Functions :

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.

Decomposition :

- Decomposition is a process of breaking down.
- It will be breaking down functions into smaller parts.
- It is another important principle of software engineering to handle problem complexity.

Abstraction :

- It refers to the construction of a simpler version of a problem by ignoring the details.
- The principle of constructing an abstraction is popularly known as modelling.
- It is the simplification of a problem by focusing on only one aspect of the problem while omitting all other aspects.

```
def my_function():
    '''
    Dostring: Reading manual
    '''
    print("Hello from a function")
    or
    return "Hello from a function"
```

In [ ]:

```
```

In [18]:

```
# Creating a fucntion to check whether the no. is positive, negative, or

def check_no(no):
    '''
    This fucntn tells if the given no. is odd or even
    Input : any valid int
    Output : odd even zero
    Created by : Harsh
    Last edited : 3 Aug 2022
    '''
    if type(no)==int:
        if no<0:
            print('--No is negative--')
        elif no>0:
            print('--No is positive--')
        else:
            print('--No is zero--')
    else:
        print('Not allowed')
```

In [17]:
```
while True:
    no = input('Enter the no. : ')
    if no=='done':
        break
    else:
        check_no(int(no))
```

```
Enter the no. : 0
--No is zero--
Enter the no. : -1
--No is negative--
Enter the no. : 11
--No is positive--
Enter the no. : done
```

In [ ]:

Fetching the documentation of the function:

```
function_name.__doc__
```

In [20]:
```
print(check_no.__doc__)
```

```

    This fucntn tells if the given no. is odd or even
    Input : any valid identifier
    Output : odd even zero
    
```

In [ ]:

Parameter v/s arguments

- difficulty level : parameter
- easy/hard : argument

```
def fname(parameter)
    pass

x = fname(argument) # peop;e gove argument
print(x)
```

In [ ]:

Note :

- The default return type of a fuctn is None
- Whenever a fuctn is created
  - a global frame is created
- when the code inside the fuctn is run
  - a specific room is created inside the frame & when the working of the fuctn is done the room gets destroyed again

In [ ]:

Differnet types of argument in python

- Default argument
- Positional argument
- Keyword argument
- Arbitrary argument

When-ever the fuctn is created the main goal is it should not crash in any scenerio and do execute it's work properly

```
In [6]: def calc(a,b):
            print(f'Addn of {a} & {b} is {a+b}')
            print(f'Mul of {a} & {b} is {a*b}')
            print(f'Div of {a} & {b} is {round(a/b,2)}')
```

```
In [7]: calc(2,3)
```

```
Addn of 2 & 3 is 5
Mul of 2 & 3 is 6
Div of 2 & 3 is 0.67
```

```
In [ ]:
```

```
In [8]: calc()
```

```
-----------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
Input In [8], in <module>
----> 1 calc()

TypeError: calc() missing 2 required positional arguments: 'a' and 'b'
```

```
In [9]: calc(12)
```

```
-----------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
Input In [9], in <module>
----> 1 calc(12)

TypeError: calc() missing 1 required positional argument: 'b'
```

```
In [ ]:
```

Python Default Arguments

- Function arguments can have default values in Python.
- We can provide a default value to an argument by using the assignment operator (=).

In [12]:
```
'''
to solve the above issue we will use the
default argument
'''
def calc(a=1,b=1):
    print(f'Addn of {a} & {b} is {a+b}')
    print(f'Mul of {a} & {b} is {a*b}')
    print(f'Div of {a} & {b} is {round(a/b,2)}')
```

In [13]:
```
calc()
```

```
Addn of 1 & 1 is 2
Mul of 1 & 1 is 1
Div of 1 & 1 is 1.0
```

In [ ]:

In [15]:
```
calc(2)
```

```
Addn of 2 & 1 is 3
Mul of 2 & 1 is 2
Div of 2 & 1 is 2.0
```

In [16]:
```
calc(2,3)
```

```
Addn of 2 & 3 is 5
Mul of 2 & 3 is 6
Div of 2 & 3 is 0.67
```

In [ ]:

Positional Arguments

- During a function call, values passed through arguments should be in the order of parameters in the function definition.
- This is called positional arguments.

When we call a function with some values, these values get assigned to the arguments according to their position.

In [19]:
```
calc(2,3)
```

```
Addn of 2 & 3 is 5
Mul of 2 & 3 is 6
Div of 2 & 3 is 0.67
```

In [ ]:

Keyword arguments

- Python allows functions to be called using keyword arguments.
- When we call functions in this way, the order (position) of the arguments can be changed.

In [17]:
```
calc(b=2,a=3)
```

```
Addn of 3 & 2 is 5
Mul of 3 & 2 is 6
Div of 3 & 2 is 1.5
```

In [ ]:
```
```

Arbitrary Arguments

- Sometimes, we do not know in advance the number of arguments that will be passed into a function.
- Python allows us to handle this kind of situation through function calls with an arbitrary number of arguments.

In the function definition,

- we use an asterisk (*) before the parameter name to denote this kind of argument. Here is an example.

In [22]:
```
def friends_name(*name):
    for i in name:
        print(f'hi friend {i},')
```

In [25]:
```
friends_name('suga','jin','rin','harsh')
```

```
hi friend suga,
hi friend jin,
hi friend rin,
hi friend harsh,
```

In [ ]:
```
```

In [26]:
```
def total_calc(*no):
    res = 0
    for i in no:
        res+=i
    return res
```

In [27]:
```
print(total_calc(1,2,3,4,5))
```

```
15
```

In [ ]:
```
```

NOte :

- Asterisk converts the inp type to tuple

```
In [28]: def total_calc(*no):
             print('no passed : ',no)
             print('type : ',type(no))
             res = 0
             for i in no:
                 res+=i
             return res
```

```
In [29]: print(total_calc(1,1,1,1))

         no passed :  (1, 1, 1, 1)
         type :  <class 'tuple'>
         4
```

```
In [ ]:
```

local variables :

- can be accessed only inside the function in which they are declared,

whereas global variables :

- can be accessed throughout the program body by all functions

Note :

- In the case of fuctn
  - If the local variable is not found the fucnt will use the global var
  - the value of the global var cannot be changed inside the local var
  - To change the value of the global var inside local var we need to use global keyword

Everything in python is an object

```
In [ ]:
```

```
In [30]: # everthing in py. is an obj even fuctn

         def total_calc(*no):
             print('no passed : ',no)
             print('type : ',type(no))
             res = 0
             for i in no:
                 res+=i
             return res

         # just like int is an obj similiarly fuctn is an obj

         # so we can use Aliasing
         x = total_calc
```

```
In [31]: x(1,2,3)

         no passed :  (1, 2, 3)
         type :  <class 'tuple'>

Out[31]: 6
```

In [33]:
```
type(x)
```
Out[33]:
```
function
```

In [34]:
```
# That means we can also del the fuctn
del total_calc
```

In [35]:
```
total_calc(11,11)
```
```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call l
ast)
Input In [35], in <module>
----> 1 total_calc(11,11)

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

In [ ]:

In [36]:
```
# everthing in py. is an obj even fuctn

def total_calc(*no):
    print('no passed : ',no)
    print('type : ',type(no))
    res = 0
    for i in no:
        res+=i
    return res

# just like int is an obj similiarly fuctn is an obj

# so we can use Aliasing
x = total_calc
```

In [37]:
```
 # so fuctn is a data type just like int
# so we can also store the fuctn inside the list as a item

l = [1,2,x]
```

In [38]:
```
l
```
Out[38]:
```
[1, 2, <function __main__.total_calc(*no)>]
```

In [39]:
```
# accessign the fuctn item/datatype inside the list
l[-1](1,1,1,10)
```
```
no passed :  (1, 1, 1, 10)
type :  <class 'tuple'>
```
Out[39]:
```
13
```

In [ ]:

In [42]:
```
l2 = [1,9,80,x(10,11)]
```

```
no passed :  (10, 11)
type :  <class 'tuple'>
```

In [43]:
```
l2
```

Out[43]:
```
[1, 9, 80, 21]
```

In [ ]:

Different types of awesome things we can do inside a fuctn :

- Renaming a fuctn
- Deleting a fuctn
- Storing a fuctn
- Returning a fuctn
- fuctn as argument

Benefits of using fucntions

- Code readability (debug easy,code organized)
- Code Modularity (code organizedbreak huge prog different module)
- Code Re-usability (write once use forever)

In [ ]: