**Функции: объявление и вызов**

Вот мы с вами и подошли к одному из фундаментальных моментов в изучении языка Python – функциям. Что это такое? Смотрите. Например, уже знакомая вам функция print() выводит сообщения в консоль. Фактически же при ее вызове выполняется определенный фрагмент программы, результатом которого и является вывод информации в заданном виде. И это очень удобно. Благодаря наличию таких функций нам не нужно каждый раз писать дублирующие инструкции для выполнения типовых операций. Собственно, это главное предназначение функций – многократное выполнение определенного фрагмента программы.

Язык Python позволяет программисту создавать свои собственные функции. Для этого используется следующий синтаксис:

def <имя функции>([список аргументов]):  
       оператор 1  
        оператор 2  
        …  
        оператор N

Здесь имя функции придумывается программистом подобно именам переменных и, так как функция – это определенное действие, то ее имя следует выбирать как глагол,

Далее, идет набор операторов, которые образуют **тело функции**. Именно они начинают выполнятся при ее вызове.

Давайте зададим простейшую функцию, которая будет выводить «hello» в консоль:

**def** sayHello():

**print**("hello")

Смотрите, мы здесь придумали имя функции «sayHello», записали пустые круглые скобки без аргументов и через двоеточие определили тело функции в виде конструкции print("hello"). Но это лишь определение функции. Самого вызова здесь еще нет и если запустить программу, то ничего не произойдет.

Чтобы вызвать эту функцию, нужно указать ее имя и в конце обязательно поставить круглые скобки даже если мы не передаем ей никаких аргументов:

sayHello()

Эти круглые скобки являются оператором вызова функции с указанным именем. Теперь, при запуске программы в консоли появится сообщение «hello».

Имя функции без круглых скобок – это фактически ссылка на функцию:

**print**( type(sayHello) )

то есть, ссылка на специальный объект, представляющий ту или иную функцию. А раз это ссылка, то мы можем выполнить такую операцию:

f = sayHello

тем самым определить ее синоним и вызвать ее уже через это второе имя:

f()

Как мы говорили в самом начале, функции, как правило, создаются для их многократного вызова. И действительно, мы теперь, можем ее вызывать в любом месте нашей программы необходимое число раз, например, так:

sayHello()

**print**("---------------")

sayHello()

Здесь будет уже два вызова этой функции. И так далее. Причем, обратите внимание, мы вызываем функцию только после ее определения. То есть, если записать ее вызвать в самом начале программы, то возникнет ошибка, т.к. данная функция не была определена.

Также и с функциями: мы их сначала определяем и только потом можем вызывать. Поэтому определение функций обычно идет в самом начале, а потом уже их вызовы в основной программе.

Если нужно определить еще одну функцию, то мы ее можем записать после первой:

**def** myAbs(x):

    x = -x **if**(x<0) **else** x

Имена функций должны быть уникальными (также как и имена переменных), поэтому я назвал ее myAbs, т.к. функция abs уже существует. И предполагаю, что она будет вычислять модуль переданного ей числа. Соответственно, в круглых скобках обозначаю этот аргумент. Если теперь мы ее вызовем:

**print**( myAbs(-5) )

то увидим значение None. Это произошло потому, что функция myAbs явно не возвращает никакого значения. По идее, мы ожидаем возврата переменной x. Для этого нужно записать оператор return, после которого через пробел указываем возвращаемую величину:

**def** myAbs(x):

    x = -x **if**(x<0) **else** x

**return** x

Теперь, при вызове функции, получим ожидаемое значение 5. Как это в деталях работает? Вызывая функцию с аргументом -5, переменная x начинает ссылаться на этот числовой объект. Далее, выполняется тело функции и идет проверка: если x<0, то x=-x (меняем знак числа), иначе x не меняется. Затем, выполняется оператор return и функция myAbs возвращает вычисленное значение x.

Такой подход позволяет передавать функции самые разные значения, например, так:

**print**( myAbs(15) )

a = 100

**print**( myAbs(a) )

И это делает ее работу универсальной – с любыми числовыми данными. Причем, как только встречается оператор return функция завершает свою работу. То есть, если после данного оператора будут идти еще какие-либо конструкции:

**def** myAbs(x):

    x = -x **if**(x<0) **else** x

**return** x

**print**(x)

То при вызове этой функции:

val = myAbs(-5.8)

Ничего в консоль выведено не будет, т.к. вызов был завершен на операторе return. А вот если поставить print до этого оператора: то мы увидим значение 5.8.

**def** myAbs(x):

    x = -x **if**(x<0) **else** x

**print**(x)

**return** x

Используя эту особенность, можно определять такие функции:

**def** isPositive(x):

**if** x >=0:

**return** True

**else**:

**return** False

В данном случае мы будем получать значения True для неотрицательных чисел и False – для отрицательных. И далее, ее можно использовать так:

p = []

**for** a **in** range(-5, 11):

**if** isPositive(a):

        p.append(a)

**print**(p)

В результате, список будет содержать только положительные числа. Правда, в данном случае, функцию можно записать гораздо короче:

**def** isPositive(x):

**return** x >=0

Здесь сам оператор >= будет возвращать значение True или False.

Если нужно создать функцию, принимающую два аргумента, например, для вычисления периметра прямоугольника, то это делается так:

**def** getSquare(w, h):

**return** 2\*(w+h)

То есть, аргументы перечисляются через запятую, а тело функции состоит всего из одного оператора return, в котором сразу выполняются необходимые вычисления.

Вызовем эту функцию:

p = getSquare(10, 5.5)

**print**(p)

И увидим результат ее работы – значение 31,0. При этом, на первое значение 10 ссылается первый аргумент w, а на второе 5.5 – второй аргумент h. Вот так можно определять различное число аргументов у функций.

Далее, при вызове функций мы должны им передавать ровно столько параметров, сколько указано в их определении. Например, вот такие вызовы работать не будут:

myAbs()

myAbs(1, 2)

sayHello("abc")

Здесь указано или слишком много, или слишком мало **фактических параметров**.

Однако у любой функции можно добавить **формальные параметры** со значениями по умолчанию:

**def** sayHello(msg, end="!"):

**print**(msg+end)

И теперь, можно вызвать эту функцию так:

sayHello("Hello")

или так:

sayHello("Hello", "?")

Смотрите, если формальный параметр не указан, то берется его значение по умолчанию. Если же мы его явно задаем, то берется переданное значение. Здесь нужно помнить только одно правило: формальные аргументы должны быть записаны последними в списке аргументов функции. То есть, вот такая запись:

**def** sayHello(end="!", msg):

приведет к синтаксической ошибке.

Теперь, давайте добавим этой функции еще один вот такой формальный параметр:

**def** sayHello(msg, end="!", sep = ": "):

**print**("Message"+sep+msg+end)

И функция будет выводить сообщение в формате: «Message»+sep+msg+end. Вызвать эту функцию мы можем таким образом:

sayHello("Hello", "?", " ")

и каждому параметру здесь будет соответствовать свое значение в соответствии с указанным порядком. А можно ли вызвать эту функцию, указав только первый и последний аргумент? Оказывается да, Python позволяет это делать. Вот таким образом:

sayHello("Hello", sep=" ")

Мы здесь вторым аргументом явно указываем имя формального параметра и присваиваем ему желаемое значение. В результате аргументы msg и sep будут принимать переданные значения, а аргумент end – значение по умолчанию. Это называется **именованные параметры**, когда мы указываем не просто значение, но еще и имя параметра.

Если нам требуется сразу вернуть **несколько значений**, то это можно сделать так. Предположим наша функция будет сразу определять и периметр и площадь прямоугольника:

**def** perAndSq(w, h):

**return** 2\*(w+h), w\*h

И, далее, вызываем ее:

res = perAndSq(2.3, 5)

**print**(res)

получаем результат в виде кортежа из двух чисел. Или, так:

per, sq = perAndSq(2.3, 5)

**print**(per, sq)

Аналогичным образом можно возвращать и списки и словари и вообще любые типы данных.

Далее, в теле функции можно записывать самые разные конструкции языка Python. Например, для возведения числа в целую степень, можно определить такую функцию:

**def** myPow(x, n):

    sx = 1

**while** n > 0:

        sx \*= x

        n -= 1

**return** sx

И, затем, вызвать ее:

p = myPow(3, 5)

**print**(p)

Интересной особенностью Python в определении функций является возможность переопределять уже существующие функции. Например, у нас задана вот такая функция:

**def** sayHello():

**print**("hello")

Тогда ниже мы можем ее переопределить, если укажем то же самое имя:

**def** sayHello():

**print**("------- hello --------")

Теперь, при ее вызове:

sayHello()

увидим выполнение последнего, переопределенного варианта. Если дальше ее переопределить вот так:

**def** sayHello(msg):

**print**(msg)

то все равно будет доступна только одна такая функция, но теперь уже с одним обязательным аргументом:

sayHello("привет мир")

Когда это может пригодиться на практике? Например, если мы хотим определить некоторую функцию в зависимости от условия:

TYPE\_FUNC = True

**if** TYPE\_FUNC:

**def** sayHello():

**print**("hello")

**else**:

**def** sayHello(msg):

**print**(msg)

sayHello()

Здесь при значении переменной TYPE\_FUNC равной True будет определен первый вариант функции, а иначе – второй вариант. Иногда это бывает полезно.

**Элементы функционального подхода к программированию**

При написании программ приветствуется такой подход, который называется **функциональным программированием**. Продемонстрирую его на следующем примере. Предположим, нам нужна функция, которая находит максимальное значение из двух чисел:

**def** max2(a, b):

**if** a > b:

**return** a

**return** b

И вызвать мы ее можем так:

**print**( max2(2, -3) )

Затем, нам потребовалась функция, которая бы находила максимальное из трех чисел. Как ее можно реализовать? Используя идею функционального программирования, это можно сделать следующим образом:

**def** max3(a, b, c):

**return** max2(a, max2(b, c))

И вызвать так:

**print**( max3(2, -3, 5) )

Смотрите, здесь оператор return возвращает значение, которое возвращает функция max2. Но, прежде чем она будет выполнена, вызовется другая функция max2, которая определит максимальное среди чисел b и c. То есть, прежде чем вызвать первую функцию max2 необходимо вычислить ее параметры: первый просто берется их x, а второй вычисляется вложенной функцией max2.

Вот так это работает и вот что из себя представляет элемент функционального подхода к программированию.

Причем, благодаря гибкости языка Python, мы можем вызвать эту функцию и для нахождения максимальной строки:

**print**( max3("ab", "cd", "abc") )

так как строки могут спокойно сравниваться между собой. И вообще, любые величины, которые можно сравнивать на больше и меньше, можно подставлять в качестве аргументов функции max3 и max2.

**Задания для самоподготовки**

**Подвиг 1.**

Объявите функцию, которая принимает список, находит максимальное, минимальное и сумму значений этого списка и выводит результат в виде строки (без кавычек): "Min = v\_min, max = v\_max, sum = v\_sum"

где v\_min, v\_max, v\_sum - вычисленные значения минимального, максимального и суммы значений списка.

**Sample Input:**

8 11 5 -10 12 0

**Sample Output:**

Min = -10, max = 12, sum = 26

**Подвиг 2.**

Объявите функцию с двумя параметрами width и height (ширина и высота прямоугольника), которая выводит сообщение (без кавычек):

"Периметр прямоугольника, равен x"

где x - вычисленный периметр прямоугольника. После объявления функции прочитайте (с помощью функции input) два целых числа, записанных в одну строку через пробел, и вызовите функцию с этими значениями.

**Sample Input:**

8 11

**Sample Output:**

Периметр прямоугольника, равен 38

**Подвиг 3.**

Объявите функцию с именем is\_triangle, которая принимает три стороны треугольника (целые числа) и проверяет, можно ли из переданных аргументов составить треугольник. (Напомню, что у любого треугольника длина третьей стороны всегда должна быть меньше суммы двух других). Если  проверка проходит, вернуть булево значение True, иначе - значение False.

**Подвиг 4.**

Объявите функцию, которая принимает строку (в качестве аргумента) и возвращает False, если длина строки меньше 6 символов. Иначе возвращается значение True.

После объявления функции прочитайте (с помощью функции input) список названий городов, записанных в одну строку через пробел. Затем, используя генератор списка и созданную функцию, сформируйте список из названий городов длиной не менее шести символов на основе введенного исходного списка. Результат отобразите на экране командой:print(\*lst)где lst - итоговый сформированный список.

**Sample Input:**

Москва Уфа Пермь Самара Вологда

**Sample Output:**

Москва Самара Вологда