Functions and Modularity

Computer Science definition

- A function is a block of `organized`, `reusable` code that is used to perform a `single` related action.
- Functions provide better `modularity` for your application at a high degree of code reusing.

Mathematical background

The binary relation between of two sets say x and y that associates every element of first set (x) to exactly one element of the second set (y).

$$y = f(x)$$

or

$$y = f(x_1, x_2, x_3, \dots, x_n)$$

In computer science, functions do the same thing but in a different way.

Here,

- y is an outcome
- $f()$ is a function
- x is an input

Simply it is said as - acting upon the parameter x with a certain functionality f and storing the result in y.

Image by author

Function structure

```python
def some_action(*args, **kwargs):
    """
    docstring of a function
    """

    # write the function flow

    return None
```

- args $\rightarrow$ Arguments (basically input parameters)
- kwargs $\rightarrow$ Keyword arguments
- `outcome = some_action(*args, **kwargs)`

Types of functions

- Parameterized function structure

```python
def parameter_func(param1, param2, param3):
    # do something
    return None
```

- Non-parameterized function structure

```python
def non_parameter_func():
    # do something
    return None
```

Note: It is always good to have params' in function that signifies input receival and output returning.

In [ ]:

Different practises and uses of Functions

Suppose you are given a set of numbers and your task is to identify which number is odd and which is not (even).

The numbers are from 1 to 100.

3 things to remember

- start value
- end value
- even and odd logic

A newbie programmer

He / She will check all the 100 numbers individually with 100 if conditions and make the code messy.

```python
num = 1
if num % 2 == 0:
    print("even")
else:
    print("odd")
################
num = 2
if num % 2 == 0:
    print("even")
else:
    print("odd")
#################
num = 3
if num % 2 == 0:
    print("even")
else:
    print("odd")
#################
...
#################
num = 100
if num % 2 == 0:
    print("even")
else:
    print("odd")
```

An intermediate programmer

He / She will define a function to implement the task and repeat the function by `calling` it 100 times.

```python
# function definition
def check_odd_even(num):
    if num % 2 == 0:
        return True
    else:
        return False
#########################

num = 1
num_type = check_odd_even(num)
print(num_type)  # False
##############
num = 2
num_type = check_odd_even(num)
print(num_type)  # True
##############
num = 3
num_type = check_odd_even(num)
print(num_type)  # False
##############
...
##############
num = 100
num_type = check_odd_even(num)
print(num_type)  # True
```

A pro programmer (like you)

He / She will make a dynamic function. Even when task increases to check from 1 to 1000, he / she will be able to do it easily.

```python
def check_odd_even(start, end):
    """
    Checks if a number is even or odd within the given range
    :param int start: Integer number
    :param int end: Integer number
    :return dict categorised_numbers: Dictionary of odd_numbers and even_numbers
    """
    even_numbers = []
    odd_numbers = []

    for num in range(start, end + 1):
        if num % 2 == 0:
            even_numbers.append(num)
        else:
            odd_numbers.append(num)

    categorised_numbers = {
        'odd_numbers' : odd_numbers,
        'even_numbers' : even_numbers
    }
    return categorised_numbers
```

Task accomplished

In [1]:

```python
def check_odd_even(start, end):
    """
    Function docstring
    ------------------
    Checks if a number is even or odd within the given range
    :param int start: Integer number
    :param int end: Integer number
    :return dict categorised_numbers: Dictionary of odd_numbers and even_numbers
    """
    even_numbers = []
    odd_numbers = []
    provided_numbers = list(range(start, end + 1))

    for num in provided_numbers:
        if num % 2 == 0:
            even_numbers.append(num)
        else:
            odd_numbers.append(num)

    categorised_numbers = {
        'odd_numbers' : odd_numbers,
        'even_numbers' : even_numbers
    }
    return categorised_numbers
```

Function calling

In [2]:

```python
output = check_odd_even(start=1, end=10)
print(output)
```

```
{'odd_numbers': [1, 3, 5, 7, 9], 'even_numbers': [2, 4, 6, 8, 10]}
```

`help(<any_function>)`

In [3]:

```python
help(check_odd_even)
```

```
Help on function check_odd_even in module __main__:

check_odd_even(start, end)
    Function docstring
    ------------------
    Checks if a number is even or odd within the given range
    :param int start: Integer number
    :param int end: Integer number
    :return dict categorised_numbers: Dictionary of odd_numbers and even_numbers
```

`type(<any_function>)`

In [4]:

```python
type(check_odd_even)
```

Out[4]:

```
function
```

In [5]:

```python
re = check_odd_even(1, 10)
print(type(re))
```

```
<class 'dict'>
```

In [ ]:

Pros of using functions

- Increases readability of a program
- Organized code is always better than messy code
- Easy to understand and makes it reusable

In [ ]:

One line `if` - `else` statement

Note - This can be only used with `if` and `else`. We cannot use this along with `elif`.

In [6]:

```python
## one line if-else statement
## ternary operator

s = 0
if s == 0:
    r = True
else:
    r = False
# print(r)

## show example

r = True if (s == 0) else False
print(r)
```

```
True
```

Function chaining

In [7]:

```python
# show example

def which_greater(num1, num2):
    greater_num = num1 if (num1 > num2) else num2
    return greater_num

def which_greatest(num1, num2, num3):
    gr = which_greater(num1, num2)
    greatest_num = which_greater(gr, num3)
    # foo = which_greater(which_greater(num1, num2), num3)
    return greatest_num

# print(which_greatest(1, 2, 3))

def calculate_greatest():
    n_list = []
    for i in range(3):
        n = int(input("Enter num - {} : ".format(i + 1)))
        n_list.append(n)

    print("Given numbers : ", n_list)
    greatest = which_greatest(num1=n_list[0], num2=n_list[1], num3=n_list[2])

    return "The greatest number is " + str(greatest)
```

In [8]:

```python
calculate_greatest()
```

```
Enter num - 1 : 12
Enter num - 2 : 32
Enter num - 3 : 43
Given numbers :  [12, 32, 43]
```

Out[8]:

```
'The greatest number is 43'
```

Note: A function can use `n` number of other functions within. It can also be imported from different files to obtain modularity.

In [ ]:

Modularity in Python

Modularity simply encourages the separation of the `functionality` in a program into `distinct` and `independent units` such that `every unit` has everything required to execute.

- Code reusability
- Simplicity
- Organized code structure
- Easy to debug errors

Modularity flow

Image by author

Steps to implement modularity

- Create two files
  - basic_math.py
  - operate.py
- Write functions namely `add()`, `subtract()`, `product()`, `divide()` in `basic_math.py`.
- Import the functions of `basic_math.py` module in `operate.py` and reuse it.
- `from basic_math import <function_name>`

In [ ]:

List of built-in modules

- os
- sys
- math
- random
- time
- collections
- ...

syntax -

```python
import <package_or_module_name>
```

`time` example

In [9]:

```python
import time

for i in range(1, 10):
    print("Hi" * i)
    print("*" * 2*i)
    time.sleep(3)
```

```
Hi
**
HiHi
****
HiHiHi
******
HiHiHiHi
********
HiHiHiHiHi
**********
HiHiHiHiHiHi
************
HiHiHiHiHiHiHi
**************
HiHiHiHiHiHiHiHi
****************
HiHiHiHiHiHiHiHiHi
******************
```

In [10]:

```python
import time

s = []
for i in range(10):
    print(i+1 , " --> iteration")
    s.append(i)
    print("\t", s)
    time.sleep(2)
    print("######")
```

```
1  --> iteration
         [0]
######
2  --> iteration
         [0, 1]
######
3  --> iteration
         [0, 1, 2]
######
4  --> iteration
         [0, 1, 2, 3]
######
5  --> iteration
         [0, 1, 2, 3, 4]
######
6  --> iteration
         [0, 1, 2, 3, 4, 5]
######
7  --> iteration
         [0, 1, 2, 3, 4, 5, 6]
######
8  --> iteration
         [0, 1, 2, 3, 4, 5, 6, 7]
######
9  --> iteration
         [0, 1, 2, 3, 4, 5, 6, 7, 8]
######
10  --> iteration
         [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
######
```

`random` example

In [11]:

```python
import random

gaming_input = ['r', 'p', 's']
for i in range(10):
    computer_input = random.choice(gaming_input)
    print("computer selected - ", computer_input)
    time.sleep(2)
```

```
computer selected -  r
computer selected -  p
computer selected -  s
computer selected -  r
computer selected -  r
computer selected -  r
computer selected -  r
computer selected -  r
computer selected -  r
computer selected -  p
```

`collections` example

In [12]:

```python
import collections

# print(dir(collections))

list_value = ["hello", "hi", "greetings", "India", "hello", "hello", "hi"]

# use Counter() function/method from collections
c = collections.Counter(list_value)
print(c)
```

```
Counter({'hello': 3, 'hi': 2, 'greetings': 1, 'India': 1})
```

In [ ]:

What did we learn?

- Function definition
- Function types
- Use of doc string
- One liner if else statement
- Complex functionality of a function calling another function and so on
- Modularity in python (creating modules)
- List of built-in modules