Introduction to Python - Functions

May 27, 2020

[45]:
```
from IPython.core.display import display, HTML, Image
display(HTML("<style>.container { width:99% !important; }</style>"))
Image('https://www.python.org/images/python-logo.gif')
```

```
<IPython.core.display.HTML object>
```

[45]:
```
<IPython.core.display.Image object>
```

1 Agenda

- What is a function in Python?
  - Syntax of Function
  - Example of a function.
  - How to call a function in python?
- Python Function Arguments (Parameters or Arguments?)
  - Positional Arguments
  - Keyword Arguments
  - Default Arguments
  - Arbitrary arguments
- Python local, global variables and their scope and lifetime
- Understanding variable passing to functions in detail
- Void functions and return statement
- Types of functions
  - Built-in functions
  - User-defined functions
  - Recursive functions
  - Python Anonymous/Lambda Function
    - Use of Lambda Function in python
- Why we need to have good understanding of functions?

1.1 What is a function?

$y = F(x_1, x_2)$

$y = F(x_1, x_2, x_3) = \sqrt{(x_1 * x_2)} + (1 + x_1)^2 - 2 * x_3^3$

- Functions are building blocks of a Python program.

- Functions contain a group of related statements that perform a specific task.
- Functions allow reusing a group of statements which might be required to run multiple times in Python.
- Functions allow user to modularize a Python code

1.2 How to define a function?

1.2.1 Syntax

[52]:
```
def function_name(arguments):
    """
    Docstring :
    This is a sample function.
    Here you can write details about the function for future reference.
    More details coming up.
    """

    print(f'Welcome to functions. This is a {arguments}.')
    ...
    ...
    ...
    return True
```

[48]:
```
?function_name
```

[49]:
```
def compute_sum(l):
    """
    This function takes a list as argument and computes the sum of all elements␣
 ↪in the list.
    After computing the sum.
    """

    list_sum = 0
    for element in l:
        list_sum += element

    print(f'Sum of the list is {list_sum}.')
```

[50]:
```
?compute_sum
```

1.2.2 Calling/Invoking a function

[53]:
```
function_name('Python function')
```

```
Welcome to functions. This is a Python function.
```

[53]:
```
True
```

[54]:
```
l = [2, 4, 6, 8, 10]

compute_sum(l)
```

```
Sum of the list is 30.
```

[57]:
```
compute_sum()
```

```
␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-57-8be732bac68c> in <module>
   ----> 1 compute_sum()


       TypeError: compute_sum() missing 1 required positional argument: 'l'
```

1.2.3 Docstring

[55]:
```
print(function_name.__doc__)
```

```
    Docstring :
    This is a sample function.
    Here you can write details about the function for future reference.
    More details coming up.
```

[56]:
```
print(compute_sum.__doc__)
```

```
    This function takes a list as argument and computes the sum of all elements
in the list.
    After computing the sum.
```

[ ]:

1.3 Function Arguments/Parameters

The functions you define can take multiple arguments.

That can sometimes be useful, and you'll occasionally write such functions.

1.3.1 Positional Arguments

These are specified as a comma-separated list of parameters inside the parentheses.

User need to correctly follow the order of parameters.

[73]:
```
def record_user_information(name, occupation, age):

    print(f"Name = {name}")
    print(f"Occupation = {occupation}")
    print(f"Age = {age}")
```

[74]:
```
record_user_information('Nimo', 'Student', 22)
```

```
Name = Nimo
Occupation = Student
Age = 22
```

[75]:
```
record_user_information('Nimo', 23, 'Student')
```

```
Name = Nimo
Occupation = 23
Age = Student
```

[61]:
```
record_user_information('Nimo', 'Student')
```

```
      ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-61-7c69be76cfee> in <module>
   ----> 1 record_user_information('Nimo', 'Student')


       TypeError: record_user_information() missing 1 required positional␣
↪argument: 'age'
```

[62]:
```
record_user_information('Nimo', 'Student', 23, 'Delhi')
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-62-2b5f82db18f1> in <module>
----> 1 record_user_information('Nimo', 'Student', 23, 'Delhi')

TypeError: record_user_information() takes 3 positional arguments but 4␣
↪were given
```

[ ]:
```

```

1.3.2 Keyword Arguments

Second way of passing parameters is in the form **keyword** = **value**

Each **keyword** must match a parameter in the Python function definition.

[76]:
```
def record_user_information(name, occupation, age):

    print(f"Name = {name}")
    print(f"Occupation = {occupation}")
    print(f"Age = {age}")
```

[77]:
```
record_user_information(name = 'Nimo', occupation = 'Student', age = 22)
```

```
Name = Nimo
Occupation = Student
Age = 22
```

[78]:
```
record_user_information(occupation = 'Student', age = 22, name = 'Nimo')
```

```
Name = Nimo
Occupation = Student
Age = 22
```

[67]:
```
record_user_information(occupation = 'Student', age = 22)
```

```
␣
↪---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-67-0272edf60cd9> in <module>
----> 1 record_user_information(occupation = 'Student', age = 22)


TypeError: record_user_information() missing 1 required positional␣
↪argument: 'name'
```

[68]:
```
record_user_information(occupation = 'Student', age = 22, name = 'Nimo', city =␣
↪'Delhi')
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-68-df854a2ec280> in <module>
----> 1 record_user_information(occupation = 'Student', age = 22, name =␣
↪'Nimo', city = 'Delhi')


TypeError: record_user_information() got an unexpected keyword argument␣
↪'city'
```

[ ]:
```

```

1.3.3 Default Arguments

- While defining a function you can specify a argument's value by using **name=value**, then **value** becomes a default value for that argument.
- Parameters defined this way are referred to as default or optional arguments.
- **All positional arguments must be defined before the default ones.**

[79]:
```
def record_user_information(name, occupation, age=25):

    print(f"Name = {name}")
    print(f"Occupation = {occupation}")
    print(f"Age = {age}")
```

[80]:
```
record_user_information(occupation = 'Student', age = 22, name = 'Nimo')
```

```
Name = Nimo
Occupation = Student
Age = 22
```

[82]:
```
record_user_information(occupation = 'Student', name = 'Nimo')
```

```
Name = Nimo
Occupation = Student
Age = 25
```

[83]:
```
record_user_information(occupation = 'Student', age = 24)
```

```
      ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-83-497cf995ce9d> in <module>
   ----> 1 record_user_information(occupation = 'Student', age = 24)


       TypeError: record_user_information() missing 1 required positional␣
↪argument: 'name'
```

[84]:
```
def record_user_information(name='Dummy', occupation='Student', age=25):

    print(f"Name = {name}")
    print(f"Occupation = {occupation}")
    print(f"Age = {age}")
```

[85]:
```
record_user_information()
```

```
Name = Dummy
Occupation = Student
Age = 25
```

[86]:
```
record_user_information(occupation = 'Software Engineer', age = 24)
```

```
Name = Dummy
Occupation = Software Engineer
Age = 24
```

[87]:
```
def record_user_information(name='Dummy', occupation='Student', age):

    print(f"Name = {name}")
    print(f"Occupation = {occupation}")
    print(f"Age = {age}")
```

```
  File "<ipython-input-87-52ab8dacffc7>", line 1
    def record_user_information(name='Dummy', occupation='Student', age):
                                ^
SyntaxError: non-default argument follows default argument
```

[88]:
```
def record_user_information(age, name='Dummy', occupation='Student'):

    print(f"Name = {name}")
    print(f"Occupation = {occupation}")
    print(f"Age = {age}")
```

Positional arguments must agree in order and number with the parameters declared in the function definition.

Keyword arguments must agree with declared parameters in number, but they may be specified in arbitrary order.

Default arguments allow some arguments to be omitted when the function is called.

1.3.4 Arbitrary Arguments

*args - contains all the extra positional arguments

**kwargs - contains all the extra keyword arguments

All the positional arguments must appear before keyword arguments.

[89]:
```
def record_user_information(name, occupation, *args, **kwargs):

    print(f"Name = {name}")
    print(f"Occupation = {occupation}")

    print("-"*20)

    for arg in args:
        print(arg)

    print("-"*20)

    for k in kwargs:
        print(k, kwargs[k])
```

[92]:
```
record_user_information('Nemo', 'ML engineer', 'Delhi', '12LPA', 'CS',␣
↪graduate_year=2014, years_of_experience=6, city_tier='tier-1')
```

```
Name = Nemo
Occupation = ML engineer
--------------------
Delhi
12LPA
CS
--------------------
graduate_year 2014
years_of_experience 6
city_tier tier-1
```

[95]:
```
record_user_information('Rohan', 'Data Scientist')
```

```
Name = Rohan
Occupation = Data Scientist
--------------------
--------------------
```

[72]:
```
record_user_information('Nemo', 'ML engineer', 'Delhi', graduate_year=2014,␣
↪years_of_experience=6, '12LPA')
```

```
          File "<ipython-input-72-8c7307ddfec0>", line 1
        record_user_information('Nemo', 'ML engineer', 'Delhi',␣
↪graduate_year=2014, years_of_experience=6, '12LPA')
                                                                                ␣
↪                           ^
    SyntaxError: positional argument follows keyword argument
```

[96]:
```
record_user_information('Nemo', 'ML engineer', 'Delhi',  '12LPA',␣
↪graduate_year=2014, years_of_experience=6)
```

```
Name = Nemo
Occupation = ML engineer
--------------------
Delhi
12LPA
--------------------
graduate_year 2014
years_of_experience 6
```

[ ]:
```

```

[ ]:

1.4 Python Local, Global variables

Local Variables

A variable declared inside the function's body or in the local scope is known as local variable.

[108]:
```
def scope_of_variables():
    z = "machine learning"
    print(z)

scope_of_variables()
print(z)
```

```
machine learning


␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-108-e92e3d506bb2> in <module>
  4
  5 scope_of_variables()
----> 6 print(z)


NameError: name 'z' is not defined
```

Global Variables

In Python, a variable declared outside of the function or in global scope is known as a global variable.

This variable can be accessed inside or outside of the function.

[109]:
```
y = 21

def scope_of_variables():
    print(y)

scope_of_variables()
```

```
21
```

[110]:
```
y = 21

def scope_of_variables():
    z = y*10
    print(z)

scope_of_variables()
print(y)
```

```
210
21
```

[112]:
```
y = 21

def scope_of_variables():
    y = y*10
    print(y)

scope_of_variables()
print(y)
```

```
␣
↪---------------------------------------------------------------------------

UnboundLocalError                         Traceback (most recent call␣
↪last)

<ipython-input-112-45013664a230> in <module>
      5     print(y)
      6
----> 7 scope_of_variables()
      8 print(y)


<ipython-input-112-45013664a230> in scope_of_variables()
      2
      3 def scope_of_variables():
----> 4     y = y*10
      5     print(y)
      6


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

1.4.1 Global Keyword

[113]:
```
help()
```

```
Welcome to Python 3.7's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not

help>

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
```

Global keyword allows you to modify the variable outside of the current scope.

It is used to create a global variable and make changes to the variable in a local context.

The basic rules for global keyword in Python are: - a variable created inside a function, it is local by default. - a variable created outside of a function, it is global by default. You don't have to use global keyword. - *Global* keyword is used to read and write a global variable inside a function. - *Global* keyword outside a function has no effect.

[103]:
```
y = 21

def scope_of_variables():
    global y
    y = y*10

print(y)
scope_of_variables()
print(y)
```

```
21
210
```

1.4.2 Global and local variables

[104]:
```
def scope_of_variables(y):
    y = 41

y = 31

print(y)
scope_of_variables(y)
print(y)
```

```
31
31
```

[105]:
```
def scope_of_variables():
    global y
    y = 41

y = 31

print(y)
scope_of_variables()
print(y)
```

```
31
41
```

Names listed in a global statement must not be defined as formal parameters Arguments of a function are considered to be local variables by default

[106]:
```
def scope_of_variables(x):
    global x
    x = x**2

x = 11
```

```
scope_of_variables(x)
```

```
  File "<ipython-input-106-291ff511f78e>", line 5
    x = 11

    ^
SyntaxError: name 'x' is parameter and global
```

[ ]:

[ ]:

1.5 Pass-By-Value vs Pass-By-Reference

Function : id(object)

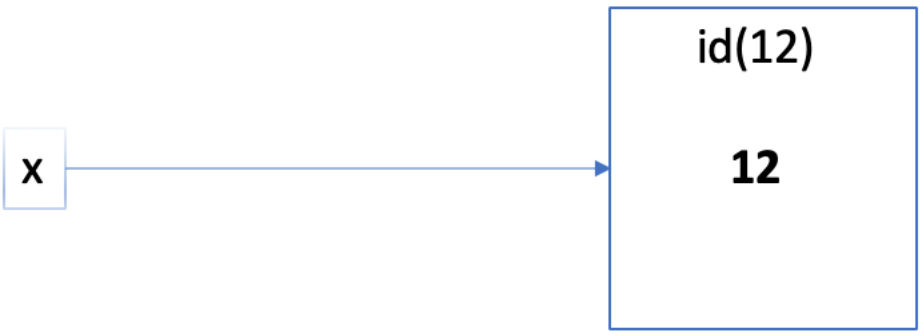
- This identity has to be unique and constant for this object during the lifetime.
- Similar to C/C++, it is actually the memory address, here in Python it is the unique id. This function is generally used internally in Python.

[119]:
```
x = 12
print(f"x = {x} and id({x}) = {id(x)}")
```

```
x = 12 and id(12) = 4340352480
```

[120]:
```
Image('images-functions/x12_23.png', width=500)
```

[120]:



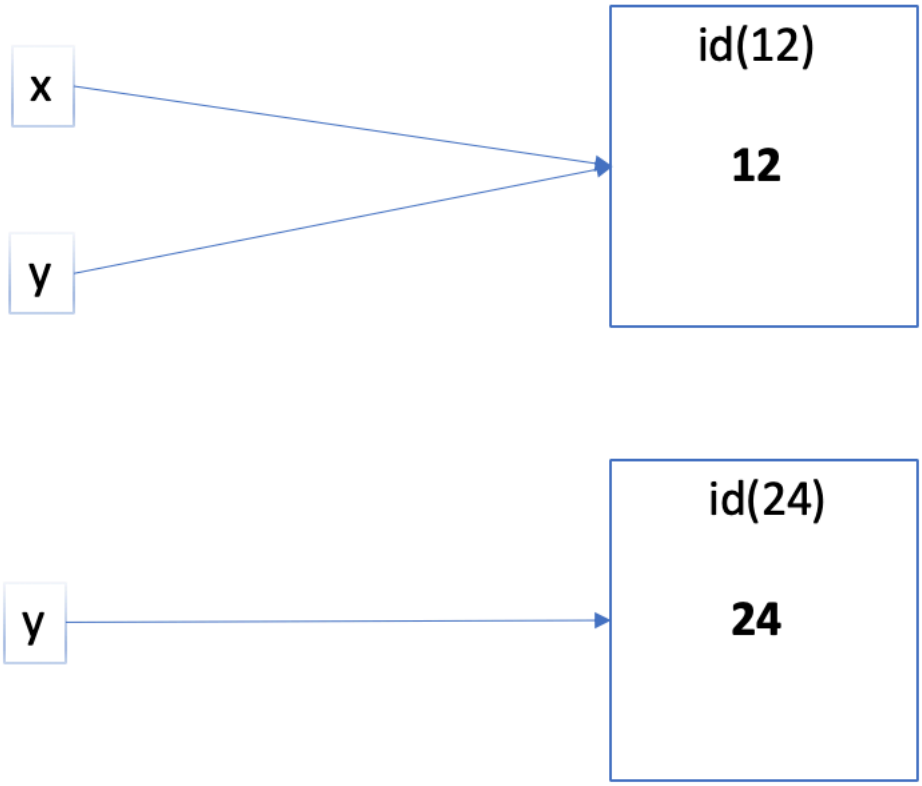
[121]:
```
y = 12
print(f"y = {y} and id({y}) = {id(y)}")
```

```
y = y+12
print(f"y = {y} and id({y}) = {id(y)}")
```

```
y = 12 and id(12) = 4340352480
y = 24 and id(24) = 4340352864
```

[122]:
```
Image('images-functions/x_y.png', width=500)
```

[122]:



[123]:
```
z = 24
print(f"z = {z} and id({z}) = {id(z)}")
```

```
z = 24 and id(24) = 4340352864
```

1.5.1 Call by value

[124]:
```
def argument_passing(x):
    print(f'Argument : {x} with id = {id(x)}')
    x = 45
    print(f'Argument updated : {x} with id = {id(x)}')
```

```
x = 4
argument_passing(x)

print(f"Original : {x} with id = {id(x)}")
```

```
Argument : 4 with id = 4340352224
Argument updated : 45 with id = 4340353536
Original : 4 with id = 4340352224
```

[17]:
```
def update_whole_list(l):
    l = 'foo'

name_list = ['ML', 'DS', 'CS', 'SQL', 'SPARK']

print(f"BEFORE calling the function : \t {name_list}")

update_whole_list(name_list)

print(f"AFTER calling the function : \t {name_list}")
```

```
BEFORE calling the function :    ['ML', 'DS', 'CS', 'SQL', 'SPARK']
AFTER calling the function :     ['ML', 'DS', 'CS', 'SQL', 'SPARK']
```

1.5.2 Call by reference

[43]:
```
def update_name(l):
    l[0] = 'Machine Learning'
    l[1] = 'Data Science'


name_list = ['ML', 'DS', 'CS', 'SQL', 'SPARK']

print(f"BEFORE calling the function : \t {name_list} and id = {id(name_list)}")

update_name(name_list)

print(f"AFTER calling the function : \t {name_list} and id = {id(name_list)}")
```

```
BEFORE calling the function :    ['ML', 'DS', 'CS', 'SQL', 'SPARK'] and id =
4431151008
AFTER calling the function :     ['Machine Learning', 'Data Science', 'CS',
'SQL', 'SPARK'] and id = 4431151008
```

1.5.3 Summary

- Passing an immutable object, like an int, float, str, tuple, or frozenset, to a Python function acts like pass-by-value.

- Passing a mutable object such as a **list, dict, or set** acts somewhat---but not exactly---like pass-by-reference. The function can't reassign the object completely, but it can change items in place within the object, and these changes will be reflected in the calling environment.

[ ]:

[ ]:

1.6 Void functions & Return statement

We can define two types of functions in Python : - functions which do not return anything to its caller. These functions are called void functions and default **None** values is returned by these functions. - functions which do return some values to its caller.

[55]:
```
def void_function(arguments):
    """
        Docstring :

        This is a sample function.
        Here you can write details about the function for future reference.
        More details coming up.

    """

    print(f'Welcome to functions. This is a {arguments}.')
    ...
    ...
    ...
```

[58]:
```
def non_void_function(arguments):
    """
        Docstring :

        This is a sample function.
        Here you can write details about the function for future reference.
        More details coming up.

    """
    ...
    ...
    ...
    return (f'Welcome to functions. This is a {arguments}.')
```

[59]:
```
type(void_function("ML"))
```

```
Welcome to functions. This is a ML.
```

[59]: NoneType

[60]:
```
type(non_void_function("ML"))
```

[60]: str

Return statement is used in following two ways in Python :

- It immediately terminates the function and passes execution control back to the caller.
- It provides a mechanism by which the function can pass data back to the caller.

1.6.1 Example

[125]:
```
def compute_area_of_rectangle(l, b):
    area = l*b

    return area
```

[126]:
```
return_value = compute_area_of_rectangle(2, 3)
```

[128]:
```
return_value
```

[128]: 6

[129]:
```
type(return_value)
```

[129]: int

[130]:
```
def compute_area_of_rectangle(l, b):
    area = l*b

    return

    print(f'Area is {area}.')
```

[131]:
```
return_value = compute_area_of_rectangle(2, 3)
```

[132]:
```
type(return_value)
```

[132]: NoneType

[133]:
```
## Compute sum of all multiples of k in [1, n]. Ask user for both n and k

n = int(input("Enter n. "))
k = int(input("Enter k. "))

def compute_sum(n, k):
```

```
    final_sum = 0

    for i in range(1, n+1):
        if i%k==0: final_sum += i

    return final_sum

final_sum = compute_sum(n, k)
final_prod = compute_prod(n, k)
print(f'Sum of multiples of {k} in 1 to {n} is {final_sum}.')
```

```
Enter n. 43
Enter k. 7
Sum of multiples of 7 in 1 to 43 is 147.
```

[ ]:

[ ]:

[ ]:

2 Function types

2.1 Built-in vs User-defined functions

The Python has a number of functions that are pre-defined and built in for specific tasks.

We have seen a few built-in functions so far **input**, **print** etc

You can find more details of existing built-in functions **here**.

[128]:
```
Image('images-functions/built-in-functions.png', width=1000)
```

[128]:

		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

[127]:
```
pow(2, 3)
```

[127]:
```
8
```

[42]:
```
max(12, 34, 34, 342), max(12, 234)
```

[42]:
```
(342, 234)
```

[44]:
```
?abs
```

[45]:
```
?max
```

[46]:
```
sorted([21, 32, 1, 2, 43])
```

[46]:
```
[1, 2, 21, 32, 43]
```

2.2 Recursive functions

A function can call other functions.

It is even possible for the function to call itself. These types of construct are termed as **recursive functions**.

2.2.1 Function calling another function

[85]:
```
import numpy as np

def mean(l):
    return np.mean(l)

def std_dev(l):
    return np.std(l)

def get_stats(l):
    return mean(l), std_dev(l)
```

[86]:
```
print(get_stats([1,2,3,4,5]))
```

```
(3.0, 1.4142135623730951)
```

2.2.2 Funtction calling itself

[87]:
```
def recursive_function():
    ...
    ...
    ...
    recursive_function()
    ...
    ...
```

[ ]:
```
n! = 1*2*3....(n-1)*n
```

[140]:
```
def compute_factorial(n):
    if n==1:
        return 1

    return n*compute_factorial(n-1)
```

[141]:
```
compute_factorial(6)
```

[141]: 720

2.2.3 Pros and cons of recursion

Pros

1. Compact code
2. Allows user to solve a bigger problem by reducing it to smaller sub problems.

Cons

1. Recruisve solution is sometimes complicated.
2. User need to carefully define base conditions, if these base conditions are not specified then you code can go in infinite loop.

[90]:
```
def compute_factorial(n):

    return n*compute_factorial(n-1)
```

[91]:
```
compute_factorial(4)
```

```
␣
↪---------------------------------------------------------------------------

       RecursionError                            Traceback (most recent call␣
↪last)

       <ipython-input-91-c5aebdefa2df> in <module>
   ----> 1 compute_factorial(4)


       <ipython-input-90-168e54eaafd3> in compute_factorial(n)
         1 def compute_factorial(n):
         2
   ----> 3     return n*compute_factorial(n-1)


       … last 1 frames repeated, from the frame below …


       <ipython-input-90-168e54eaafd3> in compute_factorial(n)
         1 def compute_factorial(n):
         2
   ----> 3     return n*compute_factorial(n-1)


       RecursionError: maximum recursion depth exceeded
```

2.3 Python Anonymous/Lambda Function

[93]:
```
help()
```

```
Welcome to Python 3.7's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.7/tutorial/.
```

```
Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not

help> quit

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
```

2.3.1 Syntax of Lambda function

- These functions are called anonymous because they are not declared in the standard manner by using the def keyword.
- You can use the lambda keyword to create small anonymous functions.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

```
func_name = lambda arg1, arg2, …, argn : … # expression
```

2.3.2 Example 1

[142]:
```
cube = lambda x : x*x*x
```

[144]:
```
cube(6)
```

[144]: 216

[146]:
```
add = lambda arg1, arg2 : arg1 + arg2
```

[147]:
```
add(12, 23)
```

[147]: 35

2.3.3 Example 2

Mean

Standard deviation

Skewness

Kurtosis

[120]:
```
from scipy.stats import kurtosis, skew

stats = lambda l : (np.mean(l), np.std(l), skew(l), kurtosis(l))
```

[121]:
```
stats([1, 2, 3, 4, 5, 7, 8, 9, 10, 13])
```

[121]: (6.2, 3.655133376499413, 0.2821067299980273, -0.9877191724335761)

2.3.4 Example 3

[122]:
```
standardization = lambda l : (np.array(l) - np.mean(l))/(1e-7 + np.std(l))
```

[123]:
```
standardization([1, 2, 3, 4, 5, 7, 8, 9, 10, 13])
```

[123]:
```
array([-1.42265666, -1.14906884, -0.87548102, -0.6018932 , -0.32830538,
        0.21887026,  0.49245808,  0.7660459 ,  1.03963372,  1.86039718])
```

[ ]:

[ ]:

2.4 Why we need to have good understanding of functions?

The advantages of using functions are:

- Reducing duplication of code

- Decomposing complex problems into simpler pieces
- Improving clarity of the code
- Reuse of code
- Information hiding
- Maintainability

[ ]: