Functions in Python

What is a function in Python?

- In Python, a function is a group of related statements that performs a specific task.
- Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.
- Furthermore, it avoids repetition and makes the code reusable.

```
def function_name(parameters):
        """docstring"""
        statement(s)
```

```
def greet(name):
    """
    This function greets to
    the person passed in as
    a parameter
    """
    print("Hello, " + name + ". Good morning!")

greet('Paul')
```

```
# function call
greet('Paul')

# function definition
def greet(name):
    """
    This function greets to
    the person passed in as
    a parameter
    """
    print("Hello, " + name + ". Good morning!")

# Erro: name 'greet' is not defined
```

Docstrings

- The first string after the function header is called the docstring and is short for documentation string. It is briefly used to explain what a function does.
- Although optional, documentation is a good programming practice.

```
>>> print(greet.__doc__)

    This function greets to
    the person passed in as
    a parameter
```

return statement

- The return statement is used to exit a function and go back to the place from where it was called.
- If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

Syntax of return

```
return [expression_list]
```

For example:

```
>>> print(greet("May"))
Hello, May. Good morning!
None
```

Example of return

```
def absolute_value(num):
    """This function returns the absolute
    value of the entered number"""

    if num >= 0:
        return num
    else:
        return -num


print(absolute_value(2))

print(absolute_value(-4))
```

Output

```
2
4
```

Scope and Lifetime of variables

```
def my_func():
        x = 10
        print("Value inside function:",x)

x = 20
my_func()
print("Value outside function:",x)
```

local scope

global scope

Output

```
Value inside function: 10
Value outside function: 20
```

Using Global and Local variables in the same code

```
x = "global "

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

foo()
```

Output

```
global global
local
```

Global variable and Local variable with same name

```
x = 5

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


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

Output

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

Nonlocal variables are used in nested functions whose local scope is not defined. This means that 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()
```

```
>>> def outer():
...     x="local"
...     def inner():
...         '''nonlocal x'''
...         x="nonlocal"
...         print("inner",x)
...     inner()
...     print("outer",x)
...
...
>>> outer()
inner nonlocal
outer local
```

Output

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

Global variable

- In Python, 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:

- When we create a variable inside a function, it is local by default.
- When we define a variable outside of a function, it is global by default. You don't have to use global keyword.
- We use global keyword to read and write a global variable inside a function.
- Use of global keyword outside a function has no effect.

```
c = 1 # global variable

def add():
    c = c + 2 # increment c by 2
    print(c)

add()
```

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

```
c = 0 # global variable

def add():
    global c
    c = c + 2 # increment by 2
    print("Inside add():", c)

add()
print("In main:", c)
```

```
Inside add(): 2
In main: 2
```

Create a `config.py` file, to store global variables

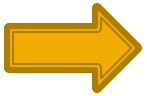
```
a = 0
b = "empty"
```

Create a `update.py` file, to change global variables

```
import config

config.a = 10
config.b = "alphabet"
```

Create a `main.py` file, to test changes in value

```
import config
import update

print(config.a)
print(config.b)
```



```
10
alphabet
```

Types of Functions

Basically, we can divide functions into the following two types:

- Built-in functions - Functions that are built into Python.
- User-defined functions - Functions defined by the users themselves.

Arguments

- non-default arguments cannot follow default arguments.

```
def greet(name, msg="Good morning!"):
    """
    This function greets to
    the person with the
    provided message.

    If the message is not provided,
    it defaults to "Good
    morning!"
    """

    print("Hello", name + ', ' + msg)


greet("Kate")
greet("Bruce", "How do you do?")
```

```
Hello Kate, Good morning!
Hello Bruce, How do you do?
```

```
def greet(msg = "Good morning!", name):
```

```
SyntaxError: non-default argument follows default argument
```

Python Keyword Arguments

```
# 2 keyword arguments
greet(name = "Bruce",msg = "How do you do?")

# 2 keyword arguments (out of order)
greet(msg = "How do you do?",name = "Bruce")

1 positional, 1 keyword argument
greet("Bruce", msg = "How do you do?")
```

```
greet(name="Bruce","How do you do?")
```

```
SyntaxError: non-keyword arg after keyword arg
```

Python Arbitrary Arguments

- we do not know in advance the number of arguments that will be passed into a function.

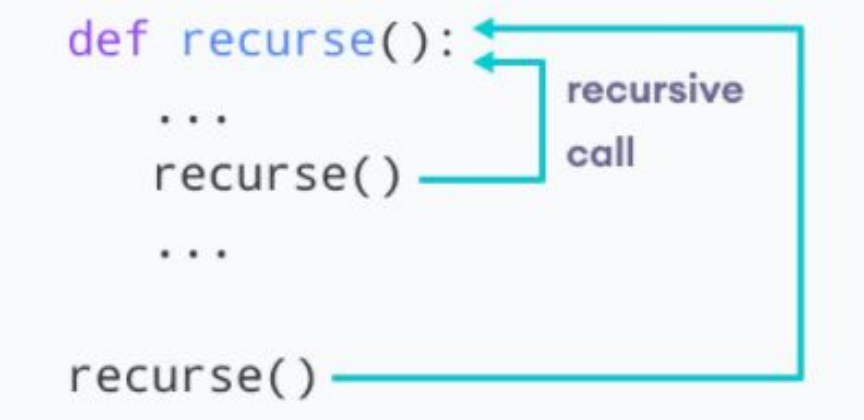
```
def greet(*names):
    """This function greets all
    the person in the names tuple."""

    # names is a tuple with arguments
    for name in names:
        print("Hello", name)


greet("Monica", "Luke", "Steve", "John")
```

Output

```
Hello Monica
Hello Luke
Hello Steve
Hello John
```

Recursion

- Recursion is the process of defining something in terms of itself.



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

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


num = 3
print("The factorial of", num, "is", factorial(num))
```

Output

```
The factorial of 3 is 6
```

```
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
```

Pros and Cons

Advantages of Recursion

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

lambda functions in Python

- In Python, an anonymous function is a function that is defined without a name.
- While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword.
- Hence, anonymous functions are also called **lambda functions.**

```
lambda arguments: expression
```

```
# Program to show the use of lambda functions
double = lambda x: x * 2

print(double(5))
```

Output

```
10
```

function

```
def double(x):
   return x * 2
```

Lambda function

- We use lambda functions when we require a nameless function for a short period of time.
- Lambda functions are used along with built-in functions like filter(), map() etc.
- The **map() & filter() function** in Python takes in a function and a list as **arguments**.

map()

```
# Program to double each item in a list using map()

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

print(new_list)
```

Output

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

filter()

```
# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

print(new_list)
```

Output

```
[4, 6, 8, 12]
```

Modules

- >>> import example
  >>> example.add(4,5)

example.py
def add(a, b):
.....

- import math math.pi
- import math as m m.pi
- from math import pi , e pi or e can be used
- from math import *
- import sys sys.path used to list directories
- dir() function to find out names that are defined inside a module