Functions

Functions

- Functions in programming is to bundle a set of instructions that you want to use
  - repeatedly or that, because of their complexity,
  - are better self-contained in a sub-program
  - and called when needed.

Types of Functions

- There are three types of functions in Python:
- Built-in functions, such as help() to ask for help, min() to get the minimum value, print()to print an object to the terminal,… You can find an overview with more of these functions
- User-Defined Functions (UDFs), which are functions that users create to help them out; And
- Anonymous functions, which are also called lambda functions because they are not declared with the standard def keyword.

How To Define A Function: User-Defined Functions (UDFs)

- The four steps to defining a function in Python are the following:

1. Use the keyword def to declare the function and follow this up with the function name.
2. Add parameters to the function: they should be within the parentheses of the function. End your line with a colon.
3. Add statements that the functions should execute.
4. End your function with a return statement if the function should output something. Without the return statement, your function will return an object None.

Example 1

```
def functionname( parameters ):
        "function_docstring"
        function_suite
        return [expression]

def hello():
        print("Hello World")
        return
```

Example 2

```
def hello():
  name = str(input("Enter your name: "))
  if name:
    print ("Hello " + str(name))
  else:
    print("Hello World")
  return

hello()
```

- No parameter no return value
- With Parameter no return value
- Without Parameter With Return value
- With Parameter and with return value

Parameters vs Arguments

- Parameters are the names used when defining a function , and into which arguments will be mapped.
- In other words, arguments are the things which are supplied to any function call, while the function or method code refers to the arguments by their parameter names.

Function Arguments in Python

- Default arguments
- Required arguments
- Keyword arguments
- Variable number of arguments

Required Arguments

- As the name kind of gives away, the required arguments of a UDF are those that have to be in there. These arguments need to be passed during the function call and in exactly the right order, just like in the following example:

```
def plus(a,b):
  return a + b
```

How to call a function

```
plus(2,3)
```

Default Arguments

- Default arguments are those that take a default value if no argument value is passed during the function call. You can assign this default value by with the assignment operator =, just like in the following example:

```
# Define `plus()` function
def plus(a,b = 2):
  return a + b

# Call `plus()` with only `a` parameter
plus(a=1)

# Call `plus()` with `a` and `b` parameters
plus(a=1, b=3)
```

Keyword Arguments

- If you want to make sure that you call all the parameters in the right order, you can use the keyword arguments in your function call. You use these to identify the arguments by their parameter name. Let's take the example from above to make this a bit more clear:

```
# Define `plus()` function
def plus(a,b):
  return a + b
  # Call `plus()` function with parameters
plus(2,3)
# Call `plus()` function with keyword arguments
plus(a=1, b=2)
# Define `plus()` function
def plus(a,b):
  return a + b
# Call `plus()` function with keyword arguments
plus(b=2, a=1)
```

Variable Number of Arguments

- In cases where you don't know the exact number of arguments that you want to pass to a function, you can use the following syntax with *args:

```
def plus(*args): # arguments
  return sum(args)

# Calculate the sum
plus(1,4,5)
```

Example (variable argument)

```
# Define `plus()` function to accept a variable number of arguments
def plus(*args):
  sum = 0
  for i in args:
    sum += i
  return sum

# Calculate the sum
plus(20,30,40,50)
plus(210,130,401)
```

Global vs Local Variables

- In general, variables that are defined inside a function body have a local scope, and those defined outside have a global scope. That means that local variables are defined within a function block and can only be accessed inside that function, while global variables can be accessed by all functions that might be in your script:

```
# Global variable `init`
init = 1

# Define `plus()` function to accept a variable number of arguments
def plus(*args):
  # Local variable `sum`
  sum = 0
  for i in args:
    sum += i
  return sum

# Access the global variable
print("this is the initialized value " + str(init))

# (Try to) access the local variable
print("this is the sum " + str(sum))
```

Global and local variables

```
def f():
    s = 2
    print s  #2

# Global scope
s = 5
f()
print s #5
```

Local and Global Variables

```
# Uses global keyword to modify global 'a'
a=5
def h():
    global a
    a = 3
    print('Inside h() : ',a)

# Global scope
h()
print('global : ',a)
```

Anonymous Functions in Python

- Anonymous functions are also called lambda functions in Python because instead of declaring them with the standard def keyword, you use the lambda keyword.

```
double = lambda x: x*2
double(5)
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

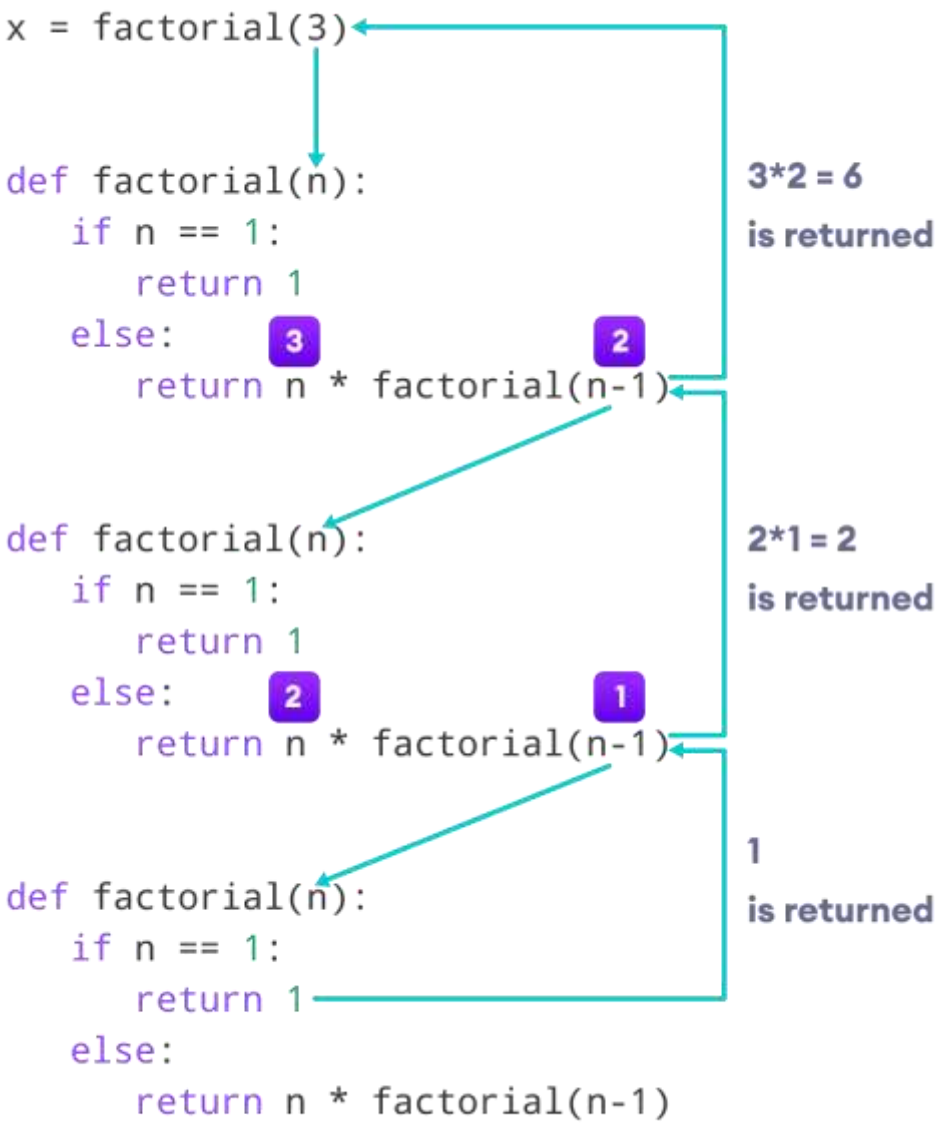
Recursive Function

```
def calc_factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * calc_factorial(x-1))

num = 4
print("The factorial of", num, "is", calc_factorial(num))
```

```
factorial(3)          # 1st call with 3
3 * factorial(2)  # 2nd call with 2
3 * 2 * factorial(1) # 3rd call with 1
3 * 2 * 1          # return from 3rd call as number=1
3 * 2              # return from 2nd call
 6                 # return from 1st call
```


x = factorial(3)
def factorial(n):
if n == 1:
return 1
else:
return n * factorial(n-1)
3
2
def factorial(n):
if n == 1:
return 1
else:
return n * factorial(n-1)
2
1
def factorial(n):
if n == 1:
return 1
else:
return n * factorial(n-1)
3*2 = 6
is returned
2*1 = 2
is returned
1
is returned

Example 1

```
# Simple function call and function  definition

def add(a,b):
    print(a+b)
add(3,4)
p=10
q=32
add(p,q)
```

Example 2

```
# Function returns value


def add(a,b):
    return a+b
x=add(3,4)
print(x)
print(add(10,23))
```

Example 3

```
# Function return more than one value
def swap(a,b):
    return b,a
a,b=swap(5,10)
print(a,b)
```

Example 4

```
# function with default arguments
def add(a,b=10):
    return a+b
x=add(5,25)
print(x)
y=add(10)
print(y)
```

Example 5

```
#function with Default arguments and keyword arguments

def SI(p,n,r=9):
    return p*n*r/100
a=SI(10000,4,15)
print(a)
b=SI(10000,4);
print(b)
c=SI(r=15,p=10000,n=4)
print(c)
```

Example 6

```
#function with list arguments


def name(a,b,c):
    print(a,b,c)
c=[1,2,3,4,5,6,7,8,9]
name("hello",2015,c)
```

Example 7

```
# Function with more than one Definition

a = 10
if a%2==0:
        def func():
                print('even')
else:
        def func():
                print('odd')
func()
```

Example 8

```
# Function with Default and keyword arguments

def func(spam, eggs, toast=0, ham=0):
    print(spam, eggs, toast, ham)
func(1, 2)
func(1, ham=1, eggs=0)
func(spam=1, eggs=0)
func(toast=1, eggs=2, spam=3)
func(1, 2, 3, 4)
```

Example 9

```
# check whether palindrome or not using function
def palindrome(x):
        if x[::-1]==x:
                return 'True'
        else:
                return'False'
s=palindrome(input('Enter the string: '))
print(s)
```