GeekBrains

Основы Python

Урок 3. Функции

Пришло время познакомиться с механизмом группировки строк кода по блокам — функциям. Они обеспечивают возможность использования этих блоков кода повторно в любой точке программы. Функции могут быть именными и анонимными, принимать параметры и возвращать результат. Важное понятие урока — область видимости переменных. Функции могут документироваться для описания их назначения, принимаемых параметров и возвращаемого результата. Наконец, функции могут быть встроенными и пользовательскими (самописными).

Оглавление

На этом уроке студент:

1. Узнает, как реализовать в программе именные функции.
2. Узнает, для чего в функциях используется оператор return.
3. Узнает, какие бывают аргументы функций и как их передать.
4. Научится использовать в программах анонимные функции.
5. Познакомится с новыми встроенными функциями.
6. Научится использовать функцию range() при реализации циклов.
7. Узнает, что такое область видимости переменных в функции и какие бывают области видимости.
8. Научится документировать функции.

Именные функции

В Python функция представляет собой объект, принимающий аргументы, выполняющий с ними определенные операции и возвращающий результат (значение). Функция определяется с помощью инструкции **def**, после которой следует имя функции. Функции в Python относятся к объектам первого класса, т.е., к элементам, которые могут быть переданы в качестве параметра, возвращены из функции, присвоены переменной.

Пример:

```
def my_sum(arg_1, arg_2):
    return arg_1 + arg_2

print(my_sum(20, 30))
print(my_sum("abra", "kadabra"))
```

Результат:

```
50
abrakadabra
```

В примере представлена простейшая функция, которая принимает два параметра. В зависимости от типов данных параметров, результатом функции может быть число или строка. Инструкция **return** указывает, что функция должна вернуть.

Функция может содержать вложенные функции и возвращать объекты различных типов (списки, словари, функции).

Пример:

```
def ext_func(var_1):
    def int_func(var_2):
        return var_1 + var_2
    return int_func

f_obj = ext_func(200) # f_obj - функция
print(f_obj(300))
```

Результат:

```
500
```

Структура функции определяется спецификой решаемой задачи. Оператор **return** не используется, если функция выполняет некоторые действия, но не возвращает значения. В этом случае возвращаемое значение — **None**.

Пример:

```
def my_func():
    pass

print(my_func())
```

Результат:

```
None
```

В этом примере в теле функции реализован оператор-заглушка. Его использование равноценно отсутствию операции. **Pass** может применяться в тех случаях, когда код на текущий момент не написан.

Оператор return

О назначении данного оператора уже говорилось выше. Функции могут могут принимать данные и возвращать определенный результат после обработки полученных данных. При этом для выхода из функции и передачи результата в точку вызова функции применяется оператор **return**.

return со значением

Если при выполнении логики функции интерпретатор Python встречает оператор **return**, то забирает значение, определенное после данного оператора, и выполняет выход из функции.

Рассмотрим следующий пример (расчет полной площади цилиндра):

Пример:

```
def s_calc():
    r_val = float(input("Укажите радиус: "))
    h_val = float(input("Укажите высоту: "))
    # площадь боковой поверхности цилиндра:
    s_side = 2 * 3.14 * r_val * h_val
    # площадь одного основания цилиндра:
    s_circle = 3.14 * r_val ** 2
    # полная площадь цилиндра:
    s_full = s_side + 2 * s_circle
    return s_full

s_val = s_calc()
print(s_val)
```

Результат:

```
Укажите радиус: 4
Укажите высоту: 3
175.84
```

В данном примере в главную ветку из функции возвращается значение локальной переменной **s_full**, точнее, именно не сама переменная, а ее значение (число, являющееся результатом вычисления площади цилиндра). Подробнее о локальных и глобальных переменных поговорим позднее.

В главной ветке программы значение получает глобальная переменная **s_val**, т.е., выражение **s_val = s_calc()** работает следующим образом:

1. Вызов функции **s_calc()**.
2. Возврат из функции значения.
3. Присвоение полученного значения переменной **s_val**.

Здесь есть один важный момент. Не обязательно присваивать некоторой переменной вычисленное в функции значение. Его можно вывести напрямую на экран.

Пример:

```
print(s_calc())
```

В этом случае число, вычисленное в **s_calc()**, получает непосредственно функция **print()**. И если написать только **s_calc()**, не выполнив присвоение полученных данных некоторой переменной или передав куда-то далее в программе, то синтаксической ошибки не будет, но данные будут потеряны.

return без значения

В функции можно реализовать несколько операторов **return**, но по итогам работы функций может быть выполнен только один. Это оператор **return**, которого поток выполнения программы достигнет первым.

Пример:

```
def s_calc():
    try:
        r_val = float(input("Укажите радиус: "))
        h_val = float(input("Укажите высоту: "))
    except ValueError:
        return
    s_side = 2 * 3.14 * r_val * h_val
    s_circle = 3.14 * r_val ** 2
    s_full = s_side + 2 * s_circle
    return s_full

print(s_calc())
```

В этом примере предусмотрена ситуация, когда пользователь вводит некорректные данные, например, вместо числа (радиус, высота) — строку. При этом реализована обработка исключения, которое возникнет при попытке выполнения арифметической операции умножения со строками. В ветке **except** при возникновении исключения осуществляется выход из функции без вычисления площади цилиндра.

Результат:

```
Укажите радиус: radius
None
```

В результате функция возвращает объект типа **None**. Такое значение оператор **return** возвращает по умолчанию, но можно указать его явно: **return None**.

Важно, что если в функции отсутствует оператор **return**, не значит, что она ничего не возвращает. На самом деле возвращает. Только это значение не присваивается какой-либо переменной и не выводится на экран. В Python любая функция что-то возвращает. Если оператор **return** отсутствует, то возвращаемое значение — **None**.

Возврат набора значений

В Python возможно использование оператора **return**, возвращающего из функции несколько объектов. Достаточно указать их через запятую после оператора **return**.

Пример:

```
def s_calc():
    try:
        r_val = float(input("Укажите радиус: "))
        h_val = float(input("Укажите высоту: "))
    except ValueError:
        return
    s_side = 2 * 3.14 * r_val * h_val
    s_circle = 3.14 * r_val ** 2
    s_full = s_side + 2 * s_circle
    return s_side, s_full

s_side_val, s_full_val = s_calc()
print(f"Площадь боковой пов-ти - {s_side_val}; Полная площадь - {s_full_val}")
```

Результат:

```
Укажите радиус: 4
Укажите высоту: 3
Площадь боковой пов-ти - 75.36; Полная площадь - 175.84
```

Функция **s_calc()** возвращает два значения, присваиваемые переменным **s_side_val** и **s_full_val**. Подобное групповое присвоение — важная характеристика Python.

Смысл в том, что перечисление значений через запятую формирует объект типа кортеж (tuple). При присваивании кортежа сразу набору переменных элементы кортежа сопоставляются переменным. Происходит своего рода распаковка.

Т.е., когда функция возвращает набор объектов, на деле она возвращает объект-кортеж с этими объектами (они упаковываются в кортеж перед возвратом). Если за оператором **return** следует только одна переменная, ее тип сохраняется в исходном состоянии.

Аргументы функций

Функция может принимать любое количество параметров или не принимать их вообще. Параметры могут быть позиционные и именованные, обязательные и необязательные.

Пример:

```
# позиционные параметры
def first_func(var_1, var_2, var_3):
    return var_1 + var_2 + var_3

print(first_func(10, 20, 30))

# именованные параметры
def second_func(var_2, var_1, var_3):
    print(f"var_2 - {var_2}; var_1 - {var_1}; var_3 - {var_3}")

second_func(var_1=10, var_2=20, var_3=30)
```

Результат:

```
60
var_2 - 20; var_1 - 10; var_3 - 30
```

Пример:

```
# обязательные параметры
def first_func(var_1, var_2, var_3):
    return var_1 + var_2 + var_3

print(first_func(10, 20, 30))

# var_2 и var_3 - необязательные параметры
def second_func(var_1, var_2=20, var_3=30):
    return var_1 + var_2 + var_3

print(second_func(10))
```

Результат:

```
60
60
```

Функция может принимать неопределенное число позиционных параметров. В этом случае при описании функции используется конструкция ***args**.

Пример:

```
def my_func(*args):
    return args

print(my_func(10, "text_1", 20, "text_2"))
```

Результат:

```
(10, 'text_1', 20, 'text_2')
```

Из примера следует, что **args** представляет собой кортеж, содержащий переданные в функцию аргументы. С переменной **args** можно выполнять те же операции, что и с кортежем.

Функция может принимать и неопределенное число именованных параметров. Тогда используется конструкция ****kwargs**.

Пример:

```
def my_func(**kwargs):
    return kwargs

print(my_func(el_1=10, el_2=20, el_3="text"))
```

Результат:

```
{'el_1': 10, 'el_2': 20, 'el_3': 'text'}
```

Переменная kwargs хранит словарь, с которым можно выполнять привычные для словаря операции.

Важно, что операторы * и ** в Python можно использовать и с другими именами переменных (т. е. имена **args** и **kwargs** не являются обязательными). Но помните, что хороший стиль программирования подразумевает использование имен **args** и **kwargs**, т. к. сразу становится понятно о назначении таких переменных.

Пример:

```
def my_func(**kparams):
   return kparams
```

```
print(my_func(el_1=10, el_2=20, el_3="text"))
```

Результат:

```
{'el_1': 10, 'el_2': 20, 'el_3': 'text'}
```

Анонимные функции (lambda)

Это функции, содержащие только одно выражение, но выполняющиеся быстрее именных функций. При этом используется оператор **lambda**. При использовании **lambda**-функций их не обязательно присваивать некоторой переменной, как в случае с именными функциями. **lambda**-функции, в отличие от именных, не требуют оператора **return**, в остальном — идентичны именным.

Пример:

```
my_func = lambda p_1, p_2: p_1 + p_2

print(my_func(2, 5))
print(my_func("abra", "kadabra"))

print((lambda p_1, p_2: p_1 + p_2)(2, 5))
print((lambda p_1, p_2: p_1 + p_2)("abra", "kadabra"))

new_func = lambda *args: args
print(new_func(10, 20, 30, 40))
```

Результат:

```
7
abrakadabra
7
abrakadabra
(10, 20, 30, 40)
```

Другое название **lambda**-функции — анонимная или несвязанная.

Еще пример:

```
def named_func(param):
    return param ** 0.5
```

```
print(named_func(100))

square_rt = lambda param: param ** 0.5
print(square_rt(100))
```

Результат:

```
10.0
10.0
```

Еще раз о встроенных функциях

В языке Python предусмотрены встроенные функции. Их логика работы скрыта от разработчиков, а имена зарезервированы. Достаточно знать, какие данные эти функции могут принимать и какой результат возвращать. С частью функций мы уже познакомились ранее (**input()**, **type()**, **int()**, **str()**, **float()**, **bool()**). Переводная версия документации, в которой описаны встроенные функции и их назначение, доступна по ссылке.

Рассмотрим еще две группы встроенных функций.

Функции для операций с символами

Функция	Описание
ord()	Принимает Unicode-символ и возвращает соответствующий код (целое число)
chr()	Принимает целое число и возвращает Unicode-символ, соответствующий переданному числу (коду)
len()	Принимает любой объект-последовательность (строка, набор байтов, список, кортеж) или объект-коллекцию (словарь, множество) и возвращает число элементов последовательности

Пример:

```
print(ord("g"))
print(chr(103))
print(len("abracadabra"))
```

Результат:

```
103
g
11
```

Математические функции

Функция	Описание
abs()	Принимает целое число или число с плавающей точкой. Возвращает абсолютное значение числа (по модулю)
round()	Принимает число с плавающей точкой. Округляет число до ближайшего целого числа. Также может принимать число знаков после запятой, до которых необходимо выполнить округление
divmod()	Принимает два числа, возвращает также два числа (частное и остаток от деления чисел)
pow()	Принимает два числа. Позволяет возвести первое число в указанную степень
max()	Принимает итерируемый объект и возвращает самый большой элемент
min()	Принимает итерируемый объект и возвращает наименьший элемент
sum()	Суммирует элементы последовательности

Пример:

```
# abs()
print(abs(2))
print(abs(-2))
```

Результат:

```
2
2
```

Пример:

```
# round()
print(round(2.6743))
print(round(-2.678))
print(round(2.6743, 2))
print(round(-2.678, 2))
```

Результат:

```
3
-3
2.67
-2.68
```

Пример:

```
# divmod()
print(divmod(4, 2))
print(divmod(5, 2))
```

Результат:

```
(2, 0)
(2, 1)
```

Пример:

```
# pow()
print(pow(2, 4))
```

Результат:

```
16
```

Пример:

```
# max()
iter_obj = [20, 2, 5, 100]
print(max(iter_obj))
```

```
iter_obj = (20, 2, 5, 100)
print(max(iter_obj))
iter_obj = "abrakadabra"
print(max(iter_obj))
```

Результат:

```
100
100
r
```

Пример:

```
# min()
iter_obj = [20, 2, 5, 100]
print(min(iter_obj))
iter_obj = (20, 2, 5, 100)
print(min(iter_obj))
iter_obj = "abrakadabra"
print(min(iter_obj))
```

Результат:

```
2
2
a
```

Пример:

```
# sum()
iter_obj = [20, 2, 5, 100]
print(sum(iter_obj))
iter_obj = (20, 2, 5, 100)
print(sum(iter_obj))
```

Результат:

```
127
127
```

Функция range() для многократно выполняемых действий

Данная функция отвечает за генерацию набора чисел в пределах указанного диапазона. Для выбора чисел из диапазона можно использовать еще один параметр — шаг.

Пример:

```
print(list(range(7))) # целые числа в диапазоне [0, 7)
print(list(range(2, 8))) # целые числа в диапазоне [2, 8)
print(list(range(1, 9, 2))) # целые числа в диапазоне [1, 9) с шагом 2
print(list(range(1, -7, -2))) # целые числа в диапазоне [1, -7) с шагом -2
print(list(range(0))) # целые числа в диапазоне (0, 0)
print(list(range(1, 0))) # целые числа в диапазоне (1, 0)
```

Результат:

```
[0, 1, 2, 3, 4, 5, 6]
[2, 3, 4, 5, 6, 7]
[1, 3, 5, 7]
[1, -1, -3, -5]
[]
[]
```

Функция range() может использоваться в циклах:

Пример:

```
for el in range(4, 20, 4):
    res = el / 2
    print(f"Результат деления {el} на 2: {int(res)}")
```

Результат:

```
Результат деления 4 на 2: 2
Результат деления 8 на 2: 4
Результат деления 12 на 2: 6
Результат деления 16 на 2: 8
```

Области видимости переменных в функциях

Понятие «Область видимости» определяет, когда и в каких точках программы разработчик может применять свои пользовательские объекты (переменные, функции). В Python доступны следующие области видимости: локальная, глобальная, нелокальная.

Локальная область видимости

Переменная, объявленная в рамках функции, по умолчанию имеет локальную область видимости. Рассмотрим еще раз пример, представленный ранее.

Пример:

```
def full_s_calc():
    r_val = float(input("Укажите радиус: "))
    h_val = float(input("Укажите высоту: "))
    s_side = 2 * 3.14 * r_val * h_val
    s_circle = 3.14 * r_val ** 2
    s_full = s_side + 2 * s_circle
    return s_full

s_val = full_s_calc()
print(s_val)
print(s_side)
```

Результат:

```
Укажите радиус: 4
Укажите высоту: 3
175.84
Traceback (most recent call last):
  File "my_file.py", line 11, in <module>
    print(s_side)
NameError: name 's_side' is not defined
```

В этом примере попытки обратиться к переменным **r_val**, **h_val**, **s_side**, **s_circle** приведут к аварийному завершению работы программы, т. к. они локальные и доступны только в пределах функции **full_s_calc()**. Для решения этой проблемы необходимо перевести нужные локальные переменные в глобальную область видимости.

Глобальная область видимости

Оператор **global** позволяет определить глобальную область видимости для переменной, объявленной в рамках функции.

Пример:

```
def full_s_calc():
    global r_val, h_val, s_side, s_circle
    r_val = float(input("Укажите радиус: "))
    h_val = float(input("Укажите высоту: "))
    s_side = 2 * 3.14 * r_val * h_val
    s_circle = 3.14 * r_val ** 2
    s_full = s_side + 2 * s_circle
    return s_full

s_val = full_s_calc()
print(s_val)
print(s_circle)
```

Результат:

```
Укажите радиус: 4
Укажите высоту: 3
175.84
50.24
```

Не локальная область видимости

Перевод переменной в область видимости объемлющей функции.

Пример:

```
def ext_func():
    my_var = 0
    def int_func():
        my_var += 1
        return my_var
    return int_func

func_obj = ext_func()
print(func_obj)
print(func_obj())
print(func_obj())
print(func_obj())
```

Результат:

```
UnboundLocalError: local variable 'my_var' referenced before assignment
```

Ошибка возникает из-за того, что Python пытается увеличить значение переменной **my_var** на единицу, но исходное значение этой переменной неопределено. Т.е., оно как бы определено, но вне области видимости функции **int_func()** и потому по умолчанию недоступно.

Решение проблемы — перевод переменной **my_var** в не локальную зону видимости (в зону видимости объемлющей функции).

Пример:

```
def ext_func():
    my_var = 0
    def int_func():
        nonlocal my_var
        my_var += 1
        return my_var
    return int_func

func_obj = ext_func()
print(func_obj)
print(func_obj())
print(func_obj())
print(func_obj())
```

Результат:

```
<function ext_func.<locals>.int_func at 0x0000009E70C658C8>
1
2
3
```

Еще один интересный момент. В данном примере объемлющая функция **ext_func()** возвращает нам объект-функцию:

```
<function ext_func.<locals>.int_func at 0x0000009E70C658C8>
```

Следовательно, переменная **func_obj** начинает ссылаться на объект-функцию, и значит, допустим вызов этой функции:

```
func_obj()
```

Документирование кода функций

Качественное документирование кода повышает его читаемость и ускоряет понимание, что особенно важно в командной разработке. Рекомендуемый формат документирования кода описывает стандарт PEP-257. Это не правила, не синтаксис, а набор рекомендуемых соглашений для разработчиков.

В Python принято сопровождать функции строками документации. Они даже применяются при описании переменных, логики работы классов, а также логики работы целых файлов (в Python они называются модулями).

Однострочные строки документации

Однострочники, как правило, используются для описания явных элементов логики программы. Умещаются в одной строке, идентифицируются тройными кавычками в начале и конце. Пустые строки до и после однострочников не ставятся.

Пример:

```
def get_path():
    """Возвращает путь до директории."""
    global my_path
    if my_path: return my_path
```

Однострочник желательно не использовать в качестве подписи параметров функции.

Пример:

```
def my_func(param_1, param_2):
    """my_func(param_1, param_2) -> tuple"""
```

Лучше такой вариант:

```
def my_func(param_1, param_2):
    """Выполняет обработку параметров и возвращает кортеж."""
```

Многострочные строки документации

Многострочники принято реализовывать следующим образом: однострочник, пустая строка и далее несколько строк более подробного описания. Первая строка может находиться с кавычками на одной строке или на следующей. Желательно везде придерживаться одного с первой строкой отступа. В строках документации функций желательно описать основное поведение функции, а также аргументы (позиционные или именованные) и возвращаемые значения, ограничения на ее применение.

Пример:

```
def my_func(p_1=0, p_2=1):
    """Возвращает частное от деления.

    Именованные параметры:
    p_1 -- делимое (по умолчанию 0.0)
    p_2 -- делитель (по умолчанию 1.0)

    """
    return p_1/p_2
```

Алгоритм создания функции

Благодаря функциям разработчик получает возможность многократного использования кода, что повышает модульность проекта, упрощает его последующую модификацию. Для создания функции можно использовать алгоритм, который рассмотрим на примере определения площади прямоугольника:

1) Определить название функции. Оно должно быть информативным, чтобы было понятно назначение функции.
2) Определить в строках документации назначение функции, типы данных ее параметров и тип данных результата. Можно также указать пример вызова функции с параметром и возвращаемый результат.
3) Определить информативные имена параметров, передаваемых в функцию, написать тело функции с возвращаемым результатом (при необходимости).

Пример:

```
def rectangle_area_calc(length, width):
    """
    Возвращает площадь прямоугольника по длине и ширине

    (number, number) -> number
```

```
    >>> rectangle_area_calc(10, 10)
    100
    """
    return length * width
```

Сводная таблица «Функции builtins»

Интерпретатор Python предоставляет разработчику ряд встроенных функций:

Функция	Описание
`abs()`	Возвращает абсолютное значение числа (целого или с плавающей точкой)
`all()`	Возвращает True, если все элементы итерируемого объекта являются истинными
`any()`	Возвращает True, если какой-либо элемент итерируемого объекта равен True
`ascii()`	Возвращает строку, содержащую печатаемое представление объекта
`bin()`	Преобразует целое число в двоичную строку с префиксом **0b**
`bool()`	Возвращает логическое значение (True или False)
`breakpoint()`	Перемещает в отладчик
`bytearray()`	Возвращает массив байтов
`bytes()`	Возвращает объект bytes, представляющий собой неизменяемый набор целых чисел в диапазоне от 0 до 256
`callable()`	Возвращает True, если аргумент функции поддерживает возможность вызова
`chr()`	Возвращает символ, соответствующий коду Unicode (целому числу)
`classmethod()`	Преобразует функцию в метод класса, а не только его экземпляра
`compile()`	Компилирует исходный код в объект кода, либо в объект абстрактного синтаксического дерева
`complex()`	Помогает преобразовать в комплексное число

`delattr()`	Удалить из объекта указанный атрибут
`dict()`	Вызов конструктора, создающего словарь
`dir()`	Возвращает список имен, определенных в модуле
`divmod()`	Возвращает частное-остаток от деления чисел
`enumerate()`	Возвращает генератор пар счетчик-элемент для элементов указанного набора
`eval()`	Выполняет разбор и запуск указанного выражения
`exec()`	Выполняет переданный в функцию код
`filter()`	Выполняет фильтрацию элементов объекта
`float()`	Преобразует объект к числу с плавающей точкой
`format()`	Форматирование переданного объекта
`frozenset()`	Создание неизменяемого множества
`getattr()`	Получить значение атрибута объекта
`globals()`	Получить словарь с глобальной таблицей символов модуля
`hasattr()`	Возвращает True, если объект содержит указанный атрибут
`hash()`	Получить хеш объекта
`help()`	Вызов встроенной справки
`hex()`	Возвращает целое число в виде строки в шестнадцатеричном формате
`id()`	Получить идентификатор объекта
`input()`	Запросить строковый пользовательский ввод
`int()`	Преобразовать объект в целочисленный формат
`isinstance()`	Возвращает True, если переданный объект является экземпляром указанного класса
`issubclass()`	Возвращает True, если указанный класс является подклассом другого класса
`iter()`	Получить объект итератора

`len()`	Получить количество элементов в объекте
`list()`	Создание объекта-списка
`locals()`	Получить текущую локальную таблицу символов в виде словаря
`map()`	Применить указанную функцию к каждому элементу коллекции
`max()`	Возвращает элемент с максимальным значением из набора
`memoryview()`	Возвращает объект — представление в памяти для переданного аргумента
`min()`	Возвращает элемент с наименьшим значением из набора
`next()`	Получить очередной элемент итератора
`object()`	Создать новый базовый класс
`oct()`	Возвращает заданное целое число в восьмеричном формате в виде строки
`open()`	Открыть файл и вернуть его объект
`ord()`	Вернуть числовой код символа
`pow()`	Возвести число в степень
`print()`	Отправить указанный объект текстовым потоком
`property()`	Вернуть свойство
`range()`	Определить диапазон с шагом (при необходимости)
`repr()`	Получить для переданного объекта формальное строковое представление
`reversed()`	Получить обратный итератор для переданного набора значений
`round()`	Получить число с плавающей точкой. Округленное до нужного числа знаков после запятой
`set()`	Создать изменяемое множество
`setattr()`	Связать с объектом указанный атрибут
`slice()`	Выполнить срез в последовательности
`sorted()`	Вернуть список, состоящий из элементов объекта, поддерживающего итерирование

`staticmethod()`	Определить указанную функцию в качестве статичного метода
`str()`	Преобразовать объект к строковому типу
`sum()`	Выполнить суммирование элементов объекта и вернуть результат
`super()`	Вернуть объект-посредник, перенаправляющий вызовы методов родителю
`tuple()`	Создать кортеж
`type()`	Определить тип объекта
`vars()`	Получить словарь из атрибута __dict__ объекта
`zip()`	Вернуть итератор для кортежей, где каждый i-й кортеж содержит i-й элемент каждой из коллекций

Практическое задание

1) Реализовать функцию, принимающую два числа (позиционные аргументы) и выполняющую их деление. Числа запрашивать у пользователя, предусмотреть обработку ситуации деления на ноль.
2) Реализовать функцию, принимающую несколько параметров, описывающих данные пользователя: имя, фамилия, год рождения, город проживания, email, телефон. Функция должна принимать параметры как именованные аргументы. Реализовать вывод данных о пользователе одной строкой.
3) Реализовать функцию **my_func()**, которая принимает три позиционных аргумента, и возвращает сумму наибольших двух аргументов.
4) Программа принимает действительное положительное число **x** и целое отрицательное число **y**. Необходимо выполнить возведение числа **x** в степень **y**. Задание необходимо реализовать в виде функции **my_func(x, y)**. При решении задания необходимо обойтись без встроенной функции возведения числа в степень.

   Подсказка: попробуйте решить задачу двумя способами. Первый — возведение в степень с помощью оператора **. Второй — более сложная реализация без оператора **, предусматривающая использование цикла.

5) Программа запрашивает у пользователя строку чисел, разделенных пробелом. При нажатии Enter должна выводиться сумма чисел. Пользователь может продолжить ввод чисел, разделенных пробелом и снова нажать Enter. Сумма вновь введенных чисел будет добавляться к уже подсчитанной сумме. Но если вместо числа вводится специальный символ, выполнение программы завершается. Если специальный символ введен после нескольких

чисел, то вначале нужно добавить сумму этих чисел к полученной ранее сумме и после этого завершить программу.

6) Реализовать функцию **int_func()**, принимающую слово из маленьких латинских букв и возвращающую его же, но с прописной первой буквой. Например, **print(int_func('text'))** -> Text.

   Продолжить работу над заданием. В программу должна попадать строка из слов, разделенных пробелом. Каждое слово состоит из латинских букв в нижнем регистре. Сделать вывод исходной строки, но каждое слово должно начинаться с заглавной буквы. Необходимо использовать написанную ранее функцию **int_func()**.

Дополнительные материалы

1) Встроенные функции.
2) Функции в Python.
3) Функция с переменным количеством аргументов в Python: *args и **kwargs.
4) Как определять функции в Python 3.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1) Язык программирования Python 3 для начинающих и чайников.
2) Программирование в Python.
3) Учим Python качественно(habr).
4) Самоучитель по Python.
5) Лутц М. Изучаем Python. — М.: Символ-Плюс, 2011 (4-е издание).