МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный электротехнический
университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

ИНФОРМАТИКА

Методические указания к лабораторным работам

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2019

ТЕРМИНЫ И ПОЯСНЕНИЯ К ОБОЗНАЧЕНИЯМ

№	Обозначения	Пояснения
1	4 пробела или знак табуляции	*Отступы в языке Python* - особенность синтаксиса языка Python, в котором не используются фигурные скобки для обозначения отдельных фрагментов программы. Вместо скобок - отступы слева.
2	< >	Носят иллюстрирующий характер, используются для того чтобы показать место расположения параметра или аргумента, не являются частью синтаксиса языка Python 3
3	>>>	Специальное обозначение интерактивного режима, при котором можно вводить инструкции непосредственно в командной строке
4	[ ]	Необязательные элементы, например, аргументы функции
5	>?	Ожидание ввода данных в интерактивном режиме в Python консоли в среде PyCharm
6	...	Продолжение определения составной инструкции после нажатия Enter

Все примеры, изложенные в пособии, предназначены для выполнения в командной строке Linux. Как пользоваться командной строкой Linux, можно посмотреть в [5], [6].

ГЛАВА 1. УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ ЯЗЫКА PYTHON

Цель и задачи главы

Цель первой главы - изучить основные управляющие конструкции языка Python.

Задачи настоящей главы:

- дать общее представление о принципах программирования на языке Python;
- рассмотреть ввод, вывод информации средствами языка Python;
- рассмотреть в общем смысле концепцию объектно-ориентированного программирования (ООП), лежащую в основе языка Python;
- рассмотреть основные встроенные типы данных, операции над этими данными;

На момент написания данного учебного пособия существует 2 версии языка Python: Python 2 и Python 3, не обладающие обратной совместимостью, то есть программы, написанные на Python 2 нельзя запустить на Python 3 и наоборот. Далее в пособии при упоминании Python имеется в виду 3-я версия языка (3.5).

Для изучения примеров можно использовать как командную строку и предварительно установленный на компьютер Python-интерпретатор, онлайн-отладчики (например, OnlineGBD python 3), интегрированные среды разработки онлайн (такие как, например, Online Python compiler, Online Python IDE, and online Python REPL), так и интегрированную среду разработки PyCharm Community. Исходный код примеров, приведенных в настоящем методическом пособии, находится в git-репозитории. Инструкция по запуску примеров в среде PyCharm находится на wiki-странице git-репозитория.

1.1 Запуск программы

Python - это не только язык программирования, но и название интерпретатора. Интерпретатор - это специальная программа, которая исполняет другие программы. Таким образом, программы, которые вы будете писать на языке Python, будут выполняться с помощью интерпретатора. В данном разделе подробно рассматривается не то, как интерпретатор работает с исходным кодом, а как пользователь взаимодействует с интерпретатором.

В языке Python существует два способа запуска программ. Первый способ будет использоваться нами достаточно часто, он называется

интерактивный режим. Второй способ - это запуск программного кода с помощью интерпретатора.

Рассмотрим работу в интерактивном режиме. Если вы наберете имя интерпретатора (в примере далее - python) в консоли (также ее называют командная строка или терминал) Linux и нажмете Enter, увидите следующее:

```
user@user-desktop:~$ python
Python 2.7.12 (default, Nov 12 2018, 14:36:49)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Во второй строке вывода вы видите версию интерпретатора, в данном случае это Python 2.7.12. Вы можете вызвать любую другую версию, которая установлена на вашем компьютере, просто указав ее при запуске:

```
user@user-desktop:~$ python3.6
Python 3.6.8 (default, Dec 24 2018, 19:24:27)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

В качестве имени интерпретатора можно указать просто python3. В этом случае будет вызвана та версия интерпретатора, которая указана в системе для вызова python3.

Для того, чтобы завершить взаимодействие с интерпретатором, достаточно нажать клавиши *ctrl+D* или набрать команду *exit()*.

Обратите внимание на символы, которые находится в самом низу выведенного сообщения:

```
>>>
```

После этих символов вы можете набирать любые инструкции языка Python и сразу же видеть результат. Например:

```
>>> 2 + 2
4
```

При этом для того, чтобы вывести результат на экран, в интерактивном режиме вам не нужно указывать специальные инструкции, это произойдет автоматически.

Если вы используете интегрированную среду разработки IDE (Integrated Development Environment), например PyCharm, вы можете также воспользоваться интерактивным режимом с помощью следующего пункта меню приложения: Tools → Python Console.

Интерактивный режим удобен, когда нужно быстро протестировать небольшой фрагмент программного кода. Обратите внимание, что

инструкции будут выполняться одна за одной:

```
>>> 2 + 2
4
>>> 3 + 3
6
```

Такие инструкции, которые можно разместить в одной строке, называют *простыми*. В одной строке можно разместить несколько простых инструкций, разделив их символом “;” – однако такой стиль программирования не приветствуется.

Уже в этой главе вы узнаете про существование *составных* инструкций. Такие инструкции в языке Python состоят из нескольких строк и требуют наличие отступов. Таким образом, это инструкции, которые включают в себя другие инструкции. Их синтаксис можно описать следующим образом:

```
<заголовок составной инструкции>:
    <тело составной инструкции>
```

Заголовок составной инструкции заканчивается символом “:”; все внутренние инструкции должны начинаться с одинакового отступа: это особенности Python. Обычно используются 4 пробела.

К составным инструкциям относятся циклы, условный оператор, функции и др. Позже мы подробно будем разбирать каждую из них, сейчас важно подчеркнуть, что составная инструкция неделима: нельзя написать заголовок и пропустить тело, это вызовет ошибку.

В интерактивном режиме наравне с однострочными инструкциями вы можете использовать и многострочные:

```
>>> if 1 > 0:
...     print(1)
...
1
```

Здесь:

- *if 1 > 0* является заголовком составной инструкции,
- *print()* - функция для вывода в консоль (подробнее см. в разделе “1.42 Ввод и вывод данных в Python”).

После нажатия Enter вы увидите приглашение к вводу тела составной инструкции или ее завершения:

```
...
```

Если вы хотите ввести команды, которые относятся к заголовку, не забывайте про отступы, как показано у нас в примере:

```
...     print(1)
```

Они используются согласно синтаксису Python. Таких инструкций может быть сколько угодно и они могут быть сколь угодно вложенными.

Далее, после нажатия *Enter*, вы опять увидите многоточие:

```
...
```

Если вы хотите завершить свою составную инструкцию, нажмите *Enter*. Обратите внимание, что вы не можете сразу начать писать следующую инструкцию, которая не относится к составной, это вызовет ошибку:

```
>>> if 1 > 0:
...         print(1)
... print('Hello!')
  File "<stdin>", line 3
    print('Hello!')
        ^
SyntaxError: invalid syntax
```

Выше показан один из многих вариантов возникновения ошибок, которые подробнее рассматриваются в разделе “3.2.6 Исключения”.

В данном случае инструкция *print('Hello!')* не имела отношения к телу составной инструкции, это мы можем понять по отсутствию отступов в начале строки. Чтобы код выше не вызывал ошибок, нам следует дважды нажать клавишу Enter после завершения тела составной инструкции, например так:

```
>>> if 1 > 0:
...     print(1)
...
1
>>> print('Hello!')
Hello!
```

Второй способ запуска программ - это сохранить программный код в файле и выполнить его с помощью интерпретатора. Файл с программным кодом называется *модуль*. Модуль обычно имеет расширение *.py*. Для запуска модуля в терминале Linux запустите нужную версию интерпретатора и укажите модуль, который хотите запустить, например:

```
user@user-desktop:~$ python3.6 main.py
```

При этом результат уже не будет выведен автоматически, как в интерактивном режиме.

1.2 Процедурное программирование

Существует множество способов организации программного кода. Такие способы принято называть *парадигмами программирования*. Выделяют функциональное, логическое, процедурное и объектно-ориентированное

программирование. Мы будем рассматривать некоторые из них в “3.1 Парадигмы программирования”. *Процедурное программирование* - один из популярных стилей программирования у начинающих программистов, предполагающий использование *функций*.

Функция – это именованная часть программы, которую можно использовать повторно, обращаясь к ней по имени (вызывая функцию). Давайте подробнее разберем, что означает это определение.

Предположим, вы написали функцию, которая решает квадратное уравнение. В данный момент нас не интересует, каким образом происходит вычисление значений корней, но важно, что после *вызова* (т.е. использования) функции эти значения нам известны. Мы можем написать программу, которая будет вызывать эту функцию столько раз, сколько нам нужно и там, где нам нужно (т.е. вне функции).

Функция может вызывать сама себя, такие функции называются *рекурсивными*.

Функции можно вызывать (т.е. использовать) и определять. У функции могут быть *аргументы* (или параметры) - некоторые данные, которые передаются в функцию из вызывающей ее программы. Определение функции содержит *имя*, *названия ее аргументов* и *набор инструкций*, которые содержатся в функции (такой набор обычно называют *телом функции*). Для примера выше это может выглядеть так:

1. Имя: *решить_уравнение*
2. Параметры: *a*, *b*, *c* (коэффициенты квадратного уравнения).
3. Тело: вызов инструкции вычисления значения дискриминанта и вызов инструкции вычисления значений корней, например:

   вычислить_дискриминант: $D = b^2-4 \cdot a \cdot c$

   вычислить_корни_уравнения: $x_{1,2} = (-b \pm \sqrt{D})/(2 \cdot a)$

Таким образом, определение функции может выглядеть следующим образом:

решить_уравнение(a, b, c):

   вычислить_дискриминант: $D = b^2-4 \cdot a \cdot c$

   вычислить_корни_уравнения: $x_{1,2} = (-b \pm \sqrt{D})/(2 \cdot a)$

Запись, показывающая имя функции, количество и названия её аргументов называется *прототипом*. Прототип часто используется с целью показать синтаксис функции. Для примера выше прототип может выглядеть так:

решить_уравнение(a, b, c)
при этом вызов функции выглядит так:
решить_уравнение(1, 2, 3)

Таким образом, в прототипе указаны названия аргументов, а в вызове - конкретные значения аргументов.

Для удобства работы с различными значениями, можно использовать удобный способ хранения данных – использовать переменные. Переменная – это понятное человеку название (имя) для некоторой области памяти, где хранится определенное значение.

Рассмотрим небольшой пример. Допустим, сразу в нескольких местах программы необходимо использовать значение *5*. Предупреждая ситуацию, что со временем вместо *5* может понадобиться использовать какое-либо другое значение, создадим переменную:

```
a = 5
```

a - это название (имя), области памяти, где хранится значение *5*. При смене значения переменной *a*, например:

```
a = 3.14
```

во всех местах в программе, где используется переменная *a*, будет подставлено обновленное ее значение.

1.3 Объекты в Python

Функции - это один из способов организации программы, но не единственный. В функциях мы оперируем переменными, которые хранят некоторые значения.

На самом деле, все переменные в языке Python являются *объектами*. Чтобы лучше понять, что это значит, кратко рассмотрим несколько определений.

1) Объекты

Объект - конкретная сущность некоторой предметной области. Например, рассмотрим предметную область - планеты, в которой есть знакомые нам *объекты*: Земля, Венера, Юпитер и др. В данном случае слово *планеты* обозначает некоторую абстракцию (обобщение), речь о которой пойдет в следующем пункте.

2) Классы

Класс - это тип объекта. Например, можно рассматривать в качестве класса планеты. При этом конкретные примеры планет: Марс, Меркурий и т.д. - будут объектами.

Класс описывает общее поведение: общие черты, свойства и характеристики, а также общие действия, функции, которые можно выполнять над объектами класса. Об этом подробнее речь пойдет в следующем пункте.

3) Поля и методы классов

Поля классов - это общие свойства, характеристики классов. Например, что общего можно выделить у всех планет? Планету можно охарактеризовать длиной радиуса, величиной массы, удаленностью от солнца - для каждой отдельно взятой планеты эти характеристики будут принимать конкретные значения. Формальный синтаксис обращения к полю объекта класса такой:

объект.поле

Например, представим, что у нас есть класс “Планета”. Пусть в данном классе определено поле *масса*, хранящее информацию соответственно о массе планеты. Допустим, у нас есть объект класса “Планета”, название которого - Венера. Тогда, чтобы у объекта Венера узнать массу, надо обратиться к соответствующему полю объекта следующим образом:

Венера.масса

Методы классов - это функции для работы с объектами классов. Методы, как и поля, определены в самих классах. Формальный синтаксис вызова метода у объекта класса таков:

объект.метод([параметры])

Представим, что в упомянутом выше классе “Планета” определена функция получения радиуса *получитьРадиус()*. Тогда, чтобы вызвать данный метод и узнать радиус Венеры, объекта класса “Планета”, можно поступить следующим образом:

Венера.получитьРадиус()

В данном случае метод не имеет параметров, поэтому мы используем пустые скобки.

И поля, и методы классов иногда называют одним словом: *атрибуты* класса. Вызов методов объекта класса и обращение к полям объекта класса выполняются с использованием символа “.”.

Практически всё, что вам встретится в языке Python, является объектом: числа, строки и даже функции. Все они имеют свой собственный класс, в котором определены атрибуты: поля и методы.

Теперь, после обсуждения терминологии, мы можем перейти непосредственно к программированию на языке Python.

1.4 Базовые типы данных

Прежде чем начать писать программы на Python, давайте рассмотрим с какими данными мы можем работать. Ответ на этот вопрос кроется в типах данных языка Python. Так, например, для задания чисел используются три различных типа данных: целочисленные (*int*), с плавающей точкой (*float*) и комплексные (*complex*). Чтобы лучше понять, как работать с числами в Python, рассмотрим следующие простые примеры. Давайте создадим переменную *int_var* и присвоим ей целое значение *5*, создадим переменную *float_var* и присвоим ей вещественное значение *5.5*, и создадим переменную *complex_var* и присвоим ей комплексное значение *5.5 + 5.5j* (*j* - мнимая единица). Присваивание значения выполняется с помощью оператора присваивания =:

```
>>> int_var = 5
>>> float_var = 5.5
>>> complex_var = 5.5 + 5j
```

Программируя на языке Python, мы не можем задавать тип переменных при создании, но обязательно должны присвоить им некоторые значения, прежде чем использовать в программе.

Для задания строк используется тип *str*. Строковые литералы (значения типа *str*) представляют собой последовательности символов произвольной длины, которые могут быть заданы как в одинарных, так и в двойных кавычках:

```
>>> string1 = "I am a string"
>>> string2 = 'I am a string also'
```

Если требуется задать многострочный литерал, то применяются три кавычки (двойные " или одинарные '), идущие подряд:

```
>>> s = '''i
... am
... a
... multiline
... string
... '''
```

Подробнее многострочные строки (литералы) будут рассматриваться в “1.15 Строки str”.

Для задания значений логических выражений служит тип *bool*. Он допускает только два значения - *True* и *False* (подробности - в разделе “1.12 Логический тип данных”).

Для оперирования данными, которые представляют собой

последовательность элементов одного типа (однородные данные) и элементов разных типов (разнородные данные) используется тип *list*. Для задания списков используются квадратные скобки. Рассмотрим примеры однородных списков:

```
>>> numbers = [1, 2, 3, 4, 5]
>>> strings = ["one", "two", "three", "four", "five"]
```

Примеры разнородных списков:

```
>>> list0 = [1, 3.14, 2, 9.8, 3+4j]
>>> list1 = ["string", 1+4j, True]
```

К каждому элементу списка возможен доступ по его номеру (индексу), нумерация в языке Python начинается с нуля.

```
>>> list1[0] # Выведет string
string
>>> list2[2] # Выведет True
True
```

Для того, чтобы узнать, сколько элементов хранится в списке или в строке в Python есть встроенная функция для определения длины *len()*. Она применима не только к строкам и спискам, но и к некоторым другим типам данных, о которых узнаем позднее. Приведем пример использования такой функции для определения длины списка:

```
>>> len(list1) # Возвращает 3
3
```

Состав списка может быть изменен за счет добавления новых элементов (методы *append*, *insert*), изменения существующих (например: *list1[0] = 55*), а также удаления элементов (метод *remove*). Подробнее про списки - “1.24 Списки list”.

Для работы с данными, которые можно представить в виде последовательности элементов как одинаковых, так и разных типов, с невозможностью вносить изменения (менять элементы, менять количество элементов и пр.) в Python служит тип *tuple* (кортеж). Его использование практически аналогично типу *list*, за исключением двух особенностей:

- вместо квадратных скобок используются круглые,
- ни элементы кортежа, ни их количество и порядок не могут быть изменены.

Пример создания кортежа (разнородного):

```
>>> tpl = (1, '2', ['H2O', 1])
>>> tpl
(1, '2', ['H2O', 1])
```

Попытка изменить элемент кортежа:

```
>>> tpl[0] = 10
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Для кортежей, как и для списков, доступна функция определения длины кортежа *len()*, которая работает следующим образом:

```
>>> len(tpl)
3
```

Помимо строк *str*, списков *list*, кортежей *tuple*, в Python есть еще такие структуры данных как словари *dict* и множества *set*. Все эти структуры данных имеют общее название - *коллекции*. Они часто будут использоваться в примерах, а подробное описание коллекций можно найти в “Главе 4. Введение в алгоритмы и структуры данных”.

1.5 Приведение типов

Приведение (преобразование) типов - очень важная особенность языка Python, обеспечивающая переход от одного типа переменной к другому типу для этой же переменной, при этом переменную можно использовать также, как и раньше, меняется только ее тип. Например, в случае когда вы считываете число с помощью функции *input()* (см. раздел “1.42 Ввод и вывод данных в Python”), которая возвращает строку *str*, а работать в программе надо с числом. Отметим, что переходы между типами надо выполнять очень осторожно, так как для успешного приведения типов необходимо соблюсти множество ограничений.

В общем случае, для приведения простых типов можно вызвать метод с таким же названием, как и необходимый вам тип, и передать ему значение, тип которого вы собираетесь приводить. Метод вернет значение с преобразованным типом.

```
>>> a = 12
>>> a # Выведет целое число 12
12
>>> float(a) # Выведет вещественное число 12.0
12.0
>>> a # Снова выведет целое число 12
12
```

Обратите внимание на то, что в примере выше исходная переменная *a* не изменяется (результат выполнения функции *float()* никуда не присвоили).

Примеры функций для приведения простых типов:

- *int()* - приведение к целочисленному типу,

- *float()* - приведение к числу с плавающей точкой,
- *bool()* - приведение к булевому типу,
- *str()* - приведение к строке.

А что же с более “сложными” типами данных, такими как упомянутые ранее *list*, *tuple*? Мы предлагаем вам самостоятельно запустить код, предложенный ниже, и посмотреть, как в данном случае сработают функции приведения типов. Обратите внимание на способ инициализации нескольких переменных в одну строку: слева от знака равно перечислены через запятую переменные *a*, *b*, *c*, а справа от знака равно - через запятую соответственно значения переменных *12*, *13* и *14*:

```
>>> a, b, c = 12, 13, 14
>>> list([a, b, c])
[12, 13, 14]
>>> tuple((a, b, c))
(12, 13, 14)
```

Обратите внимание на квадратные скобки [], участвующие при передаче аргумента функции *list*, и на круглые скобки (), участвующие при передаче аргумента функции *tuple*.

1.5.1 Приведение к типу str

Программистам часто приходится обрабатывать информацию, представленную в текстовой форме (считанную из файла, введенную пользователем с клавиатуры). Для хранения такой информации используется строковый тип данных. Забегая немного вперед, скажем, что и ввод данных в программу, и вывод информации из программы осуществляются с помощью строк. Поэтому программистам часто приходится выполнять преобразование типа переменной к строковому типу *str*. Для большей ясности рассмотрим ряд примеров, представленных в табл. 1.1 ниже.

Таблица 1.1 – Примеры преобразования чисел к типу *str*

	Получение строки из вещественного числа	**Получение строки из целого числа**	**Получение строки из комплексного числа**
Пример кода	`>>> f = 6.626070040` `>>> s = str(f)`	`>>> i = 384000` `>>> s = str(i)`	`>>> c = 10 + 12j` `>>> s = str(c)`
Результат	`>>> s` `'6.62607004'`	`>>> s` `'384000'`	`>>> s` `'(10+12j)'`

И во всех выше представленных случаях можно выполнить проверку типа результата преобразования:

```
>>> type(s)
```

```
<class 'str'>
```

Любой встроенный тип данных может быть преобразован к типу *str*.

1.6 Числовые типы данных: int, float, complex

В предыдущем разделе упоминались числовые типы языка Python, представленные далее в табл. 1.2, далее рассмотрим их подробнее.

Таблица 1.2 – Встроенные числовые типы данных

№	Тип	Пояснение	Пример
1	*int*	Целые числа неограниченной точности	5, -5, 555555555555
2	*float*	Числа с плавающей точкой	7.1, 9., 8.5e-4, 8.5E4, 3E-11
3	*complex*	Комплексные числа (j - мнимая единица)	4 + 1.2j, 7j

В языке Python не требуется использовать спецификаторы или отдельные типы для хранения больших значений: в типе *int* используется неограниченная точность для представления целочисленных значений. Это означает, что величина числа ограничена только объемом доступной памяти.

Представление вещественных чисел *float* реализовано также, как представление чисел типа *double* в языке программирования С. Подробнее о представлении чисел в памяти можно будет ознакомиться в разделе “2.2 Формат представления данных в компьютере”.

Комплексное число типа *complex* состоит из двух чисел типа *float* - действительной и мнимой частей. После мнимой части записывается мнимая единица, которая обозначается как *j* или *J*, например:

2 - 5.6j или 2 - 5.6J

где 2 - действительная часть (*real*);

-5.6 - мнимая часть (*imaginary*).

Изученные нами типы *int*, *float* и *complex* являются классами. Вспомним пример из “1.4 Базовые типы данных”:

```
>>> int_var = 5
>>> float_var = 5.5
>>> complex_var = 5.5 + 5j
```

В примерах выше переменная *int_var* - это объект класса *int*, переменная *float_var* - это объект класса *float*, и переменная *complex_var* - это объект класса *complex*. То есть в строке кода:

```
>>> int_var = 5
```

мы создали объект класса *int*.

А в этой строке:

```
>>> float_var = 5.5
```

мы создали объект класса *float*.

И, наконец, в этой строке:

```
>>> complex_var = 5.5 + 5j
```

мы создали объект класса *complex*.

Для проверки типа создаваемого объекта можно воспользоваться специальной функцией *type(obj)*, которая в качестве своего аргумента принимает объект *obj*, а в качестве результата возвращает информацию о типе *obj*. В нашем случае это будет выглядеть следующим образом:

```
>>> type(int_var)
<class 'int'>
```

что является подтверждением создания объекта класса (типа) *int*.

Аналогично можно уточнить тип и созданного объекта *float_var*, а именно:

```
>>> type(float_var)
<class 'float'>
```

что является подтверждением создания объекта класса (типа) *float*.

И, наконец, для объекта *complex_var*:

```
>>> type(complex_var)
<class 'complex'>
```

что является подтверждением создания объекта класса (типа) *complex*.

1.6.1 Подробнее про int

1) Переход от float к int

В Python возможен переход от вещественных чисел к целым с помощью функции *int()*, примеры использования которой представлены в табл. 1.3.

Таблица 1.3 – Переход от вещественных чисел к целым

	Положительное вещественное число	**Отрицательное вещественное число**	**Вещественное число с нулевой целой частью**	**Отрицательное вещественное число с нулевой целой частью**
Пример кода	`>>> a = 3.4` `>>> b = int(a)`	`>>> a = -3.4` `>>> b = int(a)`	`>>> a = 0.4597` `>>> b = int(a)`	`>>> a = -0.4597` `>>> b = int(a)`

Итог	`>>> b`
	`3`
	`>>> type(b)`
	`<class 'int'>`
		`>>> b`
		`-3`
		`>>> type(b)`
		`<class 'int'>`
			`>>> b`
			`0`
			`>>> type(b)`
			`<class 'int'>`
				`>>> b`
				`0`
				`>>> type(b)`
				`<class 'int'>`

Обратите внимание на отбрасывание дробной части.

2) Переход от str к int

Ранее мы познакомились с возможностью приведения любого типа к типу *str*. А возможно ли привести тип *str* к другому типу? Да, возможно.

Начнем с приведения типа *str* к типу *int*. Это можно сделать, используя функцию *int(string, base)*, где *string* - строка, в которой хранится запись числа в системе счисления с основанием *base*. В результате преобразования получаем десятичное число, например:

```
>>> int('11', base=3)
4
```

Таким образом мы преобразовали 11_3 в десятичное число 4_{10}.

Обратите внимание, что в качестве аргумента функции мы используем представление числа в виде строки *string* (например, “11”) в указанной системе счисления *base* (3), а в качестве результата работы функции мы получаем десятичное число (в примере это 4). Функцию *int()* можно использовать и с одним параметром - строкой, как показано в примере далее:

```
>>> int('11')
11
```

В результате выполнения приведенного выше фрагмента кода строка ‘11’ была преобразована в целое число 11. При этом следует помнить, что когда мы не указываем параметр *base*, то это означает, что будет выполнено преобразование в десятичную систему счисления. То есть у аргумента *base* значение по умолчанию равно *10*. Подробнее этот вопрос будет рассматриваться в разделе "1.3.6. Передача аргументов в функцию".

1.6.2 Подробнее про float

1) Примеры методов

Для работы с вещественными числами в классе *float* определено несколько методов: *is_integer(), is_intreger_ratio(), hex()*. Рассмотрим некоторые из их подробнее.

Метод *is_integer()* позволяет проверить, является ли объект *a* целым числом, а именно:

```
>>> a = 3.4
>>> a.is_integer()
False
```

Обратите внимание, что в строке выше мы вызываем *метод класса* у конкретного *объекта класса*. В данном случае метод класса - это `is_integer()`, а объект класса - это a. Вызов метода у объекта

осуществляется через оператор “точка”: `a.is_integer()`.

Результаты выполнения кода означают, что объект *a* не является целым числом. Продолжим эксперимент: присвоим объекту *a* новое значение - целое положительное число 3.

```
>>> a = 3
```

Снова проверим его на целочисленность с помощью уже известного метода `is_integer()` (внимательный читатель может заподозрить подвох):

```
>>> a = 3
>>> a.is_integer()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'int' object has no attribute 'is_integer'
```

В результате получили ошибку - об этом можно догадаться по наличию слова *Error* в выведенном сообщении. Текст ошибок надо читать очень внимательно, потому что порой решение ошибки уже содержится в ее описании. Обратим ваше внимание, что в некоторых источниках при объяснении и описании подобных ситуаций вместо слова “ошибка” может встречаться слово “исключение”. Объяснение этому будет дано позднее (см. раздел “3.2.5 Исключения”), сейчас мы примем эти два термина как синонимы.

Рассмотрим текст ошибки из примера выше. Среда, которая предназначена для программирования на языке Python, в большинстве случаев подсказывает вам следующую информацию: строка, где произошла ошибка (после ключевого слова *line*) и тип ошибки. В нашем случае произошла ошибка типа *AttributeError*. Текст ошибки, представленный после типа ошибки, значит следующее: объект типа *int* не имеет атрибута *is_integer*, то есть для объектов класса ‘int’ не определен метод ‘is_integer’. Продолжим эксперимент. Присвоим переменной *a* целое положительное число 3 следующим образом, вызовем метод *is_integer()*:

```
>>> a = 3.0
>>> a.is_integer()
True
```

При этом класс объекта a будет следующим:

```
>>> type(a)
<class 'float'>
```

Обратите особое внимание на разницу в записи одного и того же числа: *3* и *3.0*, в связи с чем переменные, хранящие эти числа, имеют различные типы: *int* и *float* соответственно.

Еще один метод класса *float*, заслуживающий внимания, это - *as_integer_ratio()*, который работает следующим образом:

```
>>> a = 3.4
>>> a.as_integer_ratio()
(7656119366529843, 2251799813685248)
```

Данный метод возвращает пару целых чисел, частным которых является вещественное число, хранимое в переменной *a*. Проверка:

```
>>> 7656119366529843 / 2251799813685248
3.4
```

Более очевидный результат можно увидеть далее в примере:

```
>>> a = 3.0
>>> a.as_integer_ratio()
(3, 1)
```

2) Переход от int к float

Если в определенный момент необходимо перейти от типа *int* к типу *float*, то это можно сделать, например, так:

```
>>> i = -3
>>> f = float(i)
>>> f
-3.0
>>> type(f)
<class 'float'>
```

3) Переход от str к float

При использовании вещественных чисел может возникнуть необходимость преобразования строки *string* в число типа *float*, например, в случае, если вы решили считать вещественное число с клавиатуры с использованием функции *input()* (подробнее в разделе “1.42 Ввод и вывод данных в Python”). Для таких случаев можно использовать следующее выражение: *float(str)*. Продемонстрируем это на примере. Пусть переменной *s* будет присвоена строка с вещественным числом:

```
>>> s = '3.1458'
```

Попробуем получить из строки *s* вещественное число и выведем результат:

```
>>> s = '3.1458'
>>> f = float(s)
>>> f
3.1458
>>> type(f)
<class 'float'>
```

Преобразование прошло успешно. Имейте ввиду, что в случае передачи

некорректной строки в функцию *float* мы можем столкнуться с ошибками, например:

```
>>> s = '3.1458A'
>>> f = float(s)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: could not convert string to float: '3.1458A'
```

Информационное сообщение ошибки выше означает, что интерпретатор не может конвертировать строку в вещественное число '3.1458A' (в связи с наличием символа “A”).

Аналогично можно получить отрицательное вещественное число из строки:

```
>>> s = '-3.1458'
>>> f = float(s)
>>> f
-3.1458
>>> type(f)
<class 'float'>
```

Результат говорит об успешном преобразовании в тип данных *float*.

1.6.3 Подробне про complex

1) Примеры создания комплексных чисел

Для создания комплексной переменной можно воспользоваться простым способом, который был рассмотрен в “1.4 Базовые типы данных”. Также для создания объектов (переменных) такого типа используется функция *complex(real, imag)*, куда передаются 2 параметра: действительная *real* и мнимая части *imag*. При работе с числами этого типа используется математика комплексных чисел, для использования которой можно подключать специальный модуль *cmath* (о работе с модулями подробнее будет рассмотрено в “1.46 Импорт собственных модулей”).

Примеры создания комплексной переменной рассмотрены далее. Для начала введем следующие переменные:

```
>>> a = 12
>>> b = 9
>>> c = 1.2
>>> d = 6.3
```

Примеры того, что действительная и мнимая части могут принимать как целые, так и вещественные значения, представлены в табл. 1.4.

Таблица 1.4 – Примеры создания комплексной переменной с использованием функции complex()

Комбинация типов	Примеры

complex(int, int)	`>>> x = complex(a, b)`
	`>>> x`
	`(12+9j)`

complex(float, float)	`>>> y = complex(c, d)`
	`>>> y`
	`(1.2+6.3j)`

complex(int, float)	`>>> z = complex(a, c)`
	`>>> z`
	`(12+1.2j)`

complex(float, int)	`>>> w = complex(d, b)`
	`>>> w`
	`(6.3+9j)`

Проверить результаты создания переменных в таблице выше можно с помощью уже известной функции *type()*, например, для переменной *x*.

```
>>> type(x)
<class 'complex'>
```

Для других переменных *y*, *z*, *w* проведите проверку самостоятельно.

2) Некоторые методы класса complex

Рассмотрим некоторые методы класса *complex* на примере переменной *x*:

```
>>> x
(12+9j)
```

1. Получение сопряженного числа используя метод *conjugate()*:

```
>>> x.conjugate()
(12-9j)
```

2. Получение действительной части, используя поле *real* класса *complex*:

```
>>> x.real
12.0
```

3. Получение мнимой части, используя поле *imag* класса *complex*:

```
>>> x.imag
9.0
```

Обратите внимание, что переменные *a* и *b*, на основе которых было создано комплексное число *x*, являются переменными типа *int*, т.е. целыми числами, а тип возвращаемых значений действительной *real* и мнимой части *imag* - *float*.

Дополнительные операции, доступные не только для чисел, будут рассмотрены в разделе “1.8 Операции в языке Python”.

1.7 Представление чисел

Примеры представления чисел в табл. 1.5. Обратите внимание на

особенности представления чисел в 3-й строке таблицы. Для представления восьмеричных, шестнадцатеричных и двоичных чисел используются специальные символы: *o*, *x*, *b* соответственно.

Таблица 1.5 – Представления чисел

№	Представление числа	Описание
1	111111111111111	Целое число
2	2.1e-1, 2E3, 0.123	Вещественные числа
3	0**o**157, 0**x**9f, 0**b**10111	Восьмеричное, шестнадцатеричное и двоичное числа
4	3+4.8j, 22j, 5.6J	Комплексные числа

Если в записи числа обнаруживается точка или экспонента, интерпретатор Python создает объект класса *float* - вещественное число и использует вещественную (нецелочисленную) математику, когда такой объект участвует в выражении. Правила записи вещественных чисел в языке Python ничем не отличаются от правил записи чисел типа *double* в языке С и потому вещественные числа в языке Python обеспечивают такую же точность представления значений, как и в языке С [4].

1.7.1 Представление чисел с использованием e, E

Наряду с привычной записью вещественных чисел через “точку”, реже, но всё же используется запись через экспоненту *e* или *E*. Приведем несколько примеров. Присвоим переменной *a* следующее число и выведем его:

```
>>> a = 2.007e-100
>>> a
2.007e-100
```

Проверим тип данной переменной (результат будет следующей строкой):

```
>>> type(a)
<class 'float'>
```

Или вот такой пример числа с экспонентой:

```
>>> a = -1.000007e-0
>>> a
-1.000007
>>> type(a)
<class 'float'>
```

Пример с использованием E:

```
>>> a = 2E3
>>> a
2000.0
>>> type(a)
<class 'float'>
```

Подробнее о представлении вещественных чисел будет рассмотрено во второй главе “2.2.5.1) Основные сведения: мантисса, порядок и смещенный порядок”.

1.7.2 Восьмеричные oct(), шестнадцатеричные hex() и двоичные bin() числа

Примечательно, что литералы, где используются *o*, *x*, *b* являются всего лишь альтернативными формами записи целых чисел. Для преобразования целого числа в строку с представлением в любой из трех систем счисления можно использовать встроенные функции *hex(int)*, *oct(int)* и *bin(int)*, где *int* - целое число. Рассмотрим следующий пример. Создадим объект *a*, присвоив ему восьмеричное число, и выведем объект и его тип на экран:

```
>>> a = 0o123
>>> a
83
>>> type(a)
<class 'int'>
```

Как вы думаете, что произойдет в результате выполнения следующей строки?

```
>>> a = 0o888
```

Поскольку в восьмеричной системе счисления могут использоваться цифры от 0 до 7, мы получаем следующую ошибку:

```
  File "<input>", line 1
    a = 0o888
         ^
SyntaxError: invalid token
```

Интерпретатор не может распознать данный набор символов ни как восьмеричное число, поскольку в нем присутствуют “8”, ни как десятичное, поскольку есть символ *o*, не как строку, поскольку строка должна находиться в кавычках.

Из представления числа с использованием *o*, *x*, *b* можно явно получать целые числа в 10-ой системе счисления. Чтобы лучше понять, рассмотрим ряд примеров:

```
>>> int(0b1011001)
89
```

```
>>> int(0o1234567)
342391
>>> int(0x159ADF)
1415903
```

Таким образом, из двоичного, восьмеричного и шестнадцатеричного представления чисел были получены их преобразования в десятичную систему счисления.

1.8 Операции в языке Python

В табл. 1.6 приведены некоторые часто используемые операции языка Python. Большинство из них применимы не только к числам, но и к другим объектам.

Таблица 1.6 – Операции в языке Python

<table>
<tr><th>№</th><th>Операция</th><th>Пример</th><th>Пояснение</th></tr>
<tr><td>1</td><td>or</td><td>x or y</td><td>Логическая операция ИЛИ (значение y вычисляется, только если значение x ложно)</td></tr>
<tr><td>2</td><td>and</td><td>x and y</td><td>Логическая операция И (значение y вычисляется, только если значение x истинно)</td></tr>
<tr><td>3</td><td>not</td><td>not x</td><td>Логическая операция НЕ (отрицание, инверсия)</td></tr>
<tr><td>4</td><td>in
not in</td><td>x in y
x not in y</td><td>Проверка на вхождение (для итерируемых объектов и множеств)</td></tr>
<tr><td>5</td><td>is
not is</td><td>x is y
x is not y</td><td>Проверка идентичности объектов</td></tr>
<tr><td>6</td><td>>
>=
<=
<</td><td>x > y
x >= y
x < y
x <= y</td><td>Операции сравнения</td></tr>
<tr><td>7</td><td>==
!=</td><td>x == y
x != y</td><td>Операция проверки на равенство
Операция проверки на неравенство</td></tr>
<tr><td>8</td><td>|</td><td>x | y</td><td>Битовая операция ИЛИ</td></tr>
<tr><td>9</td><td>^</td><td>x ^ y</td><td>Битовая операция «исключающее ИЛИ»</td></tr>
</table>

			(XOR)
10	`&`	`x & y`	Битовая операция И
11	`<<` `>>`	`x >> y` `x << y`	Сдвиг значения *x* вправо на *y* битов Сдвиг значения *x* влево на *y* битов
12	`+`	`x + y`	Сложение
13	`-`	`x - y`	Вычитание
14	`*`	`x * y`	Умножение, дублирование
15	`%`	`x % y`	Взятие остатка от деления, форматирование для строк
16	`/` `//`	`x / y` `x // y`	Деление истинное Деление с округлением вниз
17	`-`	`-x`	Унарный знак «минус»
18	`~`	`~y`	Битовая операция НЕ (инверсия)
19	`**`	`x ** y`	Возведение в степень

Когда в одном выражении используются несколько операций, возникает вопрос определения приоритета операции, как в математике. Чем выше приоритет операции, тем ниже она находится в табл. 1.6, и тем раньше она выполняется в смешанных выражениях (вверху таблицы операции с наименьшим приоритетом, внизу - с наибольшим).

Отметим, что операции, представленные в таблице выше, можно поделить на бинарные и унарные. Для использования бинарной операции требуются два операнда. Операнд - это, в общем случае, переменная, к которой применяется операция. То есть, для применения бинарной операции требуются 2 переменные: левый операнд и правый операнд. Бинарные операции в табл. выше представлены в строках, чьи номера выделены полужирным шрифтом. Для применения унарных операций, напротив, требуется только один операнд. Унарные операции в табл. выше представлены в строках, чьи номера выделены курсивом.

Операции, представленные в табл. 1.6, можно поделить на группы, как

это сделано в табл. 1.7.

Таблица 1.7 – Группы операций в языке Python

№	Название группы операций	Операция
1	Логические	or, and, not
2	Вхождения	in, not in
3	Идентичности	is, is not
4	Сравнения	>, >=, <, <= ==, !=
5	Битовые	|, &, ^, >>, <<, ~
6	Математические	+, -, *, /, //, %, **

Далее в разделах операции будут рассмотрены подробнее по группам, представленным в табл. выше.

1.9 Математические операции над числами

Рассмотрим группу математических операций над числовыми типами данных. Отметим, что математические операции можно разделить на коммутативные: “+”, “*” и некоммутативные операции “-”, “/”. Примеры работы математических операций представлены в табл. 1.8.

Таблица 1.8 – Примеры математических операций над числовыми данными

Операция	Тип данных		
	int	**float**	**complex**

+	>>> a = 90
	>>> b = 1001
	>>> a + b
	1091
		>>> a_f = 2.4
		>>> b_f = 0.6
		>>> a_f + b_f
		3.0
			>>> c = 4 - 5j
			>>> d = 12 + 4j
			>>> d + c
			(16-1j)

-	>>> a = 90
	>>> b = 1001
	>>> a - b
	-911
	>>> b - a
	911
		>>> a_f = 2.5
		>>> b_f = 5.0
		>>> a_f - b_f
		-2.5
		>>> b_f - a_f
		2.5
			>>> c = 4 - 5j
			>>> d = 12 + 4j
			>>> c - d
			(-8-9j)
			>>> d - c
			(8+9j)

*	>>> a = 90
	>>> b = 1001
	>>> a * b
	90090
		>>> a_f = 2.5
		>>> b_f = 5.0
		>>> a_f * b_f
		12.5
			>>> c = 4 - 5j
			>>> d = 12 + 4j
			>>> d * c
			(68-44j)

/	>>> a = 25 >>> b = 50	>>> a_f = 2.5 >>> b_f = 5.0	>>> c = 4 - 2j >>> d = 2 + 4j

	>>> a / b
	0.5
	>>> b / a
	2.5
		>>> b_f / a_f
		2.0
		>>> a_f / b_f
		0.5
			>>> c / d
			1j
			>>> d / c
			-1j

//	>>> a = 4
	>>> b = 2
	>>> a // b
	2
	>>> b // a
	0
		>>> a_f = 2.5
		>>> b_f = 5.0
		>>> a_f // b_f
		0.0
		>>> b_f // a_f
		2.0
			>>> c // d
			Traceback (most recent call last):
			File "<input>", line 1, in <module>
			TypeError: can't take floor of complex number.

%	>>> a = 90
	>>> b = 1001
	>>> a % b
	90
	>>> b % a
	11
		>>> a_f = 5.5
		>>> b_f = 2.5
		>>> a_f % b_f
		0.5
		>>> b_f % a_f
		2.5
			>>> c % d
			Traceback (most recent call last):
			File "<input>", line 1, in <module>
			TypeError: can't mod complex numbers.

**	>>> a = 3
	>>> b = 5
	>>> a ** b
	243
	>>> b ** a
	125
		>>> a_f = 2.5
		>>> b_f = 2.0
		>>> a_f ** b_f
		6.25
			>>> c = 4 - 2j
			>>> d = 2
			>>> c ** d
			(12-16j)

Отметим, что в результате деления целых чисел получаем вещественное число типа *float*. Операция // выполняет деление чисел и возвращает целую часть от деления - целое число типа *int*. Операция % выполняет деление чисел и возвращает в качестве результата остаток от деления, например, для двух целых чисел это будет целое число типа int, если одно или сразу два числа – вещественные, тогда тип результата float. Деление с использованием / называют *истинным*, т.к. данная операция возвращает результат с дробной частью. Обратите внимание, что результат *истинного* деления всегда получается вещественным, а операции // и % недоступны для комплексных чисел.

1.9.1 Округление и точность при работе с float

Округление вещественных чисел в интерпретаторе Python обладает определенными особенностями. Давайте попробуем провести простые математические операции с вещественными числами:

```
>>> a = 0.1 + 0.2
>>> a
0.30000000000000004
```

Или вот такой пример:

```
>>> a = 0.1 + 0.2 - 0.3
>>> a
```

```
5.551115123125783e-17
```

В результате получили не самое очевидное представление нуля. Все эти особенности представления вещественных чисел в памяти связаны с вопросами *точности*, о которой речь пойдет в “Главе 2. Моделирование работы компьютера”.

1.9.2 Возведение в степень **

Если в выражении имеется несколько операций, они выполняются в направлении слева направо. Исключение составляет операция возведения в степень – эти операции выполняются справа налево, например, следующий код позволяет получить число 512 (не 64):

```
>>> 2**3**2
512
```

1.9.3 Интерпретация математических выражений с различными типами

В выражениях, где участвуют значения различных типов, интерпретатор сначала выполняет преобразование типов операндов к типу самого сложного операнда, а потом применяет математику, специфичную для этого типа [4].

```
>>> x = 12
>>> c = 5 - 12j
>>> y = 5.34
>>> x ** y - c**2 + x*c / y
(579329.3033362366+93.03370786516854j)
```

Интерпретатор Python в выражении выше работает с целым, вещественным и комплексным типами. Он ранжирует по возрастанию сложность работы с числовыми типами следующим образом: целые => вещественные => комплексные. Поэтому при обработке выражения с операндами разных типов интерпретатор сначала выполняет преобразование от целых к вещественным, а затем - от вещественных к комплексным.

1.10 Комбинированные операции присваивания

Ранее мы уже сталкивались с операцией присваивания (=). В Python существуют так называемые *комбинированные* операции присваивания, позволяющие сократить запись некоторых выражений. Рассмотрим, как они работают (см. табл. 1.9).

Таблица 1.9 – Примеры комбинированных операций

№	Операция	Развернутая запись	Сокращенная запись
1	`+=`	`>>> x = 125`	`>>> x = 125`

		>>> y = 11
		>>> x = x + y
		>>> x
		136
			>>> y = 11
			>>> x += y
			>>> x
			136

2	-=	>>> x = 125
		>>> y = 11
		>>> x = x - y
		>>> x
		114
			>>> x = 125
			>>> y = 11
			>>> x -= y
			>>> x
			114

3	*=	>>> x = 125
		>>> y = 11
		>>> x = x * y
		>>> x
		1375
			>>> x = 125
			>>> y = 11
			>>> x *= y
			>>> x
			1375

4	**=	>>> x = 5
		>>> y = 3
		>>> x = x ** y
		>>> x
		125
			>>> x = 5
			>>> y = 3
			>>> x **= y
			>>> x
			125

5	/=	>>> x = 6
		>>> y = 3
		>>> x = x / y
		>>> x
		2.0
			>>> x = 6
			>>> y = 3
			>>> x /= y
			>>> x
			2.0

6	//=	>>> x = 125
		>>> y = 11
		>>> x = x // y
		>>> x
		11
			>>> x = 125
			>>> y = 11
			>>> x //= y
			>>> x
			11

7	%=	>>> x = 125
		>>> y = 11
		>>> x = x % y
		>>> x
		4
			>>> x = 125
			>>> y = 11
			>>> x %= y
			>>> x
			4

8	^=	>>> x = 125
		>>> y = 11
		>>> x = x ^ y
		>>> x
		188
			>>> x = 125
			>>> y = 11
			>>> x ^= y
			>>> x
			118

9	|=	>>> x = 125 >>> y = 11	>>> x = 125 >>> y = 11

<table>
<tr><td></td><td></td><td>>>> x = x | y
>>> x
127</td><td>>>> x |= y
>>> x
127</td></tr>
<tr><td>10</td><td>&=</td><td>>>> x = 125
>>> y = 11
>>> x = x & y
>>> x
9</td><td>>>> x = 125
>>> y = 11
>>> x &= y
>>> x
9</td></tr>
<tr><td>11</td><td>>>=</td><td>>>> x = 1225
>>> y = 7
>>> x = x >> y
>>> x
9</td><td>>>> x = 1225
>>> y = 7
>>> x >>= y
>>> x
9</td></tr>
<tr><td>12</td><td><<=</td><td>>>> x = 1225
>>> y = 7
>>> x = x << y
>>> x
156800</td><td>>>> x = 1225
>>> y = 7
>>> x <<= y
>>> x
156800</td></tr>
</table>

Подробнее о поразрядных операциях (>>, <<, &, |, ~) можно узнать в разделе “2.2.2 Побитовые (поразрядные) операции”.

Следует отдельно отметить, что в Python нет операций инкремента ++ (увеличение на 1) и декремента – (уменьшения на 1), которые используются, например, в языке C. Их отсутствие компенсируется наличием комбинированных присваиваний += и -=.

1.11 Операции сравнения

Рассмотрим группу операций сравнения, которые представлены в табл. 1.1.6 в разделе “1.8 Операции в языке Python”. С помощью представленных выше операций можно строить выражения, например:

```
>>> x, y, z = 3, 12, -1
>>> x < y
True
>>> z > x
False
>>> z < y
True
```

Если обратить внимание на пример кода выше, то слева и справа от каждой операции присваивания стоят переменные *x*, *y*, *z*, играя роль левых и правых операндов в соответствующих случаях - в этом и заключается бинарность операции.

Другие примеры сравнения целых чисел:

```
>>> 3 < 5
True
```

Примеры сравнения вещественных чисел:

```
>>> 3.1 >= 3.0
False
```

И сравнение целых и вещественных чисел:

```
>>> 3.0 == 3
True
```

Стоит отметить, что не все числовые типы можно сравнивать. Например, не все операции сравнения работают с комплексными числами. Чтобы убедиться в этом, можно запустить такой пример кода:

```
>>> 12.34873928 + 3j <= 23.208403 + 8j
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: '<=' not supported between instances of 'complex' and 'complex'
```

В тоже время для сравнения комплексных чисел доступны операции проверки на равенство. Например:

```
>>> 4+7j == 3+1j
False
>>> 4+7j != 3+1j
True
```

1.12 Логический тип данных bool

Логический тип данных уже встречался вам в первом разделе. Напомним, что переменные (объекты) типа *bool* удобно использовать в качестве возвращаемого значения, когда ответ бинарный: да или нет. Для логического типа данных доступны операции сравнения, а также математические операции из табл 1.1.6. Создадим две булевых переменных:

```
>>> t = True
>>> t
True
>>> f = False
>>> f
False
```

С переменными типа *bool* можно выполнять математические операции:

```
>>> f + t
1
>>> f / t
0.0
```

Обратите внимание на возвращаемый результат: в первом случае - *int*, во

втором - *float*.

В Python применяется вычисление логических выражений *по сокращенной схеме*. В самом простом случае это работает так: простейшие логические операции *or* и *and* не вычисляют второй операнд, если результат определяется первым операндом. То есть для случая с операцией *or*: если первый операнд *True*, то второй не будет вычисляться. Для операции *and* - если первый операнд *False*, то второй не будет вычисляться.

1.13 Преобразование в тип bool

Отдельного внимания заслуживают вопросы преобразования других типов в логический тип *bool*. В Python можно к типу *bool* привести любой другой тип данных, однако стоит помнить интерпретацию *True* и *False*. Для чисел всё, что не нуль - это *True*, и только нуль - это *False*, что и продемонстрировано в табл. 1.10.

Таблица 1.10 – Интерпретация True и False для чисел

№	Тип	*value*	*bool(value)*	Пример
1	int	0	False	`>>> bool(0)` `False`
2	float	0.0	False	`>>> bool(0.0)` `False`
3	complex	0j	False	`>>> bool(oj)` `False`

4	int,
	float,
	complex
		любое другое, не 0	True	`>>> bool(-1)`
				`True`
				`>>> bool(10.0)`
				`True`
				`>>> bool(9-3j)`
				`True`

Что же касается других типов данных: строки, списки, кортежи, словари, то нужно понимать, что логический 0 (*False*) для таких типов ассоциируется с “*пустотой”*. То есть при преобразовании в *bool* объекта типа строка, словарь, кортеж или список получить *False* можно будет при условии, что объект “пустой”. Примеры представлены в табл. 1.11.

Таблица 1.11 – Интерпретация True и False для строки, словаря, кортежа и списка

№	Тип	*value*	*bool(value)*	Пример

1	list	[]	False	`>>> bool([])` `False`
2	dict	{}	False	`>>> bool({})` `False`
3	tuple	()	False	`>>> bool(())` `False`
4	str	‘’	False	`>>> bool('')` `False`

5	list,
	dict,
	tuple,
	str
		любое другое	True	`>>> bool([1, 2, 'a'])`
				`True`
				`>>> bool({'a', 3})`
				`True`
				`>>> bool(('a', 'b'))`
				`True`
				`>>> bool('a')`
				`True`

1.14 Логические операции

Рассмотрим группу логических операций из табл. 1.6, которые представлены в разделе “1.8 Операции в языке Python”. Примеры выполнения операций and, or и not представлены в табл. 1.12.

Таблица 1.12 – Примеры логических операций

X	Y	and	or	not

True	True	`>>> x = True`
		`>>> y = True`
		`>>> x and y`
		`True`
			`>>> x = True`
			`>>> y = True`
			`>>> x or y`
			`True`
				`>>> x = True`
				`>>> not x`
				`False`

False	True	`>>> x = False`
		`>>> y = True`
		`>>> x and y`
		`False`
			`>>> x = True`
			`>>> y = False`
			`>>> x or y`
			`True`
				`>>> x = False`
				`>>> not x`
				`True`

False	False	`>>> x = False`
		`>>> y = False`
		`>>> x and y`
		`False`
			`>>> x = False`
			`>>> y = False`
			`>>> x or y`
			`False`
				`>>> x = True`
				`>>> y = False`
				`>>> not (x and y)`
				`True`
				`>>> not (x or y)`
				`False`

Акцентируем ваше внимание на том, что с помощью операций сравнения и логических операций можно строить такие выражения, как, например:

```
>>> x, y, z = 3, 12, -1
```

```
>>> x < y <= z
False
```

Сначала выполнится часть выражения: *x < y*, т.е. *3 < 12*, что даст в результате *True*. Затем выполняется вторая часть: *y <= z*, т.е. *12 <= -1*, что даст в результате *False*. И далее будет выполнено сопоставление двух частей выражения: *True* и *False*, в результате чего будет получено *False*. Сложное логическое выражение можно представить в более подробном виде:

```
>>> x < y and y <= z
False
```

1.15 Строки str

Особого внимания заслуживает один из самых популярных типов данных - строки. Строки (тип *str)* - неизменяемый тип данных, поэтому при работе со строками результат выполнения методов нужно сохранять. Напротив, для объектов *изменяемых* типов данных вызов метода вносит изменения в сам объект, у которого вызывается. Более подробнее об изменяемости и неизменяемости можно узнать в разделе “1.38 Изменяемые и неизменяемые объекты”.

Для инициализации переменных типа str можно использовать как одинарные кавычки: ‘example’, так и двойные: “example”.

В Python есть возможность создавать многострочные переменные типа *str*, с чем мы уже сталкивались ранее в разделе “1.4 Базовые типы данных”. Форма определения многострочных строковых литералов (блочная строка) – тройные кавычки или апострофы. Когда используется такая форма, все строки в программном коде объединяются в одну строку, а там, где в исходном тексте выполняется переход на новую строку, вставляется символ «конец строки»: ‘\n’. Например:

```
>>> a = """aaa
... aaaa
... bbb"""
>>> a
'aaa\naaaa\nbbb'
```

Строку, содержащую кавычки другого вида, отличные от тех, в которые обрамлена вся строка, можно создавать следующим образом:

```
>>> s = "abc' ' defg"
>>> s
"abc' ' defg"
```

Бывают случаи, когда внутри строки строки должны быть специальные символы, такие как, например, двойные или одинарные кавычки (при этом

строка обрамлена двойными или одинарными кавычками соответственно), слеш (косая черта) или бэкслеш (обратная косая черта) или ряд других символов (так называемые управляющие символы). Например, в следующем случае создание строки завершится ошибкой:

```
>>> s = "Maggy said: "Hello, world!""
  File "<input>", line 1
    s = "Maggy said: "Hello, world!""
                          ^
SyntaxError: invalid syntax
```

Чтобы избежать подобных ситуаций, используют *экранирование*, что подразумевает использование экранированных последовательностей или *escape*-последовательностей, которые могут состоять из одного или нескольких символов после обратной косой черты \. Нам уже знаком пример такой последовательности - переход на новую строку ‘\n’. Предыдущий пример можно исправить, добавив внутри строки вместо " экранированную последовательность \":

```
>>> s = "Maggy said: \"Hello, world!\""
>>> s
'Maggy said: "Hello, world!"'
```

Другие примеры использования экранированных последовательностей приведены в табл. А.2 в приложении А.

Для вывода строки также можно воспользоваться функцией *print()*:

```
>>> print(a)
aaa
aaaa
bbb
```

Обратите внимание, что строка выглядит иначе. Объяснения этому будут даны в “Глава 3. Парадигмы программирования”.

Ранее упоминалась функция *len(obj)*, которая помогает определить длину переданного на вход объекта. Покажем, как данная функция работает на строках:

```
>>> s = 'example'
>>> n = len(s)
>>> n
7
```

1.16 Доступ к элементам строки по индексу

У строк в языке Python есть возможность индексации, т.е. обращения к элементам строки по индексу. В роли индекса выступает номер элемента. Для индексации используются квадратные скобки. Индекс обязательно

является целым числом, при этом может быть положительным и отрицательным. Для лучшего понимания приведем пример:

```
>>> index = 3
>>> my_str = 'AQBWCF'
>>> my_str[index]
'W'
```

При попытке получения элемента с индексом, превышающим длину строки, как в следующем примере, мы получим ошибку:

```
>>> index = 10
>>> my_str = 'AQBWCF'
>>> my_str[index]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: string index out of range
```

Как можно догадаться, такого рода ошибки будут возникать в подобных ситуациях при работе со списками, кортежами и другими хранилищами данных, где возможно обращение по индексу.

Использование отрицательного индекса означает обход строки с конца. Если первый элемент - нулевой (0), то элемент справа от него - первый (1), а элемент слева от него - “минус первый” (-1) .

Проиллюстрировать можно следующим примером (см. табл. 1.13):

```
>>> c = 'asdfghj'
>>> c[3]
'f'
>>> c[-3]
'g'
```

Таблица 1.13 – Пояснение примера прямой обратной индексации строк

index→	0	1	2	3	4	5	6
Строка с	*a*	*s*	*d*	*f*	*g*	*h*	*j*
←index	-7	-6	-5	-4	-3	-2	-1

Следовательно, следующий пример кода с той же строкой *c*, определенной выше возвращает True:

```
>>> c[0] == c[-7]
True
```

Стоить помнить, что строки - неизменяемые, то есть при попытке изменить элемент строки через обращение по индексу столкнемся со следующей ошибкой:

```
>>> c = 'asdfghj'
>>> c[3] = '1'
```

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Перейдем к более общему понятию - срезам.

1.17 Срезы для строк

Обращение по индексу в строке - это частный случай среза. Срез - возможность языка Python получить подстроку из строки. Для совершения среза используются те же квадратные скобки, в которых через двоеточие перечисляются параметры - целые числа *[start:stop:step]*: start - индекс, откуда начать срез; *stop* - индекс, до которого выполнить срез (не включая указанный индекс!), *step* - с каким шагом выполнить срез. Эти параметры имеют значения по умолчанию: *start* равен 0, *stop* равен длине строки, *step* равен 1. Когда не указано ни одного параметра, то операция [:] позволяет создать такую же строку:

```
>>> s = "hello!"
>>> a = s[:]
>>> s
>>> a
hello!
hello!
```

Если указан только один параметр в квадратных скобках, то это уже знакомое нам обращение по индексу. Если указаны два параметра - то это срез, для которого указаны начальная позиция *start* и конечная *stop*, шаг при этом считается по умолчанию равным 1. Проиллюстрируем:

```
>>> s = 'asdfghjklpt'
>>> s[3]
'f'
>>> s[3:8]
'fghjk'
>>> s[3:8:2]
'fhk'
```

Также, как и в случае с обращением по индексу, в срезах есть возможность использовать отрицательные параметры. Проиллюстрируем на примере определенной выше строки *s*:

```
>>> s[len(s):2:-2]
'ljg'
```

Стоит помнить, что при значении параметра *step*<0 порядок использования *step* и *stop* должен быть изменен на противоположный: [*stop*:*start*:*step*], не включая *start*.

Пример:

```
>>> s[::-1]
'lkjhgfdsa'
```

Приведенный пример кода позволил получить перевернутую строку, другими словами - извлек все элементы строки *s* в обратном порядке.

При указании неверных параметров в срезе вы можете получить пустую строку, а указав нулевой шаг вы столкнетесь с такой ошибкой:

```
>>> s[len(s):2:0]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: slice step cannot be zero
```

Некоторые математические операции допустимы к выполнению на строках, поговорим о них далее подробнее.

1.18 Некоторые операции для строк

1.18.1 Математические операции над строками

Для работы со строками доступны математические операции: + и *. Операция + используется для конкатенации (сложения строк), операция * используется для умножения строки на число. Например:

```
>>> 'aa' + 'bb'
'aabb'
```

Стоить иметь ввиду, что операция + может работать только с объектами одного типа. То есть код, представленный далее, вызовет ошибку:

```
>>> 'aa' + 1
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Умножение допустимо использовать между строкой и целым числом, например:

```
>>> 'a' * 4
'aaaa'
```

Эта операция - *дублирование*.

При использовании операции * к ошибке может привести умножение строки на строку. Например, при выполнении следующего фрагмента кода:

```
>>> 'a' * 'c'
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

К ошибке приведет также использование математических операций “-” для строк, например:

```
>>> 'start' - 't'
Traceback (most recent call last):
```

```
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Аналогичные ошибки будут наблюдаться при попытке использовать другие математические операции для строк, или для строк и чисел.

1.18.2 Логические операции над строками

Над строками можно выполнять уже знакомые нам логические операции из табл. 1.6 раздела “1.8 Операции в языке Python”. Примеры работы таких операций на строках приведены в табл. 1.14.

Таблица 1.14 – Логические операции на строках

№	Выражение	Результат
1	`'a' and 'a'`	`'a'`
2	`'a' and 0`	`0`
3	`'a' and ''`	`''`
4	`not ""`	`True`
5	`not 'a'`	`False`

Обратите внимание на строки 4 и 5 в табл. 1.14. Пустая строка при приведении к типу bool преобразуется в тип *False*. Любая другая непустая строка, например, '1' или '0', при преобразовании в bool даст результат *True*.

1.19 Операция проверки вхождения

Операции проверки вхождения *in*, *not in*, представленные в табл. 1.6 раздела “1.8 Операции в языке Python”, можно использовать на строках для определения, является ли одна строка частью другой (проверка вхождения подстроки в строку), например:

```
>>> 't' in 'start'
True
```

Как видим из примера выше, строка может быть представлена одним символом. Пример для более длинной подстроки:

```
>>> 'ello' not in 'Hello everybody!'
False
```

Однако, успешность применения данной операции зависит от регистра, а именно:

```
>>> 't' in "STarT"
False
```

Операция проверки вхождения работают для объектов одного типа. То есть пример кода далее приведет к ошибке:

```
>>> 1 in '11111'
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'in <string>' requires string as left operand, not int
```

Если же исправим следующим образом:

```
>>> '1' in '11111'
True
```

1.20 Сравнение строк

Доступные для применения на строках операции сравнения >, >=, <, <=, ==, != представлены в табл. 1.6 раздела “1.8 Операции в языке Python”. Рассмотрим небольшие примеры, чтобы понять, как работают эти операции.

```
>>> x, y, z = '123', 'abc', '1a0'
>>> x < y < z and ~ x < y and y < z
False
```

Операции сравнения строк работают с учетом лексикографического порядка в соответствии с таблицей Unicode, которая подробнее будет рассмотрена в “2.2.6 Формат представления текстовой информации”, где для упрощения понимания каждому символу поставлен в соответствие некоторый код - порядковый номер.

Простые примеры:

```
>>> 'a' < 'b'
True
>>> 'a' < '3'
False
```

Особое внимание надо уделить проверке строки на пустоту:

```
>>> not ''
True
>>> not 'a'
False
```

1.21 Другие методы для работы со строками

1.21.1 Неизменяемость строк на примере работы методов

Мы уже упоминали, что строки в языке Python – неизменяемые. Подробнее о неизменяемости мы поговорим позже, а сейчас рассмотрим, что это означает на практике.

1) Замена подстроки (символа) в строке

Для замены символов в строке есть специальный метод *replace()*, который работает следующим образом:

```
>>> s  = " hello, everybody "
>>> s.replace(',', '!')
```

```
' hello! everybody '
```

Для сохранения результата можно создать новую строку и присвоить ей результат выполнения метода, при этом исходная строка не изменится:

```
>>> s  = " hello, everybody "
>>> new_s = s.replace(',', '!')
>>> new_s
' hello! everybody '
>>> s
' hello, everybody '
```

2) Удаление пробелов из строки

Обратите внимание, что в предыдущем примере справа и слева от строки *s* - пробелы. В языке Python есть метод для удаления пробельных символов (*whitespace* символы: пробел, перевод строки, табуляция и др.): *strip()*, который возвращает копию строки без пробелов в начале и конце строки:

```
>>> s.strip()
'hello, everybody'
```

Также есть метод для удаления пробельных символов в начале строки *lstrip(),*:

```
>>> s.lstrip()
'hello, everybody '
```

И метод для удаления пробельных символов в конце строки *rstrip()*:

```
>>> s.rstrip()
' hello, everybody'
```

Вышепредставленные методы позволяют удалять из начала и конца заданной строки все вхождения другой определенной подстроки. Например, удаление подстроки из начала и конца строки:

```
>>> s = '!!!hello, everybody!!!'
>>> s.strip("!")
'hello, everybody'
```

Удаление подстроки с начала строки:

```
>>> s = 'hello, everybody'
>>> s.lstrip('hello')
', everybody'
```

Удаление подстроки в конце строки:

```
>>> s = 'hello, everybody'
>>> s.rstrip('body')
'hello, ever'
```

3) Верхний и нижний регистр строк

Для строк есть метод, позволяющий привести строку к верхнему регистру, *upper()*:

```
>>> s.upper()
' HELLO, EVERYBODY '
```

И метод, который позволяет привести строку к нижнему регистру *lower()*:

```
>>> s.lower()
' hello, everybody '
```

Для проверки, все ли символы строки в нижнем регистре, есть метод *islower()*:

```
>>> s.islower()
True
```

Для проверки, что все символы строки в верхнем регистре, есть аналогичный метод *isupper()*:

```
>>> s.isupper()
False
```

Вышеприведенные примеры не меняют саму переменную *s*, в чем можно убедиться, выведя строку *s*:

```
>>> s
' hello, everybody '
```

Таким образом, все представленные методы возвращают *измененную копию* исходной строки.

1.21.2 Поиск подстроки в строке

Для поиска подстроки в заданной строке есть специальный метод *find()*. Он выполняет поиск <u>с начала</u> строки и возвращает индекс, с которого в заданной строке начинается искомая подстрока, например:

```
>>> s  = " hello, everybody "
>>> s.find('body')
13
```

В случае, если искомая подстрока отсутствует, метод возвращает *-1*:

```
>>> s.find('head')
-1
```

Искомая подстрока может состоять из одного символа:

```
>>> s.find(',')
6
```

Альтернативный метод *rfind()* выполняет поиск подстроки <u>с конца</u> заданной строки:

```
>>> s.rfind('body')
13
```

В случае, если искомой подстроки нет в заданной строке, метод *rfind()*, также как и *find()*, возвращает *-1* (искомая подстрока может состоять из одного символа):

```
>>> s.rfind('i')
-1
```

Если же в качестве искомой подстроки в методы *find()* и *rfind()* передать пустую строку, то результат будет следующим:

```
>>> s  = "hello, everybody"
>>> s.find('')
0
>>> s.rfind('')
16
```

Данные методы чувствительны к регистру. Убедитесь в этом самостоятельно, выполнив, например, такой фрагмент кода:

```
>>> s  = "hello, everybody"
>>> s.find(H)
```

1.21.3 Количество вхождений подстроки в строку

Помимо метода поиска индекса элемента в строке, есть метод, определяющий количество вхождений подстроки в строку без пересечений, называемый *count()*. Пример работы метода:

```
>>> s = """Oh, jingle bells, jingle bells
... Jingle all the way
... Oh, what fun it is to ride
... In a one horse open sleigh
... Jingle bells, jingle bells
... Jingle all the way
... Oh, what fun it is to ride
... In a one horse open sleigh"""
>>> s.count('Jingle')
3
```

Если искомой подстроки в строке нет, то метод возвращает 0:

```
>>> s.count('Hello')
0
```

При использовании данного метода можно указывать индекс в строке, с которого надо вести поиск вхождений. Определим длину строки, затем выберем индекс, с которого хотим выполнить поиск:

```
>>> len(s)
203
>>> s[77:]
'In a one horse open sleigh\nJingle bells, jingle bells\nJingle
all the way\nOh, what fun it is to ride\nIn a one horse open sleigh'
>>> s.count('Jingle', 77)
```

```
2
```

Указывая индекс начала поиска, мы также можем указать еще один параметр - индекс, до которого (не включая) следует вести поиск. Продемонстрируем:

```
>>> s[77:130]
'In a one horse open sleigh\nJingle bells, jingle bells'
>>> s.count('Jingle', 77, 130)
1
```

1.21.4 Методы startswith() и endswith()

Методы *startswith(prefix)* и *endswith(suffix)* позволяют определять, начинается ли строка с определенного префикса *prefix* и оканчивается ли строка определенным суффиксом *suffix* соответственно. Рассмотрим примеры, взяв ту же строку s, что и в предыдущем подразделе:

```
>>> s.startswith('Oh,')
True
>>> s.endswith('gh')
True
```

Данные методы позволяют ограничить область поиска, указав индекс начала поиска, например:

```
>>> s[50:]
'Oh, what fun it is to ride\nIn a one horse open sleigh\nJingle bells, jingle bells\nJingle all the way\nOh, what fun it is to ride\nIn a one horse open sleigh'
>>> s.startswith('Oh', 50)
True
```

Указывая индекс начала поиска, можно указать и индекс, до которого нужно искать:

```
>>> s[50:103]
'Oh, what fun it is to ride\nIn a one horse open sleigh'
>>> s.endswith('gh', 50, 103)
True
```

1.21.5 Методы isalpha() и isdigit()

Существует метод проверки строки, который определяет, состоит ли строка только из символов *isalpha()*:

```
>>> s  = " hello, everybody "
>>> s.isalpha()
False
```

Аналогично метод *isdigit()* определяет, состоит ли строка только из цифр:

```
>>> s.isdigit()
False
```

Если вы хотите преобразовать строку в число, используя для проверки методы *isdigit()*, следует учитывать, что метод распознает только цифры. Поэтому для строк вида ‘1.23’, ‘-12’ метод вернет False:

```
>>> y = '1.23'
>>> y.isdigit()
False
>>> y = '-12'
>>> y.isdigit()
False
```

1.21.6 Метод join()

Также особого внимания заслуживает метод *join()*, который будет подробнее рассматриваться в разделе “1.35 Методы split и join”. Для строк метод работает следующим образом:

```
>>> '_separator_'.join('target')
't_separator_a_separator_r_separator_g_separator_e_separator_t'
```

Строка, у которой вызывается метод *join()*, становится разделителем в строке, которая передается в качестве аргумента в метод *join()*.

1.21.7 Форматирование строк

Когда нужно создать строку, подставив в неё некоторые данные, полученные в процессе выполнения программы (пользовательский ввод, данные из файлов, результаты вычисления выражений), используют специальные средства форматирования строк - операция %, и метод *format()*. Например, имеются следующие переменные:

```
>>> name = 'Ivan'
>>> surname = 'Petrov'
>>> age = 18
```

Пример использования % для вывода строки и числа (об этом говорит спецификаторы s и d после символа %):

```
>>> 'Student: Name: %s,  Surname: %s, Age: %d' % (name, surname, age)
'Student: Name: Ivan,  Surname: Petrov, Age: 18'
```

Пример использования функции *format()*:

```
>>> 'Student: Name: {}, Surname: {}, Age: {}'.format(name, surname, age)
'Student: Name: Ivan, Surname: Petrov, Age: 18'
```

Рассмотрим подробнее оба подхода.

1) Метод format()

Метод *format()* работает следующим образом:

```
>>> s = "We love {}.".format("open-source")
>>> s
```

```
'We love open-source.'
```

Для определения места, куда нужно подставить определенный объект, используются фигурные скобки { } прямо в самой строке. Сам подставляемый объект передается в функцию *format(obj)*. Соответственно, можно передавать несколько объектов в функцию *format(obj)*, выделив для них предварительно несколько мест, например:

```
>>> s = "We love {} {}.".format("open source", 'software')
>>> s
'We love open source software.'
```

Более того, указывая в фигурных скобках индексы, можно регулировать порядок подстановки объектов, а именно:

```
s = "We love {1} {0}.".format('software', "open source")
s
'We love open source software.'
```

2) Использование % для форматирования строк

Форматирование можно выполнять с помощью операции %, который используется вместе со спецификаторами, знакомыми из языка C. Например, когда на нужное место в строке нужно подставить другую строку, используют “%s”:

```
>>> 'Hello, %s!' % 'student'
'Hello, student!'
```

Если нужно вывести вещественное число, то используют “%f”:

```
>>> 'height: %f m.' % h
"height: 1.800000 m."
```

Количество выводимых знаков после запятой можно регулировать следующим образом:

```
>>> 'height: %.2f m.' % h
"height: 1.80 m."
```

В одной строке можно комбинировать различные спецификаторы:

```
>>> 'name: %s, surname: %s, id: %d, h: %.2f' % (name, surname, id, h)
'name: Ivan, surname: Petrov, id: 100032, h: 1.80'
```

Как и в самом первом примере данного подраздела, после строки и операции % передали несколько аргументов, заключенных в круглые скобки - кортеж. Так приходится делать, когда требуется передать для форматирования больше одного аргумента.

Другие примеры использования % с различными спецификаторами рассмотрены в табл. А.1. в приложении А.

1.22 Ветвление

Начнем рассмотрение управляющих инструкций Python - специальных

конструкций, которые позволяют управлять порядком вычислений в программе. Простейшей из таких конструкций является условный оператор. Его синтаксис представлен ниже:

```
if <условие 1>:
    <действие 1> # Данный блок сработает, если <условие 1> истинно
elif <условие 2>: # Необязательный блок
    <действие 2> # Данный блок сработает, если <условие 1> ложно И <условие 2> истинно
else:  # Необязательный блок
    <действие 3> # Данный блок сработает, если ложными окажутся оба условия выше
```

Данная конструкция проверяет истинность набора условий и в зависимости от того, какие из них выполняются, переводит управление к соответствующим блокам кода. Условием может быть любое выражение, которое можно привести к типу *bool*, например:

```
a == 5
```

Еще пример с усложненным условием:

```
a > 6 and b == 6
```

Или просто:

```
True
```

Использование последнего примера в условном операторе:

```
>>> c = True
>>> if c:
...    c
...
True
>>> if c:
...     c = False
...
>>> c
False
```

И, отмечая, что *c* равно *False*, сделаем следующее:

```
>>> if c:
...     c = False
... else:
...     print('c is not True')
...
c is not True
```

Действия в теле условного оператора также могут быть достаточно произвольными, например:

```
>>> a = 6
>>> if a > 2:
...   print("a is greater than 2")
```

```
... else:
...     print("a is less than 2")
...
a is greater than 2
```

Конструкция *if-elif-else* может содержать произвольное количество блоков *elif*, при этом проверка условий осуществляется последовательно до первого срабатывания. Как только найден *elif* с истинным условием, выполняется его блок действия и далее управление передается инструкции следующей за *if-elif-else*. В случае, если истинного блока *elif* не найдено управление передается действию, ассоциированному с *else* (если оно задано), либо также инструкции следующей за *if-elif-else*. Отдельно стоит отметить, что блок *else* всегда один.

1.23 Циклы

Для выполнения повторяющихся действий используются операторы циклов. В Python это цикл для перебора значений (*for*) и цикл с условием (*while*).

1.23.1 Цикл for

Для определения циклов с известным количеством итераций используется конструкция *for*. Синтаксис цикла *for* следующий:

for <переменная> in <коллекция>:
    <действие 1>
else: # необязательный блок
    <действие 2>

Операция *in* - это уже знакомая операция проверки на вхождение, (рассматривалась в разделе “1.19 Операция проверки вхождения”), которая также используется для итерирования (обхода по элементам) объектов. Не все типы данных могут быть итерируемыми, т.е. переменные не всех типов данных могут состоять из элементов, которые можно было бы обойти в цикле.

Главное отличие цикла *for* от цикла *while*, который рассмотрим далее, заключается в том, что количество итераций явно определяется числом элементов в коллекции. При этом, в роли коллекции могут выступать любые итерируемые объекты, например, списки, строки и т.д.:

```
>>> for i in [1, 2, 3, 4]:
...     print(i, "Hello!")
...
1 Hello!
```

```
2 Hello!
3 Hello!
4 Hello!
```

1.23.2 Цикл while

Цикл *while* использует следующий синтаксис:

while <условие>:

<действие 1> # тело цикла

else: # необязательный блок

<действие 2>

Обратите ваше внимание на новый термин - *итерация*, что в простом объяснении означает один шаг цикла, или однократное выполнение тела цикла.

Принцип работы этого цикла заключается в следующем: сначала происходит проверка условия на истинность, если она пройдена, то выполняется действие 1, за которым вновь следует проверка и выполнение действия 1 и так до тех пор, пока условие не станет ложным. После того, как условие становится ложным, происходит выполнение действия 2.

Пример цикла для вывода трех строк в терминал:

```
>>> a = 0
... while a < 3:
...     print("I am a string")
...     a = a + 1
...
I am a string
I am a string
I am a string
```

Обратите внимание, что переменная *a* используется в условии выхода из цикла как своего рода счетчик итераций. В языке Python счетчики необходимо предварительно инициализировать.

Часто *while* используется для проверки условия, истинность которого определяется кодом вне *while*. Примером может быть ожидание ввода специального символа в последовательности, вводимой пользователем:

```
>>> a = input()
... sum = 0
... while a != "q":
...     sum = sum + int(a) # Действия над a
...     a =  input() # Повторный ввод
...
>? 12
>? 14
```

```
>? q
>>>
```

Цикл *while* таит в себе определенную опасность - при невнимательном программировании условия он может стать бесконечным и тогда выполнение программы никогда не закончится (“программа зациклилась”).

Про функцию *input()* можно подробнее прочитать в разделе “1.42 Ввод и вывод данных в Python”.

1.23.3 Функция range(start, stop[, step])

Когда требуется перебрать не элементы последовательности, а ее индексы, можно использовать функцию *range()*:

```
>>> for i in range(len([‘a’, ‘b’, ‘c’, ‘d’])):
...     print(i, "Hello!")
...
0 Hello!
1 Hello!
2 Hello!
3 Hello!
```

Функция *range()* принимает следующие параметры:

range (start, stop[, step])

где *start* - это начальное значение, *stop* - значение, до которого необходимо сгенерировать последовательность, *step* - шаг (по умолчанию равен единице). Все аргументы функции *range()* должны быть целыми числами. В разделе “1.41 Передача аргументов в функцию” можно найти объяснение формулировки “по умолчанию”.

Важным отличием цикла *for* от *while* является возможность использовать переменную, содержащую текущий элемент последовательности в теле цикла, не определяя ее вне цикла. Например, данный код вычисляет сумму чисел от 1 до 4 и выводит результат:

```
>>> sum = 0
... for i in [1, 2, 3, 4]:
...    sum = sum + i
... else:
...    print(sum)
...
10
```

Часто могут возникать ситуации, когда в зависимости от дополнительных условий требуется управлять выполнением цикла. Для этого служат ключевые слова *break* (прервать выполнение цикла) и *continue* (прервать выполнение текущей итерации и перейти к следующей). Ниже

представлен код, который вычисляет сумму вводимых пользователем *положительных* чисел до момента, когда будет введен символ 'q':

```
>>> a = input()
... sum = 0
... while True:
...     if a == 'q': # Прерывание цикла, если введен q
...         break
...     if int(a) <= 0: # Пропуск итерации,
...                           # если введено отрицательное число:
...         continue
...     sum = sum + int(a) # Действия над a
...     a =  input() # Повторный ввод
...
>? 12
>? 14
>? q
```

1.24 Списки list

Одной из наиболее часто используемых структур данных в языке Python является список. Это изменяемая упорядоченная последовательность элементов произвольного типа, которая не всегда последовательно расположена в памяти. Таким образом, не следует путать списки в Python с массивами, элементы которых, напротив, расположены в памяти строго последовательно, что обеспечивает быстрый доступ к элементам по индексу (для работы с массивами в Python есть специальный модуль *array*; о модулях и работе с ними речь пойдет в разделе “1.45 Модули”).

Списки в Python обозначаются квадратными скобками. Таким образом, следующей строчкой мы можем создать пустой список *s*:

```
>>> s = []
>>> s
[]
```

Конечно, список можно создавать не обязательно пустым, например:

```
>>> s = [1, 'Abc', [10, 20, 30], 3.14]
>>> s
[1, 'Abc', [10, 20, 30], 3.14]
```

Это пример списка, в котором первый его элемент - целое число, второй элемент - строка, третий - список, а четвертый - число с плавающей точкой. Вложенность списков может быть любой.

Один из самых интересных способов создания списка - это способ на основе строки:

```
>>> a = "Hello"
>>> b = list(a)
>>> b
['H', 'e', 'l', 'l', 'o']
```

Длину списка, как уже демонстрировалось ранее, можно узнать с помощью функции *len(obj)*, передав ей список следующим образом:

```
>>> b = ['H', 'e', 'l', 'l', 'o']
>>> len(b)
5
```

1.24.1 Методы для работы с однородными списками

Если список состоит только из чисел, то можно вычислить сумму всех его элементов, используя встроенную функцию *sum():*

```
>>> num_list = [-23, 1, 0, 14, 9, 2, -1]
>>> sum(num_list)
2
```

Если список состоит не из чисел, то функцией sum() воспользоваться будет нельзя:

```
>>> sym_list = ['a', 'e', '1', 'w']
>>> sum(sym_list)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

1.25 Изменяемость списков

Списки - изменяемый тип данных. Это означает, что мы можем после создания списка добавлять элементы, удалять их, изменять существующие. В этом нам может помочь набор методов для работы со списком.

Наиболее часто используемый метод *append()*, который добавляет элемент в конец списка. Удалить элементы можно по значению используя метод *remove()* или по индексу, используя метод *pop()*. Метод *pop()* возвращает удаленный элемент. Если в метод *pop()* не передать параметр, то будет удален последний элемент.

```
>>> s = [1,2,3]
>>> s.append(1) # добавляем элемент в конец списка
>>> s
[1, 2, 3, 1]
>>> s.remove(1) # удаляем первый элемент со значением 1
>>> s
[2, 3, 1]
>>> s.pop(1) # удаляем элемент с индексом 1. метод вернет его значение
3
>>> s
```

```
[2, 1]
```

Изменять элементы списка можно, например, обратившись к ним по индексу.

```
>>> s[0] = [1, 2] # меняем значение нулевого элемента
>>> s
[[1, 2], 1]
```

Для списков, которые состоят из элементов *одного* типа, доступна сортировка с помощью методы *sort()*. Пример:

```
>>> L = [12, 19476, 0, -1213]
>>> L.sort()
>>> L
[-1213, 0, 12, 19476]
```

1.26 Генераторы списков

Иногда бывает нужно создать список, заполненный по определенному условию. Например, состоящий из упорядоченных элементов от 1 до 10. Можно, конечно, создать пустой список и в цикле добавить в него необходимы элементы, но Python предоставляет более элегантное решение для этого: генераторы списков.

Например, создать список с элементами от 1 до 10 с помощью генератора `[i for i in range(1, 11)]` можно таким способом всего в одну строчку:

```
>>> s = [i for i in range(1, 11)]
>>> s
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Синтаксис генератора устроен следующим образом: сначала описывается выражение (в данном примере просто i), потом указывается над каким элементом это выражение применяется (в данном случае тоже просто i), а потом - откуда этот элемент должен быть получен. В данном случае из *range()*. В выражении могут быть использованы любые другие необходимые переменные.

Таким образом, если в качестве первого элемента находится выражение, то можно создать еще более сложный список.

```
>>> s = [[i**2] for i in range(1,11)]
>>> s
[[1], [4], [9], [16], [25], [36], [49], [64], [81], [100]]
```

В данном случае выражение определяет, что каждый элемент списка - это тоже список из одного элемента, который является значением *i*, возведенным в квадрат.

Подобным образом можно создавать и многомерные списки. Например,

можно создать “матрицу” 3×3, инициализировав её нулями:

```
>>> matrix = [[0 for x in range(3)] for y in range(3)]
>>> matrix
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

1.27 Доступ к элементам списка

Доступ к элементам списка можно осуществлять по индексу способом, рассмотренным ранее - с использованием квадратных скобок *[index]*. Например, доступ к элементу с индексом 5 списка *s* можно получит следующим образом:

```
>>> s = ['A', 'B', 'C', 'D']
>>> s[5]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list index out of range
>>> s[1]
'B'
```

По индексу возможно не только получение значения элемента, но и изменение элемента, например:

```
>>> s = ['A', 'B', 'C', 'D']
>>> s[3] = [1, 2, 3]
>>> s
['A', 'B', 'C', [1, 2, 3]]
```

Важной особенностью является то, что индексы списка могут иметь отрицательное значение (как и в случае со строками см. раздел “1.16 Доступ к элементам строки по индексу”). В этом случае можно получить значение элемента по индексу начиная с конца списка таким образом, что последний элемент имеет индекс *-1*. Для списка *s*, введенного ранее, каждый элемент будет иметь 2 индекса как показано в табл. 1.15:

Таблица 1.15 – Операции проверки вхождения

Элемент	'A'	'B'	'C'	'D'
Индекс с начала	0	1	2	3
Индекс с конца	-4	-3	-2	-1

Небольшой пример:

```
>>> s = ['A', 'B', 'C', 'D']
>>> s[3]
'D'
>>> s[-3]
'B'
```

1.28 Срезы в списках

В Python также существует возможность получить подмножество элементов списка с использованием срезов. Для списка s срез списка можно получить следующим образом: *s[i:j:k]*, где *i* определяет индекс первого элемента среза, *j-1* - индекс последнего включенного в срез элемента а *k* - шаг изменения индекса.

Некоторые или даже все из этих индексов могут быть пропущены. По умолчанию *i = 0*, *j* - количеству элементов списка, *k = 1*.

```
>>> s = [0,10,20,30,40,50,60]
>>> s[2:5]
[20, 30, 40]
```

Шаг *k* может быть и отрицательным числом, но в этом случае порядок использования *i* и *j* должен быть изменен на противоположный. Для списка из предыдущего примера:

```
>>> s[5:0:-2]
[50, 30, 10]
```

С помощью срезов можно создавать копию списка. Например:

```
>>> b = [10, [20, 30], 'Abc']
>>> a = list(b)
>>> a
[10, [20, 30], 'Abc']
```

Можно создать новый список с помощью срезов:

```
>>> a = b[:]
>>> a
[10, [20, 30], 'Abc']
```

Также для копирования списка можно воспользоваться методом *copy():*

```
>>> a = b.copy()
>>> a
[10, [20, 30], 'Abc']
```

В обоих случаях мы получим списки, которые на первый взгляд будут одинаковые. Например, такие:

```
>>> a
[10, [20, 30], 'Abc']
>>> b
[10, [20, 30], 'Abc']
```

1.29 Сравнение и другие операции над списками

Операции сравнения ==, !=, <, <=, >, >=, представленные в табл. 1.1.6, можно применять и над списками. Сравнение происходит поэлементно, включая вложенные списки. Таким образом

```
>>> [1,[2, 3],4] == [1,[2, 3],4]
```

```
True
>>> [1,[2, 3],4] == [1,[2, 5],4]
False
```

Однако, для операций сравнения существует важное ограничение, чтобы сравниваемые элементы совпадали по типу. Поэтому результат такого сравнения:

```
>>> [1,[2, 'B'],4] > [1,[2, 'A'],4]
True
```

Но такая запись:

```
>>> [1,[2, 3],4] > [1,[2, 'A'],4]
```

Вызовет ошибку TypeError: '>' not supported between instances of 'int' and 'str'

Для того, чтобы проверить входит ли элемент в список используется операция *in*:

```
>>> a = [1,2,3]
>>> 1 in a
True
```

Для того, чтобы из двух списков получить один, можно использовать операцию +:

```
>>> a = [1, 2, 3]
>>> b = [10, 20, 30]
>>> c = a + b
>>> c
[1, 2, 3, 10, 20, 30]
```

1.30 Словари. Изменяемость словарей

Помимо списков, важную роль играют словари. Словарь - неупорядоченная коллекция (о коллекциях подробнее - в “Главе 4. Введение в алгоритмы и структуры данных”), доступ к элементам которой осуществляется по ключу. В других языках или источниках вы также можете услышать, что такую структуру хранения данных могут называть *ассоциативным массивом* или *map*. В Python для словаря используются фигурные скобки {}. Например, создать пустой словарь можно так:

```
>>> d = {}
>>> d
{}
```

В фигурные скобки { } можно передать пары <ключ>:<значение> через запятую. Например:

```
>>> countries = {"Russia": "Moscow", "Finland": "Helsinki", "USA": "New York", "USA": "Washington, D.C."}
>>> countries
```

```
{'Russia': 'Moscow', 'Finland': 'Helsinki', 'USA': 'Washington, D.C.'}
```

Сам словарь является изменяемым объектом и значения в нем могут быть любыми изменяемыми или неизменяемыми объектами, но для ключей существуют строгие ограничения. Ключи в словаре должны быть неизменяемыми (если говорить точнее - хэшируемыми, об этом подробнее говорится в “Главе 4. Введение в алгоритмы и структуры данных”) уникальными объектами.

Доступ к значению по ключу можно получить используя квадратные скобки. Например, для словаря *countries* это можно сделать следующим образом:

```
>>> countries["Russia"]
'Moscow'
```

Аналогично можно и изменить или добавить новое значение по ключу:

```
>>> countries["Australia"] = "Canberra"
```

Также, словарь является итерируемым объектом и позволяет перебрать все его ключи следующим способом:

```
>>> countries = {"Russia": "Moscow",
"Finland": "Helsinki",
"USA": "New York",
"USA": "Washington, D.C."}
>>> for country in countries:
...     print(country)
...
Russia
Finland
USA
```

Длину словарей, также как и для уже изученных строк, списков и частично изученных кортежей, можно узнать с помощью функции *len(obj)* следующим образом:

```
>>> countries = {"Russia": "Moscow", "Finland": "Helsinki", "USA": "Washington, D.C."}
>>> len(countries)
3
```

1.30.1 Другие способы создания словаря

Помимо приведенного выше способа создания словаря, его также можно создать из списка кортежей:

```
>>> d = dict([(1,1),(2,4),(3,9)])
>>> d
{1: 1, 2: 4, 3: 9}
```

Или из списка списков:

```
>>> d = dict([[1,1],[2,4],[3,9]])
>>> d
{1: 1, 2: 4, 3: 9}
```

Если же передать функции простой (одноуровневый) список, то можно столкнуться со следующей ошибкой:

```
>>> d = dict([1, 2, 3, 4, 5])
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: cannot convert dictionary update sequence element #0 to a sequence
```

Но возможности работы со словарями в Python таковы, что и из простого списка можно получить словарь. Для этих целей доступен метод *fromkeys()*. Работает он следующим образом:

```
>>> l = [1, 2, 3, 4, 5]
>>> d = dict.fromkeys(l)
>>> d
{1: None, 2: None, 3: None, 4: None, 5: None}
```

Метод *fromkeys()* позволяет также установить одинаковые значения ключам:

```
>>> d = dict.fromkeys(l, 0)
>>> d
{1: 0, 2: 0, 3: 0, 4: 0, 5: 0}
```

1.30.2 Методы словарей

При работе со словарями может возникнуть необходимость получения элементов словаря. Это можно сделать с помощью метода *items()*, как показано далее:

```
>>> d = dict([[1,1],[2,4],[3,9]])
>>> d.items()
dict_items([(1, 1), (2, 4), (3, 9)])
```

Обратите внимание, что метод *items()* возвращает объект типа *dict_items*. Чтобы в дальнейшем работать с элементами словаря привычным образом, можно привести полученный результат к типу *list*:

```
>>> l_dict = list(d.items())
>>> l_dict
[(1, 1), (2, 4), (3, 9)]
```

Обратите внимание, что каждый элемент полученного списка - кортеж:

```
>>> type(l_dict[0])
<class 'tuple'>
```

При работе со словарями можно получить ключи словаря, используя метод *keys()*. Так как метод *keys()*, как и в случае с методом *items()*, возвращает объект особо типа - *dict_keys*, для наглядности результатов

используется преобразование к типу *list*:

```
>>> d.keys()
dict_keys([1, 2, 3])
>>> list(d.keys())
[1, 2, 3]
```

Для получения значений словаря, аналогично, существует метод *values()*. Так как же как и методы *keys()* и *items()*, метод *values()* возвращает объект особо типа - *dict_values()*, для наглядности результатов воспользуемся преобразованием к типу *list*:

```
>>> d.values()
dict_values([1, 4, 9])
>>> list(d.values())
[1, 4, 9]
```

Отметим, что преобразование результатов методов *items()*, *keys()*, *values()* возможно и к другому типу коллекций, например, преобразование к кортежу. Наиболее частое применение методов *items()*, *keys()*, *values()* можно встретить в циклах при обходе словарей.

Обратим внимание на идентичность поведения следующих двух фрагментов кода с обходом словаря:

```
>>> for elem in d:
...     print(elem)
1
2
3
>>> for elem in d.keys():
...     print(elem)
1
2
3
```

Для объединения словарей можно использовать метод *update()*. Этот метод объединяет ключи и значения одного словаря с ключами и значениями другого, перезаписывая значения с одинаковыми ключами.

```
>>> d = {'a' : 1, 'b' : 2, 'c' : 3}
>>> d.update({'a' : 11, 'bb' : 2, 'c' : 33})
>>> d
{'a': 11, 'b': 2, 'c': 33, 'bb': 2}
```

По аналогии со списками, для словарей доступен метод pop(key), который возвращает значение из словаря по ключу, исключая данную пару из словаря. Рассмотрим пример:

```
>>> d = {'a' : 1, 'b' : 2, 'c' : 3}
>>> d
```

```
{'a': 1, 'b': 2, 'c': 3}
>>> d.pop('b')
2
>>> d
{'a': 1, 'c': 3}
```

Для извлечения элементов из словаря также существует метод popitem(), который возвращает произвольную пару, извлекая ее:

```
>>> d = {'a' : 1, 'b' : 2, 'c' : 3}
>>> d.popitem()
('c', 3)
>>> d.popitem()
('b', 2)
>>> d.popitem()
('a', 1)
>>> d.popitem()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
>>> d
{}
```

Для получения значения по ключу можно воспользоваться как ранее рассмотренной операцией квадратные скобки [ ], так и специальным методом *get()*:

```
>>> d = {'a' : 1, 'b' : 2, 'c' : 3}
>>> d['b']
2
>>> d.get('a')
1
```

В случае, если будет совершена попытка обратиться по несуществующему ключу с использованием квадратных скобок [ ], получим такой результат:

```
>>> d['d']
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'd'
```

В случае передачи в метод *get()* несуществующего ключа в качестве результата будет получено *None*:

```
>>> d.get('d')
None
```

Метод *get()* интересен еще тем, что ему можно передать второй параметр - значение по умолчанию (*default*), которое будет возвращено в случае отсутствия ключа, а именно:

```
>>> d.get('d', -1)
```

```
-1
```

1.31 Генераторы словарей

По аналогии с генераторами списков, в языке Python доступны генераторы словарей. Они очень похожи как с точки зрения использования, так и с точки зрения назначения - их также используют при необходимости инициализировать словарь элементами по определенному правилу.

Например, мы можем создать словарь, где ключами будут целые числа в диапазоне от 0 до 5, а значениями - квадраты этих чисел

```
>>> squares = {i: i ** 2 for i in range(6)}
>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Единственное отличие генератора словаря от генератора списка заключается в том, что в генераторе словаря указывается через двоеточие пара ключ-значение, а не один элемент как в случае со списками.

1.32 Многомерные словари

Мы уже обсуждали ранее, что в словаре ключ должен быть неизменяемым объектом, а значение может быть изменяемым. Других ограничение на значение нет, поэтому значением может быть в том числе другой словарь. Пусть у нас есть словари, ключи которых целые числа, а значения - эти же числа прописью на разных языках:

```
>>> rus = {1 : 'один', 2 : 'два', 3 : 'три'}
>>> eng = {1 : 'one', 2 : 'two', 3 : 'three'}
```

Было бы удобно объединить эти словари и уметь получать написания чисел в зависимости от языка. Для этого, можно создать словарь, где в качестве ключа был бы признак языка (например, сокращенное написание), а в качестве ключа - словарь для перевода. На примере это выглядит следующим образом:

```
>>> translate = {'rus' : {1 : 'один', 2 : 'два', 3 : 'три'}, 'eng' : {1 : 'one', 2 : 'two', 3 : 'three'}}
```

Пользоваться таким словарем можно также как и одномерным. При обращении по ключу, мы сможем получить доступ к значению

```
>>> translate['rus']
{1: 'один', 2: 'два', 3: 'три'}
```

А так как значение тоже словарь, обратиться по ключу мы можем и в нем

```
>>> translate['rus'][2]
'два'
```

Это же справедливо и для большей глубины вложенности.

1.33 Операции над словарями

Помимо операций добавления и получения значения по ключу с которыми мы уже успели познакомиться, над словарем доступны и другие операции. Давайте рассмотрим наиболее часто используемые. Например, проверить вхождение ключа в словарь можно используя операция проверки вхождения *in***:**

```
>>> d = {'a' : 1, 'b' : 2, 'c' : 3}
>>> 'a' in d
True
```

Операция проверки идентичности применима к словарям:

```
>>> d is rus
False
```

Другие операции +, *, /, // и пр. к словарям не применяются.

1.34 Операции сравнения и логические операции над словарями

В языке Python над словарями возможны только операции сравнения такие как равенство или неравенство. Остальные операции сравнения недопустимы.

Равными считаются словари, содержащие одинаковый набор пар ключ-значение.

```
>>> d1 = {'a' : 1, 'b' : 1}
>>> d2 = {'a' : 1, 'b' : 1}
>>> d2 == d1
True
>>> d2['b'] = 2
>>> d2 != d1
True
```

Но логические операции *and* и *or* над словарями применять можно:

```
>>> d1 = {'H' : 24, 'D' : 32, 'V' : 18}
>>> d2 = {'A' : 10, 'B' : 11, 'H' : 24}
>>> d1 and d2 # возвращает второй аргумент - словарь d2
{'A': 10, 'B': 11, 'H': 24}
>>> d2 and d1 # возвращает второй аргумент - словарь d1
{'H': 24, 'D': 32, 'V': 18}
>>> d1 or d2 # возвращает первый аргумент - словарь d1
{'H': 24, 'D': 32, 'V': 18}
>>> d2 or d1 # возвращает первый аргумент - словарь d2
{'A': 10, 'B': 11, 'H': 24}
```

Логическое отрицание not:

```
>>> bool(d2)
True
>>> not d2
```

```
False
```

1.35 Методы split и join

Методы *split()* для строк и *join()* для списков представляют особый интерес. Данные методы работы с последовательностями позволяют получить другую последовательность, как показано в примерах далее.

Синтаксис метода *split()*:

<строка>.split(<разделитель>)

Этот метод разбивает строку на части, возвращая список элементов-строк, полученный путем разделения исходной строки <строка> по символу или строке <разделителю>, переданному в качестве аргумента в метод *split()*. Сам разделитель при этом в эти строки не входит. Пример использования, когда в качестве разделителя выбран пробел:

```
>>> string_var = "some text"
>>> list_var = string_var.split(" ")
>>> list_var
['some', 'text']
```

Результатом выполнения данной программы будет список *['some', 'text']*. Следует отметить, что пробельные символы в данной функции используются в качестве разделителя в случаях, даже когда мы вызываем функцию без аргументов, например:

```
>>> list_var = string_var.split()
>>> list_var
['some', 'text']
```

Рассмотрим пример с методом *split(),* когда в строке встречается множество разделителей подряд:

```
 >>> string_var = ' s   o m e   t   e x t! '
>>> string_var.split()
['s', 'o', 'm', 'e', 't', 'e', 'x', 't!']
```

Несмотря на большое количество подряд идущих пробелов в строке, в итоговом списке нет элементов - пустых строк.

Метод *join()* выполняет обратную к методу *split()* операцию объединения списка в строку. Синтаксис метода *join()*:

<разделитель>.join(<объект>)

Он конкатенирует строки в переданном ему итерируемом объекте <объект> с помощью разделителя-строки, для которой этот метод вызывается:

```
>>> '!'.join(['Hi', 'qwe', 'aaa'])
'Hi!qwe!aaa'
>>> '---'.join('123456789')
```

```
'1---2---3---4---5---6---7---8---9'
```

В качестве итерируемых объектов могут выступать объекты уже знакомых нам коллекций : списки, кортежи, словари, строки и др.

Если требуется склеить все элементы итерируемого объекта без разделителя между ними, можно использовать пустую строку, а именно:

```
>>> ''.join(['Hi', 'qwe', 'aaa'])
'Hiqweaaa'
```

А что будет, если в списке среди элементов будут не только строки? Рассмотрим пример:

```
>>> L = [1, '2', 3, 'AbD']
>>> ''.join(L)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int found
```

Ошибка говорит о том, что интерпретатор ожидал от нулевого элемента тип *str*, а оказался - *int*. Такие списки нельзя преобразовать в строку с помощью метода *join()*.

В заключение, хотим добавить, что хотя *split()* и *join()* и являются методами строк, но они тесно связаны со списками и важны при работе с ними.

1.36 Динамическая типизация. Переменные, ссылки и объекты

Язык Python - динамически типизированный язык. В этом разделе мы постараемся разобраться, что это значит и как мы можем это использовать.

В языке Python вы можете написать следующий код, который не вызовет ошибку, в отличие, например, от языка С:

```
>>> a = 10
>>> a = [1, 2, 3]
>>> a = 'hello'
```

Здесь интересны два момента. Во-первых, мы не указываем тип явно, т.е. не пишем

```
str a = 'hello'
```

Во-вторых, мы можем присвоить переменной a сначала число, потом список, потом строку, не боясь вызвать ошибку.

Это является следствием того, что типы данных в языке Python автоматически определяются во время выполнения программы, а не в результате объявлений в программном коде, как например, в языке С. Языки, подобные Python, называются *динамически типизированными*. Языки, подобные С, где типы переменных устанавливаются на этапе компиляции

(этап «перевода» программы с языка высокого уровня на понятный компьютеру язык нулей и единиц (байт-код)), называются *статически типизированными*. Также это означает, что на этапе выполнения программы у компилятора уже есть информация, какая переменная к какому типу относится.

Рассмотрим более подробно на примере инициализации переменной.

Пусть наш код состоит из одной строки:

```
>>> a = 10
```

После того, как интерпретатор дошел до ее выполнения, он создал объект 10, затем создал переменную a (поскольку она встречается в коде впервые). При этом сама переменная a не хранит информации о типе, тип есть только у объекта 10. После этого в переменную a записывается ссылка на объект 10. Говорят, что переменная ссылается на объект или переменная хранит ссылку на объект (см. рис. 1.1).

Рисунок 1.1 - Связь переменной и объекта

Когда переменная участвует в выражении, например:

```
>>>  b = a * 2
```

ее имя замещается объектом, на который она в настоящий момент ссылается, независимо от того, что это за объект.

Таким образом, любая переменная, которую вы используете в своей программе, хранит ссылку на определенный объект. При этом вы можете изменить эту ссылку на объект другого типа. Именно поэтому возможна следующая последовательность операций:

```
>>> a = 10
>>> a = [1, 2, 3]
>>> a = 'hello'
```

Перед использованием переменной её обязательно нужно проинициализировать, иначе возникнет ошибка:

```
>>> a = c + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

Если вы знакомы с указателями в языке C, вы наверняка заметили, что между ссылками в языке Python и указателями в языке C много общего.

Ссылки в языке Python реализованы как указатели, но при использовании они автоматически разыменовываются, поэтому у вас нет возможности, как в языке C, работать напрямую с адресом объекта.

1.37 Разделяемые ссылки

Рассмотрим, как действует интерпретатор, когда выполняется следующий код:

```
>>> a = 10
>>> b = a
```

При выполнении второй строки кода создается переменная `b`, в которую записывается ссылка на объект 10 (не на переменную `a`!). Это происходит, поскольку при использовании переменная `a` замещается объектом, на который она ссылается, т.е. 10. Это приводит к тому, что и `a`, и `b` ссылаются на один и тот же объект 10. В языке Python это называется *разделяемая ссылка:* когда несколько переменных хранят ссылку на один и тот же объект (см. рис. 1.2).

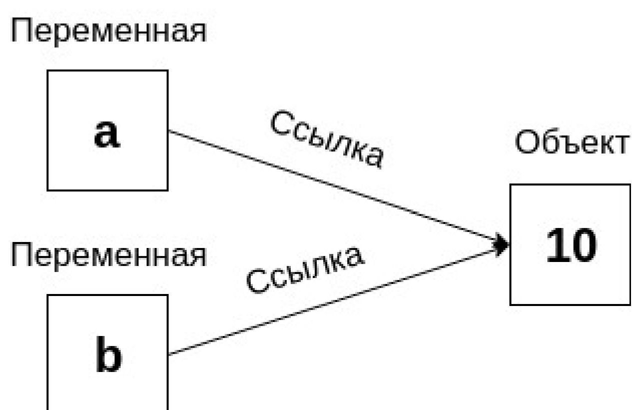


Рисунок 1.2 - Разделяемая ссылка

Что произойдет, если мы решим, что переменная `a` должна ссылаться на другой объект?

```
>>> a = 10
>>> b = a
>>> a = 20
```

Поскольку происходит операция присваивания, интерпретатор создаст объект 20 и запишет ссылку на него в существующую переменную `a` (см. рис. 1.3).

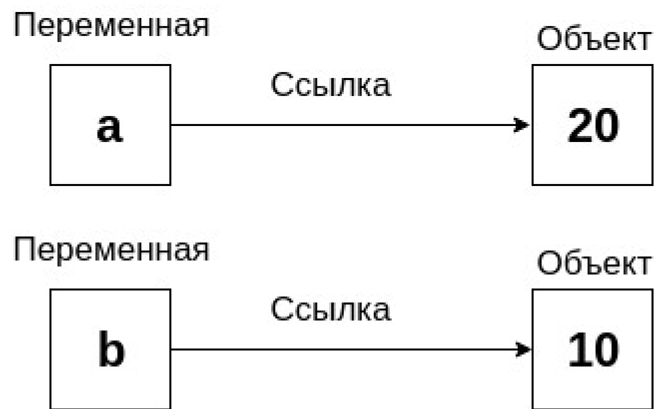


Рисунок 1.3 - Изменение ссылки одной из переменной в разделяемой ссылке

Таким образом, при выводе `a` и `b`, мы получим следующий результат:

```
>>> a
20
>>> b
10
```

1.38 Неизменяемые и изменяемые объекты

В языке Python объекты делятся на два вида: изменяемые и неизменяемые. Ранее этот вопрос поднимался при обсуждении строк (“1.15 Строки str”, “1.21.1 Неизменяемость строк на примере работы методов”), списков (“1.25 Изменяемость списков”) и словарей (“1.30 Словари. Изменяемость словарей”). Как уже отмечалось, к неизменяемым объектам относятся значения числовых типов, строковые значения, значения логического типа, None и объекты типа кортеж. Неизменность объекта гарантирует нам, что не существует способа изменить сам объект. Рассмотрим на примере объекта True типа bool.

Предположим, что мы написали следующий код:

```
>>> a = True
>>> b = a
```

True - неизменяемый объект, значит, нам гарантируется, что мы можем использовать переменные `a` и `b`, не опасаясь, что само значение может измениться с True на False. Мы можем изменить ссылку, которая хранится в переменной `a` на ссылку на объект False:

```
>>> a = False
```

но это никак не меняет объект True.

Теперь давайте рассмотрим изменяемые объекты и их отличие от неизменяемых. К изменяемым объектам из рассмотренных нами в языке Python относятся списки и словари. Рассмотрим изменяемость объектов на примере списка.

Список можно представить в памяти следующим образом (см. рис. 1.4):

```
>>> L = [10, 20, 30]
```

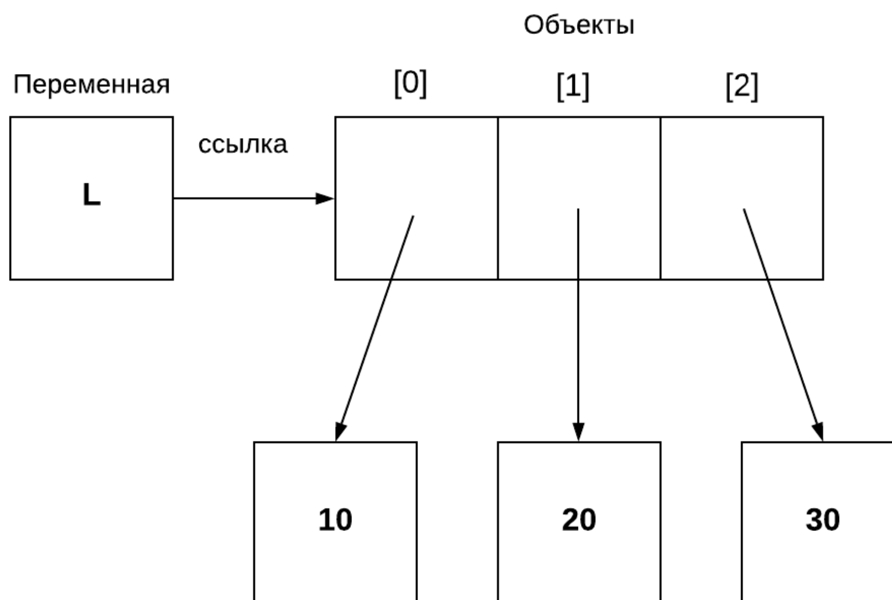


Рисунок 1.4 - Представление списка

Фактически L[0] хранит ссылку на объект 10, L[1] хранит на 20, L[2] - на 30.

Предположим, нам следует выполнить следующую операцию:

```
>>> L[0] = 11
```

В таком случае происходит следующее (см. рис. 1.5):

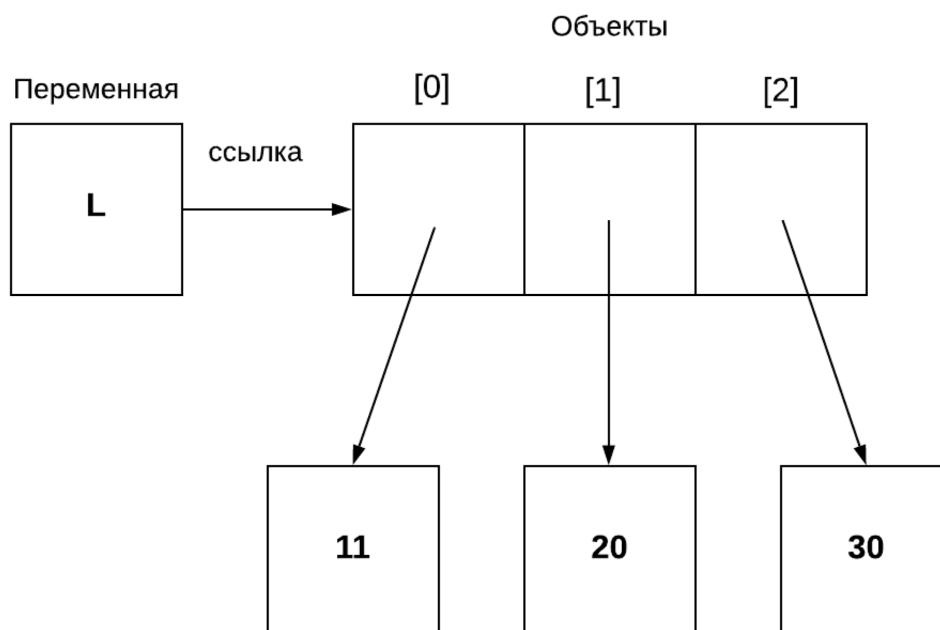


Рисунок 1.5 - Результат выполнения операции над списком

Таким образом, операция изменила непосредственно объект списка.

Рассмотрим разделяемые ссылки на примере списков:

```
>>> L1 = [1, 2, 3, 4, 5]
>>> L2 = L1
>>> L1[2] = 'hello!'
>>> L1
[1, 2, 'hello!', 4, 5]
>>> L2
[1, 2, 'hello!', 4, 5]
```

Изменилась не только переменная L1, но и переменная L2. Важно понимать, что такое поведение является стандартным для всех *изменяемых объектов* языка Python.

Методы неизменяемых объектов не вносят изменения в объекты, они лишь возвращают новый объект, который впоследствии можно использовать. Поэтому не забывайте присваивать переменной результат, который возвращает метод.

Например, строки являются неизменяемым объектом. Метод *replace()* не изменяет объект строки, а создает новый объект, который нужно не забыть сохранить в переменной:

```
>>> st = 'qwerty qwerty'
>>> st.replace(' ',  --')
'qwerty--qwerty'
>>> st # объект, на который ссылается st, не меняется
'qwerty qwerty'
>>> res = st.replace(' ', '--') # сохранили новый объект в res
>>> res
'qwerty--qwerty'
```

1.39 Функции. Общие сведения

Мы уже использовали функции, например, когда хотели узнать длину строки:

```
len("Hello, World!")
```

Напомним, что *len()* - это функция языка Python, которая выводит длину переданного ей объекта на консоль.

Рассмотрим, как создать (определить) свою собственную функцию на языке Python.

Синтаксис определения функции:

def <имя_функции>(<аргументы>):

<инструкции>

…
<инструкции>
return <значение_1>, … <значение_n> # необязательная часть

Здесь:

- def - инструкция, которая сообщает интерпретатору, что начинается определение функции;
- <имя_функции> - наименование функции, например, print;
- <аргументы> - параметры, которые передаются в функцию и которые вы можете использовать в функции;
- <инструкции> - код, который выполняется при вызове функции. Обратите внимание, что он расположен после отступов с начала строки, как и у любой составной инструкции (см. "1.1 Запуск программы");
- <значения> - это данные, которые передаются обратно в программу, где был совершен вызов функции. Здесь необходимо обратить внимание, что данная инструкция не обязательна.
- *return* - оператор, который возвращает данные и управление в место вызова функции.

Рассмотрим пример определения и вызова функций.

Определение	**Вызов**

def magic_sum(a, b):
    sum = a + b
    return sum
	s = magic_sum(10, 25) # s == 35
	s = magic_sum(1.0, 2.5) # s == 3.5
	s = magic_sum("Hello ", "World!")
	# s == "Hello, World!"

Обратите внимание, что мы можем передавать в функцию *magic_sum* объекты любой природы, которые поддерживают операцию сложения: строки, числа, списки - поскольку в языке Python не требуется явно определять, с каким типом данных вы хотите работать. Python не накладывает ограничений на типы аргументов функции, в связи с чем не гарантируется корректная работа функций на объектах, для типа которых функции не предназначены.

1.40 Аргументы функции

Переменную называют *локальной*, если она объявлена внутри блока кода

(например, внутри функции) и не видна за ее пределами. Например:

```
>>> def func(arg):
...     arg = 0
```

переменная *arg* является локальной и не существует за пределами функции:

```
>>> arg
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'arg' is not defined
```

Аргументы в функцию передаются посредством операции *присваивания* объектов локальным переменным функции. Рассмотрим, что это означает на практике.

```
>>> def func(arg):
...   arg = 0
>>> x = 100
>>> func(x)
>>> x
100
```

При вызове функции с аргументом x равным 100, интерпретатор присваивает ссылку на объект 100 локальной переменной *arg*. После этого значение локальной переменной *arg* меняется на 0, значение x остается неизменным.

Рассмотрим пример с изменяемым объектом:

```
>>> def func(arg):
...         arg.append(5)
>>> x = [1, 2, 3, 4]
>>> func(x)
>>> x
[1, 2, 3, 4, 5]
```

Здесь мы наблюдаем следующую ситуацию: переменная x и локальная переменная функции *func() arg разделяют* объект список, который меняется при вызове функции *append()*, т.е. в данном случае мы опять имеем дело с разделяемой ссылкой.

Что же произойдет при выполнении следующего блока кода?

```
>>> def func(arg):
...     arg = [1, 2]
>>> x = [1, 2, 3, 4]
>>> func(x)
```

В данном случае локальная переменная *arg* потеряет ссылку на список [1, 2, 3, 4] и будет хранить ссылку на новый список [1, 2], что не изменит объект [1, 2, 3, 4], а значит вывод переменной x останется прежним:

```
>>> x
```

```
[1, 2, 3, 4]
```

1.41 Передача аргументов в функцию

В языке Python мы не можем иметь две функции с одинаковым именем, но разным числом аргументов, как например, в языке C++, но можем указывать значения по умолчанию в определении функции.

Синтаксис:

def <имя_функции>(<аргумент>$_1$ = <значение>$_1$, …<аргумент>$_n$ = <значение>$_n$):
<инструкции>
…
<инструкции>

Рассмотрим использование значений по умолчанию на примере:

```
>>> def magic_sum(a, b, c=10):
...         sum = (a + b)*c
...         return sum
```

В данном случае значение по умолчанию имеет переменная c, равная 10. Это означает, что если не указать значение этого аргумента c при вызове, оно будет равным 10:

```
>>> magic_sum(1, 1)
20
```

также можно его указать и вызвать функцию, как обычно:

```
>>> magic_sum(1, 1, 5)
10
```

Аргументы со значением по умолчанию должны следовать строго после обычных аргументов функции.

При вызове функции существует похожий синтаксис для указания интерпретатору имен аргументов безотносительно их позиций. Такой вызов называют вызовом функции с именованными аргументами.

Синтаксис:

<имя_функции>(<аргумент>$_1$=<значение>$_1$, …, <аргумент>$_n$=<значение>$_n$)

Обратите внимание, что в отличие от предыдущего пункта, здесь описан *вызов* функции, а не ее *определение*.

Рассмотрим вызов функции magic_sum с использованием именованных аргументов:

```
>>> magic_sum(a=1, b=3, c=111)
444
```

Мы можем менять порядок следования аргументов:

```
>>> magic_sum(c=111, b=3, a=1)
444
```

В одном вызове функции могут сочетаться именованные и позиционные

(т.е. неименованные, стоящие на той же позиции, что и в определении функции) аргументы, но позиционные всегда идут перед именованными.

1.42 Ввод и вывод данных в Python

Изучив основные типы данных, операции и функции, перейдем к рассмотрению функций, без которых невозможно взаимодействие пользователя и программы – это функции ввода и вывода в командную строку. В языке Python для этого используются две встроенные функции: *input()* и *print()*.

На примере использования функции *print()* можно посмотреть, как работать с именованными аргументами и аргументами по умолчанию.

Полный прототип функции *print()* имеет вид:

print([arg1 , ...][, sep=” ”][, end=”\n”][, file=sys.stdout])

Здесь:

- *arg1* - объект, который нужно вывести. Объектов может быть несколько;
- *sep* - это строка-разделитель при выводе аргументов в командную строку. Значение по умолчанию - пробел;
- *end* - строка, добавляемая в конец вывода в командную строку. Значение по умолчанию - символ перевода строки '\n';
- *file* - указание на используемый поток вывода, по умолчанию - *stdout*.

Обратите внимание, что функция *print()* возвращает константу *None*.

Рассмотрим примеры использования функции *print()*. Фрагмент кода ниже выведет строку "Hello!":

```
>>> print("Hello!")
Hello!
```

При синтаксисе, описанном выше, *print()* будет добавлять после выведенной строки символ перевода строки '\n' (курсор будет перенесен на новую строку), однако это можно изменить. Для этого достаточно преобразовать фрагмент кода следующим образом:

```
>>> print("Hello!", end="")
... print("World")
Hello!World
```

Функции *print()* можно одновременно задать несколько аргументов, например, несколько строк для вывода:

```
>>> print("Hello", "World")
```

После выполнения данной программы они будут выведены через пробел:

```
Hello World.
```

Результата, полученного в примере выше, можно добиться, не указывая аргументы. В таком случае аргументы будут иметь значение по-умолчанию. Например:

```
>>> print("Hello", end=' ')
... print("World")
```

В результате будет выведено:

```
Hello World
```

Функция *print()* работает и с другими объектами языка Python, например:

```
>>> a = 12
>>> b = 8.0
>>> print(a)
... print(b)
... print('a + b = {}'.format(a + b))
12
8.0
a + b = 20.0
```

Для ввода данных используется функция *input()*. Данная функция может быть вызвана как с аргументом, так и без (как в случае с необязательным аргументом *end* для функции *print()*). В случае с функцией *input()* данный аргумент обозначает текст приглашения для ввода пользовательских данных, а именно:

```
>>> a = input("Приглашение: ")
Приглашение: >? Hello
>>> a
'Hello'
```

В таком случае пользователю сначала выводится текст "Приглашение: ", сразу после которого можно будет вводить данные. После того, как пользователь введет данные и нажмет клавишу *Enter*, функция *input()* вернет полученную строку в переменную *a*.

Вызов функции *input()* без параметра:

```
>>> a = input()
>? Hello
>>> a
'Hello'
```

Важной особенностью *input()* является то, что возвращаемое ей значение имеет *строковый тип*. Поэтому, если ваша программа подразумевает ввод числовых значений через *input()*, то им потребуется дополнительная обработка, например, приведение типов. Кроме того, обратите внимание: с

помощью *input()* возможен только однострочный ввод!

1.43 Области видимости переменных

Область видимости - это место в коде, где определяются переменные и где они могут быть найдены. Пространством имён называется некоторое абстрактное место, где находятся логически сгруппированные имена (переменные, функции, классы и т.д.).

Любая функция содержит в себе пространство имён: переменные, определяемые внутри функции, видны только в пределах этой функции. Такие переменные называются *локальными*, поэтому говорят, что функции образуют локальную область видимости.

Если присваивание производится за пределами всех функций, переменная является *глобальной* для всего модуля. Таким образом, модули образуют глобальную область видимости.

Одинаковые имена внутри функции и за ее пределами не создают конфликта, поскольку находятся в разных областях видимости. Поэтому, данный код не изменит переменную *x*:

```
>>> def func():
...     x = 525
...
>>> x = 797
>>> func()
>>> x
797
```

Для того, чтобы изменить объявленную в модуле переменную внутри функции, необходимо использовать инструкцию *global*:

```
>>> def func():
...     global x
...     x = 525
...
>>> x = 797
>>> func()
>>> x
525
```

Инструкция *global* позволяет изменять глобальную переменную, при этом доступ к переменной внутри функции есть и без этой инструкции:

```
>>> def f():
...     print(x)
...
>>> x = 797
>>> f() # на консоль будет выведено 797
```

Стоит отметить, что глобальные переменные лучше не использовать без крайней необходимости.

В языке Python вы не можете объявить переменную таким образом, чтобы она стала видна всем модулям, но вы можете объявить переменную в модуле и импортировать его.

1.44 Идентичность объектов

Наряду с использованием операцией проверки равенства (==) может встречаться использование операции проверки идентичности *is*. Операция == проверяет равенство значений двух объектов, а операция *is* проверяет равенство ссылок, которые хранятся в переменных, т.е. проверяет, расположены ли объекты, на которые ссылаются переменные, по одному и тому же адресу в памяти.

Проиллюстрируем это на примерах (см. рис. 1.6):

```
>>> x = [1, 2, 3]
>>> y = x
>>> x is y
True
```

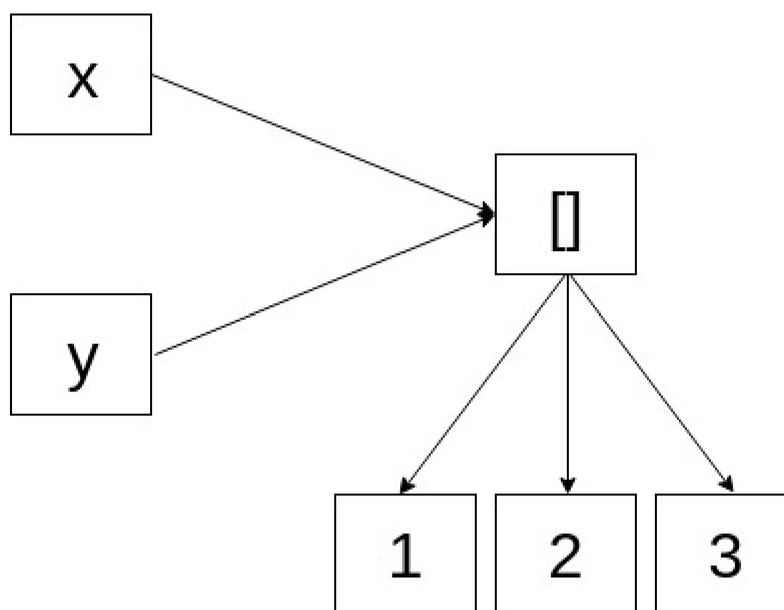


Рисунок 1.6 - Операция проверки идентичности

При этом, если x и y будут ссылаться на разные объекты с одинаковым содержимым, результат работы операции *is* будет False:

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> x is y
False
```

Следующий пример проиллюстрирует, что иногда два, казалось бы,

разных объекта могут оказаться одним объектом:

```
>>> x = 10
>>> y = 10
>>> x == y
True
>>> x is y
True
```

Такой механизм называется кэширование - интерпретатор с целью оптимизации кэширует и повторно использует небольшие числа и строки. Если запустить программу выше на более длинных числах x и y, результат будет ожидаемым:

```
>>> x = 102030
>>> y = 102030
>>> x == y
True
>>> x is y
False
```

1.45 Модули

До настоящего времени мы использовали только интерактивный режим для запуска программ. Как уже упоминалось ранее, в Python можно сохранить весь программный код в файл и запустить его. Такой файл в языке Python называется модуль.

Вы можете использовать существующие модули в своих программах, для этого вам необходимо их импортировать. Для примера рассмотрим модуль math, который содержит имена (т.е. переменные и функции) для работы с числами:

```
>>> import math
```

После импорта модуля вы можете использовать все имена, которые в нем определены, используя следующий синтаксис:

<имя_модуля>.<имя>

```
>>> math.pi
3.141592653589793
>>> math.sqrt(4)
2.0
```

- pi - это переменная, определенная в модуле math, которая хранит число Пи;
- *sqrt()* - это функция, определенная в модуле math, которая возвращает квадратный корень из неотрицательного числа.

Также в Python есть синтаксис для импорта отдельных имен из модуля:

from <имя_модуля> импорт <имена>

После импорта таким способом вы можете использовать импортированные имена без указания имени модуля:

```
>>> from math import factorial
>>> factorial(4)
24
```

- *factorial()* - это функция, определенная в модуле math, которая находит факториал целого неотрицательного числа.

Если вы хотите импортировать несколько имен, перечислите их через запятую:

```
>>> from math import factorial, pi, sqrt
```

1.46 Импорт собственных модулей

В “1.3 Объекты в Python” упоминали, что у объектов есть поля и методы, которые обычно называют атрибутами. Модуль в языке Python тоже является объектом типа *module*, у которого есть свои атрибуты.

```
>>> import math
>>> type(math)
<class 'module'>
```

В том числе, у любого модуля Python есть поле “__name__”.

```
>>> math.__name__
math
```

Когда мы импортируем модуль в интерактивном режиме или в другой модуль, поле “__name__” у импортированного модуля всегда показывает его имя. Однако, если вывести это поле у модуля, который мы запускаем, мы увидим, что значение поля “__name__” будет другим:

Листинг 1.1 – Файл test_module.py

```
print(__name__)
Запуск:
python3 test_module.py
__main__
```

Мы видим, что атрибут “__name__” файла test_module.py равен “__main__”.

Чем нам поможет информация, которую мы только что узнали? Дело в том, что код модуля при его импорте выполняется и это не всегда то поведение, которое нам требуется. Проиллюстрируем на примере:

Листинг 1.2 - Файл test_module.py

```
print('Hello!')
```

Листинг 1.3 - Файла main.py

```
import test_module
```

```
Запуск:
python3 main.py
Hello!
```

Несмотря на то, что мы запустили файл main.py, который не содержит ничего, кроме инструкции импорта, мы видим, что на консоль была выведена строка “Hello!”.

Чтобы этого избежать при импорте и оставить запуск необходимых инструкций для случая, когда модуль запускается как главный, нужно использовать следующую конструкцию:

if __name__ == ‘__main__’:

<код, который выполняется, если мы запускаем этот модуль как отдельную программу>

Подобный подход к написанию модулей является хорошим тоном.

1.47 Копирование списков

Проиллюстрировать механизм работы разделяемых ссылок можно на примере копирования списков. Ранее мы приводили пример копирования списков с использованием срезов [:] и метода *copy()*, которые позволяют получить на первый взгляд одинаковые списки:

```
>>> b = [10, [20, 30], 'Abc']
>>> a = b.copy()
>>> a
[10, [20, 30], 'Abc']
>>> b
[10, [20, 30], 'Abc']
```

Изменив один из списков, посмотрим на результат:

```
>>> a[0] = 'Qwe'
>>> a
['Qwe', [20, 30], 'Abc']
>>> b
[10, [20, 30], 'Abc']
```

Совсем другой результат мы получим, когда изменим вложенный список:

```
>>> a[1][0] = 400
>>> a
['Qwe', [400, 30], 'Abc']
>>> b
[10, [400, 30], 'Abc']
```

Как видим, значения элемента изменились в обоих списках. Связано это с тем, что по умолчанию копирование списков осуществляется поверхностно: то есть копируются только ссылки, а не объекты. Это

приводит к тому, что копирование вложенных составных объектов (таких как словари и списки) не происходит.

Чтобы создать полную копию списка, включая все вложенные элементы, копирование следует выполнять с помощью функции *deepcopy()*. Для этого необходимо предварительно импортировать модуль *copy*:

```
>>> from copy import deepcopy
>>> a = deepcopy(b)
>>> a
['Qwe', [400, 30], 'Abc']
>>> b
['Qwe', [400, 30], 'Abc']
>>> a[1][0] /= 4
>>> a
['Qwe', [100.0, 30], 'Abc']
>>> b
['Qwe', [400, 30], 'Abc']
```

Здесь уже значения элемента - вложенного списка - изменяются независимо.

1.48 Задача на попадание в интервал

Чтобы лучше понять, как работать с операциями сравнения, логическими операциями, оператором ветвления *if-elif-else*, функциями ввода-вывода, рассмотрим задачу на проверку попадания целого числа в заранее определенный интервал.

Пусть задан интервал (1, 5] U [10; ∞). Опишем этот интервал с использованием логических операций и операций сравнения. Сперва рассмотрим левую часть (до знака объединения U). Ее можно описать как показано в табл. 1.16.

Таблица 1.16 – Описание интервала (1, 5]

на русском языке	строго больше одного	и	меньше или равно пяти
на языке Python	> 1	and	<= 5

Аналогичным образом попробуем описать часть интервала справа от знака объединения U. Для начала проанализируем правую границу интервала, где фигурирует знак бесконечности ∞. Благодаря такому ограничению правая граница данного интервала не имеет смысла (т.к. все целые числа заведомо меньше бесконечности) и на языке Python запись будет выглядеть так, как показано в табл. 1.17.

Таблица 1.17 – Описание интервала [10; ∞)

на русском языке	больше или равно десяти	и	меньше бесконечности
на языке Python	>= 10		

Объединение интервалов через U равносильно логическому объединению (операция *or*). Обозначим неизвестное число через x. Таким образом, взяв для примера *x = 14*, на языке Python интервал (1, 5] U [10; ∞) можно описать как:

```
>>> x = 14
>>> (x > 1 and x <= 5) or x >= 10
True
```

Чтобы реализовать проверку попадания в интервал используем условный оператор. Пусть в случае успеха (число попало в интервал), выводится сообщение “*True*”, иначе - “*False*”. Итак, код на Python:

```
>>> if x > 1 and x <= 5 or x >= 10:
...     print('True')
... else:
...    print('False')
...
```

Считывание числа с клавиатуры реализуем с помощью функции *input()*:

```
>>> x = int(input())
```

Будьте внимательны! В случае, если пользователь введет не число, программа выдаст ошибку преобразования строки *str*, которую возвращает функция *input()*, в тип *int*. Во избежание ошибки разделим считывание строки с клавиатуры и преобразование в тип *int*. В некоторых редакторах в интерактивном режиме переход на следующую строку без выполнения кода (продолжение редактирования кода) осуществляется по нажатию комбинации клавиш *Ctrl+Enter*.

```
>>> x = input()
... x = int(x)
>? 1
```

Даже в такой записи имеет место всё та же опасность, что и ранее. Но мы можем использовать специальный метод строк, а именно *isdigit()*, с которым познакомились в разделе “1.21 Другие функции для работы со строками”. Метод *isdigit()* возвращает результат типа *bool*, а именно: *True*, если строка состоит только из чисел, и *False* - иначе. Используя метод *isdigit()* можно проверить строку, введя сначала число 12, а при следующем запуске - символ *s*:

```
>>> x = input()
... if x.isdigit():
```

```
...     x = int(x)
...     # to do smth with int x
... else:
...     print('Incorrect x')
...
>? 12
>>> x = input()
... if x.isdigit():
...     x = int(x)
...     # to do smth with int x
... else:
...     print('Incorrect x')
...
>? s
Incorrect x
```

Стоит отметить, что это не единственный способ проверять корректность ввода. Более того, в случае проверки вещественного числа функция *isdigit()* вернет *False* из-за введенной пользователем точки “.” при считывании с клавиатуры:

```
>>> x = input()
... if x.isdigit():
...     x = int(x)
...     # to do smth with int x
... else:
...     print('Incorrect x')
...
>? 13.123
Incorrect x
```

Также можно оставить считывание и преобразование типа в одной строке, но тогда нужно будет иметь дело с обработкой возникающих по факту выполнения программы ошибок. Для корректной обработки ошибок в таком случае следует изучить раздел “3.2.5 Исключения”.

В примере кода выше вместо строки с комментарием добавим условие попадания в интервал. Окончательно код примет такой вид:

```
>>> x = input()
... if x.isdigit():
...     x = int(x)
...     if x > 1 and x <= 5 or x >= 10:
...         print('True')
...     else:
...         print('False')
... else:
...     print('Incorrect x')
```

```
...
```

Попробуйте запустить данный код и получить результат его работы. Подумайте, как можно упростить большое условие, представленное в примере выше.

1.49 Упражнения и вопросы первой главы для самоконтроля

1.49.1 Базовые типы данных. Ввод, вывод.

1. Что будет выведено программой? Сколько символов будет выведено?
```
>>> a = 50
>>> a
```
2. Как выглядит код для чтения пользовательской строки из терминала в переменную *b*?
3. Что нужно знать при считывании числовых значений с помощью *input()*?
4. Какая ветка условия сработает в следующем коде
```
>>> a = 100
... if a > 200 :
...   print("Ветка 1")
... elif a > 90:
...   a = a + 21
...   print("Ветка 2")
... elif a > 120:
...   print("Ветка 3")
... else:
...   print("Ветка 4")
...
```
5. Упростите следующую запись и проверьте работоспособность полученного кода:
```
>>> a = -10
... if a > 10:
...     print("1")
... else:
...     if a > 5:
...         print("2")
...     else:
...         if a > 3:
...             print("3")
...         elif a >= -3:
...             print("4")
...         elif a >= -5:
...             print("5")
...         else:
```

```
...             if -5 < a >= -10:
...                 print("6")
...             else:
...                 print(None)
...
```

6. Зачем нужен блок *else* в циклах?
7. Как сделать бесконечный цикл *while*?
8. Как задать аргументы *range* так, чтобы получился список всех целых чисел между 50 и 60?
9. Как задать аргументы *range* так, чтобы получился список всех чисел, кратных четырем, между 0 и 100?

1.49.2 Числа, строки и операции

1. Допустим, дан такой пример кода:

```
>>> x, y, z = 100, 'abs', -12+4j
```

Что хранится в каждой переменной и какой тип у нее тип?

2. Какие математические операции недоступны для комплексных чисел?
3. Какие операции сравнения недоступны для комплексных чисел?
4. Какие математические операции доступны для строк? Приведите примеры, объясните результаты.
5. Что будет получено в результате выполнения следующего фрагмента кода:

```
>>> bool("0")
>>> bool("1")
```

6. Запустите примеры кода, представленные ниже и ответьте на вопрос, как работают логические операции для переменных типа *bool*?

```
>>> f and t
>>> t or f
>>> not t
>>> not f or t
```

7. Проверьте, какой тип данных возвращают следующие строки кода:

```
>>> f + t
1
>>> f / t
0.0
```

8. Посмотрите, что будет возвращать метод строк *islower()*, если в строке есть хоть один символ в верхнем регистре. Аналогично, посмотрите, что будет возвращать метод строк *isupper()*, если в строке есть хоть один символ в нижнем регистре.
9. Проверьте, какой тип данных возвращают функция *bin(int)*, *hex(int)* и

oct(int).

10. Среди встроенных функций, помимо рассмотренной *sum(obj)*, есть еще функции *min(obj)* и *max(obj)*. Познакомьтесь с ними самостоятельно и ответьте на вопрос: какое ограничение их использование накладывает на *obj*?
11. Запустите следующий фрагмент кода:

```
>>> hex_v = 0xA12G
```

Объясните полученные результаты.

1.49.3 Словари и списки

1. Создайте единичную матрицу 3 на 3 с помощью генератора списков.
2. Назовите известные вам способы изменения объекта списка.
3. Назовите известные вам способы изменения объекта словаря.
4. Чем отличается функция copy.copy() от copy.deepcopy()? Приведите пример, когда copy.copy() недостаточно.
5. Как перебрать в цикле все ключи словаря? Все значения? Как одновременно в цикле использовать и ключи, и значения?
6. Что будет выведено программой?

```
>>> st = '   \t\n 1 2 3\n'
>>> st.split()
>>> st.split(' ')
```

7. Скажите, что будет храниться в переменной s после выполнения следующего фрагмента кода:

```
>>> s = 'q'
>>> t = '123'
>>> s.join(t)
```

1.49.4 Функции и модули

1. Что такое динамическая и статическая типизация?
2. Что такое переменная, ссылка и объект? Проиллюстрируйте рисунком на примере объекта строки.
3. Нарисуйте, что происходит в памяти, когда выполняется фрагмент кода ниже:

```
>>> st1 = 'hello'
>>> st2 = st1
>>> st1.replace('h', 'H')
```

4. Нарисуйте, что происходит в памяти, когда выполняется фрагмент кода ниже:

```
>>> st1 = 'hello'
>>> st2 = st1
```

```
>>> st1 = st1.replace('h', 'H')
```

5. Нарисуйте, что происходит в памяти, когда выполняется фрагмент кода ниже:

```
>>> d1 = {'k':1, 'p':2}
>>> d2 = d1
>>> d2.update({'k': 3})
```

6. Чем отличается прототип функции от вызова? Приведите примеры.
7. Создайте функцию, которая получает два аргумента и возвращает их среднее арифметическое.
8. В функции из пункта выше сделайте второй аргумент аргументом по умолчанию со значением, равным 10.
9. Вызовите функцию из пункта 7, используя именованные аргументы.
10. Что будет выведено в результате работы программы?

```
>>> def func(arg):
...         arg.update({‘1’:’1’})
...
>>> x = {‘2’: ’2’, ‘4’: ‘4’}
>>> func(x)
>>> x
```

11. Что будет выведено в результате работы программы?

```
>>> def func(arg):
...         arg += 10
...
>>> x = 100.6
>>> func(x)
>>> x
```

12. Что будет выведено в результате работы программы?

```
>>> def func(x):
...    x = x.split()
...
>>> x = ‘hello, world’
>>> func(x)
>>> x
```

13. Что будет выведено в результате работы программы?

```
>>> def func():
...    global x
...    x = x.split()
...
>>> x = ‘hello, world’
>>> func(x)
>>> x
```

14. Чем отличаются операции is и ==?

15. Как импортировать модуль с названием sys?
16. Как импортировать переменную path из модуля sys?
17. Чему равно значение __name__ для модуля, который импортируется? Чему равно его значение, когда модуль запускается?
18. Как с помощью функции print() вывести три переменные со значениями ‘hello’, [1, 2, 3], 4.5, которые будут разделены точкой с запятой?

Выводы по главе

По результатам изучения первой главы настоящего учебного пособия мы достигли следующих результатов:

- познакомились с основными типами данных языка Python;
- научились использовать различные операции;
- освоили основные управляющие конструкции языка Python: ветвление, циклы;
- изучили функции ввода и вывода, и как в целом устроены функции и модули в языке Python;
- написали свой собственный модуль;
- поняли особенности работы с областями видимости и разделяемыми ссылками.

ГЛАВА 2. МОДЕЛИРОВАНИЕ РАБОТЫ КОМПЬЮТЕРА

Цели и задачи главы

Данная глава посвящена описанию основ архитектуры вычислительных устройств, исследованию способов и форматов представления данных, а также знакомству с машиной Тьюринга и решению ряда практических задач. Цель – сформировать понимание принципов, как устроено и как работает вычислительное устройство.

Задачи:

- кратко рассмотреть историю развития вычислительных устройств,
- исследовать, как внутри устроено вычислительное устройство,
- рассмотреть основы работы с представлениями чисел в различных системах счисления,
- изучить поразрядные операции,
- освоить форматы представления данных различных видов в памяти и алгоритмы получения этих представлений,
- реализовать машину Тьюринга на Python для моделирования работы вычислительного устройства.

2.1 Введение в архитектуру ЭВМ

2.1.1 Позиционные системы счисления: 2, 8, 16, 10

Каждый день каждый человек сталкивается с десятичной системой счисления. В этом нет ничего удивительного: у человека на руках десять пальцев, что привело к появлению десятичной системы как основной для подсчетов.

В компьютере десятичная система не используется, хотя так было не всегда. Самые первые вычислительные устройства и компьютеры использовали десятичную систему счисления, поскольку идея использования другой системы счисления не была столь очевидна. Однако поскольку в компьютере используются электрические сигналы, самым простым способом будет использовать в нем двоичную систему счисления: каждый разряд числа в двоичной системе в таком случае является либо нулем, либо единицей. За единицу (1) в электронике принимают наличие напряжения, а за ноль (0) – его отсутствие. Позже мы рассмотрим работу с таким представлением информации более детально.

В компьютере один двоичный разряд (то есть 1 или 0) называют *битом*.

А так как представить одно из двух состояний можно наличием или отсутствием электрического тока, то, например, простая лампочка может отображать один бит информации: горящая лампочка соответствует единице, негорящая – нулю. Бит – это минимальная часть информации, которая может быть закодирована и использована. Группируя биты вместе, можно получить другие единицы хранения данных, которые позволят хранить больше, чем просто 0 или 1. Например, следующая после бита единица информации – *байт*. Так было не всегда, но на данный момент размер байта практически всегда равен строго 8 битам. Таким образом, с помощью различных комбинаций нулей и единиц, с помощью одного байта можно закодировать числа от 0 до 255. То есть 2^8 различных значений. Байты также объединяются в более крупные единицы. Например, килобайт (Кбайт), который равен 2^{10} байт, мегабайт (2^{20}), Гигабайт (2^{30}) и так далее. Обратите внимание, что, например, приставка кило в случае с байтами обозначает не 1000 байт, а 1024.

Давайте рассмотрим способ перевода числа из десятичной системы счисления в двоичную и обратно. На самом деле, описанный далее алгоритм применим для перевода из десятичной системы счисления в абсолютно любую другую.

Для того, чтобы перевести целое число из десятичной системы системы счисления в двоичную, требуется делить число нацело на два (то есть на основание системы счисления), запоминая остатки до тех пор пока число не станет меньше двух (основания системы счисления в которую мы переводим число). Результатом перевода является комбинация из запомненных остатков в обратном порядке.

Давайте рассмотрим описанный алгоритм на примере. Пусть требуется перевести в двоичную систему счисления число 199.

199/2 = 99 (остаток 1)
99/2 = 49 (остаток 1)
49/2 = 24 (остаток 1)
24/2 = 12 (остаток 0)
12/2 = 6 (остаток 0)
6/2 = 3 (остаток 0)
3/2 = 1 (остаток 1)
1/2 = 0 (остаток 1)

Теперь запишем остатки в обратном по ходу деления порядке и получим число 11000111_2, что равно 199_{10}. Далее мы разберем как перевести число из двоичной системы счисления в десятичную и проверим наш результат.

Позиционная система счисления это такая форма записи числа, в которой значение каждой цифры зависит от ее места в записи. Нумерация позиций начинается с нуля и возрастает справа налево. Любое число в позиционной системе счисления можно представить как сумму цифр этого числа, умноженных на основание системы счисления в степени, которая равна позиции цифры в числе. Например, в десятичной системе:

$$56789_{10} = 5 * 10^4 + 6 * 10^3 + 7 * 10^2 + 8 * 10^1 + 9 * 10^0$$

При представлении числа в двоичной системе счисления основание системы счисления должно быть 2, а не 10. Например, число 11000111_2 можно записать как:

$$1*2^7+1*2^6+0*2^5+0*2^4+0*2^3+1*2^2+1*2^1+1*2^0 \text{, т.е. } 199_{10}$$

Также, может потребоваться определить минимальное количество битов требуется для представления того или иного числа, записанного в десятичной системе счисления. Для этого достаточно перевести его в двоичную систему счисления. В языке Python это можно сделать, использовав метод *bit_length()*.

Например:

```
>>> n = 199
>>> n.bit_length()
8
```

У двоичной системы счисления есть один серьезный недостаток: записанные числа обычно получаются очень длинными. Поэтому бóльшую популярность имеют восьмеричная и шестнадцатеричная системы счисления.

Представление числа в восьмеричной системе счисления:

$$567_8 = 5 * 8^2 + 6 * 8^1 + 7 * 8^0$$

Шестнадцатеричная система счисления (для ее использования потребуется расширенный набор: цифры от 0 до 9 и первые шесть латинских букв: A, B, C, D, E, F – всего 16 знаков):

$$56A8C_{16} = 5 * 16^4 + 6 * 16^3 + A * 16^2 + 8 * 16^1 + C * 16^0$$

Используемые в шестнадцатеричной системе буквы обозначают соответственно: $A_{16}=10_{10}$, $B_{16}=11_{10}$, $C_{16}=12_{10}$, $D_{16}=13_{10}$, $E_{16}=14_{10}$ $F_{16}=15_{10}$.

Так как операции перевода между двоичной, восьмеричной и шестнадцатеричной системами счисления выполняются достаточно часто, переводить из, например, двоичной в десятичную, а потом в восьмеричную

неудобно и долго.

1) Прямые переходы 2->8, 2->16

Для того, чтобы перевести число записанное в двоичной системе счисления в восьмеричную, достаточно разбить его на группы по три разряда (т.к. для записи чисел в восьмеричной системе счисления требуется три разряда: $8=2^3$), начиная справа и перевести в восьмеричную систему счисления каждую группу.

Например:

$$70_{10} = 1000110_2 = 1|000|110 = 106_8$$

Аналогичным образом можно легко перевести число из двоичной в шестнадцатеричную и обратно. Только на этот раз размер группы будет по 4 разряда, т.к. для записи чисел в шестнадцатеричной системе счисления требуется четыре разряда: $8=2^4$. Давайте попробуем перевести шестнадцатеричное число в двоичную систему счисления:

$$76E_{16} = 0111|0110|1110 = 11101101110_2$$

Отметим, что преобразование из восьмеричной в шестнадцатеричную систему счисления через двоичную также получается быстрее, чем через десятичную.

2) Работа с системами счисления в Python

В предыдущей главе рассматривалась функция int() (в разделе “1.6.1 Подробнее про int”). Было показано, что с помощью данной функции можно не только выполнять преобразование вещественных чисел float в целые, но и получать целые из разных систем счисления, которые записаны в строке (“1.6.1.2) Переход от str и int”).

Ранее в разделе “1.7.2 Восьмеричные oct(), шестнадцатеричные hex() и двоичные bin() числа” были продемонстрированы функции *bin(int)*, *oct(int)* и *hex(int)* для получения двоичных, восьмеричных и шестнадцатеричных чисел соответственно. Там же было показано, что можно выполнять обратный переход, т.е. переход от представления числа, где используются символы *o*, *x*, *b*, к числу в десятичной системе счисления:

int(bin), int(oct), int(hex)

Таким образом, в зависимости от префикса 0x, 0o или 0b будет определена система счисления как соответственно, шестнадцатеричная, восьмеричная или двоичная и будет выполнено преобразование

```
>>> int(0x10)
16
>>> int(0o10)
8
>>> int(0b10)
2
```

2.1.2 История развития компьютера

Рассмотрим историю развития компьютера, чтобы лучше понимать, как появились современные компьютеры в том виде, в котором они есть сейчас.

Начало современным компьютерам положило стремление человека автоматизировать вычисления, которые приходилось делать вручную на бумаге. Одними из первых инструментов для этого была арифмометры - механические устройства, позволяющие выполнять простейшие арифметические действия: сложение и вычитание, а в дальнейшем - и умножение с делением. Но при больших объемах вычислений это все равно давало недостаточный уровень автоматизации, заставляя человека вводить каждую пару чисел и записывать результат.

Чарльз Беббидж (26.12.1791–18.10.1871) в 1820–1833 гг. пытался создать механическую машину, которой он дал имя – разностная машина. Такая машина должна была осуществлять вычисления для создания математических таблиц и их печать. Беббидж построил модель машины, которая состояла из валиков и зубчатых колес, вращаемых вручную при помощи специального рычага. Не завершив создание разностной машины, Беббидж приступил к созданию аналитической машины – более общей и сложной в своей конструкции.

Модель аналитической машины имела “хранилище”, которое являлось памятью, и “мельницу” – вычислительное арифметическое устройство. По задумке Беббиджа в машину можно было загрузить программу на перфокарте, чтобы обозначить, какую именно операцию необходимо осуществить. Кроме того можно было получить результат работы машины в виде напечатанной таблицы. Аналитическая машина должна была уметь совершать сложение, вычитание, умножение и деление. При жизни Беббиджа она так и не была построена. Ада Лавлейс (10.12.1815–27.11.1852) создала первую в мире *программу* для аналитической машины Беббиджа. Именно поэтому Ада Лавлейс считается самым первым программистом.

Машины, подобные разностной и аналитической, назывались *механическими*.

Чтобы перейти к следующему виду компьютеров, необходимо уделить внимание такому важному изобретению, как *реле*.

Реле – это коммутационное устройство, позволяющее механически соединять или разъединять цепь электронной схемы посредством управляющего сигнала.

Конструктивно оно состоит из катушки с сердечником. При прохождении тока через катушку создается магнитное поле, которое притягивает якорь реле. Он, в свою очередь, способен механически замыкать цепь на один или другой контакт (см. рис. 2.1).

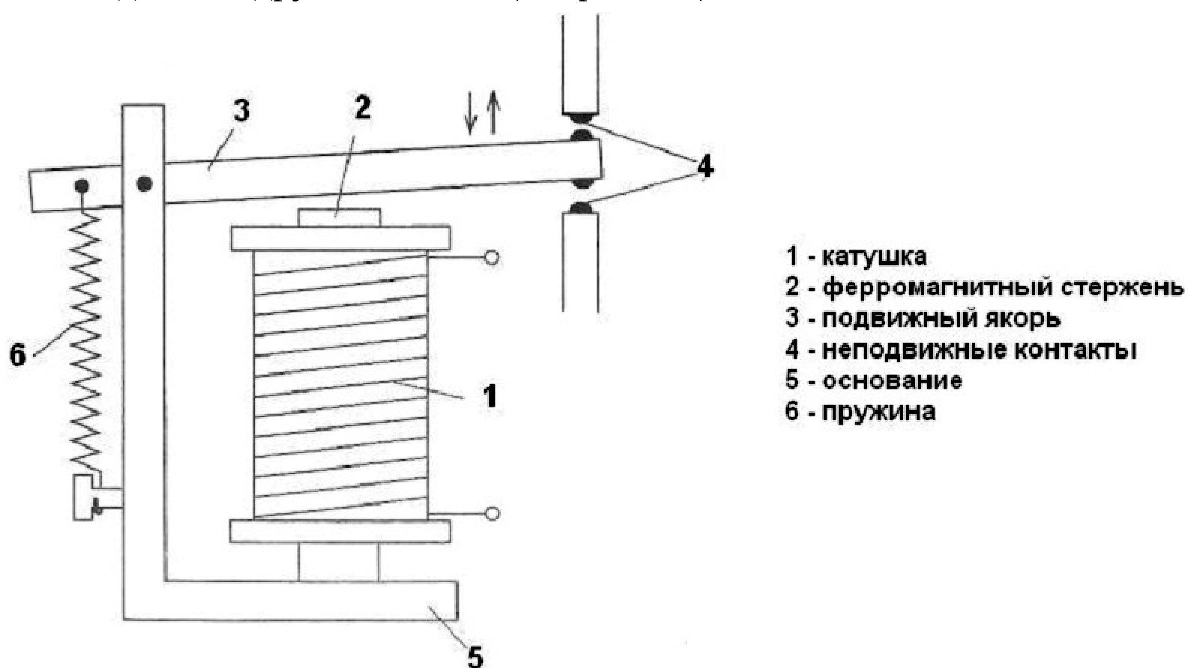


Рисунок 2.1 – Устройство реле

Замкнутое реле может служить аналогом единицы, разомкнутое – аналогом нуля.

Реле можно использовать как переключатель, а это значит, что с его помощью можно выполнить логические операции, создать сумматор (логический узел, выполняющий арифметическое сложение двоичных разрядов) и более сложные вычислительные схемы.

Релейные компьютеры начали появляться в 30-х годах XX-го века. В отличие от машин, подобных машине Беббиджа, они назывались *электромеханическими*. Такие компьютеры были весьма громоздкими и мало походили на современные. Например, известные релейные компьютеры Mark

I и Mark II использовали десятичную систему счисления и перфоленту как устройство ввода/вывода.

Поскольку реле сделано из металлической пластины, любое попадание грязи между контактов может привести к его неисправности. Кстати, вам наверняка известен термин *баг*: ошибка в компьютерной программе. Этот термин появился после того, как из реле компьютера Mark II извлекли жучка (от англ. “bug” – жук).

Реле были значительно удобнее зубчатых колес, поскольку их приводило в действие электричество, а не специальные рычаги. И все же, релейные компьютеры обладали рядом недостатков и достаточно быстро устарели.

В дальнейшем реле были заменены вакуумными лампами – устройствами, работающими за счёт управления интенсивностью потока электронов, движущихся в вакууме с помощью тока малой величины. Из них также, как и из реле, можно было собрать нужные логические вентили. Такие компьютеры стали называться *электронными*, в них перестали присутствовать механические части.

Вакуумные лампы работали значительно быстрее реле, примерно в 1000 раз. Однако они потребляли большое количество электроэнергии, часто выходили из строя и имели высокую стоимость.

Первый электронный цифровой вычислитель появился в 1945 году и назывался он ENIAC (ЭНИАК). Он был самым большим компьютером в истории и был построен из 18000 вакуумных ламп. Вычисления в нем проводились в десятичной системе счисления.

В создании следующего компьютера EDVAC принимал участие знаменитый математик Джон фон Нейман (28.12.1903–8.02.1967). Он предложил использовать двоичную систему счисления, а также изменить архитектуру компьютера таким образом, чтобы программа и данные хранились в одной памяти в виде набора бит и внешне никак не отличались друг от друга. Этот и другие принципы вошли в историю как принципы архитектуры Фон Неймана. Именно эти принципы определили направление развития компьютеров на десятилетия:

1. Адресность.
2. Однородность памяти.
3. Программное управление.

Давайте подробнее разберем каждый принцип.

Адресность: все данные хранятся в ячейках памяти. Каждая ячейка имеет свой номер или адрес. Процессор может получить доступ к ячейке, т.е. прочитать данные или записать данные, используя ее адрес.

Однородность памяти: память для данных и команд общая, т.е. в любой ячейке памяти может храниться как инструкция, так и данные.

Программное управление: управление вычислительным процессом происходит на основании предварительно загруженной в память программы. Это набор последовательных инструкций и данных, с которыми работают эти инструкции.

Следующее изобретение в 1956 году принесло своим создателям Уильяму Шокли (13.02.1910–12.08.1989), Джону Бардину (23.05.1908–30.01.1991) и Уолтеру Браттейну (10.02.1902–13.10.1987) Нобелевскую премию и положило начало твердотельной электроники. Мы говорим о транзисторах, полупроводниковых элементах, создаваемых на основе кремния. Основное назначение транзистора – управление сильным сигналом с помощью гораздо более слабого. Благодаря этому свойству, его можно использовать как усилитель сигнала или как управляемую “заслонку”. Это можно сравнить с водопроводной заслонкой, которая способна открываться от небольшого напора воды по отдельной трубке (её можно называть управляющей трубкой).

Наиболее часто встречается так называемый NPN (Negative-Positive-Negative) транзистор. Он, как и любой транзистор, имеет три контакта (см. рис. 2.2):

- Коллектор (C) – на него подается более мощный сигнал.
- База (B) – на него подается слабый управляющий сигнал.
- Эмиттер (E) – через него проходит ток с коллектора и базы, когда на базу подан сигнал и транзистор становится “открытым”.

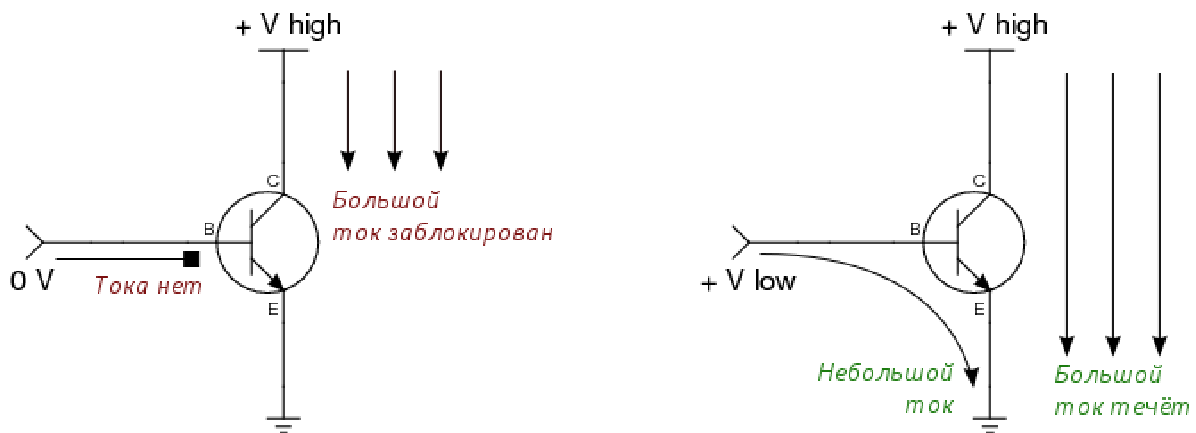


Рисунок 2.2 – Строение и принцип работы транзистора

Пришедшие на смену вакуумным лампам транзисторы имеют большое количество преимуществ: малые габариты, потребление меньшего количества энергии, редкие выходы из строя.

Из пар транзисторов собираются логические вентили – базовые элементы схем, выполняющие элементарную логическую операцию. Более подробно мы рассмотрим их в разделе “2.1.4.2) Логические вентили” Из вентилей, в свою очередь, собираются триггеры, сумматоры и другие устройства.

Одно из наиболее значимых устройств – триггер. Триггер – электронное устройство, которое может длительно сохранять свое состояние. Он обладает способностью большое количество времени находиться в одном из двух устойчивых состояний, используя которые можно собрать простейшую модель памяти, позволяющую хранить большое количество бит.

На смену компьютерам *первого поколения* (1944 – 1954 гг), где использовались электронные лампы, пришли компьютеры, изготовленные на основе транзисторов, которые относятся ко *второму поколению* (1955 – 1965 гг).

Во времена второго поколения компьютеров была создана *шина* – набор параллельно соединенных проводов, которая нужна для соединения отдельных компонентов компьютера. Более подробно о шине мы поговорим в разделе “2.1.5.2) Шины (параллельные и последовательные)”.

Следующее, *третье поколение* (1965 – 1980 гг), началось с появления интегральных схем. Идея заключается в помещении в единый неразборный корпус электронной схемы различной сложности. Таким образом, каждый чип на плате стал включать в себя большое количество компонент, представляющих собой определенную электронную схему.

Компьютеры, которые были построены на основе интегральных схем, были более быстродействующими, компактными и недорогими по сравнению с компьютерами, построенными на транзисторах.

До 1964 года новые выпускаемые компьютеры были несовместимы между собой. В 1964 году компания IBM выпустила линейку компьютеров System/360 – семейство компьютеров от менее производительных к более, использующих практически одинаковый набор команд на языке ассемблер. Компьютеры этой линейки были совместимы между собой, т.е. программы, написанные на одном компьютере за некоторыми исключениями могли запускаться на другом.

Следующее поколение – *четвертое* – началось с появлением сверхбольших интегральных схем и продолжается до сих пор. Сейчас на интегральной схеме содержатся миллиарды транзисторов, что позволило увеличить быстродействие компьютеров, увеличить размер памяти а также уменьшить размер компьютера.

2.1.3 Булева алгебра, основные операции

Булева алгебра получила свое название в честь математика Джорджа Буля (2.11.1815-8.12.1864), работавшего над математической формулировкой законов логики.

Изобретенная Дж. Булем алгебра подобна обычной алгебре, однако, в отличие от нее, простейшая булева алгебра содержит только 0 и 1. Так как булева алгебра часто используется в логике, 0 обычно называют ложью, а 1 – истиной.

Базовыми операциями булевой алгебры являются ∧ конъюнкция (логическое И), ∨ дизъюнкция (логическое ИЛИ) и ¬ отрицание.

Для демонстрации результата операции или даже выражения используют таблицы истинности, в которых рассматриваются все возможные комбинации значений операндов выражения (см. рис. 2.3).

a	b	a∧b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a∨b
0	0	0
0	1	1
1	0	1
1	1	1

a	¬a
0	1
1	0

Рисунок 2.3 – Таблицы истинности для операций конъюнкции, дизъюнкции и отрицания

2.1.4 Цифровой логический уровень

Ранее мы рассмотрели, как физически устроены электронные схемы (физический уровень). Далее следует рассмотреть уровень цифровых устройств, в котором пойдет речь о том как и для чего могут быть использованы схемы физического уровня.

1) Логические 0 и 1

Компоненты электронных схем во многом состоящие из транзисторов, называют микросхемами транзисторно-транзисторной логики (ТТЛ). Компоненты цифровых электронных систем оперирует дискретными сигналами, которые имеют только два выделенных уровня напряжения: высокий (единица) и низкий (ноль) (см. рис. 2.4).

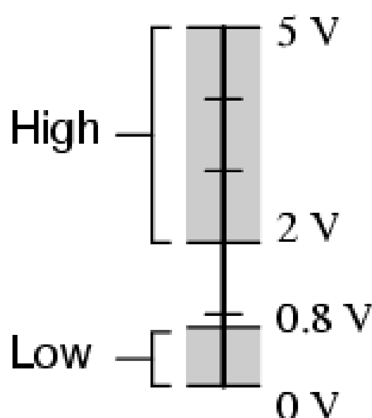


Рисунок 2.4 – Логические 0 и 1

Напряжение около 0,2 В на выходе вентиля ТТЛ соответствует нулю, напряжение 3,4 В – единице. Поскольку эти напряжения могут несколько варьироваться, иногда говорят не о нуле и единице, а о низком (*low*) и высоком (*high*) уровнях выходного сигнала.

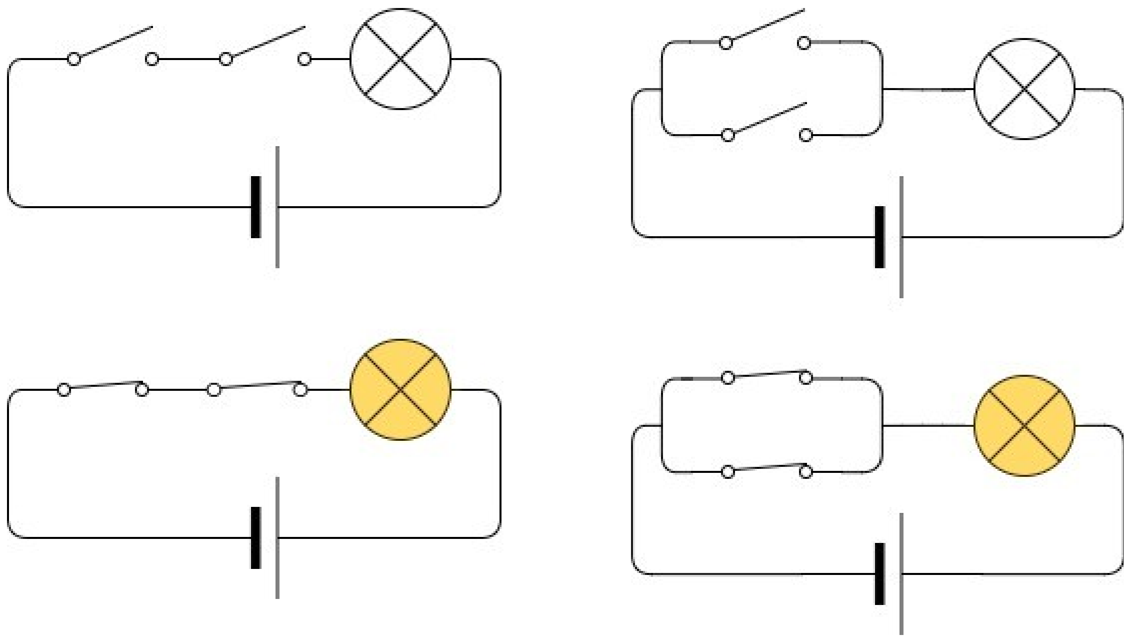
Напряжения между 0,8 и 2 В в схеме быть не должно, так как такой сигнал не может быть корректно расценен ни как единица, ни как ноль.

2) Логические вентили

В разделе “2.1.1 Позиционные системы счисления: 2, 8, 16, 10” мы уже говорили о том, что ноль и единица могут быть представлены в компьютере как отсутствие и наличие напряжения соответственно. Таким же образом может быть представлен и логический сигнал. Так как отсутствие и наличие напряжения в цепи можно контролировать с помощью простого

переключателя, давайте на их примере покажем практическое применение булевой алгебры.

Разомкнутый переключатель препятствует протеканию тока в цепи, что равносильно логическому нулю. Замкнутый – наоборот и соответствует логической единице. Последовательное и параллельное соединение переключателей реализуют схемы, согласно таблицам истинности для AND и OR (см. рис. 2.5).



Логическое И Логическое ИЛИ

Рисунок 2.5 – Соответствие последовательной и параллельной схем логическим операциям И и ИЛИ

Но все эти примеры имеют одну особенность: переключатели в них механические и управляющим воздействием не может быть логический сигнал. Поэтому для построения логических схем используют логические вентили. Это электронные элементы, выполняющие простейшие логические операции, принимающие на вход множество логических сигналов и выдающие на выход логический сигнал – результат этой простейшей операции.

Поведение каждого вентиля в зависимости от поданных на него данных можно также представить в виде таблицы истинности той операции, которую он реализует. Например, таблица истинности для логической операции И отражает поведение вентиля И, реализующего эту же операцию.

Ниже на рис. 2.6 приведены графические обозначения наиболее часто используемых вентилей. Обратите внимание на обозначение инвертора и отличие вентиля ИЛИ от вентиля ИЛИ-НЕ.

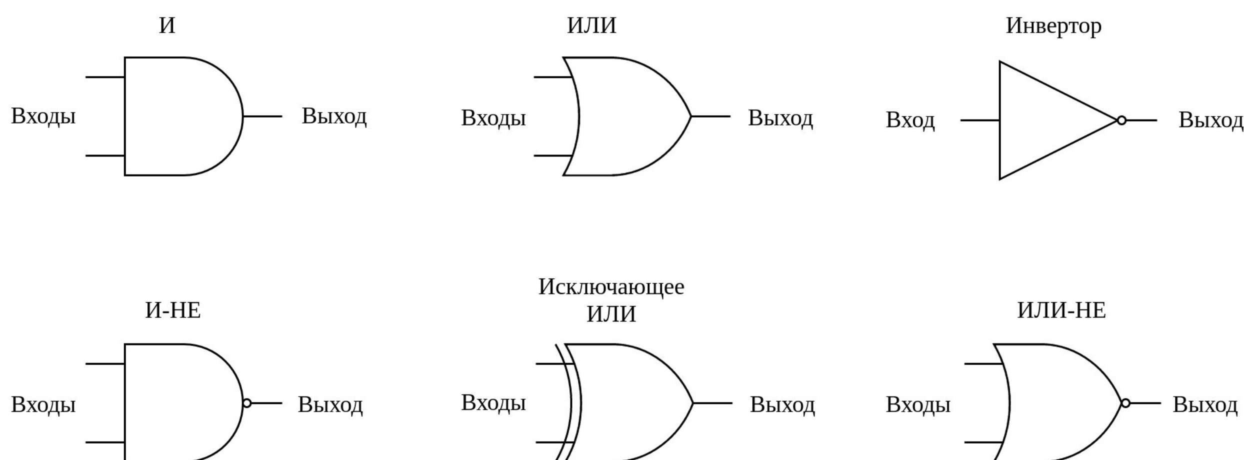


Рисунок 2.6 – Графическое изображение вентилей

Из логических операций которые не встречались ранее мы можем заметить только исключающее ИЛИ. Его результат можно описать как “одно из двух, но не оба”. То есть на выходе такого логического вентиля будет единица, когда на одном из входе будет истина, а на другом - ложь (см. табл. 2.1).

Таблица 2.1 – Таблица истинности для операции исключающее ИЛИ

a	**b**	**исключающее ИЛИ**
0	0	0
0	1	1
1	0	1
1	1	0

С помощью вентилей можно собрать сумматор – устройство, которое может складывать два двоичных числа. Составим сумматор для двух восьмиразрядных чисел.

Для начала рассмотрим, как выполняется сложение двух одноразрядных двоичных чисел. Так как для сложения порядок аргументов не важен, будем рассматривать только три случая. В качестве исходных данных у нас два числа a и b, а в качестве результата у нас сумма (S) и бит переноса (C) (см. табл. 2.2). Бит переноса появляется тогда, имеет место “единица в уме”, а именно: 1 + 1 => **1**0, где старший разряд результата и есть бит переноса.

Таблица 2.2 – Сложение двух одноразрядных двоичных чисел

a	**b**	**S**	**C**
0	0	0	0
0	1	1	0
1	1	0	1

Если обратить внимание на значения двух последних столбцов, то становится заметно, что в столбце S такой результат может предоставить операция исключающее ИЛИ, а в столбце C – операция И. Схема, собранная из таких вентилей представлена на рис. 2.7 .

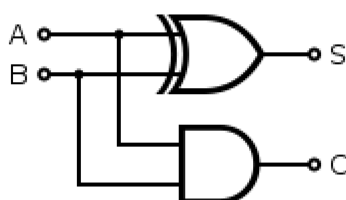


Рисунок 2.7 – Неполный сумматор (полусумматор)

Данная схема называется неполным сумматором или полусумматором. Что же в ней неполного? Дело в том, что такая схема не позволяет складывать два произвольных разряда двоичного числа из-за того, что она не учитывает бит переноса после сложения предыдущего разряда. Давайте усовершенствуем наш полусумматор. Для этого добавим в таблицу входной бит переноса *Cin*, а выходной по аналогии переименуем в *Cout* (см. табл. 2.3.)

Таблица 2.3 – Добавление бита переноса

a	**b**	**Cin**	**S**	**Cout**
0	0	0	0	0
0	0	**1**	**1**	0
0	1	0	1	0
0	1	**1**	0	**1**
1	1	0	0	1
1	1	**1**	**1**	1

Можно заметить, что с добавлением бита переноса Cin, значение S полусумматора требуется еще раз сложить с битом Cin, чтобы получить

полную сумму всех трех битов, используя уже известный нам полусумматор. Выходной бит переноса Cout может быть единицей теперь еще и в случае, когда единице равны один из входных бит и бит Cin. Давайте на рис. 2.8 графически изобразим наши наблюдения, усовершенствовав схему полусумматора.

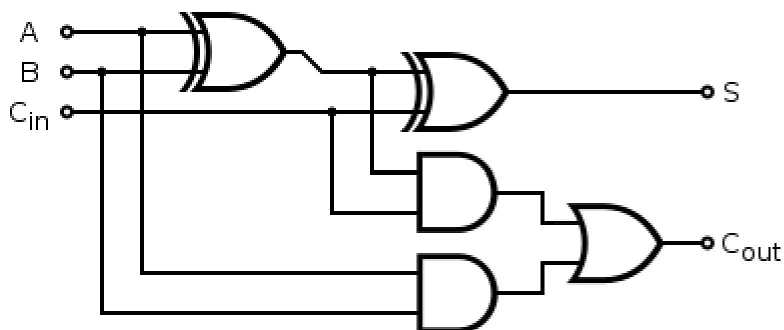


Рисунок 2.8 – Полный сумматор

Данная схема называется – *полный сумматор*. Она позволяет складывать 2 разряда двоичного числа, учитывая перенос от сложения предыдущего разряда и сообщая о бите переноса текущего. Фактически, она состоит из двух полусумматоров и вентиля ИЛИ.

Теперь, умея складывать 2 произвольных разряда двоичного числа, мы можем собрать в цепочку столько полных сумматоров, сколько разрядов нам требуется сложить. Пример для сумматора четырехразрядных чисел представлен на рис. 2.9.

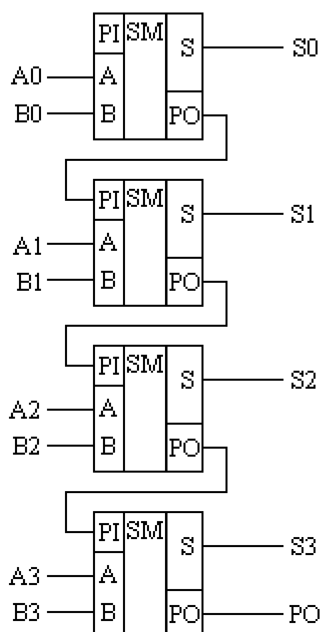


Рисунок 2.9 – Сумматор для четырехразрядных чисел

Обозначения на рис. 2.9: SM – сумматор, PO – бит переноса на выходе, (Cout), PI – бит переноса, передаваемый на вход (Cin).

2.1.5 Как устроена вычислительное устройство

Давайте рассмотрим устройство простейшего компьютера. Схематично его можно представить как показано на рис. 2.10.

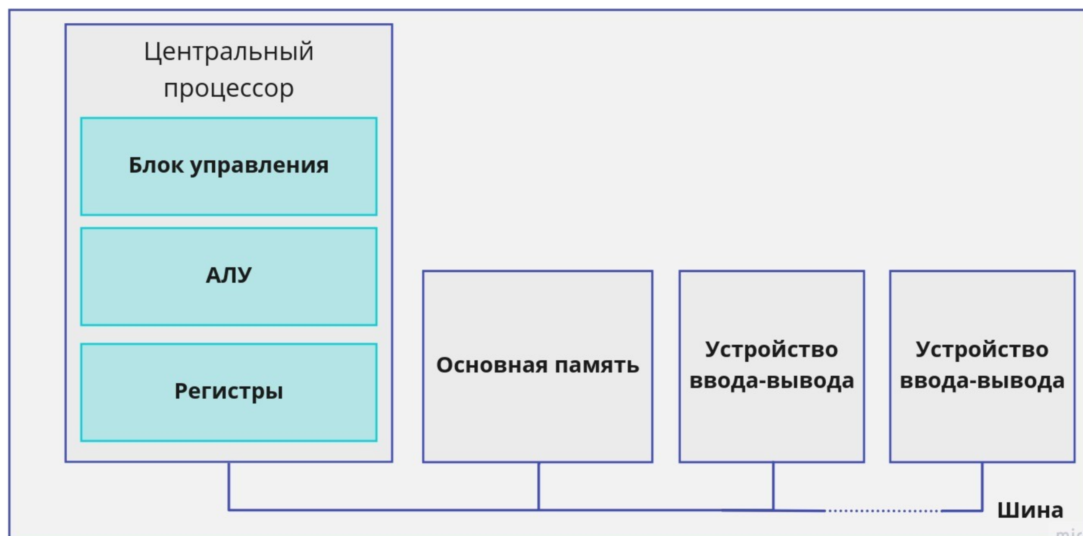


Рисунок 2.10 – Схематичное устройство вычислительной системы

Далее мы более подробно разберем назначение и устройство каждого его компонента.

1) Память

Память является основным устройством хранения данных. В ней, согласно архитектуре Фон Неймана на равных хранятся и команды для процессора, и данные, которые требуются для выполнения тех или иных инструкций. Конструктивно память можно представить в виде некоторого устройства, имеющего несколько управляющих ножек и две специальные группы ножек: адресные входы и ножки данных. На эти ножки в двоичном виде подается адрес той ячейки памяти, значение которой требуется прочитать. После этого память выставляет на ножках данных значение той ячейки, адрес которой был подан на вход. Для записи требуется также подать адрес интересующей ячейки памяти на адресные входы, но теперь на ножках данных мы сами устанавливаем значение, которое будет записано в указанную ячейку после сигнала записи на управляющую ножку.

Количество ножек для адресных входов связано с объемом самой памяти, а количество выходов - с объемом данных, которые могут быть

прочитаны за одну операцию чтения.

Конструктивно простейшую память можно реализовать используя триггеры. Если каждый триггер позволяет хранить всего один бит информации, но объединяя их вместе можно получить достаточно большой объем. Более подробно устройство памяти описывается в главе 3 книги [7].

2) Шины (параллельные и последовательные)

В компьютере ни одно устройство не взаимодействует с другим устройством напрямую, поскольку пришлось бы соединять все устройства между собой отдельно. В качестве решения используется такое устройство как *шина*. Шина представляет собой большой набор проводов, которые подключены к каждому устройству, при этом она позволяет общаться одному устройству с любым другим. Например, к шине может быть подключена память: ее адресные входы, ножки данных и управляющие ножки (см. рис. 2.11).

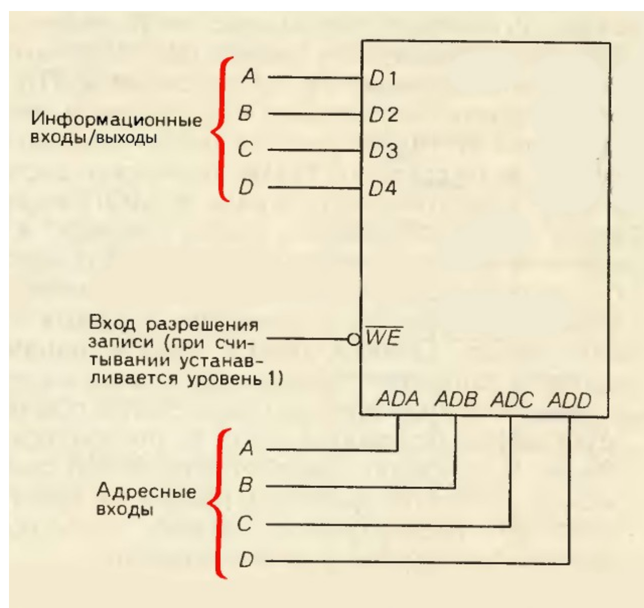


Рисунок 2.11 – Подключение разных устройств к шине

Однако, как разрешить ситуацию когда два устройства одновременно решили обратиться к третьему? Для этого используется *контроллер шины* или *арбитр шины*. Это особая микросхема, которая позволяет организовать работу с шиной таким образом, чтобы в один момент времени шина на запись принадлежала только одному устройству.

Существует алгоритм, определяющий кто имеет право сейчас работать с шиной – писать или читать. Рассмотрим его на следующем примере. Предположим, у шины есть 3 потребителя. Первый потребитель хочет что-то

записать и подает запрос: нужен доступ к шине. Контроллер знает, занята ли шина. Если нет, он помечает себе, что шина занята и выдает сообщение запрашивающему устройству, что шина свободна, а всем остальным устройствам, что шина занята.

Минусы такого подхода очевидны: при большом количестве устройств, активно использующих шину, общение всех устройств сильно замедляется. Поэтому в современных компьютерах используется несколько шин, связанных друг с другом. Одна шина может соединять более быстрые устройства, а другая - более медленные, что позволяет общаться быстрым устройствам между собой не ожидая более медленных.

Таким образом, мы изучили как организуются вычислительные системы с точки зрения хранения любой информации, включая как данные, так и инструкции, которые нужно выполнить над данными, и способ доступа к этой информации из других устройств посредством шины.

Теперь перейдем к устройству, которое занимается чтением данных и команд из памяти, а также выполнением последних – к процессору.

3) Процессор

Процессор включает в себя следующие элементы: блок управления, арифметико-логическое устройство (АЛУ) и регистры. Регистры – ячейки для хранения данных внутри процессора. Они отличаются очень высокой скоростью и очень маленьким объемом. Блок управления отвечает за чтение команд из памяти и определение их типа. АЛУ выполняет простейшие арифметические и логические операции над переданными ему аргументами.

Среди регистров самый важный – IP (Instruction Pointer) или счетчик команд. Несмотря на свое название, в нем хранится адрес команды, которая должна быть выполнена следующей. В современном процессоре есть довольно много различных регистров, но более подробно мы их рассматривать не будем.

Стоит заметить, что процессоры могут быть специализированы для выполнения каких-то определенных задач, например, работы с аудио или видео данными и могут иметь дополнительные блоки.

4) Базовый цикл работы процессора в архитектуре Фон Неймана

На самом деле, вся работа процессора сводится к последовательному выполнению команд. Сам процесс выполнения состоит из нескольких этапов:

1. Чтение команды из памяти.
2. Определение типа выполняемой команды.
3. Чтение из памяти аргументов команды.
4. Выполнение команды.
5. Сохранение в память результата.

Сначала из памяти по адресу, который хранится в регистре IP, считывается закодированная команда. Сразу после этого значение IP изменяется на адрес следующей команды. После определяется тип считанной команды. В зависимости от типа будут задействованы те или иные блоки АЛУ для выполнения операции и определено, имеет ли данная команда аргументы. Если команда их имеет, они считываются из памяти в регистры процессора. После этого выполняется текущая операция, результат её записывается обратно в память, и цикл повторяется сначала.

5) Генератор частот

Для того, чтобы все операции между устройствами были согласованы, они должны работать в общем масштабе времени, выполняя свои шаги синхронно. Например, в момент, когда процессор собирается прочитать данные из памяти, память должна эти данные выставить на ножки данных.

Для этого используют генератор частоты. Для простоты можно считать, что на один из проводов шины генератор частоты подает импульсы, которые обозначают моменты, когда устройство может выполнять какие-то действия. Некоторые элементы в момент импульса реализуют свои собственные функции. Выполнение тех или иных функций зависит от того, как поданы сигналы на входы элементов.

Именно генератор частоты является первым устройством, которое начинает работу после запуска компьютера. После того, как процессор начинает получать импульсы, он начинает читать и исполнять команды. Адрес самой первой команды жестко задан производителем процессора и не меняется в процессе его использования.

6) Периферия

К компьютеру можно подключить большое количество устройств ввода и вывода, например, монитор, клавиатуру и так далее. Давайте рассмотрим как взаимодействие с этими устройствами укладывается в нашу схему.

Пусть мы подключили устройство ввода – клавиатуру. Что произойдет, когда мы нажмем кнопку?

Мы ожидаем, что процессор, который в данный момент занят обработкой других команд, поменяет порядок их исполнения, обработает команды, связанные с реакцией на нажатие кнопки и продолжит свою работу дальше. Для того, чтобы это произошло, используется прерывание – сигнал процессору об определенном внешнем событии.

У процессора есть отдельная ножка I (*interrupt*), и если на эту ножку подать напряжение и сообщить тип прерывания, аппаратно включится другая схема, которая возьмет инструкцию с другого адреса согласно типу прерывания. После аппаратного прерывания в адрес следующей инструкции загружается адрес обработчика прерываний.

Обработчик прерывания – это подпрограмма, которая также находится в памяти, и также выполняется последовательно. После ее выполнения процессор должен продолжить исполнение инструкции, на которой был прерван.

Прерывания бывают *аппаратные* и *программные*. К аппаратным прерываниям относят еще и исключительные ситуации (например, *Segmentation Fault* или *Stack Corruption*). Вы могли видеть сообщения от операционной системы об их возникновении при написании различных программ. Программные прерывания возникают, когда программа, которая находится в памяти, нуждается в помощи других устройств. К таким устройствам относятся, например, устройства ввода и вывода информации.

7) Разрядность процессора и шины

На быстродействие процессора влияет максимальная тактовая частота, поскольку она определяет время, затрачиваемое на выполнение каждой команды. На быстродействии сказывается и ширина шины данных. Хотя 4-разрядный процессор и способен складывать 32-разрядные числа, делает он это гораздо медленнее 32-разрядного процессора. А вот связь между быстродействием и объемом адресуемой памяти уже не так очевидна. На первый взгляд, размер адресного пространства не имеет отношения к быстродействию и лишь накладывает ограничение на способность процессора решать определенные задачи, требующие значительной памяти.

Ширина шины адреса определяет максимальный объём физической памяти, непосредственно адресуемой процессором. Для 20-разрядной адресной шины процессора i8086 используется двадцать двоичных (или пять шестнадцатеричных) разрядов.

Компьютеру нужна память для хранения кодов команд, которые процессор будет исполнять. Компьютеру нужно устройство ввода, чтобы эти коды попадали в память, а также устройство вывода, позволяющее просматривать результаты работы программы. Напомним, что оперативная память энергозависима – при отключении питания ее содержимое стирается. Поэтому компьютер нужно снабдить долговременным запоминающим устройством, в котором можно хранить данные и программы, когда компьютер выключен.

8) Достоинства и недостатки архитектуры Фон Неймана

В архитектуре Фон Неймана программы и данные хранятся в общей единой памяти. Обращение процессора к этой памяти всегда одинаковое вне зависимости от того, что он читает, команду или данные. Это позволяет эффективно расходовать имеющийся объем памяти. Использование единой шины для передачи всех этих данных также повышает надежность системы.

Однако, плюсы одновременно являются и недостатками. При таком подходе шина становится “бутылочным горлышком”, не позволяя читать команды и данные одновременно, что ускорило бы выполнение программы. Стоит отметить, что на сегодняшний день существуют компьютеры, построенные в соответствии с Гарвардской архитектурой, где программа хранится в отдельной памяти, но такие компьютеры, как правило, предназначены для решения узкоспециализированных задач.

9) Введение в ассемблер

Команды, как и данные, хранятся в памяти одинаково – в виде набора байт. Поэтому сначала программистам приходилось писать программы путем ручного ввода кода каждой инструкции. Это сильно затрудняло не только чтение программ, но и поиск ошибок, что впоследствии привело к появлению языка ассемблера.

Ассемблер – это язык программирования, в котором уже используются человекочитаемые символьные обозначения для команд процессора, регистров. В этом языке можно присваивать символьные имена адресам памяти и использовать в коде программ не только двоичную систему счисления.

Программа, написанная на таком языке, как и на любых других языках программирования, фактически является просто текстовым файлом. Для того, чтобы получить из этого текстового файла программу в том виде, в

котором она может быть загружена, в память используется компилятор. Компилятор – это программа, которая переводит программу, написанную на языке более высокого уровня, в язык более низкого. Для программы на языке ассемблера более низким уровнем являются машинные команды.

Можно предположить, что раз ассемблер так тесно связан с машинными командами, то он очень сильно зависит от модели процессора. И это действительно так, поэтому переносить программы на ассемблере с компьютера на компьютер так просто нельзя.

Несмотря на то, что язык ассемблера стал первым шагом к созданию инструмента, ориентированного в большей степени на программиста, мыслить при написании программы приходилось все равно на уровне машинных инструкций. Для программиста удобнее абстрагироваться от тех или иных деталей работы конкретного процессора и писать программы, которые бы описывали действия так, чтобы они могли быть выполнены на различных компьютерах. Такой язык был бы больше отдален от машинных инструкций, но ближе к пониманию программиста.

Возьмем для примера язык Си. С одной стороны, он все еще позволяет программисту работать напрямую с памятью. С другой стороны, он предоставляет достаточно большой функционал, который существует изолированно от какого-либо конкретного компьютера. Ключевой особенностью тут является то, что программа на языке Си, перенесенная с одного компьютера на другой, должна быть просто заново *скомпилирована* для конкретного компьютера, а не переписана.

Таким образом, любая программа, написанная на языке высокого уровня может быть преобразована к программе более низкого уровня и это может повторяться до тех пор, пока не будет получен машинный код, готовый для выполнения на конкретном процессоре.

10) Преобразование кода программы

Напомним, что компиляция – это преобразование программы из одного языка на другой, более низкого уровня, который совсем необязательно должен быть языком машинных инструкций, как в случае с программой на языке С.

Существует еще один подход к выполнению программы – интерпретация. В этом случае специальная программа-интерпретатор

получает на вход программу, написанную на интерпретируемом языке программирования и выполняет инструкции, записанные в этой программе без предварительной обработки.

Также важно отметить, что результатом компиляции программы может быть и байт-код. Байт-код – это низкоуровневое, платформенно независимое представление исходного текста программы. Байт-код может быть передан интерпретатору и выполнен. Например, интерпретатор Python транслирует каждую исходную инструкцию в группы инструкций байт-кода, разбивая ее на отдельные составляющие. Такая трансляция в байт-код производится для повышения скорости: байт-код выполняется намного быстрее, чем исходные инструкции в текстовом файле программы.

Таким образом, в случае интерпретируемых языков программу можно переносить с одного компьютера на другой без каких-либо изменений и без последующей компиляции. Достаточно иметь на каждом компьютере интерпретатор для программ соответствующего языка программирования.

2.2 Формат представления данных в компьютере

В памяти компьютеров хранится самая разная информация, начиная от видео-файлов, изображений, музыкальных файлов, заканчивая различными текстовыми документами и другими файлами. Спускаясь от уровня пользователя до физического уровня – памяти компьютера, все хранимые данные преобразуются в числа, и уже числа в памяти компьютера приобретают бинарный вид – нули и единицы. Как хранятся числа в памяти компьютера будет рассмотрено далее.

В математике мы можем работать со сколь угодно большими числами, и нас ограничивает разве что ширина тетрадного листа, где мы записываем числа. Однако для работы с числами в компьютерах ситуация другая. Физически память компьютера ограничена, это значит, что на самом низком уровне для хранения чисел отведено фиксированное количество двоичных разрядов (ячеек памяти, куда записываются биты 0 и 1). Для хранения чисел в памяти компьютеров, ограниченной n разрядами, используют несколько приемов, таких как: прямой код, обратный код и дополнительный код.

Использование того или иного приема зависит от того, какое число перед нами: знаковое (отрицательное или неотрицательное) или беззнаковое (только неотрицательное). Начнем рассмотрение с представления беззнаковых чисел.

2.2.1 Формат представления целых беззнаковых чисел

Чтобы представить беззнаковое целое число в памяти компьютера, его необходимо записать в регистр, совершив следующие действия:

1. Перевести число в двоичную систему счисления.
2. Записать биты таким образом, чтобы самый младший бит располагался в самом правом разряде регистра в памяти, а биты располагались друг за другом непрерывно. Если в левых ячейках памяти остались "незанятые" биты, они заполняются нулями.

Диапазон значений представления беззнаковых чисел зависит от разрядности архитектуры, то есть от того, сколько разрядов в памяти компьютера. Например, для 8-разрядной архитектуры минимальное число, которое можно представить восемью битами, это: $0000\,0000_2=0_{10}$, а максимальное число: $1111\,1111_2=255_{10}=2^8-1$. Таким образом, значения варьируются от 0 до 255 (всего 256 значений). Если архитектура 16-разрядная, то минимальное число: $0000\,0000\,0000\,0000_2=0_{10}$, а максимальное число: $1111\,1111\,1111\,1111_2=65535_{10}=2^{16}-1$, то есть значения варьируются от 0 до 65535. Таким образом, диапазон представления целых беззнаковых чисел таков:

$$0\ldots2^n-1$$

где *n* – разрядность архитектуры.

Например, дано число 123_{10}. Для 8-разрядной архитектуры в двоичном виде оно представимо так, как показано далее (верхняя строка в табл. 2.4 – нумерация разрядов):

Таблица 2.4 – Двоичное представление числа

Разряды:	7	6	5	4	3	2	1	0
123_{10}	0	1	1	1	1	0	1	1

2.2.2 Побитовые (поразрядные) операции

Итак, у нас есть двоичное представление числа $123_{10}=1111011_2$. Если представить, что первый бит в таком представлении числа стал равен нулю, получим двоичное представление $0\,111011_2=111011_2$ другого десятичного числа:

59_{10}. Отметим, что любой бит в представлении числа можно как обнулить, так и установить равным единице. Операции обнуления и установления в единицу можно выполнять не только с одним битом, но и сразу с группами бит. Операции, которые изменяют значения бит называются *побитовыми*. Побитовые операции в двоичном представления числа изменяют значения нулей и единиц в соответствии с разрядами, поэтому такие операции еще называют *поразрядными* (т.к. один разряд соответствует одному биту: нулю или единице).

Побитовые операции представлены в табл. 1.3 в разделе “1.8 Операции в языке Python” (в языке C и Python они обозначаются одинаково). Отметим, что операции |, &, ^, >>, << являются бинарными, а операция ~ – унарной (см. объяснения про унарные и бинарные операции в “1.14 Логические операции”).

Остановимся подробнее на каждой битовой операции. При выполнении операций И, ИЛИ, исключающее ИЛИ для двух чисел, для каждой пары разрядов этих чисел независимо выполняется одна из этих операций.

a	b	a&b	a|b	a^b	~a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

```
11010011        11010011        11010011
   &               |               ^
10001100        10001100        10001100
------------    ------------    ------------
10000000        11011111        01011111
```

```
~11010011       11010011>>3     11010011<<3
------------    ------------    ------------
 00101100        00011010        10011000
```

Рисунок 2.12 – Таблицы истинности для битовых операций

В табл. 2.5 рассмотрим примеры их работы на языке Python, предварительно введя следующие переменные:

```
>>> x = 2134
>>> y = 8291
```

Таблица 2.5 – Примеры работы битовых операций

№	Пример кода	Результат
1	`>>> bin(x)`	`'0b100001010110'`
2	`>>> bin(y)`	`'0b10000001100011'`
3	`>>> x | y`	`10359`
4	`>>> bin(x | y)`	`'0b10100001110111'`

5	`>>> x & y`	`66`
6	`>>> bin(x & y)`	`'0b1000010'`
7	`>>> x ^ y`	`10293`
8	`>>> bin(x ^ y)`	`'0b10100000110101'`
9	`>>> x >> 4`	`133`
10	`>>> bin(x >> 4)`	`'0b10000101'`
11	`>>> y << 3`	`66328`
12	`>>> bin(y << 3)`	`'0b10000001100011000'`

Обратите внимание, что функция *bin()* возвращает в качестве результата строку *str*, в которой присутствует характерный литерал *b*.

Битовые представлением для каждой операции представлены в табл. 2.6 – 2.8, где первая строка – разряды; вторая и третья строки – значения оперируемых переменных; третья строка – результат выполнения побитовой операции; полужирным курсивом выделены незначащие биты.

Таблица 2.6 – Побитовое ИЛИ

Разряды:	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	***0***	***0***	1	0	0	0	0	1	0	1	0	1	1	0
Y	1	0	0	0	0	0	0	1	1	0	0	0	1	1
|	1	0	1	0	0	0	0	1	1	1	0	1	1	1

Таблица 2.7 – Побитовое И

Разряды:	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	***0***	***0***	1	0	0	0	0	1	0	1	0	1	1	0
Y	1	0	0	0	0	0	0	1	1	0	0	0	1	1
&	0	0	0	0	0	0	0	1	0	0	0	0	1	0

Таблица 2.8 – Побитовое исключающее ИЛИ

Разряды:	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	***0***	***0***	1	0	0	0	0	1	0	1	0	1	1	0
Y	1	0	0	0	0	0	0	1	1	0	0	0	1	1

^	1	0	1	0	0	0	0	0	1	1	0	1	0	1

Операции << и >> выполняют сдвиг всех разрядов числа на некоторое количество разрядов. Сдвинутые разряды (например, при сдвиге вправо) теряются. Места сдвинутых разрядов (при сдвиге влево) заполняются нулями. Использование побитового сдвига эквивалентно умножению на два в определенной степени: для сдвига влево степень положительная, для сдвига вправо – отрицательная. Продемонстрируем на десятичном числе (см. табл. 2.9):

Таблица 2.9 – Демонстрация сдвигов

Сдвиг влево	Сдвиг вправо

`>>> a = 2`
`>>> deg = 1`
`>>> a << deg`
`4`
`>>> a * 2 ** deg`
`4`
	`>>> a = 2`
	`>>> deg = 1`
	`>>> a >> deg`
	`1`
	`>>> a * 2 ** (-deg)`
	`1.0`

Возвратимся к определенным ранее *x*, *y*. Побитовый сдвиг *x* вправо на 4 бита и побитовый сдвиг *y* влево на 3 бита представлены в табл. 2.10 – 2.11 соответственно.

Таблица 2.10 – Побитовый сдвиг *x* вправо на 4 бита

Разряды:	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	*0*	*0*	1	0	0	0	0	1	0	1	0	1	1	0
>>	*0*	*0*	*0*	*0*	*0*	*0*	1	0	0	0	0	1	0	1

Таблица 2.11 – Побитовый сдвиг *y* влево на 3 бита

Разряды:	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Y	1	0	0	0	0	0	0	1	1	0	0	0	1	1
<<	0	0	0	0	1	1	0	0	0	1	1	0	0	0

Результат, полученный при выполнения примера кода выше, отличается от представленного в таблице: не были вытеснены старшие три разряда, но были дописаны три младших, равных нулю.

Операция отрицания инвертирует каждый отдельно взятый бит числа. Рассмотрим следующий пример:

```
>>> y = 8291
... print(' y =', bin(y))
... print('~y =', bin(~y))
...
 y = 0b10000001100011
~y = -0b10000001100100
```

Ожидалось: 01111110011100. Как добиться ожидаемого результата побитового отрицания? Давайте еще раз посмотрим на таблицу с результатами выполнения побитового исключающего ИЛИ: если биты одинаковые, то получается ноль, если биты разные – единица. При выполнении побитового отрицания все единицы преобразуются в нули (по логике исключающего ИЛИ: сделать ^ с единицами), а нули – в единицы (по логике исключающего ИЛИ: сделать ^ с нулями), то есть для инверсии числа достаточно сделать ^ с числом также же битовой длины, состоящим полностью из единиц. Назовем такое число *z*. Как составить z? Сперва рассмотрим функцию, которая позволяет узнать длину числа в битовой записи, а именно: *int.bit_length()*. Узнаем битовую длину нашего числа *y*:

```
>>> y = 8291
... b_len = y.bit_length()
>>> b_len
14
```

Чтобы узнать количество бит в записи числа можно воспользоваться другим способом – методом *log()* из *math*, рассмотренным в разделе “1.45 Модули”, как показано далее:

```
>>> import math
>>> int(math.log(y, 2)) + 1
14
```

Более того, длину битового представления можно узнать, используя рассмотренный ранее побитовый сдвиг вправо и цикл из раздела “1.23.2 Цикл while”:

```
>>> shift = y >> 1
>>> k = 1
>>> while shift != 0:
...     k += 1
...     shift >>= 1
>>> k
14
```

Таким образом, нам необходимо число, состоящее из 14 единиц. Для

этого сначала запишем его в привычном представлении в табл. 2.12.

Таблица 2.12 – Представление числа z

Разряды:	13	12	11	10	9	8	7	6	5	4	3	2	1	0
z	1	1	1	1	1	1	1	1	1	1	1	1	1	1

В десятичной системе счисления данное число $z = 11111111111111_2 = 16383_{10}$. Можно заметить, что ближайшее число к 16383_{10} – это 2 в степени 14, то есть: $2^{14} = 16384_{10}$. А числа 16383_{10} и 16384_{10} отличаются друг от друга только на единицу, а именно:

$$16384_{10} - 1_{10} = 16383_{10}$$

Перепишем в двоичном виде:

$$100\ 0000\ 0000\ 0000_2 - 1_2 = 11\ 1111\ 1111\ 1111_2$$

Добавим в выражение слева степень двойки:

$$2^{14} - 1 = 16383_{10}$$

Запрограммируем это преобразование и посмотрим, что получилось:

```
>>> y = 8291
... b_len = y.bit_length()
... z = 2 ** b_len - 1
>>> z
16383
>>> bin(z)
'0b11111111111111'
```

Видим, что это именно тот результат, который и был нам нужен. Давайте теперь попробуем применить исключающее ИЛИ для *y* и полученного числа *z*. Пример кода:

```
>>> y = 8291
... b_len = y.bit_length()
... z = 2 ** b_len - 1
>>> bin(y)
'0b10000001100011'
>>> bin(z)
'0b11111111111111'
>>> bin(y ^ z)
'0b1111110011100'
```

Представим выполненную побитовую операцию для исключающего ИЛИ в табл. 2.13.

Таблица 2.13 – Выполнение операции побитового исключающего ИЛИ

Разряды:	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Y	1	0	0	0	0	0	0	1	1	0	0	0	1	1

Z	1	1	1	1	1	1	1	1	1	1	1	1	1	1
^	*0*	1	1	1	1	1	1	0	0	1	1	1	0	0

Действительно, получили в результате побитовое отрицание. Обращаем ваше внимание, что для используемого таким образом числа *z* есть специальное название - *маска*. Маска – это специальная последовательность бит (в большинстве рассматриваемых случаев – число), которое позволяет установить в ноль или единицу определенные биты в определенном объекте. Операция применения маски называется *маскированием*. Например, используя битовую операцию И с маской 0b000010000 можно узнать значение 4-го бита (считаем с нуля). Чтобы установить третий бит числа в единицу, достаточно выполнить операцию ИЛИ с маской: 0b1000.

Представленных выше побитовых операций достаточно для выполнения любых манипуляций с битами.

2.2.3 Конечная точность представления, переполнение

Как уже упоминалось ранее, в памяти компьютеров для хранения данных выделено фиксированное количество бит. Представьте, что есть компьютер с 8-битной архитектурой, т.е. для представления данных доступно 8 разрядов. Что будет, если вдруг нам придется сохранить в памяти этого компьютера результат сложения двух чисел, например, 147_{10} и 209_{10}? В десятичной системе счисления результат вычислить просто:

$$147_{10} + 209_{10} = 356_{10}$$

Запишем выполнение операции сложения чисел в двоичном представлении в табл. 2.14, учитывая, что для хранения данных всего 8 разрядов.

Таблица 2.14 – Сложение десятичных чисел в двоичном представлении

Разряды:	7	6	5	4	3	2	1	0
147_{10}	1	0	0	1	0	0	1	1
209_{10}	1	1	0	1	0	0	0	1
+	**0**	1	1	0	0	1	0	0

Переведем результат из последней строки в десятичную систему счисления, и получим 100_{10}, что не совпадает с ожидаемым результатом. Видим, что старший бит, равный единице, на самом деле, расположен в девятом разряде и никак не может разместиться в 8-битной архитектуре.

Такая ситуация называется *переполнением* – когда определенные данные не умещаются в памяти, ограниченной определенным количеством бит, или, когда результат выполнения операции не помещается в наперед заданное количество разрядов: не хватает ячеек памяти для записи числа.

Из вышесказанного следует определение: числа конечной точности – это числа, представимые в фиксированном количестве разрядов. Арифметические операции с числами конечной точности имеют ограничения и могут вызвать переполнение. При работе с числами в языке Python переполнения нет. Нужно понимать, что программы на Python выполняются на том же самом аппаратном обеспечении и с использованием тех же самых принципов, а нюансы реализации скрыты от Python-программиста, внутри языка используются специальные библиотеки для работы с большими числами.

Вещественные числа типа *float*, т.е. числа с плавающей точкой, представлены в памяти компьютера как дроби с основанием 2 (двоичная система счисления) (о представлении вещественных чисел в памяти речь пойдет в разделе “2.2.5 Формат представления чисел с плавающей точкой”). Термин “плавающая точка” возникает при представлении числа в формате, например, $123 \cdot 10^{-2}$, 123E-2, что в записи с фиксированной точкой, соответственно, равно: 1,23. В языке Python стандартный тип *float* имеет двойную точность, то есть хранится в виде 64-битной строки. В других языках двойная точность достигается при использовании типа *double*.

Обращаясь к технической стороне представления информации в битовом виде, следует помнить, что в современных компьютерах минимальный размер блока памяти, с которым можно выполнять операции, – один байт (8 бит).

2.2.4 Формат представления целых знаковых чисел

Наряду с беззнаковыми числами в памяти компьютера могут храниться и знаковые числа. В течение развития истории вычислительной техники для представления знаковых целых чисел укоренились три формы представления: прямой код, обратный код и дополнительный код (перечислены в порядке увеличения сложности алгоритмов представления). Рассмотрим каждый код в отдельности.

1) Прямой код

В прямом коде представления знаковых чисел самый старший бит

отводится под знак (знаковый бит): 0 – число неотрицательное, 1 – число отрицательное. Чтобы представить число в прямом коде для *n*-разрядной архитектуры необходимо:

1. Во все биты, кроме старшего, записать двоичное представление модуля числа.
2. Сделать старший бит знаковым.

Выделение знакового бита уменьшает на один разряд общее количество разрядов для представления чисел в *n*-разрядной архитектуре по сравнению с диапазоном представления беззнаковых чисел. Например, для 8-битной архитектуры значения, представляемые в обратном коде, меняются от $-127...127$. При этом минимальное число -127_{10} принимает вид (прямой чертой отделяем знаковый бит): $1 \vee 111\,1111_2$, максимальное – $127_{10} = 0¿111\,1111_2$. Однако, выделение знакового бита позволяет двояко представить ноль: отрицательный $-0_{10} = 1 \vee 000\,0000_2$ и положительный $0_{10} = 0 \vee 000\,0000_2$ (всего получается 256 штук значений: 127 значений положительных + 127 значений отрицательных + два представление нуля).

Таким образом, диапазон представления знаковых чисел в прямом коде для *n*-разрядной архитектуры имеет вид:

$$-2^{n-1}+1...2^{n-1}-1$$

В табл. 2.15 – 2.17 рассмотрим пример представления числа -123_{10} в прямом коде для 16-, 8- и 4-битной разрядностей (старший бит отводится под знак).

Таблица 2.15 – Представление числа -123_{10} в прямом коде, разрядность:16

Разряды:	***15***	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-123_{10}	***1***	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1

Таблица 2.16 – Представление числа -123_{10} в прямом коде, разрядность:8

Разряды:	***7***	6	5	4	3	2	1	0
-123_{10}	***1***	1	1	1	1	0	1	1

Таблица 2.17 – Представление числа -123_{10} в прямом коде, разрядность:4

Разряды:	***3***	2	1	0
-123_{10}	***1***	0	1	1

В последнем примере количество бит для представления числа недостаточно, наблюдается переполнение.

- ***Сложение чисел в прямом коде***

Рассмотрим несколько примеров выполнения математических операций над числами в прямом коде. Сложение чисел в прямом коде должно приводить к результату, представленному также в прямом коде. Отметим, что вычитание чисел будем интерпретировать как сложение с отрицательными числами. Допустим, есть два отрицательных числа: -123_{10} и -2_{10}. Сложим их в фиксированной, 8-разрядной архитектуре (ожидается результат: -125_{10}) в табл. 2.18.

Таблица 2.18 – Сложение чисел -123_{10} и -2_{10}, разрядность: 8

Разряды:	***7***	6	5	4	3	2	1	0
-123_{10}	***1***	1	1	1	1	0	1	1
-2_{10}	***1***	0	0	0	0	0	1	0
?	***0***	1	1	1	1	1	0	1

Сложение знаковых битов ($1_2+1_2=10_2$) приводит к переполнению, поэтому результат – положительное число 125_{10} – является неверным.

Следующий пример, сложение -123_{10} и 2_{10}, представлен в табл. 2.19. Ожидается -121_{10}.

Таблица 2.19 – Сложение чисел -123_{10} и 2_{10}, разрядность: 8

Разряды:	***7***	6	5	4	3	2	1	0
-123_{10}	***1***	1	1	1	1	0	1	1
2_{10}	***0***	0	0	0	0	0	1	0
?	***1***	1	1	1	1	1	0	1

Полученный результат -125_{10} некорректный.

Теперь сложим числа 123_{10} и -2_{10} и представим результат в табл. 2.20. Ожидается положительное число 121_{10}.

Таблица 2.20 – Сложение чисел 123_{10} и -2_{10}, разрядность: 8

Разряды:	***7***	6	5	4	3	2	1	0
123_{10}	***0***	1	1	1	1	0	1	1
-2_{10}	***1***	0	0	0	0	0	1	0

?	***1***	1	1	1	1	1	0	1

Результат: -125_{10} – снова некорректный.

- ***Особенности прямого кода***

Таким образом, особенности представления знаковых чисел в прямом коде таковы:

- Уменьшается диапазон представления, т.к. один бит был отведен под знак (поэтому выделяют группы знаковых и беззнаковых чисел, и для представления второй группы диапазон шире).
- Операция сложения “+” приводит к некорректным результатам.
- Среди прочих недостатков – возможность представления положительного и отрицательного нуля: +0 и -0.

Наличие положительного и отрицательного нуля добавляет неопределенность при использовании операций сравнения.

В связи с вышеперечисленными особенностям прямой код для представления отрицательных чисел не используется.

2) Обратный код

Разработка обратного кода была нацелена на исключение недостатков прямого кода, но, к сожалению, и обратный код оказался неидеальным. В обратном коде также старший бит – знаковый. Обратный код для положительного числа совпадает с прямым кодом этого числа. Для представления отрицательного числа в обратном коде необходимо:

1. Во все биты, кроме старшего, записать инвертированное двоичное представление модуля числа.
2. Старший бит сделать знаковым.

Для 8-битной архитектуры минимальное число, представимое в обратном коде с учетом знакового бита: $-127_{10}=1\vee 000\,0000_2$ (помните, что для отрицательного числа инвертированы биты модуля числа), а максимальное число: $127_{10}=0\vee 111\,1111_2$(представление положительных чисел совпадает с прямым кодом). К сожалению, двоякое представление нуля осталось: отрицательный ноль $-0_{10}=1\vee 111\,1111_2$ и положительный ноль$0_{10}=0\vee 000\,0000_2$ (всего получается 256 штук значений: 127 значений положительных + 127 значений отрицательных + два представление нуля). Таким образом, диапазон представления знаковых чисел в обратном коде для *n*-разрядной архитектуры так же, как и в случае прямого кода, составляет:

$$-2^{n-1}+1 \ldots 2^{n-1}-1$$

Рассмотрим представление числа -123_{10} в обратном коде для 8-битной архитектуры в табл. 2.21.

Таблица 2.21 – Представление числа -123_{10} в прямом и обратном кодах, с разрядностью 8

Разряды:	***7***	6	5	4	3	2	1	0	Код:
-123_{10}	***1***	1	1	1	1	0	1	1	пр. код
-123_{10}	***1***	0	0	0	0	1	0	0	обр. код

Обратный код еще называют дополнением до единицы.

- ***Сложение чисел в обратном коде***

Рассмотрим несколько примеров выполнения математических операций над числами в обратном коде. Сложение чисел в обратном коде должно приводить к результату, представленному также в обратном коде. Допустим, необходимо выполнить сложение двух чисел: 23_{10} - 58_{10}, в результате: -35_{10} в обратном коде. В табл. 2.22 в первой строке перечислены разряды (0...7) 8-битной архитектуры, в первом столбце – номера строк для представления промежуточных результатов.

Таблица 2.22 – Сложение чисел 23_{10} - 58_{10} в обратном коде, разрядность: 8

№	Разряды:	***7***	6	5	4	3	2	1	0	Код:
1	23_{10}	***0***	0	0	1	0	1	1	1	пр. код
2	-58_{10}	***1***	0	1	1	1	0	1	0	пр. код
3	-58_{10}	***1***	1	0	0	0	1	0	1	обр. код
4	-35_{10}	***1***	1	0	1	1	1	0	0	обр. код
5	-35_{10}	***1***	0	1	0	0	0	1	1	пр. код

В результате операции получили корректный результат – строка 4 – число -35_{10}.

Следующий пример, -13_{10} + 81_{10}, представлен в табл. 2.23. Ожидаемый результат – положительное число: 68_{10}.

Рассматривая фактический результат выполнения операции в 4-й строке таблицы видим, что результат неверный, при этом произошло переполнение в строке 3. Для достижения корректности результата в таких случаях

приходится выполнять дополнительное действие: к фактическому результату добавлять вытесненный в результате переполнения бит (строка 5). Сложение с вытесненным битом позволяет получить корректный результат, представленный в сроке 6.

Таблица 2.23 – Сложение чисел $-13_{10} + 81_{10}$ в обратном коде, разрядность: 8

№	Разряды:	*7*	6	5	4	3	2	1	0	Код:
1	-13_{10}	***1***	0	0	0	1	1	0	1	пр. код
2	-13_{10}	***1***	1	1	1	0	0	1	0	обр. код
3	81_{10}	***0***	1	0	1	0	0	0	1	пр. код
4	67_{10}	***0***	1	0	0	0	0	1	1	рез-тат
5	+								1	*перенос*
6	68_{10}	***0***	1	0	0	0	1	0	0	пр. код

В табл. 2.24 рассмотрим пример -15_{10} - 15_{10}, ожидаемый результат: -30_{10} в обратном коде.

Таблица 2.24 - Сложение чисел -15_{10} - 15_{10} в обратном коде, разрядность: 8

№	Разряды:	*7*	6	5	4	3	2	1	0	Код:
1	-15_{10}	***1***	0	0	0	1	1	1	1	пр. код
2	-15_{10}	***1***	1	1	1	0	0	0	0	обр. код
3	-15_{10}	***1***	1	1	1	0	0	0	0	обр. код
4	-31_{10}	***1***	1	1	0	0	0	0	0	рез-тат
5	+								1	*перенос*
6	-30_{10}	***1***	1	1	0	0	0	0	1	обр. код
7	-30_{10}	***1***	0	0	1	1	1	1	0	пр. код

Рассматривая результаты сложения двух отрицательных чисел столкнулись с необходимостью прибавления вытесненного бита в строке 5 (результат прибавления 1 в строке 6 – корректный результат сложения). Как и в предыдущем примере операция является дополнительной. В строке 7 дано представление в прямом коде.

- ***Особенности обратного кода***

Особенности представления знаковых чисел в обратном коде такие же, как и в случае с прямым кодом, а именно:

- Не удалось избежать положительного +0 и отрицательного нуля -0.
- Операция сложения все еще не работает.

3) Дополнительный код

Принимая во внимание критические недостатки прямого и обратного кода, был разработан *дополнительный* код для представления знаковых чисел. В дополнительном коде старший бит – знаковый. Дополнительный код положительного числа совпадает с прямым кодом этого числа. Для представления отрицательного числа в обратном коде необходимо:

1. Представить число в обратном коде.
2. Прибавить единицу к представлению числа в обратном коде.

В табл. 2.25 рассмотрим представление в дополнительном коде чисел для архитектуры из трех бит. В первом столбце представлены числа в десятичной системе счисления. Во втором столбце – их двоичное представление. В третьем столбце – инвертированное двоичное представление. В четвертом столбце вертикальной чертой выделен знаковый (старший) бит. В пятом столбце строится дополнительный код – выполняется дополнение до двух. В шестом столбце – десятичное представление чисел в дополнительном коде (сокращение СС – система счисления).

Таблица 2.25 – Представление в дополнительном коде чисел для архитектуры из трех бит

В 10-СС	**В 2-СС**	**Инвертируем**	**Выделяем знаковый бит**	**Дополняем до двух**	**В 10-СС и доп. кода**
0	000	111	1|11	0|00	0
1	001	110	1|10	1|11	-1
2	010	101	1|01	1|10	-2
3	011	100	1|00	1|01	-3
4	100	011	0|11	1|00	-4
5	101	010	0|10	0|11	3
6	110	001	0|01	0|10	2
7	111	000	0|00	0|01	1

Из таблицы видно, что ноль остался на месте (и его представление единственно) и что левая граница включает на одно значение больше, чем правая.

Для 8-битной архитектуры минимальное число в дополнительном коде

$-128_{10}=1\vee 000\,0000_2$, максимальное – $127_{10}=0\vee 111\,1111_2$. Представления нуля единственно: $0\vee 000\,0000_2$. С учетом единственного представления нуля и расширения левой границы до -128, всего получаем 256 штук значений в дополнительном коде для 8-битной архитектуры.

Таким образом, диапазон значений в дополнительном коде таков:

$$-2^{n-1}\ldots 2^{n-1}-1$$

Дополнительный код называют еще дополнением до двух – дополнение до двойки в степени *n*, где *n* – число разрядов.

В табл. 2.26 представлен пример представления числа -123_{10} для 8-битной архитектуры в дополнительном коде.

Таблица 2.26 – Представление числа -123_{10} в прямом, в обратном и в дополнительном кодах, разрядность:8

Разряды:	***7***	6	5	4	3	2	1	0	Код:
-123_{10}	***1***	1	1	1	1	0	1	1	пр. код
-123_{10}	***1***	0	0	0	0	1	0	0	обр. код
-123_{10}	***1***	0	0	0	0	1	0	1	доп. код

- ***Сложение чисел в дополнительном коде***

Дополнительный код позволяет складывать положительные и отрицательные числа, используя обычные правила сложения (без выполнения дополнительных действий). Давайте в качестве примера рассмотрим в табл. 2.27 сложение таких чисел: -13_{10} + 81_{10}, где ожидаемый результат – положительное число 68_{10}.

Видим, что результат – положительное число 68_{10} – получен сразу в корректном представлении – в прямом коде, не пришлось выполнять дополнительных действий.

Таблица 2.27 – Сложение чисел -13_{10} + 81_{10}, разрядность: 8

№	Разряды:	***7***	6	5	4	3	2	1	0	
1	-13_{10}	***1***	0	0	0	1	1	0	1	пр. код
2	-13_{10}	***1***	1	1	1	0	0	1	0	обр. код
3	-13_{10}	***1***	1	1	1	0	0	1	1	доп. код
4	81_{10}	***0***	1	0	1	0	0	0	1	пр. код
5	68_{10}	***0***	1	0	0	0	1	0	0	пр. код

Следующий пример, 23_{10} - 58_{10}, представлен в табл. 2.28. Ожидаем

результат: отрицательное число -35_{10} в дополнительном коде.

Таблица 2.28 – Сложение чисел 23_{10} - 58_{10}, разрядность: 8

№	Разряды:	*7*	6	5	4	3	2	1	0	Код:
1	23_{10}	***0***	0	0	1	0	1	1	1	пр. код
2	-58_{10}	***1***	0	1	1	1	0	1	0	пр. код
3	-58_{10}	***1***	1	0	0	0	1	0	1	обр. код
4	-58_{10}	***1***	1	0	1	1	1	**1**	**0**	доп. код
5	-35_{10}	***1***	1	0	1	1	1	0	1	доп. код
6	-35_{10}	***1***	0	1	0	0	0	1	0	инверсия
7	-35_{10}	***1***	0	1	0	0	0	1	1	+1 => пр. код

Строки 6 (инвертирование числа – это -35_{10} в обратном коде) и 7 (прибавление единицы – это -35_{10} в прямом коде) демонстрируют правильность результата, представленного в строке 5 – число -35_{10} в дополнительном коде.

В табл. 2.29 представлен пример сложения чисел -15_{10} - 15_{10}, ожидаемый результат – отрицательное число -30_{10} в дополнительном коде.

Таблица 2.29 - Сложение чисел -15_{10} - 15_{10}, разрядность: 8

№	Разряды:	*7*	6	5	4	3	2	1	0	Код:
1	-15_{10}	***1***	0	0	0	1	1	1	1	пр. код
2	-15_{10}	***1***	1	1	1	0	0	0	0	обр. код
3	-15_{10}	***1***	1	1	1	0	0	0	1	доп. код
4	-30_{10}	***1***	1	1	0	0	0	1	0	доп. код
5	-30_{10}	***1***	0	0	1	1	1	0	1	обр. код
6	-30_{10}	***1***	0	0	1	1	1	1	0	пр. код

Как и в примере для обратного кода, в строке 4 произошло переполнение. В случае с дополнительным кодом, переполнение – основа правильного выполнения операции, и поэтому в этой строке видим корректный результат – число -30_{10} в дополнительном коде. Результат подтверждается представлением числа -30_{10} в обратном коде в строке 5 и представлением его же в прямом коде в строке 6.

- ***Особенности дополнительного кода***

Дополнительный код обладает следующими преимуществами:

- Для операции сложения: нет необходимости совершать дополнительные действия, помимо основной операции сложения.
- В дополнительном коде ноль имеет единственную беззнаковую запись.

2.2.5 Формат представления чисел с плавающей точкой

Формат представления вещественных чисел отличается от представления целых чисел. Отличие заключается в способе двоичного представления.

Вспомним определения рациональных и иррациональных чисел. *Рациональные* числа – это такие числа, которые можно представить в виде отношения двух целых чисел. Некоторые рациональные числа не так просто представить в виде десятичной дроби, например, 1/3. Разделив 1 на 3, получим: 0.3333333(3) (говорят – три в периоде).

Числа, которые нельзя представить в виде отношения целых чисел, называют *иррациональными*. Они записываются в виде бесконечных десятичных дробей без каких бы то ни было повторений. В качестве примера можно привести числа π и e.

Всё множество рациональных и иррациональных чисел называют *вещественными* числами.

Из предыдущего раздела нам известно, что, например, в 32-битовой ячейке памяти можно хранить положительные целые числа от 0 до 4 294 967 295. А как хранить дроби?

1) Основные сведения: мантисса, порядок и смещенный порядок

Начнем с того, что любое целое число можно представить в виде набора степеней основания определенной системы счисления, например, так: $150\,000\,000\,000_{10}$ выглядит как: $1{,}5 \cdot 10^{11}$, а число $0{,}00000000005_{10}$ – так: $5 \cdot 10^{-11}$ или $0{,}5 \cdot 10^{-10}$, или $0{,}05 \cdot 10^{-9}$ и т.д.

В таком представлении имеется несколько новых определений. Так, число перед степенью 10 называют *мантиссой*. Степень, в которую возводится 10, называется *порядком*. Рассмотрим в табл. 2.30 простые примеры.

Таблица 2.30 – Мантисса и порядок. Примеры

№	Число	Мантисса	Порядок
1	$123 \cdot 10^{-2}$	123	-2
2	$1{,}5 \cdot 10^{11}$	1,5	11
3	$5 \cdot 10^{-1}$	5	-1

Можно по-разному выбирать мантиссу и порядок: $1{,}5 \cdot 10^{11} = 15 \cdot 10^{10} =$

$150 \cdot 10^9 = 0{,}15 \cdot 10^{12} = 0{,}015 \cdot 10^{13}$, но первый вариант наиболее предпочтителен, т.к. в целой части остается одна цифра, далее покажем, почему это важно. Традиционно значащая часть в записи вещественного числа – *мантисса* – заключена между 1 (включительно) и 10 (не включая) (для десятичной системы счисления). Нотация, использующая в представлении чисел порядок и мантиссу, называется *научной*, и ее общий вид таков:

$$N = M \cdot n^p$$

где N – представляемое число, M – мантисса в представлении числа, n – основание показательной функции (системы счисления), p – порядок (всегда целое число).

В компьютерах научная нотация стала основой для записи чисел в формате с плавающей точкой (*floating point*). Плавающей точка называется потому что достоверно неизвестно, сколько десятичных разрядов присутствует в записи числа.

Научная нотация чисел используется также для записи двоичных чисел. В двоичной записи цифры справа от запятой (разделителя целой и дробной частей) соответствуют отрицательным степеням 2. Научная нотация подразумевает, что в записи мантиссы двоичных чисел в целой части остается одна единица: ноль не может быть первым в двоичной записи числа (незначащие нули), так что в начале числа всегда стоит единица. Зная это, можно сэкономить один разряд при записи числа в память, опуская первую единицу. Например, дробное число в двоичной записи:

$$111.1101_2 = 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4}$$

в научной нотации можно представить так:

$$111.1101_2 = 1.111101 \cdot 2^2$$

Порядок говорит о том, на сколько надо сдвинуть запятую, а знак порядка – в какую сторону надо сдвигать, чтобы получить исходное число.

В большинстве современных компьютеров для хранения чисел с плавающей точкой применяется стандарт IEEE 754 [10], который обеспечивает представление вещественных чисел в уже знакомом нам виде:

$$(-1)^s \times c \times b^q$$

где s – знак (0 или 1), c – мантисса (коэффициент), b – основание (например, 2 или 10), q – экспонента (порядок).

Такое представление применяется для реализации двоичной арифметики с плавающей точкой. Стандарт задает ряд основных форматов, которые

определяются по количеству битов, используемых в их кодировке. Существуют три базовых формата двоичной плавающей запятой:

- кодировка 32 битами (одинарная точность, binary32),
- кодировка 64 битами (двойная точность, binary64),
- кодировка 128 битами (четырехкратная точность, binary128),

и два десятичных формата с плавающей запятой:

- кодировка 64 битами (decimal64),
- кодировка 128 битами (decimal128).

Описание форматов половинной точности (half precision) и расширенной точности (extended precision) можно найти в [10]. Далее рассмотрим одиночный (одинарная точность, binary32) и двойной (двойная точность, binary64) форматы представления плавающей запятой в двоичном виде, которые относятся к версии стандарта IEEE 754-1985 [10].

2) Одинарная точность

Согласно стандарту для представления чисел с плавающей точкой с одинарной точностью (*single precision*) под хранение числа отводится 4 байта (аналогично тому, как представлен тип *float* в языке С). Диапазон представляемых значений составляет от $-3.4\times10^{38}..3.4\times10^{38}$. Одинарная точность обеспечивает относительную точность 7-8 десятичных цифр в указанном диапазоне.

Для представления используем следующие положения:

- 1 бит – знак (0 - положительные числа, 1 - отрицательные)
- 8 бит – порядок
- 23 бита – дробная значащая часть числа - мантисса
- 127 – смещение, используемое для получения смещенного порядка (смещенный порядок: истинный порядок + 127)

Рассмотрим пример: 111.1101_2. Первое, что нужно сделать для получения представления числа с одинарной точностью – это сдвинуть запятую за старшую единицу числа, то есть: $1.111101_2\cdot2^2$, выделив тем самым мантиссу – 1.111101_2. В качестве мантиссы в память записывается только дробная часть (одна единица слева от запятой осознанно отбрасывается – таково соглашение). Истинный порядок в рассматриваемом примере – 2, смещенный порядок – 129:

$2_{10}+127_{10}=129_{10}=10000001_2$

Запишем в табл. 2.31 выделенные мантиссу и порядок (помним про

первый знаковый бит, в данном случае число положительное).

Таблица 2.31 – Порядок и мантисса

знак	порядок								мантисса								
0	1	0	0	0	0	0	0	1	1	1	1	1	0	1	0	…	0

Число с одинарной точностью занимает 32 бита (или 4 байта): 1 бит для знака (0 – для положительных чисел и 1 – для отрицательных), 8 бит для порядка, 23 бита для дробной значащей части числа, в которой самый младший бит стоит справа. Первый бит, который соответствует отброшенной единице слева от запятой, не включается, хранится только 23-битовая дробная часть, но подразумевается, что точность составляет 24 бита.

Особого внимания заслуживает порядок. 8-битовый порядок может принимать значения от 0 до 255. Он является смещенным (*biased*) (обозначим как *E*), т. е. для нахождения истинного значения порядка (с учетом знака) необходимо вычесть из *E* число, называемое смещением (*bias*). Для чисел одинарной точности с плавающей точкой смещение равно 127. Использование смещенного порядка позволяет записывать в 8-бит и отрицательные, и положительные истинные порядки без использования знакового бита.

Если значение порядка заключено в пределах от 1 до 254, число, представленное конкретными значениями *s* (бита знака), *p* (порядка) и *m* (дробная часть мантиссы), выглядит так:

$$(-1)^s \cdot 1.m \cdot 2^{E-127}$$

где смещенный порядок $E = p + 127$ определяется на основании истинного порядка p.

Для формата одинарной точности есть специальные случаи, которые заслуживают особого внимания:

- Если порядок и мантисса равны 0, число равно 0. Обычно 0 представляется нулевыми значениями всех 32 битов. Если бит знака равен 1, число называется отрицательным 0. Он символизирует очень маленькое число, для записи которого недостаточно цифр и степени в простой точности, но это число меньше 0, а не равно ему.
- Если порядок равен 255 и мантисса равна 0, число в зависимости от знака является -∞ или +∞.
- Если порядок равен 255 и мантисса не равна 0, значение считается

недопустимым числом и является NaN (Not a Number).

Рассмотрим ряд примеров, чтобы понять на практике, как работает представления с одинарной точностью.

Допустим, от нас требуется побитовое представление отрицательного числа -12.625_{10} в памяти компьютера для одинарной точности. Первое, что мы сделаем, это переведем число в двоичную систему счисления. Напомним, что алгоритм перевода дробного числа из десятичной системы счисления в двоичную может быть таким:

- целая часть числа переводится привычным образом – последовательным делением на два и фиксированием остатка,
- дробная часть – последовательным умножением на два и фиксированием целой части, со своевременным отбрасываем из целой части единицы (продолжаем умножение до получения числа 1,0). В худшем случае, умножение на два может продолжаться бесконечно, но нужно понимать, что для представления мантиссы отведено всего лишь 23 бита, часть из которых будет занята битами целой части.

Таким образом, выполнили перевод в двоичную систему счисления:

$$-12.625_{10}=1100.101_2$$

Далее требуется выполнить сдвиг запятой за самую старшую единицу, скорректировав при этом степень двойки:

$$1100.101_2=1.100101_2\cdot 10^{101}=1.100101_2\cdot 2^3$$

Имея такую запись, понимаем, что истинный порядок равен $3_{10}=101_2$, смещенный:

$$E=3_{10}+127_{10}=130_{10}=1000\,0010_2$$

Этой информации достаточно, чтобы представить число в требуемом виде:

1 | 1000 0010 | 100 1010 0000 0000

3) Двойная точность

Согласно стандарту IEEE 754 для хранения чисел с плавающей точкой в двойной точности (*double precision*) используется 8 байт (что соответствует типу *double* в C и типу *float* в Python). Такое количество бит обеспечивает точность в 15-17 десятичных цифр в диапазоне $-1.7\times10^{308}..1.7\times10^{308}$. Формат представления таков:

- 1 бит - знак (0 - положительные числа, 1 - отрицательные)
- 11 бит - порядок

- 52 бита - дробная значащая часть числа - мантисса
- 1023 - смещение (для получения смещенного порядка)

Обратимся к примеру: $111{,}1101_2$. Выделим мантиссу, перенеся запятую к самой старшей единице, а именно: $1{,}111101 * 2^2$, таким образом, 1,111101 - мантисса, 2 - истинный порядок, 1025 - смещенный порядок (см. табл. 2.32).

Таблица 2.32 – Порядок и мантисса

знак	порядок							мантисса										
0	0	1	0	...	0	0	1	1	1	1	1	0	1	0	...	0	0	0

Для формата двойной точности есть специальные случаи, которые заслуживают особого внимания:

1. Для значений 0, бесконечности и NaN применяются те же правила, что и в простой точности.
2. Сложение осуществляется с преобразованием к одинаковой степени.

Например: $1.11 \cdot 2^5 + 10.01 \cdot 2^3$. В данном случае сложение выполняется следующим образом:

$$1.11 \cdot 2^5 + 10.01 \cdot 2^3 = 111 \cdot 2^3 + 10.01 \cdot 2^3 = 1001.01 \cdot 2^3 = ¿$$
$$¿\, 1001010_2 = 1.001010 \cdot 2^6$$

4) Сравнение чисел с заданной точностью

Мы уже обращали внимание на опасность представления вещественных (в особенности иррациональных) чисел в конечно разрядной архитектуре, обусловленную технической невозможностью обеспечить точное представление некоторых значений. Ограничения аппаратных средств, реализующих вещественную математику, могут послужить причиной появления неожиданного поведения программы, работающей с вещественными числами. Странное поведение может выражаться при следующих обстоятельствах:

```
>>> 0.1+0.1+0.1
0.30000000000000004
>>> 0.2+0.2+0.2+0.2+0.2+0.2+0.2+0.2
1.5999999999999999
```

В [4, с. 164] отмечается, что все цифры в приведенном выше результате действительно присутствуют в аппаратной части компьютера, выполняющей операции над числами с плавающей точкой.

При решении реальных задач такой неожиданный “хвост”

вещественного числа может привести к негативным последствиям. Давайте представим робота-хирурга, выполняющего сложную операцию, от точности которого зависит жизнь человека. Или представим работу банковской системы, где ежесекундно происходят вычисления процентных ставок, когда имеет место накопление ошибки, связанное с особенностями представления числа. В случаях, когда необходимо особенно осторожно обращаться с вещественными числами, прибегают к использованию *представления с заданной точностью*. Заданная точность отражает количество знаков после запятой, которым можно доверять, т.е. те цифры, которые действительно принадлежат определенному числу, а не появились в связи с аппаратными ограничениями. Примерами такой точности могут послужить: $e = 10^{-5}$, где принимаются во внимание только 5 знаков после запятой. Рассмотрим пример использования точности – задача сравнения вещественных чисел. Допустим, есть два числа:

```
>>> x = 3.1415926535
>>> y = 3.1415976535
```

Сравнивая эти два числа визуально, видим, что *y* больше *x*. Но если положить точность для этих двух чисел равной $e = 10^{-5}$, то y окажется равным *x*. Если положить точность равной $e = 10^{-7}$, то y будет больше *x*.

Сравнение с точностью происходит следующим образом: определяется количество знаков после запятой, которым можно доверять, например, при $e = 10^{-5}$ – это пять знаков, и все цифры, идущие после них, отбрасываются при сравнении. То есть сравниваются два таких числа: $x = 3.14159$ и $y = 3.14159$, которые, действительно, равны. Если точность составляет $e = 10^{-7}$, то это семь знаков, которым можно доверять, а, значит, сравниваются числа $x = 3.1415926$ и $y = 3.1415976$, и видно, что *y* больше *x*.

Как программно реализуется сравнение с точностью? Необходимо вычислять разность сравниваемых чисел и сравнивать ее с заданной точностью. Для примера выше продемонстрируем:

```
>>> x = 3.1415926535
>>> y = 3.1415976535
>>> eps1 = 10E-5
>>> eps2 = 10E-7
```

Согласно первой заданной точности имеем равенство чисел:

```
>>> if abs(x - y) <= eps1:
...     print('Равны')
... else:
...     print('Не равны')
```

```
...
Равны
```

Согласно второй заданной точности имеем, что *y* и *x* не равны:

```
>>> if abs(x - y) <= eps2:
...     print('Равны')
... else:
...     print('Не равны')
...
Не равны
```

2.2.6 Формат представления текстовой информации

В качестве базового представления память использует двоичные ячейки, поэтому любую информацию, которую необходимо хранить, сначала нужно преобразовать в цифровую форму (для дальнейшего представления в битах). Для представления текста в цифровом формате необходимо придумать систему кодирования, которая бы позволяла каждому знаку текста сопоставить уникальный цифровой код. Отметим, что цифровые коды понадобятся и для цифр, и для знаков препинания, поскольку и те, и другие могут встречаться в тексте наравне с буквами. Оказывается, что цифры удобно кодировать тем же самым образом, что и буквы, поэтому коды цифр в тексте не связаны с их реальным численным значением. Таким образом, нужны цифровые коды для всех буквенно-цифровых (*alphanumeric*) символов. В итоге приходим к тому, что отдельно взятый код является кодом символа *(character code*). Нужно понимать, что в данном случае речь идет не о шрифте, способе начертания и т.д.: ставится задача представления только простого текста, состоящего из 26 букв латинского алфавита и цифр.

Обращаясь к истории представления данных, отметим, что первая система кодирования букв и цифр была 5-битной, так называемый код Бодо, названный в честь своего создателя – Эмиля Бодо (1845-1903 гг.), который в 1870 году разработал эту систему для своего телеграфа. Для ввода информации в 5-битной системе кодирования использовалась клавиатура, на которой было 5 клавиш. Нажатие определенной клавиши соответствовало передаче одного бита в 5-битном коде.

Стоит отметить, что в реальной жизни мы часто имеем дело с большими компьютерными системами, поэтому для обмена информацией всем пользователям лучше договориться и применять одни и те же системы кодирования. Такая договоренность сможет обеспечить легкий перенос информации между компьютерами.

Вернемся к разработке системы кодирования. В системах кодирования цифровые коды букв лучше располагать упорядоченно: последовательные цифровые коды соответствуют буквам, отсортированным в алфавитном порядке. Система кодирования может быть представлена в табличном виде, где в одном столбце перечисляются символы, а в другом - их цифровые коды соответственно.

История вычислительной техники настолько богата, что ученые этой сферы достаточно давно придумали кодировки, которые активно используются и по сей день. Среди них мы рассмотрим ASCII и Unicode.

Кодировка ASCII (American Standard Code for Information Interchange) начала своё существование в далеком в 1963 году благодаря разработкам ученых из США. Данная кодировка является 7-битовой, таким образом, в ней доступно всего лишь 128 цифровых кодов. Цифровые коды с 1 по 95 относятся к отображаемым символам, т.е. к символам, у которых конкретное визуальное представление (строчные и заглавные буквы из английского алфавита, цифры и др.). В наборе ASCII есть также 33 управляющих символа, которые при печати или на экране не отображаются, а используются для выполнения определенных действий. Например, символ перевода строки \n (line feed), символ горизонтальной табуляции \t (tab), символ вертикальной табуляции \v (vertical tab), перевод (возврат) каретки \r (carriage return, CR), возврат каретки на один символ \b (backspace), перевод страницы \f (form feed), звуковой сигнал \a (alarm), пустой символ \0 (NULL) [11].

При разработке таблицы ASCII потребности других алфавитов учитывались мало, и, конечно, о нелатинских алфавитах речь не шла. В стандарте ASCII считается, что оставшиеся 10 кодов другие страны могут переопределять согласно своим потребностям.

Возвращаясь к количеству бит, поскольку в большинстве компьютерных систем символы хранятся как 8-битовые значения, как следствие, появляется возможность расширения набора символов таблицы ASCII до 256 символов (2^8=256). В расширенной кодировке значения кодов с 00h до 7Fh (первые 128 символов) остались неизменными, а коды с 80h по FFh соответствуют буквам с диакритическими знаками [12] или буквам нелатинских алфавитов. К сожалению, на протяжении последних десятилетий появилось множество различных вариантов расширения таблицы ASCII даже для одного языка, что, конечно же, приводит к многочисленным сложностям. С момента

расширения, ASCII стала восприниматься как половина 8-битной кодировки, а «расширенной ASCII» стали называть ASCII с задействованным 8-м битом (например, КОИ-8).

Осознав необходимость единой и всеобщей системы кодирования символов, которая подходила бы для всех языков мира, в 1988 г. несколько крупных компьютерных компаний начали разработку кодировки Unicode, которая должна прийти на смену ASCII. В отличие от ASCII кодировка Unicode является не 7-, а 16-битовой. По 2 байта занимают все символы Unicode до единого. Это значит, что в Unicode коды принимают значения от 0000h до FFFFh, а всего их доступно 65 536 ($=2^{16}$). Этого достаточно для любых языков мира, по крайней мере для тех, что будут использоваться в компьютерах, и при необходимости возможно расширение. Отметим, что первые 128 символов таблицы Unicode (коды от 0000h до 007Fh) совпадают с символами таблицы ASCII. Unicode включает и греческие, и кириллические, и арабские, и др. символы. Таким образом, Unicode – это уникальная система кодирования, позволяющая получить цифровой код для любого символа, независимо ни от платформы, ни от программы, ни от языка, но только при условии, что этот язык поддерживается стандартом.

1) Функции Python для работы с кодировками

Чтобы обращаться к кодам символов, которые хранятся в таблице Unicode, в Python есть специальные функции получения кода символа:

```
>>> ord('Щ')
1065
```

А также символа по определенному коду:

```
>>> chr(15000)
'㪘'
```

При передаче некорректного кода в функцию *chr* можно получить ошибку:

```
>>> chr(-15)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: chr() arg not in range(0x110000)
```

Для аргумента функции *chr* есть ограничения принимаемого значения: от 0 до 1 114 112 (0x110000). Ответ на вопрос, почему максимальное значение аргумента сильно превышает размер таблицы Юникод, представляем к самостоятельному изучению читателем.

2.3 Машина Тьюринга

2.3.1 Основные сведения

В предыдущих разделах мы узнали о том, из каких компонент строятся вычислительные системы. Несмотря на краткость изложения, очевидно, что из описанных компонент можно сконструировать множество различных вычислительных устройств, которые будут вполне работоспособными. Однако, чтобы изучать принципы работы таких устройств необходима некоторая общая модель их работы. Такие модели называются абстрактными исполнителями или абстрактными вычислительными машинами. В данном разделе будет рассмотрена наиболее популярная модель – машина Тьюринга, схема которой представлена на рис. 2.13.

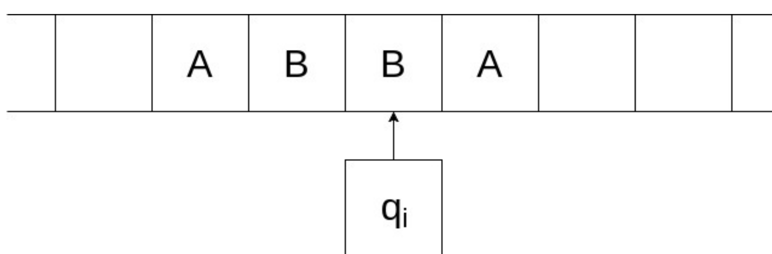


Рисунок 2.13 – Машина Тьюринга

Машина Тьюринга (МТ) была разработана британским математиком, Аланом Тьюрингом (23.06.1912-07.06.1954), в 1936 году для изучения свойств алгоритмов и их вычисления на реальных устройствах. В дополнение к данной модели А. Тьюринг также ввел понятие *вычислимости* – свойства, определяющего возможность вычисления функции с помощью машины Тьюринга. Если для какой-то функции существует вычисляющая ее машина Тьюринга, то такую функцию называют алгоритмически вычислимой. При этом, далеко не каждая функция является вычислимой – решения многих математических проблем не входят в класс вычислимых функций, например *проблема останова* [13].

Машина Тьюринга состоит из неподвижной *ленты* (аналог памяти в реальной вычислительной машине) и *автомата* (процессора). Лента (память) используется для хранения информации. Она бесконечна в обе стороны и разбита на ячейки, которые никак не нумеруются и не именуются. В каждой клетке может быть либо записан один символ, либо ничего не записано. Все возможные символы, которые могут храниться на ленте, образуют *алфавит*.

Алфавит из примера на рис. 2.13. можно записать таким образом: {‘A’, ‘B’, ''}. Важно отметить, что при рассмотрении машины Тьюринга принимается допущение о том, что алфавит всегда конечен.

Помимо машины Тьюринга существуют и другие абстрактные вычислители. Так, например, в качестве машин, оперирующими лентами с данными также выступают машина Поста и машина Минского [14]. Существуют также абстрактные вычислители, оперирующие двумерной памятью, например, муравей Ленгтона [15].

2.3.2 Как работает машина Тьюринга (таблица состояний)

Рассмотрим подробнее, каким образом работает машина Тьюринга. С точки зрения стороннего наблюдателя, ее функционирование заключается в последовательном перемещении автомата вдоль ленты с возможным (но необязательным) изменением символов, хранящихся в ее ячейках. В каждый момент времени автомат размещается целиком только под одной из клеток ленты (автомат не может находиться между клетками) и может прочитать ее содержимое; содержимое других клеток автомат не видит. Перемещение процессора задается программой – правилами перехода. В процессе работы машины Тьюринга в каждый момент времени автомат находится в одном состоянии, которое обычно обозначается буквой q с номерами: q_0, q_1, q_2 и т.д. Правила перехода задает действия, которые автомат должен выполнить, в зависимости от текущего состояния и символа на ленте, а также следующее состояние q_n, в которое автомату необходимо перейти. Существует конечное состояние (терминальное), в котором автомат останавливается (машина Тьюринг останавливает свою работу).

Программа для машины Тьюринга представляется в виде таблицы переходов (см. табл. 2.33). Столбцы соответствуют символам алфавита, а строки – состояниям автомата.

В ячейках таблицы указываются тройки <Symbol', [L, R, N], q'>:

- Symbol' – символ, который необходимо записать в видимую ячейку ленты.
- [L, R, N] – одно из направлений, куда нужно перейти на ленте:
    - R - направо,
    - L - налево,
    - N - остаться на месте.
- q' – состояние, в которое необходимо перейти автомату.

Таблица 2.33 – Структура программы для машины Тьюринга

	$Symbol_1$	$Symbol_2$	...	$Symbol_{n-1}$	$Symbol_n$
q_1					
...			<Symbol', [L, R, N], q'>		
q_m					

Исходя из вышесказанного можно сделать вывод о том, что поведение машины Тьюринга задается конкретной таблицей состояний, а значения на ленте до начала работы машины выполняют роль входных данных.

Рассмотрим примеры описания состояний машины Тьюринга. В качестве задачи рассмотрим поиск определенного символа на ленте. Для простоты будем считать, что на ленте встречаются только два вида символов: 0 и 1. Пусть нам требуется найти 1. При желании, данную задачу можно решить на алфавите любого размера, однако размер программы увеличивается пропорционально. Рассмотрим общий алгоритм решения задачи:

1. Если текущий символ “0”, то сдвигаем автомат вправо и повторяем шаг №1.
2. Если текущий символ это ”1”, то останавливаем каретку.

Очевидно, что у нашей машины будет только два состояния: “клетка не найдена” (q_0) и “клетка найдена” (q_1), являющееся конечным. Для компактности не будем приводить конечные состояния в таблице. Составим таблицу для программы (см. табл. 2.34).

Таблица 2.34 – Структура программы для машины Тьюринга

Состояние \ Символ	0	1
q_0	<0,R,q_0>	<1,N,q_1>

Стоит обратить внимание на то, что в рамках задачи не требуется изменять значения на ленте, поэтому символ для записи в ячейку дублирует символ, указанный в заголовке соответствующей колонки.

Рассмотрим более сложную задачу – инвертирование двоичного числа,

записанного на ленте, например:

11000 => 00111

Будем считать, что число на ленте одно. Кроме числа на ленте находятся пробелы. Начальная позиция автомата находится на старшем разряде числа. Таким образом, алфавит имеет вид {0, 1, ' '}.

Как и в предыдущем примере, для решения будет достаточно двух состояний – “конец числа не достигнут” (q_0), “конец числа достигнут” (q_1). Очевидно, что состояние q_0 является начальным. Находясь в нем, автомат должен выполнить следующие действия:

1. инвертировать текущий символ,
2. сдвинуться вправо,
3. перейти в состояние q_0.

Если в состоянии q_0 и видит на ленте ' ', он выполняет три действия: записывает вместо пробела пробел (т.е. на самом деле ничего не записывает), не двигается по ленте и переходит в состояние q_1.

Таблица состояний машины Тьюринга представлена в табл. 2.35.

Таблица 2.35 – Таблица состояний машины Тьюринга

Состояние \ Символ	0	1	‘ ‘
q_0	1; R; q_0	0; R; q_0	‘ ‘; N; q_1

Несмотря на кажущуюся простоту подобного подхода к заданию поведения машины Тьюринга, с ее помощью можно промоделировать работу любой программы для современных компьютеров. Если свести поведение компьютерной программы к операциям чтения и записи данных, то становится возможным выполнить ее на машине Тьюринга. При этом программы машины Тьюринга можно преобразовать для выполнения на других вычислителях (справедливо и обратное).

2.4 Упражнения и вопросы для самоконтроля

2.4.1 Введение в архитектуру

1. Переведите число в 16ю с.с.:

100011110001

2. Какие основные отличия архитектуры Фон Неймана?
3. Постройте схему из логических вентилей для 4х разрядного сумматора.
4. Какой базовый цикл процессора в архитектуре Фон Неймана?
5. Зачем нужен генератор частот?
6. Какой процессор быстрее: 4х разрядный или 8 разрядный?
7. Что означает разрядность процессора?
8. В чем разница между компиляцией и интерпретацией?
9. Что такое байт-код?

2.4.2 Формат представления данных

1. Подумайте, на какую математическую операцию похожи побитовые сдвиги?
2. Как будет выглядеть маска, устанавливающая в единицу третий, пятый и седьмой бит некоторого числа?
3. Как будет выглядеть маска, делающая второй и четвертый биты некоторого числа равными нулю?
4. Какой результат выражения 211_{10} + 150_{10} будет записан в памяти при вычислении в 8-битном компьютере?
5. Какой результат выражения 14511_{10} - 176_{10} будет записан в памяти при вычислении в 12-битном компьютере?
6. Как будет выглядеть представление числа 101.0265_{10} согласно стандарту IEEE 754-1985 в формате одинарной точности?
7. Как будет выглядеть представление числа -11.0115_{10} согласно стандарту IEEE 754-1985 в формате двойной точности?
8. Напишите программу, которая определяет для пары чисел: x = 0.0001928491 и y = 0.00019384, какое из больше другого согласно заданным точностям: e_1 = 10E-4, e_2 = 10E-5, e_3 = 10E-6, e_4 = 10E-7.

2.4.3 Машина Тьюринга

1. Напишите программу, которая стирает (заменяет пробелом) содержимое первых N ячеек ленты, которые заполнены случайными числами в диапазоне 1-4. Алфавит ленты: $A=\{1, 2, 3, 4, ''\}$
2. Какой подход можно использовать для создания программы по

подсчету количества определенных символов на ленте? Что ограничивает применимость данной программы?

3. Является ли вычислимой на машине Тьюринга функция умножения двух одноразрядных чисел?
4. Напишите программу, которая двигает каретку циклически в пределах первых 4 клеток (сначала на 4 клетки вправо, затем на 4 клетки влево).
5. Напишите программу, которая выполняет сложение двух одноразрядных двоичных чисел. Числа записаны подряд в первые две ячейки, результат необходимо записать в третью ячейку (в случае переполнения - отбросить старший бит).

Выводы по главе

В данной главе были рассмотрены основы архитектуры вычислительного устройства. Были изучены способы и форматы представления различных данных на компьютере. Был изучен принцип работы вычислительных устройств на примере Машины Тьюринга.

ГЛАВА 3. ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

Цели и задачи главы

Цель – рассмотреть понятия парадигм программирования и освоить некоторые из них на практике.

Задачи:

- дать определение парадигме,
- провести классификацию парадигм с примерами языков программирования,
- подробнее рассмотреть реализацию функционального программирования на Python с решением задач на практике,
- изучить ключевые вопросы объектно-ориентированного программирования и на примере ряда практических задач на Python,
- уделить отдельное внимание исключениям, рассмотреть обработку исключительных ситуаций и способы их генерации на Python.

3.1 Введение

Изучение парадигм программирования позволит понять ключевые принципы и правила, согласно которым разрабатываются программы, а также понять главную идею, как программы, написанные в разных парадигмах, работают. Что же касается языка Python, то изучение парадигм программирования позволит по достоинству оценить все плюсы мультипарадигмальности языка.

Парадигмы программирования делятся на две большие группы: императивные и декларативные. Каждая из этих групп, в свою очередь, делится на подгруппы. Например, логическое программирование, функциональное программирование, процедурное и объектно-ориентированное программирование.

Почему мы показываем некоторые парадигмы на примере Python? Потому что Python, как уже упоминалось выше, мультипарадигмальный язык. Использование Python для изучения парадигм позволит увидеть достоинства использования различных парадигм совместно, например, ООП и функциональное программирование.

3.2 Парадигмы программирования

3.2.1 Определение парадигмы

Термин “парадигма программирования” имеет множество определений, но в общем его можно описать так: парадигма программирования - это подход к программированию, описанный совокупностью идей и понятий, определяющих стиль написания компьютерных программ.

Однако, не следует считать, что парадигма программирования однозначно определяется каким-то конкретным языком программирования. Есть языки программирования которые поддерживают несколько парадигм при реализации программ. Такие языки называются мультипарадигменными и идея их авторов заключается в том, что для каждой конкретной задачи может быть в большей степени уместна та или иная парадигма и универсального подхода не существует.

3.2.2 Императивная парадигма

Императивная парадигма, пожалуй, наиболее привычна человеку. Её можно сравнить с последовательностью приказов в повелительном наклонении, каждый их которых определяет команду, которую должен выполнить компьютер. Примером наиболее низкоуровневой реализации данной парадигмы является машинный код, а наиболее низкоуровневым императивным языком - язык ассемблера. Исходя из этого, большинство языков программирования поддерживают эту парадигму. Наиболее известные вам: Pascal, Python, C, C++, Java и другие

Для императивного подхода характерны следующие свойства:

- Исходный код программ состоит из инструкций.
- Инструкции выполняются последовательно.
- Доступны данные после выполнения предыдущих инструкций.
- Используется оператор присваивания, именованные переменные, подпрограммы.

3.2.3 Декларативная парадигма

Декларативная парадигма является противоположностью императивной. Если императивный подход описывает то, как именно решать задачу, то декларативный подход не предполагает последовательного описания инструкций. В декларативном подходе есть только описание того, как поставлена задача и как должен выглядеть результат. Соответственно, в таких программах отсутствуют привычные операторы цикла, подпрограммы и операторы присваивания.

Примером языка использующего декларативный подход может быть язык структурированных запросов -SQL (structured query language) и Prolog.

Один язык может сочетать в себе императивную парадигму и подвиды декларативной. Например, преимущественно императивный язык Python поддерживает функциональную парадигму - подвид декларативной парадигмы. Более подробно об этой парадигме будет рассказано далее.

3.2.4 Логическое программирование

Парадигма логического программирования основана фактически на автоматическом доказательстве некоторых логических утверждений (теорем) и является подвидом декларативной парадигмы.

Самым известным представителем языка логического программирования является Prolog. Он используется в области искусственного интеллекта и компьютерной лингвистики.

Логика программы выражается в терминах отношений, представленных в виде фактов и правил. Для того чтобы инициировать вычисления, выполняется специальный запрос к базе знаний, на которые система логического программирования генерирует ответы только «истина» и «ложь».

При использовании языка Prolog в задачах, на которые он направлен, программист может получить большой выигрыш в скорости написания

программ и читаемости кода программы.

Рассмотрим пример программы на языке Prolog:

```
/* Факты */
parent(tom, bob). /*tom - родитель bob'a*/
parent(tom, ann).
parent(ann, alex).
parent(ann, mary).
parent(bob, liza).
parent(liza, kate).
parent(stiven, liza).

male(tom).
male(alex).
male(bob).
female(ann).
female(mary).
female(kate).
female(liza).

/*Правило для вывода цели*/
grandparent(X,Y):-parent(Z,X),parent(Y,Z).
```

И при запросе

```
grandparent(liza, X).
```

Результатом будет дедушка Лизы - Том. Потому что это можно вывести согласно правилу grandparent и фактам parent(bob,liza) и parent(tom,bob)

```
X = tom
```

3.2.5 Процедурное программирование

Процедурное программирование относится к императивному подходу, в котором последовательно выполняемые команды группируются в подпрограммы средствами самого языка. Идея такого подхода к программированию заключается в разбиении большой задачи на небольшие подзадачи, которые решаются шаг за шагом.

Более подробно идею процедурного подхода мы уже рассматривали в разделе 1.2

3.2.6 Функциональное программирование

3.2.6.1 Определение

Следующее, что мы рассмотрим – это функциональное программирование – полезный инструмент для решения таких задач как, например, фильтрация или генерация последовательностей значений в зависимости от определенного условия.

Начнем с определения. Функциональное программирование относится к декларативной парадигме. Также функциональное программирование является разделом дискретной математики. Но в то же время функциональное программирование рассматривают как самостоятельную парадигму, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании). По сравнению с процедурным подходом есть важное отличие – в функциональном программировании не предполагается явное хранение состояния программы.

О каких функциях идет речь в функциональном программировании? Во-первых, это функции высшего порядка. *Функции высшего порядка* – это такие функции, которые могут принимать на вход и/или возвращать другие функции.

Во-вторых, чистые функции. *Чистая функция* – это такая функция, которая зависит только от своих параметров и не взаимодействует с внешними данными. Определение чистой функции значит, что для одних и тех же данных гарантировано получится один и тот же результат. О чистых функциях также говорят, что функция детерминирована и не имеет побочных эффектов. В отношении чистых функций выделяют ряд характеристик, многие из которых можно использовать для оптимизации кода:

1. Если результат чистой функции не используется, вызов чистой функции может быть удален без вреда для других выражений.
2. Результат вызова чистой функции может быть мемоизирован, то есть сохранен в таблице значений вместе с аргументами вызова.

Мемоизация (*memoization*) – свойство функций сохранять (кешировать) результаты вычислений, чтобы не вычислять их впоследствии повторно.

Если в дальнейшем функция вызывается с этими же аргументами, её результат может быть взят прямо из таблицы, не вычисляясь повторно (иногда это называется принципом прозрачности ссылок). Мемоизация, ценой небольшого расхода памяти, позволяет существенно увеличить производительность и уменьшить порядок роста некоторых рекурсивных алгоритмов.

3. Если нет никакой зависимости по данным между двумя чистыми функциями, то порядок их вычисления можно поменять или распараллелить.

Функциональному подходу свойственны следующие особенности:

- Данные неизменяемые.
- Программа представляется в виде совокупность чистых функций.
- Отсутствие циклов.
- Использование функций высшего порядка.
- Функция может быть сохранена в переменную.
- Функция не зависит от имени, по которому мы к ней обращаемся.

Примеры, где используется функциональный подход, – многочисленны, вот некоторые из них: при создании мозаичного оконного менеджера, интерфейсов к базам данных, при разработке графических приложений, игр, при анализе и обработке текстов, генерации html-страниц, даже при разработке веб серверов. Функциональное программирование применяется при разработке языков программирования, компиляторов. Существуют функциональные языки программирования, например, LISP, Haskell, Scala и R.

Далее рассмотрим конкретные реализации функционального подхода на языке Python.

3.2.6.2 Про итератор и итерируемый объект

Термины: *итератор* (*iterator* или *iterator object*) и *итерируемый объект* (*iterable* или *iterable object*) будут часто встречаться при работе с различными парадигмами программирования, поэтому давайте рассмотрим их отдельно.

Итератор

Первое, что мы рассмотрим – это итератор. Что такое объект-итератор? Это специальный объект, который делает проще переходы по элементам

другого объекта. *Итератор* – это своего рода *перечислитель* для определенного объекта (например, списка, строки, словаря), который позволяет перейти к следующему элементу этого объекта, либо бросает исключение, если элементов больше нет.

Основное место в программе, где вы так или иначе сталкивались или уже использовали итераторы – это цикл for. Например, вы перебираете элементы в некотором списке с помощью цикла for:

```
>>> symbols = ['a', 'halo', ['h', 'o', 'w'], 'hello']
>>> for item in symbols:
...     print(item)
...
a
halo
['h', 'o', 'w']
hello
```

Видим, что при каждой итерации цикла происходит обращение к итератору списка, и итератор выдает следующий элемент. Если элементов в объекте больше нет, то генерируется исключение, обрабатываемое в рамках цикла for незаметно для пользователя.

Откуда берется итератор? Для встроенных типов данных, которые можно перебирать в цикле, наличие итератора предусмотрено самим языком. Можно посмотреть, как работает итератор. Для этого воспользуемся функцией *iter()* (эта функция вызывает метод итератора *__iter__()*)*:*

```
>>> sym_iter = iter(symbols)
>>> sym_iter
<list_iterator object at 0x7f45e0df04e0>
>>> type(sym_iter)
<class 'list_iterator'>
```

В строках выше мы получили итератор. Чтобы перейти к следующему элементу последовательности, надо вызвать функцию *next()*, куда передается объект-итератор. Функция *next()* позволяет извлечь следующий элемент из итератора:

```
>>> next(sym_iter)
'a'
>>> next(sym_iter)
```

```
'halo'
>>> next(sym_iter)
['h', 'o', 'w']
>>> next(sym_iter)
'hello'
>>> next(sym_iter)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration
```

Как мы видим, в случае, когда элементов больше нет, выбрасывается исключение *StopIteration*. Можно представить, что итератор – это специальная коробка с элементами, которая должна быть использована в цикле – на каждой итерации цикла из коробки извлекаются элементы и обрабатываются в теле цикла.

Встроенная функция *next()* вызывает метод *__next__()* у итератора, который продемонстрирован далее:

```
>>> sym_iter = iter(symbols)
>>> sym_iter.__next__()
'a'
>>> sym_iter.__next__()
'halo'
>>> sym_iter.__next__()
['h', 'o', 'w']
>>> sym_iter.__next__()
'hello'
>>> sym_iter.__next__()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration
```

Еще одна особенность использования итераторов – можно вызывать функцию *iter()* с итератором:

```
>>> iter(sym_iter)
<list_iterator object at 0x7fd68b0993c8>
```

Стоит отдельно обратить внимание, что вызов функции *iter()* всегда возвращает один и тот же объект-итератор:

```
>>> iter(sym_iter)
<list_iterator object at 0x7fd68b0993c8>
>>>
>>> iter(sym_iter)
```

```
<list_iterator object at 0x7fd68b0993c8>
```

Таким образом, получить данные из итератора можно только один раз (второй запуск такого цикла ничего не выведет на экран). Внутренний механизм цикла *for* сначала вызывает функцию *iter()* переданного объекта. Для итератора этот метод возвращает сам объект-итератор. После этого происходит обращение к методу *__next__()* до тех пор, пока не будет сгенерировано исключение *StopIteration*. По завершении цикла все элементы итератора будут исчерпаны (чтобы снова пройтись по итератору, надо создавать новый итератор).

Итерируемый объект

Итерируемый объект – объект, по которому можно итерироваться (то есть который можно обходить в цикле, например, цикле *for*). Чем же тогда итерируемый объект отличается от итератора?

Итератор – это надстройка над итерируемым объектом. Из итерируемого объекта всегда можно получить итератор с помощью встроенной функции *iter()*.

```
>>> symbols = ['a', 'halo', ['h', 'o', 'w'], 'hello']
>>> iter(symbols)
<list_iterator object at 0x7f5ab6690c18>
```

При каждом вызове функции *iter()* будет создан новый объект-итератор:

```
>>> iter(symbols)
<list_iterator object at 0x7fd68b099c50>
>>> iter(symbols)
<list_iterator object at 0x7fd68b099b00>
>>> iter(symbols)
<list_iterator object at 0x7fd68b0c8358>
```

Функцию *next()* нельзя вызывать с итерируемым объектом:

```
>>> next(symbols)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'list' object has no attribute '__next__'
```

На основе итератора можно создать итерируемый объект, например:

```
>>> sym_iter = iter(symbols)
```

```
>>> sym_iter
<list_iterator object at 0x7f5ab669e748>
>>> new_symbols = list(sym_iter)
>>> new_symbols
['a', 'halo', ['h', 'o', 'w'], 'hello']
```

И это, конечно, можно сделать только один раз, то есть повторная попытка создать другой итерируемый объект на основе того же итератора приведет к созданию пустого итерируемого объекта:

```
>>> new_symbols_1 = list(sym_iter)
>>> new_symbols_1
[]
```

В цикле *for* можно обходить как итераторы, так и итерируемые объекты. Внутренний механизм цикла *for* сначала вызывает метод *__iter__()* объекта. Если передан итерируемый объект, создается итератор для этого объекта. После этого на каждой итерации вызываться метод *__next__()* до тех пор, пока не будет возбуждено исключение *StopIteration*. Как помним, если обходить в цикле *for* итератор, то по завершении все элементы будут исчерпаны (второй раз по итератору не пройти, надо создавать новый). Для итерируемых объектов после цикла *for* все элементы продолжают быть доступны.

Примеры типов итерируемых объектов в Python – список, словарь, строка и другие коллекции (про коллекции будет подробнее в Главе 4), а объекты типа, возвращаемого функцией *range()*.

3.2.6.3 Функция map()

Функция *map()* принимает на вход два параметра: функцию и последовательность (итерируемый объект, *iterable*). Функция работает следующим образом: применяет к элементам итерируемого объекта (объектов) переданную функцию. Функция *map()* в Python возвращает объект-итератор типа *map*.

Рассмотрим синтаксис функции *map*:

map(<функция>, <объект_1> [, <объект_2>, … ,<объект_N-1>])

- *<функция>*: функция, которую следует применить к элементам

итерируемого объекта или объектов (или имя функции, которая должна быть применена).

- *<объект_1>*: итерируемый объект, к элементам которого требуется применить функцию.

Количество *N* итерируемых объектов определяется тем, сколько аргументов принимает *<функция>*. Можно передать несколько итерируемых аргументов в функцию *map()*, в этом случае указанная функция должна иметь столько же аргументов. Функция будет применяться к элементам этих итерируемых объектов. Работа функции *map()* будет завершена, когда итерируемый объект с наименьшей длиной будет обработан.

Давайте посмотрим, как работает функция *map()* на примерах.

Допустим, с помощью функции *input()* были прочитаны данные, в которых хранится список чисел, разделенных пробельными символами. Например, так:

```
>>> num_list = input().split()
>? 1 2 3 4 5 6 7 8 9
>>> num_list
['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Как видно из примера выше, изначально все эти числа имеют строковый тип данных. Допустим, нужно превратить каждый элемент списка в целое число. Можно поступить следующим образом:

```
>>> new_list = []
... for item in num_list:
...     new_list.append(int(item))
...
>>> new_list
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Но можно использовать функциональное программирование, применив функцию *map()*:

```
>>> new_list = map(int, num_list)
>>> new_list
<map object at 0x7fa0fff93b70>
```

Обратите внимание на тип возвращаемого результата:

```
>>> type(new_list)
<class 'map'>
```

Как уже говорилось выше, функция map() возвращает объект-итератор типа *map*. Обратите внимание, что после того, как к итератору произошло обращение, из него извлекаются элементы. Эту особенность можно

проиллюстрировать следующим примером:

```
>>> m = map(int, ['1', '2', '3'])
>>> print(m)
<map object at 0x7fa0fffc33c8>
>>> for i in m:
...     print(i)
...
1
2
3
>>> for i in m:
...     print(i)
...
```

Чтобы использовать результаты работы функции как обычные последовательности, например, обращаться по индексу, добавлять элементы и пр., можно обернуть вызов функции *map()* в функцию *list()*, например:

```
>>> new_list = list(map(int, num_list))
>>> new_list
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

В данном примере мы применили функцию *int()*, которая получает на вход один аргумент – строку, к элементам списка *num_list*. Функция *map()* взяла каждый элемент списка, передала в качестве аргумента в функцию *int()* и специальным образом сохранила результат.

Вспоминая, что строки – это тоже итерируемый объект, можно поступать следующим образом:

```
>>> list(map(int, '123456'))
[1, 2, 3, 4, 5, 6]
```

Функция *map()* взяла каждый элемент строки, передала в качестве аргумента в функцию *int()* и вернула результат в виде объекта-итератора, а мы уже на основе объекта-итератора создали список.

3.2.6.4 Функция map() с функциями пользователя

Функция *map()* также работает и с функциями, созданными пользователем. Например, есть функция, которая переводит часы в минуты (принимает на вход одно число – часы, и преобразует его в минуты, умножая на коэффициент 60):

```
>>> def hours_to_minutes(num_hour):
...     return num_hour * 60
```

Давайте определим список часов, например:

```
>>> hours_list = [1.0, 6.5, 7.4, 2.4, 9]
```

И применим к элементам этого списка определенную выше функцию *hours_to_minutes()* с помощью функции *map()*:

```
>>> minutes_list = list(map(hours_to_minutes, hours_list))
>>> minutes_list
[60.0, 390.0, 444.0, 144.0, 540.0]
```

В результате мы получаем преобразованный список, все элементы которого были умножены на 60.

Теперь посмотрим пример с несколькими итерируемыми объектами в функции. Функция *magic_sum()* принимает три аргумента и возвращает их сумму:

```
>>> def magic_sum(x, y, z):
...     return x + y + z
```

Пусть у нас есть три списка одинаковой длины, содержащие целочисленные значения:

```
>>> L_1 = [1, 2, 3, 4]
... L_2 = [10, 20, 30, 40]
... L_3 = [100, 200, 300, 400]
```

Нам необходимо получить сумму поэлементную сумму для данных списков, то есть другой список, где первый элемент – сумма первых элементов из определенных выше списков. То есть в результата хотим список следующего вида:

```
>>> S = [
...  L_1[0] + L_2[0] + L_3[0],
...  L_1[1] + L_2[1] + L_3[1],
...  L_1[2] + L_2[2] + L_3[2],
...  L_1[3] + L_2[3] + L_3[3]
... ]
>>> S
[111, 222, 333, 444]
```

Это легко сделать с помощью функции *map()* (сразу преобразуем к списку):

```
>>> list(map(magic_sum, L_1, L_2, L_3))
[111, 222, 333, 444]
```

Функция *magic_sum()* принимает три аргумента, поэтому мы передаем ей три списка для обработки. Что будет если списки будут разной длины?

```
>>> L1 = [1, 2, 3]
... L2 = [10, 20, 30, 40]
... L3 = [100, 200, 300, 400, 500]
...
```

```
>>> list(map(magic_sum, L1, L2, L3))
[111, 222, 333]
```

В примере выше получили в результате список, длина которого равно минимальной длине списков, переданных на вход функции *map()*.

Запомните, что количество итерируемых объектов в функции *map()* должно быть столько же, сколько аргументов в функции, которая будет применяться к этим объектам. Если передать недостаточное или избыточное количество объектов, можно столкнуться с ошибкой, например:

```
>>> list(map(magic_sum, L1, L2))
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: magic_sum() missing 1 required positional argument: 'z'
>>> list(map(magic_sum, L1, L2, L3, L3))
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: magic_sum() takes 3 positional arguments but 4 were given
```

3.2.6.5 lambda-выражения

Одна из самых популярных реализаций функционального программирования во многих языках программирования – это лямбда-выражения. Лямбда-выражения – это специальный элемент синтаксиса для создания анонимных (т.е. не имеющих имени) функций сразу в том месте, где эту функцию необходимо вызвать. Используя лямбда-выражения можно объявлять функции в любом месте кода, в том числе внутри других функций. Синтаксис определения следующий:

lambda аргумент1, аргумент2,..., аргументN : выражение

Давайте определим функцию деления при помощи лямбда-выражения:

```
>>> div = lambda x, y : x / y
>>> print(div(3, 2))
1.5
```

Лямбда-выражение может иметь неограниченное количество аргументов, однако производит только одно действие и чаще всего используется только в одном месте кода. В Python лямбда-выражения упрощают запись и использование однострочных операций.

Лямбда-выражения очень полезны при обработке списков, что мы далее и рассмотрим.

3.2.6.6 Функция map() и lambda-выражения

Мы уже знаем, что функция *map()* принимает два и более аргументов: функцию и итерируемые объекты. В качестве аргумента-функции также может быть использовано лямбда-выражение.

Пример программы, которая умножает на два каждый элемент списка:

```
>>> list(map(lambda x : x*2, [1, 2, 3, 4]))
[2, 4, 6, 8]
```

Ранее мы рассматривали пример, который переводит часы в минуты:

```
>>> def hours_to_minutes(num_hour):
...     return num_hour * 60
...
... hours_list = [1.0, 6.5, 7.4, 2.4, 9]
... minutes_list = list(map(hours_to_minutes, hours_list))
...
>>> minutes_list
[60.0, 390.0, 444.0, 144.0, 540]
```

При помощи лямбда-выражений указанный выше код может быть упрощен:

```
>>> hours_list = [1.0, 6.5, 7.4, 2.4, 9]
>>> minutes_list = list(map(lambda x: x*60, hours_list))
[60.0, 390.0, 444.0, 144.0, 540.0]
```

3.2.6.7 Функция filter()

На ряду с функцией *map()*, есть еще одна очень полезная реализация функционального программирования – функция *filter()*. Синтаксис функции:

filter(<функция>, <объект>)

Функция *<функция>* применяется для каждого элемента итерируемого объекта *<объект>* и возвращает *объект-итератор*, состоящий из тех элементов итерируемого объекта *<объект>*, для которых *<функция>* является истиной.

Напоминаем, что после того, как к итератору произошло обращение, из него извлекаются элементы. Следующий фрагмент кода иллюстрирует эту особенность:

```
>>> def check_num(num):
...     return num >= 0 and num % 3 == 0
>>> number_list = range(-10, 10)
>>> filtered = filter(check_num, number_list)
>>> for item in filtered:
...     print(item, end=' ')
... print()
```

```
0 3 6 9
```

После выполнения кода выше, где происходит обращение к итератору, и из него извлечены все элементы, и далее получить элементы невозможно. После повторного выполнения цикла на экране ничего не появится.

Чтобы воспользоваться результатами работы функции filter и после обращения к объекту-итератору (как и в случае с функцией *map()*), нужно обернуть вызов функции *filter()* в функцию *list()*, например:

```
>>> list(filter(check_num, number_list))
[0, 3, 6, 9]
```

Вспоминая, что строки – это тоже итерируемый объект, можно использовать функцию filter() и для фильтрации элементов в строках. Например, пусть есть функция, которая проверяет, что в переданной строке x хранится четно число:

```
>>> def check_str(x):
...     return int(x) % 2 == 0
...
>> number_str = '12345678'
>>> list(filter(check_str, number_str))
['2', '4', '4', '6', '8']
```

Еще больше интересных примеров можно найти в [16]-[18].

3.2.6.8 Функция filter() и lambda-выражения

Для функции *filter(<функция>, <объект>)* в качестве аргумента *<функция>* может быть передано *lambda*-выражение.

Принцип работы функции *filter* остается такой же: функция возвращает объект-итератор, состоящий из тех элементов итерируемого объекта *<объект>*, для которых *<функция>* является истиной. Как и в случае с обычной функцией в качества аргумента, *lambda*-выражение применяется для каждого элемента итерируемого объекта *<объект>*.

Помните, что после того, как к итератору произошло обращение в цикле, из него извлекаются элементы.

Давайте рассмотрим пример использования *filter* с *lambda*-выражениями. Например, выполнение такого кода:

```
>>> number_list = range(-10, 10)
>>> list(filter(lambda x: x >= 0 and x % 3 == 0, number_list))
[0, 3, 6, 9]
```

Другие примеры использования функции filter и лямбда выражений

можно найти в [19].

3.2.6.9 Функция zip()

Функция *zip(*iterables)* получает на вход несколько итерируемых объектов (чаще – списков) и возвращает объект-итератор (в Python 3, в более ранних версиях языка – *list*), состоящий из элементов-кортежей.

Первый элемент-кортеж формируется из первых элементов всех списков-аргументов, второй - из вторых элементов всех списков-аргументов и т.д.

Одно из возможных применений функции *zip* - для итерации по нескольким объектам в цикле:

```
>>> number_list = [1, 2, 3, 4, 5]
>>> str_list = ['one', 'two', 'three']
...
>>> string = 'ABCDEFG'
>>> for item in zip(number_list, str_list, string):
...     print(item)
...
(1, 'one', 'A')
(2, 'two', 'B')
(3, 'three', 'C')
>>> type(item)
<class 'tuple'>
```

При этом надо помнить, что длина возвращаемого функций zip объекта определяется минимальной длиной среди длин объектов-аргументов.

Чтобы использовать результаты работы функции *zip* несколько раз (не только в одном цикл), можно обернуть вызов функции *zip* в функцию *list()* (передать в функцию *list()* объект-итератор, возвращаемый функцией *zip()*). Об этом следующий пример:

```
>>> number_list = [1, 2, 3, 4, 5]
... str_list = ['one', 'two', 'three']
... string = 'ABCDEFG'
...
>>> zip_obj = zip(number_list, str_list, string)
>>> type(zip_obj)
<class 'zip'>
>>> list_obj = list(zip_obj)
>>> type(list_obj)
<class 'list'>
>>> for item in list_obj:
```

```
...     print(item)
...
(1, 'one', 'A')
(2, 'two', 'B')
(3, 'three', 'C')
>>> type(item)
<class 'tuple'>
```

Полезные ссылки, в которых можно найти еще больше примеров: [20], [21].

3.2.6.10 Функция zip() и словари dict

Очень полезное применение функции *zip* – создание словарей *dict*. Рассмотрим примеры. Допустим, есть два списка: *key_list* - условный список ключей, *values_list* - условный список значений. Выполнение следующего фрагмента кода

```
>>> key_list = [0, 1, 2, 3, 4, 5]
>>> values_list = ['Ann', 'Jho', 'Andrew', 'Bob', 'Sara', 'Jack']
>>> d = dict(zip(key_list, values_list))
>>> d
{0: 'Ann', 1: 'Jho', 2: 'Andrew', 3: 'Bob', 4: 'Sara', 5: 'Jack'}
```

Посмотрим, что будет, если в *key_list* есть повторяющиеся элементы:

```
>>> key_list = [0, 0, 2, 3, 5, 5]
>>> values_list = ['Ann', 'Jho', 'Andrew', 'Bob', 'Sara', 'Jack']
>>> d = dict(zip(key_list, values_list))
>>> d
{0: 'Jho', 2: 'Andrew', 3: 'Bob', 5: 'Jack'}
```

Произошло обновление значения *'Ann'* на *'Jho'* по ключу 0 и значения *'Sara'* на *'Jack'* по ключу 5.

Если в качестве ключей требуются, например, индексы элементов, то можно использовать функцию *range*, например:

```
>>> key_list = range(0, 6)
>>> values_list = ['Ann', 'Jho', 'Andrew', 'Bob', 'Sara', 'Jack']
...
>>> d = dict(zip(key_list, values_list))
>>> d
{0: 'Ann', 1: 'Jho', 2: 'Andrew', 3: 'Bob', 4: 'Sara', 5: 'Jack'}
```

Можно использовать функцию *range* с шагом, например, следующий фрагмент кода:

```
 >>> key_list = range(0, 6, 2)
>>> values_list = ('Ann', 'Jho', 'Andrew', 'Bob', 'Sara', 'Jack')
```

```
>>> d = dict(zip(key_list, values_list))
>>> d
{0: 'Ann', 2: 'Jho', 4: 'Andrew'}
```

3.2.7 Парадигмы в языках программирования

Многие языки программирования поддерживают одновременно несколько парадигм. К императивной парадигме программирования относят такие языки как Pascal, Python, C, C++, Java. Все они реализуют еще и концепцию процедурного программирования.

В отношении декларативного подхода можно выделить такие языки как SQL, Prolog, причем последний является еще и языком логического программирования.

Среди языков функционального программирования можно выделить LISP, F#, Haskel.

3.3 Объектно-ориентированное программирование

3.3.1 Основные понятия. Класс, объект, поля, методы

Следующая парадигма, которую мы бы хотели осветить подробно - это Объектно-Ориентированная Парадигма.

Человек мыслит объектами. Совершая любое действие в своей жизни, человек мысленно оперирует огромным множеством объектов: чашка, книга, дерево, ботинки и т.д. Для написания программы нам также может потребоваться мыслить в терминах объектов.

Представим, что мы разрабатываем программу для работы с различными геометрическими фигурами. Для одних фигур (например, эллипс, квадрат, шестиугольник) нам надо узнать площадь, для других (например, цилиндр, куб) — объем. Мы хотим хранить цвет и координаты каждой фигуры, а еще у нас в наличии не один цилиндр, а пять.

Такие данные, как и функции для их обработки, можно описать в процедурном стиле. Но такая программа может оказаться сложной для

восприятия, большое количество кода будет дублироваться. Для подобных программ лучше всего использовать Объектно-Ориентированную Парадигму (ООП). Тогда у нас будет свой класс для каждой фигуры, в каждом классе будет храниться описание данного типа фигур и действия, которые можно с ней сделать (например, вычисление объема).

3.3.1.1 ООП в Python

Python является объектно-ориентированным языком, но при этом на Python вы можете писать программы в процедурном стиле. Тем не менее, всё, с чем вы сталкивались, даже используя процедурный стиль, является объектом: модули, функции, списки, строки и т.д. Любая программа на языке Python представляет собой совокупность объектов.

Как уже говорилось ранее, объект - конкретная сущность предметной области, тогда как класс - это тип объекта. Примерами могут служить класс “Планета” и объекты: Меркурий, Венера, Земля, Марс; класс “Целые числа” и объекты 2, 4, 10, класс “Функции” и объекты int(), type().

Классы содержат атрибуты, которые подразделяются на *поля* и *методы*.

Под *методом* понимают функцию, которая определена внутри класса. Например, в классе str определены методы split(), join(), title(). Есть два способа вызвать метод в языке Python:

1. <объект>.<название_метода>(<аргумент_1>, … <аргумент_n>)

   Пример:

```
>>> st = 'Qw-Er-Ty' # создание объекта класса str
>>> st.split('-') # вызов метода split() с аргументом ‘-’
['Qw', 'Er', 'Ty']
```

2. <имя_класса>.<название_метода>(<объект>, <аргумент_1>, … <аргумент_n>)

```
>>> st = 'Qw-Er-Ty' # создание объекта класса str
>>> str.split(st, '-') # вызов метода split() с аргументом ‘-’ для объекта st
['Qw', 'Er', 'Ty']
```

Поле - это переменная, которая определена внутри класса. Узнать содержимое поля в языке Python можно используя следующий синтаксис:

1. <объект>.<название_поля>

Пример:
```
>>> a = 2+5j  # создание объекта класса complex
>>> a.imag  # вывод значения поля imag
5.0
```

3.3.1.2 Вызов конструктора

Конструктор - это специальный метод, который нужен для создания объектов класса. Мы не раз вызывали конструктор, когда осуществляли создание объекта или приводили тип:
```
>>> str(25) # вызов конструктора для создания строкового объекта
'25'
>>> list('QwErTy') # вызов конструктора для создания объекта списка
['Q', 'w', 'E', 'r', 'T', 'y']
```

3.3.1.3 Создание класса

До сих пор мы ООП в Python использовались только встроенными объектами языка Python, в этом разделе мы начнем создавать свои классы и объекты.

Синтаксис создания класса:
```
class <Название_класса>:
     <Тело_класса>
```

Создадим класс *Mammal* с полем *age*:
```
>>> class Mammal:
...     age = 0
```

Обратите внимание, что мы определили *поле класса*. Это означает, что для всех экземпляров класса *Mammal* поле *age* будет одинаковым. Мы еще вернемся к подробному обсуждению этого вопроса.

В примере выше мы определили класс и не определили конструктор. Тем не менее, мы можем создать объект класса *Mammal*, поскольку в Python все классы наделены конструктором по умолчанию:
```
>>> Mammal()
<__main__.Mammal object at 0x7efda8122358>
```
Мы создали объект класса *Mammal*, однако мы никак не воспользовались теми возможностями, которые есть в ООП, например, не

определили поля объекта. Чтобы инициализировать поля объекта при его создании, мы можем создать конструктор. Синтаксис конструктора:

```
def __init__(self, <аргумент_1>, …, <аргумент_n> ):
    <Тело_конструктора>
```

Обратите внимание на несколько важных моментов:

1. Конструктор в языке Python всегда имеет название *__init__()*.
2. При создании объекта вызывается конструктор.
3. Чтобы вызвать конструктор, мы используем название класса.
4. Конструктор - это метод, который ничего не возвращает.
5. Первый аргумент любого метода класса в языке Python - экземпляр класса (т.е. объект), для которого этот метод вызывается. Обычно он имеет название *self*; в других языках часто используется ключевое слово *this*.
6. В теле конструктора обычно происходит инициализация различных полей класса через обращение к экземпляру *self*.
7. Все методы имеют доступ к полям объекта.
8. В Python нельзя создать два конструктора с разным количеством аргументов, как, например, в языке C++, однако, используя механизм аргументов по умолчанию, можно добиться аналогичного поведения.

Давайте изменим класс *Mammal* (см. листинг 3.1), добавим в него конструктор и два метода.

Листинг 3.1 – Добавление конструктора в класс

```
class Mammal:
    age = 0
    def __init__(self, name):
        '''Конструктор класса Mammal.
        self - объект, для создания которого
               был вызван конструктор'''
        self.name = name  # поле объекта класса Mammal

    def sleep(self):
        print('Zzzzzzz')

    def say_hello(self, friend_name):
        print('Hello, {}! I am {}'.format(friend_name, self.name))
```

Теперь давайте создадим экземпляр класса Mammal:

```
>>> mammal = Mammal('Fedor')
```

Теперь мы можем вызвать методы класса Mammal:

```
>>> mammal.sleep()
Zzzzzzz
```

Обратите внимание, что в определении метода *sleep()* мы указали объект, с которым мы работаем (*self*), но при вызове объект, как аргумент метода, не указывается.

Вызовем второй метод:

```
>>> mammal.say_hello('Sam')
Hello, Sam! I am Fedor.
```

Методам класса доступны поля объектов через переменную *self*. Если мы создадим новый объект со значением поля *name*, результат вызова *say_hello()* будет другим.

3.3.1.4 Поля класса

В прошлом подразделе мы работали с полями *объекта*, теперь давайте подробно разберем работу с полями *класса*.

В классе *Mammal* у нас есть поле age (см. листинг 3.2):

Листинг 3.2 - Определение класса Mammal

```
class Mammal:
     age = 0
```

Каким образом мы можем получить доступ к полю класса? Сделать это, как в случае с методами, можно двумя способами:

```
>>> Mammal.age  # используя имя класса
0
>>> mammal = Mammal()
>>> mammal.age # используя объект класса
0
```

Обратите внимание, что в первом случае мы не создавали объект класса, однако всё равно смогли получить доступ к полю *age*.

В общем случае поле *age* будет одинаковым для всех экземпляров класса. Мы можем изменить его для всех экземпляров класса, если изменим его путем присваивания нового значения *Mammal.age*:

```
>>> class Mammal: # описание класса Mammal с полем класса age
```

```
...     age = 0
...
>>> a = Mammal() # экземпляр класса Mammal
>>> b = Mammal() # экземпляр класса Mammal
>>> a.age
0
>>> Mammal.age = 7 # изменение поля age для всех экземпляров класса
>>> a.age
7
>>> b.age
7
>>> Mammal.age
7
>>> c = Mammal()
>>> c.age
7
```

Мы можем поменять значение этого поля для конкретного экземпляра, при этом его значение для других объектов (в том числе новых) не изменится:

```
>>> a.age = 1
>>> a.age
1
>>> Mammal.age
0
>>> d = Mammal()
>>> d.age
0
```

Такое поведение характерно для неизменяемых объектов. Если мы создадим изменяемое поле класса, например, список, мы сможем изменить именно объект поля, а не ссылку. Например:

```
>>> class Mammal:
...     avaliable_names = ['Fedor', 'Sam', 'Christopher']
...
>>> a = Mammal()
>>> b = Mammal()
>>> Mammal.avaliable_names
['Fedor', 'Sam', 'Christopher']
>>> a.avaliable_names.append('Sigmund')
>>> a.avaliable_names
['Fedor', 'Sam', 'Christopher', 'Sigmund']
>>> b.avaliable_names
['Fedor', 'Sam', 'Christopher', 'Sigmund']
>>> Mammal.avaliable_names
['Fedor', 'Sam', 'Christopher', 'Sigmund']
```

При этом мы также можем изменить поле для конкретного объекта путем присваивания:

```
>>> a.avaliable_names = []
>>> a.avaliable_names
```

```
[]
>>> b.avaliable_names
['Fedor', 'Sam', 'Christopher']
>>> Mammal.avaliable_names
['Fedor', 'Sam', 'Christopher']
```

Для остальных объектов поле останется неизменным.

3.3.2 Наследование как часть парадигмы

Объектно-ориентированная парадигма базируется на нескольких принципах: наследование, инкапсуляция, полиморфизм.

Наследование - специальный механизм, при котором мы можем расширять классы, усложняя их функциональность.

В наследовании могут участвовать минимум два класса: *суперкласс* (или *класс-родитель, или базовый класс*) - это такой класс, который был расширен. Все расширения, дополнения и усложнения класса-родителя реализованы в *классе-наследнике* (или *производном* классе, или *классе-потомке*) - это второй участник механизма наследования. Схематично наследование представлено на рисунке 3.1.

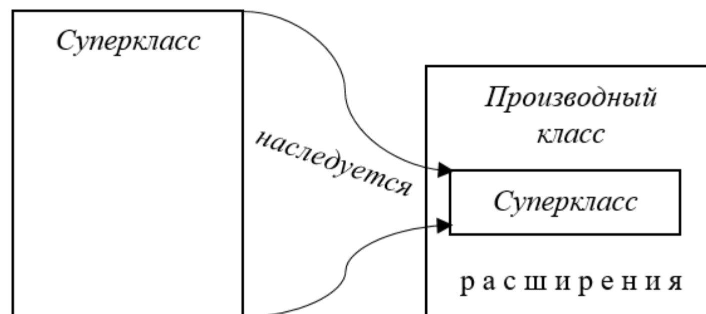


Рисунок 3.1 - Схематичное представление наследования

Наследование позволяет повторно использовать функциональность базового класса, при этом не меняя базовый класс, а также расширять ее, добавляя новые атрибуты.

Давайте создадим базовый класс Mammal:

```
>>> class Mammal:
...       def __init__(self, name, age):
...           self.name = name
...           self.age = age
...       def sleep(self):
...           print('Zzzzzzz')
```

Mammal - это класс, который описывает некоторое млекопитающее, у которого есть имя и есть возраст (поля *name* и *age*) и который иногда спит (метод *sleep()*). Создадим класс *Dog*, который будет обладать теми же

атрибутами, а также будет иметь свои:

```
>>> class Dog(Mammal):
...     def __init__(self, name, age, breed):
...         self.name = name
...         self.age = age
...         self.breed = breed
...         self.is_angry = False
...     def sleep(self):
...         print('Zzzzzzz')
...         self.is_angry = False
```

В классе *Dog* мы повторяем код класса *Mammal*. Давайте перепишем класс *Dog* таким образом, чтобы отсутствовало повторение: с помощью механизма наследования.

3.3.2.1 Наследование в Python

Синтаксис:

```
class A: # класс-родитель
    pass

class B(A): # класс-потомок
    pass
```

Класс-родитель указывается в скобках при определении класса-потомка; классов-родителей может быть сколько угодно. С возможностью наследования от нескольких родителей связаны некоторые возможные ошибки, однако в рамках данного пособия мы будем наследоваться исключительно от одного класса.

Иногда в процессе написания метода в классе-наследнике может понадобиться вызвать метод суперкласса. Это можно сделать через имя суперкласса или через функцию *super()*:

```
class B(A):
    def method(self, arg):
        # То же самое, что super(B, self).method(arg)
        super().method(arg)
```

Итак, перепишем класс Dog, используя наследование:

```
>>> class Dog(Mammal):
...     def __init__(self, name, age, breed):
...         super().__init__(name, age)
...         self.breed = breed
...         self.is_angry = False
...     def sleep(self):
```

```
...          super().sleep()
...          self.is_angry = False
```

Нам не пришлось писать код класса Mammal как это было сделано до использования наследования.

3.3.2.2 isinstance() и issubclass()

В языке Python есть две полезные функции, которые помогают понять связь объектов/классов с суперклассами.

1. Первая функция:

isinstance(obj_, class_)

Функция возвращает True, если obj_ является экземпляром класса class_ или если class_ является суперклассом для класса, объектом которого является obj_. При этом class_ может являться суперклассом суперкласса (и т.д.). Проиллюстрируем на примере:

```
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
>>> class C(B):
...     pass
...
>>> c = C()
>>> isinstance(c, A)
True
```

Функция isinstance() похожа на функцию type(), однако также учитывает классы-родители.

2. Вторая функция:

issubclass(class1, class2)

Функция возвращает True, если class1 является наследником класса class2 (или наследником наследника, с любым уровнем вложенности). Класс считается наследником самого себя. Рассмотрим ряд примеров:

```
>>> issubclass(B, A)
True
>>> issubclass(C, B)
True
>>> issubclass(C, A)
```

```
True
>>> issubclass(A, A)
True
>>> issubclass(A, B)
False
```

3.3.3 Инкапсуляция

Под инкапсуляцией часто понимают сокрытие внутренней реализации от пользователя. В других языках программирования это достигается использованием модификаторов доступа; таким образом, в описании класса мы можем указать, какой атрибут будет доступен извне, а какой нет.

В языке Python этот механизм лишь указывает, что атрибут не должен быть изменен. Рассмотрим на примере:

```
>>> class A:
...     def __init__(self):
...         self.__private_field = [1, 2, 3]
...
```

Если вы указываете атрибут с двумя нижними подчеркиваниями в начале имени (при этом без них в конце), вы сообщаете интерпретатору и пользователям вашего класса, что доступ к этому атрибуту ограниченный:

```
>>> a = A()
>>> a.__private_field
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute '__private_field'
```

Однако, в отличие от многих языков программирования, доступ к таким атрибутам в Python получить всё же можно. Воспользуемся функцией *dir()*, которая возвращает список атрибутов объекта (а также всех базовых классов):

```
>>> dir(a)
['_A__private_field', '__class__', '__delattr__', '__dict__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

Мы видим в списке атрибут, похожий по названию на созданный нами. Попробуем его вывести:

```
>>> a._A__private_field
[1, 2, 3]
```

Механизма, строго запрещающего доступ к атрибутам в языке Python, нет, поэтому может показаться, что в языке отсутствует инкапсуляция. Однако это не так: под инкапсуляцией понимают также сокрытие деталей реализации за интерфейсом объекта. Это означает, что когда вы используете какой-либо метод класса, например, метод *append()* класса *list*, вам неважно как он реализован. Это позволяет менять реализацию такого метода, при этом не влияя на его использование.

3.3.4 Полиморфизм

В некоторых языках существует возможность создать несколько функций с одинаковым именем, но разными типами аргументов. Это называется перегрузкой функций. В языке Python мы не можем воспользоваться таким механизмом, поскольку, во-первых, в языке нет объявления типа, а во-вторых, нельзя создать функцию с тем же именем в той же области видимости, например, в модуле: как и в случае с инициализацией переменной, сохранится только последнее определение функции, первое при этом будет перезаписано.

Однако, это не означает, что в Python нет полиморфизма, скорее наоборот, вы встречаетесь с ним на каждом шагу.

Например, напишем функцию, которая складывает два переданных аргумента и выводит результат сложения:

```
>>> def magic_sum(a, b):
...     print(a + b)
...
```

Мы можем передать в такую функцию числа любого типа, строки, а также списки и кортежи. Таким образом, наша функция magic_sum() может работать с разными типами данных, если они поддерживают операцию сложения. Такое свойство функции говорит о том, что она *полиморфна*.

Когда говорят о полиморфизме в контексте ООП, обычно говорят о переопределении методов.

Например, вы создаете класс-наследник от класса *list*, чтобы получился новый список, с небольшим отличием: вы хотите, чтобы метод *append()* добавлял только целые числа. Для этого вы создаете класс-наследник класса list и *переопределяете* метод *append()*:

```
>>> class IntList(list):
...    def append(self, element):
```

```
...       # проверка типа добавляемого элемента
...       if type(element) == int:
...         super().append(element) # вызов метода супер-класса
...
>>> numbers = IntList()
>>> numbers.append(4)
>>> numbers.append(5)
>>> numbers.append('19')
>>> numbers
[4, 5]
```

Мы переопределили метод *append()* и теперь для объектов класса *IntList* будет выполняться именно переопределенный метод. При этом все методы базового класса, даже если мы их не переопределили, по-прежнему нам доступны:

```
>>>numbers.pop(0)
4
```

В Python существует возможность переопределения не только методов класса, но и встроенных операций/операторов выражений. Это означает, что любые действия, которые вы привыкли делать с объектами, вы можете определить в вашем классе: сложение, вывод на экран, сравнение и т.д. За это отвечают методы класса со специальными именами: такое имя начинается и заканчивается двумя нижними подчеркиваниями.

Рассмотрим методы вывода на экран __str__ и __repr__.

Как вы уже знаете, Python предоставляет возможность работы с интерпретатором в интерактивном режиме. Если в интерактивном режиме вы определите метод __repr__ в вашем классе, он будет вызываться каждый раз при выводе объекта или преобразования его в строку.

```
>>> class A:
...     def __repr__(self):
...         return "I'm an object"
...
>>> a = A()
>>> a
I'm an object
>>> str(a)
"I'm object"
```

Если бы метод __repr__ не был определен, вывод объекта выглядел бы примерно так:

```
>>>A()
<__main__.A object at 0x7ff123dcd438>
```

У метода __repr__ есть родственный метод __str__. В интерактивном режиме он не вызывается для вывода объекта, но используется для преобразования к строке:

```
>>> class A:
...     def __str__(self):
...         return '__str__'
...
>>> A()
<__main__.A object at 0x7ff123dcd438>
>>> str(A())
'__str__'
```

При этом, если определены оба этих метода, __repr__ будет использоваться для вывода, а __str__ - для преобразования к строке:

```
>>> class A:
...     def __str__(self):
...         return '__str__'
...     def __repr__(self):
...         return '__repr__'
...
>>> A()
__repr__
>>> str(A())
'__str__'
```

Вышеописанный механизм применим только к интерактивному режиму, для кода, описанного в файле, он работает иначе. Функция *print()* использует __str__, если этот метод определен в классе, иначе использует __repr__. Рассмотрим на примере листинга 3.3.

Листинг 3.3 – Пример работы методов __str__ и __repr__

```
class A:
    def __str__(self):
        return '__str__'

    def __repr__(self):
        return '__repr__'


print(A()) # Будет выведено __str__
print(str(A()) # Будет выведено __str__
```

Данные методы всегда возвращают строку и попытка вернуть объект другого типа приведет к ошибке.

3.3.5 Исключения

До этого момента мы уже часто использовали термины исключения или исключительные ситуации. Исключения нужны для обработки возможных ошибок и особых ситуаций. Давайте подробно рассмотрим что это такое.

3.3.5.1 Характеристики и определение исключений

Исключения - это специальный класс объектов в языке Python. Исключения предназначены для управления поведением программой, когда возникает ошибка, или, другими словами, для управления теми участками программного кода, где может возникнуть ошибка. О каких ошибках идет речь? Например, ошибка выхода за границы последовательности, ошибка при вызове несуществующего метода у определенного объекта и др., а также все те, что сложно программисту предусмотреть во время написания кода.

Начинающий программист на Python часто сталкивается с синтаксическими ошибками. Синтаксические ошибки отлавливаются на этапе *компиляции* программы на языке Python. О наличии ошибки в коде вы узнаете по специальному сообщению компилятора (примеры будут чуть позднее). Но вы можете узнать о наличии синтаксических ошибок заранее, с помощью IDE. В IDE во время написания кода синтаксические ошибки подчеркиваются (подсвечиваются средствами IDE), тем самым обращая на себя внимание программиста в попытке предотвратить их появление во время выполнения программы.

Исключения могут возникать во время *выполнения* программы. В языке Python исключения могут возбуждаться (генерироваться) автоматически, когда в программном коде допущены ошибки, а также могут возбуждаться и перехватываться (отлавливаться) самим программным кодом, который пишет программист, то есть исключениями в Python можно управлять.

Как реализованы исключения? Исключения в Python реализованы на основе ООП-парадигмы с использованием наследования и других принципов. Любое исключение – это объект. У объекта-исключения есть определенный тип, то есть, как мы уже знаем, определенный класс. Классы исключений выстроены в специальную иерархию. Есть основной класс *BaseException* - базовое исключение, от которого берут начало все остальные. Берут начало – в контексте ООП-парадигмы – наследуются. От *BaseException* наследуются *системные* и *обычные* исключения. Системными

исключениями являются: *SystemExit*, *GeneratorExit* и *KeyboardInterrupt*. У этих исключений нет встроенных наследников; вмешиваться в работу системных исключений не рекомендуется.

Вторая группа наследников класса *BaseException* – это обычные исключения – класс *Exception*. Встроенные наследники класса *Exception* вам уже хорошо знакомы: *AttributeError*, *SyntaxError*, *TypeError*, *ValueError* и многие другие. При реализации своих собственных исключений рекомендуют наследоваться как раз от класса *Exception*, то есть использовать класс *Exception* в качестве класса-родителя.

Рассмотрим ряд простых примеров. Пример синтаксической ошибки:

```
>>> data = '1'
...    print(data)
...
  File "<input>", line 2
    print(data)
    ^
IndentationError: unexpected indent
```

Исключение *IndentationError* наследуется от класса *SyntaxError*, который, свою очередь, является наследником *Exception*.

Пример возникновения исключительной ситуации *ValueError*:

```
>>> data = '1q'
>>> int(data)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1q'
```

Исключения, такие как *ValueError*, появляются уже в процессе выполнения кода, когда типы и значения обрабатываемых переменных уже известны. Во время выбрасывания исключения создается специальный объект, например, в случае выше это объект *ValueError*. Встроенные в Python типы исключений можно использовать при возбуждении (генерации, выбрасывании) исключительной ситуации (что будет подробнее обсуждаться позже), для этого нужно создать экземпляр такого исключения, например:

```
>>> v_err = ValueError("это ошибка")
>>> v_err
ValueError('это ошибка',)
>>> type(v_err)
<class 'ValueError'>
```

У объектов встроенных классов исключений есть свои встроенные поля и методы, например, поле *__class__* и метод *__hash__()*:

```
>>> v_err.__class__
<class 'ValueError'>
>>> v_err.__hash__()
8784964547269
```

Методы и поля данного класса унаследованы от класса-родителя *Exception*. Класс *Exception* является, в свою очередь, наследником другого класса – *BaseException*, для работы с которым также определен ряд встроенных методов и полей.

Таким образом, у любого исключения в языке Python есть:

- Тип (класс), например: TypeError, ValueError, …

```
>>> type(v_err)
<class 'ValueError'>
```

- Сообщение, которое содержит описание исключительной ситуации, например:

```
"invalid literal for int() with base 10: '1q'"
```

- Состояние стека вызовов функций на момент возникновения ошибки. Это позволяет уточнить место возникновения исключительной ситуации. Что такое стек вызовов, рассмотрим далее.

Стек вызовов

Стек вызовов — специальное место, куда интерпретатор размещает имена функций для их вызова. Над функцией интерпретатор размещает аргументы этой функции. Когда вызванная функция закончила выполнение, интерпретатор снимает функцию со стека.

В Python содержимое стека всегда начинается с функции *module*. Эта функция размещается интерпретатором в стеке как раз в тот момент, когда мы запускаем интерпретатор. Функция *module* выполняет запросы программиста, например, при вводе инструкций построчно. Функция *module* создает объекты всех других функций, переменных и т.д., которые мы определяем в своем коде.

Рассмотрим пример, представленный в листинге 3.4.

Листинг 3.4 – Файл call_stack.py

```
def func(arr):
```

```
    return arr[0] + arr[5]

def main():
    arr = list('12345')
    print(func(arr))

main()
```

После запуска файла в командной строке:

```
python3 call_stack.py
```

Получаем сообщение об исключительной ситуации, а именно:

```
Traceback (most recent call last):
  File "call_stack.py", line 8, in <module>
    main()
  File "call_stack.py", line 6, in main
    print(func(arr))
  File "call_stack.py", line 2, in func
    return arr[0] + arr[5]
IndexError: list index out of range
```

Сообщение состоит из содержимого стека вызовов *Traceback* и имени (с дополнительными данными) исключения. В содержимом стека перечислены все строки, которые были активны в момент появления исключения, в порядке от более старых к более новым. Из представленной информации видно, что ошибка произошла во 2-й строке в файле *call_stack.py* в инструкции *return*.

Трассировочная информация – это объект, который представляет стек вызовов в точке, где возникло исключение. В документации языка Python к модулю *traceback* описываются инструменты, которые могут использоваться вместе с этим объектом для создания сообщений об ошибках вручную. С помощью встроенной функции *print_exc*, расположенной в стандартном модуле *traceback* (за дополнительной информацией обращайтесь к руководству по библиотеке языка Python) можно самостоятельно выводить стек вызовов, что будет продемонстрировано в следующем разделе.

3.3.5.2 try+except [as] [else] [finally]

Программист может как создавать экземпляры классов исключений и работать с ними, так и обрабатывать их в коде – отлавливать, то есть предупреждать ситуацию, когда может возникнуть исключительная ситуация. Конструкция *try-except* нужна для того, чтобы перехватить и

обработать исключительные ситуации. Она имеет следующий синтаксис:

try:

<*Инструкция*>

except <*Тип_Исключения*>*:*

<*Обработка_Исключения*>

В блок try помещаются инструкции при выполнении которых может возникнуть исключение. После ключевого слова except указывается тип отлавливаемого исключения. Таких типов может быть несколько и тогда они указываются в виде кортежа:

except (RuntimeError, TypeError, NameError):

Если конкретный тип исключения не указан, этим except блоком будут пойманы будут все исключения. Блоков Except также может быть несколько.

Если в блоке *try* исключения не возникает, то тело блока except не выполняется.

В предыдущем разделе, когда рассматривали стек вызовов, сказали, что с с помощью модуля *traceback* и встроенной функции print_exc, можно самостоятельно выводить стек вызовов. Давайте рассмотрим пример:

```
>>> import traceback
>>> def safe(entry, *args):
...     try:
...         entry(*args)
...         # Перехватывать любые исключения
...     except:
...         print('You are here!')
...         traceback.print_exc()
...
>>> def f(a, b, c):
...     return a + b + c
...
>>>
>>> safe(f, 12, 15, 'F')
You are here!
Traceback (most recent call last):
  File "<input>", line 3, in safe
  File "<input>", line 2, in f
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Также помимо блока except, могут быть два необязательных блока *else* и

finally.

```
try:
    <Инструкция>
except <Тип_Исключения>:
    <Обработка_Исключения>
else:
    # код для обработки случая,
    # когда в try-блоке не было поймано исключение <Тип_Исключения>
finally:
    # код, который нужно выполнить при любом исходе
```

Код в блоке else, выполняется тогда и только тогда, если в try-блоке не случилось исключительной ситуации (else-блок выполняется в случае, если утверждение "Исключительная ситуация произошла" ложно). Блок else может быть только один.

Код в блоке finally выполняется в любом случае, вне зависимости от того, произошла исключительная ситуация или нет. Блок finally также может быть только один.

3.3.5.3 Иерархия исключений

При написании нескольких except-блоков нужно понимать иерархию, в которую в Python выстроены исключения. Нужно иметь ввиду эту иерархию при выстраивании последовательности except-блоков: в какой except-блок попадем первым, а в какой, может быть, не попадем вообще.

Базовый класс для всех исключений - BaseException, но программистам рекомендуется использовать Exception для создания собственных исключений.

Поскольку при обнаружении исключительной ситуации в try-except используется функция isinstance(), мы можем отлавливать как объект указанного исключения, так и объект классов-наследников этого исключения, никогда не имеет смысла проверять сначала базовый потом наследника

При обработке исключительной ситуации мы попадем в except-блок, если был такой предусмотрен для конкретного типа исключения; или в

"общий" - без указания типа исключения. Другие строчки кода, после той, где случилось исключение, не будут выполнены. Соответственно, другие возможные исключительные ситуации не будут реализованы, и в те except-блоки, которые были для них предусмотрены, мы никогда не попадем.

В конструкции try-except помимо самого факта возникновения исключения мы также можем поймать объект, содержащий информацию об этом исключении. Для этого используется следующий синтаксис:

try:

    <Инструкция>

except <Тип_Исключения> as <Имя_Объекта_Исключения>:

    <Обработка_Объекта_Исключения>

Например:

```
>>> try:
...     a = int('1d2')
... except ValueError as e:
...     print(e)
...     print(e.args[0])
...
```

Тогда значение на экран будет выведено:

```
invalid literal for int() with base 10: '1d2'
invalid literal for int() with base 10: '1d2'
```

где *e.args[0]*, очевидно, равен строке *'invalid literal for int() with base 10: \\'1d2\\''.*

3.3.5.4 Инструкция raise

Мы можем самостоятельно сгенерировать исключение с помощью инструкции *raise*.

Синтаксис:

raise <Создание объекта исключения>

Пример:

```
>>> raise TypeError('Тип переменной указан неверно!')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: Тип переменной указан неверно!
```

Инструкция *raise* генерирует исключение, которое должно быть объектом класса, являющегося наследником класса Exception. Обычно

наследники класса *Exception* имеют суффикс **Error*.

С помощью raise можно выбросить и несколько исключений одновременно, например:

```
>>> try:
...     raise Exception(
...         [Exception('bad'),
...          Exception('really bad'),
...          Exception('really really bad')]
...     )
... except Exception as e:
...     print(e)
...     print(type(e))
...     for exp in e.args[0]:
...         print(exp)
...
```

И получим следующий вывод:

```
[Exception('bad',), Exception('really bad',), Exception('really really bad',)]
<class 'Exception'>
bad
really bad
really really bad
```

3.4 Упражнения и вопросы для самоконтроля

Парадигмы программирования

1. Чем обусловлено существование мультипарадигменных языков программирования?
2. В чем отличие между императивной и декларативной парадигмой?
3. Приведите пример декларативных языков программирования
4. К какой парадигме относят процедурное программирование? К какой относят логическое?

Функциональная парадигма программирования

1. Дайте определение итератору и итерируемому объекту. Опишите основные отличия.
2. Что будет выведено на экран в случае:

   ```
   >>> map(int, ['12', '-3', '100'])
   ```

3. Какой результат выполнения программы:

```
>>> list(filter(lambda x: len(str(x)) == 2 and int(x) % 3 == 0, [30, 24, 18, 36]))
```

4. Напишите программу, которая создает словарь на основе двух последовательностей: ['Anna', 'Nikol', 'Sasha'] и [1994, 1993, 1995] и использованием функции *zip*.
5. Что будет в результаты выполнения фрагмента кода:

```
>>> d = {}
... for key, value in zip('hello', range(6)):
...     if not key in d:
...         d[key] = value
```

6. Что будет в результате выполнения следующего кода:

```
>>> d = {1: 'Anna', 2: 'Jho', 3:'Nikita'}
>>> list(filter(lambda x: len(x) % 2 == 0, d.values()))
```

7. Что будет выведено на экран будет в результате выполнения следующего кода:

```
>>> def process(item1, item2):
...     return item1.find(item2) + 12
...
>>> in_1 = ['32', '41', '-21', '213', '12']
>>> in_2 = ['1390', '12', '-12', '20', '-2']
>>> list(map(process, in_1, in_2))
```

8. Что будет выведено на экран будет в результате выполнения следующего кода:

```
>>> values = [12, 'a8d8a', 31, ']ds*6^', 123,  '\\ssd', 31, 1, 24, '*1&ChA', '?!&s(8ps']
>>> dict(zip((filter(lambda x: type(x) == int, values)),
             (map(lambda x: str(x)[2:], values))))
```

ООП

1. Какой метод требуется перегрузить, чтобы вывести объект на экран в интерактивном режиме?
2. Как вызвать метода базового класса из метода класса-наследника?
3. Создайте класс A с полями объекта a, b, c. Создайте класс-

наследник класса А и перегрузите конструктор с использованием функции super().

4. Приведите пример инкапсуляции в Python.
5. Создайте класс своего исключения со своим сообщением.
6. Создайте список для хранения только чисел с плавающей точкой. Реализуйте метод сравнения элементов вашего списка.

Исключения

1. Опишите как с помощью одного *except* блока поймать исключения нескольких типов.
2. Чем при обработке исключений блок else отличается от блока *finally*? Может ли быть ситуация при которой выполнятся оба блока?
3. Что будет выведено на экран при вызове следующей функции? Объясните почему.

```
def f():
    try:
         10/20
         return
    except ZeroDivisionError:
        print('zero division detected')
    else:
        print('no zero division detected')
    finally:
        print('finally')
```

4. Как обратиться в полю объекта-исключения в *except* блоке? Приведите пример такого *except* блока.
5. Можно ли с помощью инструкции raise выбросить сразу несколько исключений одновременно?

1. В чем отличие синтаксических ошибок от исключений?
2. Что такое стек вызовов?
3. Что такое *traceback*?

Выводы по главе

Было рассмотрено понятие парадигмы программирования. Некоторые парадигмы программирования освоили на практике, решив ряд задач с использованием Python.

Была представлена классификация парадигм с примерами языков программирования. Особое внимание было уделено реализации функционального программирования на Python с решением задач на практике. Изучены ключевые вопросы теории объектно-ориентированного программирования, а также рассмотрены примеры задач на Python. Было определено понятие исключительной ситуации, решены задачи обработки исключительных ситуаций, а также генерация исключений на Python.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Керниган Б., Ритчи Д. Язык программирования Си. Пер. с англ., 3-е изд., испр. — СПб.: "Невский Диалект", 2001. - 352 с: ил. [электронный ресурс] URL: http://elisey-ka.ru/c/Керниган%20Б.%20и%20Ритчи%20Д.%20-%20Язык%20программирования%20Си%20(издание%203-е).pdf (дата обращения: 07.07.2019)

[2] BitwiseOperators. [электронный ресурс] URL: https://wiki.python.org/moin/BitwiseOperators (дата обращения: 09.09.2019)

[3] Python 3 - Bitwise Operators Example. [электронный ресурс] URL: https://www.tutorialspoint.com/python3/bitwise_operators_example.htm (дата обращения: 09.09.2019)

[4] Лутц М. Изучаем Python, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 1280 с., ил.

[5] Шоттс У. Командная строка Linux. Полное руководство. – СПб.: Питер, 2017. – 480 с.

[6] Командная строка Linux: краткий курс для начинающих. [электронный ресурс] URL: https://community.vscale.io/hc/ru/community/posts/209004205-%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%BD%D0%B0%D1%8F-%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0-Linux-%D0%BA%D1%80%D0%B0%D1%82%D0%BA%D0%B8%D0%B9-%D0%BA%D1%83%D1%80%D1%81-%D0%B4%D0%BB%D1%8F-%D0%BD%D0%B0%D1%87%D0%B8%D0%BD%D0%B0%D1%8E%D1%89%D0%B8%D1%85 (дата обращения: 09.09.2019)

[7] Таненбаум Э., Остин Т. Архитектура компьютера. 6-е изд. – СПб.: Питер, 2013. – 816 с.

[8] Стандарт IEEE 754-2008. [электронный ресурс] URL: https://ru.wikipedia.org/wiki/IEEE_754-2008 (дата обращения: 09.09.2019)

[9] Представление вещественных чисел. [электронный ресурс] URL: https://neerc.ifmo.ru/wiki/index.php?title=Представление_вещественных_чисел

[10] Числа с плавающей точкой/запятой согласно стандарту IEEE754-2008. [электронный ресурс] URL: http://www.learn2prog.ru/informatika/ieee754-2008.php (дата обращения: 09.09.2019)

[11] Управляющие символы ASCII. [электронный ресурс] URL:

https://ru.wikipedia.org/wiki/%D0%A3%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D1%8F%D1%8E%D1%89%D0%B8%D0%B5_%D1%81%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D1%8B (дата обращения: 09.09.2019)

[12] Диактрические знаки. [электронный ресурс] URL: https://www.setup.ru/wiki/%D0%94%D0%B8%D0%B0%D0%BA%D1%80%D0%B8%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5%20%D0%B7%D0%BD%D0%B0%D0%BA%D0%B8 (дата обращения: 09.09.2019)

[13] Alan Turing, On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction, Proceedings of the London Mathematical Society, Series 2, Volume 43 (1938), pp 544–546, doi:10.1112/plms/s2-43.6.544], The mortal matrix problem [Cassaigne, Julien; Halava, Vesa; Harju, Tero; Nicolas, Francois (2014). "Tighter Undecidability Bounds for Matrix Mortality, Zero-in-the-Corner Problems, and More". arXiv:1404.0644 [cs.DM].

[14] Stephen C. Kleene, Introduction to Meta-Mathematics, North-Holland Publishing Company, New York, 10th edition 1991, first published 1952. Chapter XIII is an excellent description of Turing machines;

[15] Langton, Chris G. (1986). "Studying artificial life with cellular automata" (PDF). Physica D: Nonlinear Phenomena. 22 (1–3): 120–149. doi:10.1016/0167-2789(86)90237-X. hdl:2027.42/26022.

[16] w3schools : Python filter() function

[17] book.pythontips : Filter

[18] younglinux : Фильтрация последовательностей

[19] Lambda and filter in Python examples

[20] Документация Python

[21] Примеры работы с zip()

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 3

ТЕРМИНЫ И ПОЯСНЕНИЯ К ОБОЗНАЧЕНИЯМ 5

ГЛАВА 1. УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ ЯЗЫКА PYTHON 6

Цель и задачи главы 6

1.1 Запуск программы 6

1.2 Процедурное программирование 10

1.3 Объекты в Python 12

1.4 Базовые типы данных 13

1.5 Приведение типов 16

1.5.1 Приведение к типу str 17

1.6 Числовые типы данных: int, float, complex 18

1.6.1 Подробнее про int 20

1) Переход от float к int 20

2) Переход от str к int 20

1.6.2 Подробнее про float 21

1) Примеры методов 21

2) Переход от int к float 23

3) Переход от str к float 23

1.6.3 Подробне про complex 24

1) Примеры создания комплексных чисел 24

2) Некоторые методы класса complex 25

1.7 Представление чисел 26

1.7.1 Представление чисел с использованием e, E 26

1.7.2 Восьмеричные oct(), шестнадцатеричные hex() и двоичные bin() числа 27

1.8 Операции в языке Python 28

1.9 Математические операции над числами 31

1.9.1 Округление и точность при работе с float 32

1.9.2 Возведение в степень ** 33

1.9.3 Интерпретация математических выражений с различными типами 33

1.10 Комбинированные операции присваивания 33

1.11 Операции сравнения 35

1.12 Логический тип данных bool 36

1.13 Преобразование в тип bool 37
1.14 Логические операции 39
1.15 Строки str 39
1.16 Доступ к элементам строки по индексу 41
1.17 Срезы для строк 43
1.18 Некоторые операции для строк 44
1.18.1 Математические операции над строками 44
1.18.2 Логические операции над строками 45
1.19 Операция проверки вхождения 46
1.20 Сравнение строк 46
1.21 Другие методы для работы со строками 47
1.21.1 Неизменяемость строк на примере работы методов 47
1) Замена подстроки (символа) в строке 47
2) Удаление пробелов из строки 47
3) Верхний и нижний регистр строк 48
1.21.2 Поиск подстроки в строке 49
1.21.3 Количество вхождений подстроки в строку 50
1.21.4 Методы startswith() и endswith() 51
1.21.5 Методы isalpha() и isdigit() 51
1.21.6 Метод join() 52
1.21.7 Форматирование строк 52
1) Метод format() 53
2) Использование % для форматирования строк 53
1.22 Ветвление 54
1.23 Циклы 55
1.23.1 Цикл for 56
1.23.2 Цикл while 56
1.23.3 Функция range(start, stop[, step]) 58
1.24 Списки list 59
1.24.1 Методы для работы с однородными списками 60
1.25 Изменяемость списков 60
1.26 Генераторы списков 61
1.27 Доступ к элементам списка 62
1.28 Срезы в списках 63
1.29 Сравнение и другие операции над списками 64
1.30 Словари. Изменяемость словарей 65

1.30.1 Другие способы создания словаря 66
1.30.2 Методы словарей 67
1.31 Генераторы словарей 70
1.32 Многомерные словари 70
1.33 Операции над словарями 71
1.34 Операции сравнения и логические операции над словарями 71
1.35 Методы split и join 72
1.36 Динамическая типизация. Переменные, ссылки и объекты 74
1.37 Разделяемые ссылки 75
1.38 Неизменяемые и изменяемые объекты 77
1.39 Функции. Общие сведения 79
1.40 Аргументы функции 81
1.41 Передача аргументов в функцию 82
1.42 Ввод и вывод данных в Python 83
1.43 Области видимости переменных 85
1.44 Идентичность объектов 86
1.45 Модули 88
1.46 Импорт собственных модулей 89
1.47 Копирование списков 90
1.48 Задача на попадание в интервал 91
1.49 Упражнения и вопросы первой главы для самоконтроля 94
1.49.1 Базовые типы данных. Ввод, вывод. 94
1.49.2 Числа, строки и операции 95
1.49.3 Словари и списки 96
1.49.4 Функции и модули 97
Выводы по главе 99
ГЛАВА 2. МОДЕЛИРОВАНИЕ РАБОТЫ КОМПЬЮТЕРА 100
Цели и задачи главы 100
2.1 Введение в архитектуру ЭВМ 100
2.1.1 Позиционные системы счисления: 2, 8, 16, 10 100
1) Прямые переходы 2->8, 2->16 103
2) Работа с системами счисления в Python 103
2.1.2 История развития компьютера 104
2.1.3 Булева алгебра, основные операции 110
2.1.4 Цифровой логический уровень 111
1) Логические 0 и 1 111

2) Логические вентили 111
2.1.5 Как устроена вычислительное устройство 116
1) Память 117
2) Шины (параллельные и последовательные) 117
3) Процессор 119
4) Базовый цикл работы процессора в архитектуре Фон Неймана 119
5) Генератор частот 120
6) Периферия 120
7) Разрядность процессора и шины 121
8) Достоинства и недостатки архитектуры Фон Неймана 122
9) Введение в ассемблер 122
10) Преобразование кода программы 123
2.2 Формат представления данных в компьютере 124
2.2.1 Формат представления целых беззнаковых чисел 125
2.2.2 Побитовые (поразрядные) операции 125
2.2.3 Конечная точность представления, переполнение 131
2.2.4 Формат представления целых знаковых чисел 133
1) Прямой код 133
Сложение чисел в прямом коде 134
Особенности прямого кода 135
2) Обратный код 136
Сложение чисел в обратном коде 137
Особенности обратного кода 138
3) Дополнительный код 139
Сложение чисел в дополнительном коде 140
Особенности дополнительного кода 142
2.2.5 Формат представления чисел с плавающей точкой 142
1) Основные сведения: мантисса, порядок и смещенный порядок 142
2) Одинарная точность 144
3) Двойная точность 147
4) Сравнение чисел с заданной точностью 148
2.2.6 Формат представления текстовой информации 150
1) Функции Python для работы с кодировками 152
2.3 Машина Тьюринга 153
2.3.1 Основные сведения 153
2.3.2 Как работает машина Тьюринга (таблица состояний) 154

2.4 Упражнения и вопросы для самоконтроля 157
2.4.1 Введение в архитектуру 157
2.4.2 Формат представления данных 157
2.4.3 Машина Тьюринга 158
Выводы по главе 158
ГЛАВА 3. ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ 159
Цели и задачи главы 159
3.1 Введение 159
3.2 Парадигмы программирования 160
3.2.1 Определение парадигмы 160
3.2.2 Императивная парадигма 160
3.2.3 Декларативная парадигма 161
3.2.4 Логическое программирование 161
3.2.5 Процедурное программирование 162
3.2.6 Функциональное программирование 163
3.2.6.1 Определение 163
3.2.6.2 Про итератор и итерируемый объект 164
Итератор 164
Итерируемый объект 167
3.2.6.3 Функция map() 168
3.2.6.4 Функция map() с функциями пользователя 170
3.2.6.5 lambda-выражения 172
3.2.6.6 Функция map() и lambda-выражения 173
3.2.6.7 Функция filter() 173
3.2.6.8 Функция filter() и lambda-выражения 174
3.2.6.9 Функция zip() 175
3.2.6.10 Функция zip() и словари dict 176
3.2.7 Парадигмы в языках программирования 177
3.3 Объектно-ориентированное программирование 177
3.3.1 Основные понятия. Класс, объект, поля, методы 177
3.3.1.1 ООП в Python 178
3.3.1.2 Вызов конструктора 179
3.3.1.3 Создание класса 179
3.3.1.4 Поля класса 181
3.3.2 Наследование как часть парадигмы 183
3.3.2.1 Наследование в Python 184

3.3.2.2 isinstance() и issubclass() 185
3.3.3 Инкапсуляция 186
3.3.4 Полиморфизм 188
3.3.5 Исключения 191
3.3.5.1 Характеристики и определение исключений 191
Стек вызовов 193
3.3.5.2 try+except [as] [else] [finally] 195
3.3.5.3 Иерархия исключений 196
3.3.5.4 Инструкция raise 197
3.4 Упражнения и вопросы для самоконтроля 198
Парадигмы программирования 198
Функциональная парадигма программирования 199
ООП 200
Исключения 200
Выводы по главе 201

ПРИЛОЖЕНИЕ А. ПРИМЕРЫ И ПОЯСНЕНИЯ

Форматирование строк с использованием %

Форматирование строк рассматривалось в разделе 3.7.7 Форматирование строк. Далее в таблице представлены краткие примеры по использованию различных спецификаторов.

Таблица А.1 - Примеры форматирования строк с использованием операции %

№	Спецификатор	Пояснение	Пример
1	`'%d', '%i', '%u'`	Десятичное число.	`>>> 'Str format, %d!' % 100` `'Str format, 100!'`

2	`'%o'`	Число в восьмеричной системе счисления.	`>>> 'Str format, %o!' % 0o127`
			`'Str format, 127!'`
			`>>> 'Str format, %o!' % 127`
			`'Str format, 177!'`

3	`'%x'`	Число в шестнадцатеричной системе счисления (буквы в нижнем регистре)	`>>> 'Str format, %x!' % 0x127`
			`'Str format, 127!'`
			`>>> 'Str format, %x!' % 127`
			`'Str format, 7f!'`

4	`'%X'`	Число в шестнадцатеричной системе счисления (буквы в верхнем регистре).	`>>> 'Str format, %X!' % 127` `'Str format, 7F!'`

5	`'%e'`	Число с плавающей точкой с экспонентой (экспонента в нижнем регистре).	`>>> 'Str format, %e!' % 2.007e-100`
			`'Str format, 2.007000e-100!'`
			`>>> 'Str format, %e!' % 0.000000002`
			`'Str format, 2.000000e-09!'`

6	`'%E'`	Число с плавающей точкой с экспонентой	`>>> 'Str format, %E!' % 0.000000002` `'Str format, 2.000000E-09!'`

		(экспонента в верхнем регистре).	

7	'%f', '%F'	Число с плавающей точкой (обычный формат).	`>>> 'Str format, %f!' % 0.0000127`
			`'Str format, 0.000013!'`
			`>>> 'Str format, %F!' % 0.0000127`
			`'Str format, 0.000013!'`

8	'%g'	Число с плавающей точкой. с экспонентой (экспонента в нижнем регистре), если она меньше, чем -4 или точности, иначе обычный формат.	`>>> 'Str format, %g!' % 0.000000002` `'Str format, 2e-09!'`
9	'%G'	Число с плавающей точкой. с экспонентой (экспонента в верхнем регистре), если она меньше, чем -4 или точности, иначе обычный формат.	`>>> 'Str format, %G!' % 0.000000002` `'Str format, 2E-09!'`

10	'%c'	Символ (строка из одного символа или число - код символа).	`>>> 'Str format, %c!' % 'P'`
			`'Str format, P!'`
			`>>> 'Str format, %c!' % 1056`
			`'Str format, P!'`

11	'%r'	Строка (литерал python).	`>>> 'Str format, %r!' % s` `"Str format, 'example'!"`

13	'%%'	Знак '%'.	`>>> 'Str format, %%!'` `'Str format, %%!'`

Экранирование

В разделе “3.1 Введение в строки” познакомились с таким понятием как экранирование. В табл. далее собраны основные экранированные последовательности, приведены краткие примеры.

Таблица А.2 - Примеры экранированных последовательностей

№	Экранирование	Пояснение	Пример
2	`\\`	Для добавления одного символа обратного слеша \	`>>> print('Hello \\ world')` `Hello \ world`
3	`\'`	Для добавления одной одинарной кавычки '	`print('Hello \' world')` `Hello ' world`
4	`\"`	Для добавления одной двойной кавычки "	`print('Hello \" world')` `Hello " world`

5	`\n`	Для перевода на новую строку	`>>> print('Hello \n world')`
			`Hello`
			` world`

6	`\r`	Для возврата каретки	`>>> print('Hello \r world')`
			`Hello`
			` world`

7	`\t`	Для добавления символа горизонтальной табуляции	`>>> print('Hello \t world')` `Hello        world`
8	`\u`	Для добавления 16-битового символа таблицы Unicode в 16-ричном представлении	`>>> print('Hello \u2E80 world')` `Hello ⺀ world`
9	`\U`	Для добавления 32-битового символа	`>>> print('Hello \U0001f300 world')` `Hello  world`

		таблицы Unicode в 32-ричном представлении	
10	`\x`	Для добавления 16-ричного значения	`>>> print('Hello \xF0 world')` `Hello ð world`
11	`\o`	Для добавления 8-ричного значения	`>>> print('Hello \o129 world')` `Hello \o129 world`
12	`\0`	Для добавления специального символа Null	`>>> print('Hello \0 world')` `Hello   world`