Тема 2. Встроенные типы и операции с ними

1 Знакомство с Python

1.1 Введение в стандарты программирования на Python

Одно из важнейших требований к коду Python-разработчика – следование стандарту PEP-8. Это описание рекомендованного стиля кода. Причём PEP-8 действует для основного текста программы, а для строк документации разработчику рекомендуется придерживаться положений PEP-257. Документ содержит объёмное описание стандарта.

Избегайте дополнительных пробелов в скобках (круглых, квадратных, фигурных).

Некорректно	Корректно
`x = [ '2', 3 ]`	`x = ['2', 3]`
`y = ( x [ 0 ] , x [1] )`	`y = (x[0], x[1])`
`z = { 'key' : y [ 0 ] }`	`z = {'key': y[0]}`

Используйте пробелы вокруг арифметических операций.

Некорректно	Корректно
`(a+b)+c=a+(b+c)`	`(a + b) + c = a + (b + c)`

Имена переменных и функций, атрибутов и методов класса задавайте в нижнем регистре. Разделяйте подчёркиванием входящие в имена слова.

Некорректно	Корректно
`MyVar, myVar, Var, VAR, MyFunc, myFunc, Func, FUNC`	`var, my_var, func, my_func`

При оформлении блоков кода в Python позаботьтесь об отступах. Рекомендуемый отступ составляет четыре пробельных символа. Знаки табуляции применять не рекомендуется. В популярных IDE не требуется ставить пробелы вручную. При переходе на очередную строку программного кода число пробельных символов определяется автоматически.

Некорректно	Корректно
`Главная инструкция:` `  Вложенная инструкция`	`Главная инструкция:` `  Вложенная инструкция`

Визуально фрагменты кода идентичны, но в первом отступ выполнен с помощью табуляции, а во втором – с помощью четырёх пробельных символов.

Будьте внимательнее при комбинировании апострофов и кавычек. При определении строк кавычки и апострофы равнозначны. Но во время их комбинирования возможны ошибки:

Некорректно	Корректно
`print("This is my string - "text"")` `print('This is my string - 'text'')`	`print("This is my string - 'text'")` `print('This is my string - "text"')`

1.2 Из чего состоит программа

Суть любой программы – получение, обработка и вывод данных. Данные в Python представлены объектами. Программы на языке Python можно разложить на такие составляющие, как модули, инструкции, выражения и объекты.

При этом:

1. Программы делятся на модули (файлы с расширением .py).
2. Модули содержат инструкции.
3. Инструкции состоят из выражений.
4. Выражения создают и обрабатывают объекты.

Понятия (выражения, операции, инструкции) довольно условны, но для эффективного понимания языка определимся с терминами, которыми мы будем оперировать.

Операция (англ. statement) – наименьшая автономная часть языка программирования; команда.

Если в оболочке Python мы введём:

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

Получим результат – 6.

– 2 + 4 – операция;
– 2 и 4 – операнды;
– + – оператор;
– 6 – результат операции.

Операции, которые возвращают результат, будем называть выражениями. Действия, которые не возвращают результат, а указывают интерпретатору, что делать, — это инструкции.

Выражение – это операция, которая возвращает значение.

Инструкция – операция, которая не возвращает значение.

Чтобы сохранить некоторые значения (данные) и воспользоваться ими далее в программе, используются переменные. Рассмотрим такой пример:

```
a = 10
b = a + 5
print("10+5 =", b)
```

Разберём каждую строчку нашей программы:

1. a = 10 – создаём переменную **a** и присваиваем ей значение 10, то есть теперь в переменной **a** у нас хранится значение 10.

2. b = a + 5 – создаём новую переменную **b**, затем присваиваем ей выражение (a + 5). Так как в переменной **a** у нас хранится значение 10, вместо **a** подставляется её значение – 10. Получаем: b = (10 + 5). Или после сложения: b = 15.

3. print("10 + 5 = ", b) – команда (функция) print() выводит на экран значение аргументов или, проще говоря, те данные, которые указали в скобках. print() может выводить сразу несколько значений, для этого аргументы указываются через запятую. Наша функция имеет два аргумента: "10 + 5 = " и b. Первый аргумент – это строка текста. Строку текста легко можно отличить по верхним кавычкам "". Второй аргумент – переменная **b**, но на экран выводится не имя переменной, а её значение.

Об аргументах будет сказано подробнее в теме функций. Пока просто запомните, что это данные, которые указываем в скобках через запятую. После каждой запятой идёт новый аргумент.

Переменная – поименованная область памяти, имя или адрес, который можно использовать для осуществления доступа к данным, находящимся в переменной (по этому адресу).

Присваивание переменной – передача в переменную нового значения.

Значение переменной – информация, хранящаяся в переменной. В переменной может храниться текст, целое число, число с десятичной точкой и т. д.

Знак = – операция присваивания, а также инструкция. То есть такая операция не возвращает результата.

1.3 Динамическая типизация как один из важнейших аспектов программирования на Python

Python поддерживает динамическую типизацию, то есть тип переменной определяется автоматически во время исполнения. Поэтому вместо «присваивания значения переменной» лучше говорить о «связывании значения с некоторым именем».

```
>>> a = 8
```

Рассмотрим, как Python обработает это выражение:

В памяти будет создан объект целого типа (int), переменная **a** получит ссылку на этот объект.

Чтобы лучше понять суть динамической типизации, рассмотрим следующий пример:

```
>>> a = 4
>>> a = a + 1
>>> a = "text"
```

В памяти создаётся объект типа int (целое), переменная **a** получает на него ссылку.

В правой части оператора = стоит выражение, и сначала будет вычислен результат выражения. После вычисления результата создаётся новый объект типа int (со значением 5). Переменная **a** получит ссылку на новый объект в памяти. На старый объект int (со значением 4) она больше не будет ссылаться.

Затем создаётся новый объект типа str (строка), переменная **a** снова изменит ссылку.

В отличие от языков со статической типизацией, таких как C++ или Pascal, переменная в Python не имеет типа! Правильно говорить: «Переменная указывает на объект такого-то типа». То есть именно объект в памяти имеет тип, а переменная – просто указатель.

Поэтому когда мы связываем с переменной некоторое значение, просто переносим указатель на другой объект. Python предоставляет мощную коллекцию объектных типов, встроенных напрямую в язык.

Встроенные типы данных (часть):

Название типа	Описание
`int`	Это функция, возвращающая целое число в десятичной системе счисления. Пример: 2, 4, 8, -10, -2
`float`	Это функция, возвращающая число с плавающей запятой. Пример: 2.6, -5.2
`str`	Это функция, возвращающая строку (неизменяемую последовательность символов)
`bool`	Это функция, возвращающая булево значение (True или False) для объекта
`list`	Функция, возвращающая изменяемую упорядоченную коллекцию объектов произвольных типов. Пример: [2, 2.4, “Hello”]

`tuple`	Функция, возвращающая неизменяемую упорядоченную коллекцию объектов произвольных типов. Кортеж. Пример:(2, 2.4, “Hello”)
`dict`	Функция, возвращающая неупорядоченную коллекцию произвольных объектов с доступом по ключу. Пример: {“name”: “Вася”, “age”: 10}

1.4 Механизмы реализации ввода/вывода данных

Основное назначение компьютерных программ — обработка данных. Программа может получать их разными способами, например, запрашивать у пользователя. Результат обработки данных может быть возвращён пользователю посредством вывода на экран в текстовой форме.

Чтобы запросить данные у пользователя с клавиатуры, воспользуемся функцией **input()**.

Функция **input()** может получать необязательный аргумент — строку, которая будет выведена в качестве приглашения/уточнения. В качестве результата она вернёт введённые пользователем данные.

```
name = input("Введите ваше имя: ")
```

Введите ваше имя: <здесь программа остановится и будет ждать ввода с клавиатуры>.

Переменной **name** будет присвоена строка введённых символов.

Обратите внимание: input() всегда возвращает строку. Если вы хотите работать с цифрами, используйте функции преобразования типов **int()**, **float()**.

```
a = int(input("Введите целое число: "))
```

Применять функцию **str()** к вводимым строковым данным не требуется.

Неправильно:

```
a = str(input("Введите текст: "))
```

Правильно:

```
a = input("Введите текст: ")
```

Для вывода в консоль пользуемся функцией **print()**.

Функция **print()** принимает неограниченное количество аргументов, которые будут выведены на экран.

```
name = "Иван
print("Меня зовут", name)
Меня зовут Иван
```

1.5 Арифметические и логические операции в Python

Список доступных арифметических операций в Python приводится в таблице:

Арифметические операторы в Python

Оператор	Описание	Примеры
+	Сложение	`print(398 + 20) -> 418`
-	Вычитание	`print(200 - 50) -> 150`
*	Умножение	`print(34 * 7) -> 238`

/	Деление	`print(36 / 6) -> 6.0`
		`print(36 / 5) -> 7.2`
		`print(round(36 / 7, 2)) -> 5.14`
		`print(round(-36 / -7, 3)) -> 5.143`

//	Целочисленное деление	`print(36 // 6) -> 6`
		`print(36 // 5) -> 7`
		`print(-9 // 4) -> -3`
		`print(5 // -2) -> -3`

%	Остаток от деления	`print(36 % 6) -> 0` `print(36 % 5) -> 1`
**	Возведение в степень	`print(2 ** 16) -> 65536`

Обратите внимание на операцию целочисленного деления с участием отрицательных чисел в качестве делимого или делителя. Сравним два примера:

Пример	Результат
Деление	
`print(9 / 4)`	2.25
`print(-9 / 4)`	-2.25
Целочисленное деление	
`print(9 // 4)`	2
`print(-9 // 4)`	-3

Такой результат обусловлен тем, что целочисленное деление в Python 3 округляет итоговое значение в меньшую сторону. То есть для числа 2.25 это 2, а для числа -2.25 — -3.

С логическими операциями мы отлично знакомы с уроков математики.

Логические операторы в Python

Оператор	Описание	Примеры
>	Больше	`print(40 > 40) -> False`
<	Меньше	`print(3 < 9) -> True`
==	Равно	`print(10 == 10) -> True`
!=	Не равно	`print(2 != 2) -> False`
>=	Больше или равно	`print(40 >= 1) -> True`
<=	Меньше или равно	`print(3 <= 1) -> False`

and	Логическое «И». Возвращает значение «Истина», если оба операнда имеют значение «Истина»	`print(True and True) -> True`
		`print(True and False) -> False`
		`print(False and True) -> False`
		`print(False and False) -> False`

or	Логическое «ИЛИ». Возвращает значение «Истина», если хотя бы один из операндов имеет значение «Истина»	`print(True or True) -> True`
		`print(True or False) -> True`
		`print(False or True) -> True`
		`print(False or False) -> False`

not	Логическое «НЕ». Изменяет логическое значение операнда на противоположное	`print(not True) -> False` `print(not False) -> True`
in	Оператор проверки принадлежности. Возвращает значение «Истина», если элемент присутствует в последовательности	`print(10 in [10, 20, 30]) -> True`

is	Оператор проверки тождественности. Возвращает значение «Истина», если операнды ссылаются на один объект	`x = 3`
		`y = 3`
		`print(x is y) -> True`

Операторные скобки

В любом языке программирования нужно выделять блоки кода. Для этого используются специальные синтаксические конструкции, показывающие начало и конец блока. В Pascal это ключевые слова begin… end; в C++ – фигурные скобки {...}. В Python – операторные скобки, одинаковые отступы слева перед всеми инструкциями блока.

Подобный синтаксис языка хорош тем, что заставляет программиста правильно табулировать свой код, улучшая читабельность.

1.6 Следования, ветвления и циклы в Python, вложенные инструкции

Разветвления

Оператор if

В теории программирования доказано, что программу для решения любой задачи можно составить из трёх структур, называемых следованием, ветвлением и циклом.

Следованием называется конструкция, представляющая собой последовательное выполнение двух или более операторов (простых или составных).

Ветвление задаёт выполнение либо одного, либо другого оператора в зависимости от выполнения какого-либо условия.

Со следованием всё просто: все команды (инструкции) выполняются последовательно, пока программа не завершится.

Познакомимся с ветвлениями.

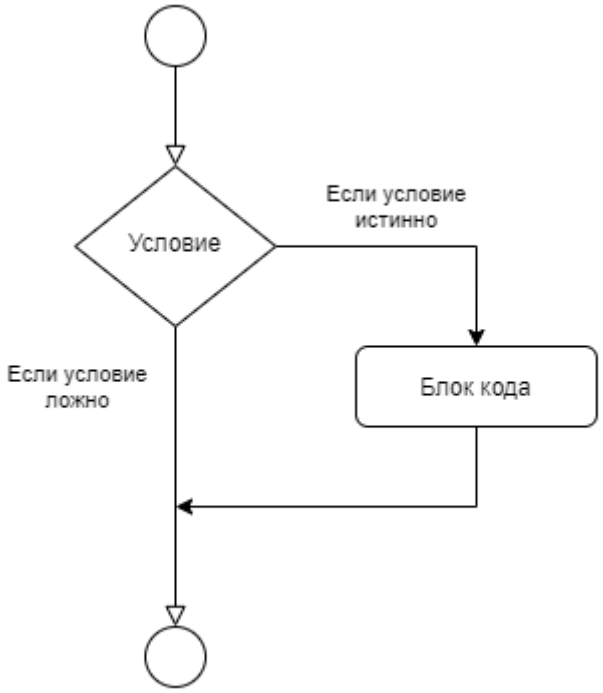

Рисунок 1 – Схема ветвления **if**.

Описание схемы

Оператор **if** называют инструкцией. В качестве выражения может выступать любое выражение, которое будет автоматически преобразовано в логическое.

Цель **if** – выполнить некоторый блок кода при определённом условии.

Если выражение истинно (**True**), то выполняется «Блок кода». При ложном выражении (**False**) «Блок кода» пропускается, программа выполняется дальше.

Синтаксис выглядит следующим образом:

```
if condition:
    #block of statements
else:
    #another block of statements (else-block)
```

Пример:

```
# Если число положительное, мы выводим соответствующее сообщение

num = int(input("Enter integer number: "))
if num > 0:
    print(num, "is a positive number.")
print("This is always printed.")
```

Рассмотрим этот пример подробнее.

Запрашивается целое число, если оно больше 0, то выводится сообщение, что число положительное.

В этом примере мы также увидели текстовое описание, которому предшествует символ **#.** В Python так обозначаются однострочные комментарии в коде. Многострочный комментарий в этом случае потребует символа # перед каждой строкой, что при большом блоке-комментарии ухудшает презентабельность кода. В этом случае лучше использовать многострочный комментарий:

```
'''
Строка 1
Строка 2
Строка 3
'''

"""
Строка 1
Строка 2
Строка 3
"""
```

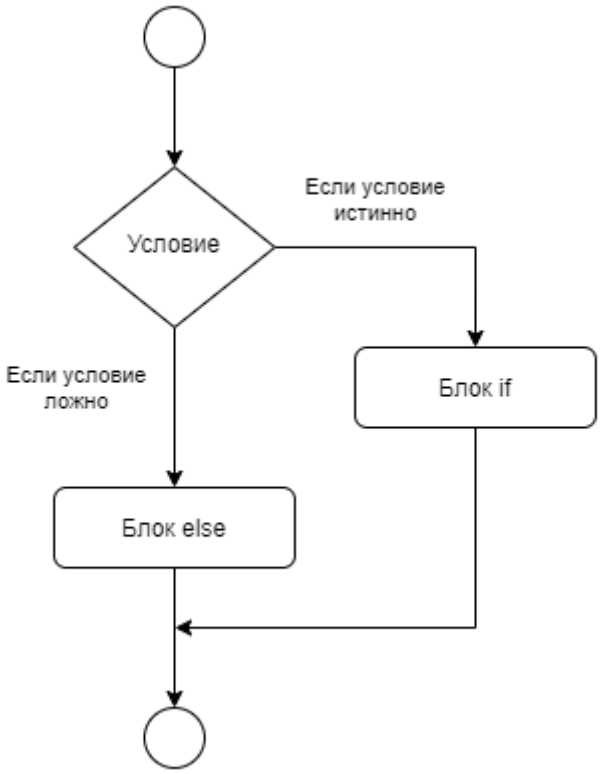

Рисунок 2 – Схема ветвления **if else**

Описание схемы

Если выражение истинно (**True**), то выполняется «Блок if», при ложном выражении (**False**) – «Блок else». То есть выполняется либо первый блок, либо второй. Программа для проверки того, является ли число четным или нет:

```
num = int(input("enter the number?"))
if num%2 == 0:
    print("Number is even...")
else:
    print("Number is odd...")
```

Вложенные инструкции

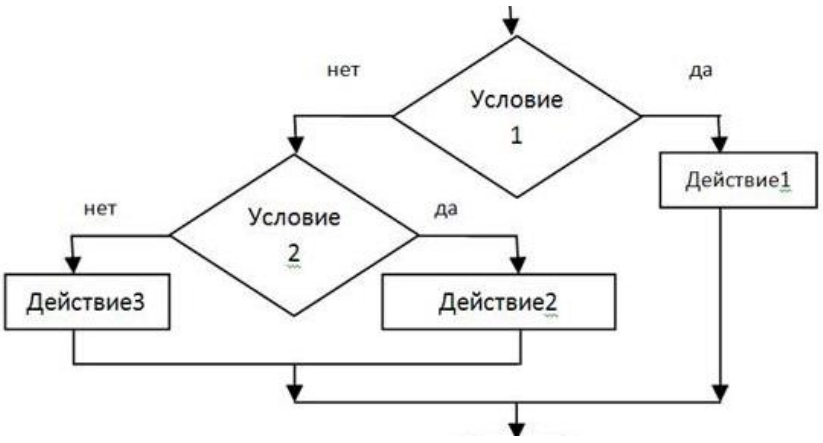

Рисунок 3 – Схема вложенных инструкций ветвления

Внутри блока условной инструкции могут находиться любые другие инструкции, в том числе и условная. Это вложенные инструкции. Синтаксис вложенной условной инструкции:

```
if                                                    условие1:
...
    if                                                условие2:
    ...
    else:
    ...
    ...
else:
...
```

Вместо многоточий можно писать произвольные инструкции. Обратите внимание на размеры отступов перед инструкциями. Блок вложенной условной инструкции отделяется четырьмя пробельными символами.

Уровень вложенности условных инструкций может быть произвольным. То есть внутри одной условной инструкции может быть вторая, а внутри неё — ещё одна и т. д. Условие 2 проверяется, только если верно условие 1.

Вложенные инструкции на одном уровне вложенности

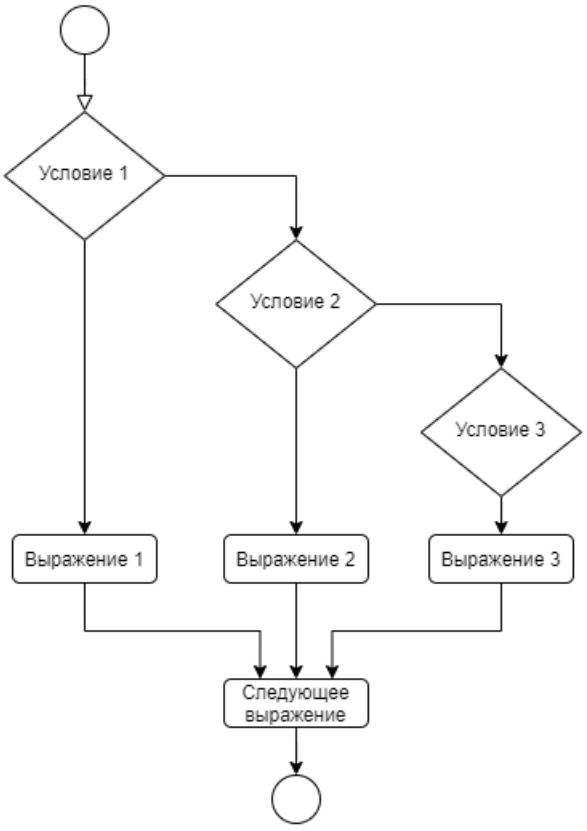

Рисунок 4 – Полная схема инструкции ветвления

Описание схемы

Оператор **elif** переводится как «иначе если». Логическое выражение, стоящее после оператора elif, проверяется, только если все вышестоящие условия ложные. То есть в этой схеме может выполниться только один блок кода: первый, второй, третий или четвёртый. Если одно из выражений истинно, то нижестоящие условия проверяться не будут.

Синтаксис:

```
if expression 1:
    # block of statements

elif expression 2:
    # block of statements

elif expression 3:
    # block of statements

else:
    # block of statements
```

Пример:

Чтобы проверялись все условия, независимо от результата предыдущего, следует использовать несколько независимых операторов **if**.

Рассмотрим ещё один пример:

```
numb_1 = int(input("Введите первое целое число: "))
numb_2 = int(input("Введите второе целое число: "))

if numb_1 != numb_2:
    print("Числа не равны")
    if numb_1 > numb_2:
    print("Первое число больше второго")
    elif numb_1 < numb_2:
    print("Первое число меньше второго")
elif numb_1 == numb_2:
    print("Числа равны")
```

Результат:

```
Введите первое число: 40
Введите второе число: 20
Числа не равны
Первое число больше второго
```

Пример:

```
numb_1 = float(input("Введите первое вещественное число: "))
numb_2 = float(input("Введите второе вещественное число: "))

if numb_1 >= numb_2:
    print("Первая ветвь")
    if numb_1 > numb_2:
    print("Первое число больше второго")
    else:
    print("Числа равны")
elif numb_1 <= numb_2:
    print("Вторая ветвь")
    if numb_1 < numb_2:
    print("Первое число меньше второго")
    else:
    print("Числа равны")
```

Результат:

```
Введите первое вещественное число: 4.6
Введите второе вещественное число: 1.2
Первая ветвь
Первое число больше второго
```

Знакомство с циклами

Цикл задаёт многократное выполнение оператора.

Все программы, которые мы писали до сих пор, запускались, выполняли необходимые действия, выводили результат и завершали свою работу. Чтобы выполнить любую из наших программ с другим набором данных, нужно запустить её заново. Но как много реальных программ вы знаете, которые немедленно завершают свою работу после выполнения некоторых действий?

Практически все программы работают непрерывно: выполнив одни действия, ожидают новых инструкций. И так до тех пор, пока пользователь не завершит работу программы. Работу большинства программ можно представить в таком виде: получение данных/инструкций --> обработка данных --> вывод результата --> получение данных/инструкций --> обработка данных --> вывод результата … Так будет происходить, пока пользователь не завершит работу с программой. Это и есть работа программы в цикле.

Циклы – это инструкции, выполняющие одну и ту же последовательность действий, пока актуально заданное условие.

В Python существуют два типа циклов: **while** и **for in**. В этой лекции мы познакомимся только с первым циклом.

Рисунок 5 – Схема цикла **while**

Описание схемы

Цикл while (“пока”) позволяет выполнить одну и ту же последовательность действий, пока проверяемое условие истинно. Условие записывается до тела цикла и проверяется до выполнения тела цикла. Как правило, цикл while используется, когда невозможно определить точное значение количества проходов исполнения цикла.

Синтаксис цикла while в простейшем случае выглядит так:

```
while условие:
    блок инструкций
```

При выполнении цикла `while` сначала проверяется условие. Если оно ложно, то выполнение цикла прекращается и управление передается на следующую инструкцию после тела цикла `while`. Если условие истинно, то выполняется инструкция, после чего условие проверяется снова и снова выполняется инструкция. Так продолжается до тех пор, пока условие будет истинно. Как только условие станет ложно, работа цикла завершится и управление передастся следующей инструкции после цикла.

Один шаг цикла (однократное выполнение тела цикла) называется **итерацией.** Пример цикла:

```
number = int(input('Введите целое число от 0 до 9: '))
while number < 10:
    print(number)
    number = number + 1
print('программа завершена успешно')
```

Рассмотрим пример подробнее.

Программа выводит на экран все числа — от введённого числа до 9, с шагом 1. Например, если мы введём число 7, программа выведет 7, 8 и 9.

Вторая строка — это оператор цикла while и number < 10 — логическое выражение.

Третья и четвёртая строки — это тело цикла, которое будет выполняться до тех пор, пока логическое выражение number < 10 будет истинно. Пятая строчка не относится к телу цикла, так как перед ней нет отступа.

Сколько раз выполнится тело цикла, заранее неизвестно — это зависит от заданного значения переменной number.

Обратите внимание на строчку 4. При каждом выполнении этой строки в цикле её значение будет увеличиваться на единицу до тех пор, пока значение переменной number не станет больше либо равно 10. При этом значении логическое выражение number < 10 станет ложным, цикл завершится.

Зацикливание

Рассмотрим такой пример:

```
a = 5
while a > 0:
    print("!")
    a = a + 1
```

Запустив этот пример, вы увидите кучу восклицательных знаков, и так до бесконечности. Цикл при текущих условиях не завершится никогда, потому что **a** всегда больше нуля, условие **a** > 0 всегда будет верным. В программах нужно избегать бесконечных циклов. Операционная система считает зациклившуюся программу повисшей (нерабочей) и предлагает снять с неё задачу.

После тела цикла можно написать слово `else:` и после него блок операций, который будет выполнен *один раз* после окончания цикла, когда проверяемое условие станет неверно:

```
i = 1
while i <= 10:
    print(i)
    i += 1
else:
    print('Цикл окончен, i =', i)
```

Казалось бы, никакого смысла в этом нет, ведь эту же инструкцию можно просто написать *после* окончания цикла. Смысл появляется только вместе с инструкцией `break`. Если во время выполнения Питон встречает инструкцию

`break` внутри цикла, то он сразу же прекращает выполнение этого цикла и выходит из него. При этом ветка `else` исполняться не будет. Разумеется, инструкцию `break` осмысленно вызывать только внутри инструкции `if`, то есть она должна выполняться только при выполнении какого-то особенного условия.

Инструкции break, continue

В теле цикла можно использовать вспомогательные инструкции **break** и **continue**. Иногда применение этих инструкций позволяет упростить ваш код и сделать его более читабельным.

Оператор **continue** начинает следующий проход цикла, минуя оставшееся тело цикла.

Оператор **break** досрочно прерывает цикл.

Пример:

```
i = 0
while True:
    i += 1
    if i >= 10:
    # инструкция break при выполнении немедленно заканчивает
выполнение цикла
      break
    if i % 2 == 0:
    # переходим к проверке условия цикла,
    # пропуская все операторы за инструкцией
      continue
    print(i)
    #i += 1
```

Подробнее о цикле **while**, операторах **break** и **continue** читайте по ссылке.

1.7 Способы форматирования строк

В процессе работы над программой у разработчика часто возникают ситуации, когда необходимо сформировать строку. Для этого нужно подставить в неё некоторые данные, полученные в процессе выполнения программы. Это данные на основе пользовательского ввода, значения переменных, вывод из файлов и т. д. Для подстановки можно воспользоваться одним из методов форматирования строк: оператором **%**, функцией **format()**, **f**-строками. Последний вариант (**f**-строки) работает быстрее других способов. Поэтому если вы работаете под Python 3.6 и старше, используйте именно его.

Форматирование через оператор %

Если для подстановки требуется только один аргумент, то значением будет сам аргумент.

```
name = input("Enter your name: ")
print("Hello, %s!" % name)
```

Выравнивание по левой стороне.

```
print("%-10s %-10s %-10s" % ('papam1', 'papam2', 'papam3'))
```

Указание количества цифр после запятой.

```
print("%.2f" % (20.0/8))
```

Значения в подстановке могут различаться по типу. В примерах выше мы подставляли значения строкового типа (**%s**), но можно выполнять и другие подстановки.

Форматирование через оператор %

Подстановка	Тип данных
"%s"	Строка
"%d"	Десятичное число
"%f"	Число с плавающей точкой
"%o"	Число в восьмеричной системе
"%x"	Число в шестнадцатеричной системе

Форматирование через метод format()

Используется специальный символ {} для указания точки подстановки значения, передаваемого методу format. Каждая пара скобок определяет одно место для подстановки.

```
print('{}'.format(['el_1', 'el_2', 'el_3', 'el_4']))
```

Вывод данных столбцами одинаковой ширины по 20 символов с выравниванием по правой стороне.

```
print("{:>20} {:>20} {:>20}".format('my_param_1', 'my_param_2',
'my_param_3'))
```

Указание количества цифр после запятой.

```
print("{:.3f}".format(5.0/3))
```

Передать параметры можно и через указание их индексов в фигурных скобках:

```
print('Третий элемент: {2}; Второй элемент: {1}; Первый элемент:
{0}'.format('el_1', 'el_2', 'el_3'))
```

Результат:

```
Третий элемент: el_3; Второй элемент: el_2; Первый элемент: el_1
```

Форматирование через f-строки

f-строки — механизм форматирования строки с префиксом **f**. Внутри **f**-строки в паре фигурных скобок указываются имена переменных, которые необходимо подставить. Информация об **f**-строках доступна по ссылке.

```
ip = '192.168.1.4'
mask = 10

print(f"ip-params: {ip}, mask: {mask}")
```

Помимо подстановки значений переменных, в фигурных скобках допустимо применить выражение:

```
octets = ['10', '1', '1', '1']
mask = 10
```

```
print(f"ip-params: {'.'.join(octets)}, mask: {mask}")
```

Через **f**-строки также возможен вывод столбцами с одинаковым расстоянием между ними:

```
    oct1,  oct2,  oct3,  oct4  =  [10,  1,  1,  1]

print(f'IP  address:  {oct1:<8}  {oct2:<8}  {oct3:<8}
{oct4:<8}')
```

2 Встроенные типы и операции с ними

2.1 Тип данных: число.

В Python доступны следующие виды чисел: целые (тип **int**), вещественные (тип **float**), комплексные (тип **complex**).

Целые (int)

В Python 3 числа соответствуют обычным числам. Некоторые операции были рассмотрены ранее. Дополнительно над целыми числами можно производить такие операции, как взятие числа по модулю и битовые операции.

Операция	Пример
Взятие по модулю	`print(abs(-6)) -> 6`
Побитовое И	`print(4 & 6) -> 4`
Побитовое ИЛИ	`print(4 | 6) -> 6`
Побитовое исключающее ИЛИ	`print(4 ^ 6) -> 2`
Битовый сдвиг влево	`print(4 << 6) -> 256`
Битовый сдвиг вправо	`print(4 >> 6) -> 0`

Числа в Python могут быть представлены не только в десятичной, но и в других системах счисления. Для перевода между системами счисления применяются специальные функции.

Функция	Описание	Пример
`int()`	Преобразовать к целому числу в десятичном формате (по умолчанию). Также допускается выбор другой системы счисления с помощью дополнительного параметра (от 2 до 36)	`print(int(17.5)) -> 17` `print(int('10001', 2)) -> 17`
`bin()`	Преобразовать к двоичному формату	`print(bin(17))-> 0b10001`
`oct()`	Преобразовать к восьмеричному формату	`print(oct(17)) -> 0o21`
`hex()`	Преобразовать к шестнадцатеричному формату	`print(hex(17)) -> 0x11`

Вещественные (float)

Поддерживают операции, аналогичные операциям, которые выполняются с целыми числами. Более подробно рассмотрены в ранее.

Комплексные (complex)

Под комплексным числом понимается выражение вида **a + ib**, где **a** и **b** – любые действительные числа, **i** – мнимая единица.

```
n_1 = complex(5, 6)
print(n_1)
n_2 = complex(7, 8)
print(n_2)
```

Результат:

```
(5+6j)
(7+8j)
```

2.2 Тип данных: строка

Строка в Python — упорядоченный набор символов для хранения и представления текстовой информации.

Пример:

```
my_str = "простая строка"
print(my_str)
print(type(my_str))
```

Результат:

```
простая строка
<class 'str'>
```

Простейший тип данных – упорядоченная неизменяемая коллекция элементов. Со строками в Python можно выполнять множество операций, например:

Конкатенация (сцепление)

Пример:

```
s1 = 'abra'
```

```
s2 = 'kadabra'
print(s1 + s2)
```

Результат:

```
abrakadabra
```

Взятие элемента по индексу

Пример:

```
s = 'abrakadabra'
print(s[1])
```

Результат:

```
b
```

Извлечение среза

Синтаксис: **[s:f:step]**, где **s** – начало среза, **f** – окончание, **step** – шаг (опционально).

Пример:

```
s = 'abrakadabra'
print("1-", s[4:7])
print("2-", s[3:-3])
print("3-",s[:5])
print("4-",s[3:])
print("5-",s[:])
print("6-",s[::-1])
print("7-",s[1:7:2])
```

Результат:

```
1- kad
2- akada
3- abrak
4- akadabra
5- abrakadabra
6- arbadakarba
7- baa
```

Механизмы реверса строк

Рассмотрим следующие механизмы реверса строк: срез, обратная итерация, алгоритм реверса на месте.

1. Срез.

Пример:

```
string = "abrakadbra"
str_reverse = string[::-1]
print(str_reverse)
```

Результат:

```
arbdakarba
```

2. Обратная итерация.

Пример:

```
for el in reversed("abrakadbra"):
    print(el)
```

Результат:

```
a
r
b
d
a
k
a
r
b
a
```

3. Реверс на месте.

Пример:

```
string = "abrakadabra"
str_reverse = ''
symbols = list(string)
for el in range(len(string) // 2):
```

```
    tmp = symbols[el]
    symbols[el] = symbols[len(string) - el - 1]
    symbols[len(string) - el - 1] = tmp
str_reverse = ''.join(symbols)
print(str_reverse)
```

Результат:

```
arbadakarba
```

Таблица методов строк

Рассмотрим методы, применяемые в приложениях для операций со строками, и примеры их использования.

Функция	Описание	Пример
`len(строка)`	Возвращает длину строки	`print(len("my_string"))` `-> 9`

`строка.split(<разделитель>)`	Разбить строку по разделителю	`print("раз два три".split())`
		`-> ['раз', 'два', 'три']`
		`print("четыре_пять_шесть".split('_'))`
		`-> ['четыре', 'пять', 'шесть']`

`<разделитель>.join(список)`	Собрать строку из списка с указанным разделителем	`print('_'.join(['раз', 'два', 'три'])) -> раз_два_три` `print(''.join(['раз', 'два', 'три'])) -> раздватри`
`строка.title()`	Перевести первую букву каждого слова в верхний регистр, остальные - в нижний	`print("лето быстро пролетело".title())` `-> Лето Быстро Пролетело`
`строка.upper()`	Преобразовать строку к верхнему регистру	`print('простая строка'.upper())` `-> ПРОСТАЯ СТРОКА`
`строка.lower()`	Преобразовать строку к нижнему регистру	`print('ПРОСТАЯ СТРОКА'.lower())` `-> простая строка`

`строка.istitle()`	Проверить, начинаются ли слова строки с буквы в верхнем регистре	`print('Лето Быстро Пролетело'.istitle()) -> True` `print('Лето быстро Пролетело'.istitle()) -> False`
`строка.isupper()`	Проверить, состоит ли строка из символов в верхнем регистре	`print('ПРОСТАЯ СТРОКА'.isupper()) -> True` `print('простая строка'.isupper()) -> False`
`строка.islower()`	Проверить, состоит ли строка из символов в нижнем регистре	`print('простая строка'.islower()) -> True` `print('ПРОСТАЯ СТРОКА'.islower()) -> False`
`ord(символ)`	Получить ASCII-код для символа	`print(ord(',')) -> 44`
`chr(код)`	Получить символ по ASCII-коду	`print(chr(44)) -> ','`
`строка.count(подстрока, [начало], [конец])`	Вернуть количество вхождений подстроки в строку	`print('lalala'.count('la')) -> 3` `print('lalala'.count('la', 2, 4)) -> 1`
`строка.capitalize()`	Перевести первый символ строки в верхний регистр, остальные - в нижний	`print('cTPOKA'.capitalize()) -> Строка`
`строка.startswith(шаблон)`	Проверить, начинается ли строка с шаблона	`print('papapa'.startswith('pa')) -> True` `print('papapa'.startswith('не')) -> False`
`строка.endswith(шаблон)`	Проверить, заканчивается ли строка шаблоном	`print('papapa'.endswith('pa')) -> True` `print('papapa'.endswith('не')) -> False`

`строка.replace(шаблон, замена)`	Заменить в строке шаблон на указанную подстроку	`print('lalala'.replace('la', 'na')) -> 'nanana'`

`строка.index(подстрока, [начало], [конец])`	Найти подстроку в строке. Получить позицию первого вхождения или получить ValueError	`print('lalalala'.index('la'))`
		`-> 0`
		`print('lalalala'.index('la', 4, 6))`
		`-> 4`
		`print('lalalala'.index('la', 10, 20))`
		`-> ValueError: substring not found`

`строка.find(подстрока, [начало], [конец])`	Найти подстроку в строке. Получить позицию первого вхождения или получить -1	`print('lalalala'.find('la'))`
		`-> 0`
		`print('lalalala'.find('la', 4, 6))`
		`-> 4`
		`print('lalalala'.find('la', 10, 20))`
		`-> -1`

С другими методами строк вы можете ознакомиться по ссылке.

2.3 Тип данных: список

В Python массивов как таковых нет. Их роль выполняют списки. Под списками понимаются упорядоченные изменяемые наборы объектов произвольного типа. Самый простой способ создать список — применить функцию **list()** к итерируемому объекту, например, к строке. Пример:

```
print(list('обычная строка'))
```

Результат:

```
['о', 'б', 'ы', 'ч', 'н', 'а', 'я', ' ', 'с', 'т', 'р', 'о', 'к', 'а']
```

Задаем список. Пример:

```
result_list = [2, 'text', 456, 45.3, None]
print(result_list)
```

Результат:

```
[2, 'text', 456, 45.3, None]
```

result_list.append(). Для добавления единственного элемента используйте append(). result_list.append(new_el). Элемент, который требуется добавить в список - new_el. Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# append
result_list.append("new_el")
print(result_list)
```

Результат:

```
[2, 'text', 456, 45.3, None, 'new_el']
```

result_list.extend() - Дополняет список элементами из указанного объекта. Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# extend
result_list.extend([8, 9, 10])
print(result_list)
```

Результат:

```
[2, 'text', 456, 45.3, None, 8, 9, 10]
```

result_list.insert(i, x). Вставляет указанный элемент перед указанным индексом. **i** – позиция (индекс), перед которой требуется поместить элемент. Нумерация ведётся с нуля. Поддерживается отрицательная индексация. **x** – Элемент, который требуется поместить в список. Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# insert
result_list.insert(1, "ins_el")
print(result_list)
```

Результат:

```
[2, 'ins_el', 'text', 456, 45.3, None]
```

result_list.remove(x).Удаляет из списка указанный элемент. x – элемент, который требуется удалить из списка. Если элемент отсутствует в списке, возбуждается ValueError. Удаляется только первый обнаруженный в списке элемент, значение которого совпадает со значением переданного в метод. Пример:

```
result_list = [2, 'text', 456, 45.3, None, "ins_el"]

# remove
result_list.remove("ins_el")
print(result_list)
```

Результат:

```
[2, 'text', 456, 45.3, None]
```

result_list.pop(i).Возвращает элемент [на указанной позиции], удаляя его из списка. **i=None** – Позиция искомого элемента в списке (целое число). Если не указана, считается что имеется в виду последний элемент списка. Отрицательные числа поддерживаются. Чтобы удалить элемент из списка не возвращая его, воспользуйтесь list.remove(). Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# pop
result_list.pop(1)
print(result_list)
```

Результат:

```
[2, 456, 45.3, None]
```

result_list.index(x[, start[, end]]). Метод возвращает положение первого индекса, со значением **x**. Также можно указать границы поиска **start** и **end**. Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# index
print(result_list.index(None))
```

Результат:

```
4
```

result_list.count(x). Метод возвращает положение первого индекса, со значением **x**. Также можно указать границы поиска **start** и **end**.

Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# count
print(result_list.count(2))
```

Результат:

```
1
```

result_list.reverse(). Перестраивает элементы списка в обратном порядке. Внимание: Данный метод модифицирует исходный объект на месте, возвращая при этом **None**. Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# reverse
result_list.reverse()
print(result_list)
```

Результат:

```
[None, 45.3, 456, 'text', 2]
```

result_list.copy(). Возвращает копию списка. Внимание: Возвращаемая копия является поверхностной (без рекурсивного копирования вложенных элементов). Действие метода эквивалентно выражению my_list[:]. Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# copy
copy_list = result_list.copy()
print(copy_list)
```

Результат:

```
[2, 'text', 456, 45.3, None]
```

result_list.clear(). Удаляет из списка все имеющиеся в нём значения. Действие метода эквивалентно выражению del my_list[:]. Пример:

```
result_list = [2, 'text', 456, 45.3, None]

# clear
result_list.clear()
print(result_list)
```

Результат:

```
[]
```

На примере списков рассмотрим использование Python-операторов **is** и **in**. Пример:

```
my_list = [10, 20, 30]
print((40 or 50) in my_list)
```

Результат:

```
False
```

Здесь используются возможности операторов **or** и **in**. Проверяется, входит ли хотя бы одно из указанных в скобках чисел в исходный список.

Пример:

```
list_1 = [30, 'string', None, False]
list_2 = [30, 'string', None, False]

print(list_1 is list_2)

list_2 = list_1

print(list_1 is list_2)
```

Результат:

```
False
True
```

В примере используется оператор идентичности (**is**). В первом случае результат — значение **False**, так как переменные **list_1** и **list_2** ссылаются на разные объекты. Во втором случае получается значение **True**, так как ссылка слева (**list_2**) указывает на тот же объект, что и ссылка справа (**list_1**).

2.4 Тип данных: кортеж

Кортеж представляет собой аналогичную списку структуру с одним отличием. Кортеж – неизменяемая структура. Самый простой способ создать кортеж – применить функцию **tuple()** к итерируемому объекту. Пример:

```
print(tuple('обычная строка'))
```

Результат:

```
('о', 'б', 'ы', 'ч', 'н', 'а', 'я', ' ', 'с', 'т', 'р', 'о', 'к',
'а')
```

Преимущества кортежей:

– защищают от неверных действий пользователя. Кортеж – неизменяемый список, защищён от случайных и намеренных изменений;

– меньший размер по сравнению со списками.

Список:

```
my_l = [4, 234, 45.8, "text", "word", "el", True, None]
print(my_l.__sizeof__())
```

Результат:

```
104
```

Кортеж:

```
my_t = (4, 234, 45.8, "text", "word", "el", True, None)
print(my_t.__sizeof__())
```

Результат:

```
88
```

В этих примерах сравниваются список и кортеж с одинаковыми данными. Но, в итоге, кортеж – более экономичная структура хранения данных. Список занимает 104 байта, а кортеж – 88.

Кортежи, как коллекции, поддерживают те же операции, что и списки. Операции не должны изменять саму коллекцию (например, **index()**, **count()**).

2.5 Тип данных: множество

Это контейнер с неповторяющимися элементами, расположенными в случайном порядке. Множество, создаваемое с помощью функции **set()**, представляет собой изменяемый тип данных, **frozenset()** – неизменяемый.

Пример:

```
perem_1 = set('abrakadabra')
perem_2 = frozenset('abrakadabra')
print(perem_1)
print(perem_2)
perem_1.add('!')
print(perem_1)
perem_2.add('!')
print(perem_2)
```

Результат:

```
{'a', 'k', 'b', 'd', 'r'}
frozenset({'a', 'k', 'b', 'd', 'r'})
{'k', '!', 'r', 'b', 'a', 'd'}
Traceback (most recent call last):
** IDLE Internal Exception:
  File "/usr/lib/python3.5/idlelib/run.py", line 351, in runcode
    exec(code, self.locals)
  File "run.py", line 7, in <module>
    perem_2.add('!')
AttributeError: 'frozenset' object has no attribute 'add'
```

Далее будут рассмотрены методы работы с изменяемыми множествами. Зададим множество. Пример:

```
my_set = {400, None, "text", True}
print(my_set)
```

Результат:

```
{400, True, None, 'text'}
```

set.add(x). Добавляет элемент в множество. Пример:

```
my_set = {400, None, "text", True}

# add
my_set.add("another_el")
print(my_set)
```

Результат:

```
{True, 'another_el', 'text', 400, None}
```

set.remove(x). Удаляет элемент из множества. KeyError, если такого элемента не существует. Пример:

```
my_set = {400, None, "text", True}

# remove
my_set.remove("text")
print(my_set)
```

Результат:

```
{400, True, None}
```

set.discard(x). Удаляет элемент, если он находится в множестве. Пример:

```
my_set = {400, None, "text", True}

# discard
my_set.discard(400)
print(my_set)
```

Результат:

```
{True, 'text', None}
```

set.pop(). Удаляет первый элемент из множества. Так как множества не упорядочены, нельзя точно сказать, какой элемент будет первым. Пример:

```
my_set = {400, None, "text", True}

```

```
# pop
my_set.pop()
print(my_set)
```

Результат:

```
{True, 'text', None}
```

set.copy(). Копия множества. Пример:

```
my_set = {400, None, "text", True}

# copy
print(my_set.copy())
```

Результат:

```
{400, 'text', None, True}
```

set.clear(). Очистка множества. Пример:

```
my_set = {400, None, "text", True}

# clear
my_set.clear()
print(my_set)
```

Результат:

```
set()
```

Изменяемые множества (**set()**) и неизменяемые (**frozenset()**) – аналогия списков и кортежей.

Пример:

```
my_s = set('abrakadabra')
my_fs = frozenset('abrakadabra')
print(my_s == my_fs)
```

Результат:

```
True
```

Пример:

```
# вычитание
my_s = set('kadabra')
print(my_s)
my_fs = frozenset('abra')
print(my_fs)
print(my_s - my_fs)
```

Результат:

```
{'r', 'a', 'b', 'd', 'k'}
frozenset({'r', 'a', 'b'})
{'k', 'd'}
```

Пример:

```
# объединение
my_s = set('kadabra')
print(my_s)
my_fs = frozenset('abra')
print(my_fs)
print(my_s | my_fs)
```

Результат:

```
{'b', 'r', 'a', 'd', 'k'}
frozenset({'b', 'a', 'r'})
{'b', 'd', 'r', 'k', 'a'}
```

2.6 Тип данных: словарь

Словарь – неупорядоченный набор произвольных объектов с доступом по ключу. Один из вариантов создания словаря – с помощью функции **dict()**. Пример:

```
my_dict = dict(key_1='val_1', key_2='val_2')
print(my_dict)
```

Результат:

```
{'key_1': 'val_1', 'key_2': 'val_2'}
```

dict.keys(). Возвращает ключи в словаре. Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# keys
print(my_dict.keys())
```

Результат:

```
dict_keys(['key_1', 2, 'key_3', 4])
```

dict.values(). Возвращает значения в словаре. Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# values
print(my_dict.values())
```

Результат:

```
dict_values([500, 400, True, None])
```

dict.items(). Возвращает пары (ключ, значение). Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# items
print(my_dict.items())
```

Результат:

```
dict_items([('key_1', 500), (2, 400), ('key_3', True), (4,
None)])
```

dict.get(key[, default]). Возвращает значение ключа, но если его нет, не бросает исключение, а возвращает **default** (по умолчанию **None**). Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# get
print(my_dict.get(2))
```

Результат:

```
400
```

dict.popitem(). Удаляет и возвращает пару (ключ, значение). Если словарь пуст, бросает исключение **KeyError**. Помните, что словари неупорядочены. Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# popitem
print(my_dict.popitem())
print(my_dict.popitem())
print(my_dict.popitem())
print(my_dict.popitem())
```

Результат:

```
(4, None)
('key_3', True)
(2, 400)
('key_1', 500)
```

dict.setdefault(key[, default]). Возвращает значение ключа, но если его нет, не бросает исключение, а создает ключ со значением **default** (по умолчанию **None**). Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# setdefault
print(my_dict.setdefault(5))
print(my_dict.items())
```

Результат:

```
None
dict_items([('key_1', 500), (2, 400), ('key_3', True), (4, None),
(5, None)])
```

dict.pop(key[, default]). Удаляет ключ и возвращает значение. Если ключа нет, возвращает **default** (по умолчанию бросает исключение). Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# pop
print(my_dict.pop(2))
print(my_dict.items())
```

Результат:

```
400
dict_items([('key_1', 500), ('key_3', True), (4, None)])
```

dict.update([other]). Обновляет словарь, добавляя пары (ключ, значение) из **other**. Существующие ключи перезаписываются. Возвращает **None** (не новый словарь!). Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# update
my_dict.update({8: 8, 9: 9, 10: 10})
print(my_dict.items())
```

Результат:

```
dict_items([('key_1', 500), (2, 400), ('key_3', True), (4, None),
(8, 8), (9, 9), (10, 10)])
```

dict.copy(). Возвращает копию словаря. Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}
# copy
print(my_dict.copy())
```

Результат:

```
{'key_1': 500, 2: 400, 'key_3': True, 4: None}
```

dict.clear(). Очищает словарь. Пример:

```
my_dict = {"key_1": 500, 2: 400, "key_3": True, 4: None}

# clear
my_dict.clear()
print(my_dict.items())
```

Результат:

```
dict_items([])
```

2.7 Тип данных: bool

Логический тип, применяется в представлениях истинности.

Пример:

```
print(True)
print(bool(20))
print(bool('text'))

print(False)
print(bool(0))
print(bool(''))
print(bool())
```

Результат:

```
True
True
True
False
False
False
False
```

Функция **bool()** позволяет привести любое значение к логическому типу, если это значение может быть интерпретировано в качестве логического типа.

2.8 Тип данных: bytes и bytearray

Байты – единица хранения информации (текстовой, графической, звуковой). Байтовое представление похоже на обычное строковое, но с рядом отличий. Пример:

```
print(b'text')
print('текст'.encode('utf-8'))
print(bytes('text', encoding = 'utf-8'))
print(bytes([10, 20, 30, 40]))
```

Результат:

```
b'text'
b'\xd1\x82\xd0\xb5\xd0\xba\xd1\x81\xd1\x82'
b'text'
b'\n\x14\x1e('
```

Тип данных **bytearray** представляет собой изменяемый массив байтов. Пример:

```
my_var = bytearray(b"some text")
print(my_var)
print(my_var[0])

#my_var[0] = b'h' -> TypeError: an integer is required
my_var[0] = 105
print(my_var)

my_var = bytearray(b"some text")
for i in range(len(my_var)):
     my_var[i] += i

print(my_var)
```

Результат:

```
bytearray(b'some text')
115
bytearray(b'iome text')
bytearray(b'spoh$yk\x7f|')
```

2.9 Тип данных: NoneType

Значение **None** переменной сигнализирует о присвоении пустого значения этой переменной. Оно обозначает «здесь нет значения». Присвоение переменной такого значения – один из вариантов её сброса в пустое состояние. Python – язык объектно-ориентированный. **None** также принадлежит к объектам и обладает своим типом.

```
print(type(None))
```

Результат:

```
<class 'NoneType'>
```

Рассмотрим ещё один пример:

```
my_dict = {'name': 'Ivan', 'surname': 'Ivanov', 'age': 40,
'position': None}
for el in my_dict:
    if my_dict[el] == None:
        print(f"Для сотрудника пока не определён параметр: {el}")
```

Результат:

```
Для сотрудника пока не определён параметр: position
```

Здесь выполняется перебор ключей словаря и проверка, есть ли в словаре значения типа None.

2.10 Тип данных: исключение

Exceptions представляют собой ещё один тип данных и предназначены для вывода сообщений об ошибках. Пример:

```
print(500 / 0)
```

Результат:

```
Traceback (most recent call last):
  File "my_file.py", line 1, in <module>
    print(500 / 0)
ZeroDivisionError: division by zero
```

В этом случае интерпретатор вывел информацию о наличии исключения (**ZeroDivisionError**), связанного с делением на 0 (division by zero). Это только один из типов исключений. В Python предусмотрены и другие, которые будут рассматриваться далее.

2.11 О цикле for in для обхода последовательностей

В Python списки, кортежи, строки относятся к последовательностям. Для выполнения однотипных операций с каждым элементом последовательностей в Python применяются циклы **for**. Эта функция отвечает за генерацию набора чисел в пределах указанного диапазона. Общий синтаксис:

```
for [переменная-итератор] in [последовательность]:
    [действия, выполняемые для каждой переменной]
```

Пример:

```
for el in "my_string":
    print(el)
```

Результат:

```
m
y
_
```

```
s
t
r
i
n
g
```

В этом примере **el** – переменная-итератор, последовательно принимающая значения – элементы строки.

Кортежи относятся к неизменяемым последовательностям, но допускают перебор элементов и выполнение операций с ними. Пример:

```
my_tuple = (1, 2, 3, 4, 5)
my_list = []
for el in my_tuple:
    my_list.append(el * 2)
print(my_list)
```

Результат:

```
[2, 4, 6, 8, 10]
```

Проверим работу цикла **for** на примере списка. Пример:

```
orig_list = [1, 2, 3, 4, 5]
new_list = []
for el in orig_list:
    new_list.append(el / 2)
print(new_list)
```

Результат:

```
[0.5, 1.0, 1.5, 2.0, 2.5]
```

И на примере множества:

```
orig_set = {1, 2, 3, 4, 5}
new_set = set()
for el in orig_set:
    new_set.add(el / 2)
print(new_set)
```

Результат:

```
{0.5, 1.0, 2.0, 2.5, 1.5}
```

Возможности цикла **for** применяются и к словарям. Пример:

```
my_dict = {'title': 'Samsung Galaxy', 'price': 20000, 'country':
'China', 'year': '2016'}
for key, value in my_dict.items():
    print(f"{key} - {value}")
```

Результат:

```
title - Samsung Galaxy
price - 20000
country - China
year - 2016
```

2.12 Понятие тернарного оператора

Понятие тернарного оператора в Python очень близко к понятию условного выражения. Тернарные операторы позволяют вернуть некоторый результат в зависимости от истинности или ложности некоторого условия. Синтаксис тернарного оператора:

```
condition_if_true if condition else condition_if_false
```

Пример:

```
is_checked = True
mode = "checked" if is_checked else "not checked"
print(mode)
```

Результат:

```
checked
```

Применение представленного подхода позволяет выполнить быструю проверку условия вместо использования нескольких ветвей с **if**. Код получается более компактным и читабельным.

Есть и другой вариант использования этого подхода (с кортежами). Синтаксис:

```
(if_check_is_false, if_check_is_true)[param_to_check]
```

Пример:

```
checked = True
personality = ("проверено", "не проверено")[checked]
print(personality)
```

Результат:

```
не проверено
```

Это работоспособный механизм в Python, поскольку значение **True** соответствует единице, а **False** – нулю. Кроме кортежей допускается использование списков.

В Python также предусмотрена возможность использования более лаконичной версии тернарного оператора. Пример:

```
print(True or "Some")
print(False or "Some")
```

Результат:

```
True
Some
```

Этот механизм удобно использовать, когда требуется проверить возвращаемое функцией значение. Пример:

```
func_return = None
message = func_return or "Функция ничего не возвращает"
print(message)
```

Результат:

```
Функция ничего не возвращает
```

2.13 Оператор is

Проверяет тождественность (идентичность) двух объектов в памяти. Возвращает значение **True** (истина), если переменные ссылаются на один и тот же объект.

Пример:

```
a = 20
b = 20
```

```
if a is b:
    print("Переменные идентичны")
else:
    print("Переменные не идентичны")
```

Результат:

```
Переменные идентичны
```

Важная особенность использования оператора **is** заключается в том, что он не идентичен оператору ==.

== – проверка равенства значений двух объектов.

is – проверка идентичности объектов, то есть проверка того, что переменные указывают на один и тот же объект в памяти. Пример:

```
obj_1 = [10, 20, 30, 40]
obj_2 = obj_1
print(obj_1 == obj_2)
print(obj_1 is obj_2)

obj_2 = obj_1[:] # переменная obj_2 ссылается на копию obj_1
print(obj_1 == obj_2)
print(obj_1 is obj_2)
print(obj_1 is not obj_2)
```

Результат:

```
True
True
True
False
True
```

Для проверки соответствия объекта типу **NoneType** предпочтительно использовать оператор **is**. Пример:

```
obj_1 = None
print(obj_1 is None)
```

Результат:

```
True
```

3 Полезные советы для работы в Python

В завершение урока познакомимся с набором интересных приёмов, которые пригодятся вам на практике.

3.1 Объединение списков без цикла

Явный вариант решения задачи объединения списков разной длины предполагает перебор элементов в цикле. Но возможно и более лаконичное решение через функцию **sum()**. Пример:

```
my_list = [[10, 20, 30], [40, 50], [60], [70, 80, 90]]
print(sum(my_list, []))
```

Результат:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

3.2 Поиск уникальных элементов в списке

Это очень популярный трюк, предполагающий трансформацию списка во множество и поиск уникальных элементов. Пример:

```
my_list = [10, 10, 3, 4, 5, 9, 30, 30]
print(list(set(my_list)))
```

Результат:

```
[3, 4, 5, 9, 10, 30]
```

3.3 Обмен значениями через кортежи

Позволяет выполнять обмен значения без создания дополнительной переменной. Трюк допустим для любого числа переменных. Пример:

```
var_1, var_2 = 20, 30
print(var_1, var_2)
var_1, var_2 = var_2, var_1
print(var_1, var_2)
```

Результат:

```
20 30
30 20
```

Правая часть выражения может представлять собой любой итерируемый объект. Главное, чтобы число элементов в левой и правой частях совпадало.

3.4 Вывод значения несуществующего ключа в словаре

Если попытаться обратиться к несуществующему ключу словаря, возникнет исключение. Пример:

```
my_dict = {'k_1': 20, 'k_2': True, 'k_3': 'text'}
print(my_dict['k_4'])
```

Результат:

```
KeyError: 'k_4'
```

Чтобы избежать такую ситуацию, можно воспользоваться методом **get()**. Пример:

```
my_dict = {'k_1': 20, 'k_2': True, 'k_3': 'text'}
print(my_dict.get('k_4'))
```

Результат:

```
None
```

3.5 Поиск самых часто встречающихся элементов списка

Искать самый часто встречающийся элемент можно, используя встроенную функцию **max()**. Она ищет наибольшее значение не только для итерируемого объекта, но и для результатов применения к этому объекту функции. Можно преобразовать список во множество и применить метод **count** для определения количества вхождений элемента в итерируемый объект. Пример:

```
my_list = [10, 20, 20, 20, 30, 50, 70, 30]
print(max(set(my_list), key=my_list.count))
```

Результат:

```
20
```

3.6 Распаковка последовательностей при неизвестном количестве элементов

В Python оператор * соответствует операции распаковки последовательности. Переменная с этим параметром связывается с частью

списка. Она содержит все не присвоенные элементы, соответствующие текущей позиции. Пример:

```
my_list = [20, 30, 40, 50]
*el_1, el_2, el_3 = my_list
print(el_1, el_2, el_3)
el_1, *el_2, el_3 = my_list
print(el_1, el_2, el_3)
el_1, el_2, *el_3 = my_list
print(el_1, el_2, el_3)
el_1, el_2, el_3, *el_4 = my_list
print(el_1, el_2, el_3, el_4)
el_1, el_2, el_3, el_4, *el_5 = my_list
print(el_1, el_2, el_3, el_4, el_5)
```

Результат:

```
[20, 30] 40 50
20 [30, 40] 50
20 30 [40, 50]
20 30 40 [50]
20 30 40 50 []
```

3.7 Вывод с помощью функции print() без перевода строки

По умолчанию функция **print()** добавляет символ перевода строки, который можно отменить, добавив в функцию параметр **end** со значением пустой строки. Пример:

```
for el in ["ab", "ra", "kada", "bra"]:
    print(el, end='')
```

Результат:

```
abrakadabra
```

3.8 Сортировка словаря по значениям

По умолчанию элементы словаря сортируются по наименованиям ключей. Пример:

```
my_dict = {'python': 1991, 'java': 1995, 'c++': 1983}
print(sorted(my_dict))
```

Результат:

```
['c++', 'java', 'python']
```

Но можно реализовать сортировку по значениям элементов. Пример:

```
my_dict = {'python': 1991, 'java': 1995, 'c++': 1983}
print(sorted(my_dict, key=my_dict.get))
```

Результат:

```
['c++', 'python', 'java']
```

3.9 Нумерованные списки

Для реализации нумерованного списка можно воспользоваться функцией **enumerate()**. Пример:

```
for ind, el in enumerate(['ноль', 'один', 'два', 'три']):
    print(ind, el)
```

Результат:

```
0 ноль
1 один
2 два
3 три
```

Пример:

```
for ind, el in enumerate(['один', 'два', 'три'], 1):
    print(ind, el)
```

Результат:

```
1 один
2 два
3 три
```

3.10 Транспонирование матрицы

Под транспонированием понимается замена местами строк и столбцов матрицы (двумерного массива). Для этого можно воспользоваться функцией zip(). Пример:

```
old_list = [('a', 'b'), ('c', 'd'), ('e', 'f')]
new_list = zip(*old_list)
```

```
print(list(new_list))
```

Результат:

```
[('a', 'c', 'e'), ('b', 'd', 'f')]
```

4 Частые ошибки начинающих разработчиков. Как их исправить

Проблема 1. TypeError: Can't convert 'int' object to str implicitly

Пример:

```
my_var = input("Введите число: ") + 5
print(my_var)
# Ошибка:
# TypeError: can only concatenate str (not "int") to str
```

Причина: недопустимо применять оператор сложения к строке и числу.

Решение: нужно выполнить преобразование строки к числу, применив функцию **int()**. Обратите внимание, что функция **input()** всегда возвращает строку. Пример:

```
my_var = int(input("Введите число: ")) + 5
print(my_var)
```

Проблема 2. SyntaxError: invalid syntax

Пример:

```
msg = True
if msg == True:
    print("Приветственное сообщение")
# Ошибка:
# SyntaxError: invalid syntax
```

Причина: забыто двоеточие.

Решение:

```
msg = True
if msg == True:
    print("Приветственное сообщение")
```

Проблема 3. SyntaxError: invalid syntax

Пример:

```
msg = True
if msg = True:
    print("Приветственное сообщение")
# Ошибка:
# SyntaxError: invalid syntax
```

Причина: забыт знак равенства. Решение:

```
msg = True
if msg == True:
    print("Приветственное сообщение")
```

Проблема 4. ***NameError: name 'my_var' is not defined***

Пример:

```
print(my_var)
# Ошибка:
# NameError: name 'my_var' is not defined
```

Причина: переменной **my_var** нет. Возможно, переменная есть, но неправильно указано её имя, или программист забыл инициализировать переменную. Решение:

```
my_var = "какое-то значение переменной"
print(my_var)
```

Проблема 5. ***IndentationError: expected an indented block***

Пример:

```
my_var = True
if my_var == True:
print("Все верно")
# Ошибка:
# IndentationError: expected an indented block
```

Причина: требуется отступ. Решение:

```
my_var = True
if my_var == True:
    print("Все верно")
```

Проблема 6. Inconsistent use of tabs and spaces in indentation

Пример:

```
my_var = True
if my_var == True:
    print("Все верно")
    print("Работа программы завершена")
# Ошибка:
```

```
# Inconsistent use of tabs and spaces in indentation
```

Причина: использование пробелов и табуляций в отступах по одной программе (файлу-модулю).

Решение: Привести все отступы к единообразию (везде использовать пробелы).

Проблема 7. UnboundLocalError: local variable 'my_var' referenced before assignment

Пример:

```
def my_func():
    my_var += 1
    print(my_var)

my_var = 10
my_func()
# Ошибка:
# UnboundLocalError: local variable 'my_var' referenced before
assignment
```

Причина: попытка обратиться к локальной переменной, которая ещё не создана. Решение:

```
def my_func(my_var):
    my_var += 1
    print(my_var)

my_var = 10
my_func(my_var)
```

Памятка

Встроенные типы данных (часть):

Название типа	Описание	Примечание
`int`	Это функция, возвращающая целое число в десятичной системе счисления. Пример: 2, 4, 8, -10, -2	Подробнее будет рассмотрено далее
`float`	Это функция, возвращающая число с плавающей запятой. Пример: 2.6, -5.2	Подробнее будет рассмотрено далее
`str`	Это функция, возвращающая строку (неизменяемую последовательность символов)	Подробнее будет рассмотрено далее
`bool`	Это функция, возвращающая булево значение (True или False) для объекта	Подробнее будет рассмотрено далее
`list`	Функция, возвращающая изменяемую упорядоченную коллекцию объектов произвольных типов. Пример: [2, 2.4, “Hello”]	Подробнее будет рассмотрено далее
`tuple`	Функция, возвращающая неизменяемую упорядоченную коллекцию объектов произвольных типов. Кортеж. Пример:(2, 2.4, “Hello”)	Подробнее будет рассмотрено далее
`dict`	Функция, возвращающая неупорядоченную коллекцию произвольных объектов с доступом по ключу. Пример: {“name”: “Вася”, “age”: 10}	Подробнее будет рассмотрено далее

Арифметические операторы в Python

Оператор	Описание	Примеры
+	Сложение	`print(398 + 20) -> 418`
-	Вычитание	`print(200 - 50) -> 150`
*	Умножение	`print(34 * 7) -> 238`

/	Деление	`print(36 / 6) -> 6.0`
		`print(36 / 5) -> 7.2`
		`print(round(36 / 7, 2)) -> 5.14`
		`print(round(-36 / -7, 3)) -> 5.143`

//	Целочисленное деление	`print(36 // 6) -> 6`
		`print(36 // 5) -> 7`
		`print(-9 // 4) -> -3`
		`print(5 // -2) -> -3`

%	Остаток от деления	`print(36 % 6) -> 0` `print(36 % 5) -> 1`
**	Возведение в степень	`print(2 ** 16) -> 65536`

Логические операторы в Python

Оператор	Описание	Примеры
>	Больше	`print(40 > 40) -> False`
<	Меньше	`print(3 < 9) -> True`
==	Равно	`print(10 == 10) -> True`
!=	Не равно	`print(2 != 2) -> False`
>=	Больше или равно	`print(40 >= 1) -> True`
<=	Меньше или равно	`print(3 <= 1) -> False`

and	Логическое «И». Возвращает значение «Истина», если оба операнда имеют значение «Истина»	`print(True and True) -> True`
		`print(True and False) -> False`
		`print(False and True) -> False`
		`print(False and False) -> False`

or	Логическое «ИЛИ». Возвращает значение «Истина», если хотя бы один из операндов имеет значение «Истина»	`print(True or True) -> True`
		`print(True or False) -> True`
		`print(False or True) -> True`
		`print(False or False) -> False`

not	Логическое «НЕ». Изменяет логическое значение операнда на противоположное	`print(not True) -> False` `print(not False) -> True`
in	Оператор проверки принадлежности. Возвращает значение «Истина», если элемент присутствует в последовательности	`print(10 in [10, 20, 30]) -> True`

is	Оператор проверки тождественности. Возвращает значение «Истина», если операнды ссылаются на один объект	`x = 3`
		`y = 3`
		`print(x is y) -> True`

Форматирование через оператор %

Подстановка	Тип данных
"%s"	Строка
"%d"	Десятичное число
"%f"	Число с плавающей точкой
"%o"	Число в восьмеричной системе
"%x"	Число в шестнадцатеричной системе

Основные методы работы со списками

Метод	Назначение
result_list.append(el)	Добавить элемент el в конец списка result_list
result_list.extend(my_list)	Расширить список result_list — добавить в конец элементы списка my_list
result_list.insert(pos, el)	Разместить на позиции pos (индекс элемента списка) элемент el
result_list.remove(el)	Удалить из списка первый элемент со значением el
result_list.pop(pos)	Удалить элемент с индексом pos
result_list.index(el)	Получить позицию (индекс) первого элемента со значением el
result_list.count(el)	Возвращает количество элементов списка со значением el
result_list.sort([key=функция])	Выполнить сортировку списка на основе указанной функции
result_list.reverse()	Выполнить реверс списка (развернуть список)
result_list.copy()	Вернуть копию списка
result_list.clear()	Очистить список

Основные методы работы с изменяемыми множествами

Метод	Назначение
.add(el)	Добавить элемент в множество
.remove(el)	Удалить элемент из множества. Если элемент отсутствует – ошибка KeyError
.discard(el)	Удалить элемент из множества
.pop()	Удалить первый элемент из множества. Множества не упорядочены, поэтому первый элемент множества заранее не определён
.copy()	Создать копию множества
.clear()	Очистить множество

Основные методы работы со словарями

Метод	Назначение
.keys()	Возвращает список ключей словаря
.values()	Возвращает список значений
.items()	Возвращает список кортежей (ключ, значение)
.get(key)	Возвращает значение, соответствующее ключу key. Если ключ отсутствует, возвращает значение None
.popitem()	Удаляет элемент словаря и возвращает пару (ключ, значение). Если элементы отсутствуют, возникает исключение KeyError
.setdefault(key)	Возвращает значение, соответствующее ключу. Если ключ отсутствует, создаётся элемент с указанным ключом и значением None
.pop(key)	Удаляет ключ и возвращает значение, соответствующее ключу
.update(new_dict)	Добавляет пары (ключ, значение) в текущий словарь из словаря new_dict. Имеющиеся ключи перезаписываются
.copy()	Возвращает копию словаря
.clear()	Очищает словарь

Описание ключевых слов:

Название	Описание
`False`	Значение «Ложь»
`None`	«Не определён», пустой объект
`True`	Значение «Истина»
`and`	Логическое «И»
`as`	Определение псевдонима для объекта
`assert`	Генерация исключения, если условие ложно
`async`	Обозначение функций как сопрограмм для использования цикла событий
`await`	
`break`	Выход из цикла
`class`	Пользовательский тип (класс), содержащий атрибуты и методы
`continue`	Переход на очередную итерацию цикла
`def`	Определение функции
`del`	Удаление объекта
`elif`	Ещё, иначе, если
`else`	Иначе, если
`except`	Перехват исключения
`finally`	Выполнение инструкций, независимо были ли исключение или нет
`for`	Начало цикла перебора элементов набора
`from`	Указание пакета или модуля, из которого выполняется импорт
`global`	Значение переменной, присвоенное ей внутри функции, становится доступным вне этой функции
`if`	Если
`import`	Импорт модуля

`in`	Проверка на вхождение
`is`	Проверка, ссылаются ли два объекта на одно и то же место в памяти
`lambda`	Определение анонимной функции
`nonlocal`	Значение переменной, присвоенное ей внутри функции, становится доступным в объемлющей функции
`not`	Логическое «НЕ»
`or`	Логическое «ИЛИ»
`pass`	Заглушка для функции или класса. Используется, когда код класса и функции ещё не определён
`raise`	Генерация исключения
`return`	Вернуть результат
`try`	Выполнить инструкции с перехватом исключения
`while`	Начало цикла «ПОКА»
`with`	Использование менеджера контекста
`yield`	Определение функции-генератора