InterviewBit
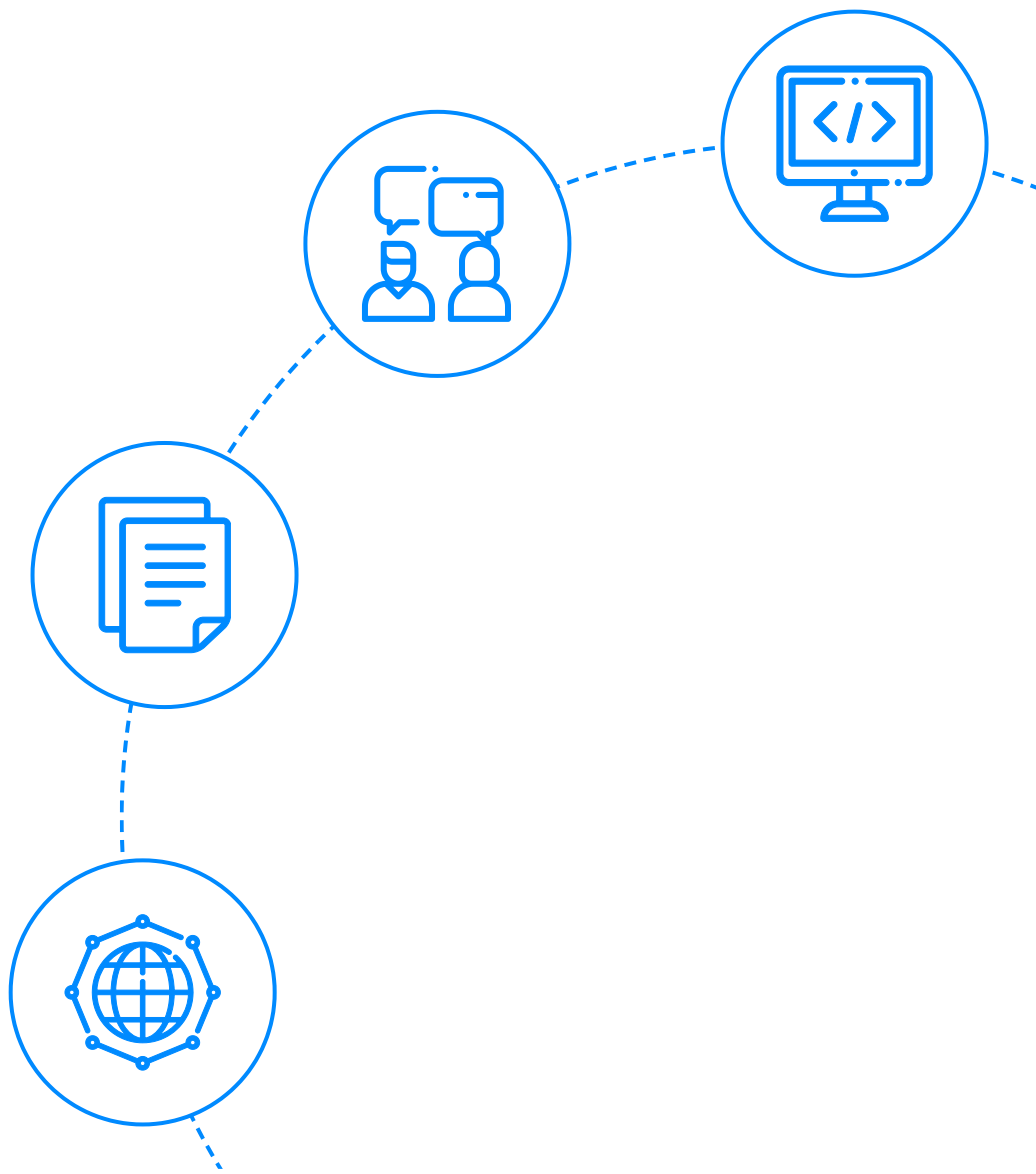
Python Cheat Sheet

To view the live version of the page, click here.

Contents

Learn Python: Basic to Advanced Concepts

Learn Python: Basic to Advanced Concepts

(.....Continued)

Let's get Started

Introduction: What is Python?

Python is a High-Level Programming Language, with high-level inbuilt data structures and dynamic binding. It is interpreted and an object-oriented programming language. Python distinguishes itself from other programming languages in its easy to write and understand syntax, which makes it charming to both beginners and experienced folks alike. The extensive applicability and library support of Python allow highly versatile and scalable software and products to be built on top of it in the real world.

The Zen of Python

The Zen of Python is basically a list of Python Aphorishm's written down in a poetic manner, to best showcase the good programming practices in Python.

> The Zen of Python, by Tim Peters
>
> 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!

Learn Python: Basic to Advanced Concepts

1. Python Arithmetic Operators

The Arithmetic Operators in the below table are in Lowest to Highest precedence.

Operators	Operation	Explanation	Examples
+	Addition	Returns sum of 2 numbers	1 + 3 = 4
-	Subtraction	Returns the difference of 2 numbers	1 - 3 = -2
*	Multiplication	Returns the product of 2 numbers	1 * 3 = 3
/	Division	Returns the value of a divided by b as a decimal value	1 / 3 = 0.33
//	Floored Division	Returns the floor of a divided by b	1 // 3 = 0
%	Remainder	Returns the remainder when a is divided by b	1 % 3 = 1

Some examples are shown below:

```
#Example for Addition
>>> 1 + 3
4
#Example for Subtraction
>>> 1 - 3
-2
#Example for Multiplication
>>> 6 * 6
36
#Example for Floored Division
>>> 4 // 2
2
#Example for Division
>>> 3 / 2
1.5000
#Example for Modulo
>>> 3 % 2
1
```

2. Python Data Types

The table below lists the different **data types in Python** along with some examples of them:

DataTypes	Examples
Integers	0, 2, -1, 5
Strings	"a", "hello", "1234", "12 Hello."
Boolean	True, False
Floating Point Numbers	16.0, -11.0, 2021.5

3. Python Variables

Variables are names given to data items that may take on one or more values during a program's runtime.

Following are the variable naming conventions in python:

- It cannot begin with a number.
- It must be a single word.
- It must consist of letters and _ symbols only.
- Variables in Python which start with _ (underscore) are considered as "Unuseful".

Some examples are shown below:

```
>>> variable_name = "Hello"
>>> variable_name
'Hello'
>>> variableName = 123
>>> variableName
123
```

4. Python Comments

Comments are lines of text/code in the program, which are ignored by the compiler during program execution.

There are multiple types of comments in python:

- Inline Comment -

We can write an Inline Comment by typing # followed by the comment.

```
# Inline Comment to calculate sum of 2 numbers
def fun(a, b):
    return a + b
```

- Multiline Comment -

We can write a Multiline Comment by typing # followed by the comment in each of the lines.

```
# Multiline
# Comment
# Function to calculate
# sum of 2 numbers
def fun(a, b):
    return a + b
```

- Docstring Comment -

Docstring comments are achieved by Typing the comment within triple quotes. (''' comment ''')

```
'''
This is a function
to find sum
of 2 numbers.
This is an example of
docstring comment.
'''
def fun(a, b):
    return a + b
```

5. Standard Python Functions

- **print() function in Python**

The print() function prints some specified message to the screen or some standard output device. We can print strings, numbers, or any other object using this function. We can print multiple tokens, and also specify to print the data separated by different delimiters using the print() function.

```
>>> print("Hello World")
Hello World
>>> var = "Interviewbit"
>>> print("My name is ", var)
('My name is ', 'Interviewbit')
>>> print("My name is " + var)
My name is Interviewbit
>>> print(123)
123
>>> a = [1, 2, 3, 4]
>>> print(a)
[1, 2, 3, 4]
```

- **input() function in Python**

The input() function in Python is used to take any form of inputs from the user/standard input device, which can later be processed accordingly in the program. It is complementary to the print() function.

```
>>> print('Enter your name.')
>>> myName = input()
>>> print('Hello, {}'.format(myName))
Enter your name.
Interviewbit
Hello, Interviewbit
```

- **len() function in Python**

The len() function is used find the length(number of elements) of any python container like string, list, dictionary, tuple, etc.

```
# For List
a = [1, 2, 3]
print(len(a))
# For string
a = "hello"
print(len(a))
# For tuple
a = ('1', '2', '3')
print(len(a))
```

- **ord() function in Python**

The ord() function in Python will return an integer that represents the Unicode Character passed into it. It takes a single character as a parameter.

Example:

```
# Print unicode of 'A'
print(ord('A'))
# Print unicode of '5'
print(ord('5'))
# Print unicode of '$'
print(ord('$'))
Output:
65
53
36
```

6. Python Type Casting

Type casting is basically a conversion of variables of a particular datatype into some other datatype, such that the conversion is valid.

Type casting can be of two types:
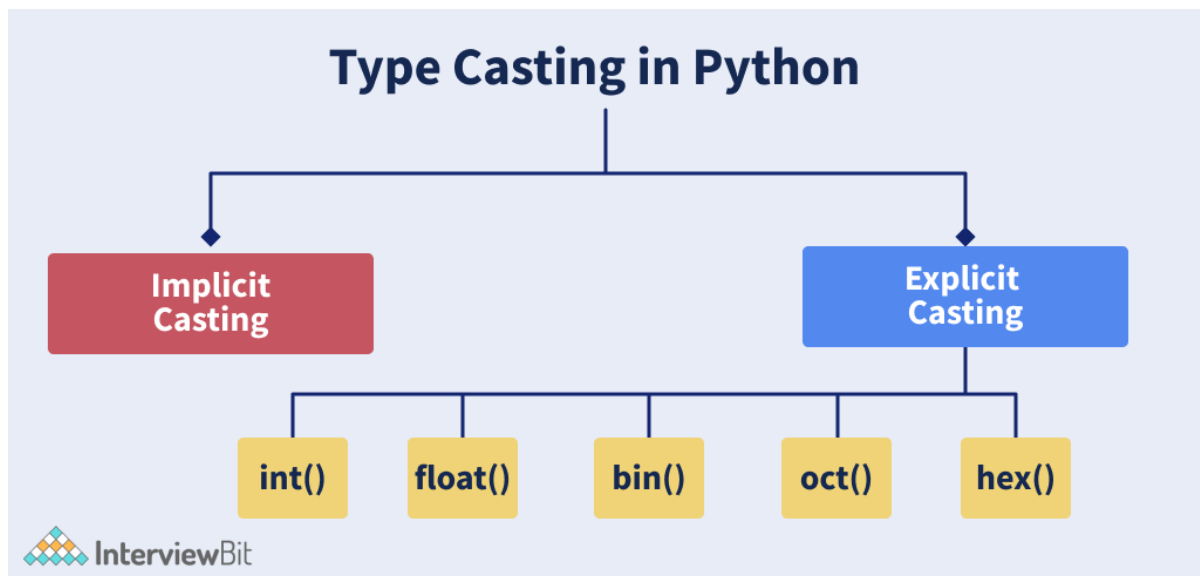
- **Implicit Type Casting:** In implicit type casting, the python compiler internally typecasts one variable into another type without the external action of the user. Example:

```
int_num = 100
float_num = 1.01
ans = int_num + float_num
print(type(int_num))
print(type(float_num))
# ans is implicitly typecasted to float type for greater precision
print(type(ans))
```

- **Explicit Type Casting:** In explicit type casting, the user explicitly forces the compiler to convert a variable from one type to another. The different ways of explicit typecasting are given below:

1. Integer to String or Float:

To typecast an integer into a string type, we use the str() method. Similarly, to typecast it into a float type, we use the float() method.

For example:

```
>>> var = 123
>>> str(var)
'123'
>>> var = 123
>>> float(var)
123.0
```

2. Float to Integer:

To typecast a float datatype into an integer datatype, we use the int() method.

For example:

```
>>> var = 7.8
>>> print(int(var))
7
```

7. Program Flow Control in Python

Relational Operators in Python

The Table gives a list of relational operators available in Python along with their functions:

Operator	What it does
==	Is equal to
>=	Is Greater than or Equal to
<=	Is Less than or Equal to
>	Is Greater than
<	Is Less than
!=	Not Equal to

Some examples are given below:

```
# Equality Operator
>>> 10 == 10
True      # 10 is equal to 10, so true
>>> 10 == "10"
False     # The first string is of type int, 2nd of type string, so false.
# Greater than
>>> 10 > 20
False     # 10 is lesser than 20, so above expression is false.
# Inequality
>>> 10 != 20
True      # 10 is not equal to 20, so the expression is true
# Greater than or equal to
>>> (2 + 3) >= (4 + 1)
True      # (2 + 3) = 5 and (4 + 1) = 5, so the expression is true.
```

Note: Never use relational operators to compare boolean operations. Use is or is not operators for it.

```
>>> True is False
False
>>> True is not False
True
```

8. Boolean Operators in Python

The Table gives a list of boolean operators available in Python along with their functions:

Operator	What it does
and	Returns True if both operands are True, else False
or	Returns True if both operands are True, else False
not	Returns value opposite to the Truth value of the expression

Examples:

```
# and operator
print(True and False)
False
# or operator
print(True or False)
True
# not operator
print(not False)
True
```

9. Conditional Statements in Python

- **If Statements:** If statement is a condition statement that will perform some operation, if the expression given in it evaluates to true as shown below:
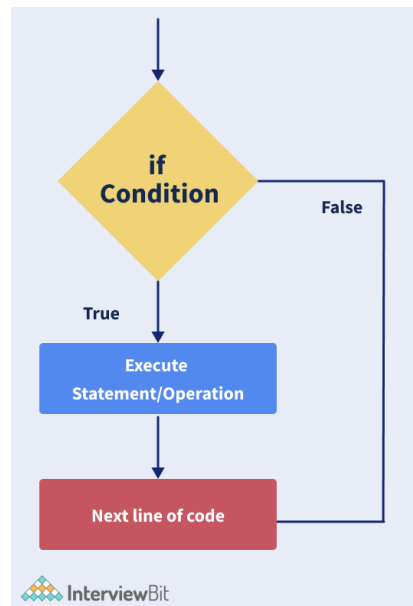
```
>>> var = "Good"
>>> if var == "Good":
...     print("Same")
...
Same
```

- **Elif Statements:** This statement is used in conjunction with the if statement to add some other condition which is evaluated if the condition in if statement fails.

```
>>> var = "Good"
>>> if var == "Good":
...     print("Same")
... elif var != "Good":
...     print("Not Same")
...
Same
```

- **Else Statements:** This statement is used to perform some operation, if all the if and elif statements evaluates to be false.

```
>>> var = "Good"
>>> if var != "Good":
...     print("Not Same")
... else:
...     print("Same")
...
Same
```

The final if-elif-else ladder looks like shown below:
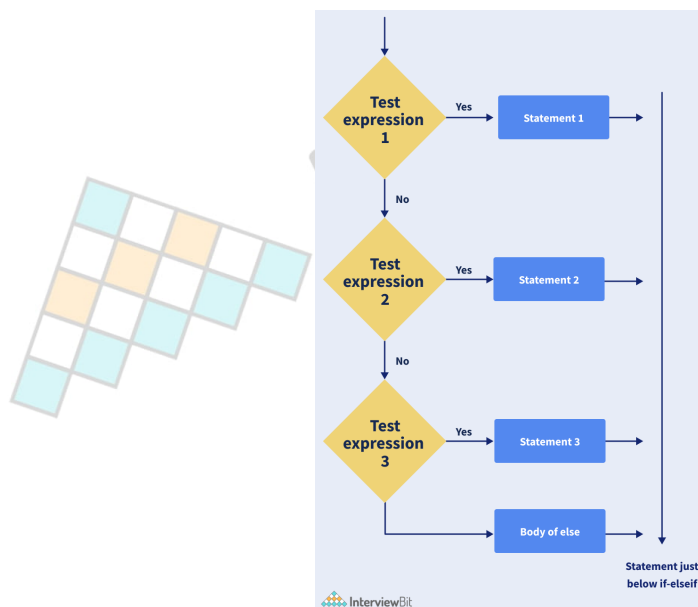
10. Loop Statements in Python

Loops in Python are statements that allow us to perform a certain operation multiple times unless some condition is met.

For Loops: For loop is used to iterate iterables like string, tuple, list, etc and perform some operation as shown in the flowchart below:
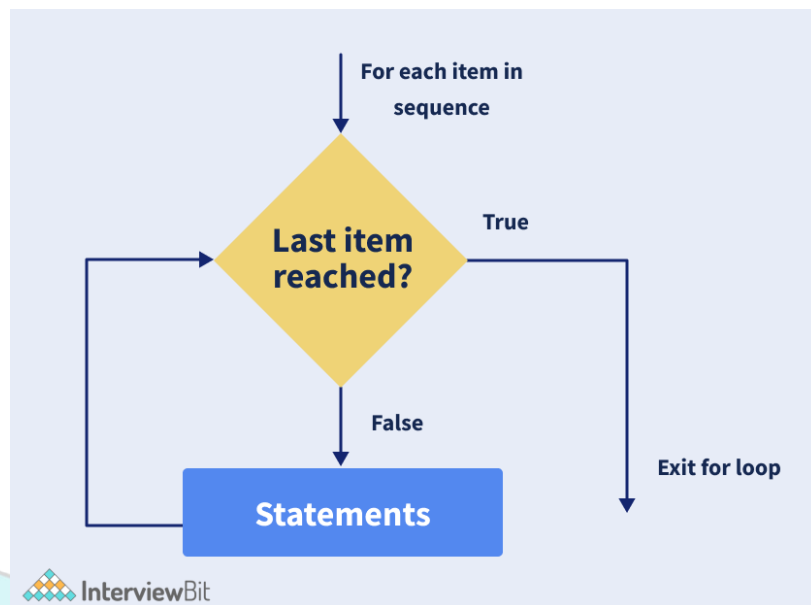
- **For with range:**
  This loop format will iterate overall numbers from 0 to Limit - 1.
  The below example prints numbers from 0 to 4.

```python
for i in range(5):
print(i)
Output:
0
1
2
3
4
```

- **For with range(start, stop, step):**

  This will run the loop from start to stop - 1, with step size = step in each iteration.
  In the below example, the start is 2, end point is 10 and the step size is 2. Hence it prints 2,4,6,8

```
for i in range(2, 10, 2):
print(i)
Output:
2
4
6
8
```

- **For with in:**

  This is used to iterate over all the elements in a python container like list, tuple, dictionary, etc.

```
a = [1, 3, 5, 7]
for ele in a:
print(ele)
Output:
1
3
5
7
```

- **While Loops:**

  This is used for executing set of statements within its block as long as the associated loop condition is evaluated to True as shown in the image below:
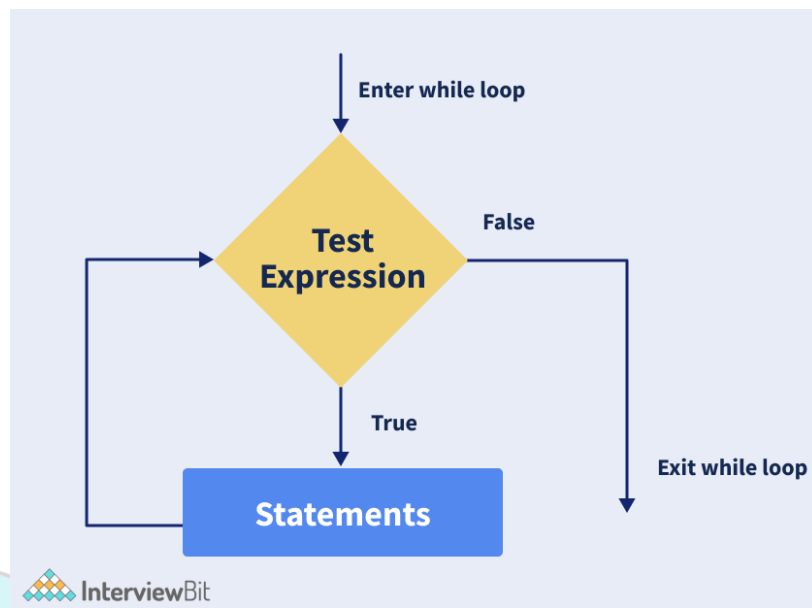
```
>>> count = 5
>>> while count > 0:
...     print(count)
...     count -= 1
...
5
4
3
2
1
```

11. Jump Statements in Python

- **break:** break statements are used to break out of the current loop, and allow execution of the next statement after it as shown in the flowchart below:
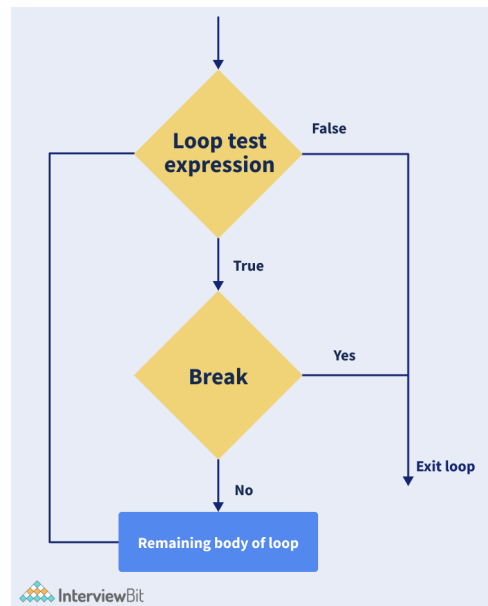
```
>>> for i in range(5):
...     print(i)
...     if i == 3:
...             break
...
0
1
2
3
```

- **continue:** continue statement allows us to send the control back to the starting of the loop, skipping all the lines of code below it in the loop. This is explained in the flowchart below:
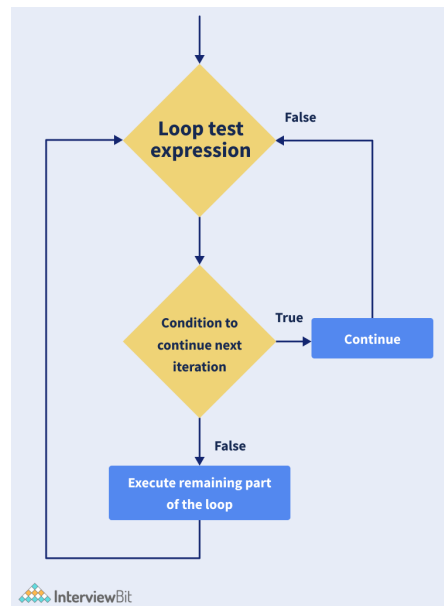
```
>>> for i in range(5):
...     if i == 3:
...         continue
...     print(i)
...
0
1
2
4
```

- **pass:** The pass statement is basically a null statement, which is generally used as a placeholder. It is used to prevent any code from executing in its scope.

```
for i in range(5):
    if i % 2 == 0:
        pass
    else:
        print(i)
Output:
1
3
```

- **return:** return statement allows us to send the control of the program outside the function we are currently in. A function can have multiple return statements, but can encounter only one of them during the course of its execution.

```python
def func(x):
    if x == 'Hello':
        return True
    else:
        return False
```

12.   Functions in Python

Functions are used to well-organized our code and enhance code readability and reusability. In Python, a function is defined using the **def** keyword. A function can return some value, or not depending upon its use case. If it has to return a value, the return statement (which has been discussed) is used. The syntax of a python function is shown in the image below:
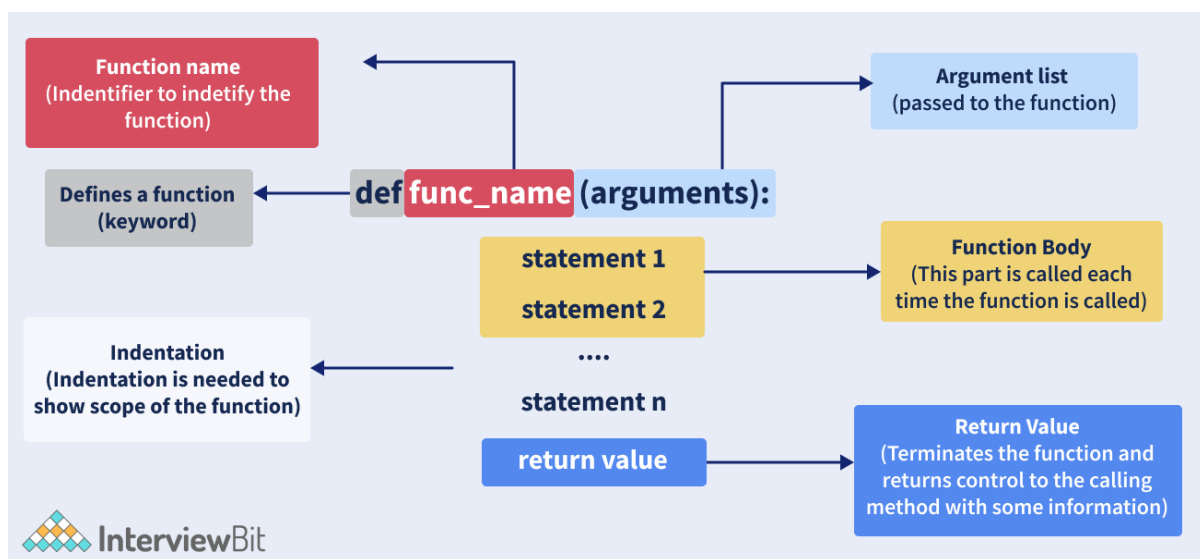
Example of a function:

```
# Function to return sum of two numbers
def getSum(a, b):
    return a + b

# Function to print sum of 2 numbers
def printSum(a, b):
    print(a + b)

print(getSum(5, 6))
printSum(5, 6)
```

13. Python Variable Scope Resolution

Scope of a variable is the part of the code, where it can be accessed freely and used by the program.
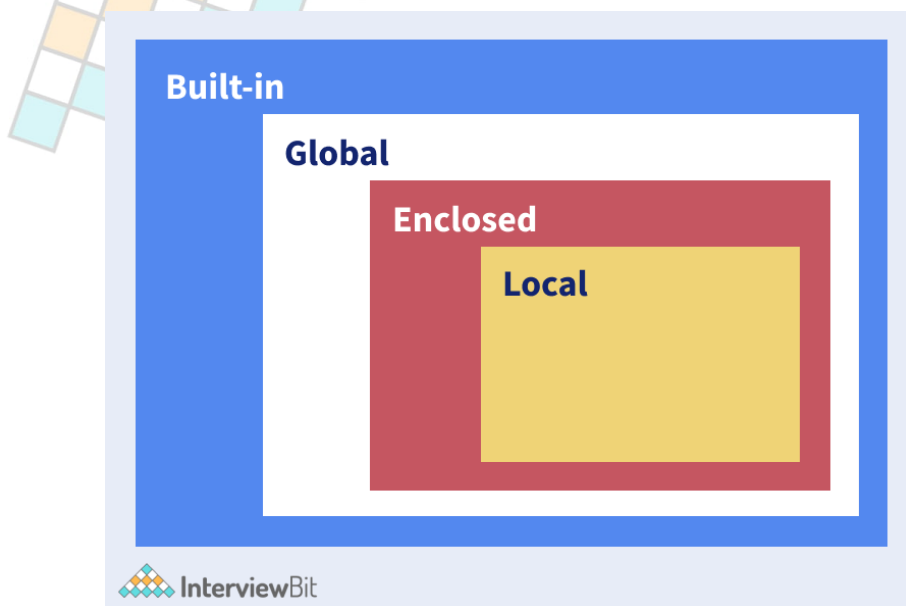
The scopes in the above image are explained as follows:

- **Built-in:** These are reserved names for Python built-in modules.
- **Global:** These variables are defined at the highest level.
- **Enclosed:** These variables are defined inside some enclosing functions.
- **Local:** These variables are defined inside the functions or class and are local to them.

The rules used in Python to resolve scope for local and global variables are as follows:

- Code in the global scope cannot use any local variables.
- Code in a function's local scope cannot use variables in any other local scope.
- However, a local scope can access global variables.
- We can use the same name for different variables if they are in different scopes.

14. Global Statement

To modify a global variable from inside a function, we use the global statement:

```python
def func():
    global value
    value = "Local"

value = "Global"
func()
print(value)
Output:
Local
```

We set the value of "value" as Global. To change its value from inside the function, we use the global keyword along with "value" to change its value to local, and then print it.

15. Importing Modules in Python

Python has various external libraries of code with useful utilities and functions. To use these modules, we need to import them into our code, using the import keyword.

For example, if we want to use the functionalities of the math module, then we can import it in our python code by using import math as shown in the example below.

```python
import math
print(math.pi)
Output:
3.141592653589793
```

If we want to perform any string manipulations, we can use the string module as import string in python. More of this is covered in the String Manipulation section below.

16. Exception Handling in Python

Exception Handling is used to handle situations in our program flow, which can inevitably crash our program and hamper its normal working. It is done in Python using try-except-finally keywords.

- **try:** The code in try section is the part of the code where the code is to be tested for exceptions.
- **except:** Here, the cases in the original code, which can break the code, are written, and what to do in that scenario by the program.
- **finally:** The code in the finally block will execute whether or not an exception has been encountered by the program.

The same has been illustrated in the image below:
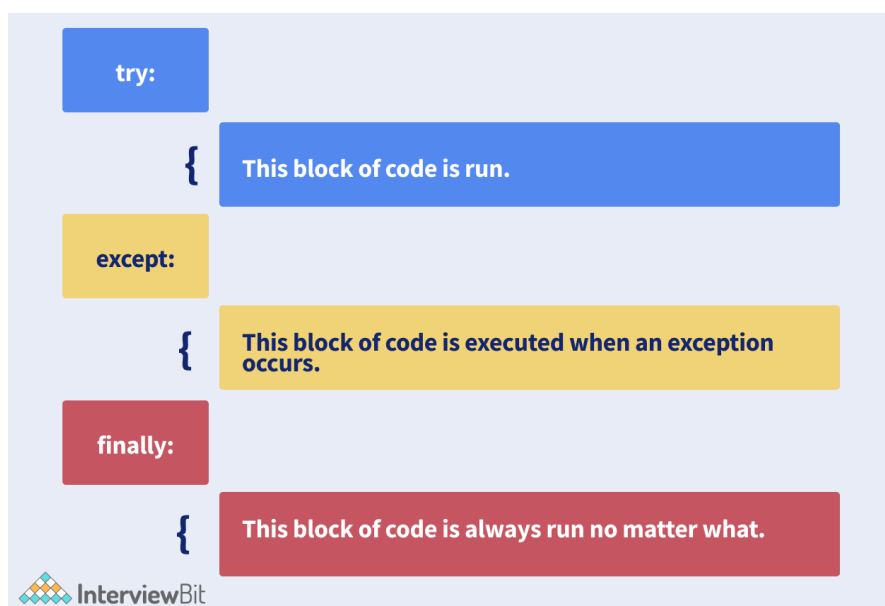
Example:

```
# divide(4, 2) will return 2 and print Division Complete
# divide(4, 0) will print error and Division Complete
# Finally block will be executed in all cases
def divide(a, denominator):
    try:
        return a / denominator
    except ZeroDivisionError as e:
        print('Divide By Zero!! Terminate!!')
    finally:
        print('Division Complete.')
```

17. Lists in Python

Lists are used to store multiple items in a single variable. Their usage and some functions are shown below with examples:

```
example = ["Sunday", "Monday", "Tuesday", "Wednesday"];
print(example)
Output:
['Sunday', 'Monday', 'Tuesday', 'Wednesday']
```

- **Accessing elements in a List:**
  Accessing elements in a list basically means getting the value of an element at some arbitrary index in the list.
  Indexes are assigned on 0 based basis in python. We can also access elements in python with negative indexes. Negative indexes represent elements, counted from the back (end) of the list.

```
example = ["Sunday", "Monday", "Tuesday", "Wednesday"];
# Positive Indexing
print(example[0], example[1])
# Negative Indexing
print(example[-1])
Output:
Sunday Monday
Wednesday
```

- **Slicing a List:**
  Slicing is the process of accessing a part or subset of a given list. The slicing is explained in the image below:

example [0:2] = [Sunday, Monday]

example [-3:-1] = [Monday, Tuesday]
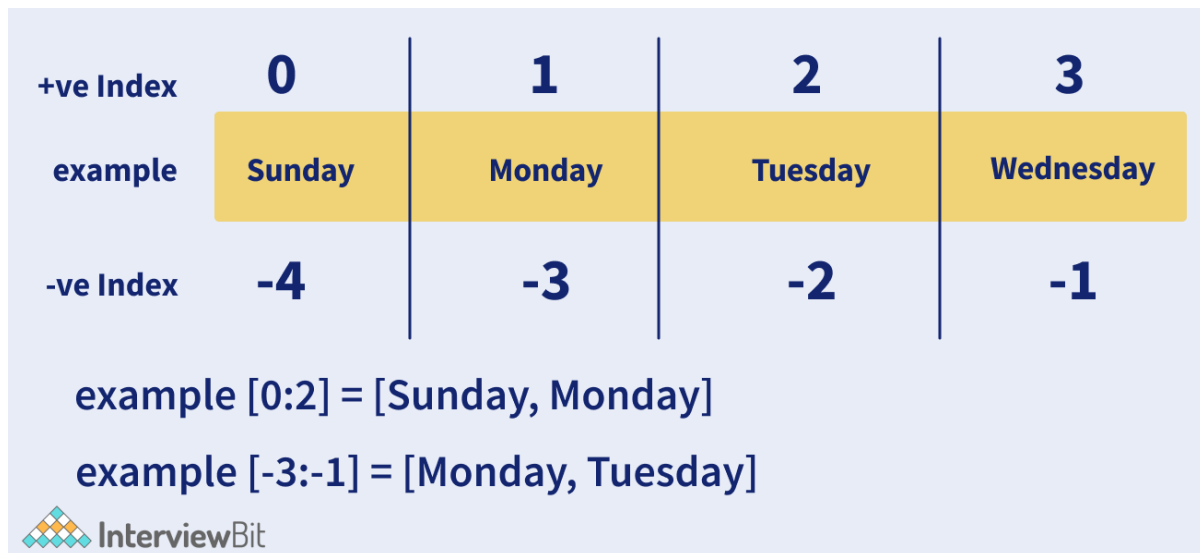
```
example = ["Sunday", "Monday", "Tuesday", "Wednesday"];
# Positive Slicing
print(example[0:2])
# Negative Slicing
print(example[-3:-1])
Output:
['Sunday', 'Monday']
['Monday', 'Tuesday']
```

- **Changing Values in a List:**
  We can change values at some particular index in a list by accessing the element with [] and then setting it to some other value.

```
example = ["Sunday", "Monday", "Tuesday", "Wednesday"];
print(example)
example[0] = "Saturday"
print(example)
Output:
['Sunday', 'Monday', 'Tuesday', 'Wednesday']
['Saturday', 'Monday', 'Tuesday', 'Wednesday']
```

- **List Concatenation and Replication:**

When we merge the contents of 2 lists into one list, it is called list concatenation.

```
example = ["Sunday", "Monday", "Tuesday", "Wednesday"];
example1 = ["Weekdays", "Weekends"]
# Concatenation
example = example + example1
print(example)
Output:
['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Weekdays', 'Weekends']
```

Copying the contents of a list, some finite number of times into the same or some list is called list replication.

```
example = ["Sunday", "Monday", "Tuesday", "Wednesday"];
example1 = ["Weekdays", "Weekends"]
# Replication
example1 = example1 * 3
print(example1)
Output:
['Weekdays', 'Weekends', 'Weekdays', 'Weekends', 'Weekdays', 'Weekends']
```

- **Delete values from Lists:**
  We can delete a particular element from a list by using the del keyword.

```
example = ["Sunday", "Monday", "Tuesday", "Wednesday"];
print(example)
del example[2]
print(example)
Output:
['Sunday', 'Monday', 'Tuesday', 'Wednesday']
['Sunday', 'Monday', 'Wednesday']
```

- **Looping through Lists:**
  The below example shows how we can iterate over all the elements present in a list.

```
example = ["Sunday", "Monday", "Tuesday", "Wednesday"];
for ex in example:
    print(ex)


Output:
Sunday
Monday
Tuesday
Wednesday
```

in and not in keywords:

With the **in** keyword, we can check if some particular element is present in the given python variable.

Similar to the not in keyword, we can check if some particular element is not present in the given python variable.

```
example = ["Sunday", "Monday", "Tuesday", "Wednesday"];
print("Sunday" in example)
print("Hello" not in example)
Output:
True
True
```

- **Adding Values in Lists:**

insert(): This function inserts an element into a particular index of a list.

```
example = ["Sunday", "Monday", "Tuesday", "Wednesday"];
print(example)
example.insert(1, 'Days')
print(example)
Output:
['Sunday', 'Monday', 'Tuesday', 'Wednesday']
['Sunday', 'Days', 'Monday', 'Tuesday', 'Wednesday']
```

append(): This function appends an element at the back of a list.

```
example = ["Sunday", "Monday", "Tuesday", "Wednesday"];
print(example)
example.append('Days')
print(example)
Output:
['Sunday', 'Monday', 'Tuesday', 'Wednesday']
['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Days']
```

- **Sorting a List:**
  Sorting a list means arranging the elements of the list in some particular order. We sort a list by using the **sort()** function.

```
# Sorts in lexicographical order
example = ["Sunday", "Monday", "Tuesday", "Wednesday"];
print(example)
# Sort in ascending order
example.sort()
print(example)
# Sort in descending order
example.sort(reverse = True)
print(example)

example = [1, 5, 3, 7, 2]
# Sort in ascending order
example.sort()
print(example)
# Sort in descending order
example.sort(reverse = True)
print(example)
Output:
['Sunday', 'Monday', 'Tuesday', 'Wednesday']
['Monday', 'Sunday', 'Tuesday', 'Wednesday']
['Wednesday', 'Tuesday', 'Sunday', 'Monday']
[1, 2, 3, 5, 7]
[7, 5, 3, 2, 1]
```

18. Tuples in Python

Tuples are entities in Python that work almost similar to that of lists, but differ in the main feature from lists, is in that they are **inmutable**.

They are initialized by writing the elements of the tuple with (), separated by commas.

```
# Defining and Initializing a tuple called example
example = ("First", "Second", "Third", "Fourth")
print(example)
print(example[1:3])
Output:
('First', 'Second', 'Third', 'Fourth')
('Second', 'Third')
```

Type Converting between Tuples, Lists, and Strings:

```
# Convert list to a tuple
tuple(['first', 'second', 'third'])
# Convert tuple to a list
list(('first', 'second', 'third'))
# Convert string to a list
list("Scaler")
```

19. Python Dictionaries

Dictionaries in Python are equivalent to Maps in C++/JAVA. They are used to store data in key-value pairs.

Printing key and values in dictionaries:

To print the keys of the dictionary, use the .keys() method and to print the values, use .values() method.

```
dict = {'first' : 'sunday', 'second' : 'monday', 'third' : 'tuesday'}
# dict.keys() method will print only the keys of the dictionary
for key in dict.keys():
    print(key)
# dict.values() method will print only the values of the corresponding keys of the dic
for value in dict.values():
    print(value)

Output:
first
second
third
sunday
monday
tuesday
```

Update key value in dictionary:

- ***Update key value which is not present in dictionary:***
  We can update a key value in a dictionary by accessing the key withing [] and setting it to a value.

```python
dict = {'first' : 'sunday', 'second' : 'monday', 'third' : 'tuesday'}
for item in dict.items():
      print(item)
dict['fourth'] = 'wednesday'
for item in dict.items():
      print(item)
Output:
 ('first', 'sunday')
 ('second', 'monday')
 ('third', 'tuesday')
 ('first', 'sunday')
 ('second', 'monday')
 ('third', 'tuesday')
 ('fourth', 'wednesday')
```

- ***Update key value which is present in the dictionary:***
  We can update a key value in a dictionary, when the key is present in the exact same way as we update a key, when the key is not present in the dictionary.

```python
dict = {'first' : 'sunday', 'second' : 'monday', 'third' : 'tuesday'}
for item in dict.items():
      print(item)
dict['third'] = 'wednesday'
for item in dict.items():
      print(item)
Output:
 ('first', 'sunday')
 ('second', 'monday')
 ('third', 'tuesday')
 ('first', 'sunday')
 ('second', 'monday')
 ('third', 'wednesday')
```

- ***Delete key-value pair from dictionary:***
  We can delete a key-value pair from a dictionary using the del keyword followed by the key value to be deleted enclosed in [].

```
dict = {'first' : 'sunday', 'second' : 'monday', 'third' : 'tuesday'}
for item in dict.items():
        print(item)
del dict['third']
for item in dict.items():
        print(item)
Output:
 ('first', 'sunday')
 ('second', 'monday')
 ('third', 'tuesday')
 ('first', 'sunday')
 ('second', 'monday')
```

Merging 2 dictionaries

We can merge 2 dictionaries into 1 by using the update() method.

```
dict1 = {'first' : 'sunday', 'second' : 'monday', 'third' : 'tuesday'}
dict2 = {1: 3, 2: 4, 3: 5}
dict1.update(dict2)
print(dict1)
Output:
{'first': 'sunday', 'second': 'monday', 'third': 'tuesday', 1: 3, 2: 4, 3: 5}
```

20. Sets in Python

Initializing Sets:

Sets are initialized using curly braces {} or set() in python.

A python set is basically an unordered collection of unique values, i.e. it will automatically remove duplicate values from the set.

```
s = {1, 2, 3}
print(s)
s = set([1, 2, 3])
print(s)
s = {1, 2, 3, 3, 2, 4, 5, 5}
print(s)
Output:
{1, 2, 3}
{1, 2, 3}
{1, 2, 3, 4, 5}
```

Inserting elements in set:

We can insert a single element into a set using the add function of sets.

```
s = {1, 2, 3, 3, 2, 4, 5, 5}
print(s)
# Insert single element
s.add(6)
print(s)
Output:
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5, 6}
```

To insert multiple elements into a set, we use the update function and pass a list of elements to be inserted as parameters.

```
s = {1, 2, 3, 3, 2, 4, 5, 5}
# Insert multiple elements
s.update([6, 7, 8])
print(s)
Output:
{1, 2, 3, 4, 5, 6, 7, 8}
```

Deleting elements from the set:

We can delete elements from a set using either the remove() or the discard() function.

```
s = {1, 2, 3, 3, 2, 4, 5, 5}
print(s)
# Remove will raise an error if the element is not in the set
s.remove(4)
print(s)
# Discard doesn't raise any errors
s.discard(1)
print(s)
Output:
{1, 2, 3, 4, 5}
{1, 2, 3, 5}
{2, 3, 5}
```

Operators in sets:

The below table shows the operators used for sets:

Operators	What it does
| (Union)	Returns all the unique elements in both the sets.
& (Intersection)	Returns all the elements common to both the sets.
- (Difference)	Returns the elements that are unique to the first set
^(Symmetric Difference)	Returns all the elements not common to both the sets.

The set operators are represented in the Venn Diagram below:
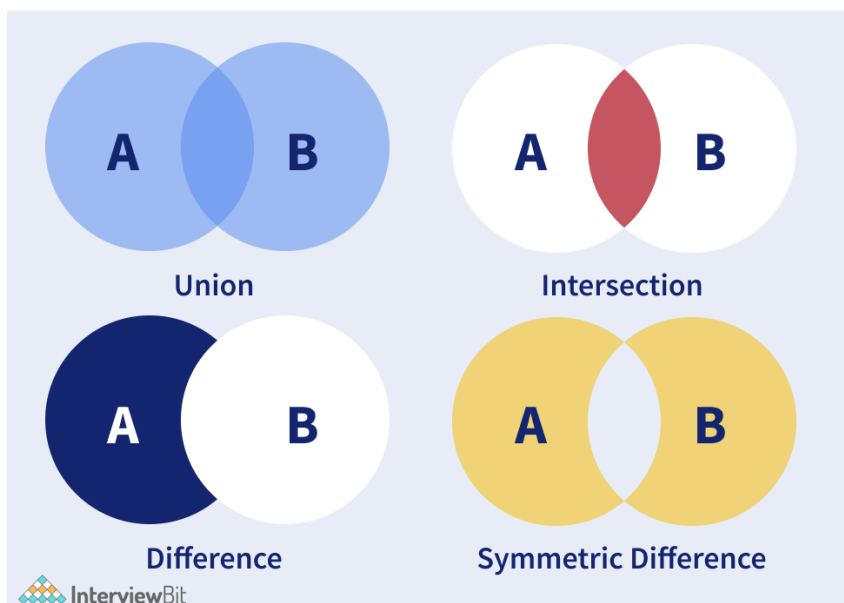
Examples:

```
a = {1, 2, 3, 3, 2, 4, 5, 5}
b = {4, 6, 7, 9, 3}
# Performs the Intersection of 2 sets and prints them
print(a & b)
# Performs the Union of 2 sets and prints them
print(a | b)
# Performs the Difference of 2 sets and prints them
print(a - b)
# Performs the Symmetric Difference of 2 sets and prints them
print(a ^ b)
Output:
{3, 4}
{1, 2, 3, 4, 5, 6, 7, 9}
{1, 2, 5}
{1, 2, 5, 6, 7, 9}
```

21. Comprehensions in Python

- **List Comprehensions:**
  It is a shorter syntax to create a new list using values of an existing list.

```
a = [0, 1, 2, 3]
# b will store values which are 1 greater than the values stored in a
b = [i + 1 for i in a]
print(b)
Output:
[1, 2, 3, 4]
```

- **Set Comprehension:**
  It is a shorter syntax to create a new set using values of an existing set.

```
a = {0, 1, 2, 3}
# b will store squares of the elements of a
b = {i ** 2 for i in a}
print(b)
Output:
{0, 1, 4, 9}
```

- **Dict Comprehension:**

   It is a shorter syntax to create a new dictionary using values of an existing dictionary.

```python
a = {'Hello':'World', 'First': 1}
# b stores elements of a in value-key pair format
b = {val: k for k , val in a.items()}
print(b)
Output:
{'World': 'Hello', 1: 'First'}
```

22. String Manipulation in Python

- **Escape Sequences:**

Escape Sequences are used to print certain characters to the output stream which carry special meaning to the language compiler.

Examples:

Escape Sequence	Results in
\t	Tab Space
\n	Newline
\\	Backslash
\'	Single Quote

- **Multiline Strings:**

Multiline Strings are used in python through triple quotes   '''

Example:

```
a = ''' Hello
        World!
        This is a
        Multiline String.'''

print(a)
Output:
Hello
World!
This is a
Multiline String.
```

- ## Strings Indexing:

Strings in Python are indexed the same way as a list of characters, based on 0-based indexing. We can access elements of a string at some index by using the [] operators.

Consider an example of the string value `Python` .


```
a = "Python"
print(a[0], a[2], a[4])
print(a[-1], a[-3], a[-5])
Output:
P t o
n h y
```

- ## Strings Slicing:

Slicing is also done the same way as in lists.

```
a = "Hello"
# Slices the string from 0 to 3 indexes
print(a[0:3])
# Slices the string from 3 to -1(same as 4) indexes
print(a[3:-1])
Output:
Hel
l
```

- **Case Conversion Functions:**

The upper() and lower() functions are used to convert a string of letters into uppercase or lowercase respectively.

The isupper() and islower() functions are used to check if a string is in all uppercase or lowercase respectively.

```
a = "Hello"
print(a)
# Converts string to uppercase
print(a.upper())
# Converts string to lowercase
print(a.lower())
# Checks if string is uppercase
print(a.isupper())
# Checks if string is lowercase
print(a.islower())
Output:
Hello
HELLO
hello
False
False
```

Other similar functions:

Function	Explanation
isspace()	Returns True if all characters in string are whitespaces
isalnum()	Returns True if given string is alphanumeric
isalpha()	Returns True if given character is alphabet
isTitle()	Returns True if string starts with an uppercase letter and then rest of the characters are lowercase

- **join() and split() Functions:**

join() function merges elements of a list with some delimiter string, and returns the result as a string.

```python
list = ["One", "Two", "Three"]
# join function
s = ','.join(list)
print(s)
Output:
One,Two,Three
```

split() function splits the into tokens, based on some delimiters and returns the result as a list.

```python
# split function
newList = s.split(',')
print(newList)
Output:
['One', 'Two', 'Three']
```

In general, a string can be split to list using split() method and a list can be joined to string using the join() method as shown in the image below:
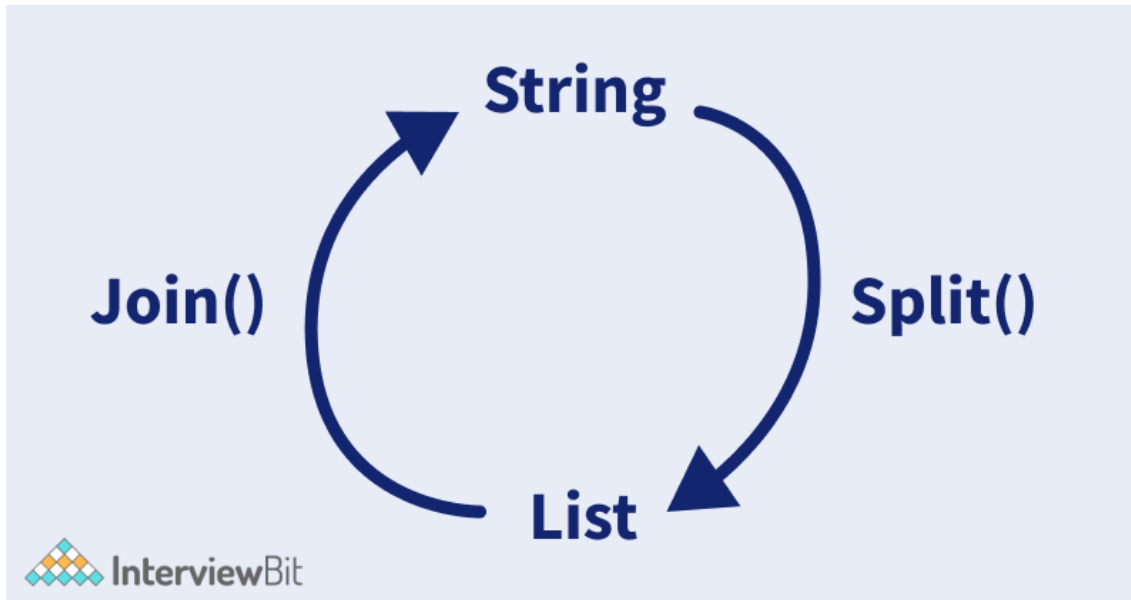
- **String Formatting:**

String Formatting is done with the str.format() function.

```
first = "first"
second = "second"
s = "Sunday is the {} day of the week, whereas Monday is the {} day of the week".format
print(s)
Output:
Sunday is the first day of the week, whereas Monday is the second day of the week
```

- **Template Strings:**

It is recommended to be used when formatting strings generated by users. They make the code less complex so are easier to understand. They can be used by importing the Template class from the string module.

Example:

```
>>> from string import Template
>>> name = 'Scaler'
>>> t = Template('Hey $name!')
>>> t.substitute(name = name)
'Hey Scaler!'
```

23. Formatting Dates in Python

To handle date and time operations in Python, we use the datetime module.

- **time class:** We can represent time values using the time class.

  Example:

```
import datetime
tm = datetime.time(1, 30, 11, 22)
print(tm)
Output:
01:30:11.000022
```

- **date class:** We can represent date values using the date class.

  Example:

```
import datetime
date = datetime.date(2000, 11, 16)
print('Date date is ', date.day, ' day of ', date.month, ' month of the year ', date
Output:
Date date is  16  day of  11  month of the year  2000
```

- **Conversion from date to time:** We can convert a date to its corresponding time using the strptime() function.

  Example:

```
from datetime import datetime
print(datetime.strptime('15/11/2000', '%d/%m/%Y'))
Output:
2000-11-15 00:00:00
```

- **time.strftime() in Python:** It converts a tuple or struct_time representing the time into a string object.

  For example:

```
from time import gmtime, strftime
s = strftime("%a, %d %b %Y %H:%M:%S + 1010", gmtime())
print(s)
Output:
Sun, 28 Nov 2021 18:51:24 + 1010
```

24. Python RegEx

- **Regex Matching**

The **re** module in python allows us to perform regex matching operations.

```
import re
landline = re.compile(r'\d\d\d\d-\d\d\d\d')
num = landline.search('LandLine Number is 2435-4153')
print('Landline Number is: {}'.format(num.group()))
Output:
Landline Number is: 2435-4153
```

The above example landline number from the string and stores it appropriately in the num variable using regex matching.

- **Parenthesis Grouping**

A group is a part of a regex pattern enclosed in parenthesis (). We can put matches into different groups using the parenthesis (). We can access the groups using group() function.

```
import re
landline = re.compile(r'(\d\d\d\d)-(\d\d\d\d)')
num = landline.search('LandLine Number is 2435-4153')
# This will print the first group, which is the entire regex enclosed in the brackets
print(num.group(0))
# This will print the second group, which is the nested regex enclosed in the 1st set c
print(num.group(1))
# This will print the third group, which is the nested regex enclosed in the 2nd set of
print(num.group(2))
Output:
2435-4153
2435
4153
```

- **Regex Symbols in Python**

There are a lot of regex symbols that have different functionalities so they are
mentioned in the table below:

Symbol	Matches
+	One or More of the preceding group
*	Zero or More of preceding group
?	Zero or One of preceding group
^name	String must begin with the name
name$	String must end with the name
.	Any character except \n
{n}	Exactly n of preceding group
{n, }	>= n of preceding group
{,n}	[0, m] of preceding group
{n, m}	[n, m] of preceding group
*?	Non Greedy matching of the preceding group
[abc]	Any character enclosed in the brackets
[^abc]	Any character not enclosed in the brackets
\d, \w, \s	Digit, word, or space respectively.
\D, \W, \S	Anything except digit, word, or space respectively

Example:

Here we define a regex pattern,

```
address = "(\\d*)\\s?(.+),\\s(.+)\\s([A-Z]{2,3})\\s(\\d{4})"
```

From the above table, we can explain some of the symbols in this pattern:

- \s?: 0 or 1 whitespace.
- (\d*): 0 or more digit characters.
- (.+): Greater than or equal to 1 characters.
- \s: Single Whitespace
- ([A-Z]{2, 3}): 2 or 3 Uppercase alphabets
- (\d{4}): 4 digit characters

25. Debugging in Python

Raising Exceptions with raise statement:

The raise statement is used to raise exceptions and consists of 3 components:

- raise keyword
- Exception() function call
- Parameter of the Exception() function, which usually contains an error message.

```
raise Exception("Error Occurred!!")
Traceback (most recent call last):
 File "./prog.py", line 1, in <module>
Exception: Error Occurred!!
```

Traceback as String

There is a function in python called traceback.format_exc() which returns the traceback displayed by Python when a raised Exception is not handled as a String type. The Traceback Module is required to be imported for this purpose.

Example:

```
import traceback
try:
    raise Exception('Error Message.')
except:
    with open('error.txt', 'w') as error_file:
        error_file.write(traceback.format_exc())
    print('The traceback info was written to error.txt.')

Output:
The traceback info was written to error.txt.
```

Assert Statements in Python:

Assert Statements/Assertions are widely used for debugging programs and checking if the code is performing some operation that is obviously wrong according to the logic of the program. The special thing about assert is that when an assert fails, the program immediately crashes, allowing us to narrow down our search space for the bug.

Writing an assert statement has the following components as a part of it,

- assert keyword
- a condition that results in a boolean value
- a display message for when the assertion fails
- a comma separating the condition and the display message.

```
>>> sum = 4
>>> assert sum == 5, 'Addition Error'
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
AssertionError: Addition Error
```

Assertions can be disabled by passing the -O option when running Python as shown in the commands below.

```
$ python -Oc "assert False"
$ python -c "assert False"
Traceback (most recent call last):
 File "<string>", line 1, in <module>
AssertionError
```

26. Logging in Python

Logging allows us to keep track of some events which occurs when some software runs. It becomes highly important in Software Development, Debugging and Running Softwares.

```python
import logging
# Create and configues the logger
logging.basicConfig(filename="newfile.log", format='%(asctime)s %(message)s', filemode=
# Creates logging object
logg = logging.getLogger()
# Sets the level of logging to DEBUG
logg.setLevel(logging.DEBUG)
# Messages
logg.debug("Debug Message")
logger.warning("Its a Warning")
logger.info("Just an information")
```

Levels of Logging:

Described in order of increasing importance

Level	Function	What it does
DEBUG	`logging.debug()`	Used for tracking any events that occur during program execution
INFO	`logging.info()`	Confirms the working of things at the end of the module in the program.
WARNING	`logging.warning()`	Used to flag some issues that might hinder the program from working in the future, but allows it to work for now.
ERROR	`logging.error()`	Records errors that might have made the program fail at some point.
CRITICAL	`logging.critical()`	Indicates or flags fatal errors in the program.

27. Lambda Function in Python

These are small anonymous functions in python, which can take any number of arguments but returns only 1 expression.

Let us understand it with an example,

Consider the function which multiplies 2 numbers:

```python
def mul(a, b):
    return a * b
print(mul(3, 5))
Output:
15
```

The equivalent Lambda function for this function will be:

```python
mul = lambda a, b: a * b
print(mul(3, 5))
Output:
15
```

The syntax for this will be as shown below:

Similarly, other functions can be written as Lambda functions too, resulting in shorter codes for the same program logic.

28. Ternary Operator in Python

The ternary operator is used as an alternative to the if-else conditional statements and provides a way to write a short crisp one-liner statement.

The syntax is as follows:

```
<expression 1> if <condition> else <expression 2>
```

```
f = 2
s = 2
# if the sum of f and s is greater than 0 the sum
# is printed, else 0 is printed
print(f + s if (f + s > 0) else 0)
Output:
4
```

29. *args and **kwargs in Python

We can pass a variable number of arguments to a function using special symbols called *args and **kwargs.

The usage is as follows:

- ***args:** For non-keyword arguments.

Example:

```
# the function will take in a variable number of arguments
# and print all of their values
def tester(*argv):
    for arg in argv:
        print(arg)
tester('Sunday', 'Monday', 'Tuesday', 'Wednesday')
Output:
Sunday
Monday
Tuesday
Wednesday
```

- ****kwargs**: For keyword arguments.

Example:

```
# The function will take variable number of arguments
# and print them as key value pairs
def tester(**kwargs):
    for key, value in kwargs.items():
        print(key, value)
tester(Sunday = 1, Monday = 2, Tuesday = 3, Wednesday = 4)
Output:
Sunday 1
Monday 2
Tuesday 3
Wednesday 4
```

30. if __name__ == "__main__" in Python

__main__ is the name of the scope in which the top-level code executes. It can check if it is running in its own scope by checking its own __name__.
Format of checking:

```
>>> if __name__ == "__main__":
...     main()
```

31. Python Dataclasses

Python Classes used for storing data objects are called Dataclasses. They have certain features like:

- Comparison with other objects of the same type is possible.
- Stores data, representing a particular data type.

Python 2.7:

The example shows the performing function of Dataclasses in older versions of python when Dataclasses were not yet introduced.

```
class Self:
    def __init__(self, x):
        self.x = x

ob = Self("One")
print(ob.x)
Output:
One
```

Python 3.7:

The example shows using dataclasses in newer versions of python.

```
@dataclass #annotation indicates that it is a dataclass module
class Self:
    x: string
ob = Self("One")
print(ob.x)
Output:
One
```

Note: It is compulsory to specify the datatype of each variable declared. If at any point we don't want to specify the type, set the type as **typing.Any.**

```
from dataclasses import dataclass
from typing import Any
@dataclass
class WithoutExplicitTypes:
    name: Any
    age: Any = 16
```

32. Python Virtual Environment

Virtual Environments are used to encapsulate certain Python Libraries of Python for single project, which might not be used in other projects, rather than installing those dependencies globally.

Installation Steps:

```
pip install virtualenv
```

```
pip install virtualenvwrapper-win
```

Usage Steps:

```
mkvirtualenv Test # Make virtual environment called Test
#setprojectdir .

deactivate # To move to something else in the command #line.

workon Test # Activate environment
```

33. Python Commands

Magic Commands are one of the new features added to the python shell. Basically, they are enhancements over the normal python code, provided by the IPython Kernel. The main syntax for these commands is that they are prefixed by as "%" character. They prove useful in solving many common problems we encounter while coding and also provide us some versatile shortcuts.
There are main kinds of commands:

- **%prefix:** The command will operate only on the given single line of code.
- **%%prefix**: The command will operate on the entire code block.

Some examples of these commands in Python are:

- %run: It is used for running an external file in Python.

```
def runner():
    print("Hello World")

runner()
%run runner.py
Output:
Hello World
```

- **%%time:** This allows us to track the amount of time taken by our code for execution.

```
%%time
for i in range(10000):
    a = a + i**2
Output:
CPU Times: user: 3.72 ms, sys: 9us, , total: 3.73ms, Wall time: 3.75ms
```

- **%%writefile:** This command will copy content from our current code cell to another external file.

```
%%writefile code.py
def func():
    print("Hello")
func()
Output:
Overwriting code.py
```

- **$pycat:** This command is used to display the contents of an external file.

```
%pycat code.py
def func():
    print("Hello")
func()
```

- **%who:** This command lists all the variables in the Python notebook.

```
a = "hello"
b = 5
c = 1
%who
Output:
a b c
```

- **%%html:** This command will let us write and execute html code in the current cell.

```
%%html
<html>
<body>
<table>
     <tr>
          <th>Name</th>
          <th>Country</th>
          <th>Age</th>
     </tr>
     <tr>
          <td>Sid</td>
          <td>India</td>
          <td>22</td>
     </tr>
     <tr>
          <td>Dave</td>
          <td>UK</td>
          <td>28</td>
     </tr>
</table>
</body>
</html>
```

Output:

Name	Country	Age
Sid	India	22
Dave	UK	28

InterviewBit

- **%env:** This command allows us to list all the environment variables, set a value for such a variable, and get the value of such a variable.

```
%env
```

```
{'TERM_SESSION_ID': 'w0t0p0:8E8D4D1D-98F1-4EF5-A51D-D4C18B1179B7',
 'SSH_AUTH_SOCK': '/private/tmp/com.apple.launchd.GMJsSYbrnd/Listeners',
 'LC_TERMINAL_VERSION': '3.3.9',
 'COLORFGBG': '7;0',
 'ITERM_PROFILE': 'Default',
 'XPC_FLAGS': '0x0',
 'PWD': '/Users/siddhesh',
 'SHELL': '/bin/zsh',
```

- **%pinfo:** This command provides detailed information regarding the object passed along with it. It works similar to that of the object? function.

```
a = "The World Makes Sense!"
%pinfo a
```

```
Type:         str
String form: The World Makes Sense!
Length:       22
Docstring:
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or
errors is specified, then the object must expose a data buffer
that will be decoded using the given encoding and error handler.
Otherwise, returns the result of object.__str__() (if defined)
or repr(object).
encoding defaults to sys.getdefaultencoding().
errors defaults to 'strict'.
```

Note: All the magic commands can be listed by using the **%lsmagic** command.

Some other useful tools for Python

- **pipenv:** It is a packaging tool for python aimed to solve common problems which are associated with the typical program workflow.
- **poetry:** It is a dependency management and packaging tool in Python.

Conclusion

We can conclude that Python is a robust, high-level, interpreted programming language. It is also an Object Oriented Programming Language that strongly follows all OOPs principles. It has various inbuilt powerful modules that follow simple syntax which makes it appealing to both beginners and experienced folks alike. A vast collection of libraries and functions makes the development of any sort much easier in Python. In this cheat sheet, we have covered the most common fundamentals of python language that would help you kickstart your career in python.

Useful Resources:

- Python Interview Questions
- Python Projects
- Python Developer: Career Guide
- Python Developer Skills
- Fast Track Python

Links to More Interview Questions

C Interview Questions	Php Interview Questions	C Sharp Interview Questions
Web Api Interview Questions	Hibernate Interview Questions	Node Js Interview Questions
Cpp Interview Questions	Oops Interview Questions	Devops Interview Questions
Machine Learning Interview Questions	Docker Interview Questions	Mysql Interview Questions
Css Interview Questions	Laravel Interview Questions	Asp Net Interview Questions
Django Interview Questions	Dot Net Interview Questions	Kubernetes Interview Questions
Operating System Interview Questions	React Native Interview Questions	Aws Interview Questions
Git Interview Questions	Java 8 Interview Questions	Mongodb Interview Questions
Dbms Interview Questions	Spring Boot Interview Questions	Power Bi Interview Questions
Pl Sql Interview Questions	Tableau Interview Questions	Linux Interview Questions
Ansible Interview Questions	Java Interview Questions	Jenkins Interview Questions