

Зыкова Г. В.  
Попов А. С.  
Сапуглецева Т. Н.

# ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ PYTHON

Учебно-методическое пособие



• ФЛИНТА •

---

Министерство науки и высшего образования Российской Федерации

Орский гуманитарно-технологический институт (филиал)  
федерального государственного бюджетного образовательного учреждения  
высшего образования «Оренбургский государственный университет»

**Зыкова Г. В., Попов А. С., Сапуглецева Т. Н.**

# **ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ PYTHON**

Учебно-методическое пособие

2-е издание, стереотипное

Москва  
Издательство «ФЛИНТА»  
2020

УДК 004.43  
ББК 32.973.2  
3 96

**Научный редактор:**

**Уткина Т. И.**, доктор педагогических наук, профессор кафедры  
математики, информатики и физики  
Орского гуманитарно-технологического института  
(филиала) ОГУ

**Рецензенты:**

**Голунова А. А.**, кандидат педагогических наук, доцент кафедры  
математики, информатики и физики  
Орского гуманитарно-технологического института  
(филиала) ОГУ;

**Пергунов В. В.**, кандидат физико-математических наук, доцент  
кафедры прикладной информатики и математики  
Орского филиала АОЧУ ВО «Московский финансово-юридический  
университет МФЮА»

**Зыкова Г. В.**

3 96 Основы программирования на языке Python [Электронный ресурс]: учеб.-метод. пособие / Г. В. Зыкова, А. С. Попов, Т. Н. Сапуглецева ; науч ред. Г. В. Зыковой. – 2-е изд., стер. – Москва : ФЛИНТА, 2020. – 135 с.

ISBN 978-5-9765-4430-7

Данное учебно-методическое пособие разработано для начального курса изучения языка программирования Python, включенного в последние годы в контрольно-измерительные материалы ЕГЭ по информатике и, соответственно, в школьный курс информатики и ИКТ.

Пособие предназначено для бакалавров направления 44.03.01 Педагогическое образование профиль «Информатика и ИКТ», а также может быть использовано школьниками и студентами среднего профессионального образования.

ISBN 978-5-9765-4430-7

© Зыкова Г.В., 2019

© Издательство «ФЛИНТА», 2020

---

## Содержание

|   |    |
|---|----|
| <b>Введение</b> .....                                   | 6  |
| <b>Предисловие</b> .....                                | 9  |
| <b>1 Типы данных. Преобразование типов данных</b> ..... | 14 |
| 1.1 Основы работы с типами данных .....                 | 14 |
| 1.2 Числа .....   | 15 |
| 1.3 Строки .....  | 17 |
| <b>2 Преобразование типов данных в Python</b> .....     | 20 |
| 2.1 Преобразование числовых типов .....                 | 20 |
| 2.2 Преобразование строк .....                          | 21 |
| <i>Контрольные вопросы</i> .....                        | 25 |
| <i>Задачи для самостоятельной работы</i> .....          | 26 |
| <b>3 Ввод и вывод данных</b> .....                      | 27 |
| <i>Контрольные вопросы</i> .....                        | 33 |
| <i>Задачи для самостоятельной работы</i> .....          | 33 |
| <b>4 Арифметические операции</b> .....                  | 35 |
| 4.1 Сложение .....                                      | 36 |
| 4.2 Унарные арифметические операции .....               | 39 |
| 4.3 Работа с комплексными числами .....                 | 40 |
| 4.4 Битовые операции .....                              | 41 |
| 4.5 Базовые операции над строками .....                 | 42 |
| 4.6 Приоритет операций .....                            | 43 |
| <i>Контрольные вопросы</i> .....                        | 44 |
| <i>Задачи для самостоятельной работы</i> .....          | 45 |
| <b>5 Логические операции</b> .....                      | 46 |
| 5.1 Операторы сравнения .....                           | 46 |
| 5.2 Логические операторы .....                          | 48 |
| 5.3 Таблицы истинности .....                            | 51 |
| <i>Контрольные вопросы</i> .....                        | 52 |
| <i>Задачи для самостоятельной работы</i> .....          | 53 |
| <b>6 Условные операторы</b> .....                       | 54 |
| 6.1 Простой условный оператор .....                     | 54 |
| 6.2 Составной условный оператор .....                   | 54 |
| 6.3 Условная конструкция <i>if</i> .....                | 55 |
| 6.4 Конструкция <i>if...else</i> .....                  | 56 |
| 6.5 Команда <i>elif</i> .....                           | 57 |
| 6.6 Вложенные условные инструкции .....                 | 58 |

---

|  |     |
|--|-----|
| 6.7 Тернарная условная операция .....                                    | 59  |
| <i>Контрольные вопросы</i> .....   | 60  |
| <i>Задачи для самостоятельной работы</i> .....                           | 60  |
| <b>7 Циклы</b> .....   | 62  |
| 7.1 Понятие цикла .....  | 62  |
| 7.2 Цикл «while» .....   | 63  |
| 7.3 Инструкции <i>break</i> и <i>continue</i> в цикле <i>while</i> ..... | 65  |
| 7.4 Вложенные циклы <i>while</i> .....                                   | 66  |
| 7.5 Цикл «for» .....   | 68  |
| 7.6 Оператор следующего прохода <i>continue</i> в цикле <i>for</i> ..... | 70  |
| 7.7 Оператор прерывания цикла <i>break</i> в цикле <i>for</i> .....      | 71  |
| 7.8 Инструкция проверки прерывания <i>else</i> в цикле <i>for</i> .....  | 72  |
| 7.9 Вложенные циклы <i>for</i> .....                                     | 73  |
| <i>Контрольные вопросы</i> .....   | 75  |
| <i>Задачи для самостоятельной работы</i> .....                           | 75  |
| <b>8 Списки</b> .....  | 77  |
| 8.1 Списки и их особенности .....  | 80  |
| 8.2 Операции над последовательностями .....                              | 81  |
| 8.3 Вложенные списки .....   | 83  |
| 8.4 Генераторы списков .....   | 84  |
| 8.5 Срезы .....  | 87  |
| 8.6 Стек и очереди .....   | 88  |
| 8.7 Методы <i>split</i> и <i>join</i> .....                              | 88  |
| <i>Контрольные вопросы</i> .....   | 90  |
| <i>Задачи для самостоятельной работы</i> .....                           | 90  |
| <b>9 Кортежи</b> .....   | 92  |
| 9.1 Операции, поддерживаемые кортежем .....                              | 94  |
| 9.2 Удаление кортежей .....  | 97  |
| 9.3 Список кортежей .....  | 97  |
| 9.4 Сортировка .....   | 97  |
| 9.5 Преобразование кортежа в список и обратно .....                      | 98  |
| 9.6 Преобразование в словарь .....                                       | 99  |
| 9.7 Преобразование кортежей в одну строку .....                          | 99  |
| <i>Контрольные вопросы</i> .....   | 100 |
| <i>Задачи для самостоятельной работы</i> .....                           | 100 |
| <b>10 Словари</b> .....  | 102 |
| 10.1 Создание .....  | 104 |
| 10.2 Добавление элемента .....   | 105 |



---

|  |     |
|--|-----|
| 10.3 Объединение словарей .....                      | 105 |
| 10.4 Удаление элемента .....                         | 106 |
| 10.5 Получение размера .....                         | 106 |
| 10.6 Перебор словаря .....                           | 107 |
| 10.7 Поиск .....                                     | 108 |
| 10.8 Сортировка .....                                | 108 |
| 10.9 Сравнение .....                                 | 109 |
| 10.10 Копирование .....                              | 110 |
| 10.11 Очистка .....                                  | 110 |
| 10.12 Генератор словарей .....                       | 110 |
| 10.13 Конвертация в строку .....                     | 111 |
| 10.14 Вложенные словари .....                        | 112 |
| <i>Контрольные вопросы</i> .....                     | 112 |
| <i>Задачи для самостоятельной работы</i> .....       | 113 |
| <b>11 Множества</b> .....                            | 115 |
| 11.1 Создание множества .....                        | 115 |
| 11.2 Изменение множества .....                       | 116 |
| 11.3 Удаление элементов множества .....              | 117 |
| 11.4 Основные операции с множествами .....           | 118 |
| 11.5 Множество в логическом контексте .....          | 119 |
| 11.6 Метод сору .....                                | 120 |
| 11.7 Frozenset .....                                 | 120 |
| <i>Контрольные вопросы</i> .....                     | 121 |
| <i>Задачи для самостоятельной работы</i> .....       | 121 |
| <b>12 Функции</b> .....                              | 123 |
| 12.1 Создание функций .....                          | 123 |
| 12.2 Работа с функциями .....                        | 124 |
| 12.3 lambda-функции .....                            | 125 |
| 12.4 Аргументы функции в Python .....                | 127 |
| 12.5 Область видимости и глобальные переменные ..... | 129 |
| 12.6 Математические функции в Python .....           | 130 |
| <i>Контрольные вопросы</i> .....                     | 132 |
| <i>Задачи для самостоятельной работы</i> .....       | 132 |
| <b>Библиографический список</b> .....                | 134 |

---

## Введение

Язык программирования Python был задуман как потомок языка программирования ABC. В настоящее время Python – это активно развивающийся высокоуровневый многоцелевой язык программирования. Он поддерживает несколько, наиболее популярных сейчас парадигм программирования, таких как структурное, объектно-ориентированное, функциональное программирование и другие. Популярности языка способствует то, что он соответствует стандартам Американского национального института стандартов и Международной организации по стандартизации.

Python – это многоцелевой язык. Его можно одинаково хорошо использовать для разработки любых программ и их тестирования. Так, например, компания Google широко использует язык Python для своей поисковой системы. Большая часть популярного видеохостинга YouTube была написана на языке Python. Также язык Python применяется в анимационной графике, научных вычислениях и тестировании аппаратного обеспечения. Таким образом, знание Python, одного из популярнейших языков программирования, открывает путь в ведущие IT-компании мира: Google, Яндекс, Mail.Ru, YouTube, Instagram.

Python пригоден для решения разнообразных задач и предлагает те же возможности, что и другие языки программирования. Любой язык, неважно – для программирования или общения, состоит, как минимум, из двух частей: словаря и синтаксиса. Язык Python организован аналогично, предоставляя синтаксис для формирования выражений, образующих исполняемые программы, и словарь – набор функциональности в виде стандартной библиотеки и подключаемых модулей. Но благодаря своей лаконичности он позволяет быстрее овладеть синтаксисом языка, чем другие языки программирования.

Python является языком общего назначения, поэтому может применяться практически в любой области разработки программного

---

обеспечения (standalone, клиент-сервер, Web-приложения) и в любой предметной области. Кроме того, Python легко интегрируется с уже существующими компонентами, что позволяет внедрять Python в уже написанные приложения.

Другая составляющая успеха Python – это его модули расширения, как стандартные, так и специфические. Стандартные модули расширения Python – это отлично спроектированная и неоднократно проверенная функциональность для решения задач, возникающих в каждом проекте по разработке программного обеспечения, обработка строк и текстов, взаимодействие с операционной системой, поддержка Web-приложений. Эти модули также написаны на языке Python, поэтому обладают его важнейшим свойством – кросс-платформенностью, позволяющей безболезненно и быстро переносить проекты с одной операционной системы на другую.

Возможности языка программирования Python:

- Работа с xml/html файлами.
- Работа с http запросами.
- GUI (графический интерфейс).
- Создание веб-сценариев.
- Работа с FTP.
- Работа с изображениями, аудио и видео файлами.
- Робототехника.
- Программирование математических и научных вычислений и многое другое.

Синтаксис языка Python проще Pascal. Большинство технических нюансов при программировании в Python решается с помощью встроенных функций. В Python проще и быстрее написать программу начинающим программистам, если она состоит из одной строки, а не из нескольких. В результате алгоритм решает ту же самую задачу, а времени на написание и отладку тратится меньше. Следовательно, можно решить больше заданий и получить больший опыт в написании программ.



---

Последние несколько лет контрольно-измерительные материалы ЕГЭ включают возможность выполнения заданий, в том числе и с использованием языка программирования Python, что актуализирует его изучение как школьниками, так и будущими учителями информатики и ИКТ.

Данное учебно-методическое пособие разработано с целью организации начального курса изучения языка программирования Python.

---

## Предисловие

Python – высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Синтаксис ядра Python минималистичен. В то же время стандартная библиотека включает большой объём полезных функций.

Python поддерживает структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное программирование. Основные архитектурные черты – динамическая типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений, высокоуровневые структуры данных. Поддерживается разбиение программ на модули, которые, в свою очередь, могут объединяться в пакеты.

Эталонной реализацией Python является интерпретатор CPython, поддерживающий большинство активно используемых платформ. Он распространяется под свободной лицензией Python Software Foundation License, позволяющей применять его без ограничений в любых приложениях, включая проприетарные. Есть реализация интерпретатора для JVM с возможностью компиляции, CLR, LLVM, другие независимые реализации. Проект PyPy использует JIT-компиляцию, которая значительно увеличивает скорость выполнения Python-программ.

Разработка языка Python была начата в конце 1980-х годов сотрудником голландского института CWI Гвидо ван Россумом. Для распределённой ОС Amoeba требовался расширяемый скриптовый язык, и Гвидо начал писать Python на досуге, позаимствовав некоторые наработки для языка ABC (Гвидо участвовал в разработке этого языка, ориентированного на обучение программированию). В феврале 1991 года Гвидо опубликовал исходный текст в группе новостей

---

alt.sources. С самого начала Python проектировался как объектно-ориентированный язык.

Название языка произошло вовсе не от вида пресмыкающихся. Автор назвал язык в честь популярного британского комедийного телешоу 1970-х «Летающий цирк Монти Пайтона». Впрочем, всё равно название языка чаще связывают именно со змеёй, нежели с передачей: пиктограммы файлов в KDE или в Microsoft Windows и даже эмблема на сайте python.org (до выхода версии 2.5) изображают змеиные головы. Важная цель разработчиков Python – создавать его забавным для использования. Это отражено в его названии, которое пришло из Монти Пайтона. Также это отражено в иногда игривом подходе к обучающим программам и справочным материалам, таким как примеры использования.

Наличие дружелюбного, отзывчивого сообщества пользователей считается наряду с дизайнерской интуицией Гвидо одним из факторов успеха Python. Развитие языка происходит согласно чётко регламентированному процессу создания, обсуждения, отбора и реализации документов PEP (англ. Python Enhancement Proposal) – предложений по развитию Python.

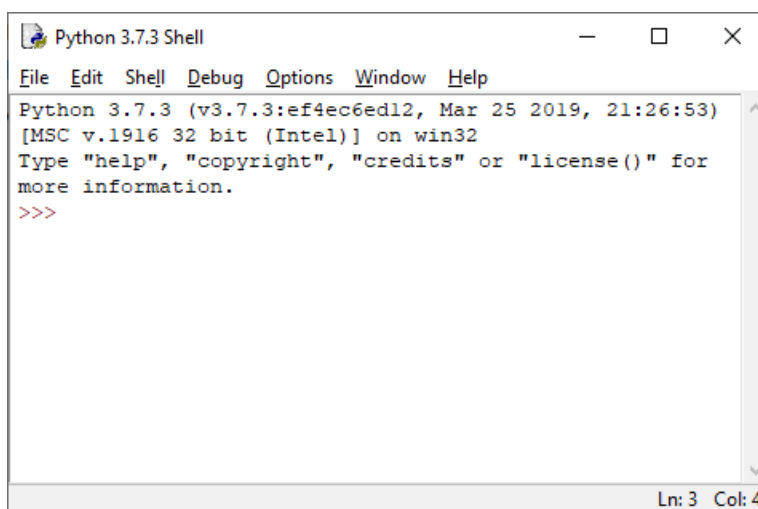
3 декабря 2008 года, после длительного тестирования, вышла первая версия Python 3000 (или Python 3.0, также используется сокращение Py3k). В Python 3000 устранены многие недостатки архитектуры с максимально возможным (но не полным) сохранением совместимости со старыми версиями Python. На сегодня поддерживаются обе ветви развития (Python 3.x и 2.x).

Python портирован и работает почти на всех известных платформах – от КПК до мейнфреймов. Существуют порты под Microsoft Windows, практически все варианты UNIX (включая FreeBSD и Linux), Plan 9, Mac OS и Mac OS X, iPhone OS 2.0 и выше, Palm OS, OS/2, Amiga, HaikuOS, AS/400 и даже OS/390, Windows Mobile, Symbian и Android.

По мере устаревания платформы её поддержка в основной ветви языка прекращается. Например, с серии 2.6 прекращена поддержка Windows 95, Windows 98 и Windows ME. Однако на этих платформах можно использовать предыдущие версии Python – на данный момент сообщество активно поддерживает версии Python начиная от 2.3 (для них выходят исправления).

При этом, в отличие от многих портируемых систем, для всех основных платформ Python имеет поддержку характерных для данной платформы технологий (например, Microsoft COM/DCOM). Более того, существует специальная версия Python для виртуальной машины Java – Jython, что позволяет интерпретатору выполняться на любой системе, поддерживающей Java, при этом классы Java могут непосредственно использоваться из Python и даже быть написанными на Python. Также несколько проектов обеспечивают интеграцию с платформой Microsoft.NET, основные из которых – IronPython и Python.Net.

Для написания программы на языке программирования Python необходимо иметь установленный одноименный пакет на персональном компьютере. При его наличии необходимо запустить программу с названием «IDLE». Таким образом запустится графический интерфейс среды разработки языка Python.



*Рис. 1. Графический интерфейс среды разработки языка Python*

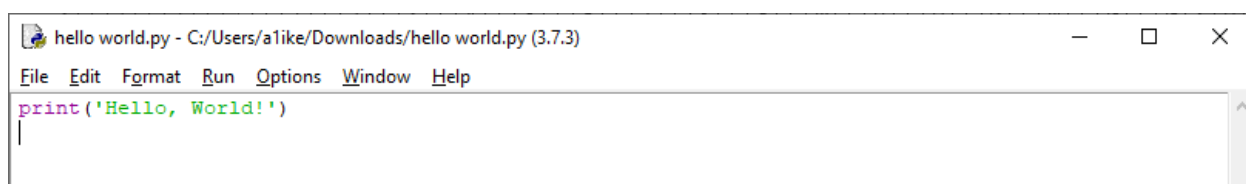
Окно среды разработки представляет собой поле ввода текста, которое занимает большую часть площади окна, а также меню. В поле ввода текста изначально записано несколько строк, в которых указана текущая версия языка, дата её публикации, а также приглашение на ввод.

Выражения, записанные в этом интерфейсе, вычисляются сразу, потому что Python относится к интерпретируемым языкам программирования. То есть, записываемые на нём команды при каждом выполнении построчно переводятся в двоичный код и выполняются сразу после перевода, а среда разработки языка Python по умолчанию работает в интерактивном режиме. В ней все команды запускаются на выполнение сразу после ввода.

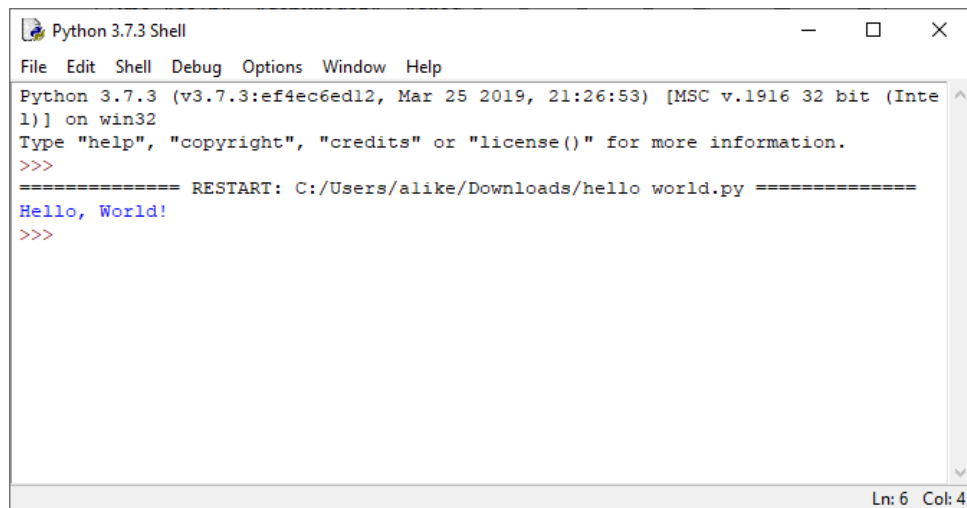
Для того, чтобы получить возможность сохранять написанные команды, в среде разработки необходимо создать новый файл. Для этого, в меню File выбирается команда New File или используется сочетание клавиш Ctrl + N при установленной англоязычной раскладке клавиатуры. После этого, поверх окна среды разработки появится окно нового файла. Файлы модулей, описанных на языке Python, сохраняются с расширением \*.py, сокращённо от названия языка.

При написании кода программы, файл, как правило, сохраняется. Для этого в меню File выбирается команда Save As или используется сочетание клавиш Ctrl + S.

После сохранения можно запустить написанную программу на выполнение. Для этого в меню Run выбирается команда Run Module или используется клавиша F5. В главном окне среды разработки будет выводиться результат выполнения программы.



*Рис. 2. Стандартный текстовый редактор для написания кода в среде Python*



The image shows a screenshot of a 'Python 3.7.3 Shell' window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area contains the following text: 'Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32', 'Type "help", "copyright", "credits" or "license()" for more information.', '>>>', '===== RESTART: C:/Users/alike/Downloads/hello world.py =====', 'Hello, World!', and '>>>'. The status bar at the bottom right indicates 'Ln: 6 Col: 4'.

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/alike/Downloads/hello world.py =====
Hello, World!
>>>
```

*Рис. 3. Выполнение написанной программы через команду Run Module*

---

# 1 ТИПЫ ДАННЫХ. ПРЕОБРАЗОВАНИЕ ТИПОВ ДАННЫХ

В Python, как и в других языках программирования, все данные для удобства делятся на типы. Тип данных определяет значения, которые можно присваивать, и действия, которые можно выполнять.

## 1.1 Основы работы с типами данных

Примером разбиения на типы данных служат разные числовые множества, изучаемые в курсе математики: целые числа ( $0, \pm 1, \pm 2 \dots$ ), иррациональные числа ( $\pi$ ) и т. п.

Как правило, в математических операциях можно комбинировать числа различных типов, например:

$$5 + \pi.$$

При этом можно оставить в качестве ответа полученное уравнение, а также можно округлить  $\pi$  до 3,14 и сложить числа:

$$5 + \pi = 5 + 3,14 = 8,14.$$

Но если попытаться решить уравнение, в котором кроме чисел будут присутствовать другие типы данных, например слова, не получится никакого вменяемого результата. К примеру,

$$\text{имя} + 7.$$

Компьютеры строго разграничивают все типы данных, потому нужно очень внимательно присваивать значения и использовать их в различных операциях.



---

## 1.2 Числа

Любая цифра воспринимается в Python как число. При этом не обязательно объявлять, какой тип данных вы вводите. Python воспринимает любое число, записанное без десятичных знаков, как целое число (например, 133), а любое число, записанное с десятичными знаками, в качестве числа с плавающей точкой (например, 138.0).

### *Целые числа*

Как и в математике, в компьютерном программировании целые числа – это натуральные числа, их противоположные (отрицательные) и 0 (... , -1, 0, 1, ...). Целое число можно отметить как *int*. Как и в других языках, в Python не нужно использовать запятую при написании многозначных чисел (к примеру, тысяча записывается как 1000, а не как 1,000).

Синтаксис вывода целого числа:

```
print(- 25)
```

Результат: -25.

Можно объявить переменную (в данном случае она является символом числа, которое нужно вывести):

```
my_int = -25  
print(my_int)
```

Результат: -25

Целые числа в Python 3 неограниченного размера.

### *Числа с плавающей точкой*

Число с плавающей точкой (или *float*) – это действительное число (то есть оно может быть как рациональным, так и иррациональным). Числа с плавающей точкой могут содержать дробную часть (например, 9.0 или -116.42). То есть Python воспринимает любое число с десятичной точкой как число с плавающей точкой.

Синтаксис вывода числа с плавающей точкой:

```
print(17.3)
```

Результат: 17.3

Можно объявить переменную:

---

```
my_flt = 17.3
print(my_flt)
```

Результат: 17.3

При работе с целыми числами и числами с плавающей точкой важно помнить, что 3 и 3.0 – не одно и то же.

$$3 \neq 3.0,$$

поскольку 3 – целое число, а 3.0 – число с плавающей точкой.

### *Логический тип*

Логические значения в Python представлены двумя величинами – логическими константами *True* (Истина) и *False* (Ложь). Логические значения получаются в результате логических операций и вычисления логических выражений.

Логический тип данных (или *Boolean*) – это тип данных, который принимает одно из двух возможных значений: *True* или *False*. Этот тип присутствует во многих языках программирования и используется для построения алгоритмов.

**Примечание.** Название этого типа данных (*Boolean*) всегда пишется с заглавной буквы, поскольку этот тип назван в честь математика Джорджа Буля, который занимался вопросами математической логики. Значения *True* и *False* также пишутся с большой буквы.

Многие математические операции можно расценивать как истинные или ложные:

$$\begin{aligned} 500 > 100 & \text{ (True);} \\ 1 > 5 & \text{ (False).} \end{aligned}$$

Как и в случае с числами, значения *Boolean* можно определять переменными:

```
my_bool = 5 > 8
```

Теперь можно вывести значение переменной с помощью функции `print()`:

```
print (my_bool)
```

---

Поскольку 5 меньше 8, на экране появится:  
False

### 1.3 Строки

Строка представляет собой последовательность из одного или нескольких символов (букв, цифр и других символов), которые могут быть постоянными или переменными. В Python строки обозначаются одинарными (') или двойными кавычками ("). Для создания строки последовательность символов заключается в кавычки:

```
'This is a string in single quotes.'  
''This is a string in single quotes.''
```

Одинарные и двойные кавычки работают одинаково. Важно только использовать один и тот же тип кавычек в рамках одной программы.

Простая программа «Hello, World» демонстрирует применение строк в программировании (последовательность символов, из которых состоит фраза 'Hello, World!', является строкой).

```
print ('Hello, World!')
```

Строки можно хранить в переменных:

```
hm = 'Hello, World!'
```

Чтобы вывести значение переменной, вводится:

```
print (hm)
```

Результат: Hello, World!

#### *Списки*

Список – это изменяемая, упорядоченная последовательность элементов. Значения, которые находятся в списке, называются элементами. Подобно тому, как строки определяются кавычками, списки определяются квадратными скобками [ ].

Список целых чисел выглядит так:

```
[-3, -2, -1, 0, 1, 2, 3]
```

Список чисел с плавающей точкой имеет такой вид:

```
[3.14, 9.23, 111.11, 312.12, 1.05]
```

---

Список строк:

```
['shark', 'cuttlefish', 'squid', 'mantis shrimp']
```

Определим список `sea_creatures` и осуществим его вывод:

```
sea_creatures = ['shark', 'cuttlefish', 'squid', 'mantis shrimp']  
print (sea_creatures)  
['chark', 'cuttlefish', 'squid', 'mantis shrimp']
```

Списки – очень гибкий тип данных, который позволяет быстро добавить, удалить или изменить данные. В Python существует тип данных, очень похожий на списки, но который нельзя изменять. Такой тип называется кортежем.

### *Кортежи*

Кортеж (*tuple*) позволяет группировать данные. Кортеж – это неизменяемая упорядоченная последовательность элементов.

Кортежи очень похожи на списки, но вместо квадратных скобок они используют круглые. Данные этого типа нельзя изменять.

Кортеж имеет такой вид:

```
('blue coral', 'staghorn coral', 'pillar coral')
```

Кортеж можно хранить в переменной и вывести на экран:

```
coral = ('blue coral', 'staghorn coral', 'pillar coral')  
print (coral)
```

Результат: ('blue coral', 'staghorn coral', 'pillar coral')

### *Словари*

Словарь – это неупорядоченный изменяемый массив данных, состоящий из пар «ключ – значение». Словари обозначаются фигурными скобками { }.

Словари обычно хранят связанные данные. Словарь имеет такой вид:

```
{'name': 'Jake', 'animal': 'dog', 'color': 'yellow', 'location': 'Tree Fort'}
```

Кроме фигурных скобок, в словарях используется двоеточие. Слева от двоеточия пишутся ключи, в данном случае это 'name', 'animal', 'color', 'location'.

---

Справа от двоеточия находятся значения. Значения могут быть представлены любым типом данных. В приведённом примере значениями являются 'Jake', 'dog', 'yellow', 'Tree Fort'.

Создать переменную для словаря, а затем вывести её на экран.

```
jake = {'name': 'Jake', 'animal': 'dog', 'color': 'yellow', 'location': 'Tree Fort'}  
print(jake)
```

Результат: {'color': 'yellow', 'animal': 'dog', 'name': 'Jake', 'location': 'Tree Fort'}.

Чтобы запросить только один из элементов словаря, используются квадратные скобки. Например:

```
print (jake ['color'])
```

Результат: yellow.

---

## 2 ПРЕОБРАЗОВАНИЕ ТИПОВ ДАННЫХ В PYTHON

В программировании часто необходимо конвертировать один тип данных в другой, чтобы получить доступ к другим функциям, например, склеить числовые значения со строками или представить целые числа в виде десятичных.

### 2.1 Преобразование числовых типов

В Python существует два числовых типа данных: целые числа и числа с плавающей точкой. Для преобразования целых чисел в числа с плавающей точкой и наоборот Python предоставляет специальные встроенные методы.

*Преобразование целых чисел в числа с плавающей точкой*

Метод `float()` преобразует целые числа в числа с плавающей точкой. Число указывается в круглых скобках:

```
float(57)
```

В результате преобразует число 57 в 57.0.

Также можно использовать переменные. Например:

```
f = 57
print (float(f))
```

Результат: 57.0

*Преобразование чисел с плавающей точкой в целые числа*

Встроенная функция `int()` предназначена для преобразования чисел с плавающей точкой в целые числа.

Например, число 390.8 преобразуется в 390:

```
int (390.8)
```

Эта функция также может работать с переменными:

```
b = 125.0
c = 390.8
print (int (b))
print (int (c))
```

---

Результат:

125

390.

Чтобы получить целое число, функция `int()` отбрасывает знаки после запятой, не округляя их (потому 390.8 не преобразовывается в 391).

### *Преобразование чисел с помощью деления*

При делении Python 3 может преобразовать целое число в число с плавающей точкой (в Python 2 такой функции нет). К примеру, разделив 5 на 2, получается 2.5.

```
a = 5 / 2
```

Результат: 2.5.

Python не преобразует тип данных во время деления; следовательно, разделив целое число на целое число, в результате получили бы целое число 2.

## **2.2 Преобразование строк**

Существует множество способов преобразования строк.

### *Преобразование чисел в строки*

Чтобы конвертировать число в строку, используется метод `str()`. Число (или переменная) помещается в круглые скобки.

Можно преобразовать целое число, например:

```
str(12)
```

Вывод:

```
'12'
```

Кавычки означают, что теперь 12 является строкой, а не числом.

Особенно полезно преобразовывать числа в строки, используя переменные. К примеру, можно отследить, сколько строк кода в день пишет тот или иной пользователь. Если пользователь пишет больше 50 строк, программа отправит ему поощрительное сообщение.

```
user = 'Michael'  
lines = 50
```



---

```
print ('Congratulations, ' + user + '! You just wrote  
' + lines + ' lines of code.')
```

Запустив этот код, получится ошибка:

`TypeError: Can't convert 'int' object to str implicitly`

Python не может склеивать строки с числами, потому нужно преобразовать значение `lines` в строку.

```
user = 'Michael'  
lines = 50  
print ('Congratulations, ' + user + '! You just wrote  
' + str(lines) + ' lines of code.')
```

Теперь, запустив код:

Congratulations, Michael! You just wrote 50 lines of code.

Метод `str()` может преобразовать в строку и число с плавающей точкой. Необходимо поместить в круглые скобки число или переменную:

```
f = 5524.53  
print (str (421.034))  
print (str (f))
```

Результат:

```
'421.034'  
'5524.53'
```

Выполняется конкатенация (склеивание) строки и преобразованного в строку числа:

```
f = 5524.53  
print ('Michael has ' + str(f) + 'points.')
```

Результат: Michael has 5524.53 points.

*Преобразование строк в числа*

Строки можно преобразовать в числа с помощью методов `int()` и `float()`.

Если в строке нет десятичных знаков, лучше преобразовать её в целое число. Для этого используется `int()`.

---

Можно попробовать расширить предыдущий пример кода, который отслеживает количество написанных строк. Пусть программа отслеживает разницу между написанными строками вчера и сегодня.

```
lines_yesterday = '50'
lines_today = '50'
lines_more = lines_today - lines_yesterday
print (lines_more)
```

Результат: `TypeError: unsupported operand type (s) for -: 'str' and 'str'`.

При запуске возникла ошибка, поскольку Python не может выполнить сложение строк. Для этого необходимо преобразовать строки в числа и снова запустить программу:

```
lines_yesterday = '50'
lines_today = '108'
lines_more = int (lines_today) - int (lines_yesterday)
print (lines_more)
```

Результат: 58.

Значение переменной *lines\_more* – это число, в данном случае это 58.

Также можно преобразовать числа в предыдущем примере в числа с плавающей точкой. Для этого используется метод *float()*.

К примеру, очки начисляются в десятичных значениях.

```
total_points = '5524.53'
new_points = '45.30'
new_total_points = total_points + new_points
print (new_total_points)
```

Результат: 5524.5345.30.

В данном случае оператор `+` склеивает две строки, а не складывает числа. Потому в результате получилось довольно странное значение.

Следует конвертировать эти строки в числа с плавающей точкой, а затем выполнить сложение.

```
total_points = '5524.53'
new_points = '45.30'
```

---

```
new_total_points = float (total_points) + float
(new_points)
print (new_total_points)
```

Результат: 5569.83.

Теперь программа возвращает ожидаемый результат.

Если попробовать преобразовать строку с десятичными значениями в целое число, получим ошибку:

```
f = '54.23'
print (int(f))
```

Результат: ValueError: invalid literal for int () with base 10: '54.23'.

### *Преобразование в кортежи и списки*

Чтобы преобразовать данные в кортеж или список, следует использовать методы *tuple()* и *list()* соответственно.

### *Преобразование списка в кортеж*

Преобразовывая список в кортеж, можно оптимизировать программу. Для преобразования в кортеж используется метод *tuple()*.

```
print (tuple (['pull request', 'open source', 'repository', 'branch']))
```

Результат: ('pull request', 'open source', 'repository', 'branch').

Выведенные на экран данные являются кортежем, а не списком, поскольку они взяты в круглые скобки.

Возможно использование *tuple()* с переменной:

```
sea_creatures = ['shark', 'cuttlefish', 'squid', 'mantis shrimp']
print (tuple (sea_creatures))
```

Результат: ('shark', 'cuttlefish', 'squid', 'mantis shrimp').

В кортеж можно преобразовать любой итерируемый тип, включая строки:

```
print (tuple ('Michael'))
```

Результат: ('M', 'i', 'c', 'h', 'a', 'e', 'l').

Конвертируя в кортеж числовой тип данных, получится ошибка:

```
print (tuple(5000))
```

Результат: TypeError: 'int' object is not iterable.

---

### *Преобразование в списки*

Можно преобразовать кортеж в список, чтобы сделать его изменяемым.

Следует обратить внимание: при этом в методах `list()` и `print()` используется две пары круглых скобок. Одни принадлежат собственно методу, а другие – кортежу.

```
print (list (('blue coral', 'staghorn coral', 'pillar coral')))
```

Результат: ['blue coral', 'staghorn coral', 'pillar coral'].

Если данные, которые вывел метод `print`, заключены в квадратные скобки, значит, кортеж преобразовался в список.

Чтобы избежать путаницы с круглыми скобками, можно создать переменную:

```
colar = ('blue coral', 'staghorn coral', 'pillar coral')  
list (colar)
```

Строки тоже можно преобразовывать в списки:

```
print (list ('Michael'))
```

Результат: ['M', 'i', 'c', 'h', 'a', 'e', 'l'].

### ***Контрольные вопросы***

1. Назовите типы данных в языке Python.
2. Что означает термин «неизменяемый» и какие типы данных языка Python являются неизменяемыми?
3. Что означает термин «последовательность» и какие типы относятся к этой категории?
4. Для преобразования каких чисел предназначена встроенная функция `int()`?
5. Изменяемая упорядоченная последовательность элементов, взятая в квадратные скобки (`[ ]`) в Python – это?
6. В каком случае используется метод `str()`?
7. С помощью каких методов можно преобразовать строки в числа?

---

8. Какой метод используется для преобразования списка в кортеж?

### *Задачи для самостоятельного решения*

1. Дано четырехзначное число. Поменяйте местами наименьшую и наибольшую цифры.

2. Найдите  $n$  пар простых чисел, которые отличаются друг от друга на 2.

3. Вывести все пятизначные числа, которые делятся на 2, у которых средняя цифра нечетная и сумма всех цифр делится на 4.

4. Дана строка, состоящая ровно из двух слов, разделенных пробелом. Переставьте эти слова местами. Результат запишите в строку и выведите получившуюся строку. При решении этой задачи нельзя пользоваться циклами и инструкцией *if*.

5. Дана строка. Если в этом числе буква  $f$  встречается только один раз, выведите её индекс. Если она встречается два и более раз, выведите индекс её первого и последнего появления. Если буква  $f$  в данной строке не встречается, ничего не выводите. При решении этой задачи нельзя использовать метод *count* и циклы.

6. Дана строка. Найдите в этой строке второе вхождение буквы  $f$  и выведите индекс этого вхождения. Если буква  $f$  в данной строке встречается только один раз, выведите число -1, а если не встречается ни разу, выведите число -2. При решении этой задачи нельзя использовать метод *count*.

7. Дана строка. Замените в этой строке все цифры 1 на слово one.

8. Дана строка. Удалите из этой строки все символы @.

9. Дана строка. Получите новую строку, вставив между двумя символами исходной строки символ \*. Выведите полученную строку.

10. Дана строка. Удалите из нее все символы, индексы которых делятся на 3.

---

### 3 ВВОД И ВЫВОД ДАННЫХ

Компьютерные программы нужны для того, чтобы быстро обрабатывать и анализировать некоторые данные, особенно это целесообразно, когда речь идет о больших объемах данных. Предположим, что есть программа, способная обработать некоторые данные. Прежде чем обработать данные, программа должна их каким-то образом получить. Так, например, пользователь может задать данные путём ввода с клавиатуры. После того, как программа обработала данные, она должна определённым способом вернуть результат их обработки пользователю, например, вывести его на экран в текстовой форме. Именно для того, чтобы организовать передачу данных от пользователя программе, и наоборот, используются инструкции ввода и вывода.

Начнём с ввода данных. Для того, чтобы дать возможность пользователю ввести данные, используется функция *input* без параметров. Эта функция возвращает значение, которое пользователь ввёл с клавиатуры в строку.

Рассмотрим синтаксис. Все функции в языке Python записываются в составе инструкций. Для вызова функции записывается её имя, после которого в скобках следуют её параметры. Так как функция *input* не имеет параметров, после её имени должны следовать пустые скобки. Так как программа записывает данные в переменную, то результат работы этой функции присваивается некоторой переменной. Таким образом, для считывания значения переменной *a* с клавиатуры нужно записать инструкцию присваивания переменной *a* значения функции *input* ().

Для вывода данных из оперативной памяти компьютера на экран монитора используется инструкция *print*. После служебного слова *print* в скобках следует список выводимых данных. Как и в любой другой операции, в инструкции вывода могут указываться литералы, переменные и выражения.

---

Рассмотрим пример.

На ввод подать 2 целых числа и вывести на экран их сумму.

```
a = input ()  
b = input ()  
print (a + b)
```

Первое число зададим равным 35.

Второе число зададим равным 42.

Очевидно, что, по нашему замыслу, программа должна была вывести на экран число 77, но вместо этого она вывела 3542. Хотя на самом деле это не названное число, а символьная строка, состоящая из четырёх цифр. Почему так произошло?

Пользователь, задавая данные с клавиатуры, вводит их в текстовой форме. То есть функция *input* возвращает данные типа *str*, по условию задачи, нужны целые числа, то есть данные типа *int*. Для того, чтобы эти данные получить, необходимо воспользоваться функцией преобразования типов.

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

В языке программирования Python есть множество функций и одна из них *print()* – это команда языка Python, которая выводит то, что находится в скобках, на экран:

```
print(1032)
```

Результат: 1032;

```
print(2.34)
```

Результат: 2.34

```
print("Hello")
```

Результат: Hello

В скобках могут быть любые типы данных. Кроме того, количество данных может быть различным:

```
print("a:", 1)
```

Результат: a: 1;

```
one = 1
```

```
two = 2
```



---

```
three = 3
print(one, two, three)
```

Результат: 1 2 3.

Можно передавать в функцию `print()` как непосредственно литералы (в данном случае `"a:"` и `1`), так и переменные, вместо которых будут выведены их значения. Аргументы функции (то, что в скобках) разделяются между собой запятыми. В выводе вместо запятых значения разделены пробелом.

Если в скобках стоит выражение, то сначала оно выполняется, после чего `print()` уже выводит результат данного выражения:

```
print("hello" + " " + "world")
```

Результат: hello world;

```
print(10 - 2.5/2)
```

Результат: 8.75.

В `print()` предусмотрены дополнительные параметры. Например, через параметр `sep` можно указать отличный от пробела разделитель строк:

```
print("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun",
sep="-")
```

Результат: Mon-Tue-Wed-Thu-Fri-Sat-Sun;

```
print(1, 2, 3, sep="//")
```

Результат: 1//2//3

Параметр `end` позволяет указывать, что делать, после вывода строки. По умолчанию происходит переход на новую строку. Однако это действие можно отменить, указав любой другой символ или строку:

```
print(10, end="")
```

Результат: 10.

Обычно, если `end` используется, то не в интерактивном режиме, а в скриптах, когда несколько выводов подряд надо разделить не переходом на новую строку, а, скажем, запятыми. Сам переход на новую строку обозначается комбинацией символов `"\n"`. Если присвоить это значение параметру `end`, то никаких изменений в работе

---

функции `print()` вы не увидите, так как это значение и так присвоено по умолчанию:

```
print(10, end='\n')
```

Результат: 10.

Однако, если надо отступить на одну дополнительную строку после вывода, то можно сделать так:

```
print(10, end='\n\n')
```

Результат: 10.

Рассмотрим еще одну возможность функции `print()` – это использование форматирования строк. На самом деле это никакого отношения к `print()` не имеет, а применяется к строкам. Но обычно используется именно в сочетании с функцией `print()`.

Форматирование может выполняться в так называемом старом стиле или с помощью строкового метода `format`. Старый стиль также называют Си-стилем, так как он схож с тем, как происходит вывод на экран в языке С. Рассмотрим пример:

```
pupil = "Ben"
old = 16
grade = 9.2
print("It's %s, %d. Level: %f" % (pupil, old, grade))
```

Результат: It's Ben, 16. Level: 9.200000.

Здесь вместо трех комбинаций символов `%s`, `%d`, `%f` подставляются значения переменных `pupil`, `old`, `grade`. Буквы `s`, `d`, `f` обозначают типы данных – строку, целое число, вещественное число. Если бы требовалось подставить три строки, то во всех случаях использовалось бы сочетание `%s`.

Хотя в качестве значения переменной `grade` было указано число 9.2, на экран оно вывелось с дополнительными нулями. Однако можно указать, сколько требуется знаков после запятой, записав перед буквой `f` точку с желаемым числом знаков в дробной части:

```
print("It's %s, %d. Level: %.1f" % (pupil, old, grade))
```

Результат: It's Ben, 16. Level: 9.2.

Теперь рассмотрим метод `format()`:

---

```
print("This is a {0}. It's {1}.".format("ball", "red"))
```

Результат: This is a ball. It's red.

```
print("This is a {0}. It's {1}.".format("cat", "white"))
```

Результат: This is a cat. It's white.

```
print("This is a {0}. It's {1} {2}.".format(1, "a", "number"))
```

Результат: This is a 1. It's a number.

В строке в фигурных скобках указаны номера данных, которые будут сюда подставлены. Далее к строке применяется метод *format()*. В его скобках указываются сами данные (можно использовать переменные). На нулевое место подставится первый аргумент метода *format()*, на место с номером 1 – второй и т. д.

При обработке вывода данных часто бывает полезным использование форматированного вывода. В этом случае можно выделить некоторое количество знаковых позиций для вывода каждого значения. Для этого используется функция «Формат», которая формирует символьную строку заданного формата. Для этого присвоим переменным *a*, *b* и *c* соответственно значения 15, 141 и 3. Дальше записываем инструкцию *print*, в которой, в кавычках, сначала запишем строку, описывающую формат вывода. Формат вывода каждого отдельного значения указывается в отдельных фигурных скобках. Он начинается с двоеточия. Дальше для целых чисел следует единственное число – количество выделяемых знаковых позиций и английская буква *d*. Выделим по пять знаковых позиций для вывода каждого числа. После строки формата ставится точка и записывается служебное слово *format*, после которого в скобках указываются выводимые значения. Укажем значения переменных *a*, *b* и *c*. Программа вывела отступы перед значениями, так как незанятые знаковые позиции заполняются пробелами. Если же знаковых позиций не хватает, они дополняются автоматически.

```
a = 15
```

```
b = 141
```

```
c = 3
```

---

```
print ('{:5d}{:5d}{:5d}'.format (a, b, c))
```

Результат: 15 141 3.

Рассмотрим расчёт и вывод частного двух чисел, причём обязательно целых. Так как числа обязательно будут целыми, то вводимые значения нужно преобразовать в вещественный тип *float*.

```
a = int (input ())  
b = int (input ())  
print (a, '/', b, '=', a / b, sep = '')
```

Введём первое число, равное 0.01, а второе – 5000.

В результате программа вывела вместо последнего числа сообщение: 2e-06.

Это называется экспоненциальной формой представления числа. Она означает, что результат равен произведению 2 и  $10^{-6}$ .

Для вывода вещественных значений также можно использовать форматированный вывод. Применим форматированный вывод для последнего числа. В качестве строки формата, в кавычках, между фигурными скобками укажем двоеточие, после которого будет следовать два целых числа, разделённых точкой – общее количество выделяемых знаковых позиций и количество выводимых знаков после запятой. Укажем 10 знаковых позиций и 7 знаков после запятой. Далее для вещественных чисел следует английская буква *f*. После кавычек поставим точку и напомним служебное слово *format*, после которого в скобках укажем выводимое значение.

```
print ('Программа, вычисляющая частное двух чисел. Введите  
два числа.')
```

```
a = int (input ())  
b = int (input ())  
print (a, '/', b, '=', '{:10.7f}'.format (a / b))
```

Запустим модуль на выполнение. Снова зададим числа 0.01 и 5000. На этот раз программа вывела ответ не в экспоненциальной, а в обычной форме.

---

## ***Контрольные вопросы***

1. С помощью какой команды производится ввод данных на языке программирования Python?
2. С помощью какой команды производится вывод данных на языке программирования Python?
3. Можно ли введенные данные сразу преобразовывать в другой тип?
4. В переменную “a” записали числовое значение 42. Что выведет на экран следующая команда: `print(a)`?
5. В переменные “a” и “b” записали числовые значения 2 и 5 соответственно. Что выведет на экран следующая команда: `print(a + b)`?
6. В переменные “a” и “b” записали текстовые значения “2” и “5” соответственно. Что выведет на экран следующая команда: `print(a + b)`?
7. Как при вводе данных преобразовывать текст в числовые значения?
8. Будет ли работать программа, если внутри команды `print` объявить команду `input`?
9. Можно ли в команде `print` перечислять данных для вывода?
10. Правда ли, что строки можно складывать?

## ***Задачи для самостоятельного решения***

1. Напишите программу, которая запрашивает имя пользователя и затем выводит приветствие с введенным именем.
2. Напишите программу, в которой запрашивается несколько раз ввод данных, а затем эти данные суммируются, выводясь на экран.
3. Напишите программу, которая запрашивает 2 ввода чисел и выводит их сумму.

- 
4. Напишите программу, которая вычисляет дискриминант квадратного уравнения (значения вводятся пользователем).
  5. Напишите программу, которая вычисляет периметр треугольника (значения вводятся пользователем).
  6. Напишите программу, которая считает квадрат суммы двух введенных чисел (значения вводятся пользователем).
  7. Напишите программу, которая возводит в куб любое введенное число (значение вводится пользователем).
  8. Напишите программу, которая вычисляет значение выражения:  $4 \cdot 100 - 54$ .
  9. Напишите программу, которая запрашивает имя пользователя и затем выводится прощание с введенным именем.
  10. Напишите программу, которая вычисляет площадь прямоугольника (значения вводятся пользователем).

## 4 АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ

Числа в Python делятся на целые; действительные (с плавающей точкой размером в 64 бита); комплексные (с плавающей точкой размером в 128 бит).

Если в качестве *операндов* (операнды – это действия операторов, производимые над данными) арифметического выражения используются только целые числа, то результат будет целое число.

Исключение: операция деления, результатом которой является вещественное число.

При совместном использовании целочисленных и вещественных переменных результат будет вещественным.

Оператор – это символ или функция, которая выполняет то или иное действие над данными.

Таблица 1

### *Математические операторы*

| Оператор       | Действие   |
|----------------|--|
| 1              | 2  |
| $x+y$          | сумма $x$ и $y$ ( $6 + 3 = 9$ )  |
| $x-y$          | разность $x$ и $y$ ( $57 - 2 = 55$ )   |
| $x*y$          | произведение $x$ и $y$ ( $3 * 3 = 9$ )   |
| $x/y$          | $x$ на $y$ : $4 / 1.6 = 2.5$ , $10 / 5 = 2.0$ , результат всегда типа float                                      |
| $x//y$         | $x$ на $y$ нацело: $11 // 4 = 2$ , $11.8 // 4 = 2.0$ , результат целый, только если оба аргумента целые $x \% y$ |
| $x ** y$       | возведение $x$ в степень $y$ : ( $2 ** 3 = 8$ )  |
| $round(x)$     | округление ( $12,3$ )=12   |
| $round(x,n)$   | округляет число $x$ до $n$ знаков после запятой  |
| $pow(x, y)$    | полный аналог записи $x ** y$  |
| $divmod(x, y)$ | выдаёт два числа: частное и остаток, обращаться следует так $q, r = divmod(x, y)$                                |
| $<$            | Меньше. Определяет, верно ли, что $x$ меньше $y$ . Все операторы сравнения возвращают True или False             |
| $>$            | Больше. Определяет, верно ли, что $x$ больше $y$   |



| 1                  | 2  |
|--------------------|--|
| <code>&lt;=</code> | Меньше или равно. Определяет, верно ли, что $x$ меньше или равно $y$ |
| <code>&gt;=</code> | Больше или равно. Определяет, верно ли, что $x$ больше или равно $y$ |
| <code>==</code>    | Равно. Проверяет, одинаковы ли объекты                               |
| <code>!=</code>    | Не равно. Проверяет, верно ли, что объекты не равны                  |

Операции сложения, вычитания, умножения и возведения в степень выдают ответ типа `int` только тогда, когда оба аргумента целые.

Если один из аргументов действительный, а другой целый, то выдают ответ типа `float`.

Если хотя бы один аргумент комплексный, то выводит ответ типа `complex`.

Операция возведения в степень выдает комплексный результат при возведении отрицательных чисел в степень кроме целой и нечетной степени.

## 4.1 Сложение

Складывать можно сами числа:

```
print(4+2)
```

Результат: 6.

Либо можно складывать переменные, но они должны заранее быть проинициализированы:

```
a = 4
```

```
b = 2
```

```
print(a + b)
```

Результат: 6.

Результат операции сложения может быть присвоен другой переменной:

---

```
a = 2
b = 4
c = a + b
print(c)
```

Результат: 6

Либо может быть присвоен ей самой, тогда можно использовать полную или сокращенную запись. Полная выглядит так:

```
a = 2
b = 4
a = a + b
print(a)
```

Результат: 6.

Сокращенная запись:

```
a = 2
b = 4
a += b
print(a)
```

Результат: 6.

Числа с плавающей точкой:

```
e = 5.5
f = 2.5
print(e + f)
```

Результат: 8.0.

В результате сложения чисел с плавающей точкой получается число с плавающей точкой, потому Python выводит 8.0, а не 8.

Таблица 2

### *Арифметические действия*

| Арифметическое действие | Сокращенная запись       | Полная запись                          |
|-------------------------|--------------------------|--|
| 1                       | 2                        | 3                                      |
| Вычитание               | <pre>print (6-2) 4</pre> | <pre>a = 5 b = 7 print(a - b) -2</pre> |

Окончание таблицы 2

| 1                                   | 2                                  | 3  |
|-------------------------------------|------------------------------------|--|
| Умножение                           | <code>print (6 * 8)</code><br>48   | <code>a = 5</code><br><code>a *= 10</code><br><code>print (a)</code><br>50       |
| Деление                             | <code>print (12 / 3)</code><br>4.0 | <code>a = 7</code><br><code>b = 4</code><br><code>print (a / b)</code><br>1.75   |
| Получение целой части<br>от деления | <code>print (9 // 3)</code><br>3   | <code>a = 7</code><br><code>b = 4</code><br><code>print (a // b)</code><br>1     |
| Получение остатка<br>от деления     | <code>print (9 % 5)</code><br>4    | <code>a = 7</code><br><code>b = 4</code><br><code>print (a % b)</code><br>3      |
| Возведение в степень                | <code>print (5 ** 4)</code><br>625 | <code>a = 10</code><br><code>b = 10</code><br><code>print (a ** b)</code><br>100 |

### *Операторы присваивания*

В Python есть составные операторы присваивания для каждой математической операции.

Операторы присваивания позволяют постепенно увеличить или уменьшить значение, а также автоматизировать некоторые вычисления.

Таблица 3

### *Операторы присваивания*

| Оператор            | Действие  | Пример   |
|---------------------|---|--|
| 1                   | 2   | 3  |
| <code>y += x</code> | Сложение и присваивание.<br>Составной оператор += выполнил сложение, а затем присвоил переменной y значение, полученное в результате сложения | <code>c = 5</code><br><code>a = 2</code><br><code>c += a</code><br>7 |

| 1          | 2  | 3                                     |
|------------|--|---------------------------------------|
| $y -= x$   | Вычитание и присваивание.<br>Отнимает значение правого операнда от левого и присваивает результат левому операнду            | $c = 5$<br>$a = 2$<br>$c -= a$<br>3   |
| $y *= x$   | Умножение и присваивание.<br>Умножает правый операнд на левый операнд и присваивает результат левому операнду                | $c = 5$<br>$a = 2$<br>$c *= a$<br>10  |
| $y /= x$   | Деление и присваивание.<br>Делит левый операнд на правый операнд и присваивает результат левому операнду                     | $c = 10$<br>$a = 2$<br>$c /= a$<br>5  |
| $y //= x$  | Деление <i>floor</i> и присваивание.<br>Выполняет деление операторов с округлением и присваивает значение левому операнду    | $c = 11$<br>$a = 2$<br>$c //= a$<br>5 |
| $y **= x$  | возведение в степень и присваивание.<br>Выполняет вычисление экспоненты от операторов и присваивает значение левому операнду | $c = 3$<br>$a = 2$<br>$c **= a$<br>9  |
| $y \% = x$ | Вывод остатка и присваивание.<br>Принимает модуль с помощью двух операндов и присваивает результат левому операнду           | $c = 5$<br>$a = 2$<br>$c \% = a$<br>1 |

## 4.2 Унарные арифметические операции

Унарное математическое выражение состоит только из одного компонента или элемента. В Python «плюс» и «минус» вместе со значением могут быть использованы в качестве одного элемента, это позволяет показать тождественность значения (+) или изменить его знак (-).

Тождественность используется нечасто. Плюс можно использовать с положительными числами:

```
i = 3.3
print(+i)
```

---

Результат: 3.3.

Если используется плюс с отрицательным числом, он также вернёт тождественное (в этом случае – отрицательное) число.

```
j = -19  
print(+j)
```

Результат: -19.

Минус позволяет изменить знак. Если вы добавите минус к положительному значению, в результате будет отображено отрицательное значение:

```
i = 3.3  
print(-i)
```

Результат: -3.3.

Если добавить минус к отрицательному значению, в результате получится положительное число:

```
j = -19  
print(-j)
```

Результат: 19.

### 4.3 Работа с комплексными числами

При создании комплексного числа используют функцию *complex(a, b)*, у которой первый аргумент передает действительную часть, а второй – мнимую часть. Либо может записать число в виде  $a + bj$ .

Создание комплексного числа:

```
z = 1 + 2j  
print(z)
```

Результат: (1+2j).

```
x = complex(3, 2)  
print(x)
```

Результат: (3+2j).

**Комплексные числа**

| Арифметическое действие                  | Запись в Python  |
|--|--|
| Сложение                                 | <code>print(x + z)</code><br><code>(4+4j)</code>   |
| Вычитание                                | <code>print(x - z)</code><br><code>(2+0j)</code>   |
| Умножение                                | <code>print(x * z)</code><br><code>(-1+8j)</code>  |
| Деление                                  | <code>print(x / z)</code><br><code>(1.4-0.8j)</code>   |
| Возведение в степень                     | <code>print(x ** z)</code><br><code>(-1.1122722036363393-</code><br><code>0.012635185355335208j)</code><br><code>print(x ** 3)</code><br><code>(-9+46j)</code> |
| Извлечение действительной и мнимой части | <code>x = 3 + 2j</code><br><code>print(x.real)</code><br><code>3.0</code><br><code>print(x.imag)</code><br><code>2.0</code>                                    |
| Получение комплексно сопряженного числа  | <code>x = 3 + 2j</code><br><code>print(x.conjugate())</code><br><code>(3-2j)</code>  |

**4.4 Битовые операции**

Над числами *int* в Python можно выполнять и *битовые операции*. Побитовые операторы предназначены для работы с данными в битовом (двоичном) формате.

Предположим, что у нас есть два числа  $a = 60$  и  $b = 13$ . В двоичном формате они будут иметь следующий вид:

$$a = 00111100$$

$$b = 00001101$$

**Битовые операции**

| Операция     | Значение  | Выполнение  |
|--------------|---|---|
| $a \& b$     | Побитовое «и»<br>Бинарный «И» оператор копирует бит в результат только если бит присутствует в обоих операндах  | $a \& b$<br>12<br>в двоичном формате<br>0000 1100     |
| $a   b$      | Бинарный «ИЛИ» оператор копирует бит, если тот присутствует в хотя бы в одном операнде  | $a   b$<br>61<br>в двоичном формате<br>0011 1101      |
| $a \wedge b$ | Побитовое исключающее «или»<br>Бинарный «ИЛИ» оператор копирует бит, если бит присутствует в одном из операндов, но не в обоих сразу                  | $a \wedge b$<br>49<br>в двоичном формате<br>0011 0001 |
| $a \ll b$    | Сдвиг влево<br>Побитовый сдвиг влево. Значение левого операнда «сдвигается» влево на количество бит, указанных в правом операнде                      | $a \ll 2$<br>240<br>в двоичном формате<br>1111 0000   |
| $a \gg b$    | Сдвиг вправо<br>Побитовый сдвиг вправо. Значение левого операнда «сдвигается» вправо на количество бит, указанных в правом операнде                   | $a \gg 2$<br>15<br>в двоичном формате<br>0000 1111    |
| $\sim a$     | Инверсия битов<br>Является унарным (то есть ему нужен только один операнд) меняет биты на обратные, там, где была единица, становится ноль и наоборот | $\sim a$<br>195<br>в двоичном формате<br>1100 0011    |

**4.5 Базовые операции над строками**

Арифметические операции. Для строк, подобно числам, определены операторы сложения «+» и умножения «\*».

---

В результате сложения содержимое двух строк записывается подряд в новую строку, например:

```
S1 = 'Py'
S2 = 'thon'
S3 = S1 + S2
print (S3)
```

Результат: 'Python'.

Можно складывать несколько строк подряд.

Умножение определено для строки и целого положительного числа, в итоге получается новая строка, которая повторяет исходную столько раз, какое было значение числа (возьмём строку S3 из прошлого примера):

```
print(S3 * 4)
```

Результат: 'PythonPythonPythonPython'.

```
print(2 * S3)
```

Результат: 'PythonPython'.

## 4.6 Приоритет операций

В Python операции выполняются в порядке их приоритета. Например:

$$u = 20 + 20 * 2$$

Сначала выполняется умножение ( $20 * 2 = 40$ ), а затем сложение ( $20 + 40$ ). Потому результат будет такой:

```
print(u)
```

Результат: 60.

Чтобы сначала выполнить операцию сложения, а затем умножить полученный результат на 5, нужно взять сложение в скобки:

```
u = (10 + 10) * 5
print(u)
```

Результат: 100.



**Приоритет операторов**

| Оператор             | Значение оператора   |
|----------------------|--|
| <, <=, >, >=, !=, == | Сравнения  |
|                      | Побитовое «ИЛИ»  |
| ^                    | Побитовое «ИСКЛЮЧИТЕЛЬНО ИЛИ»                                  |
| &                    | Побитовое «И»  |
| <<, >>               | Сдвиги   |
| +, -                 | Сложение и вычитание   |
| *, /, //, %          | Умножение, деление, целочисленное деление и остаток от деления |
| +x, -x               | Положительное, отрицательное                                   |
| ~x                   | Побитовое НЕ   |
| **                   | Возведение в степень   |

**Контрольные вопросы**

1. Какие арифметические операции реализованы в Python?
2. Чем отличаются унарные операции от бинарных? Приведите примеры.
3. Перечислите типы ответов при выполнении арифметических операций. От чего они зависят?
4. Приведите примеры выполнения сложения и вычитания.
5. Приведите примеры выполнения округления, возведения в степень.
6. Как получить целую часть и остаток от деления?
7. Как получить комплексное число? Что такое приоритет операций?
8. Какие арифметические операции возможны с комплексными числами?
9. Что такое битовые операции?
10. Какие арифметические операции над строками возможны?

---

### *Задачи для самостоятельного решения*

1. Дано целое число, большее 999. Используя одну операцию деления нацело и одну операцию взятия остатка от деления, найти цифру, соответствующую разряду сотен в записи этого числа.
2. Вводится натуральное число (целое больше нуля). Необходимо найти сумму и произведение цифр, из которых состоит это число.
3. Дана масса  $M$  в килограммах. Используя операцию деления нацело, найти количество полных тонн в ней (1 тонна = 1000 кг).
4. Дано двузначное число. Вывести число, полученное при перестановке цифр исходного числа.
5. Напишите программу (необходимые данные вводятся с клавиатуры) для вычисления всех трёх сторон прямоугольного треугольника, если даны один из острых углов и площадь.
6. Составьте арифметическое выражение и вычислите  $n$ -е чётное число (первым считается 2, вторым 4 и т. д.).
7. Вы стоите на краю дороги и от вас до ближайшего фонарного столба  $x$  метров. Расстояние между столбами  $y$  метров. На каком расстоянии от вас находится  $n$ -й столб?
8.  $x$  - вещественное число. Запишите выражение, позволяющее выделить его дробную часть.
9. Старинными русскими денежными единицами являются: 1 рубль = 100 копеек, 1 гривна = 10 копеек, 1 алтын = 3 копейки, 1 полушка = 0,25 копейки. Имеется  $A$  копеек. Разложите имеющуюся сумму в копейках на сумму из  $x$  рублей +  $y$  гривен +  $z$  алтынов +  $v$  полушек.
10. Даны два предмета, первый из них стоит  $A$  рублей и  $B$  копеек, второй стоит  $C$  рублей  $D$  копеек. Сколько рублей и копеек они стоят в сумме?
11. Дано действительное положительное число  $a$  и целое неотрицательное число  $n$ . Вычислите  $a^n$ .
12. Для натуральных чисел  $A$  и  $B$  требуется вычислить значение  $A$  в степени  $B$ .

---

## 5 ЛОГИЧЕСКИЕ ОПЕРАЦИИ

*Логический тип данных (Boolean)* служит для хранения значений, которые обладают одним из двух возможных состояний: истина (True) или ложь (False). Логический тип данных нужен, чтобы программировать альтернативные действия.

### 5.1 Операторы сравнения

В программировании операторы сравнения используются при оценке и сравнении значений для последующего сведения их к одному логическому значению (True или False).

Операторы сравнения Python 3 представлены в таблице 7.

Таблица 7

*Операторы сравнения Python 3*

| Оператор | Значение   |
|----------|--|
| ==       | Проверяет равенство между компонентами; условие истинно, если компоненты равны                     |
| !=       | Проверяет равенство между компонентами; условие истинно, если компоненты НЕ равны                  |
| <        | Оценивает значение левого компонента; условие истинно, если он меньше, чем правый                  |
| >        | Оценивает значение левого компонента; условие истинно, если он больше, чем правый                  |
| <=       | Оценивает значение левого компонента; условие истинно, если он меньше или равен правому компоненту |
| >=       | Оценивает значение левого компонента; условие истинно, если он больше или равен правому компоненту |

---

Рассмотрим пример работы. Пусть дана пара переменных. Произведём сравнение значений переменных с помощью вышеперечисленных операторов.

```
x = 5
y = 8
print("x == y:", x == y)
print("x != y:", x != y)
print("x < y:", x < y)
print("x > y:", x > y)
print("x <= y:", x <= y)
print("x >= y:", x >= y)
```

Результат:

```
x == y: False
x != y: True
x < y: True
x > y: False
x <= y: True
x >= y: False.
```

Python оценивает соотношения между значениями переменных так:

- 5 равно 8? Ложь
- 5 не равно 8? Истина
- 5 меньше 8? Истина
- 5 больше 8? Ложь
- 5 меньше или равно 8? Истина
- 5 больше или равно 8? Ложь.

Также операторы сравнения можно применять к числам с плавающей точкой и строкам.

**Примечание.** Строки чувствительны к регистру; чтобы отключить такое поведение, нужно использовать специальный метод.

Сравним две строки:

```
Hello = "Hello"
hello = "hello"
print("Hello == hello: ", Hello == hello)
```

---

Результат: `Hello == hello: False`.

Строки `Hello` и `hello` содержат одинаковый набор символов, однако они не равны, поскольку одна из них содержит символы верхнего регистра. Добавим ещё одну переменную, которая также будет содержать символы верхнего регистра, и сравним.

```
Hello = "Hello"
hello = "hello"
Hello_there = "Hello"
print("Hello == hello: ", Hello == hello)
print("Hello == Hello_there", Hello == Hello_there)
```

Результат:

`Hello == hello: False`

`Hello == Hello_there: True`.

Также для сравнения строк можно использовать операторы `>` и `<`. Python выполнит лексикографическое сравнение строк на основе значений символов ASCII.

Операторы сравнения можно применять к логическим значениям `True` и `False`:

```
t = True
f = False
print("t != f: ", t != f)
```

Результат: `t != f: True`

Важно помнить отличие операторов `=` и `==`.

`x = y` # Оператор присваивания – устанавливает равенство между `x` и `y` (то есть присваивает `x` значение `y`).

`x == y` # Оператор сравнения – проверяет равенство между `x` и `y` и оценивает выражение как истинное или ложное.

## 5.2 Логические операторы

Для сравнения значений используются три логических оператора, которые сводят результат к логическому значению `True` или `False` (табл. 8).

*Логические операторы*

| Оператор | Значение  |
|----------|---|
| And      | Оператор «и»: выражение истинно, если оба его компонента истинны                |
| Or       | Оператор «или»: выражение истинно, если хотя бы один из его компонентов истинен |
| Not      | Оператор «не»: изменяет логическое значение компонента на противоположное       |

Логические операторы обычно используются для оценки двух или более выражений.

Например, необходимо написать программу, которая проверит:

- сдал ли студент экзамен
- **и** зарегистрирован ли он.

Если оба значения истинны, студент будет переведён на следующий курс.

Другой пример: программа с логическими операторами может проверять активность пользователя в онлайн-магазине:

- использовал ли он кредит магазина
- **или** заказывал ли он товары в течение последних 6 месяцев.

Рассмотрим варианты:

| Пример   | Объяснение   |
|--|--|
| <code>print((9 &gt; 7) and (2 &lt; 4))</code><br><code>True</code> | Оба выражения истинны, потому оператор <code>and</code> возвращает <code>True</code> .   |
| <code>print((8 == 8) or (6 != 6))</code><br><code>True</code>      | Истинно только значение <code>8 == 8</code> . Поскольку хотя бы одно из предложенных условий истинно, оператор <code>or</code> возвращает <code>True</code> . Оператор <code>and</code> в таком случае выдал бы <code>False</code> . |

| Пример   | Объяснение  |
|--|---|
| <code>print(not(3 &lt;= 1))</code><br><code>False</code>                     | Выражение <code>3 &lt;= 1</code> ложно. Оператор <code>not</code> изменяет полученное логическое значение на противоположное: <code>not False = True</code> . |
| <code>print((-0.2 &gt; 1.4) and (0.8 &lt; 3.1))</code><br><code>False</code> | Одно из выражений ложно, <code>and</code> вернёт <code>False</code> .   |
| <code>print((7.5 == 8.9) or (9.2 != 9.2))</code><br><code>False</code>       | Оба выражения ложны, оператор <code>or</code> выдаст <code>False</code> .   |
| <code>print(not(-5.7 &lt;= 0.3))</code><br><code>True</code>                 | Выражение истинно, оператор <code>not</code> вернёт <code>False</code> ( <code>not True = False</code> ).   |

**Примечание.** Логические операторы можно объединять в составные выражения:

```
not((-0.2 > 1.4) and ((0.8 < 3.1) or (0.1 == 0.1)))
```

Выражение `(0.8 < 3.1) or (0.1 == 0.1)` истинно, поскольку оба математических выражения, из которых оно состоит, истинны. Оператор `or` вернёт `True`.

Полученное значение `True` становится компонентом следующего выражения: `(-0.2 > 1.4) and (True)`. Оператор `and` выдаст `False`, потому что выражение `-0.2 > 1.4` ложно. `(False) and (True) = False`.

Далее оператор `not` заменит полученное значение `False` на обратное ему логическое значение: `not(False) = True`. Значит, результат будет таким: `True`.

---

### 5.3 Таблицы истинности

Ниже представлены таблицы истинности для оператора сравнения `==` и всех логических операторов.

Таблица 9

*Таблица истинности оператора `==`*

| <b>x</b> | <b>==</b> | <b>y</b> | <b>Результат</b> |
|----------|-----------|----------|------------------|
| True     | ==        | True     | True             |
| True     | ==        | False    | False            |
| False    | ==        | True     | False            |
| False    | ==        | False    | True             |

Таблица 10

*Таблица истинности оператора `AND`*

| <b>x</b> | <b>and</b> | <b>y</b> | <b>Результат</b> |
|----------|------------|----------|------------------|
| True     | and        | True     | True             |
| True     | and        | False    | False            |
| False    | and        | True     | False            |
| False    | and        | False    | False            |



**Таблица истинности оператора OR**

| <b>x</b> | <b>or</b> | <b>y</b> | <b>Результат</b> |
|----------|-----------|----------|------------------|
| True     | or        | True     | True             |
| True     | or        | False    | True             |
| False    | or        | True     | True             |
| False    | or        | False    | False            |

**Таблица истинности оператора NOT**

| <b>Not</b> | <b>x</b> | <b>Результат</b> |
|------------|----------|------------------|
| Not        | True     | False            |
| Not        | False    | True             |

**Контрольные вопросы**

1. Что такое логический тип данных?
2. Какие операторы существуют в языке программирования Python?
3. Как проверяется равенство между компонентами в операторе сравнения?
4. Какие три логических оператора используют в сравнение?
5. Что такое таблица истинности?
6. Что такое математическая логика?
7. Что такое выражение?
8. Перечислите операторов в логических операциях.

- 
9. Приведите примеры сравнения чисел с плавающей точкой.
  10. Приведите примеры сравнения строк. Важен ли регистр?

### *Задачи для самостоятельного решения*

1. Дано два числа. Вывести на экран наибольшее из чисел.
2. Пользователь вводит два числа с клавиатуры. Вывести на экран Yes, если они отличаются друг от друга на 135, иначе вывести на экран No.
3. Дано число. Если оно больше 100 или меньше -100, то вывести на экран символ —, иначе вывести на экран символ +.
4. Пользователь вводит номер месяца (от 1 до 12). Вывести название сезона года на экран (зима, весна, лето, осень).
5. Пользователь вводит три числа. Если все числа больше 10, то вывести на экран Yes, иначе No.
6. Даны три числа. Найти количество положительных чисел среди них.
7. Пользователь вводит количество месяцев и лет. Вывести на экран количество дней за это время. Считать, что в каждом месяце 30 дней.
8. Введите 5 различных значений с клавиатуры и найдите среднее арифметическое только положительных. Если таких нет, вывести No.
9. Введите с клавиатуры 10 элементов в интервале  $[0,10]$  и подсчитайте отдельно среднее значение всех элементов, которые больше 5, и среднее значение всех элементов, которые меньше либо равны 5.
10. Ввести четыре целых числа, найти наибольшее из них.

---

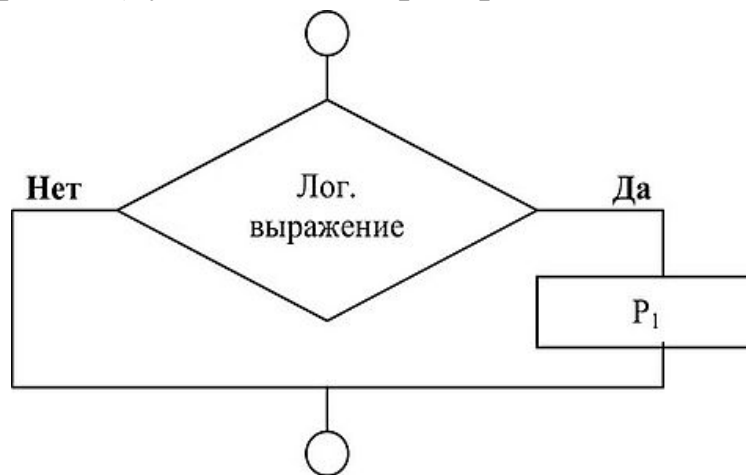
## 6 УСЛОВНЫЕ ОПЕРАТОРЫ

Условные операторы – важная часть любого языка программирования. Они позволяют выполнять команды (или наборы команд) только при наличии определённых условий. Условные операторы отвечают за изменение поведения программы в зависимости от входных параметров, определённых в проверке. Такие операторы иногда называют операторами ветвления.

Условные операторы состоят их заголовка и тела. Заголовок - это сама конструкция. Тело - это та часть, которая написана после двоеточия.

### 6.1 Простой условный оператор

Если необходимо выполнить некоторое действие только при истинности проверяемого условия, то в таком случае применяется сокращенный (простой) условный оператор.



*Рис. 4. Блок-схема сокращенного условного оператора*

### 6.2 Составной условный оператор

Если при некотором условии надо выполнить определенную последовательность операторов, то их объединяют в один составной оператор.

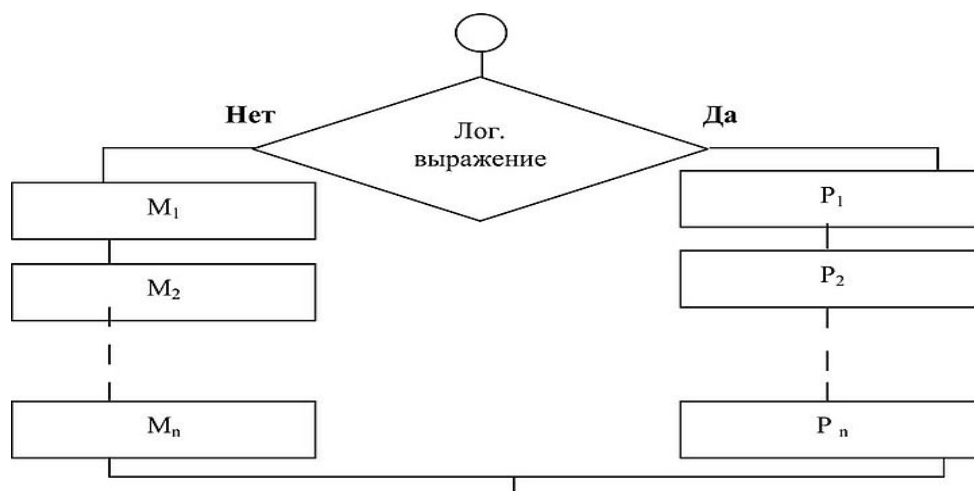


Рис. 5. Блок-схема составного условного оператора

В Python предполагается, что любое ненулевое и непустое значение равняется истине (True), в то время как ноль или пустой объект равняется лжи (False).

Существуют следующие условные конструкции:

1. *if*;
2. *if / elif / else*;
3. вложенные *if* конструкции.

### 6.3 Условная конструкция *if*

Конструкция *if* в Python работает по той же схеме, что и в других языках программирования.

Синтаксис программы:

```
if условие:  
    действие (s)
```

Программа содержит в себе логическое условие, и, если это условие истинно (равно True), выполнится блок кода, записанный внутри команды *if*. Если же логическое условие ложно (равно False), то блок кода, записанный внутри команды *if* пропускается, а выполнение кода переходит на следующую после блока *if* строчку кода.

Выражение считается истинным, когда оно:

- не равно нулю;

- не является пустым;
- является логическим.

Блок кода, который необходимо выполнить, в случае истинности выражения отделяется четырьмя пробелами слева.

Например:

```
var1 = 100
if var1:
    print ("1 - Got a true expression value")
    print (var1)
var2 = 0
if var2:
    print ("2 - Got a true expression value")
    print (var2)
print ("Good bye!")
```

## 6.4 Конструкция *if...else*

Оператор *else* может использоваться вместе с оператором *if*. Оператор *else* содержит блок кода, который будет выполнен, если результат выражения равен нулю или считается ложью.

Эта условная конструкция имеет следующий синтаксис:

```
if выражение:
    инструкция_1
    инструкция_2
    ...
    инструкция_n
else:
    инструкция_a
    инструкция_b
    ...
    инструкция_x
```

При этом *Блок инструкций 1* будет выполнен, если *Условие* истинно. Если *Условие* ложно, будет выполнен *Блок инструкций 2*.

Команда *else* является опциональной, в каждой *if*-конструкции может быть только одна команда *else*.

---

Например:

```
x= int(input())
if x>0:
    print(x)
else:
    print(-x)
```

## 6.5 Команда *elif*

В некоторых случаях программа должна обрабатывать более двух возможных результатов, чего операторы *if* и *else* обеспечить не могут.

В такой ситуации используется оператор *else if*, который в Python сокращен до *elif*.

Синтаксис команды:

```
if выражение_1:
    инструкции_(блок_1)
elif выражение_2:
    инструкции_(блок_2)
elif выражение_3:
    инструкции_(блок_3)
else:
    инструкции_(блок_4)
```

Команда *elif* позволяет проверить истинность нескольких выражений и, в зависимости от результата проверки, выполнить нужный блок кода.

Как и команда *else*, команда *elif* является опциональной, однако, в отличие от команды *else*, у одной *if*-конструкции может существовать произвольное количество команд *elif*.

Например:

```
var = 100
if var==200:
    print ("1 - Got a true expression value")
    print (var)
elif var==150:
```

---

```
    print ("2 - Got a true expression value")
    print (var)
elif var==100:
    print ("3 - Got a true expression value")
    print (var)
else:
    print ("4 - Got a true expression value")
    print (var)
print ("Good bye!")
```

## 6.6 Вложенные условные инструкции

Внутри условных инструкций можно использовать любые инструкции, в том числе и условную инструкцию. Вложенные операторы `if` позволяют добавить в код второстепенные условия, которые будут проверены, если первичное выражение истинно. При этом вложенные блоки имеют больший размер отступа.

Кроме того, во вложенной конструкции можно добавлять *if..elif...else* внутри другой такой же конструкции.

Например:

```
x= int(input())
y= int(input())
if x>0:
    if y>0: #x>0, y>0
        print("1 chetvert")
    else: #x>0, y<0
        print("4 chetvert")
else:
    if y>0: #x<0, y>0
        print("2 chetvert")
    else: #x<0, y<0
        print("3 chetvert")
```

---

## 6.7 Тернарная условная операция

Тернарные операторы наиболее широко известны как условные выражения. Такие операторы возвращают что-то в зависимости от того, является ли условие истиной или ложью. Аналогом тернарной операции в математической логике является условная дизъюнкция.

```
is_nice=True  
state = "nice" if is_nice else "not nice"  
print("The weather is ", state)
```

Такой подход позволяет быстро проверить условие, а не писать несколько строчек оператора *if*.

Рассмотрим пример. Из двух случайных чисел, одно из которых четное, а другое нечетное, определить и вывести на экран нечетное число.

В данной задаче можно выделить две подзадачи:

- 1) сгенерировать два случайных числа так, чтобы одно было четным, а другое нечетным;
- 2) определить, какое из них нечетное.

Вариант решения первой подзадачи:

Генерируем два случайных числа. Далее проверяем, являются ли оба числа четными или оба нечетными. Если это так, то увеличиваем первое число на 1. При этом в любом случае одно станет четным, а другое – нечетным. Проверку осуществляем в заголовке оператора *if*, строя сложное логическое выражение.

Вариант решения второй подзадачи:

Используем оператор ветвления. Если первое число нечетное, то выводим его, иначе выводим второе.

Проверка чисел на четность выполняется путем определения остатка от деления числа на 2. Если остаток равен нулю, значит, число четное. Если нет, то нечетное.

```
from random import random  
a = int(random() * 100)  
b = int(random() * 100)
```



---

```
if a%2 and b%2 or a%2==0 and b%2==0:
    a += 1
print(a,b)
if a%2:
    print(a)
else:
    print(b)
```

### ***Контрольные вопросы***

1. Опишите принцип работы простого условного оператора. Приведите пример.
2. Опишите принцип работы составного условного оператора. Приведите пример.
3. Опишите синтаксис условного оператора полной и сокращенной формы.
4. Приведите пример блок-схемы, иллюстрирующей работу простого условного оператора.
5. Приведите пример блок-схемы, иллюстрирующей работу составного условного оператора.
6. Сколько команд *else* может быть в конструкции *if*?
7. Приведите пример работы команды *elif*.
8. Что такое тернарная условная операция?
9. Что позволяет добавить вложенный оператор *if*?
10. Приведите пример анализа и реализации программы с использованием составного условного оператора.

### ***Задачи для самостоятельного решения***

1. Из двух случайных чисел, одно из которых четное, а другое нечетное, определить и вывести на экран четное число.
2. Вводятся пять целых чисел. Определить наибольшее отрицательное. Если отрицательных нет, вывести соответствующее сообщение.
3. Вводятся два целых числа. Проверить, делится ли первое на второе.

---

4. Вводится целое число, обозначающее код символа по таблице ASCII. Определить, это код английской буквы или какой-либо иной символ.

5. Перевести число, которое ввел пользователь, в байты и килобайты.

6. Найти корни квадратного уравнения и вывести их на экран, если они есть, в противном случае вывести сообщение о том, что действительных корней нет.

7. Определить, является ли год високосным (пользователь сам вводит год).

8. Определить октант, которому принадлежит точка. Координаты точки ввести с клавиатуры. Октант - любая из восьми областей, на которые пространство делится тремя взаимно перпендикулярными координатными плоскостями.

Расчет их ведётся в следующем порядке согласно знакам координат:

|      | x | y | z |
|------|---|---|---|
| I    | + | + | + |
| II   | — | + | + |
| III  | — | — | + |
| IV   | + | — | + |
| V    | + | + | — |
| VI   | — | + | — |
| VII  | — | — | — |
| VIII | + | — | — |

9. Вводятся координаты ( $x$ ;  $y$ ) точки и радиус круга ( $r$ ). Определить принадлежит ли данная точка кругу, если его центр находится в начале координат.

10. Вводятся три разных числа. Найти, какое из них является средним (больше одного, но меньше другого).

---

## 7 ЦИКЛЫ

Важной частью структурного программирования, помимо операторов условий, считаются циклы. Они помогают автоматизировать последовательные задачи в программе, а именно: повторить выполнение определенных участков кода. Такая необходимость возникает достаточно часто, когда нужно сделать что-нибудь много раз, следовательно, циклы упрощают эту задачу. Цикл – разновидность управляющей конструкции в высокоуровневых языках программирования, предназначенная для организации многократного исполнения набора инструкций. Также циклом может называться любая многократно исполняемая последовательность инструкций, организованная любым способом (например, с помощью условного перехода).

### 7.1 Понятие цикла

Последовательность инструкций, предназначенная для многократного исполнения, называется *телом цикла*. Единичное выполнение тела цикла называется *итерацией*. Выражение, определяющее, будет в очередной раз выполняться итерация или цикл завершится, называется *условием выхода* или *условием окончания цикла* (либо *условием продолжения*) в зависимости от того, как интерпретируется его истинность – как признак необходимости завершения или продолжения цикла. Переменная, хранящая текущий номер итерации, называется *счётчиком итераций цикла* или просто *счётчиком цикла*. Цикл необязательно содержит счётчик, счётчик не обязан быть один – условие выхода из цикла может зависеть от нескольких изменяемых в цикле переменных, а может определяться внешними условиями (например, наступлением определённого времени), в последнем случае счётчик может вообще не понадобиться.

Исполнение любого цикла включает первоначальную инициализацию переменных цикла, проверку условия выхода, исполнение тела

---

цикла и обновление переменной цикла на каждой итерации. Кроме того, большинство языков программирования предоставляет средства для досрочного управления циклом, например, операторы завершения цикла, то есть выхода из цикла независимо от истинности условия выхода (в языке Python – `break`) и операторы пропуска итерации (в языке Python – `continue`).

Достаточно часто встречаются циклические задачи, к ним можно отнести любые списки, будь то: продукты, задачи на день, запланированные экзамены. И действительно, когда люди приходят в магазин, то покупают все, что есть в их списке, не останавливаясь, пока не сделают этого.

В программировании циклы позволяют повторять некоторое действие в зависимости от соблюдения заданного условия. Таким образом организуется исполнение многократной последовательности инструкций.

*Телом цикла* называется та последовательность кода, которую нужно выполнить несколько раз. Единоразовое выполнение цикла – это *итерация*.

Python позволяет также создавать вложенные циклы. Так, сначала программа запустит внешний и в первой его итерации перейдет во вложенный. Затем она снова вернется к началу внешнего и снова вызовет внутренний. Это будет происходить до тех пор, пока последовательность не завершится или не прервется. Такие циклы полезны в том случае, если нужно перебрать определенное количество элементов в списке.

В Python есть только два цикла: *while* и *for*.

## 7.2 Цикл «while»

While с английского языка переводится, как «до тех пор, как». Оператор *while* предназначен для организации циклического процесса в программе. В языке программирования Python оператор *while* ис-

---

пользуется в случаях, если количество повторений цикла заранее неизвестно (в отличие от оператора *for*). В операторе *while* следующий шаг итерации цикла определяется на основе истинности некоторого условия.

Запись цикла *while* в Python выглядит так:

```
while [условие истинно]:  
    [сделать указанное]
```

Приведем пример использования этого цикла:

```
n = 0  
while n < 6:  
    print(n)  
    n += 2
```

Результат:

0  
2  
4

Здесь переменной *n* присваивается значение 0, после чего начинается цикл, в котором проверяется условие, чтобы число было меньше 6. В теле цикла также содержатся две инструкции: первая выводит само число на экран, а вторая увеличивает его значение на два. Цикл таким образом выполняется, пока условие продолжает быть истинным.

После тела цикла можно указать *else* и блок операций, которые необходимо выполнить, когда закончится *while*.

Обычно в ней есть смысл, только если указана инструкция *break*, но программа работает и без последней.

```
n = 3  
while n < 7:  
    print (n, "меньше 7")  
    n = n + 1  
else:  
    print (n, " не меньше 7")
```

Результат:

3 меньше 7

---

4 меньше 7  
5 меньше 7  
6 меньше 7  
7 не меньше 7.

В данной программе переменной *n* присваивается начальное значение – 3, задается условие, что, пока она меньше 7, нужно вывести ее и выражение «меньше 7», затем прибавлять к ней 1. В тех случаях, когда она уже становится равной 7, то в ход пойдет условие, указанное в *else*, и на экране появится, что переменная не меньше 7.

### 7.3 Инструкции *break* и *continue* в цикле *while*

Оператор *break* используется для выхода из цикла Python – прерывает его досрочно. Так, если во время выполнения кода, программа наткнется на *break*, то она сразу прекращает цикл и выходит из него, минуя *else*. Это необходимо, например, если при выполнении инструкций была обнаружена ошибка и дальнейшая работа бессмысленна. Посмотрим на пример его применения:

```
while True:
    name = input("Введите имя:")
    if name == "хватит":
        break
    print("Привет, ", name)
```

Результат:

Введите имя: Алексей

Привет, Алексей

Введите имя: Галина

Привет, Галина

Введите имя: Татьяна

Привет, Татьяна

Введите имя: хватит

После этого выполнение программы будет прервано.

---

Другая инструкция, которая может менять цикл, – это *continue*. Если она указана внутри кода, то все оставшиеся инструкции до конца цикла пропускаются и начинается следующая итерация.

```
n = 0
while n < 7:
    n+=1
    if n==3 or n==5:
        continue
    print(n)
```

Результат:

1  
2  
4  
6  
7

В приведенной выше программе мы видим, что на выходе 3 и 5 отсутствуют. Это связано с тем, что при  $n == 3$  или  $n == 5$  цикл встречает оператор *continue* и управление возвращается к началу цикла.

*Бесконечными циклами* в программировании называются те, в которых условие выхода из них не выполняется.

Цикл *while* становится бесконечным, когда его условие не может быть ложным. Цикл не должен быть бесконечным, поскольку это причина неустойчивой работы программы. Для того чтобы выйти из него, нужно нажать комбинацию клавиш: *CTRL+C*.

## 7.4 Вложенные циклы *while*

Для реализации вложенных циклов можно использовать как вложенные *while*, так и *for*.

*Вложенный цикл* – это цикл, который встречается внутри другого цикла. Структурно это похоже на выражение *if*.

---

Программа сначала выполняет внешний цикл. Внутри первой итерации внешнего цикла запускается внутренний, вложенный цикл. Затем программа возвращается обратно к началу внешнего цикла, завершает вторую итерацию и снова вызывает вложенный цикл. После завершения вложенного цикла программа возвращается в начало внешнего цикла. Это будет происходить до тех пор, пока последовательность не будет завершена или прервана (или пока какое-то выражение не приведёт к нарушению процесса).

Рассмотрим пример получения простых чисел (от 1 до 30) с использованием вложенных циклов `while`.

```
i = 1
while(i < 30):
    j = 2
    while(j <= (i/j)):
        if not(i%j):
            break
    j+= 1
    if (j > i/j):
        print (i, "простое число")
    i+=1
```

Результат:

1 простое число  
2 простое число  
3 простое число  
5 простое число  
7 простое число  
11 простое число  
13 простое число  
17 простое число  
19 простое число  
23 простое число  
29 простое число.



---

## 7.5 Цикл «*for*»

*Цикл со счётчиком* – цикл, в котором некоторая переменная изменяет своё значение от заданного начального значения до конечного значения с некоторым шагом, и для каждого значения этой переменной тело цикла выполняется один раз. В большинстве процедурных языков программирования реализуется оператор *for*, в котором указывается счётчик (так называемая «переменная цикла»), требуемое количество проходов (или граничное значение счётчика) и, возможно, шаг, с которым изменяется счётчик.

Ещё одним вариантом цикла является цикл, задающий выполнение некоторой операции для объектов из заданного множества, без явного указания порядка перечисления этих объектов. Такие циклы называются *совместными* (а также циклами по коллекции, циклами просмотра) и представляют собой формальную запись инструкции вида: «Выполнить операцию X для всех элементов, входящих во множество M». Совместный цикл, теоретически, никак не определяет, в каком порядке операция будет применяться к элементам множества, хотя конкретные языки программирования, разумеется, могут задавать конкретный порядок перебора элементов. Произвольность даёт возможность оптимизации исполнения цикла за счёт организации доступа не в заданном программистом, а в наиболее выгодном порядке. При наличии возможности параллельного выполнения нескольких операций возможно даже распараллеливание выполнения совместного цикла, когда одна и та же операция одновременно выполняется на разных вычислительных модулях для разных объектов, при том что логически программа остаётся последовательной.

Совместные циклы имеются в некоторых языках программирования (C#, Eiffel, Java, JavaScript, Perl, Python, PHP, LISP, Tcl и др.) – они позволяют выполнять цикл по всем элементам заданной коллекции объектов. В определении такого цикла требуется указать только коллекцию объектов и переменную, которой в теле цикла будет при-

---

своею значение обрабатываемого в данный момент объекта (или ссылка на него).

Цикл *for* в Python выполняет написанный код повторно согласно введенной переменной или счетчику. Он используется только тогда, когда необходимо совершить перебор элементов заранее известное число раз. Что это значит? У нас имеется список. Сначала из него берется первый элемент, потом – второй и так далее, но с каждым из них совершается действие, которое указано в теле *for*. Примерно это выглядит так:

```
For [элемент] in [последовательность]:  
[сделать_указанное]
```

*For* может содержать данные разных типов: цифры, слова и прочее. Очень важная часть идеологии Питона – это цикл *for*, который предоставляет удобный способ перебрать все элементы некоторой последовательности. В этом отличие Питона от Паскаля, где обязательно надо перебирать именно индексы элементов, а не сами элементы.

Рассмотрим пример:

```
for i in 10, 14, "первый", "второй", "третий":  
    print(i)
```

Результат:

10

14

первый

второй

третий.

Для упрощения часто используется функция *range()*, или диапазон. В циклах она указывает на необходимое количество повторов последовательности, уточняя, какие именно элементы из списка *for* нам необходимы в данный момент. В скобках может быть указано от одного до трех чисел:

- одно указывает на то, что нужно проверить все числа от 0 и до него (не включая последнее число);

---

– два говорят о том, что перебрать нужно все числа, находящиеся между ними;

– три числа сгенерируют список от первого до второго, но с шагом, равным третьей цифре.

Рассмотрим пример. Можно записать в следующем виде:

```
for i in [7, 8, 9, 10, 11]:
```

```
    print(i)
```

Результат:

7

8

9

10

11.

Однако это не оптимально, особенно если чисел слишком много, поэтому лучше использовать указанный выше `range()`:

```
for i in range(3):
```

```
    print(i)
```

```
for i in range(7, 12):
```

```
    print(i)
```

```
for i in range(7, 12, 2):
```

```
    print(i)
```

Результат:

Для 1 цикла

0

1

2

Для 2 цикла

7

8

9

10

11

Для 3 цикла

7

9

11

## 7.6 Оператор следующего прохода *continue* в цикле *for*

С помощью этого оператора начинается следующий проход цикла, минуя оставшиеся после него операторы в теле цикла.

---

Этот пример делает цикл по строке и по условию проверяет каждый символ на соответствие с числом 3. Если находит его, то увеличивает счётчик *a*, а в самом конце выводит общее число троек в строке.

```
a=0
for i in "3232453232455456":
    if i != "3":
        continue
    print(i)
    a=a+1
print("Число троек в строке = ",a)
```

Результат:

3  
3  
3  
3

Число троек в строке = 4.

## 7.7. Оператор прерывания цикла *break* в цикле *for*

С помощью этого оператора циклы досрочно прерываются. Удобно использовать, когда все, что нужно, уже подсчитано.

```
a=0
for i in "32324532732455456":
    if i == "7":
        break
    print(i)
    a=a+1
print("Число символов в строке до 7 = ",a)
```

Результат:

3  
2  
3  
2

---

4  
5  
3  
2

Число символов в строке до 7 = 8.

## 7.8 Инструкция проверки прерывания *else* в цикле *for*

Оператор *else*, примененный в цикле *for* или *while*, проверяет, был ли произведен выход из цикла инструкцией *break* или же «естественным» образом. Блок инструкций внутри *else* выполнится только в том случае, если выход из цикла произошёл без помощи *break*.

```
a=0
for i in "32324532732455456":
    if i == "7":
        break
    print(i)
    a=a+1
else:
    print("В строке символов не содержится символ 7")
print("Количество символов в строке (до символа 7) =" , a)
```

Вот так может быть выполнен этот код, если семёрка встречается:

Результат:

3  
2  
3  
2  
4  
5  
3  
2.

Количество символов в строке (до символа 7) = 8.

А вот так, если её нет в строке ("3232453232455456").

---

Результат:

3  
2  
3  
2  
4  
5  
3  
2  
3  
2  
4  
5  
5  
4  
5  
6.

В строке символов не содержится символ 7.

Количество символов в строке (до символа 7) = 16.

Здесь главное – не ошибиться с расстановкой отступов, у *else* их нет, так как этот оператор выше проверки условия *if*.

## 7.9 Вложенные циклы *for*

Python позволяет вкладывать циклы друг в друга.

*Вложенный цикл* – это цикл, который встречается внутри другого цикла. Структурно это похоже на выражение *if*.

Программа сначала выполняет внешний цикл. Внутри первой итерации внешнего цикла запускается внутренний, вложенный цикл. Затем программа возвращается обратно к началу внешнего цикла, завершает вторую итерацию и снова вызывает вложенный цикл. После завершения вложенного цикла программа возвращается в начало

---

внешнего цикла. Это будет происходить до тех пор, пока последовательность не будет завершена или прервана (или пока какое-то выражение не приведёт к нарушению процесса).

В данном примере внешний цикл будет перебирать список чисел (*nl*), а внутренний – алфавит (*al*).

```
nl = [1, 2, 3]
al = ['a', 'b', 'c']
for i in nl:
    print(i)
    for k in al:
        print(k)
```

Результат:

```
1
a
b
c
2
a
b
c
3
a
b
c.
```

Программа завершает первую итерацию, выводя на экран 1, после чего запускается внутренний цикл, который выводит последовательно «а, b, с». После этого программа возвращается в начало внешнего цикла, выводит 2, а затем снова обрабатывает внутренний цикл.

Вложенные циклы *for* могут быть полезны для перебора элементов в списках, составленных из списков. Если в списке, который состоит из списков, использовать только один цикл, программа будет выводить каждый внутренний список в качестве элемента.

---

## Контрольные вопросы

1. Что такое цикл?
2. Что такое тело цикла?
3. Что такое итерация?
4. Бесконечные циклы.
5. Цикл *while* в Python.
6. Цикл *for* в Python.
7. Общая форма оператора *while*.
8. Общая форма оператора *for*.
9. Что выполняет оператор *break*?
10. Что выполняет оператор *continue*?
11. В чем заключаются основные различия между инструкциями *break* и *continue*?
12. Общая форма оператора *while*, предусматривающего использование блока *else*.
13. Общая форма оператора *for*, предусматривающего использование блока *else*.
14. Отличия оператора *while* от оператора *for*.
15. Что такое вложенный цикл?

## Задачи для самостоятельного решения

1. Задан некоторый список A, содержащий целые числа. Разработать программу, которая вычисляет сумму элементов списка.
2. Создать список заданной длины и записать в него заданное число
3. Задано число  $a$  ( $1 < a < 1.5$ ). Из чисел  $1+1/2$ ,  $1+1/3$ ,  $1+1/4$  ..., напечатать те, которые не меньше  $a$ .
4. Задан список строк. В каждой строке подсчитать количество вхождений заданного символа.
5. Напишите программу для поиска всех делителей числа.



- 
6. Вывести на экран все числа от 50 до 150 включительно.
  7. Напишите программу, определяющую наименьшее общее кратное чисел  $a$  и  $b$ .
  8. Напишите программу, которая находит самую большую цифру в числе.
  9. Заполнить список ста нулями, кроме первого и последнего элементов, которые должны быть равны единице.
  10. Сформировать возрастающий список из чётных чисел (количество элементов 45).
  11. 5. Пользователь вводит число  $x$ . Определить, содержит ли список данное число  $x$ . Если содержит, то вывести на экран число 7145, если не содержит, то вывести на экран число 5741.
  12. Найти сумму  $n$ -элементов ряда  $1, -0.5, 0.25, -0.125, \dots$
  13. Определить количество простых чисел в диапазоне от  $a$  до  $b$ .
  14. Посчитать сумму и произведение цифр числа

---

## 8 СПИСКИ

Большинство программ работает не с отдельными переменными, а с набором переменных. Например, программа может обрабатывать информацию об учащихся класса, считывая список учащихся с клавиатуры или из файла, при этом изменение количества учащихся в классе не должно требовать модификации исходного кода программы.

Для хранения таких данных можно использовать структуру данных, называемую в Python *список* (в большинстве же языков программирования используется другой термин – «массив»).

Списки – это самое общее представление последовательностей, реализованных в языке Python.

*Списки* – это упорядоченные по местоположению коллекции объектов произвольных типов, размер которых не ограничен. Кроме того, в отличие от строк, списки являются изменяемыми – они могут модифицироваться как с помощью операций присваивания по смещениям, так и с помощью разнообразных методов работы со списками.

Списки можно задать перечислением.

```
Primes = [2, 3, 5, 7, 11, 13]
Rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue',
           'Indigo', 'Violet']
```

В списке `Primes` – 6 элементов, а именно:

```
Primes[0] == 2,
Primes[1] == 3,
Primes[2] == 5,
Primes[3] == 7,
Primes[4] == 11,
Primes[5] == 13.
```

Список `Rainbow` состоит из 7 элементов, каждый из которых является строкой.

---

Так же, как и символы в строке, элементы списка можно индексировать отрицательными числами с конца, например, `Primes[-1] == 13`, `Primes[-6] == 2`.

Длину списка, то есть количество элементов в нем, можно узнать при помощи функции *len*, например: `len(Primes) == 6`.

В отличие от строк, элементы списка можно изменять, присваивая им новые значения.

```
Rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue',
           'Indigo', 'Violet']
print(Rainbow[0])
Rainbow[0] = 'красный'
print('Выведем радугу')
for i in range(len(Rainbow)):
    print(Rainbow[i])
```

В результате выполнения данного кода получим следующий результат:

```
красный
Orange
Yellow
Green
Blue
Indigo
Violet.
```

Рассмотрим несколько способов создания и считывания списков. Прежде всего, можно создать пустой список (не содержащий элементов, длины 0), а в конец списка можно добавлять элементы при помощи метода *append*. Например, пусть программа получает на вход количество элементов в списке *n*, а потом *n* элементов списка по одному в отдельной строке:

```
a = [] # создаем пустой список
n = int(input()) # считываем количество элементов в списке
for i in range(n):
    new_element = int(input()) # считываем очередной элемент
```

---

```
a.append(new_element) # добавляем его в список
print (a)
```

В этом примере создается пустой список, далее считывается количество элементов в списке, затем по одному считываются элементы списка и добавляются в его конец. То же самое можно записать, сэкономив переменную *n*:

```
a = []
for i in range(int(input())):
    a.append(int(input()))
print (a)
```

Для списков целиком определены следующие операции: конкатенация списков (сложение списков, то есть приписывание к одному списку другого) и повторение списков (умножение списка на число).

Например:

```
a = [1, 2, 3]
b = [4, 5]
c = a + b
d = b * 3
```

В результате список *c* будет равен [1, 2, 3, 4, 5], а список *d* будет равен [4, 5, 4, 5, 4, 5].

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* считывается *i*-й элемент списка, а затем вывести элементы списка на экран:

```
a=[0]*int(input())
for i in range(len(a)):
    a[i]=int(input())
```

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:

```
for i in range(len(a)):
```

---

```
print(a[i])
```

Здесь в цикле меняется индекс элемента `i`, затем выводится элемент списка с индексом `i`.

```
for elem in a:  
    print(elem, end = ' ')
```

В этом примере элементы списка выводятся в одну строку, разделенные пробелом, при этом в цикле меняется не индекс элемента списка, а само значение переменной (например, в цикле `for elem in ['red', 'green', 'blue']` переменная `elem` будет последовательно принимать значения `'red'`, `'green'`, `'blue'`).

Последовательностями в Питоне являются строки, списки, значения функции `range()` (это не списки) и ещё кое-какие другие объекты.

## 8.1 Списки и их особенности

С точки зрения производительности (performance), списки имеют следующие особенности.

1. Время доступа к элементу есть величина постоянная и не зависит от размера списка.
2. Время на добавление одного элемента в конец списка есть величина постоянная.
3. Время на вставку зависит от того, сколько элементов находится справа от него, то есть чем ближе элемент к концу списка, тем быстрее идет его вставка.
4. Удаление элемента происходит так же, как и в пункте 3.
5. Время, необходимое на реверс списка, пропорционально его размеру –  $O(n)$ .
6. Время, необходимое на сортировку, зависит логарифмически от размера списка.

Элементы списка не обязательно должны быть одного типа. Приведем вариант статического определения списка:

---

```
lst = ['spam', 'drums', 100, 1234]
```

Как и для строк, для списков нумерация индексов начинается с нуля. Для списка можно получить срез, объединить несколько списков и так далее:

```
lst[1:3]
```

Результат: ['drums', 100].

Можно менять как отдельные элементы списка, так и диапазон:

```
lst[3] = 'piano'
```

```
lst[0:2] = [1, 2]
```

```
lst
```

Результат: [1, 2, 100, 'piano'].

Вставка:

```
lst[1:1] = ['guitar', 'microphone']
```

```
lst
```

Результат: [1, 'guitar', 'microphone', 2, 100, 'piano'].

Можно сделать выборку из списка с определенной частотой:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
```

```
numbers[::4]
```

Результат: [1, 5, 9].

## 8.2 Операции над последовательностями

К операциям над списками относят:

- 1) копирование списка;
- 2) сложение и умножение списков;
- 3) итерацию – проход в цикле по элементам списка;
- 4) конструктор списков (list comprehension);
- 5) распаковку списка – sequence unpacking.

Создание копии списка.

`L1 = L2[:]` – создание второй копии списка. Здесь создается вторая копия объекта.

`L1 = list(L2)` – тоже создание второй копии списка.

---

$L1 = L2$  – создание второй ссылки, а не копии. 3-й вариант показывает, что создаются две ссылки на один и тот же объект, а не две копии.

Сложение или конкатенация списков:

$L1 + L2$

Умножение или повтор списков:

$L1 * 2$

Итерацию списков в питоне можно делать несколькими способами:

- простая итерация списка – *for x in L:*
- сортированная итерация – *for x in sorted(L):*
- уникальная итерация – *for x in set(L):*
- итерация в обратном порядке – *for x in reversed(L):*
- исключаящая итерация (например, вывести элементы 1-го списка, которых нет во 2-м списке) – *for item in set(L).difference(L2).*

Списки в языке Python являются аналогом массивов в других языках программирования, но они обладают более широкими возможностями. Кроме того, размер списков не ограничен, благодаря чему они могут увеличиваться и уменьшаться по мере необходимости в результате выполнения операций, характерных для списков. В списках применяются следующие операции и методы ( $L$  – некоторый список):

- *x in L* – проверка, содержится ли элемент в списке. Возвращает True или False.
- *x not in L* – проверка того, что элемент не содержится в списке.
- *min(L)* – наименьший элемент списка.
- *max(L)* – наибольший элемент списка.
- *L.append(x)* – увеличивает размер списка  $L$  и вставляет в конец новый элемент  $x$ .

- 
- `L.clear()` – очищает список.
  - `L.count(x)` – количество вхождений элемента `x` в список.
  - `L.copy()` – поверхностная копия списка.
  - `L1.extend(L2)` – расширяет список `L1`, добавляя в конец все элементы списка `L2`.
  - `L.index(x)` – индекс первого вхождения элемента `x` в список, при его отсутствии генерирует исключение `ValueError`.
  - `L.index(x, [start [, end]])` – возвращает положение первого элемента со значением `x` (при этом поиск ведется от `start` до `end`).
  - `L.insert(i, x)` – вставляет на `i`-ый элемент значение `x`.
  - `L.pop(n)` или `del L[n]` – удаляет из списка `L` элемент с заданным номером `n`, что приводит к уменьшению списка.
  - `L.remove(x)` – удаляет первый элемент в списке, имеющий значение `x`. `ValueError`, если такого элемента не существует.
  - `L.reverse()` – запись списка в обратном порядке.
  - `L.sort([key=функция])` – сортирует список на основе функции.

Хотя списки не имеют фиксированного размера, язык Python, тем не менее, не допускает возможности обращаться к несуществующим элементам списка. Обращение к элементам списка по индексам, значения которых выходят за пределы списка, всегда является ошибкой.

### 8.3 Вложенные списки

Одна из особенностей базовых типов языка Python состоит в том, что они поддерживают возможность создания вложенных конструкций произвольной глубины и в любых комбинациях (например, можно создать список, содержащий словарь, который содержит другой список, и так далее). Одно из очевидных применений этой особенности – представление матриц, или «многомерных массивов», в



---

языке Python. Делается это с помощью списка, содержащего вложенные списки:

Например, можно создать матрицу 3\*3 в виде вложенных списков:

```
M = [[1, 2, 3],  
      [4, 5, 6],  
      [7, 8, 9]]
```

Результат вывода матрицы M: `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`.

Здесь был реализован список, состоящий из трех других списков. В результате была получена матрица чисел 3x3. Обращаться к такой структуре можно разными способами:

`M[1]` – получить вторую строку (`[4, 5, 6]`).

`M[1][2]` – получить вторую строку, а затем третий элемент этой строки (6).

Первая операция в этом примере возвращает вторую строку целиком, а вторая – третий элемент в этой строке. Соединение операций индексирования позволяет все дальше и дальше погружаться в глубь вложенной структуры объектов.

## 8.4 Генераторы списков

Помимо обычных операций над последовательностями и методов списков, Python предоставляет возможность выполнять более сложные операции над списками, известные как выражения генераторов списков (`list comprehension expression`), которые представляют эффективный способ обработки таких структур, как приведенная в примере матрица. Предположим, например, что требуется извлечь из матрицы второй столбец. Строку легко можно получить, выполнив операцию индексирования, потому что матрица хранится в виде строк, однако, благодаря генераторам списков, получить столбец ничуть не сложнее:

```
col2 = [row[1] for row in M]  
col2
```

---

Результат вывода col2: [2, 5, 8].

Генераторы списков следуют традиции системы представления множеств; они позволяют создавать новые списки, выполняя выражение для каждого элемента в последовательности, по одному за раз, слева направо. Генераторы списков заключены в квадратные скобки (чтобы отразить тот факт, что они создают список) и составлены из выражения и конструкции цикла, которые используют одно и то же имя переменной (в данном случае row). В предыдущем примере генератор списков интерпретируется так: «Получить элементы row[1] из каждой строки матрицы M и создать из них новый список». Результатом является новый список, содержащий значения из второго столбца матрицы. На практике генераторы списков могут приобретать еще более сложную форму:

```
[row[1] + 1 for row in M]
```

Результат: [3, 6, 9].

```
[row[1] for row in M if row[1] % 2 == 0]
```

Результат: [2, 8].

Первая операция в этом примере прибавляет 1 к значениям всех отобранных элементов, а вторая использует условный оператор `if` для исключения из результата нечетных чисел с помощью операции деления по модулю — `%` (остаток от деления).

Генераторы списков, применяемые к спискам, возвращают в качестве результатов новые списки, но могут использоваться и для любых других объектов, допускающих выполнение итераций.

Для создания списка, заполненного одинаковыми элементами, можно использовать оператор повторения списка, например (список заполняется пятью нулями):

```
n = 5
```

```
a = [0] * 5
```

Для создания списков, заполненных по более сложным формулам, можно использовать генераторы — выражения, позволяющие заполнить список некоторой формулой. Общий вид генератора следующий:

---

[выражение for переменная in последовательность]

*переменная* – идентификатор некоторой переменной;

*последовательность* – последовательность значений, который принимает данная переменная (это может быть список, строка или объект, полученный при помощи функции range);

*выражение* – некоторое выражение, как правило зависящее от использованной в генераторе переменной, которым будут заполнены элементы списка.

Вот несколько примеров использования генераторов.

Создать список, состоящий из n нулей можно и при помощи генератора:

```
n = 5
a = [0 for i in range(n)]
```

Создать список, заполненный квадратами целых чисел можно так:

```
n = 5
a = [i ** 2 for i in range(n)]
```

Если нужно заполнить список квадратами чисел от 1 до n, то можно изменить параметры функции range на range(1, n + 1):

```
n = 5
a = [i ** 2 for i in range(1, n + 1)]
```

Вот так можно получить список, заполненный случайными числами от 1 до 9 (используя функцию randrange из модуля random):

```
from random import randrange
n = 10
a = [randrange(1, 10) for i in range(n)]
```

А в этом примере список будет состоять из строк, считанных со стандартного ввода: сначала нужно ввести число элементов списка (это значение будет использовано в качестве аргумента функции range), потом заданное количество строк:

```
a = [input() for I in range(int(input()))]
```

---

## 8.5 Срезы

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

$A[i:j]$  – срез из  $j-i$  элементов  $A[i], A[i+1], \dots, A[j-1]$ .

$A[i:j:-1]$  – срез из  $i-j$  элементов  $A[i], A[i-1], \dots, A[j+1]$  (то есть меняется порядок элементов).

$A[i:j:k]$  – срез с шагом  $k$ :  $A[i], A[i+k], A[i+2*k], \dots$ . 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. Но можно менять и целиком срезы.

Например:

```
A = [1, 2, 3, 4, 5]
```

```
A[2:4] = [7, 8, 9]
```

Результат: получится список, у которого вместо двух элементов среза  $A[2:4]$  вставлен новый список уже из трех элементов. Теперь список стал равен  $[1, 2, 7, 8, 9, 5]$ .

```
A = [1, 2, 3, 4, 5, 6, 7]
```

```
A[::-2] = [10, 20, 30, 40]
```

Результат: получится список  $[40, 2, 30, 4, 20, 6, 10]$ . Здесь  $A[::-2]$  – это список из элементов  $A[-1], A[-3], A[-5], A[-7]$ , которым присваиваются значения 10, 20, 30, 40 соответственно.

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

Следует отметить, что  $A[i]$  – это элемент списка, а не срез!

---

## 8.6 Стек и очереди

Список можно использовать как *стек* – когда последний добавленный элемент извлекается первым (LIFO, last-in, first-out). Для извлечения элемента с вершины стека есть метод `pop()`:

```
stack = [1, 2, 3, 4, 5]
stack.append(6)
stack.append(7)
stack.pop()
stack
```

Результат: `[1, 2, 3, 4, 5, 6]`.

Список можно использовать как *очередь* – элементы извлекаются в том же порядке, в котором они добавлялись (FIFO, first-in, first-out). Для извлечения элемента используется метод `pop()` с индексом 0:

```
queue = ['rock', 'in', 'roll']
queue.append('alive')
queue.pop(0)
queue
```

Результат: `['in', 'roll', 'alive']`.

## 8.7 Методы `split` и `join`

Элементы списка могут вводиться по одному в строке, в этом случае строку целиком можно считать функцией `input()`. После этого можно использовать метод строки `split()`, возвращающий список строк, которые получатся, если исходную строку разрезать на части по пробелам.

```
s = input()
a = s.split()
```

Если при запуске этой программы ввести строку `1 2 3`, то список `a` будет равен `['1', '2', '3']`. Заметим, что список будет состоять из строк, а не из чисел. Если хочется получить список именно из чисел, то можно затем элементы списка по одному преобразовать в числа:

```
a = input().split()
```

---

```
for i in range(len(a)):
    a[i] = int(a[i])
```

Используя генераторы, то же самое можно сделать в одну строку:

```
a = [int(s) for s in input().split()]
```

Если нужно считать список действительных чисел, то нужно заменить тип *int* на тип *float*.

У метода *split()* есть необязательный параметр, который определяет, какая строка будет использоваться в качестве разделителя между элементами списка. Например, вызов метода *split('.')* вернет список, полученный разрезанием исходной строки по символам '.':

```
a='192.168.0.1'.split('.')
```

Результат: ['192', '168', '0', '1'].

В Питоне можно вывести список строк при помощи однострочной команды. Для этого используется метод строки *join*. У этого метода один параметр: список строк. В результате возвращается строка, полученная соединением элементов переданного списка в одну строку, при этом между элементами списка вставляется разделитель, равный той строке, к которой применяется метод.

```
a = ['red', 'green', 'blue']
print(' '.join(a))
```

Результат: red green blue.

```
a = ['red', 'green', 'blue']
print('').join(a)
```

Результат: redgreenblue.

```
a = ['red', 'green', 'blue']
print('***'.join(a))
```

Результат: red\*\*\*green\*\*\*blue.

Если же список состоит из чисел, то придется использовать генераторы. Вывести элементы списка чисел, разделяя их пробелами, можно так:

```
a = [1, 2, 3]
print(' '.join([str(i) for i in a]))
```

Результат: 1 2 3.

---

## *Контрольные вопросы*

1. Что такое список (list) в Python?
2. Какими способами задаются списки?
3. Особенности, характерные для списков.
4. Основные операции, выполняемые над списками.
5. Главная отличительная особенность списков от строк.
6. Как создать копию списка?
7. Представление матриц – «многомерных массивов» в языке Python.
8. Генератор, как способ удобной записи списков.
9. Использование срезов со списками.
10. Методы, используемые для извлечения элементов из списков.

## *Задачи для самостоятельного решения*

1. Напишите программу, на вход которой подаётся список чисел одной строкой. Программа должна для каждого элемента этого списка вывести сумму двух его соседей. Для элементов списка, являющихся крайними, одним из соседей считается элемент, находящийся на противоположном конце этого списка. Например, если на вход подаётся список «1 3 5 6 10», то на выход ожидается список «13 6 9 15 7». Если на вход пришло только одно число, надо вывести его же. Вывод должен содержать одну строку с числами нового списка, разделёнными пробелом.

2. Напишите программу, которая принимает на вход список чисел в одной строке и выводит на экран в одну строку значения, которые повторяются в нём более одного раза. Выводимые числа не должны повторяться, порядок их вывода может быть произвольным. Например: 4 8 0 3 4 2 0 3.

3. Выполните обработку элементов прямоугольной матрицы А, имеющей N строк и М столбцов. Все элементы имеют целый тип. Да-

---

но целое число  $N$ . Определите, какие столбцы имеют хотя бы одно такое число, а какие не имеют.

4. Известно, что на доске  $8 \times 8$  можно расставить 8 ферзей так, чтобы они не били друг друга. Вам дана расстановка 8 ферзей на доске, определите, есть ли среди них пара бьющих друг друга. Программа получает на вход восемь пар чисел, каждое число от 1 до 8 – координаты 8 ферзей. Если ферзи не бьют друг друга, выведите слово NO, иначе выведите YES.

5.  $N$  кеглей выставили в один ряд, занумеровав их слева направо числами от 1 до  $N$ . Затем по этому ряду бросили  $K$  шаров, при этом  $i$ -й шар сбил все кегли с номерами от  $l_i$  до  $r_i$  включительно. Определите, какие кегли остались стоять на месте. Программа получает на вход количество кеглей  $N$  и количество бросков  $K$ . Далее идет  $K$  пар чисел  $l_i, r_i$ , при этом  $1 \leq l_i \leq r_i \leq N$ . Программа должна вывести последовательность из  $N$  символов, где  $j$ -й символ есть “I”, если  $j$ -я кегля осталась стоять, или “.”, если  $j$ -я кегля была сбита.

6. Дан список. Выведите те его элементы, которые встречаются в списке только один раз. Элементы нужно выводить в том порядке, в котором они встречаются в списке.

7. Дан список чисел. Посчитайте, сколько в нем пар элементов, равных друг другу. Считается, что любые два элемента, равные друг другу образуют одну пару, которую необходимо посчитать.

8. Дан список целых чисел, число  $k$  и значение  $C$ . Необходимо вставить в список на позицию с индексом  $k$  элемент, равный  $C$ , сдвинув все элементы, имевшие индекс не менее  $k$ , вправо.

9. Дан список из чисел и индекс элемента в списке  $k$ . Удалите из списка элемент с индексом  $k$ , сдвинув влево все элементы, стоящие правее элемента с индексом  $k$ . Программа получает на вход список, затем число  $k$ .

10. В списке все элементы различны. Поменяйте местами минимальный и максимальный элемент этого списка.



---

## 9 КОРТЕЖИ

*Кортеж (tuple)* – это неизменяемая последовательность, обычно используемая для хранения разнотипных объектов. По своей сути кортеж напоминает список. Но большой отличительной особенностью кортежа от списка является то, что кортеж неизменяем (т. е. добавлять и удалять элементы кортежа нельзя).

Кортежи можно создавать различными способами (перечислением элементов или с использованием функции *tuple* (*[<Последовательность>]*), которая позволяет преобразовать любую последовательность в кортеж). Рассмотрим некоторые способы создания кортежей:

```
a = ()
aa = tuple()
b = (1, 2, 3, 4, 5)
bb = tuple([1, 2, 3, 4, 5])
c = 6, 7, 8
d = (9,)
```

Результат:

*a* и *aa* – пустые кортежи,

*b* и *bb* – кортежи, содержащие числа 1, 2, 3, 4, 5,

*c* – кортеж содержит числа 6, 7, 8,

*d* – одноэлементный кортеж, в нем содержится один элемент число 9 (следует обратить внимание, что после числа стоит запятая, именно она показывает, что создается кортеж).

*Свойства кортежей:*

- Нельзя добавлять элементы к кортежу. Кортежи не имеют методов *append()* или *extend()*.

- Нельзя удалять элементы из кортежа. Кортежи не имеют методов *remove()* или *pop()*.

- Можно искать элементы в кортеже, поскольку это не изменяет кортеж.

---

– Также можно использовать оператор *in*, чтобы проверить, существует ли элемент в кортеже.

– Кортежи делают код безопаснее в том случае, если есть «защищенные от записи» данные, которые не должны изменяться. Использование кортежей вместо списков избавит от необходимости использовать оператор *assert*, дающий понять, что данные неизменяемы и что нужно приложить особые усилия (и особую функцию), чтобы это обойти.

– Некоторые кортежи могут использоваться в качестве ключей словаря (конкретно, кортежи, содержащие неизменяемые значения, например, строки, числа и другие кортежи). Списки никогда не могут использоваться в качестве ключей словаря, потому что списки – изменяемые объекты.

Существует несколько причин, по которым стоит использовать кортежи вместо списков. Одна из них – это желание обезопасить данные от случайного изменения.

Использовать кортежи в задаче следует по нескольким причинам – во-первых, это экономия места. Дело в том, что кортежи в памяти занимают меньший объем, по сравнению со списками. Это можно проверить с помощью оператора *sizeof*. Во-вторых, прирост производительности, который связан с тем, что кортежи работают быстрее, чем списки (то есть на операции перебора элементов и тому подобные будет тратиться меньше времени). Важно также отметить, что кортежи можно использовать в качестве ключа у словаря.

```
a = [1, 2, 3, 4]
b = (1, 2, 3, 4)
sa = a.__sizeof__()
sb = b.__sizeof__()
```

Результат: для *sa* – 72 байта, *sb* – 56 байт.

---

## 9.1 Операции, поддерживаемые кортежем

Так как кортеж является последовательностью, можно определить его длину. Для этого необходимо использовать оператор *len*.

При помощи оператора *in* можно проверить истинное или ложное нахождение какого-либо элемента в кортеже.

Также элементы кортежей можно объединять. В программе такое объединение обозначается знаком «+». Важно правильно указать, какой кортеж стоит слева, а какой справа.

Кортеж можно дублировать. Для этого в программе необходимо, с помощью знака «\*» указать, какое количество раз повторяются элементы кортежа.

Используя операторы *min* и *max*, можно найти минимальное и максимальное числа кортежа.

С помощью оператора *sum* находится сумма всех элементов кортежа.

```
tup = (1, 2, 3, 4, 5, 6)
tup1 = (10, 11, 12)
a = len(tup)
b = 7 in tup
c = 4 in tup
d = tup + tup1
e = tup1 + tup
f = tup * 3
mx = max(tup)
mn = min(tup)
s = sum(tup)
```

Результат:

```
a = 6
b = False
c = True
d = (1, 2, 3, 4, 5, 6, 10, 11, 12)
e = (10, 11, 12, 1, 2, 3, 4, 5, 6)
f = (1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6)
```

---

$mx = 6$

$mn = 1$

$s = 21$ .

Доступ к элементам кортежа осуществляется так же, как к элементам списка – через указание индекса. Все элементы кортежа пронумерованы, начиная с нулевого индекса. И для того, чтобы обратиться к элементу кортежа, необходимо обратиться к его индексу.

Метод *count* считает, сколько раз встречался тот или иной элемент.

```
a = (1, 5, 4, 3, 4, 3, 4, 4, 5, 7)
```

```
n1 = a.index(3)
```

```
n2 = a[3]
```

```
k1 = a.count(3)
```

```
k2 = a.count(5)
```

Результат:

$n1 = 3$

$n2 = 3$

$k1 = 2$

$k2 = 4$ .

Из кортежа можно извлечь как элемент, так и срез. В этом случае мы получим кортеж, состоящий из элементов, расположенных в промежутке среза. Следует уточнить, что при указании среза используются не номера элементов, а номера промежутков между ними. Перед первым элементом находится промежуток с индексом 0.

```
b = (5, 3.6, "квадрат", 15, 'B')
```

```
print (b [1])
```

```
print (b [2:4])
```

Результат:

3.6

('квадрат', 15).

Может возникнуть ситуация, когда внутри кортежа находится список. Также с помощью индекса разрешается обращаться к этому списку.

---

```
a = (1, 4, 6, 7, [10, 20, 30])
print(a[4])
```

Результат: [10, 20, 30].

Так как список – изменяемый объект, можно вносить в него корректировки, используя метод `append`.

```
a = (1, 4, 6, 7, [10, 20, 30])
a[4].append(100)
print(a[4])
print(a)
```

Результат:

[10, 20, 30, 100]

(1, 4, 6, 7, [10, 20, 30, 100]).

Так как кортеж является последовательностью, то пользователь вправе обходить элементы кортежа с помощью цикла *for*.

```
a=(1, 2, 3)
for i in a
print(i)
```

Результат:

1

2

3.

При данном варианте обходят по значению, то есть переменная *i* сохраняет поочередно все значения из этого кортежа.

Но так как эта последовательность упорядоченная, ее можно обойти по индексу. Для этого используется *range*, который зависит от длины кортежа.

Теперь *i* будет принимать значения индексов, то есть изменяться от нуля до длины (не включительно), и необходимо обращаться по индексу к этому кортежу.

```
A=(1, 2, 3, 100, 32, „hello“)
for i in range ( len (a)):
print( a[i])
```

Результат:

1, 2, 3, 100, 32, hello.

---

## 9.2 Удаление кортежей

Удалить отдельные элементы из кортежа невозможно. Но можно удалить кортеж целиком, используя оператор *del*.

## 9.3 Список кортежей

Иногда бывают задачи, в которых нужно хранить таблицы с данными, которые называются матрицами (или двумерными массивами). Для работы с массивами используется библиотека *array*. К сожалению, не получится с ее помощью сделать массив кортежей, потому что элементом массива не может быть кортеж.

Из этой ситуации есть выход – создание в Python списка кортежей. Вариант объявления такого списка представлен ниже:

```
a=[ (1,2,3) , (4,5,6) ]
```

## 9.4 Сортировка

Иногда нужно отсортировать имеющиеся элементы списка. Благодаря встроенной функции *sorted*, все это делается достаточно легко.

```
a = ('One', 'Two', 'Three')
a = tuple(sorted(a))
print(a)
```

Результат: ('One', 'Three', 'Two').

Видно, что произошла сортировка кортежа Python по алфавиту. Стандартную сортировку можно провести и по числовым элементам.

```
a= (3, 2, 5, 1, 12, 7)
a= tuple (sorted (a))
print(a)
```

Результат: 1, 2, 3, 5, 7, 12.

Можно заметить, что произошла сортировка по возрастанию.

---

Также обратите внимание, что функция *sorted* возвращает список, но с помощью *tuple* привели результат сортировки к кортежу.

## 9.5 Преобразование кортежа в список и обратно

На базе кортежа можно создать список, верно и обратное утверждение. И если это уже список, можно без каких-либо трудностей использовать методы, присущие им. После всех выполненных операций, можно также свободно список преобразовать его в кортеж.

Преобразование кортежа в список осуществляется функцией *list*.

```
a = (1, 4, 6, 7, [10, 20, 30])
a=list(a)
print(a)
```

Результат: [1, 4, 6, 7, [10, 20, 30]].

Преобразование списка в кортеж осуществляется функцией *tuple*.

```
a = [1, 4, 6, 7, [10, 20, 30]]
a=tuple(a)
print(a)
```

Результат: (1, 4, 6, 7, [10, 20, 30]).

Преобразование кортежа в список и работа с его элементами как элементами списка, а затем – обратное преобразование в кортеж.

```
a = (1, 4, 2, 7, [20, 10, 30])
a=list(a)
print(a)
a.append(100)
print(a)
a[4].reverse()
a.reverse()
print(a)
a=tuple(a)
print(a)
```

Результат:

[1, 4, 2, 7, [20, 10, 30]]

---

```
[1, 4, 2, 7, [20, 10, 30], 100]
[100, [30, 10, 20], 7, 2, 4, 1]
(100, [30, 10, 20], 7, 2, 4, 1)
```

## 9.6 Преобразование в словарь

Словарь – это еще одна структура, используемая в Python. Он, как и список, является изменяемым, но при этом неупорядоченным. Это значит, что обратиться к определенному элементу посредством указания индекса не получится. Чтобы лучше понять, можно провести аналогию с обычным англо-русским словарем. В нем для каждого слова есть перевод: house – дом, flat – квартира, window – окно. Если перенести такую структуру в программный код, то получится такая запись, оформляемая фигурными скобками:

```
{ 'house': 'дом', 'flat': 'квартира', 'window': 'окно' }
```

Последовательность пар при выводе на экран не определяется никаким правилом, отображаются они в произвольном порядке.

Для того, чтобы в Python преобразовать кортеж в словарь, необходимо использовать приведение типов с помощью *dict*.

```
A= (('a', 2), ('b', 4))
a= dict(a)
print(a)
```

Результат: {'a': 2, 'b': 4}.

Для создания словаря понадобился кортеж кортежей. Причем вложенные кортежи состоят из двух элементов каждый. Иначе не получается провести преобразование к словарю.

## 9.7 Преобразование кортежей в одну строку

Чтобы вывести в Python кортеж в одну строку, используется функция *join*.

```
a= (one, two, three)
b= ' '.join(a)
```



---

```
c= '\, \. join (a)
print(b)
print(c)
```

Результат:

onetwothree

one, two, three.

### ***Контрольные вопросы***

1. Понятие кортежа в Python.
2. Применение кортежей в Python.
3. Создание кортежей в Python.
4. Доступ к элементам кортежа в Python.
5. Удаление кортежей в Python.
6. Преобразование кортежа в список.
7. Преобразование списка в кортеж.
8. Перебор кортежей.
9. Встроенные методы кортежей.
10. Свойства кортежей.

### ***Задачи для самостоятельного решения***

1. Дан кортеж  $a = (53, 65, 1, 92)$ . Найти сумму элементов кортежа, занимаемый объем и дублировать его в число, кратное 17.
2. Дан кортеж  $n = (13, 29, 432, 98, 1, 0)$ . Отсортируйте имеющиеся элементы кортежа по возрастанию. Найдите занимаемый объем и длину.
3. Даны кортежи  $r = (1765, 986, 5, 36)$  и  $n = (2, 65, 333, 46)$ . Объедините в один кортеж. Найдите максимальное и минимальное значения в каждом кортеже.
4. Даны два кортежа. Найдите все числа, которые входят как в первый, так и во второй кортеж и выведите их в порядке возрастания.

---

5. Дан кортеж, который может содержать до 100000 чисел каждый. Определите, сколько в нем встречается различных чисел.

6. Создайте пустой кортеж и кортеж, состоящий из 10 элементов. Переместите элементы из второго кортежа в пустой и найдите сумму этих элементов.

7. Дан кортеж  $an = (12, 49, 369, Python, 38)$ . Удалите из него слово. Преобразуйте в словарь.

8. Создайте кортеж, элементы которого являются степенями числа 8 и который состоит из 7 элементов. Преобразуйте кортеж в список. Посчитайте занимаемый объем кортежа и списка.

9. Дан список кортежей  $grades = [("Ann4), ("Bod 2), ("Elizabeth 3), ("Dan 5)]$ . Вывести на экран последовательность сообщений вида *Hello, Ann! Your grade is 4.*

10. Создайте кортеж, который будет представлять страны и состоять из кортежей, каждый из которых — отдельная страна.

---

## 10 СЛОВАРИ

*Словарем* в языке программирования Python называется неупорядоченный набор данных произвольного типа с доступом по ключу. Их иногда ещё называют *ассоциативными массивами* или *хеш-таблицами*.

«Неупорядоченный» значит, что последовательность расположения пар не важна. Язык программирования ее не учитывает, поэтому обращение к элементам по индексам невозможно.

Элементами такой коллекции выступают пары объектов, каждая из которых включает в себя *ключ* и *значение*.

Если вспомнить такую структуру, как список, то доступ к его элементам осуществляется по индексу, который представляет собой целое неотрицательное число, причем разработчик алгоритма непосредственно не участвует в его создании (индекса). В словаре аналогом индекса является ключ, при этом ответственность за его формирование ложится на программиста.

Ключами словаря могут являться только объекты, поддерживающие хеширование (то есть преобразование по определённому алгоритму входного массива данных произвольной длины в выходную битовую строку фиксированной длины). Таким образом, использовать в качестве ключей списки, словари и другие изменяемые типы не получится.

Если в словарь будет добавлено несколько значений с одним и тем же ключом, словарь сохранит последнее.

Значением словаря может быть любой тип данных. Значения словарей вполне могут быть структурами, например, другими словарями или списками. Для работы со словарями доступны функции, меняющие их содержимое и выполняющие различные операции над ними. Его можно конвертировать в другие типы данных, например, в строку.

Чтобы более ясно понять, что же представляет собой ключ и значение в словаре, рассмотрим пример:

```
a = {'name': "Milla", 'age': "15", 'city': "Moscow"}  
print(a)
```

Результат: {'name': 'Milla', 'age': '15', 'city': 'Moscow'}.

Словарь выступает как контейнер, в котором собрано много различных коробок. Каждая коробка подписана и имеет свое содержимое. В нашем случае это коробка *name*, *age*, *city* – это и будет являться ключами, а значения либо содержание этих коробок – *Milla*, *15*, *Moscow*. Если объяснить более научным языком, то значение *Milla* привязано к ключу *name*, и чтобы получить доступ к этому значению, мы должны использовать ключ.

Таблица 13

### *Методы и функции для работы со словарями в Python*

| Название | Назначение                                   |
|----------|--|
| 1        | 2  |
| update   | Объединение содержимого двух словарей в один |
| len      | Получение размера                            |
| items    | Возвращает пары (ключи и значения)           |
| keys     | Возвращает ключи                             |
| values   | Возвращает значения                          |
| copy     | Копирует содержимое в другой словарь         |
| clear    | Полная очистка всех элементов                |
| eval     | Конвертация строки в словарь                 |
| dict     | Создание словаря                             |
| deepcopy | Создает полную копию словаря                 |
| fromkeys | Создание словаря                             |
| get      | Получить значение по ключу                   |
| has_key  | Проверка значения по ключу                   |

| 1                      | 2                              |
|------------------------|--------------------------------|
| <code>iteritems</code> | Возвращает итератор            |
| <code>iterkeys</code>  | Возвращает итератор ключей     |
| <code>pop</code>       | Извлекает значение по ключу    |
| <code>del</code>       | Оператор удаляет пару по ключу |

## 10.1 Создание

Перед тем как начать работу со словарем, его нужно создать. Сделать это можно базовыми средствами языка, присвоив свободной переменной произвольное количество пар объектов. Элементы необходимо поместить в фигурные скобки, а между ключом и значением должен стоять символ двоеточия. Следующий пример демонстрирует создание словаря под именем *a*, который включает в себя ключи в виде чисел и значения в виде строк.

```
a = {1: "one", 2: "two", 3: "three"}
print(a)
```

Результат: {1: 'one', 2: 'two', 3: 'three'}.

Вывести содержимое словаря можно стандартной функцией *print*, указав для нее в качестве аргумента нужный набор данных. Для заполнения словаря также используется метод *dict*, получающий произвольное количество пар ключей и значений. В таком случае быть ключом может только строка, как это показано в следующем примере кода.

```
a = dict(one = 1, two = 2, three = 3)
print(a)
```

Результат: {'one': 1, 'two': 2, 'three': 3}.

Как и в прошлом примере, функция *print* отображает содержимое словаря *a*. В данном случае имеется пара объектов, представленных также в виде чисел и строк.

---

Так же можно создать словарь с помощью `fromkeys()` – создает словарь по списку ключей с пустыми значениями:

```
D = {}.fromkeys(['name', 'age'], 123)
```

Результат: `{'name': 123, 'age': 123}`.

И с помощью конструктора:

```
d = dict((x, x**2) for x in range(5))
```

Результат: `{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}`.

## 10.2 Добавление элемента

В Python 3 содержимое словаря можно в любой момент изменить по своему усмотрению. К примеру, для того чтобы внести в коллекцию новую пару объектов, необходимо всего лишь указать новый ключ в квадратных скобках, а также соответствующее ему значение.

```
a = {1: "one", 2: "two", 3: "three"}
```

```
a[4] = "four"
```

```
print(a)
```

Результат: `{1: 'one', 2: 'two', 3: 'three', 4: 'four'}`

В приведенном выше коде применяется оператор присваивания, благодаря чему новая пара (4: “four”) помещается в конец уже созданной ранее коллекции *a*.

## 10.3 Объединение словарей

В том случае, если возникла необходимость в перемещении данных из одного словаря в другой, стоит воспользоваться функцией объединения *update*. Вызвать ее нужно на объекте, который предполагается расширить новыми парами ключей и значений.

```
a = {1: "one", 2: "two", 3: "three"}
```

```
b = {4: "four", 5: "five"}
```

```
a.update(b)
```

```
print(a)
```

Результат: `{1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five'}`.

---

Результатом работы метода *print* станет вывод на экран обновленного содержимого словаря под названием *a*.

После объединения новые элементы были автоматически записаны в конец коллекции.

## 10.4 Удаление элемента

Если словарь содержит лишнюю информацию, от нее можно избавиться при помощи специальной операции *del*. Для ее выполнения необходимо указать имя коллекции, а также ключ в квадратных скобках.

```
a = {1: "one", 2: "two", 3: "three"}
del a[3]
print(a)
```

Результат: {1: 'one', 2: 'two'}.

Так как операция получила ключ 3, в результате ее работы удалилось и значение *three*.

## 10.5 Получение размера

Функция *len* позволяет в любой момент определить текущее количество элементов словаря, если передать ей в качестве аргумента имя коллекции. В приведенном ниже примере метод *print* осуществляет вывод на экран размерность словаря *a*.

```
a = {1: "one", 2: "two", 3: "three"}
print(len(a))
```

Результат: 3.

Стоит заметить, что функция *len* возвращает точное количество пар, но не объектов. В этом случае имеется словарь, который содержит в себе ровно 3 пары.

---

## 10.6 Перебор словаря

Произвести перебор элементов словаря можно несколькими способами, в зависимости от желаемой для получения информации о его содержимом.

Перебор элементов можно осуществить с целью получения для последующей обработки:

1. пар ключ – значение;
2. перебор всех ключей;
3. перебор значений.

В данном примере показывается, как вывести на экран все пары этой коллекции в формате ключ: значение. Для этого используется цикл *for* и функция *items*, работающая с элементами словаря.

```
a = {1: "one", 2: "two", 3: "three"}
for key, value in a.items():
    print (key, ":", value)
```

Результат:

```
1: one
2: two
3: three.
```

Чтобы получить только ключи, следует применить метод *keys*, вызывая его на словаре.

```
a = {1: "one", 2: "two", 3: "three"}
for key in a.keys():
    print(key)
```

Результат:

```
1
2
3.
```

Аналогичным образом нужно поступить, чтобы вывести только значения словаря. Однако в таком случае в цикле *for* используется метод *values*.

```
a = {1: "one", 2: "two", 3: "three"}
```



---

```
for val in a.values():  
    print(val)
```

Результат:

one

two

three.

В обоих случаях отображается только выбранная часть пары, ключ или значение.

## 10.7 Поиск

Проверить наличие определенного ключа можно при помощи операции *in*. Для этого достаточно вывести результат ее выполнения для словаря по имени *a*.

```
a = {1: "one", 2: "two", 3: "three"}  
print (2 in a)  
print (4 in a)
```

Результат:

True

False.

Как можно заметить, проверка ключа 2 дала положительный результат (*True*). Во втором случае вывелось значение *False*, поскольку ключа 4 в словаре не обнаружено.

## 10.8 Сортировка

Средства языка дают возможность проводить в Python сортировку словаря по ключам и значениям, в зависимости от необходимости. В следующем примере имеется коллекция данных по имени *a*, в которой содержится информация в произвольном порядке. Ключами здесь выступают числа, а значениями являются строки. Сортировка осуществляется за счет импортированного модуля *operator* и встроенного метода *itemgetter*, получающего 0 или 1.

---

```
import operator
a = {2 : "two", 3 : "three", 1 : "one"}
b = sorted(a.items(), key = operator.itemgetter(0))
print(b)
b = sorted(a.items(), key = operator.itemgetter(1))
print(b)
```

Результат:

```
[(1, 'one'), (2, 'two'), (3, 'three')]
[(1, 'one'), (3, 'three'), (2, 'two')].
```

Как можно заметить, аргумент 0 позволяет отсортировать словарь по ключу, в то время как 1 дает возможность вывести его содержимое в алфавитном порядке значений.

## 10.9 Сравнение

Иногда нужно удостовериться, что два словаря содержат абсолютно одинаковые данные, либо узнать, какая коллекция больше или меньше по размеру. В этом случае на помощь приходит метод *cmp*, получающий в качестве параметров два словаря.

```
a = {1: "one", 2: "two", 3: "three"}
b = {4: "four", 5: "five"}
c = {1: "one", 2: "two", 3: "three"}
print(cmp(a, b))
print(cmp(b, c))
print(cmp(a, c))
```

Результат:

```
1
-1
0.
```

Приведенный код продемонстрировал выполнение метода *cmp* с тремя комбинациями аргументов. Как видно из результатов выдачи, функция возвращает 1, если первый больше второго; -1, если наоборот; и 0, когда данные полностью идентичны.

---

## 10.10 Копирование

Метод *copy* используется для копирования содержимого одного словаря в другой. Данный пример демонстрирует перенос ключей и значений из коллекции *a* в *b*.

```
a = {1: "one", 2: "two", 3: "three"}
b = a.copy()
print(b)
```

Результат: {1: 'one', 2: 'two', 3: 'three'}.

Как можно заметить, порядок и содержимое всех пар было сохранено в новом наборе.

## 10.11 Очистка

Чтобы избавиться от всех элементов словаря, стоит вызвать для него функцию *clear*.

```
a = {1: "one", 2: "two", 3: "three"}
a.clear()
print(a)
```

Результат: {}.

В результате получается абсолютно пустой набор данных.

## 10.12 Генератор словарей

*Генераторы словарей* – однострочные выражения, возвращающие словарь. Как и с другими наборами данных, производить заполнение словарей можно при помощи генераторов. В следующем примере демонстрируется создание числовых пар коллекции с использованием генератора словарей Python с методом *range*, получающим в качестве аргумента 5.

```
a = {a: a * a for a in range(5)}
print(a)
```

Результат: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}.

---

Таким образом, на выходе получается словарь *a*, включающий в себя ровно 5 пар. Ключами являются числа от 0 до 4, а значениями выступают их математические квадраты.

### 10.13 Конвертация в строку

Словарь можно очень легко преобразовать в строку для более удобной работы с цельным представлением его содержимого. Чтобы сделать это, потребуется функция *str*. Как можно видеть из результатов выполнения метода *type*, конвертация прошла успешно.

```
a = {1: "one", 2: "two", 3: "three"}
b = str(a)
print(b)
print(type(b))
```

Результат:

```
{1: 'one', 2: 'two', 3: 'three'}
<class 'str'>.
```

Аналогичным образом происходит обратное преобразование строки Python в словарь. Важно, чтобы ее текстовое содержимое подходило по структуре под рассматриваемую коллекцию.

```
a = '{1: "one", 2: "two", 3: "three"}'
b = eval(a)
print(b)
print(type(b))
```

Результат:

```
{1: 'one', 2: 'two', 3: 'three'}
<class 'dict'>.
```

Как видно из примера, метод *eval* конвертирует весь текст строки в новый словарь.

---

## 10.14 Вложенные словари

В Python словари могут быть вложенными, то есть выступать частью другого, более крупного словаря. При помощи фигурных скобок и двоеточий можно обозначить границы этого набора данных и указать программе пары ключей со значениями.

```
a = {  
    "First": {  
        1: "one",  
        2: "two",  
        3: "three"  
    },  
    "Second": {  
        4: "four",  
        5: "five"  
    }  
}  
print(a)
```

Результат: {'First': {1: 'one', 2: 'two', 3: 'three'}, 'Second': {4: 'four', 5: 'five'}}.

В примере, описанном выше, создается словарь *a*, включающий в себя два других словаря (*First* и *Second*). Те, в свою очередь, содержат несколько пар ключей и значений.

### ***Контрольные вопросы***

1. Словари. Способы создания словаря.
2. Методы и функции словарей.
3. Добавление элемента в словарь.
4. Объединение словарей.
5. Функция `len`. Перебор словаря.
6. Поиск при помощи операции `in`.
7. Сортировка словаря по ключам и значениям.

- 
8. Сравнение и очистка словарей.
  9. Генератор словарей.
  10. Конвертация в строку. Вложенные словари.

### *Задачи для самостоятельного решения*

1. Дана база данных о продажах некоторого интернет-магазина. Каждая строка входного файла представляет собой запись вида Покупатель товар количество, где Покупатель – имя покупателя (строка без пробелов), товар – название товара (строка без пробелов), количество – количество приобретенных единиц товара. Создайте список всех покупателей, а для каждого покупателя подсчитайте количество приобретенных им единиц каждого вида товаров.

2. Дан список стран и городов каждой страны. Затем даны названия городов. Для каждого города укажите, в какой стране он находится.

3. Дан текст: в первой строке записано количество строк в тексте, а затем сами строки. Выведите все слова, встречающиеся в тексте, по одному на каждую строку. Слова должны быть отсортированы по убыванию их количества появления в тексте, а при одинаковой частоте появления – в лексикографическом порядке.

4. В файловую систему одного суперкомпьютера проник вирус, который сломал контроль за правами доступа к файлам. Для каждого файла известно, с какими действиями можно к нему обращаться: запись W, чтение R, запуск X. В первой строке содержится число N – количество файлов, содержащихся в данной файловой системе. В следующих N строчках содержатся имена файлов и допустимых с ними операций, разделенные пробелами. Далее указано число M – количество запросов к файлам. В последних M строках указан запрос вида Операция Файл. К одному и тому же файлу может быть применено любое количество запросов. Вам требуется восстановить контроль над правами доступа к файлам (ваша программа для каждого

---

запроса должна будет возвращать ОК если над файлом выполняется допустимая операция, или же Access denied, если операция недопустима.

5. Дан текст: в первой строке задано число строк, далее идут сами строки. Выведите слово, которое в этом тексте встречается чаще всего. Если таких слов несколько, выведите то, которое меньше в лексикографическом порядке.

6. Вам дан словарь, состоящий из пар слов. Каждое слово является синонимом к парному ему слову. Все слова в словаре различны. Для слова из словаря, записанного в последней строке, определите его синоним.

7. В единственной строке записан текст. Для каждого слова из данного текста подсчитайте, сколько раз оно встречалось в этом тексте ранее. Словом считается последовательность непробельных символов идущих подряд, слова разделены одним или большим числом пробелов или символами конца строки.

8. У исполнителя «Удвоитель» две команды, которым присвоены номера: 1 – прибавь 1, 2 – умножь на 2. Выполняя первую из них, Удвоитель прибавляет к числу на экране единицу, а выполняя вторую, умножает его на два. Необходимо написать алгоритм, который запрашивает у пользователя начальное число и конечное число, а затем по алгоритму находит минимальное число операций, которые нужно проделать для решения этой задачи.

9. Дан англо-латинский словарь. Сделайте из него латино-английский.

10. Проверьте наличие в базе данных телефона по имени.

---

## 11 МНОЖЕСТВА

*Множество* в языке Python – это структура данных, которая эквивалентна множествам в математике. Множество может состоять из различных элементов. Во множестве можно перебирать, удалять и добавлять элементы множества, проверять принадлежность элемента. Кроме того, можно также выполнять операции над множествами, а именно: пересечение, объединение и разность. Так как элементы не индексируются, множества не поддерживают никаких операций среза и индексирования.

Элементы множества хранятся при помощи алгоритмов. К примеру, в отличие от массивов, где элементы хранятся в виде последовательного списка. Это позволяет выполнять некоторые операции быстрее. Также множество не может содержать дубликаты элементов.

Изменяемые типы данных не могут быть элементами множества. Элементами множества могут быть только числа, строки, кортежи.

### 11.1 Создание множества

Множество можно создать путем передачи всех элементов множества внутри фигурных скобок `{}` и разделить элементы при помощи запятых. Множество может содержать любые типы и содержать разное количество элементов. Для создания множества с несколькими элементами нужно отделить их друг от друга запятыми и поместить внутрь фигурных скобок.

```
num_set={1,2,3,4,5,6}  
print(num_set)
```

Результат: `{1, 2, 3, 4, 5, 6}`.

Также множество можно создавать из списка. Стоит отметить, что элементы множества находятся в произвольном порядке. Если за-



---

пустить один и тот же код несколько раз, можно получить разный порядок.

Можно создать и пустое множество. Для создания пустого множества необходимо вызвать `set()` без аргументов. Невозможно создать пустое множество с помощью двух фигурных скобок. Фигурные скобки создают пустой словарь, а не множество.

```
x=set()  
print(x,type(x))
```

Результат: `set() <class 'set'>`.

## 11.2 Изменение множества

Существует несколько способов добавить элементы в существующее множество: метод `add()` и метод `update()`.

`Add()` принимает один аргумент, который может быть любого типа, и добавляет данное значение в множество. Невозможно добавить значение, которое уже присутствует в множестве. Если попытаться это сделать, то ничего не произойдет, и это не приведет к возникновению ошибки.

```
mon=set(["Jan","Feb","Mar"])  
mon.add("Jan")  
mon.add("Apr")  
print(mon)
```

Результат: `{'Mar', 'Apr', 'Feb', 'Jan'}`.

`Update()` принимает один аргумент – множество – и добавляет все его элементы к исходному множеству. Как уже говорилось, множество не может содержать дубликаты, а следовательно, повторяющиеся значения игнорируются.

```
mon=set(["Jan","Feb","Mar"])  
mon.update(["Apr"])  
mon.update(["Jan"])  
print(mon)
```

Результат: `{'Mar', 'Apr', 'Feb', 'Jan'}`.

---

### 11.3 Удаление элементов множества

Python даёт нам возможность удалять элемент из множества, но не используя индекс, так как множество элементов не индексированы. Существует несколько способов удаления значений из множества. Первые два, *discard()* и *remove()*, немного различаются.

Метод *discard()* принимает в качестве аргумента одиночное значение и удаляет это значение из множества. Метод *remove()* также принимает в качестве аргумента одиночное значение и также удаляет его из множества. Если при *discard()*, придав элементу значение, которого нет в множестве, произойдет нулевое действие, то при *remove()* в результате программа выдаст `KeyError` (исключение, возникающее при обращении к элементу по ключу, отсутствующему в отображении).

Кроме вышеперечисленных методов, множества имеют метод *pop()*. *Pop()* удаляет один элемент из множества и возвращает его значение. Главная особенность этого метода состоит в том, что удаляется произвольный элемент. Невозможно проконтролировать, какое значение было удалено.

```
num_set={1,2,3,4,5,6}
print(num_set)
num_set.discard(3)
print(num_set)
num_set.remove(6)
print(num_set)
num_set.pop()
print(num_set)
```

Результат:

```
{1, 2, 3, 4, 5, 6}
{1, 2, 4, 5, 6}
{1, 2, 4, 5}
{2, 4, 5}.
```

---

## 11.4 Основные операции с множествами

Тип *set* в Python поддерживает несколько основных операций над множествами.

Метод *union()* (объединение) возвращает новое множество, содержащее все элементы каждого из множеств. Допустим, имеется два множества *A* и *B*. *Объединение* этих множеств – это множество со всеми элементами обоих множеств. Объединение может состоять из нескольких (более двух) множеств, и все их элементы сложатся в одно большое множество.

Метод *intersection()* (пересечение) возвращает новое множество, содержащее все элементы, которые есть и в первом множестве, и во втором. К примеру, существуют два множества: *A* и *B*. Их пересечение представляет собой множество элементов, которые являются общими для *A* и для *B*.

Метод *difference()* (разность) возвращает новое множество, содержащее все элементы, которые есть в множестве *A*, но которых нет в множестве *B*. К примеру, существуют два множества: *A* и *B*. Разница между *A* и *B* – это множество со всеми элементами, содержащимися в *A*, но не в *B*.

Метод *symmetric\_difference()* (симметрическая разность) возвращает новое множество, которое содержит только уникальные элементы обоих множеств. Симметричная разница между множествами *A* и *B* – это множество с элементами, которые находятся в *A* и *B*, за исключением тех элементов, которые являются общими для обоих множеств.

```
x={1,2,3,4,8}
y={1,2,4,5,6,9}
z={1,5,7,8,9}
S=x.union(y,z)
P=x.intersection(y)
R=y.difference(x)
SR=x.symmetric_difference(z)
```

---

```
print(S)
print(P)
print(R)
print(SR)
```

Результат:

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
{1, 2, 4}
{9, 5, 6}
{2, 3, 4, 5, 7, 9}.
```

## 11.5 Множество в логическом контексте

Существует возможность сравнить множества в зависимости от того, какие элементы в них содержатся. Можно сказать, является ли множество родительским или дочерним от другого множества. Результат такого сравнения будет либо True, либо False.

```
x={1, 2, 4}
y={1, 2, 4, 5, 6, 9}
z={1, 2, 4}
a=x<=y
b=x>=y
c=x!=z
d=x==z
print(a)
print(b)
print(c)
print(d)
```

Результат:

```
True
False
False
True.
```

В логическом контексте:

1. Пустое множество – ложь.

---

2. Любое множество, содержащее хотя бы один элемент – истина.

Значения элементов не важны.

## 11.6 Метод *copy*

Метод *copy()* возвращает копию множества. Операция присваивания не копирует объект, а создаёт ссылку на объект. Данный модуль предоставляет общие (поверхностная и глубокая) операции копирования.

```
str_set={"Alexey", "Galina", "Tatyana"}
x=str_set.copy()
print(x)
```

Результат: {'Alexey', 'Galina', 'Tatyana'}.

Для операции глубокого копирования часто возникают две проблемы, которых нет у операции поверхностного копирования.

Рекурсивные объекты (составные объекты, которые явно или неявно содержат ссылки на себя) могут стать причиной рекурсивного цикла. Поскольку глубокая копия копирует всё, она может скопировать слишком много, например, административные структуры данных, которые должны быть разделяемы даже между копиями.

## 11.7 *Frozenset*

*Frozenset*, или *замороженное множество*, – это класс с характеристиками множества, однако, как только элементы становятся назначенными, их нельзя менять.

```
x=frozenset([1, 2, 3, 4, 5, 6])
y=frozenset([4, 5, 6, 7, 8, 9])
print(x)
print(y)
```

Результат:

`frozenset({1, 2, 3, 4, 5, 6})`

---

`frozenset({4, 5, 6, 7, 8, 9})`.

Единственное отличие *set* от *frozenset* заключается в том, что *set* – изменяемый тип данных, а *frozenset* – нет.

### ***Контрольные вопросы***

1. Назовите ряд особенностей, определяющих множество.
2. Назовите главное отличие множества от массивов. В каком виде хранятся элементы множества?
3. Какие типы данных могут быть элементами множества, а какие нет?
4. Могут ли во множестве существовать дубликаты элементов?
5. Назовите различия между `discard()` и `remove()`.
6. Назовите главную особенность метода `pop()`.
7. Назовите основные операции над множествами.
8. Существует ли возможность сравнения множеств?
9. Изменяемы ли элементы множества?
10. Изменяемо ли само множество?

### ***Задачи для самостоятельного решения***

1. Каждая семья, живущая в доме N, выписывает газету, или журнал, или и то, и другое. 75 семей выписывают газету, 27 – журнал, 13 - и журнал, и газету. Сколько семей живёт в доме N?
2. Каждая семья, живущая в доме N, выписывает газету, или журнал, или и то, и другое. 116 семей выписывают газету, 37 – журнал, 29 - и журнал, и газету. Сколько семей живёт в доме N?
3. Дан список чисел, который может содержать до 100000 чисел каждый. Определите, сколько в нем встречается различных чисел.
4. Дан список чисел, который может содержать до 9000 чисел каждый. Определите, сколько в нем встречается различных чисел.

---

5. Даны два списка чисел. Найдите все числа, которые входят как в первый, так и во второй список, и выведите их в порядке возрастания.

6. Из 52 школьников 23 собирают значки, 35 – марки, 16 – и значки, и марки. Сколько школьников не увлекаются коллекционированием?

7. Из 89 школьников 33 собирают значки, 25 – марки, 6 – и значки, и марки. Сколько школьников не увлекаются коллекционированием?

8. В воскресенье 19 учеников нашего класса побывали в планетарии, 10 – в цирке и 6 – на стадионе. Планетарий и цирк посетили 5 человек, планетарий и стадион – 3, цирк и стадион – 1. Сколько учеников в классе, если никто не успел посетить все три места, а 3 ученика болели и в выходные были дома?

9. На уроке литературы учитель решил узнать, кто из 40 учеников читал книги А, В и С. Результаты опросы выглядят так: А – 25 уч., В – 22 уч., С – 22 уч.; одну из книг А или В – 33 уч., А или С – 32 уч., В или С – 31 уч., все три книги прочитали 10 учеников. Сколько учеников прочитали по одной книге? Сколько учеников прочитали ровно 2 книги? Сколько учеников не прочитали ни одной книги?

10. Каждый из  $N$  школьников некоторой школы знает  $M_i$  языков. Определите, какие языки знают все школьники и языки, которые знает хотя бы один из школьников. Первая строка входных данных содержит количество школьников  $N$ . Далее идет  $N$  чисел  $M_i$ , после каждого из чисел идет  $M_i$  строк, содержащих названия языков, которые знает  $i$ -й школьник. Длина названий языков не превышает 1000 символов, количество различных языков не более 1000.  $1 \leq N \leq 1000$ ,  $1 \leq M_i \leq 500$ .

---

## 12 ФУНКЦИИ

По своей сути *функции* в Python практически ничем не отличаются от функций из других языков программирования. *Функцией* называют именованный фрагмент программного кода, к которому можно обратиться из другого места программы (но есть *lambda-функции*, у которых нет имени). Как правило, функции создаются для работы с данными, которые передаются ей в качестве аргументов, также функция может формировать некоторое возвращаемое значение.

### 12.1 Создание функций

Для создания функции используется ключевое слово *def*, после которого указывается имя и список аргументов в круглых скобках. Если их нет, то скобки остаются пустыми, но они обязательно должны быть. Далее идет двоеточие, обозначающее окончание заголовка функции. Тело функции выделяется так же, как тело условия (или цикла): четырьмя пробелами.

При вызове функции в скобках указывается нужное количество переменных или выражений, значения которых будут переданы функции в качестве параметров. Если у функции нет параметров, скобки (пустые) при вызове все равно нужны.

Таким образом, самая простая функция, которая ничего не делает, будет выглядеть так:

```
def fun():  
    pas
```

Возврат значения функцией осуществляется с помощью ключевого слова *return*, после которого указывается возвращаемое значение.

```
def fun():  
    return 1
```



---

Результат:

```
>>> fun()
```

1.

## 12.2 Работа с функциями

Во многих случаях функции используют для обработки данных. Эти данные могут быть глобальными либо передаваться в функцию через аргументы. Список аргументов определяется на этапе реализации и указывается в круглых скобках после имени функции. Например, операцию сложения двух аргументов можно реализовать следующим образом:

```
def summa(a, b):  
    return a + b
```

Результат:

```
>>> summa(3, 4)
```

7.

В качестве примеров работы с функциями рассмотрим вычисление числа Фибоначчи с использованием рекурсии и вычисление факториала с использованием цикла.

Вычисление числа Фибоначчи:

```
def fibb(n):  
    if n==0:  
        return 0  
    elif n==1:  
        return 1  
    elif n==2:  
        return 1  
    else:  
        return fibb(n-1)+fibb(n-2)
```

Результат:

```
>>> fibb(10)
```

55.

Вычисление факториала:

---

```
def factorial(n):  
    prod = 1  
    for i in range(1, n+1):  
        prod *= i  
    return prod
```

Результат:

```
>>> factorial(5)  
120.
```

Функцию можно присвоить переменной и использовать ее, если необходимо сократить имя. В качестве примера можно привести вариант использования функции вычисления факториала из пакета *math*.

```
>>> import math  
>>> f = math.factorial  
>>> print(f(5))
```

Результат: 120.

## 12.3 lambda-функции

*Lambda-функция* – это безымянная функция с произвольным числом аргументов, вычисляющая одно выражение. Тело такой функции не может содержать более одной инструкции (или выражения). Данную функцию можно использовать в рамках каких-либо конвейерных вычислений (например внутри *filter()*, *map()* и *reduce()*) либо самостоятельно в тех местах, где требуется произвести вычисления, которые удобно «завернуть» в функцию.

```
>>> (lambda x: x**2)(5)
```

Результат: 25.

Lambda-функцию можно присвоить какой-либо переменной и в дальнейшем использовать ее в качестве имени функции.

```
>>> sqrt = lambda x: x**0.5  
>>> sqrt(25)
```

Результат: 5.0.

---

Списки можно обрабатывать *lambda*-функциями внутри таких функций, как *map()*, *filter()*, *reduce()*.

Функция *map* принимает два аргумента, первый – это функция, которая будет применена к каждому элементу списка, а второй – это список, который нужно обработать.

Вычисление куба каждого числа последовательности:

```
>>> l = [1, 2, 3, 4, 5, 6, 7]
>>> list(map(lambda x: x**3, l))
```

Результат: [1, 8, 27, 64, 125, 216, 343].

Функция *filter()* ожидает два аргумента, должна возвращать логическое значение. Первый аргумент вызывается для каждого элемента итерации, и фильтр возвращает только те элементы, для которых второй аргумент возвращает *true*.

Подобно функции *map*, функция *filter* также возвращает список элементов. В отличие от *map*, функция *filter* может принять только один итерируемый объект.

Выберем четные числа числовой последовательности:

```
a = [1, 2, 3, 4, 5, 6]
b=list(filter(lambda x : x % 2 == 0, a))
print(b)
```

Результат: [2, 4, 6].

Функция *reduce()* принимает два аргумента: функцию и последовательность. *reduce()* последовательно применяет функцию-аргумент к элементам списка, возвращает единичное значение. В Python 2.x функция *reduce* доступна как встроенная, в то время как в Python 3 она была перемещена в модуль *functools*.

Вычисление наибольшего элемента в списке при помощи *reduce*:

```
from functools import reduce
s = [-1, 24, 17, 19, 9, 32, 2, 54, 3]
mx = reduce(lambda a,b: a if (a > b) else b, s)
print (mx)
```

Результат: 54.

---

## 12.4 Аргументы функции в Python

Вызывая функцию, мы можем передавать ей следующие типы аргументов:

- обязательные аргументы (Required arguments);
- аргументы-ключевые слова (Keyword argument);
- аргументы по умолчанию (Default argument);
- аргументы произвольной длины (Variable-length arguments).

### *Обязательные аргументы функции*

Если при создании функции мы указали количество передаваемых ей аргументов и их порядок, то и вызывать ее мы должны с тем же количеством аргументов, заданных в нужном порядке.

```
def bigger(a,b):  
    if a > b:  
        print(a)  
    else:  
        print (b)
```

Результат:

```
>>> bigger(5,3)  
5.
```

Если функция `bigger` будет вызвана с другим количеством аргументов или отсутствием их, то получим сообщение об ошибке.

### *Аргументы – ключевые слова*

Аргументы – ключевые слова используются при вызове функции. Благодаря ключевым аргументам, можно задавать произвольный (то есть не такой каким он описан при создании функции) порядок аргументов.

```
def st(name, age):  
    print (name, "is", age, "years old")
```

Результат:

```
>>> st(age=20,name="Tatyana")  
Tatyana is 20 years old.
```

---

### *Аргументы, заданные по умолчанию*

Аргумент по умолчанию – это аргумент, значение для которого задано изначально, при создании функции.

```
def space(planet_name, center="Star") :  
    print (planet_name, "is orbiting a", center)
```

Результат:

```
>>> space("Mars")
```

Mars is orbiting a Star

```
>>> space("Mars","Luna")
```

Mars is orbiting a Luna.

### *Аргументы произвольной длины*

Иногда возникает ситуация, когда заранее не известно, какое количество аргументов будет необходимо принять функции. В этом случае следует использовать аргументы произвольной длины. Они задаются произвольным именем переменной, перед которой ставится звездочка (\*).

Функция может принимать и произвольное число именованных аргументов, тогда перед именем ставится \*\*.

Чаще всего при написании программ используются параметры *\*args* и *\*\*kwargs*.

Сами слова «args» и «kwargs» не так важны. Это просто сокращение. Их можно назвать и *\*lol* и *\*\*omg*, и они будут работать таким же образом. Главное здесь – это количество звездочек.

```
def f1(*args) :  
    for i in args:  
        print (i)  
def f2(**kwargs) :  
    return kwargs
```

Результат:

```
>>> f1("Hello","wordls")
```

Hello

wordls

```
>>> f1(1,2,3,4)
```

---

```
1
2
3
4
>>> f2(a=1,b=2,c=3)
{'a': 1, 'b': 2, 'c': 3}
>>> f2(a="python")
{'a': 'python'}.
```

## 12.5 Область видимости и глобальные переменные

Концепт области в Python такой же, как и в большей части языков программирования. *Область видимости* указывает на то, когда и где переменная может быть использована. Если определяются переменные внутри функции (локальные переменные), эти переменные могут быть использованы только внутри этой функции. Когда функция заканчивается, их можно больше не использовать, так как они находятся вне области видимости.

```
def f_a():
    a = 1
    b = 2
    return a+b
def f_b():
    c = 3
    return a+c
print(f_a())
print(f_b())
```

Результат: `NameError: name 'a' is not defined`.

Это вызвано тем, что переменная определена только внутри первой функции (*f\_a*, то есть является локальной), но не во второй (*f\_b*). Можно обойти этот момент, указав в Python, что переменная *a* – глобальная (*global* – видна во всех частях программы).

```
def f_a():
    global a
```

---

```
a = 1
b = 2
return a+b
def f_b():
    c = 3
    return a+c
print(f_a())
print(f_b())
```

Результат:

3

4.

## 12.6 Математические функции в Python

При работе с математическими функциями в Python необходимо импортировать библиотеку *math*. Это можно сделать, выполнив следующую команду:

```
from math import *
```

*ceil(x)* – возвращает округленное  $x$  как ближайшее целое значение типа *int*, большее или равное  $x$  (округление «вверх»).

*fabs(x)* – возвращает абсолютное значение (модуль) числа  $x$ . В Python есть встроенная функция *abs*, но она возвращает модуль числа с тем же типом, что число, здесь же всегда *float abs (fabs)*.

*factorial(x)* – возвращает факториал целого числа  $x$ , если  $x$  не целое возбуждается исключение *ValueError*.

*floor(x)* – в противоположность *ceil(x)* возвращает округленное  $x$  как ближайшее целое значение типа *int*, меньшее или равное  $x$  (округление «вниз»).

*frexp(x)* – представляет число в экспоненциальной записи  $x=m*2^n$  и возвращает мантиссу  $m$  (действительное число, модуль которого лежит в интервале от 0.5 включительно до 1 не включительно) и порядок  $n$  (целое число) как пару чисел  $(m, n)$ . Если  $x = 0$ , то возвращает  $(0.0, 0)$ .

---

*fsum(iterable)* – возвращает *float* сумму от числовых элементов итерируемого объекта.

*isinf(x)* – проверяет, является ли *float* объект *x* плюс или минус бесконечностью, результат соответственно True или False.

*isnan(x)* – проверяет, является ли *float* объект *x* объектом *NaN* (not a number).

*ldexp(x, i)* – возвращает значение  $x \cdot 2^i$ , то есть осуществляет действие, обратное функции *frexp(x)*.

*modf(x)* – возвращает дробную и целую часть *float* числа. Оба результата сохраняют знак исходного числа *x* и представлены типом *float*.

*trunc(x)* – возвращает целую часть числа *x* в виде *int* объекта.

*exp(x)* – возвращает  $e^x$ .

*log(x[, base])* – при передаче функции одного аргумента *x* возвращает натуральный логарифм *x* (логарифм по основанию  $e = 2.7182\dots$ ). При передаче двух аргументов второй берется как основание логарифма.

*log10(x)* – возвращает десятичный логарифм *x*.

*pow(x, y)* – возвращает *x* в степени *y*. В отличие от операции **\*\*** приводит оба аргумента к типу *float*.

*sqrt(x)* – квадратный корень из *x*.

*acos(x)* – возвращает арккосинус *x*, в радианах.

*asin(x)* – возвращает арксинус *x*, в радианах.

*atan(x)* – возвращает арктангенс *x*, в радианах.

*atan2(y, x)* – возвращает *atan(y/x)*, в радианах. Результат лежит в интервале  $[-\pi, \pi]$ . Вектор, конец которого задается точкой (*x*, *y*), образует угол с положительным направлением оси *x*. Поэтому эта функция имеет более общее назначение, чем предыдущая. Например и *atan(1)*, и *atan2(1, 1)* дадут в результате  $\pi/4$ , но *atan2(-1, -1)* – это уже  $-3\pi/4$ .



---

*Cos(x)* – возвращает косинус  $x$ , где  $x$  выражен в радианах.

*hyp(x, y)* – возвращает  $\sqrt{x^2+y^2}$ . Удобно для вычисления гипотенузы (*hyp*) и длины вектора.

*Sin(x)* – возвращает синус  $x$ , где  $x$  выражен в радианах.

*tan(x)* – возвращает тангенс  $x$ , где  $x$  выражен в радианах.

*degrees(x)* – конвертирует значение угла  $x$  из радиан в градусы.

*radians(x)* – конвертирует значение угла  $x$  из градусов в радианы.

### ***Контрольные вопросы***

1. Что такое функция?
2. Способы создания функций в Python.
3. Пустая функция.
4. Оператор return.
5. Lambda-функция.
6. Обязательные аргументы (Required arguments).
7. Аргументы-ключевые слова (Keyword argument).
8. Аргументы по умолчанию (Default argument).
9. Аргументы произвольной длины (Variable-length arguments).
10. Область видимости переменных.

### ***Задачи для самостоятельного решения***

1. Написать функцию, принимающую 3 аргумента: первые два – числа, третий – операция, которая должна быть произведена над ними. Если третий аргумент +, сложить их; если -, то вычесть; \* - умножить; / - разделить (первое на второе). В остальных случаях вернуть строку «Неизвестная операция».

2. Написать функцию, принимающую 1 аргумент – год – и возвращающую True, если год високосный, и False иначе.

---

3. Написать функцию, принимающую 1 аргумент – сторону квадрата – и возвращающую 3 значения (с помощью кортежа): периметр квадрата, площадь квадрата и диагональ квадрата.

4. Написать функцию, принимающую 1 аргумент – номер месяца (от 1 до 12) – и возвращающую время года, которому этот месяц принадлежит (зима, весна, лето или осень).

5. Пользователь делает вклад в размере  $a$  рублей сроком на  $years$  лет под 10% годовых (каждый год размер его вклада увеличивается на 10%. Эти деньги прибавляются к сумме вклада, и на них в следующем году тоже будут проценты). Написать функцию, принимающую аргументы  $a$  и  $years$  и возвращающую сумму, которая будет на счету пользователя.

6. Написать функцию, принимающую 1 аргумент – число от 0 до 1000 – и возвращающую True, если оно простое, и False – иначе.

7. Написать функцию `date`, принимающую 3 аргумента – день, месяц и год. Вернуть True, если такая дата есть в нашем календаре, и False иначе.

8. Написать функцию, принимающая 2 аргумента: строку, которую нужно зашифровать, и ключ шифрования, который возвращает строку, зашифрованную путем применения функции XOR (^) над символами строки с ключом.

9. Написать также функцию, которая по зашифрованной строке и ключу восстанавливает исходную строку.

10. Написать функцию, принимающую 1 аргумент - сторону квадрата – и возвращающую 3 значения (с помощью кортежа): периметр квадрата, площадь квадрата и диагональ квадрата.

---

## Библиографический список

1. Гуриков, С. Р. Основы алгоритмизации и программирования на Python : учеб. пособие / С. Р. Гуриков. – М. : ФОРУМ : ИНФРА-М, 2017. – 343 с.
2. Зрюмова, А. Г. Информатика : учебное пособие / А. Г. Зрюмова, Е. А. Зрюмов, С. П. Пронин. – Барнаул : Изд-во АлтГТУ, 2011. – 177 с. – ISBN 978-5-7568-0843-8
3. Лутц, М. Изучаем Python : учебник / пер. с англ. / М. Лутц. – 4-е издание Спб. : Символ-Плюс, 2011. – 1280 с.
4. МакГрат, М. Программирование на PYTHON / Майк МакГрат. – М. : Эксмо, 2015. – 192 с.
5. Маккинли, У. Python и анализ данных / У. Маккинли. – М. : ДМК Пресс, 2015. – 482 с.
6. Мэтиз, Э. Изучаем Python. Программирование игр, визуализация данных, веб-приложения : учебник / пер. с англ. / Э. Мэтиз. – Спб. : Питер, 2017. – 496 с.
7. Ночка, Е. И. Основы алгоритмизации и программирования на Питон : учебник / Е. И. Ночка. – М. : КУРС, НИЦ ИНФРА-М, 2017. – 208 с.
8. Саммерфилд, М. Python на практике : создание качественных программ с использованием параллелизма, библиотек и паттернов / пер. с англ. / М. Саммерфилд. – М. : ДМК Пресс, 2014. – 337 с.
9. Чан, У. Дж. Python. Создание приложений / Уэсли Дж. Чан. – М. : Вильямс, 2016. – 808 с.

---

*Учебное издание*

**Галина Владимировна Зыкова,  
Алексей Сергеевич Попов,  
Татьяна Николаевна Сапуглецева**

**ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ PYTHON**

*Учебное пособие*

**Под редакцией Г.В. Зыковой**

Подписано в печать 01.04.2020

Электронное издание для распространения через Интернет.

ООО «ФЛИНТА», 117342, г. Москва, ул. Бутлерова, д. 17-Б, комн. 324.

Тел./факс: (495) 334-82-65; тел. (495) 336-03-11.

E-mail: [flinta@mail.ru](mailto:flinta@mail.ru); WebSite: [www.flinta.ru](http://www.flinta.ru)