МИНОБРНАУКИ РОССИИ

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

ИНФОРМАТИКА. ВВЕДЕНИЕ В PYTHON

Учебное пособие

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2019

УДК
ББК
Б49

Кринкин К. В., Берленко Т. А., Заславский М. М., Чайка К. В., Размочаева Н. В.

Б49 Учебное пособие “Информатика. Введение в Python”: учеб.-метод. пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2019. 111 с.

ISBN 978-5-7629-

Представлены материалы для освоения инструмента к изучению дисциплины “Информатика”. Рассматриваются теоретические основы и базовые задачи для освоения языка Python в рамках дисциплины “Информатика”.

Предназначено для студентов направлений «Программная инженерия» и «Прикладная математика и информатика».

Утверждено
редакционно-издательским советом университета
в качестве учебно-методического пособия

ISBN 978-5-7629-

ВВЕДЕНИЕ

Цель учебного пособия – изучить и освоить на практике основы программирования на языке Python для дальнейшего решения задач дисциплины “Информатика”.

Задачи:

- 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;
- изучить основные управляющие конструкции языка Python, такие как циклы и ветвление;
- научиться работать с функциями и модулями, научиться писать свои собственные функции и модули.

Информационные технологии (операционные системы, программное обеспечение общего и специализированного назначения, информационные справочные системы) и материально-техническая база, используемые при осуществлении образовательного процесса по освоению материалов учебного пособия, соответствуют требованиям федерального государственного образовательного стандарта высшего образования.

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. С ЧЕГО НАЧИНАЕТСЯ PYTHON

1.1 Версия языка Python

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.

1.2 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 “7.4 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 или исключений. как их еще называют в Python.

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.

1.3 Процедурное программирование

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

1.4 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.

ГЛАВА 2. ЧИСЛОВЫЕ ТИПЫ ДАННЫХ И ОПЕРАЦИИ НАД НИМИ

1.1 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 “ГЛАВА 3. СТРОКОВЫЙ ТИП ДАННЫХ, ОПЕРАЦИИ НАД СТРОКАМИ, МЕТОДЫ СТРОК”.

Для задания значений логических выражений служит тип *bool*. Он допускает только два значения – *True* и *False* (см. “1.9 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 – “5.1 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.

1.2 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 “7.4 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*.

1.2.1 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 ниже.

Таблица 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*.

1.3 Числовые типы данных: int, float, complex

В предыдущем разделе упоминались числовые типы языка Python, представленные далее в табл. 2, далее рассмотрим их подробнее.

Таблица 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 C [1].

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 “1.1 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*.

1.3.1 Подробнее про int

1.3.1.1 Переход от float к int

В Python возможен переход от вещественных чисел к целым с помощью функции *int*, примеры использования которой представлены в табл. 3.

Таблица 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.

1.3.1.2 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 “7.3. Передача аргументов в функцию”.

1.3.2 Подробнее про float

1.3.2.1 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”.

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)
```

1.3.2.2 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'>
```

1.3.2.3 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 “7.4 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*.

1.3.3 Подробне про complex

1.3.3.1 Примеры создания комплексных чисел

Для создания комплексной переменной можно воспользоваться простым способом, который был рассмотрен в “1.1 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 разделе “7.8 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. 4.

Таблица 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.

1.3.3.2 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 “1.5 Операции в языке Python”.

1.4 Представление чисел

Примеры представления чисел в табл. 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 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 C и потому вещественные числа в языке Python обеспечивают такую же точность представления значений, как и в языке C [4].

1.4.1 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'>
```

1.4.2 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.

1.5 Операции в языке Python

В табл. 6 приведены некоторые часто используемые операции языка Python. Большинство из них применимы не только к числам, но и к другим объектам.

Таблица 6 – Операции в языке Python

№	Операция	Пример	Пояснение
1	`or`	`x or y`	Логическая операция ИЛИ (значение *y* вычисляется, только если значение *x* ложно)
2	`and`	`x and y`	Логическая операция И (значение y вычисляется, только если значение x истинно)
3	`not`	`not x`	Логическая операция НЕ (отрицание, инверсия)
4	`in` `not in`	`x in y` `x not in y`	Проверка на вхождение (для итерируемых объектов и множеств)
5	`is` `not is`	`x is y` `x is not y`	Проверка идентичности объектов

6	`>`
	`>=`
	`<=`
	`<`
		`x > y`
		`x >= y`
		`x < y`
		`x <= y`
			Операции сравнения

7	`==` `!=`	`x == y` `x != y`	Операция проверки на равенство Операция проверки на неравенство
8	`|`	`x | y`	Битовая операция ИЛИ
9	`^`	`x ^ y`	Битовая операция «исключающее ИЛИ» (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. 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. 6, можно поделить на группы, как это сделано в табл. 7.

Таблица 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. Подробнее про битовые операции можно познакомиться в [2] и [3].

1.6 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. 8.

Таблица 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 / b
		>>> a_f = 2.5
		>>> b_f = 5.0
		>>> b_f / a_f
			>>> c = 4 - 2j
			>>> d = 2 + 4j
			>>> c / d

	0.5 >>> b / a 2.5	2.0 >>> a_f / b_f 0.5	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, например, для двух целых чисел это будет целое число типа *int*, если одно или сразу два числа – вещественные, тогда тип результата *float*. Операция % выполняет деление чисел и возвращает в качестве результата остаток от деления – целое число типа *int*. Деление с использованием / называют *истинным*, т.к. данная операция возвращает результат с дробной частью. Обратите внимание, что результат *истинного* деления всегда получается вещественным. Операции // и % недоступны для комплексных чисел.

1.6.1 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* представления чисел.

1.6.2 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
```

1.6.3 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 (подробнее в [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.

1.7 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. 9).

Таблица 9 – Примеры комбинированных операций

№	Операция	Развернутая запись	Сокращенная запись

1	+=	>>> x = 125
		>>> y = 11
		>>> x = x + y
		>>> x
		136
			>>> x = 125
			>>> 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 = x | y
		>>> x
		127
			>>> x = 125
			>>> y = 11
			>>> x |= y
			>>> x
			127

10	&=	>>> x = 125
		>>> y = 11
		>>> x = x & y
		>>> x
		9
			>>> x = 125
			>>> y = 11
			>>> x &= y
			>>> x
			9

11	>>=	>>> x = 1225
		>>> y = 7
		>>> x = x >> y
		>>> x
		9
			>>> x = 1225
			>>> y = 7
			>>> x >>= y
			>>> x
			9

12	<<=	>>> x = 1225
		>>> y = 7
		>>> x = x << y
		>>> x
		156800
			>>> x = 1225
			>>> y = 7
			>>> x <<= y
			>>> x
			156800

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 –=.

1.8 Операции сравнения

Рассмотрим группу операций сравнения, которые представлены в табл. 6 в разделе “1.5 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
```

1.9 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. 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.

1.10 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. 10.

Таблица 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. 11.

Таблица 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`

1.11 Логические операции

Рассмотрим группу логических операций из табл. 6, которые представлены в разделе “1.5 Операции в языке Python”. Примеры выполнения операций *and*, *or* и *not* представлены в табл. 12.

Таблица 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
```

ГЛАВА 3. СТРОКОВЫЙ ТИП ДАННЫХ, ОПЕРАЦИИ НАД СТРОКАМИ, МЕТОДЫ СТРОК

3.1 Введение в строки

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 "6.3 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 "1.1 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.

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
```

3.2 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. 13):

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

Таблица 13 – Пояснение примера

index→	0	1	2	3	4	5	6
Строка c	*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.

3.3 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.

3.4 Некоторые операции для строк

3.4.1 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.

3.4.2 Логические операции над строками

Над строками можно выполнять уже знакомые нам логические операции из табл. 6 раздела “1.5 Операции в языке Python”. Примеры работы таких операций на строках приведены в табл. 14.

Таблица 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. 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*.

3.5 Операция проверки вхождения

Операции проверки вхождения *in*, *not in*, представленные в табл. 6 раздела “1.5 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
```

3.6 Сравнение строк

Доступные для применения на строках операции сравнения >, >=, <, <=, ==, != представлены в табл. 6 раздела “1.5 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 обычно рассматривается при изучении вопросов 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
```

3.7 Другие методы для работы со строками

3.7.1 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.

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 '
```

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'
```

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.

3.7.2 Поиск подстроки в строке

Для поиска подстроки в заданной строке есть специальный метод *find*. Он выполняет поиск с начала 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 с конца 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)
```

3.7.3 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
```

3.7.4 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
```

3.7.5 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
```

3.7.6 Метод *join*

Также особого внимания заслуживает метод *join*, который будет подробнее рассматриваться в разделе “5.12 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*.

3.7.7 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.

3.7.7.1 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.'
```

3.7.7.2 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.

ГЛАВА 4. ЦИКЛЫ И УСЛОВНЫЙ ОПЕРАТОР

4.1 Оператор ветвления

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.

4.2 Циклы

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

4.2.1 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 “3.5 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!
```

4.2.2 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 “7.4 Ввод и вывод данных в Python”.

4.2.3 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 “7.3 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
```

ГЛАВА 5. ТИПЫ ДАННЫХ ДЛЯ РАБОТЫ С ПОСЛЕДОВАТЕЛЬНОСТЯМИ

5.1 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 "7.7 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
```

5.1.1 Методы для работы с однородными списками

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

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

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

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

5.2 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]
```

5.3 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]]
```

5.4 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 "3.2 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. 15:

Таблица 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'
```

5.5 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']
```

5.6 Сравнение и другие операции над списками

Операции сравнения ==, !=, <, <=, >, >=, представленные в табл. 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]
```

5.7 Словари. Изменяемость словарей

Помимо списков, важную роль играют словари. Словарь – неупорядоченная коллекция, 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) 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
```

5.7.1. 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:

```
>>> key_list = [1, 2, 3, 4, 5]
>>> d = dict.fromkeys(key_list)
>>> 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}
```

5.7.2. 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
```

5.8 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.

5.9 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.

5.10 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.

5.11 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
```

5.12 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.

ГЛАВА 6. ДИНАМИЧЕСКАЯ ТИПИЗАЦИЯ И РАЗДЕЛЯЕМЫЕ ССЫЛКИ

6.1 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 C:

```
>>> 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 C. Языки, подобные Python, называются *динамически типизированными*. Языки, подобные C, 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).

Рисунок 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.

6.2 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. 2).

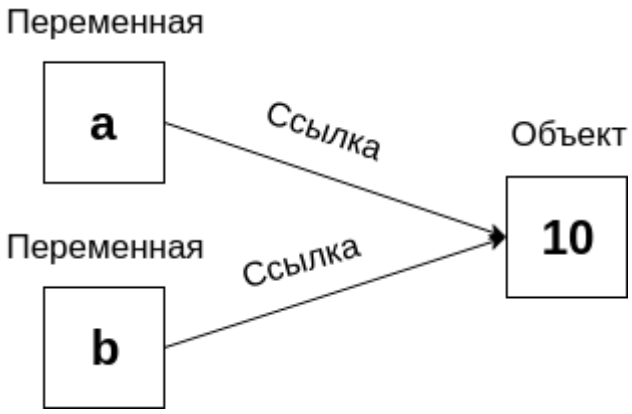

Рисунок 2 – Разделяемая ссылка

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. 3).

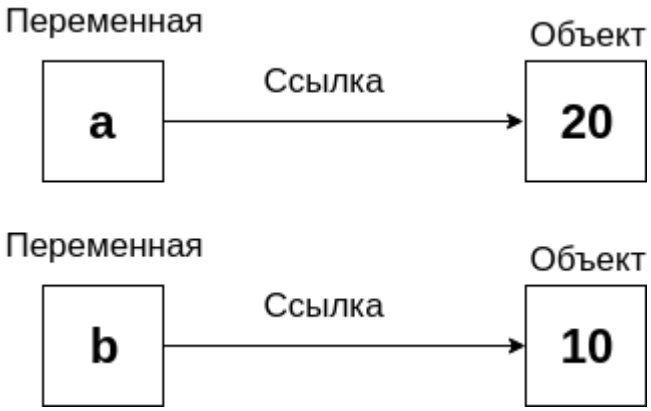

Рисунок 3 – Изменение ссылки одной из переменной в разделяемой ссылке

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

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

6.3 Неизменяемые и изменяемые объекты

В языке Python объекты делятся на два вида: изменяемые и неизменяемые. Ранее этот вопрос поднимался при обсуждении строк (3.1 Введение в строки , 3.7.1 Неизменяемость строк на примере работы методов), списков (5.2 Изменяемость списков) и словарей (5.7 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. 4):

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

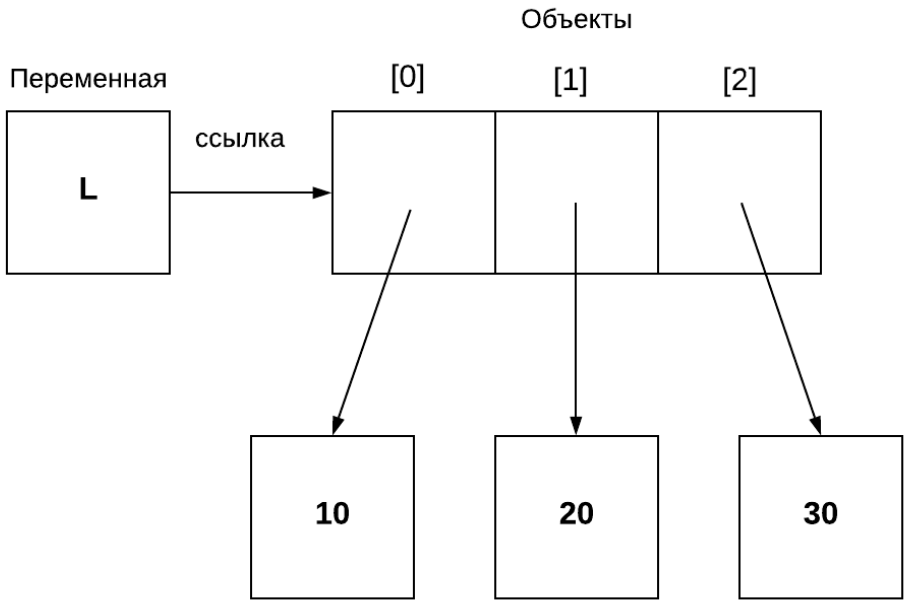

Рисунок 4 – Представление списка

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. 5):

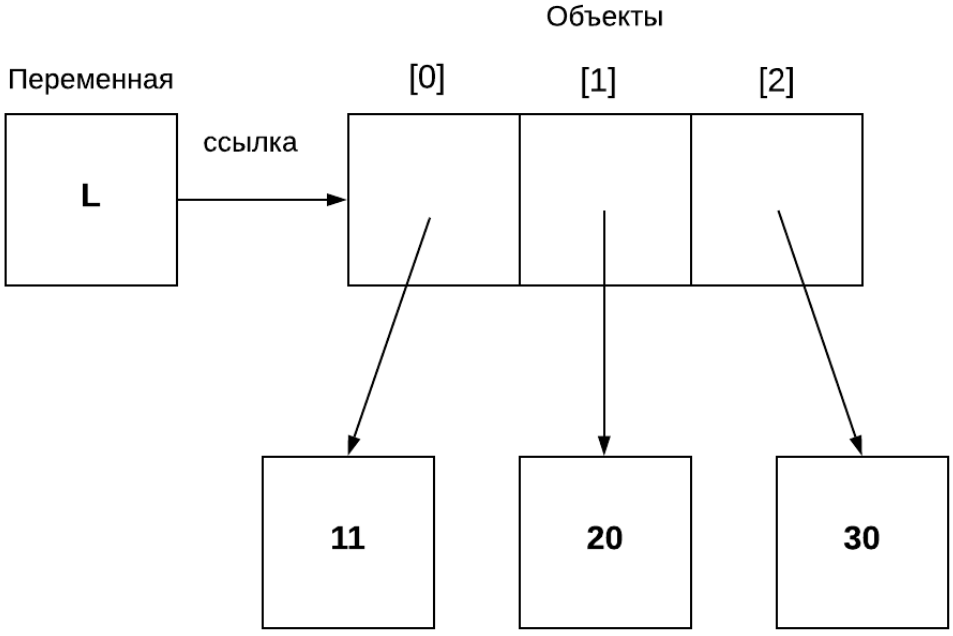

Рисунок 5 – Результат выполнения операции над списком

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'
```

ГЛАВА 7. ФУНКЦИИ И МОДУЛИ В PYTHON

7.1 Общие сведения о функциях

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. “1.2 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):	s = magic_sum(10, 25) # s == 35

sum = a + b
return sum
	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.

7.2 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]
```

7.3 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.

7.4 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!

7.5 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.

7.6 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. 6):

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

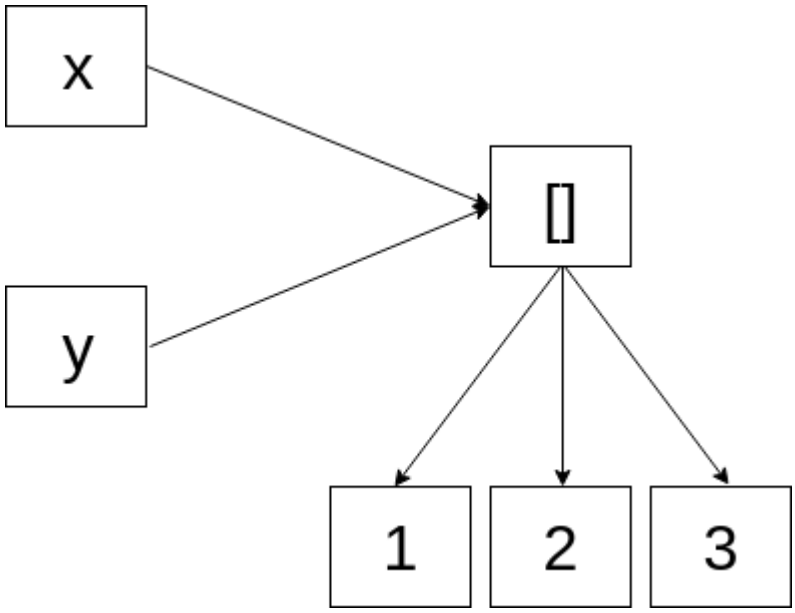

Рисунок 6 – Операция проверки идентичности

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
```

7.7 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
```

7.8 Импорт собственных модулей

В “1.4 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 – 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 2 – Файл test_module.py

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

Листинг 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.

7.9 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.

7.10 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.16.

Таблица 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. 17.

Таблица 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 "3.7 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.

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.

УПРАЖНЕНИЯ И ВОПРОСЫ ДЛЯ САМОКОНТРОЛЯ

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?

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.

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)
```

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?

ЗАКЛЮЧЕНИЕ

По результатам изучения учебного пособия достигли следующих результатов:

- познакомились с основными типами данных языка Python;
- научились использовать различные операции для работы с числами, строками и последовательностями;
- 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.

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)

СОДЕРЖАНИЕ

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 (экспонента в верхнем регистре).	>>> 'Str format, %E!' % 0.000000002 'Str format, 2.000000E-09!'

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	>>> 'Str format, %c!' % 'P'
			'Str format, P!'
			>>> 'Str format, %c!' % 1056
			'Str format, P!'

		символа).	
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 таблицы Unicode в 32-ричном представлении	>>> print('Hello \U0001f300 world') Hello 🌀 world
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