09-Python Basic

Functions, Parameters, Arguments, Generator, Iterator

Posted on September 12, 2021
Last updated on February 2, 2023

9. Functions

If a group of statements is repeatedly required then it is not recommended to write these statements everytime seperately.

We have to define these statements as a single unit and we can call that unit any number of times based on our requirement without rewriting. This unit is nothing but function.

- The main advantage of functions is `code Reusability.`
- Python supports 2 types of functions
    1. Built in Functions
    2. User Defined Functions

9.1 Built in Functions

The functions which are coming along with Python software automatically,are called built in functions or pre defined functions

```
Eg:
id() , type(), input(),
map(), reduce(), filter(), lambda(), eval()
```

9.2 User Defined Functions

The functions which are developed by programmer explicitly according to business requirements ,are called user defined functions.

```
# Syntax
def function_name(parameters):
    """ doc string"""
    ----
    -----
    return value
```

9.3 Return Statement

Function can take input values as parameters and executes business logic, and `returns output` to the caller with return statement.

```
def add(x,y):
    return x+y

result=add(10,20)
print("The sum is",result)
```

```
# Returning multiple values
def sum_sub(a,b):
    sum=a+b
    sub=a-b
    return sum,sub
x,y=sum_sub(100,50)
print(x,y)
```

9.4 Parameters

Parameters are `inputs to the function`. If a function contains parameters,then at the time of calling,compulsory we should provide values otherwise,otherwise we will get error.

A parameter is the `variable` defined within the parentheses `during function definition`.

```
# Here a,b are the parameters
def sum(a,b):
    print(a+b)
sum(1,2)
```

9.5 Arguments

An argument is a value that is passed to a function when it is called. It might be a `variable`, value or object passed to a function or method as input. They are written when we are `calling the function`

```
def f1(a,b):
    ------
    ------
    ------
f1(10,20)
```

There are 4 types are actual arguments are allowed in Python.

1. Positional arguments
2. Keyword arguments
3. Default arguments
4. Variable length arguments

Positional arguments

The number of arguments and position of arguments must be matched. If we change the order then result may be changed.

```
def sub(a,b):
    print(a-b)
sub(100,200)  # -100
sub(200,100)  # 100
```

Keyword arguments

Here the order of arguments is not important but number of arguments must be matched.

```
def wish(name, msg):
    print("Hello", name, msg)

wish(name="Amrit", msg="Good Morning")
# Hello Amrit Good Morning
wish(msg="Good Morning", name="Amrit")
# Hello Amrit Good Morning
```

- We can use both positional and keyword arguments simultaneously.
- But first we have to take positional arguments and then keyword arguments,otherwise we will get syntaxerror.

```
def wish(name, msg):
    print("Hello", name, msg)

wish("Amrit","GoodMorning")  # valid
wish("Amrit",msg="GoodMorning") # valid
wish(name="Amrit","GoodMorning") # invalid
# SyntaxError: positional argument follows keyword argument
```

Default Arguments

Sometimes we can provide default values for our positional arguments.

```
def wish(name="Guest"):
    print("Hello", name, "GoodMorning")

wish("Amrit")
# Hello Amrit Good Morning
wish()
# Hello Amrit Good Morning
```

- SyntaxError: non-default argument follows default argument

```
def wish(name="Guest", msg="Good Morning"): # valid
def wish(name, msg="Good Morning"):      # valid
def wish(name="Guest", msg):  # Invalid
```

Variable length arguments

Sometimes we can `pass variable number of arguments` to our function, such type of arguments are called variable length arguments.

We can declare a variable length argument `with * symbol` as follows

```
def sum(*n):
    total=0
    for n1 in n:
        total=total+n1
    print(total)

sum()  # 0
sum(10)  # 10
sum(10, 20)  # 30
sum(10, 20, 30, 40)  # 100
```

We can mix variable length arguments with positional arguments.

```
def f1(n1,*s):
    print(n1, s)

f1(10)
# 10 ()
f1(10,20)
# 10 (20,)
f1(10, "A", 30)
# 10 ('A', 30)
```

We can declare `key word variable length arguments` also. For this we have to `use **`.

```
def display(**kwargs):
    print(kwargs)

display(n1=10)
# {'n1': 10}

display(rno=100,name="Amrit")
# {'rno': 100, 'name': 'Amrit'}
```

```
def test7(*args , **kwargs) :
    return args , kwargs

test7(2,3,4,5,a= 34, b = 98)
# ((2, 3, 4, 5), {'a': 34, 'b': 98})
```

9.6 Anonymous/lambda Functions

Sometimes we can declare a function without any name,such type of nameless functions are called `anonymous` functions

The main purpose of anonymous function is just for `instant` use(i.e for `one time usage`)

AKA `lambda functions` or `shorthand function`

```
# Syntax
lambda argument_list : expression
```

```
print((lambda x: x + 1)(2)) # 3

s = lambda n:n*n
print(s(4)) # 64

s = lambda a,b:a+b
print(s(2, 3)) # 5

s = lambda a,b:a if a>b else b
print(s(10,20)) # 20
print(s(100,200)) # 200

print((lambda x, y, z=3: x + y + z)(1, 2))
```

Note

Lambda Function internally returns expression value and we are `not required to write return statement explicitly`

Sometimes we can pass function as argument to another function. In such cases lambda functions are best choice.

We can use lambda functions very commonly with filter(), map() and reduc() functions, because these functions expect function as argument.

9.7 map() function

`map()` function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)

```
# str to int
num = ['1', '2', '3']
res = list(map(int, num))
print(res)

# Square
def square(number):
        return number ** 2

numbers = [1, 2, 3, 4, 5]
print(list(map(square, numbers)))

# Square using map and lambda
print(list(map(lambda x:x*x, numbers)))


first_it = [1, 2, 3]
second_it = [4, 5, 6, 7]
print(list(map(pow, first_it, second_it)))

# x- y
print(list(map(lambda x, y: x - y, [2, 4, 6], [1, 3, 5])))
# [1, 1, 1]

print(list(map(lambda x, y, z: x + y + z, [2, 4], [1, 3], [7, 8])))
# [10, 15]



# Eg 5
string_it = ["processing", "strings", "with", "map"]
print(list(map(str.upper, string_it)))
# ['PROCESSING', 'STRINGS', 'WITH', 'MAP']


with_spaces = ["processing ", "  strings", "with   ", " map   "]
print(list(map(str.strip, with_spaces)))
# ['processing', 'strings', 'with', 'map']

with_dots = ["processing..", "...strings", "with....", "..map.."]
print(list(map(lambda s: s.strip("."), with_dots)))
# ['processing', 'strings', 'with', 'map']
```

```
# Eg 6
import re
def remove_punctuation(word):
        return re.sub(r'[!?.:;,"()-]', "", word)

text = """Some people, when ,,confronted... with a problem, think"""
words = text.split()
print(list(map(remove_punctuation, words)))
```

9.8 reduce() function

reduce() function `reduces sequence of elements` into a single element by applying the specified function. reduce(function, sequence) reduce() function present in functools module and hence we should write import statement.

```
from functools import *
l = [10,20,30,40,50]
result=reduce(lambda x,y:x+y,l)
print(result) # 150
```

```
l = [1,2,3,4,5,4]
reduce(lambda x , y , z : x+y+z , l) # TypeError

reduce(lambda x , y : x+y , [1])
# 1

reduce(lambda x,y : x if x> y else y , l)
# 5
```

9.9 filter() function

We can use filter() function to `filter values` from the given `sequence` based on some condition.

filter(function, sequence)

where function argument is responsible to perform conditional check sequence can be list or tuple or string.

```
#  Working of filter()
def check_even(number):
    if number % 2 == 0:
          return True
    return False

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(list(filter(check_even, numbers)))
# Output: [2, 4, 6, 8, 10]
```

```
# Using Lambda Function Inside filter()
numbers = [1, 2, 3, 4, 5, 6, 7]

print(list(filter(lambda x: (x%2 == 0), numbers)))
#[2, 4, 6]


l2 = ["sudh" , "pwskills" , "kumar" , "bengalore" , "krish"]
print(list(filter(lambda x : len(x) < 6 , l2)))
```

9.10 eval() function

- The eval function evaluates the String like a `python expression` and returns the result as an integer.
- `eval(expression, globals=None, locals=None)`
    - The globals must be represented as a dictionary and the locals as a mapped object.

```
x = 1
print(eval('x + 1'))
print(eval("1024 + 1024"))
```

```
code = compile("5 + 4", "<string>", "eval")
print(eval(code))
print()

x = 100
print(eval("x + 10", {"x": x}))

print(eval("x + y", {"x": x, "y": x}))
```

9.10 Generators

Generator is a function which is responsible to `generate a sequence of values`.

We can write generator functions just like ordinary functions, but it uses `yield keyword` to return values.

```
def simpleGeneratorFun():
  yield 1
  yield 2
  yield 3

for value in simpleGeneratorFun():
  print(value, end=' ')
# 1 2 3

x = simpleGeneratorFun()
print(list(x))
# [1, 2, 3]
```

Advantages of Generator Functions

1. when compared with class level iterators, generators are very easy to use
2. Improves `memory utilization` and `performance`.
3. Generators are best suitable for reading data from large number of large files
4. Generators work great for web scraping and crawling.

We will `get MemoryError` in this case because all these values are required to store in the memory.

```
# Normal Collection
l=[x*x for x in range(10000000000000000)]
print(l[0])
```

We `won't get any MemoryError` because the values won't be stored at the beginning

```
# Generators
g=(x*x for x in range(10000000000000000))
print(next(g))
# 0
```

9.11 Iterable and Iterator in Python

- Iteration means to access each item of something one after another generally `using a loop`
- list, tuples, dictionaries, etc. –> all are iterables
- One important property it has
  - an `__iter__()` method or `iter()` method
  - which allows any iterable to return an iterator object.
- `iter()` and `next()`

```
# list of cars
cars = ['Audi', 'BMW', 'Jaguar', 'Kia', 'MG', 'Skoda']

# get iterator object using the iter() / __iter__() method
cars_iter = iter(cars)
cars_iter = cars.__iter__()

# use the next / __next__() method to iterate through the list
print(next(cars_iter))
# OP: Audi

print(cars_iter.__next__())
# OP: BMW
```

Create an Iterator Class

- To create an object/class as an iterator you have to implement the methods `__iter__() and __next__()` to your object.
- To prevent the iteration to go on forever, we can use the `StopIteration` statement.

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
      if self.a <= 3:
          x = self.a
          self.a += 1
          return x
      else:
          raise StopIteration

my_class = MyNumbers()
my_iter = iter(my_class)

print(next(my_iter))
print(next(my_iter))
print(next(my_iter))
print(next(my_iter))
print(next(my_iter))
```

.

.

.

.

.

.