Владимирский государственный университет

Л. А. АРТЮШИНА   Е. А. ТРОИЦКАЯ   Т. В. СПИРИНА

ПРОГРАММИРОВАНИЕ НА PYTHON 3

Учебно-практическое пособие

Владимир 2024

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»

Л. А. АРТЮШИНА Е. А. ТРОИЦКАЯ Т. В. СПИРИНА

ПРОГРАММИРОВАНИЕ НА PYTHON 3

Учебно-практическое пособие

Электронное издание

Допущено Федеральным учебно-методическим объединением в системе высшего образования по укрупненной группе специальностей и направлений высшего образования 09.00.00 Информатика и вычислительная техника в качестве учебно-практического пособия для студентов высших учебных заведений, обучающихся по направлению подготовки бакалавра 09.03.02 Информационные системы и технологии

Владимир 2024

ISBN 978-5-9984-1745-0

© ВлГУ, 2024
© Артюшина Л. А.,
Троицкая Е. А., Спирина Т. В., 2024

УДК 004.43
ББК 32.973

Рецензенты:
Доктор технических наук, профессор
зав. кафедрой информационных систем и программной инженерии
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича
И. Е. Жигалов

Доктор педагогических наук, профессор
зав. кафедрой математики, информатики и методики обучения
Шуйского филиала ФГБОУ ВО
«Ивановский государственный университет»
С. А. Зайцева

Артюшина Л. А. Программирование на Python 3 [Электронный ресурс] : учеб.-практ. пособие / Л. А. Артюшина, Е. А. Троицкая, Т. В. Спирина ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2024. – 423 с. – ISBN 978-5-9984-1745-0. – Электрон. дан. (7,21 Мб). – 1 электрон. опт. диск (CD-ROM). – Систем. требования: Intel от 1,3 ГГц ; Windows XP/7/8/10 ; Adobe Reader ; дисковод CD-ROM. – Загл. с титул. экрана.

Рассмотрена технология проектирования и написания эффективных и надежных программ на языках Python 3. Представлены примеры и иллюстрации, поясняющие теорию. Приведены ссылки на дополнительную литературу, вопросы для самостоятельной работы, тесты самоконтроля с ответами, более 300 упражнений, позволяющих проверить знания по всем рассматриваемым темам.

Предназначено для студентов вузов 1 – 3-го курсов всех форм обучения направлений подготовки 09.03.02 «Информационные системы и технологии» при проведении лекционных, практических или лабораторных работ по дисциплинам «Технологии и методы программирования», «Языки программирования». Может быть рекомендовано в качестве дополнительной литературы для подготовки к занятиям по дисциплинам «Алгоритмы и структуры данных», «Анализ данных».

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС ВО.

Ил. 130. Табл. 15. Библиогр.: 116 назв.

ISBN 978-5-9984-1745-0

© ВлГУ, 2024
© Артюшина Л. А.,
Троицкая Е. А., Спирина Т. В., 2024

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 9

Глава 1. ВВЕДЕНИЕ В ЯЗЫКИ ПРОГРАММИРОВАНИЯ 10
1.1. История языков программирования 10
1.1.1. Экскурс в терминологию 10
1.1.2. Ранние языки программирования 11
1.2. Классификация языков программирования 12
1.2.1. Императивные языки программирования 12
1.2.2. Первые языки программирования высокого уровня 14
1.2.3. Декларативные языки программирования 16
1.2.4. Функциональные языки программирования 17
1.2.5. Логические языки программирования 18
1.2.6. Объектно-ориентированные языки программирования 20
1.2.7. Языки сценариев 22
1.2.8. Параллельное программирование 24
1.3. Инструментальные средства разработки ПО 25
1.3.1. CASE-средства 25
1.3.2. RAD-средства 28
1.4. Дополнительные источники по языкам программирования 31
Вопросы для самоконтроля к главе 1 32
Тест самоконтроля к главе 1 33

Глава 2. ЗНАКОМСТВО СО СРЕДОЙ РАЗРАБОТКИ PYCHARM И ЯЗЫКОМ ПРОГРАММИРОВАНИЯ PYTHON 3.8 35
2.4. Комментарии 38
2.5. Отладка программного кода в среде PyCharm 39
Практическая работа № 1. ЗНАКОМСТВО СО СРЕДОЙ PYCHARM и PYTHON 3.8 44
2.6. Дополнительные источники по IDE PyCharm 44
Вопросы для самоконтроля к главе 2 44
Тест самоконтроля к главе 2 45

Глава 3. ОСНОВНЫЕ КОНЦЕПЦИИ ТЕСТИРОВАНИЯ 46
3.1. Автономное тестирование с помощью модуля unittest Python. Тестовые случаи 48

3.2. Методы модуля unittest 52
3.3. Дополнительные источники по модулю unittest 56
Практическая работа № 2. UNIT-ТЕСТИРОВАНИЕ ПРОГРАММНОГО КОДА 56
Вопросы для самоконтроля к главе 3 58
Тест самоконтроля к главе 3 59

Глава 4. ПОРЯДОК ИСПОЛНЕНИЯ ПРОГРАММЫ, УСЛОВНЫЕ ОПЕРАТОРЫ, ЦИКЛЫ 61
4.1. Ветвления 61
4.2. Циклы 63
Практическая работа № 3. УСЛОВНЫЕ ОПЕРАТОРЫ. ЦИКЛЫ 67
Вопросы для самоконтроля к главе 4 69
Тест самоконтроля к главе 4 69

Глава 5. ПЛАНОВАЯ ОБРАБОТКА ОШИБОК ПРИ ПОМОЩИ ИСКЛЮЧЕНИЙ В PYTHON 74
5.1. Обработка исключений. Блок try-exept 75
5.2. Типы исключений 76
5.3. Обработка набора исключений 79
5.4. Страхование от ошибок 80
5.5. Принудительная генерация исключений 82
5.6. Дополнительные источники по плановой обработке ошибок 83
Практическая работа № 4. ПЛАНОВАЯ ОБРАБОТКА ОШИБОК В PYTHON 83
Вопросы для самоконтроля к главе 5 84
Тест самоконтроля к главе 5 84

Глава 6. ФОРМАТИРОВАННЫЙ ВЫВОД В PYTHON 87
6.1. Оператор форматирования % 87
6.2. Метод format () 92
6.3. Дополнительные источники по вопросу форматированного вывода 94
Практическая работа № 5. ФОРМАТИРОВАННЫЙ ВВОД-ВЫВОД В PYTHON 95
Вопросы для самоконтроля к главе 6 96
Тест самоконтроля к главе 6 97

Глава 7. ОСНОВНЫЕ ТИПЫ ДАННЫХ ЯЗЫКА PYTHON....... 99
7.1. Типы данных в технологиях разработки программного обеспечения 99
7.2. Типы данных в Python 3 103
7.2.1. Числовой тип 104
Вопросы для самоконтроля к параграфу 7.2.1 110
Тест самоконтроля к параграфу 7.2.1 110
7.2.2. Статический массив 113
7.2.3. Реализация статического массива в Python 118
7.2.4. Динамический массив 121
7.2.5. Реализация динамического массива в Python. Списки 127
Вопросы для самоконтроля к параграфам 7.2.2. – 7.2.5 144
Тест самоконтроля к параграфам 7.2.2. – 7.2.5 145
7.2.6. Дополнительные источники по типам данных Python 147
Практическая работа № 6. ОРГАНИЗАЦИЯ СПИСКОВ. ОПЕРАЦИИ СО СПИСКАМИ 148
7.2.7. Строковый тип 151
Создание строки 153
7.2.7.1. Алгоритмы работы со строками 160
7.2.7.2. Регулярные выражения 166
7.2.7.3. f-строки 175
7.2.7.4. Unit-тестирование строк 176
7.2.7.5. Дополнительные источники по работе со строками 176
Практическая работа № 7. РАБОТА СО СТРОКАМИ 177
Тест самоконтроля к параграфу 7.2.7 186
7.2.8. Словари. Кортежи. Множества 189
7.2.9. Дополнительные источники по вопросу работы со словарями, множествами и кортежами 204
Практическая работа № 8. СЛОВАРИ. МНОЖЕСТВА. КОРТЕЖИ 205
Вопросы для самоконтроля к параграфам 7.2.7 – 7.2.8 210
Тест самоконтроля к параграфам 7.2.7 – 7.2.8 210

Глава 8. ОЦЕНКА ЭФФЕКТИВНОСТИ АЛГОРИТМОВ РАБОТЫ С ОСНОВНЫМИ ТИПАМИ ДАННЫХ ПО ПАМЯТИ И ВРЕМЕНИ ИСПОЛНЕНИЯ 214
8.1. Оценка временной сложности алгоритма 215
8.2. Правила вычисления времени выполнения программ 220
8.3. Анализ сортировки вставками 222

8.4. Анализ рекурсивных алгоритмов 224
8.5. Дополнительные источники по оценке временной сложности алгоритмов 226
Практическая работа № 9. ОЦЕНКА ВРЕМЕННОЙ СЛОЖНОСТИ АЛГОРИТМА 227
Вопросы для самоконтроля к главе 8 230
Тест самоконтроля к главе 8 231

Глава 9. ФАЙЛЫ 235
9.1. Открытие файла 235
9.2. Чтение файла 238
9.3. Запись в файл 241
9.4. Другие методы работы с файлами 241
9.5. Средства, напоминающие файлы 243
9.6. Дополнительные источники по работе с файлами 243
Практическая работа № 10. ФАЙЛЫ 243
Вопросы для самоконтроля к главе 9 246
Тест самоконтроля к главе 9 247

Глава 10. ФУНКЦИИ И ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ 248
10.1. Объявление и вызов функции 249
10.2. Рекурсия. Рекурсивные функции 252
10.3. Функции-генераторы 254
10.4. Функции с переменным числом аргументов 255
10.5. Значения аргументов функции "по-умолчанию" 257
10.6. Лямбда-функции 257
10.7. Вложенные функции 259
10.8. Как сделать вызов функции из другого файла в Python 264
Практическая работа № 11. ФУНКЦИИ 267
Практическая работа № 12. РЕКУРСИЯ 269
Вопросы для самоконтроля к главе 10 271
Тест самоконтроля к главе 10 272
10.9. Дополнительные источники по работе с функциями 274

Глава 11. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ 276
11.1. Описание классов и создание объектов 279
11.2. Декораторы 283

11.3. Наследование, множественное наследование 287
11.4. Порядок разрешения методов в Python. Ромбовидное наследование 288
11.5. Перегрузка операторов 290
11.6. Дополнительные источники по объектно-ориентированному программированию в Python 293
Практическая работа № 13. СОЗДАНИЕ ПОЛЬЗОВАТЕЛЬСКИХ КЛАССОВ 293
Вопросы для самоконтроля к главе 11 295
Тест самоконтроля к главе 11.. 296

Глава 12. РЕАЛИЗАЦИЯ ДИНАМИЧЕСКИХ СТРУКТУР ДАННЫХ В PYTHON. ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ В PYTHON 298
12.1. Односвязный список.. 301
12.1.1.Создание, заполнение и вывод элементов односвязного списка.. 302
12.1.2. Вставка элементов в односвязный список.............................. 308
12.1.3. Удаление элемента односвязного списка 312
12.2. Двусвязный (двунаправленный) список 315
12.2.1.Создание, обход двусвязного списка, вставка элемента в список.. 315
12.2.2. Удаление элемента двусвязного списка.................................. 321
12.3. Циклические списки.. 329
Практическая работа № 14. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. СВЯЗНЫЕ СПИСКИ... 332
12.4. Стеки 336
12.5. Очереди... 340
12.6. Дек 342
Практическая работа № 15. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. СТЕКИ, ОЧЕРЕДИ И ДЕКИ 345
12.7. Бинарные деревья 347
Структура и формирование бинарного дерева.................................. 347
12.7.1. Способы обхода и удаления вершин бинарного дерева 357
Практическая работа № 16. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. БИНАРНЫЕ ДЕРЕВЬЯ... 361
12.8. Графы.. 362
12.8.1. Графы: основные термины 363
12.8.2. Способы представления графа в памяти................................ 365

12.8.3. Способы обхода графа 369
12.8.4. Алгоритмы на графах 377
Практическая работа № 17. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ В PYTHON. ГРАФЫ 384
Вопросы для самоконтроля к главе 12 391
Тест самоконтроля к главе 12 394
12.9. Дополнительные источники по работе с динамическими структурами данных в Python 396

Глава 13. БИБЛИОТЕКИ (ПАКЕТЫ) PYTHON 398
13.1. Библиотека NumPy 398
13.2. Другие библиотеки, поддерживающие Python 399
Вопросы для самоконтроля к главе 13 401
Тест самоконтроля к главе 13 401
13.3. Дополнительные источники по работе с библиотеками 404

Глава 14. СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ 405
14.1. Базовые понятия 405
14.2. Управление версиями в среде PyCharm 409
Вопросы для самоконтроля к главе 14 411
Тест самоконтроля к главе 14 412
14.3. Дополнительные ресурсы по работе с СКВ 413

ЗАКЛЮЧЕНИЕ 414

ПРИЛОЖЕНИЕ 415

ВВЕДЕНИЕ

Python известен как универсальный язык программирования общего назначения. Это язык высокого уровня с открытым исходным кодом, который легко выучить благодаря использованию простого английского синтаксиса.

Почему Python 3? Потому что эта версия языка программирования позволяет выстраивать все рабочие процессы от веб-разработки и компьютерной графики до машинного обучения, нейронных сетей и анализа данных в одной среде.

В пособии подробно представлены все этапы процесса разработки программ: постановка задачи; анализ и исследование задачи, модели; разработка алгоритма; программирование; тестирование и отладка; анализ результатов решения задачи и уточнение в случае необходимости математической модели с повторным выполнением предыдущих этапов. Учебный материал разбит на главы в соответствии с логикой изложения материала. Рассматриваются основные типы данных, структуры данных, алгоритмы для работы с ними. Большое внимание уделено таким важным вопросам реальной практики программирования, как оценка временной сложности алгоритмов, автономное тестирование, инструментальные средства разработки программного обеспечения и системы контроля версий. Для расширения кругозора обучаемых в книгу включена глава об истории языков программирования.

Пособие содержит большое количество примеров, демонстрирующих и поясняющих каждый этап процесса разработки программного кода. В главах предусмотрены вопросы и тесты для самоконтроля, ссылки на дополнительную литературу, практические работы, включающие в себя 22 варианта индивидуальных заданий.

Глава 1. ВВЕДЕНИЕ В ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Языки программирования – разнообразные знаковые системы для записи алгоритмов.

Алгоритм – упорядоченная последовательность инструкций.

Знаковая система – это набор символов (алфавит языка), система правил образования из этих символов конструкций, с помощью которых представляются определенные компоненты алгоритма (синтаксис языка), и система правил истолкования этих конструкций, позволяющих однозначно воспроизводить процесс переработки данных (семантика языка)».

Таким образом, язык программирования служит двум связанным между собой **целям**: он дает программисту аппарат для задания действий, которые должны быть выполнены, и формирует концепции, которыми пользуется программист, размышляя о том, что делать.

1.1. История языков программирования

1.1.1. Экскурс в терминологию

Одной из самых революционных идей приведших к созданию автоматических цифровых вычислительных машин, была высказанная в 20-х годах 19 века Чарльзом Беббиджем мысль о предварительной записи порядка действия машины для последующей автоматической реализации вычислений – программе. И, хотя использованная Бебиджем запись программы на перфокартах, придуманная для управления ткацкими станками французским изобретателем Жозефом Мари Жаккаром, технически не имеет ничего общего с современными приемами хранения программ в ПК, принцип здесь по существу один. С этого момента начинается история программирования.

Аду Левлейс, современницу Беббиджа, называют первым в мире программистом. Она теоретически разработала некоторые приемы управления последовательностью вычислений (алгоритм вычисления чисел Бернулли на вычислительной машине и т.д.), которые используются в программировании и сейчас. Ею же была описана и одна из важнейших конструкций практически любого современного языка программирования – цикл.

Революционным моментом в истории языков программирования стало появление системы кодирования машинных команд с помощью специальных символов, предложенной Джоном Моучли. Предложенная им система вдохновила одну из его сотрудниц Грейс Мюррей Хоппер. При работе на компьютере «Марк-1» (1944 г.) она и возглавляемая ей группа придумали подпрограммы. И еще одно фундаментальное понятие техники программирования «отладка» – также было введено Хоппер и ее группой.

В конце 40-х годов Дж. Моучли создал систему под названием «Short Code», которая являлась примитивным языком программирования высокого уровня. В ней программист записывал решаемую задачу в виде математических формул, а затем, используя специальную таблицу, преобразовывал эти формулы в двухлитерные коды. В дальнейшем специальная программа компьютера превращала эти коды в двоичный машинный код. Система, разработанная Дж. Моучли, считается одним из первых **примитивных интерпретаторов.**

Уже в 1951 г. Хоппер создала **первый в мире компилятор**, ею же был введен сам этот термин. Компилятор Хоппер осуществлял функцию объединения команд и в ходе трансляции производил организацию подпрограмм, выделение памяти компьютера, преобразование команд высокого уровня (в то время псевдокодов) в машинные команды. «Подпрограммы находятся в библиотеке (компьютера), а когда вы подбираете материал из библиотеки – это называется компиляцией» – так она объясняла происхождение введенного ею термина.

1.1.2. Ранние языки программирования

Как и следовало ожидать, первые языки программирования (Plankalkül [plannkalkyːl]), Assembler [əˈsɛmblə] и т.д.), как и первые ЭВМ, были довольно примитивными и ориентироваными на численные расчеты из области математики и физики, а также прикладные задачи из области военного дела.

Программы, написанные на ранних языках программирования, представляли собой линейные последовательности элементарных операций с регистрами, в которых хранились данные. Нужно отметить, что ранние языки программирования были оптимизированы под ту аппаратную архитектуру конкретного компьютера, для которого они предназначались, и хотя они обеспечивали высокую эффектив-

ность вычислений, до стандартизации было еще далеко. Программа, которая была вполне работоспособной на одной вычислительной машине, зачастую не могла быть выполнена на другой.

Таким образом, ранние языки программирования существенно зависели от того, что принято называть средой вычислений и приблизительно соответствовали современным машинным кодам или языкам ассемблера.

Пример кода, выводящего сообщение «Hello, world!»:
BB 11 01 B9 0D 00 B4 0E 8A 07 43 CD 10 E2 F9 CD 20 48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21

где BB 11 01, B9 0D 00, B4 0E, 8A 07 — команды присвоения значений регистрам

43 – инкремент регистра BX

CD 10, CD 20 – вызов программных прерываний

E2 F9 – команда для организации цикла

Полужирным курсивом показаны данные (строка «*Hello, world!*»)

1.2. Классификация языков программирования

1.2.1. Императивные языки программирования

В начале 1950-х гг. машинный язык был единственным языком. Для спасения программистов от сурового машинного языка программирования, были созданы языки высокого уровня (т.е. немашинные языки), которые стали своеобразным связующим мостом между человеком и машинным языком компьютера.

Основная идея: Автомат, последовательно изменяющий свои состояния под управлением некоторой схемы, наиболее просто реализуется технически. Поэтому первые компьютеры были императивными. И остались такими и в наши дни, несмотря на все эксперименты с оригинальными вычислительными устройствами.

Даже типичное определение алгоритма (описание последовательности действий для решения какой-либо задачи), несет на себе сильнейший отпечаток императивного подхода. Стоит ли говорить о том, почему императивное программирование - практически наиболее "популярное"?

Одна из характерных черт императивного программирования - наличие переменных с операцией "разрушающего присвоения". То

есть, была переменная А, было у нее значение X. Алгоритм предписывает на очередном шаге присвоить переменной А значение Y. То значение, которое было у А, будет "навсегда забыто". На практике это означает "переход между состояниями под управлением функции переходов".

Синтаксис описания алгоритмов в простейшем языке, поддерживающем императивную модель программирования, был примерно таким:

Оператор ::= Простой оператор | Структурный оператор
Простой оператор ::= Оператор присваивания | Оператор вызова | Оператор возврата

```
Структурный оператор ::= Оператор последовательно-
го исполнения | Оператор ветвления | Оператор цик-
ла
Оператор присваивания ::= Переменная:=Выражение;
Оператор вызова ::= Имя подпрограммы (Список пара-
метров);
Оператор возврата ::=return[ Выражение ];
Оператор последовательного исполнения ::=begin
Оператор* end
Оператор ветвления ::= if Выражение then Оператор*
(elseif Выражение then Оператор*)* [ else Опера-
тор* ] end
Оператор цикла ::= while Выражение do Оператор*
end
```

Императивные языки программирования работали(ют) через трансляционные программы, которые переводят «исходный код» (гибрид английских слов и математических выражений, который считывает машина) в машинные коды и, в конечном итоге, «заставляют» компьютер выполнять соответствующие команды.

Существует два основных вида трансляторов:

– интерпретаторы, которые сканируют и проверяют исходный код в один шаг;
– компиляторы, которые сканируют исходный код для производства текста программы на машинном языке, которая затем выполняется отдельно.

Разработки в этом направлении привели к созданию императивных языков Кобол (COBOL – Common Business Oriented Language, 1960 г.), Фортран (Fortran, 1954 г.), Алгол (Algol, 1950-ые гг.)

Фортран позволял использовать именованные переменные, составные выражения, подпрограммы и многие другие элементы, распространённые в императивных языках, был разработан фирмой IBM .

Алгол – соперник Фортрана, был разработан в Европе . оба языка программирования имели хорошо разработанные средства математических вычислений.

В Коболе по сравнению с Фортраном и Алголом, были лучше развиты средства обработки текстов, организация вывода данных в форме требуемого документа. Он задумывался как основной язык для массовой обработки данных в сферах управления и бизнеса.

То есть для первых языков программирования высокого уровня **предметная ориентация** языков была характерной чертой.

Пример фрагмента программы на Фортран для вычисления выражения присваивания b+a:

```
program addNumbers
! Type declarations
real :: a, b, result
a = 12.0
b = 15.0
result = a + b
print *, 'The total is ', result
end program
```

1.2.2. Первые языки программирования высокого уровня

В середине 60-х годов Томас Курц и Джон Камени создали специализированный язык программирования, который состоял из простых слов английского языка. Новый язык назвали «универсальным символическим кодом для начинающих» (Beginner All-Purpose Symbolic Instruction Code, или, сокращенно, BASIC). Годом рождения нового языка можно считать 1964. Сегодня универсальный язык BASIC (Бейсик) имеет множество версий, приобрел большую популярность и получил широкое распространение среди пользователей ПК различных категорий во всем мире. В значительно мере этому способствовало то, что BASIC начали использовать как встроенный

язык персональных компьютеров, широкое распространение которых началось в конце 70-х годов.

Однако Бейсик – не структурный язык, и поэтому он плохо подходит для обучения качественному программированию. Справедливости ради следует заметить, что последние версии Бейсика для ПК (например, QBasic) стали более структурными и по своим изобразительным возможностям приближаются к таким языкам, как Паскаль.

Разработчики ориентировали языки на разные классы задач, в той или иной мере привязывали их к конкретной архитектуре ПК, реализовывали личные вкусы и идеи. В 60-е годы были предприняты попытки преодолеть эту «разноголосицу» путем создания универсального языка программирования. Первым детищем этого направления стал PL/1 (Programm Language One), разработанный фирмой IBM в 1967 году. Этот язык претендовал на возможность решать любые задачи: вычислительные, обработки текстов, накопления и поиска информации. Однако он оказался слишком сложным, транслятор с него – недостаточно оптимальным и содержал ряд не выявленных ошибок.

Однако линия на универсализацию языков была поддержана. Старые языки были модернизированы в универсальные варианты: Алгол-68 (1968 г.), Фортран-77. Предполагалось, что подобные языки будут развиваться и усовершенствоваться, станут вытеснять все остальные. Однако ни одна из этих попыток не увенчалась успехом.

Значительным событием в истории языков программирования стало создание в 1971 году языка Паскаль. Его автор – швейцарский ученый Никлаус Вирт. Вирт назвал его в честь великого французского математика и религиозного философа XVII века Блеза Паскаля, который изобрел первое суммирующее устройство, именно поэтому новому языку было присвоено его имя. Этот язык первоначально разрабатывался как учебный язык структурного программирования, и, действительно, сейчас он является одним из основных языков обучения программированию в школах и вузах.

В 1975 году два события стали вехами в истории программирования Билл Гейтс и Пол Аллен заявили о себе, разработав свою версию Basic, а Вирт и Йенсен выпустили классическое описание языка «Pascal User Manual and Report».

Не менее впечатляющей, в том числе и финансовой, удачи добился Филипп Кан, француз, разработавший в 1983 году систему Турбо-Паскаль. Суть его идеи состояла в объединении последова-

тельных этапов обработки программы – компиляции, редактирования связей, отладки и диагностики ошибок – в едином интерфейсе. Турбо-Паскаль – это не только язык и транслятор с него, но еще и операционная оболочка, позволяющая пользователю удобно работать на Паскале. Этот язык вышел за рамки учебного предназначения и стал языком профессионального программирования с универсальными возможностями.

В силу названных достоинств Турбо-Паскаль стал источником многих современных языков программирования. С тех пор появилось несколько его версий, последняя – седьмая.

Фирма Borland/Inprise завершила линию продуктов Турбо-Паскаль и перешла к выпуску системы визуальной разработки для Windows –Delphi.

Большоt влияние на современное программирование наложил язык Си (первая версия – 1972 год), являющийся очень популярным в середе разработчиков систем программного обеспечения, включая операционные системы. Этот язык создавался как инструментальный язык для разработки операционных систем, трансляторов, баз данных и других системных и прикладных программ. Си сочетает в себе как черты языка высокого уровня, так и машинно-ориентированного языка, допуская программиста ко всем машинным ресурсам, чего не обеспечивают такие языки как Бейсик и Паскаль.

1.2.3. Декларативные языки программирования

В 60-х г.г. возникает новый подход к программированию, который до сих пор успешно конкурирует с императивным, а именно, декларативный подход. Суть подхода состоит в том, что программа представляет собой не набор команд, а описание действий, которые необходимо осуществить. Этот подход существенно проще и прозрачнее формализуем математическими средствами. Отсюда следует тот факт, что программы проще проверять на наличие ошибок (тестировать), а также на соответствие заданной технической спецификации (верифицировать). Высокая степень абстракции также является преимуществом данного подхода. Фактически, программист оперирует не набором инструкций, а абстрактными понятиями, которые могут быть достаточно обобщенными.

Программа состоит из набора фактов и набора правил вывода. Вы начинаете программу, задавая вопрос, и набор правил вывода

языка использует факты и правила для ответа на ваш вопрос. Примерами декларативного языка программирования являются Пролог, SQL.

Пролог разработан во Франции в 1972 году для решения проблем «искусственного интеллекта». Пролог позволяет в формальном виде описывать различные утверждения, логику рассуждений и заставляет ПК давать ответы на заданные вопросы.

Язык структурированных запросов SQL был создан фирмой IBM в начале 70-х годов XIX века для работы с реляционными базами данных. В круг ответственности SQL входит добавление данных, извлечение по запросу, обновление и их удаление, а также создание и изменение схемы реляционной БД, контроль прав доступа к данным.

Пример инструкции, удаляющей запись из таблицы на языке SQL:

```
DELETE FROM Customers WHERE CustomerName='Alfreds
Futterkiste';
```

1.2.4. Функциональные языки программирования

При функциональном подходе процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании). Например, APL – предшественник современных научных вычислительных сред, таких как MATLAB. Пример кода, вычисляющего угол между векторами на языке APL:

$$\cos\varphi = \frac{(\vec{a},\vec{b})}{|\vec{a}|\cdot|\vec{b}|} = \frac{2\cdot3+4\cdot1}{\sqrt{2^2+4^2\cdot\sqrt{3^2+1^2}}} = \frac{10}{\sqrt{200}} = \frac{1}{\sqrt{2}} = \frac{\sqrt{2}}{2}.$$

Искомый угол φ равен:

$$\varphi = \arccos(\cos\varphi) = \arccos\left(\cos\frac{\sqrt{2}}{2}\right) = 45^0 .$$

Текст программы:

```
>> a=[2 4];
>> b=[3 1];

% вычисление (a,b)
>> c=times (a,b);
>> c=sum(c);
```

```
% вычисление |a⃗|·|b⃗|
>> a=a.^2;
>> b=b.^2;
>> s1=sqrt (plus (a(1), a(2)));
>> s2=sqrt (plus (b(1), b(2)));
>> cos_fi=c/ (s1*s2);
>> fi=acosd (cos_fi)

#будет напечатано
    fi =
       45.0000
```

Поскольку функция является естественным формализмом для языков функционального программирования, реализация различных аспектов программирования, связанных с функциями, существенно упрощается.

В частности, интуитивно прозрачным становится написание рекурсивных функций, т.е. функций, вызывающих самих себя в качестве аргумента. Кроме того, естественной становится и реализация обработки рекурсивных структур данных (например, списков, деревьев и др.).

Естественно, языки функционального программирования не лишены недостатков. Часто к ним относят нелинейную структуру программы и относительно невысокую эффективность реализации. Однако, первый недостаток достаточно субъективен, а второй успешно преодолен современными реализациями, в частности, рядом последних трансляторов языка SML, включая и компилятор для среды Microsoft.NET.

1.2.5. Логические языки программирования

В 70-х гг. XX века возникла еще одна ветвь языков декларативного программирования, связанная с проектами в области искусственного интеллекта, а именно, языки логического программирования.

Первым языком логического программирования был язык Planner, в нем была заложена возможность автоматического вывода результата из данных и заданных правил перебора вариантов. В настоящее время самым известным языком логического программирования является Пролог (Prolog).

Согласно логическому подходу к программированию, программа представляет собой совокупность правил или логических высказываний. Кроме того, в программе допустимы логические причинно-следственные связи, в частности, на основе операции импликации.

Продемонстрируем изложеное на примере утверждений (предикатов):

- Марк изучает книгу (учебник, документацию).
- Маша видит клавиатуру (мышку, книгу, тетрадь, Марка)
- Миша изучает математику (документацию, учебник)
- Саша старше Лёши
- У нас есть объекты: книга, учебник, документация, мышь, тетрадь, математика, Марк, Маша, Саша, Леша.

Между объектами существует связь, которую можно выразить, например, через глагол «изучает» как в наших утверждениях. Строка программного кода на Прологе, соответствующая первому утверждению будет выглядеть:

```
study(mark, book).

Зададим вопрос:
"Марк изучает книгу?"
На Прологе это выглядит:
    ?- study(mark, book).
Получим ответ:
    true.
```

По сути мы спросили "есть ли факт study(mark, study) в базе?", на что нам Пролог ответил «Да» (true).

Таким образом, языки логического программирования базируются на классической логике и применимы для систем логического вывода, в частности, для так называемых экспертных систем (ЭС). Базовый цикл работы ЭС представлен на рисунке 1.

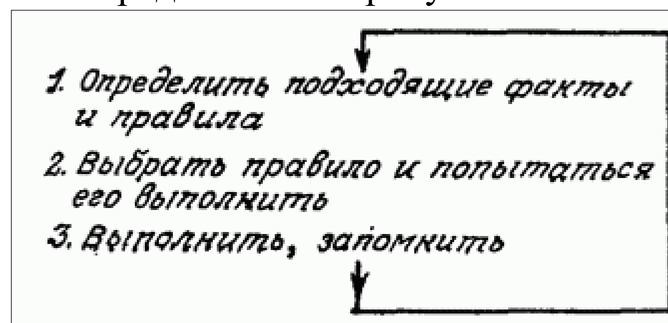


Рис.1. Базовый цикл ЭС

Важным преимуществом подхода является достаточно высокий уровень машинной независимости, а также возможность откатов – возвращения к предыдущей подцели при отрицательном результате анализа одного из вариантов в процессе поиска решения (скажем, очередного хода при игре в шахматы), что избавляет от необходимости поиска решения полным перебором вариантов и увеличивает эффективность реализации.

Рассмотрим еще один пример, демонстрирующий вывод факта на основе правил:

```
Правило 1. «ЕСЛИ Двигатель не заводится И Фары не горят, ТО Сел аккумулятор».
Правило 2. «ЕСЛИ Указатель бензина находится на нуле, ТО Двигатель не заводится».
```

Предположим, что в рабочую память (РП) от пользователя ЭС поступили факты: фары не горят и указатель бензина находится на нуле.

Рассмотрим основные шаги алгоритма прямого вывода.

1. Сопоставление фактов из РП с образцами правил из базы правил (БП). Правило 1 не может сработать, а Правило 2 срабатывает, так как образец, совпадающий с его антецедентом, присутствует в РП.
2. Действие сработавшего Правила 2. В РП заносится заключение этого правила — образец «Двигатель не заводится».
3. Второй цикл сопоставления фактов в РП с образцами правил. Теперь срабатывает Правило 1, так как конъюнкция условий в его антецеденте становится истинной.
4. Действие Правила 1, которое заключается в выдаче пользователю окончательного диагноза – «Сел аккумулятор».
5. Конец работы (БП исчерпана).

1.2.6. Объектно-ориентированные языки программирования

В чистом объектно-ориентированном программировании (ООП) все, что можно – процессы, данные, события – являются **объектами**. Все объекты располагают некоторыми "собственными" данными, представленными как **ссылки** на другие объекты.

Объекты могут обмениваться между собой **сообщениями**. При получении объектом сообщения запускается соответствующий ему обработчик, иначе называемый **методом**.

У объекта есть ассоциативный контейнер, который позволяет получить по сообщению метод для его обработки. Кроме этого, у объекта есть **объект-предок**.

Если метод для обработки сообщения не найден, сообщение будет перенаправлено объекту-предку. Эту структуру в целом (таблица обработчиков + предки) из соображений эффективности выделяют в отдельный объект, называемый **классом** данного объекта.

У самого объекта будет ссылка на объект, представляющий его класс. Объект по отношению к своему классу является **экземпляром**.

Так как классы тоже представлены, как объекты, существует класс, определяющий поведение, общее для всех классов. Такой класс принято называть **мета-классом**.

Важно выделить следующие основные **свойства** объектов:

- так как один объект может воздействовать на другой исключительно при помощи посылки последнему сообщений, он не может как-либо непосредственно работать с собственными данными "собеседника", и, следовательно, не может нарушить их внутреннюю согласованность. Это свойство (сокрытие данных) принято называть **инкапсуляцией**;
- так как объекты взаимодействуют исключительно за счет обмена сообщениями, объекты-собеседники могут ничего не знать о реализации обработчиков сообщений у своего визави. Взаимодействие происходит исключительно в терминах сообщений/событий, которые достаточно легко привязать к предметной области. Это свойство (описание взаимодействия исключительно в терминах предметной области) называют **абстракцией**;
- объекты взаимодействуют исключительно через посылку сообщений друг другу. Поэтому если в каком-либо сценарии взаимодействия объектов заменить произвольный объект другим, способным обрабатывать те же сообщения, сценарий так же будет реализуем. Это свойство (возможность подмены объекта другим объектом со сходной структурой класса) называется **полиморфизмом**;
- отношение "потомок-предок" на классах принято называть **наследованием.**

Все эти свойства по-отдельности встречаются и в других методологиях программирования. Так, локальные переменные инкапсулированы в процедуре. Объектно-ориентированное программирование достаточно гармонично их сочетает.

Пример кода на рисунке 2 демонстрирует описанную выше иерархию и взаимодействие объекта и класса.

Класс DateClass содержит один метод print () и три поля данных (день, месяц, год) – характеристики объектов этого класса. В основной программе создается и инициализируется экземпляр класса – объект today. Метод print, примененный к объекту, выводит на экран значения дня, месяца и года.

```
1  #include <iostream>
2
3  class DateClass
4  {
5  public:
6      int m_day;
7      int m_month;
8      int m_year;
9
10     void print()
11     {
12         std::cout << m_day << "/" << m_month << "/" << m_year;
13     }
14 };
15
16 int main()
17 {
18     DateClass today { 12, 11, 2018 };
19
20     today.m_day = 18; // используем оператор выбора членов для выбо
21     today.print(); // используем оператор выбора членов для вызова
22
23     return 0;
24 }
```

Рис.2. Фрагмент кода на объектно-ориентированном языке

1.2.7. Языки сценариев

Развитием событийно управляемой концепции объектно-ориентированного подхода стало появление в 90-х гг. целого класса языков программирования, которые получили название языков сценариев или скриптов. Сегодня наиболее популярными и активно используемыми являются: JavaScript, JScript, PHP, Python, Perl.

В рамках данного подхода программа представляет собой совокупность возможных сценариев обработки данных, выбор которых инициируется наступлением того или иного события (щелчок по кнопке мыши, попадание курсора в ту или иную позицию, изменение атрибутов того или иного объекта, переполнение буфера памяти и т.д.). События могут инициироваться как операционной системой, так и пользователем.

Основные достоинства языков данного класса унаследованы от объектно-ориентированных языков программирования. Это интуитивная ясность описаний, близость к предметной области, высокая

степень абстракции, хорошая переносимость, широкие возможности повторного использования кода.

Существенной положительной чертой языков сценариев является их совместимость с передовыми инструментальными средствами автоматизированного проектирования и быстрой реализации программного обеспечения, или так называемыми CASE- (ComputerAided Software Engineering) и RAD- (Rapid Application Development) средствами.

Одним из наиболее передовых инструментальных комплексов для быстрой разработки приложений является Microsoft Visual Studio .NET (VS).

Пример разработки окна приложения в VS представлен на рис.3.

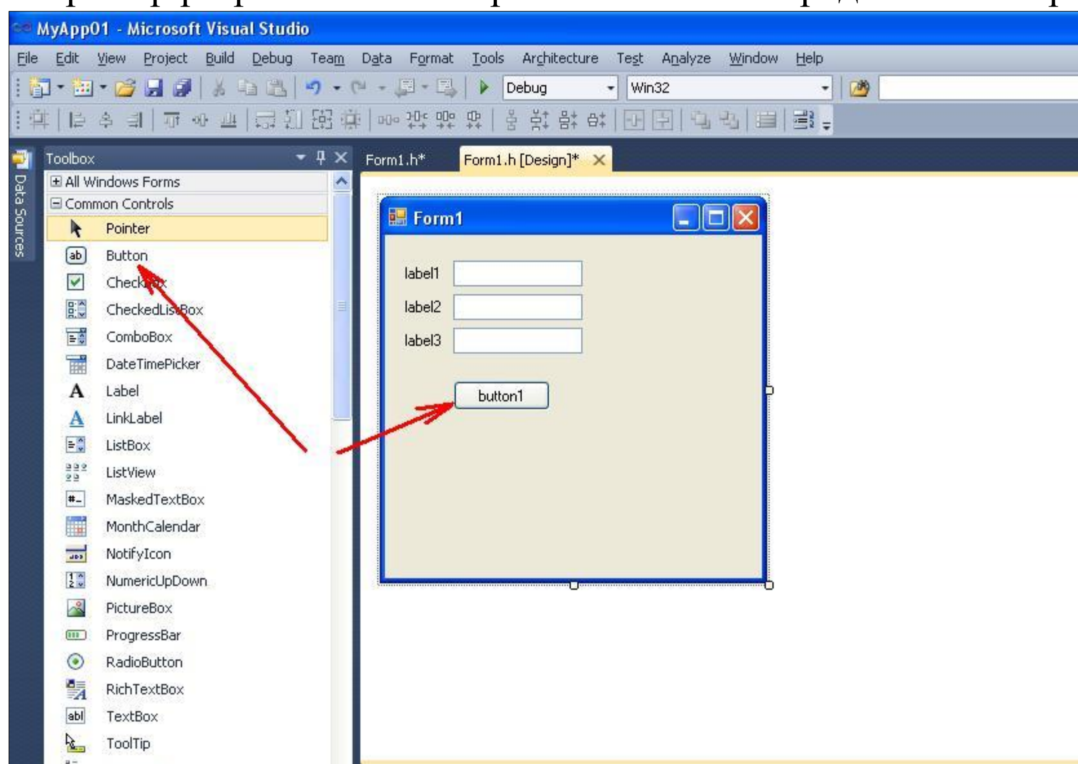


Рис.3. Компонент button

Естественно, что вместе с достоинствами объектно-ориентированного подхода языки сценариев унаследовали и ряд недостатков. К последним относятся: сложность тестирования и верификации программ и возможности возникновения в ходе эксплуатации множественных побочных эффектов, проявляющихся за счет сложной природы взаимодействия объектов и среды, представленной интерфейсами с подчас многочисленными одновременно работающими пользователями программного обеспечения, операционной системой и внешними источниками данных

1.2.8. Параллельное программирование

Еще одним весьма важным классом языков программирования являются языки параллельного программирования (Ocam, Orca, Parallaxis, Linda и т.д.) и расширения существующих языков (CC++, Charm++ , Fortran-DVM и т.д.).

Пример фрагмента кода на языке Си, определяющий характеристики системного таймера:

```
int main(int argc, char **argv)
{
   double time_start, time_finish, tick;
   int rank, i;
   int len;
   char *name;
   name
 =
(char*)malloc(MPI_MAX_PROCESSOR_NAME*sizeof(char))
;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Get_processor_name(name, &len);
   tick = MPI_Wtick();
   time_start = MPI_Wtime();
   for (i = 0; i<NTIMES; i++)
      time_finish = MPI_Wtime();

   printf ("processor %s, process %d: tick= %lf,
time= %lf\n", name, rank, tick, (time_finish-
time_start)/NTIMES);
   MPI_Finalize();
}
```

С помощью процедуры malloc() динамически выделяется память, MPI_Init() инициализирует параллельную часть приложения, MPI_Comm_rank () следит за процессами, а MPI_Get_processor_name () возвращает в строке NAME имя узла, на котором запущен процесс.

Как видно из примера, программы, написанные на этих языках, представляют собой совокупность описаний процессов, которые могут выполняться как в действительности одновременно, так и в псевдопараллельном режиме. В последнем случае устройство, обрабатывающее процессы, функционирует в режиме разделения времени, вы-

деляя время на обработку данных, поступающих от процессов, по мере необходимости (а иногда с учетом последовательности или приоритетности выполнения операций).

Языки параллельных вычислений позволяют достичь заметного выигрыша в эффективности при обработке больших массивов информации, поступающих, например, от одновременно работающих пользователей, либо имеющих высокую интенсивность (видеоинформация или звуковые данные высокого качества).

Другой весьма значимой областью применения языков параллельных вычислений являются системы реального времени, в которых пользователю необходимо получить ответ от системы непосредственно после запроса. Системы такого рода отвечают за жизнеобеспечение и принятие ответственных решений.

1.3. Инструментальные средства разработки ПО

Мы уже упоминали о таких средствах, рассказывая о языках сценариев. В этом разделе поговорим о CASE и RAD-средствах подробно.

1.3.1. CASE-средства

В состав CASE – средств входят четыре основных компонента.

1. средства централизованного хранения всей информации о проекте (репозиторий);
2. средства ввода. Служат для ввода данных в репозиторий, организации взаимодействия участников проекта с CASE–средством. Должны поддерживать различные методологии анализа, проектирования, тестирования, контроля;
3. средства анализа и разработки. Предназначены для анализа различных видов графических и текстовых описаний и их преобразования в процессе разработки;
4. средства вывода. Служат для кодогенерации, создания различного вида документов, управления проектом.

К CASE–средствам может быть отнесено любое программное средство, обеспечивающее автоматическую помощь при разработке ПО, сопровождении или управлении проектом, базирующееся на следующих основополагающих принципах:

- графическая ориентация. Использование мощной графики для описания и документирования систем и для улучшения интерфейса с пользователем;
- интеграция. Обеспечивает легкость передачи данных между своими компонентами и другими средствами, входящими в состав линейки CASE–средств. Это позволяет поддерживать совокупность процессов жизненного цикла (ЖЦ) ПО;
- локализация всей проектной информации в репозитории. Исполнителям проекта доступны соответствующие разделы репозитория в соответствии с их уровнем доступа. Это обеспечивает поддержку принципа коллективной работы;

Кроме перечисленных принципов в основе концептуального построения CASE– средств лежат следующие положения:

- человеческий фактор, его учет позволяет привести процессы ЖЦ к легкой, удобной и экономичной форме;
- использование базовых программных средств, применяющихся в других приложениях (СУБД, компиляторов различных языков программирования, отладчиков и др.);
- автоматизированная (или автоматическая) кодогенерация. При автоматизированной кодогенерации выполняется частичная генерация кодов программного средства, остальные участки программируются вручную. При автоматической кодогенерации выполняется полная генерация кодов;
- ограничение сложности. Позволяет поддерживать сложность компонентов разрабатываемого программного средства или системы на уровне, доступном для понимания, использования и модификации;
- доступность для различных категорий пользователей, в том числе заказчиков, специалистов в предметной области, системных аналитиков, проектировщиков, программистов, тестировщиков, инженеров по качеству, менеджеров проекта. CASE–средства содержат инструменты различного функционального назначения, поддерживающие различные этапы основных, вспомогательных и организационных процессов ЖЦ;
- рентабельность, обеспечивает быструю окупаемость денежных средств, вложенных в CASE–средства, за счет сокращения сроков и стоимости проектов.
- сопровождаемость. CASE–средства обладают способностью адаптации к изменяющимся требованиям и целям проекта.

Классификация Case-средств

Классификация по типам отражает функциональное назначение CASE – средств в ЖЦ ПС.

1. Анализ и проектирование. Средства этого типа используются для поддержки начальных этапов процесса разработки: анализа предметной области, разработки требований к системе, проектирования системной архитектуры, технического проектирования программной архитектуры, технического проектирования программных средств. На выходе генерируются спецификации системы, ее компонентов и интерфейсов, архитектура системы, архитектура программного средства, технический проект, включая алгоритмы и структуры данных. Примерами являются: AllFusion Process Modeler (BPwin), CASE.Аналитик, Design/IDEF, Telelogic DOORS, Telelogic Modeler, Telelogic TAU, Telelogic Rhapsody, Telelogic Statemate.
2. Проектирование баз данных и файлов. Средства этого типа обеспечивают логическое моделирование данных, автоматическое преобразование моделей данных в третью нормальную форму, автоматическую генерацию схем баз данных и описаний форматов файлов на уровне программного кода. Примерами являются: AllFusion Process Modeler (BPwin), CA Erwin Data Model Validator, S-Designor, Silverrun, Designer2000, Telelogic TAU, Telelogic Rhapsody.
3. Программирование и тестирование. Средства этого типа поддерживают программирование и тестирование. Данные средства автоматически выполняют кодогенерацию на основе спецификаций или моделей. Содержат графические редакторы, средства поддержки работы с репозиторием, генераторы и анализаторы кодов, генераторы тестов, анализаторы покрытия тестами, отладчики. Примерами являются: TAU/Developer, TUA/Tester, Logiscope Audit, Logiscope RuleChecker, Logiscope TestChecker, Logiscope Reviewer, Rhapsody Developer.
4. Сопровождение. Общей целью средств этого типа является поддержка корректировки, изменений, преобразований, реинженерия существующей системы, поддержка документации по проекту. К данным средствам относятся средства документирования, анализаторы программ, средства управления изменениями и конфигурацией ПС и систем, средства реструктурирования и реинженении, средства обеспечения мобильности. Примерами являются: Telelo-

gic DocExpress, Telelogic Synegry, Telelogic Change, AllFusion Change Management Suite.
5. Окружение. К средствам данного типа относятся средства поддержки интеграции CASE – средств и данных. Примерами являются: Telelogic Rhapsody Gateway, Telelogic Rhapsody Interface Pack, AllFusion Data Profiler, AllFusion Model Manager, AllFusion Model Navigator.
6. Управление проектом. К средствам данного типа относятся средства поддержки процесса управления ЖЦ. Их функциями являются планирование, контроль, руководство, организация взаимодействия. Примерами являются: Telelogic Focal Point, Telelogic dashboard, AllFusion Process Management Suite, Advisor.

1.3.2. RAD-средства

Концепция быстрой разработки приложений была создана в 1970-х годах, но официально представлена в 1991 году. Эта методология ориентирована на быструю разработку приложений посредством нескольких итераций и постоянного взаимодействия с клиентом.

Модель RAD делает упор на гибкие и быстрые выпуски прототипов, удобство использования собственных приложений, адаптацию в соответствии с отзывами пользователей и быструю доставку, а не на долгосрочное планирование и единый начальный набор требований. Благодаря этому, модель RAD стала более популярной по сравнению с каскадной и итеративной моделями разработки ПО.

Основные преимущества концепции быстрой разработки приложений:

- более быстрая разработка и доставка;
- повышенная гибкость и адаптивность;
- улучшенное управление рисками;
- меньшее программирования вручную и более быстрое тестирование;
- постоянные, актуальные отзывы пользователей в реальном времени;

Этапы быстрой разработки приложений. Быстрая разработка приложений включает четыре этапа, необходимых для завершения проекта. Ее цель — сократить время планирования и сосредоточиться на разработке и создании продукта. Даже если некоторые шаги по-

вторяются, в результате получается продукт, которым будут довольны и ваша команда, и заинтересованные лица.

Этап 1. Определение требований проекта. Все участники — вы, разработчики, пользователи ПО и заинтересованные лица — определяете, исследуете и утверждаете объем и требования проекта, включая цели, ожидания, сроки и бюджет. Во время краткого совещания по представлению проекта заинтересованные лица предлагают свое видение, а лица, принимающие решения, и разработчики помогают согласовать список требований.

Этап 2. Создание прототипов. Команда начинает разработку моделей и прототипов. Цель состоит в том, чтобы быстро создать работающую модель и представить ее заинтересованным лицам. Для достижения целей и выполнения требований разработчики и дизайнеры работают вместе. На ранних этапах разработки прототипа они могут применять обходные решения, которые позволяют создать рабочий продукт без ущерба для качества. В это время важную роль имеют взаимодействие с пользователем, тестирование и отзывы. Постоянные отзывы помогают команде видеть живую систему, а не абстрактный дизайн. Таким образом, ошибки обнаруживаются и исправляются на более ранних этапах, что позволяет придерживаться графика заинтересованных лиц и создавать лучшую структуру проекта для внесения будущих дополнений.

Этап 3. Создание, тестирование и внедрение отзывов. Имея рабочий прототип, пора превратить его в рабочую модель. Разработчики собирают отзывы пользователей и создают продукт. Обязательно включите в процесс платформу создания приложений, чтобы воплотить свою идею в жизнь. Благодаря программированию приложений, тестированию системы и интеграции модулей прототип и бета-системы преобразуются в рабочую модель. Поскольку команды используют с минимумом программирования и решения для быстрой разработки приложений, можно быстро вносить любые изменения. Программное обеспечение и приложения тщательно тестируются, и заинтересованные лица могут предлагать изменения или новые идеи по мере обнаружения проблем. Много ошибок быть не должно, поскольку преимущество быстрой разработки приложений в том, что вы можете видеть большинство из них в реальном времени на этапе создания прототипа и сразу же исправлять. Когда заинтересованные лица будут довольны вашим продуктом, можно завершить его разработку.

Этап 4. Заключительный этап — создание оптимизированной версии вашего конечного продукта: стабильного и простого в обслуживании для долгого срока службы. Характеристики, функции и внешний вид окончательно согласовываются с заинтересованными лицами. После перехода в рабочую среду пользователи могут проводить полномасштабное тестирование или обучение. Теперь продукт готов к презентации заинтересованным лицам.

К известным инструментальным средствам разработки программного обеспечения можно отнести Power Apps Studio, позволяющего создать приложение с нуля и набора данных или одного из готовых шаблонов (рис. 4). Затем доработать его так, как вам нужно.

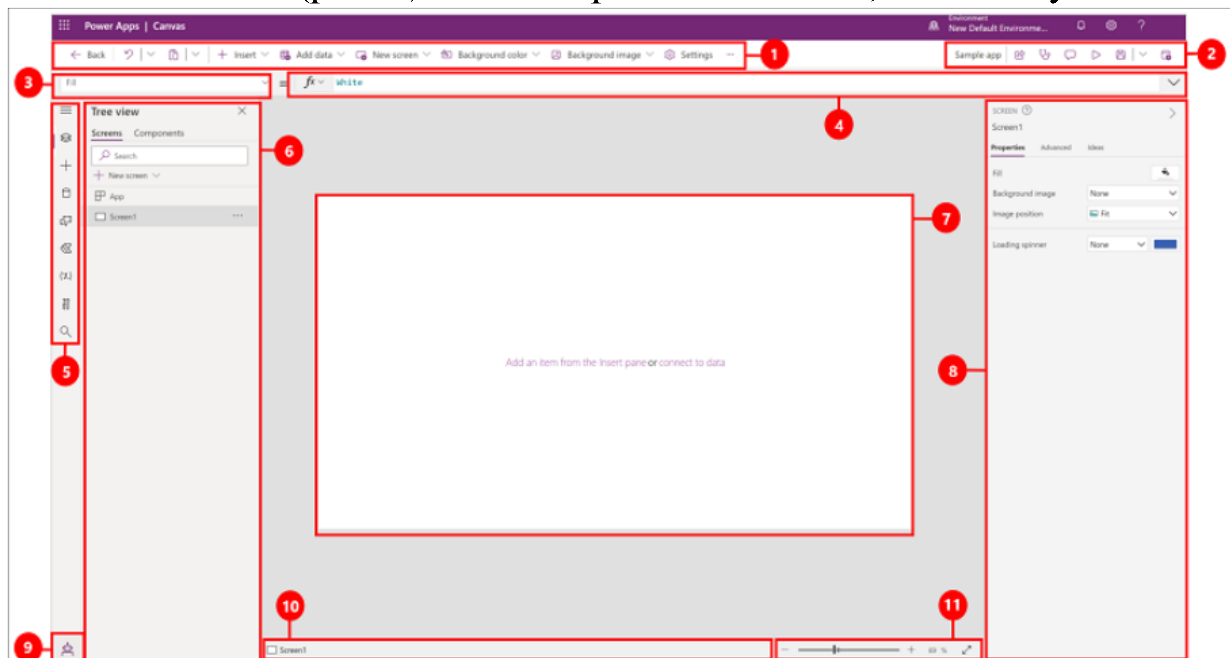


Рис.4. Внешний вид окна приложения Power Apps Studio

1. Современная панель команд Power Apps Studio: динамическая панель команд, которая показывает набор команд в зависимости от выбранного элемента управления.
2. Действия приложения: параметры переименования, совместного использования, выполнения средства проверки приложений, добавления комментариев, предварительного просмотра, сохранения или публикации приложения.
3. Список свойств: список свойств для выбранного объекта.
4. Панель формул: создание или редактирование формулы для выбранного свойства с одной или несколькими функциями.
5. Меню разработки приложений: панель выбора для переключения между источниками данных и параметрами вставки.

6. Варианты разработки приложений: панель сведений с параметрами, относящимися к выбранному пункту меню для создания приложения.
7. Холст/экран: основной холст для построения структуры приложения.
8. Область свойств — список свойств выбранного объекта в формате пользовательского интерфейса.
9. Виртуальный агент: помощь в создании приложения от виртуального агента.
10. Селектор экрана: переключение между разными экранами в приложении.
11. Изменить размер экрана холста: изменение размера холста во время разработки в Power Apps Studio.

1.4. Дополнительные источники по языкам программирования

1. Орлов С.А. Теория и практика языков программирования: Учебник для вузов. Стандарт 3-го поколения. – СПб.: Питер, 2013. – 688 с.: ил.
2. История языков программирования. Режим доступа: https://translated.turbopages.org/proxy_u/en-ru.ru.d9c85772-636b5acb-37133951-74722d776562/https/en.wikipedia.org/wiki/Evolution_of_programming_languages
3. Какой язык программирования считается первым? Режим доступа: https://skillbox.ru/media/code/kakoy_yazyk_programmirovaniya_schitaetsya_pervym/
4. Первые языки программирования. Режим доступа: https://it-black.ru/pervyye-yazyki-programmirovaniya/
5. Основы параллельного программирования. Режим доступа: https://intuit.ru/EDI/08_01_19_2/1546899581-11707/tutorial/450/objects/1/files/01.pdf
6. Технологии параллельного программирования. Режим доступа: https://moluch.ru/conf/tech/archive/332/15178/
7. Сравнительный обзор Case-средств для проектирования программных систем. Режим доступа: https://scienceforum.ru/2015/article/2015008920
8. Декларативный подход к программированию. Режим доступа: https://prognote.ru/other/declarative-approach-to-programming/

9. Программирование на функциональном уровне. Режим доступа: https://prognote.ru/other/functional-level-programming/
10. Что такое логическое программирование и зачем оно нужно? Режим доступа: https://habr.com/ru/post/322900/
11. Объектно-ориентированный подход к программированию. Режим доступа: https://intuit.ru/studies/courses/50/50/lecture/1492

Вопросы для самоконтроля к главе 1

1. Чем язык программирования отличается от естественного языка?
2. Перечислите основные революционные идеи в истории языков программирования.
3. Что было характерно для ранних языков программирования?
4. Чем язык высокого уровня отличается от машинного языка?
5. Охарактеризуйте императивный подход в программировании.
6. Дайте характеристику императивного подхода применительно к компьютерной технике.
7. Чем интерпретатор отличается от компилятора?
8. Перечислите основные языки программирования высоко уровня. Дайте краткую характеристику каждому из них.
9. Чем декларативные языки отличаются от других языков программирования. Приведите примеры.
10. Чем функциональные языки отличаются от других языков программирования. Приведите примеры.
11. Кратко опишите основные характеристики объектно-ориентированных языков программирования. Почему ООП стал приоритетным при разработке большинства программных проектов?
12. Перечислите и поясните основные свойства объектов.
13. Дайте краткую характеристику языкам параллельного программирования.
14. Как понимается процесс в параллельном программировании?
15. Обоснуйте необходимость использования CASE-средств для моделирования процессов.

Тест самоконтроля к главе 1

1. Первый в мире компилятор был создан: а) в 40-х гг. XX века; б) в конце XIX века Ч. Беббиджем в) в 50-ых гг. XX века; г) в 1951 г.
2. Первая идея о программе была высказана: а) Бьерном Страуструпом; б) Адой Лавлейс; в) Чарльзом Беббиджем; г) Конрадом Цузе.
3. Алгоритмическая конструкция цикл впервые была описана: а) Аланом Тьюрингом; б) Адой Лавлейс; в) Чарльзом Беббиджем; г) Конрадом Цузе.
4. Термин «отладка» применительно к программированию был предложен: а) Бьерном Страуструпом; б) Джоном Моучли; в) Грейс Хоппер; г) Конрадом Цузе.
5. Первый интерпретатор был разработан: а) Бьерном Страуструпом; б) Джоном Моучли; в) Грейс Хоппер; г) Конрадом Цузе.
6. Первый в мире компилятор был создан: а) Бьерном Страуструпом; б) Джоном Моучли; в) Грейс Хоппер; г) Конрадом Цузе.
7. Первые языки программирования были ориентированы на: а) вычислительные расчеты; б) военное дело; в) организацию управления различными объектами; г) работу с структурами данных.
8. Термин «императивность» применительно к программированию означает: а) описание последовательности действий в программе; б) описание цели программы; в) описание в программе последовательности действий, приводящих к результату; г) описание результата работы программы.
9. Термин «императивность» применительно к компьютерной технике означает: а) описание последовательности действий устройства; б) последовательное изменение состояний автомата некоторой схемой; в) описание в программе последовательности действий автомата, приводящих к результату; г) описание результата работы программы.
10. Операция «разрушающего присвоения» предполагает: а) изменение значения некоторой переменной без сохранения ее текущего значения; б) задание начального значения некоторой переменной; в) изменение значения некоторой переменной с предварительным сохранением ее текущего значения; г) копирование значения переменной.

11. Первыми языками программирования высокого уровня являются: а) Кобол; б) Фортран; в) PL/1; г) Си; д) Лисп
12. Первым универсальным языком программирования является: а) Алгол 68; б) Фортран 77; в) Паскаль; г) PL/1; д) Турбо-Паскаль.
13. Первым языком профессионального программирования является: а) Алгол 68; б) Фортран 77; в) Турбо-Паскаль; г) PL/1; д) Лисп.
14. Декларативными являются следующие языки программирования: а) Турбо-Паскаль; б) Пролог; в) SQL; г) Javascript; д) C++; е) Python; ж) Erlang.
15. Функциональными являются следующие языки программирования: а) Scala ; б) Пролог; в) SQL; г) Javascript; д) Лисп.
16. Основными свойствами объектов как экземпляров классов являются: а) ссылки на другие объекты; б) инкапсуляция; в) обмен сообщениями; г) полиморфизм; д) наличие объекта-предка.
17. Язык программирования PHP нашел наиболее широкое применение в: а) автоматизированном проектировании; б) разработке web-приложений; в) создании и управлении базами данных.
18. Параллельная программа – это… а) программа, работающая одновременно на нескольких компьютерах; б) программа, обрабатывающая большой объем данных; в) программа, осуществляющая обмен сообщениями в сети; г) программа, содержащая несколько процессов, работающих совместно.
19. К Case-средствам относятся: а) BPwin; б) SQL; в) Telelogic Statemate; г) Pycharm.

Глава 2. ЗНАКОМСТВО СО СРЕДОЙ РАЗРАБОТКИ PYCHARM И ЯЗЫКОМ ПРОГРАММИРОВАНИЯ PYTHON 3.8

Python — язык простой: у него лаконичный и в то же время понятный синтаксис.

Python применяют IT-гиганты: Яндекс, Google, Facebook, Dropbox, Mozilla и Microsoft. На нём написаны такие приложения, как YouTube, Instagram, PayPal, внутренние сервисы Facebook и сервисы Wargaming.

Знакомство с Python начнём с «Дзен Питона» Тима Питерса:

- Красивое лучше, чем уродливое.
- Явное лучше, чем неявное.
- Простое лучше, чем сложное.
- Сложное лучше, чем запутанное.
- Плоское лучше, чем вложенное.
- Разреженное лучше, чем плотное.
- Читаемость имеет значение.
- Особые случаи не настолько особые, чтобы нарушать правила.
- При этом практичность важнее безупречности.
- Ошибка никогда не должна замалчиваться. Если только вы сами этого не захотите.
- Встретив двусмысленность, отбрось искушение угадать.
- Должен существовать один и, желательно, только один очевидный способ сделать что-то. Хотя он поначалу может быть и не очевиден, если вы не голландец.
- Сейчас лучше, чем никогда. Хотя никогда зачастую лучше, чем прямо сейчас.
- Если реализацию сложно объяснить – идея плоха.
- Если реализацию легко объяснить – идея, возможно, хороша.
- Пространства имён – отличная штука! Будем делать их больше!

2.1. Установка Pycharm

PyCharm – это кросс-платформенная среда разработки (IDE), которая совместима с Windows, MacOS, Linux. Существуют – бесплатная версия *PyCharm Community Edition* и платная полнофункциональная версия *Professional Edition*. На наших занятиях мы будем

пользоваться бесплатной версией, скачать которую можно по ссылке: https://www.jetbrains.com/ru-ru/pycharm/

2.2. Создание проекта

Для создания проекта выполните последовательно следующие действия:

1. Запустите среду разработки PyCharm.
2. В открывшемся окне выберите Create new Project (рис.5)

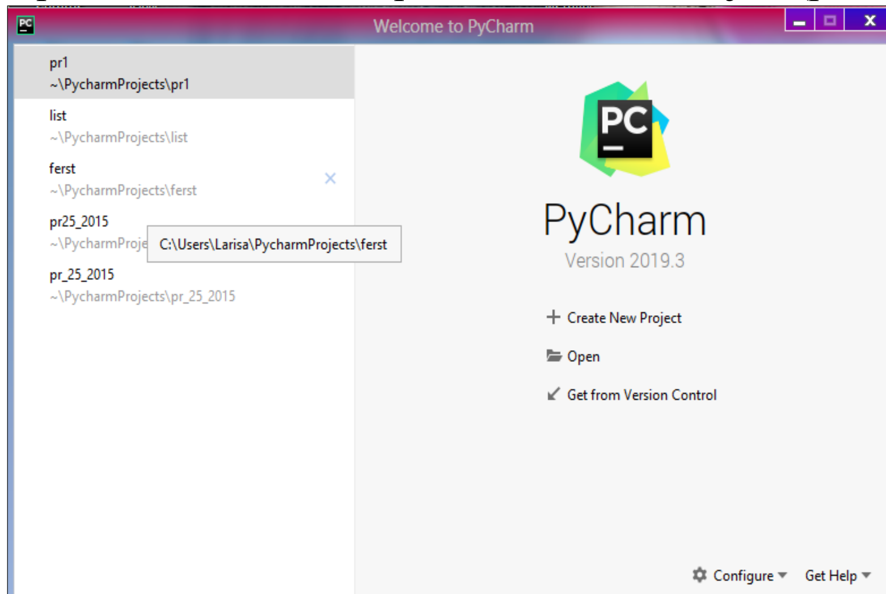


Рис.5. Окно запуска PyCharm

3. В открывшемся окне создания проекта задайте имя проекта, например, project1 (рис.6). Нажмите кнопку Create.

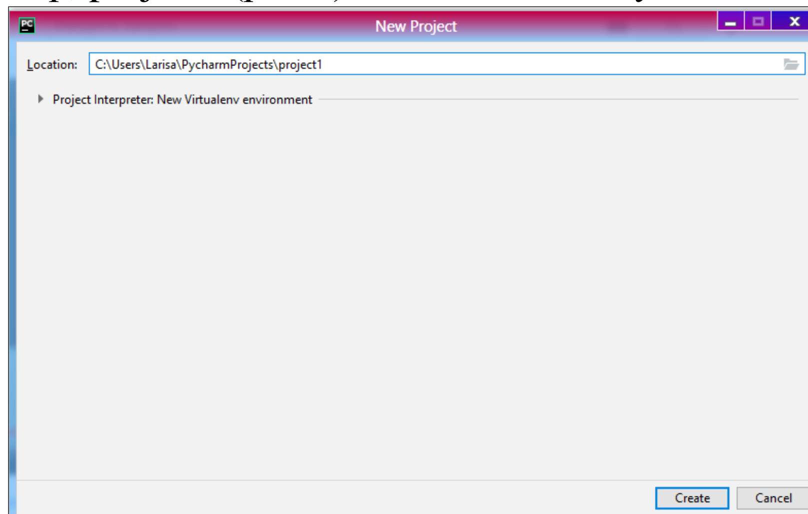


Рис.6. Создание проекта

4. Добавим к созданному проекту python-файл. Для этого встаем курсором на папку с проектом, нажимаем правую кнопку мыши, последовательно выбираем new, python file (рис.7).

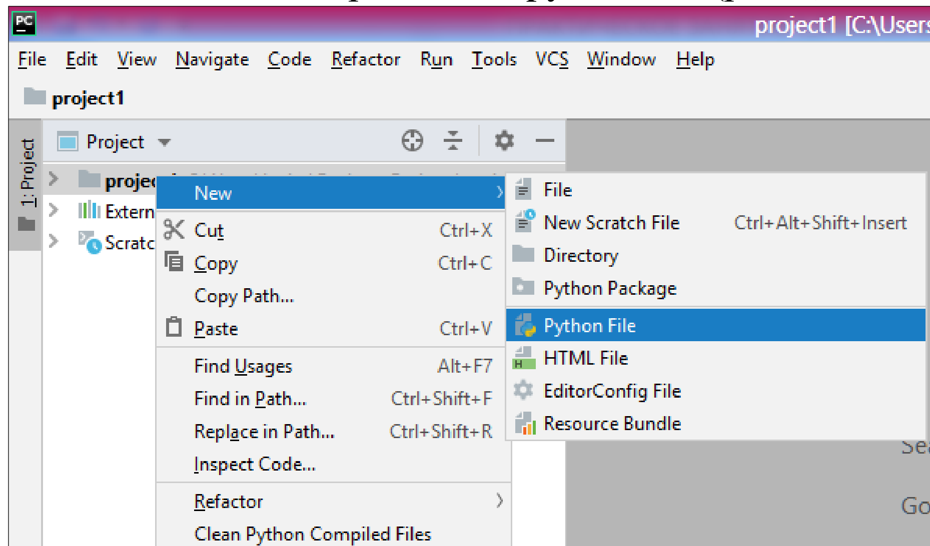


Рис.7. Добавление файла к проекту

5. Задаем имя файла, например, pr1 (рис.8). Нажимаем клавишу Enter.

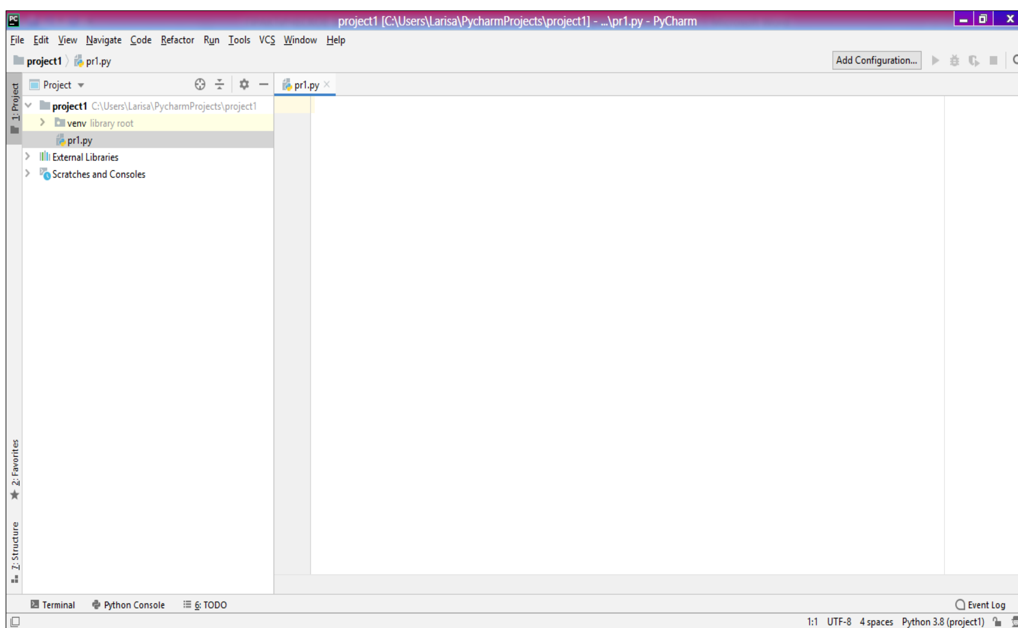


Рис.8. Задание имени добавляемому файлу

Можно писать код.

2.3. Сохранение, открытие и закрытие проекта

Для сохранения или открытия существующего проекта во вкладке File панели инструментов выберите команды Save as или Open соответственно (рис.5).

Закрыть существующий проект можно, последовательно выбрав в панели инструментов File, Close Project (рис.9).

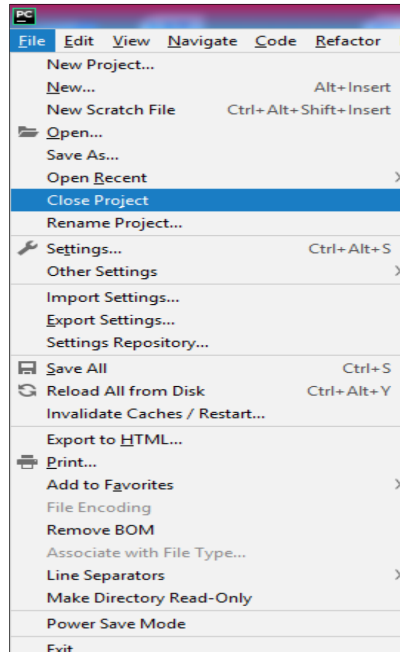


Рис.9. Окно закрытия проекта

2.4. Комментарии

Комментарий начинается с символа #.

Одна строка:

```
# Код вызова
print ('Здравствуй, МИР!')
```

Часть кода (три кавычки в англ. раскладке):

```
"""
print ('Здравствуй, МИР!')
print ('Привет, я Анфиса')
"""
```

Именование переменных

В именах переменных используйте только латинский алфавит, цифры и подчеркивание. Если название состоит из нескольких слов, разделяйте их символом нижнего подчёркивания: new_message. Такой стиль написания называется snake case, потому что слова могут получаться ***очень_длинные_как_змея_или_даже_как_две.***

Python допускает использование цифр в именах переменных, но не на первой позиции.

Составляйте названия переменных из английских слов. Когда ваша программа вырастет до многих сотен строк, названия-слова помогут быстро сориентироваться в коде.

Сравните сами: переменную с названием урока можно назвать буквой l, а можно — словом lesson. Второй вариант лучше: ведь то, что когда-то разработчик сократил lesson до одной буквы, со временем забудут и он, и его коллеги. Код станет сложнее для чтения.

Не рекомендуется использовать русские слова в английской раскладке. Рано или поздно ваш код будет читать человек, не владеющий русским. Он может не понять, что к чему. Сразу называйте переменные по-английски: child, а не транслитерацией – rebyonok.

Вывод на экран

Вывод числа и строки по отдельности рассмотрен в примерах выше.

Для того, чтобы вывести число и строку одновременно, надо преобразовать число в строку, например:

```
count=8
st1='У вас '
st2=' новых сообщений'
print (st1+ str(count) +st2)
```

Можно напечатать и без оператора +: в скобках функции print() перечислите через запятую аргументы, которые она должна напечатать. Запятая между аргументами по умолчанию заменяется на пробел. Например:

```
weather = 'облачно'
print('На улице сейчас', weather)
```

или

```
temperature = -25
weather = 'солнечно'
print('Сегодня',weather)
print('Температура воздуха',
temperature,'градусов')
```

2.5. Отладка программного кода в среде PyCharm

Отладка – этап разработки компьютерной программы, на котором обнаруживают, локализуют и устраняют ошибки. Чтобы понять, где возникла ошибка:

- узнают текущие значения переменных;
- выясняют, по какому из возможных путей выполнялась программа.

Существуют две взаимодополняющие технологии отладки:

- использование отладчиков – программ, которые включают в себя пользовательский интерфейс для пошагового выполнения программы: оператор за оператором, функция за функцией, с остановками на некоторых строках исходного кода или при достижении определённого условия.
- вывод текущего состояния программы с помощью расположенных в критических точках программы операторов вывода – на экран, принтер, громкоговоритель или в файл (журналирование).

На примере программного кода, представленного на рис.10, рассмотрим, как использовать для поиска и устранения ошибок встроенный в Pycharm отладчик. Для этого перейдем в программе на интересующую нас строку, например, 12. С этой строки начнем отладку.

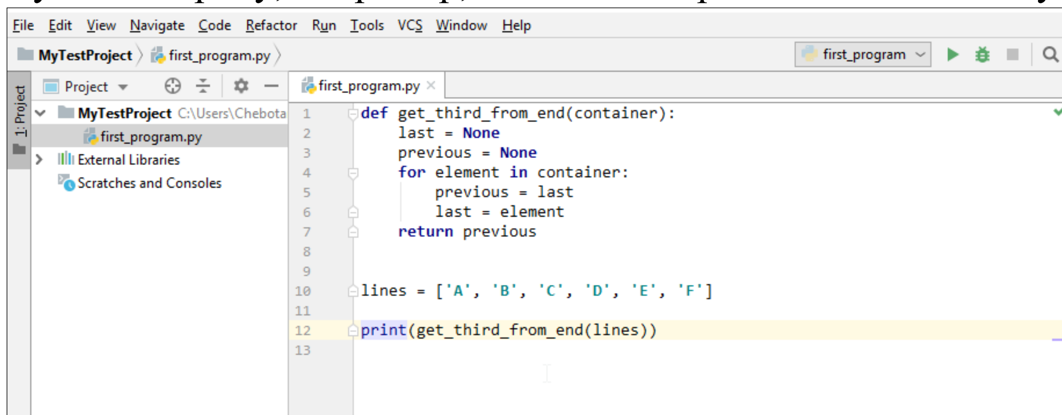


Рис. 10. Процесс отладки программы. Шаг 1

В верхней панели инструментов последовательно выберем *Run -> Toggle Line Breakpoint или Run -> Toggle Temporary Line Breakpoint* (рис.11)

Теперь рядом со строкой появилась красная жирная точка (рис.11). Это точка останова. Теперь при выполнении программы в режиме отладки среда остановит её в этом месте, и можно будет узнать состояние программы.

Нажмем правой кнопкой на названии файла с программным кодом и выберем строку Debug (рис.13). Теперь программа запустилась и остановилась на указанной строке. Текущее положение интерпретатора Python в программе отмечается синей строкой. В нижней части

экрана появилась вкладка отладки. Там виден список переменных, доступных из данной точки программы (рис.14).

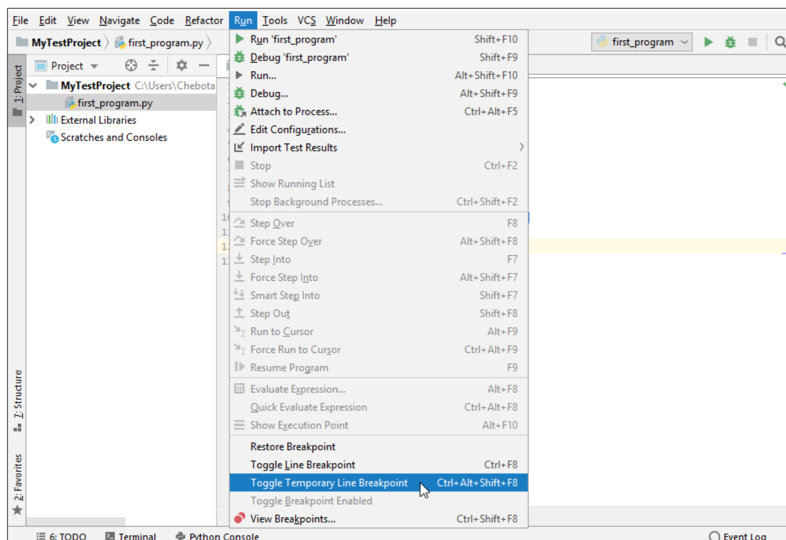


Рис. 11. Процесс отладки программы. Шаг 2

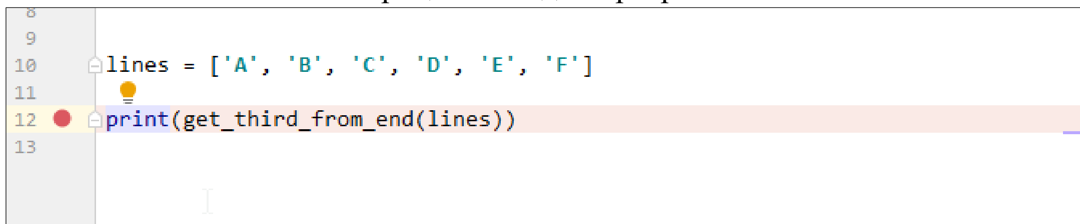


Рис.12. Процесс отладки программы. Шаг 3

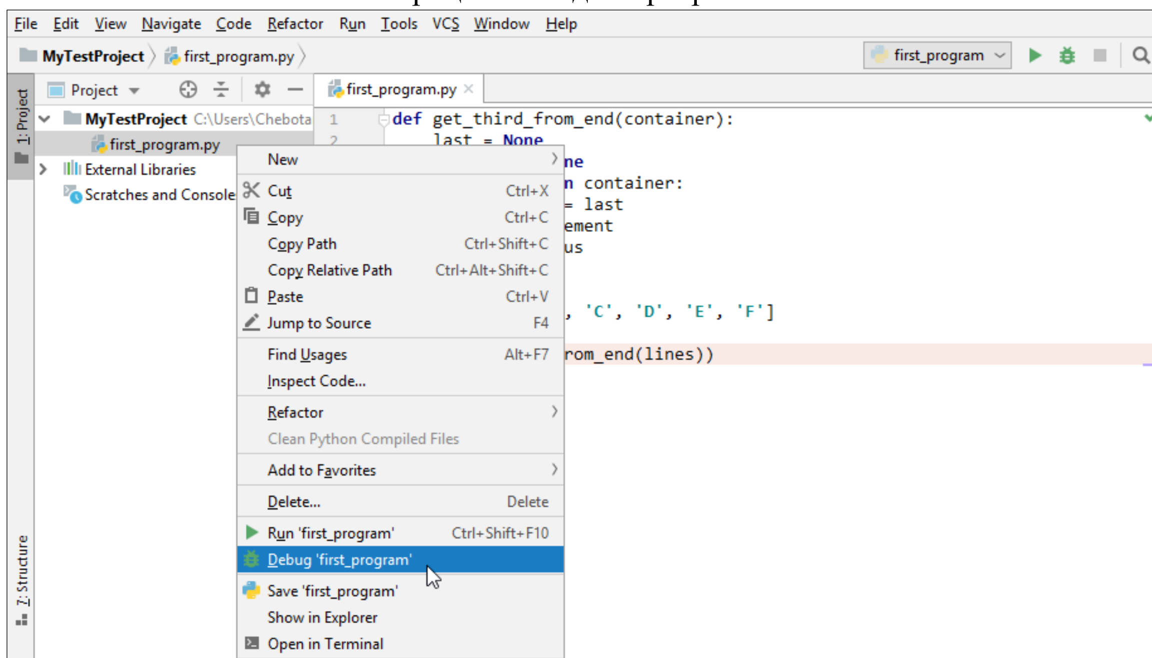


Рис. 13. Процесс отладки программы. Шаг 4

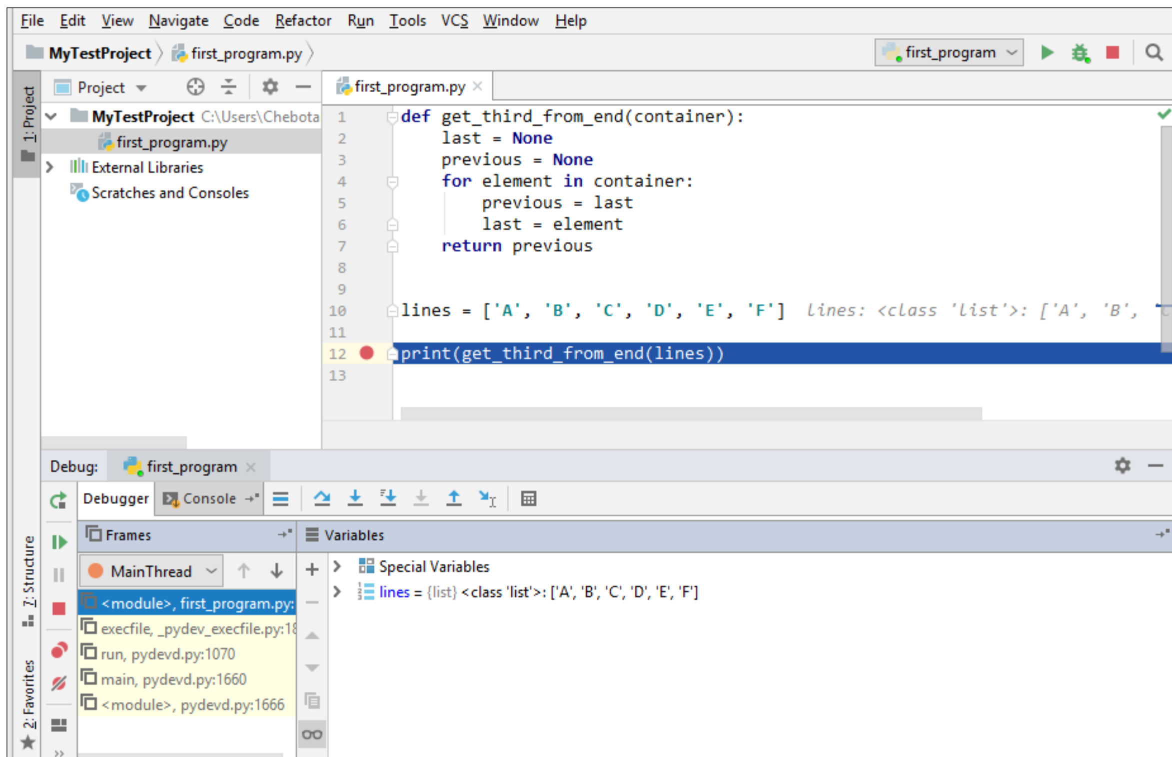


Рис. 14. Процесс отладки программы. Шаг 5

Для того, чтобы последовать, вслед за интерпретатором Python, внутрь функции (на строку 2), в главном меню выберите *Run -> Step Into*, или нажмите клавишу F7 (рис.15).

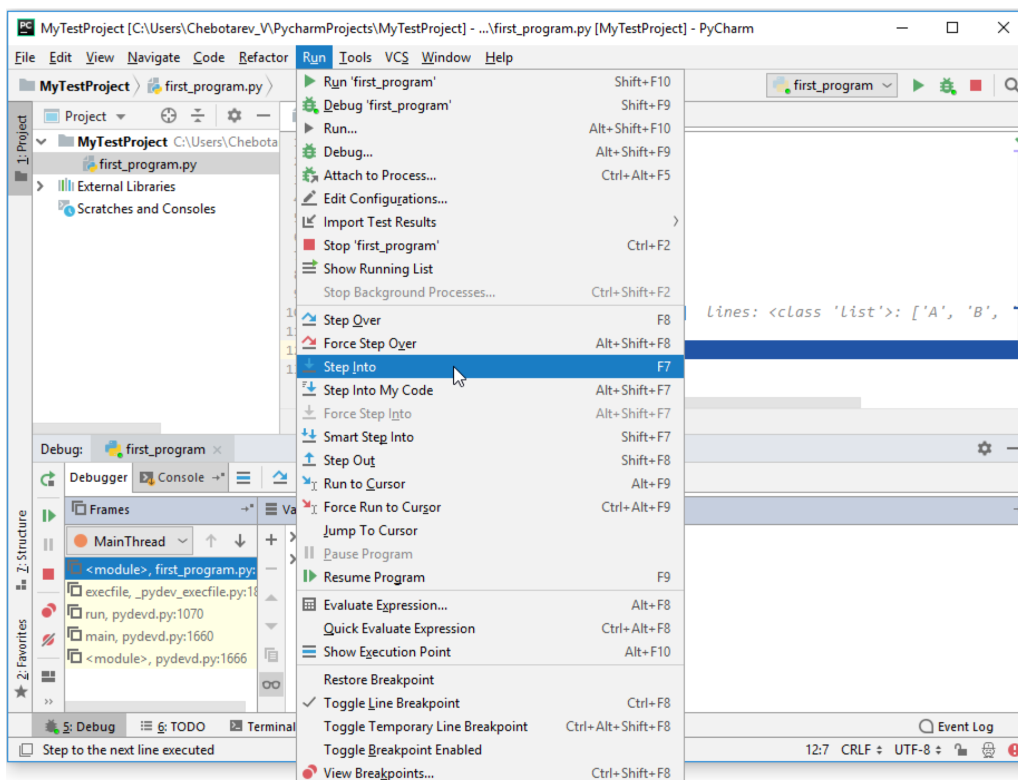


Рис. 15. Процесс отладки программы. Шаг 6

Как и ожидалось, интерпретатор переместил фокус своего внимания на строку 2, внутрь функции get_third_from_end() рис.16:

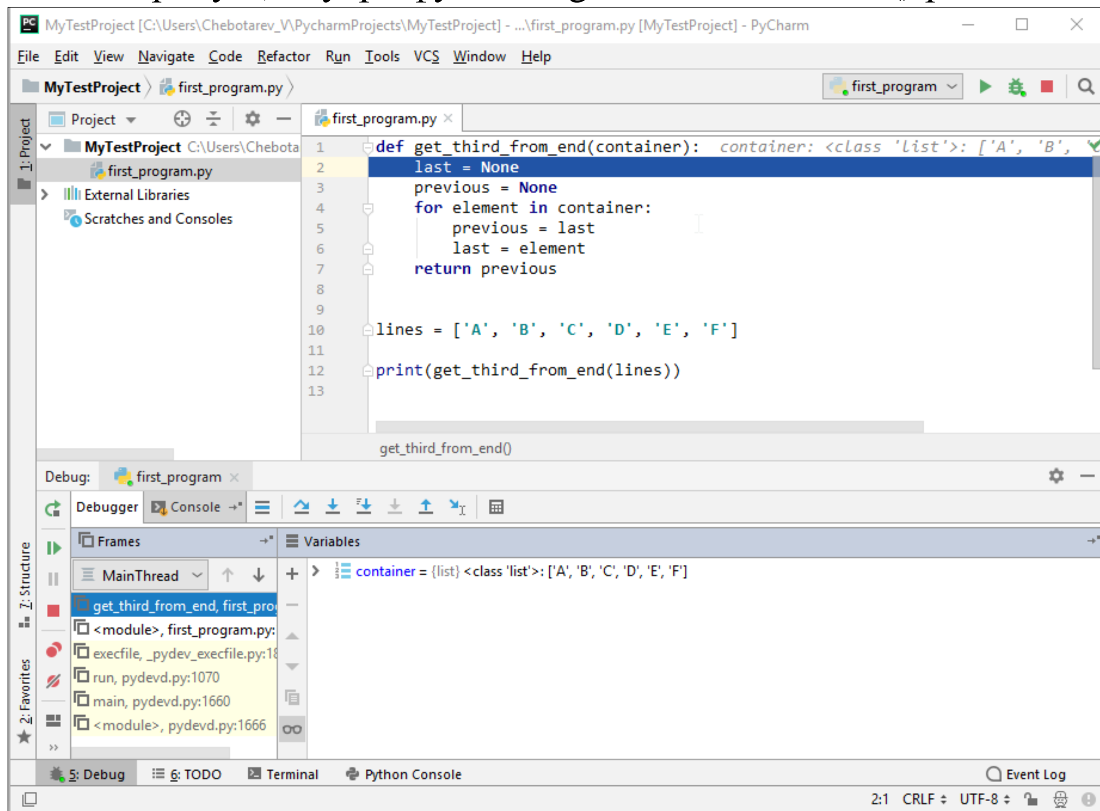


Рис. 16. Процесс отладки программы. Шаг 7

Чтобы сделать шаг вперёд, не заглядывая в используемые функции, используйте команду Run -> Step Over, или клавишу F8 (рис.17):

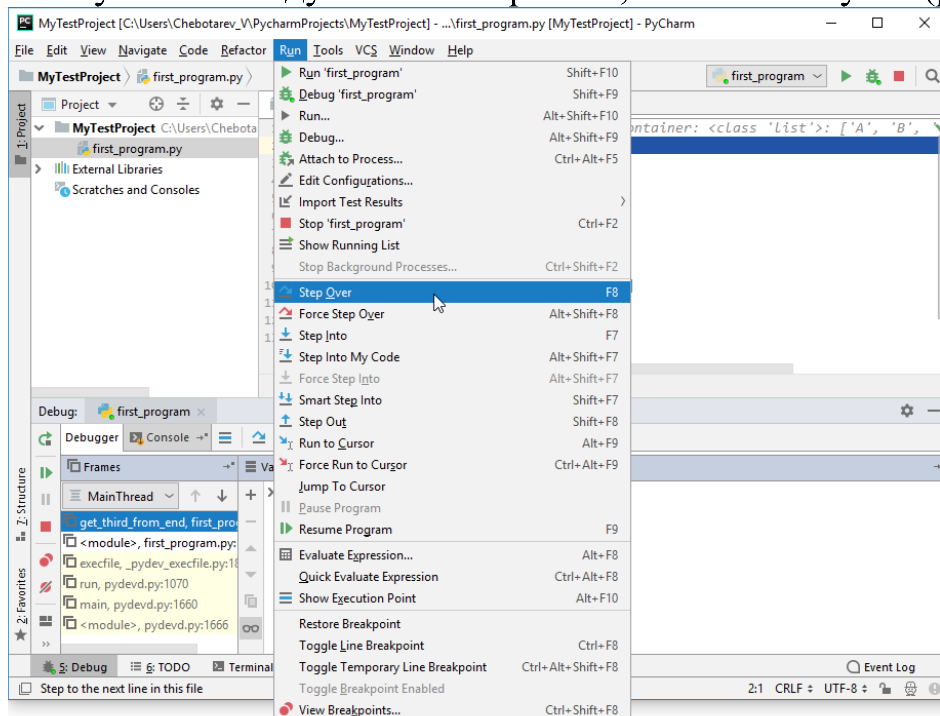


Рис. 17. Процесс отладки программы. Шаг 8

Теперь программа переместилась ещё на шаг вперёд. В нижней вкладке при этом появилась новая переменнаяl ast, она была создана в строке 2.

Таким образом можно легко узнавать, что происходит в каждой точке Вашей программы. Это часто необходимо для поиска ошибок, когда Вы не понимаете, почему программа выдаёт некоторый неверный ответ.

Практическая работа № 1
ЗНАКОМСТВО СО СРЕДОЙ PYCHARM и PYTHON 3.8

Задание

Воспроизвести порядок действий по созданию проекта из пп. 2.1, 2.2 и 2.3.

2.6. Дополнительные источники по IDE PyCharm

1. Постолит А. В. Python, Django и PyCharm для начинающих. - СПб.: БХВ-Петербург, 2021. -464 с.: ил. - (Для начинающих).
2. Установка, настройка и использование Pycharm https://www.youtube.com/watch?v=wquEFeQAjPQ
3. Особенности и возможности Pycharm. Режим доступа: https://www.youtube.com/watch?v=DpscmxH2LQU
4. Руководство по настройке и установке Pycharm. Режим доступа: https://www.jetbrains.com/help/pycharm/installation-guide.html
5. Инструкция по началу работы с Pycharm. Режим доступа: https://py-charm.blogspot.com/2017/09/blog-post.html
6. Отладка программ в среде разработки Pycharm. Режим доступа: https://pythonhelp.ru/post/2018-09-26-debugging/
7. Эффективная разработка на Python в среде Pycharm. Режим доступа:https://waksoft.susu.ru/2019/11/07/pycharm-effektivnaya-razrabotka-na-python/#running-code-in-pycharm

Вопросы для самоконтроля к главе 2

1. Какие библиотеки на базе Python позволяет использовать PyCharm? Кратко охарактеризуйте каждую из них.
2. Опишите последовательность шагов по созданию проекта на Python в среде PyCharm.

3. Назовите способы запуска и отладки программного кода в среде PyCharm.

Тест самоконтроля к главе 2

1. PyCharm поддерживает следующие библиотеки из перечисленных:
   а) Anaconda; б) Tensorflow; в) NumPy; г) Matplotlib; д) Kaggle; е) Colaboratory.
2. Составьте правильную последовательность команд по созданию проекта в среде PyCharm:
   а) Create; б) Create new Project; в) new; г) project1; д) python file.
3. Укажите способы запуска программы на Python в среде PyCharm (для Windows):
   а) Ctrl+Shift+F5; б) Ctrl+Shift+F10; в) Run main; г) F5.
4. В PyCharm программный код создается в контексте …
   а) проекта; б) консольного приложения; в) командного режима.
5. Интеллектуальным ассистентом кодирования в среде PyCharm является:
   а) Visual StudioInteliCode; б) Intelligent Coding Assistance; в) Whisper.
6. Процесс отладки программы в среде PyCharm начинается с…
   а) просмотра исключений; б) установки определенного значения для переменной; в) установки точки останова; г) использования консоли.
7. Встроенным инструментом отладки в среде PyCharm является…
   а) Intelligent Coding Assistance; б) Whisper; в) Debuger; г) Toggle Line Breakpoint.
8. Модулем тестирования (по умолчанию) в среде PyCharm является:
   а) unittest; б) pytest; в) nose; г) tox.
9. Фрейворком, созданным для редактирования проектов в среде PyCharm, является:
   а) Debuger; б) Whisper; в) Profiler; г) Alcazar.
10. Найти фрагмент текста в текущем файле в среде PyCharm можно:
    а) cmd+F; б) ctrl+F; в) ctrl+E; г) alt+F.

Глава 3. ОСНОВНЫЕ КОНЦЕПЦИИ ТЕСТИРОВАНИЯ

Невозможно представить себе процесс создания программного продукта без тестирования. Тестирование осуществляется практических на всех этапах его разработки, начиная с создания функций, методов и классов (модульное тестирование), и, заканчивая функциональным и нагрузочным тестированием уже готового, развернутого продукта.

Как вы уже поняли, существует большое количество различных видов тестирования. В данном пособии мы остановимся на некоторых из них.

Начнем с простого примера. Допустим, вы написали небольшую программу, вычисляющую периметр треугольника по трем введенным сторонам. В основной программе вы вводите длины сторон и выводите результат, в функции – вычисляете периметр. Правильность работы программы можно проверить, экспериментируя с вводимыми данными. Это называется *исследовательским тестированием*, является формой *ручного тестирования*.

Исследовательское тестирование – это форма тестирования, которая проводится без плана. В этом случае вы просто изучаете, как работает приложение.

Чтобы получить полный набор ручных тестов, вам необходимо составить список всех типов входных данных, которые могут быть аргументами функции, их возможных значений и все ожидаемые результаты.

Каждый раз, когда вы будете вносить изменения в свой код, вам нужно будет проверять правильность работы функции.

Не похоже на веселье, не так ли?

Вот где приходит на помощь *автоматизированное тестирование* – выполнение плана тестирования (частей вашего приложения, которые вы хотите протестировать, порядок их тестирования, и ожидаемые результаты) с помощью сценария тестирования. Такое тестирование называется *модульным (автономным)*, в терминологии Python – юнит-тестами (unit-tests).

Автономный тест – это часть кода, которая вызывает единицу работы (объект) и затем проверяет ее конечный результат. Если предположения о конечном результате не подтверждаются, считается, что автономный тест завершился неудачно. Объектом автономного тести-

рования может быть как единственный метод, функция или класс, так и их совокупность.

Важно понимать, что такое хороший автономный тест.

Хороший автономный тест должен обладать следующими свойствами:

– должен быть автоматизированным и повторяемым;
– его должно быть просто реализовать;
– должен сохранять актуальность и завтра;
– любой должен иметь возможность выполнить его одним нажатием кнопки;
– должен работать быстро;
– его результаты должны быть стабильны (тест всегда должен возвращать один и тот же результат, если между двумя последовательными запусками ничего не менялось);
– должен полностью контролировать тестируемую автономную единицу;
– должен быть полностью изолирован (работать независимо от других тестов);
– если тест завершается неудачно, то должно быть легко понять, каков ожидаемый результат и в каком месте искать ошибку.

Если ваше приложение содержит несколько функций, методов или классов то помимо проверки корректности работы каждого из них, вам надо проверить правильность их работы друг с другом. Это – *интеграционное тестирование*.

В силу своей специфики интеграционные тесты, в отличие от автономных, не являются быстрыми и стабильными и пользуются одной или несколькими реально существующими зависимостями между тестируемыми автономными компонентами.

Так, тесты, в которых используется системное время или реальная файловая система или реальная база данных, попадают в категорию интеграционных.

В Python есть инструменты и библиотеки, которые помогут вам создавать автоматизированные тесты для вашего приложения. В следующем параграфе мы рассмотрим эти инструменты и библиотеки.

3.1. Автономное тестирование с помощью модуля unittest Python. Тестовые случаи

Модуль unittest поддерживает автоматизацию тестов, использование общего кода для настройки и завершения тестов, объединение тестов в группы, а также позволяет отделять тесты от фреймворка для вывода информации.

Для автоматизации тестов, unittest поддерживает некоторые важные концепции[1]:

- испытательный стенд (test fixture) - выполняется подготовка, необходимая для выполнения тестов и любых связанных действий по очистке. Это может включать, например, создание временных или прокси-баз данных, каталогов или запуск серверного процесса;
- тестовый случай (test case) - это отдельная единица тестирования, проверяет конкретный ответ на определенный набор входных данных. Модуль unittest предоставляет базовый класс testcase, который можно использовать для создания новых тестовых случаев;
- набор тестов (test suite) - несколько тестовых случаев, наборов тестов или и того и другого, используется для объединения тестов, которые должны быть выполнены вместе;
- исполнитель тестов (test runner) - компонент, который управляет выполнением тестов и предоставляет пользователю результат. Может использовать графический или текстовый интерфейс или возвращать специальное значение, которое сообщает о результатах выполнения тестов.

Тестовый пример создается путем создания подклассов класса unittest.TestCase и определяются методами, имена которых начинаются со слова *test*. Это соглашение об именах информирует исполнителя тестов о том, какие методы представляют тесты.

Рассмотрим пример применения двух концепций тестового случая и набора тестов на примере разработки программы вычисления значения функции m(a):

$$m(a,b) = \begin{cases} \frac{a}{2}, a > 1 \text{ и } b = 0 \\ a + 1, a = 2 \text{ или } b > 1 \\ 0 \end{cases}$$

[1] https://docs.python.org/3/library/unittest.html#module-unittest

Из-за очевидной простоты алгоритма структурную схему программы можно получить на первом шаге декомпозиции рис. 18.

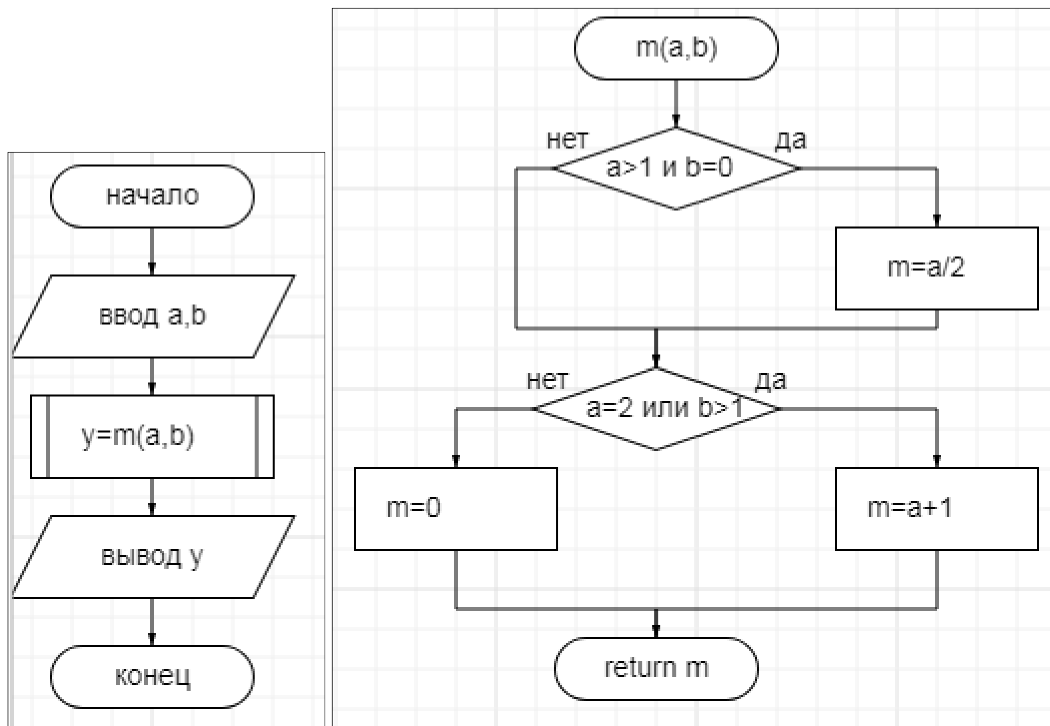


Рис.18. Блок-схема алгоритма программы примера

Соответствующий программный код представлен в файле first.py:

```
#first.py

#функция
def m (a,b):
 if a>1 and b==1:
    return a/2
 if a==2 or b>1:
    return a+1
 else: return 0

#основная программа
a= float(input())
b=float(input())
print (m(a,b))
```

Автоматизируем тесты с помощью модуля unittest. Для этого:

1) составим список всех данных, используемых в программе, и их типов. В нашем случае, это переменные вещественного типа a, b.

2) перечислим все случаи возможных значений этих данных:

1) a>1, b=0;
2) a>1, b≠0;
3) a<=1, b=0;
4) a<=1, b≠0;
5) a=2, b>1;
6) a=2, b<=1;
7) a≠2, b>1;
8) a≠2, b <=1.

3) Для каждого случая составим тесты:
   – a=2, b=0 – проверяет комбинацию (1);
   – a=2, b=1 – проверяет комбинации (2), (6);
   – a=2, b=3 – проверяет комбинацию (5);
   – a=1, b=0 – проверяет комбинацию (3);
   – a=1, b=1 – проверяет комбинации (4), (8);
   – a=1, b=3 – проверяет комбинацию (7).

4) для каждого случая укажем ожидаемый результат (таблица):

№	Тестовый случай (a,b)	Результат (y)
1.	a=2, b=0	3
2.	a=2, b=1	1
3.	a=2, b=3	3
4.	a=1, b=0	0
5.	a=1, b=1	0
6.	a=1, b=3	2

5) создадим файл для тестов с именем utest_first.py.

Содержание файла utest_first.py с тестами для каждого тестового случая:

```
import unittest
import first

class CalcTest(unittest.TestCase):
    def test_m1(self):
        self.assertEqual(first.m(2, 0), 3)
    def test_m2(self):
        self.assertEqual(first.m(2, 1), 1)
    def test_m3(self):
        self.assertEqual(first.m(2, 3), 3)
    def test_m4(self):
        self.assertEqual(first.m(1, 0), 0)
    def test_m5(self):
        self.assertEqual(first.m(1, 1), 0)
    def test_m6(self):
```

```
        self.assertEqual(first.m(1, 3), 2)

#точка входа командной строки
if __name__ == '__main__':
    unittest.main()
```

В примере с помощью ключевого слова class создан класс CalcTest объектом которого является unittest.TestCase (концепция тестового случая) – это то, на чем основан класс, или наследуется от него.

Ключевое слово *self* – это ссылка на текущий экземпляр класса. В таких языках как *Java*, *C#*, C++ аналогом является ключевое слово *this*. Через *self* вы получаете доступ к атрибутам и методам класса внутри него. Метод *assertEqual* сравнивает полученный результат с ожидаемым.

Если все шесть тестов завершились успешно, на экране монитора мы увидим *OK* (рис.19). Если на какой-то тест будет неуспешен, выведется соответствующее сообщение.

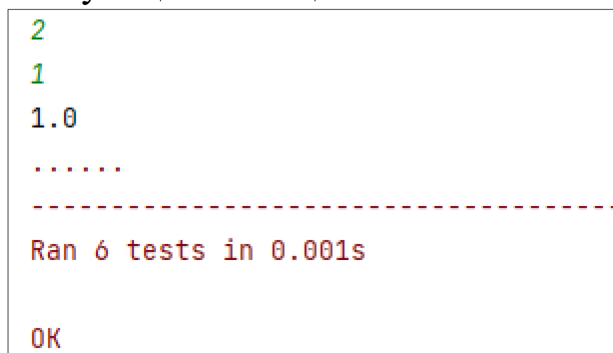


Рис.19. Результат работы модуля unittest для случая успешного завершения всех тестов

Если ваш проект состоит из нескольких файлов, в тестовый файл будет добавляться все большего количества тестов. Скоро вы обнаружите, что этот файл становится более большим по объему и сложным по структуре.

Поэтому лучше создать папку с именем **tests/** и разбить тесты на несколько файлов. Принято давать имя тестовому файлу начинающееся со слова **tests_**. Некоторые очень большие проекты разбивают каталог tests на несколько подкаталогов в зависимости от их назначения или использования.

В следующем параграфе познакомиться с методами класса TestCase более подробно.

3.2. Методы модуля unittest

Инструкции assert в Python — это булевы выражения, которые проверяют, является условие истинным или ложным. Если оно истинно, то программа переходит к выполнению следующей строчки кода. Если оно ложно, то программа останавливается и возвращает ошибку.

Инструкция *assert*:

- принимает в качестве аргументов выражение и сообщение, которое является необязательным аргументом;
- используется для проверки типов, значений аргумента и вывода функции;
- для отладки, т.к. приостанавливает программу в случае ошибки.

В качестве примера работы этой инструкции рассмотрим обработку списка (рис.20). Важно, чтобы аргумент функции *avg()* не был «пустым» (в нашем случае – пустым списком). Если этого не сделать, вернется ошибка *Assertion Error* и будет выведено заданное нами сообщение (рис. 21):

```
#для непустого списка

def avg(ranks):
    assert len(ranks) != 0,'Список не должен быть пустым'
    return round(sum(ranks)/len(ranks), 2)

#основная программа
ranks = [62, 65, 75]
print("Среднее значение:", avg(ranks))
```

Рис.20. Результат работы инструкции assert для непустого списка

```
#для пустого списка

def avg(ranks):
assert len(ranks) != 0,'Список не должен быть пустым'
return round(sum(ranks)/len(ranks), 2)
#основная программа
```

```
ranks = []
print("Среднее значение:", avg(ranks))
```

```
Traceback (most recent call last):
  File "C:/Users/Larisa/PycharmProjects/Spiski/l9.py", line 6, in <module>
    print("Среднее значение:", avg(ranks))
  File "C:/Users/Larisa/PycharmProjects/Spiski/l9.py", line 2, in avg
    assert len(ranks) != 0, 'Список не должен быть пустым'
AssertionError: Список не должен быть пустым

Process finished with exit code 1
```

Рис.21. Результат работы инструкции assert для непустого списка

Как видно из примеров, инструкция assert генерирует исключения Assertion Error. Их можно перехватывать и обрабатывать, как и любые другие исключения, с помощью блока try-except. Например (рис.22):

```
def avg(ranks):
    try:
        assert len(ranks) != 0
        return round(sum(ranks)/len(ranks), 2)
    except AssertionError:
        print ('Список не должен быть пустым')

#основная программа
ranks = []
print("Среднее значение:", avg(ranks))
#будет выведено
```

```
Список не должен быть пустым
Среднее значение: None

Process finished with exit code 0
```

Рис.22. Результат обработка исключения Assertion Error с помощью блока try-except

У инструкции есть методы. В предыдущем параграфе мы уже рассмотрели один из них – *assertEqual ()*. Рассмотрим другие методы инструкции *assert*.

Методы assertTrue () или assertFalse ()

Служат для проверки условия.

СИНТАКСИС:

assertTrue(expr, msg=None)

assertFalse(expr, msg=None)

где *expr* – выражение, значением которого может быть ИСТИНА (True) или ЛОЖЬ (False), что эквивалентно bool(expr) is True (False), а не *expr is True*.

msg – сообщение, не является обязательным аргументом.

Этого метода следует избегать, когда доступны более конкретные методы (например, *assertEqual(a, b)* вместо *assertTrue(a = = b))*, поскольку они обеспечивают лучшее сообщение об ошибке в случае сбоя.

В качестве базового рассмотрим пример, в котором, в зависимости от значения переменной testValue, тест считается успешным:

```
# unit test case
import unittest

class TestStringMethods(unittest.TestCase):

    # test function
    def test_positive(self):
        testValue = True

# текст сообщения, если тест успешен
        message = "Test value is true."
# если значение переменной=ИСТИНА
        self.assertTrue(testValue, message)


if __name__ == '__main__':
    unittest.main()
```

Значение переменной testValue – ИСТИНА, поэтому тест считается успешным.

```
    #Будет выведено
    .
---------------------------------------------------------
-
    Ran 1 test in 0.000s
    OK
    Process finished with exit code 0
```

В случае неуспешного прохождения теста будет выведено:

```
---------------------------------------------------------
-
```

```
    Traceback (most recent call last):
      File
"C:/Users/User/PycharmProjects/utest1/utest_list.p
y", line 11, in test_negative
        self.assertTrue(testValue, message)
    AssertionError: False is not true : Test value
is not true.
--------------------------------------------------
-
    Ran 1 test in 0.001s
    FAILED (failures=1)
    Process finished with exit code 1
```

Метод assertRaises ()

Используется тогда, когда необходимо определить, какая из функций вызвала конкретное исключение.

В примере ниже при вводе вещественного числа 4.5 показано, что именно функция parse_int() генерирует исключение. При вводе целого числа 5 программа завершается успешно (рис.23а, б):

```
# first.py

# функция вовзвращает целое число
def parse_int(s):
    return int(s)
s=input ()
parse_int(s)

#Содержание файла с тестами utest_first.py
import unittest
import first

class TestConversion(unittest.TestCase):

    def test_bad_int(self):
        self.assertRaises(ValueError,
            first.parse_int(), 'N/A')

    unittest.main()
```

```
4.5
Traceback (most recent call last):
  File "C:/Users/Larisa/PycharmProjects/p1/first.py", line 6, in <module>
    parse_int(s)
  File "C:/Users/Larisa/PycharmProjects/p1/first.py", line 4, in parse_int
    return int(s)
ValueError: invalid literal for int() with base 10: '4.5'

Process finished with exit code 1
```

Рис.23а. Результат работы программы при вводе вещественного числа

```
5

Process finished with exit code 0
```

Рис.23б. Результат работы программы при вводе целого числа

Этот метод можно использовать для создания отчетов по тестам. Более подробно с этими и другими методами можно познакомиться по ссылкам, представленным в п.3 параграфа 3.3.

3.3. Дополнительные источники по модулю unittest

1. Документация по использованию модуля unittest. Режим доступа: https://docs.python.org/3/library/unittest.html
2. Руководство по созданию тестов в коде на Python в среде PyCharm. Режим доступа: «Unit Tests and Doctests in PyCharm»
3. Методы модуля unittest. Режим доступа: https://docs.python.org/3/library/unittest.html#organizing-tests
4. Ошероув Р. Искусство автономного тестирования с примерами на C#. 2-е издание / пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2014. – 360 с.: ил.

Практическая работа № 2
UNIT-ТЕСТИРОВАНИЕ ПРОГРАММНОГО КОДА

Задание

Напишите программу, вычисляющую значение выражения. Шаги вычислений представьте в виде функций.

Оформите отчет по практической работе, содержащий:

– титульный лист;
– содержание;
– ответ на вопросы самоконтроля.

– для каждого варианта:
 - текст задания с указанием варианта;
 - блок-схему алгоритма решения задачи;
 - программный код на языке Python версии 3.8, соответствующий блок-схеме;
 - unit-тесты;
 - содержание файла utest_... .py с тестами для каждого тестового случая;
 - скриншоты результатов работы программы и тестового файла.

Варианты заданий

	Текст задания		Текст задания
1	$a = -1.3;\ b = -0.91;$ $c = 0.75; x = 2.32; k = 8$ $y = sin\frac{a-x}{c} + 10^4\sqrt[3]{\frac{a-kx^3}{2b}}$	2	k=2; x=0.32; d=1.25; n=-4; b=0.75; c=2.2 $y = 10^{-3}tgk - \frac{(x-d)(x^2+b^2}{\sqrt[3]{x^2+b^2-cd}}$
3	k=2; x=0.32; d=1.25; n=-4; b=0.75; c=2.2 $y = -\frac{coskx}{sin0.3} + d^n \cdot b\sqrt{c}$	4	$i = 5;\ k = -2; x = 0.1; a = 25$ b=2.35 $y = tgik - \frac{ax^3-b}{(a+b)^2} + 10^3e^{-5}$
5	$c = 0.05;\ d = 2.5; x = 1.35;$ $a = -1.25$ $y = \frac{\sqrt{\lvert c-d\rvert + (a+c)^2}}{\sin 2i} + 10^{-3}$	6	$i = 5;\ k = -2; x = 0.1; a = 25$ $y = i + \sqrt[3]{\frac{10^3\lvert xk\rvert}{a+b}}$
7	$c = 0.05;\ d = 2.5; x = 1.35;$ $a = -1.25$ $y = 10^{-3}e^{ix} - \frac{\lvert c-d\rvert + a^2}{\sqrt[3]{(a+c)^2}}$	8	$k = 2;\ c = 0.31; x = -2.5;$ $a = 0.93; b = 5.61$ $y = \frac{ln\lvert kx\rvert}{sin\ 0.9} - \sqrt{\lceil x-a^2\rceil}$
9	$k = 0.02;\ c = 0.31; x = -2.5;$ $a = 0.93; b = 5.61$ $y = -\frac{10^4a-b}{coskx} + \sqrt[3]{x-a^2}$+cx	10	$k = -2;\ a = 3.5; x = 1.523;$ $b = 0.35$ $y = 10^4\frac{ax}{b^2} - \left\lvert\frac{a-b}{kx}\right\rvert + \frac{ln3}{\sqrt{ax+b^2}}$

11	$k = -2;\ a = 3.5; x = 1.523;$ $b = 0.35$ $y = -e^{kx} + cosb\,\frac{3i+6}{ak^2 \log_a b}$	12	$a = 1.7; b = -1.25; c = -0.3;$ $x = 2.5; k = 3$ $y = \sqrt{\frac{abc}{2.4}} - \frac{0.7abc}{sin\frac{3\pi}{2}} - \lvert b - c\rvert$
13	$a = 1.7; b = -1.25; c = -0.3;$ $x = 2.5; k = 3$ $y = 10^4\sqrt[5]{\lvert coskx\rvert} - \frac{\lvert b-a\rvert}{kx}$	14	$a = 1.3; b = 2.42; c = 0.83;$ $x = 1.5; k = 2$ $y = \frac{\lvert a^2 - b^2\rvert}{\sin(0.001 \cdot kx)} - \frac{k^2}{e^{kx}}$
15	$a = 1.3; b = 2.42; c = 0.83;$ $x = 1.5; k = 2$ $y = \frac{k^2 + tg3k}{e^{kx}} - 10^4 - \frac{c}{x}$	16	$a = 1.3; b = 2.42; c = 0.83;$ $x = 1.5; k = 2$ $y = \sqrt[5]{\lvert\sin(k^{-2}x - bc\rvert} + \frac{\sqrt[3]{lnx + a^2}}{0.47x^2}$
17	$a = -2.5; b = 1.35; x = 2.75;$ $i = 3; c = -0.72$ $y = \frac{1.5(a-b)^2}{\lvert a-b\rvert c} + \frac{i}{5} + 10^3\sqrt{\lvert a\rvert}$	18	$a = -2.5; b = 1.35; x = 2.75;$ $i = 3; c = -0.72$ $y = 10^3\sqrt{\lvert a-b\rvert} - \frac{2.5(a+x^2)}{ix^2 + a^2bc}$
19	$a = 3.5; i = 2; b = -0.7; x = 0.8$ $y = 10^4(\sin i)^2 - \frac{0.32x^2 + 4x}{cosi^{-3} \cdot a}$	20	$a = 3.5; i = 2; b = -0.7; x = 0.8$ $y = 10^4(\sin i)^2\sqrt[6]{0.32x^3 - b} + \lvert b\rvert$
21	$a = 4.72; b = 1.25; d = -0.01;$ $x = 2.25; i = 2; k = 3$ $y = \frac{ax^2 + \lvert d\rvert}{(a+b)^2} - 10^4\sqrt[5]{\frac{kx}{(a+b)^2}}$	22	$a = 4.72; b = 1.25; d = -0.01;$ $x = 2.25; i = 2; k = 3$ $y = -\frac{cosi^{-2}}{sin(kx)^{-3}} + 10^4(\sin i)^2$

Вопросы для самоконтроля к главе 3

1. Перечислите порядок действий при создании проекта в PyCharm.
2. С какими операционными системами совместима среда PyCharm?

3. Какие библиотеки можно использовать в PyCharm?
4. Какие концепции тестирования поддерживает модуль unittest? Дайте краткую характеристику каждой из концепций.
5. Чем автоматизированное тестирование отличается от ручного?
6. Что характерно для исследовательского тестирования?
7. В чем разница между модульным и интеграционным тестированием?
8. Какие тесты попадают в категорию интеграционных? Приведите примеры.
9. Какие тесты можно считать модульными (автономными)?
10. С помощью какого метода можно проверить, какая функция сгенерировала исключение? Приведите пример.
11. Опишите алгоритм автоматизации тестов с помощью модуля unittest.
12. Какими методами модуля unittest проверяется истинность или ложность значения некоего выражения? Приведите пример.
13. С помощью какого метода модуля unittest можно проверить тестовые случаи? Приведите пример.

Тест самоконтроля к главе 3

1. Выберите верные утверждения о ручном тестировании: а) проверяет корректность одного компонента приложения; б) проверяет корректность каждого компонента приложения; в) проводится по строгому плану; г) проводится без строго плана; д) проверяет работу компонент друг с другом; е) быстро работают.
2. Выберите верные утверждения об автоматизированном тестировании: а) проверяет корректность одного компонента приложения; б) проверяет корректность каждого компонента приложения; в) проводится по строгому плану; г) проводится без строго плана; д) проверяет работу компонент друг с другом; е) быстро работают.
3. Выберите верные утверждения о модульном тестировании: а) проверяет корректность одного компонента приложения; б) проверяет корректность каждого компонента приложения; в) проводится по строгому плану; г) проводится без строго плана; д) проверяет работу компонент друг с другом; е) быстро работают.
4. Выберите верные утверждения об интеграционном тестировании: а) проверяет корректность одного компонента приложения; б) проверяет корректность каждого компонента приложения; в) прово-

дится по строгому плану; г) проводится без строго плана; д) проверяет работу компонент друг с другом.

5. Метод *assertEqual ()* служит для: а) проверки условий; б) отладки; в) для определения функции, вызвавшей исключение; г) создания отчетов по тестам; д) сравнения полученного результата с ожидаемым.

6) 5. Метод *assertTrue ()* служит для: а) проверки условий; б) отладки; в) для определения функции, вызвавшей исключение; г) создания отчетов по тестам; д) сравнения полученного результата с ожидаемым.

7. Метод *assertRaises ()* служит для: а) проверки условий; б) отладки; в) для определения функции, вызвавшей исключение; г) создания отчетов по тестам; д) сравнения полученного результата с ожидаемым.

Глава 4. ПОРЯДОК ИСПОЛНЕНИЯ ПРОГРАММЫ, УСЛОВНЫЕ ОПЕРАТОРЫ, ЦИКЛЫ

4.1. Ветвления

В языке Python существует несколько типов ветвлений. Рассмотрим каждый из них.

Ветвление объявляется условным оператором if, который служит для выбора направления работы программы в зависимости от условий, сложившихся в данной точке программы на момент ее выполнения.

Общая форма записи условного оператора:

if <условие>:

<блок кода1, который выполнится, если условие истинно>

 # Внимание: этот блок имеет отступ в 4 пробела от начала строки

else:

<блок кода2, который выполнится, если условие ложно>

 # Этот блок тоже отбит от края строки 4 пробелами

Если на момент выполнения условие истинно, программа передает управление блоку кода1 и далее первому оператору за пределами конструкции if-else. При этом блок кода2 не выполняется. Если на момент выполнения условие ложно, выполняется блок кода2, а блок кода1 не выполняется. В табл. 1 указаны простейшие операции отношения.

Таблица 1. Операции отношения

Python	Значение
<	Меньше чем
≤	Меньше или равно чем
=	Равно
≥	Больше или равно чем
>	Больше чем
≠	Не равно

Сокращенные варианты записи

При программировании обыденной является ситуация, когда требуется некоторое действие в ответ на сложившиеся условия. Например, если получены неверные исходные данные от пользователя, то выдать сообщение об ошибке и выйти из программы. В таких

случаях используется сокращенная запись оператора условия с отсутствующим блоком else:

if <условие>:
<блок кода, который выполнится, если условие вернуло True>
 # Внимание: этот блок имеет отступ в 4 пробела от начала строки

Здесь в случае истинности условия управление передается блоку кода. В случае ложности условия этот блок пропускается.

Составные логические выражения

В программировании распространены двойные условия, которые в математике записываются в виде f<b<c. В программе такие условия должны быть переформулированы с использованием простых операций сравнения и логических операций «И», «ИЛИ», «НЕ». Обозначение логических операций и пример их использования приведены в таблице 1.

Таблица 2. Логические операторы

Python	Значение
OR (ИЛИ)	Логическое сложение
AND (И)	Логическое умножение
NOT (НЕ)	Отрицание

Например:

```
a=int (input())
b=int (input())

if (a<b):
    print ('меньшее число a')
    else:
        print ('меньшее число b')
```

Вложенные операторы условия

Операторы условия могут быть вложенными друг в друга в соответствии с тем программным алгоритмом, который они реализуют. Допускается произвольная степень их вложенности. Например:

```
a=int (input())
b=int (input())
x=int (input())
```

```
if (a<=b):
    if x>=a and x<=b:
        print ('x принадлежит отрезку ab')
```

Множественное ветвление

Используется в том случае, если в программе присутствует большое дерево ветвлений и все ветвления зависят от значения какой-либо одной переменной:

```
b=int(input())
if b==0:
    print ('штиль')
elif b==1:
    print ('тихий ветер')
elif b==2:
    print('легкий ветер')
```

Как только выполниться одно из условий – все нижеследующие elif пропускаются.

Можно предусмотреть действия в случае, если ни одно условие не будет выполняться. Для этого в код добавляется ветвь else:

```
b=int(input())
if b==0:
    print ('штиль')
elif b==1:
    print ('тихий ветер')
elif b==2:
    print('легкий ветер')
else:
    print ('введено неверное значение')
```

4.2. Циклы

Действие циклов заключается в последовательном повторении определенной части программы некоторое количество раз. Повторение продолжается до тех пор, пока выполняется соответствующее условие. Когда значение выражения, задающего условие, становится ложным, выполнение цикла прекращается, а управление передается оператору, следующему непосредственно за циклом.

В Python существуют два типа циклов: for, while. В отличие от многих Си-подобных языков и от привычного многим Pascal, в Python нет аналога цикла do-while или repeat-until.

Цикл for организует выполнение фрагмента программы фиксированное количество раз. Как правило, этот тип цикла используется тогда, когда число повторений известно заранее. Соответственно, while – когда число повторений заранее не известно.

Чтобы программа поняла, что сейчас начнётся цикл — нужно сообщить ей об этом специальными словами: **объявить цикл**.

Объявление цикла for

Цикл for в Python объявляется ключевыми словами for и in, после них задаётся **условие цикла**. После условия ставится двоеточие.

for переменная in список_элементов:

В условии цикла после for указывают имя переменной, в которую будут поочерёдно передаваться элементы из обрабатываемого списка, а после in ставится имя списка, который надо обработать. Цикл автоматически прекратит работу, когда переберёт все элементы списка.

Имя переменной в цикле вы можете дать любое, но традиционно эти имена образуют от имени обрабатываемого списка, в единственном числе. Например, если список называется musicians, то переменную лучше назвать musician; если список называется pigs — переменную называют pig.

Тело цикла for

За условием с новой строки следует тело цикла. Каждая строка тела цикла отбивается от начала строки четырьмя пробелами:

for переменная in список_элементов:

код, который выполняется для каждого элемента

Примеры использования цикла for

```
bremen_musicians = ['Трубадур', 'Петух',
'Кот','Пес', 'Осел']
print ('Представляем музыкантов:')
for musician in bremen_musicians:
    print (musician)
```

Будет выведено (рис.24)

```
Представляем музыкантов:
Трубадур
Петух
КотПес
Осел

Process finished with exit code 0
```

Рис.24. Пример работы цикла for

Переменная musician принимает последовательно значения всех элементов из списка bremen_musicians.

Четыре отступа перед вложенным кодом

Код, который размещён в теле цикла, отбивается от начала строки **четырьмя отступами**. По этим отступам Python определяет, где начинается и кончается код, относящийся к телу цикла.

Это строгое техническое условие: без отступов в блоках кода программа просто не заработает.

Цикл берёт значение первого элемента из списка bremen_musicians и передаёт его в переменную musician. Затем выполняется код в теле цикла: печатается содержимое переменной musician.

После этого начнётся новый «круг», со следующим элементом списка. И так будет продолжаться до тех пор, пока цикл не переберёт весь список.

Каждый такой «круг» называется **итерацией цикла**.

Когда список закончится — программа выйдет из цикла; после этого сработает код, который написан после цикла.

Пример кода:

```
for i in range (1,6): # range - диапазон
    print (i)

#будет напечатано
1
2
3
4
5
```

Как видно из примера, range (a,b) возвращает числа от a b-1.
Функция reversed () «переворачивает» списки и диапазоны значений.
Например:

```
for i in reversed (range (1,6)):
        print (i)

#будет напечатано
5
4
3
2
1
```

Числа могут быть и отрицательными, важно лишь, чтобы первое число было меньше второго. Также вместо списка в цикл можно передать переменную, в которой сохранен список. Например:

```
around_zero = range (-3, 3)

#  Вместо  списка  в  цикл  передаётся  переменная
around_zero
for i in around_zero:
    # перебрать все числа в диапазоне от -3 до 3
    print (i)

#будет напечатано
-3
-2
-1
0
1
2
```

Объявление цикла while

Цикл while в Python объявляется ключевыми словами while, после которого задаётся **условие цикла**. После условия ставится двоеточие: *while условие:*

здесь условие является выражением логического типа, определяющим условие выполнения цикла. Цикл автоматически прекратит работу, когда условие перестанет выполняться.

Тело цикла while

За условием с новой строки следует тело цикла. Каждая строка тела цикла отбивается от начала строки четырьмя пробелами:

while условие:

код, который выполняется

тело цикла выполняется до тех пор, пока выполняется условие цикла.

Примеры использования цикла while

Следующая программа демонстрирует механизм работы цикла while:

```
i = 0

while i < 10:
    print (i)
    i = i + 1
```

Пока значение переменной i меньше десяти, оно выводится на экран. Как только значение превысит число 9, произойдет завершение цикла.

Практическая работа № 3
УСЛОВНЫЕ ОПЕРАТОРЫ. ЦИКЛЫ

Задание. В вариантах 1-3, 5-6, 8-11, 13-16, 19-21 вывести на экран в виде таблицы значения функции F на интервале от $x_{нач}$ до $x_{кон}$ с шагом dx, где a, b, c – действительные числа. Значения $x_{нач}$, $x_{кон}$,dx, a, b, c ввести с клавиатуры.

В вариантах 4, 7,12, 17, 18, 22 вывести на экран результат работы программы также в виде таблицы.

Оформите отчет по практической работе, содержащий:

- титульный лист (Приложение 1);
- содержание;
- для каждого варианта:
  - текст задания с указанием варианта;
  - блок-схему алгоритма решения задачи;
  - программный код на языке Python версии 3.8, соответствующий блок-схеме;
  - тесты;
  - скриншоты результатов работы программы для каждого тестового случая.

ВАРИАНТЫ ЗАДАНИЙ

№	Функции	№	Функции
1	$F=\begin{cases} ax^2+2\,, x<0, b\neq 0 \\ \dfrac{x-a}{x-c}, x>0, b=0 \\ \dfrac{x}{c} \text{ в остальных случаях} \end{cases}$	2	$F=\begin{cases} ax^2+b^2x, a<0, x\neq 0 \\ x-\dfrac{a}{x-c}, a>0, x=0 \\ \dfrac{x}{c} \text{ в остальных случаях} \end{cases}$
3	$F=\begin{cases} \dfrac{1}{ax}-b, x+5<0, c=0 \\ \dfrac{x-a}{x}, x+5, c\neq 0 \\ \dfrac{10x}{c-4} \text{ в остальных случаях} \end{cases}$	4	Даны положительные числа A, B, C. Вычислить, сколько квадратов со стороной C можно разместить в прямоугольнике размера A x B. Операции умножения и деления не использовать.
5	$F=\begin{cases} ax^2+bx+c, a<0, c\neq 0 \\ -\dfrac{a}{x-c}, a>0, c=0 \\ a(x+c) \text{в остальных случаях} \end{cases}$	6	$F=\begin{cases} ax^2+\dfrac{b}{c}, x<1, c\neq 0 \\ \dfrac{x-a}{(x-c)^2}, x>1.5, c=0 \\ \dfrac{x^2}{a} \text{ в остальных случаях} \end{cases}$

7	Программу проверки числа на простоту можно значительно усовершенствовать, если проверять делимость числа только на 2 и на все нечетные числа, а также, если воспользоваться тем фактом, что любое составное число n имеет нетривиальный делитель, не превосходящий n0.5 (докажите). Напишите программу проверки числа на простоту с учетом этих требований.	8	$F = \begin{cases} ax^5 + b^4 + c, x < 0.6, b + c \neq 0 \\ \frac{x-a}{x-c}, x > 0.6, b + c = 0 \\ \frac{x}{c} + \frac{x}{a} \text{ в остальных случаях} \end{cases}$
9	$F = \begin{cases} a - \frac{x}{10+b} x < 0, b \neq 0 \\ \frac{x-a}{x-c}, x > 0, b = 0 \\ 3x + \frac{2}{c} \text{ в остальных случаях} \end{cases}$	10	$F = \begin{cases} ax^2 + b, x - 1 < 0, b - x \neq 0 \\ \frac{x-a}{x}, x - 1 > 0, b + x = 0 \\ \frac{x}{c} \text{ в остальных случаях} \end{cases}$
11	$F = \begin{cases} ax^2 + b^2x, c < 0, b \neq 0 \\ \frac{x+a}{x+c}, c > 0, b = 0 \\ \frac{x}{c} \text{ в остальных случаях} \end{cases}$	12	Напишите программу, которая по данному целому n>1 находит разложение числа n на простые множители (печатает все простые делители n с учетом кратности).
13	$F = \begin{cases} -ax^2 - b, x < 5, c \neq 0 \\ \frac{x-a}{x}, x > 5, c = 0 \\ \frac{-x}{c} \text{ в остальных случаях} \end{cases}$	14	$F = \begin{cases} -ax^2 + b, x < 0, b \neq 0 \\ \frac{x}{x-c} + 5.5, x > 0, b = 0 \\ \frac{x}{-c} \text{ в остальных случаях} \end{cases}$
15	$F = \begin{cases} -ax^2, c < 0, a \neq 0 \\ \frac{a-x}{cx}, c > 0, a = 0 \\ \frac{x}{c} \text{ в остальных случаях} \end{cases}$	16	$F = \begin{cases} a(x+c)^2 - b, x = 0, b \neq 0 \\ \frac{x-a}{-c}, x = 0, b = 0 \\ a + \frac{x}{c} \text{ в остальных случаях} \end{cases}$
17	Для чисел 23436, 190187200, 380457890232 вычислите и выведите на экран их делители	18	Дано натуральное число N>1. Найти наименьшее целое число K, при котором выполняется неравенство 3k > N
19	$F = \begin{cases} ax^2 - bx + c, x < 3, b \neq 0 \\ x - \frac{a}{x}, a > 0, x > 3 \\ 1 + \frac{x}{c} \text{ в остальных случаях} \end{cases}$	20	$F = \begin{cases} -ax - c, c < 0, x \neq 0 \\ \frac{x-a}{-c}, c > 0, x = 0 \\ \frac{bx}{c-a} \text{ в остальных случаях} \end{cases}$
21	Напишите программу для вычисления значений членов бесконечного ряда $x \cdot \frac{x^2}{2!} \cdot \ldots \cdot \frac{x^n}{n!}$ до члена ряда, не превышающего точность ε.	22	Среди чисел 1< N <100 найти все пары чисел, для которых их сумма равна их произведению.

Вопросы для самоконтроля к главе 4

1. Для каких видов программирования алгоритмов применяется условный оператор (оператор цикла)?
2. Назовите основные разновидности условных операторов Python и форматы их записи.
3. В каких случаях какой вид условного оператора используется? Приведите примеры.
4. Назовите основные разновидности циклов Python и форматы их записи.
5. В каких случаях какой вид цикла используется? Приведите примеры.
6. Что такое вложенные циклы? Приведите примеры.

Тест самоконтроля к главе 4

1.В результате работы программы на экран будет выведено:

```
num1=4
num2=6
num1+=num1
num2*=num1
print (num1)
```

a) 100; б) 10; в) 200; г) 60.

2.В результате работы программы на экран будет выведено:

```
total=0
for i in range (1,6):
   total+=i
print (total)
```

b) 100; б) 10; в) 15; г) 60.

3. В результате работы программы на экран будет выведено:

```
total=0
for i in range (1,6):
   total+=i
 print (total, end='')
```

c) 100; б) 1361015; в) 18; г) 123456.

4. Условный оператор применяется для программирования алгоритмов:

а) сложных ; б) линейных; в) ветвящихся; г) циклических; д) верного ответа нет.

5. Ветвящийся алгоритм подразумевает: а) выполнение лишь нескольких, удовлетворяющих заданному условию частей программы; б) неоднократное повторение отдельных частей программы; в) последовательное выполнение всех элементов программы; г) верного ответа нет.
6. Циклический алгоритм подразумевает: а) выполнение лишь нескольких, удовлетворяющих заданному условию частей программы; б) неоднократное повторение отдельных частей программы; в) последовательное выполнение всех элементов программы; г) верного ответа нет.
7. Составьте правильную последовательность написания оператора ветвления в Python: а) Оператор 2; б) else; д) if; е) : ; ж) then; з) Оператор 1; и) Условие.
8. Что выведет данный код:

```
if 10>2:
   print (1)
else:
   0
```

а) 1; б) 0; в) ошибку; г) ничего.

9. Что выведет данный код:

```
if 10>0:
   pass
else:
   pass
elif 10>9:
   print(1)
```

а) 1; б) 0; в) ошибку; г) ничего.

10. Что выведет данный код:

```
if False:
   print (1)
else:
   print (0)
```

а) 1; б) 0; в) ошибку; г) ничего.

11. Что выведет данный код:

```
if True:
   print (1)
elif:
   print (0)
```

а) 1; б) 0; в) ошибку; г) ничего.

12.Что выведет данный код:

```
if 0:
   print (0)
elif 1:
   print (1)
else:
   print (2)
```

а) 2; б) 1; в) ошибку; г) 0.

13. что выведет данный код:

```
if bool("False")==bool("true"):
   print (bool("False"))
else:
   print (0)
```

а) False; б) 0; в) ошибку; г) True.

14. Определите, что будет напечатано в результате выполнения следующего кода:

```
s = 0
for k in range(3,11):
     s = s + k
print(s)
```

а) 52; б) 0; в) ошибку; г) 45.

15. Определите, что будет напечатано в результате выполнения следующего кода:

```
s = 0
for k in range(-5,11):
     s = s + 2 * k
print(s)
```

а) 52; б) 80; в) ошибку; г) 45.

16. Определите, что будет напечатано в результате выполнения следующего кода:

```
s = 1
for k in range(1,30):
     s = (k-5) * s
print(s)
```

а) 52; б) 80; в) ошибку; г) 0.

17. Определите, что будет напечатано в результате выполнения следующего кода:

```
s = 1
for k in range(30):
```

```
    s = (-1) * s
print(s)
```
а) 52;   б) 1;   в) ошибку;   г) 0.

18. Определите, что будет напечатано в результате выполнения следующего кода:

```
z = 30
for n in range(30):
    if n > 10:
        z = z – n
            else:
                z = z + n
print(z)
```
а) 52;   б) 1;   в) -321;   г) 35.

19. Определите, что будет напечатано в результате выполнения следующего кода:

```
z = 30
for n in range(10):
    if n < 0:
        z = z-2 * n
        else:
        z = n - z
print(z)
```
а) 52;   б) 1;   в) -321;   г) 35.

20. Определите, что будет напечатано в результате выполнения следующего кода:

```
a = 23
b = 4
while a > b:
    if a % 2 == 0:
        b = b + a
    else:
        a = a-2 * b + 1
print(b)
```
а) 20;   б) 1;   в) -321;   г) 35.

21. Определите, что будет напечатано в результате выполнения следующего кода:

```
s = 0
m = 123
while m > 0:
```

```
    d = m % 10
    s = s + d
    m = m // 10
print(s)
```
а) 20; б) 6; в) -21; г) 35.

22. Определите, что будет напечатано в результате выполнения следующего кода:

```
c = 0
m = 123
while m > 1:
    d = m % 10
    c = (c + d) * 10
    m = m // 10
print(c)
```
а) 20; б) 6; в) 320; г) 35.

23. В результате выполнения программы, записанной ниже, на экран будет выведено два числа A и B. Укажите такое наибольшее число x, при вводе которого на экран будет выведено сначала 3, а потом 5.

```
x = int(input())
A = 0
B = 0
while x > 0:
    A = A + 1
    if B < x % 10:
        B = x % 10
    x = x // 10
print(A)
print(B)
```
а) 555; б) 6; в) 320; г) 35.

24. Определите, что будет напечатано в результате выполнения следующего кода:

```
print(23 // 7)
```
а) 555; б) 6; в) 320; г) 3.

25. Логические операторы записаны как and, not, or, xor и пронумерованы, начиная с единицы. Расположите их согласно приоритету выполнения.

а)1234; б) 2133; в) 3421; г) 4312.

Глава 5. ПЛАНОВАЯ ОБРАБОТКА ОШИБОК ПРИ ПОМОЩИ ИСКЛЮЧЕНИЙ В PYTHON

В *Python* выделяют два различных вида ошибок: *синтаксические* ошибки и *исключения*.

Синтаксические ошибки возникают в случае, если программа написана с нарушениями требований *Python* к синтаксису.

Исключениями (*exceptions*) в языках программирования называют проблемы, возникающие в ходе выполнения программы, которые допускают возможность дальнейшей ее работы в рамках основного алгоритма. Типичным примером исключения является деление на ноль, невозможность считать данные из файла (устройства), отсутствие доступной памяти, доступ к закрытой области памяти и т.п. Для обработки таких ситуаций в языках программирования, как правило, предусматривается специальный механизм, который называется обработка исключений (*exception handling*).

Они возникают в случае, если синтаксически программа корректна, но в процессе выполнения возникает ошибка (деление на ноль и т.п.).

Ошибки во время работы программы неизбежны. Поэтому существенной частью любой программы является их обработка. В Python все ошибки происходят во время выполнения, поэтому даже если ошибка не является синтаксической, она вызвана определенной операцией в определенной строке кода. В тексте необработанного исключения выводятся имя файла и номер строки, чтобы Вы знали, где искать ошибку. Последняя строка сообщения показывает, что произошло (рис.25).

```
Traceback (most recent call last):

File "…\src\debug\tserver\_sandbox.py", line 5, in <module>

File "…\src\debug\tserver\_sandbox.py", line 4, in <fragment>

builtins.IOError: [Errno 2] No such file or directory: 'A.txt'
```

Рис.25. Вид сообщений об ошибке

5.1. Обработка исключений. Блок try-exept

Инструкция try работает следующим образом:

– Сначала выполняется ветвь try (инструкции, находящиеся между ключевыми словами try и except).
– Если не возникает исключительной ситуации, ветвь except пропускается и выполнение инструкции try завершается.
– Если во время выполнения ветви try генерируется исключение, оставшаяся часть ветви пропускается. Далее, если тип (класс) исключения соответствует указанному после ключевого слова except, выполняется ветвь except и выполнение инструкции try завершается.
– Если исключение не соответствует указанному после ключевого слова except, то оно передается внешнему блоку try или, если обработчик не найден, исключение считается не перехваченным, выполнение прерывается и выводится сообщение об ошибке.

Например (рис.26):

```
#значение переменной должно быть целым числом
try:
    a= int (input())
    b= a**2
    print (‘квадрат введенного числа ’, b)
except ValueError:
    print (‘введено нецелое число, попробуйте
еще раз’)
```

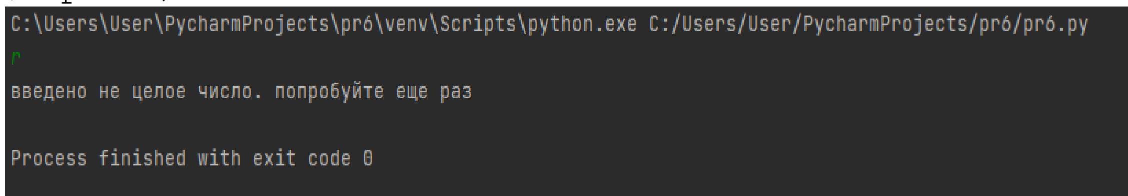


Рис.26. Результат работы программы с блоком try

Без блока try выполнение программы завершится некорректно (рис.27):

```
#значение переменной должно быть целым числом
try:
    a= int (input())
except ValueError:
    print (‘введено нецелое число, попробуйте еще
раз’)
```

```
#основная программа
b= a**2
print ('квадрат введенного числа ', b)
```

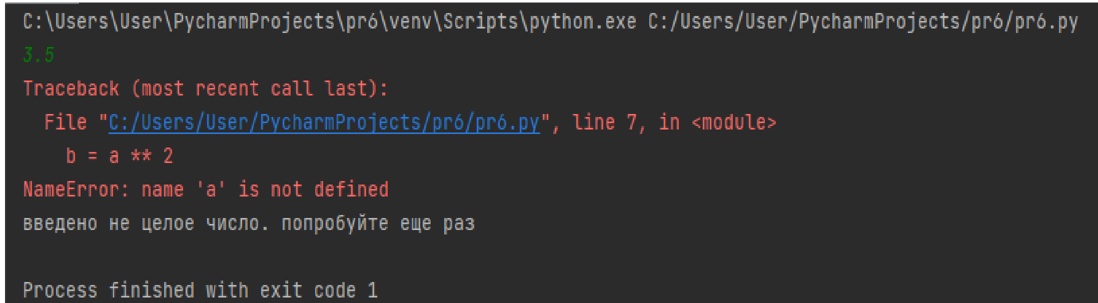


Рис.27. Результат работы программы без блока try

5.2. Типы исключений

В Python существует 20 типов встроенных исключений.

Иерархия классов для встроенных исключений в Python выглядит так:

BaseException
- SystemExit
- KeyboardInterrupt
- GeneratorExit
- Exception
  - ArithmeticError
  - AssertionError
  - ...
  - ...
  - ...
  - ValueError
  - Warning

Все исключения в Python наследуются от базового BaseException:

– SystemExit — системное исключение, вызываемое функцией sys.exit() во время выхода из приложения;

– KeyboardInterrupt — возникает при завершении программы пользователем (чаще всего при нажатии клавиш Ctrl+C);

– GeneratorExit — вызывается методом close объекта generator;

– Exception — исключения, которые можно и нужно обрабатывать (предыдущие были системными и их трогать не рекомендуется).

От Exception наследуются:

1) StopIteration — вызывается функцией next в том случае если в итераторе закончились элементы;

2) ArithmeticError — ошибки, возникающие при вычислении, бывают следующие типы:
   – FloatingPointError – ошибки при выполнении вычислений с плавающей точкой (встречаются редко);
   – OverflowError — результат вычислений большой для текущего представления (не появляется при операциях с целыми числами, но может появиться в некоторых других случаях);
   – ZeroDivisionError — возникает при попытке деления на ноль.
3) AssertionError — выражение, используемое в функции assert неверно;
4) AttributeError — у объекта отсутствует нужный атрибут;
5) BufferError — операция, для выполнения которой требуется буфер, не выполнена;
6) EOFError — ошибка чтения из файла;
7) ImportError — ошибка импортирования модуля;
8) LookupError — неверный индекс, делится на два типа:
   – IndexError — индекс выходит за пределы диапазона элементов;
   – KeyError — индекс отсутствует (для словарей, множеств и подобных объектов);
9) MemoryError — память переполнена;
10) NameError — отсутствует переменная с данным именем;
11) ReferenceError — попытка доступа к объекту с помощью слабой ссылки, когда объект не существует;
12) RuntimeError — генерируется в случае, когда исключение не может быть классифицировано или не подпадает под любую другую категорию;
13) NotImplementedError — абстрактные методы класса нуждаются в переопределении;
14) SyntaxError — ошибка синтаксиса;
15) SystemError — сигнализирует о внутренней ошибке;
16) TypeError — операция не может быть выполнена с переменной этого типа;
17) ValueError — возникает, когда в функцию передается объект правильного типа, но имеющий некорректное значение;
18) UnicodeError — исключение, связанное с кодирование текста в unicode, бывает трех видов:
   – UnicodeEncodeError — ошибка кодирования;
   – UnicodeDecodeError — ошибка декодирования;

– UnicodeTranslateError — ошибка перевода unicode.

19) Warning — предупреждение, некритическая ошибка.

Из иерархии видно, что большая часть исключений наследуется от Exception.

На что влияет эта иерархия и как ее можно использовать при обработке исключений?

Правило следующее: если мы в качестве обрабатываемого исключения указать базовый класс, будут обрабатываться все исключения-наследники этого класса.

Например, если мы в качестве базового класса укажем ArithmeticError, будут обрабатываться FloatingPointError, OverflowError и ZeroDivisionError. В примере ниже при вводе чисел 1 и 0 мы увидим сообщение «Деление на ноль», хотя явно исключение ZeroDivisionError не прописывали:

```
try:
   x, y = map (int, input ().split())
   res = x / y
except ArithmeticError:
   print ("Деление на ноль")
```

В качестве еще одного примера рассмотрим повторяющийся до введения корректных данных код: пользователю будет выдаваться приглашение до тех пор, пока не будет введено целое число (рис.28):

```
#значение переменной должно быть целым числом
while 1:

    try:
        a= int (input())
        b= a**2
        print ('квадрат введенного числа ', b)
        break
    except ValueError:
        print ('введено нецелое число, попробуйте
еще раз')
```

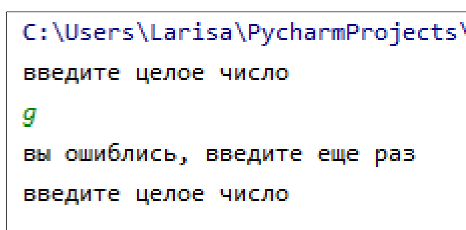


Рис. 28. Демонстрация работы корректного кода

5.3. Обработка набора исключений

Можно для каждого типа исключения использовать свой блок except:

```
try:
    val = int (input("input number:"))
    tmp= a/val
    print (tmp)
except (ValueError):
    print ('Ошибка типа данных')
except (ZeroDivisionError):
    print ('Деление на ноль')
```

Чтобы «отловить» несколько исключений в одном блоке *except* их необходимо перечислить в круглых скобках через запятую после оператора *except*:

```
try:
    val = int (input("input number:"))
    tmp= a/val
    print (tmp)
except (ValueError, ZeroDivisionError):
    print ('Error!')
```

Существует возможность передать подробную информацию о произошедшем исключении в код внутри блока *except*:

```
print ("start")

try:
    val = int (input("input number:"))
    tmp= a/val
    print (tmp)
except ValueError as ve:
    print ("ValueError! {0}".format(ve))
except ZeroDivisionError as zde:
    print ("ZeroError! {0}".format(zde)))
except Exception as ex:
    print ("Error! {0}".format(ex)))

print ("stop")
```

На рис. 29 представлен результат работы программы при вводе числа 0:

```
C:\Users\Larisa\PycharmProjects\6_1\venv\Scripts\python.exe
start
input number: 0
ZeroDivisionError! division by zero
stop
```

Рис.29. Демонстрация вывода подробной информации о произошедшем исключении

Если мы в блоке exept не укажем исключения, в момент исполнения программы будут «отлавливаться» все возможные исключения.

Пример такого кода приведен ниже:

```
try:
    x, y = map (int, input ().split())
    res = x / y
except:
    print (”Ошибка типа данных или деление на
ноль”)
```

Python допускает вложенность блоков try-except:

```
x = 10
y = 0

try:
    print("внешний блок try")
    try:
        print("вложенный блок try")
        print(x / y)
    except TypeError as te:
        print("вложенный блок except")
        print(te)
except ZeroDivisionError as ze:
    print("внешний блок except ")
    print(ze)
```

Иерархия следующая: если возникшеее исключение относится к вложенному блоку try-except, он его и обрабатывает. Если вложенный блок не может этого сделать, для обработки используется внешний (родительский) блок try-except.

5.4. Страхование от ошибок

Еще один вариант записи инструкции try – с определением “страховочной” ветви finally.

Finally — выполняется всегда. При обработке исключений можно после блока try использовать блок finally. Он похож на блок

except, но команды, написанные внутри него, выполняются обязательно.

Если в блоке try не возникнет исключения, то блок finally выполнится так же, как и при наличии ошибки, и программа возобновит свою работу.

Обычно try/except используется для перехвата исключений и восстановления нормальной работы приложения, а try/finally для того, чтобы гарантировать выполнение определенных действий (например, для закрытия внешних ресурсов, таких как ранее открытые файлы). В примере ниже блок finally сработал при срабатывании блока try (рис. 30):

```
file = open('ok.txt', 'r')

try:
    lines = file.readlines()
    print(lines[5])
finally:
    file.close()
    if file.closed:
        print("файл закрыт!")
```

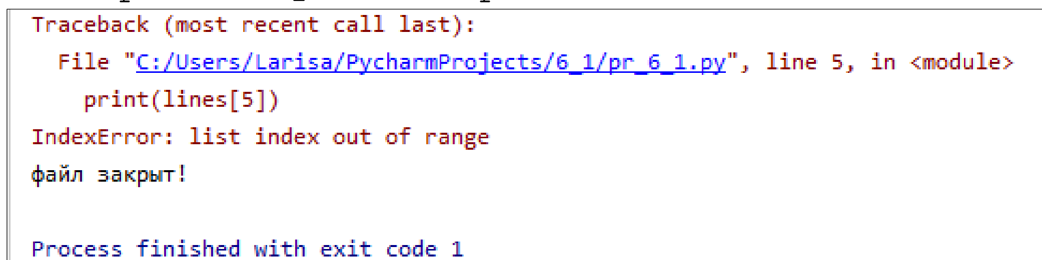


Рис. 30. Демонстрация работы finally

Можно использовать одновременно три блока try/except/finally. В этом случае помещаем:

- в try — код, который может вызвать исключения;
- в except — код, который должен выполниться при возникновении исключения;
- в finally — код, который должен выполниться в любом случае.

Например:

```
res = 0

try:
    a=int (input ("введите целое число"))
    b=int (input ("введите целое число"))
    res=a+b
```

```
    print (res)
except TypeError:
    print ("введены не числа")
finally:
    print ("dd")
```

Демонстрация работы программного кода представлена на рис.31аб.

```
2
3
5
dd

Process finished with exit code 0
```

Рис. 31а. Демонстрация одновременной работы трех блоков try/except/finally

Если введен символ, блок finally сработал все равно:

```
e
dd
Traceback (most recent call last):
  File "C:/Users/Larisa/PycharmProjects/6_1/pr_6_1.py", line 3, in <module>
    a = int (input())
ValueError: invalid literal for int() with base 10: 'e'
```

Рис. 31б. Демонстрация одновременной работы трех блоков try/except/finally

5.5. Принудительная генерация исключений

Инструкция raise позволяет программисту принудительно сгенерировать исключение. Например, в коде все в порядке, но мы генерируем исключение (рис.32):

```
res = 0

try:
   res+=1
   raise Exception("Some exception")
except Exception as e:
   print("Exception exception " + str(e))
```

```
C:\Users\Larisa\PycharmProjects\6_1\venv\Scripts\pythor
Exception exception Some exception

Process finished with exit code 0
```

Рис. 32. Демонстрация принудительной генерации исключений

Таким образом, можно “вручную” вызывать исключения при необходимости.

5.6. Дополнительные источники по плановой обработке ошибок

1. Встроенные исключения – режим доступа: https://translated.turbopages.org/proxy_u/en-ru.ru.002c50d3-62c3f098-2b0072dc-74722d776562/https/docs.python.org/3.8/library/exceptions.html
2. Конкретные исключения – режим доступа: https://translated.turbopages.org/proxy_u/en-ru.ru.002c50d3-62c3f098-2b0072dc-74722d776562/https/docs.python.org/3.8/library/exceptions.html#concrete-exceptions
3. Предупреждения – режим доступа: https://translated.turbopages.org/proxy_u/en-ru.ru.002c50d3-62c3f098-2b0072dc-74722d776562/https/docs.python.org/3.8/library/exceptions.html#warnings

Практическая работа № 4
ПЛАНОВАЯ ОБРАБОТКА ОШИБОК В PYTHON

Задание

Доработайте программный код индивидуального варианта задания практических работ №2 и №3. Для этого выявите и обработайте все возможные исключительные ситуации.

Оформите отчет по практической работе, содержащий:

- титульный лист;
- содержание;
- ответ на вопросы самоконтроля.
- для каждого варианта:
  - текст задания с указанием варианта;
  - блок-схему алгоритма решения задачи, включающую плановую обработку ошибок;
  - программный код на языке Python версии 3.8, соответствующий блок-схеме;
  - unit-тесты;

- содержание файла utest_... .py с тестами для каждого тестового случая;
- скриншоты результатов работы тестового файла и программы на тестовых данных и данных, генерирующих исключения.

Вопросы для самоконтроля к главе 5

1. Чем исключения отличаются от ошибок?
2. Чем структура вывода необработанного исключения отличается от обработанного?
3. Перечислите типы исключений, обрабатывать которые не рекомендуется. Почему?
4. Перечислите типы исключений, обработка которых рекомендована. Почему?
5. Расскажите, как в Python реализовано страхование от ошибок?
6. Как можно “вручную” вызывать исключения?

Тест самоконтроля к главе 5

1. Выберите виды ошибок в Python: а) трассировки; б) арифметические; в) синтаксические; г) исключения; д) предупреждения.
2. Синтаксические ошибки в Python возникают: а) при попытке записи в недоступное для записи место; б) нарушении требований языка программирования к синтаксису; в) в ходе выполнения программы и допускают возможность дальнейшей ее работы; г) при неправильно спроектированной работе приложения; д) в ходе выполнения программы и не допускают возможность дальнейшей ее работы.
3. Исключения в Python возникают: а) при попытке записи в недоступное для записи место; б) нарушении требований языка программирования к синтаксису; в) в ходе выполнения программы и допускают возможность дальнейшей ее работы; г) при неправильно спроектированной работе приложения; д) в ходе выполнения программы и не допускают возможность дальнейшей ее работы.
4. Что будет выведено на экран данным фрагментом кода:
   try: print (1)
   except Exception: print (0)
   а) 1; б) 0; в) Error; г) ничего не выведет.
5. Что будет выведено на экран данным фрагментом кода:

try:

b= 4 + ‘2’

print (b)

except TypeError: print (‘Error’)

а) 1; б) 0; в) Error; г) ничего не выведет.

6. Что будет выведено на экран данным фрагментом кода:

a= int (“there”)

print (a)

а) ValueError; б) TypeError; в) Error; г) there.

7.Что будет выведено на экран данным фрагментом кода:

try:

b= 4 /0

print (b)

except ZeroDivisionError:

b=0

print (b)

а) ValueError; б) TypeError; в) Error; г) 0.

8. Что будет выведено на экран данным фрагментом кода:

a=”3” – “2”

print (a)

а) ValueError; б) TypeError; в) Error; г) 1.

9. Что будет выведено на экран данным фрагментом кода:

a= 10

print (b)

а) ValueError; б) TypeError; в) NameError; г) 10.

10. Что будет выведено на экран данным фрагментом кода:

try:

b= “10”

print b

except SyntaxError:

print (1)

а) ValueError; б) TypeError; в) NameError; г) SyntaxError; д) 1.

11. В тексте необработанного исключения выводятся: а) имя файла б) номер строки, содержащей ошибку; в) сообщение, содержащее тип ошибки; г) сообщение, содержащее тип ошибки и пояснения; д) номер и содержимое строки, содержащей ошибку.

12. К исключениям, обрабатывать которые не рекомендуется, относятся: а) Exception; б) KeyboardInterrupt; в) ArithmeticError; г) SystemExit.

13. К исключениям, обрабатывать которые рекомендуется, относятся: а) Exception; б) KeyboardInterrupt; в) ArithmeticError; г) SystemExit.
14. Исключение ZeroDivisionError возникает при: а) отсутствии нужного атрибута у объекта; б) попытке деления на ноль; в) ошибке чтения из файла; г) отсутствует переменная с данным именем.
15. Исключение EOFError возникает при: а) отсутствии нужного атрибута у объекта; б) попытке деления на ноль; в) ошибке чтения из файла; г) отсутствует переменная с данным именем.
16. Исключение NameError возникает при: а) отсутствии нужного атрибута у объекта; б) попытке деления на ноль; в) ошибке чтения из файла; г) отсутствует переменная с данным именем.

Глава 6. ФОРМАТИРОВАННЫЙ ВЫВОД В PYTHON

В Python, начиная с версии 2.6, осуществить вывод данных требуемым образом можно двумя способами: с помощью операции % и метода format (). Каждому из способов в книге посвящен отдельный раздел.

6.1. Оператор форматирования %

Форматирование имеет следующий синтаксис:

<Строка специального формата> % <Значения>

Внутри параметра *<Строка специального формата>* могут быть указаны спецификаторы, имеющие следующий синтаксис:
%[(<Ключ>)][<Флаг>][<Ширина>][.<Точность>]<Тип преобразования>

Количество спецификаторов внутри строки должно быть равно количеству элементов в параметре <Значения>. Если используется только один спецификатор, то параметр <Значения> может содержать одно значение, в противном случае необходимо перечислить значения через запятую внутри круглых скобок, создавая тем самым кортеж. Например:

```
# Один элемент
print ("%s" % 10)

#будет напечатано
'10'

# Несколько элементов
 print ("%s - %s - %s" % (10,  20, 30))

#будет напечатано
'10 - 20 - 30'
```

Параметры внутри спецификатора имеют следующий смысл:

– *<Ключ>* - ключ словаря. Если задан ключ, то в параметре <Значения> необходимо указать словарь, а не кортеж.

Пример:

```
print  ("%(year)s  -  %(month)s  -  %(day)s"  %
{"year":"1967",  "month": "12", "day": "30"})

#будет напечатано
1967-12-30
```

– <Флаг> - флаг преобразования. Может содержать следующие значения:
  - для восьмеричных значений добавляет в начало комбинацию символов O0, для шестнадцатеричных значений добавляет комбинацию символов Ox, для вещественных чисел предписывает всегда выводить дробную точку, даже если задано значение 0 в параметре <Точность>. Например:

```
print("%#o %#x %#o" % (77, 10, 12))

#будет напечатано
0o115  0o14

print("%#.0F %.0F" % (300,  300))

#будет напечатано
300. 300
```

  - 0 (цифра) задает наличие ведущих нулей для числового значения. Например:

```
print (" %d - %05d " % (3,  3))# 5 - ширина
поля

#будет напечатано
 3 - 30003
```

  - – задает выравнивание по левой границе области. По умолчанию используется выравнивание по правой границе. Если флаг указан одновременно с флагом 0, то действие флага о будет отменено. Например:

```
print (" %d - %05d " % (3,  3))
print ("%5d - %-5d"  % (3,  3))

#будет напечатано
 3 – 30003
    3 - 3
```

- пробел вставляет пробел перед положительным числом. Перед отрицательным числом будет стоять минус. Например:

```
print ("%  d % d " % (-3,  3))

#будет напечатано
 -3 3
```

- + задает обязательный вывод знака, как для отрицательных, так и для положительных чисел. Если флаг +указан одновременно с флагом пробел, то действие флага пробел будет отменено. Например:

```
print ("% +d % +d " % (-3,  3))

#будет напечатано
 -3 +3
```

– <Ширина> - минимальная ширина поля. Если строка не помещается в указанную ширину, то значение игнорируется и строка выводится полностью:

```
print ("'%10d' - '%-10d'"  % (3234567,  3))

#будет напечатано
'   3234567'  - '3         '

print   ("'   %30s'   '%20s'"   %(   "string", "string"))

#будет напечатано
'                  string' '          string'
```

– <Точность> – количество знаков после точки для вещественных чисел. Перед этим параметром обязательно должна стоять точка. Пример:

```
import math

#выводится строка, числа с точностью до 6 и 2 знаков
print ( "%s %f %.2f" % (math.pi, math.pi,math.pi))

#будет напечатано
3.141592653589793   3.141593   3.14
```

– <Тип преобразования> задает тип преобразования. Параметр является обязательным. В параметре могут быть указаны следующие символы:

- s – преобразует любой объект в строку с помощью функции str(). Например:

```
print ("%s%s%s" % (10,  10.5,[1,  2, 3]))

#будет напечатано
 1010.5 [1, 2, 3]
```

- r – преобразует объект в строку с помощью функции repr():

```
#список остается числом.
print ("%r%r" % ([1,2,3],'vjht'))

#будет напечатано
[1, 2, 3] 'vjht'
1
```

- + a – преобразует объект в строку с помощью функции ascii():

```
print("%a"% ("строка"))

#будет намечатано
'\u0441\u0442\u0440\u043e\u043a\u0430'
```

- c – выводит одиночный символ или преобразует числовое значение в символ. В качестве примера выведем числовое значение и соответствующий этому значению символ:

```
for i in range(33, 127):
    print("%0d %c" % (i,  i))

#будет напечатано
33!
34"
35#
36$
37%
38&
39'
```

- d и i – возвращают целую часть числа. Например:

```
print("%d  %d   %d" % (10,  25.6, -80.134))
```

```
#будет напечатано
10   25   -80
```

- f и F - вещественное число в десятичном представлении. Например:

```
print("%f  %f %f" % (300,  18.65781452,-
  2.5))

#будет напечатано
300.000000 18.657815 -12.500000
300. 300
```

- e - вещественное число в экспоненциальной форме (буква "e" в нижнем регистре):

```
print ( "%e %e" % (3000,  18657.81452))

#будет напечатано
3.000000e+03  1.865781+04
```

- g - эквивалентно f или e (выбирается более короткая запись числа):

```
print("%g  %g  %g"%  (0.086578,    0.000086578,
1.865E-005))

#будет напечатано
0.086578  8.6578e-05  1.865e-05
```

Если внутри строки необходимо использовать символ процента, то этот символ следует удвоить, иначе будет выведено сообщение об ошибке:

```
print("%%  %s" % ("- это символ процента"))

#будет напечатано
% - это символ процента
```

Форматирование строк очень удобно использовать при передаче данных в шаблон HTML-страницы. Для этого заполняем словарь данными и указываем его справа от символа %,

```
<head><title>%(title)s</title>
     </head>
     <body>
    <h1>%(h1)s</h1>
    <div>%(content)s</div>
    </body>
```

```
</html>"""
arr = {"title": "Это название документа",
 "h1": "Это заголовок первого уровня",
 "content": "Это основное содержание страни-
цы"}

# Подставляем значения и выводим шаблон
print(html % arr) input()
```

Результат выполнения представлен на рис. 33:

```
<html>
<head><title>Это название документа</title>
</head>
<body>
<h1>Это заголовок первого уровня</h1>
<div>Это основное содержание страницы</div>
 </body>
 </html>
```

Рис. 33. Форматирование строк в шаблоне HTML-страницы

6.2. Метод format ()

Метод имеет следующий синтаксис:
*<Строка>= <Строка специального формата>.format(*args, **kwargs)*

В параметре *<Строка специального формата>* внутри символов { и } указываются спецификаторы, имеющие синтаксис: *{[<Поле>][!<Функция>][:<Формат>]}*

Все символы, расположенные вне фигурных скобок, выводятся без преобразований. Если внутри строки необходимо использовать символы { и }, то эти символы следует удвоить, иначе возбуждается исключение ValueError.

В параметре <Поле> можно указать индекс позиции, в которой необходимо вывести данные, или ключ. Нумерация начинается с нуля. Например:

```
print ("Символы {{ и
}}{0}".format("специальные"))

#будет напечатано
```

```
Символы { и } – специальные
```

В качестве еще одного примера выведем два числа и строку в позициях 0, 2 и 1 соответственно с «–» в качестве разделителя и спецификатором {1: 3.3f}, задающим для числа 12.3 по три знакоместа для целой и дробной части:

```
print ("{0} - {2} - {1: 3.3f}".format(10, 12.3,
"string"))

#будет напечатано
10 – string – 12.300
```

Существует краткая форма записи оператора, при которой параметр <*Поле*> не указывается. В этом случае скобки нумеруются слева направо, начиная с нуля. Например:

```
print ("{}-{}-{}-{n}".format(1,2,3,n=4))
print ("{}-{n}-{}-{}".format(1,2,3,n=4))

#будет напечатано
1 – 2 – 3 – 4
1 – 4 – 2 – 3
```

В качестве примера аргумента-ключа выведем фразу «Сергей, ваш баланс составляет 134 рубля». В качестве данных здесь использованы сочетания ключ-значение: name="Сергей" и blc=134:

```
print  ("{name},ваш  баланс  составляет{blc:5.2f}
рубля". format(name="Сергей",blc=134))

#будет напечатано
Сергей, ваш баланс составляет 134.00 рубля
```

Допустимо комбинировать позиционные и именованные параметры. В этом случае в методе format () именованные параметры указываются в самом конце. Например:

```
print("{0}- {1}- {2}".format(10, 12.3, "string"))
arr = [3, 5.6, "more"]
print("{0}- {1}- {2}".format(*arr))

#будет напечатано
10-  12.3-  string
3-  5.6-  more
```

Параметр <*функция*> задает функцию, с помощью которой обрабатываются данные перед вставкой в строку. Если указано значение «s», то данные обрабатываются функцией str (), если значение «r», то данные обрабатываются функцией repr(), а если значение «a», то функцией ASCII().

В качестве примера выведем на экран слово строка с помощью функций str() и ASCII(). В первом случае на экране мы увидим слово «строка», во втором – шестнадцатеричные коды символов, образующих это слово:

```
print ("{!s}".format("строка"))
print ("{!a}".format("строка"))

#будет напечатано
строка
'\u0441\u0442\u0440\u043e\u043a\u0430'
```

Символ «!» в коде означает преобразование формата.

6.3. Дополнительные источники по вопросу форматированного вывода

1. Васильев А. Программирование на Python в примерах и задачах. –Москва: Эксмо, 2021. – 616 с.
2. Использование метода format () – режим доступа: https://translated.turbopages.org/proxy_u/en-ru.ru.d363332f-62c3f7e0-ac7343f0-74722d776562/https/docs.python.org/3.8/tutorial/inputoutput.html
3. Форматирование строк – режим доступа: https://translated.turbopages.org/proxy_u/en-ru.ru.d363332f-62c3f7e0-ac7343f0-74722d776562/https/docs.python.org/3.8/tutorial/inputoutput.html#formatted-string-literals
4. Ручное форматирование – режим доступа: https://translated.turbopages.org/proxy_u/en-ru.ru.d363332f-62c3f7e0-ac7343f0-74722d776562/https/docs.python.org/3.8/tutorial/inputoutput.html#manual-string-formatting
5. Оператор форматирования % - режим доступа: https://translated.turbopages.org/proxy_u/en-ru.ru.d363332f-62c3f7e0-ac7343f0-

74722d776562/https/docs.python.org/3.8/tutorial/inputoutput.html#old-string-formatting

Практическая работа № 5
ФОРМАТИРОВАННЫЙ ВВОД-ВЫВОД В PYTHON

Задание

Выведите на экран результаты работы программы (или символы) согласно требованиям форматирования, указанным в индивидуальном варианте.

Оформите отчет по практической работе, содержащий:

- титульный лист;
- содержание;
- ответ на вопросы самоконтроля.
- для каждого варианта:
  - текст задания с указанием варианта;
  - программный код на языке Python версии 3.8, соответствующий блок-схеме;
  - скриншоты результата работы программы.

ВАРИАНТЫ ЗАДАНИЙ

№	Шаблон	№	Шаблон
1	1 2 2 3 3 3 4 4 4 4 5 5 5 5 5	2	Hello | Hello Hello Hello......... ....Hello.....
3	Напишите программу, которая принимает 3 входных данных, и подставляет эти данные в предложение.		3.141592653589 +3.14 -3.14 3 003 3,141,592, 653
5	5 5 5 5 5 4 4 4 4 3 3 3 2 2 1	6	12 1234 12.235 00012 0012.235
7	+12.230000 -12.230000 12.230000 -12.230000 12.230000 -12.230000	8	12 12.235 12000 - 12.235

9	cat. cat . cat . ‘cat’	10	It's now: 2020/01/18 12:19:24 Real part: 1.0 and Imaginary part: 2.0 Adam's age is: 23
11	Number before inserting commas : 1000000000 Number after inserting commas: 1,000,000,000 Number after inserting commas: ‘1.000.000.000’	12	1 1 2 1 2 3 1 2 3 4 1 2 3 4 5
13	5 4 3 2 1 1 2 3 4 5 5 4 3 2 2 3 4 5 5 4 3 3 4 5 5 4 4 5 5 5	14	Bob - 35- Bob- 35- Bob - 35 -
15	1.3333333333333333 1.333333 1.33 1.33 1.333333e+00	16	*+1,234 0123.4570 123 123.456790 1100
17	‘1; A; 2; B’ 1 A 2 B ‘sym: Z; num: 45’ sym: Z; num: 45	18	5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
19	1 1 1 1 1 2 2 2 2 3 3 3 4 4 5	20	‘| - 000000000000 1.234|’ | - 000000000000 1.234| - 000000000000 1.234 1. +1.j
21	Правильные ответы: 86,361 Правильные ответы: 86,36% Правильные ответы: 86,4% Правильные ответы: 0,86	22	1 2 3 4 5 6 7 8 9 10 11 12 13 14

Вопросы для самоконтроля к главе 6

1. Какие способы форматированного вывода в Python вы знаете? Чем они отличаются? В чем схожи? Приведите примеры.
2. Какой синтаксис имеет оператор %?
3. Поясните смысл спецификатора <Ключ> оператора %.

4. Поясните смысл спецификатора <Флаг> оператора %.
5. Поясните смысл спецификатора <Ширина> оператора %.
6. Поясните смысл спецификатора <Точность> оператора %.
7. Поясните смысл спецификатора <Тип преобразования> оператора %.
8. Какой синтаксис имеет оператор format?
9. Поясните смысл параметра <Строка специального формата> метода format ()?

Тест самоконтроля к главе 6

1. Что выведет данный код?

   a="P".format ("Y")
   print (a)

   а) РУ; б) Р; в) У; г) УР.

2. Что выведет данный код?

   a="{2}{0}".format ("p","y","b")
   print (a)

   а) bp; б) p; в) y; г) pb.

3. Что выведет код?

   a="{2}{0}".format ("ypc")*
   print (a)

   а) ус; б) ур с; в) су; г) ур.

4. Что выведет код?

   a="P{}".format ("y")
   print (a)

   а) у; б) ур; в) уР; г) Ру.

5. Что выведет код?

   a="{0}".format ((0,1))
   print (a)

   а) 0; б) 10; в) 01; г) (0,1).

6. Что выведет код?

   n=(4,5)
   b="{0[0]}; {0[1]}".format (n)
   print (b)

   а) 4; 5; б) 54; в) 45; г) 4, 5.

7. Что выведет код?

   b=("{0}"+"{1}").format ("2","4")
   print (b)

а) 4; 2; б) 42; в) 24; г) 2, 4.

8. Выберите все способы передачи в строку "I have N Apples " значения переменной var = 10 вместо N.

   а) *"I have %i apples" % var;* б) *fr"I have {var} apples";* в) *"I have (&s = var) apples".format(var);* г) *"I have {} apples".format(var).*

9. Как в Python3.8 и выше вывести название переменной var и ее значение одной строкой?

   а) *f"{var}";* б) *"{var:=}";* в) *"{!var}";* г) *f"{var=}".*

10. Как вывести значение переменной number=10 с точностью до 3 знаков после запятой? а) print (f"{number, 3f}"); б) print (f"{number:.3f}"); в) print (f"{number:&3f}"); г) print (f"{number.% 3f}").

11. Как в Python3.8 и выше разделить большое число по сотням символом «_»? Например, число number=200000000 привести к виду 200_000_000. а) print (f"{number:_}"); б) print (f"{number, _}"); в) print(f"{number.%_}"); г)print (f"{number:_s}").

12. Как в Python3.8 и выше разделить большое число по сотням символом , (запятая)? Например, число number=200000000 привести к виду 200,000,000. а) print (f"{number:,}"); б) print (f"{number:,s}"); в) print (f"{number, _}"); г) print (f"{number,:}").

13. Как в Python3.8 и выше представить десятичное число в формате «процентный»? Например, число number=0.34 привести к виду 34%. а) print(f"{number-%}"); б) print(f"{number.%}"); в) print(f"{number, %}"); г) print(f"{number:%}").

14. Имеется объект типа datetime (дата и время) – now=date (2022,11,5). Как в Python3.8 и выше вывести год, месяц, день в формате год-месяц-день? а) print(f"{now=(%Y-%m-%d)}"); б) print (f"{now=[%Y-%m-%d]}"); в) print (f"{now:%Y-%m-%d}"); г) print (f"{now=%Y-%m-%d}").

15. Каким из указанных ниже способов в Python3.8 и выше можно добавить перед передаваемой переменной s= "hello" 3 (три) отступа вида «*». В результате должно быть выведено "***hello": а) print (f'{s:*>3}'); б) print (f'{s:*<3}'); в) print (f'{s:*>8}'); г) print (f'{s:*<8}').

Глава 7. ОСНОВНЫЕ ТИПЫ ДАННЫХ ЯЗЫКА PYTHON

7.1. Типы данных в технологиях разработки программного обеспечения

Современные технологии разработки программного обеспечения располагают широким набором формальных методов, которые помогают убедиться, что система ведет себя в соответствии с некоторой спецификацией ее поведения, явной или неявной.

На одном конце спектра находятся мощные конструкции, такие как логика Хоара, языки алгебраической спецификации, модальные логики и различные семантики. Они способны выразить самые широкие требования к корректности программ, однако часто очень трудны в использовании и требуют от программистов высочайшей квалификации.

На другом конце спектра располагаются намного более скромные методы такие, что применять их могут даже программисты, не знакомые с теоретическими основами этих методов.

Примерами таких методов являются программы проверки моделей, мониторинг во время исполнения и т.д.. Самые популярные – это системы типов (type systems).

Ниже приведем одну из различных трактовок термина «система типов» - это гибко управляемый синтаксический метод доказательства отсутствия в программе определенных видов поведения при помощи классификации выражений языка по разновидностям вычисляемых ими значений.

В более общем смысле, термин «системы типов» относится к намного более обширной области исследований в таких областях, как логика, математика и философия.

В этом смысле типы были впервые формально описаны в начале 1900-х годов как средство избегания логических парадоксов, угрожавших основаниям математики. Например, парадокса Рассела.

Поясним его. На неформальном языке парадокс можно описать следующим образом. Условимся называть множество «обычным», если оно не является своим собственным элементом. Например, множество всех людей является «обычным», так как само множество – не человек. Примером «необычного» множества является множество всех множеств, так как оно само является множеством, а, следовательно, само является собственным элементом.

Можно рассмотреть множество, состоящее только из всех «обычных» множеств. Парадокс возникает при попытке определить, является ли это множество «обычным» или нет, то есть содержит ли оно себя в качестве элемента.

Есть две возможности.

С одной стороны, если оно «обычное», то оно должно включать себя в качестве элемента, так как оно по определению состоит из всех «обычных» множеств. Но тогда оно не может быть «обычным», так как «обычные» множества – это те, которые себя не включают.

Остаётся предположить, что это множество «необычное». Однако оно не может включать себя в качестве элемента, так как оно по определению должно состоять только из «обычных» множеств. Но если оно не включает себя в качестве элемента, то это «обычное» множество.

В любом случае получается противоречие.

В течение XX века типы превратились в стандартный инструмент логики, особенно в теории доказательств, и глубоко проникли в язык философии и науки.

В этой области основными достижениями были теория типов Рассела (1910 г.), простая теория типов Рамсея (1925 г.), основа для простого типизированного лямбда-исчисления Чёрча (1940 г.), конструктивная теория типов, Мартина-Лёфа (1973 г., 1984 г.) и чистые системы типов Берарди, Терлоу и Барендрегта (1988, 1989, 1992 года соответственно).

Внутри самой информатики как научной дисциплины изучение систем типов разделяется на две ветви.

Одна ветвь, в которой исследуются приложения к языкам программирования.

Вторая ветвь изучает соответствия между различными «чистыми типизированными лямбда-исчислениями» и разновидностями логики.

В обоих направлениях используются аналогичные понятия, системы записи и методы, однако есть важные различия в подходе.

Например, при исследовании типизированного лямбда-исчисления обычно имеют дело с системами, в которых для всякого правильно типизированного вычисления гарантировано завершение, тогда как большинство языков программирования жертвуют этим свойством ради таких инструментов, как рекурсивные функции.

Систему типов можно рассматривать как статическую аппроксимацию поведения программы во время выполнения.

Иногда слово «статический» добавляется явным образом – например, когда говорят о «языках программирования со статической типизацией», чтобы отличить тот анализ, который производится при компиляции от динамической или латентной типизации в языках типа Scheme.

При динамической типизации теги типов используются для различения видов объектов, находящихся в куче.

Будучи статическими, системы типов обязательно консервативны. Они способны однозначно доказать отсутствие определенных нежелательных видов поведения, но не могут доказать их наличие и, следовательно, иногда вынуждены отвергать программы, которые на самом деле правильные и при выполнении ведут себя корректным образом. Так, например, программа if … then S else B будет отвергнута как неверно типизированная, даже если «сложная проверка» всегда выдает значение «истина», поскольку статический анализ неспособен это заметить.

Виды анализа, воплощенные в большинстве систем типов, неспособны совсем запретить любое нежелательное поведение программ, они гарантируют только отсутствие в программах определенных видов ошибок.

Например, большинство систем типов могут статически проверить, что в качестве аргументов элементарных арифметических операций всегда используются числа, что объект-получатель в вызове метода всегда имеет соответствующий метод, и т. п., но не могут обеспечить, чтобы делитель всегда не был равен нулю или чтобы индексы массива никогда не выходили за предельные значения.

Виды некорректного поведения, которые в каждом конкретном языке программирования могут быть устранены при помощи типов, часто называют ошибками типов времени выполнения (run-time type errors).

Набор таких ошибок в каждом языке свой. Несмотря на то, что наборы ошибок времени выполнения в разных языках существенно пересекаются, в принципе каждая система типов сопровождается описанием ошибок, которые она должна предотвращать.

Процедуры проверки типов обычно встроены в компиляторы или компоновщики. В качестве примера приведем несколько кодов сообщений компилятора системы Турбо-Паскаль об ошибках: 1 – вы-

ход за границы памяти; 2 – не указан идентификатор; 3 - неизвестный идентификатор.

Самое очевидное достоинство статической проверки типов – это то, что она помогает раньше обнаруживать некоторые ошибки в программах. Рано обнаруженные ошибки могут быть немедленно исправлены, а не прятаться долго в коде, чтобы потом неожиданно всплыть, когда программист занят чем-то совершенно другим — или даже после того, как программа передана пользователям. Более того, зачастую место возникновения ошибки можно точнее определить при проверке типов, а не во время выполнения, когда их последствия могут обнаружиться не сразу, а лишь спустя некоторое время после того, как программа начинает работать неправильно.

Для некоторых видов программ процедура проверки типов может служить инструментом сопровождения. Например, программист, которому требуется изменить определение сложной структуры, может не искать вручную все места в программе, где требуется изменить код, имеющий дело с этой структурой. Как только изменяется определение типа данных, во всех этих фрагментах кода возникают ошибки типов, и их можно найти путем простого прогона компилятора и исправления тех мест, где проверка типов не проходит.

Самые ранние системы типов в языках программирования использовались для простейшего различения между целыми числами и числами с плавающей точкой (например, в Фортране).

В конце 1950-х и начале 1960-х эта классификация была расширена на структурированные данные (массивы записей и т. п.) и функции высшего порядка.

В начале 1970-х появились еще более сложные понятия (параметрический полиморфизм, абстрактные типы данных, системы модулей и подтипы), и системы типов превратились в отдельное направление исследований. Тогда же специалисты по информатике обнаружили связь между системами типов в языках программирования и типами, изучаемыми в математической логике, и это привело к плодотворному взаимодействию между двумя областями, которое продолжается по сегодняшний день.

7.2. Типы данных в Python 3

Все данные в языке Python представлены объектами.

Различают: встроенные, предоставляемые языком Python, и создаваемые пользователем с помощью других инструментов, таких как библиотеки расширений, написанные на языке С

Каждый объект характеризуется значением и набором операций. Для доступа к объекту предназначены переменные. При инициализации в переменной сохраняется ссылка на объект (адрес объекта в памяти компьютера). Благодаря этой ссылке можно в дальнейшем изменять объект из программы.

Начнем с обзора встроенных объектных типов языка Python.

Если вы знакомы с языками программирования С или С++, то вы уже знаете, что значительная доля работы приходится на реализацию объектов, которые предназначены для представления компонентов внутри приложений. В таких языках программирования необходимо заниматься проработкой структур данных, управлять выделением памяти, реализовывать функции поиска и доступа к элементам структур и т. д. Это часто способствует появлению ошибок. В типичных программах на языке Python в этом нет необходимости. Python предоставляет мощную коллекцию объектных типов, встроенных непосредственно в язык.

Поэтому, если у вас нет потребности в специальных видах обработки, обычно нет никакой необходимости создавать собственные реализации объектов, предназначенных для решения поставленных задач.

В Python версий 3.* существуют 7 встроенных типов объектов:

- числа;
- строки;
- булевы величины;
- списки;
- словари;
- множества;
- кортежи.

7.2.1. Числовой тип

В Python для хранения числа отводится 64 бита. Различают: целые числа со знаком (int), с плавающей точкой с точностью до 15 десятичных знаков (float) и комплексные (a±bj, где j – мнимая единица).

Чтобы определить число, в Python используется запись, состоящая из цифр и символа «.». Например, 123, 12.3.

Пример программы, вычисляющей сумму двух мнимых чисел:

```
a= 12.5
b=1+5j
c=2-1j
a=a**2
sum=b+c
print (a,b,c)
print ("сумма",sum)
#будет выведено
156.25  (1+5j)  (2-1j)
сумма (3+4j)
```

Целые числа

По умолчанию для записи чисел используется десятичная система счисления, но для целых чисел также можно использовать шестнадцатеричную, восьмеричную и двоичную системы:

- int(x) - преобразование к целому числу в десятичной системе счисления:
  - если на входе строка:
    ```
    a = int ('19')
    print (a)

    #будет напечатано
    19
    ```
  - если на входе число, записанное в системе счисления, отличной от десятичной:
    ```
    #преобразование двоичного числа
    a = int (0b10011)
    print (a)

    #будет напечатано
    19

    #обратное преобразование
    ```

```
bin (x)
a = bin (19)

#будет напечатано
print (a)
0b10011

#преобразование целого числа в восьмерич-
ную строку
a = oct(19)
print (a)

#будет напечатано
0o23

#преобразование целого числа в шестнадца-
теричную строку
a = hex(19)
print (a)

#будет напечатано
0x13
```

Для работы с большими значениями можно использовать экспоненциальную запись числа. Слева от e расположено основание экспоненты (любое десятичное число), а справа — степень экспоненты:

```
a = 2.99e6
print (a)

#будет напечатано
2990000.0
```

Целые числа преобразуются в строки функцией str(). Обратное преобразование можно осуществить функцией int(). Для целых чисел поддерживается набор типовых операторов (см. таблицу 3).

Таблица 3. Типовые операторы Python для работы с числами

Оператор	Описание
x + y	сложение
x - y	вычитание

x * y	умножение
x / y	деление
x // y	получение целой части от деления
x % y	остаток от деления
-x	смена знака числа
abs (x)	модуль числа
x ** y или pow (x, y)	возведение в степень
~x	побитовое не
^	побитовое исключающее или
&	побитовое исключающее и

Дробные числа

Для десятичных дробей (иначе их называют «числа с плавающей запятой») в Python есть специальный тип данных - **float**:

```
first = 87.2
second = 50.2
third = 50.242
print (first + second + third)

# Будет напечатано
187.642
```

Дробные числа преобразуют в строки так же, как и целые — функцией str ():

```
first = 87.2
second = 50.2
third = 50.242
print(str(first) + str(second) + str(third))

#будет напечатано
87.250.250.242

first = 87.2
second = 50.2
third = 50.242
```

```
print(str(first) + ' '+ str(second)+'
'+str(third))

#будет напечатано
87.2   50.2    50.242
```

Следует помнить, что вычисления не с целыми числами будут неточными.

Ввод значений в программу

Выше мы присваивали значения переменным непосредственно в программном коде (имеется в виду, что значение, присваиваемое переменной, в программном коде было представлено текстовым литералом или числом). Но существует возможность вводить значение через окно оболочки интерпретатора непосредственно в процессе выполнения программы. Для этого используют встроенную функцию input(), которая в качестве результата возвращает значение, введенное пользователем с клавиатуры.

В качестве примера введем с клавиатуры целое число, строку, значение элемента словаря по ключу:

```
#целое число
a= int (input ())
print (a)

#строка
st= (input ())
print (st)

#значение элемента словаря
garden = {}
union=['овощ','оружие']
garden['земляника']=input("земляника:")
```

Константы

В Python используются четыре вида числовых констант, использование которых требует подключения математической библиотеки *math*:

- *math.pi* – возвращает значение математической константы π с точностью, которая зависит от конкретной платформы;
- *math.e* – возвращает значение математической константы e с точностью, которая зависит от конкретной платформы;

– *math.Inf* – возвращает положительную бесконечность, значение которое является типом float и может присутствовать в математических выражениях. Например, если вычесть из бесконечности единицу, получится по-прежнему бесконечность.
– *math.Nan* – возвращает значение "не число" которое является типом *float* и может присутствовать в математических выражениях как результат недопустимых арифметических операций. Например, при делении на ноль.

Преобразование типов

Если умножить число на число, никаких неожиданностей не будет:

```
six = 6
print (six * 7)

#будет напечатано
42
```

А умножение строки на число скопирует строку несколько раз:

```
lol = 'xo'
print(lol * 10)

#будет напечатано
xoxoxoxoxoxoxoxoxoxo
```

Для совместного вывода числа и строки, надо преобразовать число в строку:

```
count = 8
st1='У вас '
st2=' новых сообщений'
print (st1+ str(count) +st2)

#будет напечатано
У вас 8 новых сообщений
```

Приведение к целому типу

```
#Функция int() просто убирает всё, что после запя-
той
a = int (3.14)
print(a)

#будет напечатано
3
```

```
a = int (2.72)
print(a)

#будет напечатано
2
```

Функция int() одинаково работает с положительными и отрицательными числами:

```
a = int (-3.14)
print(a)

#будет напечатано
-3
```

Можно сделать несколько преобразований в одной строке: сначала превратить дробь в целое число, а затем преобразовать в строку:

```
fraction = 1.5  # Дробь
print("Целая часть = " + str (int (fraction)))
```

Вернётся строка, представляющая собой целочисленную часть дроби: «Целая часть = 1».

Еще один пример:

```
speed_kmh = 1079252848.8
# переменную speed_kms сделаем типа int
speed_kms = speed_kmh/3600
speed_kms=int(speed_kms)
print('Скорость света равна', speed_kms, 'км/с')

#будет напечатано
Скорость света равна 299 792 км/с
```

Приведение к вещественному типу

```
first = '87.2'   # Строка
second = '50.2'  # Тоже строка
third = '50.242' # И это строка
#В итоге получится число
print    (float(first)    +    float(second)    +
float(third))

# будет напечатано
187.642
```

Логический тип данных (*bool*) в *Python*. Логический тип данных (bool) (или булевый тип) это примитивный тип данных, который принимает 2 значения — истина (*True*) или ложь (*False*).

True и *False* являются экземплярами класса *bool*, который в свою очередь является подклассом *int*. Поэтому *True* и *False* в Python ведут себя как числа 1 и 0. Отличие только в том, что будет выводено на экран. Например, команда print (True + 4) выведет на экран число 5, так как True - это число 1. А команда print (5 * False) выведет число 0, так как False - это число 0 и при умножении на любое число результатом будет 0.

Вопросы для самоконтроля к параграфу 7.2.1

1. Для чего использовались типы данных в ранних языках программирования?
2. Каким образом представлены типы данных в Python? Что характерно для такого способа представления данных?
3. Перечислите встроенные типы данных языка Python и кратко охарактеризуйте каждый тип.
4. Как можно привести вещественное число к целому типу?
5. Как можно привести число к вещественному типу?
6. Как преобразовать число к строке и наоборот? В каких случаях это возможно, а в каких нет?
7. Перечислите и кратко охарактеризуйте типы числовых констант Python.

Тест самоконтроля к параграфу 7.2.1

1. Типы данных в ранних языках программирования использовались для: а) проверки моделей; б) мониторинга во время исполнения программы; в) доказательства отсутствия в программе определенных видов поведения; г) различения типов числовых данных.
2. В языке Python различают следующие типы объектов: а) встроенные; б) созданные с помощью различных инструментов; в) создаваемые пользователем; г) строки и числа.
3. Объект характеризуется: а) типом; б) значением; в) набором операций; г) ссылкой на класс-предок.

4. К встроенным типам данных языка Python относятся: а) записи; б) множества; в) массивы; г) словари; д) файлы.
5. В Python существует … типа числовых констант: а) четыре; б) три; в) пять; г) семь.
6. Функция int выполняет: а) возвращает остаток от деления; б) возвращает целую часть от деления; в) отбрасывает дробную часть числа, округляя его; г) отбрасывает дробную часть числа, не округляя его;
7. В результате работы функции int (3.45)вы получите: а) 4; б) 3; в) 3.5; г) 3.4.
8. В результате работы функции round (3.45) вы получите: а) 4; б) 3; в) 3.5; г) 3.4.
9. Для проверки типа переменной в Python используется встроенная функция: а) func (); б) tape(); в) round(); г) type().
10. Какие из перечисленных ниже типов строковых данных поддерживает язык Python 3.*: а) unicode; б) bytes; в) koi-8 г) windows 1251.
11. Что выведет код:
    ```
    number=200+True
    print (number)
    ```
    а) ничего; б) ошибку; в) 200; г) 201; д) 198.
12. Что выведет код:
    ```
    number=200+~True
    print (number)
    ```
    а) ничего; б) ошибку; в) 200; г) 201; д) 198.
13. Что выведет код:
    ```
    number=200 - True
    print (number)
    ```
    а) 199; б) ошибку; в) 200; г) 201; д) 198.
14. Что выведет код:
    ```
    number=200 - ~True
    print (number)
    ```
    а) 202; б) ошибку; в) 200; г) 201; д) 198.
15. Что выведет код:
    ```
    number=~True
    print (number)
    ```
    а) ничего; б) -2; в) ошибку; г) -1; д) 1.
16. Что выведет код:

number=True
print (number)
а) true; б) -2; в) ошибку; г) -1; д) 1.

17. Что выведет код:
number=~(~200)
print (number)
а) 202; б) ошибку; в) 200; г) 201; д) 198.

18. Что выведет код:
number=1^1^0
print (number)
а) ничего; б) ошибку; в) 0; г) 1; д) 11.

19. Что выведет код:
number=1^(1^0)
print (number)
а) ничего; б) ошибку; в) 0; г) 1; д) 11.

20. Что выведет код:
number=~-False
print (number)
а) ничего; б) ошибку; в) 0; г) 1; д) -1.

21. Что выведет код:
number=-False
print (number)
а) ничего; б) ошибку; в) 0; г) 1; д) -1.

22. Что выведет код:
number=~False
print (number)
а) ничего; б) ошибку; в) 0; г) 1; д) -1.

23. Что выведет код:
number=~-False
print (number)
а) false; б) ошибку; в) 0; г) 1; д) -1.

24. Что выведет код:
number=True+~False
print (number)
а) false; б) ошибку; в) 0; г) 1; д) -1.

25. Что выведет код:
number=True+~False
print (number)
а) false; б) ошибку; в) 0; г) 1; д) -1.

26. Что выведет код:

```
number=~1.1
print (number)
```

а) false; б) ошибку; в) 0; г) 1; д) -1.

27. Что выведет код:

```
number=1+ .1
print (number)
```

а) 1.1; б) ошибку; в) 0; г) 1; д) 0.1.

28. Что выведет код:

```
number= str(.1)
print (number)
```

а) 1.1; б) ошибку; в) 0; г) 1; д) 0.1.

29. Что выведет код:

```
number= str(.1)
print (number [1])
```

а) 1.1; б) ошибку; в) . ; г) 1; д) 0.1.

30. Что выведет код:

```
number= str(.10)
print (number)
```

а) 1.1; б) ошибку; в) . ; г) 1; д) 0.1.

7.2.2. Статический массив

Наряду со списками в Python реализовано классическое представление массива – статический массив.

Представим, что нам в программе требуется сохранить значения функции cos(), например, в диапазоне от 0 до $2 \cdot \pi$ с шагом 0,1.

Как это сделать? Для этого очень хорошо подходит как раз статический массив. Каждый элемент массива хранит значения функции в отдельной точке (рис.).

Другой пример. Нам надо сохранять состояние игрового поля (крестики, нолики) размером 3*3 элемента. Удобно воспользоваться статическим массивом длиной 9 элементов. Тогда каждые 3 элемента – это строка игрового поля (рис.34).

Или такая задача – в программе необходимо сохранить фамилии студентов. Выбираем статический массив и посимвольно записываем фамилию, получаем строку (рис.34).

Все это примеры, когда целесообразно использовать статический массив.

Посмотрим преимущества и недостатки этой структуры данных.

Структура статического массива:

1) Все элементы массива должны иметь один тип данных. Например, только целые числа или только вещественные или только символьный тип. Не предполагается в одном и том же массиве сохранять разные типы данных (рис.35абв).

Рис.34. Примеры статического массива

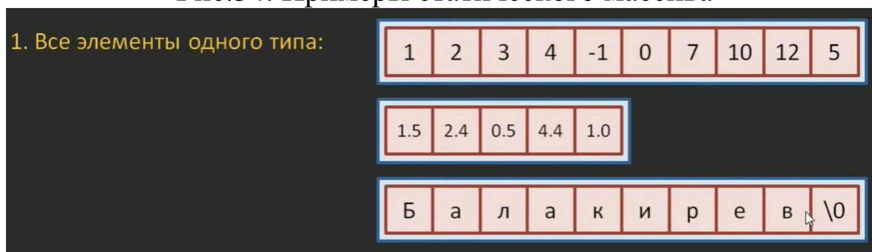


Рис.35а. Структура статического массива. Одинаковый тип данных

2) Длина массива – фиксированное число элементов и оно не меняется. Если мы задали размер массива 50 элементов, то на протяжении всей работы программы массив будет иметь длину 50 элементов. При этом значение элементов мы менять можем. Поэтому данные массивы и называются статическими (рис.).

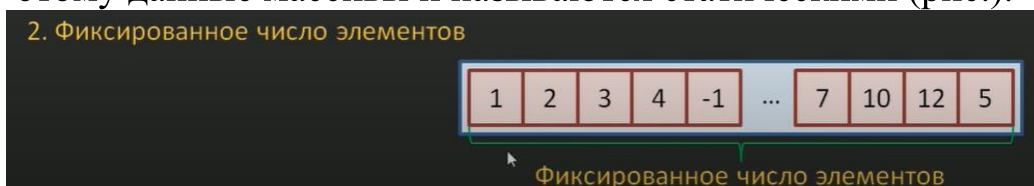


Рис.35б Структура статического массива. Длина массива

3) Все элементы статического массива располагаются в памяти последовательно друг за другом без пропусков, начиная с какого-то начального адреса. В примере целочисленный массив, каждый элемент которого занимает 4 байта (рис.35в). Таким образом, мы имеем цельную неделимую область памяти, где сохраняются значения всех элементов этого массива. Размер этого

массива в байтах можно вычислить по формуле: $size = n \cdot k$ байт.

Рис.35в Расположение элементов статического массива.

Обращение к элементам статического массива. В случае статического массива мы всегда знаем, с какого адреса данные хранятся в конкретном массиве. Эту информацию дает нам имя массива, которое, по сути, хранит адрес, начиная с которого хранятся все значения в этом массиве. Зная начальный адрес и размер, который занимает один элемент в этом массиве, легко можем обратиться к первому элементу, взять все 4 байта, начиная с адреса, указанного в имени массива. Математически это запишется, как $ar + 0 \cdot k$, где k – это размер одного элемента в памяти компьютера. Это говорит нам, начиная с какого адреса, мы можем взять первый элемент массива. Очевидно, что $ar + 0 \cdot k$ – это ar и в данном случае ar составляет 1000, мы будем ссылаться на первые 4 байта и т.д. (рис.36)

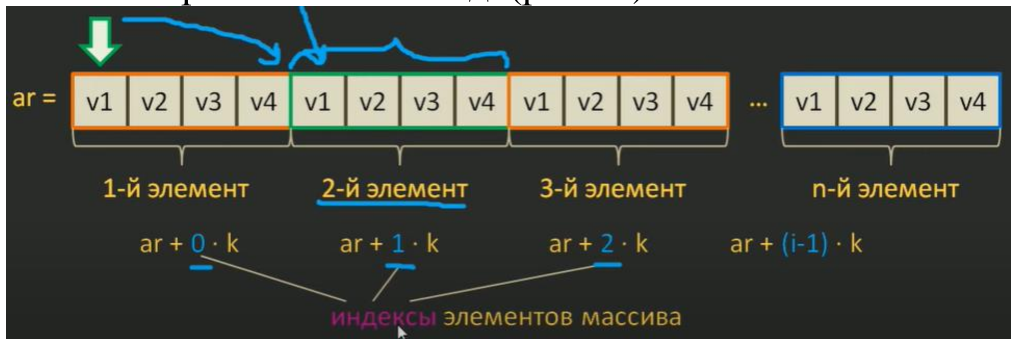


Рис.36. Иллюстрация обращения к элементам статического массива

Фактически 0, 1, 2, …, (n-1) – порядковые номера элементов массива. Формула в общем виде для j-ого индекса: $p_i = ar + j \cdot k$.

Базовые операции, выполняемые с элементами массива:

– считывание значенияvalue = ar[j];
– запись значения ar[j] = value.

Эти операции выполняются за константное время. Сложность этих операций по времени составляет O(1), что означает мгновенный доступ к элементу. Это главное преимущество данной структуры данных.

К основным операциям с массивом относятся: вставка элемента в какое-либо места массива и удаление элемента из любого места массива.

Рассмотрим, как работает вставка элемента в массив. Различают:

1) вставка с конца. Это самый простой способ. Нужно указать имя массива, в квадратных скобках индекс элемента, затем значение элемента (рис.37).

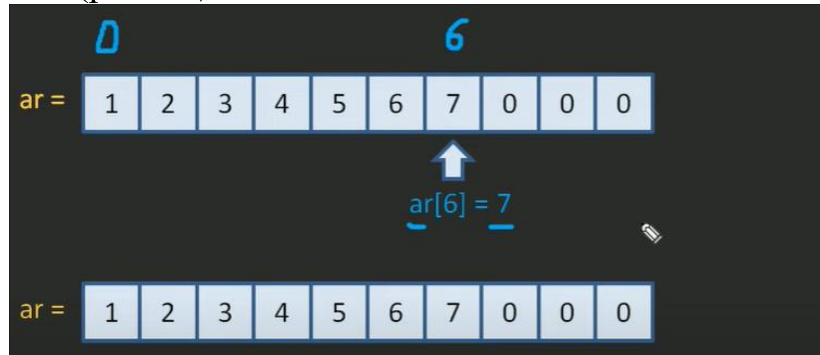


Рис.37. Иллюстрация встаки элемента в конец статического массива

2) Вставка с начала. Эта операция будет работать по-другому. Вначале мы должны сдвинуть все элементы массива вправо на один элемент. Получим массив, как на рис.38:

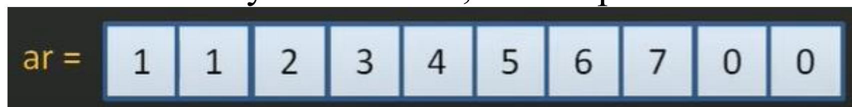


Рис.38. Иллюстрация встаки элемента в начало статического массива. Сдвиг элементов

Только после сдвига мы можем записать нужный нам элемент на место первого (рис.39):

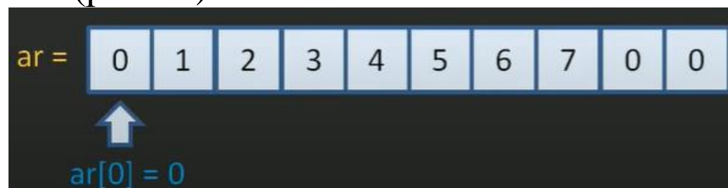


Рис.39 Иллюстрация встаки элемента в начало статического массива

Аналогичным образом осуществляется вставка элемента в любое другое место массива.

Какова вычислительная сложность такого алгоритма по времени? Если в общем случае принять длину массива n, то сложность составит O(n).

Удаление элементов из массива. Различают:

1) Удаление с конца. Допустим, мы хотим удалить 7. Мы просто уменьшаем на единицу счетчик элементов массива. Число 7 осталось, просто мы его не учитываем (рис.40).

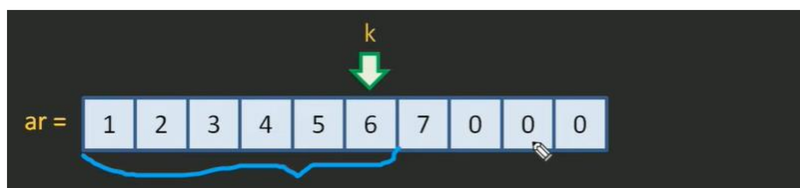


Рис.40. Иллюстрация удаления элемента с конца статического массива

2) Удаление с начала. В этом случае придется сдвинуть все элементы на одну позицию влево и изменить значение счетчика (рис.41).

Рис.41. Иллюстрация удаления элемента из начала статического массива

Получится массив, содержащий актуальные элементы. Вычислительная сложность такого алгоритма по времени, если в общем случае принять длину массива n, составит O(n).

Получим, что операции считывания/ записи выполняются быстро, а операции вставки/ удаления зависят от количества элеметов. Получим следующие оценки для основных операций (таб.4).

Таблица.4. Вычислительная сложность основных операций с статическими массивами

№	Название операции	Вычислительная сложность
1	запись	O(1)
2	чтение	O(1)
3	вставка	O(n)
4	удаление	O(n)

Глядя на результат очевидно, в случае большого количества данных и преобладания операций вставки и удаления от статического массива лучше отказаться. Если же у вас небольшое количество данных и основные операции считывание и записывание – статический массив – хороший выбор.

Подитожим сказанное.

Основными преимуществами статических массивов являются:

– скорость доступа к произвольному элементу для базовых операций;
– простота реализации и удобство использования для небольших наборов данных.

Недостатки статического массива:

- хранение данных выполняется в непрерывной области памяти, что не всегда эффективно для больших объемов данных;
- статический массив не может менять число своих элементов в процессе работы программы. Если зарезервированного места окажется недостаточно, то данные могут потеряться;
- вставка и удаление элементов выполняется за время O(n), что может быть критично при больших n.

7.2.3. Реализация статического массива в Python

Массивы в Python определяет модуль array.

Синтаксис: *Класс **array.array**(TypeCode [, инициализатор])* создает новый массив, элементы которого ограничены TypeCode, и инициализатор, который должен быть списком.

Массивы очень похожи на списки, но с ограничением на тип данных и размер каждого элемента. Размер и тип элемента в массиве определяется при его создании и может принимать следующие значения (табл.5)

Таблица 5. Коды типов данных Python для статических массивов

Код типа	Тип в Python	Минимальный размер в байтах
'b'	int	1
'B'	int	1
'h'	int	2
'H'	int	2
'i'	int	2
'I'	int	2
'l'	int	4
'L'	int	4
'q'	int	8
'Q'	int	8
'f'	float	4
'd'	float	8

Массивы изменяемы. Массивы поддерживают все списковые методы (индексация, срезы, умножения, итерации).

Методы массивов array в Python:

- **array.typecode** – TypeCode символ, использованный при создании массива.

- **array.itemsize** – размер в байтах одного элемента в массиве.
- **array.append**(x) - добавление элемента в конец массива.
- **array.buffer_info**() - кортеж (ячейка памяти, длина). Полезно для низкоуровневых операций.
- **array.byteswap**() - изменить порядок следования байтов в каждом элементе массива. Полезно при чтении данных из файла, написанного на машине с другим порядком байтов.
- **array.count**(x) - возвращает количество вхождений x в массив.
- **array.extend**(iter) - добавление элементов из объекта в массив.
- **array.frombytes**(b) - делает массив array из массива байт. Количество байт должно быть кратно размеру одного элемента в массиве.
- **array.fromfile**(F, N) - читает N элементов из файла и добавляет их в конец массива. Файл должен быть открыт на бинарное чтение. Если доступно меньше N элементов, генерируется исключение EOFError , но элементы, которые были доступны, добавляются в массив.
- **array.fromlist**(список) - добавление элементов из списка.
- **array.index**(x) - номер первого вхождения x в массив.
- **array.insert**(n, x) - включить новый пункт со значением x в массиве перед номером n. Отрицательные значения рассматриваются относительно конца массива.
- **array.pop**(i) - удаляет i-ый элемент из массива и возвращает его. По умолчанию удаляется последний элемент.
- **array.remove**(x) - удалить первое вхождение x из массива.
- **array.reverse**() - обратный порядок элементов в массиве.
- **array.tobytes**() - преобразование к байтам.
- **array.tofile**(f) - запись массива в открытый файл.
- **array.tolist**() - преобразование массива в список.

В программном коде ниже массив инициализируется значениями при его создании:

```
#подключаем модуль для работы с массивами
from array import array

#создаем статический массив целых чисел,
y=array('i',[2, 4, 6, 8])

#выводим на экран первый элемент массива
```

```
print (y[0])

#выводим на экран весь массив
for n in range (len (y)):
    print (y[n], end=' ')

#удаляем элемент с индексом 1
y.pop(1)

print(y)
```

В примере кода массив инициализируется значениями после запуска программы:

```
#подключаем модуль для работы с массивами
from array import array

#создаем статический массив целых чисел размером 5
N=5
y=array('i')

#задаем значения элементов массива с клавиатуры
for i in range (N):
    y.append(int (input()))
print (y)
```

В отличие от списка, если мы попытаемся ввести в качестве значений вещественные числа, будет выведено сообщение об ошибке (рис.42):

```
#подключаем модуль для работы с массивами
from array import array

#создаем статический массив целых чисел, специфи-
катор i
N=5
y=array('i')
for i in range (N):
    y.append(float (input()))
print (y)
```

```
2.5
Traceback (most recent call last):
  File "C:/Users/Larisa/PycharmProjects/Lecia5/arr.py", line 7, in <module>
    y.append(float (input()))
TypeError: integer argument expected, got float

Process finished with exit code 1
```

Рис.42. Сообщение интерпретатора в случае неправильно типа данных

Другие примеры объявления и инициализации статического массива:

```
#подключаем модуль для работы с массивами
from array import array

#вывод в кодировке u2641
arr = array('u', 'hello \u2641')
print (arr)

#будет напечатано
array('u', 'hello ♁')

#вывод вещественных чисел
arr = array('d', [1, 2, 3.14])
print (arr)

#будет напечатано
array('d', [1.0, 2.0, 3.14])
```

Статические массивы используются редко, когда нужно достичь высокой скорости работы. В остальных случаях массивы можно заменить другими типами данных: списками, кортежами, строками.

7.2.4. Динамический массив

Мы с вами хорошо знаем, как работает статический массив. Но у него есть существенный недостаток – неизменяемое число элементов. Когда создается статический массив, программист должен каким-то образом решить, сколько элементов этот массив должен иметь. Далеко не всегда можно указать разумное значение.

Чтобы как-то выйти из этой ситуации в мире программирования появилась более гибкая структура – динамический массив – структура

данных в виде массива, но с изменяемым, увеличивающимся числом элементов.

Приведем пример их использования. Предположим, мы просматриваем файловую систему и сохраняем имена файлов в каждом каталоге (рис.43).

Рис.43. Пример использования динамических массивов

Очевидно, число файлов может варьироваться от одного до нескольких сотен или тысяч. Если воспользоваться статическим массивом, то придется зарезервировать память под количество элементов в несколько тысяч, а это лишний расход памяти. Выходом из ситуации могут быть динамические массивы.

Для нашего примера мы можем изначально создать динамический массив размером в 100 элементов для хранения имен файлов. Если такого количества будет недостаточно, то этот размер может быть увеличен в ходе работы программы. В результате память будет расходоваться по требованию, т.е. более экономно.

Реальный размер такого массива называют физическим, а число записанных в него данных – логическим размером массива (рис.44).

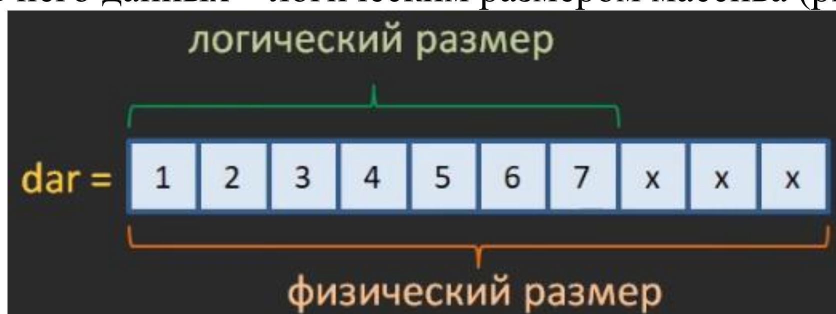


Рис.44. Разница между физическим и логическим размером массива

Давайте посмотрим, как можно организовать массив с изменяемым числом элементов. Для этого воспользуемся обычным массивом с некоторым начальным размером, допустим, в 10 элементов. Для того, чтобы программно оперировать физическим и логическим разме-

ром массива вводят две вспомогательные переменные – currentLength (логический размер), maxCapacity (физический размер).

Переменная currentLength содержит индекс следующего добавляемого элемента, а также определяет число уже записанных данных. Значение переменной maxCapacity равно максимальному количеству элементов массива, в нашем случае 10.

Теперь предположим, что мы хотим добавить элементы в конец нашего массива. Так как все элементы массива должны следовать строго друг за другом с самого начала без каких-либо пропусков, то новое значение нам следует записать в ячейку с индексом currentLength. Делается это просто. Используя имя массива, по индексу currentLength заносим нужное нам значение, в данном случае 8. Значение счетчика currentLength должно увеличиться на 1, после чего он будет указывать на следующий элемент (рис.45).

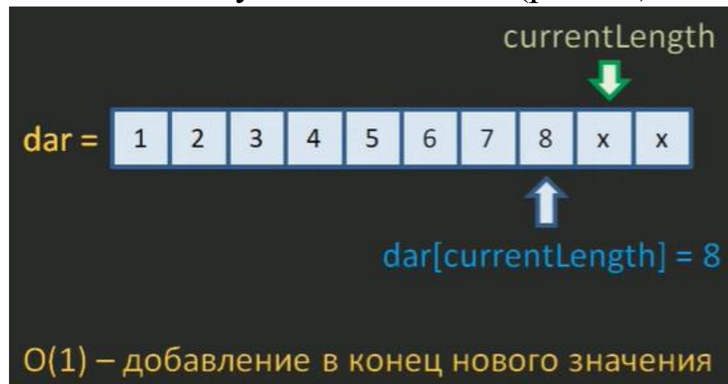


Рис. 45. Иллюстрация добавления элемента в конец динамического массива

Оценка сложности операции «добавление в конец» составляет O(1).

Допустим нам необходимо вставить новые значения после числа 5. Как в этом случае будет выглядеть алгоритм? Как в обычном статическом массиве. Необходимо сдвинуть все значения, расположенные после 5, (8, 7 и 6) на один элемент вправо (рис.46абв).

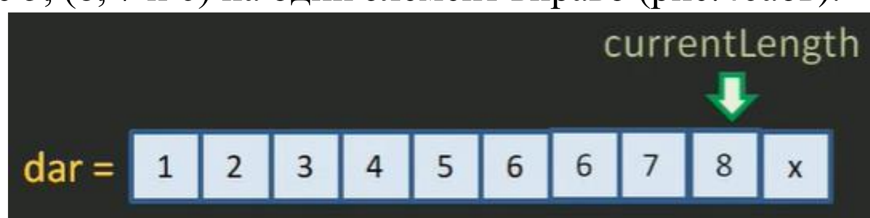


Рис.46.а. Вставка элемента после какого-то элемента. Шаг 1

Затем в элемент с индексом 5 занести новое значение, например, -4.

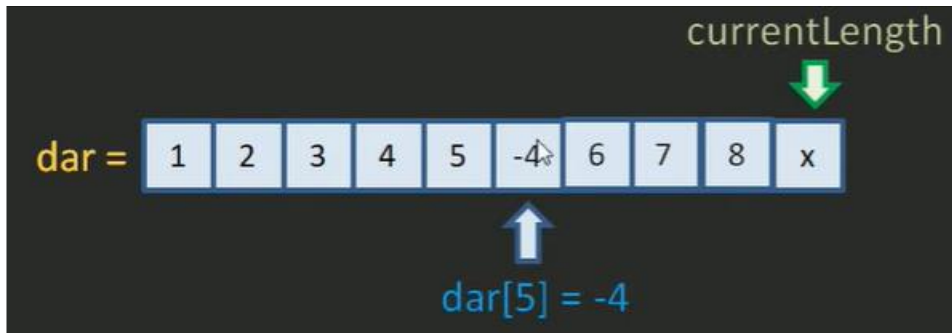


Рис.46.б. Вставка элемента после какого-то элемента. Шаг 2

После этого увеличить значение счетчика currentLength на 1, для того, чтобы он указывал на следующий элемент массива.

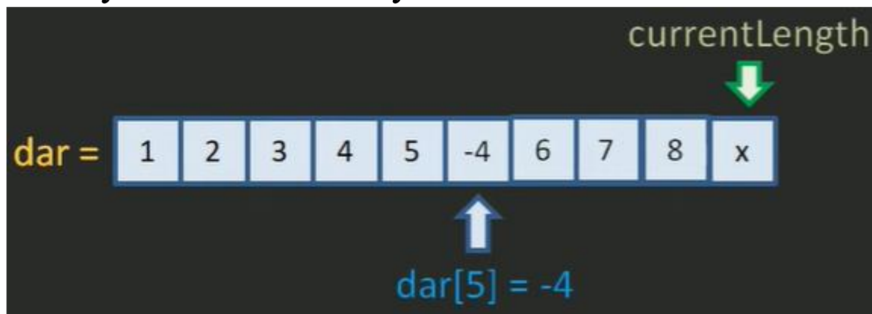


Рис.46.в. Вставка элемента после какого-то элемента. Шаг 3

Вычислительная сложность этой операции будет составлять уже O(n), где n – физический размер динамического массива.

Рассмотрим, как работают операции вставки, если все места в массиве заняты. В этом случае необходимо выделить требуемое количество байт под новый элемент (рис.47абв) и записать туда требуемое значение.

В общем случае такую операцию выполнить не получится, так как следующие ячейки памяти могут быть заняты другими процессами. В этом случае поступают следующим образом.

Вначале выделяется память под новый статический массив длиной в 2 или 3 раза больше первоначального. Размер такого массива составляет maxCapacity · 2.

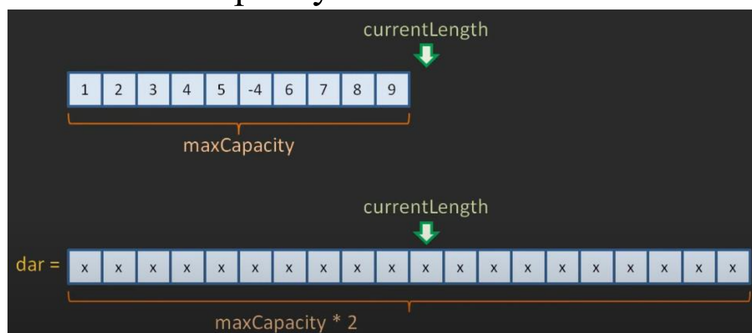


Рис.47.а. Наглядное представление операции вставки. Шаг 1

В этот новый массив копируют все прежние значения. Затем записывают новые значения по индексу currentLength.

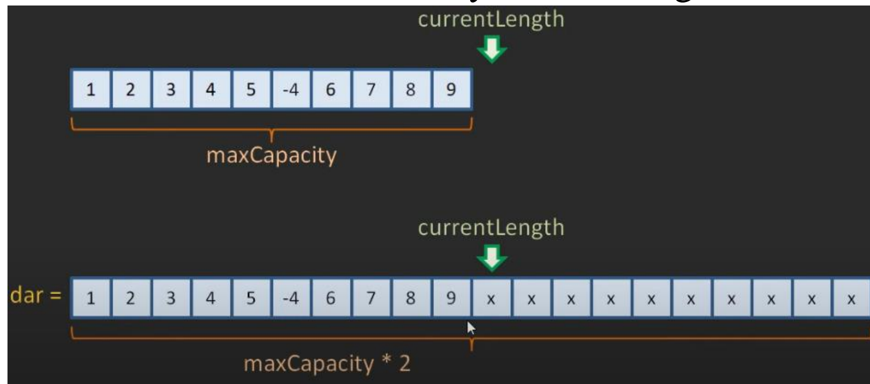


Рис.47.б. Наглядное представление операции вставки. Шаг 2

После вставки нового элемента, значение счетчика currentLength увеличивается на 1. Вычислительная сложность операции добавления значения с изменением размера массива составляет O(n).

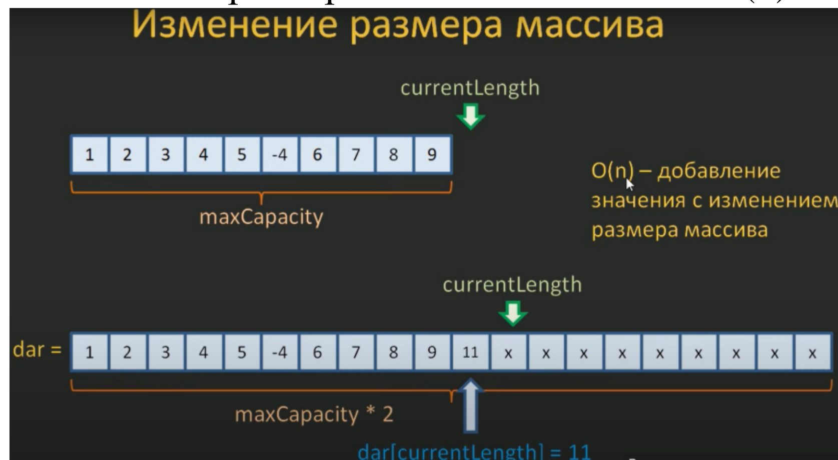


Рис.47.в. Наглядное представление операции вставки. Шаг 3

Далее, если все элементы в измененном массиве снова будут заполнены, то опять происходит удвоение или утроение размера массива и операция «вставка в конец массива» повторяется. ***Это принцип работы динамического массива.***

У вас может возникнуть вопрос: «Зачем под новый массив отводить в два или три раза больше места? Почему бы не увеличить его на требуемое число элементов?» Если нам не хватило физического размера то, скорее всего, не хватит и для нового массива с одним дополнительным элементом. Значит операцию копирования всех прежних значений в новый массив снова придется повторять. Это приводит к неоправданному расходу процессорного времени. Логичнее размер массива сразу сделать в два раза больше и минимизировать вероятность создания новых массивов с большими физическими размерами.

Исходя из таких соображений, и делают удваивание, а иногда и утраивание физического размера.

Рассмотрим, как происходит удаление элемента из динамического массива с конца. Это намного проще, так как физический размер массива не меняется. Работает операция удаления так же, как и со статическим массивом.

Допустим, мы хотим удалить элемент со значением 8 (рис.48абвг). Для этого достаточно уменьшить значение счетчика currentLength на 1 и в массиве остается ровно 8 элементов, потому что при просмотре массива мы доходим до currentLength. currentLength показывает, сколько элементов записано в динамический массив и в данном случае она принимает значение 8, поэтому ровно 8 значений мы и вычитываем. Вычислительная сложность этой операции составляет O(1).

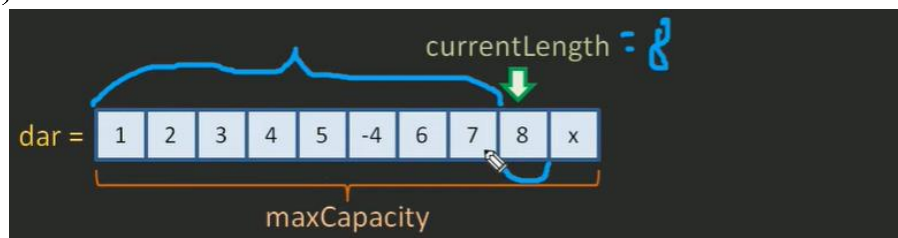


Рис.48.а. Удаление элемента из массива. Шаг 1

Рассмотрим, что будет происходить, если мы удалим элемент, находящийся не в конце, например, -4? В этом случае нам нужно сместить все элементы, стоящие правее удаляемого элемента (-4), на одну позицию влево (рис.) и переменную currentLength уменьшить на 1.

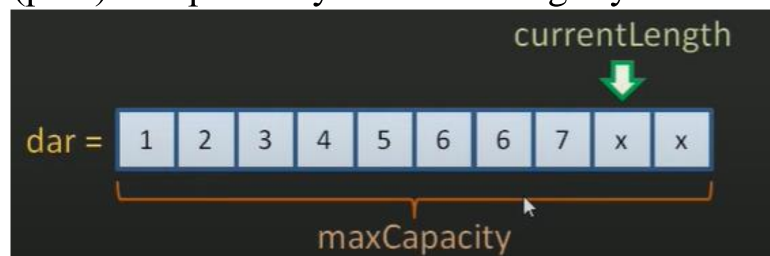


Рис.48.б. Удаление элемента из массива. Шаг 2

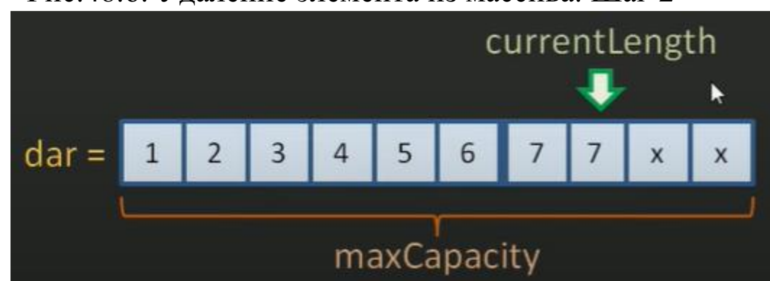


Рис.48.в. Удаление элемента из массива. Шаг 3

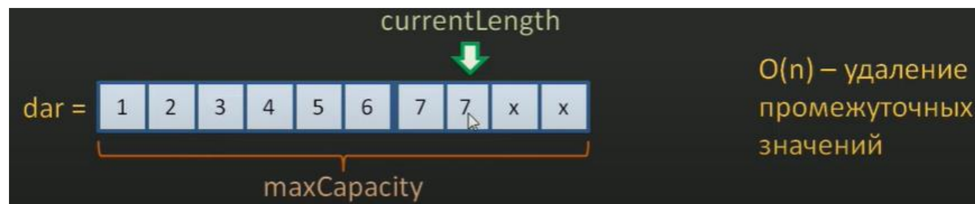


Рис.48.г. Удаление элемента из массива. Шаг 4

Сложность этой операции составляет O(n).

Еще раз отметим, что при удалении элемента физический размер массива не меняется.

7.2.5. Реализация динамического массива в Python. Списки

Всем питонистам известна коллекция типа list, которая позволяет создавать список из разных объектов. Например, так:

```
marks = [2, 2, 3, 4]
```

или так:

```
lst = [True, "Истина", 1, 1.0]
```

Можно ли считать такие списки динамическими массивами? Строго говоря, лишь с некоторым приближением, так как в массивах все элементы должны иметь единый тип данных. А во втором списке мы видим и булево значение, и строку и целое число, то есть, разные типы. Но если мы посмотрим внутрь этой структуры, то увидим, что на самом деле списки в Python реализованы как динамические массивы ссылок. И эти ссылки могут быть связаны с любым объектом, любого типа (рис.49).

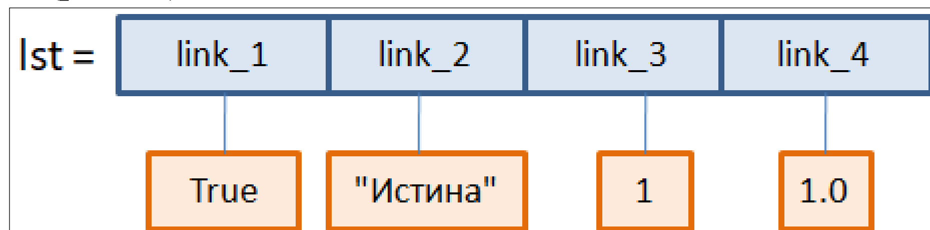


Рис.49. Структура динамического массива вPython

Отсюда и получается эффект, словно список содержит разные типы. На самом деле в нем хранятся только ссылки, ведущие на те или иные объекты и не более того. То есть, сам динамический массив содержит данные одного типа – ссылки на объекты.

Списки — это последовательности чисел, строк или каких-то ещё значений. Содержимое списка пишется в квадратных скобках, элементы списка разделяются запятой: *new_list = [<элемент1>, <элемент2>, … , <элемент n>]*

Так выглядит в коде Python пустой список:

```
s = []  # Пустой список
```

В качестве примера сохраним в переменной russian_alphabet список, состоящий из букв русского алфавита. Буква – это символ, поэтому каждый элемент – в апострофах:

```
russian_alphabet =
['а','б','в','г','д','е','ё','ж','з','и','й','к','л','м','н','о','п','р','с','т','у','ф','х','ц','ч','ш','щ','ъ','ы','ь','э','ю','я']
print (russian_alphabet)

#будет напечатано:
['а', 'б', 'в', 'г', 'д', 'е', 'ё', 'ж', 'з', 'и',
'й', 'к', 'л', 'м', 'н', 'о', 'п', 'р', 'с', 'т',
'у', 'ф', 'х', 'ц', 'ч', 'ш', 'щ', 'ъ', 'ы', 'ь',
'э', 'ю', 'я']
```

Список из чисел может выглядеть так:

```
countdown = [5, 4, 3, 2, 1, 0]
```

Можно сделать список из выражений, тогда в нём будут храниться вычисленные значения:

```
# сохраним в списках вторую и третью строки таблицы Пифагора
pithagoras_2 = [
    2*1, 2*2, 2*3, 2*4, 2*5, 2*6, 2*7, 2*8, 2*9
]
pithagoras_3 = [
    3*1, 3*2, 3*3, 3*4, 3*5, 3*6, 3*7, 3*8, 3*9
]

print (pithagoras_2)
print (pithagoras_3)

# будет напечатано:
[2, 4, 6, 8, 10, 12, 14, 16, 18]
[3, 6, 9, 12, 15, 18, 21, 24, 27]
```

Можно создать смешанный список, содержащий и символы и числа. Например:

```
list =[1, 4, 'b', 'c']
```

В этом случае мы сможем работать с числами и символами только по отдельности, например, как показано ниже:

```
list =[1, 4, 'b', 'c']

result_1=list[0]+list[1]
result_2=list[2]+list[3]

print (result_1,' ',result_2)
#будет напечатано
    5  bc
```

Если попытаться выполнить какую-то операцию одновременно с числом и символом, интерпретатор сообщит об ошибке .

Сложим первый и третий элементы, получим следующее сообщение (рис.50):

```
list=[1,4,'b','c']
result_1=list[0]+list[2]

print (result_1)
```

```
  File "C:/Users/Larisa/PycharmProjects/pr5_1/pr5_1.py", line 3, in <module>
    result_1=list[0]+list[2]
TypeError: unsupported operand type(s) for +: 'int' and 'str'

Process finished with exit code 1
```

Рис.50. Сообщение интерпретатора при попытке сложить число и символ

В этом случае необходимо преобразование – либо числа в строку, либо строки в число.

В Python для этой цели используются следующие предопределенные встроенные **функции**:

- type () возвращает тип объекта;
- int () – приведение данных к целочисленному типу;
- float () – преобразовывает данные аргументы в плавающий тип;
- str () – преобразует данные аргументы в строку;
- bool () – преобразует данные в логический тип.

Рекомендации по преобразованию типов данных в Python:

1. типы int и float можно конвертировать в str и обратно, потому что строка может включать не только символы алфавита, но и цифры.
2. строку, включающую символы алфавита и числа, нельзя конвертировать в целое или вещественное число.

С помощью функции str() приведем все элементы нашего списка к типу «строка» и выведем новую строку на экран:

```
list=[1, 4,'b','c']
result_1=" "

#приведем тип всех элементов списка к строковому типу
#сохраним в строке result_1
for i in range(0,4):
    result_1 = result_1+str(list[i])

print (result_1)

# будет напечатано:
14bc
```

Начальные значения элементов списка можно ввести с клавиатуры (рис.51):

```
seasons = [int (input('введите 5 целых чисел \n'))
for i in range(5)]
print (seasons)
```

```
введите 5 целых чисел
3
введите 5 целых чисел
6
введите 5 целых чисел
8
введите 5 целых чисел
9
введите 5 целых чисел
1
[3, 6, 8, 9, 1]
```

Рис.51. Задание значений элементов списка с клавиатуры

Основные операции со списками

Выделены следующие:

- считывание значения элемента (или группы элементов);
- изменение значения элемента;
- вставка элемента (или группы элементов) в последовательность;
- удаление элемента (или группы элементов) из последовательности;
- изменение порядка элементов в последовательности.

Считывание значения элемента (или группы элементов)

У каждого элемента списка есть свой порядковый номер – **индекс**. С помощью индекса можно получить значение элемента списка. В примере ниже выводится содержимое элементов с индексами 1 и 2:

```
print (russian_alphabet[1])
print (russian_alphabet[2])

# будет напечатано:
б
в
```

Считывание группы элементов. Срезы списков

Со списками, так же как и со строками, можно делать срезы. А именно:

1. A[i:j] - срез из j-i элементов A[i], A[i+1], …, A[j-1]:

```
a=[2, 3, 5, 7, 8, 9, 10, 11]
print (a[0:3])

#будет напечатано
[2, 3, 5]
```

2. A[i:j:-1] - срез из i-j элементов A[i], A[i-1], …, A[j+1] (то есть меняется порядок элементов, становиться обратным). В примере ниже будут выведены зачения элементов с индексами 7, 6, 5, 4, 3:

```
a=[2, 3, 5, 7, 8, 9, 10, 11]
print (a[7:2:-1])

#будет напечатано
[11, 10, 9, 8, 7]
```

Шаг можно менять. В примере ниже на экран будут выведены значения элементов с индексами 7, 5, 3:

```
a=[2, 3, 5, 7, 8, 9, 10, 11]
print (a[7:2:-2])

#будет напечатано
[11, 9, 7]
```

3. A[i:j:k] - срез с шагом k: A[i], A[i+k], A[i+2*k],… . Если значение k<0, то элементы идут в противоположном порядке. Каждое из чисел i или j может отсутствовать, что означает “начало строки” или “конец строки”. Списки, в отличие от строк, являются изменяемыми объектами: можно отдельному элементу списка присвоить новое значение. Срезы можно менять и целиком. Например:

```
A = [1, 2, 3, 4, 5]
A [2:4] = [7, 8, 9]
print (A)

#будет напечатано
[1, 2, 7, 8, 9, 5]
```

Получился список A, у которого начиная со 2-ой позиции, вставлен новый список из трех элементов 7, 8, 9 . Теперь список A стал равен [1, 2, 7, 8, 9, 5].

4. Срезы списка, элементы которого имеют отрицательные индексы. В примере ниже A[::-2] – это список из элементов A[-1], A[-3], A[-5], A[-7], которым присваиваются значения 10, 20, 30, 40 соответственно:

```
A = [1, 2, 3, 4, 5, 6, 7]
A[::-2] = [10, 20, 30, 40]

#будет напечатано
 [40, 2, 30, 4, 20, 6, 10]
```

Если не непрерывному срезу (то есть срезу с шагом k, отличному от 1), присвоить новое значение, то количество элементов в старом и новом срезе обязательно должно совпадать, в противном случае произойдет ошибка ValueError.

Для непрерывных срезов количество элементов может не совпадать, что приведет к добавлению элементов в середину списка или удалению элементов из середины. Это может быть затратной по времени операцией, если для вставки или удаления необходимо переместить в памяти значительное количество элементов.

Изменение значения элемента

Изменить значение можно **тремя способами**, аналогичными считыванию значения элементов: одиночный элемент, группа элементов и срезы. Примеры:

1) одиночный элемент. В примере ниже продемонстрировано изменение значения элемента с индексом 0:

```
A = [1, 2, 3, 4, 5, 6, 7]
A[0]=-3
print (A)

#будет напечатано
[-3, 2, 3, 4, 5, 6, 7]
```

2) замена нескольких значений:

```
A = [1, 2, 3, 4, 5, 6, 7]
for i in range (len(a)):
    A[i]=-2
    print (A)

#будет напечатано
[-2, -2, -2, -2, -2, -2, -2]
```

3) замена нескольких значений с помощью срезов. В примере ниже будут заменены указанными значениями элементы с индексами 0, 1 и 2:

```
A = [1, 2, 3, 4, 5, 6, 7]
A[:3]=[-2, 0, 0]
print (A)

#будет напечатано
[-2, 0, 0, 4, 5, 6, 7]
```

Будут заменены указанными значениями 5, 6 и 7-ой элементы:

```
A = [1, 2, 3, 4, 5, 6, 7]
A[5:7]=[-2, 0, 0]
print (A)

#будет напечатано
[1, 2, 3, 4, 5, -2, 0, 0]
```

Вставка элемента (или группы элементов) в последовательность. Вставка производится аналогично замене, номер начальной и конечной позиций должны совпадать. В примере ниже, с пятой позиции вставляются 2 элемента:

```
A = [2, 3, 5, 7, 8, 9, 10, 11]
A[5:5]=[-2, 0]
print (A)
```

```
#будет напечатано
[2, 3, 5, 7, 8, -2, 0, 9, 10, 11]
```
С пятой позиции вставляется один элемент:
```
A = [2, 3, 5, 7, 8, 9, 10, 11]
A[5:5]=[-2]
print (A)

#будет напечатано
[2, 3, 5, 7, 8, -2, 9, 10, 11]
```
Удаление элемента (или группы элементов) из последовательности. Производится аналогично, указанием диапазона индексов и пустого списка. В примере ниже будут удалены элементы с индексами 5, 6 и 7:
```
A = [2, 3, 5, 7, 8, 9, 10, 11]
A[5:7]=[]
print (A)

#будет напечатано
[2, 3, 5, 7, 8, 11]
```
Изменение порядка следования элементов списка

Упорядочивание (сортировка) списка. Под *сортировкой* будем понимать размещение набора данных в определенном порядке, что используется для облегчения поиска нужного элемента или группы элементов. От эффективности алгоритма сортировки зависит, например, производительность работы баз и банков данных.

В языке Питон есть *встроенный алгоритм сортировки* – **Timsort.** На больших списках он работает, как сортировка слиянием, а на маленьких фрагментах – как сортировка вставками. Этот алгоритм специально оптимизирован для использования в языке Питон, а также быстро работает (за линейное время), если исходный список почти отсортирован.

Сортировать можно любые объекты, которые допускают сравнения – числа, строки (они сравниваются лексикографически), списки, кортежи.

Можно объявить собственный класс объектов, определить для них функцию сравнения и тогда встроенная сортировка сможет и их сортировать.

Есть два способа вызова сортировки. Первый способ – это метод sort(), который вызывается для списка. Например:

```
import random
A = [random.randint(1, 1000) for i in range(1000)]
A.sort()
```

В этом случае проводится сортировка «на месте», то есть элементы списка переставляются. Метод sort() не возвращает значения, поэтому результат вызова метода sort нельзя использовать в арифметических выражениях или при выводе результата.

Другой способ — это функция sorted(). Она, наоборот, не модифицирует переданный ей список, а возвращает новый список. Функцию sorted() можно использовать так:

```
import random
A = [random.randint(1, 1000) for i in range(1000)]
B = sorted(A)
```

Можно использовать функцию sorted() в вводе-выводе. Вот такая программа из одной строки сортирует считанный со стандартного ввода список, состоящий из чисел, разделенных пробелами, и выводит результат на экран:

```
print(" ".join(map(str, sorted(map(int, in-
put().split())))))
```

где map() – это встроенная функция, которая позволяет обрабатывать и преобразовывать все элементы в итерируемом объекте без использования явного цикла for методом, широко известным как сопоставление (mapping). map() полезен, когда вам нужно применить функцию преобразования к каждому элементу в коллекции или в массиве, и преобразовать их в новый массив.

В коде нами использован такой инструмент, как метод. Это разновидность функции. С другими функциями, вроде print() и len(), у него много общего. Тоже мини-программа из Python, тоже аргументы в круглых скобках. Однако, в отличие от функций, метод в коде привязывается к объекту, в нашем случае методы join() и sort() – к списку.

И методу sort(), и функции sorted() можно передать дополнительный параметр reverse. Если этот параметр равен True, то сортировка будет произведена в обратном порядке, то есть по невозрастанию элементов:

```
s2="hello"
print (sorted (s2))
print (sorted (s2, reverse=True))
```

```
#будет напечатано
['e', 'h', 'l', 'l', 'o']
['o', 'l', 'l', 'h', 'e']
```

Некоторые другие операции со списками

Определение длины списка

Длина списка определяется функцией **len** (имя списка). Например, длина списка list из примера выше будет равна 4:

```
list =[1, 4, 'b', 'c']
A=len(list)

print ('Длина списка ',A)

#будет напечатано
Длина списка 4
```

Конкатенация или сложение списков

В этом случае происходит добавление одного списка в конец другого. Получится новый список:

```
pithagoras_2 = [1, 2, 3]
pithagoras_3 = [4, 5, 6]
print (pithagoras_2+pithagoras_3)
# будет напечатано
[1, 2, 3, 4, 5, 6]
```

Повторение списков (умножение списка на число)

```
pithagoras_2 =[1, 2, 3]
pithagoras_3 =[4, 5, 6]
print (pithagoras_2+pithagoras_3)
D=pithagoras_2 *3
print (D)

#будет напечатано
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

В новом списке D список pithagoras_2 повториться 3 раза.

Это позволяет по-другому организовать процесс считывания списков: сначала считать размер списка и создать список из нужного числа элементов, затем организовать цикл по переменной i, начиная с числа 0, и внутри цикла считывать i-й элемент списка (рис. 52):

```
#задаем количество элементов списка
pithagoras_2 = [0] * int(input())
#ввод значений элементов списка по длине списка
```

```
for i in range(len(pithagoras_2)):
    pithagoras_2[i] = int(input())
print (pithagoras_2)

#будет напечатано
```

```
4
6
7
8
9
[6, 7, 8, 9]

Process finished with exit code 0
```

Рис.52. Процесс считывания списка

Объединение списков

Сделать одну строку из элементов списка позволяет метод **join.** Он принимает список как аргумент и возвращает одну строку, например:

```
bayan = ['[', ':', '|', '|', '|', '|', '|', '|',
'|', '|', '|', '|', ':', ']']

#превращаем список в строку методом join()
print(''.join(bayan))

#будет напечатано:
[:||||||||||:]
```

Вызов метода записывают всегда так: *объект.имя_метода (аргументы_метода).*

В таблице 6 приведены основные методы работы со списками.

Таблица 6. Основные операции и методы работы со списками

Метод	Что делает
list.append(x)	Добавляет элемент в конец списка
list.extend(L)	Расширяет список list, добавляя в конец все элементы списка L
list.insert(i, x)	Вставляет перед i-ым элементом значение x
list.remove(x)	Удаляет первый элемент в списке, имеющий значение x. ValueError, если такого элемента не существует
list.pop([i])	Удаляет i-ый элемент и возвращает его. Если индекс не указан, удаляется последний

	элемент
list.index(x, [start [, end]])	Возвращает положение первого элемента со значением x (при этом поиск ведется от start до end)
list.count(x)	Возвращает количество элементов со значением x
list.reverse ()	Разворачивает список в обратном порядке
list.clear()	Очищает список
x in A	Проверяет, содержится ли элемент в списке. Возвращает true или false
x not in A	Проверяет, не содержится ли элемент в списке. Возвращает true или false s2="hello" print(sorted(s2)) print(sorted(s2, reverse=True)) q='B' not in s2 print (q)
min(A) / max(A)	Минимальный (максимальный) элемент списка
A.index(x)	Индекс первого вхождения элемента в список A. При отсутствии генерирует исключение ValueError
A.count(x)	Количество вхождения элемента в список A
sum(A)	Возвращает сумму элементов списка

Вывод элементов списка

Как показано в примерах выше, вывести элементы списка A можно одной инструкцией print (A), при этом будут выведены квадратные скобки вокруг элементов списка и запятые между элементами списка.

Такой вывод неудобен, чаще требуется просто вывести все элементы списка в одну строку или по одному элементу в строке. Приведем два примера, также отличающиеся организацией цикла.

1 способ. Элементы списка будут выведены в столбец, каждый раз с новой строки. Здесь в цикле меняется индекс элемента i, затем выводится элемент списка с индексом i.

```
for i in range (len (pithagoras_2)):
    print (pithagoras_2 [i])
```

```
#будет напечатано
9
8
7
6
```

2 способ. Элементы списка будут выведены в одну строку и разделены пробелом. При этом в цикле меняется не индекс элемента списка, а само значение переменной, т.е. переменная elem будет последовательно принимать значения 9, 8, 7, 6:

```
for elem in pithagoras_2:
    print (elem, end = ' ')

#будет напечатано
9  8  7  6
```

Рассмотрим другие эффективные и быстрые алгоритмы обработки списков.

В Python самый простой способ поиска объекта – использовать операторы членства, названные так, потому что они позволяют нам определить, является ли данный объект членом коллекции. Эти операторы можно использовать с любой итерируемой структурой данных в Python, включая строки, списки и кортежи:

in – возвращает True, если данный элемент является частью структуры.

not in – возвращает True, если данный элемент не является частью структуры.

Операторов членства достаточно, когда все, что нам нужно сделать, это выяснить, существует ли подстрока в данной строке, или определить, пересекаются ли две строки, списки или кортежи с точки зрения содержащихся в них объектов.

В большинстве случаев нам нужна позиция элемента в последовательности, помимо определения того, существует он или нет. Операторы членства не соответствуют этому требованию. Поэтому для определения позиции элемента используются другие типовые алгоритмы.

Рассмотрим некоторые из них.

Поиск элемента в упорядоченном списке. Рассмотрим алгоритм **быстрого поиска элемента** в упорядоченной совокупности a_1, ….., a_n. При этом будем считать, что a_1, ….., a_n – целочисленный список, b – некоторое целое число.

Пусть $a_1 < \ldots < a_n$. Рассмотрим задачу: входит ли данное число b в список $a_1, \ldots, a_n$, и если входит, то каково значение p, для которого $a_p = b$?

Тривиальный алгоритм решения этой задачи основывается на последовательных сравнениях b с элементами $a_1, \ldots, a_n$. В этом случае среднее число требуемых сравнений можно считать равным n/2. Известен алгоритм, который требует гораздо меньших затрат.

Предположим, что в списке $a_1, \ldots, a_n$ имеется элемент, равный b, т.е. существует такое p, что $a_p = b$. По результату любого сравнения $a_s < b$ $(1 \leq s \leq n)$, где $s = int\left(\frac{1+n}{2}\right)$, мы сразу определяем, лежит ли p в диапазоне от 1 до s или же в диапазоне от s + 1 до n: второе будет иметь место, если неравенство $a_s < b$ справедливо, а первое – если несправедливо.

Если s находится примерно посередине между 1 и n, то сравнение $a_s < b$ будет сужать диапазон поиска примерно вдвое. Этот прием можно использовать многократно. Получается алгоритм, называемый алгоритмом деления пополам.

В соответствии с этим алгоритмом надо взять первоначально 1 и n в качестве границ поиска индекса элемента; далее до тех пор, пока границы не совпадут, шаг за шагом сдвигать эти границы следующим образом: сравнить b с a_s где s – целая часть среднего арифметического границ, если $a_s < b$, то заменить прежнюю нижнюю границу на s + 1, оставив верхнюю границу без изменения, иначе оставить без изменения нижнюю границу, а верхнюю заменить на s. Поиск закончится, когда границы совпадут.

Сказанное можно записать в виде последовательности операторов (p и q – верхняя и нижняя границы), когда p и q совпадут, p дает результат выполнения.

Схематично процесс поиска для $a_1, \ldots, a_n$ соответственно равных 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 и b = 19 представлен на рис.53.

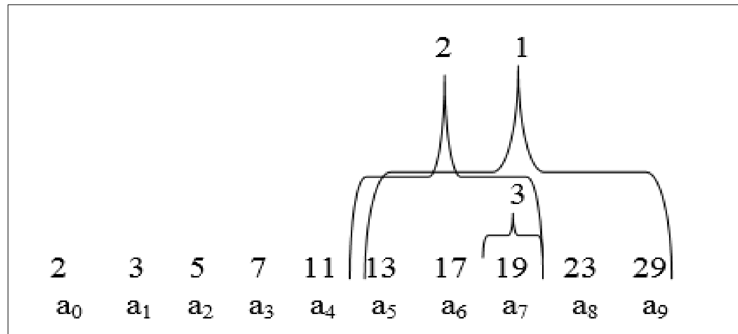


Рис. 53. Процесс поиска с использованием алгоритма деления пополам

Заметим следующее. Мы исходим из предположения, что среди элементов $a_1,..., a_n$ имеется такой, который равен b. Если заранее неизвестно, имеется ли такой элемент, то, получив p, необходимо дополнительно проверить, действительно ли a_p = b. Если обнаружится, что равенство несправедливо, то из этого будет следовать, что среди a_1, …, a_n нет элемента, равного b. Программа ниже демонстрирует применение этого алгоритма:

```
n=10
b=19
a= [2,3,5,7,11,13,17,19,23,29]
p=1;
q=n;
while (p<q):
    s=int((p+q)/2)
    if (a[s]<b):
        p=s+1
    else:
        q=s
if (a[p]==b):
    print ("number ", p)
else:
    print ("now")

#будет напечатано
number 7
```

Метод «пузырька». Данный метод относится к сортировкам обменом. Метод «пузырька» получил своё название оттого, что продвижение максимальных элементов списка к его вершине происходит постепенно, подобно всплытию пузырька на поверхность воды. Этот метод требует нескольких проходов списка. На каждом проходе сравнивается пара соседних друг с другом элементов.

Если пара расположена в порядке возрастания, переставляем эти элементы местами. В противном случае элементы остаются на исходных позициях. Процедура должна быть повторена n-1 раз для гарантированного достижения результата. Пример поэтапной работы алгоритма показан на рис. 54. Изображенные на каждом проходе перестановки должны выполняться последовательно слева направо.

Листинг программы демонстрирует реализацию этого метода:

```
a=[0, 5, 3, 7, 9]
temp=0
for i in range (len (a)):
    for j in range (len (a)-1):
        if a[j]<a[j+1]:
            temp=a[j]
            a[j]=a[j+1]
            a[j+1]=temp
for i in a:
    print (i, end = ' ')

#будет напечатано
9   7   5   3   0
```

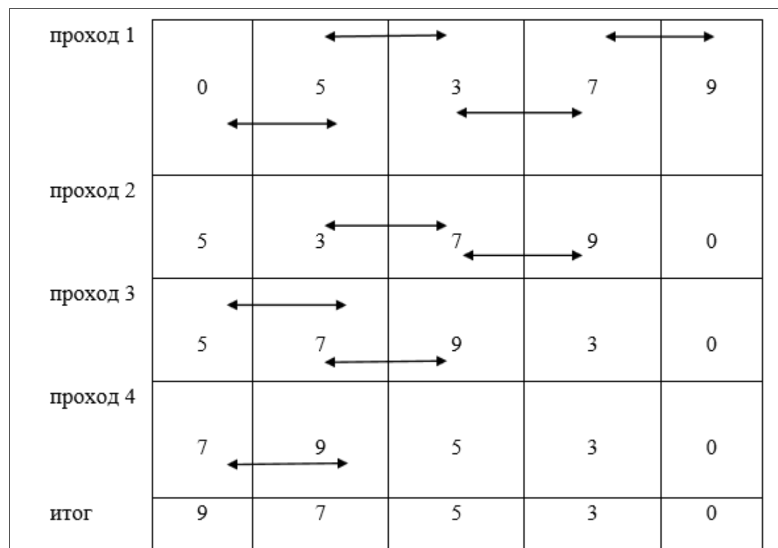


Рис. 54. Алгоритм сортировки «пузырьком»

Сортировка выбором. Алгоритм состоит в том, что выбирается наименьший элемент списка и меняется местами с первым элементом, затем рассматриваются элементы, начиная со второго, и наименьший из них меняется местами со вторым элементом и так да- лее n-1 раз (при последнем проходе цикла при необходимости меня- ются местами предпоследний и последний элементы массива). Последовательность шагов при n = 5 изображена на рис. 55 ниже.

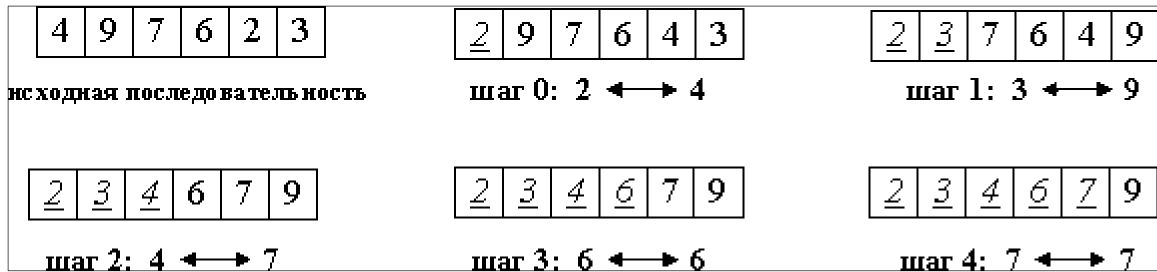


Рис.55.Алгоритм сортировки выбором

Вне зависимости от номера текущего шага i последовательность a[0]...a[i] (выделена курсивом) является упорядоченной. Таким образом, на (n-1)-м шаге вся последовательность, кроме a[n], оказывается отсортированной, а a[n] стоит на последнем месте по праву: все меньшие элементы уже ушли влево.

Код программы демонстрирует реализацию этого метода:

```
a=[1, 3, 4, 5, 6,7,1]
for i in range (len (a)-1):
    index=i
    for j in range (i+1, len (a)):
        if a[j]<a[i]:
            index=j
            temp=a[i]
            a[i]=a[index]
            a[index]=temp
for i in a:
    print (i, end = ' ')

#будет напечатано
1   1   3   4   5   6   7
```

Циклический сдвиг элементов списка. При циклическом сдвиге вправо выталкиваемые элементы с конца списка заполняют освобождающиеся места в начале списка. Например, при сдвиге вправо на 3 разряда списка A [1, 2, 3, 4, 5, 6, 7] получаем список A [5, 6, 7, 1, 2, 3, 4]. Ниже представлена соответствующая программа:

```
a=[1, 3, 4, 5, 6,7,1]
print ("введите m\n")
n=len (a)
m=int(input())
m= m % n
temp=0
for i  in range (m):
```

```
    temp=a[n-1]
    for j in range (n-1):
         a[n-j-1]=a[n-j-2]
    a[0]=temp
for i in a:
    print (i, end = ' ')

#будет напечатано
3
6  7  1  1  3  4  5
```

Вопросы для самоконтроля к параграфам 7.2.2. – 7.2.5

1. Перечислите основные характеристики типа данных «статический массив».
2. Перечислите основные характеристики типа данных «динамический массив».
3. Чем отличается динамический маасив от статического?
4. Перечислите характеристики типа данных «список», которые вы знаете.
5. Как проверить наличие элемента в списке?
6. Назовите основные операции со списками. Приведите примеры.
7. Чем операции отличаются от методов работы со списками? Продемонстрируйте на примерах.
8. Как можно изменить значение элемента списка? Приведите примеры.
9. Приведите примеры разных способов вывода элементов списка.
10. Чем отличаются методы append() и extend()?
11. Какие параметры можно передавать при срезах списков?
12. Что произойдет со списком lst1 в первом и втором случаях? Поясните результат.

Случай 1	Случай 2
lst1 = [1, 2, 3, 14, 33, 1, 9]	lst1 = [1, 2, 3, 14, 33, 1, 9]
lst2 = [1, 2, 3, 14, 33, 1, 9]	lst2 = lst1
lst2.append(789)	lst2.append(789)

13. Какие способы создания списка из символов строки вы знаете?
14. Приведите примеры многомерных списков.
15. Как перебрать элементы списка? Приведите пример.
16. Продемонстрируйте примерами способы задания значений элементов списка.

17. Расскажите, как работают со списками функции map, zip, filter, reduce?
18. Как удалить элемент списка? Приведите пример.
19. Как добавить элемент в список? Приведите пример.
20. Как перебирать более 2 списков одновременно? Приведите пример.
21. Изменяем ли список?
22. Должен ли список быть однородным?
23. В чем разница между добавлением и расширением списка? Покажите на примерах.
24. Что делает метод del () со списком? Покажите на примере.
25. Что делает метод remove () со списком? Покажите на примере.
26. Каким способом можно удалить дубликаты из списка? Покажите на примере.
27. Как можно перебрать индексы элементов списка? Покажите на примере.
28. Как можно объединить два списка? Покажите на примере.
29. С помощью какого метода можно подсчитать количество появлений одного и того же элемента в списке? Приведите пример.

Тест самоконтроля к параграфам 7.2.2. – 7.2.5

1. Для каких типов элементов списка метод sort() работает без ошибок? а) для однотипных элементов списка; б) только для строковых; в) только для целочисленных; г) для любых типов элементов.
2. Какой метод отвечает за изменение порядка следования элементов списка? а) reverse (); б) rebuild (); в) rotate (); г) reorder ().
3. Что представляет собой Python-список? а) разновидность FIFO-очереди; б) позиционно упорядоченную коллекцию с произвольными типами элементов; в) многомерный массив; г) упорядоченный массив.
4. За что отвечает следующий фрагмент кода:
   l = [123, ‘hello’, 1.23]
   l.insert(1, ‘inserted’)
   а) замена элемента с индексом 1 на указанный элемент; б) проверка наличия указанного элемента с указанной позицией; в) удаление указанного элемента; г) добавление элемента с индексом 1.
5. Какой метод отвечает за добавление элемента в конец списка? а) include (); б) append (); в) extend (); г) add ().

6. Какой метод отвечает за удаление элемента в списке с указанной позицией? а) delete (); б) pop (); в) clear (); remove ().
7. Какой метод отвечает за удаление указанного значения элемента в списке? а) pop (); б) delete (); в) clear (); г) remove ().
8. Какой максимальный размер у списка в Python? а) фиксированного размера нет; б) 100 тыс. элементов; в) 1 млн. элементов; г)10 тыс. элементов.
9. Выберите одно верное утверждение про списки: а) списки имеют фиксированный размер; б) списки представляют собой массивы; в) списки в Python являются изменяемыми; г) списки невозможно сортировать.
10. Какая функция отвечает за вывод длины списка? а) getLen(); б) length (); size (); г) len ().
11. Ниже представлены утверждения о списках. Какие из них верны? а) один и тот же объект может появляться в списке несколько раз; б) размеры списка четко не определены; в) все элементы списка должны быть одного типа; г) в списке может содержаться любой тип данных, кроме других списков.
12. Задан список a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']. Ниже представлены несколько программ. В каких из них вывод указан правильно? а) print(a[4::-2]); б) max(a[2:4] + ['grault']); в) a[:] is a; г) print(a[-6]); д) print(a[-5:-3]).
13. Задан список x = [10, [3.141, 20, [30, 'baz', 2.718]], 'foo']. Какая из команд позволяет получить доступ к символу ‘z’? а) print (x[1][2][1]); б) print (x[1][2][1][2]); в) x[4]; г) x[4][2].
14. Задан список x = [10, [3.141, 20, [30, 'baz', 2.718]], 'foo']. Какая из команд, указанных ниже позволяет получить доступ к ['baz', 2.718]? а) x[4][2]; б) x[1][2][1+2]; в) x[1][2][1]; г) x[1][2][1:];
15. Задан список a = [1, 2, 3, 4, 5]. Какая(ие) из строк кода, удаляющая(ие) элемент, даст список [1, 2, 4, 5]? а) del a[2]; б) a[2:2]=[]; в) a[2:3]=[]; г) a.remove (3); д) a[2]=[].
16. Задан список a=[‘a’, ‘b’, ‘c’]. Какие из строк кода корректно добавляют символы ‘d’ и ‘e’ в конец списка? а) a+=’de’; б) a.append ([‘d’, ‘e’]); в) a[len (a):]=[‘d’,’e’]; г) a.extend ([‘d’,’e’]; д) a+=[’d’, ‘e’]; е) a[-1:]=[‘d’,’e’].
17. Задан список a=[1, 2, 7, 8]. Какой вариант использования оператора среза даст список a=[1, 2, 3, 4, 5, 6, 7, 8]? а) a[2:]=[3, 4, 5, 6]; б) a[1:2]=[3, 4, 5, 6]; в) a[2:2]=[3, 4, 5, 6]; г) a[2]=[3, 4, 5, 6].

18. Какие из перечисленных выражений создадут список ровно из трех элементов? а) list(range(3)); б) 'asd'.split(); в) a = 1, 2, 3; г) 'a b c'.split(' ')
19. Какой будет результат выполнения кода?

```
a = [1, 2, 3]
if a[2] < 3:
   print(a[a[1]])
else:
   print(a[1])
```

а) 1; б) 2; в) 3; г) 0.
20. Отметьте все правильные утверждения о статических массивах в языке Python: а) элементы списка могут быть разных типов; б) все элементы списка должны быть одного типа; в) элементы могут нумероваться с единицы; г) элементы всегда нумеруются с нуля; д) размер списка может меняться во время работы программы.
21. Отметьте все правильные утверждения о динамических массивах в языке Python: а) элементы списка могут быть разных типов; б) все элементы списка должны быть одного типа; в) элементы могут нумероваться с единицы; г) элементы всегда нумеруются с нуля; д) размер списка может меняться во время работы программы.

7.2.6. Дополнительные источники по типам данных Python

1. Васильев А. Программирование на Python в примерах и задачах. –Москва: Эксмо, 2021. – 616 с.
2. Структуры данных. Список. Режим доступа: https://docs.python.org/3/tutorial/datastructures.html
3. Операции со списками на примерах. Режим доступа: https://pythonru.com/primery/python-spiski-primery
4. Работа со списками. Режим доступа: https://devpractice.ru/python-lesson-7-work-with-list/
5. Статические массивы. Режим доступа: https://docs-python.ru/standart-library/modul-array-python/

Практическая работа № 6
ОРГАНИЗАЦИЯ СПИСКОВ. ОПЕРАЦИИ СО СПИСКАМИ

Задание

Напишите программы к заданиям, указанным в индивидуальном варианте.

Оформите отчет по практической работе, содержащий:

- титульный лист;
- содержание;
- ответ на вопросы самоконтроля;
- для каждого варианта:
    - текст задания с указанием варианта;
    - блок-схему алгоритма решения задачи;
    - плановую обработку ошибок;
    - программный код на языке Python версии 3.8, соответствующий блок-схеме;
    - unit-тесты;
    - скриншоты результата работы программы для каждого unit-теста.

ВАРИАНТ 1. Напишите программу, в которой есть функция для заполнения вложенного списка. Список заполняется натуральными числами «змейкой»: сначала заполняется первая строка, затем последний столбец (сверху вниз), последняя строка (справа налево), первый столбец (снизу вверх), вторая строка (слева направо), и т.д.

ВАРИАНТ 2. Дан список, состоящий из строк разной длины, среди которых есть и пустые строки. Получить новый список, содержащий только непустые строки.

ВАРИАНТ 3. Дан произвольный список чисел, необходимо найти «бесполезное число» в этом списке. Решение реализовать с помощью функции. Примечание: бесполезным числом является число, полученное делением максимального числа списка на его длину.

ВАРИАНТ 4. Даны два списка, состоящие из строчных латинских букв. Построить новый список, в который войдут только общие символы из двух списков в алфавитном порядке и без повторений.

ВАРИАНТ 5. Напишите программу с функцией, которая для списка, переданного аргументом, возвращает список из двух элементов: значение наибольшего элемента в списке и индекс этого элемента в списке (если таких элементов несколько, то индекс первого из таких элементов).

ВАРИАНТ 6. Дан список, состоящий из букв слова «телегазета». Написать программу, которая проверяет любое введенное пользователем слово и выдает сообщение, можно ли из его букв составить это слово.

ВАРИАНТ 7. Напишите программу, в которой пользователь вводит целое число, а программа формирует список, который состоит из цифр, входящих в это число. Предложите способы создания списка, при котором цифры, формирующие число, включаются в список в прямом и обратном порядке.

ВАРИАНТ 8. Напишите программу, в которой создается список из случайных чисел. В матрице, которая реализуется данным вложенным списком, удаляется строка и столбец. Номер строки и номер столбца, которые нужно удалить, вводятся пользователем.

ВАРИАНТ 9. Из натуральных чисел, принадлежащих отрезку [101 000 000; 102 000 000], сформируйте список из чисел, у которых ровно три различных чётных делителя. Выведите элементы списка в порядке возрастания.

ВАРИАНТ 10. Назовём нетривиальным делителем натурального числа его делитель, не равный единице и самому числу. Например, у числа 6 есть два нетривиальных делителя: 2 и 3. Из натуральных чисел, принадлежащих отрезку [123456789; 223456789] и имеющих ровно три нетривиальных делителя, сформируйте список. Элементы списка расположите в порядке возрастания.

ВАРИАНТ 11. Напишите программу, в которой создается числовой список. Список заполняется случайными числами. Затем между каждой парой элементов этого списка вставляется новый элемент, равный сумме значений соседних элементов.

ВАРИАНТ 12. Рассмотрим произвольное натуральное число, представим его всеми возможными способами в виде произведения двух натуральных чисел и найдём для каждого такого произведения разность сомножителей. Например, для числа 16 получим: 16 = 16*1 = 8*2 = 4*4, множество разностей содержит числа 15, 6 и 0. Из натуральных чисел, принадлежащих отрезку [1 000 000; 2 000 000], у которых составленное описанным способом множество разностей будет содержать не меньше трёх элементов, не превышающих 100, сформируйте список. Элементы списка расположите в порядке возрастания.

ВАРИАНТ 13. Из натуральных чисел, принадлежащих отрезку [200 000 000; 400 000 000], которые можно представить в виде $N = 2^m$

$\cdot\ 3^n$, где m – чётное число, n – нечётное число, сформируйте список. Элементы списка расположите в порядке возрастания.

ВАРИАНТ 14. Пусть M(N) – произведение различных натуральных делителей натурального числа N, не считая единицы. Если у числа N меньше 5 таких делителей, то M(N) считается равным нулю. Из натуральных чисел, превышающих 200 000 000, найдите 5 наименьших натуральных чисел, для которых 0 < M(N) <N, сформируйте список.

ВАРИАНТ 15. Из целых чисел, принадлежащих отрезку [190061; 190080] и имеющих ровно 4 различных четных делителя, сформируйте список, состоящий из вложенных списков. Каждый из вложенных списков содержит найденные делители для каждого числа. Элементы вложенного списка расположить в порядке убывания.

ВАРИАНТ 16. Напишите программу, которая ищет среди целых чисел, принадлежащих числовому отрезку [45006; 50221], простые числа, оканчивающиеся на число 19. Сформируйте список из найденных чисел. Выведите элементы списка в порядке возрастания, слева от каждого числа выведите его номер по порядку.

ВАРИАНТ 17. Пусть M – сумма минимального и максимального натуральных делителей целого числа, не считая единицы и самого числа. Если таких делителей у числа нет, то значение M считается равным нулю. Среди целых чисел, больших 700 000, найдите такие, для которых значение M оканчивается на 8. Сформируйте список из пяти первых найденных чисел и соответствующих им значениям M.

ВАРИАНТ 18. Напишите программу, в которой создается числовой список. Список заполняется случайными числами. Затем между каждой парой элементов этого списка вставляется новый элемент, равный сумме значений соседних элемент.

ВАРИАНТ 19. На вход программы подается 10 строк, представляющих из себя различные слова. Создать список, состоящий из уникальных букв слов. Элементы списка отсортировать по возрастанию.

ВАРИАНТ 20. На вход программы подается 10 строк, представляющих из себя различные слова, и слово «ИНФОРМАТИКА». Создать список, состоящий из уникальных букв слов. Проверить, можно ли из элементов списка составить заданное слово.

ВАРИАНТ 21. В заполненном наполовину списке, не создавая дополнительный список, продублировать все элементы списка с сохранением порядка их следования. Например, из списка A=[1, 2, 3] необходимо получить список B=[1, 1, 2, 2, 3, 3]. Затем сожмите спи-

сок B, выбросив из него дубликаты, не используя дополнительный список.

ВАРИАНТ 22. Задан список целых чисел. найти в этом списке минимальный элемент m и максимальный элемент M. Вывести в порядке возрастания все целые числа из интервала (m; M), не входящие в данный список.

7.2.7. Строковый тип

Строки являются упорядоченной последовательностью символов. Длина строки ограничивается объемом оперативной памяти компьютера. Символом конца строки является «\n». Как все последовательности, строки поддерживают обращение к элементу по индексу, получение среза, конкатенацию (+), повторение (*), проверку на вхождение (in).

Строки относятся к неизменяемым типам данных, поэтому практически все строковые методы возвращают новую строку. Созданную строку изменить нельзя.

В примере ниже, при попытке изменить символ ‘P’ на символ ‘I’ компилятор выводит ошибку TypeError (рис.56):

```
s="Python"
print(s[0])
s[0]="I"
```

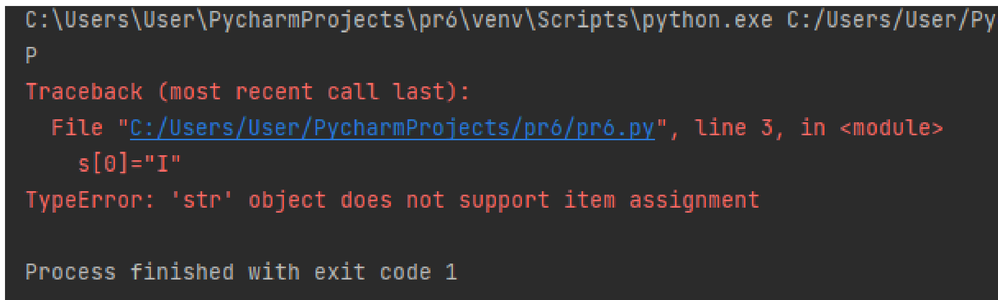


Рис.56. Сообщение компилятора при попытке изменить строку

Язык Python 3.* поддерживает следующие строковые типы:

– str – Unicode-строка. Авторами имеется в виду некая абстрактная Unicode кодировка, не конкретная, например, utf-8. При работе с такой строкой необходимо преобразовывать ее в последовательность байтов в какой-либо кодировке. Например:

```
print (type ("строка1"))

# будет напечатано
```

```
<class 'str'>

print (type ("строка1"))
i="строка1".encode(encoding='cp1251')
print (i)

#будет напечатано
<class 'str'>
b'\xf1\xf2\xf0\xee\xea\xe01'
```

– bytes – неизменяемая последовательность байтов. Каждый элемент последовательности хранит целое число от 0 до 255, которое обозначает код символа. Объект bytes поддерживает большинство строковых методов, и, если это возможно, выводится как последовательность символов. Однако, доступ по индексу возвращает целое число, а не символ. Например:

```
s=bytes ("море", "cp1251")
print (s)

#будет напечатано
b'\xec\xee\xf0\xe5'
```

Объект bytes может содержать как однобайтовые, так и многобайтовые символы. Функции и методы строк некорректно работают с многобайтовыми кодировками. Например, функция len() вернет количество байт, а не символов.

```
#кодировка однобайтовая
print (len (bytes ("море", "cp1251")))

#кодировка многобайтовая
print (len (bytes ("море", "utf-8")))

# будет напечатано
4
8
```

– bytearray – изменяемая последовательность байтов. Аналогичен типу bytes, но позволяет изменять элементы по индексу и содержит дополнительные методы, позволяющие добавлять и удалять элементы. Например:

```
l=bytearray("str", "cp1251")
```

```
#34 - код символа "
l[0]=34
print (l)

# будет напечатано
bytearray(b'"tr')
```

Во всех случаях, когда речь идет о текстовых данных, следует использовать тип str. Именно этот тип называется в Питоне строкой. Два других типа используются для хранения бинарных данных и для промежуточного хранения текстовых данных.

Создание строки

Создать строку можно несколькими способами.

Способ 1. с помощью функции str (объект, кодировка, обработка ошибок). Если параметры не указаны вообще, возвращается пустая строка.

Если указан только первый параметр, функция возвращает строковое представление любого объекта (рис.57):

```
list=[1, 2, 'ddd']
print (str(list))
```

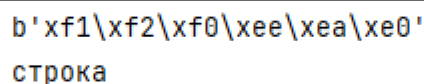
```
[1, 2, 'ddd']
```

Рис.57. Результат работы функции str

Чтобы получить строку из объекта типа bytes и bytearray следует указать кодировку во втором параметре (рис.59):

```
#вывод строки без указания кодировки
print (str (b"xf1\xf2\xf0\xee\xea\xe0"))

#вывод строки с указанием кодировки
print (str (b"\xf1\xf2\xf0\xee\xea\xe0",
'cp1251'))
```



```
b'xf1\xf2\xf0\xee\xea\xe0'
строка
```

Рис.58. Результат работы функции str в случае указания второго параметра

В третьем параметре могут быть указаны значения:

– strict – при ошибке возбуждается исключение UnicodeDecoderError, значение по умолчанию. Например:

```
print (str ("ascii","strict"))
```

Будет выведено сообщение об ошибки и пояснение к ошибке – «такая стратегия декодирования не поддерживается функцией str» (рис.60).

```
Traceback (most recent call last):
  File "C:/Users/Larisa/PycharmProjects/Spiski/spiski.py", line 3, in <module>
    print (str ("ascii","strict"))
TypeError: decoding str is not supported

Process finished with exit code 1
```

Рис.59. Результат работы функции str в случае указания третьего параметра

- replace – неизвестный символ заменяется символом, имеющим код \uFFFD;
- ignore – неизвестные символы игнорируются.

Способ 2. Указать строку между апострофами или кавычками: s="строка". Разницы нет. Все специальные символы в строках интерпретируются. Например, последовательность символов '\n'интерпретируется в символ новой строки. Примеры и результаты вывода представлены ниже:

```
s1="строка1"
s2="строка2"
print (s1, s2)
#будет выведено
строка1  строка2

s1="строка1\n"
s2="строка2"
print (s1, s2)

#будет напечатано
строка1
  строка2
```

Чтобы специальный символ выводился как есть, его необходимо экранировать с помощью символа слэш:

```
#экранируем символ \n
s1="строка1\\n"
s2="строка2"
print (s1, s2)

#будет напечатано
строка1\n  строка2
```

Кавычку или апостроф внутри строки также необходимо экранировать. Но объект, представляющий из себя несколько строк, начинающихся с новой строки, нельзя заключить в кавычки или апострофы. Это вызовет ошибку. Например (рис.60):

```
s1="строка1\n
строка2"
print (s1)
```

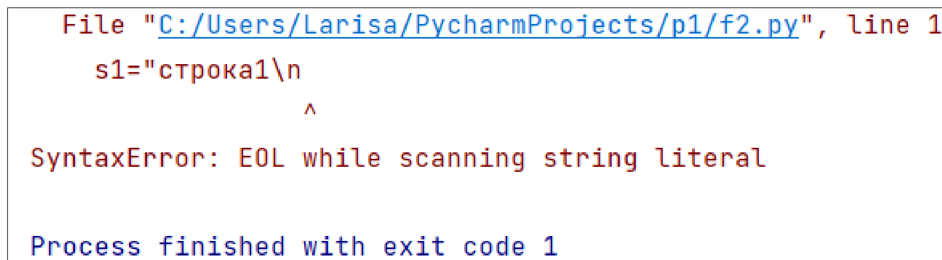


Рис.60. Результат неправильно оформленного вывода

Чтобы расположить объект на нескольких строках, следует перед символом перевода строки указать символ \, поместить две строки внутри скобок или использовать конкатенацию внутри скобок. Например (рис.61):

```
s1="строка1\
строка2"
print (s1)

#неявная конкатенация
s1=("строка1" "строка2")
print (s1)

#явная конкатенация
s1=("строка1"+"строка2")
print (s1)

#результат работы программы для всех 3-х случаев
```

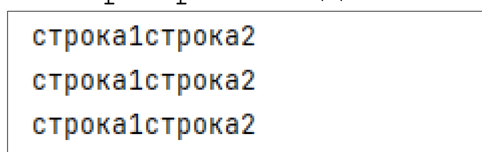


Рис.61. Вывод объекта из нескольких строк

Способ 3. Указать строку между утроенными апострофами или утроенными кавычками. Такие объекты можно разместить на нескольких строках, а также одновременно использовать апострофы и кавычки без необходимости их экранировать. Например:

```
s1="""строка1
строка2
строка3"""
print (s1)

#будет напечатано
строка1
строка2
строка3
```

Специальные символы строк

Это комбинации знаков, обозначающих служебные или непечатаемые символы, которые невозможно вставить обычным способом. Перечислим специальные символы, допустимые внутри строки, перед которой нет модификатора r:

- \n-перевод строки;
- \r-возврат каретки;
- \t-знак табуляции;
- \ v-вертикальная табуляция;
- \a- звонок;
- \b-забой;
- \f-перевод формата;
- \0-нулевой символ (не является концом строки);
- \"-кавычка;
- \ ' - апостроф;
- \N-восьмеричное значение N. Например, \74 соответствует символу <;
- \xN-шестнадцатеричное значение N. Например, \x6a соответствует символу j;
- \\-обратный слэш;
- \uxxxx- 16-битный символ Unicode. Например, \u043a соответствует русской букве к;
- \Uxxxxxxxx-32-битный символ Unicode.

Если после слэша не стоит символ, который вместе со слэшем интерпретируется как спецсимвол, то слэш сохраняется в составе строки. Например:

```
print ("Этот символ \не специальный")
```

Тем не менее, лучше экранировать слэш явным образом:

```
print ("Этот символ \\не специальный")
```

Операции над строками

Как вы уже знаете, строки относятся к последовательностям. Как и все последовательности, строки поддерживают обращение к элементу по индексу, получение среза, конкатенацию, повторение и проверку на вхождение.

Рассмотрим эти операции подробно.

К любому символу строки можно обратиться как к элементу списка. Достаточно указать его индекс в квадратных скобках. Нумерация начинается с нуля:

```
s = 'Python'
print(s[0],s[1], s[2], s[3], s[4], s[5])
```

Если символ, соответствующий указанному индексу, отсутствует в строке, то возбуждается исключение IndexError.

В качестве индекса можно указать отрицательное значение. В этом случае смещение будет отсчитываться от конца строки, а точнее, значение вычитается из длины строки, чтобы получить положительный индекс:

```
s = 'Python'
print(s[-1],s[len (s)-1])

#будет напечатано
 n  n
```

Так как строки относятся к неизменяемым типам данных, то изменить символ по индексу нельзя, будет сгенерирована ошибка TypeError (рис.62):

```
s = 'Python'
s[0]='f'
```

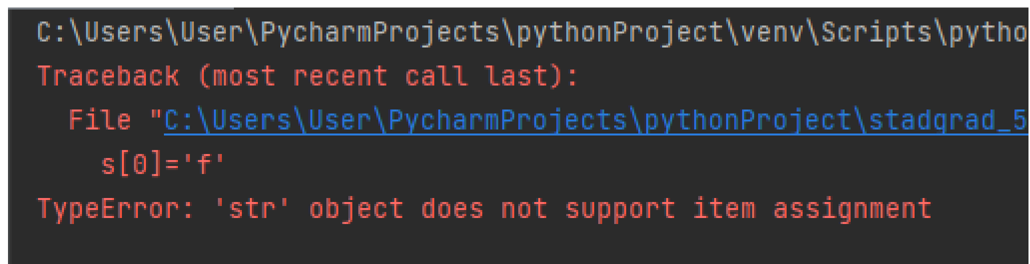


Рис.62. Результат работы программы при попытке изменить символ по его индексу

Чтобы выполнить изменение, можно воспользоваться операцией извлечения среза, которая возвращает указанный фрагмент строки. Формат операции: *[<Начало>:<Конец>:<Шаг>]*

Все параметры являются необязательными. Если параметр *<Начало>* не указан, то используется значение 0. Если параметр *<Конец>* не указан, то возвращается фрагмент до конца строки.

Следует также заметить, что символ с индексом, указанным в этом параметре, не входит в возвращаемый фрагмент.

Если параметр <*Шаг*> не указан, то используется значение 1. В качестве значения параметров можно указать отрицательные значения. Рассмотрим несколько примеров.

Сначала получим копию строки:

```
s = "Python"
```

Заменим первый символ в строке:

```
s1="J" + s[1:] #Извлекаем фрагмент от символа 1 до
конца строки
#получим
'Jython'
```

Теперь выведем символы в обратном порядке:

```
s[::-1] #Указываем отрицательное значение в пара-
метре Шаг
```

Удалим последний символ:

```
s2=s[:-1] # Возвращается фрагмент от 0до 1en (s)-1
print (s2)
#получим
'Pytho'
```

Получим первый символ в строке:

```
s[0:1]
```

Так как символ с индексом 1 не входит в диапазон, мы увидим символ с индексом 0 - 'P' .

А теперь получим последний символ:

```
s[-1:]
```

Получаем фрагмент от 1en(s)-1 до конца строки - 'n'.

И, наконец, выведем символы с индексами 2, 3 и 4:

```
s[2:5]  # Возвращаются символы с индексами 2, 3 и
4
'tho'
```

Узнать количество символов в строке позволяет функция len ().
Теперь, когда мы знаем количество символов, можно перебрать все символы с помощью цикла for:

```
s = 'Python'
for i in range(len(s)):
    print(s[i])
#результат выполнения
P
y
t
h
o
n
```

Так как строки поддерживают итерации, мы можем просто указать строку в качестве параметра цикла:

```
s = 'Python'
for i in s:
    print(i)
```

Результат выполнения будет таким же, как в примере выше.

Соединить две строки в одну строку позволяет оператор +. Кроме того, можно выполнить неявную конкатенацию строк. В этом случае две строки указываются рядом без оператора между ними:

```
s = 'Python'
s1='qt5'

s2=s+s1
print(s2)
print ('rrr' 'eee')

#будет напечатано
Pythonqt5
rrreee
```

Обратите внимание на то, что если между строками указать запятую, то мы получим кортеж, а не строку:

```
s = "Строка1", "Строка2"
print (type(s))

# Получаем кортеж, а не строку
<class 'tuple'>
```

При необходимости соединить строку с другим типом данных (например, с числом) следует произвести явное преобразование типов с помощью функции str():

```
"string" + str(10)
```

```
'string10'
```

Повторить строку указанное количество раз можно с помощью оператора *, а выполнить проверку на вхождение фрагмента в строку позволяет оператор in:

```
s="-" *  20
print(s)
print ("yt" in "Python")

#будет выведено
--------------------
True
```

7.2.7.1. Алгоритмы работы со строками

На сегодняшний день разработано большое количество алгоритмов работы со строками. В этом параграфе мы рассмотрим алгоритмы сортировки строк – LSD и MSD-сортировки, как наиболее широко используемые, на наш взгляд. Оба алгоритма реализуют так называемый **цифровой (поразрядный) подход**, использующий внутреннюю структуру сортируемых объектов.

Суть подхода. Пусть имеется множество последовательностей одинаковой длины, состоящих из элементов на которых задано отношение линейного порядка. Требуется отсортировать эти последовательности в лексикографическом порядке.

По аналогии с разрядами чисел элементы, из которых состоят сортируемые объекты, называются *разрядами*. Объекты сортируются последовательно с помощью какой-либо устойчивой сортировки в порядке от младшего разряда к старшему разряду.

Примерами объектов, которые удобно разбивать на разряды и сортировать по ним, являются числа и строки. Почему?

Для чисел понятие разряда определено, поэтому их можно представить как последовательности разрядов. Так как в различных системах счисления представления одного и того же разряда отличаются друг от друга, перед сортировкой лучше представить числа в удобной для себя системе счисления.

Строки представляют из себя последовательность символов. В качестве разрядов в данном случае выступают отдельные символы, сравнение которых обычно происходит по соответствующим им ко-

дам из таблицы кодировок. Для такого разбиения самый младший разряд — последний символ строки.

LSD сортировка строк

В качестве разряда используется значение символа. Оптимальная производительность достигается на сортировках строк одинаковой длины: автомобильные или телефонные номера, идентификаторы товаров и т. п.. Чем меньше мощность используемого алфавита, тем оптимальней алгоритм.

Так как алгоритм линеен, сложность по времени в наихудшем случае – O (n), где n - количество сортируемых объектов.

Требует (n +k) дополнительной памяти, k – диапазон возможных значений объектов.

Описание алгоритма:

1) Определяем длину строк k в сортируемой последовательности.
2) Последовательно k раз выполняем сортировку объектов, используя алгоритм распределяющего подсчета, где в качестве ключа сортировки используется значение i-го символа. Отсчет ведется от конца строки.

Рисунок 63 графически иллюстрирует описанный выше алгоритм.

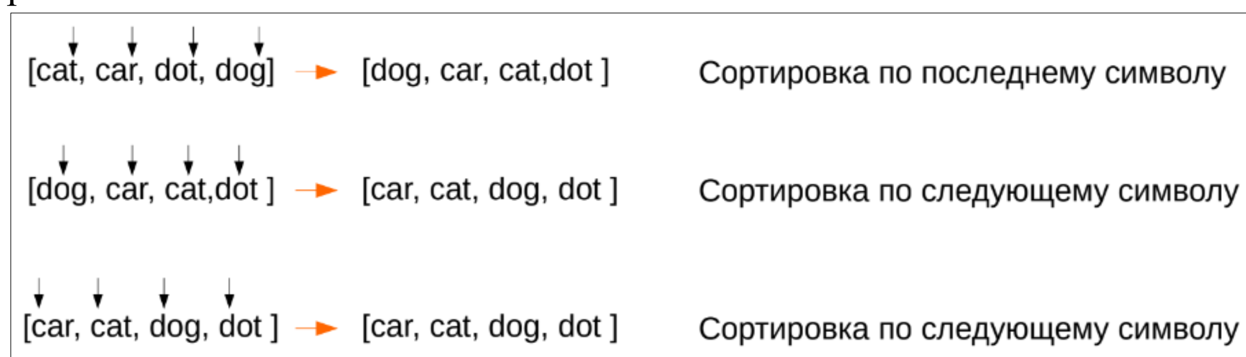


Рис.63. Графическая иллюстрация алгоритма LSD-сортировки

Алгоритм распределяющего подсчета

Предположим, что надо отсортировать числа, заданные в массиве с именем "Исходный_массив"; количество сортируемых чисел задается скаляром "Кол-во_чисел"; сортируемые числа J удовлетворяют условию: $0 \leq J \leq Max_число$

Для сортировки потребуются описания:

```
int  Max_число;      /* Верхняя граница значений */
int  *Повтор;        /* Длина этого массива = Max_число */
```

int Кол_чисел;   / Кол-во сортируемых чисел */*

*int *Исходный_массив; /* Длина этого массива >= Кол_чисел */*

*int *Результат;   /* Длина этого массива >= Кол_чисел */*

int ii,jj, kk;   / Рабочие переменные */*

1.Обнуляется служебный массив для подсчета числа повторений исходных кодов:

for (ii=0; ii<Max_число; ++ii) Повтор[ii]= 0;

2.Сортируемый массив просматривается и вычисляется количество повторений каждого числа:

for (ii= 0; ii < Кол_чисел; ++ii) { jj= Исходный_массив[ii]; Повтор[jj]= Повтор[jj] + 1;

}

3.Суммируется количество повторений каждого числа, так что значение Повтор[J] даст начальное расположение группы чисел, равных J, в отсортированном массиве:

jj= 0;

for (ii=0; ii<Max_число; ++ii) { jj= jj + Повтор[ii]; Повтор[ii]= jj;

}

4.Просматривается исходный массив, числа из него заносятся в массив результатов той же длины. Индекс занесения числа J в массив результатов равен значению J-го элемента массива Повтор. После занесения числа J значение Повтор[J] уменьшается на 1:

for (ii= 0; ii < Кол_чисел; ++ii) { jj= Исходный_массив[ii];

kk= Повтор[jj];

Результат[kk]= jj; Повтор[jj]= Повтор[jj] - 1;}

Пример реализации LSD сортировки на Python:

```
#функция для LSD-сортировки
def counting_sort(sequince, symbol_position):
    n = 256
    support = [0 for i in range(n)]
    for element in sequince:
        support[ord(element[symbol_position])] +=
1
    size = len(sequince)
    for i in range(n-1, -1, -1):
        size -= support[i]
```

```
        support[i] = size
    result = [None for i in range(len(sequince))]
    for element in sequince:
        result[support[ord(element
[symbol_position])]]= element
        support[ord(element[symbol_position])] +=
1
    return result

def radix_sort(sequince):
    max_len = len(sequince[0])
    for i in range(max_len-1, -1, -1):
        sequince = counting_sort(sequince, i)
    return sequince

#основная программа
sequince = ["cat", "car", "dot", "dog"]
result = radix_sort(sequince)
print(result)
```

MSD сортировка

Является одним из методов сортировки, который работает на основе разбиения массива объектов на подмассивы по его разрядам и последующим сортировкой каждого подмассива по отдельности. Сортировка начинается с начала строк (самой значащей цифры).

Описание алгоритма. Сначала исходный массив объектов делится на k частей, где k — основание, выбранное для представления сортируемых объектов. Эти части принято называть "корзинами". В первую корзину попадают элементы, у которых старший разряд с номером d=0 имеет значение 0. Во вторую корзину попадают элементы, у которых старший разряд с номером d=0 имеет значение 1 и так далее. Затем элементы, попавшие в разные корзины, подвергаются рекурсивному разделению по следующему разряду с номером d=1. Рекурсивный процесс разделения продолжается, пока не будут перебраны все разряды сортируемых объектов, и пока размер корзины больше единицы. То есть останавливаемся, когда $d>m$ или $l \geq r$, где m — максимальное число разрядов в сортируемых объектах, l, r — левая и правая границы отрезка массива объектов A.

В основу распределения элементов по корзинам положен метод распределяющего подсчета элементов с одинаковыми значениями в сортируемом разряде. Для этого выполняется просмотр массива и подсчет количества элементов с различными значениями в сортируемом разряде. Эти счетчики фиксируются во вспомогательном массиве счетчиков cnt. Затем счетчики используются для вычисления размеров корзин и определения границ разделения массива. В соответствии с этими границами сортируемые объекты переносятся во вспомогательный массив c, в котором размещены корзины. После того как корзины сформированы, содержимое вспомогательного массива c переносится обратно в исходный массив A и выполняется рекурсивное разделение новых частей по следующему разряду в пределах границ корзин, полученных на предыдущем шаге.

На рисунке 64 представлена графическая иллюстрация работы описанного алгоритма.

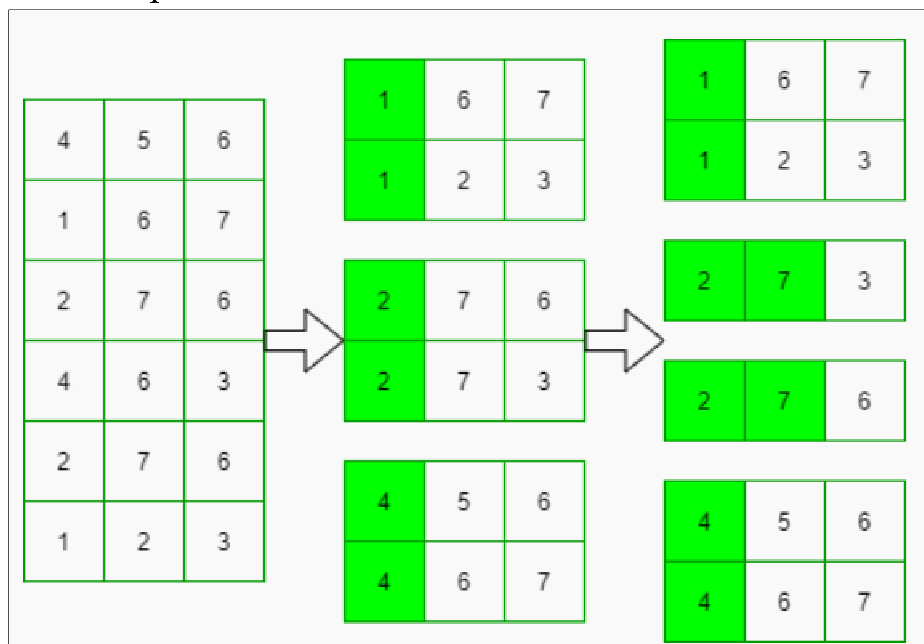


Рис.64. Графическая иллюстрация MSD сортировки

Временная сложность MSD в лучшем случае равна O(n), в худшем случае — O(n·M), где M - средняя длина строк.

Требует (n + M·B) дополнительной памяти, где M - длина самой длинной строки и B - мощность системы счисления: для десятичной системы счисления B = 10, для символов – B = 256.

Пример реализации алгоритма на Python:

```
    import numpy as np
class Radix_Sort():
```

```
    def __init__(self):
        # тестовый набор для проверки MSD
        self.A=['aaa','w','ssss','f','dddd']
        print(self.Radix_Sort_MSD(self.A))

    def count_sort_MSD(self, A, exp):
        c = np.zeros(10, dtype=int)
        B = np.zeros(len(A), dtype=int)
        for i in A:
            c[int(i / exp) % 10] += 1
        d = c.copy()

        for i in range(1, len(c)):
            c[i] += c[i - 1]
        for i in A[::-1]:
            B[c[int(i / exp) % 10] - 1] = i
            c[int(i / exp) % 10] -= 1
        return B, d

def Radix_Sort_MSD(self, A, exp=None):
    max_ = np.max(A)
    if exp == None:
        exp = 1
        while int(max_ / exp) > 0:
            exp *= 10
        exp /= 10

    A, d = self.count_sort_MSD(A, exp)

    if exp == 1:
        return A

#основная программа
    l = []
    k = 0
    for i in d:
        if i > 1:
            l.extend(self.Radix_Sort_MSD(A[k:k +
i], exp / 10))
        elif i == 1:
```

```
            l.append(A[k])
        k += i

    return l
```

где count_sort_msd — это счетчик, отсортированный по условиям базовой сортировки MSD, отличие от LSD в том, что массив C копируется, чтобы определить, сколько чисел в каждом сегменте C [i].

Алгоритмы поиска образца в строке

7.2.7.2. Регулярные выражения

Регулярные выражения предназначены для выполнения в строке сложного поиска или замены.

Он широко используется в естественной обработке языка, веб-приложениях, требующих проверки ввода текста (например, адреса электронной почты) и почти во всех проектах в области анализа данных, которые включают в себя интеллектуальную обработку текста.

В языке Python использовать регулярные выражения позволяет модуль re. Прежде чем задействовать функции из этого модуля, необходимо подключить модуль с помощью инструкции: *import re***.**

Основные методы модуль re. Синтаксис регулярных выражений

В записи регулярных выражений используются метасимволы. Основные представлены в таблице 7.

Таблица 7. Метасимволы

Символ	**Описание**
.	Один любой символ, кроме новой строки \n.
?	0 или 1 вхождение шаблона слева
+	1 и более вхождений шаблона слева
*	0 и более вхождений шаблона слева
\w	Любая цифра или буква

\d	Любая цифра [0-9] (\D — все, кроме цифры)
\s	Любой пробельный символ
\b	Граница слова
[..]	Один из символов в скобках ([^..] — любой символ, кроме тех, что в скобках)
\	Экранирование специальных символов (\. означает точку или \+ знак «плюс»)
^ и $	Начало и конец строки соответственно
{n, m}	От n до m вхождений ({,m} — от 0 до m)
a| b	Соответствует a или b
()	Группирует выражение и возвращает найденный текст
\t, \n, \r	Символ табуляции, новой строки и возврата каретки соответственно

Основные **методы** модуля:

- re.compile () - создает откомпилированный шаблон регулярного выражения;
- re.search () - метод ищет по всей строке, но возвращает только первое найденное совпадение;
- re.match() - находит совпадение, если оно происходит в начале строки;
  re.findall() - возвращает все неперекрывающиеся совпадения шаблона в строке в виде списка строк. Строка сканируется слева направо, совпадения возвращаются в найденном порядке.
- re.split() - сканирует всю строку и разделяет ее в случае нахождения разделителя.
- re.sub() – возвращает измененную строку.

Функция compile () имеет следующий *формат: = re.compile ([шаблон, флаг]).*

Основным примером является \s+. Здесь \s соответствует любому символу пробела. Добавив в конце оператор +, шаблон будет иметь не менее 1 пробела. Этот шаблон будет соответствовать даже символам tab \t, метод search () ищет заданный шаблон:

```
import re

st="АБВГДЕЕ"
#задаем шаблон поиска пробелов
p=re.compile('\s+')

print("Найдено" if p.search(st) else "Нет")
```

Посмотрим, как они работают. Основная связка метасимволов, с которой вы будете сталкиваться, это **квадратные скобки**: [и]. Они используются для создания «**класса символов**», которые вы можете сопоставить. Перечислить символы, например, так: [xyz]. Будет сопоставлен любой внесенный в скобки символ. Например:

```
import re

s="zrrr"
#задаем шаблон поиска любых сочетаний x, xy,xz,
xyz,yz,yx,y,z
p = re.compile(r"^[xyz]")

print("Найдено" if p.search(s) else "Нет")
```

Модификатор r означает, что строка не форматирована. Если этот модификатор не указывать, то все слэши необходимо экранировать.

Также можно использовать тире для **выражения ряда символов**, соответственно:

```
import re

s="абвгдее"
#поиск букв русского алфавита
p = re.compile(r"^[а-яё]")

print("Найдено" if p.search(s) else "Нет")
```

Фактически для **выполнения поиска** нам нужно добавить начальный и конечный искомые символы.

Чтобы упростить это, мы можем использовать звездочку. **Символ ‘*’**, данный символ указывает регулярному выражению, что предыдущий символ может быть сопоставлен 0 или более раз. Например:

```
import re

s="абвгдее"
#поиск букв английского алфавита до буквы f
p = re.compile('[b-f]*f')

print("Найдено" if p.search(s) else "Нет")
```

Чтобы увидеть результат поиска, используем метод group ():

```
import re

s="abcdfдee"
p = re.compile('[b-f]*f')
pp=p.search(s)
print(pp.group())

#будет напечатано
bcdf
```

Шаблон работает до первого вхождения буквы f, т.е. для строки s="abcdfдeebcf" результат будет bcdf.

Символ '.' Задает один произвольный символ, кроме новой строки \n.

```
import re

s="abcdfдeebbf"
p = re.compile('.д')
pp=p.search(s)
print(pp.group())

#будет напечатано
fд
```

Если необходимо найти конкретный символ (или подстроку), надо указать его:

```
import re

s="abcdfффдeebbf"
p = re.compile('ф.д')
pp=p.search(s)
print("Найдено" if p.search(s) else "Нет")
print(pp.group())

#будет напечатано
ффд
```

Метасимвол «^» - поиск шаблона в начале строки (привязка к первому символу строки).

Примеры:

```
import re
```

```
s="abcdfфффдееbфффдbf"
p = re.compile(r"^[0-9]+")
print("Найдено" if p.search(s) else "Нет")

#Будет напечатано
нет

import re

s="1abcdfфффдееbфффдbf"
p = re.compile(r"^[0-9]+")
print("Найдено" if p.search(s) else "Нет")

#будет напечатано
да

import re

s="ab1cdfфффдееbфффдbf"
p = re.compile(r"^[0-9]+")
print("Найдено" if p.search(s) else "Нет")

#Будет напечатано
нет
```

Аналогично для символа конца строки $

Метасимвол «{n,m}» поиск от n до m вхождений шаблона.

```
import re

s="11bcd1fфффдееbфффдbf"
p = re.compile(r"^[0-9]{1,2}")
print("найдено" if p.search(s) else "нет")

#будет напечатано
да

import re

s="222222bcd1fфффдееbфффдbf"
```

```
p = re.compile(r"^[0-1]{1,2}")
print("найдено" if p.search(s) else "нет")

#будет напечатано
нет
```

Поиск осуществляется без привязки к началу строки.

Флаги

В параметре могут быть указаны следующие *флаги* (или их комбинация через оператор |):

– I или IGNORECASE — поиск без учета регистра:

```
import re
s="АБВГДЕЕ"
#поиск букв русского алфавита, кодировка UTF-8
(re.U)
p = re.compile(r"^[а-яё]", re.I | re.U)
print("Найдено" if p.search(s) else "Нет")
#будет напечатано
Найдено
```

– M или MULTILINE — поиск в строке, состоящей из нескольких подстрок, разделенных символом новой строки ("\n").

Найдем, существуют ли подстрока, начинающаяся на символ с (\n). В строке можно использовать комментарий:

```
import re
s="222222bcd1fфффдееbфффдbf\ncrtyu"
p = re.compile('''комметарий''' r"\n", re.M)

print("Найдено" if p.search(s) else "Нет")
#будет напечатано
Найдено
```

– L или LOCALE — учитываются настройки текущей локали. Флаг re.LOCALE делает метасимволы \w, \W, \b и \B зависимыми от текущей локали, а не от базы данных Unicode. Локали - это функция библиотеки языка C, предназначенная для написания программ, учитывающих языковые различия. Например, если обрабатывается текст на русском языке, то необходимо иметь возможность чтобы \w+ соответствовало словам, но \w соответствует только классу символов [A-Za-z] и не будет соответствовать "й" или "ё". Если система настроена правильно и выбран русский язык, некоторые функции языка C сообщат программе,

что "й" также следует считать буквой. Установка флага re.LOCALE при компиляции регулярного выражения приведет к тому, что результирующий скомпилированный объект будет использовать эти функции C для \w. Компиляция будет происходить медленнее, но также позволяет \w+ сопоставлять русские слова, как и ожидается.

ВАЖНО: *Как видно из примеров, перед всеми строками, содержащими регулярные выражения, указан модификатор r. Иными словами, мы используем неформатированные строки. Если модификатор не указать, то все слэши необходимо экранировать. Например, строку: p = re.compile(r"^\w+$") нужно было бы записать так: p = re.compile("^\\w+$") Внутри регулярного выражения символы ., ^, $, *, +, ?, {, [,], \, |, (и) имеют специальное значение. Если эти символы должны трактоваться как есть, их следует экранировать с помощью слэша. Некоторые специальные символы теряют свое особое значение, если их разместить внутри квадратных скобок, — в этом случае экранировать их не нужно. Например, как уже было отмечено ранее, метасимвол «точка» по умолчанию соответствует любому символу, кроме символа перевода строки. Если необходимо найти именно точку, то перед точкой нужно указать символ \ или разместить точку внутри квадратных скобок: [.].*

Продемонстрируем это на примере проверки правильности введенной даты:

```
import re

# Вместо точки указана запятая
d = "29,12.2009"
#экранируем символ «.»,чтобы он воспринимался как
есть, а не как любой символ
p = re.compile(r"^[0-3][0-9]\.[01][0-
9].[12][09][0-9][0-9]$")
# Символ "\" не указан перед точкой
if p.search(d):
    print("Дата введена правильно")
else: print("Дата введена неправильно")

#будет напечатано
Дата введена неправильно
```

Пример поиска дат в строке (рис.65):

```
import re
```

```
string = '34-3456 / 12-05-2007 / 65-58-3015 / 56-
4532 / 11-11-2011 / 67-8945 / 12-01-2009'
#шаблон два числа для дня месяца и номера месяца
#4 числа – под месяц
pattern = r'\d{2}-\d{2}-\d{4}'
result = re.findall(pattern, string)
print(result)

#будет напечатано (рис.65)
```

```
C:\Users\User\PycharmProjects\pr6\venv\Scripts\python.exe
['12-05-2007', '65-58-3015', '11-11-2011', '12-01-2009']

Process finished with exit code 0
```

Рис.65. Результат поиска дат в строке

Пример повторного ввода даты в случае ее некорректного задания:

```
import re

# Вместо точки указана запятая
d = "29,12.2009"

#экранируем символ .,чтобы он воспринимался как
есть, а не как любой символ
p = re.compile(r"^[0-3][0-9]\.[01][0-
9].[12][09][0-9][0-9]$")

# Символ "\" не указан перед точкой
if p.search(d):
    print("Дата введена правильно")
else:
    print("Дата введена неправильно")
    print("Введите еще раз\n")
    d=input ()
```

Метод match()

Синтаксис: *re.match(pattern, string, flags=0)*. Описание параметров представлено в таблице 8.

Таблица 8. Параметры метода match ()

№	Параметр & Описание
1	pattern — строка регулярного выражения (r'g.{3}le')

2	String — строка, в которой мы будем искать соответствие с шаблоном в начале строки ('google')
3	flags — модификаторы, перечисленными в таблице ниже. Вы можете указать разные флаги с помощью побитового OR

Функция re.match возвращает объект match при успешном завершении, или None при ошибке. Можно использовать функцию group (num) или groups() объекта match для получения результатов поиска (таблица 9).

Таблица.9. Методы функции match()

№	Метод совпадения объектов и описания
1	group(num=0) — этот метод возвращает полное совпадение (или совпадение конкретной подгруппы)
2	groups() — этот метод возвращает все найденные подгруппы в tuple

Пример:

```
import re

print("Найдено" if re.match(r'TP', 'TP Tutorials
Point TP') else "Нет")

#Будет напечатано
Найдено
```

Метод split () - сканирует всю строку и разделяет ее в случае нахождения разделителя (регулярного выражения). Синтаксис: *re.split (pattern, string, maxsplit=0, flags=0)*

Параметры:

- Pattern – строка, шаблон регулярного выражения;
- String – строка для поиска;
- maxsplit=0 – число, максимальное количество делений строки;
- flags=0 – один или несколько флагов.

Возвращаемое значение – список подстрок.

Например:

```
import re

result = re.split(r'y', 'Analytics')
print (result)

#будет напечатано
```

```
['Anal', 'tics']
```

Метод Sub () – возвращает измененную строку. Синтаксис: *re.sub(pattern, repl, string, max=0)*

Метод заменяет все вхождения pattern в string на repl, если не указано на max. Возвращает измененную строку. Например:

```
import re

born = "05-03-1987 # Дата рождения"
# Удалим комментарий из строки
dob = re.sub(r'#.*$', "", born)
print("Дата рождения:", dob)

#будет напечатано
Дата рождения: 05-03-1987
```

7.2.7.3. f-строки

Начиная с версии 3.6, в Python появилась новая возможность – так называемые «f-строки». Использование f-строк напоминает применение метода format (). Отличие состоит в более демократичной форме записи. Например:

```
name = 'Chris'
food = 'creme brulee'
#обычный вывод
print ('Hello. My name is ', name,' and I like ',
food,'.')

#вывод с помощью f-строки
print (f'Hello. My name is {name} and I like
{food}.')

#будет напечатано
Hello. My name is Chris and I like creme brulee.
Hello. My name is Chris and I like creme brulee.
```

Как видно из примера, вывод с помощью f-строки менее громоздкий. При объявлении f-строк перед открывающей кавычкой пишется буква f.

7.2.7.4. Unit-тестирование строк

Тестирование строк выполняется обычными методами модуля unittest. В примере ниже, проверяется введенная строка. Если это «hello world», должно быть выведено ‘OK’, иначе ‘NO’.

#модуль с тестами

```
import unittest
import test

class CalcTest(unittest.TestCase):
    def test_1(self):
        self.assertEqual(test.test_split('hello world'), 'OK')

     def test_2 (self):
        self.assertEqual (test.test_split ('hel-lo'), 'NO')

if __name__ == '__main__':
    unittest.main()
# тестируемая программа test.py
def test_split(s):
    if s=='hello world': return 'OK'
    else: return 'NO'
#основная программа
s=input()
```

7.2.7.5. Дополнительные источники по работе со строками

1. Васильев А. Программирование на Python в примерах и задачах. –Москва: Эксмо, 2021. – 616 с.
2. Хайнеман Дж., Поллис Г., Сеяков С. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд.: Пер. с англ. – СпБ.: ООО “Альфа-книга”, 2017. – 432 с .: ил.
3. Основы работы со строками в Python. Режим доступа: https://pythonpip.ru/osnovy/osnovy-raboty-so-strokami-python
4. Функции и методы строк. Режим доступа: https://pythonworld.ru/tipy-dannyx-v-python/stroki-funkcii-i-metody-strok.html

5. Работа со строками. Готовимся к собеседованию. Режим доступа: https://tproger.ru/articles/python-strings-for-interview-part-1/
6. Приемы работы со строками. Режим доступа: https://docs-python.ru/tutorial/ispolzovanie-tekstovyh-strok-python/
7. Алгоритмы поиска образца в строке. Режим доступа: https://www.youtube.com/watch?v=S2I0covkyMc&t=394s
8. Алгоритм распределяющего подсчета. Режим доступа: https://www.youtube.com/watch?v=RKhNpG7_fhI
9. Алгоритмы LSD и MSD. Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Цифровая сортировка

Практическая работа № 7
РАБОТА СО СТРОКАМИ

Задание

Напишите программы к заданиям.

Оформите отчет по практической работе, содержащий:

- титульный лист;
- содержание;
- для каждого варианта:
  - текст задания с указанием варианта;
  - блок-схемы алгоритмов функций, используемых в решении задачи;
  - плановую обработку ошибок;
  - программный код на языке Python версии 3.8, соответствующий блок-схемам;
  - unit-тесты;
  - скриншоты результата работы программы для каждого unit-теста.

ВАРИАНТ 1. Дан список контактов в формате <фамилия, имя, номер>. Написать программу управления контактами. Для этого необходимо реализовать в виде функций поиск контакта в списке по различным ключам (фамилии, имени, номеру телефона соответственно) и вывод найденного контакта.

Большие целые числа удобно читать, когда цифры в них разделены на тройки запятыми. Найдите числа в тексте. Переформатируйте целые числа в тексте. Пример входных и выходных данных.

Ввод	Вывод
12 мало	12 мало

Лучше 123 1234 почти 12345 хорошо Стало 123456 Супер 1234567	Лучше 123 1,234 почти 12,345 хорошо Стало 123,456 Супер 1,234,567

ВАРИАНТ 2. В России применяются регистрационные знаки нескольких видов. Общего в них то, что они состоят из цифр и букв. Причём используются только 12 заглавных букв кириллицы, имеющие графические аналоги в латинском алфавите – А, В, Е, К, М, Н, О, Р, С, Т, У и Х. У частных легковых автомобилей номера – это буква, три цифры, две буквы, затем две или три цифры с кодом региона. У такси – две буквы, три цифры, затем две или три цифры с кодом региона. Напишите программу, определяющую является ли последовательность букв и цифр корректным номером указанных двух типов. На вход подаются строки, которые претендуют на то, чтобы быть номером. Определите тип номера.

Пример входных данных

Ввод	Вывод
с227НА777 КУ22777 Т22В7477 М227К19У9 С227НА777	Не соответствует заданным типам Такси Частник Частник Частник

ВАРИАНТ 3. Задайте строку, содержащую произвольный русский текст, длиной не более 200 символов. Проанализируйте строку. Выведите количество вхождений для каждого символа в этой строке. Ответ должен приводиться в грамматически правильной форме. Например, «а – 18 раз», «ф – 23 раза».

ВАРИАНТ 4. Слово — это последовательность из символов, внутри которой могут быть дефисы. На вход даётся строка. Посчитайте, сколько слов содержится в строке.

Пример входных данных.

Ввод	Вывод
Он --- серо-буро-малиновая редиска!! >>>:-> А не кот.	9

www.kot.ru	

ВАРИАНТ 5. Задайте строку, содержащую произвольный русский текст, длиной не более 40 символов. Осуществите предобработку строки. Удалите из нее все кратные рядом стоящие одинаковые символы, оставив по одному: ППППОООгоДДДДАААА →ПОГОДА.

ВАРИАНТ 6. Допустимый формат e-mail адреса регулируется стандартом RFC 5322, согласно которому e-mail состоит из уникального имени пользователя почты (Local-part), одного символа @ и доменного имени (Domain-part). Будем считать валидными только те адреса, Local-part которых состоит не более чем из 64 латинских букв, цифр и символов «'._+-», Domain-part – не более, чем из 255 латинских букв, цифр и символов «.-». Обе части адреса не могут начинаться или заканчиваться на символы «.+-» и содержать более одной точки подряд. На вход даётся текст сообщения. Необходимо найти и вывести все e-mail адреса, которые встречаются в сообщении.

Ввод	Вывод
Иван Иванович! Нужен ответ на письмо от ivanoff@ivan-chai.ru. Не забудьте поставить в копию serge'o-lupin@mail.ru - это важно.	ivanoff@ivan-chai.ru serge'o-lupin@mail.ru
foo.@ya.ru, foo@.ya.ru, fo@o.@ya.ru	NO

ВАРИАНТ 7. Задайте строку, содержащую произвольный русский текст, длиной не более 200 символов. Проанализируйте строку. Выведите символы строки, входящие в нее не более одного раза. Ответ должен приводиться в грамматически правильной форме. Например, «а б к ь».

Вариант 8. Вова подготовил одно очень важное письмо, но, к сожалению, везде в тексте указал неправильное время. Найдите все вхождения времени в текст и замените его на TBD-строку.

Примечание: время — это строка вида HH:MM:SS или HH:MM, в которой HH — число от 00 до 23, а MM и SS — число от 00 до 59. TBD – строка-заполнитель, обозначающая, что точное время события не определено и будет объявлено позже.

Пример входных и выходных данных

Ввод	Вывод
Уважаемые! Если вы к 09:00 не вернёте чемодан, то уже в 09:00:01 я за себя не отвечаю. PS. С отношением 25:50 всё нормально!	Уважаемые! Если вы к (TBD) не вернёте чемодан, то уже в (TBD) я за себя не отвечаю. PS. С отношением 25:50 всё нормально!

ВАРИАНТ 9. Написать программу, преобразующую два своих целочисленных аргумента – числитель m и знаменатель n правильной дроби (m<n<100) – в строку, представляющую собой конечную десятичную запись дроби.

Пример входных и выходных данных:

3
5
3,5

56
78
56,78

ВАРИАНТ 10. Pascal требует, чтобы действительные константы имели либо десятичную точку, либо показатель степени (начинающийся с буквы e или E), либо и то, и другое. Знак + или - может предшествовать как самому числу, так и показателю степени. Пробелы могут предшествовать или следовать за реальной константой, но они не могут быть встроены в нее. Задача – определить валидные записи вещественных констант Паскаля. Пример входных и выходных данных

Ввод	Вывод
1.2 1. 1.0e-55 e-12 6.5E 1e-12 +4.1234567890E-99999 7.6e+12.5 99	1.2 is legal. 1. is illegal. 1.0e-55 is legal. e-12 is illegal. 6.5E is illegal. 1e-12 is legal. +4.1234567890E-99999 is legal. 7.6e+12.5 is illegal. 99 is illegal.

ВАРИАНТ 11. Дана строка, содержащая текст и числа, разделенные пробелами. Выделите из строки числа. Проанализируйте числовые данные, определите: максимальное и минимальное значения.

ВАРИАНТ 12. Владимир устроился на работу в одно очень важное место. И в первом же документе он ничего не понял, там были сплошные *ФГУП НИЦ ГИДГЕО*, *ФГОУ ЧШУ АПК* и т.п. Тогда он решил собрать все аббревиатуры, чтобы потом найти их расшифровки в Интернете. Помогите ему, найдите все аббревиатуры, содержащиеся в тексте.

Примечание: аббревиатурой будем считать слова, состоящие не менее чем из двух заглавных букв, расположенных последовательно одна за другой. Если несколько таких слов разделены пробелами, то они считаются одной аббревиатурой.

Ввод	Вывод
Этот курс информатики соответствует ФГОС и ПООП, это подтверждено ФГУ ФНЦ НИИСИ РАН	ФГОС ПООП ФГУ ФНЦ НИИСИ РАН

ВАРИАНТ 13. На вход программы подаётся текст. С помощью регулярных выражений найдите все e-mail адреса в тексте. Выведите их на экран. При этом e-mail не может быть частью слова, то есть слева и справа от e-mail'а должен быть либо конец строки, либо небуква и при этом не один из символов '._+-, допустимых в адресе. Пример входных данных.

Ввод	Вывод
Василий Васильевич! Необходимо ответить на письмо от ivanoff@ivan-chai.ru. Не забудьте поставить в Копию serge'o-lupin@mail.ru – это важно!	ivanoff@ivan-chai.ru serge'o-lupin@mail.ru
foo.@ya.ru, foo@.ya.ru, foo@foo@foo +foo@ya.ru, foo@ya-ru foo@ya_ru, -foo@ya.ru-, foo@ya.ru+ foo..foo@ya.ru (boo1@ya.ru), boo2@ya.ru!, boo3@ya.ru	boo1@ya.ru boo2@ya.ru! boo3@ya.ru

ВАРИАНТ 14. Вове потребовалось срочно запутать финансовую документацию. Но так, чтобы это было обратимо. Он не приду-

мал ничего лучше, чем заменить каждое целое число (последовательность цифр) на его куб. Найдите эти числа в тексте. Преобразуйте согласно требованию. Пример входных данных.

Ввод	Вывод
Было закуплено 12 единиц техники по 410.37 рублей за единицу.	Было закуплено 1728 единиц техники по 68921000.50653 рублей за единицу.

ВАРИАНТ 15. Если вы когда-нибудь пытались собирать номера мобильных телефонов, то наверняка знаете, что почти люди используют различные способы записи номера телефона: кто-то начинает с +7, кто-то просто с 7 или 8, а некоторые вообще не пишут префикс. Трёхзначный код кто-то отделяет пробелами, кто-то при помощи дефиса, кто-то скобками. Это очень неудобно! На вход даётся номер телефона, как его мог бы ввести человек. С помощью регулярного выражения переформатируйте его в формат +7 123 456-78-90. Если с номером что-то не так (не хватает цифр или записан недопустимый символ), нужно вывести строчку «Fail!».

Пример входных и выходных данных.

Ввод	Вывод
+7 123 456-78-90	+7 123 456-78-90
8 (123) 456-78-90	+7 123 456-78-90
7 (422) 4567890	+7 422 456-78-90
1234567890	+7 123 456-78-90
123456789	Fail !
+9 123 456-76-95	Fail !

ВАРИАНТ 16. На вход подаётся текст, состоящий из строк. С помощью регулярного выражения выделите первую букву каждого слова строки. Выведите получившуюся аббревиатуру. При выводе все буквы должны быть приведены к верхнему регистру. Пример входных данных.

Ввод	Вывод
Московский государственный институт международных отношений	МГИМО
Мама мыла раму	ММР

ВАРИАНТ 17. Дан список хештегов, некоторые из которых могут повторяться. Написать программу, удаляющую повторения хештегов. Результатом должна быть строка хештэгов, разделенных пробелом и запятой и приведенных к нижнему регистру. Список может быть пустым. В этом случае должна вернуться пустая строка.

Примечание: хештегом называется непустая строка, состоящая из одного слова.

ВАРИАНТ 18. Оригинальное японское хайку состоит из 17 слогов, составляющих один столбец иероглифов. Особыми разделительными словами – кирэдзи – текст хайку делится на части из 5, 7 и снова 5 слогов. При переводе хайку на западные языки вместо разделительного слова использую разрыв строки и, таким образом, хайку записываются как трёхстишия. На вход программы подаются трёхстишия, которые претендуют на то, чтобы быть хайку. В качестве разделителя строк используется символ «/». С помощью регулярного выражения проверьте: если разделители делят текст на строки, в которых 5/7/5 слогов, то выведите *«Хайку!»*; если число строк не равно 3, то выведите строку «*Не хайку. Должно быть 3 строки»*, иначе выведите строку вида *«Не хайку. В i строке слогов не s, а j.»*, где строка i — самая ранняя, в которой количество слогов неправильное. Для простоты будем считать, что слогов ровно столько же, сколько гласных.

Ввод	Вывод
Вечер за окном. / Еще один день прожит. / Жизнь скоротечна...	Хайку!
Просто текст	Не хайку. Должно быть 3 строки.
Как вишня расцвела! / Она с коня согнала / И князя-гордеца.	Не хайку. В 1 строке слогов не 5, а 6.
На голой ветке / Ворон сидит одиноко… / Осенний вечер!	Не хайку. В 2 строке слогов не 7, а 8.
Тихо, тихо ползи, / Улитка, по склону Фудзи, / Вверх, до самых высот!	Не хайку. В 1 строке слогов не 5, а 6.
Жизнь скоротечна… / Думает ли об этом / Маленький мальчик.	Хайку!

ВАРИАНТ 19. Дан список из N стран. На вход программе подаются сведения о марках машин, изготовляемых в данной стране и импортируемых за рубеж. Каждая строка имеет формат: <Марка машины> <Название страны, куда импортируется данная марка>. Проанализируйте данные. Определите для каждой из марок, какие из них были: доставлены во все N стран, доставлены в некоторые из N стран, не доставлены ни в одну из стран.

ВАРИАНТ 20. Владимир написал свой открытый проект, именуя переменные в стиле «ИтоговаяСумма». Позднее он узнал, что в Python для имён переменных принято использовать подчёркивания

для разделения слов (under_score). С помощью регулярного выражения, найдите в строках неправильно записанные имена переменных. Исправьте ошибки. Примечание: именем переменной будем считать любую последовательность букв и цифр, внутри которой есть строчные буквы.

Пример входных и выходных данных

Ввод	Вывод
MyVar17 = OtherVar + YetAnother2Var TheAnswerToLifeTheUniverse AndEverything = 42	my_var17 = other_var + yet_another2_var the_answer_to_life_the_u niverse_and_every

ВАРИАНТ 21. Повтор слова является достаточно распространенной ошибкой. На вход программы подается текст. Необходимо исправить каждый такой повтор. Учесть, что повторяемое слово может находиться от исходного на расстоянии до 3-ех пробельных символов.

Пример входных и выходных данных.

Ввод	Вывод
Довольно распространённая ошибка ошибка — это лишний повтор повтор слова слова. Смешно, не не правда ли? Не нужно портить хор хоровод.	Довольно распространённая ошибка — это лишний повтор слова. Смешно, не правда ли? Не нужно портить хор хоровод.

ВАРИАНТ 22. Владимир написал свой реферат в текстовом редакторе WORD, по ошибке сохранив его в «старом» формате «Документ word 97-2003». Позднее, открыв его в более новом формате, он увидел, что в документе все имена и фамилии написаны слитно строчными буквами. С помощью регулярного выражения, найдите в строках неправильно записанные имена и фамилии. Исправьте ошибки.

Вопросы для самоконтроля к параграфу 7.2.7

1. Какие строковые типы поддерживает язык Python 3.*? дайте краткую характеристику каждому типу.
2. Какими способами можно создать строку. Продемонстрируйте на примерах.
3. Какими способами можно форматировать строки? Покажите на примерах.

4. Как бы вы реализовали изменение строки на языке Python? Покажите на примерах.
5. Имеется строка S со значением "s,pa,m". Укажите все способы извлечения двух символов в середине строки. Продемонстрируйте на примерах.
6. Сколько символов в строке "a\nb\x1f\000d"?
7. Перечислите основные функции для работы со строками. Приведите примеры.
8. Когда необходимы функции преобразования систем счисления? Приведите примеры.
9. Как можно вычислить значение символьной строки? Приведите примеры.
10. Как можно выделить числовые данные из строки? Приведите примеры.
11. Какой метод разбивает строку по шаблону и возвращает список подстрок? Приведите пример.
12. Какой метод ищет все совпадения с шаблоном и заменяет их указанным значением? Приведите пример.
13. В чем отличие метода finditer () от метода findall ()? Приведите пример.
14. Какие методы проверяет соответствие с любой частью строки. Приведите пример.
15. Какие методы ищут в строке первое совпадение с шаблоном? Приведите пример.
16. Как проверить, что каждое слово в строке начинается с заглавной буквы? Приведите пример.
17. Как проверить строку на вхождение в нее другой строки? Приведите пример.
18. Как найти индекс первого вхождения подстроки в строку? Приведите пример.
19. Как вычислить длину строки? Приведите пример.
20. Как подсчитать, сколько раз символ встречается в строке? Приведите пример.
21. Как сделать первый символ строки прописным символом? Приведите пример.
22. Что такое f-строки и как ими пользоваться? Чем они отличаются от других методов форматирования? Приведите пример.

23. Как найти подстроку в заданной части строки? Приведите пример.
24. Как вставить содержимое переменной в строку, воспользовавшись методом format()?Приведите пример.
25. Как узнать о том, что в строке содержатся только цифры? Приведите пример.
26. Как разделить строку по заданному символу? Приведите пример.
27. Как проверить строку на то, что она составлена только из строчных букв? Приведите пример.
28. Как проверить то, что строка начинается со строчной буквы? Приведите пример.
29. Можно ли в Python прибавить целое число к строке? Приведите пример.
30. Как «перевернуть» строку? Приведите пример.
31. В какой ситуации вы бы воспользовались методом splitlines ()? А в какой нет? Приведите пример.
32. Что общего и чем отличаются LSD и MSD сортировки строк? продемонстрируйте на примере.

Тест самоконтроля к параграфу 7.2.7

1. Имеется строка s = "Python - простой язык". Измените в ней слово Python на Питон. Используйте модуль re. Выберите правильный ответ (ы). а) re.replace ("Питон", "Python", s) б) re.sub ("Python", "Питон", s) в) re.sub ("Python", "Питон", s) г) re.replace ("Python", "Питон", s)
2. Какое регулярное выражение(я) служит для нахождения слов, которые состоят из 5 букв и оканчиваются на латинскую букву «d»? а) “[.]+d” б) “[A-Za-z]{4}d” в) ‘.{4}d’ г) “….d”.
3. Имеется строка s=”Ух, ты!!:))\-_-.”. Командой(ами), удаляющей все знаки пунктуации, является: а) re.sub (‘[.-,’, ‘’, s) б) re.sub (r’[^\w\s],’’, s) в) re.sub (‘[,!_:)-.]’,’’,s) г) re.sub (‘[,!_:)-.]’,’’,s)

   Используйте модуль re. Выберите правильный ответ (ы).
4. Имеется строка s= «Шла Саша по шассе». Командой (ами), заменяющей все буквы ш и с на букву х, является: а) re.sub (‘(ш) |

(с)’, ‘x’, s) б) re.sub (‘[шс]’, ‘x’, s) в) re.sub (‘[шШсС]’, ‘x’, s) г) re.sub (‘(ш | с)’, ‘x’, s)

5. Имеется строка s= «Боб, я, Рос, олег, Я». Командой (ами), находящей все слова, которые содержат не менее 2 букв и начинаются с большой буквы, являются: а) re.findall (‘(А-Я)(а-я)+’, s) б) re.findall (‘[А-Я][а-я]*’, s) в) re.findall (‘(А-Я)(а-я)*’, s) г) re.findall (‘[А-Я][а-я*’, s)
6. Какое из указанных ниже регулярных выражений служит для поиска дат, заданных в формате dd-YYYY? а) \D\D-\D\D\D\D б) \D\D-\d\d\d\d в) \d\d-\d\d\d\d г) \D\D-\Y\Y\Y\Y д) \d\d-\D\D\D\D е) \d\d-\y\y\y\y.
7. Какое из указанных ниже регулярных выражений служит для нахождения всех слов, исключая все числа и знаки пунктуации: а) [^\d]+ б) [\D]+ в) [A-Za-z]+ г) [^0-9]+
8. Что будет выведено в результате выполнения следующего программного кода:

```
string=”Hello”+”,”
string=string + ”World!”
print (string)
```

а) string б) Hello, World! в) Hello,World! г) World !

9. Функция len() при передаче в неё строки возвращает: а) количество символов в строке; б) количество слов в строке; в) функция не предназначена для работы со строками; г) размер строки в байтах.
10. Что будет выведено в результате выполнения следующего программного кода:

```
str1="yellow "
 str2="baloon"
 str3=str1*3+str2
 print (str3)
```

а) yellowyellowyellowbaloon б) yellow yellow yellow balloon в) синтаксическая ошибка г) 3456ballon

11. Следующий программный код выполняет…

```
str="yellow spring red"
str2=str.split()
print (str2)
```

а) разделяет предложение на символы; б) удаляет пробелы; в) делает все строчные символы прописными; г) разделяет предложение на слова.

12. Следующий программный код выведет:

```
str="key"*3
str=str[2]+str[4]
str+=str[6]
 print (str)
```

а) key; б) eee; в) yek; г) keykeykey; д) ошибка.

13. Следующий программный код выведет:

```
str="1"+"2"
print (str)
```

а) 1+2; б) 12; в) 3; г) ошибка.

14. Следующий программный код выведет:

```
str="spam"
print (str[:-1:])
```

а) spa; б) aps; в) m; г) ошибка.

15. Следующий программный код выведет:

```
lst=list ("spam")
st=””
for x in lst:
        st+=x
print (st)
```

а) spa; б) spam; в) [‘s’, ‘p’, ‘a’, ‘m’]; г) ошибка.

16. Следующий программный код выведет:

```
str="spam "
print (len[str])
```

а) 4; б) spam; в) 5; г) 3.

17. Следующий программный код выведет:

```
str="1" – "2"
print (str)
```

а) 1 - 2; б) 12; в) -2; г) ошибка.

18. Следующий программный код выведет:

```
str=b"byte"
print (str)
```

а) b’byte’; б) 62 79 74 65; в) byte; г) ошибка.

19. Следующий программный код выведет:

```
str=spam
print (str)
```

а) spam; б) ничего; в) [‘s’, ‘p’, ‘a’, ‘m’]; г) ошибка.

20. Следующий программный код выведет:

```
st="spam"
```

print (st[-1::-1])
а) masp; б) spam; в) maps; г) ошибка.
21. Следующий программный код выведет:
st="1" * int ("2")
print (st)
а) 1212; б) 11; в) 2; г) ошибка.
22. Следующий программный код выведет:
st="spam")
for x in st:
print (x, end="")
а) spam; б) s p a m; в) [‘s’, ‘p’, ‘a’, ‘m’]; г) ошибка.
23. Какая из команд проверяет, начинается ли каждое слово в строке с заглавной буквы? 1)print('The Hilton'.istitle()) ; 2)print('The dog'.istitle()); 3)print('липкий рис'.istitle())
а) 1; б) 2; в) 3 г) ни одна из команд.
24. Какая из команд проверяет, содержит ли строка определенную подстроку:
1)print('самолет' in 'Самый быстрый самолет в мире') 2)print('автомобиль' в 'Самый быстрый самолет в мире')
3) print('самолет' в 'Самый быстрый самолет в мире')
а) 1; б) 2; в) 3 г) ни одна из команд.
25. Какая из команд подсчитывает количество определенных символов в строке:
1)print('о' in 'Первый президент организации...')
2)print('Первый президент организации...'.count('о'))
3) print('о' в 'Первый президент организации...')
а) 1; б) 2; в) 3 г) ни одна из команд.

7.2.8. Словари. Кортежи. Множества

Словари

До сих пор вы хранили данные в списках. Например, у вас есть список друзей. А список английских слов, которые кто-то из этих друзей хочет выучить, мог бы выглядеть так:

```
english_words = ['hand', 'leg', 'back-end
developer']
```

Гораздо удобнее было бы хранить переводы слов с русского на английский, чтобы забытые слова было легко подсмотреть. Для этого в Python есть структура данных **словарь** (**dict**).

Когда нужно использовать словари:

- подсчет числа каких-то объектов. В этом случае нужно завести словарь, в котором ключами являются объекты, а значениями – их количество;
- хранение каких-либо данных, связанных с объектом. Ключи – объекты, значения – связанные с ними данные. Например, если нужно по названию месяца определить его порядковый номер, то это можно сделать при помощи словаря num[‘january’] = 1; num[‘february’] = 2; …;
- установка соответствия между объектами (например, “родитель –потомок”). Ключ – объект, значение – соответствующий ему объект;
- если нужен обычный массив, при этом максимальное значение индекса элемента очень велико, но при этом будут использоваться не все возможные индексы (так называемый “разреженный массив”), то можно использовать ассоциативный массив для экономии памяти.

Синтаксис:

name_dict={
  ключ1: значение1,
  ……….
  ключ n: значение n
}

Например:

```
english = {
    'рука': 'hand',
    'нога': 'leg',
    'бэкенд-разработчик': 'back-end developer'
}
```

Как видно из примеров, словарь оформляется фигурными скобками. Его заполняют пары, записанные через запятую. Первый элемент в паре называется **ключ**, а второй – **значение**, они разделяются между собой двоеточием. Русские слова здесь ключи, а их переводы на английский – значения.

Когда запрашивают в словаре значение, соответствующее определённому ключу, это называется «доступ по ключу». Так можно получить значение для какого-нибудь ключа и заменить его:

```
english = {
```

```
        'рука': 'hand',
        'нога': 'leg',
        'бэкенд-разработчик': 'back-end developer'
    }
    # доступ по ключу: как по-английски рука?
    print (english ['рука'] )

    #будет напечатано слово hand

    english ['рука'] = 'arm'
    #значение для ключа 'рука' поменялось с 'hand'
на 'arm' - другой допустимый перевод
```

Ключи в словаре похожи на индексы списков. Только индексами выступают натуральные числа, а ключами бывают и числа обоих типов, и строки, и даже булевы значения *True* и *False*.

Например:

```
    garden = {
        'земляника': 'ягода',
        'яблоко': 'фрукт',
        'лук': ['овощ', 'оружие']# значением может
быть список
    }
    print (garden ['лук'])

    # будет напечатано ['овощ', 'оружие']

    Ключами могут быть числа:
    garden = {
    1: 'ягода',
    2.3: 'фрукт'

}
print (garden[2.3])
    # будет напечатано    фрукт
```

Ключами могут быть числа и строки:

```
    garden = {
   'земляника': 'ягода',
    2.3: 'фрукт'
}
```

Список не может быть ключом:

```
    garden = {
        'земляника': 'ягода',
        'яблоко': 'фрукт',
        ['морковь', 'кабачок']: 'овощ'  # а так
нельзя! ключом список быть не может
    }
    print (garden [['морковь', 'кабачок']])  # по-
лучится ошибка
```

При этом в одном словаре не может быть двух одинаковых ключей.

Задание значений элементов словаря с клавиатуры:

```
garden = {}
union=['овощ','оружие']
garden['земляника']=input("земляника:")
garden['яблоко']=input("яблоко:")
garden['лук']=union
print (garden)

#будет напечатано
земляника: ягода
яблоко:фрукт
{'земляника': ' ягода', 'яблоко': 'фрукт', 'лук':
['овощ', 'оружие']}
```

Способы получения всех ключей и значений

1. Метод items () возвращает список со всеми парами ключ-значение. Не принимает параметров.

```
garden = {}
union=['овощ','оружие']

garden['земляника']=input("земляника:")
garden['яблоко']=input("яблоко:")
garden['лук']=union

for i,j in garden.items():
    print (i,j)
#будет напечатано
земляника:ягода
яблоко:фрукт
земляника ягода
яблоко фрукт
```

```
лук ['овощ', 'оружие']
```

2. Чтобы получить все *ключи* словаря, нужно вызвать метод keys():

```
favorite_songs = {
    'Тополиный пух': 'Иванушки international',
    'Город золотой': 'Аквариум',
    'Звезда по имени Солнце': 'Кино'
}

print (favorite_songs.keys())

# будет напечатан список песен
(['Тополиный пух', 'Город золотой', 'Звезда по
имени Солнце'])
```

3. а если нужны все значения — метод ***values():***

```
old_letters = {'ять': 'Ѣ', 'юс малый': 'Ѧ', 'юс
большой': 'Ѫ'}

print (old_letters.values())
# будет напечатан список начертаний старинных букв
# dict_values (['Ѣ', 'Ѧ', 'Ѫ'])
```

Чтобы такие списки использовать в коде, их обычно превращают в строки методом **join():**

```
#словарь отчеств
patronymic = {'Илья': 'Ильич', 'Иван': 'Ивановна',
'Пётр': 'Петровна'}
print(", ".join (patronymic.values ()))

#выводится строка: Ильич, Ивановна, Петровна
```

Перебор элементов словаря

Пройти по всем элементам словаря можно циклом for, причём есть несколько вариантов:

1 способ. Этот способ позволяет пробежать по всем ключам словаря. Обратите внимание, что track здесь – просто название переменной, оно могло быть любым и код отработал бы так же:

```
favorite_songs = {
    'Тополиный пух': 'Иванушки international',
    'Город золотой': 'Аквариум',
    'Звезда по имени Солнце': 'Кино'
}
```

```
for track in favorite_songs:
    print (track + ' это песня группы ' + favor-
ite_songs [track])
```

2 способ. Ещё можно пройти отдельно по значениям словаря:

```
favorite_songs = {
    'Тополиный пух': 'Иванушки international',
    'Город золотой': 'Аквариум',
    'Звезда по имени Солнце': 'Кино'
}
```

```
for music_band in favorite_songs.values():
    print ('Доктор, я больше не могу слушать груп-
пу ' + music_band)
#будет напечатано (рис.)
```

```
Доктор, я больше не могу слушать группу Иванушки
international
Доктор, я больше не могу слушать группу Аквариум
Доктор, я больше не могу слушать группу Кино
```

3 способ. И по ключам и значениям одновременно:

```
for track, music_band in favorite_songs.items():
    print (track + ' это песня группы ' +
music_band)
```

Здесь мы вызвали метод items() — он похож на keys() и values(), но возвращает набор пар ключ-значение, поэтому при переборе мы используем две переменных — track и music_band. Вы можете называть их и по-другому, например song и band.

Работа с элементами словаря

Основная операция: получение значения элемента по ключу, записывается так же, как и для списков: A[key].

Если элемента с заданным ключом не существует в словаре, то возникает исключение KeyError (рис.66):

```
garden = {
    'земляника': 'ягода',
    'яблоко': 'фрукт',
    'лук': ['овощ', 'оружие']
}

#вывод существующего элемента
print (garden ['земляника'])
```

```
#вывод несуществующего элемента
print (garden ['овощ'])
#будет напечатано
```

```
ягода
Traceback (most recent call last):
  File "C:/Users/Larisa/PycharmProjects/6_1/pr_6_1.py", line 10, in <module>
    print (garden ['овощ'])
KeyError: 'овощ'
```

Рис.66. Результат работы программы для несуществующего элемента словаря

Способ с перехватыванием и обработкой исключения:

```
garden = {
    'земляника': 'ягода',
    'яблоко': 'фрукт',
    'лук': ['овощ', 'оружие'],
    'овощ': 'картофель'
}

# определение несуществующего элемента в словаре
try:
    str='овощ'
    if str in garden:
        print (garden [str])
except KeyError:
    print('такого элемента нет в словаре')
    pass
```

где оператор pass – это заглушка. Используется для того, чтобы буквально ничего не делать; это означает, что когда мы не хотим выполнять операцию, можно использовать pass для пустого кода.

Другой способ определения значения по ключу — метод get: A.get (key). Например:

```
garden = {
    'земляника': 'ягода',
    'яблоко': 'фрукт',
    'лук': ['овощ', 'оружие']
}

#определение, есть ли элемент в словаре
print (garden.get ('земляника'))

#будет напечатано
```

```
ягода
```

Если элемента с ключом get нет в словаре, то возвращается значение None:

```
    garden = {
    'земляника': 'ягода',
    'яблоко': 'фрукт',
    'лук': ['овощ', 'оружие']
}
# определение  несуществующего элемента в словаре
print (garden.get ('овощ'))

#будет напечатано
None
```

В форме записи с двумя аргументами A.get (key, val) метод возвращает значение val, если элемент с ключом key отсутствует в словаре (рис.67). Например:

```
garden = {
    'земляника': 'ягода',
    'яблоко': 'фрукт',
    'лук': ['овощ', 'оружие']
}

# определение несуществующего элемента в словаре
print (garden.get ('овощ',-1))
#будет напечатано
```

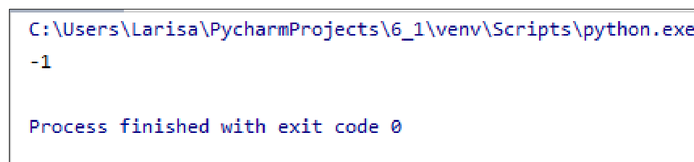


Рис.67. Результат работы метода get() с двумя аргументами

Проверить принадлежность элемента словарю можно операциями in и not in. Например:

```
garden = {
    'земляника': 'ягода',
    'яблоко': 'фрукт',
    'лук': ['овощ', 'оружие']
}
```

```
# определение существует ли элемент в словаре
if 'земляника' in garden:
    print (garden ['земляника'])
else:
    print ('такого элемента нет в словаре')
```

Проверка принадлежности с перехватыванием и обработкой исключения:

```
garden = {
    'земляника': 'ягода',
    'яблоко': 'фрукт',
    'лук': ['овощ', 'оружие'],
    'овощ': 'картофель'
}

# определение несуществующего элемента в словаре
try:
    str='овощ'
    if str in garden:
        print (garden [str])
except KeyError:
    print('такого элемента нет в словаре')
    pass
```

Для добавления нового элемента в словарь нужно просто присвоить ему какое-то значение: A[key] = value.

Для удаления элемента из словаря можно использовать операцию del A[key] (операция возбуждает исключение KeyError, если такого ключа в словаре нет). Вот два безопасных способа удаления элемента из словаря:

1) способ с предварительной проверкой наличия элемента:

```
garden = {
    'земляника': 'ягода',
    'яблоко': 'фрукт',
    'лук': ['овощ', 'оружие']
}
# удаление элемента в словаре
if 'земляника' in garden:
        del garden ['земляника']

print (garden)
```

2) способ с использованием блока try … except:

```
garden = {
    'земляника': 'ягода',
    'яблоко': 'фрукт',
    'лук': ['овощ', 'оружие']
}
# удаление  элемента в словаре
try:
    del garden ['земляника']
except KeyError:
    pass
print (garden)
```

Очищение словаря – метод clear (). Пример:

```
garden = {
    'земляника': 'ягода',
    'яблоко': 'фрукт',
    'лук': ['овощ', 'оружие']
}

# очищение словаря
garden.clear()
print (garden)
#будет напечатано (рис.68)
```

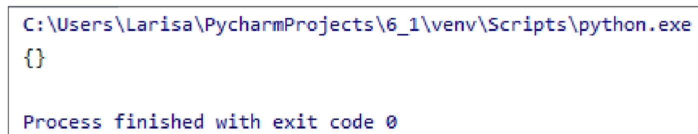


Рис.68. Результат работы метода clear()

Множества

Множества – это неупорядоченная коллекция уникальных элементов, сгруппированных под одним именем. Может быть неоднородным – включать элементы разных типов. Не относится к последовательностям!

Свойства множеств:

- множество не содержит дубликаты элементов, в отличие от списков и кортежей;
- элементы множества являются неизменными (их нельзя менять), однако само по себе множество является изменяемым, и его можно менять;

– так как элементы не индексируются, множества не поддерживают никаких операций **среза** и **индексирования**. Например:

```
s1=set()

print (s1)

#попытка изменить множество
s1={5,6,7}
print (s1[0])
```

```
#будет напечатано (рис.69)
```

```
Traceback (most recent call last):
  File "C:/Users/Larisa/PycharmProjects/p1/f2.py", line 5, in <module>
    print (s1[0])
TypeError: 'set' object is not subscriptable
set()

Process finished with exit code 1
```

Рис.69. Исключение при попытке изменить множество

Множество в Python можно **создать** несколькими способами.

Первый способ – это задать множество перечислением его элементов в фигурных скобках. Например:

```
count={1,2,3, "мир", 2.5}

print (count)
#будет напечатано
{1, 2, 3, 2.5, 'мир'}
```

Ограничение – таким образом нельзя создать пустое множество. Вместо этого будет создан пустой словарь(рис.69). Например:

```
count={}

print (type (count))
#будет напечатано
```

```
<class 'dict'>

Process finished with exit code 0
```

Рис.69. Результат создания пустого множества

Второй способ – с помощью функции set(), используя список. Например (рис.70):

```
#список
```

```
count=[1,1,1,2,2]

#создаем из списка множество
a=set(count)
print (a)

#создаем пустое множество
s1=set()
print (s1)

#заполняем пустое множество элементами
s1={5,6,7}
print (s1)
#будет напечатано
```

```
{1, 2}
set()
{5, 6, 7}

Process finished with exit code 0
```

Рис.70. Создание множества с использованием списка

Ввод значений элемента множества с клавиатуры

```
n = int(input('Количество элементов множества? '))
my_set = set()

#задаем значения элементов множества
for i in range(n):
    my_set.add(input())

print('Множество:', my_set)
#будет напечатано (рис.71)
```

```
Количество элементов множества? 5
5
6
8
9
1
Множество: {'9', '5', '8', '6', '1'}
```

Рис.71 Задание значений элементов множества с клавиатуры

Операции над множествами:

– проверка вхождения в множество;
– разница множеств;
– объединение множеств;
– пересечение множеств.

Например:

```
# Задаем множества
x={1, 2, 3, 4}
y={1,2,5,6}

# Проверка вхождения в множество
print ('e' in x)

# Разница множеств
print (x.difference(y))

# Объединение множеств
print (x | y)

# Пересечение множеств
print (x & y)
#будет напечатано (рис.72)
```

```
False
{3, 4}
{1, 2, 3, 4, 5, 6}
{1, 2}
```

Рис.72. Результат различных операций над множеством

Применение множеств

Такие операции удобны при работе с большими множествами данных – пересечение двух множеств содержит объекты, которые присутствуют в обоих множествах, а объединение содержит все объекты из обоих множеств. Ниже приводится более реальный пример применения операций над множествами – при работе со списками служащих гипотетической компании:

```
engineers =
set(['Иванов','Петров','Сидоров','Дудин'])
managers = set(['Дудин', 'Петров'])

# Кто является и инженером, и менеджером одновре-
менно?
```

```
print ('инженер и менеджер одновременно ',
engineers & managers)

# Все, кто принадлежит к той или иной категории
print ('инженер или менеджер', engineers |
managers)

# Инженеры, которые не являются менеджерами
print ('инженеры, не являющиеся менеджера-
ми',engineers. difference(managers))
#будет напечатано (рис.73)
```

```
инженер и менеджер одновременно  {'Петров', 'Дудин'}
инженер или менеджер {'Сидоров', 'Петров', 'Дудин', 'Иванов'}
инженеры, не являющиеся менеджерами {'Сидоров', 'Иванов'}
```

Рис.73. Результат практического применения различных операций над множествами

Преобразование множеств в списки и наоборот

Часто встречаемая задача – сделать список уникальным. Решить ее можно с помощью множеств. Например:

```
count=[1,1,1,2,2]

#преобразуем список в множество
a=set(count)
print ('множество a= ',a)

#преобразуем множество в список
count=list (a)
print ('список count= ',count)

#будет напечатано  [1, 2]
```

Кортежи

Объект-кортеж (tuple – произносится как «тъюпл») в общих чертах напоминает список, который невозможно изменить, – кортежи являются последовательностями, как списки, но они являются неизменяемыми, как строки.

Синтаксически, определение кортежа заключается в круглые, а не в квадратные скобки. Они также поддерживают включение объектов различных типов, вложение и операции, типичные для последовательностей:

```
#кортеж из 4 элементов
T = (1, 2, 3, 4)

#длина кортежа
l=len(T)
print ('длина ', l)

#конкатенация кортежей
C=T + (5, 6, ['eee','yyy'])
print (T)
print (C)

#извлечение элемента, среза
print (T[0])
print (T[1:3])
#будет напечатано  (рис.74)
```

```
длина  4
(1, 2, 3, 4)
(1, 2, 3, 4, 5, 6, ['eee', 'yyy'])
1
(2, 3)
```

Рис.74. Результат применения различных операций к кортежу

Единственное реальное отличие кортежей – это невозможность их изменения после создания. То есть кортежи являются неизменяемыми последовательностями. Например, при попытке изменить значение нулевого элемента кортежа T, будет выведено сообщение об ошибке (рис.75). Например:

```
T[0]=2
```

```
Traceback (most recent call last):
  File "C:/Users/Larisa/PycharmProjects/Spiski/spiski.py", line 15, in <module>
    T[0]=2
TypeError: 'tuple' object does not support item assignment
```

Рис.75. Результат работы программы при попытке изменить элемент кортежа

Для чего нужны кортежи?

Зачем нужен тип, который напоминает список, но поддерживает меньшее число операций? Откровенно говоря, на практике кортежи используются не так часто, как списки, но главное их достоинство – неизменяемость. Если коллекция объектов передается между компонентами программы в виде списка, он может быть изменен любым из компонентов. Если используются кортежи, такие изменения стано-

вятся невозможны. То есть кортежи обеспечивают своего рода сохранение целостности, что может оказаться полезным в крупных программах.

Например, семантическая сеть. Формально семантическая сеть, построенная на любом из описанных отношений, представляется в виде кортежа:

WN = <LCn,adj,v,adv, Rn,adj,v,adv, S, T, M, A>, где:

LCn,adj,v,adv = {LCn, LCadj, LCv, LCadv} – совокупность лексикализованных понятий-синсетов, сгруппированных по разным частям речи (существительные, прилагательные, глаголы и наречия соответственно);

Rn,adj,v,adv= {Rn, Radj, Rv, Radv} – наборы отношений синсетов, различающиеся для разных частей речи;

T – текстовые выражения, описанные в ресурсе;

S – отношения между текстовыми выражениями и синсетами;

M – совокупность неоднозначных текстовых выражений: $M \subset T$;

A – аксиомы транзитивности и наследования.

7.2.9. Дополнительные источники по вопросу работы со словарями, множествами и кортежами

1. Васильев А. Программирование на Python в примерах и задачах. –Москва: Эксмо, 2021. – 616 с.
2. Хайнеман Дж., Поллис Г., Сеяков С. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд.: Пер. с англ. – СпБ.: ООО “Альфа-книга”, 2017. – 432 с .: ил.
3. Создание словаря. Режим доступа: https://pythonru.com/osnovy/python-dict
4. Операции со словарями. Режим доступа: https://proglib.io/p/15-veshchey-kotorye-nuzhno-znat-o-slovaryah-python-2020-03-03
5. Словари в Python 3.8. Режим доступа: https://pythonchik.ru/osnovy/slovari-v-python
6. Методы словарей. Режим доступа: https://pythonworld.ru/tipy-dannyx-v-python/slovari-dict-funkcii-i-metody-slovarej.html
7. Словари и их методы в Python. Режим доступа: https://tproger.ru/explain/python-dictionaries/
8. Преобразование словарей в другие типы данных. Режим доступа: https://metanit.com/python/tutorial/3.3.php

9. Работа с элементами словаря. Режим доступа: https://devpractice.ru/python-lesson-9-dict/
10. Python и теория множеств. Режим доступа: https://habr.com/ru/post/516858/
11. Множества в Python. Режим доступа: https://pythonru.com/osnovy/mnozhestva-v-python
12. Кортежи в Python. Режим доступа: https://pythonchik.ru/osnovy/kortezhi-v-python

Практическая работа № 8
СЛОВАРИ. МНОЖЕСТВА. КОРТЕЖИ

Задание

Напишите программы к заданиям, указанным в индивидуальном варианте.

Оформите отчет по практической работе, содержащий:

- титульный лист;
- содержание;
- ответ на вопросы самоконтроля.
- для каждого варианта:
  - текст задания с указанием варианта;
  - блок-схему алгоритма решения задачи;
  - плановую обработку ошибок;
  - программный код на языке Python версии 3.8, соответствующий блок-схеме;
  - unit-тесты;
  - скриншоты результата работы программы для каждого unit-теста.

ВАРИАНТ 1. Напишите программу для вычисления множества чисел (в пределах первой полусотни), которые делятся или на 3, или на 4, но при этом не делятся одновременно на 3 и 4. 5.

ВАРИАНТ 2. Напишите программу для создания множества, элементами которого являются кортежи (по два элемента в каждом) с нечетными числами: (1,3), (3,5), (5,7) и так далее.

ВАРИАНТ 3. Напишите программу, которая выполняется следующим образом. Пользователь вводит целое неотрицательное число. На основе этого числа создается список из натуральных чисел от 1 до этого числа. Затем на основе этого списка создается словарь. Элементы списка служат ключами для элементов словаря. Значения элемен-

тов словаря определяются на основе значений элементов списка, но взятых в обратном порядке. Например, если пользователь вводит число 3, то создается список [1, 2, 3] и на его основе создается словарь из трех элементов. У элемента с ключом 1 значение 3, у элемента с ключом 2 значение 2, а у элемента с ключом 3 значение 1.

ВАРИАНТ 4. Напишите программу, которая выполняется следующим образом. Дан файл, содержащий текст. На основе этого текста создается словарь. Ключами словаря служат символы из текста, а значениями элементов словаря являются количества вхождений соответствующих символов в текст. Например, если в тексте "ABBCAB", то словарь будет состоять из трех элементов с ключами "A", "B" и "C", а значения элементов соответственно равны 2 (в тексте 2 буквы "A"), 3 (в тексте 3 буквы "B") и 1 (в тексте 1 буква "C").

ВАРИАНТ 5. На вход программе в виде файла подаются сведения об игрушках, некоторые из которых имеются в N детских садах. В первой строке файла сообщается количество детских садов N, которое не меньше 10, но не превосходит 100, каждая из следующих N строк имеет следующий формат: название игрушки, номер детского сада, в котором она находится. где название игрушки – строка, состоящая не более чем из 20 символов, номер детского сада – натуральное число. Создайте список игрушек. Определите игрушки из списка: а) которых нет ни в одном из детских садов; б) которые есть в каждом из детских садов.

ВАРИАНТ 6. На вход программе в виде файла подается текст из цифр и строчных латинских букв, за которыми следует точка. Определить, каких букв – гласных или согласных – больше в этом тексте.

ВАРИАНТ 7. На вход программе в виде файла подаются два числа m и n, задающие диапазон целых чисел. Из диапазона целых чисел от m до n выделить: а) множество чисел, делящихся без остатка или на k, или на l (k, l – простые); б) множество чисел, делящихся без остатка на k*l.

ВАРИАНТ 8. На вход программе в виде файла подаются строки, содержащие целые числа. Подсчитать: а) сколько раз встретилась каждая цифра десятичной записи в числе; б) количество разных цифр в десятичной записи натурального числа.

ВАРИАНТ 9. На вход программе в виде файла подается текст, все слова которого разделены пробелами. Создайте на основе этого текста два любых словаря. На основе двух словарей создайте новый словарь. В этот новый словарь включатся элементы, ключи которых

представлены в каждом из исходных словарей. Значениями элементов в создаваемом словаре являются множества из значений соответствующих элементов в исходных словарях.

ВАРИАНТ 10. На вход программе в виде файла подается текст на русском языке. Напечатать в алфавитном порядке: все согласные буквы, которые не входят ни в одно из слов; все звонкие согласные буквы, которые входят в каждое нечетное слово и не входят ни в одно четное слово.

ВАРИАНТ 11. На вход программе в виде файла подается текст. На основе текста формируется словарь. Ключами элементов словаря являются символы из текста. Значение соответствующего элемента — это исходный текст, в котором «вычеркнут» тот символ, который является ключом. Если при формировании очередного элемента словаря окажется, что такой ключ уже есть, то соответствующий символ пропускается. Например, если пользователь ввел текст "ABCABD", то в словаре будут представлены элементы с ключами "A", "B", "C" и "D" со значениями соответственно "BCABD", "ACABD", "ABABD" и "ABCAB".

ВАРИАНТ 12. На вход программе в виде файла подается текст на русском языке. Напечатать в алфавитном порядке: все гласные буквы, которые входят в каждое слово; все звонкие согласные буквы, которые входят хотя бы в одно слово.

ВАРИАНТ 13. На вход программе в виде файла подаются строки, содержащие целые положительные числа. Создайте из строк кортежи. Напишите функцию, которая сортирует кортежи по возрастанию и возвращает их. Если хотя бы один элемент кортежа не является целым числом, функция возвращает исходный кортеж.

ВАРИАНТ 14. На вход программе подается целое неотрицательное число. На основе этого числа создается список из натуральных чисел от 1 до этого числа. Затем на основе этого списка создается словарь. Элементы списка служат ключами для элементов словаря. Значения элементов словаря определяются на основе значений элементов списка с нечетными индексами.

ВАРИАНТ 15. Имеются словари с пересекающимися ключами, значениями являются положительные целые числа. Напишите две функции, выполняющие со словарями некие операции. Первая функция формирует новый словарь по следующему правилу: если в исходных словарях есть повторяющиеся ключи, выбираем среди их значений максимальное и присваиваем этому ключу. Если ключ не повто-

ряется, он переносится со своим значением в новый словарь. Вторая функция суммирует значения повторяющихся ключей и присваивает им значение полученной суммы. Если ключ не повторяется, он переносится со своим значением в новый словарь. Обе функции возвращают сформированный словарь.

ВАРИАНТ 16. На вход программе в виде файла подаются сведения о сдаче экзаменов учениками 9-ых классов. Создайте кортеж из именованных кортежей учащихся. Каждый именованный кортеж содержит следующие поля: фамилия – строка, состоящая не более чем из 20 символов, имя – строка, состоящая не более чем из 15 символов, оценки – три целых числа, соответствующие оценкам по пятибалльной системе. Требуется написать программу, которая будет выводить на экран фамилии и имена трех худших по среднему баллу учеников. Если среди остальных есть ученики, набравшие тот же средний балл, что и один из трех худших, то следует вывести и их фамилии и имена.

ВАРИАНТ 17. На вход программе в виде файла подается текст на русском языке. Напечатать в алфавитном порядке: все согласные буквы, которые входят только в одно слово; все звонкие согласные буквы, которые входят более чем в одно слово;

ВАРИАНТ 18. На вход программе в виде файла подаются строки англо-русского словаря в таком формате:

cat - кошка
dog - собака
home - домашняя папка, дом
mouse - мышь, манипулятор мышь
to do - делать, изготавливать
to make - изготавливать

Здесь английское слово (выражение) и список русских слов (выражений) разделены двумя табуляциями и минусом: \t-\t'.

Требуется создать русско-английский словарь и вывести его в файл ru-en.txt в следующем формате:

делать - to do
дом - home
домашняя папка - home
изготавливать - to do, to make
кошка - cat
манипулятор мышь - mouse
мышь - mouse
собака - dog

В выходном файле должен быть *лексикографический* порядок слов.

ВАРИАНТ 19. В одном очень дружном доме, где живет Фёдор, многие жильцы оставляют ключи от квартиры соседям по дому, например чтобы покормили животных или полили цветы. Вернувшись домой после долгих странствий, Фёдор обнаруживает, что потерял свои ключи и соседей дома нет. Но вдруг у домофона он находит чужие ключи. Помогите Федору найти ключи от своей квартиры в квартирах соседей. На ввод подается файл input.txt, в котором в первой строке записано три числа через пробел N - номер квартиры Фёдора, M - номер квартиры от которой Федор нашел ключи, K - ключ от этой квартиры. Далее i-я строка хранит описание ключей запертых в i-й квартире в формате <m_i0 - номер квартиры> <k_i0 - ключ>,<m_i1 - номер квартиры> <k_i1 - ключ>,..., причем реальные номера квартир "зашифрованы" ключем от i-й квартиры(K_i) и находятся по формуле m_ij' = m_ij - K_i. Номера квартир начинаются с 0. Нужно вывести ключ от квартиры Федора или None если его найти не получилось.

Ввод	Вывод
4 0 1	1
1 1,2 0,3 1,4 0	
3 0	
5 1,6 0	
1 1	
2 1	

Подсказка: используйте словарь для хранения ключей от еще не открытых комнат и множество для уже проверенных комнат.

ВАРИАНТ 20. Напишите функцию, принимающую на вход два аргумента (кортеж и случайное число). Требуется вернуть кортеж, начинающийся с первого появления случайного числа в нем, и заканчивающийся вторым его появлением. В случае отсутствия элемента в кортеже, вернуть пустой кортеж. В случае, если элемент присутствует в кортеже один раз, вернуть кортеж, который начинается с него.

Вопросы для самоконтроля к параграфам 7.2.7 – 7.2.8

1. Назовите четыре базовых типа данных в языке Python. Почему они называются базовыми?
2. Что означает термин «неизменяемый», и какие три базовых типа языка Python являются неизменяемыми?
3. Что означает термин «последовательность», и какие три базовых типа относятся к этой категории?
4. Что означает термин «отображение», и какой базовый тип является отображением?
5. Опишите основные свойства словарей.
6. Опишите основные свойства множеств.
7. Опишите основные свойства кортежей.
8. Представьте пример создания словаря с 0,1, 5 элементами.
9. Представьте пример создания множества с 0, 1, 5 элементами.
10. Представьте пример создания кортежа с 0, 1, 5 элементами.
11. В каком случае изменяемость кортежей может быть нарушена?
12. Чем именованный кортеж отличается от словаря?

Тест самоконтроля к параграфам 7.2.7 – 7.2.8

1. Выберите верные утверждения: а) словари изменяемы; б) словари могут быть любой глубины; в) словарь может содержать объект любого типа кроме словаря; г) доступ к элементам словаря осуществляется с помощью ключа; д) все ключи в словаре должны быть одного и того же типа.
2. Ниже представлены фрагменты кода. Какой из них удалит элемент с ключом “baz” из словаря? а) del (d[‘baz’]); б) d[‘baz’]; в) del.d[‘baz’]; г) del d(‘baz’).
3. Объявлен словарь — d = {'foo': 100, 'bar': 200, 'baz': 300}. Какой результат будет у d['bar':'baz']? а) возникнет ошибка; б) (200, 300); в) [200; 300]; г) 200 300.
4. Объявлен словарь: X = [‘a’, ‘b’, {‘foo’: 1, ‘bar’: {‘x’: 10, ‘y’:20, ‘z’: 30}, ‘baz’: 3}, ‘c’, ‘d’]. Как получить доступ к значению 30? а) x[1][2][2]; б) x[2][‘bar’][‘z’]; в) x[2][‘bar’][2]; г) x[1][1][3].
5. Ниже представлен список ключей. Какие из них синтаксически правильны? а) (‘foo’, ‘bar’); б) (3+2j); в) ‘foo’; г) [‘foo’, ‘bar’]; д) dict (foo=1, bar=2); е) len.

6. Объявлен словарь: X = [‘a’, ‘b’, {‘foo’: 1, ‘bar’: {‘x’: 10, ‘y’:20, ‘z’: 30}, ‘baz’: 3}, ‘c’, ‘d’]. Что будет выведено командой print ('z' in x[2])? а) true; б) false; в) 30; г) 3.
7. Объявлен словарь d = {'foo': 100, 'bar': 200, 'baz': 300}. Какой метод удалит элемент со значением 200? а) del d (200); б) del d (‘bar’); в) d.pop (200); г) d.pop (‘bar’).
8. Какие параметры принимает метод fromkeys() на вход? а) последовательность ключей и последовательность значений; б) значение; в) последовательность ключей и значение; г) ключ и значение.
9. Какой метод отвечает за копирование словаря? а) copy (); б) build (); в) put (); г) erase ().
10. Какой метод используется для возвращения ключей словаря? а) key(); б) get_keys(); в) keys(); г) keys_list().
11. Какой метод позволяет добавлять новые элементы в словарь? а) input(); б) update(); в) insert(); г)dict_in().
12. Какие параметры принимает метод pop() на вход? а) ключ и его значение; б) у метода нет параметров; в) ключ; г) значение.
13. Что представляют собой словари в Python? а) неупорядоченные коллекции; б) json-объекты; в) отсортированные списки; г) массивы.
14. Что возвращает метод items()? а) пары ключ/значение; б) значения; в) ключи; г) типы ключей и значений.
15. Какая функция отвечает за создание словаря? а) list(); б) dictionary(); в) create_dict(); г) dict().
16. Можно ли изменять ключи в Python-словаре? а) ключи можно изменять по требованию; б) ключи неизменны; в) не более 1 раза в момент инициализации.
17. Какой метод отвечает за удаление пары из словаря по ключу? а) clear (); б) pop (); в) del (); г) delete ().
18. Какие параметры принимает метод fromkeys () на вход? а) последовательность ключей и последовательность значений; б) значение; в) последовательность ключей и значение; г) ключ и значение.
19. За возвращение значения ключа в словаре отвечает метод … а) copy (); б) build (); в) put (); г) get ().
20. Кортеж используется, когда нам необходимо записать … а) данные, которые обновляются с некой периодичностью (1 раз в

месяц, 1 раз в год и т.д.); б) данные, которые предварительно были записаны в виде кортежа; в) данные, которые в дальнейшем нельзя изменять; г) данные, которые в дальнейшем можно изменять.

21. Что выведет этот код:
    sample = (10, 20, 30)
    sample.append(60)
    print(sample)
    а) (10, 20, 30, 60); б) [10, 20, 30, 60]; в) Ошибку; г) (10, 20, 30).
22. Кортеж – это… а) неупорядоченная неизменяемая коллекция объектов произвольных типов; б) упорядоченная изменяемая коллекция объектов произвольных типов; в) упорядоченная неизменяемая коллекция объектов произвольных типов; г) упорядоченная неизменяемая коллекция объектов одного типа; д) неупорядоченная неизменяемая коллекция объектов одного типа; е) упорядоченная изменяемая коллекция объектов одного типа.
23. Какие из следующих утверждений о кортежах верны? а) кортежи можно складывать; б) кортеж занимает больше памяти чем список; в) кортеж может быть ключом словаря; г) у кортежа есть метод append (); д) кортеж занимает меньше памяти, чем список; е) кортеж может быть частью другого кортежа.
24. Какая из функций, указанных ниже, создает пустой кортеж? а) такой функции нет в списке; б) tupl () в) typl() г) tuple () д) typle().
25. Задан кортеж lake = ("Python", 51, False, "22"). Каким из способов, указанных ниже, можно получить последний элемент кортежа? а) lake [1]; б) lake (1); в) lake [3]; г) lake (3); д) lake[4].
26. Какие из указанных ниже способов разделяют кортеж name_and_age =("Bob", 34) на значения? а) (name, age) = name_and_age; б) (name age) = name_and_age; в) (name) = name_and_age (age) = name_and_age; г) (name,) = name_and_age.
27. множество в python – это … а) это любая коллекция элементов; б)это список, содержащий в себе только функции; в) это контейнер, значения в котором не повторяются; г) это список, содержащий вложенные списки.

28. Какой из способов, указанных ниже, объявляет множество a? а) a = { }; б) a = []; в) a = set (); г) a = set.
29. Что выведет код?

```
a=[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
x= len( set(a))
 print (x)
```

а) 6; б) 5; в) 12; г) 0.
30. Чем отличаются методы remove () и discard (), применяемые к множеству? а) ничем не отличаются; б) метод remove() удаляет элемент если он есть в множестве, и бросает ошибку если элемента в множестве нет; в) метод discard() удаляет элемент, если он есть в множестве; г) метод discard() удаляет элемент если он есть в множестве, и бросает ошибку, если такого элемента нет; д) метод remove() удаляет элемент, если он есть в множестве.

Глава 8. ОЦЕНКА ЭФФЕКТИВНОСТИ АЛГОРИТМОВ РАБОТЫ С ОСНОВНЫМИ ТИПАМИ ДАННЫХ ПО ПАМЯТИ И ВРЕМЕНИ ИСПОЛНЕНИЯ

Программисту, работающему с данными, крайне важно выбирать правильные структуры данных для решения поставленной задачи, ведь выбор неправильного типа данных плохо влияет на производительность приложения.

Помимо структур данных программисту очень важно не ошибиться с алгоритмом их обработки, т.к. в алгоритме выполняется ряд операций. Эти операции могут включать в себя итерацию по коллекции, копирование элемента или всей коллекции, добавление элемента, вставку элемента, удаление элемента или обновление элемента в коллекции. Каждая из перечисленных операций имеет свою временную сложность.

Продемонстрируем это на примере вычисления «бета-значения» списка, содержащего показания счетчика (целые положительные числа), количество которых не превышает 10 000. «Бета-значением» будем считать минимальное четное произведение двух показаний, отстоящих друг от друга на расстоянии не менее шести элементов.

Решение «в лоб» этой задачи – запомнить входные данные в списке, после чего проверить все возможные пары элементов, удовлетворяющих требованию расстояния. Такая программа не будет эффективна ни по времени (используется алгоритм прямого перебора) ни по памяти (сохраняются все показания, т.е. размер списка определяется количеством входных элементов).

Одним из возможных эффективных по времени и памяти способом решения этой задачи является использование списка, размер которого определяется минимальным требованием к расстоянию (в данном случае 6), и использование алгоритма FIFO. Время его работы будет пропорционально значению минимального требования к расстоянию.

Сложность алгоритма – количественная характеристика, которая говорит о том, сколько времени он работает (временная сложность) либо о том, какой объем памяти требуется для его работы (эффективность по памяти).

Разумеется, для оценки временной сложности алгоритма и его эффективности по памяти существую специальные методы анализа.

На примере языка Python и его структур данных ниже мы разберемся с классами сложности различных операций и научимся комбинировать их, чтобы вычислить сложность целой функции.

Такой тип математического анализа называется статическим. Так как в отличие от динамического анализа, при котором проводятся измерения параметров работающего кода, не требует запуска программы.

Развитие технологии привело к тому, что память стала дешевой и компактной. Как правило, она не является критическим ресурсом – ее вполне достаточно для решения задачи. Поэтому чаще всего под анализом сложности алгоритма понимают исследование его временной сложности. Однако, оценка объема занимаемой памяти также не теряет своей актуальности.

Далее под сложностью будем понимать именно временную сложность алгоритма.

8.1. Оценка временной сложности алгоритма

В каких единицах измерять временную сложность алгоритмов? Понятно, что обычные единицы измерения времени (секунды и т.д.) здесь не подходят – одна и та же программа при одних и тех же входных данных на разных компьютерах будет выполняться, вообще говоря, разное время.

Физическое время выполнения алгоритма – это величина $\tau \cdot t$, где t – число действий (элементарных шагов, команд), а τ – среднее время выполнения одного элементарного действия.

Число t определяется описанием алгоритма и не зависит от физической реализации модели, а τ зависит от скорости обработки сигналов в элементах и узлах ЭВМ. Поэтому объективной математической характеристикой временной сложности алгоритма является число элементарных действий, выполняемых в ходе работы алгоритма.

Однако далеко не всегда ясно, какие операции следует считать элементарными. Кроме того, разные операции требуют для своего выполнения разного времени, да и перевод операций, используемых в описании алгоритма, в операции, используемые в компьютере, – дело очень неоднозначное, зависящее от таких, например, факторов, как свойства компилятора и квалификация программиста. Поэтому утверждение, что такой-то алгоритм при таких-то входных данных требует 150 000 000 операций, фактически не несет информации о ре-

альном времени вычислений. На самом деле задача анализа сложности алгоритма состоит в исследовании того, как меняется время работы при увеличении объема входных данных. Оно, конечно, растет, но с какой скоростью?

Временную сложность алгоритма выражают функцией T(n) зависимости времени работы (числа элементарных действий) от размера входа. Размер входа определяется для каждой задачи индивидуально.

Обычно у задачи есть какой-нибудь естественный параметр, характеризующий объем входных данных, и сложность оценивается по отношению к этому параметру. Например:

– в задачах обработки одномерных массивов под размером входа принято считать количество элементов в массиве;
– в задачах обработки двумерных массивов размером входа так же является количество элементов в массиве, но часто бывает полезно выразить это значение через количество строк и столбцов массива;
– в задачах обработки чисел (длинная арифметика, проверка на простоту и т.д.) более естественно считать размером общее число битов, необходимое для представления данных в памяти компьютера;
– в задачах обработки графов разумно за размер входа принять количество вершин графа, а иногда представить двумя значениями: число вершин и число ребер графа.

К элементарным действиям, определяющим временную сложность алгоритма, следует отнести, прежде всего, операции сравнения и присваивания.

Пример. Алгоритм сложения n чисел $x_1, x_2, \ldots, x_n$.

```
import random
sum=0
x=[]
n=int (input ())
for i in range (n):
    a=random.randint (1,10000)
    print (a)
    x.append(a)
    sum+=x[i]
print ('сумма ', sum)
```

Анализ алгоритма программы показывает, что, если числа не слишком велики, то можно измерять объем входных данных числом слагаемых. Считая элементарной операцией сложение, приходим к выводу, что временная сложность алгоритма равна n, где n – количество вводимых чисел.

Если же складываются очень большие числа, то для сложения нужно использовать специальные программы («длинную арифметику») и объем входных данных правильнее измерять максимальной длиной представления слагаемых (например, в битах).

Рассмотрим другой пример – поиск элемента в списке методом линейного поиска:

```
A=[44,1,9,2,7,8,9,10,16,11,12,14,13,15]
print ('задайте элемент для поиска')
x=int(input ())
flag=0
for i in range (len (A)):
    if A[i]==x:
        print ('искомый элемент есть в списке')
        flag=1
if flag==0: print('Искомого элемента в списке
нет')
```

Операцией, по которой будем оценивать временную сложность алгоритма, можно считать сравнение.

Число выполняемых сравнений в худшем случае (если элемента нет в списке) будет равно количеству элементов в списке (len (A)).

В лучшем случае, если x в списке стоит на первом месте, – 1.

Если предварительно упорядочить список A, тогда в среднем будет выполняться len (A)/2 сравнений.

Это наиболее распространенные ***меры сложности*** – ***трудоемкость в худшем, в среднем и в лучшем случаях***. Чаще они различаются гораздо сильнее, чем в этом примере.

Для оценки временной сложности алгоритма нас будет интересовать время работы в худшем случае, которое определяется как **максимальное время**, так как:

– зная время работы в худшем случае можно гарантировать, что при любом входе работа будет длиться не дольше;
– на практике худший случай встречается часто, например, поиск несуществующего элемента в списке;

– часто время работы в среднем случае часто оказывается ближе к оценке худшего случая.

Для сравнения алгоритмов по степени роста времени работы введены асимптотические обозначения.

Определение 1. Говорят, что время работы алгоритма T(n) имеет порядок роста g(n), если существуют натуральное число n_0 и положительные константы c_1 и c_2 ($0 < c_1 \le c_2$), такие, что для любого натурального n начиная с n_0 выполняется неравенство: $c_1 \cdot g(n) \le T(n) \le c_2 \cdot g(n)$.

Обозначение: T(n) = Θ(g(n))[2].

$$T(n) = \Theta\big(g(n)\big), \text{если } \exists n_0 \in N, 0 < c_1 \le c_2:$$

$$\forall n \ge n_0 \; c_1 \cdot g(n) \le T(n) \le c_2 \cdot g(n)$$

Замечание: говорят, что время работы алгоритма T(n) имеет порядок роста g(n) (T(n) = Θ(g(n))), если и сверху и снизу время работы ограничено функцией с одной и той же степенью роста. Если это не так, то говорят о верхних (худший случай) и нижних (лучший случай) оценках.

Определение 2. Говорят, что время работы алгоритма T(n) имеет нижнюю оценку g(n), если существуют натуральное число n_0 и положительная константа c, такие, что для любого натурального n, начиная с n_0, выполняется неравенство $c \cdot g(n) \le T(n)$.

Обозначение: T(n) = Ω(g(n))[3].

$$T(n) = \Omega\big(g(n)\big), \text{если } \exists n_0 \in N, \mathrm{c} > 0:$$

$$\forall n \ge n_0 \; c \cdot g(n) \le T(n)$$

Определение 3. Говорят, что время работы алгоритма T(n) имеет верхнюю оценку g(n), если существуют натуральное число n0 и положительная константа c, такие, что для любого натурального n, начиная с n_0, выполняется неравенство $T(n) \le c \cdot g(n)$.

Обозначение: T(n) = O(g(n))[4].

$$T(n) = O\big(g(n)\big), \text{если} \exists n_0 \in N, c > 0:$$

$$\forall n \ge n_0 \; T(n) \le c \cdot g(n)$$

Таким образом, для линейного поиска (см. пример поиска элемента) справедливы следующие оценки: T(n) = Ω(1) и T(n) = O(n).

Пример. Доказать, что функция $T(n) = 3 \cdot n^3 + 2 \cdot n^2$ имеет верхнюю оценку O(n^3).

[2] Читается как «Тэта большое от g от n»

[3] Читается как «Омега большое от g от n»

[4] Читается как «О большое от g от n»

Положим n_0=1, тогда $\forall n \geq 1$ должно выполняться неравенство $3 \cdot n^3 + 2 \cdot n^2 \leq c \cdot n^3$.

Преобразуем это неравенство $3 + \frac{2}{n} \leq c$, т.е. $c \geq 5$.

Таким образом, найдены натуральное число n_0=1 и положительная константа c=5, такие, что для любого натурального n, начиная с n_0, выполняется неравенство $3 \cdot n^3 + 2 \cdot n^2 \leq c \cdot n^3$. Следовательно, $T(n)=O(n^3)$.

Ограниченность показателя степени роста

Итак, программы можно оценить с помощью функций времени работы, пренебрегая при этом константами пропорциональности.

С этой точки зрения программа с временем работы $O(n^2)$ лучше программы с временем работы $O(n^3)$. Однако, константы пропорциональности зависят не только от используемых компилятора и компьютера, но и от свойств самой программы.

Пусть одна программа выполняется за $100 \cdot n^2$ миллисекунд, а вторая – за $5 \cdot n^3$ миллисекунд. Может ли вторая программа быть предпочтительней первой? Ответ зависит от размера входа: при $n<20$ предпочтительней вторая программа, во всех остальных случаях предпочтительнее будет первая программа.

Таким образом, чем меньше степень роста, тем больше размер задачи, которую можно решить за фиксированный промежуток времени (см. таблица 10).

Таблица 10. Зависимомть роста размера задачи от размера входа

T(n)	Максимальный размер задачи, решаемый за данное время		Рост размера задачи (количество раз)
	10^3секунд	10^4секунд	
$100 \cdot n$	10	100	10
$5 \cdot n^2$	14	45	3,2
$\frac{n^3}{2}$	12	27	2,3
2^n	10	13	1,3

Помимо указанных выше, функциями, часто встречающимися при анализе алгоритмов, также являются:

– log n (логарифмическое время),

– n (линейное время);

– $n \cdot \log n$;

– n^2 (квадратичное время);
– 2^n (экспоненциальное время).

Первые четыре функции имеют невысокую скорость роста и алгоритмы, время работы которых оценивается этими функциями, можно считать быстродействующими. Экспоненциальная функция – отдельный случай.

Определение. Полиномиальным алгоритмом (или алгоритмом полиномиальной временной сложности) называется алгоритм, временная сложность которого равна O(p(n)), где p(n) – некоторая полиномиальная функция, а n – входная длина.

Однако не всякая задача может быть решена за полиномиальное время. Примером может служить «Задача об укладке рюкзака»: имеется n предметов, для каждого из которых известны стоимость C_i и объем V_i . Необходимо определить какие предметы необходимо уложить в рюкзак, чтобы их суммарный объем не превышал допустимый объем V рюкзака, а их общая стоимость C была наибольшей.

При условии, что каждый из n предметов может быть упакован или не упакован в рюкзак, всего будет 2^n вариантов взятых предметов, т.е. решением является не полиномиальный, а экспоненциальный алгоритм ($O(a^n)$). Такие задачи называют трудно решаемыми.

Отличие полиномиальных и экспоненциальных алгоритмов будет более заметно, если обратиться к таблице, в которой отображено время работы алгоритма на компьютере, выполняющем 1 000 000 оп/с (см. таблицу 11).

Таблица 11. Время работы алгоритма

n O(n)	10	20	30	40	50
n	0,00001 с.	0,00002 с.	0,00003 с.	0,00004 с.	0,00005 с.
n^2	0,0001 с.	0,0004 с.	0,0009 с.	0,00016 с.	0,0025 с.
n^5	0,1 с.	3,2 с.	24,3 с.	1,7 мин.	5,2 мин.
2^n	0,001 с.	1 с.	17,9 мин.	12,7 дней.	35,7 лет
3^n	0,059 с.	58 мин.	6,5 лет	3855 стол.	$2 \cdot 10^8$ стол.
n!	3,6 с.	771,5 стол.	$8 \cdot 10^{16}$стол.	……	……

8.2. Правила вычисления времени выполнения программ

Правило 1. Правило сумм. Пусть $T_1(n)$ и $T_2(n)$ – время выполнения двух последовательных фрагментов программы P_1 и P_2 соответ-

ственно. Пусть $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$. Тогда $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$.

Пример. Пусть имеется два фрагмента со временем выполнения O(f(n)) и O(g(n)), где

$$f(n) = \begin{cases} n^4 - \text{если } n - \text{четно} \\ n^2 - \text{нечетно} \end{cases}, f(n) = \begin{cases} n^2 - \text{ если } n - \text{четно} \\ n^3 - \text{нечетно} \end{cases}$$

Так как T(n)= O(max(f(n), g(n))),

то $T(n) = O(\begin{cases} n^4 - \text{ если } n - \text{четно} \\ n^3 - \text{нечетно} \end{cases})$

Следствие. Если $g(n) \le f(n)$ для всех $n \ge n_0$, то

$O(f(n)+g(n))= O(f(n))$.

Пример. $O(n^2+n) = O(n^2)$, т. е. слагаемыми, имеющими меньший порядок роста, в силу правила сумм можно пренебречь.

Правило 2. Правило произведений. Пусть $T_1(n)$ и $T_2(n)$ – время выполнения двух вложенных фрагментов программы P_1 и P_2 соответственно. Пусть $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$. Тогда $T_1(n) \cdot T_2(n) = O(f(n) \cdot g(n))$.

Пример. $O(n^2/2)=O(n^2)$, т. е. постоянным множителем, в силу правила произведений можно пренебречь.

Однако существуют ситуации, когда постоянный множитель следует учитывать. Это бывает, когда сравниваются разные алгоритмы для одной задачи.

Правило 3. Время выполнения операторов присваивания, чтения, записи, сравнения обычно пропорционально единице, т. е. равно O(1).

Правило 4. Время выполнения последовательности операторов определяется с помощью правила сумм, следовательно, равно наибольшему времени выполнения оператора в данной последовательности.

Правило 5. Время выполнения условных операторов состоит из времени условно исполняемых операторов и времени вычисления самого логического выражения, т. е. O(if–then–else)=O(if)+O(max(then, else)).

Правило 6. Время выполнения цикла является суммой времени всех исполняемых итераций цикла, в свою очередь состоящих из времени выполнения операторов тела цикла и времени вычисления условия прекращения цикла.

Часто время выполнения цикла вычисляется, пренебрегая определением констант пропорциональности, как произведение количе-

ства выполненных итераций цикла на наибольшее возможное время выполнения операторов тела цикла.

Правило 7. Вызов процедур. Определять время работы такой программы следует в порядке, обратном порядку вызова процедур.

Пример. Допустим, в основной программе вызываются процедуры A и D. В свою очередь, в процедуре A вызывается процедура B, а в B – C. Из процедуры D вызывается процедура E. Время работы такой программы:

1) T(C) → T(B) → T(A);
2) T(E) → T(D);
3) T(A) + T(D).

Правило 8. Анализ времени работы рекурсивной процедуры.

Время работы рекурсивных алгоритмов складывается из:

- D(n) – времени разбиения исходной задачи на подзадачи (действия на входе в рекурсию);
- $T(n_1) + \ldots + T(n_k)$ – времени решения каждой подзадачи в отдельности (количество подзадач k определяется числом рекурсивных вызовов);
- S(n) – времени на соединение полученных решений (действия на выходе из рекурсии).

8.3. Анализ сортировки вставками

В качестве примера вычислим временную сложность алгоритма сортировки простыми вставками.

Запишем алгоритм так, чтобы в каждой строке располагалось по одной команде (таблица 11). Обозначим через константу c_i стоимость одного выполнения i-й строки.

Определим количество выполнений каждой строки в ходе исполнения алгоритма. Заголовок цикла проверяется на один раз больше, чем команда внутри цикла, поэтому 1-я строка выполняется n раз, а команды строк 2, 3, 7 – (n–1) раз, так как содержатся в цикле и не зависят ни от каких условий. Для 4-й строки (заголовка цикла While) нельзя однозначно сказать, сколько раз она будет выполняться на i-й итерации внешнего цикла, так как это зависит от расположения элементов в массиве. Поэтому формально это число обозначим через t_i.

Таблица 12. Оценка временной сложности алгоритма сортировки простыми вставками

№ строки алгоритма	Алгоритм	Стоимость одного выполнения строки	Количество выполнений строки
1	for i in range (0,n):	c_1	n
2	x=A[i]	c_2	n-1
3	j=i-1	c_3	n-1
4	while (j>=0) and (A[j]>=x):	c_4	$\sum_{i=0}^{n} t_i$
5	A[j+1]=A[j]	c_5	$\sum_{i=0}^{n} (t_i - 1)$
6	j=j-1	c_6	$\sum_{i=0}^{n} (t_i - 1)$
7	A[j+1]=x	c_7	n-1

Полный текст кода алгоритма представлен ниже:

```
n=8
A=[4,1,67,89,2,6,90,23]
for i in range (0,n):
    x=A[i]
    j=i-1
    while (j>=0) and (A[j]>=x):
        A[j+1]=A[j]
        j=j-1
    A[j+1]=x
print (A)
```

Тогда время работы алгоритма будет определяться выражением:

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{i=0}^{n} t_i + c_5 \cdot \sum_{i=0}^{n} (t_i - 1) + c_6 \cdot \sum_{i=0}^{n} (t_i - 1) + c_7 \cdot (n-1)$$

Так как функция времени работы алгоритма T(n) должна зависеть только от n, то необходимо избавиться от формально введенных t_i. Для этого рассмотрим работу алгоритма в лучшем и худшем (с точки зрения числа выполняемых операций) случаях.

Лучший случай – массив уже отсортирован, тогда на каждой итерации внешнего цикла действие цикла while (4-я строка алгоритма) прекращается сразу же после первой проверки, следовательно, все

t_i=1, тогда $\sum_{i=0}^{n} t_i = n - 1, \sum_{i=0}^{n}(t_i - 1) = 0$. То есть, $T_{\text{луч.сл.}}(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_5 \cdot 0 + c_6 \cdot 0 + c_7 \cdot (n-1) = (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n + (-c_2 - c_3 - c_4 - c_7) = A \cdot n + B$ – линейная зависимость.

Худший случай – массив отсортирован в обратном порядке, тогда на каждой i-й итерации внешнего цикла 4-я строка алгоритма выполняется i раз, следовательно, все t_i=i. Тогда

$$\sum_{i=0}^{n} t_i = \sum_{i=0}^{n} i = \frac{(2+n)\cdot(n-1)}{2} = \frac{n^2+n+2}{2},$$

$$\sum_{i=0}^{n}(t_i - 1) = \sum_{i=0}^{n}(i-1) = \frac{(1+(n-1))\cdot(n-1)}{2} = \frac{n^2-n}{2},$$

$$T_{\text{худ}_{\text{сл}}}(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \frac{n^2+n+2}{2} + c_5 \cdot \frac{n^2-n}{2} + c_6 \cdot \frac{n^2-n}{2} + c_7 \cdot (n-1)$$
$$= (c_4 + c_5 + c_6) \cdot n^2 + (c_1 + c_2 + c_3) + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7) \cdot n + (-c_2 - c_3 - c_4 - c_7) = A' \cdot n^2 + B' \cdot n + C'$$

Получаем квадратичную зависимость.

Таким образом, в соответствии с правилами 1 и 2, время работы алгоритма сортировки вставками (метод прямого включения) имеет нижнюю оценку Ω(n) и верхнюю оценку $O(n^2)$.

8.4. Анализ рекурсивных алгоритмов

Проанализируем рекурсивный алгоритм на примере анализа сортировки слиянием. Основной метод анализа временной сложности рекурсивных алгоритмов – декомпозиция - разделение задачи на части меньшей размерности, получение решение для полученных частей и объединение решений.

Полный код программы представлен ниже:

```
def merge_sort(array, left_index, right_index):
    if left_index >= right_index:
        return
    middle = (left_index + right_index)//2
    merge_sort(array, left_index, middle)
    merge_sort(array, middle + 1, right_index)
    merge(array, left_index, right_index, middle)
```

```
# делаем копии списков, которые будем объединять
def merge(array, left_index, right_index, middle):
     """второй параметр не является инклюзивным,
     поэтому увеличиваем его значение на 1"""
    left_copy = array[left_index:middle + 1]
    right_copy = array[middle+1:right_index+1]

    """ зададим начальные значения переменных, ис-
пользуемых для сортировки, для того, чтобы отсле-
живать индексы элементов"""
        left_copy_index = 0
    right_copy_index = 0
    sorted_index = left_index

    """сортируем обе половины списка"""
    while left_copy_index < len(left_copy) and
right_copy_index < len(right_copy):

        if left_copy[left_copy_index] <=
right_copy[right_copy_index]:
            array[sorted_index] =
left_copy[left_copy_index]
            left_copy_index = left_copy_index + 1
        else:
            array[sorted_index] =
right_copy[right_copy_index]
           right_copy_index = right_copy_index + 1

            orted_index = sorted_index + 1
while left_copy_index < len(left_copy):
        array[sorted_index] =
left_copy[left_copy_index]
        left_copy_index = left_copy_index + 1
        sorted_index = sorted_index + 1

    while right_copy_index < len(right_copy):
        array[sorted_index] =
right_copy[right_copy_index]
        right_copy_index = right_copy_index + 1
```

```
        sorted_index = sorted_index + 1

array = [33, 42, 9, 37, 8, 47, 5, 29, 49, 31, 4,
        48, 16, 22, 26]
merge_sort(array, 0, len(array) -1)
print(array)
```

Основная идея алгоритма состоит в том, чтобы разделить список на половинки и отсортировать каждую рекурсивно. Делать это необходимо до тех пор, пока не будут получены подсписки, имеющие только один элемент.

Если список состоит из одного элемента (n=1, лучший случай), выполняется только проверка условия (left_index >= right_index), что дает оценку для временной сложности Q(1) .

Если в списке более одного элемента (n>1, худший случай), рекурсивно вызываем сортировку полученных массивов половинной длины, что дает временную сложность – $2 \cdot Q(\frac{n}{2})$. Затем объединяем возвращенные отсортированные списки за время Q (n).

Тогда ожидаемая трудоемкость на сортировку составит: T(n)= 2 * Q(n/2)+ Q(1)+ Q (n).

Так как сложность Q(1) меньше сложности Q (n), то с учетом правил 1 и 2, слагаемое Q (1) можно отбросить.

Таким образом, время работы сортировки слиянием определяется соотношением:

$$T(n) = \begin{cases} 2 \cdot T\left(\frac{n}{2}\right) + Q(n), \text{если } n > 1 \\ Q(1), \text{если } n = 1 \end{cases}$$

8.5. Дополнительные источники по оценке временной сложности алгоритмов

1. Коварцев А.Н., Даниленко А.Н. Алгоритмы и анализ сложности: учебник. – Самара: Изд-во Самарского университета, 2018. – 128 с.: ил.
2. Математический аппарат анализа сложности алгоритмов. Режим доступа: https://pro-prof.com/archives/1660
3. Алгоритмы и анализ сложности. Режим доступа: http://repo.ssau.ru/bitstream/Uchebnye-izdaniya/Algoritmy-i-analiz-slozhnosti-Elektronnyi-resurs-uchebnik-73319.pdf
4. Сложность алгоритмов. Режим доступа: https://teach-in.ru/file/synopsis/pdf/algorithm-M.pdf

Практическая работа № 9
ОЦЕНКА ВРЕМЕННОЙ СЛОЖНОСТИ АЛГОРИТМА

Задание

Оформите отчет по практической работе, содержащий:

- титульный лист;
- содержание;
- ответ на вопросы самоконтроля;
- для каждого варианта:
  - текст задания с указанием варианта;
  - блок-схема алгоритма решения задачи;
  - плановая обработка ошибок;
  - программный код на языке Python версии 3.8, соответствующий блок-схеме;
  - unit-тесты;
  - скриншоты результата работы программы для каждого unit-теста.
  - анализ временной сложности выбранного алгоритма программы.

ВАРИАНТЫ ЗАДАНИЙ

Вариант 1. Выполните анализ временной сложности использованного алгоритма линейного поиска с барьером (ЛПБ) в списке. ЛПБ является модификацией линейного поиска, осуществляемого путем последовательного сравнения очередного рассматриваемого значения с искомым до тех пор, пока не совпадут эти значения или границы поиска. Отличие – не проверяет каждый раз в цикле условие, связанное с границами списка. Это можно обеспечить, установив в данном множестве так называемый барьер. Под барьером понимается любой элемент, который удовлетворяет условию поиска. Тем самым будет ограничено изменение индекса. Выход из цикла, в котором теперь остается только условие поиска, может произойти либо на найденном элементе, либо на барьере. Существует два способа установки барьера: дополнительным элементом или вместо крайнего элемента списка.

ВАРИАНТ 2. Выполните анализ временной сложности использованного алгоритма функции для заполнения вложенного списка. Список заполняется натуральными числами «змейкой»: сначала заполняется первая строка, затем последний столбец (сверху вниз), по-

следняя строка (справа налево), первый столбец (снизу вверх), вторая строка (слева направо), и т.д.

ВАРИАНТ 3. Дана строка, содержащая произвольный русский текст, длиной не более 200 символов. Выполните анализ временной сложности алгоритма подсчета количества вхождений каждого символа в эту строку.

ВАРИАНТ 4. Дан произвольный список чисел. Выполните анализ временной сложности алгоритма поиска «бесполезного числа» в этом списке. Решение реализовать с помощью функции. Примечание: бесполезным числом является число, полученное делением максимального числа списка на его длину.

ВАРИАНТ 5. Даны два списка, состоящие из строчных латинских букв. Выполните анализ временной сложности алгоритма построения нового списка, в который войдут только общие символы из двух списков в алфавитном порядке и без повторений.

ВАРИАНТ 6. Выполните анализ временной сложности алгоритма функции, которая для списка, переданного ее аргументом, возвращает список из двух элементов: значение наибольшего элемента в списке и индекс этого элемента в списке (если таких элементов несколько, то индекс первого из таких элементов).

ВАРИАНТ 7. Дано слово «сообщение». Оценить временную сложность алгоритма программы, которая проверяет любое введенное пользователем слово и выдает сообщение, можно ли из его букв составить слово «сообщение».

ВАРИАНТ 8. Дана строка, содержащая произвольный русский текст, длиной не более 50 символов. Оцените временную сложность алгоритма анализа строка, выводящего символы строки, входящие в нее не более одного раза.

ВАРИАНТ 9. Из натуральных чисел, принадлежащих отрезку [101 000 000; 102 000 000], сформируйте матрицу из чисел, у которых ровно три различных чётных делителя. Оцените временную сложность использованного алгоритма.

ВАРИАНТ 10. Назовём нетривиальным делителем натурального числа его делитель, не равный единице и самому числу. Например, у числа 6 есть два нетривиальных делителя: 2 и 3. Из натуральных чисел, принадлежащих отрезку [123456789; 223456789] и имеющих ровно три нетривиальных делителя, сформируйте список. Оцените временную сложность алгоритма программы.

ВАРИАНТ 11. Задайте список и заполните его случайными числами. Вставьте между каждой парой элементов этого списка новый элемент, равный сумме значений соседних элементов. Оцените временную сложность алгоритма программы.

ВАРИАНТ 12. Произвольное натуральное число можно представить различными способами в виде произведения двух натуральных чисел. Например, для числа 16 получим: 16 = 16*1 = 8*2 = 4*4. Для каждого такого произведения разность сомножителей будет содержать числа 15, 6 и 0. Оцените временную сложность алгоритма поиска среди натуральных чисел, принадлежащих отрезку [1 000 000; 2 000 000], таких, у которых множество разностей содержит не меньше трёх элементов, не превышающих 100.

ВАРИАНТ 13. Оцените временную сложность алгоритма поиска среди натуральных чисел, принадлежащих отрезку [200 000 000; 400 000 000] тех, которые можно представить в виде $N = 2^m \cdot 3^n$, где m – чётное число, n – нечётное число.

ВАРИАНТ 14. Пусть M(N) – произведение различных делителей натурального числа N, не считая единицы и самого числа. Если у числа N меньше 5 таких делителей, то M(N) считается равным нулю. Оцените временную сложность алгоритма поиска среди натуральных чисел, превышающих 200 000 000, 5-ти наименьших натуральных чисел, для которых $0 < M(N) < N$.

ВАРИАНТ 15. Оцените временную сложность алгоритмов программы формирования списка из целых чисел, принадлежащих отрезку [190061; 190080], и имеющих ровно 4 различных четных делителя. Каждый элемент списка представляет собой вложенный список найденных делителей каждого из найденных чисел.

ВАРИАНТ 16. Напишите программу, которая ищет среди целых чисел, принадлежащих числовому отрезку [35000; 45221], простые числа, оканчивающиеся на число 19. Сформируйте список из найденных чисел. Выведите элементы списка в порядке возрастания, слева от каждого числа выведите его номер по порядку.

ВАРИАНТ 17. Пусть M – сумма минимального и максимального натуральных делителей целого числа, не считая единицы и самого числа. Если таких делителей у числа нет, то значение M считается равным нулю. Среди целых чисел, больших 700 000, найдите такие, для которых значение M оканчивается на 8. Сформируйте список из пяти первых найденных чисел и соответствующих им значениям M.

ВАРИАНТ 18. Напишите программу, в которой создается числовой список. Список заполняется случайными числами. Затем между каждой парой элементов этого списка вставляется новый элемент, равный сумме значений соседних элемент.

ВАРИАНТ 19. На вход программы подается 10 строк, представляющих из себя различные слова. Создать список, состоящий из уникальных букв слов. Элементы списка отсортировать по возрастанию.

ВАРИАНТ 20. На вход программы подается 10 строк, представляющих из себя различные слова, и слово «ИНФОРМАТИКА». Создать список, состоящий из уникальных букв слов. Проверить, можно ли из элементов списка составить заданное слово.

Вопросы для самоконтроля к главе 8

1. Объясните термин сложность алгоритма.
2. Назовите показатели, характеризующие сложность алгоритмов.
3. Как Вы считаете, какие факторы влияют на сложность алгоритма?
4. От чего зависит расход памяти и время выполнения алгоритма?
5. Когда возможно практическое применение алгоритма?
6. Что является объективной характеристикой временной сложности алгоритма?
7. Что представляет собой функция времени работы алгоритма?
8. Что дает оценка времени работы алгоритма в худшем, лучшем и среднем случаях? Какова их связь с асимптотическими обозначениями?
9. Из чего складывается время работы рекурсивного алгоритма?
10. В чем заключаются сложности каждого из способов решения рекуррентных соотношений?
11. Какие функции используются для представления верхней оценки сложности алгоритма?
12. У каких известных вам алгоритмов сложность является константной, а у каких - линейной?
13. Как влияет размер списка на временную сложность алгоритма?
14. Как влияет количество циклов повторения исследуемого алгоритма на погрешность определения времени его выполнения?
15. В каких случаях не применима теорема о рекуррентных соотношениях?

16. Даны следующие функции от n:

$f_1(n) = n^2;$

$f_2(n) = n^2 + 1000 \cdot n;$

$f_3(n) = \begin{cases} n, \text{если } n \text{ нечетно;} \\ n^3, \text{если } n \text{ нечетно.} \end{cases}$

$f_4(n) = \begin{cases} n, \text{если } n > 100; \\ n^3, \text{если } n \leq 100. \end{cases}$

Указать для каждой пары функций, когда f_i(n имеет порядок роста $O(f_j(n))$ и когда $f_i(n)$ есть $\Omega(f_j(n))$.

17. Можно ли утверждать, что $2^{n+1} = O(2^{n+1})$; $2^{2 \cdot n} = O(2^n)$?
18. Доказать по определению, что следующие утверждения истинны:
    a. 17 имеет порядок O(1);
    b. $n \cdot \frac{n-1}{2}$ имеет порядок $O(n^2)$;
    c. $\max(n^3, 10 \cdot n^2)$ имеет порядок $O(n^3)$.

Тест самоконтроля к главе 8

1. Задан список X [N]. Определите число операций сложения, которые выполняются при работе этой программы:

```
N=int (input ())
X=[]
for i in range (N):
    a=int(input())
    X.append(a)
S=X[0]+X[N-1]
for k in range(N):
    X[k]+=X[k]+S
```

При вводе ответа для обозначения операции умножения используйте символ *, степени – символ ^.

2. Задан список X[N]. Определите число операций умножения, которые выполняются при работе этой программы:

```
N=int (input ())
X=[]
for i in range (N):
    a=int(input())
    X.append(a)
S=X[0]*X[N-1]
for k in range (N):
```

```
    X[k]=2*X[k]+S
for i in range(3):
      S=S*2
      print(S)
```
При вводе ответа для обозначения операции умножения используйте символ *, степени – символ ^.

3. Задан список X[N]. Определите число операций сложения, которые выполняются при работе этой программы:
```
N=int (input ())
X=[]
for i in range (N):
    a=int(input())
    X.append(a)
S=X[1]+X[N-1]+3
for k in range(N):
    for m in range(N):
      X[k]=X[k]+S
```
При вводе ответа для обозначения операции умножения используйте символ *, степени – символ ^.

4. Количество операций при выполнении некоторого алгоритма равно
$T(N) = 5 \cdot N^2 + 3 \cdot N + 1$
Определите наиболее точную оценку временной сложности алгоритма:
а) O(1); б) O(N); в) $O(N)^2$; г) $O(N)^3$; д) $O(2^N)$.

5. Задан список X[N]. Определите наиболее точную оценку временной сложности алгоритма:
```
N=int (input ())
X=[]
for i in range (N):
    a=int(input())
    X.append(a)
S=X[1]+X[N-1]
for k in range (1,N+1):
    for m in range (k+1):
      X[k]=X[k]+X[m]+S
```
а) O(1); б) O(N); в) $O(N)^2$; г) $O(N)^3$; д) $O(2^N)$.

6. Задан массив X[1..N]. Определите наиболее точную оценку временной сложности алгоритма:

```
N=int (input())
X=[]
for i in range(N):
    a=float(input())
    X.append(a)
S=X[1]+X[N-1]
for k in range(1,N):
    for m in range (1,N):
      for q in range(k):
          X[k]=X[k]+X[q]+S
```

а) O(log N); б) O(N); в) $O(N)^2$; г) $O(N)^3$; д) $O(2^N)$.

7. Задан массив X[1..N]. Определите наиболее точную оценку временной сложности алгоритма:

```
def Rec(N):
    Rec=0
    if N > 3:
        Rec=Rec+Rec(N-1)+2*Rec(N-2)
N=int(input())
```

а) O(log N); б) O(N); в) $O(N)^2$; г) $O(N)^3$; д) $O(2^N)$.

8. Задан массив X[1..N]. Определите наиболее точную оценку временной сложности алгоритма:

```
import math
N=int (input ())
X=[]
for i in range (N):
    a=int(input())
    X.append(a)
S=X[1]+X[N-1]
for k in range (1,int (math.sqrt (N))):
    for m in range (1,2*((N-1)**2)):
        X[k]=X[k]+X[m]+S
```

а) O(log N); б) O(N); в) $O(N)^2$; г) $O(N)^3$; д) $O(2^N)$.

9. Задан массив X[1..N]. Определите наиболее точную оценку временной сложности алгоритма:

```
N=int (input())
X=[]
for i in range(N+1):
    a=float(input())
    X.append(a)
L=1
```

```
R=N+1
while L<R-1:
    c=L+(R-L)//2
    if R < X[c]:
        R=c
    else:
        L=c
```

а)O(logN); б) O(N); в)O(N^2); г) O(N^3); O(2^n).

10. Задан массив X[1..N]. Определите наиболее точную оценку временной сложности алгоритма:

```
N=int (input ())
X=[]
for i in range (N):
    a=int(input())
    X.append(a)
S=X[1]+X[N-1]
for k in range (1,N):
    for m in range (5):
        X[k]=X[k]+S
```

а)O(logN); б) O(N); в)O(N^2); г) O(N^3); O(2^n).

11. Задан массив X[1..N]. Определите наиболее точную оценку временной сложности алгоритма:

```
N=int (input ())
R=int (input ())
X=[]
for i in range (N):
    a=int(input())
    X.append(a)
k=0
for i in range (1,N):
    if X[i] == R:
      k=i
      break
```

а)O(logN); б) O(N); в)O(N^2); г) O(N^3); O(2^n).

Глава 9. ФАЙЛЫ

Объекты-файлы – это основной интерфейс между программным кодом на языке Python и внешними файлами на компьютере. Рассмотрим основные операции работы с файлами.

9.1. Открытие файла

Python предоставляет функцию open (), которая принимает два аргумента: имя файла и режим доступа, в котором осуществляется доступ к файлу. Функция возвращает файловый объект, который можно использовать для выполнения различных операций, таких как чтение, запись и т. д. Если файл не может быть открыт, бросается исключение OSError.

Синтаксис:

file object = open(<file-name>, <mode>, < buffering >, < encoding >, <errors>, < newline >, < closefd >, < opener >)

где:

```
fp = open(file, mode='r', buffering=-1, encoding=None,
          errors=None, newline=None, closefd=True, opener=None)
```

Например, режим перевода строк (newline), errors – строка, указывающая, как должны обрабатываться ошибки кодирования и декодирования.

Режимы доступа (mode)**:**

- r - открывает файл только для чтения,
- w - открыт для записи (перед записью файл будет очищен),
- x - эксклюзивное создание, бросается исключение FileExistsError, если файл уже существует.
- A - открыт для добавления в конец файла (на некоторых Unix-системах пишет в конец файла вне зависимости от позиции курсора)
- + - символ обновления (чтение + запись).
- t - символ текстового режима.
- B - символ двоичного режима (для операционных систем, которые различают текстовые и двоичные файлы).

Различные комбинации режимов:

Режимы 'r+' и 'r+b' открывают файл и устанавливают курсор на начало файла, запись в файл начинается с места остановки курсора при его чтении перед записью. Если файл перед записью не читался то запись осуществляется в начало файла.

Режимы 'w' и 'wb' создают новый файл или открывают существующий файл только для записи, с последующей его очисткой (стирает все перед записью).

У режимов 'w+' и 'wb+' поведение такое же, как в предыдущем случае, только, если не закрывать файл после записи, его еще можно потом прочитать.

Python различает двоичный и текстовый ввод-вывод. Файлы, открытые в двоичном режиме, возвращают содержимое в виде байтов (шестнадцатеричные коды) без какого-либо декодирования. В текстовом режиме содержимое файла возвращается как строки, которые были сначала декодированы с использованием кодировки, используемой системой по умолчанию или с использованием переданного аргумента encoding.

Аргумент **buffer** - необязательное целое число, используемое для установки политики буферизации.

- 0 - отключить буферизацию, только для бинарного режима;
- 1 - построчная буферизация, только для текстового режима;
- int число > 1 - размер буфера в байтах.
- -1 - по умолчанию.

У текстовых файлов, если buffering не задан, используется построчная буферизаця. Двоичные файлы буферизируются кусками фиксированного размера. Для многих систем буфер равен 4096 или 8192 байт. Позволяет записать последовательность байтов в текстовый поток. Например:

```
import io
f=open(r'f.txt', 'w', encoding='cp1251')
f.buffer.write (bytes ("Строка", "cp1251"))
f.close ()
```

Аргумент encoding: имя кодировки, используемой для декодирования или кодирования файла. При чтении из файла происходит попытка преобразовать данные в кодировку Unicode, при записи строка преобразуется в последовательность байтов в определенной кодировке. По умолчанию используется кодировка, используемая в системе. Его следует использовать только в текстовом режиме.

При работе с файлами в кодировках UTF-8, UTF-16 и UTF-32 следует учитывать, что в начале файла могут быть служебные слова, называемые сокращенно BOM – метка порядка файлов), для UTF-8 является необязательным:

- encoding=”utf-8” – маркер BOM был прочитан из файла;

– encoding="utf-8-sig" - маркер BOM не попал в результат.

Для кодировок UTF-16 и UTF-32 является обязательным и обрабатывается автоматически. Для разновидностей этих кодировок, например, UTF-16, маркер необходимо самим добавлять в начало файла при записи, а при чтении необходимо самим удалить его.

```
f=open(r'f.txt', 'w', encoding="utf-8-sig")
f.write("Строка")
f.close()
with open (r'f.txt', 'r', encoding="utf-8") as f:
    for line in f:
        print (repr (line))
    with open (r'f.txt', 'r', encoding="utf-8-
sig") as f:
    for line in f:
        print (repr (line))
```

r – означает что файл открыт в текстовом режиме для чтения/записи.

Функция repr () вернет строку, содержащую печатаемое формальное представление объекта. В нашем случае:

```
\ufeffСтрока
Строка
```

Аргумент errors: необязательная строка, которая указывает, как должны обрабатываться ошибки кодирования и декодирования, не может использоваться в двоичном режиме. Может принимать следующие стандартные значения:

– 'strict' –ошибка кодирования возбуждает исключение ValueError. (значение None делает тоже самое);
– 'ignore' – ошибка кодирования игнорируется;
– 'replace' – ошибка кодирования заменяется на замещающий маркер (например - ?). Используется только с текстовыми кодировками.
– 'backslashreplace' - ошибка кодирования заменяется на escape-последовательность с обратной косой чертой. Используется только с текстовыми кодировками.

Аргумент newline (перевод_строки) – задает режим работы универсальных символов новых строк. Может иметь значения: None, ' ', '\n', '\r', '\r\n\'.

Аргумент closed – флаг закрытия файлового дескриптора. возвращает true, если файл закрыт и false в противном случае.

Аргумент opener. Функция open() возвращает файловый объект, тип которого зависит от режима используемого при открытии (рис.76). Если файл открывается:

- В текстовом режиме – функция open() возвращает объект io.TextIOWrapper.
- В двоичном режиме чтения с буферизацией – функция open() возвращает объект io.BufferedReader.
- В двоичном режиме записи с буферизацией – функция open() возвращает объект io.BufferedWriter.
- В двоичном режиме без буферизации – функция open() возвращает объект io.FileIO.

```
>>> #открываем файл в текстовом режиме
>>> open('E:\\Утес.txt', 'rt')
    <_io.TextIOWrapper name='E:\\Утес.txt' mode='rt' encoding='cp1251'>
>>>
>>> #открываем файл на чтение в двоичном режиме с буферизацией
>>> open('E:\\Утес.txt', 'rb')
    <_io.BufferedReader name='E:\\Утес.txt'>
>>>
>>> #открываем файл на запись в двоичном режиме с буферизацией
>>> open('E:\\Утес1.txt', 'wb')
    <_io.BufferedWriter name='E:\\Утес1.txt'>
>>>
>>> #открываем файл в двоичном режиме без буферизации
>>> open('E:\\Утес.txt', 'rb', 0)
    <_io.FileIO name='E:\\Утес.txt' mode='rb' closefd=True>
```

Рис.76. Режимы работы функции open

9.2. Чтение файла

Первый способ - метод read, читающий весь файл целиком, если был вызван без аргументов, и n символов, если был вызван с аргументом (целым числом n). Например:

```
a=open ('f.txt')
print (a. read(1), end='')
print (a. read())
```

Начиная с версии 2.6, Питон поддерживает протокол менеджеров контекста. Этот протокол гарантирует закрытие файла вне зависимости от того, произошло исключение в блоке кода или нет. Конструкция with:

```
with open('f.txt') as a:

    #считали в строку
    str=a.read ()

print (str)
```

Второй способ - прочитать файл построчно и сохранить в строку:

```
a=open ('f.txt')
str=' '. join(a)
```

или в список

```
#открыли файл для чтения
a=open ('f.txt','r')

#считали в строку
str=a.read()
a.close()

#строку преобразовали в список
list=str.split(' ')
```

Третий способ – прочитать файл построчно с помощью функции readline (size). Считывает из файла одну строку при каждом вызове, если size не указан.

ВАЖНО: Возвращаемая строка содержит символ перевода строки. Исключением является последняя строка. Если она не завершается этим символом. При достижении конца файла возвращается пустая строка.

Если необязательный аргумент size присутствует и неотрицателен, то метод читает строку частями по size байтов, пока не достигнет символ новой строки \n. Если size отрицателен, то считывается строка полностью. Если длина строки меньше size, то будет считана одна строка. Например:

```
a=open ('f.txt')
str=a.readline ()
```

Четвертый способ. readlines() – считывает все содержимое файла в список. Каждый элемент списка будет содержать одну строку, включая символ перевода строки:

```
f=open(r'f.txt', 'w', encoding="utf-8-sig")
f.write("Строка1\n")
f.write("Строка2\n")
f.write("Строка3\n")
f.close()
l=[]
with open (r'f.txt', 'r', encoding="utf-8") as f:
    l=f.readlines ()
print (l)
#будет напечатано
\ufeffСтрока1\n Строка2\n Строка3\n
```
Удалим символ перевода строки
```
# определим пустой список
list=[]

# откроем файл и считаем его содержимое в список
with open('f.txt', 'r') as a:
    for line in a:
    # удалим заключительный символ перехода строки
        currentPlace = line[:-1]

# добавим элемент в конец списка
        list.append(currentPlace)
print (list)
#будет напечатано
[Berlin, Moscow, London, ‵ ‵]
```
Или, если название было в отдельных строках
```
['Berlin', 'Moscow ', London', ' ']
```
Нам понадобилось удалить перенос строки в самом её конце. Здесь нам помогает то, что Python позволяет применять списочные операции к строкам. В строке 8 кода выше удаление сделано просто как операция над самой строкой, что сохраняет всё, кроме последнего элемента. Он содержит символ “\n”, обозначающий перенос строки в системах UNIX/Linux.

Пятый способ. Метод _next_() позволяет считывать построчно:
```
f=open(r'f.txt', 'w', encoding="utf-8-sig")
f.write("Строка1\n")
f.write("Строка2\n")
f.write("Строка3\n")
```

```
f=open(r'f.txt', 'r', encoding="utf-8-sig")
for line in f:
    print (line.rstrip ('\n'), end=" ")
f.close()
```
где символ перевода строки удаляется методом rstrip ().

9.3. Запись в файл

Метод write () – записывает строку или последовательность байтов в файл. Если в качестве параметров указана строка, файл должен быть открыт в текстовом режиме. Для записи последовательности байтов необходимо открыть файл в бинарном режиме.
```
list=['1','2','3']

#преобразуем список в строку
str=''.join(list)

a=open ('f.txt','w')

#запишем строку в файл
a.write (str)
a.close()
```

9.4. Другие методы работы с файлами

Метод truncate ([ ]) – обрезает файл до указанного количества символов или байтов. Например:
```
f=open(r'f.txt', 'r+', encoding="cp1251")
print (f.read())
f.truncate(3)
f.close()
with open (r'f.txt', 'r', encoding="cp1251") as f:
    print (f. read())

#будет напечатано
rrrrt
rrr
```

Метод tell () – возвращает позицию указателя относительно начала файла в виде целого числа. В Windows метод считывает сим-

вол \r (возврат каретки) как дополнительный байт, хотя это символ удаляется при открытии файла в текстовом режиме. Например:

```
with open(r'f.txt', 'w', encoding="cp1251") as f:
    f.write("String1\nString2")

f= open (r'f.txt', 'r', encoding="cp1251")
print (f. tell())
    #указатель расположен в начале файла
    #перемещаем указатель
    f.readline ()
print (f. tell())
    9 # возвращает 9=8+’\r’
```

Чтобы избежать этого несоответствия, следует открывать файл в бинарном режиме.

Метод seek() в Python, перемещает указатель в файле

Перемещает указатель чтения/записи в файле.

Синтаксис: *file.seek (<смещение>[, <позиция>])*

File - объект файла

смещение - int байтов, смещение указателя чтения/записи файла.

позиция- int, абсолютное позиционирование указателя.

Возвращаемое значение: целое число int, новая позиция указателя.

В параметре Позиция могут быть указаны следующие атрибуты из модуля io:

– io.SEEK_SET или 0 - начало файла (по умолчанию).
– io.SEEK_CUR или 1- текущая позиция указателя.
– io.SEEK_END или 2 - конец файла.

Пример:

```
import io

f=open(r'f.txt', 'rb')
# на 9 байтов от указателя
print (f.seek (9, io.SEEK_CUR))

    # -9 байтов от конца файла
print (f.seek (-9, io.SEEK_END))
    9
    7
```

9.5. Средства, напоминающие файлы

Функция open – это рабочая лошадка в большинстве операций с файлами, которые можно выполнять в языке Python. Для решения более специфичных задач Python поддерживает и другие инструментальные средства, напоминающие файлы: каналы, очереди, сокеты, файлы с доступом по ключу, хранилища объектов, файлы с доступом по дескриптору, интерфейсы к реляционным и объектно-ориентированным базам данных и многие другие. Файлы с доступом по дескриптору, например, поддерживают возможность блокировки и другие низкоуровневые операции, а сокеты представляют собой интерфейс к сетевым взаимодействиям.

9.6. Дополнительные источники по работе с файлами

1. Васильев А. Программирование на Python в примерах и задачах. –Москва: Эксмо, 2021. – 616 с.
2. Хайнеман Дж., Поллис Г., Сеяков С. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд.: Пер. с англ. – СпБ.: ООО “Альфа-книга”, 2017. – 432 с .: ил.
3. Работа с файлами. Режим доступа: https://pythonru.com/osnovy/fajly-v-python-vvod-vyvod
4. Работа с файлами в Python 3: документация. Режим доступа: https://docs.python.org/3/tutorial/inputoutput.html

Практическая работа № 10
ФАЙЛЫ

Задание

Напишите программы к заданиям, указанным в индивидуальном варианте.

Оформите отчет по практической работе, содержащий:

- титульный лист;
- содержание;
- ответ на вопросы самоконтроля.
- для каждого варианта:
  - текст задания с указанием варианта;
  - блок-схему алгоритма решения задачи;
  - плановую обработку ошибок;

- программный код на языке Python версии 3.8, соответствующий блок-схеме;
- unit-тесты;
- скриншоты результата работы программы для каждого unit-теста.

ВАРИАНТ 1. Имеется пункт взимания платежей за проезд по автостраде. Каждая проезжающая машина должна заплатить за проезд 180 рублей, часть машин платит за проезд, часть проезжает бесплатно. В кассе ведется учет числа проехавших за день машин и оплата (или неоплата) каждой машины. Внесите проверку на ошибки неправильного ввода данных. Записывайте текущие данные кассы в файл. После окончания ввода, по данным файла подсчитайте количество проехавших машин и суммарную выручку за день.

ВАРИАНТ 2. В цикле запрашивайте у пользователей данные, состоящие из имени, отчества, фамилии и номера работника. Внесите проверку на ошибки неправильного ввода данных. Записывайте данные в файл. После окончания ввода всех данных выведите на экран все данные из файла в формате таблицы, где порядок имен столбцов задан следующим образом: порядковый номер, фамилия, имя, отчество. Отсортируйте выведенные данные в алфавитном порядке.

ВАРИАНТ 3. Напишите функцию, фиксирующую время обращения пользователя (целое число часов, минут и секунд). Внесите в функцию проверку на ошибки неправильного ввода пользователем времени. Для часов диапазон составляет от 0 до 23, для минут и секунд – от 0 до 59. Записывайте введенные значения в файл. Выведите записанные значения в формате 12:59:32.

ВАРИАНТ 4. Для описания данных о цветных планшетных сканерах описать структуру (список, словарь, множество и т.д.) вида: наименование модели, цена, горизонтальный размер области сканирования, вертикальный размер области сканирования, оптическое разрешение, число градаций серого. Написать функцию, которая записывает в бинарный файл данные о сканере из приведенной структуры (6-8 записей). Структура файла: в первых двух байтах размещается количество сделанных в файл записей, далее размещаются данные о сканерах. Вывести на экран содержимое полученного файла.

ВАРИАНТ 5. Для хранения данных о цветных планшетных сканерах описать структуру из варианта 4. Написать функцию, которая записывает в бинарный файл данные о сканере из приведенной структуры (6-8 записей). Структура файла: в первых двух байтах размеща-

ется количество сделанных в файл записей, далее размещаются данные о сканерах. написать функцию, которая сортирует записи в бинарном файле по цене. Вывести на экран содержимое полученного файла.

ВАРИАНТ 6. Данные о цветных планшетных сканерах из варианта 4 хранятся в текстовом файле. Написать функцию, которая записывает содержимое файла в структуру, описанного в варианте 4 вида. Вывести на экран данные только тех сканеров, число градаций серого которых больше 8. Если таких сканеров нет, вывести соответствующее сообщение.

ВАРИАНТ 7. Данные о цветных планшетных сканерах из варианта 4 хранятся в текстовом файле. Написать функцию, которая читает из файла данные только для тех сканеров, цена которых находится в заданном вами диапазоне.

ВАРИАНТ 8. Определить, какой символ чаще других встречается в данном файле.

ВАРИАНТ 9. Определить, сколько строк, состоящих из одного, двух, трех и т.д. символов, содержится в данном файле. Считать, что длина каждой строки – не более 30 символов.

ВАРИАНТ 10. Даны два непустых файла. Определить номер строки и номер первого символа в этой строке, отличающего содержимое одного файла от другого. Если содержимое файлов полностью совпадает или один файл является началом другого, то вывести соответствующие сообщения.

ВАРИАНТ 11. В файле записана непустая последовательность целых чисел (целое число – это последовательность десятичных цифр, возможно начинающаяся знаком + или -). Определить, сколько четных положительных чисел содержится в файле.

ВАРИАНТ 12. Создать файл, являющийся результатом конкатенации (слияния) других файлов. Имя файла-результата и имена соединяемых файлов задаются пользователем.

ВАРИАНТ 13. Дан файл f. Создать файл newf, полученный из файла f заменой всех его прописных букв соответствующими строчными.

ВАРИАНТ 14. Дан файл и две строки. Все вхождения первой строки в файл (в том числе в качестве подстроки) заменить второй строкой.

ВАРИАНТ 15. В файле записана непустая последовательность целых чисел (целое число – это последовательность десятичных

цифр, начинающаяся знаком + или -). Создать новый файл, где все отрицательные числа заменены нулем.

ВАРИАНТ 16. В файле записана непустая последовательность целых чисел, являющихся числами Фибоначчи. Приписать еще n чисел Фибоначчи. Число n спросить у пользователя (предварительно сообщив, сколько чисел уже имеется в файле).

ВАРИАНТ 17. Считая, что файл разбит на строки, длина каждой из которых не превосходит 50 символов, написать программу, которая, дополняя короткие строки исходного файла пробелами справа, формирует новый файл, все строки в котором имеют длину 50.

ВАРИАНТ 18. Написать программу, которая, игнорируя исходное деление файла на строки, переформатирует его, разбивая строки так, чтобы каждая строка оканчивалась точкой, либо содержала ровно 60 литер, если среди них нет точки. Результат записать в другой файл.

ВАРИАНТ 19. В файле записаны числа. Создать новый файл, содержащий длины всех убывающих подпоследовательностей элементов исходного файла (длина – количество элементов в последовательности). Напр., исходный файл 1.7 4.5 3.4 2.2 8.5 1.2, результат – 3 2, где 3 – длина последовательности 4.5 3.4 2.2, а 2 - длина последовательности 8.5 1.2.

ВАРИАНТ 20. В файле записана непустая последовательность целых чисел. Определить, сколько чисел этой последовательности являются точными квадратами, и «подчеркнуть» их в данном файле (т.е. поставить минусы в соответствующих позициях следующей строки).

Вопросы для самоконтроля к главе 9

1. Что такое файл?
2. Какие типы файлов поддерживает Python? Дайте краткую характеристику каждому типу.
3. Как открыть файл?
4. Перечислите основные режимы открытия файла. Чем они отличаются друг от друга?
5. Для чего необходимо создание специального объекта файл?
6. Как открыть файл?
7. Какие способы чтения файла предоставляет Python? Дайте краткую характеристику каждому из способов.
8. Какие способы записи в файл реализованы в Python? Дайте краткую характеристику каждому из способов.

9. Какие способы закрытия файла реализованы в Python? Дайте краткую характеристику каждому из способов.

Тест самоконтроля к главе 9

1. Для открытия файла используется метод…

а) open (); б) close (); в) write (); г) read ().

2. Режимом «только для чтения» является:

а)wr; б) r; в) rb; г) r+; д) w; е) wb.

3. Режимом «только для записи» является:

а)wr; б) r; в) rb; г) r+; д) w; е) wb.

4. Каким из указанных ниже методов файл открывается для чтения одной строки?

а) rewrite; б) add; в) readline; г) read; д) rename.

5. Переименование файла в Python осуществляет ….

а) rewrite; б) add; в) readline; г) read; д) rename.

6. В Python узнать текущую позицию указателя в файле можно с помощью функции:

а) rewrite; б) tell; в) seek; г) read; д) rename.

7. В Python изменить текущую позицию указателя в файле можно с помощью функции:

а) rewrite; б) tell; в) seek; г) read; д) rename.

8. Каким из указанных ниже методов файл открывается для построчного чтения?

а) rewrite; б) add; в) readline; г) readlines; д) rename.

9. Каким из указанных ниже методов файл открывается для добавления строки?

а) writelines; б) add; в) readline; г) readlines; д) write.

10. Каким из указанных ниже методов файл открывается для добавления строк?

а) writelines; б) add; в) readline; г) readlines; д) write.

Глава 10. ФУНКЦИИ И ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

С понятием пользовательской функции мы познакомились в главе 3, изучая способы тестирования программного кода.

Кроме этого, на протяжении всей книги мы постоянно пользовались встроенными библиотечными функциями, такими как print, input, sort, max, min и т.п., для решения различных задач.

Теперь пришло время больше узнать о функциях.

Функцией в программировании называется последовательность инструкций, которая выполняет вычисления. С чем можно сравнить функцию? Направшивается аналогия с «черным ящиком», когда мы знаем, что поступает на вход и что при этом получается на выходе, а внутренности «черного ящика» от нас скрыты. Он получает на вход пластиковую карточку, пин-код и денежную сумму, на выходе мы ожидаем получить запрашиваемую денежную сумму. Нас не очень интересует принцип работы банкомата до тех пор, пока он работает без сбоев.

Примером механизма работы функции как «черного ящика» может служить встроенная встроеная библиотечная функция вычисления модуля числа *abs()* (рис.77).

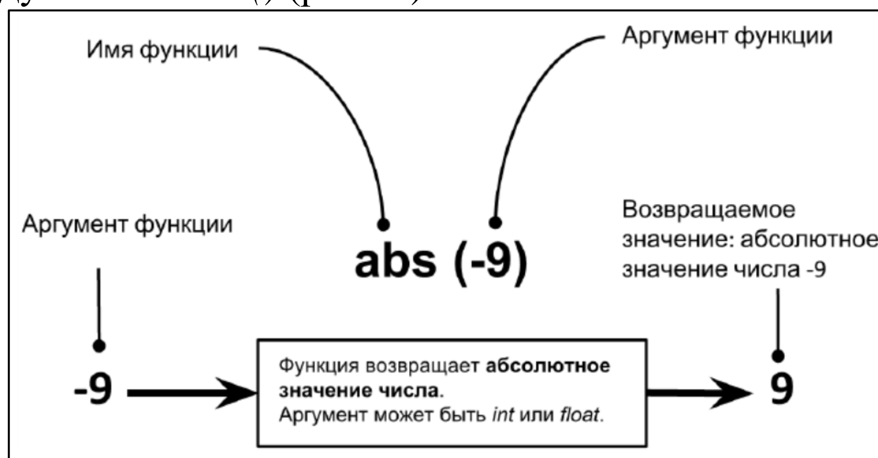


Рис.77. Механизм работы функции

Функция принимает на вход один аргумент – объект числового типа и возвращает абсолютное значение этого объекта. Механизм реализации преобразования скрыт от нас.

В этой главе рассмотрим ряд тем, имеющих непосредственное отношение к созданию и использованию функций.

10.1. Объявление и вызов функции

Функцией называется блок программного кода, у которого есть имя и который можно выполнить, вызвав функцию в нужном месте программного кода.

Прежде чем функцию вызывать, ее следует описать – т.е. создать блок кода, который будет выполняться при вызове функции.

Вы уже сталкивались с объявлением функции, вспомним, как это делается. В самом простом случае при объявлении функции указывается ключевое слово def, после которого следует имя функции, в круглых скобках перечисляются аргументы функции, затем ставится двоеточие и далее следует блок, который содержит описание функции:

```
def имя (аргументы):
    #описание функции
```

Если у функции нет аргументов, то после имени функции указываются пустые круглые скобки. Функция может возвращать значение, а может и не возвращать.

В случае, если функция возвращает значение, мы используем инструкцию return, после которой указываем возвращаемое значение. Например:

```
def calculate_sum (a,b):
  sum = a+b
  return sum
# вызов функции в основной программе
print(calculate_sum(2,3))
#будет выведено 5
```

В нашем примере переменная sum получает свое значение в теле функции. Такие переменные называются локальными. Они доступны только в теле функции. Поэтому если в двух разных функциях используются переменные с одинаковыми именами, то на самом деле это разные переменные. Каждая из них видима и доступна только в своей функции. Это же правило относится и к аргументам функций.

Помимо локальных, в функциях могут использоваться и глобальные переменные. Это переменные, объявленные за пределами всех функций. Областью видимости глобальных переменных является вся программа. В примере ниже глобальная переменная name является аргументом двух разных функций, а переменная s, являясь локальной, получает разные значения в каждой из функций:

```
def next_day (name):
    s='Hello!'+name
    return s
def answer (name):
    s=name+', давай дружить'
    return s
name=input()
print (next_day(name))
print (answer (name))
#будет выведено
Андрей
Hello! Андрей
Андрей, давай дружить
```

Тип возвращаемого результата определяется в процессе выполнения функции. Это означает, если функции передаются разные аргументы, то функция может возвращать значения разных типов. Такая особенность функций Python имеет последствия, и ее следует принимать в расчет при работе с функциями.

Результат функции не обязательно возвращается инструкцией return. Например, есть так называемые функции-генераторы, в теле которых для формирования результата используется инструкция yield. Функции-генераторы будут рассмотрены ниже в параграфе 11.1.2.

Если функция не возвращает значения, то при вызове функции просто выполняется последовательность команд. В примере ниже функции input и next_day не содержит аргументов, функция next_day ничего не возвращает:

```
def next_day ():
    print ('Hello!')
name=input()
print (name, end=' ')
next_day()
```

Аргументами функции могут быть не только переменные базовых типов, но и сложные типы данных, такие как, списки, кортежи, множества, словари, файлы, объекты классов и сами функции.

В случае, если аргументом функции является список, кортеж, множество или словарь, в качестве аргумента функции достаточно указать его имя. Например, функция summa вычисляет сумму двух элементов списка l:

```
l=[1,2,3,4,5]
def summa (l):
    s=l[1]+l[3]
    return s
print (summa(l))
```

В примере ниже функция summa вычисляет сумму значений элементов словаря:

```
l={'q':1,'w':2,'e':3,'r':4,'t':5}
def summa (l):
    s=0
    for i in l.values():
        s=s+i
    return s
print (summa(l))
```

Функция summa вычисляет сумму чисел, хранящихся в файле f.txt:

```
f=open('f.txt')
def summa (f):
    s=0
    for line in f:
        q=int(line)
        s+=q
    return s
print (summa(f))
```

Удвоим значение полученной в примере выше суммы чисел с помощью функции pv, аргументом которой является функция summa:

```
f=open('f.txt')
def summa (f):
    s=0
    for line in f:
        q=int(line)
        s+=q
    return s
def pv (summa):
    t=2
    t*=summa
    return t
print (pv(summa(f)))
```

Пример функций, аргументами которых являются объекты класса, представлен ниже на классах Smallobj и Sm. В программе создает-

ся две функции. Первая является конструктором и задает значения поля класса, вторая show () предназначена для отображения содержимого поля класса:

```
class Smallobj:
     def __init__(self, data):
        """конструктор, инициализирующий открытое
поле somedata значением"""
        self.somedata=data
     def show (self):
        print (self.somedata)
#создадим дочерний класс
class Sm(Smallobj):
     pass
#основная программа
s1=Smallobj (2)
print (s1.somedata)
s1.show()
#создаем экземпляр дочернего класса и пользуемся
методами родительского класса
s2=Sm (3)
s2.show()
```

Во всех рассмотренных выше примерах функции вызывались привычной для нас инструкцией. В ней вызывается функция, а аргументы, которые ей передаются, указаны в круглых скобках через запятую. Это так называемая позиционная передача аргументов, или передача аргументов по позиции: порядок передачи аргументов при вызове функции соответствует порядку, в котором аргументы объявлялись в описании функции.

Есть иной способ передачи аргументов, когда для аргумента явно указывается название и значение. Такой способ называется передачей по имени, или передачей по ключу, а соответствующий аргумент иногда называют именованным. Пример такого способа передачи аргументов дает, например, команда:

```
show(second="B", third="C", first="A").
```

10.2. Рекурсия. Рекурсивные функции

Рекурсией называется прямое или косвенное обращение функции (подпрограммы) к самой себе. Функция, которая вызывает саму себя, называется *рекурсивной*.

Рассмотрим пример вычисления факториала функции:

$$F(n) = \begin{cases} 1, \text{если } n = 0; \\ n \cdot F(n-1), \text{если } n > 0. \end{cases}$$

Рекурсивная функция может быть представлена на языке Python следующим образом:

```
def F(n):
if n==0: return 1
if n>0: return n*F(n-1)
#основная программа
print ('введите натуральное число\n')
n=int (input())
print (F(n))
```

Вызов F(7) влечет за собой ряд обращений к функции F с фактическими параметрами 6, 5, …, 2, 1, 0.

При вызове F(0) вычисляется непосредственное значение функции, что останавливает процесс повторений; после этого следует процесс возврата и вычисление значений функции F для аргументов 1, 2, …, 6, 7, последнее вычисленное значение F(7) возвращается на место первого вызова функции (рис.78).

```
F(7) -> F(6) -> F(5) -> ... -> F(1) -> F(0).
```

Рис.78. Порядок обращения функции при рекурсивном вызове

Из данного примера видно, что рекурсия используется для того, чтобы запрограммировать повторяющиеся вычисления. Повторение осуществляется с помощью подпрограммы, внутри которой есть вызов самой себя: когда процесс выполнения программы достигает соответствующего места, вызывается новое выполнение данной процедуры.

Очевидно, любая рекурсивная функция должна содержать условие остановки процесса повторения. Например, в случае факториала процесс повторений останавливается, когда n принимает значение 0;

При любом вызове рекурсивной функции в память компьютера заносится следующая информация:

1. текущие значения параметров, передаваемых через значение;
2. местонахождение (адреса) параметров-переменных;
3. адрес возврата, т. е. адрес оператора, который следует за оператором вызова.

Таким образом, при рекурсивном вызове занимаемая область памяти увеличивается очень быстро, что ведет к риску переполнения

памяти компьютера. Этого можно избежать путем замены рекурсии на итерацию (операторы for, while, repeat).

В качестве примера представим не рекурсивную форму функции для вычисления факториала:

```
    def F_not_recursive (n):
    p = 1
    for i in range (1,n+1):
        p = p * i
    return p

#основная программа
print ('введите натуральное число\n')
n=int (input())

print (F_not_recursive(n))
```

Рекурсивно или итеративно?

Каковы же преимущества рекурсивных функций? Можно ли с помощью итеративных получить тот же результат? Когда лучше использовать одни, а когда — другие?

Важно учитывать временную и пространственную сложности. Рекурсивные функции занимают больше места в памяти по сравнению с итеративными из-за постоянного добавления новых слоев в стек памяти. Однако их производительность куда выше.

Рекурсия может быть медленной, если реализована неправильно. Из-за этого вычисления будут происходить чаще, чем требуется.

Написание итеративных функций зачастую требуется большего количества кода. Например, пример функции для вычисления факториала, но с итеративным подходом. Выглядит не так изящно, не правда ли?

Рекурсию необходимо использовать в случаях, когда написание не рекурсивных алгоритмов является очень сложным.

10.3. Функции-генераторы

Очевидный плюс использования функций-генераторов — они в оперативной памяти не хранят все объекты, которые используют. Есть отложенное выполнение. Такая функция возобновляет работу там, где остановилась. Она генерирует значения, а не вычисляет их. Чтобы передать аргументы генератору и вернуть значения, применяют ключевое слово yield. Когда элементы объекта-генератора закан-

чиваются, возбуждается исключение типа StopIteration. Общий принцип работы функций этого типа представлен в примере ниже:

```
def gen_demo(times, var):
    for _ in range(times):
        yield var

gen_inst = gen_demo(25, "Gen demo text")

for _ in range(26):
    print(next(gen_inst))
#будет напечатано
Gen demo text
Gen demo text
. . . . . . .
Gen demo text
StopIteration
```

10.4. Функции с переменным числом аргументов

В Python можно передать переменное количество аргументов в функцию двумя способами:

- *args для позиционных аргументов;
- **kwargs для именованных аргументов.

Рассмотрим первый способ – *args.

Символ '*' перед именем аргумента означает, что при вызове функции, на место этого параметра передается список аргументов, передаваемых как кортеж и доступных внутри функции под тем же именем, что и имя параметра, только без символа '*'.

Например, использование *nums в качестве параметра позволяет последовательно передать 2, 4 и 5 аргументов в функцию adder (). Внутри функции мы проходимся в цикле по этим аргументам, чтобы вычислить их сумму, и вывести результат:

```
def adder(*nums):
    sum = 0
    for n in nums:
        sum += n
    print("Sum: ", sum)
#основная программа
adder(3, 5)
adder(4, 5, 6, 7)
```

```
adder(1, 2, 3, 5, 6)
#будет выведено
Sum:  8
Sum:  22
Sum:  17
```

Второй способ – **kwargs. Например:

```
def intro(**data):
    print("\nData type of argument: ",type(data))

    for key, value in data.items():
        print("{} is {}".format(key, value))

intro(Firstname="Sita", Lastname="Sharma", Age=22,
Phone=1234567890)
intro(Firstname="John", Lastname="Wood",
Email="johnwood@nomail.com", Country="Wakanda",
Age=25, Phone=9876543210)
```

При запуске программы будет выведено:

```
Data type of argument:  <class 'dict'>
Firstname is Sita
Lastname is Sharma
Age is 22
Phone is 1234567890
Data type of argument:  <class 'dict'>
Firstname is John
Lastname is Wood
Email is johnwood@nomail.com
Country is Wakanda
Age is 25
Phone is 9876543210
```

В этом случае в функцию были переданы два словаря разной длины. Затем с помощью цикла было выведено их содержимое.

Если вы хотите применить оба способа одновременно, используйте синтаксис следующего вида: `func(fargs, *args, **kwargs)`, порядок следования аргументов важен.

10.5. Значения аргументов функции "по-умолчанию"

В функциях иногда полезно указывать значение по умолчанию для одного или нескольких параметров. В этом случае соответствующий аргумент может быть опущен при вызове. Это позволяет вызывать функцию с меньшим количеством аргументов, чем предписано. Например:

```
def ask_ok(prompt, retries=4, reminder='Попробуйте
еще раз!'):
    while True:
        ok = input(prompt)
        if ok in ('ok', 'y', 'yes'):
            return True
        if ok in ('n', 'no'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('Неверный ответ поль-
зователя')
        print(reminder)
```

Вызовем функцию ask_ok несколькими способами.

Первый способ – укажем в качестве аргумента только один параметр `prompt`, в качестве ответа введем сначала `OK`, потом `no`:

```
ask_ok('Хочешь продолжить?')
#будет выведено
  Хочешь продолжить? ok
  Попробуйте еще раз!
  Хочешь продолжить? no
  Process finished with exit code 0
```

Второй способ – укажем в качестве аргументов два параметра `retries и reminder`:

```
ask_ok( 2, 'Вводите только "ok" или "no"!')
#будет выведено
2
```

10.6. Лямбда-функции

Название функции можно рассматривать как некоторую «переменную», которая ссылается на объект функции. Возникает вопрос, можно ли получить собственно функцию, не привязывая ее к какому-то конкретному имени? Такая возможность существует. Состоит она

в том, чтобы использовать анонимные функции. Такого рода функции получили название лямбда-функций.

Синтаксис описания лямбда-функций следующий: указывается ключевое слово lambda, после которого перечисляются аргументы, ставится двоеточие и в той же строке – значение, которое функция возвращает:

```
lambda аргументы: результат
```

Лямбда-функции могут иметь любое количество аргументов, но у каждой может быть только одно выражение. Выражение вычисляется и возвращается. Аргументы перечисляются через запятую. Если аргументов нет, они просто пропускаются. В результате создается объект для анонимной функции. Значением лямбда-функции является ссылка на этот объект. Такую ссылку в качестве значения можно присвоить переменной, после чего эта переменная сама «станет» функцией. Рассмотрим примеры.

Выведем на экран с помощью лямбда-функции все нечетные числа на интервале от 1 до 20:

```
L=lambda n: 2*n+1
for n in range (10):
    print ("нечетные числа:", L(n))
 #будет выведено
нечетные числа: 1
нечетные числа: 3
нечетные числа: 5
нечетные числа: 7
нечетные числа: 9
нечетные числа: 11
нечетные числа: 13
нечетные числа: 15
нечетные числа: 17
нечетные числа: 19
```

В вышеуказанном коде `L=lambda n: 2*n+1` — это лямбда-функция. Здесь n — это аргумент, а $2 \cdot n + 1$ — это выражение, которое вычисляется и возвращается. Эта функция безымянная. Она возвращает функциональный объект с идентификатором L. Сейчас мы можем считать её обычной функцией. Инструкция `L=lambda n: 2*n+1` эквивалентна:

```
def L(n):
    return 2*n+1
```

```
for n in range(0,10):
    print ("нечетные числа:", L(n))
```

10.7. Вложенные функции

Вложенные функции – это функции, которые мы определяем внутри других функций. В Python такая функция имеет прямой доступ к переменным и именам, определенным во включающей её функции.

Пример создания функции с двумя внутренними функциями:

```
def parent():
    def first_child():
        print("Привет из функции first_child().")
    def second_child():
        print("Привет из функции second_child().")
    second_child()
    first_child()
parent ()
#будет выведено
Привет из функции second_child().
Привет из функции first_child().
```

В этом коде мы определяем две функции first_child() и second_child() внутри функции parent().

Для вывода на экран строк "Привет из функции first_child()." и "Привет из функции second_child()." мы вызываем функции first_child() и second_child() в последней строке функции parent ().

Из приведенного примера видна основная особенность внутренних функций — их способность обращаться к переменным и объектам из включающей («внешней») функции. Другими словами, включающая функция предоставляет пространство имен, доступное вложенным в нее функциям.

Приведем еще один пример:

```
def outer_func(who):
    def inner_func():
        print(f"Hello, {who}")
    inner_func()
who=input("ВВедите имя: ")
outer_func(who)
#будет напечатано
Введите имя: Борис
Hello, Борис
```

С помощью переменной who мы передаем строку в качестве аргумента функции outer_func(). Функция inner_func() будет обращаться к этому аргументу через имя who. Хотя это имя определяется в локальной области видимости outer_func(), с точки зрения inner_func() оно является глобальным.

Применение вложенных функций в Python. Функции высших порядков

Инкапсуляция. В этом случае внутренние функции используются для защиты или полного сокрытия функции из области глобальной видимости. Рассмотрим поясняющий пример:

```
def increment(number):
    def inner_increment():
        return number + 1
    return inner_increment()
print(increment(10))
#будет выведено
11
```

Вызовем вложенную функцию inner_increment()

```
print(inner_increment())
#будет выведено
NameError: name 'inner_increment' is not defined
```

У нас нет прямого доступа к вложенной функции inner_increment(). Попытавшись обратиться к ней, мы получаем ошибку NameError. Функция increment() полностью скрывает функцию inner_increment(), предотвращая доступ из глобальной области, т.е. функция не может быть вызвана нигде, кроме включающей ее функции.

Другим распространенным вариантом применения вложенных функций являются **внутренние вспомогательные функции**.

Потребность в них возникает, когда нам нужна функция, выполняющая один и тот же фрагмент кода в нескольких местах своего тела. Например, мы хотим написать функцию для обработки файла CSV, содержащего информацию о точках доступа Wi-Fi во городе Владимире. Программный код ниже позволяет узнать общее количество точек доступа, а также информацию о компании, которая их предоставляет:

```
import csv
from collections import Counter
```

```
def process_hotspots(file):
    def most_common_provider(file_obj):
        hotspots = []
        with file_obj as csv_file:
            content = csv.DictReader(csv_file)
            for row in content:
                hotspots.append(row["Provider"])
        counter = Counter(hotspots)
        print(
f"Во Владимире {len(hotspots)} точек Wi-Fi.\n"

f"{counter.most_common(1)[0][1]} из них предостав-
ляет"
f"{counter.most_common(1)[0][0]}."
)
    if isinstance(file, str):
        # Получаем путь к файлу
        file_obj = open(file, "r")
        most_common_provider(file_obj)
    else:
        # Забираем объект файла
        most common provider(file)
```

Функция process_hotspots() принимает аргумент file и проверяет, является ли файл строковым путем к физическому файлу или файловым объектом. Затем функция process_hotspots() вызывает вспомогательную внутреннюю функцию most_common_provider(), которая принимает файловый объект и выполняет следующие операции:

- считывает содержимое файла в генератор, который создает словари с помощью метода csv.dictreader;
- составляет список провайдеров wi-fi;
- подсчитывает количество точек доступа wi-fi для каждого поставщика с помощью объекта collections.counter.
- печатает сообщение с полученной информацией.

Запустив функцию, мы получим следующий результат:

```
from hotspots import process_hotspots

file_obj = open("./NYC_Wi-
Fi_Hotspot_Locations.csv", "r")
```

```
process_hotspots(file_obj)
Во Владимире 3319 точек Wi-Fi.
1868 из них предоставляет LinkNYC - Citybridge.

process_hotspots("./NYC_Wi-
Fi_Hotspot_Locations.csv")
Во Владимире 3319 точек Wi-Fi.
1868 из них предоставляет LinkNYC - Citybridge.
```

Из примера видно, - независимо от того, вызываем ли мы функцию process_hotspots() со строковым путем к файлу или с файловым объектом, получается один и тот же результат.

Следующий пример использования вложенных функций – **замыкания** – динамически создаваемые функции, возвращаемые другими функциями. Замыкание «заставляет» вложенную функцию при ее вызове сохранять состояние своего окружения. То есть замыкание – это не только сама внутренняя функция, но и окружающая ее среда.

Чтобы определить замыкание, необходимо:

1. создать вложенную функцию;
2. сослаться на переменные из включающей функции;
3. вернуть вложенную функцию.

Рассмотрим пример:

```
def generate_power(exponent):
    def power(base):
        return base ** exponent
    return power
raise_two = generate_power(2)
raise_three = generate_power(3)
print (raise_two(4))
print (raise_two(5))
print (raise_three(4))
print (raise_three(5))
#будет напечатано
16
25
64
125
```

В примере мы определяем функцию generate_power(), которая представляет собой настоящую фабрику для создания замыканий. Эта функция при каждом вызове создает и возвращает новую функцию-замыкание. Функция power(), которая является внутренней функцией,

принимает единственный аргумент base и возвращает результат выражения base ** exponent. Последняя строка возвращает функцию power как функциональный объект, не вызывая его.

Откуда функция power() получает значение показателя степени exponent? Вот где в игру вступает замыкание. В этом примере power() получает значение экспоненты из внешней функции generate_power(). Когда мы вызываем generate_power(), Python:

1. определяет новый экземпляр – функцию power() – который принимает аргумент base;
2. делает «снимок» окружения power(). В данном случае это текущее значение переменной exponent;
3. Возвращает в качестве результата функцию power() вместе с окружением.

Таким образом, когда мы вызываем экземпляр power(), возвращаемый функцией generate_power(), функция generate_power() запоминает значение степени exponent.

Еще одно применение вложенных функций – **декораторы**.

Декоратор – это функция, которая принимает в качестве аргумента другую функцию и изменяет ее поведение, не изменяя ее программный код. Рассмотрим пример:

```
def func ():
    print ('Это функция func, которую декорируют')

def decorator_function(func):
    def wrapper():
        print('Функция-декоратор!')
        print('Декорируемая функция:
{}'.format(func))
        print('Выполняем задекорированную функцию
func')
        func()
        print('Выходим из обёртки')
    return wrapper
func = decorator_function(func)
func()

#будет напечатано
Функция-декоратор!
```

```
Декорируемая функция: <function func at
0x000001F72A93C940>
Выполняем задекорированную функцию func
Это функция func, которую декорируют
Выходим из обёртки
```

В рассматриваемом примере функция decorator_function() является декоратором. Это функция высшего порядка, так как принимает в качестве аргумента другую функцию func() и возвращает в качестве результата функцию wrapper().

В теле декоратора функция wrapper() выполняет роль «обертки» для функции fun(), изменяет её поведение, не изменяя ее программного кода. Декоратор возвращает эту обёртку, теперь переменная func указывает на внутреннюю функцию wrapper(), которая содержит ссылку на оригинал - функцию fun(), и вызывает эту функцию между двумя вызовами print().

10.8. Как сделать вызов функции из другого файла в Python

Со временем в вашем проекте будет много функций. Эти функции желательно хранить отдельно от основной программы и вызывать, когда нужно.

Сделать это можно несколькими способами. Общий вид команды:

```
from имя файла (или пакета) import …
```

Если проект небольшой, можно все функции сохранить в одном отдельном файле.

Если вам нужно вызвать все функции, вы импортируете весь файл, иначе – указываете конкретные функции.

Продемонстрируем сказанное на примерах.

Пусть мы имеем две функции, вычисляющие : sum () – сумму двух чисел, product () – произведение двух чисел. Обе функции хранятся отдельно от основной программы в файле *pr2.py*:

```
#содержимое файла pr2.py
def summ(a, b):
        return a + b

def product(a, b):
        return a * b
```

Код основной программы сохранен в файле *main.py*. Первой строкой с помощью конструкции *from pr2 import* * мы подключаем все функции файла *pr2*:

```
from pr2 import *

print ('input a b')
a=int (input())
b=int (input())

print ('1-summa, 2 - product')
n=int (input())

if n==1: print (summ(a,b))
if n==2: print (product(a,b))
```

Допустим, для вычислений нам нужна только одна функция sum (). В этом случае первая строка программы файла *main.py* будет выглядеть:

```
from pr2 import summ
```

Добавим в наш проект функцию *val ()* с помощью которой будет задавать значения двух переменных *a,b*. Сохраним ее в файле *inVal.py* подкаталога *vvod*. Наши две функции сохраним в файле cal.py пакета *calculators*.

Для импорта, каталог, в котором лежит файл, должен быть пакетом (*package*), то есть содержать в себе файл ***init****.py*.

Создадим пакеты. Для этого последовательно выполните следующие действия: подведите курсор к имени проекта *pr2_1* и нажмите правую кнопку мыши. В появившемся меню выберите *Python Package*, задайте имя пакета. (рис.79)

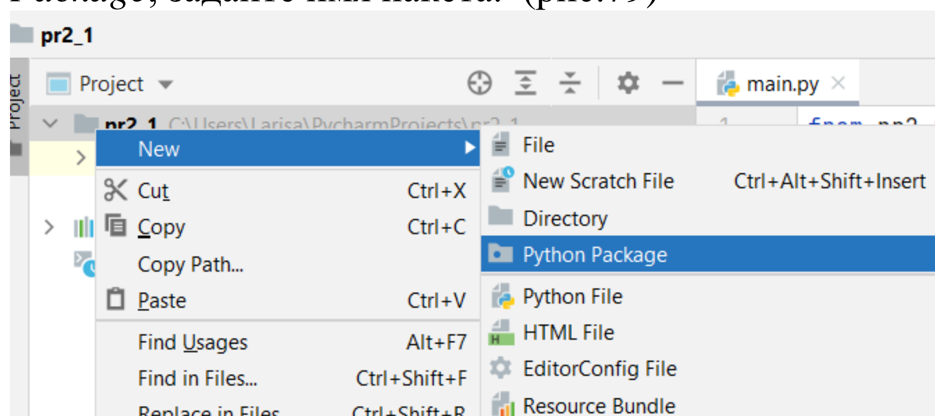


Рис.79. Процесс создания пакета

Для создания файла встаньте курсором на имя пакета, нажмите правую кнопку мыши, в появившемся меню выберите Python File. Задайте имя файла (рис.80)

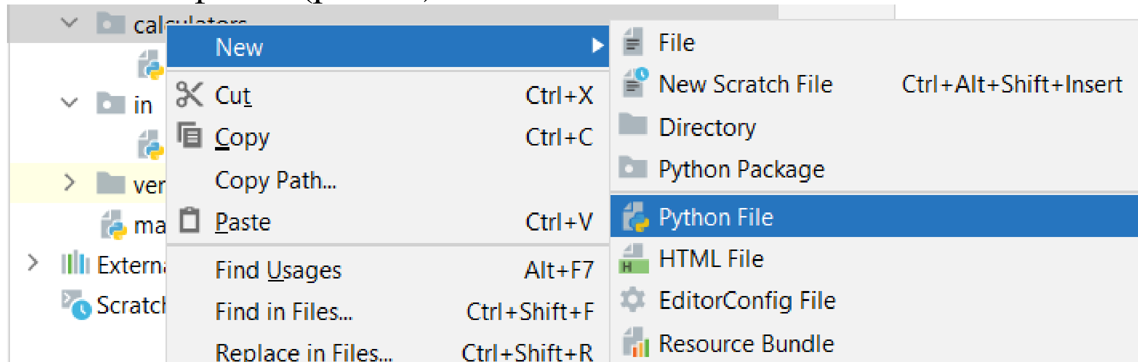


Рис.80. Создание файла в пакете

Структура проекта pr2_1 представлена на рис.81.

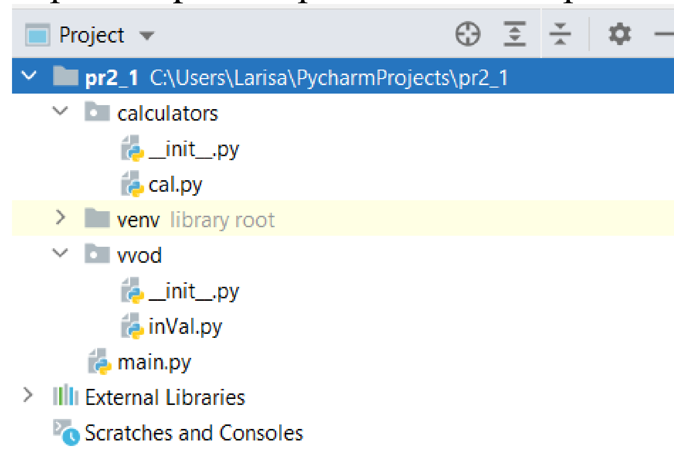


Рис.81. Структура проекта pr2_1

Содержимое файла *inVal.py* пакета *vvod*:

```
def val ():
    print ('input a b')
    a=int (input())
    b=int (input())
    return a,b
```

Содержимое файла *cal.py* пакета *calculators*:

```
def summ(a, b):
    return a + b

def product(a, b):
    return a * b
```

Содержимое файла *main.py*

```
from vvod import inVal
from calculators import cal
m,c=inVal.val ()
```

```
print ('1-summa, 2 - product')
n=int (input())

if n==1: print (cal.summ(m,c))
if n==2: print (cal.product(m,c))
```

Практическая работа № 11
ФУНКЦИИ

Задание. Решите задачу индивидуального варианта с помощью различных видов функций.

Оформите отчет по практической работе, содержащий:

- титульный лист;
- содержание;
- для каждого варианта:
  - текст задания с указанием варианта;
  - программный код на языке Python версии 3.8, содержащий функции, группированные по разным пакетам. Наличие в программном коде плановой обработки ошибок обязательно;
  - unit-тесты;
  - оценка временной сложности алгоритма решения задачи;
  - скриншоты результата работы программы для каждого unit-теста.

ВАРИАНТЫ ЗАДАНИЙ

№	Вариант		Вариант
1.	Дан список целых чисел. Напишите программу, в которой описывается лямбда-функция, удваивающая значения элементов списка.	2.	Напишите программу, в которой используется функция-генератор, создающая итерируемый объект с названиями месяцев.
3.	Дан список целых чисел. Создайте лямбда-функцию(ии), которая(ые) ищет в списке положительные числа и сохраняет их в новом списке.	4.	Определите функцию, вычисляющую количество дней в том или ином месяце. Учесть два случая: заданный год является високосным, заданный год не является

			високосным.
5.	С помощью функции-генератора напишите программу «Детектор полиндромов», определяющую, является ли число палиндромом.	6.	С помощью функции-генератора получить все шестизначные счастливые номера. Счастливым называется такое шестизначное число, у которог сумма первых трех цифр равна сумме его последних трех цифр.
7.	Создайте функцию, имитирующую работу простейшего калькулятора. Каждое арифметическое действие оформите в виде вложенной функции.	8.	С помощью функции напишите программу, которая будет выводить нечетные числа из списка и остановится, если встретит число 19.
9.	Напишите программу с функцией, которая в качестве аргументов получает ссылку на другую функцию и два целых числа. В качестве результата функция возвращает наибольшее значение переданной первым аргументом функции в целочисленных точках диапазона (границы которого определяются вторым и третьим аргументом).	10.	Напишите программу, в которой описывается функция с произвольным количеством числовых аргументов. В качестве результата возвращается список из трех элементов: среднее значение аргументов, максимальное значение среди аргументов и минимальное значение среди аргументов.
11.	Даны два числа. С помощью лямбда-функции напишите одну функции с разными аргументами, удваивающую и утраивающую значения этих чисел.	12.	Даны два числа. Напишите функцию, содержащую в себе две функции, одна из которых удваивает, другая утраивает значения этих чисел.
13.	Напишите программу, в которой используется функция-генератор, создающая итерируемый объект со степенями двойки. Количество элементов определяется аргументом	14.	Дан файл, содержащий целые числа. С помощью функции-генератора считайте данные из файла в список.

	функции-генератора.		
15.	С помощью встроенной функции range() создайте список [18, 14, 10, 6, 2].	16.	С помощью функции-генератора напишите программу «Детектор полиндромов», определяющей является ли слово полиндромом.
17.	Дан список целых чисел. С помощью лямбда-функции отберите из списка четные числа.	18.	Напишите функции-генераторы, реализующие поток чисел в заданном интервале и суммирующие сгенерированные числа.
19.	Рассчитайте значение выражения: $y = \frac{1+sin1}{3} + \frac{5+sin5}{3} + \frac{3+sin3}{3}$. Однотипные действия выполните с помощью функции.	20.	Даны три квадратных уравнения: $\begin{cases} ax^2 + bx + c = 0 \\ bx^2 + ax + c = 0 \\ cx^2 + ax + b = 0 \end{cases}$ Сколько из них имеет вещественные корни? Определите функцию, позволяющую распознавать наличие вещественных корней в квадратном уравнении.

Практическая работа № 12
РЕКУРСИЯ

Задание. Решите задачу индивидуального варианта двумя способами: итерацией и рекурсией.

Оформите отчет по практической работе, содержащий:

- титульный лист;
- содержание;
- для каждого варианта:
  - текст задания с указанием варианта;
  - программный код на языке Python версии 3.8, содержащий функции, группированные по разным пакетам. Наличие в программном коде плановой обработки ошибок обязательно;
  - Наличие в программном коде плановой обработки ошибок обязательно;

- unit-тесты;
- оценка временной сложности алгоритма решения задачи (для каждого способа);
- скриншоты результата работы программы для каждого unit-теста;
- вывод, какой способ решения задачи более оптимален по времени выполнения.

№	Вариант		Вариант
1.	Рассмотреть последовательность, образованную дробями $\frac{1}{1}, \frac{2}{1}, \frac{3}{2}, \ldots$, Определить формулу(ы) по которой вычисляется следующее число. Найти первый член такой последовательности, который отличается от предыдущего не более чем на 0,001.	2.	Напишите программу для вычисления суммы заданных положительных целых чисел a и b без прямого использования оператора +.
3.	Найти все натуральные числа, не превосходящие N, сумма цифр каждого из которых в некоторой степени дает это число (например: $7^4 = 2\,401$, $8^3 = 512$, $9^2 = 81$ и т. д.).	4.	Дана последовательность чисел 1, 2, 5, 12, 29, 70, … Определить формулу(ы) по которой вычисляется следующий член последовательности. Выяснить, входит ли заданное число a в последовательность.
5.	Вывести N автоморфных чисел (числа, совпадающие с младшими цифрами своего квадрата, например: 25 и 625).	6.	Получить N троек чисел Пифагора (натуральные числа a, b и c называются числами Пифагора, если выполняется условие $a^2 + b^2 = c^2$).
7.	Дана последовательность: 1; 11; 21; 1112; 3112; 211213; 312213; 212223; 114213; … Определите формулу, по которой получаются значения элементов последовательности. Выведите первых тридцать членов этой последовательности.	8.	Сколько слов длины 5 можно составить из букв a, b и c так, чтобы буквы a и b не стояли рядом? Определите формулу, по которой получаются значения элементов последовательности. Выведите все возможные комбинации слов.
9.	Напишите программу с функцией, которая в качестве аргументов получает ссылку на другую функцию и два целых числа. В качестве результата функция возвращает наибольшее значение переданной первым аргументом функции в целочисленных точках диапазона	10.	Напишите программу, в которой описывается функция с произвольным количеством числовых аргументов. В качестве результата возвращается список из трех элементов: среднее значение аргументов, максимальное значение среди аргументов и мини-

	(границы которого определяются вторым и третьим аргументом).		мальное значение среди аргументов.
11.	Задана последовательность: 1; 2; 4; 8; 16; 32; 64, …. Определите формулу, по которой получаются значения элементов последовательности. Выведите первых тридцать членов этой последовательности.	12.	Опишите функцию equals (N, S) (где N и S – неотрицательные целые числа), которая проверяет, совпадает ли сумма цифр в десятичной записи числа N со значением S (например: equals (1234567, 28) = True, equals (10, 7) = False).
13.	Дана произвольная последовательность чисел. Напишите программу поиска максимального числа в этой последовательности.	14.	Напишите формулу последовательности, членами которой являются числа: -3; -1; 3; 11; …. Выведите первых тридцать членов этой последовательности.
15.	Напишите формулу последовательности, членами которой являются числа: 39; 37; 35; …. Выведите первых тридцать членов этой последовательности.	16.	Даны две дроби A/B и C/D (A, B, C, D – натуральные числа). С помощью функции напишите программу деления дроби на дробь. Ответ должен быть несократимой дробью.
17.	Из заданного числа вычли сумму его цифр. Из результата вновь вычли сумму его цифр и т. д. Определите, через какое количество таких действий получится нуль?	18.	Натуральное число, в записи которого n цифр, называется числом Армстронга, если сумма его цифр, возведенная в степень n, равна самому числу. Найти все числа Армстронга от 1 до к.
19.	Найти все простые натуральные числа, не превосходящие n, двоичная запись которых представляет собой палиндром, т. е. читается одинаково слева направо и справа налево.	20.	Напишите программу, которая выводит символы из текста, «через один»: то есть отображается первый, третий, пятый и так далее, символы.

Вопросы для самоконтроля к главе 10

1. Покажите на примере, как осуществляется объявление и вызов функции?
2. Чем функция отличается от метода?
3. Чем функция-генератор отличается от обычной функции. Покажите разницу на примере.
4. Какими способами можно задать функцию с переменным числом аргументов. Покажите разницу в способах задания на примерах.

5. В каких случаях целесообразно задавать значения аргументов функции «по-умолчанию»? Приведите примеры.
6. Что такое лямбда-функция? Чем лямбда-функция отличается от обычной функции?
7. Когда имеет смысл применять лямбда-функцию? Приведите пример.
8. Что такое вложенная функция? Приведите пример.
9. Какие способы задания вложенных функций вы знаете? Приведите примеры.
10. Что такое рекурсия?
11. Какая функция называется рекурсивной? Приведите примеры.
12. Как выполняется рекурсивная функция?
13. В чем разница между рекурсией и итерацией?
14. В каких случаях рекурсия предпочтительней итерации? Приведите пример.

Тест самоконтроля к главе 10

1. Что такое функция?
а) объект, принимающий аргументы и возвращающий значение; б) структура, определяющая поведение объекта; в) любой код в Python; г) код, заключенный в круглые скобки.
2. Какое ключевое слово используется для создания функции?
а) fun; б) function; в) void; г) def.
3.Сколько параметров может принимать функция?
а) ни одного, функция не принимает значения, только возвращает; б) 1; в) 2; г) бесконечно много.
4. Какое ключевое слово используется для возврата значения из функции? а) get; б) post; в) return; г) answer.
5. Что выведет этот код?

```
def get_sum (a,b):
        pass
print (get_sum (4, 2))
```

а) 4; б) None; в) 6; г) 2.
6. Каким ключевым словом обозначается анонимная функция?
а) lambda; б) alpha; в) anonym; г) таких функций не существует.
7. Что выведет этот код?

```
def get_sum (a=2,b=3):
        print (a+b)
get_sum (4)
```

а) 5; б) 4; в) 7; г) 2.
8. Для чего используется ключевое слово global?
а) чтобы сделать переменную доступной из любой точки мира; б) чтобы указать, что эта переменная имеет самое важное значение в программе; в) такого ключевого слова в Python нет; г) чтобы переменную можно было изменять за пределами текущей области видимости.
9. Что такое рекурсивная функция?
а) функция, которая используется другими функциями; б) функция, которая обращается сама к себе; в) функция, которая выполняется с последней строки по первую; г) функция, которая имеет переменное число аргументов.
10. В каком случае правильно создана анонимная функция? а) lambda x+1 : x; б) x + 1= lambda x; в) lambda x : x + 1; г) def lambda ():.
11. Функцией, отвечающей в Python за фильтрацию элементов по условию, является: а) where (); б) filter (); в) equal (); г) sort ().
12. Параметрами, принимаемыми на вход функцией filter() являются …
а) количество элементов в итерируемом объекте, итерируемый объект;
б) значение-условие, итерируемый объект; в) имя созданной пользователем функции, итерируемый объект; г) итерируемый объект и значение-условие.
13. Вложенные функции – это …
а) функции с переменным числом аргументов; б) функции, определяемые внутри других функций; в) функции, принимаемые в качестве аргументов объекты класса; г) функция, которая обращается сама к себе.
14. Что выведет код?
```
def f (1)
    print(1)
f(2)
```
а) ошибку; б) ничего; в) 1; г) 2.
15) Что выведет код?
```
def f (a,b):
    print(a,b)
f(2)
```
а) ошибку; б) 10, None; в) 1; г) 2.
16) Что выведет код?

```
def f (a):
    print("a")
f(2)
```

а) ошибку; б) «а»; в) 1; г) 2.

17) Что выведет код?

```
def f (a):
    print(a)
f(2)
```

а) ошибку; б) «а»; в) 1; г) 2.

18) Что выведет код?

```
def f (a):
    print(a.pop(1))
f([1,2,3])
```

а) ошибку; б) а; в) 1; г) 2.

19) Как называются функции, принимающие в качестве аргументов другие функции?

а) функции с переменным числом аргументов; б) функции высшего порядка; в) функции переменного порядка; г) функции с функциональным аргументом.

20) Какая функция позволяет заменить цикл при обработке итерируемого объекта?

а) for (); б) maps(); в) with(); г) map ().

21) В каком из предложенных случаев будет применяться функция reduce ()? а) построение нового списка; б) удаление дубликатов списка; в) подсчет суммы элементов списка; г) удаление элементов списка со значением 0 из списка.

10.9. Дополнительные источники по работе с функциями

1. Васильев А. Программирование на Python в примерах и задачах. – Москва: Эксмо, 2021. – 616 с.
2. Хайнеман Дж., Поллис Г., Сеяков С. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд.: Пер. с англ. – СпБ.: ООО “Альфа-книга”, 2017. – 432 с .: ил.
3. Лутц М. Изучаем Python, 3_е издание – Пер. с англ. – СПб.: Символ_Плюс, 2009. – 848 с., ил.
4. Значения аргументов функции «по умолчанию». Режим доступа: https://docs-python.ru/tutorial/opredelenie-funktsij-python/znachenie-argumenta-umolchaniju-funktsijah/

5. Лямбда-функции. Режим доступа: https://habr.com/ru/company/piter/blog/674234/
6. Создание вложенных функций. Режим доступа: https://proglib.io/p/kak-v-python-primenyayutsya-vlozhennye-funkcii-2021-02-09
7. Что такое декораторы и как с ними работать. Режим доступа: https://proglib.io/p/vse-chto-nuzhno-znat-o-dekoratorah-python-2020-05-09.
8. Функции-генераторы. Режим доступа: https://pythonist.ru/generatory-python-ih-sozdanie-i-ispolzovanie/

Глава 11. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Моделирование реального мира в методологии ООП

В реальном мире нам приходится иметь дело с физическими объектами, такими, например, как люди. Эти объекты представляют собой совокупность свойств и поведения.

Примерами свойств (данных, характеристик, атрибутов) для людей могут являться цвет глаз или место работы; для машин — мощность двигателя и количество дверей. Таким образом, свойства объектов имеют определенное значение, например голубой для цвета глаз или 4 для количества дверей автомобиля.

Поведение — это некоторая реакция объекта в ответ на внешнее воздействие. Например, если вы нажмете на тормоз автомобиля, это повлечет за собой его остановку. Остановка является примером поведения. Поведение сходно с функцией: вы вызываете функцию, чтобы совершить какое-либо действие (например, вывести на экран учетную запись), и функция совершает это действие.

Объектно-ориентированный подход

Основополагающей идеей объектно-ориентированного подхода является объединение данных и действий, производимых над этими данными, в единое целое, которое называется объектом. Функции объекта, называемые методами (или функциями-членами), обычно предназначены для доступа к данным объекта. Если необходимо считать какие-либо данные объекта, нужно вызвать соответствующий метод, который выполнит считывание и возвратит требуемое значение. Прямой доступ к данным невозможен. Данные сокрыты от внешнего воздействия, что защищает их от случайного изменения. Говорят, что данные и методы инкапсулированы. Термин инкапсуляция данных являются ключевым в описании объектно-ориентированных языков. Если необходимо изменить данные объекта, то, очевидно, это действие также будет возложено на методы объекта. Никакие другие функции не могут изменять данные класса. Такой подход облегчает проектирование, написание, отладку и использование программного кода.

Когда начинается процесс моделирования предметной области, вы сталкиваетесь с проблемой разбиения ее на объекты.

Что должно представляться в программе в виде объектов? Окончательный ответ на этот вопрос может дать только ваше вообра-

жение, однако приведем несколько советов, которые могут оказаться полезными:

- Физические объекты.
  - Автомобили при моделировании уличного движения.
  - Схемные элементы при моделировании цепи электрического тока.
  - Страны при создании экономической модели.
  - Самолеты при моделировании диспетчерской системы.
- Элементы интерфейса.
  - Окна.
  - Меню.
  - Графические объекты (линии, прямоугольники, круги).
  - Мышь, клавиатура, дисковые устройства, принтеры.
- Структуры данных.
  - Массивы.
  - Стеки.
  - Связанные списки.
  - Бинарные деревья.
- Группы людей.
  - Сотрудники.
  - Студенты.
  - Покупатели.
  - Продавцы.
- Хранилища данных.
  - Описи инвентаря.
  - Списки сотрудников.
  - Словари.
  - Географические координаты городов мира.
- Пользовательские типы данных.
  - Время.
  - Величины углов.
  - Комплексные числа.
  - Точки на плоскости.
- Участники компьютерных игр.
  - Автомобили в гонках.
  - Позиции в настольных играх (шашки, шахматы).
  - Животные в играх, связанных с живой природой.
  - Друзья и враги в приключенческих играх.

Объекты являются экземплярами классов. **Класс** является своего рода формой, определяющей, какие данные и функции (из каких элементов состоят, сколько памяти нужно, значения элементов и набор операций, которые можно применять к этим значениям, как их создавать и уничтожать) будут включены в объект класса. При объявлении класса не создаются никакие объекты этого класса. Класс является описанием совокупности сходных между собой объектов, обладающих определенным набором характеристик. Объект класса также называют экземпляром класса.

Понятие класса приводит нас к понятию **наследования.** В повседневной жизни мы часто сталкиваемся с разбиением классов на подклассы: например, класс наземный транспорт делится на классы автомобили, грузовики, автобусы, мотоциклы и т. д.

Кроме общих свойств, для нашего примера, наличие колес, подкласс может обладать и собственными свойствами. Например, автобусы имеют большее число посадочных мест для пассажиров, чем мотоциклы и грузовики (рис.82).

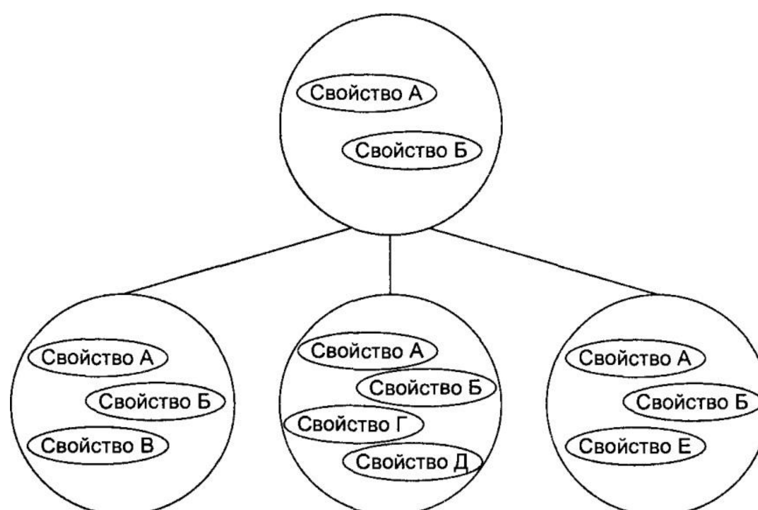


Рис.82. Производные классы. Наследование

Класс, который порождает все остальные классы, называется **базовым классом (суперклассом)**, остальные классы наследуют его свойства, одновременно обладая собственными свойствами. Такие классы называются **производными (подклассами).**

Полиморфизм, перегрузка методов и операторов

Использование операций и функций различным образом в зависимости от того, с какими типами величин они работают, называется **полиморфизмом**. Когда существующая операция или метод, например = или +, наделяется возможностью совершать действия над операндами нового типа, говорят, что такая операция (метод) является

перегруженной. Перегрузка представляет собой частный случай полиморфизма. Например, применение операции = к символьным и целым числам.

Таким образом, при проектировании в парадигме ООП **программа** представляет собой набор *взаимодействующих* друг с другом объектов. Посредством взаимодействия объекты в программе выполняют определенные *вычисления*.

Вычисления осуществляются путем взаимодействия (обмена данными через сообщения) между объектами, при котором один объект требует, чтобы другой объект выполнил некоторое действие.

Сообщения - это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия.

11.1. Описание классов и создание объектов

Шаблон описания класса выглядит достаточно просто: после ключевого слова class указывается название класса, ставится двоеточие и далее следует блок с собственно описанием класса. Общая конструкция с описанием класса, показана ниже.

```
Class имя класса:
     #описание класса
```

В самом простом случае класс может не содержать описания. В этом случае блок описания класса состоит из ключевого слова pass. Для создания объекта на основе класса переменной в качестве значения присваивается инструкция, которая состоит из названия класса и в простом случае пустых круглых скобок. Пример описания класса и создания на его основе двух объектов представлен ниже:

```
Class MyClass:
    pass
a=MyClass ()
b=MyClass ()
```

В этой программе описан класс MyClass. На основе класса командами a=MyClass() и b=MyClass() создаются два объекта, ссылки на которые записываются в переменные a и b. Так как класс не содержит описания, объекты класса не содержат никаких данных и методов для их обработки.

Приведем пример класса, содержащего описание. Описание класса включает в себя поля (свойства объектов класса) и методы.

```
class Employee:
    """Базовый класс для всех сотрудников"""
    emp_count = 0
#конструктор, инициализирующий значения полей
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.emp_count += 1
#метод, подсчитывающий и выводящий количество со-
трудников
    def display_count(self):
        print('Всего сотрудников: %d' % Employ-
ee.empCount)
class Ingeneer (Employee):
        def __init__(self, name, salary):
             super().__init__(name, salary)

Классы Employee и Ingeneer. Для того, чтобы Необ-
ходимо вызвать метод инициализации родителя
# В Python 3.x это делается при помощи функции su-
per()
```

Классы образуют иерархическую древовидную структуру. Фактически ООП в языке Python сводится к выражению:

```
object.attribute
```

Когда подобное выражение применяется к объекту, полученному с помощью инструкции class, интерпретатор начинает поиск в дереве связанных объектов, который заканчивается, как только будет встречено первое появление атрибута attribute. Просматриваются все классы выше объекта, снизу вверх и слева направо (рис.83).

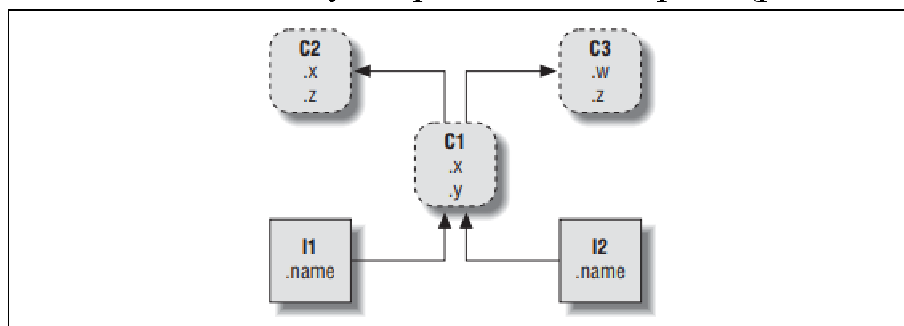


Рис.83. Иерархия классов

На рисунке дерево связывает вместе три объекта классов C1, C2 и C3 и два объекта экземпляров I1 и I2 в иерархию наследования.

Если мы напишем: I2.w, получится выражение вида

object.attribute, оно приводит к запуску поиска в дереве с рис. 22.1 – интерпретатор приступает к поиску атрибута w, начиная с I2, и движется вверх по дереву. Он будет просматривать объекты в следующем порядке: I2, C1, C2, C3 и остановится, как только будет найден первый атрибут с таким именем, или возбудит исключение, если атрибут w вообще не будет найден. Поиск будет продолжаться, пока не будет достигнут объект C3, поскольку атрибут w имеется только в этом объекте. Другими словами, имя I2.w в терминах автоматического поиска ищется как C3.w.

Синтаксис программного кода, реализующих дерево объектов классов с рис.84:

```
#создаем классы и объекты классов
 class C2:. . .
 class C2:. . .
 #связываем классы с суперклассом
 Class C1 (C2, C3): . . .
 #создаем экземпляры классов
 I1=C1()
 I2=C2()
```

Из-за особенностей поиска в дереве наследования имеет большое значение, к какому из объектов присоединяется тот или иной атрибут, – тем самым определяется его область видимости. Атрибуты, присоединяемые к экземплярам, принадлежат только этим конкретным экземплярам, но атрибуты, присоединяемые к классам, совместно используются всеми подклассами и экземплярами.

Атрибуты обычно присоединяются к экземплярам с помощью присваивания значений специальному аргументу с именем self, передаваемому функциям внутри классов. Например, классы определяют поведение своих экземпляров с помощью функций, создаваемых инструкциями def внутри инструкции class, она обычно называется методом и автоматически принимает специальный первый аргумент с именем self, который содержит ссылку на обрабатываемый экземпляр (объект класса). Синтаксис (рис. 84):

```
class C1(C2, C3):               # Создать и связать класс C1
    def setname(self, who):     # Присвоить: C1.setname
        self.name = who         # Self - либо I1, либо I2

I1 = C1()                       # Создать два экземпляра
I2 = C1()
I1.setname('bob')               # Записать 'bob' в I1.name
I2.setname('mel')               # Записать 'mel' в I2.name
print I1.name                   # Выведет 'bob'
```

Рис. 84. Синтаксис классов

Пример:

```
class Smallobj:
     def __init__(self, data):
        """конструктор, инициализирующий открытое
поле somedata значением"""
        self.somedata=data
     def show (self):
        print (self.somedata)
#создадим дочерний класс
class Sm(Smallobj):
     pass
#основная программа
s1=Smallobj (2)
print (s1.somedata)
s1.show()

#создаем экземпляр дочернего класса и пользуемся
методами родительского класса
s2=Sm (3)
s2.show()
```

Связь между родительским и дочерним классом определяется тем, что наследуемый класс передается в качестве аргумента, принимаемого дочерним классом.

Уровни доступа к полям и методам класса (в общем виде):

1. Private. Приватные члены класса недоступны для методов, расположенных вне класса, с ними можно работать только внутри класса. Классы-потомки также не имеют доступа к приватным членам базового класса.
2. Protected.Доступ к защищенным ресурсам класса возможен только внутри этого класса и также внутри унаследованных от него классов-потомков.
3. Public. Публичные методы наоборот – доступны за пределами класса. объявляются публичными сразу по умолчанию. Все члены класса в Python являются публичными по умолчанию.

В Python поддерживается соглашение, в соответствии с которым:

– если переменная/метод начинается с одного нижнего подчеркивания (_protected_example), то она/он считается защищенным (protected).

– если переменная/метод начинается с двух нижних подчеркиваний (__private_example), то она/он считается приватным (private).

В Python на самом деле вы можете обратиться к методам, несмотря на их «ограниченный» доступ. На уровне соглашения – взрослые люди просто не будут их вызывать вне класса. Например:

```
class Car:
    def _start_engine(self):
        return "Engine's sound."

    def run(self):
        return self._start_engine()
car = Car()
assert "Engine's sound." == car.run()
assert "Engine's sound." == car._start_engine()

#будет напечатано
Process finished with exit code 0
```

Если бы метод_start_engine () обновлял какие-то переменные класса или сохранял состояние, а не просто возвращал строку текста, вы могли что-то поломать в работе с классом. Авторы библиотек рассчитывают, что никто не будет пользоваться защищенными и приватными методами классов, прежде не заглянув в их код. В примере для этих целей есть публичный метод run ().

Есть другой способ «переложить» отвественность с человека на интерпретатор – декораторы.

11.2. Декораторы

Декоратор — это функция, которая позволяет «обернуть» другую функцию, расширив ее функциональность без непосредственного изменения ее кода. Такая обертка позволяет программам работать с другими программами как со своими данными. Это очень похоже на работу вложенных функций.

Рассмотрим пример:

```
class Rectangle:
    def __init__(self, a, b):
        self.a = a
        self.b = b
```

```
    @property
    def area(self):
        return self.a * self.b
#основная программа
rect = Rectangle(5, 6)

print(rect.area)

#будет напечатано
30
```

В классе Прямоугольник (Rectangle) есть защищенный с помощью декоратора метод area (), вычисляющий площадь прямоугольника. Метод доступен потомкам класса и недоступен объектам других классов.

Попытка вызова его объектом класса Square приводит к ошибке:

```
class Rectangle:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    @property
    def area(self):
        return self.a * self.b
class Square ():
    def __init__(self, c, d):
        self.c=c
        self.d=d
rect = Rectangle(5, 6)

print(rect.area)
sq=Square (5, 6)

print (sq.area)

#будет напечатано
Traceback (most recent call last):
  File
"C:/Users/Larisa/PycharmProjects/stroky/decorator.
py", line 16, in <module>
    print (sq.area)
```

```
AttributeError: 'Square' object has no attribute 'area'
```

Если мы сделаем класс Rectangle родителем класса Square, ошибки не будет:

```
class Rectangle:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    @property
    def area(self):
        return self.a * self.b
class Square (Rectangle):
    pass
rect = Rectangle(5, 6)
print(rect.area)
sq=Square (5, 6)
print (sq.area)
#будет напечатано
30
30
```

Свойства, которые должны иметь все объекты класса Rectangle, определяются в специальном методе с именем __init__() – конструкторе. Методу __init__() можно передать любое количество параметров, но первым параметром всегда является автоматически создаваемая переменная с именем self. Переменная self ссылается на только что созданный экземпляр класса, за счет чего метод __init__() сразу может определить новые атрибуты. Это очень удобный способ задать параметры объекта при его создании.

Конструктор – метод класса, выполняющийся автоматически в момент создания объекта класса. Является типичным представителем большого класса методов, которые называются ***методами перегрузки операторов.***

В качестве примера перегрузки предположим, что вас привлекли к реализации приложения базы данных, где хранится информация о служащих. Как программист, использующий объектно-ориентированные особенности языка Python, вы могли бы начать работу с реализации общих суперклассов, который определяет поведение, общее для всех категорий служащих в вашей организации, что и сделано в классе Salary (заработная плата):

```
class Salary(object): #начисление зарплаты общее: оклад*на 12 месяцев
class Employee: # Общий суперкласс Работники

#Общее поведение
      def getTotal (self):...#начисление зарплаты

      def giveRaise (self):...#начисление надбавки

      def retire (self): ...#увольнение
```

Реализовав это общее поведение, можно специализировать его для каждой категории служащих, чтобы отразить отличия разных категорий от стандарта. То есть можно создать подклассы, которые изменяют лишь ту часть поведения, которая отличает их от типового представления служащего, – остальное поведение будет унаследовано от общего класса Salary.

Например, если зарплата инженеров начисляется в соответствии с какими-то особыми правилами (то есть не по почасовому тарифу), в подклассе можно переопределить всего один метод:

```
class Engineer(Employee): # Специализированный подкласс
     def getTotal (self): ... # Особенная реализация
```

Поскольку эта версия находится в дереве классов ниже, она будет замещать (переопределять) общую версию метода в классе Employee.

Затем можно создать экземпляры разновидностей классов служащих в соответствии с принадлежностью имеющихся служащих классам, чтобы обеспечить корректное поведение:

```
bob = Employee() # Поведение по умолчанию
mel = Engineer() # Особые правила начисления зарплаты

company = [bob, mel] # Составной объект
for emp in company:
# Вызвать версию метода для данного объекта
  print (emp.computeSalary())
class Salary(object):
    def __init__(self, pay):
        self.pay = pay
```

```
    def getTotal(self):
        return (self.pay * 12)

class Employee(object):
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus
    def annualSalary(self):
        return "Total: " + str(self.pay.getTotal()
+ self.bonus)

class Engineer(Salary, Employee):
    def __init__(self,pay=100,bonus=10):
        self.pay = pay
        self.bonus=bonus
    _pay_times = 6
    # метод, начисляющий оклад
    def getTotal(self):
        return (self.pay * self._pay_times)
```

11.3. Наследование, множественное наследование

Язык программирования Python является языком, поддерживающим возможность множественного наследования. То есть, возможность у класса потомка наследовать функционал не от одного, а от нескольких родителей. Благодаря этому мы можем создавать сложные структуры, сохраняя простой и легко поддерживаемый код. Например, у нас есть два класса Auto (Автомобиль) и Boat (лодка):

```
class Auto:
    def ride(self):
        print("Ездит по земле")
class Boat:
    def swim(self):
        print("Плавает в воде")
```

Допустим, нам необходимо запрограммировать автомобиль-амфибию, который будет плавать в воде, и ездить по земле. Вместо написания нового класса, можем просто унаследовать свойства уже существующих классов:

```
class Auto:
    def ride(self):
        print("Ездит по земле")
class Boat:
    def swim(self):
        print("Плавает в океане")
class Amphibian(Auto, Boat):
    pass
a = Amphibian()
a.ride()
a.swim()
#Будет напечатано
Ездит по земле
Плавает в океане
```

Обратите внимание, что объект класса Amphibian, будет одновременно объектом классов Auto и Boat. Наглядно убедиться в этом поможет код ниже:

```
>>> a =Amphibian()
>>> isinstance(a,Auto)
True
>>> isinstance(a,Boat)
True
>>> isinstance(a,Amphibian)
True
```

Классы, подобные Amphibian, получили специальное назавание – классы-примеси (или миксины).

11.4. Порядок разрешения методов в Python. Ромбовидное наследование

Как было показано в предыдущем параграфе, классы-наследники могут использовать родительские методы. Но что, если у нескольких родителей будут одинаковые методы? Какой метод в таком случае будет использовать наследник? Рассмотрим классический пример:

```
class A:
  def hi(self):
    print("A")
class B(A):
```

```
  def hi(self):
    print("B")
class C(A):
  def hi(self):
    print("C")
class D(B, C):
  pass
d = D()
d.hi()
#будет напечатано
B
```

Такая ситуация получила название **ромбовидное наследование** (**diamond problem**) решается в **Python** путем установления порядка разрешения методов. В Python3.* для определения порядка используется **алгоритм поиска в ширину**, рассмотренный подробно в параграфе 12.1. Просмотреть, в каком порядке будут проинспектированы родительские классы, можно при помощи метода mro():

```
class A:
  def hi(self):
    print("A")
class B(A):
  def hi(self):
    print("B")
class C(A):
  def hi(self):
    print("C")
class D(B, C):
  pass
print (D.mro())
#будет напечатано
B
[<class '__main__.D'>, <class '__main__.B'>,
<class '__main__.C'>, <class '__main__.A'>, <class
'object'>]
```

Для того, чтобы использовать метод конкретного родителя, например hi() класса C, нужно напрямую вызвать его по имени класса, передав объект-наследник в качестве аргумента:

```
d = D()
C.hi(d)
```

```
#будет напечатано
C
```

11.5. Перегрузка операторов

Перегрузка операторов в Python – это возможность с помощью специальных методов в классах переопределять различные операторы языка. Имена таких методов включают двойное подчеркивание спереди и сзади. Под операторами в данном контексте понимаются не только знаки +, -, *, /, обеспечивающие операции сложения, вычитания и др., но также специфика синтаксиса языка, обеспечивающая операции создания объекта, вызова объекта как функции, обращение к элементу объекта по индексу, вывод объекта и другое.

Вы уже знакомы с одним из методов перегрузки операторов – это __init__() – конструктор объектов класса. Есть еще и другие (рис.85,86):

- __del__() – деструктор объектов класса, вызывается при удалении объектов;
- __str__() – преобразование объекта к строковому представлению, вызывается, когда объект передается функциям print() и str();
- __add__() – метод перегрузки оператора сложения, вызывается, когда объект участвует в операции сложения, будучи операндом с левой стороны;
- __setattr__() – вызывается, когда атрибуту объекта выполняется присваивание.

```
class Stars:
    def __init__(self, n):
        self.qty = '*' * n

    def __add__(self, n):
        return self.qty + '*' * n


a = Stars(3)

b = a + 5

print(b)  # вывод: ********
```

Рис.85. Метод перегрузки операции сложения

```python
class A:
    def __init__(self, v):
        self.field = v

    def __setattr__(self, attr, value):
        if attr == 'field':
            self.__dict__[attr] = value
        else:
            raise AttributeError


a = A(15)

a.field = -3

a.field2 = 10
```

Рис.86 Метод преобразования объекта к строковому представлению

В Python существует много других методов перегрузки операторов (табл.13).

Таблица 13. Сводная таблица методов перегрузки арифметических и логических операторов

Оператор	Выражение	Специальная функция
Сложение	p1 + p2	p1.__add__(p2)
Вычитание	p1 - p2	p1.__sub__(p2)
Умножение	p1 * p2	p1.__mul__(p2)
Возведение в степень	p1 ** p2	p1.__pow__(p2)
Деление	p1 / p2	p1.__truediv__(p2)
Целочисленное деление	p1 // p2	p1.__floordiv__(p2)
Взятие остатка	p1 % p2	p1.__mod__(p2)
Побитовый сдвиг влево	p1 << p2	p1.__lshift__(p2)
Побитовый сдвиг вправо	p1 >> p2	p1.__rshift__(p2)
Побитовое И (AND)	p1 & p2	p1.__and__(p2)
Побитовое ИЛИ (OR)	p1 | p2	p1.__or__(p2)
Побитовое исключающее ИЛИ (XOR)	p1 ^ p2	p1.__xor__(p2)
Побитовое НЕ (NOT)	~p1	p1.__invert__()

Python не ограничивается перегрузкой арифметических операторов. Перегружать можно и операторы сравнения. Сводная таблица методов их перегрузки представлена ниже (табл.14).

Таблица 14. Сводная таблица методов перегрузки операторов сравнения

Оператор	Выражение	Специальная функция
Меньше чем	p1 < p2	p1.__lt__(p2)
Меньше или равно	p1 <= p2	p1.__le__(p2)
Равно	p1 == p2	p1.__eq__(p2)
Не равно	p1 != p2	p1.__ne__(p2)
Больше чем	p1 > p2	p1.__gt__(p2)
Больше или равно	p1 >= p2	p1.__ge__(p2)

В коде ниже на примере сравнения координат точек p1, p2 и p3 показан алгоритм перегрузки оператора «меньше чем» с помощью метода __lt__():

```
# Перегрузка оператора «меньше чем»
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __lt__(self, other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y **
2)
        return self_mag < other_mag

# задаем значения точек p1, p2, p3
p1 = Point(1,1)
p2 = Point(-2,-3)
p3 = Point(1,-1)

# используем «меньше чем»
print(p1<p2)
print(p2<p3)
print(p1<p3)

#будет напечатано
True
False
False
```

11.6. Дополнительные источники по объектно-ориентированному программированию в Python

1. Лутц М. Изучаем Python, 3_е издание – Пер. с англ. – СПб.: Символ_Плюс, 2009. – 848 с., ил.
2. Концепция объектно-ориентированного программирования. Режим доступа: https://youtu.be/Z7AY41tE-3U?si=OXFJO1s-6K-SBDLL
3. Методы классов. Параметр self. Объектно-ориентированное программирование Python. Режим доступа: https://www.youtube.com/watch?v=Lw8TeLS4_IA&t=1s
4. Наследование в объектно-ориентированном программировании. Режим доступа: https://www.youtube.com/watch?v=7WVYqjdMa6U&t=1s

Практическая работа № 13
СОЗДАНИЕ ПОЛЬЗОВАТЕЛЬСКИХ КЛАССОВ

Задание

I. Разработать программу на языке Python, в которой:
   1. создается класс, описывающий поведение объектов, представляющих (см. колонку "Поведение") таких персонаажей:
      a. 1 – пользователей компьютера;
      b. 2 – литературных персонажей;
      c. 3 – студентов;
      d. 4 – героев мультипликации;
      e. 5 – исторических персонажей;
      f. 6 – персонажей художественных фильмов;
   2. класс должен иметь следулющие специальные методы: __init__(), __str__() и __del__();
   3. класс должен иметь такие атрибуты и/или методы (см. колонку "Атрибуты/методы"):
      a. 1 – статический метод;
      b. 2 – атрибут класса;
      c. 3 – метод экземпляра класса;
      d. 4 – закрытый атрибут
      e. 5 – закрытый метод

4. осуществляется управление двумя атрибутами класса, для первого устанавливается режим "только чтение", для второго – согласно колонке "Управление):
   a. 1 выполняется чтение атрибута и запись в него;
   b. 2 выполняется чтение и удаление атрибута;
   c. 3 выполняется чтение, запись и удаление атрибута;
5. Создается иерархия классов;
6. создаются объекты класса и проверяется их работа.

II. Оформить отчет по практической работе, содержащий:

- титульный лист (Приложение 1);
- содержание;
- ответ на вопросы самоконтроля;
- для каждого варианта:
  - текст задания с указанием варианта;
  - программный код на языке Python версии 3.8 и выше, соответствующий заданию. Наличие в программном коде плановой обработки ошибок обязательно;
  - unit-тесты;
  - оценка временной сложности алгоритма методов (на выбор);
  - скриншоты результата работы программы для каждого unit-теста.

ВАРИАНТЫ ЗАДАНИЙ

Номер варианта	Поведение	Атрибуты/ методы	Управление
1	1	1,3,4	1
2	2	2,3,5	2
3	3	1,3,5	3
4	4	2,3,4	1
5	5	1,3,4	2
6	6	2,3,5	3
7	1	1,3,5	1
8	2	2,3,4	2
9	3	1,3,4	3
10	4	2,3,5	1
11	5	1,3,5	2
12	6	2,3,4	3

13	1	1,3,4	1
14	2	2,3,5	2
15	3	1,3,5	3
16	4	2,3,4	1
17	5	1,3,4	2
18	6	2,3,5	3
19	1	1,3,5	1
20	2	2,3,4	2

Вопросы для самоконтроля к главе 11

1. Каково основное назначение ООП в языке Python?
2. Дайте краткую характеристику основным концепциям ООП в языке Python.
3. Где выполняется поиск унаследованных атрибутов?
4. В чем разница между объектом класса и объектом экземпляра класса?
5. В чем состоит особенность первого аргумента в методах классов?
6. Для чего служит метод __init__?
7. Как создать класс? Приведите пример.
8. Как создать объект (экземпляр) класса? Приведите примеры.
9. Как определяются суперклассы для класса? Приведите пример.
10. Где и как создаются атрибуты классов? Приведите пример.
11. Где и как создаются атрибуты экземпляров классов? Приведите пример.
12. Что в языке Python означает слово self для классов?
13. Как можно расширить унаследованный метод вместо полного его замещения? Приведите пример.
14. Как производится перегрузка операторов в классах на языке Python? Приведите пример.
15. Когда может потребоваться перегрузка операторов в ваших классах?
16. Какой метод перегрузки оператора используется наиболее часто?
17. Что такое множественное наследование?
18. Объясните, что такое делегирование?

Тест самоконтроля к главе 11

1. Какое ключевое слово используется для вызова методов родительского класса? а) super; б) sub; в) superclass; г) upper.
2. Какая из нижеперечисленных переменных имеет видимость private? а)_key_; б) _fabric; в)__flag; г) keyword_.
3. Выберите верное утверждение: а) инкапсуляция – создание новых классов на основе существующих; б) полимиорфизм - переопределение методов, унаследованных от суперкласса в подклассах; в) подкласс – класс, на основе которого создаются другие классы; г) наследование – переопределение методов в классах-потомках.
4. Какой механизм позволяет создавать одни классы на основе других? а) абстракция; б) полиморфизм; в) наследование; г) инкапсуляция.
5. Как можно получить доступ к переменным, имеющим видимость private, в Python? а) обращением напрямую к классу; б) обращением к экземпляру класса; в) посредством специального символа «__»; г) через реализованные методы get (), set().
6. Для создания класса в Python используется ключевое слово ….
7. Какой метод Python отвечает за инициализацию полей класса? а) интерфейс; б) объект; в) get-метод; г) конструктор.
8. Для того, чтобы воспользоваться методом класса нужно создать …а) конструктор; б) экземпляр; в) класс-потомок; г) другой метод.
9. Для создания конструктора в Python используется ключевое слово …а) main; б) _init_; в) key; г) _class_.
10. Реализацией общей концепции ООП является… а) процедура; б) метод; в) класс; г) функция.
11. Что относится к основным принципам ООП? а) инкапсуляция, полиморфизм, наследование, абстракция; б) инкапсуляция, полиморфизм, делегирование, абстракция; в) полиморфизм, разделение интерфейса, наследование, абстракция; г) инкапсуляция, наследование, абстракция, открытость/ закрытость.
12. Что будет выведено на экран?

```
class A:
  a=None
print (A.a)
```

а) 0; б) False; в) None; г) Error.

13. Какой из перечисленных ниже принципов ООП обеспечивает возможность использования общего интерфейса для нескольких различных типов данных? а) инкапсуляция; б) полиморфизм; в) абстракция; г) наследование.
14. Какая из нижеперечисленных переменных имеет видимость private? а) private field = 0; б) field = 0; в) _field = 0; г) __field = 0.
15. Какая из перечисленных ниже команд создает конструктор класса А? а) А(параметры конструктора); б) def __init__(параметры конструктора); в) def init (параметры конструктора); г) def __A_ (параметры конструктора).
16. Какое количество конструкторов может иметь класс в Python? а) один; б) два; в) бесконечно много; г) не более двух.
17. При определении в классе деструктора с двумя параметрами будет …а) сгенерирована ошибка; б) сгенерировано предупреждение; в) ошибка или предупреждение сгенерированы не будут.
18. При определении в классе двух методов с одинаковыми именами и разным количеством параметров будет …а) сгенерирована ошибка; б) сгенерировано предупрждение; в) ошибка или предупреждение сгенерированы не будут.
19. Значением по умолчанию поля класса может быть … а) значение переменной; б) константа; в) результат вызова функции; г) в Python возможность указания значений полей по умолчанию не предусмотрена.
20. В языке Python объектами являются…а) экземпляры классов; б) экземпляры классов и переменные; в) экземпляры классов, переменные, функции; г) все типы данных.
21. Деструктор класса задается методом с именем: а) __del_; б) __delete__; в) __destr__; г) __destruct__.
22. Укажите результат выполнения кода:

```
x=0
class F:
    count=x
    def __init__(self):
        self.count+=1
    def __del(self):
        self.count+=1
obj = F()
print (obj.count)
```

а) 0; б) 1; в) 2; г) пустая строка.

Глава 12. РЕАЛИЗАЦИЯ ДИНАМИЧЕСКИХ СТРУКТУР ДАННЫХ В PYTHON. ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ В PYTHON

Переменные, описанные в главах 7 и 8, называются *статическими переменными*. Число статических переменных устанавливается в момент написания программы и не может изменяться в процессе ее выполнения. Однако очень часто встречаются задачи, в которых необходимое число переменных заранее неизвестно. Предположим, что необходимо обработать данные о людях, которые стоят в очереди в кассу за билетами. Длина очереди заранее неизвестна. Каждый раз, когда появляется новый человек, необходимо создать переменную соответствующего типа. После того, как человек уходит, соответствующая переменная становится излишней и желательно уничтожить ее в процессе выполнения программы.

Переменные, которые создаются и уничтожаются в процессе выполнения программы, называются *динамическими переменными*.

Динамическими структурами данных называются структуры данных, память под которые выделяется и освобождается по мере необходимости.

Динамические структуры данных являются необходимым инструментом увеличения эффективности программ на любом языке программирования. За счет чего это происходит?

Дело в том, что динамические структуры данных в процессе существования в памяти имеют ряд отличий от статических структур данных, а именно, могут изменять не только число составляющих их элементов, но и характер связей между элементами. При этом изменение содержимого самих элементов данных не учитывается.

Указанные особенности динамических структур приводят к тому, что на этапе создания машинного кода программа-компилятор не может выделить для всей структуры в целом участок памяти фиксированного размера, а также не может сопоставить с отдельными компонентами структуры конкретные адреса.

Для решения проблемы адресации динамических структур данных используется *метод*, называемый *динамическим распределением памяти*, то есть память под отдельные элементы выделяется в момент, когда они «начинают существовать» в процессе выполнения программы, а не вовремя компиляции. В этом случае компилятор вы-

деляет фиксированный объем памяти для хранения адреса динамически размещаемого элемента, а не самого элемента.

Динамическая структура данных характеризуется тем что:

- не имеет имени;
- ей выделяется память в процессе выполнения программы;
- количество элементов структуры может не фиксироваться;
- размерность структуры может меняться в процессе выполнения программы;
- в процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.

Каждой динамической структуре данных сопоставляется статическая переменная типа *указатель*, значением которой является *адрес объекта этой структуры*. То есть посредством указателя осуществляется доступ к динамической структуре.

Сами динамические величины не требуют описания в программе, поскольку во время компиляции память под них не выделяется. Во время компиляции память выделяется только под статические величины. Указатели – это статические величины, поэтому они требуют описания.

Необходимость в динамических структурах данных обычно возникает в следующих случаях:

- используются переменные, имеющие довольно большой размер, например, массивы большой размерности, необходимые в одних частях программы и совершенно ненужные в других;
- в процессе работы программы нужен список или иная структура, размер которой изменяется в широких трудно предсказуемых пределах;
- когда размер данных, обрабатываемых в программе, превышает объем сегмента данных.

Поскольку элементы динамической структуры располагаются по непредсказуемым адресам памяти, адрес элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента. Для установления связи между элементами динамической структуры и нужны указатели, с помощью которых устанавливаются явные связи между элементами. Такое представление данных в памяти называется *связным.*

Достоинства связного представления данных – в возможности обеспечения значительной изменчивости структур:

- размер структуры ограничивается только доступным объемом машинной памяти;
- при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей;
- большая гибкость структуры.

Вместе с тем, связное представление не лишено и недостатков, основными из которых являются:

- на поля, содержащие указатели для связывания элементов друг с другом, расходуется дополнительная память;
- доступ к элементам связной структуры может быть менее эффективным по времени.

Последний недостаток является наиболее серьезным и именно им ограничивается применимость связного представления данных. Если в смежном представлении данных для вычисления адреса любого элемента нам во всех случаях достаточно было номера элемента и информации, содержащейся в дескрипторе структуры, то для связного представления адрес элемента не может быть вычислен из исходных данных. Дескриптор связной структуры содержит один или несколько указателей, позволяющих войти в структуру, далее поиск требуемого элемента выполняется следованием по цепочке указателей от элемента к элементу. Поэтому связное представление практически никогда не применяется в задачах, где логическая структура данных имеет вид вектора или массива – с доступом по номеру элемента, но часто применяется в задачах, где логическая структура требует другой исходной информации доступа (таблицы, списки, деревья и т.д.).

Порядок работы с динамическими структурами данных следующий:

1. объявить динамическую структуру;
2. создать для объекта структуры место в динамической памяти;
3. работать при помощи указателя;
4. освободить занятое структурой место.

Классификация динамических структур данных

Во многих задачах требуется использовать данные, у которых *конфигурация*, размеры и состав могут меняться в процессе выполнения программы. Для их представления используют динамические информационные структуры. К таким структурам относят:

– однонаправленные (односвязные) списки;
– двунаправленные (двусвязные) списки;
– циклические списки;
– стек;
– дек;
– очередь;
– бинарные деревья;
– графы.

Они отличаются способом связи отдельных элементов и/или допустимыми операциями. Динамическая структура может занимать несмежные участки оперативной памяти.

Рассмотрим каждый шаг работы с динамическими структурами более подробно.

12.1. Односвязный список

Каждая компонента любой динамической структуры представляет собой запись, содержащую, по крайней мере, два поля: одно поле типа указатель, а второе – для размещения данных. В общем случае запись может содержать не один, а несколько указателей и несколько полей данных. Поле данных может быть переменной, списком или любой другой структурой (рис.85).

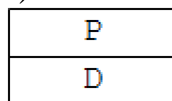


Рис.85. Общий вид компоненты динамической структуры

где:

- P – адресное поле (указатель);
- D – информационное поле, содержащее данные.

В общем виде объявление элемента динамической структуры данных выглядит следующим образом:

```
    struct имя_типа {
                    информационное поле;
                    адресное поле;
                   };
```

Видно, что поля структуры образуют пару. Реализация пары на Python может выглядеть так:

```
def make_pair(x,y):
    return lambda n: x if n==0 else y
```

```
def first(p):
    return p(0)

def second(p):
    return p(1)
```

Как видно из примера функция *make_pair()* возвращает пару через другую функцию, принимающую на вход аргумент (любое число), и возвращающую первый элемент, если аргумент равен 0, и второй – в противном случае. Для доступа к элементам пары созданы функции *first()* и *second()*.

Полный код программы представлен ниже:

```
def make_pair(x,y):
    return lambda n: x if n==0 else y
def first(p):
    return p(0)

def second(p):
    return p(1)

#основная программа
p = make_pair(5,6)
first(p) #5
second(p) #6
p1 = make_pair('hello',6)
first(p1) #'hello'
```

12.1.1. Создание, заполнение и вывод элементов односвязного списка

Реализуем с помощью пары связный список — базовую динамическую структура данных, состоящую из *узлов*, каждый из которых содержит как собственно данные, так и ссылки одного из двух видов:

– одна ссылка на следующий или предыдущий элемент (односвязный список);
– две ссылки на следующий и предыдущий узлы списка (двусвязный список).

Связный список относится к рекурсивным структурам данных, поскольку его определение рекурсивно. Представляет собой либо:

- пустой список, представленный значением *None;*

- узел, содержащий данные и ссылку на следующий элемент связного списка.

Структурная схема односвязного списка представлена на рис.86.

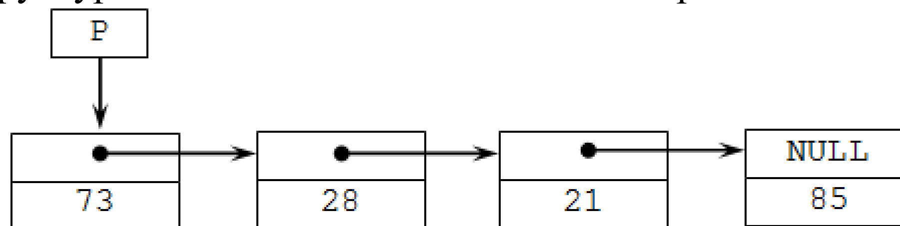


Рис.86. Схематичное представление односвязного списка

Данная структура состоит из 4 элементов. Ее первый элемент имеет поле *Data*, равное 73, и связан с помощью своего поля *Next* со вторым элементом, поле *Data* которого равно 28, и так далее до последнего, четвертого элемента, поле *Data* которого равно 85, а поле *Next* равно *NULL*, то есть нулевому адресу, что является признаком завершения структуры. Здесь *P* является указателем, который указывает на первый элемент структуры.

Создадим класс *Node,* представляющий из себя узел односвязного списка:

```
class Node:
    def __init__(self, data=None, next=None):
        self.data = data
        self.next  = next
    def __str__(self):
        return str(self.data)
```

Конструктор класса *__init__()* задает начальное значение *None* двум полям объекта структуры. Поле для того, чтобы в узле можно было хранить любое значение, использована функция *__str__()*, преобразующая любое значение в строку.

Создадим и инициализируем значениями 4 узла односвязного списка в тексте программы:

```
node1 = Node(73)
node2 = Node(28)
node3 = Node(21)
node4=Node (85)
```

В тексте программы свяжем узлы так, чтобы первый узел ссылаться на второй, а второй — на третий, третий – на четвертый. Ссылка в четвертом узле имеет значение *None*, что означает конец списка:

```
node1.next = node2
node2.next = node3
node3.next = node4
```

За вывод элементов списка отвечает функция *def print_list():*

```
def print_list(node):
    while node:
        print(node),
        node = node.next
    print
```

Вызовем функцию *print_list()*, передав ей ссылку на первый узел:

```
print_list(node1)
```

Внутри функции *print_list()* нет переменных, ссылающихся на другие узлы односвязного списка. Функция получает ссылку на следующий узел, используя значение атрибута *next* каждого узла.

Полный текст кода программы примера:

```
class Node:
    def __init__(self, data=None, next=None):
        self.data = data
        self.next  = next

    def __str__(self):
        return str(self.data)

def print_list(node):
    while node:
        print(node),
        node = node.next
    print
#основная программа
node1 = Node(73)
node2 = Node(28)
node3 = Node(21)
node4=Node (85)

node1.next = node2
node2.next = node3
node3.next = node4

print_list(node1)
```

Автоматизируем процесс создания односвязного списка, более полно реализовав концепцию указателей.

В Python существует несколько способов создания односвязного списка. Стандартное решение добавляет один узел к концу списка, делая список немного похожим на стэк (рис.87).

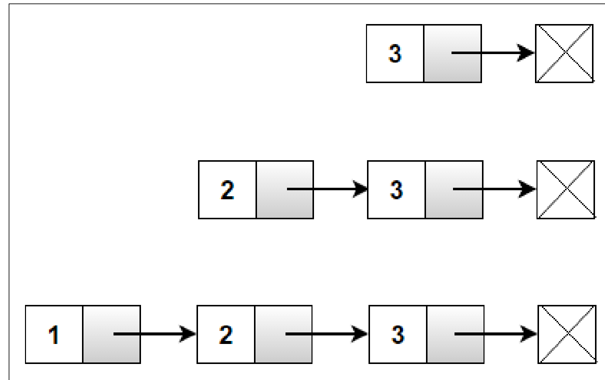


Рис.87. Стандартный способ заполнения односвязного списка с конца

```
# Узел связного списка
class Node:
   def __init__(self, data=None, next=None):
      self.data = data
      self.next = next

# Функция для печати связанного списка
def printList(head):
   ptr = head
   while ptr is not None:
      print(ptr.data, end=' —> ')
      ptr = ptr.next

   print('None')

# Функция построения односвязанного списка из за-
данного набора данных
def construct(keys):
   head = None

   # начать с конца списка
   for i in reversed(range(len(keys))):
      # выделяет новый узел в куче и устанавливает
его данные
      head = Node(keys[i], head)
```

```
   return head

    #основная программа
if __name__ == '__main__':

   # значения, из которых формируется связный спи-
сок
   keys = [73, 21, 28, 85]

   # указывает на головной узел связанного списка
   head = construct(keys)

   # распечатать связанный список
   printList(head)
```

Можно записать приведенный выше код в одну строку, передав следующий узел в качестве аргумента Node конструктору *construct()*:

```
# Узел связного списка
class Node:
    def __init__(self, data, next_node):
        self.data = data
        self.next = next_node

# Функция для печати заданного списка
def printList(head):
    ptr = head
    while ptr is not None:
        print(ptr.data, end=' —> ')
        ptr = ptr.next

    print('None')
# Функция для реализации связанного списка, содер-
жащего четыре узла
def construct():
    return Node(1, Node(2, Node(3, Node(4,
None))))

#основная программа
if __name__ == '__main__':
    keys = [73, 21, 28, 85]
```

```
    # `head` указывает на головной узел связанного
списка
    head = construct()

    # печать списка
    printList(head)
```

Задать значения элементов связного списка можно с клавиатуры. Пример такого ввода представлен ниже:

```
#ввод значений с клавиатуры

#узел списка
class LinkedListNode:
    def __init__(self, value, next):
        self.value = value
        self.next = next

#вывод текущего элемента связного списка и переход
к следующему элементу
def printList(head):
   ptr = head
   while ptr is not None:
      print(ptr.value, end=' —> ')
      ptr = ptr.next
   print('None')

#основная программа

#создание и инициализация начальными значениями
объекта структуры «связный список»
head=LinkedListNode(0,None)

#ввод значений элементов связного списка
for i in range (4):
    a= int (input())
    head = LinkedListNode(a,head)
printList(head)
```

12.1.2. Вставка элементов в односвязный список

В динамические структуры легко добавлять элементы, так как для этого достаточно изменить значения адресных полей. Вставка первого и последующих элементов списка отличаются друг от друга. Поэтому в методе, реализующем данную операцию, сначала осуществляется проверка, на какое место вставляется элемент. Далее реализуется соответствующий алгоритм добавления. Схематично процесс вставки элемента представлен на рис. 88.

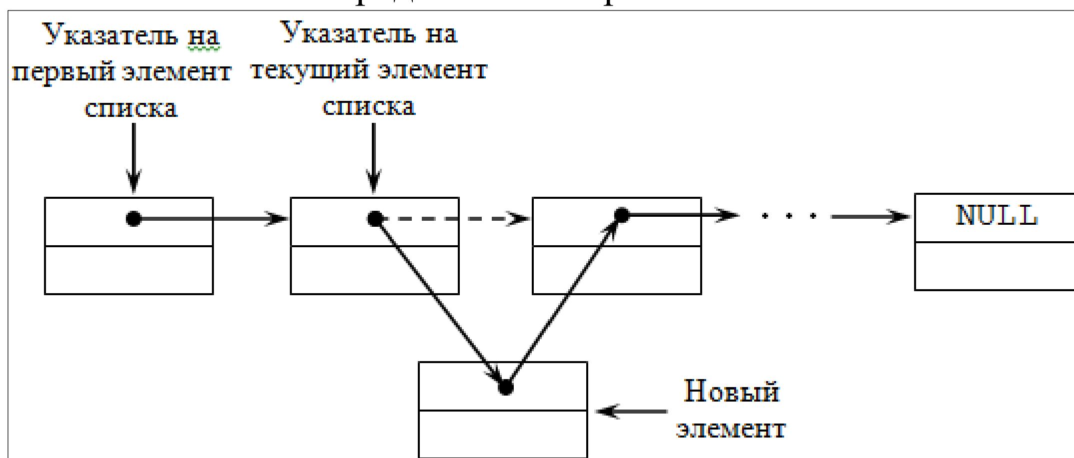


Рис.88. Графическое представление процесса вставки элемента в односвязный список

Самый простой способ вставить элемент в односвязанный список - это *добавить его в начало списка*. Следующая функция реализует это:

```
def insert_at_start(self, data):
    new_node = Node(data)
    new_node.ref = self.start_node
    self.start_node = new_node
```

В приведенном выше программном коде мы создаем метод *insert_at_start()*, принимающий один параметр – значение элемента, который мы хотим вставить в список. Для этого внутри метода мы создаем объект *Node* и устанавливаем его ссылку на *start_node* переменную, поскольку она ранее хранила ссылку на первый узел. После вставки нового узла в начало списка первый узел становится вторым узлом. Мы добавляем ссылку на *start_node* в *ref* нового узла. Так как теперь *new_node* является первым узлом, мы устанавливаем значение *start_node* переменной в *new_node*.

Добавим элементы в начало списка:

```
new_linked_list.insert_at_start(10)
new_linked_list.insert_at_start(5)
new_linked_list.insert_at_start(18)
```

```
#будет напечатано
18
5
10
```

Из примера видно, как элементы последовательно сдвигаются в конец списка при добавлении следующего элемента.

Вставка элементов в конец односвязного списка. Следущий метод используется для добавления элементов в конец односвязанного списка:

```
def insert_at_end(self, data):
    new_node = Node(data)
    if self.start_node is None:
        self.start_node = new_node
        return
    n = self.start_node
    while n.ref is not None:
        n = n.ref
    n.ref = new_node
```

Значение элемента, который мы хотим вставить, передается в качестве аргумента метода. Метод состоит из двух частей. Сначала мы проверяем, пуст ли список. Если связанный список пуст, все, что мы должны сделать, это установить значение *start_node* переменной в *new_node* объекта.

Если список уже содержит какие-то узлы. Мы инициализируем переменную *n* с помощью начального узла. Затем перебираем все узлы в списке, используя цикл while, как мы это делали в случае с методами *traverse_list()* и *insert_at_start()*. Цикл завершается, когда мы достигнем последнего узла. Затем мы устанавливаем ссылку последнего узла *new_node* на вновь созданный узел.

Добавим элемент в конец уже созданного нами списка:

```
new_linked_list.insert_at_end (11)
#будет напечатано
18
5
10
11
```

Из примера видно, что существующие элементы не меняют своего положения при добавлении следующего элемента.

Вставка элемента после другого элемента. Часто требуется добавить элемент в список после другого элемента. Для этого воспользуемся методом *insert_after_item()* как показано ниже:

```
def insert_after_item(self, x, data):
    n = self.start_node
    while n is not None:
        if n.item == x:
            break
        n = n.ref
    if n is None:
        print("item not in the list")
    else:
        new_node = Node(data)
        new_node.ref = n.ref
        n.ref = new_node
```

Метод *insert_after_item()* принимает два параметра: *x* и *data*. Первый параметр - это элемент, после которого вы хотите вставить новый узел, а второй параметр содержит значение для нового узла.

Начнем с создания новой переменной n и присвоения start_node переменной значения start_node. Затем, используя цикл while, просматриваем связанный список до тех пор, пока n не станет равной None. Во время каждой итерации проверяем, равно ли значение, хранящееся в текущем узле, значению параметра x. Если сравнение вернет истину, мы прервем цикл. Затем, если элемент найден, n не будет равно None. Ссылка new_node устанавливается на ссылку, сохраненную n, а ссылка n устанавливается на добавленный узел (new_node).

Вставим с помощью этого метода элемент, равный 555, сразу после элемента со значением 10:

```
new_linked_list.insert_after_item(10,555)

#будет напечатано
18
5
10
555
11
```

Вставка элемента *перед другим элементом*:

```
def insert_before_item(self, x, data):
    if self.start_node is None:
        print("List has no element")
        return
    if x == self.start_node.item:
        new_node = Node(data)
        new_node.ref = self.start_node
        self.start_node = new_node
        return
    n = self.start_node
    print(n.ref)
    while n.ref is not None:
        if n.ref.item == x:
            break
        n = n.ref
    if n.ref is None:
        print("item not in the list")
    else:
        new_node = Node(data)
        new_node.ref = n.ref
        n.ref = new_node
```

Метод состоит из трех частей. Давайте подробно рассмотрим каждую часть. Вначале мы проверяем, пуст ли список. Если он пуст, мы просто печатаем, что в списке нет элемента, и возвращаемся из метода:

```
if self.start_node is None:
     print("List has no element")
     return
```

Затем проверяем, является ли элемент первым в списке:

```
 if x == self.start_node.item:
     new_node = Node(data)
     new_node.ref = self.start_node
     self.start_node = new_node
     return
```

Если элемент, после которого мы хотим вставить новый узел, первый, мы устанавливаем ссылку на вновь вставленный узел через start_node, а затем устанавливаем значение start_node равное new_node.

Если список не пустой и элемент не является первым, мы создаем новую переменную n и присваиваем start_node переменной значение start_node:

```
n = self.start_node
print(n.ref)
```

Затем мы просматриваем связанный список, используя цикл while, который выполняется до тех пор, пока значение *n.ref* не станет равным *None*. Во время каждой итерации проверяем, равно ли значение, хранящееся в ссылке текущего узла, значению, переданному параметром *x*. Если сравнение вернет истину, мы прервем цикл.

Если элемент найден, значение *n.ref* не будет равно *None*. Ссылка *new_node* устанавливается на ссылку *n.ref*, которая указывает на новый узел (*new_node*):

```
if n.ref is None:
 print("item not in the list")
else:
 new_node = Node(data)
 new_node.ref = n.ref
 n.ref = new_node
```

12.1.3. Удаление элемента односвязного списка

Из динамических структур можно удалять элементы, так как для этого достаточно изменить значения адресных полей. Операция удаления элемента однонаправленного списка осуществляет удаление элемента, на который установлен указатель текущего элемента.

После удаления указатель текущего элемента устанавливается на предшествующий элемент списка или на новое начало списка, если удаляется первый.

Алгоритмы удаления первого и последующих элементов списка отличаются друг от друга. Поэтому в функции, реализующей данную операцию, осуществляется проверка, какой элемент удаляется. Далее реализуется соответствующий алгоритм удаления. Наглядно процесс удаления элемента представлен на рис.89.

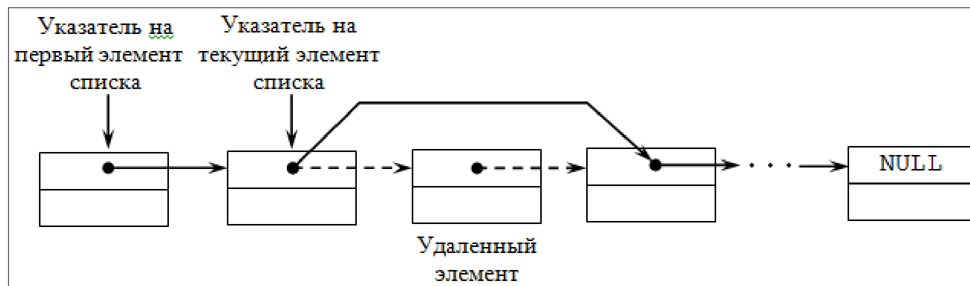


Рис.89. Графичекое представление процесса удаления элемента из односвязного списка

Удалить элемент односвязного списка можно несколькими способами, аналогичными вставке элемента в список.

Для того, чтобы *удалить элемент или элементы из начала* связанного списка, нужно установить ссылку *start_node* на второй узел. Это можно сделать, просто присвоив значение ссылки начального узла (который указывает на второй узел) второму узлу, как показано ниже:

```
def delete_at_start(self):
    if self.start_node is None:
        print("The list has no element to delete")
        return
    self.start_node = self.start_node.ref
```

Для того, чтобы *удалить элемент или элементы с конца* связанного списка, нужно перебрать список до предпоследнего элемента, затем установить значение поля ref этого элемента равным None, что сделает этот элемент последним в списке.

Программный код для метода *delete_at_end ()* выглядит следующим образом:

```
def delete_at_end(self):
    if self.start_node is None:
        print("The list has no element to delete")
        return
    n = self.start_node
    while n.ref.ref is not None:
        n = n.ref
    n.ref = None
```

Чтобы удалить элемент по значению, сначала нужно найти узел, содержащий указанное значение. Как только удаляемый элемент найден, ссылка на узел перед этим элементом устанавливается на узел, который расположен после удаляемого элемента. Программный код соответствующего метода представлен ниже:

```
def delete_element_by_value(self, x):
    if self.start_node is None:
        print("The list has no element to delete")
        return
    # Deleting first node
    if self.start_node.item == x:
        self.start_node = self.start_node.ref
        return
    n = self.start_node
    while n.ref is not None:
        if n.ref.item == x:
            break
        n = n.ref
    if n.ref is None:
        print("item not found in the list")
    else:
        n.ref = n.ref.ref
```

Сначала мы проверяем, пуст ли список. Затем, находится ли удаляемый элемент в начале связанного списка. Если элемент найден в начале, мы удаляем его, устанавливая ссылку на второй узел.

Если удаляемый элемент не является первым, мы обходим связанный список и проверяем, равно ли значение текущего узла значению, которое нужно удалить. Если сравнение возвращает истину, мы устанавливаем ссылку предыдущего узла на узел, который существует после удаляемого узла.

Таким образом, однонаправленный список имеет только один указатель в каждом элементе. Это позволяет минимизировать расход памяти на организацию такого списка.

Однако, это позволяет осуществлять переходы между элементами только в одном направлении, что зачастую увеличивает время, затрачиваемое на обработку списка. Например, для перехода к предыдущему элементу необходимо осуществить просмотр списка с начала до элемента, указатель которого установлен на текущий элемент.

12.2. Двусвязный (двунаправленный) список

Для ускорения многих операций целесообразно применять переходы между элементами списка в обоих направлениях. Это реализуется с помощью двунаправленных списков, которые являются сложной динамической структурой.

В двусвязном списке каждый узел имеет три компонента: значение узла, ссылку на предыдущий узел и ссылку на следующий узел. Такая организация позволяет осуществлять обход списка, как в прямом, так и в обратном направлении.

Для начального узла двусвязного списка ссылка на предыдущий узел равна *None*. Точно так же для последнего узла в двусвязном списке ссылка на следующий узел равна *None*.

Одним из основных недостатков двусвязного списка является больший в сравнении с односвязным списком объем занимаемой памяти, связанный с необходимостью хранения дополнительной ссылки для каждого узла. Как следствие, требуются дополнительные шаги для выполнения различных операций со списком.

Графичеки структура двусвязного списка представлена на рис.90.

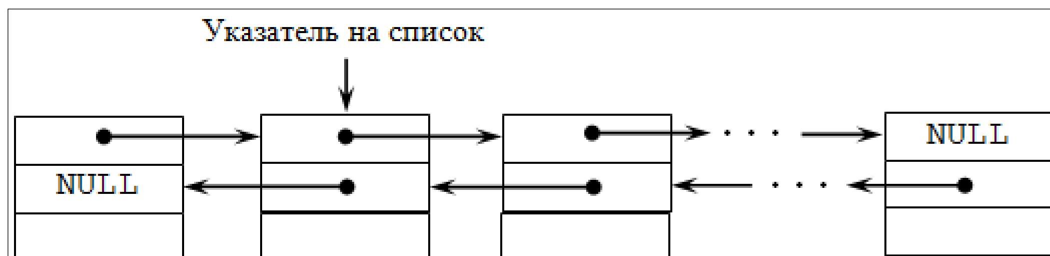


Рис.90. графическое представление структуры двусвязного списка

12.2.1.Создание, обход двусвязного списка, вставка элемента в список

В качестве примера создадим в *Python* простой двусвязный список. По аналогии с односвязным списком опишем узел с помощью класса *Node* с тремя атрибутами: *item*, *nref* и *pref*. В *item* будем хранить фактические данные, в *nref* – ссылку на следующий узел, а в *pref* - ссылку на предыдущий узел двусвязного списка. Зададим значение первого элемента списка равное 59:

```
class Node:
    def __init__(self, data):
        self.item = data
```

```
        self.nref = None
        self.pref = None
#основная программа
n=Node (59)
print(n.item)
```

Задать значение первому элементу списка можно с клавиатуры. В этом случае изменится код основной программы:

```
#основная программа
print ('задайте значение первого элемента списка')
a=int (input())
n=Node(a)
print (n.item)
```

Программный код ниже создает первый элемент в пустом двусвязном списке с помощью методов класса *DoublyLinkedList:*

```
class Node:
    def __init__(self, data):
        self.item = data
        self.nref = None
        self.pref = None

#класс содержит методы работы со списком
class DoublyLinkedList:
    def __init__(self):
        self.start_node = None

    def insert_in_emptylist(self, data):
        if self.start_node is None:
            new_node = Node(data)
            self.start_node = new_node
        else: print("list is not empty")

    def traverse_list(self):
        if self.start_node is None:
            print("List has no element")
            return
        else:
            n = self.start_node
            while n is not None:
                print(n.item, " ")
```

```
                n = n.nref

#основная программа
new_linked_list = DoublyLinkedList()
new_linked_list.insert_in_emptylist(50)
new_linked_list.traverse_list()
#будет напечатано
50
```

Метод *insert_in_emptylist()* сначала проверяет, имеет ли переменная *self.start_node* значение *None.*

Если значение переменной равно *None*, список пуст. В этом случае создается первый узел, и его значение инициализируется значением, переданным в качестве параметра функции *insert_in_emptylist()*, и значение *None* для переменной *self.start_node* устанавливается для нового узла. Если список не пустой, выводится соответствующее сообщение.

С помощью метода *traverse_list()* по аналогии с односвязным списком реализован обход двусвязного списка.

Можно вставить элемент в список одним из следующих способов:

– после другого элемента;
– перед другим элементом.

Наглядное представление процесса встаки элемента в двусвязный список представлено на рис.91.

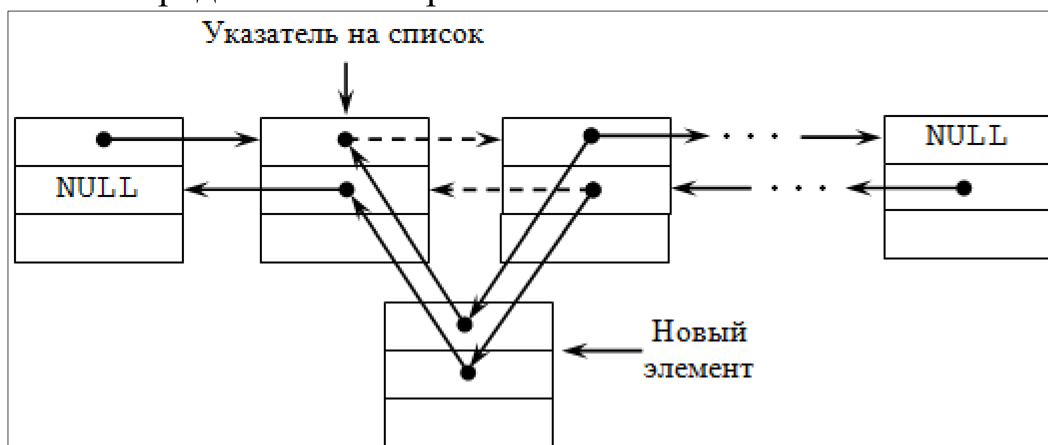


Рис.91. Графическое представление процесса вставки элемента в двусвязный список

Рассмотрим каждый способ подробно.

Чтобы *вставить элемент после другого элемента*, сначала необходимо проверить, пуст ли список. Если список пуст, необходи-

мо вначале создать первый узел списка. Можно для пользователя вывести соответствующее сообщение.

В противном случае перебираются все узлы двусвязного списка вплоть до последнего. Затем последовательно выполняются следующие операции:

1) установить предыдущую ссылку вновь вставленного узла на выбранный узел;
2) установить следующую ссылку вновь вставленного узла на следующую ссылку выбранного;
3) если выбранный узел не является последним узлом, установить предыдущую ссылку следующего узла на вновь добавленный узел;
4) установить следующую ссылку выбранного узла на вновь вставленный узел.

Программный код соответствующего метода выглядит следующим образом:

```
def insert_after_item(self, x, data):
    if self.start_node is None:
        print("List is empty")
        return
    else:
        n = self.start_node
        while n is not None:
            if n.item == x:
                break
            n = n.nref
        if n is None:
            print("item not in the list")
        else:
            new_node = Node(data)
            new_node.pref = n
            new_node.nref = n.nref
        if n.nref is not None:
            n.nref.prev = new_node
        n.nref = new_node
```

Чтобы *вставить элемент перед другим элементом*, необходимо сначала проверить, не пуст ли список. Если список пуст, можно вывести соответствующее сообщение. В противном случае необходимо перебрать все узлы двусвязного списка до нужного нам узла. Если такой узел не найден, вывести соответствующее сообщение. Если узел

найден, выбрать его, затем последовательно выполнить следующие операции:

1) установить следующую ссылку вновь вставленного узла на выбранный узел;
2) установить предыдущую ссылку вновь вставленного узла на предыдущую ссылку выбранного узла;
3) установить следующую ссылку узла, предшествующего выбранному узлу, на добавленный узел.
4) установить предыдущую ссылку выбранного узла на вновь вставленный узел.

Программный код соответствующего метода представлен ниже:

```
def insert_before_item(self, x, data):
    if self.start_node is None:
        print("List is empty")
        return
    else:
        n = self.start_node
        while n is not None:
            if n.item == x:
                break
            n = n.nref
        if n is None:
            print("item not in the list")
        else:
            new_node = Node(data)
            new_node.nref = n
            new_node.pref = n.pref
            if n.pref is not None:
                n.pref.nref = new_node
                n.pref = new_node
```

Можно не выделять отдельным шагом создание первого элемента и *вставить элемент в начало списка*.

Аналогично описанным выше методам, сначала проверяем, является ли список пустым. Если список пуст, используем логику, определенную в методе insert_in_emptylist(). Если список не пустой, нужно последовательно выполнить три операции:

1) для нового узла ссылка на следующий узел будет установлена через переменную *start_node*.
2) для *start_node* ссылка на предыдущий узел будет установлена на вновь вставленный узел.

3) *start_node* станет вновь вставленным узлом.

Программный код соответствующего метода представлен ниже:

```
def insert_at_start(self, data):
    if self.start_node is None:
        new_node = Node(data)
        self.start_node = new_node
        print("node inserted")
        return
    new_node = Node(data)
    new_node.nref = self.start_node
    self.start_node.pref = new_node
    self.start_node = new_node
```

Аналогичным образом можно вставлять элементы с конца списка. Эта операция похожа на вставку элемента в начало списка. Сначала нам нужно проверить, пуст ли список. Если список пуст, для вставки элемента мы можем использовать метод *insert_in_emptylist()*. Если список уже содержит какой-то элемент, мы проходим по списку, до тех пор, пока ссылка на следующий узел не станет равной *None*. Это означает, что текущий узел является последним узлом и можно добавить элемент после него.

Программный код метода, описывающий последовательность действий представлен ниже:

```
def insert_at_end(self, data):
    if self.start_node is None:
        new_node = Node(data)
        self.start_node = new_node
        return
    n = self.start_node
    while n.nref is not None:
        n = n.nref
    new_node = Node(data)
    n.nref = new_node
    new_node.pref = n
```

Удаление элементов из двусвязного списка. Как и со вставкой, существует несколько способов удаления элементов из двусвязного списка. Рассмотрим некоторые из них.

Самый простой способ удалить элемент из двусвязного списка – с начала. Для этого все, что вам нужно сделать, это установить значение начального узла на следующий узел, а затем установить для предыдущей ссылки начального узла значение None. Однако, прежде

чем мы это сделаем, нам нужно выполнить две проверки. Во-первых, нам нужно узнать, пуст ли список. Затем мы должны увидеть, сколько элементов содержит список. Если список содержит только один элемент, мы можем просто установить для начального узла значение *None*.

Программный код метода представлен ниже:

```
def delete_at_start(self):
    if self.start_node is None:
        print("The list has no element to delete")
        return
    if self.start_node.nref is None:
        self.start_node = None
        return
    self.start_node = self.start_node.nref
    self.start_prev = None
```

12.2.2. Удаление элемента двусвязного списка

Из динамических структур можно удалять элементы, так как для этого достаточно изменить значения адресных полей. Операция удаления элемента из двунаправленного списка осуществляется во многом аналогично удалению из однонаправленного списка (рис.92).

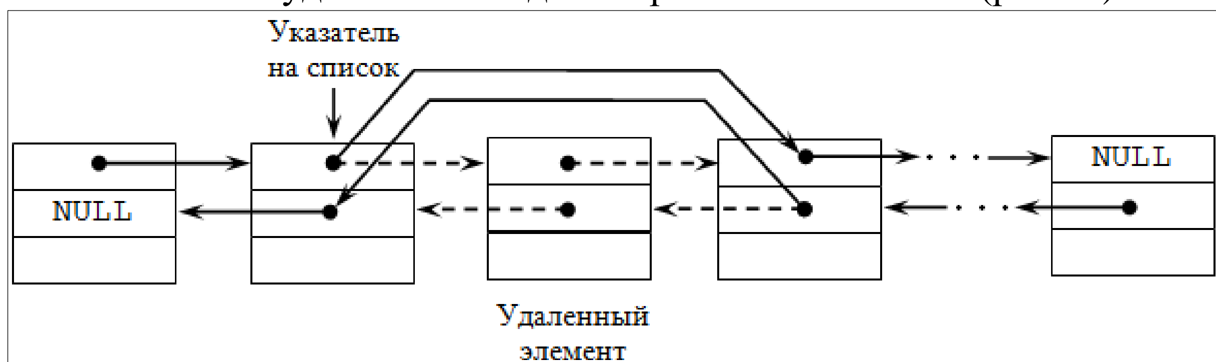


Рис.92. Графическое представление процесса удаления элемента из двусвязного списка

Чтобы *удалить элемент с конца*, мы снова проверяем, пуст ли список. Если список содержит единственный элемент, все, что нам нужно сделать, это установить для начального узла значение None. Если в списке более одного элемента, мы перебираем список до тех пор, пока не будет достигнут последний узел. Как только мы достигаем последнего узла, мы устанавливаем следующую ссылку узла, предшествующего последнему, на None, что фактически удаляет последний узел.

Программный код метода представлен ниже:

```
def delete_at_end(self):
    if self.start_node is None:
        print("The list has no element to delete")
        return
    if self.start_node.nref is None:
        self.start_node = None
        return
    n = self.start_node
    while n.nref is not None:
        n = n.nref
    n.pref.nref = None
```

Удаление элемента по значению – самая сложная из всех функций удаления в двусвязных списках, поскольку для удаления элемента по значению необходимо обработать несколько случаев. Давайте сначала посмотрим, как выглядит функция, а затем мы увидим объяснение отдельного фрагмента кода:

```
def delete_element_by_value(self, x):
    if self.start_node is None:
        print("The list has no element to delete")
        return
    if self.start_node.nref is None:
        if self.start_node.item == x:
            self.start_node = None
        else: print("Item not found")
        return
    if self.start_node.item == x:
        self.start_node = self.start_node.nref
        self.start_node.pref = None
        return
    n = self.start_node
    while n.nref is not None:
        if n.item == x:
            break
        n = n.nref
    if n.nref is not None:
        n.pref.nref = n.nref
        n.nref.pref = n.pref
    else:
        if n.item == x:
```

```
            n.pref.nref = None
        else: print("Element not found")
```

В приведенном выше программном коде мы создаем функцию *delete_element_by_value()*, которая принимает значение узла в качестве параметра и удаляет этот узел.

Вначале мы проверяем, пуст ли список. Если да, выводится соответствующее сообщение. Эта логика реализована в следующем фрагменте кода:

```
if self.start_node is None:
    print("The list has no element to delete")
    return
```

Затем мы проверяем, есть ли в списке единственный элемент, и является ли он тем элементом, который мы хотим удалить. Если этот элемент является тем, который мы хотим удалить, устанавливаем для переменной *start_node* значение *None*, это означает, что теперь в списке не будет элемента.

Если есть только один элемент, и это не тот элемент, который мы хотим удалить, отобразим сообщение о том, что удаляемый элемент не найден. Следующий фрагмент кода реализует эту логику:

```
if self.start_node.nref is None:
    if self.start_node.item == x:
        self.start_node = None
    else: print("Item not found")
    return
```

Затем мы обрабатываем случай, когда список имеет более одного элемента, но элемент, который нужно удалить, является первым элементом. В этом случае мы выполняем логику, написанную для метода *delete_at_start()*. Следующий фрагмент кода реализует:

```
if self.start_node.item == x:
    self.start_node = self.start_node.nref
    self.start_node.pref = None
    return
```

Если список содержит несколько элементов и удаляемый элемент не является первым элементом, мы просматриваем все элементы в списке, кроме последнего, и смотрим, имеет ли какой-либо из узлов значение, которое нужно удалить.

Если узел найден, выполняем последовательно следующие операции:

1) установить значение следующей ссылки предыдущего узла на следующую ссылку удаляемого узла;
2) установить предыдущее значение следующего узла на предыдущую ссылку удаляемого узла;
3) если удаляемый узел является последним узлом, для следующей ссылки узла, предшествующего последнему узлу, устанавливить значение *None*.

Следующий программный код реализует эту логику:

```
n = self.start_node
while n.nref is not None:
    if n.item == x:
        break
    n = n.nref
if n.nref is not None:
    n.pref.nref = n.nref
    n.nref.pref = n.pref
else:
    if n.item == x:
        n.pref.nref = None
    else: print("Element not found")
```

Полный код программы, демонстрирующий работу всех методов, представлен ниже:

```
class Node:
    def __init__(self, data):
        self.item = data
        self.nref = None
        self.pref = None

#класс содержит методы работы со списком
class DoublyLinkedList:
    def __init__(self):
        self.start_node = None
    #вставка элемента в пустой список
    def insert_in_emptylist(self, data):
        if self.start_node is None:
            new_node = Node(data)
            self.start_node = new_node
        else: print("list is not empty")

    #вставка элемента в начало списка
```

```
def insert_at_start(self, data):
    if self.start_node is None:
        new_node = Node(data)
        self.start_node = new_node
        print("node inserted")
        return
    new_node = Node(data)
    new_node.nref = self.start_node
    self.start_node.pref = new_node
    self.start_node = new_node

#вставка элемента в конец списка
def insert_at_end(self, data):
    if self.start_node is None:
        new_node = Node(data)
        self.start_node = new_node
        return
    n = self.start_node
    while n.nref is not None:
        n = n.nref
    new_node = Node(data)
    n.nref = new_node
    new_node.pref = n

#обход списка
def traverse_list(self):
    if self.start_node is None:
        print("List has no element")
        return
    else:
        n = self.start_node
        while n is not None:
            print(n.item, " ")
            n = n.nref
#вставка элемента после другого элемента
def insert_after_item(self, x, data):
    if self.start_node is None:
        print("List is empty")
        return
    else:
```

```
            n = self.start_node
            while n is not None:
                if n.item == x:
                    break
                n = n.nref
            if n is None:
                print("item not in the list")
            else:
                new_node = Node(data)
                new_node.pref = n
                new_node.nref = n.nref
            if n.nref is not None:
                n.nref.prev = new_node
            n.nref = new_node
    #вставка элемента перед выбранным элементом
    def insert_before_item(self, x, data):
        if self.start_node is None:
            print("List is empty")
            return
        else:
            n = self.start_node
            while n is not None:
                if n.item == x:
                    break
                n = n.nref
            if n is None:
                print("item not in the list")
            else:
                new_node = Node(data)
                new_node.nref = n
                new_node.pref = n.pref
                if n.pref is not None:
                    n.pref.nref = new_node
                    n.pref = new_node
    #удаление элемента с начала списка
    def delete_at_start(self):
        if self.start_node is None:
            print("The list has no element to de-
lete")
            return
```

```
        if self.start_node.nref is None:
            self.start_node = None
            return
        self.start_node = self.start_node.nref
        self.start_prev = None
    #удаление элемента с конца
    def delete_at_end(self):
        if self.start_node is None:
            print("The list has no element to de-
lete")
            return
        if self.start_node.nref is None:
            self.start_node = None
            return
        n = self.start_node
        while n.nref is not None:
            n = n.nref
        n.pref.nref = None
    #удаление элемента по значению
    def delete_element_by_value(self, x):
        if self.start_node is None:
            print("The list has no element to de-
lete")
            return
        if self.start_node.nref is None:
            if self.start_node.item == x:
                self.start_node = None
            else: print("Item not found")
            return
        if self.start_node.item == x:
            self.start_node = self.start_node.nref
            self.start_node.pref = None
            return
        n = self.start_node
        while n.nref is not None:
            if n.item == x:
                break
            n = n.nref
        if n.nref is not None:
            n.pref.nref = n.nref
```

```
            n.nref.pref = n.pref
        else:
            if n.item == x:
                n.pref.nref = None
            else: print("Element not found")

#основная программа
new_linked_list = DoublyLinkedList()

#создание первого узла пустого списка
new_linked_list.insert_in_emptylist(50)

#добавим элементы с начала списка
new_linked_list.insert_at_start(10)
new_linked_list.insert_at_start(5)
new_linked_list.insert_at_start(18)

#вставка элемента после выбранного элемента
new_linked_list. insert_after_item(5, 222)

#вставка элемента перед выбранным элементом
new_linked_list.insert_before_item(222,99)

#вставка нескольких элементов с конца списка
new_linked_list.insert_at_end(12)
new_linked_list.insert_at_end(121)

#вывод элементов списка/ обход списка
new_linked_list.traverse_list()
print ('|||||||||')
#удаление элемента с начала списка
new_linked_list.delete_at_start()

#удаление элемента списка по значению
new_linked_list.delete_element_by_value(50)

#удаление элемента по значению
new_linked_list.delete_at_end()

#вывод элементов списка после удаления элемента
```

```
сначала, с конца, по значению
new_linked_list.traverse_list()
```

Краткие итоги

1. Список является динамической структурой, для элементов которого определены операции включения, исключения.
2. В связанном списке элементы линейно упорядочены указателями, входящими в состав элементов списка.
3. Линейные связные списки являются простейшими динамическими структурами данных и в зависимости от организации связей делятся на однонаправленные и двунаправленные.
4. В однонаправленном (односвязном) списке каждый из элементов содержит информационную часть и указатель на следующий элемент списка. Адресное поле последнего элемента имеет значение NULL.
5. Каждый элемент списка содержит ключ, который идентифицирует этот элемент.
6. Основными операциями с однонаправленными списками, являются: создание списка; печать (просмотр) списка; вставка элемента в список; удаление элемента из списка; поиск элемента в списке; проверка пустоты списка; удаление списка.
7. В двунаправленном (двусвязном) списке каждый из элементов содержит информационную часть и два указателя на соседние элементы.
8. Основные операции, выполняемые над двунаправленным списком, те же, что и для однонаправленного списка.

12.3. Циклические списки

Циклический или круговой связанный список — это разновидность связанного списка. Узел является элементом списка и состоит из двух частей: данных и указателя *next.*

Аналогично одно- и двусвязным спискам, поле данных может быть переменной, списком или любой другой структурой, *next* — указатель на место размещения следующего узла. В отличие от одно- и двусвязного списков указатель последнего узла будет указывать на адрес головного узла.

На рисунке 93 представлена структура циклического списка.

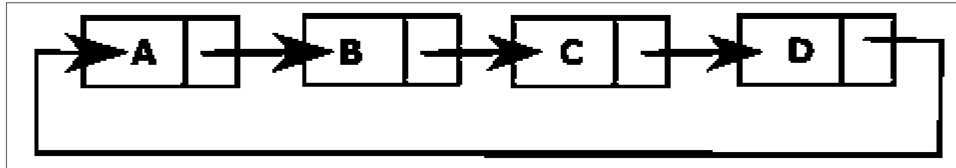


Рис.93. Структура циклического списка

Узел A представляет голову, а узел D – хвост списка. Указатель узла A ссылается на узел B, B указывает на узел C, C указывает на узел D, который указывает снова на узел A.

Опишем алгоритм работы с циклическим списком.

По аналогии с одно- и дву-направленным списками опишем структуру узла с помощью класс *Node*, содержащего два атрибута *data* и *next*. Далее добавим класс *CreateList,* содержащий: два атрибута *head* – указатель на первый элемент списка, *tail* – указатель на последний элемент списка; два метода - *add()*- добавляет узел в список, *display()*- выводит на экран значения всех узлов списка.

Программный код, демонстрирующий работу с циклическим списком:

```
#представление узла списка
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CreateList:
#задаем начальные значения указателей головы и
хвоста равными null
    def __init__(self):
        self.head = Node(None)
        self.tail = Node(None)
        self.head.next = self.tail
        self.tail.next = self.head

#добавление нового узла в конец списка
    def add(self, data):
        newNode = Node(data)
        # проверяем, пуст ли список
        if self.head.data is None:
#если список пуст, голова и хвост будут указывать
на новый узел
```

```
            self.head = newNode
            self.tail = newNode
            newNode.next = self.head
        else:
  #иначе хвост будет указывать на новый узел
            self.tail.next = newNode
            # новый узел становится новым хвостом
            self.tail = newNode;
            # хвост указывает на заголовок
            self.tail.next = self.head

            # отображает все узлы списка

    def display(self):
        current = self.head
        if self.head is None:
            print("Список пуст")
            return
        else:
            print("Узлы циклического списка ")
#печать каждого узла, переход к следующему узлу
            print(current.data),
            while (current.next != self.head):
                current = current.next
                print(current.data),

#основная программа
cl = CreateList()
# добавление значений в список
cl.add(1)
cl.add(2)
cl.add(3)
cl.add(4)
cl.display()
```

Практическая работа № 14
ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. СВЯЗНЫЕ СПИСКИ

Указания к работе. При выполнении практической работы для каждого задания требуется написать программу на языке Python 3.8, в которой выполнено формирование однонаправленного или двунаправленного списка в соответствии с постановкой задачи, ввод данных элементов списка с учетом типа информационного поля, их обработка и вывод на экран в указанном формате. Для хранения данных списков следует использовать ресурсы динамической памяти. Ввод данных осуществляется с учетом требований к входным данным, содержащихся в постановке задачи. Ограничениями на входные данные являются максимальный размер строковых данных, диапазоны числовых типов полей структуры и допустимый размер области динамической памяти в языке Python 3.8.

Порядок выполнения работы.

1. Для организации односвязного списка определить структурный тип, содержащий указатель на свой тип и поля, указанные в индивидуальном задании.
2. Определить функции добавления (удаления) элемента с начала (с конца) списка, поиск элемента по его значению, просмотра содержимого списка.
3. В функции main() создать заглавное звено списка и проверить работу функций вставки, удаления и просмотра.
4. Преобразовать односвязный список в двусвязный. Определить для двусвязного списка все указанные выше функции.
5. Преобразовать линейный список в кольцевой. Определить для кольцевого списка все указанные выше функции работы со списком.

Требования к отчету.

Отчет по практической работе должен соответствовать следующей структуре.

- титульный лист;
- словесная постановка задачи. В этом подразделе проводится полное описание задачи. Описывается суть задачи, анализ входящих в нее физических величин, область их допустимых значений, единицы их измерения, возможные ограничения, анализ

условий при которых задача имеет решение (не имеет решения), анализ ожидаемых результатов.

- математическая модель. В этом подразделе вводятся математические описания физических величин и математическое описание их взаимодействий. Цель подраздела – представить решаемую задачу в математической формулировке.
- алгоритм решения задачи. В подразделе описывается разработка структуры алгоритма, обосновывается абстракция данных, задача разбивается на подзадачи. Схема алгоритма выполняется по ЕСПД (ГОСТ 19.003-80 и ГОСТ 19.002-80).
- листинг программы. Подраздел должен содержать текст программы на языке программирования *Python 3.8*, реализованный в среде ms *PyCharm*.
- контрольный тест. Подраздел содержит наборы исходных данных и полученные в ходе выполнения программы результаты.
- выводы по лабораторной работе.

ВАРИАНТЫ ЗДАНИЙ

№ варианта	Текст задания
1	В файле хранится последовательность символов. Сформируйте из символов последовательности однонаправленный список с информационным полем типа char. Добавьте в этот список элементы с нечетными номерами.
2	В файле хранится последовательность латинских букв, оканчивающаяся точкой. Среди букв есть специальный символ *Ch*, появление которого означает отмену предыдущего символа. Учитывая вхождение этого символа, преобразуйте последовательность. Сформируйте однонаправленный список с символьным информационным полем.
3	Даны действительные числа a1, a2, ..., a2n (n >=2, заранее неизвестно и вводится с клавиатуры). Вычислите: max (min(a1, a2n), min(a3, a2n-2), ... , min (a2n-1, a2)). Сформируйте однонаправленный список с информационным полем типа float.
4	В файле хранится строка, состоящая из прописных и строчных латинских букв. В каждой паре первая буква прописная, а вторая строчная. Пары образуются в том порядке, в котором буквы следуют в строке. Исходные данные: переменная s — строка, содержащая прописные и

	строчные буквы в любом порядке. Результат: пары букв и набор букв, оставшихся без пары. Сформируйте однонаправленный список с символьным информационным полем.
5	Создайте последовательность, содержащую степени двойки, количество членов последовательности заранее неизвестно и вводится с клавиатуры. Сформируйте однонаправленный список из членов последовательности, значение которых четно.
6	Определите формулу генерации чисел, которые при делении на 5 дают в остатке 3, Сформируйте однонаправленный список из таких чисел.
7	Сформируйте однонаправленный список элементы которого содержали бы два информационных поля: значение первого типа char, значение второго задается случайным образом из диапазона целых чисел от 50 до 100.
8	В файле хранятся данные о ноутбуках вида: char model[21] // наименование size // габаритные размеры float w // вес int price // цена На основе данных из файла сформируйте однонаправленный список. Организуйте запись данных из списка обратно в файл.
9	На вход программы подаются числа. Сформируйте однонаправленный список с двумя информационными полями. В первое поместите положительные, а во второе — отрицательные числа.
10	Задана последовательность чисел. Сформируйте однонаправленный список значением информационного поля которого является удвоенное произведение элементов последовательности. Выведите элементы списка, отстоящие друг от друга на 2 шага, начиная с конца.
11	Дан текстовый файл, содержащий строки. Сформируйте однонаправленный список из строк текстового файла. Определите, сколько слов в в каждой строке начинается на гласную букву.
12	Сформируйте однонаправленный список L2, являющийся копией списка L1, начинающегося с заданного узла.

13	Дан текстовый файл, содержащий строки. Сформируйте однонаправленный список из строк текстового файла. Определите, сколько строк содержат минимальное количество символов.
14	Сформируйте однонаправленный список с информационным полем типа char. Проверьте, упорядочены ли элементы списка по алфавиту? Если нет, упорядочите элементы списка.
15	Определить симметричность произвольного текста любой длины. Текст должен оканчиваться точкой. Задачу решить с помощью двух однонаправленных списков.
16	Задана числовая последовательность $x_1, \dots, x_n$. Вычислить значение выражения $x_1 \cdot x_n + x_2 \cdot x_{n-1} + \dots + x_n \cdot x_1$. Значения элементов последовательности вводятся с клавиатуры и динамически размещаются в памяти в виде однонаправленного списка.
17	В файле хранятся данные о ноутбуках вида: char model[21] // наименование size // габаритные размеры float w // вес int price // цена На основе данных из файла сформируйте однонаправленные списки для каждой модели ноутбука.
18	В файле хранятся данные о планшетных сканерах (наименование модели и цена). Извлеките данные о сканере из описанного выше файла в однонаправленный список с информационными полями вида: наименование модели (char model [25]) и цена int price. Отсортируйте элементы списка по наименованию модели сканера.
19	Для хранения данных о планшетных сканерах описать словарь вида: char model[25] // наименование модели; int price // цена Извлеките данные о сканере из описанного выше словаря в однонаправленный список с несколькими информационными полями.
20	В файле хранятся данные о ноутбуках вида: char model[21] // наименование size // габаритные размеры

	float w // вес int price // цена На основе данных из файла сформируйте однонаправленный список только для тех ноутбуков, габаритные размеры которых не превышают 14 дюймов.

12.4. Стеки

Стек представляет собой линейную структуру данных, которая следует порядку LIFO (последним пришел, первым ушел), т. е. элементы могут быть вставлены или удалены только с одного конца.

Поддерживает следующие стандартные операции:

- push – помещает элемент в верхнюю часть стека;
- pop - удаляет и веозвращает элемент с вершины стека;
- peek – возвращает элемент на вершину стека, не удаляя его;
- size – возвращает общее количество элементов в стеке;
- isEmpty – проверяет, пуст ли стек;
- isFull – проверяет, заполнен ли стек.

Рис.94 наглядно демонстрирует работу операций pop и push.

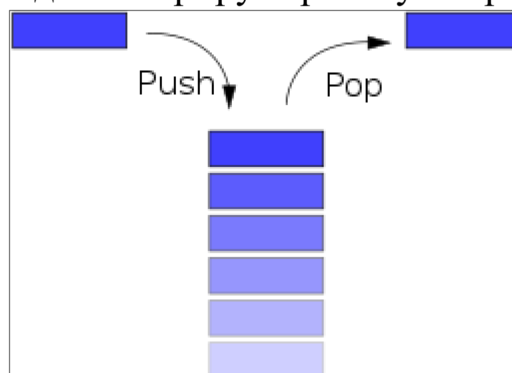


Рис.94. Алгоритм работы стека

Реализовать стек в Python можно двумя способами: в виде списка и с использованием deque объекта.

Ниже приведена программная реализация пользовательского стека в Python с использованием списка:

```
# Реализация пользовательского stack в Python
class Stack:
    # Конструктор для инициализации stack
    def __init__(self, size):
        self.arr = [None] * size
        self.capacity = size
```

```
        self.top = -1

    # Функция добавления элемента val в stack
    def push(self, val):
        if self.isFull():
            print(Переполнение стека, выход')
            exit(-1)

        print(f'Добавьте {val} стек…')
        self.top = self.top + 1
        self.arr[self.top] = val

    # Функция для извлечения верхнего элемента из
stack
    def pop(self):
        # проверка, пуст ли стек
        if self.isEmpty():
            print(Преполнение стека, выход)
            exit(-1)

        print(f'Удаление {self.peek()} из стека')

        # уменьшает размер stack на 1 и (опцио-
нально) возвращает извлеченный элемент
        top = self.arr[self.top]
        self.top = self.top - 1
        return top

    # Функция для возврата верхнего элемента стека
    def peek(self):
        if self.isEmpty():
            exit(-1)
        return self.arr[self.top]

    # Функция возврата размера стека
    def size(self):
        return self.top + 1

    # Функция для проверки, пуст стек или нет
    def isEmpty(self):
```

```
        return self.size() == 0

    # Функция проверки заполнения стека
    def isFull(self):
        return self.size() == self.capacity


if __name__ == '__main__':

    stack = Stack(3)

    stack.push(1)  # Вставка 1 в стек
    stack.push(2)  # Вставка 2 в стек

    stack.pop()  # снятие верхнего элемента (2)
    stack.pop()  # снятие верхнего элемента (1)

    stack.push(3)  # Вставка 3 в stack

    print('Top element is', stack.peek())
    print('The stack size is', stack.size())

    stack.pop()  # снятие верхнего элемента (3)

    # проверяет, пуст ли stack
    if stack.isEmpty():
        print('The stack is empty')
    else:
        print('The stack is not empty')
```

Класс Deque содержится в коллекции collection. Deque объект обозначает двустороннюю очередь – обобщение стеков и очередей, которые поддерживают добавление и удаление с любой стороны очереди в любом направлении.

Ниже приведен простой пример, демонстрирующий использование deque объекта для реализации структуры данных stack в Python:

```
# Реализация стека с использованием класса deque
from collections import deque

if __name__ == '__main__':
```

```
    stack = deque()

    print('Inserting A into the stack…')
    stack.append('A')

    print('Inserting B into the stack…')
    stack.append('B')

    print('Inserting C into the stack…')
    stack.append('C')

    print('Inserting D into the stack…')
    stack.append('D')

    print('Top element is', stack[-1])  # печатает
верхнюю часть stack (D)

    print(f'Removing {stack.pop()} from the
stack')  # снятие верхнего элемента (D)
    print(f'Removing {stack.pop()} from the
stack')  # снимает следующую вершину (C)

    # возвращает общее количество элементов в
stack.
    print('The stack size is', len(stack))

    print(f'Removing {stack.pop()} from the
stack')  # снятие верхнего элемента (B)
    print(f'Removing {stack.pop()} from the
stack')  # удаление следующей вершины (A)

    # проверяет, пуст ли stack
    if len(stack) == 0:
        print('The stack is empty')
    else:
        print('The stack is not empty')
```

12.5. Очереди

Очередь (Queue) поддерживает тот же набор операций, что и стек, но имеет противоположную семантику. Для описания очереди используется аббревиатура FIFO (First In, First Out), так как первым из очереди извлекается элемент, раньше всех в неё добавленный. Название этой структуры говорит само за себя: принцип работы совпадает с обычными очередями в магазине или на почте.

Реализация очереди похожа на реализацию стека, отличается наличием двух указателей: для первого элемента очереди (“головы”) и для последнего (“хвоста”). Структура очереди показана на рис. 95.

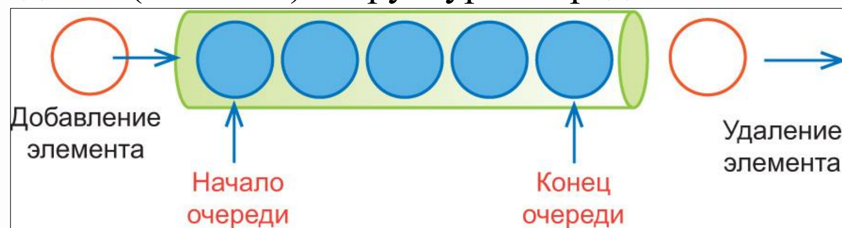


Рис.95. Структурная модель очереди

Очередь поддерживает следующие стандартные операции:

- enqueue - вставляет элемент в конец (справа) очереди;
- dequeue – удаляет элемент из передней (левой части) очереди и возвращает его;
- peek – возвращает элемент в начале очереди, не удаляя его;
- isEmpty – проверяет, пуста ли очередь;
- size – возвращает общее количество элементов, присутствующих в очереди.

Программный код ниже демонстрирует создание и работу с очередью:

```
class Queue:

    # Инициализировать очередь
    def __init__(self, size=1000):
        self.q = [None] * size  # Список для
хранения элементов queue
        self.capacity = size  # максимальная ем-
кость queue
        self.front = 0  # front указывает на
первый элемент очереди
        self.rear = -1  # rear указывает на по-
следний элемент очереди
```

```
        self.count = 0  # текущий размер очереди

 # Функция удаления переднего элемента из очере-
ди
    def dequeue(self):
        # проверка, пуста ли очередь
        if self.isEmpty():
            print('Queue Underflow!! Terminating
process.')
            exit(-1)
        x = self.q[self.front]
        print('Removing element…', x)
        self.front = (self.front + 1) %
self.capacity
        self.count = self.count - 1
        return x

# Функция добавления элемента в очередь
    def enqueue(self, value):
        # проверка на переполнение очереди
        if self.isFull():
            print('Overflow!! Terminating pro-
cess.')
            exit(-1)
        print('Inserting element…', value)
        self.rear = (self.rear + 1) %
self.capacity
        self.q[self.rear] = value
        self.count = self.count + 1

    # Функция возврата первого элемента очереди
    def peek(self):
        if self.isEmpty():
            print('Queue UnderFlow!! Terminating
process.')
            exit(-1)
        return self.q[self.front]

    # Функция возврата размера очереди
    def size(self):
```

```
        return self.count

    # Функция для проверки, пуста или нет оче-
редь
    def isEmpty(self):
        return self.size() == 0

    # Функция проверки заполнения очереди
    def isFull(self):
        return self.size() == self.capacity


if __name__ == '__main__':

    # создает очередь емкостью 5
    q = Queue(5)

    q.enqueue(1)
    q.enqueue(2)
    q.enqueue(3)

    print('The queue size is', q.size())
    print('The front element is', q.peek())
    q.dequeue()
    print('The front element is', q.peek())

    q.dequeue()
    q.dequeue()

    if q.isEmpty():
        print('The queue is empty')
    else:
        print('The queue is not empty')
```

12.6. Дек

Дек (англ. Deque — double ended queue, «двухсторонняя очередь»). Дек можно определять как двухстороннюю очередь, или как стек, имеющий два конца. Это означает, что данный вид списка позволяет добавлять элементы в начало и в конец. Тоже самое спра-

ведливо и для операции извлечения. Эта структура данных одновременно работает по двум принципам организации данных: FIFO и LIFO. Графически модель данных типа дек представлена на рис.96.

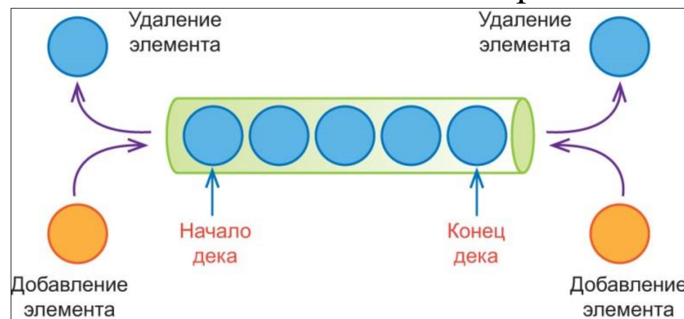


Рис.96. Структурная модель дека

Реализован в Python с помощью модуля deque библиотеки collections, содержащего встроенные методы, реализующие стандартные операции с очередями (табл.15).

Таблица 15. Стандартные операции для двусторонней очереди модуля deque

№	Операция	Описание
1	append ()	добавление элемента данных в правый конец двухсторонней очереди
2	appendleft ()	добавление элемента данных в левый конец двухсторонней очереди
3	pop ()	удаление элемента данных с правого конца двухсторонней очереди
4	popleft ()	удаление элемента данных с правого конца двухсторонней очереди
5	index (element, begin, end)	возврат первого значения индекса, указанного в параметрах
6	remove ()	удаление первого вхождения значения, указанного в качестве параметра
7	count ()	подсчет общего количества вхождений значения, указанного в качестве параметра
8	reverse ()	изменение порядка следования элементов двусторонней очереди
9	rotate ()	поворот двухсторонней очереди на число, указанное в качестве параметра. Если число отрицательное, то вращение происходит против часовой стрелки. В противном случае – по часовой стрел-

		ке.

Рассмотрим, как в Python создать и работать со структурой дек с помощью модуля deque.

Сначала импортируем необходимые библиотеки и модули:

```
# importing the deque module
# from the collections library
from collections import deque
```

Так как мы используем встроенные методы, нам не нужно создавать класс для реализации методов. В примере ниже с помощью конструктора мы создаем и инициализируем значениями двустороннюю очередь:

```
my_deque = collections.deque([10, 20, 30, 40, 50])
```

С помощью метода append () добавим число 60 в конец двусторонней очереди, т.е. справа:

```
my_deque.append(60)
```

Метод print () выведем элементы очереди на экран:

```
print(my_deque)
```

С помощью метода appendleft () добавим число 70 в начало двусторонней очереди, т.е. слева:

```
my_deque.appendleft(70)
```

Команда my_deque.pop () удалит число 60, а my_deque.popleft () удалит число 70:

```
my_deque.pop()
my_deque.popleft()
```

Полный программный код примера:

```
# импортируем библиотеку collections
# для модуля deque
import collections

# декларируем двустороннюю очередь
my_deque = collections.deque([10, 20, 30, 40, 50])

# вставляем число 60 в конец очереди
my_deque.append(60)

# печатаем результат
print("The deque after appending at right: ")
```

```
print(my_deque)

# вставляем число 70 в начало очереди
my_deque.appendleft(70)

# печатаем результат
print("The deque after appending at left: ")
print(my_deque)

# удаляем число 60 справа с конца очереди
my_deque.pop()

# печатаем результат
print("The deque after removing from right: ")
print(my_deque)

# удаляем число 70 слева с головы очереди
my_deque.popleft()

# печатаем результат
print("The deque after removing from left: ")
print(my_deque)
```

Практическая работа № 15
ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. СТЕКИ, ОЧЕРЕДИ И ДЕКИ

Указания к работе. При выполнении практической работы для каждого задания требуется написать программу на языке Python 3.8, в которой выполнено формирование динамической структуры данных в соответствии с постановкой задачи, ввод данных, их обработка и вывод на экран в указанном формате. Ввод данных осуществляется с учетом требований к входным данным, содержащихся в постановке задачи. Ограничениями на входные данные являются максимальный размер строковых данных, диапазоны числовых типов полей структуры и допустимый размер области динамической памяти в языке Python 3.8.

Порядок выполнения работы:

1. Для исходных данных, указанных в индивидуальном задании практической работы №12, осуществить их чтение в стек.
2. Определить функции добавления (удаления) элемента, поиск элемента по его значению, просмотра содержимого соответствующей динамической структуры.
3. В функции main() создать меню и проверить работу функций вставки, удаления и просмотра.
4. Преобразовать стек в очередь и дек. Опеределить для них все указанные выше функции.

Требования к отчету.

Отчет по практической работе должен соответствовать следующей структуре.

- титульный лист;
- словесная постановка задачи. В этом подразделе проводится полное описание задачи. Описывается суть задачи, анализ входящих в нее физических величин, область их допустимых значений, единицы их измерения, возможные ограничения, анализ условий при которых задача имеет решение (не имеет решения), анализ ожидаемых результатов.
- математическая модель. В этом подразделе вводятся математические описания физических величин и математическое описание их взаимодействий. Цель подраздела – представить решаемую задачу в математической формулировке.
- алгоритм решения задачи. В подразделе описывается разработка структуры алгоритма, обосновывается абстракция данных, задача разбивается на подзадачи. Схема алгоритма выполняется по ЕСПД (ГОСТ 19.003-80 и ГОСТ 19.002-80).
- листинг программы. Подраздел должен содержать текст программы на языке программирования *Python 3.8*, реализованный в среде ms *PyCharm*.
- контрольный тест. Подраздел содержит наборы исходных данных и полученные в ходе выполнения программы результаты.
- выводы по лабораторной работе.

12.7. Бинарные деревья

Структура и формирование бинарного дерева

Дерево – иерархическая структура данных, имеющая некий начальный узел (вершину), от которого можно произвести связи к другим узлам, а от них к следующим и т.д. Примерами дерева могут служить гинеологическое древо семьи, структура каталогов и файлов и т.д.

Начальный узел дерева называется корнем (или родителем). Другие элементы дерева называются дочерними узлами (потомками или детьми), причем узел слева называется левым, а узел справа называется правым. Если у вершины нет ни одного потомка, она называется листовой. Таким образом, дерево является упорядоченной ориентированной структурой данных.

Чаще всего в практике программирования используют бинарные деревья.

Дерево, в котором каждый узел имеет не более двух потомков, называется бинарным или двоичным. Графически модель бинарного дерева представлена на рис.97.

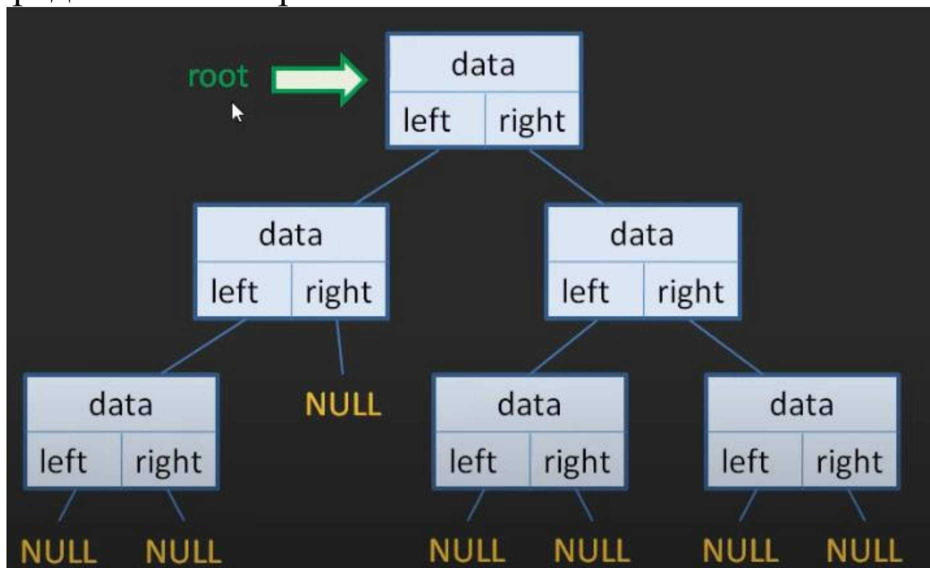


Рис.97. Структура бинарного дерева

Если у вершины предполагается множество потомков, в практике программирования задачу зачастую сводят к бинарному дереву или какой-либо другой структуре данных. Поэтому в курсе дисциплины мы рассмотрим только бинарные деревья. Рассмотрим его структуру.

Узел такого дерева содержит как минимум три поля: поле data для хранения каких-либо данных, а также указатели left и right, ссы-

лающиеся на левую и правую вершины соответственно. Иногда в структуру бинарного дерева добавляют еще одно поле – ссылку на родительский узел.

На вершину дерева, которая называется корнем, всегда ведет указатель и он обычно называется root. Через этот указатель происходит вся работа с бинарным деревом, а именно добавление, удаление и поиск элементов, обход всех узлов и т.д.. То есть из корневого узла можно пройти до любого другого узла дерева.

Количество вершин, которые надо пройти до какого-либо узла определяет уровень дерева. Иерархия уровней представлена на рис.98.

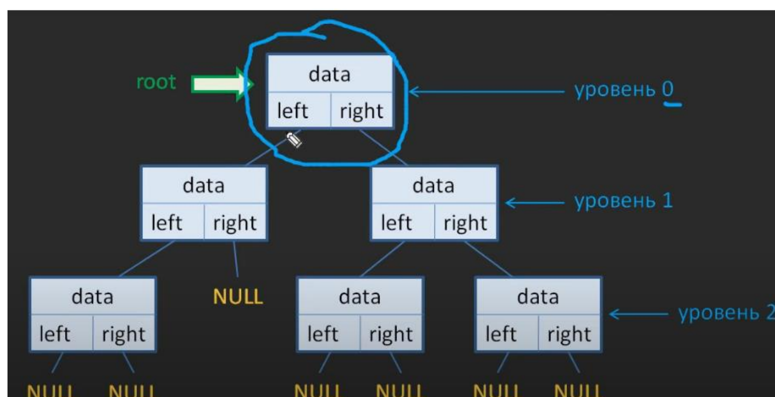


Рис.98. Иерархия уровней бинарного дерева

Если дерево не содержит ни одной вершины, указатель root принимает значение null, это же касается указателей left и right.

Разберем, как можно формировать бинарное дерево.

Обычно новые узлы добавляются к свободному указателю, т.е. тому, который принимает значение null. То есть первая вершина всегда добавляется как корневая, root=null, дальше мы можем решать, куда добавить следующую вершину: либо слева, либо справа. Строго говоря, никаких ограничений здесь нет, алгоритм добавления узла может быть самым разным, исходя из поставленной задачи. Например, если нам нужно сформировать гинеологическое древо, то очевидно, что корневая вершина – это сам индивид, вершины-потомки – это его папа, мама, дедушки и бабушки.

Однако, существует целый класс задач при котором в вершинах дерева хранятся значения, которые можно сравнивать. В этом случае добавление вершины осуществляется по следующим правилам:

– если добавляемое значение меньше значения в родительском узле, то новая вершина добавляется в левую ветвь, иначе – в правую;

– если добавляемое значение уже присутствует в дереве, то оно игнорируется, дубли в бинарном дереве отсутствуют.

Рассмотрим, как работает добавление узла согласно этим правилам. Пусть у нас имеется последовательность целых чисел 10, 5, 7, 16, 13, 2, 20. Нам необходимо добавить их в бинарное дерево. Первое число 10 добавляется в корень дерева. Второе число 5, согласно первому правилу, добавляется в левый узел относительно корня. Третье число 7, 7 < 10, поэтому мы идем по левой ветви, но в ней уже есть узел, значение которого равно 5. Сравниваем числа 5 и 7, 7 > 5, следовательно, число 7 добавляется в правую ветвь узла со значением 5 и т.д.

Следующее число – 16, его значение сравнивается с корнем дерева, 16 > 10, следовательно, мы переходим на правую ветвь, она свободна, 16 добавляется в правый узел относительно корня дерева.

Следующее число 13, 13>10, уходим на правую ветвь дерева, так как 13 <16, оно размещается слева относительно узла 16 и т.д. Наглядно процесс формирования дерева представлен на рис.99.

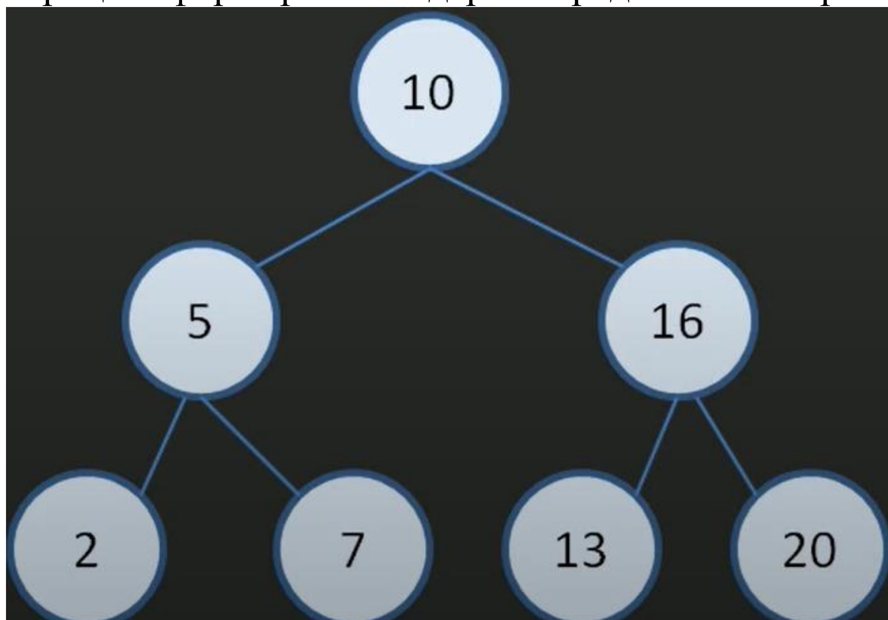


Рис.99. Наглядное представление процесса формирования бинарного дерева

Благодаря такому подходу к формированию бинарного дерева, в левой ветви у нас будут находиться вершины со значениями меньше корневой, а в правой – больше корневой. Этот вывод справедлив для любого поддерева, так в поддереве с корневой вершиной 5, слева от нее располагаются вершины с меньшими, а справа – с большими относительно нее значениями.

Что это дает?

Появляются несколько интересных эффектов.

Первый – ускорение поиска заданного значения. Например, нам надо определить, хранится ли число 2 в заданном массиве? Если бы мы хранили числа в обычном массиве, нам бы пришлось сравнивать число 2 со всеми числами, начиная с первого. Временная сложность алгоритма поиска составила бы O(n), где n – число элементов в массиве.

При использовании двоичного дерева поиск устроен иначе. Мы начинаем с корневой вершины. Смотрим, равно ли ее значение 2? Нет, это 10, $2 < 10$, значит, мы продолжаем поиск в левом поддереве, и все вершины правого поддерева отбрасываем, за счет этого происходит резкое ускорение поиска нужного значения.

Продолжаем поиск. Мы начинаем с корневой вершины левого поддерева. Смотрим, равно ли ее значение 2? Нет, это 5, $2 < 5$, значит, мы продолжаем поиск в левом ветви этого поддерева и все вершины правой ветви отбрасываем.

Переходим к следующей вершине. Смотрим, равно ли ее значение 2? Да, прерываем поиск. В отличие от обычного массива, мы провели 3 сравнения вместо 7. Временная сложность алгоритма – O(log n).

Рассмотрим работу алгоритма поиска в бинарном дереве в случае отсутствия искомого значения. Например, будем искать число 11.

Мы начинаем с корневой вершины, $11 \neq 10, 11 > 10$, переходим к следующей вершине 16, $11 \neq 16, 11 < 16$, переходим в левую ветвь. $11 \neq 13$ и далее никаких потомков нет, т.е. мы дошли до листовой вершины и не нашли нужное значение. В этом случае поиск завершен, нужного значения нет в бинарном дереве.

Второй эффект получается, если на вход подаются данные, упорядоченные по убыванию или возрастанию. В этом случае деревья выглядят иначе – фактически они вырождаются в односвязный список (рис.100) и объем вычислений для поиска заданного элемента будет составлять O(n), также как в обычных списках, поэтому выигрыша во времени нет.

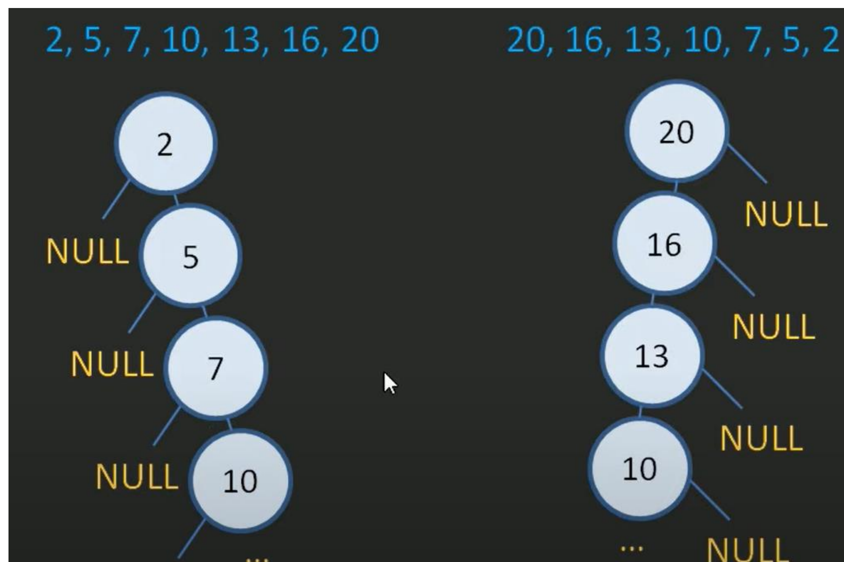


Рис.100. Вырожденные деревья

Такие вырожденные деревья и подобные им называются несбалансированными.

Дерево называется сбалансированным, если в нем поддеревья от одной вершины отличаются не более, чем на один уровень. Именно в сбалансированных деревьях поиск элемента осуществляется за минимальное число шагов. Поэтому на практике стараются строить деревья, близкие к сбалансированому. Наглядно разница между сбалансированными и не сбалансированными деревьями показана на рис.101.

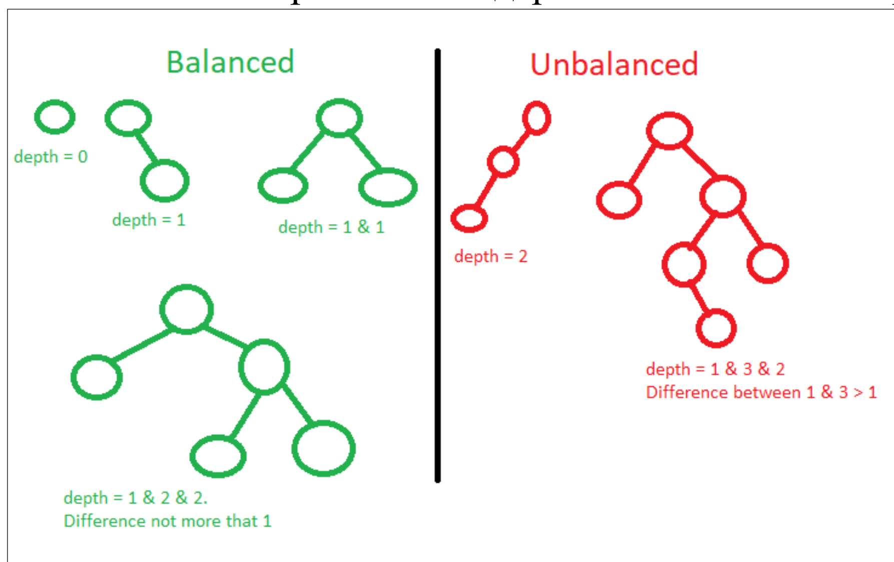


Рис.101 Сбалансированные и не сбалансированные деревья

По аналогии со связными списками и очередями создадим класс `Node`, описывающий вершину бинарного дерева. Этот класс содержит конструктор, в который мы передадим данные о вершине (значения узла и указателей на левую и правую ветвь):

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = self.right = None
```

Когда мы будем создавать объекты этого класса, по сути, будем создавать листовые вершины, так как потомки у этого класса отсутствуют.

Для работы с деревом определим второй класс Tree. У этого класса также будет конструктор, в котором мы укажем ссылку на корень дерева. По умолчанию ее значение равно null:

```
class Tree:
    def __init__(self):
        self.root = None
```

Пропишем набор методов класса Tree. Метод append () – для добавления вершин, параметр метода – вершина дерева, объект obj класса Node. Метод будет работать согласно описанным выше правилам.

Если указатель root принимает значение None, в бинарном дереве нет ни одного объекта, вершина добавляется как корневая, поэтому root должен ссылаться на объект obj. Вернем этот объект:

```
def append(self, obj):
    if self.root is None:
        self.root = obj
        return obj
```

Если root $\neq$ None, то мы должны найти вершину (родителя), к которой затем добавим новую вершину:

```
s, p, fl_find = self.__find(self.root, None,
obj.data)
```

где s – вершина, к которой мы добавляем новую вершину, p – указатель на следующую вершину, fl_find – флаг, если значение флага равно True, такая вершина уже есть, False – такой вершины нет.

Если добавляемое значение меньше того, что хранится в переменной s, мы добавляем его в левую ветвь, иначе – в правую и возвращаем объект, который добавили:

```
        if not fl_find and s:
            if obj.data < s.data:
                s.left = obj
            else:
                s.right = obj
```

```
        return obj
```

Полный текст кода метода append ():

```
def append(self, obj):
        if self.root is None:
            self.root = obj
            return obj

        s, p, fl_find = self.__find(self.root,
None, obj.data)

        if not fl_find and s:
            if obj.data < s.data:
                s.left = obj
            else:
                s.right = obj

        return obj
```

Для поиска вершины, к которой можно присоединить новую вершину метод append () должен получить ссылку на родителя. С этой целью пропишем специальный метод find() поиска такой родительской вершины. В этот метод мы передадим:

- корень дерева, т.к. вершина ищется от корня;
- ссылку на родительскую вершину для корня (None);
- данные, записанные в объект obj, относительно которых мы и будем искать ту вершину, к которой потом присоединяем новую:

```
s, p, fl_find = self.__find(self.root, None,
obj.data)
```

Пропишем метод find (). Так как мы каждый раз обращаемся к предыдущей вершине – это рекурсивный метод. Как он работает?

Поиск начинается с корневой вершины.Если значение value равно значению в текущей вершине, то добавлять такое значение не нужно. В этом случае мы вернем объект node, родительский объект parent, флаг того, нужно ли добавлять вершину:

```
if value == node.data:
    return node, parent, True
```

Если значение value меньше значения корневой вершины, значит надо двигаться по левой ветви. Предварительно необходимо про-

верить, существует ли она? В качестве параметров в метод передадим объект left, родительский объект node и значение value:

```
if value < node.data:
        if node.left:
            return self.__find(node.left, node,
value)
```

Иначе – по правой. Аналогично проверяем, существует ли она:

```
if value > node.data:
    if node.right:
        return self.__find(node.right, node, val-
ue)
```

Если все рекурсии проходят, метод вернет объекты node, parent и значение флага False. Это означает, что нужная вершина найдена, и к ней можно присоединить новую вершину:

```
return node, parent, False
```

Полный текст программного кода метода find ():

```
def __find(self, node, parent, value):
    if node is None:
        return None, parent, False

    if value == node.data:
        return node, parent, True

    if value < node.data:
        if node.left:
            return self.__find(node.left, node,
value)

    if value > node.data:
        if node.right:
            return self.__find(node.right, node,
value)

    return node, parent, False
```

Следующий метод show_tree() будет отображать сформированное бинарное дерево. Параметр метода – объект node. Если node принимает значение None, то отображать нечего – дерево пустое или мы дошли до листовой вершины, возвращаемся в программу через оператор return:

```
if node is None:
```

```
    return
```

Далее по рекурсии вызываем этот метод и отображаем текущее значение левой ветви:

```
self.show_tree(node.left)
        print(node.data)
```

Рекурсивно вызываем метод и переходим на правую ветвь:

```
self.show_tree(node. right)
```

Поясним алгоритм работы метода. Сначала мы доходим по левой ветви до последнего левого листового узла, в нашем случае это 2. Указатели на левую и правую ветвь у этой вершины равны None, поэтому поднимаемся на уровень выше и попадаем в вершину со значением 3. Правый указатель у этой вершины равен None, поэтому мы поднимаемся на уровень выше и попадаем в вершину со значением 5, переходим в правую ветвь. Двигаясь по левой ветке правой ветви, доходим до последнего левого листового узла, его значение равно 6. Затем последовательно начинаем подниматься на более высокие уровни (рис.102)

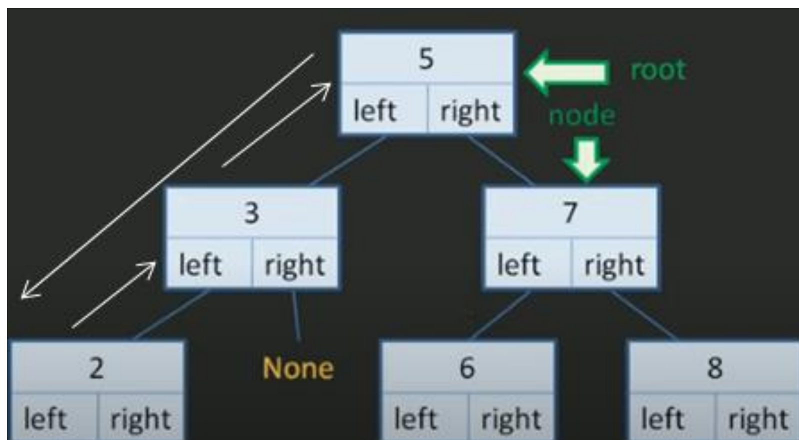


Рис.102. Алгоритм метода show_tree()

Метод show_tree() реализует известный алгоритм обхода двоичного дерева в глубину. Подробно он будет рассмотрен в следующем параграфе. Полный код метода представлен ниже:

```
def show_tree(self, node):
        if node is None:
            return
        self.show_tree(node.left)
        print(node.data)
        self.show_tree(node. right)
```

Полный листинг программного кода на Python, реализующий структуру и процесс формирования бинарного дерева представлен ниже:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = self.right = None

class Tree:
    def __init__(self):
        self.root = None

    def __find(self, node, parent, value):
        if node is None:
            return None, parent, False

        if value == node.data:
            return node, parent, True

        if value < node.data:
            if node.left:
                return self.__find(node.left,
node, value)

        if value > node.data:
            if node.right:
                return self.__find(node.right,
node, value)

        return node, parent, False

    def append(self, obj):
        if self.root is None:
            self.root = obj
            return obj

        s, p, fl_find = self.__find(self.root,
None, obj.data)

        if not fl_find and s:
            if obj.data < s.data:
                s.left = obj
            else:
```

```
                s.right = obj

        return obj

    def show_tree(self, node):
        if node is None:
            return
        self.show_tree(node.right)
        print(node.data)
        self.show_tree(node.left)

#основная программа
#список из добавляемых значений
v = [10, 5, 7, 16, 13, 2, 20]

#создаем объект класса бинарное дерево
t = Tree()

#используем цикл для чтения списка и добавления
считанного значения в узел дерева
for x in v:
    t.append(Node(x))

#отобразим дерево, начиная с его корня
t.show_tree(t.root)
```

Значения будут выведены в порядке возрастания

2
5
7
10
15
16
20

12.7.1. Способы обхода и удаления вершин бинарного дерева

В практике программирования существует два основных способа обхода вершин дерева: в ширину и в глубину. Рассмотрим эти подходы.

При обходе в ширину перебор вершин дерева выполняется по уровням слева направо.

То есть мы сначала берем корневую вершину и отображаем ее значение, потом узлы следующего уровня в порядке слева направо и так до тех пор, пока узлы существуют. Графически процесс показан на рис.103

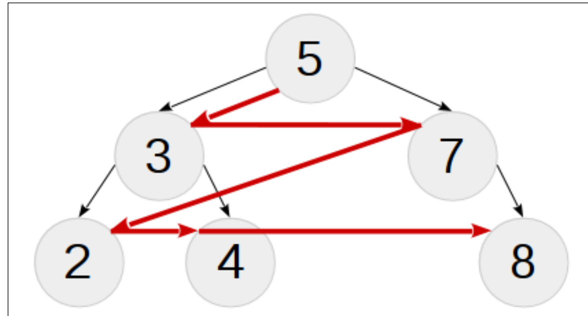


Рис.103. Обход дерева в ширину

При обходе дерева в глубину мы сначала доходим до листовой левой вершины и обрабатываем ее, затем переходим на уровень выше и производим обработку вершины этого уровня, затем пытаемся пройти по другой ветви этой вершины при ее наличии (рис.104).

Рассмотрим более подробно, как работают эти алгоритмы. Начнем с алгоритма обхода в ширину (рис.104).

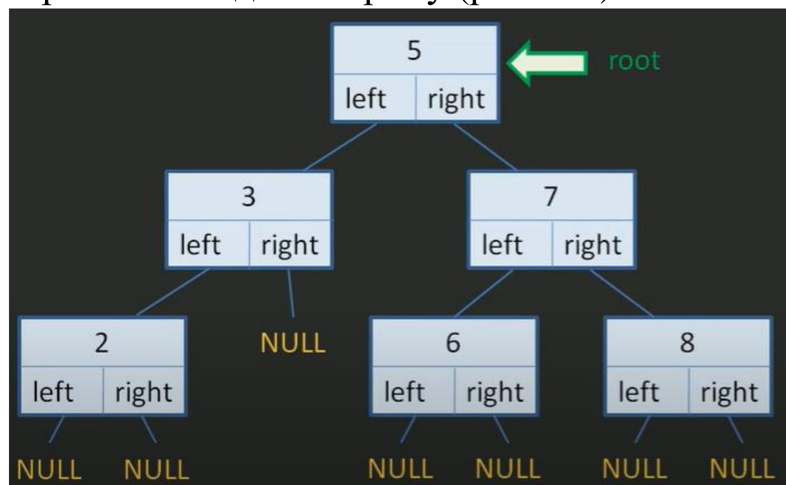


Рис.104. Наглядное представление первого шага алгоритма обхода в ширину

Как видно из рис.104, изначально у нас имеется указатель root, который всегда ссылается на корневую вершину.

Сформируем еще один временный указатель p, который изначально будет ссылаться на корневую вершину. Сформируем список v, состоящий из вершин текущего уровня. На первом шаге он будет содержать указатель на корневую вершину: v=[p].

Далее организуем цикл while и будем отрабатывать его до тех пор, пока список не станет пустым (рис.105).

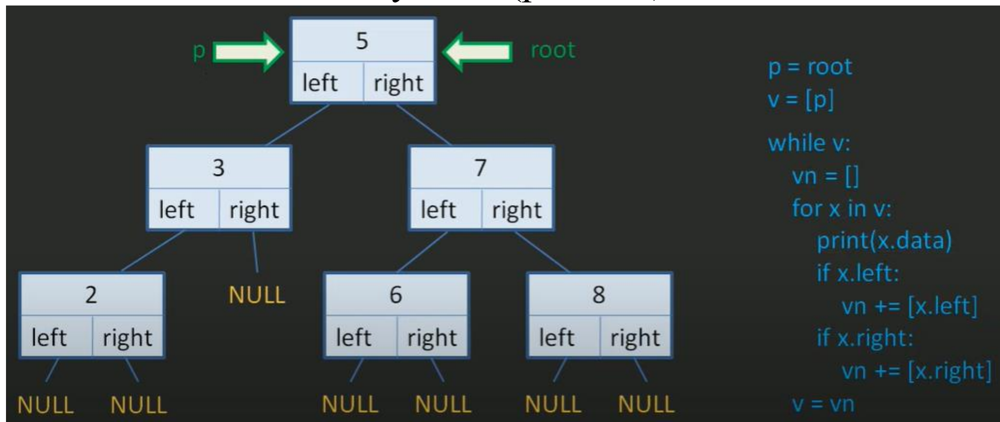


Рис.105. Наглядное представление шагов алгоритма обхода в ширину

Внутри цикла while мы создаем еще один список vn для узлов следующего уровня. Далее мы перебираем список v, выводим на экран значения узлов (поле data) и параллельно формируем вершины следующего уровня, добавляя их в список vn.

Когда v ссылается на корневую вершину, на экране отобразится число 5. Будет сформирован новый список vn, состоящий из указателей вершин следующего уровня, в нашем случае это узлы со значениями 3 и 7. Затем командой v=vn указатели на эти вершины записываются в список v, список vn очищается, а список v становится непустым, цикл while повторяется. Так продолжается до тех пор, пока мы не дойдем до листовых узлов. В нашем случае – до третьего уровня, содержащего числа 2, 6 и 8.

Таким образом мы обойдем все узлы бинарного дерева ровно один раз, значения будут отображены в следующем порядке: 5, 3, 7, 2, 6, 8.

Рассмотрим алгоритм обхода бинарного дерева в глубину. В предыдущем параграфе на его основе был реализован метод вывода на экран вершин бинарного дерева. Рассмотрим ее код более подробно.

Запускается функция из основной программы, начиная от корня дерева: *t.show_tree(t.root)*, поэтому изначально параметр node = root.

Далее мы проверяем, если эта вершина существует, тогда условие *if node is None* не срабатывает, и мы переходим к блоку операторов:

self.show_tree(node. left)
print(node.data)
self.show_tree(node. right)

Что происходит в этом блоке?

Рекурсивно вызывая функцию show_tree (), в качестве параметра мы передаем ей левого потомка командой self.show_tree(node.left). Далее функция снова рекурсивно вызывается командой self.show_tree(node.left) и так будет происходить до тех пор, пока мы не дойдем по левой ветви до левой листовой вершины.

В этом случае параметр node будет ссылаться на None. Это означает, что дальше идти некуда, сработает условие if node is None, которое не даст выполняться рекурсивному вызову self.show_tree(node. left) и мы перейдем к функции print(node.data), которая напечатает значение последнего левого листового узла – 2. Затем функция снова будет рекурсивно вызвана для правого потомка командой self.show_tree(node. right). Так как у последнего левого листового узла нет правого потомка, для этого узла функция show_tree ()завершит свою работу. По рекурсии произойдет откат к предыдущему узлу со значением 3.

Для этого узла сработает функция print(node.data) и на экране мы увидим значение 3. Затем функция будет рекурсивно вызвана для правого потомка этого узла командой self.show_tree(node. right). Правого потомка у этого узла нет, поэтому по рекурсии произойдет откат назад к корневой вершине со значением 5. Для этого корневого узла также отработает функция print(node.data) и мы увидим на экране значение 5. Далее пытаемся пройти по правой ветви командой self.show_tree(node. right). Правая ветвь у этого узла существует, выводим значение этого узла на экран командой print (node.data). Программный код метода представлен ниже:

```
def show_tree(self, node):
    if node is None:
        return
    self.show_tree(node.left)
    print(node.data)
    self.show_tree(node. right)
```

Практическая работа № 16
ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. БИНАРНЫЕ ДЕРЕВЬЯ

Указания к работе. При выполнении практической работы для каждого задания требуется написать программу на языке Python 3.8, в которой выполнено формирование динамической структуры данных в соответствии с постановкой задачи, ввод данных, их обработка и вывод на экран в указанном формате. Ввод данных осуществляется с учетом требований к входным данным, содержащихся в постановке задачи. Ограничениями на входные данные являются максимальный размер строковых данных, диапазоны числовых типов полей структуры и допустимый размер области динамической памяти в языке Python 3.8.

Порядок выполнения работы:

1. Для исходных данных, указанных в индивидуальном задании практической работы №12, осуществить их чтение в бинарное дерево.
2. Определить функции добавления (удаления) элемента, поиск элемента по его значению, просмотра содержимого соответствующей динамической структуры.
3. В функции main() создать меню и проверить работу функций вставки, удаления и просмотра.

Требования к отчету.

Отчет по практической работе должен соответствовать следующей структуре.

- титульный лист;
- словесная постановка задачи. В этом подразделе проводится полное описание задачи. Описывается суть задачи, анализ входящих в нее физических величин, область их допустимых значений, единицы их измерения, возможные ограничения, анализ условий при которых задача имеет решение (не имеет решения), анализ ожидаемых результатов.
- математическая модель. В этом подразделе вводятся математические описания физических величин и математическое описание их взаимодействий. Цель подраздела – представить решаемую задачу в математической формулировке.
- алгоритм решения задачи. В подразделе описывается разработка структуры алгоритма, обосновывается абстракция данных, зада-

ча разбивается на подзадачи. Схема алгоритма выполняется по ЕСПД (ГОСТ 19.003-80 и ГОСТ 19.002-80).
- листинг программы. Подраздел должен содержать текст программы на языке программирования *Python 3.8*, реализованный в среде ms *PyCharm*.
- контрольный тест. Подраздел содержит наборы исходных данных и полученные в ходе выполнения программы результаты.
- выводы по лабораторной работе.

12.8. Графы

В реальной жизни мы сталкиваемся с графами повсюду. Настало время рассказать о них поподробнее.

Географическиекарты. Какой маршрут из Москвы в Лондон требует наименьших расходов? Какой требует наименьшего времени? Требуется информация о связях между городами и о стоимости этих связей.

Микросхемы. Транзисторы, резисторы и конденсаторы связаны между собой проводниками. Есть ли короткие замыкания в системе? Можно ли так переставить компоненты, чтобы не было пересечения проводников?

Расписания задач. Одна задача не может быть начата без решения других, следовательно, имеются связи между задачами. Как составить график решения задач так, чтобы весь процесс завершился за наименьшее время?

Компьютерные сети. Узлы конечные устройства, компьютеры, планшеты, телефоны, коммутаторы, маршрутизаторы... Каждая связь обладает свойствами латентности и пропускной способности. По какому маршруту послать сообщение, чтобы оно было доставлено до адресата за наименьшеевремя? Есть ли в сети ¾ критические узлы, отказ которых приведёт к разделению сети на несвязные компоненты?

Структура программы. Узлы функции в программе. Связи может ли одна функция вызвать другую (статический анализ) или что она вызовет в процессе исполнения программы (динамический анализ). Чтобы узнать, какие ресурсы потребуется выделять системе, требуется граф исполнения программы.

Чтобы узнать, какие ресурсы потребуется выделять, требуется граф исполнения программы.

В математике, **граф** — это абстрактное представление множества объектов и связей между ними. Графом называют пару (V, E), где V это множество **вершин**, а E множество пар, каждая из которых представляет собой связь, пары называют **рёбрами**.

12.8.1. Графы: основные термины

Определение 1. Ориентированный граф: G=(V, E) есть пара из V – конечного множества и E – подмножества множества $V * V$. обозначается как $(v_i,\ v_i)$.

Определение 2. Вершины графа: элементы множества V (vertex, vertices).

Определение 3. Ребра графа: элементы множества E (edges).

Определение 4. Неориентированный граф: ребра есть неупорядоченные пары.

Определение 5. Петля: ребро из вершины v_1в вершину v_2, где $v_1 = v_2$ (рис.106).

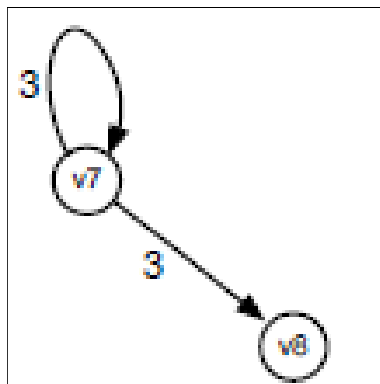


Рис.106. Петля в графе

Определение 6. Смежные вершины: вершина v_j смежна вершине v_i если имеется ребро $(v_i,\ v_j)$.

Определение 7. Множество смежных вершин: обозначается $Ad_j[v]$.

Определение 8. Степень вершины: величина $|Ad_j[v]|$.

Определение 9. Путь из v_0 в v_n: последовательность ребер, таких, что $e_1 = (v_0, v_1), e_2 = (v_1, v_2), \ldots, e_n = (v_{n-1}, v_n)$.

Определение 10. Простой путь: путь, в котором все вершины попарно различны.

Определение 11. Длина пути: количество n ребер в пути.

Определение 12. Цикл: путь, в котором $v_0 = v_n$.

Определение 13. Неориентированный связный граф: для любой пары вершин существует путь между ними.

Определение 14. Связная компонента вершины v: множество вершин неориентированного графа, до которых существует путь из v.

Определение 15. Расстояние между $\sigma(v_i, v_j)$: длина кратчайшего пути из v_i в v_j:

$\sigma(u, v) = 0 \Leftrightarrow u = v,$

$\sigma(u, v) \leq \sigma(u, v') + \sigma(u', v)$.

Поясним некоторые определения. В ориентированном графе, связи являются направленными, то есть, пары (a, b) и (b, a) – это две разные связи. В свою очередь в неориентированном графе, связи ненаправленные, и поэтому если существует связь (a, b) то существует связь (b, a). Графически неориентированный и ориентированный графы представлены на рис.107 а,б.

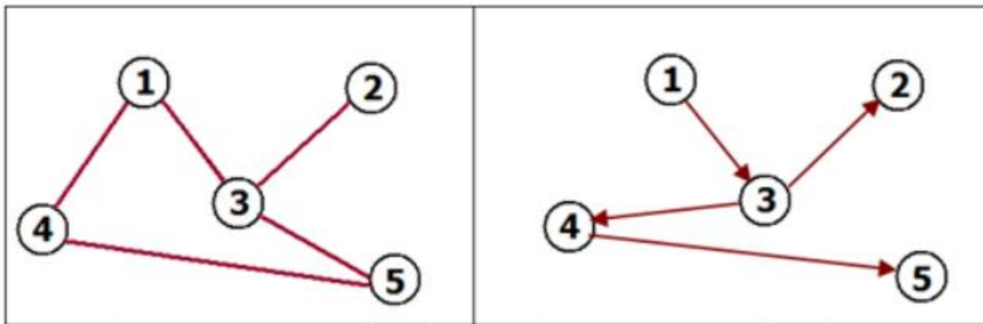


Рис.107а. Неориентированный граф Рис.107б. Ориентированный граф

Примером неориентированного графа в жизни является соседство. Если (1) сосед (3), то (3) сосед (1).

Примером ориентированного графа являются ссылки. Например, сайт (1) может ссылаться на сайт (3). Но совсем не обязательно, хотя возможно, что сайт (3) ссылается на сайт (1).

Степень вершины ($|Ad_j[v]|$) может быть входящая и исходящая. Для неориентированных графов входящая степень равна исходящей. Входящая степень вершины V – это количество ребер вида (input, **v**), то есть количество ребер которые «входят» в V. Исходящая степень вершины V – это количество ребер вида (**v** , out), то есть количество ребер которые «выходят» из V.

Путь в графе в зависимости от вида графа путь может быть ориентированным или неориентированным. На рис 107.а, путем является, например, последовательность [(1), (4), (5)]. На рис 107.б, [(1), (3), (4), (5)]. Также можно указать и другие пути.

Графы обладают множеством других разных свойств, например, они могут быть связными, двудольными, полными, с взвешенными

ребрами и т.д. Эти свойства мы рассмотрим далее, по мере изложения материала.

12.8.2. Способы представления графа в памяти

Существует два способа представления графа: в виде списков смежности, в виде матрицы смежности и в виде множеств. Оба способа подходят для представления ориентированных и неориентированных графов.

Матрица смежности. Этот способ является удобным для представления **плотных** графов, в которых количество рёбер (|E|) примерно равно квадрату количества вершин ($|V|^2$). В данном представлении мы заполняем матрицу A размером |V| x |V| следущим образом: если существует ребро из вершины i в вершину j, A[i][j] = 1, иначе A[i][j] = 0 (рис.)

Данный способ подходит как для ориентированных, так и для неориентированных графов. Для неориентированных графов матрица A является симметричной (A[i][j] == A[j][i]), т.к. если существует ребро между вершинами i и j, то оно является и ребром из вершины i в вершину j, и ребром из j в i (рис а, б).

Если граф невзвешенный (каждое ребро и/или вершина не имеет веса), наличие связи обозначается единицей или логической истиной. Отсутствие – нулем или логической ложью. Матрица смежности для графов и сами графы представлены на рис.108 абвг.

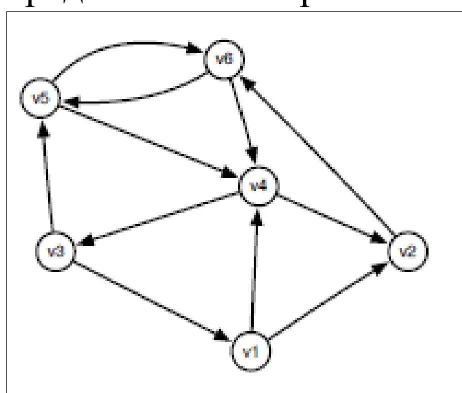


Рис.108а. Ориентированный граф

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	0	1
3	1	0	0	0	1	0
4	0	1	1	0	0	0
5	0	0	1	1	0	1
6	0	0	0	1	1	0

Рис.108б. Матрица смежности для ориентированного графа

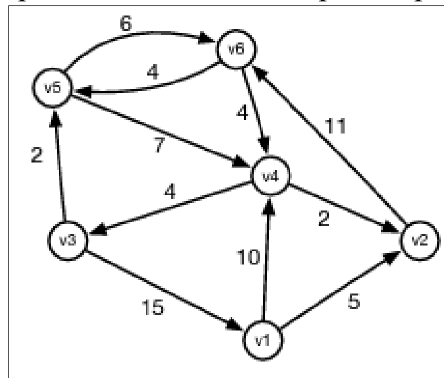


Рис.108в. Ориентированный взвешенный граф

	1	2	3	4	5	6
1	0	5	0	10	0	0
2	0	0	0	0	0	11
3	15	0	0	0	2	0
4	0	2	4	0	0	0
5	0	0	0	7	0	6
6	0	0	0	4	4	0

Рис.108г. Матрица смежности для взвешенного ориентированного графа

В случае неориентированного графа матрица смежности позволяет сократить размер используемой памяти практически в два раза, храня элементы только в верхней части матрицы, над главной диагональю. Можно также быстро проверить, есть ли ребро между вершинами V и U, просто посмотрев в ячейку A[v][u]. С другой стороны этот способ очень громоздкий, так как требует O ($|V|^2$) памяти для хранения матрицы. Кроме этого, матрица смежности не позволяет описывать графы, содержащие кратные рёбра (мультирёбра).

Множества смежности. В этом случае для каждого узла указывается множество смежных с ним узлов или соседей. Данный способ представления больше подходит для разреженных графов, таких, у которых количество рёбер гораздо меньше, чем количество вершин в квадрате ($|E| << |V|^2$). Используется массив Adj содержащий |V| мно-

жеств. В каждом множестве Adj[v] содержатся все вершины u, так что между v и u есть ребро. Для графа рис а, множества смежности будут выглядеть следующим образом:
v_1: {2, 4}
v_2: {6}
v_3: {1, 5}
v_4: {2, 3}
v_5: {3, 4, 6}
v_6: {4, 5}

Память, требуемая для представления списков смежности, равна O (|E| + |V|), что является лучшим показателем в сравнении с матрицами смежности для разреженных графов.

Для взвешенного графа элементы множеств смежности есть пары, один элемент которых – номер смежного узла, другой – вес связи. Для графа рис в, множества смежности будут выглядеть следующим образом:
v_1: {(2, 5), (4, 10)}
v_2: {(6, 11)}
v_3: {(1, 15), (5, 2)}
v_4: {(2, 2), (3, 4)}
v_5: {(4, 7), (6, 6)}
v_6: {(4, 4), (5, 4)}

Списки смежности. Для некоторых алгоритмов оказывается достаточным, если граф представлен списка троек: {откуда, куда, стоимость}. Мультирёбра в этом представлении вполне возможны. Главный недостаток этого способа представления заключается в отсутствии быстрого способа проверки существования ребра (u, v). Для графа рис в, списки смежности будут выглядеть следующим образом: {{1, 2, 5}, {1, 4, 10}, {2, 6, 11}, {3, 1, 15}, {3, 5, 2}, {4, 2, 2}, {5, 4, 7}, {5, 6, 6}, {6, 4, 4}, {6, 5, 4}}.

Где еще применяются графы? В первую очередь, в социальных сетях. Вершинами графа являются люди, а ребрами отношения между ними, например, дружбы. Граф будет неориентированным, в случае, если я могу дружить только с теми, кто дружит со мной. Граф будет ориентированным в случае, если можно добавить человека в друзья, без того, чтобы он добавлял вас. Если и он добавит вас в друзья, вы будете «взаимными» друзьями.

Ещё одно из применений графа – рейтинг сайтов. Представим, что вы хотите сделать поисковую систему. В этом случае вам необхо-

димо вести учет количества ссылок сайтов друг на друга. Например, сайт A ссылается на сайт B, а сайт B может ссылаться на сайт A, а может и нет. В этом случае необходимо учесть, сколько ссылок ссылается на сайт A и при этом учесть, сколько сайтов ссылается на сайт B, который ссылается на сайт A. Получится матрица смежности ссылок. На ее основе можно организовать систему подсчёта рейтинга сайтов.

Пример реализации графа (рис.109) с помощью множеств смежности на Python представлен ниже:

```
a, b, c, d, e, f, g, h = range(8)
N = [
    {b, c, d, e, f}, # a
    {c, e}, # b
    {d}, # c
    {e}, # d
    {f}, # e
    {c, g, h}, # f
    {f, h}, # g
    {f, g} # h
]
```

где a, b, c, d, e, f, g, h – вершины графа, пронумерованные от 0 до 7. В списке N в виде элементов множества перечислены вершины, связанные с конкретной вершиной. Например, с вершиной a связаны вершины b, c, d, e, f.

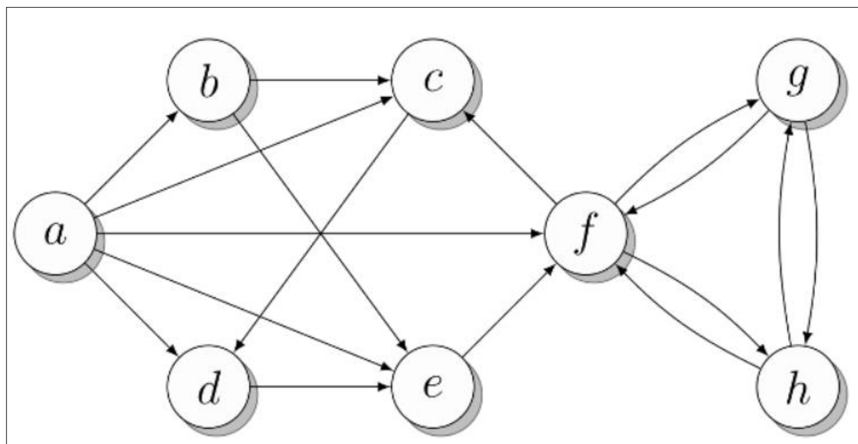


Рис.109. Направленный граф

Пример представления этого же графа в виде матрицы смежности:

```
a, b, c, d, e, f, g, h = range(8)
# a b c d e f g h
N =  [[0,1,1,1,1,1,0,0], # a
```

```
    [0,0,1,0,1,0,0,0], # b
    [0,0,0,1,0,0,0,0], # c
    [0,0,0,0,1,0,0,0], # d
    [0,0,0,0,0,1,0,0], # e
    [0,0,1,0,0,0,1,1], # f
    [0,0,0,0,0,1,0,1], # g
     [0,0,0,0,0,1,1,0]] #
```

Реализация графа в виде списка смежности. В качестве значения «стоимость» использован вес ребра, значение которого выбрано случайным образом:

```
a, b, c, d, e, f, g, h = range(8)
N = [
   {b:2, c:1, d:3, e:9, f:4}, # a
   {c:4, e:3}, # b
   {d:8}, # c
   {e:7}, # d
   {f:5}, # e
   {c:2, g:2, h:2}, # f
   {f:1, h:6}, # g
   {f:9, g:8} # h
]
```

Реализация графа рис. в виде списка двоек:

```
a, b, c, d, e, f, g, h = range(8)
N = [
   [b, c, d, e, f], # a
   [c, e], # b
   [d], # c
   [e], # d
   [f], # e
   [c, g, h], # f
   [f, h], # g
   [f, g] # h
]
```

12.8.3. Способы обхода графа

Аналогично рассмотренным ранее динамическим структурам данных обход и поиск в графе осуществляется в глубину и в ширину.

Рассмотрим, как эти методы работают на графах.

Пусть дан некоторый граф (рис.110) и мы хотим обойти по ребрам все его вершины.

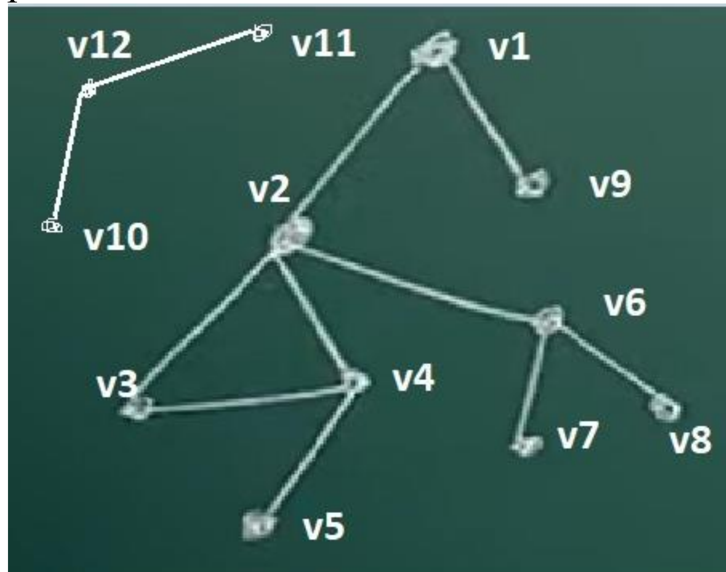


Рис.110. Пример графа, не все вершины которого соединены

Какие задачи в реальной жизни будут соответствовать обходу графа? Например, поиск потерявщегося человека в лабиринте, вам нужно будет пройти по разным проходам лабиринта и заглянуть во все пещеры. Или у вас есть файловая система, и вы хотите найти какой-то файл. Для этого вы сначала идете во все директории текущей папки, затем последовательно во все директории других папок и т.д.

Пешеры и папки – это вершины, проходы – ребра графа. Формально вы хотите попасть в каждую вершину графа, при этом необязательно, что вы пройдете по каждому ребру. Как это можно сделать?

Есть начальная вершина, в которой мы стоим (v1). Пойдем из нее в первую попавшуюся соседнюю вершину, допустим v2. Из нее опять пойдем в первую попавшуюся соседнюю вершину, в которой мы еще не были, например в v3. Далее в v4. Из этой вершины идти наверх бессмысленно, так как там мы уже были, следовательно, пойдем вниз в вершину v5. Таким образом, мы попали в вершину, все соседи которой помечены как посещенные.

Из вершины v5 мы возвращаемся назад в вершину v4 и смотрим, есть ли у этой вершины какие-нибудь еще непомеченные соседи, в которых мы еще не были. У вершины v4 таких соседей тоже нет, поэтому мы переходим в вершину, соседнюю с v4 – v3. У этой вершины тоже нет непомеченных соседей, поэтому мы поднимаемся выше в вершину v2. У этой вершины есть непомеченный сосед – вершина v6, идем в эту вершину. Из этой вершины идем, например, в вершину v7.

Здесь новых соседей нет, поэтому поднимаемся наверх в вершину v6. У этой вершины есть непомеченный сосед v8.

Так как у вершины v8 нет непомеченных соседей, рассуждая аналогичным образом, мы последовательно поднимаемся наверх до вершины v1, у которой есть непомеченный сосед v9. Так как у этой вершины нет непомеченных соседей, мы поднимаемся наверх и оказываемся в начальной вершине v1. В этот момент мы останавливаемся, так как обошли все вершины, до которых мы смогли дойти из выбранной нами начальной вершины.

Важно! Целью этого алгоритма является обязательный просмотр всех вершин, но не всех ребер. В нашем примере мы просмотрели все вершины, но при этом ребро, соединяющее вершины v2 и v4 осталось непройденным.

Рассмотрим стандартную реализацию обхода графа в глубину на языке Python. Как было объяснено выше, алгоритм обхода в глубину помещает каждую вершину графа в одну из двух категорий:

1. пройденные (visited).
2. не пройденные (not visited).

Цель алгоритма состоит в том, чтобы пометить каждую вершину как “пройденная”, избегая при этом циклов.

Алгоритм работает следующим образом:

1. поместим любую вершину графа на вершину стека.
2. возьмите верхний элемент стека и добавьте его в список “пройденных”.
3. создайте список смежных вершин для этой вершины. Добавьте те вершины, которых нет в списке “пройденных”, в вверх стека.
4. повторять шаги 2 и 3, пока стек не станет пустым.

Проиллюстрируем графически, как работает алгоритм поиска в глубину (рис.111абв). Будем использовать неориентированный граф с пятью вершинами.

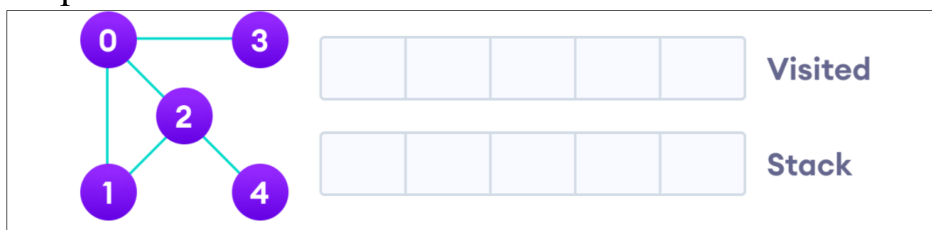


Рис.111а. Начальные значения для алгоритма обхода в глубину

Начнем мы с вершины “0”. В первую очередь алгоритм поиска в глубину поместит эту вершину в список “Пройденные” (на рисунке “Visited”), а ее смежные вершины — в стек (на рисунке Stack).

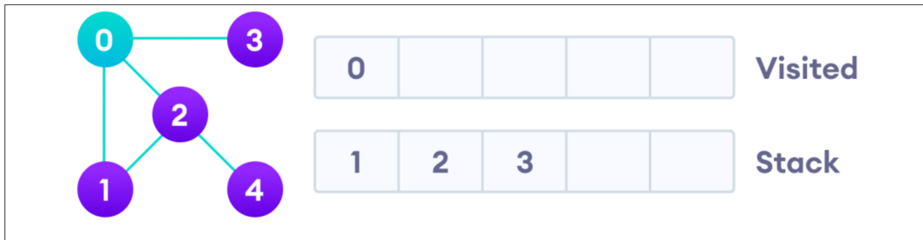


Рис.111б. Шаг 1 алгоритма обхода в глубину

Затем мы берем следующий элемент стека - вершину “1”, и переходим к ее соседним вершинам. Поскольку вершина “0” уже просмотрена, мы переходим к следующей вершине “2” (Рис.).

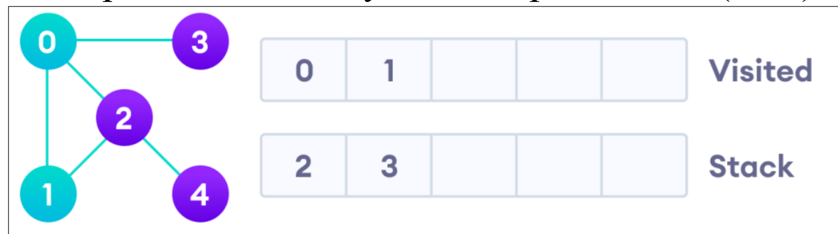


Рис.111в. Шаг 2 алгоритма обхода в глубину

Вершина “2” смежна непройденной вершине “4”, следовательно, мы добавляем ее в стек, а затем проходим через нее (рис. 112а,б).

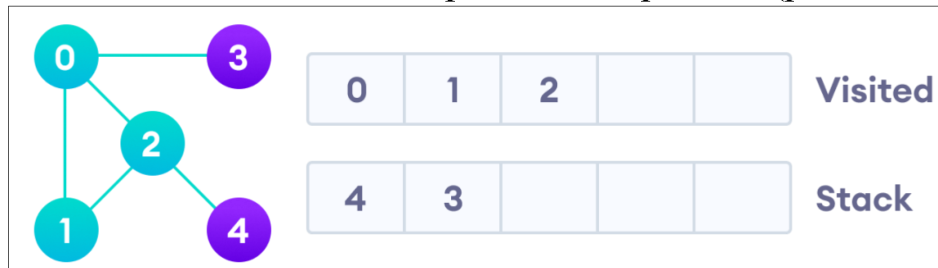


Рис.112а. Шаг 2 алгоритма обхода в глубину

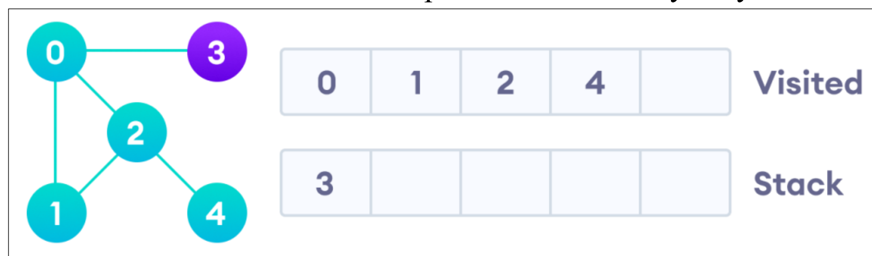


Рис.112б. Шаг 2 алгоритма обхода в глубину

Переходим к вершине 3. После проверки всех ее смежных вершин, стек остался пустым, следовательно, алгоритм обхода графа в глубину завершил свою работу (рис.115в).

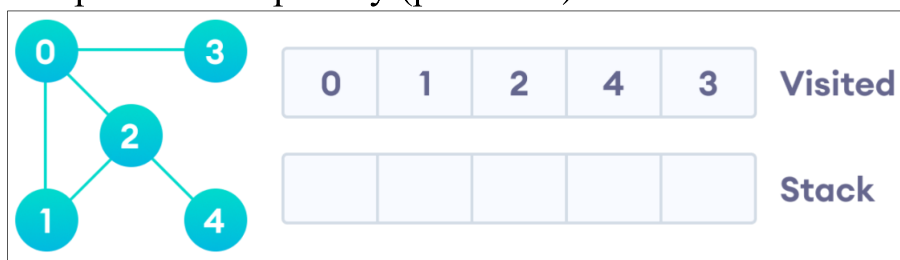


Рис.112г. Завершающий шаг алгоритма обхода в глубину

Программный код на Python, реализующий иллюстрацию алгоритма представлен ниже:

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)

    print(start)

    for next in graph[start] - visited:
        dfs(graph, next, visited)
    return visited


graph = {'0': set(['1', '2']),
         '1': set(['0', '3', '4']),
         '2': set(['0']),
         '3': set(['1']),
         '4': set(['2', '3'])}

dfs(graph, '0')
```

Какова сложность этого алгоритма по времени выполнения? Как всегда, это зависит от того, как мы храним граф.

Допустим мы храним граф матрицей смежности. Тогда для каждой вершины нужно посмотреть на всех ее соседей. Для этого надо просмотреть всю строку матрицы. Всего вершин V, просмотреть их можно за V операций. Получаем сложность алгоритма по времени равна V^2.

Если у нас есть список смежных вершин, то мы для каждой вершины смотрим только на ее соседей, причем один раз. Получается, что количество соседей равно количеству ребер вершины (E). Сложность алгоритма по времени равна E.

В случае множеств смежности, для того, чтобы найти смежные вершины нам необходимо просмотреть для каждой вершины все ребра. В этом случае сложность алгоритма $V \cdot E$.

Таким образом, алгоритм обхода в глубину справляется лучше, если граф представлен в виде списка смежности.

Рассмотрим алгоритм обхода графа в ширину (рис.113). Является вариантом так называемого волнового алгоритма, который позво-

ляет не искать каждый раз вершины, помеченные на предыдущем шаге, а запоминать их. Это позволяет при дальнейшей обработке каждый раз не тратить время на то, чтобы обойти весь массив вершин и найти там отмеченные ранее вершины.

Как действует волновой алгоритм? Выбираем начальную вершину, от которой мы хотим посчитать расстояние до всех остальных вершин графа. Помечаем эту вершину нулем. Далее берем соседние с ней вершины и помечаем их единицами. Затем непомеченные соседние с ними вершины помечаем двойками. Непомеченные соседи этих вершин помечаем тройками и т.д. Процесс продолжается до тех пор, пока все вершины не кончатся либо мы не дойдем до числа (V-1)[5].

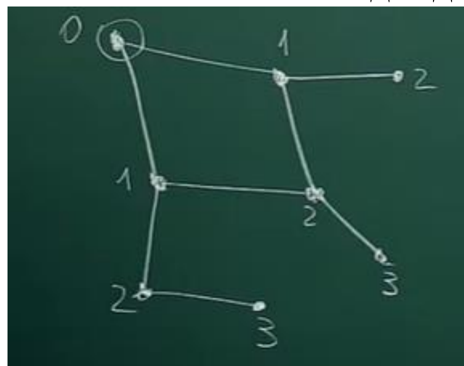


Рис.113. Демонстрация алгоритма обхода графа в ширину

Информацию о том, какие вершины помечены на предыдущем шаге, сохраняем в списки. Количество списков равно количеству расставленных меток. В нашем случае будет три списка (метки 1, 2, 3). Количество элементов в каждом списке равно количеству вершин с определенной меткой. Так в первом списке будет 2 элемента, во втором – 3 элемента, в третьем – 2 элемента. Когда мы записываем метку 1 в первый список, нам сразу же будет нужен второй список вершин с метками 2 , являющимися соседями вершины с меткой 1. При этом массив с метками 1 нам еще нужен, так как мы его еще не обработали. Аналогичным образом заполняются другие списки. Графически процесс заполнения списков представлен на рис.114.

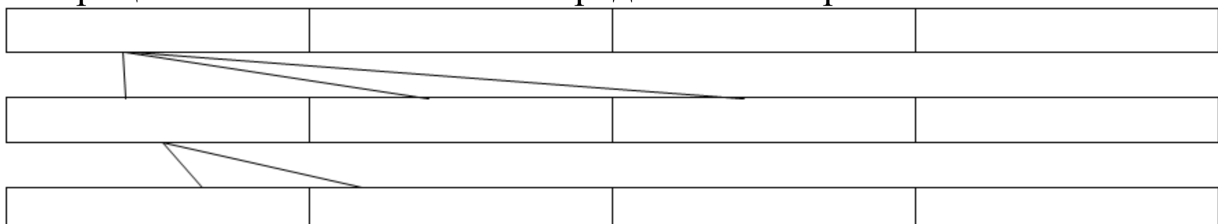


Рис.114. Процесс заполнения списков метками вершин

[5] самое большое расстояние, если вершины соединить в цепочку

Можно использование два массива, производя запись то в один, то в другой список, как в рассмотренном выше алгоритме обхода в глубину.

Можно использовать один списк. В этом случае метки вершин будут записаны по возрастанию, т.е. сначала все единицы, затем все двойки и т.д. По такому списку мы будем двигаться слева направо, последовательно беря вершину и ее соседей. Если вершины не помечены, помечать, добавляя их номера в конец списка. Таким образом, у нас получится очередь.

В этом случае алгоритм обхода в ширину можно сформулировать следующим образом: берем из очереди очередную вершину и ее необработанных соседей, обрабатываем и помещаем в конец очереди.

Вопрос: как мы можем обработать соседа, если мы не знаем, где у нас проходит границами между соседями? А это нам и не нужно. При обработке вершины, мы можем посмотреть, что в этой вершине записано, и в ее соседей записывать число на единицу больше записанного в текущей вершине.

Таким образом, у нас будет один цикл, который будет брать вершину из очереди, обрабатывать ее, добавляя в очередь всех ее соседей до тех пор, пока очередь не опустеет. Это означает, что мы обработали все вершины, до которых смогли дойти.

Посчитаем сложность различных реализаций этого алгоритма.

Граф представлен в виде матрицы или множеств смежности. В этом случае волновой алгоритм проходил бы каждую из V вершин ровно V раз. . Получаем сложность алгоритма по времени равна V^v, т.е. алгоритм работал бы не быстрее, чем V^2.

Граф представлен в виде списка смежных вершин. В этом случае вершина попадает в очередь ровно один раз. Для каждой вершины мы смотрим на всех ее соседей, сложность алгоритма по времени равна $2 \cdot E$, т.е. пропопцианальна E $(\sim E)$.

Таким образом, алгоритм обхода в ширину справляется лучше, если граф представлен в виде списка смежности. Реализуем на Python граф в виде списков смежности.

Зададим функцию обхода графа, начиная с какой-то вершины v. в структуре visited будем хранить вершины, которые мы уже посещали, visited является множеством. В списке new_vertices будем хранить вершины, которые еще надо исследовать, используем его как очередь. Обход продолжаем до тех пор, пока to_explore не станет пустым списком. Очередную вершину, которую будем обрабатывать (u)

встроенными методами работы со списками: забираем элемент из начала списка `to_explore` методом pop (), добавляем элемент в конец списка `to_explore` – методом extend(), обновляем список просмотренных вершин (структуру visited) просмотренными вершинами списка `new_vertices`:

```
def bfs(graf, v):
    visited={v}
        to_explore =[v]
    while to_explore:
        u=to_explore.pop (0)
        print (u)
        new_vertices=[i for i in graph [u] if i
not in visited]
        to_explore.extend(new_vertices)
        visited.update(new_vertices)

#задаем граф в виде списков смежности
a, b, c, d, e, f, g, h = range(8)
graph = [
   [b, c, d, e, f], # a
   [c, e], # b
   [d], # c
   [e], # d
   [f], # e
   [c, g, h], # f
   [f, h], # g
   [f, g] # h
]
print (graph)
bfs(graph,0)
```

graph [u] - это список из смежных вершин нашей вершины u, которую мы рассматриваем, i- это какой- то элемент такого списка. Соответственно, мы берем только те вершины, которые мы еще не посещали, а не посещали мы вершины, которых еще нет во множестве visited. Такие вершины мы помещаем в список new_vertices.

функция print(graph) выведет граф в виде списка смежности. Будет напечатано:

```
[[1, 2, 3, 4, 5], [2, 4], [3], [4], [5], [2, 6, 7], [5, 7], [5, 6]]
```

После вызова функции bfs (graph, 0)мы увидим вершины в следующем порядке (рис.115):

```
[[1, 2, 3, 4, 5], [2, 4], [3], [4], [5], [2, 6, 7], [5, 7], [5, 6]]
0
1
2
3
4
5
6
7
```

Рис.115. Порядок следования вершин в случае списка смежности

12.8.4. Алгоритмы на графах

Рассмотрим алгоритм, предложенный нидерландским ученым Эдегером Дейкстрой в 1959 году.

Суть этого алгоритма состоит в определении наикратчающих маршрутов от некоторой стартовой вершины графа до всех остальных его вершин (рис.116).

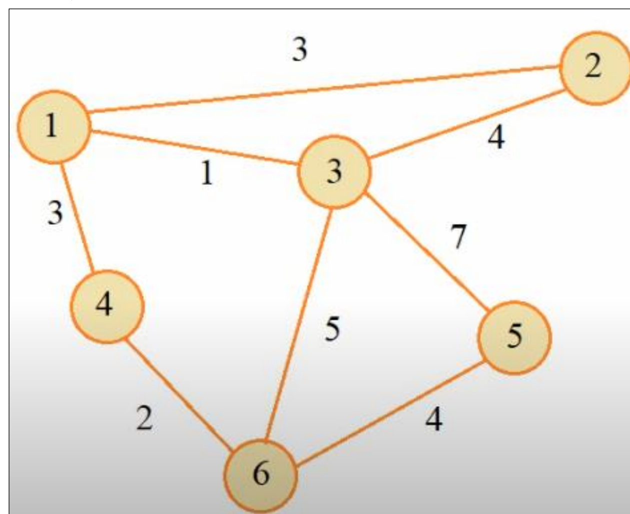


Рис.116. Граф для задачи маршрутизации

Исходными данными для этого алгоритма является граф с дугами, имеющими положительные веса, отрицательные не допустимы.

Где в реальной жизни применяются такие графы? Зачем такой алгоритм? В действительности с помощью этого графа можно формализовать множество задач. Например, задачу маршрутизации информационных пакетов, где стартовый узел – компьютер клиента, кото-

рый отправляет запрос определенноу серверу (рис.117), веса дуг – среднее время передачи информационного пакета.

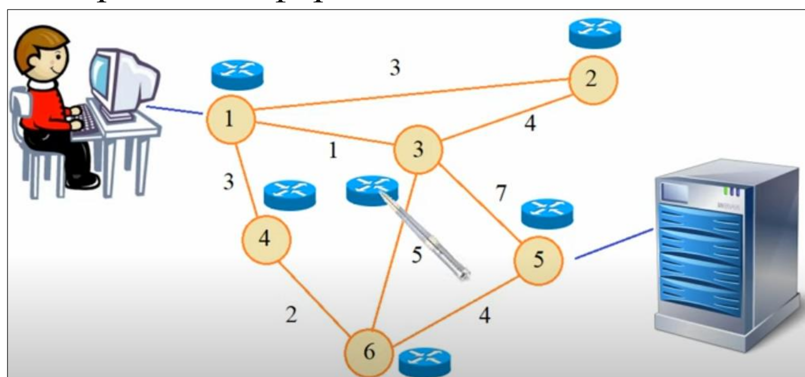


Рис.117. Граф для задачи маршрутизации

Пусть сервер находится в узле 5 (рис.117). С помощью алгоритма Дейкстры можно найти маршрут с наименьшими временными затратами для передачи зароса от клиента к серверу. Похожих задач в практике очень много, поэтому алгоритм Дейксты и получил такую популярность.

Давайте посмотрим, как он работает.

Основная его идея очень проста – полный направленный поик. В начале мы должны определится со стартовой вершиной. Пусть это будет вершина с меткой 1 (рис.). Дальнейший процесс будем представлять в виде таблицы (см. таб абвг). В строке будем отображать номер итерации, в столбцах – номера вершин графа. На первой итерации работы этого алгоритма для стартовой вершины мы запишем нулевой вес, так как мы уже находимся в ней, т.е. для того, чтобы попасть в нее никуда двигаться не нужно. Для остальных вершин в качестве значений пишем бесконечность, так как не знаем, какие веса нам понадобятся для того, чтобы перейти из вершины 1 во все остальные вершины (табл.16а).

Таблица.16а. Работа алгоритма Дейкстры. Шаг 1.

Итерация	1	2	3	4	5	6
1	0	∞	∞	∞	∞	∞

На втором шаге алгоритма мы просматриваем все вершины, в которые можно попасть из вершины 1. Это вершины с метками 2, 3 и 4. Для каждой из вершин мы формируем вес пути как вес предыдущей вершины и вес дуги, соединяющий вершины 1 и 2. Аналогичным образом поступаем с вершинами 3 и 4. Получаем следующие веса (таб.16б).

Итерация	1	2	3	4	5	6
1	0	∞	∞	∞	∞	∞
2	0	0+3=3	0+1=1	0+3=3	∞	∞

К вершинам 5 и 6 напрямую из вершины 1 мы пройти не можем, поэтому дублируем значение бесконечность (∞). В получившейся второй строчке мы должны выбрать вершину с минимальным весом. В нашем случае – это вершина с меткой 3 (рис.118).

№	1	2	3	4	5	6
1	0	∞	∞	∞	∞	∞
2		3	1	3	∞	∞

Рис.118 Веса дуг на 2-ой итерации алгоритма

Далее необходимо взять эту вершину и повторить туже операцию. Из вершины 3 можно попасть в вершины 2, 5, 6. Вершину 1 мы не учитываем, так как она уже была рассмотрена, и возврат назад алгоритмом не предусмотрен.

Для вершин 2, 5, 6 мы считаем веса ребер (дуг) по той же самой схеме. На третьей итерации работы алгоритма для вершины 2 вес дуги будет равен 1 + 4=5, 5 >3, поэтому для вершины 2 мы оставляем меньший вес 3. Для вершины 5 мы получаем вес 1+7=8, 8< ∞, оставляем этой вершине вес 8. Аналогично для вершины 6: 1+5=6, 6 < ∞. Вес вершины 6. Значение вершины 4 мы дублируем, так как она находятся в рассмотрении. В итоге получаем таблицу, как на рис.119.

№	1	2	3	4	5	6
1	0	∞	∞	∞	∞	∞
2		3	1	3	∞	∞
3		3		3	8	6

Рис.119. Веса дуг на 3-ой итерации алгоритма

Далее, из всех полученных значений нужно выбрать минимальное. В нашем случае две вершины (2 и 4) имеют минимальное значение, равное 3. Можно выбрать любую из них.

Возьмем вершину 2. Из этой вершины мы можем перейти только в уже рассмотренные вершины 1 и 3, следовательно, вес 3 – это минимальный вес для этой вершины, т.е. из вершины 1 в вершину 2 можно пройти с минимальным весом, равным 3. Исключаем вершину

2 из рассмотрения. Следовательно, на четвертой итерации работы алгоритма будет взята вершина с меткой 4.

Повторяем операции. Из 4-ой вершины мы можем пройти только в вершину 6. Вес соответствующей дуги будет равен 3+2 = 5. Таким образом, на 4-ой итерации для вершины 6 появляется вес 5, значение 5-ой вершины, так как она еще не рассматривалась, дублируется (рис.120).

№	1	2	3	4	5	6
1	0	∞	∞	∞	∞	∞
2		3	1	3	∞	∞
3		3		3	8	6
4				3	8	6
5					8	5

Рис.120 Веса дуг на 4-ой итерации алгоритма

Далее (5 итерация) снова рассматриваем оставшиеся вершины 5 и 6. Выбираем вершину с минимальным весом – это вершина 6. Так как вершина 5 еще не рассматривалась, сформируем вес дуги из вершины 5 в вершину 6: 5 + 4=9, 9 > 8, оставляем 8.

Таким образом, на 6-ой итерации алгоритма получаем следующие веса дуг (рис.121).

№	1	2	3	4	5	6
1	0	∞	∞	∞	∞	∞
2		3	1	3	∞	∞
3		3		3	8	6
4				3	8	6
5					8	5
6					8	

Рис.121. Веса дуг на 6-ой итерации алгоритма

Мы прошли шесть итераций алгоритма Дейкстры, чтобы определить маршрут с наименьшими весами из стартовой вершины 1 ко всем вершинам графа. Из таблицы видно, минимальная длина пути из вершины 1 в вершину 2 равна 3, из вершины 1 в вершину 3 – 1, из вершины 1 в вершину 4 – 3 и т.д.

Посмотрим, как реализовать этот алгоритм на языке Pyton. Сначала опишем граф с весами дуг с помощью матрицы смежности (рис.122).

	1	2	3	4	5	6
1	0	3	1	3	0	0
2	3	0	4	0	0	0
3	1	4	0	0	7	5
4	3	0	0	0	0	2
5	0	0	7	0	0	4
6	0	0	5	2	4	0

Рис.122. Матрица смежности графа примера

Реализация представления графа в виде матрицы смежности:

```
D = ((0, 3, 1, 3, math.inf, math.inf),
     (3, 0, 4, math.inf, math.inf, math.inf),
     (1, 4, 0, math.inf, 7, 5),
     (3, math.inf, math.inf, 0, math.inf, 2),
     (math.inf, math.inf, 7, math.inf, 0, 4),
     (math.inf, math.inf, 5, 2, 4, 0))
```

Далее определяем число вершин в графе. Функция len (D) вернет число строк в графе, это и будет количество вершин. Затем в виде списка последнюю строку таблицы, во все ячейки которой запишем значение бесконечность.

```
N = len(D)
T = [math.inf]*N
```

Затем с помощью переменной v определяем текущую вершину, которая будет рассматриваться на i-ой итерации. Стартовая вершина определена как 0, т.к. в Python нумерация начинается с 0. Через S обозначим множество просмотренных вершин, зададим вес начальной вершины `T[v]` – 0.

```
v = 0
S = {v}
T[v] = 0
```

Далее в цикле с помощью встроенной функции enumerate () мы одновременно перебираем все порядковые номера и все значения вершин, связанных с текущей вершиной v. Сначала v=1 и функция enumerate () вернет индексы и значения вершин первой строки матрицы: 1, 2, 3, 4, 5, 6 и 0, 3, 1, 3, 0, 0 соответственно.

Задаем значение индекса вершины, для которой уже подсчитан минимальный вес, равным -1, и пока такое значение не найдено (while v != -1), в цикле for индексы рассматриваемых вершин считываем в переменную j, вес вершин – в переменную dw.

Если индекс рассматриваемой вершины не содержится в множестве просмотренных вершин S (условие if j not in S:), ее вес добавляется к весу рассматриваемой вершины (w = T[v] + dw). Если полученный вес меньше веса, записанного для этой вершины (условие if w < T[j]:), мы сохраняем новое значение связи для j-ой вершины (T[j] = w) и связываем вершину v с j-ой вершиной с помощью списка M (M [j] = v):

```
while v != -1:
    for j, dw in enumerate(D[v]):
        if j not in S:
            w = T[v] + dw
            if w < T[j]:
                T[j] = w
                M[j] = v
```

После того, как операция подсчета весов всех j-ых вершин для текущей вершины v завершена (while v != -1:), с помощью функции arg_min () выбираем следующий узел с минимальным весом, чтобы на следующей итерации отталкиваться от него:

```
def arg_min(T, S):
    amin = -1
    m = math.inf
    for i, t in enumerate(T):
        if t < m and i not in S:
            m = t
            amin = i

    return amin
```

```
v = arg_min(T, S)
    if v >= 0:
        S.add(v)
```

Полный код программы примера:

```
import math

def arg_min(T, S):
    amin = -1
    m = math.inf  # максимальное значение
    for i, t in enumerate(T):
        if t < m and i not in S:
            m = t
            amin = i

    return amin

#описываем граф
D = ((0, 3, 1, 3, math.inf, math.inf),
     (3, 0, 4, math.inf, math.inf, math.inf),
     (1, 4, 0, math.inf, 7, 5),
     (3, math.inf, math.inf, 0, math.inf, 2),
     (math.inf, math.inf, 7, math.inf, 0, 4),
     (math.inf, math.inf, 5, 2, 4, 0))

N = len(D)  # число вершин в графе
T = [math.inf]*N   # последняя строка таблицы

v = 0       # стартовая вершина (нумерация с нуля)
S = {v}     # просмотренные вершины
T[v] = 0    # нулевой вес для стартовой вершины
M = [0]*N   # оптимальные связи между вершинами

while v != -1:          # цикл, пока не просмотрим 
все вершины
    for j, dw in enumerate(D[v]):   # перебираем 
все связанные вершины с вершиной v
        if j not in S:           # если вершина 
еще не просмотрена
            w = T[v] + dw
            if w < T[j]:
```

```
                T[j] = w
                M[j] = v # связываем вершину j с
вершиной v

    v = arg_min(T, S)     # выбираем следующий узел
с наименьшим весом
    if v >= 0:          # выбрана очередная вершина
        S.add(v)        # добавляем новую вершину в
рассмотрение

#print(T, M, sep="\n")

# формирование оптимального маршрута:
start = 0
end = 4
P = [end]
while end != start:
    end = M[P[-1]]
    P.append(end)

print(P)
```

Практическая работа № 17
ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ В PYTHON. ГРАФЫ

Задание. Сеть состоит из 10 вершин (рис.123). Пропускные способности дуг заданы таблицей. Найти максимальный поток между вершинами 1 и 10.

Сформируйте граф любым из рассмотренных в п. 13.8.3 способом. Выполните вывод графа на экран.

Каждое задание необходимо решить в соответствии с изученными методами формирования, вывода и обработки динамических структур данных в языке *Python*. Обработку следует выполнить на основе базовых алгоритмов обхода, поиска, вставки и удаления элемента, удаление всей динамической структуры. При объявлении динамической структуры выполните комментирование используемых полей. Программу для решения каждого задания необходимо разработать методом структурной декомпозиции, оформив комментарии к коду.

Следует реализовать каждое задание в соответствии с приведенными этапами:

- изучить словесную постановку задачи, выделив при этом все виды данных;
- сформулировать математическую постановку задачи;
- выбрать метод решения задачи, если это необходимо;
- разработать графическую схему алгоритма;
- записать разработанный алгоритм на языке *Python 3.8*;
- разработать контрольные тесты к программе;
- отладить программу;
- представить отчет по работе.

Требования к отчету.

Отчет по практической работе должен соответствовать следующей структуре.

- титульный лист;
- словесная постановка задачи. В этом подразделе проводится полное описание задачи. Описывается суть задачи, анализ входящих в нее физических величин, область их допустимых значений, единицы их измерения, возможные ограничения, анализ условий при которых задача имеет решение (не имеет решения), анализ ожидаемых результатов.
- математическая модель. В этом подразделе вводятся математические описания физических величин и математическое описание их взаимодействий. Цель подраздела – представить решаемую задачу в математической формулировке.
- алгоритм решения задачи. В подразделе описывается разработка структуры алгоритма, обосновывается абстракция данных, задача разбивается на подзадачи. Схема алгоритма выполняется по ЕСПД (ГОСТ 19.003-80 и ГОСТ 19.002-80).
- листинг программы. Подраздел должен содержать текст программы на языке программирования *Python 3.8*, реализованный в среде ms *PyCharm*.
- контрольный тест. Подраздел содержит наборы исходных данных и полученные в ходе выполнения программы результаты.
- выводы по практической работе.

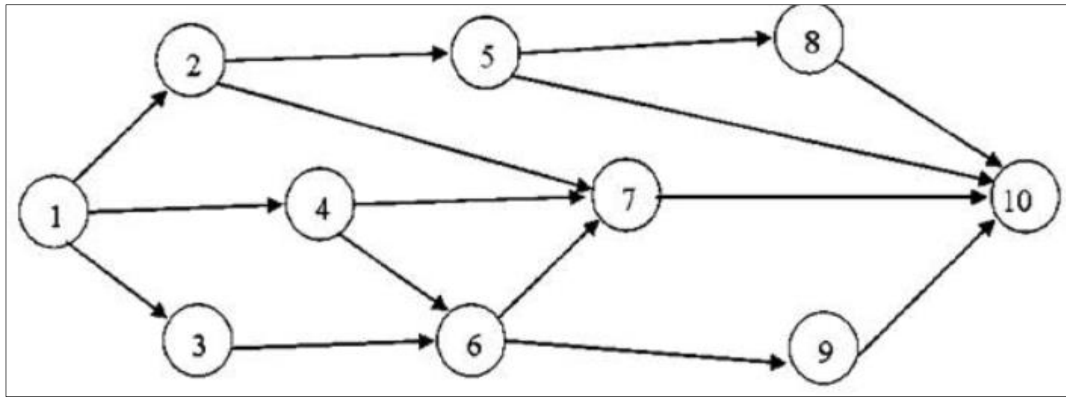


Рис.123. Граф для индивидуальных заданий

ВАРИАНТЫ ЗАДАНИЙ

Вар.1	1	2	3	4	5	6	7	8	9	10
1	0	8	2	9	0	0	0	0	0	0
2	0	0	0	0	3	0	2	0	0	0
3	0	0	0	0	0	7	0	0	0	0
4	0	0	0	0	0	3	2	0	0	0
5	0	0	0	0	0	0	0	1	0	1
6	0	0	0	0	0	0	4	0	9	0
7	0	0	0	0	0	0	0	0	0	10
8	0	0	0	0	0	0	0	0	0	2
9	0	0	0	0	0	0	0	0	0	4
10	0	0	0	0	0	0	0	0	0	0
Вар.2										
	1	2	3	4	5	6	7	8	9	10
1	0	5	5	4	0	0	0	0	0	0
2	0	0	0	0	6	0	3	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	4	0	0	0
5	0	0	0	0	0	0	0	9	0	7
6	0	0	0	0	0	0	1	0	5	0
7	0	0	0	0	0	0	0	0	0	6
8	0	0	0	0	0	0	0	0	0	10
9	0	0	0	0	0	0	0	0	0	10
10	0	0	0	0	0	0	0	0	0	0
Вар.3	1	2	3	4	5	6	7	8	9	10
1	0	9	6	7	0	0	0	0	0	0
2	0	0	0	0	2	0	8	0	0	0
3	0	0	0	0	0	7	0	0	0	0
4	0	0	0	0	0	1	5	0	0	0
5	0	0	0	0	0	0	0	10	0	1
6	0	0	0	0	0	0	10	0	5	0
7	0	0	0	0	0	0	0	0	0	2
8	0	0	0	0	0	0	0	0	0	10
9	0	0	0	0	0	0	0	0	0	5
10	0	0	0	0	0	0	0	0	0	0

Вар.4	1	2	3	4	5	6	7	8	9	10
1	0	10	10	10	0	0	0	0	0	0
2	0	0	0	0	10	0	10	0	0	0
3	0	0	0	0	0	1	0	0	0	0
4	0	0	0	0	0	6	7	0	0	0
5	0	0	0	0	0	0	0	4	0	4
6	0	0	0	0	0	0	4	0	3	0
7	0	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	0	7
9	0	0	0	0	0	0	0	0	0	4
10	0	0	0	0	0	0	0	0	0	0
Вар.5	1	2	3	4	5	6	7	8	9	10
1	0	6	5	6	0	0	0	0	0	0
2	0	0	0	0	10	0	3	0	0	0
3	0	0	0	0	0	5	0	0	0	0
4	0	0	0	0	0	7	3	0	0	0
5	0	0	0	0	0	0	0	8	0	2
6	0	0	0	0	0	0	3	0	6	0
7	0	0	0	0	0	0	0	0	0	10
8	0	0	0	0	0	0	0	0	0	7
9	0	0	0	0	0	0	0	0	0	8
10	0	0	0	0	0	0	0	0	0	0
Вар.6	1	2	3	4	5	6	7	8	9	10
1	0	10	7	4	0	0	0	0	0	0
2	0	0	0	0	7	0	6	0	0	0
3	0	0	0	0	0	7	0	0	0	0
4	0	0	0	0	0	4	2	0	0	0
5	0	0	0	0	0	0	0	3	0	2
6	0	0	0	0	0	0	7	0	6	0
7	0	0	0	0	0	0	0	0	0	7
8	0	0	0	0	0	0	0	0	0	5
9	0	0	0	0	0	0	0	0	0	10
10	0	0	0	0	0	0	0	0	0	0
Вар.7	1	2	3	4	5	6	7	8	9	10
1	0	5	2	6	0	0	0	0	0	0
2	0	0	0	0	6	0	9	0	0	0
3	0	0	0	0	0	6	0	0	0	0
4	0	0	0	0	0	9	8	0	0	0
5	0	0	0	0	0	0	0	6	0	7
6	0	0	0	0	0	0	9	0	10	0
7	0	0	0	0	0	0	0	0	0	9
8	0	0	0	0	0	0	0	0	0	4
9	0	0	0	0	0	0	0	0	0	6

10	0	0	0	0	0	0	0	0	0	0
Вар.8	1	2	3	4	5	6	7	8	9	10
1	0	2	9	2	0	0	0	0	0	0
2	0	0	0	0	2	0	1	0	0	0
3	0	0	0	0	0	2	0	0	0	0
4	0	0	0	0	0	0	8	6	0	0
5	0	0	0	0	0	0	0	1	0	8
6	0	0	0	0	0	0	1	0	6	0
7	0	0	0	0	0	0	0	0	0	7
8	0	0	0	0	0	0	0	0	0	8
9	0	0	0	0	0	0	0	0	0	2
10	0	0	0	0	0	0	0	0	0	0
Вар.9	1	2	3	4	5	6	7	8	9	10
1	0	5	10	2	0	0	0	0	0	0
2	0	0	0	0	2	0	9	0	0	0
3	0	0	0	0	0	8	0	0	0	0
4	0	0	0	0	0	5	7	0	0	0
5	0	0	0	0	0	0	0	1	0	1
6	0	0	0	0	0	0	2	0	3	0
7	0	0	0	0	0	0	0	0	0	4
8	0	0	0	0	0	0	0	0	0	9
9	0	0	0	0	0	0	0	0	0	6
10	0	0	0	0	0	0	0	0	0	0
Вар.10	1	2	3	4	5	6	7	8	9	10
1	0	3	10	9	0	0	0	0	0	0
2	0	0	0	0	2	0	2	0	0	0
3	0	0	0	0	0	2	0	0	0	0
4	0	0	0	0	0	4	9	0	0	0
5	0	0	0	0	0	0	0	6	0	7
6	0	0	0	0	0	0	10	0	3	0
7	0	0	0	0	0	0	0	0	0	5
8	0	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0	7
10	0	0	0	0	0	0	0	0	0	0
Вар.11	1	2	3	4	5	6	7	8	9	10
1	0	7	9	6	0	0	0	0	0	0
2	0	0	0	0	5	0	10	0	0	0
3	0	0	0	0	0	6	0	0	0	0
4	0	0	0	0	0	2	10	0	0	0
5	0	0	0	0	0	0	0	1	0	5
6	0	0	0	0	0	0	6	0	9	0
7	0	0	0	0	0	0	0	0	0	9
8	0	0	0	0	0	0	0	0	0	9

9	0	0	0	0	0	0	0	0	0	6
10	0	0	0	0	0	0	0	0	0	0
Вар.12	1	2	3	4	5	6	7	8	9	10
1	0	8	6	4	0	0	0	0	0	0
2	0	0	0	0	9	0	2	0	0	0
3	0	0	0	0	0	8	0	0	0	0
4	0	0	0	0	0	2	2	0	0	0
5	0	0	0	0	0	0	0	5	0	5
6	0	0	0	0	0	0	1	0	5	0
7	0	0	0	0	0	0	0	0	0	6
8	0	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0
Вар.13	1	2	3	4	5	6	7	8	9	10
1	0	1	6	4	0	0	0	0	0	0
2	0	0	0	0	9	0	1	0	0	0
3	0	0	0	0	0	1	0	0	0	0
4	0	0	0	0	0	2	2	0	0	0
5	0	0	0	0	0	0	0	2	0	5
6	0	0	0	0	0	0	1	0	5	0
7	0	0	0	0	0	0	0	0	0	6
8	0	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0
Вар.14	1	2	3	4	5	6	7	8	9	10
1	0	8	6	4	0	0	0	0	0	0
2	0	0	0	0	2	6	2	0	0	0
3	0	0	0	0	0	4	0	0	0	0
4	0	0	0	0	0	5	2	0	0	0
5	0	0	0	0	0	0	0	3	0	3
6	0	0	0	0	0	0	0	0	6	0
7	0	0	0	0	0	0	0	0	0	8
8	0	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0
Вар.15	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	0	0	0	0	0	0
2	0	0	0	0	2	0	2	0	0	0
3	0	0	0	0	0	3	0	0	0	0
4	0	0	0	0	0	4	4	0	0	0
5	0	0	0	0	0	0	0	5	0	5
6	0	0	0	0	0	0	6	0	6	0
7	0	0	0	0	0	0	0	0	0	7

8	0	0	0	0	0	0	0	0	0	8
9	0	0	0	0	0	0	0	0	0	9
10	0	0	0	0	0	0	0	0	0	0
Вар.16	1	2	3	4	5	6	7	8	9	10
1	0	10	10	10	0	0	0	0	0	0
2	0	0	0	0	9	0	9	0	0	0
3	0	0	0	0	0	8	0	0	0	0
4	0	0	0	0	0	7	7	0	0	0
5	0	0	0	0	0	0	0	6	0	6
6	0	0	0	0	0	0	5	0	5	0
7	0	0	0	0	0	0	0	0	0	4
8	0	0	0	0	0	0	0	0	0	3
9	0	0	0	0	0	0	0	0	0	2
10	0	0	0	0	0	0	0	0	0	0
Вар.17	1	2	3	4	5	6	7	8	9	10
1	0	10	9	8	0	0	0	0	0	0
2	0	0	0	0	7	0	6	0	0	0
3	0	0	0	0	0	5	0	0	0	0
4	0	0	0	0	0	4	3	0	0	0
5	0	0	0	0	0	0	0	2	0	1
6	0	0	0	0	0	0	10	0	9	0
7	0	0	0	0	0	0	0	0	0	8
8	0	0	0	0	0	0	0	0	0	7
9	0	0	0	0	0	0	0	0	0	6
10	0	0	0	0	0	0	0	0	0	0
Вар.18	1	2	3	4	5	6	7	8	9	10
1	0	1	2	3	0	0	0	0	0	0
2	0	0	0	0	4	0	5	0	0	0
3	0	0	0	0	0	6	0	0	0	0
4	0	0	0	0	0	7	8	0	0	0
5	0	0	0	0	0	0	0	9	0	10
6	0	0	0	0	0	0	11	0	5	0
7	0	0	0	0	0	0	0	0	0	16
8	0	0	0	0	0	0	0	0	0	17
9	0	0	0	0	0	0	0	0	0	21
10	0	0	0	0	0	0	0	0	0	0
Вар.19	1	2	3	4	5	6	7	8	9	10
1	0	18	16	14	0	0	0	0	0	0
2	0	0	0	0	19	0	12	0	0	0
3	0	0	0	0	0	18	0	0	0	0
4	0	0	0	0	0	12	12	0	0	0
5	0	0	0	0	0	0	0	15	0	15
6	0	0	0	0	0	0	11	0	15	0

7	0	0	0	0	0	0	0	0	0	16
8	0	0	0	0	0	0	0	0	0	11
9	0	0	0	0	0	0	0	0	0	11
10	0	0	0	0	0	0	0	0	0	0
Вар.20	1	2	3	4	5	6	7	8	9	10
1	0	81	61	41	0	0	0	0	0	0
2	0	0	0	0	91	0	21	0	0	0
3	0	0	0	0	0	81	0	0	0	0
4	0	0	0	0	0	21	21	0	0	0
5	0	0	0	0	0	0	0	51	0	51
6	0	0	0	0	0	0	11	0	51	0
7	0	0	0	0	0	0	0	0	0	61
8	0	0	0	0	0	0	0	0	0	21
9	0	0	0	0	0	0	0	0	0	31
10	0	0	0	0	0	0	0	0	0	0

Вопросы для самоконтроля к главе 12

1. Какие переменные и структуры данных называются статическими?
2. Какие переменные и структуры данных называются динамическими?
3. В чем разница между статическими и динамическими переменными?
4. В чем разница между статическими и динамическими структурами данных?
5. Дайте краткую характеристику методу динамического распределения памяти.
6. Перечислите характеристики динамической структуры данных.
7. Для чего нужны указатели?
8. Приведите примеры ситуаций, когда лучше использовать динамические, а не статические структуры данных. С чем это связано?
9. Каков порядок работы с динамическими структурами?
10. Перечислите основные виды динамических структур. Дайте краткую характеристику каждому виду.
11. Какова структура односвязного списка?
12. Любой ли список является связным? Обоснуйте ответ.
13. Как создать узел односвязного списка в Python? Приведите пример программного кода.

14. Какими способами в Python можно создать и инициализировать значениями объект односвязного списка? Приведите пример программного кода.
15. Как в Python связать узлы односвязного списка между собой? Приведите пример программного кода.
16. Как в Python можно добавить элемент в односвязный список? Приведите пример.
17. В чем принципиальные отличия выполнения добавления элемента на первую и любую другую позиции в однонаправленном списке?
18. Как в Python произвести обход односвязного списка?
19. Как в Python удалить элемент односвязного списка? Приведите пример программного кода.
20. В чем принципиальные отличия выполнения удаления элемента на первую и любую другую позиции в однонаправленном списке?
21. Почему при работе с однонаправленным списком необходимо позиционирование на первый элемент списка?
22. Какова структура двусвязного списка?
23. Как в Python создать узел двусвязного списка? Приведите пример.
24. Какими способами в Python можно создать и инициализировать значениями объект двусвязного списка? Приведите пример.
25. Как в Python связать узлы двусвязного списка между собой? Приведите пример.
26. Какими способами в Python можно вставить элмент в двусвязный список? Покажите на примерах программного кода.
27. В чем принципиальные отличия выполнения добавления элемента на первую и любую другую позиции в двунаправленном списке?
28. Покажите на примере как в Python произвести обход двусвязного списка?
29. Покажите на примере как в Python удалить элемент двусвязного списка?
30. В чем принципиальные отличия выполнения удаления элемента на первой и любой другой позиции в двунаправленном списке?
31. Почему при работе с двунаправленным списком не обязательно позиционирование на первый элемент списка?
32. В чем отличие первого элемента однонаправленного (двунаправленного) списка от остальных элементов этого же списка?
33. В чем отличие последнего элемента однонаправленного (двунаправленного) списка от остальных элементов этого же списка?

34. В чем принципиальные отличия выполнения добавления (удаления) элемента на первую и любую другую позиции в однонаправленном списке?
35. В чем принципиальные отличия выполнения основных операций в однонаправленных и двунаправленных списках?
36. С какой целью в программах выполняется проверка на пустоту однонаправленного (двунаправленного) списка?
37. С какой целью в программах выполняется удаление однонаправленного (двунаправленного) списка по окончании работы с ним? Как изменится работа программы, если операцию удаления списка не выполнять?
38. Какова структура циклического списка?
39. В чем принципиальное отличие циклического списка от одно- и дву- направленного списков? Продемонстрируйте на примерах программного кода.
40. Покажите на примерах различные способы удаления (добавления) элемента циклического списка в Python.
41. Покажите на примере программного кода Python поиск элемента циклического списка.
42. Чем очередь отличается от списков и стека? Продемонстрируйте на примерах.
43. Чем дек отличается от списков, стека и очереди? Продемонстрируйте на примерах.
44. Приведите примеры ориентированных и неориентированных графов.
45. Какие способы представления графов вам известны? Чем они отличаются друг от друга?
46. Дайте определение графа. Перечислить способы представления графа для решения задач с помощью компьютера.
47. Постройте ориентированный граф с пятью вершинами. Представьте граф: а) матрицей смежности; б) списком смежности в) множеством смежности.
48. Постройте ориентированный граф с пятью вершинами. Представьте граф в виде структуры с оглавлением.
49. Перечислите переменные, которые необходимы для программы обхода графа в ширину/ в глубину. Объясните назначение этих переменных.

Тест самоконтроля к главе 12

1. К ссылочным типам данных в Python относятся…а) числа; б) списки; в) булевы значения; г) строки.
2. Как называется область хранения данных ссылочного типа? а) оперативная память; б) куча; в) хэш-память; г) стек.
3. Структура компоненты любой динамической структуры представляет из себя: а) запись, хранящая длину и значения элементов динамической структуры; б) запись, хранящая значения элементов динамической структуры; в) запись, содержащую два поля: указатель на адрес следующего элемента и значение текущего элемента; г) запись, хранящая индексы и значения элементов динамической структуры.
4. Элемент одно- или двусвязного списка называется …. а) индекс; б) куча; в) поле; г) узел.
5. Значение указателя в самом последнем узле динамической структуры в Python равно … а) null; б) none; в) 0; г) zero.
6. В самом начале значение указателя в первом узле динамической структуры в Python равно … а) null; б) none; в) 0; г) zero.
7. Выберите одно верное утверждение о конструкции list Python: а) списки имеют фиксированный размер; б) списки в Python являются изменяемыми; в) списки невозможно отсортировать.
8. Какой максимальный размер у списка (list) в Python? а) фиксированного размера нет; б) 10 тыс.элементов; в) 100 тыс.элементов; г) 1 млн.элементов.
9. Имеются следующие параметры: a – количество рекурсивных вызовов; b – коэффициент, на который размер входных данных сжимается перед рекурсивными вызовами; d – показатель степени в границе объема работы, выполняемой вне рекурсивных вызовов. Выберите вариант, в котором значения a, b и d соответствуют алгоритму двоичного поиска: а) 1; 2; 0; б) 2; 1; 1; в) 2; 2; 2; г) 1; 1; 2.
10. Какова временная сложность алгоритма проверки на дубликаты массива A длиной n?

```
for i in range (1, n):
    for j in range(i + 1, n):
    if A[i] == A[j]: return TRUE
```

else: return FALSE

а) O (1); б) O (log n); в) O(n^2); г) O(n).

11. Алгоритм Беллмана-Форда находит в ориентированном графе кратчайшие пути от исходной вершины до всех остальных. Каким будет время работы алгоритма как функции от m (числа ребер) и n (числа вершин)? а) O (mn); б) O(n^2); в) O(n^3); г) O(mn^2).
12. Имеется куча с n объектами. Какую из задач ниже можно решить с помощью операций Вставить и Извлечь минимум с временем O(1) и дополнительной работы с временем O(1)? а) найти объект с максимальным ключем; б) ни один из перечисленных вариантов; в) найти объект с медианным ключем; г) найти объект с пятым наименьшим ключем.
13. У неориентированного графа ребра есть …. а) тройки вида (V, E, R); б) неупорядоченные пары; в) упорядоченные пары; г) ребро из вершины v_1в вершину v_2, где $v_1 = v_2$.
14. Петля в графе есть … а) тройки вида (V, E, R); б) неупорядоченные пары; в) упорядоченные пары; г) ребро из вершины v_1в вершину v_2, где $v_1 = v_2$.
15. У ориентированного графа ребра есть …. а) тройки вида (V, E, R); б) неупорядоченные пары; в) упорядоченные пары; г) ребро из вершины v_1в вершину v_2, где $v_1 = v_2$.
16. Простой путь в графе – это …а) путь, в котором количество ребер равно n; б) количество вершин нечетно; в) все вершины различны; г) все вершины попарно различны.
17. Длина пути в графе – это …а) количество n ребер в пути; б) количество n вершин в пути; в) количество n ребер и вершин пути.
18. Расстояние между вершинами в графе – это …а) длина любого пути из одной вершины в другую; б) длина кратчайшего пути из одной вершины в другую; в) длина самого длинного пути из одной вершины в другую.
19. Дерево – это …. структура данных: а) иерархическая; б) базовая; в) сетевая; г) неориентированная.
20. Дерево – структура данных, имеющая …а) начальный узел; б) ребра; в) связи с другими узлами; г) набор узлов, связанных между собой.

21. Начальный узел дерева называется …. а) потомком; б) дочерними узлами; в) корнем; г) родителем.
22. Дек – это … а) двухсторонняя очередь; б) стек, имеющий два конца; в) линейная структура данных, которая следует порядку «последним пришел, первым ушел»; г) линейная структура данных, которая следует порядку «первым пришел, первым ушел».
23. Стек –это … а) двухсторонняя очередь; б) стек, имеющий два конца; в) линейная структура данных, которая следует порядку «последним пришел, первым ушел»; г) линейная структура данных, которая следует порядку «первым пришел, первым ушел».
24. Очередь – это … а) двухсторонняя очередь; б) стек, имеющий два конца; в) линейная структура данных, которая следует порядку «последним пришел, первым ушел»; г) линейная структура данных, которая следует порядку «первым пришел, первым ушел».

12.9. Дополнительные источники по работе с динамическими структурами данных в Python

1. Хайнеман Дж., Поллис Г., Сеяков С. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд.: Пер. с англ. – СпБ.: ООО "Альфа-книга", 2017. – 432 с .: ил.
2. Ахмад Имран. 40 алгоритмов, которые должен знать каждый программист на Python. — СПб.: Питер, 2023. — 368 с.: ил.
3. Создание односвязного списка. Режим доступа: https://www.techiedelight.com/ru/linked-list-implementation-python/
4. Вставка, удаление и поиск элементов связного списка. Режим доступа: https://ru.hexlet.io/courses/basic-algorithms/lessons/linked-list/theory_unit
5. Работа с односвязным списком. Режим доступа: https://rukovodstvo.net/posts/id_1160/#insertingitems
6. Двусвязный список. Режим доступа: https://tonais.ru/list/dvusvyazniy-spisok-python
7. Односвязный и двусвязный списки. Режим доступа: https://intuit.ru/studies/courses/648/504/lecture/11456
8. Циклические списки. Режим доступа: https://habr.com/ru/companies/otus/articles/470828/

9. Реализация стека в Python. Режим доступа: https://www.techiedelight.com/ru/stack-implementation-python/
10. Стек, дек и очередь в Python. Режим доступа: https://www.youtube.com/watch?v=G8CBmWg31sg
11. Реализация очереди в Python. Режим доступа: https://www.techiedelight.com/ru/queue-implementation-python/
12. Реализация данных типа дек с помощью модуля collection. Режим доступа: https://pythonpip.ru/osnovy/modul-deque-v-python-i-ego-metody
13. Деревья. Бинарные деревья. Режим доступа: https://www.youtube.com/watch?v=DQ8lZT3zLdM&list=PLA0M1Bcd0w8x4jEp1r_aN3xlnlbfx9RQ2&index=18
14. Способы обхода вершин бинарного дерева. Режим доступа: https://www.youtube.com/watch?v=oYyEqfi_4fo&list=PLA0M1Bcd0w8x4jEp1r_aN3xlnlbfx9RQ2&index=19
15. Реализация бинарного дерева на Python. Режим доступа: https://www.youtube.com/watch?v=mdkwm5FUpFs&list=PLA0M1Bcd0w8x4jEp1r_aN3xlnlbfx9RQ2&index=21
16. Фомирование бинарного дерева. Режим доступа: https://www.youtube.com/watch?v=mdkwm5FUpFs&list=PLA0M1Bcd0w8x4jEp1r_aN3xlnlbfx9RQ2&index=22
17. Понятие графа. Режим доступа: https://habr.com/ru/articles/112421/
18. Обход графа в глубину. Режим доступа: https://www.youtube.com/watch?v=Tzc7Z-mOwxY
19. Обход графа в ширину. Режим доступа: https://www.youtube.com/watch?v=4A5vN9p0YTY
20. Реализация алгоритма в ширину на Python. Режим доступа: https://yandex.ru/video/preview/7951851896801434488?t=0
21. Алгоритм Дейкстры. Режим доступа: https://www.youtube.com/watch?v=MCfjc_UIP1M&t=93s
22. Алгоритм Флойда. Режим доступа: https://www.youtube.com/watch?v=ipWZ-d1l00s

Глава 13. БИБЛИОТЕКИ (ПАКЕТЫ) PYTHON

Язык Python стал популярен, благодаря своей коллекции библиотек. Рассмотрим основные из них.

13.1. Библиотека NumPy

У этой библиотеки есть несколько важных особенностей, которые сделали ее популярным инструментом.

Во-первых, исходный ее код в свободном доступе хранится на GitHub, поэтому NumPy называют open-source модулем для Python.

Во-вторых, библиотека написана на языках C и Fortran. Это компилируемые языки (языки программирования, текст которых преобразуется в машинный код — набор инструкций для конкретного типа процессора). Преобразование происходит с помощью специальной программы-компилятора, на которой вычисления производятся гораздо быстрее и эффективнее, чем на интерпретируемых языках, к которым относится и сам Python.

В-третьих, NumPy применима для научных вычислений, с ее помощью решаются научные задачи физики, математики, химии и даже когнитивной психологии. Полное название библиотеки – Numerical Python extensions, или «Числовые расширения Python».

В состав входят средства:

- численных вычислений:
    - комплексные математические функции, генераторы случайных чисел, процедуры линейной алгебры, преобразования Фурье и т.д.;
- обработки массивов:
    - многомерных массивов, маскированных массивов и матриц, набор подпрограмм для быстрых операций с массивами, включая математические, логические, манипуляции с фигурами, сортировку, выбор, ввод-вывод.

В-четвертых, вычислительные мощности библиотеки позволяют использовать ее в машинном обучении и анализе данных.

Подключить пакет NumPy к своему проекту можно командой:

```
import numpy as np
```

Если команда выполняется без ошибок, пакет присутствует на вашем устройстве. В противном случае его можно установить командой:

```
pip install numpy
```

13.2. Другие библиотеки, поддерживающие Python

TensorFlow. Базовый язык C++, но имеет API для Python. Разработана компанией Google. Вычисления производятся с использованием графов потоков данных. Поддерживает:

- числовые вычисления на основе многомерных массивов (аналогично NumPy);
- GPU и распределенная обработка данных;
- автоматическое дифференцирование;
- построение, обучение и экспорт модели для машинного обучения и т.д.

Theano. Базовый язык Python. Разработана университетом Монреаля. Вычисления производятся с использованием графов потоков данных. Эффективна в обработке многомерных массивов. Вычисления выражаются NumPy-подобным синтаксисом.

PyTorch. Базовый язык Lua. Библиотека верхего уровня (API) для языков Lua, Python, Java, C++ разных уровней абстракции. Разработана Ронаном Коллабертом. Имеет множество модульных элементов, которые легко комбинировать. Эта особенность позволяет легко работать на GPU.

CNTK – набор средств с открытым кодом для коммерческого распределенного глубокого обучения. Базовый язык C++. Поддерживает API для C++, C#, Python, Java. Разработана Microsoft. Описывает нейронные сети как ряд вычислительных шагов с помощью направленного графа. Обеспечивает скорость обучения, сравнимую с библиотекой TensorFlow, а на рекуррентных сетях превосходит ее. Имеет API разных уровней абстракции. Позволяет пользователю легко реализовать и объединить популярные типы моделей нейронных сетей, такие как перенаправленные каналы (DNN), сверточная нейронная сеть (CNN) и рекуррентные нейронные сети (RNN/LSTMs).

Caffe – платформа глубокого обучения, разработанная подразделением компании Microsoft Berkeley Vision and Learning Center (BVLC). Базовый язык C++. Обеспечивает скорость обучения, сравнимую с TensorFlow, а на рекуррентных сетях превосходит его. Имеет

API для C++, C#, Python, Java разных уровней абстракции, позволяющих поддерживать различные типы архитектур глубокого обучения, таких как CNN, LSTM и FC.

Keras – библиотека верхнего уровня (API), написанная на Python. Имеет узкую специализацию – это инструмент для специалистов по машинному обучению, которые работают с языком Python. Используется в качестве вычислительной основы библиотеки TensorFlow и Theano. Позволяет создавать и обучать нейронные сети на очень высоком уровне абстракции.

Помимо указанных библиотек, что для решения задач машинного обучения существуют очень мощные и удобные библиотеки, такие как Scikit-learn, Xgboost, LightGBM, CatBoost. Ряд задач с разнородными табличными данными до сих пор решается лучше именно с помощью них, а не нейронных сетей.

Scikit-learn одна из наиболее широко используемых библиотек для Data Science и Machine Learning, так как содержит методы, алгоритмы и инструменты, необходимые для решения задач классификации, регрессии, кластеризации, а также оценки производительности модели машинного обучения. Построена на библиотеках Numpy, sciPy и matplotlib в Python. Имеет открытый исходный код.

Xgboost — библиотека с открытым исходным кодом, используемая в машинном обучении и предоставляющая функциональность для решения задач, связанных с регуляризацией градиентного бустинга – алгоритма машинного обучения для задач классификации и регрессии, строящего модель предсказания в форме ансамбля слабых предсказывающих моделей, обычно деревьев решений. ПомимоPython, библиотека, поддерживается такими языками программирования как C++, Java, R, Julia, Perl и Scala. Библиотека работает под ОС Linux, Windows, и macOS. Может работать как на одной машине, так и на системах распределенной обработки Apache Hadoop, Apache Spark иApache Flink. В последнее время эта библиотека приобрела большую популярность и привлекла внимание как выбор многих команд-победителей соревнований по машинному обучению.

LightGBM – библиотека, предоставляющая функциональность для решения задач, связанных с регуляризацией градиентного бустинга, и принятия решений. Получила широкую известность, благодаря более высокой скоростью обучения по сравнению с другими библиотеками, малым использованием памяти и возможностью обработки больших датасетов.

CatBoost – библиотека, созданная инженерами и специалистами Яндекса в качестве преемника Матрикснета — алгоритма, применяемого для ранжирования и прогнозирования. CatBoost использует более универсальный алгоритм, который умеет напрямую работать не только с числовыми, но и с категориальными признаками.

Вопросы для самоконтроля к главе 13

1. Дайте краткую характеристику каждой из указанных в тексте параграфа 13.2 библиотек. Что у них общего? Чем отличаются?
2. Дайте краткую характеристику библиотекам Scikit-learn, Xgboost, LightGBM, CatBoost.

Тест самоконтроля к главе 13

1. Какую операцию выполняет данный фрагмент кода: d = np.eye(5): а) создает пустой массив; б) удаляет из массива 5 элементов; в) создает единичную матрицу; г) очищает массив.
2. Какой из перечисленных ниже методов библиотеки numpy используется для для объединения массивов по горизонтали? а) hstack(); б) stack(); в) hor_stack(); г) stackH().
3. Какая функция библиотеки numpy отвечает за создание массива из заданного количества чисел? а) linspace(); б) range(); в) space(); г) num_range().
4. Какая функция библиотеки numpy отвечает за создание массива из одних нулей? а) zero(); б) zero_array(); в) zeros(); г) empty().
5. В результате выполнения кода на экран будет выведено:
   b = np.array([[1.5, 2, 3], [4, 5, 6]])
   d = b.copy()
   d is a
   а) True; б) False; в) d=[[1.5, 2, 3], [4, 5, 6]]; г) [1.5, 2, 3, 4, 5, 6].
6. Какой метод библиотеки numpy используется для изменения формы массива? а) resize(); б) reshape(); в) rearray(); г) recreate().
7. Какой метод библиотеки numpy отвечает за объединение массивов по вертикали? а) ver_stack(); б) vstack(); в) stack(); hstack().
8. Функция empty() библиотеки numpy: а) создает массив со случайным содержимым; б) очищает массив; в) создает пустой массив; г) удаляет массив.

9. Какой из методов библиотеки numpy отвечает за создание массива, используя кортежи? а) tuple_num_array(); б) arrays(); в) array_tuple(); г) numpy.array().
10. Каков правильный синтаксис для создания массива NumPy? а) np.object ([1, 2, 3, 4, 5]); б) np.array ([1, 2, 3, 4, 5]); в) np.createArray ([1, 2, 3, 4, 5]); г) np.Createarray ([1, 2, 3, 4, 5]).
11. Какой из следующих массивов является двумерным (2D) массивом? а) [1, 2, 3, 4, 5]; б) [[1, 2, 3], [4, 5,6]]; в) [[1, 2], [3, 4], [5,6]]; г) 42.
12. Каков правильный синтаксис для проверки количества измерений в массиве? а) arr.ndim(); б) arr.dim(); в) arr.dim; г) arr.ndim.
13. Каков правильный синтаксис для печати первого элемента массива? а) print (myArr[0]); б) print (myArr,1); в) print (myArr[1]); г) print (myArr).
14. Каков правильный синтаксис для печати числа 8 из массива ниже: arr = np.array([[1,2,3,4,5], [6,7,8,9,10]]). а) print (arr[1,2]); б) print (arr[3,0]); в) print (arr[7,2]); г) print (8).
15. Каков правильный синтаксис для печати чисел [3, 4, 5] из массива ниже: arr = np.array([1,2,3,4,5,6,7]. а) print (arr[2,6]); б) print (arr[3,6]); в) print (arr[2:4]); г) print (arr[2:5]).
16. Какой синтаксис будет печатать последние 4 числа из массива ниже: arr = np.array([1,2,3,4,5,6,7]). а) print (arr[3:]); б) print (arr[4]); в) print (arr[:4]); г) print (arr[4:]).
17. Каков правильный синтаксис для проверки типа данных массива? а) arr.type; б) arr.dtype; в) arr.ntype; г) arr.datatype.
18. Каков правильный синтаксис для создания массива типа float? а) arr=np.array([1, 2, 3, 4]).toFloat(); б) arr=np.float ([1, 2, 3, 4]); в) arr=np.array([1, 2, 3, 4]),dtype='f').
19. Что означает ФОРМА массива в NumPy? а) количество столбцов; б) количество элементов в каждом измерении; в) количество рядов.
20. Каков правильный синтаксис для возврата формы массива? а) arr.shape(); б) arr.shape; в) shape (arr).
21. Каков правильный метод объединения двух или более массивов? а) join(); б) array_join(); в) concatenate().
22. Каким из указанных методов можно разделить массив? а) vstack(); б) hstack(); в) все три метода корректные; г) array_split().
23. Каков правильный метод поиска определенного значения в массиве? f) where(); б) find (); в) search ().

24. Каков правильный синтаксис для возврата индекса всех элементов со значением 4 из массива arr = np.array([1,4,3,4,5,4,4])? а) np.where (arr==4); б) arr.search(4); в) arr.where ().
25. Каков правильный метод сортировки элементов массива? а) sort (); б) orderby (); в) order ().
26. Как при использовании модуля random библиотеки NumPy вернуть целое число из диапазона от 0 до 100? а) random.rand (100); б) random.rand (); в) random.randint (); г) random.randint (100).
27. Как при использовании модуля random библиотеки NumPy вернуть нормальное распределение данных с 1000 чисел, сосредоточенных вокруг числа 50, со стандартным отклонением 0,2? а) random.normal (size=1000, normal=50, s=0.1); б) random.normal (size=1000, mean=50, devation=0.2); в) random.normal (size=1000, loc=50, scale=0.2).
28. Каков правильный синтаксис для математического добавления чисел массива arr1 к числам массива arr2? а) sum (arr1, arr2); б) np.add(arr1, arr2); в) np.append (arr1, arr2).
29. Каков правильный синтаксис для вычитания чисел из arr1 с числами из arr2? а) np.sub (arr1, arr2); б) np.minus (arr1, arr2); в) np.subtract (arr1, arr2); г) np.min (arr1, arr2).
30. Каков правильный метод округления десятичных знаков в NumPy? а) np.around (); б) np.trunc (); в) np.fix (); г) все три метода корректные.
31. Что будет выведено на экран в результате кумулятивного суммирования NumPy? arr = np. array ([1,2,3]) print (np.cumsum (arr)). а) [1 3 6]; б) [9]; в) [6]; г) [3 6 9].
32. Какой модуль не входит в стандартную библиотеку Python? а) lib; б) ABC; в) requests; г) sqlite.
33. Какой модуль стандартной библиотеки Python содержит итератор, бесконечно генерирует числа? а) iterators; б) collections; в) functools; г) itertools.
34. Модуль collections не содержит …а) список; б) стек; в) кортеж; г) множество.
35. В каком модуле содержатся функции для работы с комплексными числами? а) math; б) cmath; в) complex; г) все эти функции доступны без импортирования дополнительных модулей.

13.3. Дополнительные источники по работе с библиотеками

1. Сборник библиотек. Режим доступа: http://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy
2. Библиотека Numpy: установка и первое знакомство. Режим доступа: https://www.youtube.com/watch?v=eDuuKvIWzew&t=12s
3. Официальный сайт библиотеки Numpy. Режим доступа: www.numpy.org
4. Исходный код библиотеки Numpy. Режим доступа: https://github.com/numpy/numpy
5. Официальный сайт библиотеки Keras. Режим доступа: https://ru-keras.com/home/
6. Официальный сайт библиотеки Tensorflow. Режим доступа: https://www.tensorflow.org/?hl=ru
7. Описание библиотек Theano, PyTorch, Tensorflow, Keras, Numpy, CNTK. Режим доступа: https://waksoft.susu.ru/2021/01/19/9-luchshih-bibliotek-python-dlya-speczialistov-po-dannym-i-inzhenerov-po-mashinnomu-obucheniyu/
8. Описание библиотеки Caffe. Режим доступа: https://ru.wikipedia.org/wiki/Caffe
9. Как начать работу с библиотекой Keras? Режим доступа: https://timeweb.com/ru/community/articles/kak-nachat-rabotu-s-keras
10. Библиотека Scikit-learn. Режим доступа: https://biconsult.ru/products/rukovodstvo-po-scikit-learn-dlya-mashinnogo-obucheniya
11. Библиотека LightGBM. Режим доступа: https://pythonru.com/biblioteki/lightgbm
12. Библиотека CatBoost. Режим доступа: https://yandex.ru/dev/catboost/

Глава 14. СИСТЕМЫ КОНТРОЛЯ ВЕРСИЙ

14.1. Базовые понятия

Контроль версий – это практика отслеживания изменений программного кода и управления ими.

Система контроля версий (СКВ) — это программные инструменты управления изменениями в исходном коде в процессе разработки программного обеспечения.

При разработке программного обеспечения рано или поздно приходится вносить сложные исправления (или изменения). Если проект небольшой, то, обычно, делается его резервная копия. Но что делать, если вы сопровождаете огромный проект, состоящий из сотен и даже тысяч файлов?

Копировать каждый раз весь проект и вручную описывать версию слишком трудоемко и затратно по памяти. Не мудрено запутаться в версиях и использовать, вместо сохраненной правильной версии, промежуточную недоделанную версию с массой ошибок. Хорошо, если потом можно быстро откатиться на правильную версию, но ведь бывают случаи, когда, наводя порядок в архиве, удаляются единственно верные копии с последними наработками.

Все это значительно усложняется, когда проект ведут несколько человек, иногда территориально удаленных друг от друга на сотни и тысячи километров. В этом случае у каждого будут образовываться свои архивы версий, и начнется настоящий кошмар сопровождения разработанного программного обеспечения.

Сообщество разработчиков пришло к выводу о необходимости специализированного программного обеспечения, позволяющего просто и надежно сопровождать большие программные проекты. И в скором времени появилось множество программ для контроля версий (Git, CVS, Subversion, Bazaar, Monotone, Aegis и др.), позволяющих удовлетворять любые, даже самые изощренные пожелания. Интерфейсы некоторых СКВ Git представлена на рис.124аб.

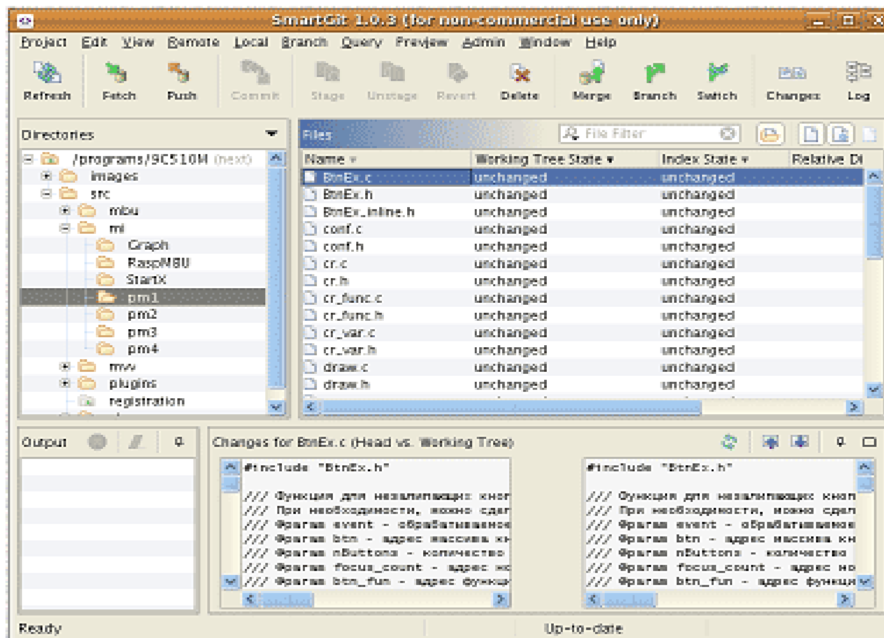


Рис.124а. Интерфейс СКВ Git

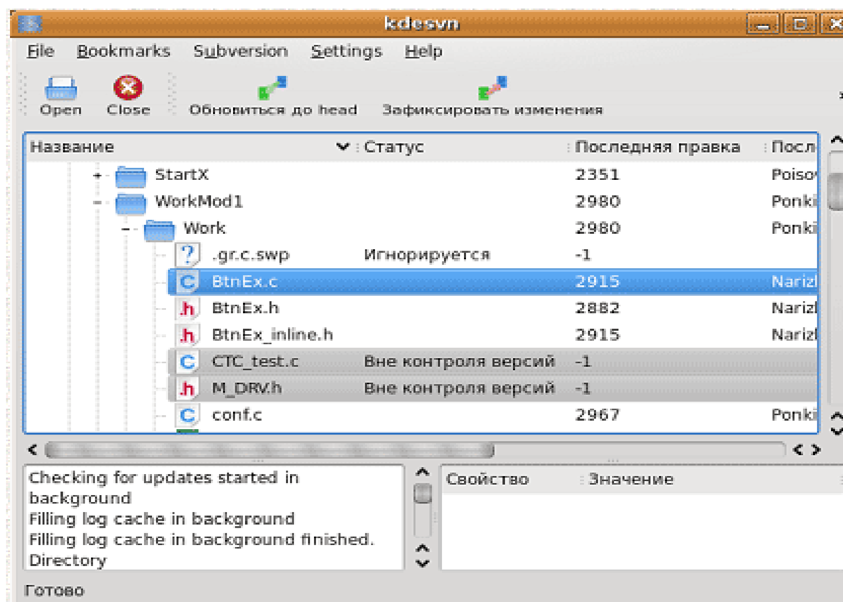


Рис.124б. Интерфейс СКВ Subversion

СКВ позволяют:

1) сохранять все этапы разработки. Внося изменения в один или несколько файлов проекта, программист записывает изменения в репозиторий – хранилище всех версий и изменений проекта. Таким образом, сохраняется не весь проект целиком, в репозиторий добавляются только файлы, претерпевшие изменения.
2) обновлять ПО клиентов до последней версии разработанного программного обеспечения. Обычно над разработкой проектов

трудится целая команда специалистов, они постоянно добавляют в репозиторий измененные файлы, поэтому одной из основных задач СКВ является возможность отслеживать все эти изменения и быстро обновлять ПО клиентов до актуальной версии.

3) объединять изменения. Часто несколько программистов одновременно изменяют одни и те же файлы. Если изменения не пересекаются, то системы контроля версий позволяют легко и просто объединить эти изменения.
4) решать конфликты. Если несколько человек изменили один и тот же участок кода, то, автоматически объединить такие изменения невозможно. В этом случае СКВ предоставляют инструменты, позволяющие вручную внести необходимые правки в текст программ, а затем автоматически объединить конфликтующие части кода.
5) откатываться к предыдущим версиям. Если выбранное направление в развитии проекта оказалось тупиковым или содержащим ошибки, СКВ позволяют вернуть разработанное ПО к одной из последней рабочей версии, просто, скопировав из репозитория нужную версию ПО, либо отдельные файлы.
6) сопровождение нескольких направлений развития ПО. Не всегда можно сразу сохранять внесенные изменения. Часто приходится достаточно долго разрабатывать и отлаживать отдельные правки прежде, чем их можно объединить с основным программным обеспечением. В этом случае многие СКВ позволяют организовывать параллельные ветки по контролю нескольких направлений развития ПО, быстро переключаться между ними, а затем объединять их в единое целое.

Кроме перечисленного, СКВ обладают удобным интерфейсом для быстрого и простого выполнения перечисленных выше действий и надежного контроля версий разрабатываемого программного обеспечения.

Каждой СКВ присущи свои достоинства и недостатки, но, в общем, все они основываются на одном и том же принципе: регистрации изменений в программном коде и сохранение каждого нового обнаруженного изменения в новой версии, к которой можно вернуться в любой момент времени.

Несмотря на единый принцип работы, СКВ можно разделить на две группы: ***централизованные*** и ***распределенные***. У них много общего, но есть и принципиальные отличия.

Централизованные системы, такие как CVS и Subversion, для сохранения всех рабочих файлов контролируемого проекта используют репозиторий, размещенный на отдельном сервере (рис.125).

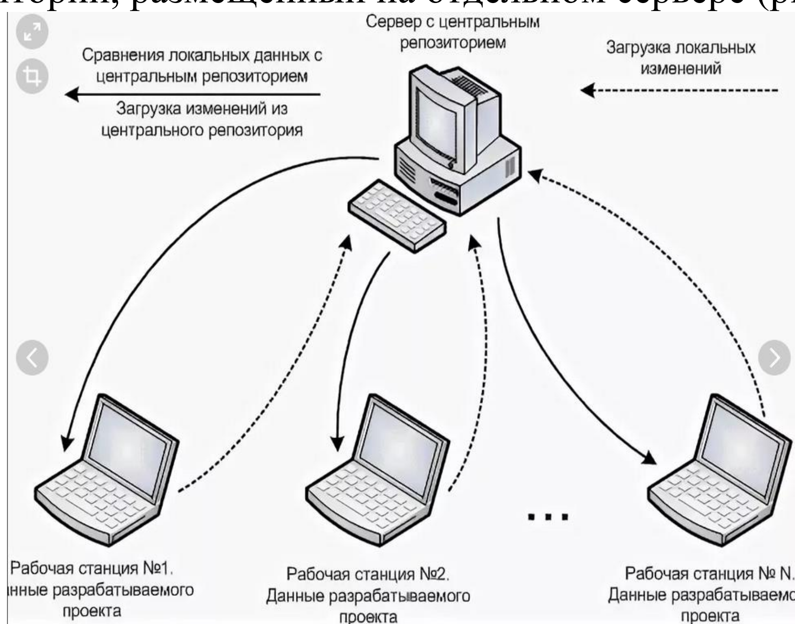


Рис.125. Структура централизованной СКВ

В этом случае для работы с СКВ каждый участник проекта сначала скачивает с сервера последнюю версию программного продукта, вносит в нее свои изменения и загружает на сервер полученный результат. При этом работа ведется с последней версией программного продукта, а, если необходимо вернуться к одной из предыдущих версий разработки, приходится каждый раз запрашивать сервер и скачивать необходимую версию.

Недостатки централизованных СКВ

После внесения всех корректировок в скаченную версию, она заливается на сервер в качестве последней версии разрабатываемого продукта. При этом, скорее всего, придется решать множество конфликтов, что бывает очень сложным, так как откат на предыдущую версию может затронуть большое количество уже сохраненных изменений.

В ***распределенных системах***, таких как Git (рис.126), когда пользователи загружают данные из репозитория сервера, скачиваются все сохраненные изменения, а не только последняя версия. Естественно, каждый раз скачивать весь репозиторий не нужно, так как

достаточно скопировать только те изменения, которых нет в локальном проекте пользователя.

Несмотря на это, операции копирования данных из репозитория могут быть достаточно долгими, но это с лихвой окупается при дальнейшем использовании распределенной системы контроля версий.

По сути, используя распределенные системы контроля, каждый пользователь имеет свой личный репозиторий, в который он локально сохраняет все свои изменения. При необходимости создает параллельные ветки контроля версий проекта, отслеживающие сложные изменения, которые пока нельзя сохранять в основной версии разрабатываемого проекта.

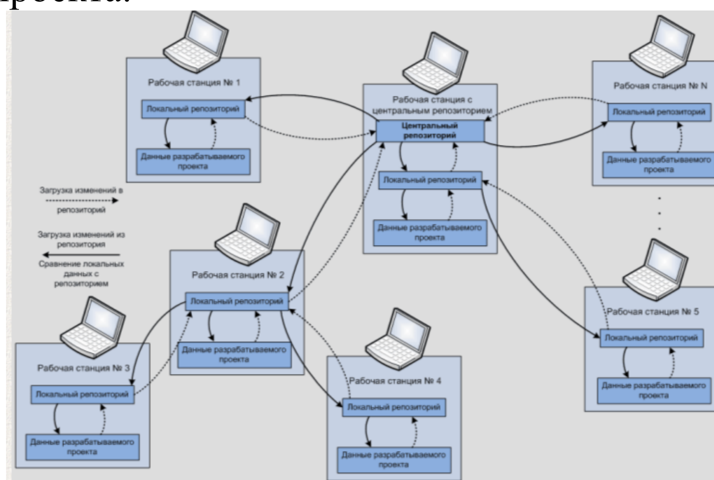


Рис.126. Структура распределенной СКВ

Стоит отметить, что назначение СКВ не ограничивается сопровождением разработки ПО. Их можно использовать для ведения документации, управления почтой, синхронизации данных на нескольких компьютерах в сети, а также для многих других задач.

На данный момент СКВ все более прочно входят в мир высоких технологий, и уже сейчас без них не мыслима разработка серьезных проектов.

14.2. Управление версиями в среде PyCharm

Инструкция по работе с Git в PyCharm:

1) включите систему контроля версий VCS- Enable Version Control Integration. Для этого в Главном меню выберите вкладку VCS, VCS Operations (рис.127):

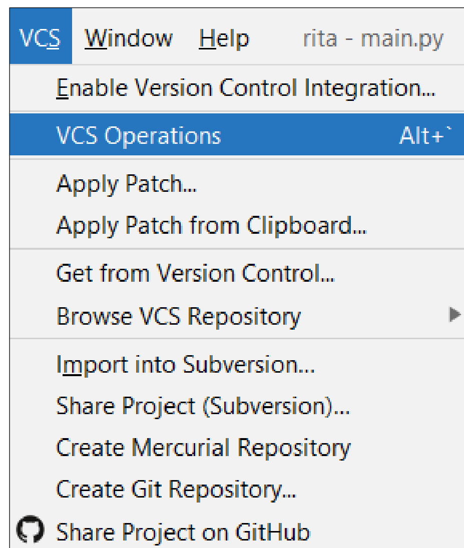


Рис.127. Включение системы контроля версий

2) в выпадающем списке выберите Create Git Repository (рис.128). Система контроля версий включена;

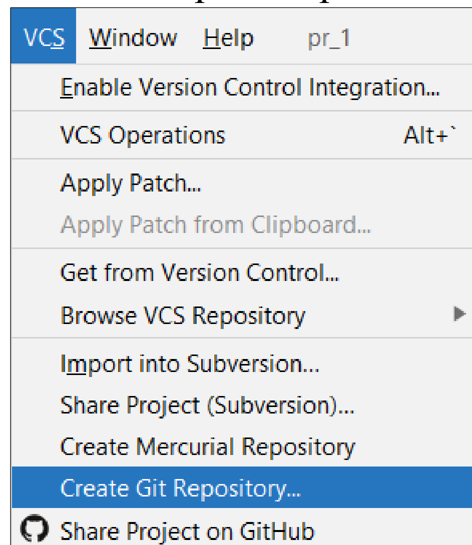


Рис.128. Создание репозитория

3) открываем общий доступ к файлам проекта. Для на вкладке VCS выберите Share Project On GitHub (рис.129). Далее следуйте инструкциям;

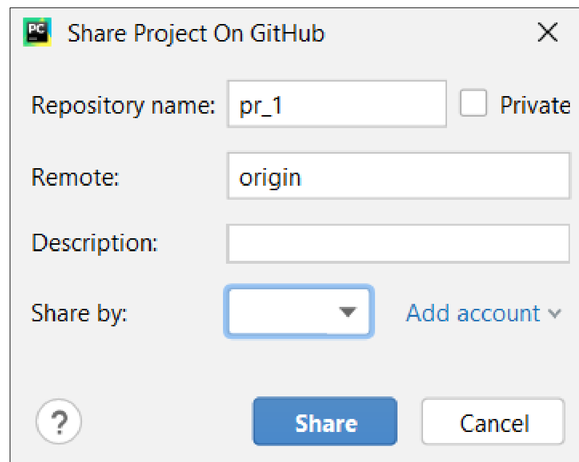


Рис.129. Открытие общего доступа к файлам проекта

4) в корне проекта создаем файл .gitignore добавляем туда же папки venv и .idea (рис.130);

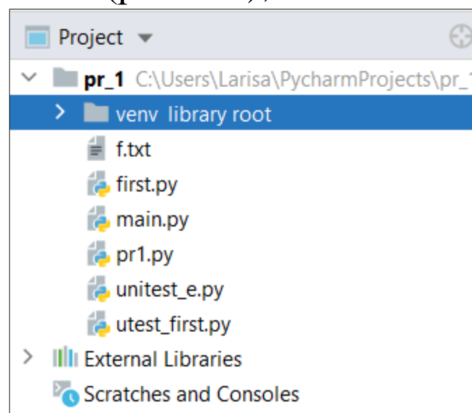


Рис.130. Добавление необходимых папок в проект

5) во вкладке Git нажимаем обновить, выделяем нужные файлы и выбираем Add to VCS (добавить в отслеживание);
6) когда код написан выделяем нужные файлы и выбираем Commit (Сохранить локально). Если хотим увидеть изменения, выбираем Show Diff. Если хотим откатить версию кода назад, то выбираем Rollback;
7) для сохранения на удаленный сервер выбираем Push.

Вопросы для самоконтроля к главе 14

1. Чем вызвана необходимость использования программ СКВ?
2. Что представляет собой СКВ?
3. Приведите примеры программ СКВ.
4. Что представляет собой репозиторий?
5. Перечислите возможности программ контроля версий.
6. Охарактеризуйте принцип работы централизованной СКВ.

7. Охарактеризуйте принцип работы распределенной СКВ.
8. Укажите представителей централизованной СКВ.
9. Укажите представителей распределенной СКВ.
10. Для каких целей можно использовать СКВ кроме сопровождения разработки программных продуктов?

Тест самоконтроля к главе 14

1. Какие из перечисленных ниже программ являются СКВ? а) Git; б) CVN; в) CVS; г) IGT.
2. Что такое GitHub? а) программа для работы с Git; б) драйвер для работы с Git; в) web-сервис для хостинга IT-проектов и их совместной разработки; г) утилита для работы с локальной версией Git.
3. Что из перечисленного ниже может быть репозиторием Git? а) любая директория /папка на моем ПК; б) любая папка, находящаяся в каталоге СКВ Git; в) подкаталог уже существующего каталога в СКВ Git.
4. Какое действие выполняет команда git status? а) Показывает состояние проекта; б) показывает, является ли пользователь авторизованным в СКВ Git; в) показывает путь к репозиторию.
5. Какое действие выполняет команда git add? а) создает файл с указанным именем и добавляет его в СКВ Git; б) добавляет локальный файл в удаленный репозиторий так, чтобы другие участники проекта могли его видеть; в) аналогична команде git commit; г) начинает отслеживать указанный файл(ы).
6. Можно ли отследить хронологию событий с помощью Git? а) да, можно отслеживать только добавленные файлы; б) да, можно отслеживать даты добавления файлов, а также какие файлы добавлены в проект; в) нет, нельзя; г) да, можно отследить дату, автора и изменения, которые были внесены.
7. Сколько человек могут работать в Git над одним проектом? а) не более 10; б) не более 50; в) не более 5; г) неограниченное количество.
8. В каком редакторе можно работать с Git? а) во всех редакторах; б) в редактор Word; в) в редакторе Atom; г) блокнот.
9. Перейти к нужной папке можно командой… а) go имя папки; б) to имя папки; в) open имя папки; г) cd имя папки.

10. Git – это ….а) облачное хранилище; б) сервер для проектов; в) распределенная система управления версиями; г) веб-сервер для хостинга IT-проектов.
11. GitHub – это …а) технология контроля версий проекта; б) графический интерфейс для работы с Git; в) веб-сервис для хостинга IT-проектов.

14.3. Дополнительные ресурсы по работе с СКВ

1. Васильева М.А., Филипченко К.М. Система контроля версий. Основы командной разработки. Учебное пособие. Изд-во: Лань, 2022 г. – 144 с.
2. Начало работы с СКВ Git – Режим доступа: https://blog.skillfactory.ru/glossary/git/
3. Обзор программного обеспечения для управления версиями. Режим доступа: https://bitbucket.org/product/ru/version-control-software
4. Управление ресурсами в среде Pycharm. Режим доступа: https://waksoft.susu.ru/2019/11/07/pycharm-effektivnaya-razrabotka-na-python/#using-version-control-in-pycharm
5. Инструкция по работе с версиями в среде PyCharm. Режим доступа: https://dzen.ru/list/gadgets/pycharm-upravlenie-versiiami

ЗАКЛЮЧЕНИЕ

Учебно-практическое пособие отвечает требованиям, предъявляемым к современной учебно-методической литературе, призванной обеспечить реализацию компетентностного подхода при обучении бакалавров, за счет включения в его содержание помимо теоретических сведений дидактического аппарата издания− контрольных вопросов и заданий, практических работ и тестов. Указанный дидактический инструментарий позволяет проверить уровень овладения студентами конкретной компетенцией.

В результате изучения материалов учебно-практического пособия будущие специалисты смогут использовать основные приемы обработки и представления данных, осуществлять постановку, анализ и исследование задачи, модели, разрабатывать алгоритмы, программировать, тестировать и отлаживать программный код.

ПРИЛОЖЕНИЕ

ОТВЕТЫ К ТЕСТАМ САМОКОНТРОЛЯ

Ответы к тесту самоконтроля к главе 1

№ вопроса	Ответ
1	г
2	в
3	б
4	в
5	б
6	в
7	аб
8	в
9	б
10	а
11	аб
12	г
13	в
14	бв
15	ад
16	бг
17	б
18	г
19	ав

Ответы к тесту самоконтроля главы 2

№ вопроса	Ответ	№ вопроса	Ответ
1	абвг	8	а
2	багвд	9	г
3	бв	10	б
4	а		
5	б		
6	в		
7	в		

Ответы к тесту самоконтроля к главе 3

№ вопроса	Ответ
1	бв
2	бв
3	абе
4	вд
5	д
6	а
7	вг

Ответы к тесту самоконтроля к главе 4

№ вопроса	Ответ
1	а
2	в
3	б
4	в
5	а
6	б
7	диезбеа
8	а
9	в
10	б
11	в
12	б
13	г
14	а
15	б
16	г
17	б
18	в
19	г
20	а
21	б
22	в
23	а
24	г
25	б

Ответы к тесту самоконтроля к главе 5

№ вопроса	Ответ
1	вг
2	бд
3	в
4	а
5	в
6	а
7	г
8	б
9	в
10	г
11	агд
12	бг
13	ав
14	б
15	в
16	г

Ответы к тесту самоконтроля к главе 6

№ вопроса	Ответ
1	б
2	а
3	в
4	г
5	г
6	а
7	в
8	абг
9	г
10	б
11	а
12	а
13	г
14	в
15	в

Ответы к тесту самоконтроля параграфа 7.2.1

№ вопроса	Ответ	№ вопроса	Ответ	№ вопроса	Ответ	№ вопроса	Ответ
1	г	20	д				
2	ав	21	в				
3	бв	22	д				
4	бгд	23	а				
5	а	24	в				
6	г	25	г				
7	б	26	б				
8	б	27	а				
9	г	28	д				
10	аб	29	в				
11	201	30	д				
12	д						
13	а						
14	а						
15	б						
16	а						
17	в						
18	в						
19	в						

Ответы к тесту самоконтроля к параграфам 7.2.2 -7.2.5

№ вопроса	Ответ	№ вопроса	Ответ
1	а	12	абд
2	а	13	б
3	б	14	г
4	г	15	авг
5	б	16	авгд
6	б	17	в
7	г	18	аг
8	а	19	б
9	в	20	бг
10	г	21	бг
11	аб		

Ответы к тесту самоконтроля параграфа 7.2.7

№ вопроса	Ответ	№ вопроса	Ответ
1	б	24	а
2	бвг	25	б
3	г		
4	в		
5	г		
6	в		
7	В		
8	Б		
9	А		
10	Б		
11	Г		
12	Д		
13	Б		
14	А		
15	Б		
16	В		
17	Г		
18	А		
19	Г		
20	В		
21	Б		
22	Б		
23	А		

Ответы к тесту самоконтроля параграфа 7.2.8

№ вопроса	Ответ	№ вопроса	Ответ	№ вопроса	Ответ
1	абг	11	б	21	в
2	а	12	в	22	в
3	а	13	б	23	авде
4	б	14	б	24	г
5	абве	15	г	25	в
6	б	16	а	26	а
7	г	17	в	27	В
8	в	18	в	28	в
9	б	19	г	29	а
10	в	20	в	30	б

Ответы к тесту самоконтроля к главе 8

№ вопроса	Ответ
1	$2 \cdot n+1$
2	$n + 4$
3	$n^2 + 2$
4	в
5	в
6	г
7	д
8	г
9	а
10	б
11	б

Ответы к тесту самоконтроля к главе 9

№ вопроса	Ответ
1	а
2	бв
3	де
4	в
5	д
6	б
7	в
8	г
9	д
10	а

Ответы к тесту самоконтроля к главе 10

№ вопроса	Ответ	№ вопроса	Ответ
1	а	15	а
2	г	16	б
3	г	17	г
4	в	18	г
5	б	19	б
6	а	20	г
7	в	21	в
8	г		
9	б		
10	в		
11	б		
12	в		
13	б		
14	а		

Ответы к тесту самоконтроля к главе 11

№ вопроса	Ответ	№ вопроса	Ответ		
1	а	6	class	12	в
2	в	7	г	13	б
3	б	8	б	14	г
4	в	9	б	15	б
5	г	10	в	16	а
		11	а	17	в
				18	в
				19	абв
				20	г
				21	а
				22	б

Ответы к тесту самоконтроля к главе 12

№ вопроса	Ответ	№ вопроса	Ответ
1	бг	13	б
2	б	14	г
3	в	15	в
4	г	16	г
5	а	17	а
6	а	18	б
7	б	19	а
8	а	20	ав
9	а	21	вг
10	в	22	аб
11	а	23	в
12	г	24	г

Ответы к тесту самоконтроля к главе 13

№ вопроса	ответ	№ вопроса	ответ	№ вопроса	ответ
1	в	14	а	27	б
2	а	15	г	28	б
3	а	16	а	29	в
4	в	17	б	30	г
5	б	18	в	31	а
6	б	19	б	32	а
7	б	20	б	33	а
8	в	21	а	34	б
9	г	22	г	35	б
10	б	23	а		
11	б	24	а		
12	г	25	а		
13	а	26	в		

Ответы к тесту самоконтроля к главе 14

№ вопроса	Ответ	№ вопроса	Ответ
1	ав	7	г
2	в	8	а
3	абв	9	г
4	а	10	в
5	г	11	в
6	г		

Учебное электронное издание

АРТЮШИНА Лариса Андреевна
ТРОИЦКАЯ Елена Анатольевна
СПИРИНА Татьяна Венедиктовна

ПРОГРАММИРОВАНИЕ НА PYTHON 3

Учебно-практическое пособие

Издается в авторской редакции

Системные требования: Intel от 1,3 ГГц; Windows XP/7/8/10;
Adobe Reader; дисковод CD-ROM.

Тираж 25 экз.

Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых
Изд-во ВлГУ
rio.vlgu@yandex.ru

Институт информационных технологий и электроники
кафедра информатики и защиты информации
larisa-artusina@yandex.ru