In [1]:
```
#1 What is the difference between a function and a method in Python?
```

```
In Python, functions & methods are both block of code that perform specific tasks, but
they differ in their context and usage

Function-
         It is defined using the def keyword.
         It takes input parameter and returns a value.

Method-
       It is a function associated with an object or class.
       It operates on the data within that object or class.
       Methods are defined within a class and are called using dot notation on an
object.
       The first parameter is typically self, which refers to  the instance of class.
```

In [2]:
```
#example of Function
def add_numbers(x,y):
    return x+y
```

In [4]:
```
result=add_numbers(5,3)
print(result)
```

```
8
```

In [6]:
```
#Example of Method
class Dog:
    def bark(self):
        print('Woof!')
```

In [7]:
```
my_dog=Dog()
my_dog.bark()
```

```
Woof!
```

In [8]:
```
#2 Explain the concept of function arguments and parameters in Python?
```

```
In Python, function arguments and parameters are closely related but distinct concepts:

Parameters-
       are the variables listed inside the parenthesis in the function definition.They
act as placeholder for the values that will be passed to the function when it's called.
       Parameters define the input that a function expects to receive,allowing you to
create reusable code that can work with different values.

Arguments-
       are the actual values passed to a function when it's called.They are used to fill
in the placeholders defined by the parameters.
       Arguments provide the specific data that the function will work with during it's
execution.
```

In [9]:
```
#Example of Parameter
def greet(name):  #'name' is the parameter
    print('Hello',name)
```

In [10]:
```
#Example of Argument
greet('Alexa') # 'Alexa' is the argument
```

```
Hello Alexa
```

In [11]:
```
#3 What are the different ways to define and call a function in Python?
```

```
In Python,we can define functions using the def keyword, and call them by their name
followed by parenthesis.Here are the different ways:
```

In [29]:
```
#basic function statement
def fun():
    print('I love my India')
fun()
```

```
I love my India
```

In [31]:
```
#python function with Parameters
def add(num1:int,num2:int):
    num3=num1+num2
    return num3
add(2,3)
```

Out[31]:
```
5
```

In [32]:
```
#python function Arguments
def evenOdd(x):
    if (x%2==0):
        print('even')
    else:
        print('odd')
evenOdd(3)
evenOdd(10)
```

```
odd
even
```

In [33]:
```
def greet(name):
    print("Hello! " + name + " How are you?")

greet("John")
```

```
Hello! John How are you?
```

In [34]:
```
def subtractNum():
    print(34 - 4)

subtractNum()
```

```
30
```

In [36]:
```
#4 What is the purpose of the `return` statement in a Python function?
```

```
The Python return statement is a special statement that you can use inside a function or
method to send the function's result back to the caller. A return statement consists of
the return keyword followed by an optional return value.
```

```
The return statement causes your function to exit and hand back a value to its caller.
The point of functions in general is to take in inputs and return something. The return
statement is used when a function is ready to return a value to its caller.
```

In [ ]:
```
#5 What are iterators in Python and how do they differ from iterables?
```

```
Iterable is an object, that one can iterate over. It generates an Iterator when passed
to iter() method. An iterator is an object, which is used to iterate over an iterable
object using the __next__() method. Iterators have the __next__() method, which returns
the next item of the object.

Note: Every iterator is also an iterable, but not every iterable is an iterator in
Python.

For example, a list is iterable but a list is not an iterator. An iterator can be
created from an iterable by using the function iter(). To make this possible, the class
of an object needs either a method __iter__, which returns an iterator, with sequential
indexes starting with 0.
```

In [38]:
```
s="GFG"
s=iter(s)
print(s)
```

```
<str_iterator object at 0x000001EEF5AB7D60>
```

In [39]:
```
print(next(s))
```

```
G
```

In [40]:
```
print(next(s))
```

```
F
```

In [41]:
```
cities = ["Berlin", "Vienna", "Zurich"]
iterator_obj = iter(cities)

print(next(iterator_obj))
print(next(iterator_obj))
print(next(iterator_obj))
```

```
Berlin
Vienna
Zurich
```

In [42]:
```
#6 Explain the concept of generators in Python and how they are defined.
```

```
In Python, a generator is a function that returns an iterator that produces a sequence
of values when iterated over.

Generators are useful when we want to produce a large sequence of values, but we don't
want to store all of them in memory at once.

In Python, similar to defining a normal function, we can define a generator function
using the def keyword, but instead of the return statement we use the yield statement.
```

In [43]:
```
def my_generator(n):

    # initialize counter
    value = 0

    # loop until counter is less than n
    while value < n:

        # produce the current value of the counter
        yield value

        # increment the counter
        value += 1

# iterate over the generator object produced by my_generator
for value in my_generator(3):

    # print each value produced by generator
    print(value)
```

```
0
1
2
```

In [ ]:
```
#7 What are the advantages of using generators over regular functions?
```

```
A generator in Python is a special type of function that allows you to iterate over a
set of values. Unlike a regular function, a generator does not return its results all at
once. Instead, it yields its values one by one, each time it is called. This makes it
possible to generate an infinite sequence of values, as long as there is sufficient
memory to store them.

The syntax for creating a generator is similar to that of a regular function, with a few
key differences. To create a generator, you use the yield keyword instead of the return
keyword, and you wrap the function definition in parentheses instead of square brackets.

Advantages of using generators over regular functions-
Memory Efficiency
Readaptibility
Speed
```

In [44]:
```
#example of a simple generator that yields the first 10 numbers
def first_ten_numbers():
    for i in range(10):
        yield i
gen = first_ten_numbers()
for num in gen:
    print(num)
```

```
0
1
2
3
4
5
6
7
8
9
```

In [45]:
```
#8 What is a lambda function in Python and when is it typically used?
```

```
Lambda Function, also referred to as ‘Anonymous function’ is same as a regular python
function but can be defined without a name. While normal functions are defined using the
def keyword, anonymous functions are defined using the lambda keyword. However,they are
restricted to single line of expression. They can take in mutliple parameters as in
regular functions.

The syntax for lambda function is given by : lambda arguments: expression

There is no return statement which is usually present in the def function syntax.
Lambda functions reduce the number of lines of code.
```

In [46]:
```
squares = lambda x: x*x
print('Using lambda: ', squares(5))
```

```
Using lambda:  25
```

In [47]:
```
add=lambda a,b:a+b
add(2,5)
```

Out[47]:
```
7
```

In [48]:
```
#9 Explain the purpose and usage of the `map()` function in Python.
```

```
The map() function is used to apply a given function to every item of an iterable, such
as a list or tuple, and returns a map object (which is an iterator).

The syntax for the map() function
map(function, iterable)
```

In [49]:
```
#example of using map() to convert a list of strings into a list of integers
s = ['1', '2', '3', '4']
res = map(int, s)
print(list(res))
```

```
[1, 2, 3, 4]
```

In [50]:
```
a = [1, 2, 3, 4]
def double(val):
  return val*2

res = list(map(double, a))
print(res)
```

```
[2, 4, 6, 8]
```

In [51]:
```
a = [1, 2, 3, 4]
res = list(map(lambda x: x * 2, a))
print(res)
```

```
[2, 4, 6, 8]
```

In [52]:
```
#10 What is the difference between `map()`, `reduce()`, and `filter()` functions in Python
```

```
The map(), filter() and reduce() functions bring a bit of functional programming to
Python. All three of these are convenience functions that can be replaced with List
Comprehensions or loops.
```

All three of these methods expect a function object as the first argument. This function object can be a pre-defined method with a name (like def add(x,y))

In [53]:
```python
#The syntax is:map(function, iterable(s))
#Without using lambdas
def starts_with_A(s):
    return s[0] == "A"

fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]
map_object = map(starts_with_A, fruit)

print(list(map_object))
```

```
[True, False, False, True, False]
```

In [54]:
```python
#The syntax is:filter(function, iterable(s))
def starts_with_A(s):
    return s[0] == "A"

fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]
filter_object = filter(starts_with_A, fruit)

print(list(filter_object))
```

```
['Apple', 'Apricot']
```

In [55]:
```python
#The syntax is:reduce(function, sequence[, initial])
from functools import reduce

def add(x, y):
    return x + y

list = [2, 4, 7, 3]
print(reduce(add, list))
```

```
16
```

In [57]:
```python
#11 Using pen & Paper write the internal mechanism for sum operation using  reduce functio
#list:[47,11,42,13];
from functools import reduce

numbers = [47, 11, 42, 13]

sum_result = reduce(lambda x, y: x + y, numbers, 0)

print(sum_result)
```

```
113
```

In [ ]: