Week-1

1. What is Python? What are the benefits of using Python?

Python is a high-level, interpreted, general-purpose programming language. Being a general-purpose language, it can be used to build almost any type of application with the right tools/libraries. Additionally, python supports objects, modules, threads, exception-handling, and automatic memory management which help in modeling real-world problems and building applications to solve these problems.

Benefits of using Python:

- Python is a general-purpose programming language that has a simple, easy-to-learn syntax that emphasizes readability and therefore reduces the cost of program maintenance. Moreover, the language is capable of scripting, is completely open-source, and supports third-party packages encouraging modularity and code reuse.
- Its high-level data structures, combined with dynamic typing and dynamic binding, attract a huge community of developers for Rapid Application Development and deployment.

2. What is a dynamically typed language?

Before we understand a dynamically typed language, we should learn about what typing is. **Typing** refers to type-checking in programming languages. In a **strongly-typed** language, such as Python, **"1" + 2** will result in a type error since these languages don't allow for "type-coercion" (implicit conversion of data types). On the other hand, a **weakly-typed** language, such as Javascript, will simply output **"12"** as result.

Type-checking can be done at two stages:

- **Static -** Data Types are checked before execution.
- **Dynamic -** Data Types are checked during execution. Python is an interpreted language, executes each statement line by line and thus type-checking is done on the fly, during execution. Hence, Python is a Dynamically Typed Language.

3. What is an Interpreted language?

An Interpreted language executes its statements line by line. Languages such as Python, Javascript, R, PHP, and Ruby are prime examples of Interpreted languages. Programs written in an interpreted language runs directly from the source code, with no intermediary compilation step.

4. What is PEP 8 and why is it important?

PEP stands for **Python Enhancement Proposal**. A PEP is an official design document providing information to the Python community, or describing a new feature for Python or its processes. **PEP 8** is especially important since it documents the style guidelines for

Python Code. Apparently contributing to the Python open-source community requires you to follow these style guidelines sincerely and strictly.

5. What is Scope in Python?

Every object in Python functions within a scope. A scope is a block of code where an object in Python remains relevant. Namespaces uniquely identify all the objects inside a program. However, these namespaces also have a scope defined for them where you could use their objects without any prefix. A few examples of scope created during code execution in Python are as follows:

- A **local scope** refers to the local objects available in the current function.
- A **global scope** refers to the objects available throughout the code execution since their inception.
- A **module-level scope** refers to the global objects of the current module accessible in the program.
- An **outermost scope** refers to all the built-in names callable in the program. The objects in this scope are searched last to find the name referenced.

Note: Local scope objects can be synced with global scope objects using keywords such as global.

6. What are the common built-in data types in Python?

There are several built-in data types in Python. Although, Python doesn't require data types to be defined explicitly during variable declarations type errors are likely to occur if the knowledge of data types and their compatibility with each other are neglected. Python provides `type()` and `isinstance()` functions to check the type of these variables. These data types can be grouped into the following categories-

1. **None Type: None** keyword represents the null values in Python. Boolean equality operations can be performed using these NoneType objects.

2. **Numeric Type:** There are three distinct numeric types - `integers`, `floating-point numbers` and `complex numbers`. Additionally, `booleans` are a subtype of integers.

3. **Sequence Types:** According to Python Docs, there are three basic Sequence Types - `lists`, `tuples`, and `range` objects. Sequence types have the `in` and `not in` operators defined for traversing their elements. These operators share the same priority as the comparison operations.

4. **Mapping Types:** A mapping object can map hashable values to random objects in Python. Mapping objects are mutable and there is currently only one standard mapping type, the `dictionary`.

5. **Set Types:** Currently, Python has two built-in set types - `set` and `frozenset`. `set` type is mutable and supports methods like *add()* and *remove()*. `the frozenset` type is immutable and can't be modified after creation.

6. **Modules:** Module is an additional built-in type supported by the Python Interpreter. It supports one special operation, i.e., `attribute access`: *mymod.myobj*, where

mymod is a module and *myobj* references a name defined in m's symbol table. The module's symbol table resides in a very special attribute of the module `__dict__`, but direct assignment to this module is neither possible nor recommended.

7. **Callable Types:** Callable types are the types to which function calls can be applied. They can be **user-defined functions, instance methods, generator functions**, and some other **built-in functions, methods** and **classes**. Refer to the documentation at docs.python.org for a detailed view of the callable types.

Q.7. Explain the ternary operator in Python.

Unlike C++, we don't have ?: in Python, but we have this:

[on true] if [expression] else [on false]

If the expression is True, the statement under [on true] is executed. Else, that under [on false] is executed.

Below is how you would use it:

```
a,b=2,3
min=a if a<b else b
print(min)
```

Above will print 2.

```
# Run this cell to see the result.
a,b=10,12
print("Hi") if a<b else print("Bye")
```

```
Hi
```

Q.8 How to check if a string is alphanumeric or not?

- If all characters of a string are either alphabet or number then the string is alphanumeric.

There are different methods associated with string data types in python. Those are-

`isnumeric()` - to check if a string is numeric or not.

`isalpha()` - to check if a string is in the alphabet or not.

`isalnum()` - to check if a string is alphanumeric or not.

`isdigit()` - to check if all the characters in the string are digit values or not. Digits are decimal numbers, exponents, subscripts, etc.

`isdigit()` vs `isnumeric()` Link

```
#Point out difference of the above methods from following code snippets-

#Eg-1
print('123'.isnumeric())
print('123'.isalpha())
```

```
print('123'.isalnum())
print('123'.isdigit())
```

```
True
False
True
True
```

```
#Eg-2
print('123F'.isnumeric())
print('123F'.isalpha())
print('123F'.isalnum())
print('123F'.isdigit())
```

```
False
False
True
False
```

```
#Eg-3
print('abc'.isnumeric())
print('abc'.isalpha())
print('abc'.isalnum())
print('abc'.isdigit())
```

```
False
True
True
False
```

```
#Eg-4
print('123²'.isnumeric())
print('123²'.isalpha())
print('123²'.isalnum())
print('123²'.isdigit())
```

```
True
False
True
True
```

```
#Eg-5
print('Ⅰ Ⅲ Ⅷ'.isdigit())
print('Ⅰ Ⅲ Ⅷ'.isnumeric())
```

```
False
True
```

9: Secret Code

You are given two strings S and N both of the same length. S is a string of lower case alphabets and N is a string of numbers.

Your task is to generate a coded string from S using N and print the result. See examples for coding patterns.

```
Example-1
Input:
    S = abcdef
    N = 123456

Output:
    bdfhjl

Explanation-
a+1=b
b+2=d
c+3=f
.. ..
.. ..
f+6 = l

That's how-
abcdef -> bdfhjl

Example-2
Input:
    S = zwhjnko
    N = 3456789

Output:
    campusx
```

```
S='zwhjnko'
N='3456789'
alphabet='abcdefghijklmnopqrstuvwxyz'
code=''
for s,n in zip(S,N):
    i=alphabet.find(s)
    code+=alphabet[(i+int(n))%26]
print(code)
```

```
campusx
```

Q.10 What is slicing - pythonic way?

Slicing is a technique that allows us to retrieve only a part of a `list`, `tuple`, or `string`. For this, we use the slicing operator-> `[::]`.

`[<int>:<int>:<int>]` This operator takes int value.

- First int value denotes the starting index for slicing.

- Second int value denotes ending index, excluded. If `4 is given` then it will only consider up to index `3`.
- Third value denotes jump. If 2 is given, it will go from start to end as per the first two values, taking a jump of `1` value.

```
(1,2,3,4,5)[2:4]
```

will result in -> `(3,4)`

It extracts values within the mentioned indexes in the slicing operator. `[2:4]` means extract from intex `2` till `4` (index `4` value not included)

Say if you want every character leaving the next one from any string -- `'caaomupeufsqx'`.

All you need to use is

```
'caaomupeufsqx'[::2]
```

This above will result in -

```
campusx
```

```
s='caaomupeufsqx'

# if colon not given in the operator, it will only extract the value at that index
print(s[10])
```

```
{"type":"string"}
```

```
# if first value not given-- it will treat it as it start from start
print(s[:10])
```

```
caaomupeuf
```

```
# if second value not given-- it will treat it as it goes till end
print(s[2:])
```

```
aomupeufsqx
```

```
# if the third value is not given-- it will take every character, no jump.
print(s[1:10:])
```

```
aaomupeuf
```

```
# if the third value is negative-- it will take a reverse jump, for this to happen start need to be greater than end.
print(s)
print(s[::-1])
```

```
caaomupeufsqx
xqsfuepumoaac
```

```
# Third value =2-- taking 1 element jump
print(s)
print(s[::2])
```

```
caaomupeufsqx
campusx
```

Q 11. What Does the // Operator Do?

In Python, the / operator performs division and returns the quotient in the float.

For example: 5 / 2 returns 2.5

The // operator, on the other hand, returns the quotient in integer.

For example: 5 // 2 returns 2

Q 12. What Does the 'is' Operator Do?

The 'is' operator compares the id of the two objects.

list1=[1,2,3]

list2=[1,2,3]

list3=list1

list1 == list2 □ True

list1 is list2 □ False

list1 is list3 □ True

Q 13: Disadvantages of Python.

https://www.geeksforgeeks.org/disadvantages-of-python

Q14 How strings are stored in Python?

- https://stackoverflow.com/questions/19224059/how-strings-are-stored-in-python-memory-model
- https://www.quora.com/How-are-strings-stored-internally-in-Python-3

Q15 What is Zen of Python?

The Zen of Python is a collection of 19 "guiding principles" for writing computer programs that influence the design of the Python programming language.
https://en.wikipedia.org/wiki/Zen_of_Python

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.

- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one-- and preferably only one --obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea -- let's do more of those!

Q16 Identity operator (is) vs ==?

->> Here's the main difference between python "==" vs "is:"

The "is" keyword is used to compare the variables and string whether they are pointing to the same object or not. If both the variables (var1 and var2) refer to the same object, they will have the same ID.

The "==" operator will compare both the variables whether their values refer to the same object or not.

Both operators are used for comparing; however, their purpose is different. Thus, they are used in different scenarios.

In Python, everything is an object and is assigned some memory.

Identity operators: The "is" and "is not" keywords are called identity operators that compare objects based on their identity. Equality operator: The "==" and "!=" are called equality operators that compare the objects based on their values. It will call the *eq*() class method of the object on the left of the operator and check for equality.

```
# Equality operator
a=10
b=10

# Case 1:
# Return True because both a and b have the same value
print(a==b)
```

```
True
```

```
# Case 2:
# Return True because both a and b is pointing to the same object
print("Id of a",id(a))
print("Id of b", id(b))
```

```
a is b
```

```
Id of a 11126976
Id of b 11126976
```

```
True
```

```
# Case 3:
# Here variable a is assigned to new variable c,
# which holds same object and same memory location
c=a
print("Id of c",id(c))
```

```
a is c
True
```

```
Id of c 11126976
```

```
True
```

```
# Case 4:
# Here variable s is assigned a list,
# and q assigned a list values same as s but on slicing of list a new list is
generated
s=[1,2,3]
p=s
q=s[:]
print("id of p", id(p))
print("Id of s", id(s))
print("id of q", id (q))
print("Comapare- s == q", s==q)
print("Identity- s is q", s is q)
print("Identity- s is p", s is p)
print("Comapare- s == p", s==p)
```

```
id of p 140194485320512
Id of s 140194485320512
id of q 140194692120816
Comapare- s == q True
Identity- s is q False
Identity- s is p True
Comapare- s == p True
```

Q17 How Python is interpreted?

Python as a language is not interpreted or compiled. Interpreted or compiled is the property of the implementation. Python is a bytecode(set of interpreter readable instructions) interpreted generally. Source code is a file with .py extension.

Python compiles the source code to a set of instructions for a virtual machine. The Python interpreter is an implementation of that virtual machine. This intermediate format is called

“bytecode”. .py source code is first compiled to give .pyc which is bytecode. This bytecode can be then interpreted by the official CPython or JIT(Just in Time compiler) compiled by PyPy.

Q18 What does _ variables represent in Python?

GssksForGeeks Article

Underscore _ is considered as "I don't Care" or "Throwaway" variable in Python

- The underscore _ is used for ignoring the specific values. If you don’t need the specific values or the values are not used, just assign the values to underscore.

** Ignore a value when unpacking

** Ignore the index

```
# Ignore a value when unpacking
x, _, y = (1, 2, 3)

print("x-",x)
print("y-", y)
```

```
x- 1
y- 3
_ 2
```

```
#Ignore the index

# Say we want to print hello 5 times, we don't need index value

for _ in range(5):
    print('hello')
```

```
hello
hello
hello
hello
hello
```

Q19 Modules vs packages vs Library

Python uses some terms that you may not be familiar with if you’re coming from a different language. Among these are scripts, modules, packages, and libraries.

- A **script** is a Python file that’s intended to be run directly. When you run it, it should do something. This means that scripts will often contain code written outside the scope of any classes or functions.
- A **module** is a Python file that’s intended to be imported into scripts or other modules. It often defines members like classes, functions, and variables intended to be used in other files that import it.

- A **package** is a collection of related modules that work together to provide certain functionality. These modules are contained within a folder and can be imported just like any other modules. This folder will often contain a special `__init__` file that tells Python it's a package, potentially containing more modules nested within subfolders

- A **library** is an umbrella term that loosely means "a bundle of code." These can have tens or even hundreds of individual modules that can provide a wide range of functionality. Matplotlib is a plotting library. The Python Standard Library contains hundreds of modules for performing common tasks, like sending emails or reading JSON data. What's special about the Standard Library is that it comes bundled with your installation of Python, so you can use its modules without having to download them from anywhere.

These are not strict definitions. Many people feel these terms are somewhat open to interpretation. Script and module are terms that you may hear used interchangeably.

https://stackoverflow.com/questions/19198166/whats-the-difference-between-a-module-and-a-library-in-python

https://www.geeksforgeeks.org/what-is-the-difference-between-pythons-module-package-and-library/

Q20 Why 0.3 – 0.2 is not equal to 0.1 in Python?

The reason behind it is called "*precision*", and it's due to the fact that computers do not compute in Decimal, but in Binary. Computers do not use a base 10 system, they use a base 2 system (also called Binary code).

https://www.geeksforgeeks.org/why-0-3-0-2-is-not-equal-to-0-1-in-python/

```
# code
print(0.3 - 0.2)
print(0.3 - 0.2 == 0.1)
```

```
0.09999999999999998
False
1.0
```

Q21 Why ~3 = -4? By bit reversing it should be 4.

Simple reason is python *not(~)* of any no x in the computer is calculated as *-x - 1.*

Answer is- **not** is performed bitwise so if `x=3` in binary it will be `00000011` which on complementing will be `11111100`. When the value of this number is printed as a decimal integer, its base-10 value is assigned to the output.

Now as the python number is **signed** so it will be negative and positive both and signed numbers are stored as two's complement. That is, the above bit pattern, when converting to base10 for storage in two's complement, requires that the complement of the bit pattern be taken, except the first bit remains as it is because this is taken as sign bit: `10000011`

Then 1 is added to the bit pattern: `10000100`

This pattern, when converted to base10, is `-4`.

Note: First bit is taken as sign bit, if it's `1` meaning no is negative, otherwise positive.

Further reading-
https://webhelp.esri.com/arcgiSDEsktop/9.3/body.cfm?tocVisable=1&ID=-1&TopicName=How%20Bitwise%20Not%20works

Q 22 - Python Docstrings

https://www.geeksforgeeks.org/python-docstrings

Week-2 Python interview questions & answers for Lecture 4- List, Lecture 5 - Tuple, Set, Dictionary and Lecture 6 - Function and Recursion:

Q1: Mutablity vs Immutablity

Mutable is a fancy way of saying that the internal state of the object is **changed/mutated**. So, the simplest definition is: An object whose internal state can be changed is mutable. On the other hand, **immutable** doesn't allow any change in the object once it has been created.

https://www.mygreatlearning.com/blog/understanding-mutable-and-immutable-in-python/.

Q2: Homogenous vs Hetrogenous

- **Homogeneous** Data Structure – Data elements will be of the same data type (ex: Array).
- **Heterogeneous** Data Structure – Data elements may not be of the same data type (ex: List, Tuples, Sets etc…).

Q3: append() vs extend() vs insert()

Append

> It adds an element at the end of the list. The argument passed in the append function is added as a single element at the end of the list and the length of the list is increased by 1.

Extend

> This method appends each element of the iterable (tuple, string, or list) to the end of the list and increases the length of the list by the number of elements of the iterable passed as an argument.

Insert

This method can be used to insert a value at any desired position. It takes two arguments-element and the index at which the element has to be inserted.

https://www.geeksforgeeks.org/difference-between-append-extend-and-insert-in-python/

Q4: What is aliasing in list?

An object with more than one reference has more than one name, then the object is said to be aliased. Example: If a refers to an object and we assign b = a, then both variables refer to the same object:

```
a = [1, 2, 3]
b = a
b is a
```

```
True
```

Q5: Define cloning in list.

In order to modify a list and also keep a copy of the original, it is required to make a copy of the list itself, not just the reference. This process is called cloning, to avoid the ambiguity of the word "copy".

```
a=[1,2,'a']

b=a[:]

a is b
```

```
False
```

Q6: Write a program in Python to delete the first element from a list.

```
def deleteHead(list):
    del list[0]

a=[1,2,3]
print(a)
deleteHead(a)
print(a)
```

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

Q7: How can we distinguish between tuples and lists?

List and Tuple in Python are the classes of Python Data Structures. The **list is dynamic**, whereas the **tuple has static** characteristics. This means that **lists can be modified** whereas **tuples cannot be modified**, the **tuple is faster** than the list because of static in nature.

Lists are denoted by the square brackets **[]** but tuples are denoted as parenthesis **()**.

https://www.geeksforgeeks.org/python-difference-between-list-and-tuple/

Q8: Swap two numbers without using third variable

```
#Method -1 for numeric variable
a=10
b=20
print(a,b)
a = a+b
b = a-b
a = a-b
print(a, b)

#Metod-2 - applies everywhere
a='abc'
b=123
print(a, b)
a, b = b, a
print(a, b)
```

```
10 20
20 10
abc 123
123 abc
```

Q9: What is the use of map () function?

The `map()` function applies a given function to each item of an iterable (list, tuple etc.) and returns an iterator.

The `map()` function returns an object of map class. The returned value can be passed to functions like

- `list()` - to convert to list
- `set()` - to convert to a set, and so on.

```
# Way to use map()

numbers = [2, 4, 6, 8, 10]

# returns square of a number
def square(number):
  return number * number

# apply square() function to each item of the numbers list
squared_numbers_iterator = map(square, numbers)

# converting to list
squared_numbers = list(squared_numbers_iterator)
print(squared_numbers)
```

```
[4, 16, 36, 64, 100]
```

Q10: Write a code to sort dictionaries using a key.

```
Input: {2: ‘Apple’, 1:’Mango’, 3:’Orange’, 4:’Banana’}

Output: 1: Mango
2: Apple
3: Orange
4: Banana
```

Below is the code to sort dictionaries using the key:

```
dict1 = {2: 'Apple', 1:'Mango', 3:'Orange', 4:'Banana'}
print(sorted(dict1.keys()))
for key in sorted(dict1):
      print("Sorted dictionary using key:",(key, dict1[key]))
```

```
[1, 2, 3, 4]
Sorted dictionary using key: (1, 'Mango')
Sorted dictionary using key: (2, 'Apple')
Sorted dictionary using key: (3, 'Orange')
Sorted dictionary using key: (4, 'Banana')
```

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

Q11: Write a code to sort dictionaries on values.

```
Input: {2: ‘Apple’, 1:’Mango’, 3:’Orange’, 4:’Banana’}

Output: 1: Mango
2: Apple
3: Orange
4: Banana
```

Below is the code to sort dictionaries using the key:

```
dict1 = {2: 'Apple', 1:'Mango', 3:'Orange', 4:'Banana'}
print(sorted(dict1.values()))
print(dict(sorted(dict1.items(), key=lambda item: item[1])))
```

```
['Apple', 'Banana', 'Mango', 'Orange']
{2: 'Apple', 4: 'Banana', 1: 'Mango', 3: 'Orange'}
```

Q12: Print the numbers pattern mentioned below without using any conditional statements?

n - size of pattern Pattern - 01120221011202210112

Think about how this pattern is made.

Solution--

In the above pattern a number is always the sum of its previous 2 numbers modulo 3.

So it's a sequence with

```
a(0)=0
a(1)=1
a(n)=(a(n-1)+a(n-2))%3; n>=2
```

```
# Solution -
n=22
x=0
y=1
print("01",end='')
for i in range(20):
    z=(x+y)%3
    print(z,end='')
    x=y
    y=z
```

```
01120221011202210112022
```

Q13: Dictionary Conversion

Suppose you are given a dictionary -

```
dict1 = {"volvo":"car", "benz":"car", "yamaha":"bike", "hero":"bike"}
```

And you are asked to convert this dict to-

```
output = {"car":["volvo", "benz"], "bike":["yamaha", "hero"]}
```

Solution:

1. You iterate through the dictionary using .items(), which gives you both the key and the value.
2. You try to add the key to your list.
3. If the list doesn't yet exist (KeyError) because it's the first entry, you create it.

```
dict1 = {"volvo":"car", "benz":"car", "yamaha":"bike", "hero":"bike"}

output ={}
for k, it in dict1.items():
    try:
        output[it].append(k)
    except KeyError:
        output[it] = [k]

print(output)
```

```
{'car': ['volvo', 'benz'], 'bike': ['yamaha', 'hero']}
```

Q14 : Adding two binary numbers given as strings in Python

Follow- https://www.geeksforgeeks.org/python-program-to-add-two-binary-numbers/

```
#Python way
num1='1001'
num2 = '1001101'
print("Num1-",num1,'->', int(num1, 2))
print("Num2-",num2,'->',int(num2, 2))
res=int(num1, 2)+int(num2, 2)
print("res:-{:b}->{}".format(res, res))
```

```
Num1- 1001 -> 9
Num2- 1001101 -> 77
res:-1010110->86
```

Q15 : Python Program that takes an array of numbers as input and outputs the next number as output.

Suppose I take an `input`:

```
array a=[1,2,3.4]
```

The `output` should be:

```
array b=[1,2,3,5]
```

```
Input-2:
```

```
a=[9,9,9]
```

```
Output-2
```

```
a=[1,0,0,0]
```

Solution:

1. Convert all elements in the list as string
2. Convert str to int and the do 1 increment
3. Convert result of above to str then list

```
#Write your program
list1=[1,2,3,4]
s=''.join(map(str, list1))
res=int(s)+1
print(list(str(res)))
```

```
['1', '2', '3', '5']
```

Q16: Write a Program to convert date from yyyy-mm-dd format to dd-mm-yyyy format.

```
# Method-1
import re

def transform_date_format(date):
   return re.sub(r'(\d{4})-(\d{1,2})-(\d{1,2})', '\\3-\\2-\\1', date)

date_input = "2021-08-01"
print(transform_date_format(date_input))

# method 2
from datetime import datetime

new_date = datetime.strptime("2021-08-01", "%Y-%m-%d").strftime("%d:%m:%Y")

print(new_data)
```

Q17: Write a Program to match a string that has the letter ‘a’ followed by 4 to 8 'b’s.

```
import re

def match_text(txt_data):
       pattern = 'ab{4,8}'
       if re.search(pattern,  txt_data):  # search for pattern in txt_data
           return 'Match found'
       else:
           return('Match not found')


print(match_text("abc"))         # prints Match not found
print(match_text("aabbbbbc"))    # prints Match found
```

Q18: Write a function to add two integers >0 without using the plus operator.

We can use bitwise operators to achieve this.

```
def add_nums(num1, num2):
   while num2 != 0:
       data = num1 & num2
       num1 = num1 ^ num2
       num2 = data << 1
   return num1

print(add_nums(2, 10))
```

Q19: Write a program) which takes a sequence of numbers and check if all numbers are unique.

You can do this by converting the list to set by using set() method and comparing the length of this set with the length of the original list. If found equal, return True.

```
def check_distinct(data_list):
    if len(data_list) == len(set(data_list)):
```

```
        return True
    else:
        return False

print(check_distinct([1,6,5,8]))     #Prints True
print(check_distinct([2,2,5,5,7,8])) #Prints False
```

Q20: Differentiate between deep and shallow copies

- Shallow copy does the task of creating new objects storing references of original elements. This does not undergo recursion to create copies of nested objects. It just copies the reference details of nested objects.
- Deep copy creates an independent and new copy of an object and even copies all the nested objects of the original element recursively.

Q21: What are lambda functions?

Lambda functions are generally inline, anonymous functions represented by a single expression. They are used for creating function objects during runtime. They can accept any number of parameters. They are usually used where functions are required only for a short period. They can be used as:

```
mul_func = lambda x,y : x*y
print(mul_func(6, 4))  # output is 24
```

Week-3

1. What is a decorator in Python?

Python offers a unique feature called decorators.

Let's start with an analogy before getting to the technical definition of the decorators. When we mention the word "decorator", what enters your mind? Well, likely something that adds beauty to an existing object. An example is when we hang a picture frame to a wall to enhance the room.

Decorators in Python add some feature or functionality to an existing function without altering it.

Let's say we have the following simple function that takes two numbers as parameters and divides them.

```
def divide(first, second):
    print ("The result is:", first/second)
```

Now if we call this function by passing the two values 16 and 4, it will return the following output:

```
divide(16, 4)
```

The output is:

```
The result is: 4.0
```

```
def divide(first, second):
    print ("The result is:", first/second)

divide(16, 4)
```

```
The result is: 4.0
```

What will happen if we pass the number 4 first, and 16 after? The answer will be 0.25. But we don't want it to happen. We want a scenario where if we see that first < second, we swap the numbers and divide them. But we aren't allowed to change the function.

Let's create a decorator that will take the function as a parameter. This decorator will add the swipe functionality to our function.

```
def swipe_decorator(func):
    def swipe(first, second):
        if first < second:
            first, second = second, first
        return func(first, second)

    return swipe
```

Now we have generated a decorator for the divide() function. Let's see how it works.

```
divide = swipe_decorator(divide)
divide(4, 16)
```

The output is:

```
The result is: 4.0
```

We have passed the function as a parameter to the decorator. The decorator "swiped our values" and returned the function with swiped values. After that, we invoked the returned function to generate the output as expected.

```
def divide(first, second):
    print ("The result is:", first/second)

def swipe_decorator(func):
    def swipe(first, second):
        if first < second:
            first, second = second, first
        return func(first, second)
    return swipe

divide = swipe_decorator(divide)
divide(4, 16)
```

```
The result is: 4.0
```

```
# Another way of doing the same thing

def swipe_decorator(func):
    def swipe(first, second):
        if first < second:
            first, second = second, first
        return func(first, second)
    return swipe

@swipe_decorator
def divide(first, second):
    print ("The result is:", first/second)



divide(4, 16)
```

```
The result is: 4.0
```

2. How can you determine whether a class is a subclass of another class?

Ans:

This is accomplished by utilizing a Python function called issubclass(). The function returns true or false depending on if a class is a child of another class, indicating whether it is.

```
class ABC:
    pass

class PQR(ABC):
    pass

print(issubclass(ABC, PQR)) # False as ABC is not the child class of PQR
print(issubclass(PQR, ABC)) # True as PQR is a child os ABC
```

```
False
True
```

3.What does Python's MRO (Method Resolution Order) mean?

Ans:

Method Resolution Order is referred to as MRO. A class inherits from many classes under multiple inheritance. If we attempt to access a method by building an object from the child class, the methods of the child class are first searched for the method. If the method is not found in the child class, the inheritance classes are searched from left to right.

The `show` method is present in both the `Father` and `Mother` classes in the example presented below.

In MRO, methods and variables are searched from left to right because while conducting inheritance, `Father` class is written first and `Mother` class is written afterwards. So firstly

`Father` class will be searched for the show method if found then will get executed if not, `Mother` class will be searched.

```
# Example code
class Father:
  def __init__(self):
    print('You are in Father Class Constructor')

  def show(self):
    print("Father Class instance Method")

class Mother:
  def __init__(self):
    print("You are in Mother Class Constructor")

  def show(self):
    print("Mother Class instance Method")

class Son(Father, Mother):
    def __init__(self):
        print("You are in Son Class Constructor")

son = Son()
son.show()
```

```
You are in Son Class Constructor
Father Class instance Method
```

4. What's the meaning of single and double underscores in Python variable and method names

- Single Leading Underscore: `_var`
- Single Trailing Underscore: `var_`
- Double Leading Underscore: `__var`
- Double Leading and Trailing Underscore: `__var__`
- Single Underscore: `_`

1. **Single Leading Underscore:** `_var` are a Python naming convention that indicates a name is meant for internal use. It is generally not enforced by the Python interpreter and is only meant as a hint to the programmer.

Adding a single underscore in front of a variable name is more like someone putting up a tiny underscore warning sign that says:

> "Hey, this isn't really meant to be a part of the public interface of this class. Best to leave it alone."

```
class Test:
    def __init__(self):
        self.foo = 11
        self._bar = 23
```

```
t = Test()
print(t.foo) #Print 11

print(t._bar) # Print 23
```

```
11
23
```

1. **Single Trailing Underscore:** `var_` Sometimes the most fitting name for a variable is already taken by a keyword in the Python language. Therefore, names like class or def cannot be used as variable names in Python. In this case, you can append a single underscore to break the naming conflict:

```
def make_object(name, class):
    pass
```

```
  File "<ipython-input-1-88a174f47223>", line 1
    def make_object(name, class):
                            ^
SyntaxError: invalid syntax
```

```
def make_object(name, class_):
    pass
```

In summary, a single trailing underscore (postfix) is used by convention to avoid naming conflicts with Python keywords. This convention is defined and explained in PEP 8.

1. **Double Leading Underscore**: `__var`

A double underscore prefix causes the Python interpreter to rewrite the attribute name in order to avoid naming conflicts in subclasses.

This is also called *name mangling*—the interpreter changes the name of the variable in a way that makes it harder to create collisions when the class is extended later.

```
class Test:
    def __init__(self):
        self.foo = 11
        self._bar = 23
        self.__baz = 23

t = Test()
print(dir(t)) # This gives us a list with the object's attributes
```

```
['_Test__baz', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_bar', 'foo']
```

```
['_Test__baz', '__class__', '__delattr__', '__dict__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
```

```
'__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', '_bar', 'foo']
```

Let's take this list and look for our original variable names `foo`, `_bar`, and `__baz`. I promise you'll notice some interesting changes.

First of all, the `self.foo` variable appears unmodified as `foo` in the attribute list.

Next up, `self._bar` behaves the same way—it shows up on the class as `_bar`. Like I explained before, the leading underscore is just a convention in this case—a hint for the programmer.

However, with `self.__baz` things look a little different. When you search for `__baz` in that list, you'll see that there is no variable with that exact name.

So what happened to `__baz`?

> If you look closely, you'll see there's an attribute called `_Test__baz` on this object. This is the name mangling that the Python interpreter applies. It does this to protect the variable from getting overridden in subclasses.

This type of variable is also explained in the Private `attributes` and `methods` session.

1. **Double Leading and Trailing Underscore**: `__var__`

Double underscores __ are often referred to as "**dunders**" in the Python community. The reason is that double underscores appear quite often in Python code, and to avoid fatiguing their jaw muscles, Pythonistas often shorten "**double underscore**" to "**dunder**."

The names that have both leading and trailing double underscores are reserved for special use in the language. This rule covers things like `__init__` for object constructors, or `__call__` to make objects callable.

These dunder methods are often referred to as magic methods.

There are many dunder methods, here are some:-

`__str__`, `__repr__`, `__call__`, `__add__`, `__sub__`, `__len__` etc.

1. **Single Underscore** _: Sometimes used as a name for temporary or insignificant variables ("don't care). Also, it represents the result of the last expression in a Python REPL session.

5. What is the difference between OOP and SOP?

Object Oriented Programming	Structural Programming
Object-Oriented Programming is a type of programming which is based on objects rather than just functions and procedures	Provides logical structure to a program where programs are divided functions
Bottom-up approach	Top-down approach

Provides data hiding	Does not provide data hiding
Can solve problems of any complexity	Can solve moderate problems
Code can be reused thereby reducing redundancy	Does not support code reusability

6. Can you call the base class method without creating an instance?

Yes, you can call the base class without instantiating it if:

- It is a static method
- The base class is inherited by some other subclass

7. What are the limitations of inheritance?

- Increases the time and effort required to execute a program as it requires jumping back and forth between different classes.
- The parent class and the child class get tightly coupled.
- Any modifications to the program would require changes both in the parent as well as the child class
- Needs careful implementation else would lead to incorrect results

8. What is the difference between range() and xrange()?

- range() creates a static list that can be iterated through while checking some conditions. This is a function that returns a list with integer sequences.
- xrange() is same in functionality as range() but it does not return a list, instead it returns an object of xrange(). xrange() is used in generators for yielding.

range()	xrange()
In Python 3, xrange() is not supported; instead, the range() function is used to iterate in for loops.	The xrange() function is used in Python 2 to iterate in for loops.
It returns a list.	It returns a generator object as it doesn't really generate a static list at the run time.
It takes more memory as it keeps the entire list of iterating numbers in memory.	It takes less memory as it keeps only one number at a time in memory.

9. How to override the way objects are printed?

Use the `__str__` and the `__repr__` dunder methods.

Here's an example that demonstrates how an instance from the Person class can be nicely formatted when printed to the console.

```
class Person:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def __str__(self):
```

```
        return f"{self.first_name} {self.last_name} ({self.age})"

    def __repr__(self):
        return f"{self.first_name} {self.last_name} ({self.age})"

person = Person("John", "Doe", 30) # thanks to __str__
person
```

```
John Doe (30)
```

10. What is the difference between a class method, a static method and an instance method?

Let's begin by writing a (Python 3) class that contains simple examples for all three method types:

```
class MyClass:

    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

Instance Methods

The first method on `MyClass`, called `method`, is a regular instance method. That's the basic, no-frills method type you'll use most of the time. You can see the method takes one parameter, self, which points to an instance of MyClass when the method is called. But of course, instance methods can accept more than just one parameter.

Through the self parameter, instance methods can freely access attributes and other methods on the same object. This gives them a lot of power when it comes to modifying an object's state.

Not only can they modify object state, instance methods can also access the class itself through the `self.__class__` attribute. This means instance methods can also modify class state. This makes instance methods powerful in terms of access restrictions—they can freely modify state on the object instance and on the class itself.

Class Methods

Let's compare that to the second method, `MyClass.classmethod`. I marked this method with a `@classmethod` decorator to flag it as a class method. Instead of accepting a self parameter, class methods take a `cls` parameter that points to the class—and *not* the object instance—when the method is called.

Since the class method only has access to this `cls` argument, it can't modify object instance state. That would require access to self. However, class methods can still modify class state that applies across all instances of the class.

`Static Methods`

The third method, `MyClass.staticmethod` was marked with a `@staticmethod6` decorator to flag it as a static method.

This type of method doesn't take a `self` or a `cls` parameter, although, of course, it can be made to accept an arbitrary number of other parameters.

As a result, a static method cannot modify object state or class state. Static methods are restricted in what data they can access—they're primarily a way to namespace your methods.

Let's See Them in Action!

Let's take a look at how these methods behave in action when we call them. We'll start by creating an instance of the class and then calling the three different methods on it.

`MyClass` was set up in such a way that each method's implementation returns a tuple containing information we can use to trace what's going on and which parts of the class or object that method can access.

```
# Class

class MyClass:

    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

Here's what happens when we call an instance method:

```
obj = MyClass()
obj.method()
```

```
('instance method called', <__main__.MyClass at 0x7f912e031810>)
```

This confirms that, in this case, the instance method called method has access to the object instance (printed as `<MyClass instance>`) via the self argument.

When the method is called, Python replaces the `self` argument with the instance object, `obj`.

We could ignore the syntactic sugar provided by the `obj.method()` **dot-call syntax** and pass the instance object manually to get the same result:

```
MyClass.method(obj)
```

```
MyClass.method(obj)
```

```
('instance method called', <__main__.MyClass at 0x7f912e031810>)
```

Let's try out the **class method** next:

```
obj.classmethod()
```

```
('class method called', __main__.MyClass)
```

Calling `classmethod()` showed us that it doesn't have access to the `<MyClass instance>` object, but only to the `<class MyClass>` object, representing the class itself (**everything in Python is an object, even classes themselves**).

Notice how Python automatically passes the class as the first argument to the function when we call `MyClass.classmethod()`. Calling a method in Python through the **dot syntax** triggers this behavior. The `self` parameter on instance methods works the same way.

Please note that naming these parameters `self` and `cls` is just a convention. You could just as easily name them `the_object` and `the_class` and get the same result. All that matters is that they're positioned first in the parameter list for that particular method.

Time to call the **static method** now:

```
obj.staticmethod()
```

```
{"type":"string"}
```

Did you see how we called `staticmethod()` on the object and were able to do so successfully? Some developers are surprised when they learn that it's possible to call a static method on an object instance.

Behind the scenes, Python simply enforces the access restrictions by not passing in the `self` or the `cls` argument when a static method gets called using the dot syntax

This confirms that static methods can neither access the object instance state nor the class state. They work like regular functions but belong to the class' (and every instance's) namespace.

Now, let's take a look at what happens when we attempt to call these methods on the class itself, **without creating an object instance** beforehand:

```
# Class Method
print(MyClass.classmethod())
# Static method
print(MyClass.staticmethod())
#Instance Method
print(MyClass.method())
```

```
('class method called', <class '__main__.MyClass'>)
static method called

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-b561d87f2a57> in <module>
      4 print(MyClass.staticmethod())
      5 #Instance Method
----> 6 print(MyClass.method())

TypeError: method() missing 1 required positional argument: 'self'
```

We were able to call `classmethod()` and `staticmethod()` just fine, but attempting to call the instance method `method()` failed with a `TypeError`.

This is to be expected. This time we didn't create an object instance and tried calling an instance function directly on the class blueprint itself. This means there is no way for Python to populate the `self` argument and therefore the call fails with a `TypeError exception`.

This should make the distinction between these three method types a little more clear

Key Takeaways

- Instance methods need a class instance and can access the instance through `self`.
- Class methods don't need a class instance. They can't access the instance (`self`) but they have access to the class itself via `cls`.
- Static methods don't have access to `cls or self`. They work like regular functions but belong to the class' namespace.
- Static and class methods communicate and (to a certain degree) enforce developer intent about class design. This can have definite maintenance benefits.

Week-4

Q-1: Can you explain some use cases for using decorators in Python?

Decorators can be used for a variety of purposes, but one common use case is to add additional functionality to a function without having to modify the function itself. For example, you could use a decorator to add logging to a function so that you can keep track of when the function is called and what its arguments are. Decorators can also be used to cache the results of a function so that subsequent calls to the function are faster.

Q-2: How does Python's @property decorator work?

The @property decorator is used to create properties in a class. When used, the decorator will take the following syntax: @property(getter, setter, deleter, doc). The getter is used to get the value of the property, the setter is used to set the value of the property, the deleter is used to delete the property, and the doc is used to provide documentation for the property.

Q-3: How does Python's @synchronized decorator work?

The @synchronized decorator is used to create a lock on a method or function. This lock can be used to prevent race conditions from occurring. When the decorator is applied to a method, it will acquire a lock on the instance of the class that the method is being called on. This lock is then released when the method returns.

Q-4: Are there any limitations on using decorators?

Yes, there are some limitations. Decorators can only be used on functions and methods, and they need to be defined before the function or method they are decorating. Additionally, decorators can't be used on class methods that take the self argument.

Q-5: Is it possible to open multiple files using Python? If yes, then how?

Yes, it is possible to open multiple files using Python. You can do this by using the built-in open() function. When you call open(), you can specify the mode in which you want to open the file, as well as the name of the file. If you want to open multiple files, you can do so by passing a list of filenames to open().

Q-6: How should you handle exceptions when dealing with files in Python?

When working with files in Python, it is important to be aware of potential exceptions that could be raised. Some common exceptions include IOError, which is raised when there is an error reading or writing to a file, and ValueError, which is raised when there is an issue with the format of the data in the file. It is important to handle these exceptions appropriately in order to avoid potential data loss or corruption.

Q-7: Which functions allow us to check if we have reached the end of a file in Python?

The functions that allow us to check if we have reached the end of a file in Python are the eof() and tell() functions. The eof() function returns true if we have reached the end of the file, and the tell() function returns the current position of the file pointer.

Q-8: What is the usage of yield keyword in Python?

The yield keyword is used in Python to define generators. Generators are a special type of function that allow the programmer to return values one at a time, rather than returning all values at once. This can be useful when working with large data sets, as it allows the programmer to process the data one piece at a time, rather than having to load the entire data set into memory at once.

Q-9: Can you explain the difference between errors and exceptions in Python?

Errors are generally caused by things that are out of our control, like hardware failures or syntax errors in our code. Exceptions, on the other hand, are things that we can anticipate and handle gracefully. For example, we can write code to catch a ValueError exception that might be raised if we try to convert a string to an int and the string can't be parsed as a valid integer.

Q-10: How can you ensure that a certain code block runs no matter whether there's an exception or not?

You can use a finally block to ensure that a certain code block runs no matter what. A finally block will always execute, even if there is an exception.

Q-11: When should you use assertions instead of try/except blocks?

Assertions should be used to check for conditions that should never occur in your code. For example, if you are working with a list, you might want to assert that the list is never empty. If an assertion fails, it will raise an exception. Try/except blocks, on the other hand, should be used to handle conditions that might occur in your code. For example, if you are trying to open a file, you might use a try/except block to handle the case where the file does not exist.

Q-12: What's the difference between raising an error and throwing an exception in Python?

Raising an error will cause the program to stop running entirely. Throwing an exception will allow the program to continue running, but will generate an error message.

Q-13: What do you understand about custom exceptions in Python?

In Python, a custom exception is a user-defined exception class. Custom exceptions are typically used to indicate errors that are specific to your application. For example, if you were developing a software application that needed to connect to a database, you might create a custom exception class to handle any errors that occur when connecting to the database.

Q-14: What do you know about EAFP (Easier to Ask Forgiveness than Permission) programming style?

EAFP is a style of programming that is common in Python. It involves trying to execute a piece of code and catching any errors that occur. This is contrasted with the LBYL (Look Before You Leap) style, which involves checking for errors before trying to execute code. EAFP is often considered to be more Pythonic, as it is more in line with the Python philosophy of "we're all adults here" and trusting programmers to handle errors as they occur.

Q-15: What is Scope Resolution in Python?

Sometimes objects within the same scope have the same name but function differently. In such cases, scope resolution comes into play in Python automatically. A few examples of such behavior are:

- Python modules namely 'math' and 'cmath' have a lot of functions that are common to both of them - log10(), acos(), exp() etc. To resolve this ambiguity, it is necessary to prefix them with their respective module, like math.exp() and cmath.exp()
- Consider the code below, an object temp has been initialized to 10 globally and then to 20 on function call. However, the function call didn't change the value of the temp globally. Here, we can observe that Python draws a clear line between global and local variables, treating their namespaces as separate identities.

```
temp = 10   # global-scope variable
def func():
     temp = 20   # local-scope variable
     print(temp)
print(temp)   # output => 10
func()    # output => 20
print(temp)   # output => 10
```

This behavior can be overridden using the global keyword inside the function, as shown in the following example:

```
temp = 10   # global-scope variable
def func():
     global temp
     temp = 20   # local-scope variable
     print(temp)
print(temp)   # output => 10
func()    # output => 20
print(temp)   # output => 20
```

Q-16: What is pickling and unpickling?

Python library offers a feature - **serialization** out of the box. Serializing an object refers to transforming it into a format that can be stored, so as to be able to deserialize it, later on, to obtain the original object. Here, the **pickle** module comes into play.

Pickling:

- Pickling is the name of the serialization process in Python. Any object in Python can be serialized into a byte stream and dumped as a file in the memory. The process of pickling is compact but pickle objects can be compressed further. Moreover, pickle keeps track of the objects it has serialized and the serialization is portable across versions.
- The function used for the above process is `pickle.dump()`

Unpickling:

- Unpickling is the complete inverse of pickling. It deserializes the byte stream to recreate the objects stored in the file and loads the object to memory.
- The function used for the above process is `pickle.load()`

Q-17: What is the difference between .py and .pyc files?

- .py files contain the source code of a program. Whereas, .pyc file contains the bytecode of your program. We get bytecode after compilation of .py file (source code). *.pyc* files are not created for all the files that you run. It is only created for the files that you import.
- Before executing a python program, the python interpreter checks for the compiled files. If the file is present, the virtual machine executes it. If not found, it checks for the .py file. If found, compiles it to .pyc file and then the python virtual machine executes it.
- Having a .pyc file saves you the compilation time.

Q-18 Assert in python.

Python's `assert` statement is a debugging aid that tests a condition. If the assert condition is true, nothing happens, and your program continues to execute as normal. But if the condition evaluates to false, an `AssertionError` exception is raised with an optional error message.

- Python's assert statement is a debugging aid that tests a condition as an internal self-check in your program.
- Asserts should only be used to help developers identify bugs. They're not a mechanism for handling run-time errors.
- Asserts can be globally disabled with an interpreter setting.

Suppose you were building an online store with Python. You're working to add a discount coupon functionality to the system, and eventually you write the following apply_discount function:

```
def apply_discount(product, discount):
    price = int(product['price'] * (1.0 - discount))
    assert 0 <= price <= product['price']
    return price
```

Notice the assert statement in there? It will guarantee that, no matter what, discounted prices calculated by this function cannot be lower than $0 and they cannot be higher than the original price of the product.

Q-19: What are *args and **kwargs in Python?

`*args` and *`**kwargs`* parameters are nevertheless a highly useful feature in Python. And understanding their potency will make you a more effective developer. These are asked in interviews more often.

*What are *args and **kwargs parameters used for?*

They allow a function to accept optional arguments, so you can create flexible APIs in your modules and classes.

```
def foo(required, *args, **kwargs):
    print(required)
    if args:
        print(args)
    if kwargs:
        print(kwargs)
```

The above function requires at least one argument called "required," but it can accept extra positional and keyword arguments as well. If we call the function with additional

arguments, `args` will collect extra positional arguments as a tuple because the parameter name has a * prefix.

Likewise, `kwargs` will collect extra keyword arguments as a dictionary because the parameter name has a ** prefix.

Both `args` and `kwargs` can be empty if no extra arguments are passed to the function.

As we call the function with various combinations of arguments, you'll see how Python collects them inside the args and kwargs parameters.

Run below cells..

```
# Function
def foo(required, *args, **kwargs):
    print(required)
    if args:
        print(args)
    if kwargs:
        print(kwargs)


foo() # This will give error as there is one argument that is mandatory.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-adfb11582809> in <module>
----> 1 foo()
      2
      3 # This will give error as there is one argument that is mandatory.

TypeError: foo() missing 1 required positional argument: 'required'
```

```
foo('hello') # Will print hello
foo('hello', 1, 2, 3) # Will print hello and 1,2,3 as elements of a tuple
foo('hello', 1, 2, 3, key1='value', key2=999) # Will print hello and 1,2,3 as
elements of a tuple and a dict with key1 and key2 as keys.
```

```
hello
hello
(1, 2, 3)
hello
(1, 2, 3)
{'key1': 'value', 'key2': 999}
```

I want to make it clear that calling the parameters `args` and `kwargs` is simply a naming convention. The previous example would work just as well if you called them `*parms` and `**argv`. The actual syntax is just the asterisk (*) or double asterisk (**), respectively.

Forwarding Optional or Keyword Arguments

It's possible to pass optional or keyword parameters from one function to another. You can do so by using the argument-unpacking operators * and ** when calling the function you want to forward arguments to.

This also gives you an opportunity to modify the arguments before you pass them along. Here's an example:

```
def foo(x, *args, **kwargs):
    kwargs['name'] = 'Alice'
    new_args = args + ('extra', )
    bar(x, *new_args, **kwargs)
```

Key Takeaways

• *args and **kwargs let you write functions with a variable number of arguments in Python.
• *args collects extra positional arguments as a tuple. **kwargs collects the extra keyword arguments as a dictionary.
• The actual syntax is * and **. Calling them args and kwargs is just a convention (and one you should stick to).

Q-20 What does the following program print, and why?

```
increment_by_i = [lambda x: x + i for i in range(10)]
print( increment_by_i [3](4))
```

```
increment_by_i = [lambda x: x + i for i in range(10)]
print( increment_by_i [3](4))
```

```
13
```

***Solution**:*

The program prints 13 (=9 + 4) rather than the 7 (=3 + 4) than might be expected. This is because the functions created in the loop have the same scope. They use the same variable name, and consequently, all refer to the same variable, i, which is 10 at the end of the loop, hence the 13 (`9 from i and  + 4 from x -> vlaue given as paremeter`).

There are many ways to get the desired behavior. A reasonable approach is to return the lambda from a function, thereby avoiding the naming conflict.

```
def create_increment_function (x):
    return lambda y: y + x

increment_by_i = [ create_increment_function (i) for i in range(10)]
print( increment_by_i [3](4))
```

Q-21: Explain what each of these creational patterns is: builder, static factory, factory method, and abstract factory.

Solution:

Builder Pattern

The idea behind the **builder pattern** is to build a complex object in phases. It avoids mutability and inconsistent state by using an mutable inner class that has a build method that returns the desired object. Its key benefits are that it breaks down the construction process, and can give names to steps. Compared to a constructor, it deals far better with optional parameters and when the parameter list is very long.

Static Factory

A **static factory** is a function for construction of objects. Its key benefits are as follows: the function's name can make what it's doing much clearer compared to a call to a constructor. The function is not obliged to create a new object—in particular, it can return a flyweight. It can also return a subtype that's more optimized, e.g., it can choose to construct an object that uses an integer in place of a Boolean array if the array size is not more than the integer word size.

Factory Method

A **factory method** defines an interface for creating an object, but lets subclasses decide which class to instantiate. The classic example is a maze game with two modes—

1. one with regular rooms, and
2. one with magic rooms.

Below snippet implements the regular rooms.

```
from abc import ABC , abstractmethod

class Room(ABC):

    @abstractmethod
    def connect(self , room2):
        pass


class MazeGame(ABC):

    @abstractmethod
    def make_room (self):
        print("abstract make_room ")
        pass

    def addRoom(self , room):
        print("adding room")

    def __init__(self):
```

```
        room1 = self. make_room ()
        room2 = self. make_room ()
        room1.connect(room2)
        self.addRoom(room1)
        self.addRoom(room2)
```

Below snippet implements the magic rooms.

```
class MagicMazeGame (MazeGame):
    def make_room (self):
        return MagicRoom ()

class MagicRoom (Room):
    def connect(self , room2):
        print(" Connecting magic room")
```

Here's how you use the factory to create regular and magic games.

```
ordinary_maze_game = ordinary_maze_game . OrdinaryMazeGame ()
magic_maze_game = magic_maze_game . MagicMazeGame ()
```

A drawback of the **factory method** pattern is that it makes subclassing challenging

Abstract Factory

An **abstract factory** provides an interface for creating families of related objects without specifying their concrete classes. For example, a class DocumentCreator could provide interfaces to create a number of products, such as createLetter() and createResume(). Concrete implementations of this class could choose to implement these products in different ways, e.g., with modern or classic fonts, right-flush or right-ragged layout, etc. Client code gets a DocumentCreator object and calls its factory methods. Use of this pattern makes it possible to interchange concrete implementations without changing the code that uses them, even at runtime. The price for this flexibility is more planning and upfront coding, as well as code that may be harder to understand, because of the added indirections.

Week-5 Numpy

1. Why do developers prefer NumPy to similar tools like Matlab, Yorick?

Developers prefer to use NumPy over other tools like Matlab for the following reasons:

- NumPy is an accessible and open-source library. You can easily access it and use it anywhere.
- It is a high-performing library integrated with multidimensional arrays and matrics.
- Some popular libraries works on top of NumPy

  For Stats and ML: SciPy, Scikit-Learn, SpaCy, Statsmodels

Array Manipulation: Dask, PyTorch, TensorFlow

Visualization: Seaborn, Matplotlib, Altair, Bokeh, Plotly

- These libraries permit developers to quickly perform high-performance and complex mathematical, science, or finance calculations.
- As a developer, you can take advantage of the general purpose feature of Python as it makes it easy for you to connect with programming languages like C, C++, or Fortran.

2. Why is NumPy Array good compared to Python Lists?

NumPy is better than Python Lists for two primary reasons:

- NumPy Array is static and has a fixed size while creating codes. In other cases, Python Lists are dynamic and can grow dynamically.
- NumPy Array can perform vectorised operations and other advanced calculations, but Python Lists can't do these even after having a large set of functions.

3. What does bincount() function?

The main job of the `bincount()` function is to count the number of times a given value appears in the array of integers. You have to take care of one point that the function takes only positive integers or Boolean expressions as an argument. You can not pass negative integers.

Syntax: `array_object.bincount()`

```
import numpy as np
p = [10,20,30]
q = [5,1,7,1,8,13]
s = np.array([0, 2, 3, 0, 3, 3, 3, 0, 0, 4, 2, 1, 7, 5])
p = np.bincount(p)
q = np.bincount(q)
print("Occurrance of each digit in p:", p)
print("\nOccurance of each digit in q:", q)
print("\nOccurance of each digit in q:", np.bincount(s))
```

```
Occurrence of each digit in p: [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1]

Occurance of each digit in q: [0 2 0 0 0 1 0 1 1 0 0 0 0 1]

Occurance of each digit in q: [4 1 2 4 1 1 0 1]
```

4. Explain the data types supported by NumPy.

https://numpy.org/doc/stable/user/basics.types.html

```
numpy.bool_ : bool
numpy.byte : signed char
numpy.ubyte : unsigned char
numpy.short : short
numpy.ushort : unsigned short
numpy.intc : int
numpy.uintc : unsigned int
numpy.int_ : long
numpy.uint : unsigned long
numpy.longlong :  long long
numpy.ulonglong : unsigned long long
numpy.half / numpy.float16 : Half precision float
numpy.single : float
numpy.double : double
numpy.longdouble : long double
numpy.csingle : float complex
numpy.cdouble : double complex
numpy.clongdouble : long double complex
```

5. How does the flatten function differs from the ravel function?

Flatten function has been used for generating 1D versions of the multi-dimensional array. The ravel function does the same operation, but both have a difference.

The `flatten()` function always returns a copy. The `ravel()` function returns a view of the original array in most cases. Although you can't see or analyze it in the shown output, if you make changes in the array returned by `ravel()`, it might change the whole data of the array. This does not happen while using the flatten() function.

Additionally, you can use the ravel function with any easily parseable object but the flatten() function is used with true NumPy arrays.

And, last, `ravel()` is faster than `flatten()`.

`Syntax: array object.flatten() and array.object.ravel()`

5. Explaining Axis=0 or Axis=1

Follow - https://www.sharpsightlabs.com/blog/numpy-axes-explained/

6. What Is The Preferred Way To Check For An Empty (zero Element) Array?

If you are certain a variable is an array, then use the size attribute. If the variable may be a list or other sequence type, use len().

The size attribute is preferable to len for numpy array because:

```
a = np.zeros((1,0))
print(a.size) # Will print 0
print(len(a)) # will print 1
```

`numpy.any(array)` can also be used for this. If it returns `False` then the array is empty.

```
import numpy as np
a = np.zeros((1,0))
print(a.size)
len(list(a))
```

```
0
```

```
1
```

```
np.any(a)
```

```
False
```

7. How do you calculate the moving average?

Before knowing how to calculate moving average, know what it means. It refers to a series of averages of fixed-size subsets of the total observation sets. You can also call it running average, rolling average, or rolling means.

The number of observations and size of windows are required for this calculation. You can calculate the moving average using several methods.

Using convolve function is the simplest method which is based on discrete convolution. You have to use a method that calculates discrete convolution to get a rolling mean. You can convolve with a sequence of np.ones of a length equal to the desired sliding window length.

Syntax: `array object.convolve(data, window size)`

Explanation - https://stackoverflow.com/a/54628145

```
import numpy as np
def moving_average(x, wsize):
    return np.convolve(x, np.ones(wsize), 'valid') / wsize
data = np.array([1,3,2,7,4,8,2])
print(data)
print("\n Moving Average:")
print(moving_average(data,1))
```

```
[1 3 2 7 4 8 2]

 Moving Average:
[1. 3. 2. 7. 4. 8. 2.]
```

8. Vectorisation in NumPy.

Follow - https://www.askpython.com/python-modules/numpy/numpy-vectorization

9. What does numpy.r_ do (numpy)?

Documentation - https://numpy.org/doc/stable/reference/generated/numpy.r_.html

What it does is row-wise merging.

```
V = np.array([1,2,3,4,5,6 ])
Y = np.array([7,8,9,10,11,12])
print(np.r_[V[0],Y[0],V[3:5],Y[1],V[2],Y[-1]])
```

```
[ 1  7  4  5  8  3 12]
```

10. Finding local maxima/minima with Numpy in a 1D numpy array

```
[1,2,1,3,2,4,5,3]

Local MAXIMA -> [2, 3, 5] AT INDEX [1, 3, 6] - If both neighbor elements are
smaller.
Local MINIMA -> [1, 1, 2, 3] AT INDEX [0, 2, 4, 7] - If both neighbor
elements are greater.
```

Approach would be- `a = [1,2,1,3,2,4,5,3]` Say for any element at index `i` (other than extreme elements), for being maxima below condition need to be `True`.

```
a[i-1] < a[i] & a[i] > a[i+1] => a[i] > a[i-1] & a[i] > a[i+1]

So if we need to do two comparisons one for element in left and one for
element in right.
# Here element wise comparison will happen.
a[1:] > a[:-1] #  [2,1,3,2,4,5,3] > [1,2,1,3,2,4,5] =>
[True,False,True,False,True,True,False]
a[:-1] > a[1:] #  [1,2,1,3,2,4,5] > [2,1,3,2,4,5,3] =>
[False,True,False,True,False,False,True]
```

And at extreme we just need to compare with the right element for index-0 and with the left of the last element. So we will add True for index-0 left comparison and True for last index right comparison.

```
#Adding True for extreme ends. + represents concatenation
[True] + a[1:] > a[:-1] => [True] + [True,False,True,False,True,True,False]
a[:-1] > a[1:] + [True] => [False,True,False,True,False,False,True] + [True]
```

Now taking and of both side comparison

```
[True] + a[1:] > a[:-1] & a[:-1] > a[1:] + [True]
-> [True] + [True,False,True,False,True,True,False] &
[False,True,False,True,False,False,True] + [True]
```

We will use `np.r_` function for concatenating row wise. Result that we will get will be a boolean index for local maxima values. We will use this boolean index to get local maximas.

```
import numpy as np
arr = np.random.randint(0,100, 10)
print(arr)
bool_index = np.r_[True, arr[1:] > arr[:-1]] & np.r_[arr[:-1] > arr[1:],
True]
print(bool_index)
print(arr[bool_index])
```

```
[99 38 78 82 76 63 58 95  3  0]
[ True False False  True False False False  True False False]
[99 82 95]
```

```
#For minima we just need to reverse the comparison sign
import numpy as np
arr = np.random.randint(0,100, 10)
print(arr)
bool_index = np.r_[True, arr[1:] < arr[:-1]] & np.r_[arr[:-1] < arr[1:],
True]
print(bool_index)
print(arr[bool_index])
```

```
[97  2 78  1 36 85 29 42 45 76]
[False  True False  True False False  True False False False]
[ 2  1 29]
```

11. Rotate Array/ Image by 90 deg clockwise numpy.rot90(array, k)

k is int specifies no of rotation.
https://numpy.org/doc/stable/reference/generated/numpy.rot90.html

```
a = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]

Expected Output:
[[7, 4, 1],
 [8, 5, 2],
 [9, 6, 3]]
```

```
a = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

np.rot90(a, 3)
```

```
array([[7, 4, 1],
       [8, 5, 2],
       [9, 6, 3]])
```

12. Fredo and Array Update in python

Fredo is assigned a new task today. He is given an array `A` containing `N` integers. His task is to update all elements of the array to some minimum value `x`, such that the sum of this new array is strictly greater than the sum of the initial array.

Note that `x` should be as minimum as possible such that the sum of the new array is greater than the sum of the initial array.

```
a = [1,2,3,4,5]
Result-> 4
```

An initial sum of an array is `1+2+3+4+5=15` When we update all elements to 4, the sum of the array becomes `20` which is greater than the initial sum `15` . Note that if we had updated the array elements to `3`, then sum will be `15`, which is not greater than initial sum `15`. So, `4` is the minimum value to which array elements need to be updated.

```
a = np.array([1,2,3,4,5])
x = np.sum(a)//a.size + 1
print(x)
```

```
4
```

13. Get products of all element except at index

You have an array of integers, and for each index you want to find the product of every integer except the integer at that index. Write a function that takes an array of integers(1-D) and returns an array of the products.

```
Note: Think of a solution without using division operation. I don't know how
to do that, that's why I'm leaving it up to you.

The array [1, 7, 3, 4]

Would return:

[84, 12, 28, 21]

By calculating:

[7*3*4, 1*3*4, 1*7*4, 1*7*3]
```

```
#
arr = np.array([1, 7, 3, 4])

# Approach with division
print(np.prod(arr)/arr)
```

```
array([84., 12., 28., 21.])
```