Functions

Prepared by Fahad Siddiqui on 26 Feb 2020

Outline

- Functions
- Parameterized function
- Functions with default parameters
- Functions with named parameters
- Dealing with an unknown number of arguments
- Functions as Variables
- Functions within functions
- Local vs. global variables

Simple functions

In [2]:

```
def add_numbers():
    first_number = 2
    second_number = 3
    total = first_number + second_number
    print(total)

add_numbers()
```

```
5
```

In [10]:

```
# Passing parameters

def add_numbers(first_number, second_number):
    total = first_number + second_number
    print(total)


print("Output add_numbers(2,3) : " , end = "")
add_numbers(2,3)


print("\nOutput add_numbers(2.5,3) : " , end = "")
add_numbers(2.5,3)


# What about this ?
# add_numbers('Hello','World')

print("\nOutput add_numbers('Hello','World') : " , end = "")
add_numbers('Hello','World')

# Because python is not explicitly typed
'''
Python is a dynamically-typed language. Java is a statically-typed language.
In a weakly typed language, variables can be implicitly coerced to unrelated types,
whereas in a strongly typed language they cannot, and an explicit conversion is required

'''
```

```
Output add_numbers(2,3) : 5

Output add_numbers(2.5,3) : 5.5

Output add_numbers('Hello','World') : HelloWorld
```

Functions with named parameters

In [31]:

```
# Tip

class color:
   PURPLE = '\033[95m'
   CYAN = '\033[96m'
   DARKCYAN = '\033[36m'
   BLUE = '\033[94m'
   GREEN = '\033[92m'
   YELLOW = '\033[93m'
   RED = '\033[91m'
   BOLD = '\033[1m'
   UNDERLINE = '\033[4m'
   END = '\033[0m'

print(color.PURPLE + 'Hello World !'  , end = "  ")

print(color.CYAN + 'Hello World !' , end = "  ")

print(color.DARKCYAN + 'Hello World !' , end = "  ")

print(color.BLUE + 'Hello World !' , end = "  ")

print(color.GREEN + 'Hello World !' , end = "\n\n")

print(color.YELLOW + 'Hello World !'  , end = "  ")

print(color.RED + 'Hello World !' , end = "  ")

print(color.BOLD + 'Hello World !' , end = "  ")

print(color.UNDERLINE + 'Hello World !' +color.END, end = "  ")

print(color.END + 'Hello World !' , end = "  \n\n")
```

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

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

In [33]:

```
def say_names_of_couple( husband_name = "Murat" , wife_name = "Hayat"):

    husband_name = color.BOLD + husband_name + color.END
    wife_name = color.BOLD + wife_name + color.END

    print("The names of the couple are " + husband_name + " and " + wife_name)



print("With default parameters \n")
#function call
say_names_of_couple()

print("\nWith explicit parameters \n")
#function call
say_names_of_couple("Asad","Nimra")
```

```
With default parameters

The names of the couple are Murat and Hayat

With explicit parameters

The names of the couple are Asad and Nimra
```

In [40]:

```
def calc_tax(sales_total=101.37, tax_rate=0.05):
    return (sales_total * tax_rate)

print("Default : " , calc_tax() )

print("Explicit : " , calc_tax(1200, 0.25) )
```

```
Default :  5.0685
Explicit :  300.0
```

In [42]:

```
# named parameters always come after positional parameters

def give_greeting(greeting, first_name, flattering_nickname=" the wonder boy"):
    print(greeting + ", " + first_name + flattering_nickname)

give_greeting("Hello", first_name="Ali")
```

```
Hello, Ali the wonder boy
```

In [50]:

```
# passing dictionaries, lists, tuples and sets

def accept_all( dict, list, tuple, set):
    print("dict : ", dict, end="\n\n")
    print("list : ", list, end="\n\n")
    print("tuple : ", tuple, end="\n\n")
    print("set : ", set, end="\n\n")


# Calling method

dict = {
    "first_name":"Fahad",
    "last_name":"Siddiqui"
}

list = [True,False]

tuple = ("pi", 3.14)

set = {0,1,2,3}


accept_all(dict,list,tuple,set)


print("\n Changing Order \n")
# This will print all but in wrong order
accept_all(list,set,dict,tuple)

print("\n Correct order \n")
accept_all(dict,set = set,tuple=tuple , list = list)
```

```
dict :  {'first_name': 'Fahad', 'last_name': 'Siddiqui'}

list :  [True, False]

tuple :  ('pi', 3.14)

set :  {0, 1, 2, 3}


 Changing Order

dict :  [True, False]

list :  {0, 1, 2, 3}

tuple :  {'first_name': 'Fahad', 'last_name': 'Siddiqui'}

set :  ('pi', 3.14)
```

Dealing with an unknown number of arguments

In [61]:

```
'''
def display_result(winner="Real Madrid", score="1-0", overtime ="yes", injuries="none"):
    # Do something here

'''

# *other_info as single tuple

def display_result( winner, score, *other_info):
    print( "winner: ", winner,end="\n\n")
    print( "score: ", winner,end="\n\n")
    print( "other_info: ", other_info,end="\n\n")


# display_result( "Real Madrid","1-0", ("yes","none") )
display_result("Real Madrid","1-0","yes","none")
```

```
winner:  Real Madrid

score:  Real Madrid

other_info:  ('yes', 'none')
```

In [63]:

```
'''
def display_result(winner="Real Madrid", score="1-0", overtime ="yes", injuries="none"):
    # Do something here

'''

# **other_info as a dictionary

def display_result( winner, score, **other_info):
    print( "winner: ", winner,end="\n\n")
    print( "score: ", winner,end="\n\n")

    for key, value in other_info.items():
        print(key + ": " + value ,end="\n\n")

#display_result("Real Madrid", "1-0", {overtime :"yes", injuries:"none"} )

display_result("Real Madrid", "1-0", overtime ="yes", injuries="none")
```

```
winner:  Real Madrid

score:  Real Madrid

overtime: yes

injuries: none
```

Functions as Variables

In [1]:

```
def add_numbers(first_number, second_number):
    return first_number + second_number

def subtract_numbers(first_number, second_number):
    return first_number - second_number


result_of_adding = add_numbers(1, 2)

result_of_subtracting = subtract_numbers(3, 2)

# sum_of_results = add_numbers(1, 2) + subtract_numbers(3, 2)
sum_of_results  = result_of_adding   + result_of_subtracting

print("Addition: ",result_of_adding , end = "\n\n")

print("Subtraction: ",result_of_subtracting , end = "\n\n")

print("Sum: ", sum_of_results , end="\n\n")
```

```
Addition:  3

Subtraction:  1

Sum:  4
```

Functions within functions

In [25]:

```
# Encapsulation

def outer(num1):
    def inner_increment(num1):  # Hidden from outer code
        return num1 + 1
    num2 = inner_increment(num1)
    print(num1, num2)

outer(10)
```

```
10 11
```

In [34]:

```
# Function to check Type of input
x = isinstance(5, int)
print(x , end="\n\n")

x = isinstance('5', int)
print(x , end="\n\n")

x = isinstance('5', str)
print(x , end="\n\n")


x = isinstance("Hello", (float, int, str, list, dict, tuple))
print(x , end="\n\n")
```

```
True

False

True

True
```

In [27]:

```
def factorial(number):

    # Error handling
    if not isinstance(number, int):
        raise TypeError("Sorry. 'number' must be an integer.")
    if not number >= 0:
        raise ValueError("Sorry. 'number' must be zero or positive.")

    def inner_factorial(number):
        if number <= 1:
            return 1
        return number*inner_factorial(number-1)
    return inner_factorial(number)

# Call the outer function.
print(factorial(4))
print(factorial('4'))
```

```
24

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-de9d494437d8> in <module>
     15 # Call the outer function.
     16 print(factorial(4))
---> 17 print(factorial('4'))

<ipython-input-27-de9d494437d8> in factorial(number)
      3     # Error handling
      4     if not isinstance(number, int):
----> 5         raise TypeError("Sorry. 'number' must be an integer.")
      6     if not number >= 0:
      7         raise ValueError("Sorry. 'number' must be zero or positive."
)

TypeError: Sorry. 'number' must be an integer.
```

In [41]:

```
# Generates a function with base n

def generate_power(number):

    # Define the inner function ...
    def nth_power(power):
        return number ** power
    # ... that is returned by the factory function.

    return nth_power

raise_two = generate_power(2)
raise_three = generate_power(3)

# Cube of 2
print("2^2 = " , raise_two(3))

# Cube of 3
print("3^2 = " ,raise_three(3))
```

```
2^2 =  8
3^2 =  27
```

Local vs. global variables

In [14]:

```
x = "global"

def foo():
    print("x inside :", x)

foo()
print("x outside:", x)
```

```
x inside : global
x outside: global
```

In [16]:

```
# This is an error --> UnboundLocalError: local variable 'x' referenced before assignment

x = "global"

def foo():
    x = x * 2
    print(x)

foo()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-16-184dd13c151a> in <module>
      7     print(x)
      8
----> 9 foo()

<ipython-input-16-184dd13c151a> in foo()
      4
      5 def foo():
----> 6     x = x * 2
      7     print(x)
      8

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

In [17]:

```
# Local Variable - > NameError: name 'y' is not defined


def foo():
    y = "local"

foo()
print(y)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-17-2cbf28ce88b8> in <module>
      5
      6 foo()
----> 7 print(y)

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

In [18]:

```
# Global Keyword

x = "global"

def foo():
    global x
    y = "local"
    x = x * 2
    print(x)
    print(y)

foo()
```

```
globalglobal
local
```

In [19]:

```
x = 5

def foo():
    x = 10
    print("local x:", x)

foo()
print("global x:", x)
```

```
local x: 10
global x: 5
```

In [20]:

```
# Nonlocal variable are used in nested function whose local scope is not defined.
# This means, the variable can be neither in the local nor the global scope.

def outer():
    x = "local"

    def inner():
        nonlocal x
        x = "nonlocal"
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
```

```
inner: nonlocal
outer: nonlocal
```

The End!