[1 Основы Python 1](#_Toc154431421)

[Введение в написание программ 1](#_Toc154431422)

[Переменные и типы данных 3](#_Toc154431423)

[Арифметические операции с числами 7](#_Toc154431424)

[Поразрядные операции с числами 11](#_Toc154431425)

[Условные выражения 15](#_Toc154431426)

[Условная конструкция if 18](#_Toc154431427)

[Циклы 23](#_Toc154431428)

[Функции 30](#_Toc154431429)

[Параметры функции 33](#_Toc154431430)

[Оператор return и возвращение результата из функции 37](#_Toc154431431)

[Функция как тип, параметр и результат другой функции 39](#_Toc154431432)

[Лямбда-выражения 42](#_Toc154431433)

[Преобразование типов 44](#_Toc154431434)

[Область видимости переменных 46](#_Toc154431435)

[Замыкания 50](#_Toc154431436)

[Декораторы 52](#_Toc154431437)

1 Основы Python

**Введение в написание программ**

Программа на языке Python состоит из набора инструкций. Каждая инструкция помещается на новую строку. Например:

1. print(2 + 3)
2. print("Hello")

Большую роль в Python играют отступы. Неправильно поставленный отступ фактически является ошибкой. Например, в следующем случае мы получим ошибку, хотя код будет практически аналогичен приведенному выше:

|  |  |
| --- | --- |
| 1  2 | print(2 + 3)  print("Hello") |

Поэтому стоит помещать новые инструкции сначала строки. В этом одно из важных отличий пайтона от других языков программирования, как C# или Java.

Однако стоит учитывать, что некоторые конструкции языка могут состоять из нескольких строк. Например, условная конструкция **if**:

|  |  |
| --- | --- |
| 1  2 | if 1 < 2:  print("Hello") |

В данном случае если 1 меньше 2, то выводится строка "Hello". И здесь уже должен быть отступ, так как инструкция print("Hello") используется не сама по себе, а как часть условной конструкции if. Причем отступ, согласно [руководству по оформлению кода](https://www.python.org/dev/peps/pep-0008/), желательно делать из такого количество пробелов, которое кратно 4 (то есть 4, 8, 16 и т.д.) Хотя если отступов будет не 4, а 5, то программа также будет работать.

Таких конструкций не так много, поэтому особой путаницы по поводу где надо, а где не надо ставить пробелы, не должно возникнуть.

### Регистрозависимость

Python - регистрозависимый язык, поэтому выражения print и Print или PRINT представляют разные выражения. И если вместо метода print для вывода на консоль мы попробуем использовать метод Print:

|  |  |
| --- | --- |
| 1 | Print("Hello World") |

то у нас ничего не получится.

### Комментарии

Для отметки, что делает тот или иной участок кода, применяются комментарии. При трансляции и выполнении программы интерпретатор игнорирует комментарии, поэтому они не оказывают никакого влияния на работу программы. Комментарии в Python бывают блочные и строчные.

Строчные коментарии предваряются знаком решетки - **#**. Они могут располагаться на отдельной строке:

|  |  |
| --- | --- |
| 1  2  3 | # Вывод на консоль  # сообщения Hello World  print("Hello World") |

Любой набор символов после знака # представляет комментарий. То есть в примее выше первые две строки кода являются комментариями.

Также они могут располагаться на той же строке, что и инструкции языка, после выполняемых инструкций:

|  |  |
| --- | --- |
| 1 | print("Hello World")  # Вывод сообщения на консоль |

В блочных коментариях до и после текста комментария ставятся три одинарные кавычки: '''текст комментария'''. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | '''      Вывод на консоль      сообщения Hello World  '''  print("Hello World") |

### Основные функции

Python предоставляет ряд встроенных функций. Некоторые из них используются очень часто, особенно на начальных этапах изучения языка, поэтому рассмотрим их.

Основной функцией для вывода информации на консоль является функция **print()**. В качестве аргумента в эту функцию передается строка, которую мы хотим вывести:

|  |  |
| --- | --- |
| 1 | print("Hello Python") |

Если же нам необходимо вывести несколько значений на консоль, то мы можем передать их в функцию print через запятую:

|  |  |
| --- | --- |
| 1 | print("Full name:", "Tom", "Smith") |

В итоге все переданные значения склеятся через пробелы в одну строку:

Full name: Tom Smith

Если функция print отвечает за вывод, то функция **input** отвечает за ввод информации. В качестве необязательного параметра эта функция принимает приглашение к вводу и возвращает введенную строку, которую мы можем сохранить в переменную:

|  |  |
| --- | --- |
| 1  2 | name = input("Введите имя: ")  print("Привет", name) |

Консольный вывод:

Введите имя: Евгений

Привет Евгений

## Переменные и типы данных

### Переменные

Переменные предназначены для хранения данных. Название переменной в Python должно начинаться с алфавитного символа или со знака подчеркивания и может содержать алфавитно-цифровые символы и знак подчеркивания. И кроме того, название переменной не должно совпадать с названием ключевых слов языка Python. Ключевых слов не так много, их легко запомнить:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | False      await      else       import     pass  None       break      except     in         raise  True       class      finally    is         return  and        continue   for        lambda     try  as         def        from       nonlocal   while  assert     del        global     not        with  async      elif       if         or         yield |

Например, создадим переменную:

|  |  |
| --- | --- |
| 1 | name ="Tom" |

Здесь определена переменная name, которая хранит строку "Tom".

В пайтоне применяется два типа наименования переменных: **camel case** и **underscore notation**.

**Camel case** подразумевает, что каждое новое подслово в наименовании переменной начинается с большой буквы. Например:

|  |  |
| --- | --- |
| 1 | userName = Tom" |

**Underscore notation** подразумевает, что подслова в наименовании переменной разделяются знаком подчеркивания. Например:

|  |  |
| --- | --- |
| 1 | user\_name ="Tom" |

И также надо учитывать регистрозависимость, поэтому переменные name и Name будут представлять разные объекты.

|  |  |
| --- | --- |
| 1  2  3 | # две разные переменные  name = "Tom"  Name = "Tom" |

Определив переменную, мы можем использовать в программе. Например, попытаться вывести ее содержимое на консоль с помощью встроенной функции **print**:

|  |  |
| --- | --- |
| 1  2 | name = "Tom"  # определение переменной name  print(name)   # вывод значения переменной name на консоль |

Отличительной особенностью переменной является то, что мы можем менять ее значение в течение работы программы:

|  |  |
| --- | --- |
| 1  2  3  4 | name = "Tom"  # переменной name равна "Tom"  print(name)   # выводит: Tom  name = "Bob"  # меняем значение на "Bob"  print(name)   # выводит: Bob |

### Типы данных

Переменная хранит данные одного из типов данных. В Python существует множество различных типов данных. В данном случае рассмотрим только самые базовые типы: **bool**, **int**, **float**, **complex** и **str**.

### Логические значения

Тип **bool** представляет два логических значения: **True** (верно, истина) или **False** (неверно, ложь). Значение **True** служит для того, чтобы показать, что что-то истинно. Тогда как значение **False**, наоборот, показывает, что что-то ложно. Пример переменных данного типа:

|  |  |
| --- | --- |
| 1  2  3  4  5 | isMarried = False  print(isMarried)    # False    isAlive = True  print(isAlive)      # True |

### Целые числа

Тип **int** представляет целое число, например, 1, 4, 8, 50. Пример

|  |  |
| --- | --- |
| 1  2  3  4  5 | age = 21  print("Возраст:", age)    # Возраст: 21    count = 15  print("Количество:", count) # Количество: 15 |

По умолчанию стандартные числа расцениваются как числа в десятичной системе. Но Python также поддерживает числа в двоичной, восьмеричной и шестнадцатеричной системах.

Для указания, что число представляет двоичную систему, перед числом ставится префикс **0b**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | a = 0b11  b = 0b1011  c = 0b100001  print(a)    # 3 в десятичной системе  print(b)    # 11 в десятичной системе  print(c)    # 33 в десятичной системе |

Для указания, что число представляет восьмеричную систему, перед числом ставится префикс **0o**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | a = 0o7  b = 0o11  c = 0o17  print(a)    # 7 в десятичной системе  print(b)    # 9 в десятичной системе  print(c)    # 15 в десятичной системе |

Для указания, что число представляет шестнадцатеричную систему, перед числом ставится префикс **0x**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | a = 0x0A  b = 0xFF  c = 0xA1  print(a)    # 10 в десятичной системе  print(b)    # 255 в десятичной системе  print(c)    # 161 в десятичной системе |

Стоит отметить, что в какой-бы системе мы не передали число в функцию print для вывода на консоль, оно по умолчанию будет выводиться в десятичной системе.

### Дробные числа

Тип **float** представляет число с плавающей точкой, например, 1.2 или 34.76. В качесте разделителя целой и дробной частей используется точка.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | height = 1.68  pi = 3.14  weight = 68.  print(height)   # 1.68  print(pi)       # 3.14  print(weight)   # 68.0 |

Число с плавающей точкой можно определять в экспоненциальной записи:

|  |  |
| --- | --- |
| 1  2  3  4  5 | x = 3.9e3  print(x)  # 3900.0    x = 3.9e-3  print(x)  # 0.0039 |

Число float может иметь только 18 значимых символов. Так, в данном случае используются только два символа - 3.9. И если число слишком велико или слишком мало, то мы можем записывать число в подобной нотации, используя экспоненту. Число после экспоненты указывает степень числа 10, на которое надо умножить основное число - 3.9.

### Комплексные числа

Тип **complex** представляет комплексные числа в формате вещественная\_часть+мнимая\_часть**j** - после мнимой части указывается суффикс **j**

|  |  |
| --- | --- |
| 1  2 | complexNumber = 1+2j  print(complexNumber)   # (1+2j) |

### Строки

Тип **str** представляет строки. Строка представляет последовательность символов, заключенную в одинарные или двойные кавычки, например "hello" и 'hello'. В Python 3.x строки представляют набор символов в кодировке Unicode

|  |  |
| --- | --- |
| 1  2  3  4  5 | message = "Hello World!"  print(message)  # Hello World!    name = 'Tom'  print(name)  # Tom |

При этом, если строка имеет много символов, ее можно разбить на части и эти части разместить на разных строках кода. В этом случае вся строка заключается в круглые скобки, а ее отдельные части - в кавычки:

|  |  |
| --- | --- |
| 1  2  3 | text = ("Laudate omnes gentes laudate "          "Magnificat in secula ")  print(text) |

Если же мы хотим определить многострочный текст, то такой текст заключается в тройные двойные или одинарные кавычки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | '''  Это комментарий  '''  text = '''Laudate omnes gentes laudate  Magnificat in secula  Et anima mea laudate  Magnificat in secula  '''  print(text) |

При использовани тройных одинарных кавычек не стоит путать их с комментариями: если текст в тройных одинарных кавычках присваивается переменной, то это строка, а не комментарий.

#### Управляющие последовательности в строке

Строка может содержать ряд специальных символов - управляющих последовательностей. Некоторые из них:

* **\\**: позволяет добавить внутрь строки слеш
* **\'**: позволяет добавить внутрь строки одинарную кавычку
* **\"**: позволяет добавить внутрь строки двойную кавычку
* **\n**: осуществляет переход на новую строку
* **\t**: добавляет табуляцию (4 отступа)

Применим несколько последовательностей:

|  |  |
| --- | --- |
| 1  2 | text = "Message:\n\"Hello World\""  print(text) |

Консольный вывод программы:

Message:

"Hello World"

Хотя подобные последовательности могут нам помочь в некоторых делах, например, поместить в строку кавычку, сделать табуляцию, перенос на другую строку. Но они также могут и мешать. Например:

|  |  |
| --- | --- |
| 1  2 | path = "C:\python\name.txt"  print(path) |

Здесь переменная path содержит некоторый путь к файлу. Однако внутри строки встречаются символы "\n", которые будут интерпретированы как управляющая последовательность. Так, мы получим следующий консольный вывод:

C:\python

ame.txt

Чтобы избежать подобной ситуации, перед строкой ставится символ **r**

|  |  |
| --- | --- |
| 1  2 | path = r"C:\python\name.txt"  print(path) |

#### Вставка значений в строку

Python позволяет встравивать в строку значения других переменных. Для этого внутри строки переменные размещаются в фигурных скобках {}, а перед всей строкой ставится символ **f**:

|  |  |
| --- | --- |
| 1  2  3  4 | userName = "Tom"  userAge = 37  user = f"name: {userName}  age: {userAge}"  print(user)   # name: Tom  age: 37 |

В данном случае на место {userName} будет вставляться значение переменной userName. Аналогично на вместо {userAge} будет вставляться значение переменной userAge.

### Динамическая типизация

Python является языком с динамической типизацией. А это значит, что переменная не привязана жестко к определенному типу.

Тип переменной определяется исходя из значения, которое ей присвоено. Так, при присвоении строки в двойных или одинарных кавычках переменная имеет тип **str**. При присвоении целого числа Python автоматически определяет тип переменной как **int**. Чтобы определить переменную как объект float, ей присваивается дробное число, в котором разделителем целой и дробной части является точка.

При этом в процессе работы программы мы можем изменить тип переменной, присвоив ей значение другого типа:

|  |  |
| --- | --- |
| 1  2  3  4  5 | userId = "abc"  # тип str  print(userId)    userId = 234  # тип int  print(userId) |

С помощью встроенной функции **type()** динамически можно узнать текущий тип переменной:

|  |  |
| --- | --- |
| 1  2  3  4  5 | userId = "abc"      # тип str  print(type(userId)) # <class 'str'>    userId = 234        # тип int  print(type(userId)) # <class 'int'> |

## Арифметические операции с числами

Python поддерживает все распространенные арифметические операции:

* **+**

Сложение двух чисел:

|  |  |
| --- | --- |
| 1 | print(6 + 2)  # 8 |

* **-**

Вычитание двух чисел:

|  |  |
| --- | --- |
| 1 | print(6 - 2)  # 4 |

* **\***

Умножение двух чисел:

|  |  |
| --- | --- |
| 1 | print(6 \* 2)  # 12 |

* **/**

Деление двух чисел:

|  |  |
| --- | --- |
| 1 | print(6 / 2)  # 3.0 |

* **//**

Целочисленное деление двух чисел:

|  |  |
| --- | --- |
| 1  2 | print(7 / 2)  # 3.5  print(7 // 2)  # 3 |

Данная операция возвращает целочисленный результат деления, отбрасывая дробную часть

* **\*\***

Возведение в степень:

|  |  |
| --- | --- |
| 1 | print(6 \*\* 2)  # Возводим число 6 в степень 2. Результат - 36 |

* **%**

Получение остатка от деления:

|  |  |
| --- | --- |
| 1 | print(7 % 2)  # Получение остатка от деления числа 7 на 2. Результат - 1 |

В данном случае ближайшее число к 7, которое делится на 2 без остатка, это 6. Поэтому остаток от деления равен 7 - 6 = 1

При последовательном использовании нескольких арифметических операций их выполнение производится в соответствии с их приоритетом. В начале выполняются операции с большим приоритетом. Приоритеты операций в порядке убывания приведены в следующей таблице.

|  |  |
| --- | --- |
| Операции | Направление |
| \*\* | Справо налево |
| \* / // % | Слева направо |
| + - | Слева направо |

Пусть у нас выполняется следующее выражение:

|  |  |
| --- | --- |
| 1  2 | number = 3 + 4 \* 5 \*\* 2 + 7  print(number)  # 110 |

Здесь начале выполняется возведение в степень (5 \*\* 2) как операция с большим приоритетом, далее результат умножается на 4 (25 \* 4), затем происходит сложение (3 + 100) и далее опять идет сложение (103 + 7).

Чтобы переопределить порядок операций, можно использовать скобки:

|  |  |
| --- | --- |
| 1  2 | number = (3 + 4) \* (5 \*\* 2 + 7)  print(number)  # 224 |

Следует отметить, что в арифметических операциях могут принимать участие как целые, так и дробные числа. Если в одной операции участвует целое число (int) и число с плавающей точкой (float), то целое число приводится к типу float.

### Арифметические операции с присвоением

Ряд специальных операций позволяют использовать присвоить результат операции первому операнду:

* **+=**

Присвоение результата сложения

* **-=**

Присвоение результата вычитания

* **\*=**

Присвоение результата умножения

* **/=**

Присвоение результата от деления

* **//=**

Присвоение результата целочисленного деления

* **\*\*=**

Присвоение степени числа

* **%=**

Присвоение остатка от деления

Примеры операций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | number = 10  number += 5  print(number)  # 15    number -= 3  print(number)  # 12    number \*= 4  print(number)  # 48 |

### Округление и функция round

При операциях с числами типа float надо учитывать, что результат операций с ними может быть не совсем точным. Например:

|  |  |
| --- | --- |
| 1  2  3  4 | first\_number = 2.0001  second\_number = 5  third\_number = first\_number / second\_number  print(third\_number) # 0.40002000000000004 |

В данном случае мы ожидаем получить число 0.40002, однако в конце через ряд нулей появляется еще какая-то четверка. Или еще одно выражение:

|  |  |
| --- | --- |
| 1 | print(2.0001 + 0.1)  # 2.1001000000000003 |

В случае выше для округления результата мы можем использовать встроенную функцию **round()**:

|  |  |
| --- | --- |
| 1  2  3  4 | first\_number = 2.0001  second\_number = 0.1  third\_number = first\_number + second\_number  print(round(third\_number))  # 2 |

В функцию round() передается число, которое надо округлить. Если в функцию передается одно число, как в примере выше, то оно округляется до целого.

Функция round() также может принимать второе число, которое указывает, сколько знаков после запятой должно содержать получаемое число:

|  |  |
| --- | --- |
| 1  2  3  4 | first\_number = 2.0001  second\_number = 0.1  third\_number = first\_number + second\_number  print(round(third\_number, 4))  # 2.1001 |

В данном случае число third\_number округляется до 4 знаков после запятой.

Если в функцию передается только одно значение - только округляемое число, оно округляется то ближайшего целого

Примеры округлений:

|  |  |
| --- | --- |
| 1  2  3 | # округление до целого числа  print(round(2.49))  # 2 - округление до ближайшего целого 2  print(round(2.51))  # 3 |

Однако если округляемая часть равна одинаково удалена от двух целых чисел, то округление идет к ближайшему четному:

|  |  |
| --- | --- |
| 1  2 | print(round(2.5))   # 2 - ближайшее четное  print(round(3.5))   # 4 - ближайшее четное |

Округление производится до ближайшего кратного 10 в степени минус округляемая часть:

|  |  |
| --- | --- |
| 1  2  3  4  5 | # округление до двух знаков после запятой  print(round(2.554, 2))      # 2.55  print(round(2.5551, 2))      # 2.56  print(round(2.554999, 2))   # 2.55  print(round(2.499, 2))      # 2.5 |

Однако следует учитывать, что функция round() не идеальный инструмент. Например, выше при округление до целых чисел применяется правило, согласно которому, если округляемая часть одинаково удалена от двух значений, то округление производится до ближайшего четного значения. В Python в связи с тем, что десятичная часть числа не может быть точно представлена в виде числа float, то это может приводить к некоторым не совсем ожидаемым результатам. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | # округление до двух знаков после запятой  print(round(2.545, 2))   # 2.54  print(round(2.555, 2))   # 2.56 - округление до четного  print(round(2.565, 2))   # 2.56  print(round(2.575, 2))   # 2.58    print(round(2.655, 2))   # 2.65 - округление не до четного  print(round(2.665, 2))   # 2.67  print(round(2.675, 2))   # 2.67 |

## Поразрядные операции с числами

Особый класс операций представляют поразрядные операции. Они выполняются над отдельными разрядами чисел типа int. Но чтобы понять природу поразрядных операций, надо понимать что вообще представляет число в двоичном представление.

### Двоичное представление чисел

При двоичной системе каждый разряд числа может иметь только два значения - 0 и 1. Например, 0 в десятичной системе также будет равен 0 в двоичной системе, а 1 в десятичной системе будет соответствовать 1 в двоичной системе. Следующее число в десятичной системе - 2 в двоичной системе будет соответствовать 10. То есть, когда мы к 1 прибавляем 1, то результатом будет 10. И так далее.

Например, 5 в двоичном представлении 101 и имеет три разряда. Для вывода десятичного числа в двоичной системе можно применять спецификатор 0b:

|  |  |
| --- | --- |
| 1  2 | number = 5 # в двоичной форме 101  print(f"number = {number:0b}")  # number = 101 |

Без указания спецификатора функция print() выводит число в десятичной системе.

При этом Python позволяет сразу определять число в двоичной форме. Для этого число в двоичной форме указывается после префикса **0b**:

|  |  |
| --- | --- |
| 1  2  3 | number = 0b101  # определяем число в двоичной форме  print(f"number = {number:0b}")  # number = 101  print(f"number = {number}")  # number = 5 - в десятичной системе |

Еще несколько примеров сопоставления между двоичной и десятичной системами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | number1 = 1 # в двоичной системе 0b1  number2 = 2 # в двоичной системе 0b10  number3 = 3 # в двоичной системе 0b11  number4 = 4 # в двоичной системе 0b100  number5 = 5 # в двоичной системе 0b101  number6 = 6 # в двоичной системе 0b110 |

### Логические операции

Логические операции выполняются над отдельными разрядами числа. В Python есть следующие логические операции:

* **&**(логическое умножение)

Умножение производится поразрядно, и если у обоих операндов значения разрядов равно 1, то операция возвращает 1, иначе возвращается число 0. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | x1 = 2  # 010  y1 = 5  # 101  z1 = x1 & y1  print(f"z1 = {z1}")   # z1 = 0    x2 = 4  # 100  y2 = 5  # 101  z2 = x2 & y2  print(f"z2 = {z2}")   # z2 = 4  print(f"z2 = {z2:0b}")  # z2 = 100 |

В первом случае у нас два числа 2 и 5. 2 в двоичном виде представляет число 010, а 5 - 101. Поразрядно умножим числа (0\*1, 1\*0, 0\*1) и в итоге получим 000.

Во втором случае у нас вместо двойки число 4, у которого в первом разряде 1, так же как и у числа 5, поэтому в итоге получим (1\*1, 0\*0, 0 \*1) = 100, то есть число 4 в десятичном формате.

* **|** (логическое сложение)

Похоже на логическое умножение, операция также производится по двоичным разрядам, но теперь возвращается единица, если хотя бы у одного числа в данном разряде имеется единица. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | x1 = 2      # 010  y1 = 5      # 101  z1 = x1|y1  # 111    print(f"z1 = {z1}")     # z1 = 7  print(f"z1 = {z1:0b}")  # z1 = 111    x2 = 4          # 100  y2 = 5          # 101  z2 = x2 | y2    # 101  print(f"z2 = {z2}")     # z2 = 5  print(f"z2 = {z2:0b}")  # z2 = 101 |

* **^** (логическое исключающее ИЛИ)

Если значения текущего разряда у обоих чисел разные, то возвращается 1, иначе возвращается 0. Также эту операцию называют XOR. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | x = 9       #  1001  y = 5       #  0101  z = x ^ y   #  1100  print(f"z = {z}")       # z = 12  print(f"z = {z:0b}")   # z = 1100 |

Здесь число 9 в двоичной форме равно 1001. Число 5 равно 0101. Операция XOR дает следующий результат: 1^0, 0^1, 0^0, 1^1. Здесь мы видим, что первые два разряда чисел содержат разные значения, поэтому первые два разряда получат значение 1. А последние два разряда чисел содержат одинаковые значения, поэтому последние два разряда получат значение 0. Таким образом, мы получаем число 1100 или 12 в десятичной системе.

нередко данную операцию применяют для простого шифрования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | x = 45       # Значение, которое надо зашифровать - в двоичной форме 101101  key = 102    # Пусть это будет ключ - в двоичной форме 1100110    encrypt = x ^ key    # Результатом будет число 1001011 или 75  print(f"Зашифрованное число: {encrypt}")    decrypt = encrypt ^ key    # Результатом будет исходное число 45  print(f"Расшифрованное число: {decrypt}") |

Также можно применять эту операцию для обмена значений чисел:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | x = 9       #  1001  y = 5       #  0101  x = x ^ y  y = x ^ y  x = x ^ y    print(f"x = {x}")       # x = 5  print(f"y = {y}")       # y = 9 |

* **~**(инверсия)

Инвертирует число. Выражение ~x фактически аналогично -(x+1). Например:

|  |  |
| --- | --- |
| 1  2  3 | x = 5  y = ~x;  print(f"y: {y}")  # -6 |

### Операции сдвига

Операции сдвига также производятся над разрядами чисел. Сдвиг может происходить вправо и влево.

* x<<y - сдвигает число x влево на y разрядов. Например, 4<<1 сдвигает число 4 (которое в двоичном представлении 100) на один разряд влево, то есть в итоге получается 1000 или число 8 в десятичном представлении.
* x>>y - сдвигает число x вправо на y разрядов. Например, 16>>1 сдвигает число 16 (которое в двоичном представлении 10000) на один разряд вправо, то есть в итоге получается 1000 или число 8 в десятичном представлении.

Таким образом, если исходное число, которое надо сдвинуть в ту или другую строну, делится на два, то фактически получается умножение или деление на два. Поэтому подобную операцию можно использовать вместо непосредственного умножения или деления на два. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | a = 16 # в двоичной форме 10000  b = 2  c = a << b  # Сдвиг числа 10000 влево на 2 разряда, равно 1000000 или 64 в десятичной системе  print(c)   #64    d = a >> b  #Сдвиг числа  10000  вправо на 2 разряда, равно 100 или 4 в десятичной системе  print(d)   #4 |

При этом числа, которые участвую в операциях, необязательно должны быть кратны 2::

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | a = 22 # в двоичной форме 10110  b = 2  c = a << b  # Сдвиг числа 10110 влево на 2 разряда, равно 1011000 или 88 в десятичной системе  print(c)   # 88    d = a >> b  # Сдвиг числа 10110 вправо на 2 разряда, равно 101 или 5 в десятичной системе  print(d)   # 5 |

## Условные выражения

Ряд операций представляют условные выражения. Все эти операции принимают два операнда и возвращают логическое значение, которое в Python представляет тип **bool**. Существует только два логических значения - **True** (выражение истинно) и **False** (выражение ложно).

### Операции сравнения

Простейшие условные выражения представляют операции сравнения, которые сравнивают два значения. Python поддерживает следующие операции сравнения:

* **==**

Возвращает True, если оба операнда равны. Иначе возвращает False.

* **!=**

Возвращает True, если оба операнда НЕ равны. Иначе возвращает False.

* **>** (больше чем)

Возвращает True, если первый операнд больше второго.

* **<** (меньше чем)

Возвращает True, если первый операнд меньше второго.

* **>=** (больше или равно)

Возвращает True, если первый операнд больше или равен второму.

* **<=** (меньше или равно)

Возвращает True, если первый операнд меньше или равен второму.

Примеры операций сравнения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | a = 5  b = 6  result = 5 == 6  # сохраняем результат операции в переменную  print(result)  # False - 5 не равно 6  print(a != b)  # True  print(a > b)  # False - 5 меньше 6  print(a < b)  # True    bool1 = True  bool2 = False  print(bool1 == bool2)  # False - bool1 не равно bool2 |

Операции сравнения могут сравнивать различные объекты - строки, числа, логические значения, однако оба операнда операции должны представлять один и тот же тип.

### Логические операции

Для создания составных условных выражений применяются логические операции. В Python имеются следующие логические операторы:

* Оператор **and** (логическое умножение) применяется к двум операндам:

|  |  |
| --- | --- |
| 1 | x and y |

* Сначала оператор and оценивает выражение x, и если оно равно False, то возвращается его значение. Если оно равно True, то оценивается второй операнд - y и возвращается значение y.

|  |  |
| --- | --- |
| 1  2  3  4 | age = 22  weight = 58  result = age > 21 and weight == 58  print(result)  # True |

* В данном случае оператор and сравнивает результаты двух выражений: age > 21 weight == 58. И если оба этих выражений возвращают True, то оператор and также возвращает True (формально возвращается значение последнего операнда).
* Но операндами оператора and необязательно выступают значения True и False. Это могут быть любые значения. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | result = 4 and "w"  print(result)  # w, так как 4 равно True, поэтому возвращается значение последнего операнда    result = 0 and "w"  print(result)  # 0, так как 0 эквивалентно False |

* В данном случае число 0 и пустая строка "" расматриваются как False, все остальные числа и непустые строки эквивалентны True
* **or** (логическое сложение) также применяется к двум операндам:

|  |  |
| --- | --- |
| 1 | x or y |

* Сначала оператор or оценивает выражение x, и если оно равно True, то возвращается его значение. Если оно равно False, то оценивается второй операнд - y и возвращается значение y. Например

|  |  |
| --- | --- |
| 1  2  3  4 | age = 22  isMarried = False  result = age > 21 or isMarried  print(result)  # True, так как выражение age > 21 равно True |

* И также оператор or может применяться к любым значениям. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | result = 4 or "w"  print(result)  # 4, так как 4 эквивалентно True, поэтому возвращается значение первого операнда    result = 0 or "w"  print(result)  # w, так как 0 эквивалентно False, поэтому возвращается значение последнего операнда |

* **not** (логическое отрицание)

Возвращает True, если выражение равно False

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | age = 22  isMarried = False  print(not age > 21)  # False  print(not isMarried)  # True  print(not 4)  # False  print(not 0)  # True |

### Оператор in

Оператор **in** возвращает True если в некотором наборе значений есть определенное значение. Он имеет следующую форму:

|  |  |
| --- | --- |
| 1 | значение in набор\_значений |

Например, строка представляет набор символов. И с помощью оператора in мы можем проверить, есть ли в ней какая-нибудь подстрока:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | message = "hello world!"  hello = "hello"  print(hello in message)  # True - подстрока hello есть в строке "hello world!"    gold = "gold"  print(gold in message)  # False - подстроки "gold" нет в строке "hello world!" |

Если нам надо наоборот проверить, нет ли в наборе значений какого-либо значения, то мы може использовать модификацию оператора - **not in**. Она возвращает True, если в наборе значений НЕТ определенного значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | message = "hello world!"  hello = "hello"  print(hello not in message)  # False    gold = "gold"  print(gold not in message)  # True |

## Условная конструкция if

Условные конструкции используют условные выражения и в зависимости от их значения направляют выполнение программы по одному из путей. Одна из таких конструкций - это конструкция **if**. Она имеет следующее формальное определение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | if логическое\_выражение:      инструкции  [elif логическое выражение:      инструкции]  [else:      инструкции] |

В самом простом виде после ключевого слова **if** идет логическое выражение. И если это логическое выражение возвращает True, то выполняется последующий блок инструкций, каждая из которых должна начинаться с новой строки и должна иметь отступы от начала выражения if (отступ желательно делать в 4 пробела или то количество пробелов, которое кратно 4):

|  |  |
| --- | --- |
| 1  2  3  4 | language = "english"  if language == "english":      print("Hello")  print("End") |

Поскольку в данном случае значение переменной language равно "english", то будет выполняться блок if, который содержит только одну инструкцию - print("Hello"). В итоге консоль выведет следующие строки:

Hello

End

Обратите внимание в коде на последнюю строку, которая выводит сообщение "End". Она не имеет отступов от начала строки, поэтому она не принадлежит к блоку if и будет выполняться в любом случае, даже если выражение в конструкции if возвратит False.

Но если бы мы поставили бы отступы, то она также принадлежала бы к конструкции if:

|  |  |
| --- | --- |
| 1  2  3  4 | language = "english"  if language == "english":      print("Hello")      print("End") |

### Блок else

Если вдруг нам надо определить альтернативное решение на тот случай, если выражение в **if** возвратит False, то мы можем использовать блок **else**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | language = "russian"  if language == "english":      print("Hello")  else:      print("Привет")  print("End") |

Если выражение language == "english" возвращает True, то выполняется блок if, иначе выполняется блок else. И поскольку в данном случае условие language == "english" возвращает False, то будут выполняться инструкция из блока else.

Причем инструкции блока **else** также должны имет отступы от начала строки. Например, в примере выше print("End") не имеет отступа, поэтому она не входит в блок else и будет выполнятьься вне зависимости, чему равно условие language == "english". То есть консоль нам выведет следующие строки:

Привет

End

Блок else также может иметь несколько инструкций, которые должны иметь отступ от начала строки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | language = "russian"  if language == "english":      print("Hello")      print("World")  else:      print("Привет")      print("мир") |

### elif

Если необходимо ввести несколько альтернативных условий, то можно использовать дополнительные блоки **elif**, после которого идет блок инструкций.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | language = "german"  if language == "english":      print("Hello")      print("World")  elif language == "german":      print("Hallo")      print("Welt")  else:      print("Привет")      print("мир") |

Сначала Python проверяет выражение if. Если оно равно True, то выполняются инструкции из блока if. Если это условие возвращает False, то Python проверяет выражение из elif.

Если выражение после elif равно True, то выполняются инструкции из блока elif. Но если оно равно False то выполняются инструкции из блока else

При необходимости можно определить несколько блоков elif для разных условий. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | language = "german"  if language == "english":      print("Hello")  elif language == "german":      print("Hallo")  elif language == "french":      print("Salut")  else:      print("Привет") |

### Вложенные конструкции if

Конструкция if в свою очередь сама может иметь вложенные конструкции if:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | language = "english"  daytime = "morning"  if language == "english":      print("English")      if daytime == "morning":          print("Good morning")      else:          print("Good evening") |

Здесь конструкция if содержит вложенную конструкцию if/else. То есть если переменная language равна "english", тогда вложенная конструкция if/else дополнительно проверяет значение переменной daytime - равна ли она строке "morning" ли нет. И в данном случае мы получим следующий консольный вывод:

English

Good morning

Стоит учитывать, что вложенные выражения if также должны начинаться с отступов, а инструкции во вложенных конструкциях также должны иметь отступы. Отступы, расставленные не должным образом, могут изменить логику программы. Так, предыдущий пример НЕ аналогичен следующему:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | language = "english"  daytime = "morning"  if language == "english":      print("English")  if daytime == "morning":      print("Good morning")  else:       print("Good evening") |

Подобным образом можно размещать вложенные конструкции if/elif/else в блоках elif и else:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | language = "russian"  daytime = "morning"  if language == "english":      if daytime == "morning":          print("Good morning")      else:          print("Good evening")  else:      if daytime == "morning":          print("Доброе утро")      else:          print("Добрый вечер")  Вопрос 1  Что выведет следующий код:   |  |  | | --- | --- | | 1  2  3  4  5 | n = 5  if n == 5:      print("false")  else:      print("true") |   Варианты ответов   * true * false * false true   Ответ  На консоль будет выведена строка "false", потому что переменная n равна 5, соответственно условие в выражении if n == 5 будет верно. Поэтому будет выполняться инструкция print("false")  Вопрос 2  Что выведет следующий код:   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9 | n = 5  if n == 5:      print("5")      n = n + 1  elif n == 6:      print("6")      n = n + 1  else:      print("7") |   Варианты ответов   * 5 * 6 * 7 * 56 * 567   Ответ  На консоль будет выведено число 5.  Вопрос 3  Что выведет следующий код:   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8 | n = 5    if n == 5:      print("n = 5")  elif n < 6:      print("n < 6")  else:      print("n > 6") |   Ответ  На консоль будет выведена строка "n = 5".  Вопрос 4  Что выведет следующий код:   |  |  | | --- | --- | | 1  2  3  4  5 | x = 3  y = 4  z = 5    print(x if not x + y \* z else y) |   Варианты ответов   * 3 * 4 * 5 * True * False   Ответ  На консоль будет выведено число 4. В данном случае выражение if not x + y \* z задает условие, если x + y \* z возвращает False, то возвращается значение x, иначе возвращается y. В данном случае x + y \* z эквивалентно 3 + 4 \* 5. Любое число, не равное 0, эквивалентно True. Поэтому и здесь выражение x + y \* z соответствует True. Поэтому возвращается значение переменной y. |

## Циклы

Циклы позволяют выполнять некоторое действие в зависимости от соблюдения некоторого условия. В языке Python есть следующие типы циклов:

* **while**
* **for**

### Цикл while

Цикл **while** проверяет истинность некоторого условия, и если условие истинно, то выполняет инструкции цикла. Он имеет следующее формальное определение:

|  |  |
| --- | --- |
| 1  2 | while условное\_выражение:     инструкции |

После ключевого слова **while** указывается условное выражение, и пока это выражение возвращает значение True, будет выполняться блок инструкций, который идет далее.

Все инструкции, которые относятся к циклу while, располагаются на последующих строках и должны иметь отступ от начала ключевого слова while.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | number = 1    while number < 5:      print(f"number = {number}")      number += 1  print("Работа программы завершена") |

В данном случае цикл while будет выполняться, пока переменная number меньше 5.

Сам блок цикла состоит из двух инструкций:

|  |  |
| --- | --- |
| 1  2 | print(f"number = {number}")  number += 1 |

Обратите внимание, что они имеют отступы от начала оператора while - в данном случае от начала строки. Благодаря этому Python может определить, что они принадлежат циклу. В самом цикле сначала выводится значение переменной number, а потом ей присваивается новое значение. .

Также обратите внимание, что последняя инструкция print("Работа программы завершена") не имеет отступов от начала строки, поэтому она не входит в цикл while.

Весь процесс цикла можно представить следующим образом:

1. Сначала проверяется значение переменной number - меньше ли оно 5. И поскольку вначале переменная равна 1, то это условие возвращает True, и поэтому выполняются инструкции цикла

Инструкции цикла выводят на консоль строку number = 1. И далее значение переменной number увеличивается на единицу - теперь она равна 2. Однократное выполнение блока инструкций цикла называется **итерацией**. То есть таким образом, в цикле выполняется первая итерация.

1. Снова проверяется условие number < 5. Оно по прежнему равно True, так как number = 2, поэтому выполняются инструкции цикла

Инструкции цикла выводят на консоль строку number = 2. И далее значение переменной number опять увеличивается на единицу - теперь она равна 3. Таким образом, выполняется вторая итерация.

1. Опять проверяется условие number < 5. Оно по прежнему равно True, так как number = 3, поэтому выполняются инструкции цикла

Инструкции цикла выводят на консоль строку number = 3. И далее значение переменной number опять увеличивается на единицу - теперь она равна 4. То есть выполняется третья итерация.

1. Снова проверяется условие number < 5. Оно по прежнему равно True, так как number = 4, поэтому выполняются инструкции цикла

Инструкции цикла выводят на консоль строку number = 4. И далее значение переменной number опять увеличивается на единицу - теперь она равна 5. То есть выполняется четвертая итерация.

1. И вновь проверяется условие number < 5. Но теперь оно равно **False**, так как number = 5, поэтому выполняются выход из цикла. Все цикл - завершился. Дальше уже выполняются действия, которые определены после цикла. Таким образом, данный цикл произведет четыре прохода или четыре итерации

В итоге при выполнении кода мы получим следующий консольный вывод:

number = 1

number = 2

number = 3

number = 4

Работа программы завершена

Для цикла while также можно определить дополнительный блок **else**, инструкции которого выполняются, когда условие равно False:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | number = 1    while number < 5:      print(f"number = {number}")      number += 1  else:      print(f"number = {number}. Работа цикла завершена")  print("Работа программы завершена") |

То есть в данном случае сначала проверяется условие и выполняются инструкции while. Затем, когда условие становится равным False, выполняются инструкции из блока else. Обратите внимание, что инструкции из блока else также имеют отступы от начала конструкции цикла. В итоге в данном случае мы получим следующий консольный вывод:

number = 1

number = 2

number = 3

number = 4

number =5. Работа цикла завершена

Работа программы завершена

Блок else может быть полезен, если условие изначально равно False, и мы можем выполнить некоторые действия по этому поводу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | number = 10    while number < 5:      print(f"number = {number}")      number += 1  else:      print(f"number = {number}. Работа цикла завершена")  print("Работа программы завершена") |

В данном случае условие number < 5 изначально равно False, поэтому цикл не выполняет ни одной итерации и сразу переходит в блоку else.

### Цикл for

Другой тип циклов представляет конструкция **for**. Этот цикл пробегается по набору значений, помещает каждое значение в переменную, и затем в цикле мы можем с этой переменной производить различные действия. Формальное определение цикла for:

|  |  |
| --- | --- |
| 1  2 | for переменная in набор\_значений:      инструкции |

После ключевого слова **for** идет название переменной, в которую будут помещаться значения. Затем после оператора **in** указывается набор значений и двоеточие.

А со следующей строки располагается блок инструкций цикла, которые также должны иметь отступы от начала цикла.

При выполнении цикла Python последовательно получает все значения из набора и передает их переменную. Когда все значения из набора будут перебраны, цикл завершает свою работу.

В качестве набора значений, например, можно рассматривать строку, которая по сути представляет набор символов. Посмотрим на примере:

|  |  |
| --- | --- |
| 1  2  3  4 | message = "Hello"    for c in message:      print(c) |

В цикле определяется переменную c, после оператора in в качестве перебираемого набора указана переменная message, которая хранит строку "Hello". В итоге цикл for будет перебираеть последовательно все символы из строки message и помещать их в переменную c. Блок самого цикла состоит из одной инструкции, которая выводит значение переменной с на консоль. Консольный вывод программы:

H

e

l

l

o

Цикл for также может иметь дополнительный блок **else**, который выполняется после завершения цикла:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | message = "Hello"  for c in message:      print(c)  else:      print(f"Последний символ: {c}. Цикл завершен");  print("Работа программы завершена")  # инструкция не имеет отступа, поэтому не относится к else |

В данном случае мы получим следующий консольный вывод:

H

e

l

l

o

Последний символ: o. Цикл завершен

Работа программы завершена

Стоит отметить, что блок else имеет доступ ко всем переменным, которые определены в цикле for.

### Вложенные циклы

Одни циклы внутри себя могут содержать другие циклы. Рассмотрим на примере вывода таблицы умножения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | i = 1  j = 1  while i < 10:      while j < 10:          print(i \* j, end="\t")          j += 1      print("\n")      j = 1      i += 1 |

Внешний цикл while i < 10: срабатывает 9 раз пока переменная i не станет равна 10. Внутри этого цикла срабатывает внутренний цикл while j < 10:. Внутренний цикл также срабатывает 9 раз пока переменная j не станет равна 10. Причем все 9 итераций внутреннего цикла срабатывают в рамках одной итерации внешнего цикла.

В каждой итерации внутреннего цикла на консоль будет выводится произведение чисел i и j. Затем значение переменной j увеличивается на единицу. Когда внутренний цикл закончил работу, значений переменной j сбрасывается в 1, а значение переменной i увеличивается на единицу и происходит переход к следующей итерации внешнего цикла. И все повторяется, пока переменная i не станет равна 10. Соответственно внутренний цикл сработает всего 81 раз для всех итераций внешнего цикла. В итоге мы получим следующий консольный вывод:

1 2 3 4 5 6 7 8 9

2 4 6 8 10 12 14 16 18

3 6 9 12 15 18 21 24 27

4 8 12 16 20 24 28 32 36

5 10 15 20 25 30 35 40 45

6 12 18 24 30 36 42 48 54

7 14 21 28 35 42 49 56 63

8 16 24 32 40 48 56 64 72

9 18 27 36 45 54 63 72 81

Подобным образом можно определять вложенные циклы for:

|  |  |
| --- | --- |
| 1  2  3 | for c1 in  "ab":      for c2 in "ba":          print(f"{c1}{c2}") |

В данном случае внешний цикл проходит по строке "ab" и каждый символ помещает в переменную c1. Внутренний цикл проходит по строке "ba", помещает каждый символ строки в переменную c2 и выводит сочетание обоих символов на консоль. То есть в итоге мы получим все возможные сочетания символов a и b:

ab

aa

bb

ba

### Выход из цикла. break и continue

Для управления циклом мы можем использовать специальные операторы **break** и **continue**. Оператор **break** осуществляет выход из цикла. А оператор **continue** выполняет переход к следующей итерации цикла.

Оператор break может использоваться, если в цикле образуются условия, которые несовместимы с его дальнейшим выполнением. Рассмотрим следующий пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | number = 0  while number < 5:      number += 1      if number == 3 :    # если number = 3, выходим из цикла          break      print(f"number = {number}") |

Здесь цикл while проверяет условие number < 5. И пока number не равно 5, предполагается, что значение number будет выводиться на консоль. Однако внутри цикла также проверяется другое условие: if number == 3. То есть, если значение number равно 3, то с помощью оператора **break** выходим из цикла. И в итоге мы получим следующий консольный вывод:

number = 1

number = 2

В отличие от оператора break оператор **continue** выполняет переход к следующей итерации цикла без его завершения. Например, в предыдущем примере заменим break на continue:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | number = 0  while number < 5:      number += 1      if number == 3 :    # если number = 3, переходим к новой итерации цикла          continue      print(f"number = {number}") |

И в этом случае если значение переменной number равно 3, последующие инструкции после оператора continue не будут выполняться:

number = 1

number = 2

number = 4

number = 5

**Вопрос 1**

Что выведет следующий код:

|  |  |
| --- | --- |
| 1  2  3  4 | n = 5  while(n):      n = n -1      print(n, end=" ") |

Варианты ответов

* 5 4 3 2 1
* 4 3 2 1 0
* 4 3 2 1
* 5 4 3 2 1 0

Ответ

На консоль будет выведена последовательность чисел "4 3 2 1 0". В данном случае цикл while будет выполняться пока условие (n) возвращает True. Поскольку n - это число, то (n) возвратит False, когда n будет равно 0. До этого момента цикл успеет пять раз уменьшить значение n.

## Функции

Функции представляют блок кода, который выполняет определенную задачу и который можно повторно использовать в других частях программы. В предыдущих статьях уже использовались функции. В частности, функция **print()**, которая выводит некоторое значение на консоль. Python имеет множество встроенных функций и позволяет определять свои функции. Формальное определение функции:

|  |  |
| --- | --- |
| 1  2 | def имя\_функции ([параметры]):      инструкции |

Определение функции начинается с выражения **def**, которое состоит из имени функции, набора скобок с параметрами и двоеточия. Параметры в скобках необязательны. А со следующей строки идет блок инструкций, которые выполняет функция. Все инструкции функции имеют отступы от начала строки.

Например, определение простейшей функции:

|  |  |
| --- | --- |
| 1  2 | def say\_hello():      print("Hello") |

Функция называется say\_hello. Она не имеет параметров и содержит одну единственную инструкцию, которая выводит на консоль строку "Hello".

Обратите внимание, что инструкции функции должны иметь отступы от начала функции. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | def say\_hello():      print("Hello")      print("Bye") |

Здесь инструкция print("Bye") не имеет отступов от начала функции say\_hello и поэтому в эту функцию не входит. Обычно между определением функции и остальными инструкциями, которые не входят в функцию, располагаются две пустых строки.

Для вызова функции указывается имя функции, после которого в скобках идет передача значений для всех ее параметров:

|  |  |
| --- | --- |
| 1 | имя\_функции ([параметры]) |

Например, определим и вызовем функцию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def say\_hello():    # определение функции say\_hello      print("Hello")      say\_hello()         # вызов функции say\_hello  say\_hello()  say\_hello() |

Здесь три раза подряд вызывается функция say\_hello. В итоге мы получим следующий консольный вывод:

Hello

Hello

Hello

Обратите внимание, что функция сначала определяется, а потом вызывается.

Если функция имеет одну инструкцию, то ее можно разместить на одной строке с остальным определением функции:

|  |  |
| --- | --- |
| 1  2  3 | def say\_hello(): print("Hello")    say\_hello() |

Подобным образом можно определять и вызывать и другие функции. Например, определим и выполним несколько функций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def say\_hello():      print("Hello")      def say\_goodbye():      print("Good Bye")      say\_hello()  say\_goodbye() |

Консольный вывод:

Hello

Good Bye

### Локальные функции

Одни функции могут определяться внутри других функций - внутренние функции еще называют **локальными**. Локальные функции можно использовать только внутри той функции, в которой они определены. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def print\_messages():      # определение локальных функций      def say\_hello(): print("Hello")      def say\_goodbye(): print("Good Bye")      # вызов локальных функций      say\_hello()      say\_goodbye()    # Вызов функции print\_messages  print\_messages()    #say\_hello() # вне функции print\_messages функция say\_hello не доступна |

Здесь функции say\_hello() и say\_goodbye() определены внутри функции print\_messages() и поэтому по отношению к ней являются локальными. Соответственно они могут использоваться только внутри функции print\_messages()

### Организация программы и функция main

В программе может быть определено множество функций. И чтобы всех их упорядочить, одним из способов их организации является добавление специальной функции (обычно называется main), в которой потом уже вызываются другие функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def main():      say\_hello()      say\_goodbye()    def say\_hello():      print("Hello")    def say\_goodbye():      print("Good Bye")    # Вызов функции main  main() |

## Параметры функции

Функция может принимать параметры. Через параметры в функцию можно передавать данные. Банальный пример - функция print(), которая с помощью параметра принимает значение, выводимое на консоль.

Теперь определим и используем свою функцию с параметрами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def say\_hello(name):      print(f"Hello, {name}")    say\_hello("Tom")  say\_hello("Bob")  say\_hello("Alice") |

Функция say\_hello имеет параметр name, и при вызове функции мы можем передать этому параметру какой-либо значение. Внутри функции мы можем использовать параметр как обычную переменную, например, вывести значение этого параметра на консоль функцией print. Так, в выражении:

|  |  |
| --- | --- |
| 1 | say\_hello("Tom") |

Строка "Tom" будет передаваться параметру name. В итоге при выполнении программы мы получим следующий консольный вывод:

Hello, Tom

Hello, Bob

Hello, Alice

При вызове функции значения передаются параметрам по позиции. Например, определим и вызовем функцию с несколькими параметрами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def print\_person(name, age):      print(f"Name: {name}")      print(f"Age: {age}")      print\_person("Tom", 37) |

Здесь функция print\_person принимает два параметра: name и age. При вызове функции:

|  |  |
| --- | --- |
| 1 | print\_person("Tom", 37) |

Первое значение - "Tom" передается первому параметру, то есть параметру name. Второе значение - 37 передается второму параметру - age. И внутри функции значения параметров выводятся на консоль:

Name: Tom

Age: 37

### Значения по умолчанию

Некоторые параметры функции мы можем сделать необязательными, указав для них значения по умолчанию при определении функции. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def say\_hello(name="Tom"):      print(f"Hello, {name}")      say\_hello()         # здесь параметр name будет иметь значение "Tom"  say\_hello("Bob")    # здесь name = "Bob" |

Здесь параметр name является необязательным. И если мы не передаем при вызове функции для него значение, то применяется значение по умолчанию, то есть строка "Tom". Консольный вывод данной программы:

Hello, Tom

Hello, Bob

Если функция имеет несколько параметров, то необязательные параметры должны идти после обязательных. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def print\_person(name, age = 18):      print(f"Name: {name}  Age: {age}")      print\_person("Bob")  print\_person("Tom", 37) |

Здесь параметр age является необязательным и по умолчанию имеет значение 18. Перед ним расположен обязательный параметр name. Поэтому при вызове функции мы можем не передавать значение параметру age, но параметру name передать значение необходимо.

При необходимости мы можем сделать все параметры необязательными:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def print\_person(name = "Tom", age = 18):      print(f"Name: {name}  Age: {age}")      print\_person()              # Name: Tom  Age: 18  print\_person("Bob")         # Name: Bob  Age: 18  print\_person("Sam", 37)     # Name: Sam  Age: 37 |

### Передача значений параметрам по имени. Именованные параметры

В примерах выше при вызове функции значения передаются параметрами функции по позиции. Но также можно передавать значения параметрам по имени. Для этого при вызове функции указывается имя параметра и ему присваивается значение:

|  |  |
| --- | --- |
| 1  2  3  4  5 | def print\_person(name, age):      print(f"Name: {name}  Age: {age}")      print\_person(age = 22, name = "Tom") |

В данном случае значения параметрам age и name передаются по имени. И несмотря на то, что параметр name идет первым в определении функции, мы можем при вызове функции написать print\_person(age = 22, name = "Tom") и таким образом передать число 22 параметру age, а строку "Tom" параметру name.

Символ **\*** позволяет установить, какие параметры будут именнованными - то есть такие параметры, которым можно передать значения только по имени. Все параметры, которые располагаются справа от символа **\***, получают значения **только по имени**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | def print\_person(name, \*,  age, company):      print(f"Name: {name}  Age: {age}  Company: {company}")      print\_person("Bob", age = 41, company ="Microsoft")    # Name: Bob  Age: 41  company: Microsoft |

В данном случае параметры age и company являются именнованными.

Можно сделать все параметры именнованными, поставив перед списком параметров символ \*:

|  |  |
| --- | --- |
| 1  2 | def print\_person(\*,  name, age, company):      print(f"Name: {name}  Age: {age}  Company: {company}") |

Если наоборот надо определить параметры, которым можно передавать значения только по позиции, то есть **позиционные** параметры, то можно использовать символ **/**: все параметры, которые идут **до** символа **/** , являются позиционными и могут получать значения только по позиции

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def print\_person(name, /, age, company="Microsoft"):      print(f"Name: {name}  Age: {age}  Company: {company}")      print\_person("Tom", company="JetBrains", age = 24)     # Name: Tom  Age: 24  company: JetBrains  print\_person("Bob", 41)                 # Name: Bob  Age: 41  company: Microsoft |

В данном случае параметр name является позиционным.

Для одной функции можно определять одновременно позиционные и именнованные параметры.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def print\_person(name, /,  age = 18, \*, company):      print(f"Name: {name}  Age: {age}  Company: {company}")      print\_person("Sam", company ="Google")               # Name: Sam  Age: 18  company: Google  print\_person("Tom", 37, company ="JetBrains")        # Name: Tom  Age: 37  company: JetBrains  print\_person("Bob", company ="Microsoft", age = 42)  # Name: Bob  Age: 42  company: Microsoft |

В данном случае параметр name располагается слева от символа /, поэтому является позиционным и обязательным - ему можно передать значение только по позиции.

Параметр company является именнованным, так как располагается справа от символа \*. Параметр age может получать значение по имени и по позиции.

### Неопределенное количество параметров

С помощью символа звездочки можно определить параметр, через который можно передавать неопределенное количество значений. Это может быть полезно, когда мы хотим, чтобы функция получала несколько значений, но мы точно не знаем, сколько именно. Например, определим функцию подсчета суммы чисел:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def sum(\*numbers):      result = 0      for n in numbers:          result += n      print(f"sum = {result}")      sum(1, 2, 3, 4, 5)      # sum = 15  sum(3, 4, 5, 6)         # sum = 18 |

В данном случае функция sum принимает один параметр - \*numbers, но звездочка перед названием параметра указывает, что фактически на место этого параметра мы можем передать неопределенное количество значений или набор значений. В самой функции с помощью цикла for можно пройтись по этому набору, получить каждое значение из этого набора в переменную n и произвести с ним какие-нибудь действия. Например, в данном случае вычисляется сумма переданных чисел.

## Оператор return и возвращение результата из функции

### Возвращение результата

Функция может возвращать результат. Для этого в функции используется оператор **return**, после которого указывается возвращаемое значение:

|  |  |
| --- | --- |
| 1  2  3 | def имя\_функции ([параметры]):      инструкции      return возвращаемое\_значение |

Определим простейшую функцию, которая возвращает значение:

|  |  |
| --- | --- |
| 1  2 | def get\_message():      return "Hello METANIT.COM" |

Здесь после оператора **return** идет строка "Hello METANIT.COM" - это значение и будет возвращать функция get\_message().

Затем это результат функции можно присвоить переменной или использовать как обычное значение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def get\_message():      return "Hello METANIT.COM"      message = get\_message()  # получаем результат функции get\_message в переменную message  print(message)          # Hello METANIT.COM    # можно напрямую передать результат функции get\_message  print(get\_message())    # Hello METANIT.COM |

После оператора return может идти и сложное вычислямое выражение, резлуьтат которого будет возвращаться из функции. Например, определим функцию, которая увеличивает число в два раза:

|  |  |
| --- | --- |
| 1  2 | def double(number):      return 2 \* number |

Здесь функция double будет возвращать результат выражения 2 \* number:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def double(number):      return 2 \* number    result1 = double(4)     # result1 = 8  result2 = double(5)     # result2 = 10  print(f"result1 = {result1}")   # result1 = 8  print(f"result2 = {result2}")   # result2 = 10 |

Или другой пример - получение суммы чисел:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def sum(a, b):      return a + b      result = sum(4, 6)                  # result = 0  print(f"sum(4, 6) = {result}")      # sum(4, 6) = 10  print(f"sum(3, 5) = {sum(3, 5)}")   # sum(3, 5) = 8 |

### Выход из функции

Оператор **return** не только возвращает значение, но и производит выход из функции. Поэтому он должен определяться после остальных инструкций. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | def get\_message():      return "Hello METANIT.COM"      print("End of the function")    print(get\_message()) |

С точки зрения синтаксиса данная функция корректна, однако ее инструкция print("End of the function") не имеет смысла - она никогда не выполнится, так как до ее выполнения оператор return возвратит значение и произведет выход из функции.

Однако мы можем использовать оператор **return** и в таких функциях, которые не возвращают никакого значения. В этом случае после оператора return не ставится никакого возвращаемого значения. Типичная ситуация - в зависимости от опеределенных условий произвести выход из функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def print\_person(name, age):      if age > 120 or age < 1:          print("Invalid age")          return      print(f"Name: {name}  Age: {age}")      print\_person("Tom", 22)  print\_person("Bob", -102) |

Здесь функция print\_person в качестве параметров принимает имя и возраст пользователя. Однако в функции вначале мы проверяем, соответствует ли возраст некоторому диапазону (меньше 120 и больше 0). Если возраст находится вне этого диапазона, то выводим сообщение о недопустимом возрасте и с помощью оператора return выходим из функции. После этого функция заканчивает свою работу.

Однако если возраст корректен, то выводим информацию о пользователе на консоль. Консольный вывод:

Name: Tom Age: 22

Invalid age

## Функция как тип, параметр и результат другой функции

### Функция как тип

В Python функция фактически представляет отдельный тип. Так мы можем присвоить переменной какую-нибудь функцию и затем, используя переменную, вызывать данную функцию. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def say\_hello(): print("Hello")  def say\_goodbye(): print("Good Bye")    message = say\_hello  message()       # Hello  message = say\_goodbye  message()       # Good Bye |

В данном случае переменной message присваивается одна из функций. Сначала ей передается функция say\_hello():

|  |  |
| --- | --- |
| 1 | message = say\_hello |

После этого переменная message будет указывать на данную функцию, то есть фактически представлять функцию say\_hello. А это значит, что мы можем вызывать переменную message как обычную функцию:

|  |  |
| --- | --- |
| 1 | message()       # Hello |

Фактически это приведет к выполнению функции say\_hello, и на консоль будет выведена строка "Hello". Затем подобным образом мы можем передать переменной message другую функцию и вызвать ее.

Подобным образом можно через переменную вызывать функцию с параметрами и возвращать ее результат:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def sum(a, b): return a + b  def multiply(a, b): return a \* b    operation = sum  result = operation(5, 6)  print(result)   # 11    operation = multiply  print(operation(5, 6))      # 30 |

### Функция как параметр функции

Поскольку функция в Python может представлять такое же значение как строка или число, соответственно мы можем передать ее в качестве параметра в другую функцию. Например, определим функцию, которая выводит на консоль результат некоторой операции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def do\_operation(a, b, operation):      result = operation(a, b)      print(f"result = {result}")    def sum(a, b): return a + b  def multiply(a, b): return a \* b    do\_operation(5, 4, sum)         # result = 9  do\_operation(5, 4, multiply)   # result = 20 |

В данном случае функция do\_operation имеет три параметра, причем третий параметр, как предполагается, будет представлять функцию, которая принимает два параметра и возвращает некоторый результат. Иными словами третий параметр - operation представляет некоторую операцию, но на момент определения функции do\_operation мы точно не знаем, что это будет за операция. Мы только знаем, что она принимает два параметр и возвращает какой-то результат, который потом выводится на консоль.

При вызове функции do\_operation мы сможем передать в качестве третьего параметра другую функцию, например, функцию sum:

|  |  |
| --- | --- |
| 1 | do\_operation(5, 4, sum) |

То есть в данном случае параметр operation фактически будет представлять функцию sum и будет возвращать сумму дву чисел.

Затем аналогичным образов в вызов функции do\_operation можно передать третьему параметру другую функцию - multiply, которая выполнит умножение чисел:

|  |  |
| --- | --- |
| 1 | do\_operation(5, 4, multiply)   # result = 20 |

Таким образом, более гибкие по функциональности функции, которые через параметры принимают другие функции.

### Функция как результат функции

Также одна функция в Python может возвращать другую функцию. Например, определим функцию, которая в зависимости от значения параметра возвращает ту или иную операцию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | def sum(a, b): return a + b  def subtract(a, b): return a - b  def multiply(a, b): return a \* b      def select\_operation(choice):      if choice == 1:          return sum      elif choice == 2:          return subtract      else:          return multiply      operation = select\_operation(1)     # operation = sum  print(operation(10, 6))             # 16    operation = select\_operation(2)     # operation = subtract  print(operation(10, 6))             # 4    operation = select\_operation(3)     # operation = multiply  print(operation(10, 6))             # 60 |

В данном случае функция select\_operation в зависимости от значения параметра choice возвращает одну из трех функций - sum, subtract и multiply. Затем мы мы можем получить результат функции select\_operation в переменную operation:

|  |  |
| --- | --- |
| 1 | operation = select\_operation(1) |

Так, в данном случае в функцию select\_operation передается число 1, соответственно она будет возвращать функцию sum. Поэтому переменная operation фактически будет указывать на функцию sum, которая выполняет сложение двух чисел:

|  |  |
| --- | --- |
| 1 | print(operation(10, 6))             # 16 - фактически равно sum(10, 6) |

Аналогичным образом можно получить и выполнить другие функции.

## Лямбда-выражения

Лямбда-выражения в языке Python представляют небольшие анонимные функции, которые определяются с помощью оператора **lambda**. Формальное определение лямбда-выражения:

|  |  |
| --- | --- |
| 1 | lambda [параметры] : инструкция |

Определим простейшее лямбда-выражение:

|  |  |
| --- | --- |
| 1  2  3 | message = lambda: print("hello")    message()   # hello |

Здесь лямбда-выражение присваивается переменной message. Это лямбда-выражение не имеет параметров, ничего не возвращает и просто выводит строку "hello" на консоль. И через переменную message мы можем вызвать это лямбда-выражение как обычную функцию. Фактически оно аналогично следующей функции:

|  |  |
| --- | --- |
| 1  2 | def message():      print("hello") |

Если лямбда-выражение имеет параметры, то они определяются после ключевого слова **lambda**. Если лямбда-выражение возвращает какой-то результат, то он указывается после двоеточия. Например, определим лямбда-выражение, которое возвращает квадрат числа:

|  |  |
| --- | --- |
| 1  2  3  4 | square = lambda n: n \* n    print(square(4))    # 16  print(square(5))    # 25 |

В данном случае лямбда-выражение принимает один параметр - n. Справа от двоеточия идет возвращаемое значение - n\* n. Это лямбда-выражение аналогично следующей функции:

|  |  |
| --- | --- |
| 1 | def square2(n): return n \* n |

Аналогичным образом можно создавать лямбда-выражения, которые принимают несколько параметров:

|  |  |
| --- | --- |
| 1  2  3  4 | sum = lambda a, b: a + b    print(sum(4, 5))    # 9  print(sum(5, 6))    # 11 |

Хотя лямбда-выражения позволяют немного сократить определения функций, тем не менее они ограничены тем, что они могут выполнять только одно выражение. Однако они могут быть довольно удобны в тех случаях, когда необходимо использовать функцию для передачи в качестве параметра или возвращения в другой функции. Например, передача лямбда-выражения в качестве параметра:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def do\_operation(a, b, operation):      result = operation(a, b)      print(f"result = {result}")    do\_operation(5, 4, lambda a, b: a + b)  # result = 9  do\_operation(5, 4, lambda a, b: a \* b)  # result = 20 |

В данном случае нам нет необходимости определять функции, чтобы передать их в качестве параметра, как в прошлой статье.

То же самое касается и возвращение лямбда-выражений из функций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | def select\_operation(choice):      if choice == 1:          return lambda a, b: a + b      elif choice == 2:          return lambda a, b: a - b      else:          return lambda a, b: a \* b      operation = select\_operation(1)  # operation = sum  print(operation(10, 6))  # 16    operation = select\_operation(2)  # operation = subtract  print(operation(10, 6))  # 4    operation = select\_operation(3)  # operation = multiply  print(operation(10, 6))  # 60 |

## Преобразование типов

В операциях с данными могут применяться значения различных типов. Например, складываются число типа int и число типа float:

|  |  |
| --- | --- |
| 1  2  3  4 | a = 2       # число int  b = 2.5     # число float  c = a + b  print(c)    # 4.5 |

В данном случае никакой ошибки не будет. Однако Python не всегда может автоматически производить операции, в которых участвуют данные разных типов. Рассмотрим, какие в данном случае действуют правила.

### Неявные преобразования

Оба числа в арифметических операциях должны представлять один и тот же тип. Если же два операнда операции представляют разные типы данных, то Python пытается автоматически выполнить преобразования к одному из типов в соответствии со следующми правилами:

* Если один из операндов операции представляет комплексное число (тип **complex**), то другой операнд также преобразуется к типу complex.
* Иначе, если один из операндов представляет тип **float**, то второй операнд также преобразуется к типу float. Собственно так и произошло в примере выше, где значение переменной a было преобразовано в тип float
* Иначе, оба операнда должны представлять тип **int**, и в этом случае преобазование не требуется

### Явные преобразования

Но в некоторых случаях возникает необходимость вручную выполнить преобразование типов. Например, пусть у нас будет следующий код:

|  |  |
| --- | --- |
| 1  2  3 | a = "2"  b = 3  c = a + b |

Мы ожидаем, что "2" + 3 будет равно 5. Однако этот код сгенерирует исключение, так как первое число на самом деле представляет строку. И мы увидим при выполнении кода что-то наподобие:

Traceback (most recent call last):

File "/Users/eugene/PycharmProjects/HelloApp/main.py", line 3, in

c = a + b

TypeError: can only concatenate str (not "int") to str

Для преобразования типов Python предоставляет ряд встроенных функций:

* **int()**: преобразует значение в целое число
* **float()**: преобразует значение в число с плавающей точкой
* **str()**: преобразует значение в строку

#### int

Так, в предыдущем примере преобазуем строку в число с помощью функции int():

|  |  |
| --- | --- |
| 1  2  3  4 | a = "2"  b = 3  c = int(a) + b  print(c)    # 5 |

Примеры преобразований с помощью int():

|  |  |
| --- | --- |
| 1  2  3  4  5 | a = int(15)     # a = 15  b = int(3.7)    # b = 3  c = int("4")    # c = 4  e = int(False)    # e = 0  f = int(True)     # f = 1 |

Однако если значение не может быть преобразовано, то функция int выдаст ошибку ValueError: invalid literal for int() with base 10:

|  |  |
| --- | --- |
| 1  2 | b = int("a1c")    # Ошибка  c = int("4.7")    # Ошибка |

#### float

Аналогичным образом действует функция **float()**, которая преобразует в число с плавающей точкой.

|  |  |
| --- | --- |
| 1  2  3  4 | a = "2.7"  b = 3  c = float(a) + b  print(c) # 5.7 |

Примеры преобразований с помощью float():

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | a = float(15)       # a = 15.0  b = float(3.7)      # b = 3.7  c = float("4.7")    # c = 4.7  d = float("5")      # d = 5.0  e = float(False)    # e = 0.0  f = float(True)     # f = 1.0 |

Но опять же не все значения могут автоматически преобразованы в float. Так, в следующем случае Python сгенерирует ошибку:

|  |  |
| --- | --- |
| 1 | d = float("abc")  # Ошибка |

#### str

Функция **str()** преобразует значение в строку:

|  |  |
| --- | --- |
| 1  2  3  4 | a = str(False)      # a = "False"  b = str(True)       # b = "True"  c = str(5)         # c = "5"  d = str(5.7)       # d = "5.7" |

Функция str() может быть актуальна, например, при добавлении к строке значения другого типа. Например, в следующем случае мы получим ошибку:

|  |  |
| --- | --- |
| 1  2  3 | age = 22  message = "Age: " + age     # Ошибка  print(message) |

Если число складывается с число, то это стандартная операция сложения чисел. Если строка складывается со строкой, то это операция объединения строк. Но каким образом выполнить операцию сложения по отношение к строке и числу, Python не знает. И если мы в данном случае мы хотим выполнить операцию объединения строк, то число можно привести к строке с помощью функции str():

|  |  |
| --- | --- |
| 1  2  3 | age = 22  message = "Age: " + str(age)   # Age: 22  print(message) |

## Область видимости переменных

Область видимости или scope определяет контекст переменной, в рамках которого ее можно использовать. В Python есть два типа контекста: глобальный и локальный.

### Глобальный контекст

Глобальный контекст подразумевает, что переменная является глобальной, она определена вне любой из функций и доступна любой функции в программе. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | name = "Tom"      def say\_hi():      print("Hello", name)      def say\_bye():      print("Good bye", name)    say\_hi()  say\_bye() |

Здесь переменная name является глобальной и имеет глобальную область видимости. И обе определенные здесь функции могут свободно ее использовать.

### Локальный контекст

В отличие от глобальных переменных локальная переменная определяется внутри функции и доступна только из этой функции, то есть имеет локальную область видимости:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def say\_hi():      name = "Sam"      surname = "Johnson"      print("Hello", name, surname)      def say\_bye():      name = "Tom"      print("Good bye", name)    say\_hi()  say\_bye() |

В данном случае в каждой из двух функций определяется локальная переменная name. И хотя эти переменные называются одинаково, но тем не менее это две разных переменных, каждая из которых доступна только в рамках своей функции. Также в функции say\_hi() определена переменная surname, которая также является локальной, поэтому в функции say\_bye() мы ее использовать не сможем.

### Скрытие переменных

Есть еще один вариант определения переменной, когда локальная переменная скрывают глобальную с тем же именем:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | name = "Tom"      def say\_hi():      name = "Bob"        # скрываем значение глобальной переменной      print("Hello", name)      def say\_bye():      print("Good bye", name)      say\_hi()    # Hello Bob  say\_bye()   # Good bye Tom |

Здесь определена глобальная переменная name. Однако в функции say\_hi определена локальная переменная с тем же именем name. И если функция say\_bye использует глобальную переменную, то функция say\_hi использует локальную переменную, которая скрывает глобальную.

Если же мы хотим изменить в локальной функции глобальную переменную, а не определить локальную, то необходимо использовать ключевое слово **global**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | name = "Tom"      def say\_hi():      global  name      name = "Bob"        # изменяем значение глобальной переменной      print("Hello", name)      def say\_bye():      print("Good bye", name)      say\_hi()    # Hello Bob  say\_bye()   # Good bye Bob |

### nonlocal

Выражение **nonlocal** прикрепляет идентификатор к переменной из ближайщего окружающего контекста (за исключением глобального контекста). Обычно **nonlocal** применяется во вложенных функциях, когда надо прикрепить идентификатор за переменной или параметром окружающей внешней функции. Рассмотрим ситуацию, где это выражение может пригодиться:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | def outer():  # внешняя функция      n = 5        def inner():    # вложенная функция          print(n)        inner()     # 5      print(n)      outer()     # 5 |

Здесь вложенная локальная функция inner() выводит на консоль значение переменной **n**, которая определена во внешней функции outer(). Затем в функции outer() вызывается внутренняя функция inner().

При вызове функции outer() здесь мы ожидаемо увидим на консоли два раза число 5. Однако в данном случае вложенная функция inner() просто получает значение. Теперь возьмем другую ситуацию, когда вложенная функция присваивает значение переменной:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | def outer():  # внешняя функция      n = 5        def inner():    # вложенная функция          n = 25          print(n)        inner()     # 25      print(n)      outer()     # 5  # 25    - inner  # 5     - outer |

При присвоении значения во вложенной функции: n = 25 будет создаваться новая переменная n, которая скроет переменную n из окружающей внешней функции outer. В итоге мы получим при выводе два разных числа. Чтобы во вложенной функции указать, что идентификатор во вложенной функции будет представлять переменную из окружающей функции, применяется выражение nonlocal:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | def outer():  # внешняя функция      n = 5        def inner():    # вложенная функция          nonlocal n  # указываем, что n - это переменная из окружающей функции          n = 25          print(n)        inner()     # 25      print(n)      outer()          # 25 |

## Замыкания

**Замыкание** (closure) представляет функцию, которая запоминает свое лексическое окружение даже в том случае, когда она выполняется вне своей области видимости.

Технически замыкание включает три компонента:

* внешняя функция, которая определяет некоторую область видимости и в которой определены некоторые переменные и параметры - лексическое окружение
* переменные и параметры (лексическое окружение), которые определены во внешней функции
* вложенная функция, которая использует переменные и параметры внешней функции

Для определения замыканий в Python применяются локальные функции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | def outer():        # внешняя функция      n = 5           # лексическое окружение - локальная переменная        def inner():      # локальная функция          nonlocal n          n += 1        # операции с лексическим окружением          print(n)        return inner      fn = outer()   # fn = inner, так как функция outer возвращает функцию inner  # вызываем внутреннюю функцию inner  fn()    # 6  fn()    # 7  fn()    # 8 |

Здесь функция outer определяет локальную переменную n - это и есть лексическое окружение для внутренней функции:

Внутри функции outer определена внутренняя функция - локальная функция inner, которая обращается к своему лексическому окружению - переменной n - увеличивает ее значение на единицу и выводит на консоль:

|  |  |
| --- | --- |
| 1  2  3  4 | def inner():      # локальная функция      nonlocal n      n += 1        # операции с лексическим окружением      print(n) |

Эта локальная функция возвращается функцией outer:

|  |  |
| --- | --- |
| 1 | return inner |

В программе вызываем функцию outer и получаем в переменную fn локальную функцию inner:

|  |  |
| --- | --- |
| 1 | fn = outer() |

Переменная fn и представляет собой замыкание, то есть объединяет две вещи: функцию и окружение, в котором функция была создана. И несмотря на то, что мы получили локальную функцию и можем ее вызывать вне ее окружающей функции, в которой она определена, тем не менее она запомнила свое лексическое окружение и может к нему обращаться и изменять, что мы увидим по консольному выводу:

|  |  |
| --- | --- |
| 1  2  3 | fn()    # 6  fn()    # 7  fn()    # 8 |

### Применение параметров

Кроме внешних переменных к лексическому окружению также относятся параметры окружающей функции. Рассмотрим использование параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def multiply(n):      def inner(m): return n \* m        return inner      fn = multiply(5)  print(fn(5))        # 25  print(fn(6))        # 30  print(fn(7))        # 35 |

Здесь внешняя функция - multiply возвращает функцию, которая принимает число и возвращает число.

Вызов функции multiply() возвращает локальную функцию inner:

|  |  |
| --- | --- |
| 1 | def inner(m): return n \* m |

Эта функция запоминает окружение, в котором она была создана, в частности, значение параметра n. Кроме того, сама принимает параметр и возвращает произведение параметров n и m.

В итоге при вызове функции multiply определяется переменная fn, которая получает локальную функцию inner и ее лексическое окружение - значение параметра n:

|  |  |
| --- | --- |
| 1 | fn = multiply(5) |

В данном случае параметр n равен 5.

При вызове локальной функции, например, в случае:

|  |  |
| --- | --- |
| 1 | print(fn(6))        # 30 |

Число 6 передается для параметра m локальной функции, которая возвращает произведение n и m, то есть 5 \* 6 = 30.

Также можно было бы сократить этот код с помощью лямбд:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def multiply(n): return lambda m: n \* m      fn = multiply(5)  print(fn(5))        # 25  print(fn(6))        # 30  print(fn(7))        # 35 |

## Декораторы

**Декораторы** в Python представляют функцию, которая в качестве параметра получает функцию и в качестве результата также возвращает функцию. Декораторы позволяют модифицировать выполняемую функцию, значения ее параметров и ее результат без изменения исходного кода этой функции.

Рассмотрим простейший пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | # определение функции декоратора  def select(input\_func):      def output\_func():      # определяем функцию, которая будет выполняться вместо оригинальной          print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  # перед выводом оригинальной функции выводим всякую звездочки          input\_func()                # вызов оригинальной функции          print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  # после вывода оригинальной функции выводим всякую звездочки      return output\_func     # возвращаем новую функцию    # определение оригинальной функции  @select         # применение декоратора select  def hello():      print("Hello METANIT.COM")    # вызов оригинальной функции  hello() |

Вначале определяется собственно функция декоратора, которая в данном случае называется select(). В качестве параметра декоратор получает функцию (в данном случае параметр input\_func), к которой этот декоратор будет применяться:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def select(input\_func):      def output\_func():      # определяем функцию, которая будет выполняться вместо оригинальной          print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  # перед выводом оригинальной функции выводим всякую звездочки          input\_func()                # вызов оригинальной функции          print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  # после вывода оригинальной функции выводим всякую звездочки      return output\_func     # возвращаем новую функцию |

Результатом декоратора в данном случае является локальная функция output\_func, в которой вызывается входная функция input\_func. Для простоты здесь перед и после вызыва input\_func для красоты просто выводим набор символов "#".

Далее определяется стандартная функция, к которой применяется декоратор - в данном случае это функция hello, которая просто выводит на консоль некоторую строку:

|  |  |
| --- | --- |
| 1  2  3 | @select         # применение декоратора select  def hello():      print("Hello METANIT.COM") |

Для применения декоратора перед определением функции указывается символ @, после которого идет имя декоратора. То есть в данном случае к функции hello() применяется декоратор select().

Далее вызываем обычную функцию:

|  |  |
| --- | --- |
| 1 | hello() |

Поскольку к этой функции применяется декоратор select, то в результате функциия hello передается в декоратор select() в качестве параметра input\_func. И поскольку декоратор возвращает новую функцию - output\_func, то фактически в данном случае будет выполняться именно эта функция output\_func()

В итоге мы получим следующий консольный вывод:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Hello METANIT.COM

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### Получение параметров функции в декораторе

Декоратор может перехватывать передаваемые в функцию аргументы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | # определение функции декоратора  def check(input\_func):      def output\_func(\*args):      # через \*args получаем значения параметров оригинальной функции          input\_func(\*args)                # вызов оригинальной функции      return output\_func     # возвращаем новую функцию    # определение оригинальной функции  @check  def print\_person(name, age):      print(f"Name: {name}  Age: {age}")    # вызов оригинальной функции  print\_person("Tom", 38) |

Здесь функция print\_person() принимает два параметра: name (имя) и age (возраст). К этой функции применяется декоратор check()

В декораторе check возвращается локальная функция output\_func(), которая принимает некоторый набор значений в виде параметра \*args - это те значения, которые передаются в оригинальную функцию, к которой применяется декоратор. То есть в данном случае \*args будет содержать значения параметров name и age.

|  |  |
| --- | --- |
| 1  2 | def check(input\_func):      def output\_func(\*args):      # через \*args получаем значения параметров функции input\_func |

Здесь просто передаем эти значения в оригинальную функцию:

|  |  |
| --- | --- |
| 1 | input\_func(\*args) |

В итоге в данном получим следующий консольный вывод

Name: Tom Age: 38

Но что, если в функцию print\_person будет передано какое-то недопустимое значение, например, отрицательный возраст? Одним из преимуществ декораторов как раз является то, что мы можем проверить и при необходимости модифицировать значения параметров. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | # определение функции декоратора  def check(input\_func):      def output\_func(\*args):          name = args[0]          age = args[1]           # получаем значение второго параметра          if age < 0: age = 1     # если возраст отрицательный, изменяем его значение на 1          input\_func(name, age)   # передаем функции значения для параметров      return output\_func    # определение оригинальной функции  @check  def print\_person(name, age):      print(f"Name: {name}  Age: {age}")    # вызов оригинальной функции  print\_person("Tom", 38)  print\_person("Bob", -5) |

args фактически представляет набор значений, и, используя индексы, мы можем получить значения параметров по позиции и что-то с ними сделать. Так, здесь, если значение возраста меньше 0, то устанавливаем 1. Затем передаем эти значения в вызов функции. В итоге здесь получим следующий вывод:

Name: Tom Age: 38

Name: Bob Age: 1

### Получение результата функции

Подобным образом можно получить результат функции и при необходимости изменить его:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | # определение функции декоратора  def check(input\_func):      def output\_func(\*args):          result = input\_func(\*args)   # передаем функции значения для параметров          if result < 0: result = 0   # если результат функции меньше нуля, то возвращаем 0          return result      return output\_func    # определение оригинальной функции  @check  def sum(a, b):      return a + b    # вызов оригинальной функции  result1 = sum(10, 20)  print(result1)          # 30    result2 = sum(10, -20)  print(result2)          # 0 |

Здесь определена функция sum(), которая возвращает сумму чисел. В декораторе check проверяем результат функции и для простоты, если он меньше нуля, то возвращаем 0.

Консольный вывод программы:

30

0