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:

```
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:

```
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:

```
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:

```
имя_функции ([параметры])
```

Например, определим и вызовем функцию:

```
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:

```
def say_hello(): print("Hello")
say_hello()
```

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

```
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:

```
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:

```
def main():
  say_hello()
  say_goodbye()
def say_hello():
  print("Hello")
def say_goodbye():
  print("Good Bye")
```

```
# Вызов функции main

main()
```

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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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 именованными.

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

```
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.

```
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 именованные параметры.

```
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 можно передать значение только по позиции.

Параметр company является именованным, 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:

```
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.

*args и **kwargs в Python

Позиционные и именованные аргументы

Для того чтобы разобраться с *args и **kwargs, нам нужно освоить концепции позиционных (positional) и именованных (keyword) аргументов.

Сначала поговорим о том, чем они отличаются. В простейшей функции мы просто сопоставляем позиции аргументов и параметров. Аргумент №1 соответствует параметру №1, аргумент №2 — параметру №2 и так далее.

```
def printThese(a,b,c):
    print(a, "is stored in a")
    print(b, "is stored in b")
    print(c, "is stored in c")
printThese(1,2,3)
"""
1 is stored in a
```

```
2 is stored in b
3 is stored in c
"""
```

Для вызова функции необходимы все три аргумента. Если пропустить хотя бы один из них — будет выдано сообщение об ошибке.

```
def printThese(a,b,c):
  print(a, "is stored in a")
  print(b, "is stored in b")
  print(c, "is stored in c")
printThese(1,2)
"""
TypeError: printThese() missing 1 required positional argument: 'c'
"""
```

Если при объявлении функции назначить параметру значение по умолчанию — указывать соответствующий аргумент при вызове функции уже необязательно. Параметр становится опциональным.

```
def printThese(a,b,c=None):
  print(a, "is stored in a")
  print(b, "is stored in b")
  print(c, "is stored in c")
printThese(1,2)
"""
1 is stored in a
2 is stored in b
None is stored in c
"""
```

Опциональные параметры, кроме того, можно задавать при вызове функции, используя их имена.

В следующем примере установим три параметра в значение по умолчанию None и взглянем на то, как их можно назначать, используя их имена и не

обращая внимания на порядок следования аргументов, применяемых при вызове функции.

```
def printThese(a=None,b=None,c=None):
  print(a, "is stored in a")
  print(b, "is stored in b")
  print(c, "is stored in c")
printThese(c=3, a=1)
"""
1 is stored in a
None is stored in b
3 is stored in c
"""
```

Оператор «звёздочка»

Оператор * чаще всего ассоциируется у людей с операцией умножения, но в Python он имеет и другой смысл.

Этот оператор позволяет «распаковывать» объекты, внутри которых хранятся некие элементы. Вот пример:

```
a = [1,2,3]
b = [*a,4,5,6]
print(b) # [1,2,3,4,5,6]
```

Тут берется содержимое списка a, распаковывается, и помещается в список b.

Как пользоваться *args и **kwargs

Итак, мы знаем о том, что оператор «звёздочка» в Python способен «вытаскивать» из объектов составляющие их элементы. Знаем мы и о том, что существует два вида параметров функций. Вполне возможно, что вы уже додумались до этого сами, но я, на всякий случай, скажу об этом. А именно, *args — это сокращение от «arguments» (аргументы), а **kwargs — сокращение от «keyword arguments» (именованные аргументы).

Каждая из этих конструкций используется для распаковки аргументов соответствующего типа, позволяя вызывать функции со списком аргументов переменной длины. Например — создадим функцию, которая умеет выводить результаты, набранные учеником в тесте:

```
def printScores(student, *scores):
  print(f"Student Name: {student}")
  for score in scores:
    print(score)
printScores("Jonathan",100, 95, 88, 92, 99)
"""
Student Name: Jonathan
100
95
88
92
99
"""
```

При объявлении функции конструкция *args не используется. Вместо неё у меня — *scores. Ошибки здесь нет. Дело в том, что «args» — это всего лишь набор символов, которым принято обозначать аргументы. Самое главное тут — это оператор *. А то, что именно идёт после него, особой роли не играет. Благодаря использованию * мы создали список позиционных аргументов на основе того, что было передано функции при вызове.

После того, как мы разобрались с *args, с пониманием **kwargs проблем быть уже не должно. Имя, опять же, значения не имеет. Главное — это два символа **. Благодаря им создается словарь, в котором содержатся именованные аргументы, переданные функции при её вызове.

```
def printPetNames(owner, **pets):
  print(f"Owner Name: {owner}")
  for pet,name in pets.items():
    print(f"{pet}: {name}")
```

printPetNames("Jonathan", dog="Brock", fish=["Larry", "Curly", "Moe"], turtle="Shelldon")
"""
Owner Name: Jonathan
dog: Brock
fish: ['Larry', 'Curly', 'Moe']
turtle: Shelldon
"""

Итоги

Вот несколько советов, которые помогут вам избежать распространённых проблем, возникающих при работе с функциями, и расширить свои знания:

Используйте общепринятые конструкции *args и **kwargs для захвата позиционных и именованных аргументов.
Конструкцию **kwargs нельзя располагать до *args. Если это сделать — будет выдано сообщение об ошибке.
Остерегайтесь конфликтов между именованными параметрами и **kwargs, в случаях, когда значение планируется передать как **kwarg-аргумент, но имя ключа этого значения совпадает с именем именованного параметра.
Оператор *можно использовать не только в объявлениях функций, но и при их вызове.

