In [1]:
```
# Python has first-class functions, which means:

# functions can be passed as arguments to other functions
# functions can be returned as values from other functions
# functions can be assigned to variables
# functions can be stored in data structures
```

In [2]:
```
# Function definition
# keyword 'def' introduces a function definition, it must be
# followed by the function name and the parenthesized list
# of formal parameters.

def function_name( parameters ):
    statements
```

In [ ]:
```
'''The parameter list may be empty,
or it may contain any number of parameters
separated from one another by commas.

first statement of the function body can optionally
be a string literal - the function's documentation
string, or docstring.

docstrings are used by some tools to automatically
produce online or printed documentation, or to let
the user interactively browse through code.
'''
```

In [3]:
```
def hi():
    """Print two strings, Hello and Let's function."""
    print("Hello")
    print("Let's function")

print(type(hi))

hi()   # function call or function invocation
print('done')
```

```
<class 'function'>
Hello
Let's function
done
```

In [4]:
```
# docstring is available via the __doc__ attribute
hi.__doc__
```

Out[4]:
```
"Print two strings, Hello and Let's function."
```

In [ ]:
```
'''A function definition associates the function name
with the function object in the current symbol table.
The interpreter recognizes the object pointed to
by that name as a user-defined function.
Other names can also point to that same function object
and can also be used to access the function.

https://docs.python.org/3/tutorial/controlflow.html#defining-functions
'''
```

In [5]:
```
# Since we can refer to functions like any other object,
# we can point a variable to a function
```

```
def hi():
    """Print two strings, Hello and Let's function."""
    print("Hello")
    print("Let's function")

welcome = hi
welcome()
```

```
Hello
Let's function
```

In [6]:
```
# Functions, like any other object, can be passed
# as an argument to another function

def hi():
    """Print two strings, Hello and Let's function."""
    print("Hello")
    print("Let's function")

help(hi) # hi is passed as an argument to help()
```

```
Help on function hi in module __main__:

hi()
    Print two strings, Hello and Let's function.
```

In [7]:
```
# function with one parameter

def hi(name):
    """Print Hello name, and Let's function."""
    print("Hello " + name)
    print("Let's function")

# arguments are passed using 'call by value',
# where the value is always an object reference,
# not the value of the object
# so 'call by object reference' is a more appropriate term

hi('Aditi')
hi('Zahir')
```

```
Hello Aditi
Let's function
Hello Zahir
Let's function
```

In [8]:
```
#function with two parameters

def see_you(s1, s2):
    if len(s1) > len(s2):
        print(s1)
    else:
        print(s2)

see_you("Hi", "Bye")
see_you("Good evening", "Good night")
see_you("Aye", "Bye")
```

```
Bye
Good evening
Bye
```

In [9]:
```
def print_again(s, n):
    """n times printing string s"""
```

```
    for i in range(n):
        print(s)

print_again("done", 3)
```

```
done
done
done
```

In [10]:
```
# returning value(s) from function
# fruitful functions

def squared(x):
    y = x * x
    return y

num = -6
num_square = squared(num)
print(f'The result of {num} squared is {num_square}.')
```

```
The result of -6 squared is 36.
```

In [11]:
```
# all Python functions return the value None
# unless there is a return statement with a value other than None

def cubed(x):
    y = x ** 3

num = 5
num_cube = cubed(num)
print(f'The result of {num} cubed is {num_cube}.')
```

```
The result of 5 cubed is None.
```

In [12]:
```
# all Python functions return the special value None
# unless there is a return statement with a value other than None

def cubed(x):
    y = x ** 3
    return

num = 4
num_cube = cubed(num)
print(f'The result of {num} cubed is {num_cube}.')
```

```
The result of 4 cubed is None.
```

In [13]:
```
# To check if any marks in a list is greater than 90

def greater_than_ninety(list_marks):
    for marks in list_marks:
        if marks > 90:
            return True
    return False # this executes only if no marks is greater than 90


list1 = [56,77,91,88]
list2 = [56,77,78,88]

print(greater_than_ninety(list1))
print(greater_than_ninety(list2))
```

```
True
False
```

In [14]:
```
def fib(n):  # return Fibonacci series up to n
    """Return a list containing the Fibonacci series up to n."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result

print(fib(60))
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

In [15]:
```
# Functions Annotations
# indicate the intended type of the function parameters
# and the type of the function return value
# completely optional metadata information.
# they can be used by third party tools such as type checkers, IDEs, linters, etc.

def add(x: int, y: int) -> int:
    return x + y

new_val: int = add(5,7)
print(new_val)
```

```
12
```

In [16]:
```
# Annotations are stored in the __annotations__ attribute
# of the function as a dictionary
# and have no effect on any other part of the function

add.__annotations__
```

Out[16]:
```
{'x': int, 'y': int, 'return': int}
```

In [17]:
```
def add(x: int, y: int) -> int:
    return x + y

new_val: int = add('Hi', 'MCAs')
print(new_val)
```

```
HiMCAs
```

In [18]:
```
# optional parameters - that can be specified or omitted

print(int("101"))
print(int("23", 8))
print(int("101", 5))
```

```
101
19
26
```

In [19]:
```
# optional parameters - that can be specified or omitted

marks = [55, 66, 99, 21, 32, 47]
marksAscending = sorted(marks)
print(f"Sorted in ascending order: {marksAscending}")

marksDescending = sorted(marks, reverse = True)
print(f"Sorted in descending order: {marksDescending}")
```

```
Sorted in ascending order: [21, 32, 47, 55, 66, 99]
Sorted in descending order: [99, 66, 55, 47, 32, 21]
```

In [20]:
```
# when defining a function, specify a default value for a parameter
# that parameter becomes an optional parameter when the function is called
# so, the function can be called with fewer arguments than it is defined

def f(x, y = 3):
    print(x**y)

f(2, 10)
f(2)
f(2, 5)
```

```
1024
8
32
```

In [21]:
```
# Note 1: the default value is evaluated at the time
# the function is defined, not at the time that it is invoked

# Note 2: THE DEFAULT VALUE IS EVALUATED ONLY ONCE
# if the default value is set to a mutable object,
# that object will be shared in all invocations of the function.

def f(x, y = []):
    y.append(x)
    return y

print(f(2))
print(f(3))
print(f(4,[25, 35, 'Hi']))
print(f(5))
```

```
[2]
[2, 3]
[25, 35, 'Hi', 4]
[2, 3, 5]
```

In [22]:
```
# Workaround: use None
# if don't want the default to be shared between subsequent calls

def f(x, y = None):
    if y is None:
        y = []
    y.append(x)
    return y

print(f(2))
print(f(3))
print(f(4,[25, 35, 'Hi']))
print(f(5))
```

```
[2]
[3]
[25, 35, 'Hi', 4]
[5]
```