

Python Interview Questions for Freshers

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 modelling 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:

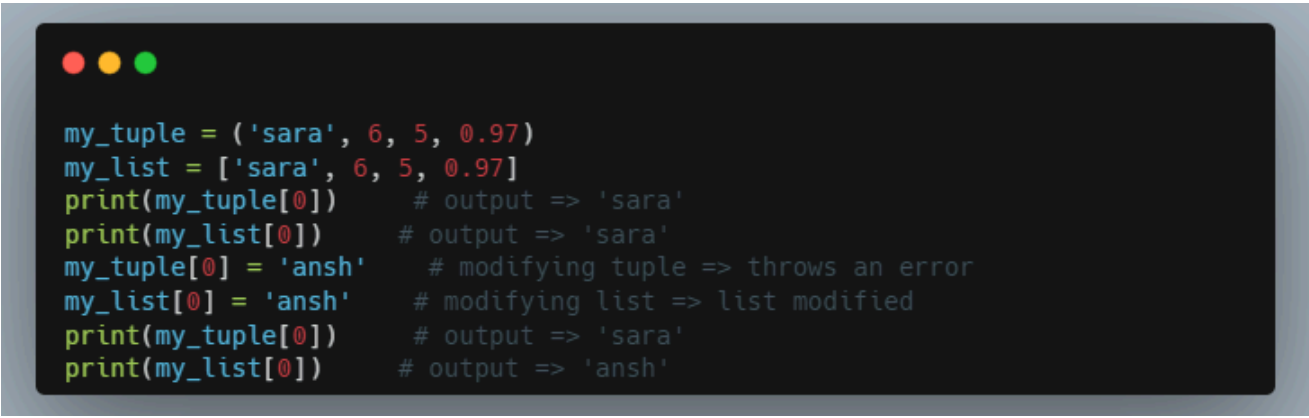
- A **local scope** refers to the local objects available in the current function.
- A **global scope** refers to the objects available throughout the code execution since their inception.
- A **module-level scope** refers to the global objects of the current module accessible in the program.
- An **outermost 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 lists and tuples? What is the key difference between the two?

Lists and **Tuples** are both **sequence data types** that can store a collection of objects in Python. The objects stored in both sequences can have **different data types**. Lists are represented with **square brackets** ['sara', 6, 0.19], while tuples are represented with **parentheses** ('ansh', 5, 0.97).

But what is the real difference between the two? The key difference between the two is that while **lists are mutable**, **tuples** on the other hand are **immutable** objects. This means that lists can be modified, appended or sliced on the go but tuples remain constant and cannot be modified in any manner. You can run the following example on Python IDLE to confirm the difference:



```

my_tuple = ('sara', 6, 5, 0.97)
my_list = ['sara', 6, 5, 0.97]
print(my_tuple[0])    # output => 'sara'
print(my_list[0])     # output => 'sara'
my_tuple[0] = 'ansh'  # modifying tuple => throws an error
my_list[0] = 'ansh'   # modifying list => list modified
print(my_tuple[0])    # output => 'sara'
print(my_list[0])     # output => 'ansh'

```

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


In Python, some common built-in data types include:

1. **Integers (int):** Represents whole numbers, positive or negative, without any decimal point.
2. **Floating-point numbers (float):** Represents real numbers, including numbers with a decimal point or in exponential form (e.g., 3.14, 2.7e5).
3. **Strings (str):** Represents sequences of characters enclosed within single, double, or triple quotes. Strings are immutable in Python.
4. **Booleans (bool):** Represents the truth values True and False, which are used for logical operations.
5. **Lists:** Ordered collections of items, which can be of different data types. Lists are mutable and can be modified.
6. **Tuples:** Similar to lists, but immutable. Once created, the elements within a tuple cannot be changed.
7. **Dictionaries (dict):** Key-value pairs where each key is unique and associated with a value. Dictionaries are unordered collections.
8. **Sets:** Unordered collections of unique elements. Sets are mutable but cannot contain duplicate elements.

These data types form the foundation of Python programming and are used extensively in writing code for various applications.

8. What is pass in Python?

The `pass` keyword represents a null operation in Python. It is generally used for the purpose of filling up empty blocks of code which may execute during runtime but has yet to be written. Without the `pass` statement in the following code, we may run into some errors during code execution.



```
def myEmptyFunc():
    # do nothing
    pass
myEmptyFunc()    # nothing happens
## Without the pass keyword
# File "<stdin>", line 3
# IndentationError: expected an indented block
```

9. What are modules and packages in Python?

Python packages and Python modules are two mechanisms that allow for **modular programming** in Python. Modularizing has several advantages -

- **Simplicity:** Working on a single module helps you focus on a relatively small portion of the problem at hand. This makes development easier and less error-prone.
- **Maintainability:** Modules are designed to enforce logical boundaries between different problem domains. If they are written in a manner that reduces interdependency, it is less likely that modifications in a module might impact other parts of the program.
- **Reusability:** Functions defined in a module can be easily reused by other parts of the application.
- **Scoping:** Modules typically define a separate namespace, which helps avoid confusion between identifiers from other parts of the program.

Modules, in general, are simply Python files with a .py extension and can have a set of functions, classes, or variables defined and implemented. They can be imported and initialized once using the import statement. If partial functionality is needed, import the requisite classes or functions using from foo import bar.

Packages allow for hierarchial structuring of the module namespace using **dot notation**. As, **modules** help avoid clashes between global variable names, in a similar manner, **packages** help avoid clashes between module names.

Creating a package is easy since it makes use of the system's inherent file structure. So just stuff the modules into a folder and there you have it, the folder name as the package name. Importing a module or its contents from this package requires the package name as prefix to the module name joined by a dot.

Note: You can technically import the package as well, but alas, it doesn't import the modules within the package to the local namespace, thus, it is practically useless.

10. What are global, protected and private attributes in Python?

- **Global** variables are public variables that are defined in the global scope. To use the variable in the global scope inside a function, we use the global keyword.


- **Protected** attributes are attributes defined with an underscore prefixed to their identifier eg. `_sara`. They can still be accessed and modified from outside the class they are defined in but a responsible developer should refrain from doing so.
- **Private** attributes are attributes with double underscore prefixed to their identifier eg. `__ansh`. They cannot be accessed or modified from the outside directly and will result in an `AttributeError` if such an attempt is made.

11. What is the use of self in Python?

Self is used to represent the instance of the class. With this keyword, you can access the attributes and methods of the class in python. It binds the attributes with the given arguments. `self` is used in different places and often thought to be a keyword. But unlike in C++, `self` is not a keyword in Python.

12. What is __init__?

`__init__` is a constructor method in Python and is automatically called to allocate memory when a new object/instance is created. All classes have a `__init__` method associated with them. It helps in distinguishing methods and attributes of a class from local variables.



```
# class definition
class Student:
    def __init__(self, fname, lname, age, section):
        self.firstname = fname
        self.lastname = lname
        self.age = age
        self.section = section
# creating a new object
stu1 = Student("Sara", "Ansh", 22, "A2")
```

What is break, continue and pass in Python?

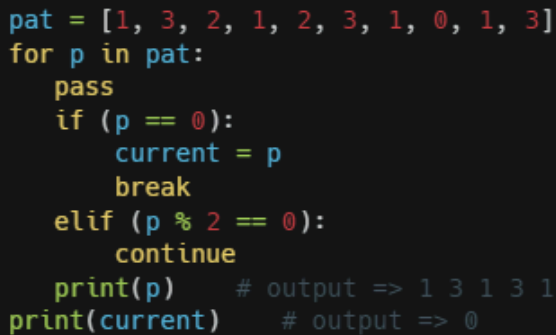
In Python, `break`, `continue`, and `pass` are control flow statements used within loops (such as `for` and `while` loops) and conditional statements (such as `if`, `elif`, and `else` statements) to alter the flow of execution.

break: When encountered within a loop, the `break` statement immediately exits the loop, regardless of whether the loop's condition has been fully satisfied. It is often used to prematurely terminate a loop when a certain condition is met.

continue: When encountered within a loop, the `continue` statement skips the remaining code within the current iteration of the loop and proceeds to the next iteration. It is useful for

skipping certain iterations based on a condition without exiting the entire loop

pass: The pass statement is a null operation and does nothing when executed. It is often used as a placeholder where syntactically required but no action is needed. It allows you to create empty code blocks without causing syntax errors.



```
pat = [1, 3, 2, 1, 2, 3, 1, 0, 1, 3]
for p in pat:
    pass
    if (p == 0):
        current = p
        break
    elif (p % 2 == 0):
        continue
    print(p)      # output => 1 3 1 3 1
print(current)   # output => 0
```

14. What are unit tests in Python?

- Unit test is a unit testing framework of Python.
- Unit testing means testing different components of software separately. Can you think about why unit testing is important? Imagine a scenario, you are building software that uses three components namely A, B, and C. Now, suppose your software breaks at a point time. How will you find which component was responsible for breaking the software? Maybe it was component A that failed, which in turn failed component B, and this actually failed the software. There can be many such combinations.
- This is why it is necessary to test each and every component properly so that we know which component might be highly responsible for the failure of the software.

15. What is docstring in Python?

- Documentation string or docstring is a multiline string used to document a specific code segment.
- The docstring should describe what the function or method does.

16. What is slicing in Python?

- As the name suggests, 'slicing' is taking parts of.
- Syntax for slicing is **[start : stop : step]**
- **start** is the starting index from where to slice a list or tuple
- **stop** is the ending index or where to stop.
- **step** is the number of steps to jump.
- Default value for **start** is 0, **stop** is number of items, **step** is 1.
- Slicing can be done on **strings, arrays, lists, and tuples**.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(numbers[1 :: 2]) #output : [2, 4, 6, 8, 10]
```

17. Explain how can you make a Python Script executable on Unix?

- Script file must begin with **#!/usr/bin/env python**

18. What is the difference between Python Arrays and lists?

- Arrays in python can only contain elements of same data types i.e., data type of array should be homogeneous. It is a thin wrapper around C language arrays and consumes far less memory than lists.
- Lists in python can contain elements of different data types i.e., data type of lists can be heterogeneous. It has the disadvantage of consuming large memory.

```
import array
a = array.array('i', [1, 2, 3])
for i in a:
    print(i, end=' ') #OUTPUT: 1 2 3
a = array.array('i', [1, 2, 'string']) #OUTPUT: TypeError: an integer is
required (got type str)
a = [1, 2, 'string']
for i in a:
    print(i, end=' ') #OUTPUT: 1 2 string
```

Python Interview Questions for Experienced

1. How is memory managed in Python?

- Memory management in Python is handled by the **Python Memory Manager**. The memory allocated by the manager is in form of a **private heap space** dedicated to Python. All Python objects are stored in this heap and being private, it is inaccessible to the programmer. Though, python does provide some core API functions to work upon the private heap space.
- Additionally, Python has an in-built garbage collection to recycle the unused memory for the private heap space.

2. What are Python namespaces? Why are they used?

A namespace in Python ensures that object names in a program are unique and can be used without any conflict. Python implements these namespaces as dictionaries with 'name as key' mapped to a corresponding 'object as value'. This allows for multiple namespaces to use the same name and map it to a separate object. A few examples of namespaces are as follows:

- **Local Namespace** includes local names inside a function. the namespace is temporarily created for a function call and gets cleared when the function returns.
- **Global Namespace** includes names from various imported packages/ modules that are being used in the current project. This namespace is created when the package is imported in the script and lasts until the execution of the script.
- **Built-in Namespace** includes built-in functions of core Python and built-in names for various types of exceptions.

The **lifecycle of a namespace** depends upon the scope of objects they are mapped to. If the scope of an object ends, the lifecycle of that namespace comes to an end. Hence, it isn't possible to access inner namespace objects from an outer namespace.

3. 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:

Follow @iron.coding

- 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

```

4. What are decorators in Python?

Decorators in Python are essentially functions that add functionality to an existing function in Python without changing the structure of the function itself. They are represented the `@decorator_name` in Python and are called in a bottom-up fashion. For example:

```

# decorator function to convert to lowercase
def lowercase_decorator(function):
    def wrapper():
        func = function()
        string_lowercase = func.lower()
        return string_lowercase
    return wrapper
# decorator function to split words
def splitter_decorator(function):
    def wrapper():
        func = function()
        string_split = func.split()
        return string_split
    return wrapper
@splitter_decorator # this is executed next
@lowercase_decorator # this is executed first
def hello():
    return 'Hello World'
hello() # output => [ 'hello' , 'world' ]

```

The beauty of the decorators lies in the fact that besides adding functionality to the output of the method, they can even **accept arguments** for functions and can further modify those arguments before passing it to the function itself. The **inner nested function**, i.e. 'wrapper' function, plays a significant role here. It is implemented to enforce **encapsulation** and thus, keep itself hidden from the global scope.

```

# decorator function to capitalize names
def names_decorator(function):
    def wrapper(arg1, arg2):
        arg1 = arg1.capitalize()
        arg2 = arg2.capitalize()
        string_hello = function(arg1, arg2)
        return string_hello
    return wrapper
@names_decorator
def say_hello(name1, name2):
    return 'Hello ' + name1 + '! Hello ' + name2 + '!'
say_hello('sara', 'ansh') # output => 'Hello Sara! Hello Ansh!'

```

5. What are Dict and List comprehensions?

Python comprehensions, like decorators, are **syntactic sugar** constructs that help **build altered** and **filtered lists**, dictionaries, or sets from a given list, dictionary, or set. Using comprehensions saves a lot of time and code that might be considerably more verbose (containing more lines of code). Let's check out some examples, where comprehensions can be truly beneficial:

- Performing mathematical operations on the entire list

Follow @iron.coding

```

my_list = [2, 3, 5, 7, 11]
squared_list = [x**2 for x in my_list] # list comprehension
# output => [4 , 9 , 25 , 49 , 121]
squared_dict = {x:x**2 for x in my_list} # dict comprehension
# output => {11: 121, 2: 4 , 3: 9 , 5: 25 , 7: 49}

```

- Performing conditional filtering operations on the entire list

```

my_list = [2, 3, 5, 7, 11]
squared_list = [x**2 for x in my_list if x%2 != 0] # list comprehension
# output => [9 , 25 , 49 , 121]
squared_dict = {x:x**2 for x in my_list if x%2 != 0} # dict comprehension
# output => {11: 121, 3: 9 , 5: 25 , 7: 49}

```

- Combining multiple lists into one

Comprehensions allow for multiple iterators and hence, can be used to combine multiple lists into one.

```
a = [1, 2, 3]
b = [7, 8, 9]
[(x + y) for (x,y) in zip(a,b)] # parallel iterators
# output => [8, 10, 12]
[(x,y) for x in a for y in b] # nested iterators
# output => [(1, 7), (1, 8), (1, 9), (2, 7), (2, 8), (2, 9), (3, 7), (3, 8), (3, 9)]
```

- **Flattening a multi-dimensional list**

A similar approach of nested iterators (as above) can be applied to flatten a multi-dimensional list or work upon its inner elements.

```
my_list = [[10,20,30],[40,50,60],[70,80,90]]
flattened = [x for temp in my_list for x in temp]
# output => [10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Note: List comprehensions have the same effect as the map method in other languages. They follow the mathematical set builder notation rather than map and filter functions in Python.

6. What is lambda in Python? Why is it used?

Lambda is an anonymous function in Python, that can accept any number of arguments, but can only have a single expression. It is generally used in situations requiring an anonymous function for a short time period. Lambda functions can be used in either of the two ways:

- Assigning lambda functions to a variable:

```
mul = lambda a, b : a * b
print(mul(2, 5)) # output => 10
```

- Wrapping lambda functions inside another function:

```
def myWrapper(n):
    return lambda a : a * n
mulFive = myWrapper(5)
print(mulFive(2))    # output => 10
```

7. How do you copy an object in Python?

In Python, the assignment statement (= operator) does not copy objects. Instead, it creates a binding between the existing object and the target variable name. To create copies of an object in Python, we need to use the **copy** module. Moreover, there are two ways of creating copies for the given object using the **copy** module -

Shallow Copy is a bit-wise copy of an object. The copied object created has an exact copy of the values in the original object. If either of the values is a reference to other objects, just the reference addresses for the same are copied.

Deep Copy copies all values recursively from source to target object, i.e. it even duplicates the objects referenced by the source object.

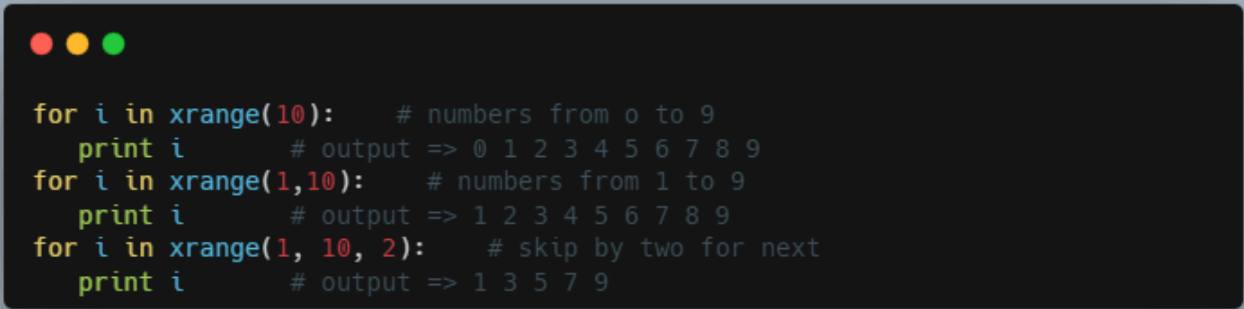
```
from copy import copy, deepcopy
list_1 = [1, 2, [3, 5], 4]
## shallow copy
list_2 = copy(list_1)
list_2[3] = 7
list_2[2].append(6)
list_2    # output => [1, 2, [3, 5, 6], 7]
list_1    # output => [1, 2, [3, 5, 6], 4]
## deep copy
list_3 = deepcopy(list_1)
list_3[3] = 8
list_3[2].append(7)
list_3    # output => [1, 2, [3, 5, 6, 7], 8]
list_1    # output => [1, 2, [3, 5, 6], 4]
```

8. What is the difference between xrange and range in Python?

xrange() and **range()** are quite similar in terms of functionality. They both generate a sequence of integers, with the only difference that **range()** returns a **Python list**, whereas, **xrange()** returns an **xrange object**.

So how does that make a difference? It sure does, because unlike **range()**, **xrange()** doesn't generate a static list, it creates the value on the go. This technique is commonly used with an object-type **generator** and has been termed as "**yielding**".

Yielding is crucial in applications where memory is a constraint. Creating a static list as in `range()` can lead to a Memory Error in such conditions, while, `xrange()` can handle it optimally by using just enough memory for the generator (significantly less in comparison).



```
for i in xrange(10):    # numbers from 0 to 9
    print i            # output => 0 1 2 3 4 5 6 7 8 9
for i in xrange(1,10):  # numbers from 1 to 9
    print i            # output => 1 2 3 4 5 6 7 8 9
for i in xrange(1, 10, 2): # skip by two for next
    print i            # output => 1 3 5 7 9
```

Note: `xrange` has been **deprecated** as of **Python 3.x**. Now `range` does exactly the same as what `xrange` used to do in **Python 2.x**, since it was way better to use `xrange()` than the original `range()` function in Python 2.x.

9. 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()`.

Note: Python has another, more primitive, serialization module called **marshall**, which exists primarily to **support .pyc files** in Python and **differs significantly from the pickle**.

10. What are generators in Python?

Generators are functions that return an iterable collection of items, one at a time, in a set manner. Generators, in general, are used to create iterators with a different approach. They

employ the use of yield keyword rather than return to return a **generator** object.
Let's try and build a generator for fibonacci numbers -

```
## generate fibonacci numbers upto n
def fib(n):
    p, q = 0, 1
    while(p < n):
        yield p
        p, q = q, p + q
x = fib(10)    # create generator object

## iterating using __next__(), for Python2, use next()
x.__next__()  # output => 0
x.__next__()  # output => 1
x.__next__()  # output => 1
x.__next__()  # output => 2
x.__next__()  # output => 3
x.__next__()  # output => 5
x.__next__()  # output => 8
x.__next__()  # error

## iterating using loop
for i in fib(10):
    print(i)    # output => 0 1 1 2 3 5 8
```

11. What is PYTHONPATH in Python?

PYTHONPATH is an environment variable which you can set to add additional directories where Python will look for modules and packages. This is especially useful in maintaining Python libraries that you do not wish to install in the global default location.

12. What is the use of help() and dir() functions?

help() function in Python is used to display the documentation of modules, classes, functions, keywords, etc. If no parameter is passed to the help() function, then an interactive **help utility** is launched on the console.

dir() function tries to return a valid list of attributes and methods of the object it is called upon. It behaves differently with different objects, as it aims to produce the most relevant data, rather than the complete information.

- For Modules/Library objects, it returns a list of all attributes, contained in that module.
- For Class Objects, it returns a list of all valid attributes and base attributes.
- With no arguments passed, it returns a list of attributes in the current scope.

13. 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 python interpreter checks for the compiled files. If the file is present, the virtual machine executes it. If not found, it checks for .py file. If found, compiles it to .pyc file and then python virtual machine executes it.
- Having .pyc file saves you the compilation time.

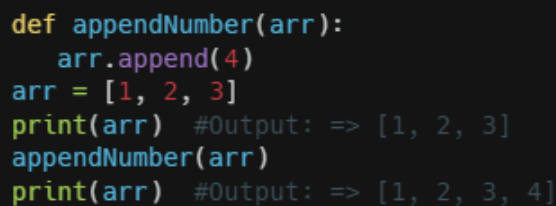
14. How Python is interpreted?

- 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.

15. How are arguments passed by value or by reference in python?

- **Pass by value:** Copy of the actual object is passed. Changing the value of the copy of the object will not change the value of the original object.
- **Pass by reference:** Reference to the actual object is passed. Changing the value of the new object will change the value of the original object.

In Python, arguments are passed by reference, i.e., reference to the actual object is passed.



```
def appendNumber(arr):  
    arr.append(4)  
arr = [1, 2, 3]  
print(arr) #Output: => [1, 2, 3]  
appendNumber(arr)  
print(arr) #Output: => [1, 2, 3, 4]
```

16. What are iterators in Python?

- An iterator is an object.
- It remembers its state i.e., where it is during iteration (see code below to see how)
- `__iter__()` method initializes an iterator.
- It has a `__next__()` method which returns the next item in iteration and points to the next element. Upon reaching the end of iterable object `__next__()` must return StopIteration exception.
- It is also self-iterable.

- Iterators are objects with which we can iterate over iterable objects like lists, strings, etc.

```
class ArrayList:
    def __init__(self, number_list):
        self.numbers = number_list
    def __iter__(self):
        self.pos = 0
        return self
    def __next__(self):
        if(self.pos < len(self.numbers)):
            self.pos += 1
            return self.numbers[self.pos - 1]
        else:
            raise StopIteration
array_obj = ArrayList([1, 2, 3])
it = iter(array_obj)
print(next(it)) #output: 2
print(next(it)) #output: 3
print(next(it))
#Throws Exception
#Traceback (most recent call last):
#...
#StopIteration
```

Follow @iron.coding

17. Explain how to delete a file in Python?

Use command **os.remove(file_name)**

```
import os
os.remove("ChangedFile.csv")
print("File Removed!")
```

18. Explain split() and join() functions in Python?

- You can use **split()** function to split a string based on a delimiter to a list of strings.
- You can use **join()** function to join a list of strings based on a delimiter to give a single string.

```
string = "This is a string."
string_list = string.split(' ') #delimiter is 'space' character or ' '
print(string_list) #output: ['This', 'is', 'a', 'string.']
print(' '.join(string_list)) #output: This is a string.
```

19. What does *args and **kwargs mean?

*args

- *args is a special syntax used in the function definition to pass variable-length arguments.
- “*” means variable length and “args” is the name used by convention. You can use any other.

```
def multiply(a, b, *argv):
    mul = a * b
    for num in argv:
        mul *= num
    return mul
print(multiply(1, 2, 3, 4, 5)) #output: 120
```

**kwargs

- **kwargs is a special syntax used in the function definition to pass variable-length keyworded arguments.
- Here, also, “kwargs” is used just by convention. You can use any other name.
- Keyworded argument means a variable that has a name when passed to a function.
- It is actually a dictionary of the variable names and its value.

```
def tellArguments(**kwargs):
    for key, value in kwargs.items():
        print(key + ": " + value)
tellArguments(arg1 = "argument 1", arg2 = "argument 2", arg3 = "argument 3")
#output:
# arg1: argument 1
# arg2: argument 2
# arg3: argument 3
```

20. What are negative indexes and why are they used?

- Negative indexes are the indexes from the end of the list or tuple or string.
- **Arr[-1]** means the last element of array **Arr[]**

```
arr = [1, 2, 3, 4, 5, 6]
#get the last element
print(arr[-1]) #output 6
#get the second last element
print(arr[-2]) #output 5
```

Python OOPS Interview Questions

1. How do you create a class in Python?

To create a class in python, we use the keyword “class” as shown in the example below:

```
class InterviewbitEmployee:
    def __init__(self, emp_name):
        self.emp_name = emp_name
```

To instantiate or create an object from the class created above, we do the following:

```
emp_1=InterviewbitEmployee("Mr. Employee")
```

To access the name attribute, we just call the attribute using the dot operator as shown below:

```
print(emp_1.emp_name)  
# Prints Mr. Employee
```

To create methods inside the class, we include the methods under the scope of the class as shown below:

Follow @iron.coding

```
class InterviewbitEmployee:  
    def __init__(self, emp_name):  
        self.emp_name = emp_name  
  
    def introduce(self):  
        print("Hello I am " + self.emp_name)
```

The self parameter in the init and introduce functions represent the reference to the current class instance which is used for accessing attributes and methods of that class. The self parameter has to be the first parameter of any method defined inside the class. The method of the class InterviewbitEmployee can be accessed as shown below:

```
emp_1.introduce()
```

The overall program would look like this:

```
class InterviewbitEmployee:
    def __init__(self, emp_name):
        self.emp_name = emp_name

    def introduce(self):
        print("Hello I am " + self.emp_name)

# create an object of InterviewbitEmployee class
emp_1 = InterviewbitEmployee("Mr Employee")
print(emp_1.emp_name)      #print employee name
emp_1.introduce()          #introduce the employee
```

2. How does inheritance work in python? Explain it with an example.

Inheritance gives the power to a class to access all attributes and methods of another class. It aids in code reusability and helps the developer to maintain applications without redundant code. The class inheriting from another class is a child class or also called a derived class. The class from which a child class derives the members are called parent class or superclass.

Python supports different kinds of inheritance, they are:

- **Single Inheritance:** Child class derives members of one parent class.

```
# Parent class
class ParentClass:
    def par_func(self):
        print("I am parent class function")

# Child class
class ChildClass(ParentClass):
    def child_func(self):
        print("I am child class function")

# Driver code
obj1 = ChildClass()
obj1.par_func()
obj1.child_func()
```

- **Multi-level Inheritance:** The members of the parent class, A, are inherited by child class which is then inherited by another child class, B. The features of the base class and the derived class are further inherited into the new derived class, C. Here, A is the grandfather class of class C.

```
# Parent class
class A:
    def __init__(self, a_name):
        self.a_name = a_name

# Intermediate class
class B(A):
    def __init__(self, b_name, a_name):
        self.b_name = b_name
        # invoke constructor of class A
        A.__init__(self, a_name)

# Child class
class C(B):
    def __init__(self, c_name, b_name, a_name):
        self.c_name = c_name
        # invoke constructor of class B
        B.__init__(self, b_name, a_name)

    def display_names(self):
        print("A name : ", self.a_name)
        print("B name : ", self.b_name)
        print("C name : ", self.c_name)

# Driver code
obj1 = C('child', 'intermediate', 'parent')
print(obj1.a_name)
obj1.display_names()
```

- **Multiple Inheritance:** This is achieved when one child class derives members from more than one parent class. All features of parent classes are inherited in the child class.

```
# Parent class1
class Parent1:
    def parent1_func(self):
        print("Hi I am first Parent")

# Parent class2
class Parent2:
    def parent2_func(self):
        print("Hi I am second Parent")

# Child class
class Child(Parent1, Parent2):
    def child_func(self):
        self.parent1_func()
        self.parent2_func()

# Driver's code
obj1 = Child()
obj1.child_func()
```

Follow @iron.coding

Hierarchical Inheritance: When a parent class is derived by more than one child class, it is called hierarchical inheritance.


```
# Base class
class A:
    def a_func(self):
        print("I am from the parent class.")

# 1st Derived class
class B(A):
    def b_func(self):
        print("I am from the first child.")

# 2nd Derived class
class C(A):
    def c_func(self):
        print("I am from the second child.")

# Driver's code
obj1 = B()
obj2 = C()
obj1.a_func()
obj1.b_func()    #child 1 method
obj2.a_func()
obj2.c_func()    #child 2 method
```

Follow @iron.coding

3. How do you access parent members in the child class?

Following are the ways using which you can access parent class members within a child class:

- **By using Parent class name:** You can use the name of the parent class to access the attributes as shown in the example below:

```
class Parent(object):
    # Constructor
    def __init__(self, name):
        self.name = name

class Child(Parent):
    # Constructor
    def __init__(self, name, age):
        Parent.name = name
        self.age = age

    def display(self):
        print(Parent.name, self.age)

# Driver Code
obj = Child("Interviewbit", 6)
obj.display()
```

By using super(): The parent class members can be accessed in child class using the super keyword.

Follow @iron.coding

```

class Parent(object):
    # Constructor
    def __init__(self, name):
        self.name = name

class Child(Parent):
    # Constructor
    def __init__(self, name, age):
        '''
        In Python 3.x, we can also use super().__init__(name)
        '''
        super(Child, self).__init__(name)
        self.age = age

    def display(self):
        # Note that Parent.name cant be used
        # here since super() is used in the constructor
        print(self.name, self.age)

# Driver Code
obj = Child("Interviewbit", 6)
obj.display()

```

Follow @iron.coding

4. Are access specifiers used in python?

Python does not make use of access specifiers specifically like private, public, protected, etc. However, it does not derive this from any variables. It has the concept of imitating the behaviour of variables by making use of a single (protected) or double underscore (private) as prefixed to the variable names. By default, the variables without prefixed underscores are public.

Example:

```
# to demonstrate access specifiers
class InterviewbitEmployee:

    # protected members
    _emp_name = None
    _age = None

    # private members
    __branch = None

    # constructor
    def __init__(self, emp_name, age, branch):
        self._emp_name = emp_name
        self._age = age
        self.__branch = branch

    #public member
    def display():
        print(self._emp_name + " "+self._age+" "+self.__branch)
```

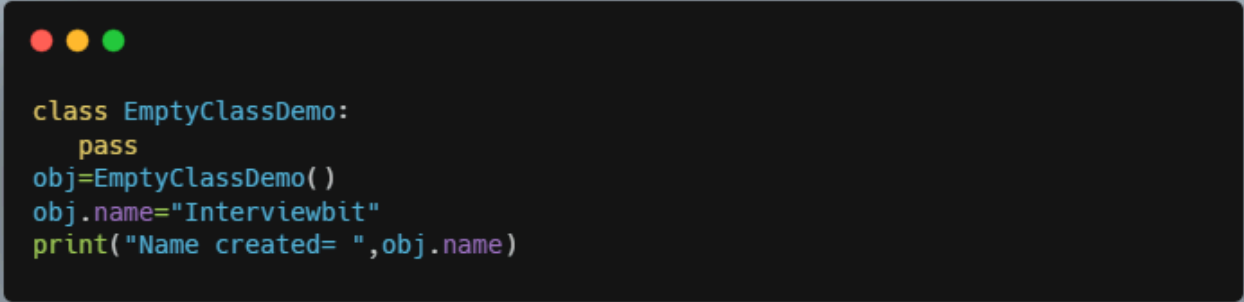
5. Is it possible to call parent class without its instance creation?

Yes, it is possible if the base class is instantiated by other child classes or if the base class is a static method.

6. How is an empty class created in python?

An empty class does not have any members defined in it. It is created by using the pass keyword (the pass command does nothing in python). We can create objects for this class outside the class.

For example-



```
class EmptyClassDemo:
    pass
obj=EmptyClassDemo()
obj.name="Interviewbit"
print("Name created= ",obj.name)
```

Output:

Name created = Interviewbit

7. Differentiate between new and override modifiers.

The new modifier is used to instruct the compiler to use the new implementation and not the base class function. The Override modifier is useful for overriding a base class function inside the child class.

8. Why is finalize used?

Finalize method is used for freeing up the unmanaged resources and clean up before the garbage collection method is invoked. This helps in performing memory management tasks.

9. What is init method in python?

The **init** method works similarly to the constructors in Java. The method is run as soon as an object is instantiated. It is useful for initializing any attributes or default behaviour of the object at the time of instantiation.

For example:

```

class InterviewbitEmployee:

    # init method / constructor
    def __init__(self, emp_name):
        self.emp_name = emp_name

    # introduce method
    def introduce(self):
        print('Hello, I am ', self.emp_name)

emp = InterviewbitEmployee('Mr Employee')    # __init__ method is called here
and initializes the object name with "Mr Employee"
emp.introduce()

```

10. How will you check if a class is a child of another class?

This is done by using a method called **issubclass()** provided by python. The method tells us if any class is a child of another class by returning true or false accordingly.

For example:

Follow @iron.coding

```

class Parent(object):
    pass

class Child(Parent):
    pass

# Driver Code
print(issubclass(Child, Parent))    #True
print(issubclass(Parent, Child))    #False

```

- We can check if an object is an instance of a class by making use of **isinstance()** method:

```
obj1 = Child()
obj2 = Parent()
print(isinstance(obj2, Child))    #False
print(isinstance(obj2, Parent))   #True
```

Python Pandas Interview Questions

1. What do you know about pandas?

- Pandas is an open-source, python-based library used in data manipulation applications requiring high performance. The name is derived from “Panel Data” having multidimensional data. This was developed in 2008 by Wes McKinney and was developed for data analysis.
- Pandas are useful in performing 5 major steps of data analysis - Load the data, clean/manipulate it, prepare it, model it, and analyze the data.

2. Define pandas dataframe.

A dataframe is a 2D mutable and tabular structure for representing data labelled with axes - rows and columns.

The syntax for creating dataframe:

```
import pandas as pd
```

```
dataframe = pd.DataFrame( data, index, columns, dtype)
```


where:

- data - Represents various forms like series, map, ndarray, lists, dict etc.
- index - Optional argument that represents an index to row labels.
- columns - Optional argument for column labels.
- Dtype - the data type of each column. Again optional.

3. How will you combine different pandas dataframes?

The dataframes can be combined using the below approaches:

- **append() method:** This is used to stack the dataframes horizontally. Syntax:




```
df1.append(df2)
```

- **concat() method:** This is used to stack dataframes vertically. This is best used when the dataframes have the same columns and similar fields. Syntax:



```
pd.concat([df1, df2])
```

- **join() method:** This is used for extracting data from various dataframes having one or more common columns.



```
df1.join(df2)
```

Follow @iron.coding

4. Can you create a series from the dictionary object in pandas?

One dimensional array capable of storing different data types is called a series. We can create pandas series from a dictionary object as shown below:

```
import pandas as pd
dict_info = {'key1' : 2.0, 'key2' : 3.1, 'key3' : 2.2}
series_obj = pd.Series(dict_info)
print (series_obj)
Output:
x      2.0
y      3.1
z      2.2
dtype: float64
```

If an index is not specified in the input method, then the keys of the dictionaries are sorted in ascending order for constructing the index. In case the index is passed, then values of the index label will be extracted from the dictionary.

5. How will you identify and deal with missing values in a dataframe?

We can identify if a dataframe has missing values by using the `isnull()` and `isna()` methods.

Follow @iron.coding

```
missing_data_count=df.isnull().sum()
```

We can handle missing values by either replacing the values in the column with 0 as follows:

```
df['column_name'].fillna(0)
```

Or by replacing it with the mean value of the column

```
df['column_name'] = df['column_name'].fillna((df['column_name'].mean()))
```

6. What do you understand by reindexing in pandas?

Reindexing is the process of conforming a dataframe to a new index with optional filling logic. If the values are missing in the previous index, then NaN/NA is placed in the location. A new object is returned unless a new index is produced that is equivalent to the current one. The copy value is set to False. This is also used for changing the index of rows and columns in the dataframe.

7. How to add new column to pandas dataframe?

A new column can be added to a pandas dataframe as follows:

Follow @iron.coding

```
import pandas as pd
data_info = {'first' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
            'second' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(data_info)
#To add new column third
df['third']=pd.Series([10,20,30],index=['a','b','c'])
print (df)
#To add new column fourth
df['fourth']=df['first']+info['third']
print (df)
```

8. How will you delete indices, rows and columns from a dataframe?

To delete an Index:

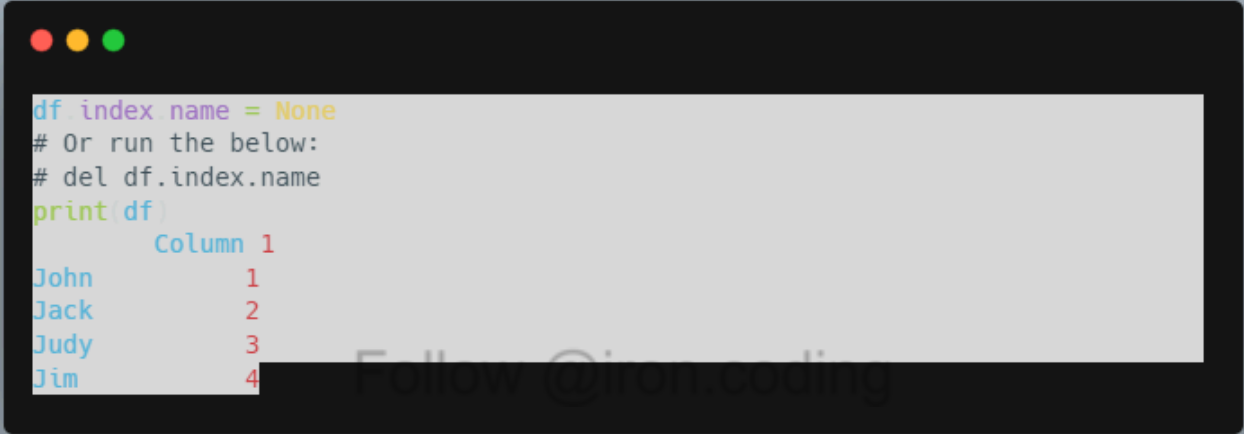
- Execute `del df.index.name` for removing the index by name.
- Alternatively, the `df.index.name` can be assigned to `None`.
- For example, if you have the below dataframe:

Column 1

Names

John	1
Jack	2
Judy	3
Jim	4

- To drop the index name “Names”:



```
df.index.name = None
# Or run the below:
# del df.index.name
print(df)
```

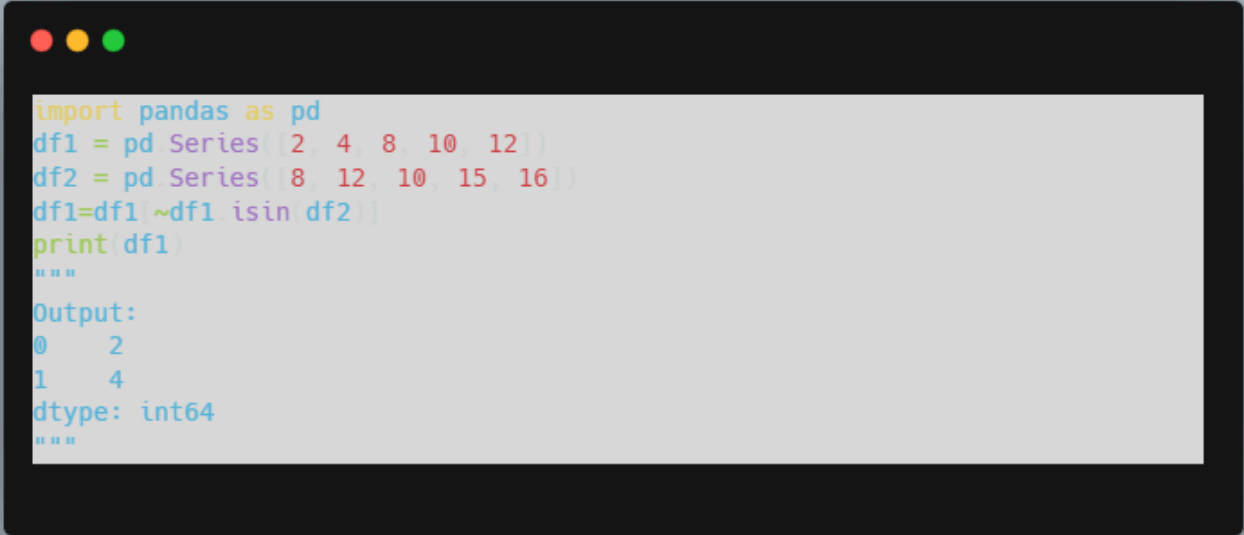
	Column 1
John	1
Jack	2
Judy	3
Jim	4

To delete row/column from dataframe:

- drop() method is used to delete row/column from dataframe.
- The axis argument is passed to the drop method where if the value is 0, it indicates to drop/delete a row and if 1 it has to drop the column.
- Additionally, we can try to delete the rows/columns in place by setting the value of inplace to True. This makes sure that the job is done without the need for reassignment.
- The duplicate values from the row/column can be deleted by using the drop_duplicates() method.

9. Can you get items of series A that are not available in another series B?

This can be achieved by using the ~ (not/negation symbol) and isin() method as shown below.



```
import pandas as pd
df1 = pd.Series([2, 4, 8, 10, 12])
df2 = pd.Series([8, 12, 10, 15, 16])
df1=df1[~df1.isin(df2)]
print df1
"""
Output:
0    2
1    4
dtype: int64
"""
```

10. How will you get the items that are not common to both the given series A and B?

We can achieve this by first performing the union of both series, then taking the intersection of both series. Then we follow the approach of getting items of union that are not there in the list of the intersection.

Follow @iron.coding

The following code demonstrates this:

```

import pandas as pd
import numpy as np
df1 = pd.Series([2, 4, 5, 8, 10])
df2 = pd.Series([8, 10, 13, 15, 17])
p_union = pd.Series(np.union1d(df1, df2)) # union of series
p_intersect = pd.Series(np.intersect1d(df1, df2)) # intersection of series
unique_elements = p_union[~p_union.isin(p_intersect)]
print(unique_elements)
"""
Output:
0      2
1      4
2      5
5     13
6     15
7     17
dtype: int64
"""

```

11. While importing data from different sources, can the pandas library recognize dates?

Follow @iron.coding

Yes, they can, but with some bit of help. We need to add the `parse_dates` argument while we are reading data from the sources. Consider an example where we read data from a CSV file, we may encounter different date-time formats that are not readable by the pandas library. In this case, pandas provide flexibility to build our custom date parser with the help of lambda functions as shown below:

```

import pandas as pd
from datetime import datetime
dateparser = lambda date_val: datetime.strptime(date_val, '%Y-%m-%d %H:%M:%S')
df = pd.read_csv("some_file.csv", parse_dates=['datetime_column'],
date_parser=dateparser)

```

Types of Python Framework

Python has three categories of frameworks and those are full-stack framework, micro-framework, and asynchronous framework. Now, let's understand what each category offers:

1. **Full Stack Framework:** As the name suggests this kind of framework provides a complete solution for web development like form generator, form validation, template layouts, etc. This type of framework can be utilized for any type of application. It is a little bit complex to use.
2. **Micro Framework:** It's a lightweight, easy-to-use framework and doesn't provide any extra features like a data abstraction layer, form validation, etc. Developers have to put in lot of effort into adding code manually to get additional features and functionalities. This is useful for small applications.
3. **Asynchronous Framework:** This framework is gaining popularity in recent times and it uses asyncio library to work. This kind of framework mainly helps in running concurrent connections in huge amounts.

Why Use A Framework?

Frameworks are widely used by developers to reuse the code for similar types of HTTP operations and to define the structure of an application with predefined functionalities. Frameworks make the life of a programmer easier as these frameworks design the project in such a manner that any other programmer who knows the framework can easily take over the application. Although there is an option of using libraries, people prefer to use frameworks as it is more idiomatic, and reliable and it's easy to extend the functionalities using the tools provided by the frameworks. Using a library can be as challenging as performing any specific operation one has to learn the functionalities to perform that operation. There is no particular flow in libraries but in the framework, there is a basic flow and then programmers are in better control of the application.

There are various frameworks of Python like:

- Bottle
- Flask
- Django
- Web2py
- AIOHTTP
- CherryPy
- Dash

Falcon
Growler
UvLoop
Pyramid
Sanic
CubicWeb
TurboGears
Hug
MorePath

These are some of the frameworks used in Python, we will be discussing each in the coming topic.

Top Python Frameworks List

Now we will look at some top Python frameworks in detail:

1. Bottle

This framework is ideal for small applications and is mainly used for building APIs. It is one of the most used Python web frameworks as it doesn't need any other dependency apart from the standard Python library to make the application, programmers simply can work with hardware. This framework creates a single source file and it comes under the micro-framework category.

Key Features of Bottle Framework are listed below:

- Using this framework one gets access to form data, cookies, file upload, and other HTTP-related metadata.
- The Request-dispatching route is another key highlight of this framework.
- It offers a built-in HTTP server
- Has plugin support for various databases.
- Third-party template engines and WSGI/HTTP servers can also be used.

2. Django

It comes under the category of full-stack framework, which has gained popularity in recent times and is considered as one of the top python web frameworks. It follows the principle of Don't Repeat Yourself(DRY).

Django has many built-in libraries and it gives the aid to migrate from one database to another. By default Django can work on these few databases: MySQL, Oracle,

PostgreSQL, and SQLite, the rest databases can be used with the help of third-party drivers. For mapping objects to database tables, it uses ORM.

Key Features are:

- Security is the most important point of this framework. When compared with other python frameworks Django proves to be more secure.
- It provides URL routing
- It provides authentication support
- Django offers a database schema migration feature.
- It provides adequate prebuilt libraries for full stack development.
- Django follows MVC-MVT architecture. In this architecture, the developer just has to give a model, view, and template then the user does the mapping of it to the URL and then the rest of the tasks are handled by Django itself.

3. Web2Py

It is a framework of the full stack category. It is an open-source and scalable framework that provides support to all operating systems. Web2Py has its web-based IDE that has all the features that an IDE should have like a debugger, a code editor, and one-click deployment. It cannot use python 3.

Key Features:

Follow @iron.coding

- The ticket system is there in which a ticket is sent to the user if something goes wrong with the framework.
- It is platform-independent.
- Backward compatibility is there which makes sure that there is advancement without breaking the ties with previous versions.
- Readability of multiple protocols
- Provides role-based access control
- There is no prerequisite for its installation or configuration.
- Give support for internalization

4. Flask

Another lightweight and micro-framework that is popular is a flask. Due to its modular design, it is more easily adaptable. Using this framework developers can make solid web applications and the creation of such applications makes it easy to use any type of extension.

Key Features:

- Compatibility with Google App Engine
- Aids jinja2 template and Werkzeug WSGI toolkit

- Inbuilt debugger
- Provision for unit testing
- For enabling client-side sessions Flask provides support for cookies
- Restful request dispatching
- Another key highlight of this framework is it is Unicode based
- HTTP request handling is also supported
- This framework also provides the option of plugging in any ORM

5. CherryPy

One of the oldest microframeworks is CherryPy. It has a minimalistic approach. CherryPy is an open-source and object-oriented framework. One can use any technology for accessing data or template creation. Applications created using this framework are stand-alone python applications that have a multithreaded server embedded in them.

Below are some of the key features of CherryPy:

- One can run multiple servers simultaneously using this framework
- Platform independent.
- Coverage, profiling, testing are some other built-in features supported by the framework.
- It runs on Android.
- It provides a good configuration system
- HTTP WSGI compliant thread pooled web server is there.
- Caching, encoding, and authentication are some more features provided by the framework.

6. Aiohttp

It's a kind of asynchronous framework. This framework can serve as a client framework also apart from being the server web framework. It is based on Python 3.5+ features like async and await. Python's asyncio library has the main role in the framework's functioning. aiohttp makes use of request objects and routers for redirection of queries.

Key features of the framework are listed below:

- Building views is easy through this framework
- It provides middleware support
- Pluggable routing and middleware support are other features that make it a top-using framework.
- It provides support for both server and client-based Web Sockets.

7. CubicWeb

It is a full-stack Python framework that makes use of cubes instead of using separate models and views. It is an open-source, free-to-use, and semantic web framework.

Key Features of the framework are:

- It provides support for multiple databases
- Provides security and reusable components
- Uses RQL(relational query language) for simplifying data related queries
- Provides support for Web Ontology Language (OWL) and Resource Description Framework(RDF)

8. Dash

It is an open-source micro-framework used for developing analytical web applications. This framework is more popular among data scientists who are not much aware of web development. For frontend rendering it uses ReactJS. Applications built using dash can also be used to run web servers like flask and then do the communication with JSON packets through HTTP requests. Since dash applications can be rendered in the browser and deployed in the server it is said to be cross-platform and mobile-ready applications.

Key Features of the framework are:

- Doesn't require much coding to develop applications
- A high level of customization is offered
- Error handling is easy
- LDAP integration (Dash Deployment Server)
- Plugin support is also there

9. Falcon

It is another micro framework that focuses on building web APIs. This framework is widely used by organizations like LinkedIn, OpenStack, RackSpace. It allows developers to make clear designs for HTTP and REST architectures.

Key Features:

- Aims at having 100% code coverage
- Upfront exception handling is supported by this framework
- It is a very extensible and optimized codebase
- Easy access for headers and bodies is offered by its request and response classes
- WSGI helpers and mocks are used for unit testing
- Cython support increases the speed of the framework

10. Giotto

It is a full-stack MVC-based framework that separates model, view, and controller in order so that developers and system admins can work independently. Giotto enables users to build apps on top of the web, IRC(Internet Relay Chat), and command-line by including a controller module.

Key Features:

- Provides feature of automatic URL routing
- Provides characteristic of Jinja2 for HTML templates
- Functional CRUD patterns
- Generic model and view
- Multiple pluggable controllers are there
- Database persistence with SQLAlchemy

11. Growler:

It is an asynchronous type of framework built on the python asyncio library. It is based on NodeJS and an express/connect framework. This framework handles the request by passing it through middleware technology.

Key Features:

Follow @iron.coding

- For writing clean and reusable code decorators are used.
- The entire application can be zipped into a single execution file by using the ziapp module.
- Also provides support for the multitude of the open-source package

Conclusion

In this blog, we have discussed different types of Python frameworks that are beneficial for building web applications. Each framework is discussed with its key features and these frameworks can easily meet the requirements of one's project and business. The choice of the framework is totally on the developer according to their project needs.

FAQs

Q.1: What is the Python Framework?

Ans: Python framework is a collection of modules or packages that are useful for creating web applications. The framework reduces the development time by

providing prebuilt implementation of redundant tasks. Hence frameworks make the life of a developer easy.

Q.2: Which Framework is best for Python for Beginners?

Ans: For beginners, Flask and Django is a better option to start with but Flask can be more beneficial as it is very simple and easy to learn and requires fewer lines of code as compared to Django. Using this framework web development becomes easier.

Q.3: How do you create a Framework?

Ans: For developing a framework one must be aware of what it consists of, a framework mainly consists of URL routing, views, models, and templates. First of all, try to learn about them and then create each section and then connect them. Once you get all the components in a running state try to build a simple project with it and test it. After the framework is complete, package it up and upload it to PyPI.

Q.4: Is Django the best Python Framework?

Ans: Django can be considered the best framework as its development speed is fast and easy, from a technical perspective it contains almost every feature, and the rest can be made available using third-party drivers. Also, this framework is a good option for database-based websites.

Q.5: Is Django better than Flask?

Ans: Both Django and Flask are very popular frameworks. If you are starting your work on the web then Flask is a better option since in this framework developers have to do everything on their own while Django offers many extraordinary features and one can learn new concepts using this framework very easily. So, one can go with any of the frameworks but to start with it's better to go with Flask and then Django as its learning curve is a little complicated.