Functions

July 7, 2022

Functions

0.1 In-Built Functions

[1]:
```
# Built-in Functions
# name_of_the_function(parameters)

print(bin(7))

print(chr(97))

print(complex(2,3))

print(help(isinstance))

print(print.__doc__)
print()
#(Shift+Tab)  - shortcut to view docstring
```

```
0b111
a
(2+3j)
Help on built-in function isinstance in module builtins:

isinstance(obj, class_or_tuple, /)
    Return whether an object is an instance of a class or of a subclass thereof.

    A tuple, as in ``isinstance(x, (A, B, …))``, may be given as the target to
    check against. This is equivalent to ``isinstance(x, A) or isinstance(x, B)
    or …`` etc.

None
print(value, …, sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
```

flush: whether to forcibly flush the stream.

0.2 User-Defined Functions

[2]:
```
#hello( ) #Throws name error - Function call should be after defining the␣
↪function

def hello(): #Defining Function
    print('Hello, World!')

hello()
hello()
hello()
hello()
hello()
hello()
hello()
hello()
hello()
#Calling Function
```

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

[2]:
```
#Identifiers are user-defined names to represent a variable, function, class,␣
↪module or any other object

# check if an identifier is valid or not is by calling the str.isidentifier()␣
↪function.

print('python'.isidentifier())

print('1python'.isidentifier())

print('python.com'.isidentifier())

print('python_com'.isidentifier())
```

```
True
False
False
True
```

0.3 Type of Arguments :

Default Arguments

Positional Arguments

Keyword Arguments

Variable Length Arguments - Aribitary Aruguments

[8]:
```
# Pass single argument to a function

def hello(name):
    print('Hello,', name)

hello('Bob') # Prints Hello, Bob

hello('Sam') # Prints Hello, Sam
```

```
Hello, Bob
Hello, Sam
```

[3]:
```
# Pass two arguments
#Positional Aruguments

def func(name, job):
    print(name, 'is a', job)

func('Bob', 'developer') # Prints Bob is a developer
```

```
Bob is a developer
```

[5]:
```
#positional Aruguments

def func(name, job):
    print(name, 'is a', job)

func( 'Bob' ,'developer') # Prints Bob  is a developer
func( 'developer','Bob' ) # Prints developer is a Bob
```

```
#func('nitheesh',234,'trainee') # throws error
```

```
Bob is a developer
developer is a Bob
```

[10]:
```
# Keyword arguments can be put in any order

def func(name, job):
    print(name, 'is a', job)

func(name='Bob', job='developer') # Prints Bob is a developer

func(job='developer', name='Bob') # Prints Bob is a developer
```

```
Bob is a developer
Bob is a developer
```

[13]:
```
# Set default value 'developer' to a 'job' parameter

def func(name,job='developer',):
    print(name, 'is a', job)

func('Bob', 'manager')# Prints Bob is a manager

func('Bob')# Prints Bob is a developer

#Positional - Keyword - Default - Order
```

```
Bob is a manager
Bob is a developer
```

[14]:
```
#Variable length Positional Aruguments - Aribitary

def print_arguments(*args):
    print(args)

print_arguments(1, 54, 60, 8, 98, 12)

print_arguments( 8, 98, 12)

print_arguments(8)
 # Prints (1, 54, 60, 8, 98, 12)
```

```
(1, 54, 60, 8, 98, 12)
(8, 98, 12)
(8,)
```

[7]:
```
#Variable Length Keyword Aruguments
```

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

print_arguments(name='Bob', age=25, job='dev',gender='male',)
```

```
{'name': 'Bob'}
```

To-Do : Args vs Kwargs

[8]:
```
import math

x = math.sin(360*2*math.pi) #mathematical expression as argument

print(x) # Prints -3.133115067780141e-14
```

```
-3.133115067780141e-14
```

To-Do : Parameters vs Aruguments

0.4 Aruguments Passing : Pass by Reference vs Value

Python's argument passing model is neither "Pass by Value" nor "Pass by Reference" but it is "Pass by Object Reference".

If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like Call-by-value. It's different, if we pass mutable arguments.

[16]:
```
# Passed by reference : It means if you change what a parameter refers to␣
 ↪within a function,
# the change also reflects back in the calling function.
def val(lst):
    lst.append(4)
    print(lst," - ",id(lst))

lst=[1,2,3]
print(lst," - ",id(lst))
val(lst)
print(lst," - ",id(lst))
```

```
[1, 2, 3]  -  2129788300160
[1, 2, 3, 4]  -  2129788300160
[1, 2, 3, 4]  -  2129788300160
```

[17]:
```
#Passed by value Any changes made in the function variable won't change the␣
 ↪nature of the caller variable
def val(x):
    x=15
    print(x," - ",id(x))
```

```
x=10
val(x)
print(x," - ",id(x))
```

```
15  -  140710238759136
10  -  140710238758976
```

0.5 Variable Scope - Local & Global

Scope : Scope refers to the region within the code where a particular variable is visible

[2]:
```
#local Variable

def myfunc():
    y = 42      # local scope y
    print("within the function : ",y)
myfunc()        # prints 42

print(y) # Triggers NameError: y does not exist
```

```
within the function :  42
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-d5652346c62a> in <module>
      6 myfunc()        # prints 42
      7
----> 8 print(y) # Triggers NameError: x does not exist

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

[8]:
```
#Global variable

x = 42          # global scope x

def myfunc():
    print(f'Inside the function {x}')     # x is 42 inside def
myfunc( )

print(f'Outside the function {x} ')        # x is 42 outside def
```

```
Inside the function 42
Outside the function 42
```

[3]:
```
#Reassiging Global Variable

x = 42          # global scope x
```

```
def myfunc():
    x = 0
    print("Local X:",x)    # local x is 0

myfunc( )

print("Global X :",x)        # global x is still 42
```

```
Local X: 0
Global X : 42
```

[9]:
```
#Global Keyword

x = 42          # global scope x

def myfunc():
    global x    # declare x global
    x = 0
    print(x)    # global x is now 0

myfunc( )
print(x)        # x is 0
```

```
0
0
```

[6]:
```
#global Keyword

x=42
print(x)
def myfunc():
    global x
    x = x + 1   # global x is now 43
    print(x)

myfunc()
print(x)        # x is 43
```

```
42
43
43
```

[5]:
```
#Globals() - returns the dictionary of current global symbol table.

a = 100
b = 4

def foo():
```

```
    x = 100 # x is a local variable
    print(x)

#print(globals())
```

[13]:
```
print( globals( )['a'])
```

```
100
```

[11]:
```
#Globals

a = 5

def  func():
    c = 10
    d = c + a
    globals( )['a'] = d # Calling globals()
    print(a)


func()
print(a)
```

```
15
15
```

[6]:
```
# Non-Local
x=50 # Global
def f1():
    #outer function
    x = 42  #Non-local variable - Enclosing Scope
    def f2():  # nested function
        #inner function
        x = 0
        print(f'Inner Function - Local {x}')    # x is 0
    f2()
    print((f'Outer Function - Non Local {x}') )
f1()
print((f'Outside of Functions - Global {x}') )
```

```
Inner Function - Local 0
Outer Function - Non Local 42
Outside of Functions - Global 50
```

[17]:
```
#Non-Local

def f1():
    x = 42  #Non-local variable
    def f2():  # nested function
```

```
        nonlocal x
        x = 0
        print(x)    # x is 0
    f2()
    print(x)
f1()
```

```
0
0
```

1 Return

[11]:
```
#Retruning Values

def sum(a, b):
    return a + b
x = sum(3, 4)
print(x) # Prints 7`
print(sum(4,5))
```

```
7
9
```

[19]:
```
#Retruning Multiple Values

def func(a, b):
    return a+b, a-b , a*b

result = func(3, 2)

print(result) # Prints (5, 1 , 6)
```

```
(5, 1, 6)
```

[8]:
```
#Retruning Multiple Values

def func(a, b):
    return a+b, a-b

result = func(3, 2)
print(result)

add, sub = func(3, 2)
print(add) # Prints 5
print(sub) # Prints 1
```

```
(5, 1)
5
1
```

[21]:
```
#Calling one function in other function
def function1():
    def function2(a,b):
        return a+b
    c=function2(3,5)
    print(c)
    return c+5

function1()
```

```
8
```

[21]: 13

[15]:
```
def multiply(a, b):
    c = a*b
    return c
    print(c) #-Return is end of function

multiply(10, 3)
```

[15]: 30

To-Do : Retrun vs Yeild

1.0.1 Docstring

[9]:
```
def hello():
    """ This function prints message on the screen
        param : No parameters Required
        return: No Value Returned
        """
    print('Hello, World!')
hello()
print(help(hello))

print(hello.__doc__)
```

```
Hello, World!
Help on function hello in module __main__:

hello()
    This function prints message on the screen
    param : No parameters Required
    return: No Value Returned

None
 This function prints message on the screen
```

```
    param : No parameters Required
    return: No Value Returned
```

1.1 Function is an First-Class Object

Functions Can Be Stored in Data structures

[27]:
```
from yell import yell #importing a function from python file - Realted concept
 ↪- modules

funcs = [yell, str.lower, str.capitalize]

print(funcs)
```

```
[<function yell at 0x0000022E6196EB80>, <method 'lower' of 'str' objects>,
<method 'capitalize' of 'str' objects>]
```

[28]:
```
#Accessing Functions From List

for f in funcs:
    print(f, f('hey there'))

#Accessing Single Function

print(funcs[0]('heyho'))
```

```
<function yell at 0x0000022E6196EB80> HEY THERE!
<method 'lower' of 'str' objects> hey there
<method 'capitalize' of 'str' objects> Hey there
HEYHO!
```

Functions Can Be Passed To Other Functions

[30]:
```
def greet(func):
    greeting = func('Hi, I am a Python program') #yell()
    print(greeting)

greet(yell)

def whisper(text):
    return text.lower() + '...'
greet(whisper)
```

```
HI, I AM A PYTHON PROGRAM!
hi, i am a python program...
```

Functions Can Be Nested

[34]:
```
def speak(text):
    def whispers(t):
        return t.lower() + '...'
```

```
    return whisper(text)

s = speak('Hello, World')

print(s)

#whispers('Yo') #Inner Function cannot Be Accessed outside outer function
```

```
hello, world...
```

[35]:
```
#Nested Functions

def get_speak_func(volume):
    def whisper(text):
        return text.lower() + '...'
    def yell(text):
        return text.upper() + '!'
    if volume > 0.5:
        return yell
    else:
        return whisper

s = get_speak_func(0.3)

print(s('HELLO'))

s = get_speak_func(0.7)

print(s('hello'))
```

```
hello...
HELLO!
```

[36]:
```
#Nested Functions

def get_speak_func(volume,text):
    def whisper():
        return text.lower() + '...'
    def yell():
        return text.upper() + '!'
    if volume > 0.5:
        return yell
    else:
        return whisper

s = get_speak_func(0.3,'HELLO')()

print(s)
```

hello…

Assigning and Deleting of Function

[17]:
```
from yell import yell

#Assiging To Variable

bark = yell

print(bark('i am from bark'))

#delting function

del yell

#yell('hello?') - Throws an error - NameError: name 'yell' is not defined

print(bark('function deleted'))
```

```
I AM FROM BARK!
FUNCTION DELETED!
```

1.2 Python Function Redefinition

[19]:
```
#function redefination - Realted concepts - Method Overloading , Method Over␣
↪Riding
from time import gmtime, strftime
def show_message(msg):
    print(msg)
show_message("Ready.")

def show_message(msg):
    print(strftime("%H:%M:%S", gmtime()))
    print(msg)
show_message("Processing")
```

```
Ready.
04:15:51
Processing
```

[20]:
```
#Similar to variables reassiging
x = 10
x = 15
print(x)
```

```
15
```

1.3 Code Challenges

Challenge 1 : We have been assigned to work on an existing program for an ATM system. The current program has bugs and needs to be fixed so we can provide the best experience for our users. Fix the scoping bugs to get the program to function properly. When fixed, the program should output :

Your balance is 1000

Your new balance is 500

You will gain interest on: 500

You will be taxed: 65.0

[ ]:
```
#Fix the scoping bugs
def print_balance():
    balance = 1000
    print("Your balance is " + str(balance))

def deduct(amount):
    print("Your new balance is " + str(balance - amount))

def calculate_interest_on_savings():
  print("You will gain interest on: " + str(savings))
  def calculate_taxes():
    savings = 500
    tax_amount = savings * 0.13
    print("You will be taxed: " + str(tax_amount))
  calculate_taxes()

print_balance()
deduct(500)
calculate_interest_on_savings()
```

Challenge 2 : Describe the scope of the variables a, b, c and d in this example

[3]:
```
#Describe the scope
def my_function(a):
    b = a - 2
    return b

c = 3

if c > 2:
    d = my_function(5)
    print(d)
```

```
3
```

What is the lifetime of these variables? When will they be created and destroyed ?