Python Revision

Plan for Job Interviews

By Mayank Kumar

# 

# **INTRODUCTION**

## **About Python**

* General purpose dynamic, interpreted, interactive, object-oriented and high level programming language.
* Supports Object Oriented programming approach to develop applications.
* Supports multiple programming patterns, including object-oriented, imperative, and functional or procedural programming styles.
* Created by [Guido van Rossum](https://en.wikipedia.org/wiki/Guido_van_Rossum) during the 1985-1990 timeline.
* First released in 1991.

## **Characteristics**

* It provided rich data types and easier to read syntax.
* It is platform independent.
* Libraries are more cross-platform compatible with Linux, MacOS and Windows.
* It supports Functional and Structural programming as well as OOP.
* No compilation step, hence editing, debugging and testing is fast.

## **Python 2 vs Python 3**

* **Division operator :** Python 2 doesn't give decimal output whereas Python 3 gives correct decimal output.
* **Print function :** Print 'Hello World' is only supported in Python 2  
  Print('Hello World') is supported in Python 3
* **Unicode :** In Python 2, implicit str type is ASCII. But in Python 3.x implicit str type is Unicode.
* **Xrange :** Python 2 supports xrange and range function but Python 3 supports only range function
* **Error Handling :** In Python 3 'as' keyword is required
* **\_future\_ module**

## **First Python Program**

| print('Hello World') |
| --- |

## **Identifiers**

* It is a name used to identify a variable, function, class, module or any other object.
* It starts with A to Z or a to z or \_(underscore) followed by 0 or more alphanumeric characters.
* Python doesn't allow punctuation characters i.e., '@','$' etc
* Python is a case sensitive language.

### Naming conventions for Python Identifiers

* Class names start with an UPPER case letter, all other identifiers start with lowercase.
* **Single Leading Underscore** \_var: Naming convention indicating name is meant for internal use. A hint for programmers and not enforced by programmers.
* **Double Leading Underscore** \_\_var: Triggers name mangling when used in class context. Enforced by the Python interpreter.
* **Single Trailing Underscore** var\_: Used by convention to avoid naming conflicts with Python keywords.
* **Double Trailing Underscore** \_\_var\_\_: Indicates special methods defined by Python language.

reserved for special use like the\_\_init\_\_ method for object constructors, or \_\_call\_\_ method to make objects callable.

* **Underscore** \_: Used as a name for temporary variables.

### Testing If an Identifier is Valid

You can test whether a Python identifier is valid or not by using the **keyword.iskeyword()** function. It returns “**True**” if the keyword is correct or “**False**” otherwise.

## **Reserved Keywords**

Reserved means —> you can not use these keywords as constants or variables or as any other identifiers.

Following list shows the 35 reserved keywords in Python 3.x:

| ***False******def******if******raise******None******del******import******return******True******elif******in******try******and******else******is******while******as******except******lambda******with******assert******finally******nonlocal******yield******break******for******not******class******from******or******continue******global******pass*** |
| --- |

## **Multi-line Statements**

* Statements in Python normally end with a new line
* Python, however, does allow the new line continuation character (\) to denote that the line should continue.

| *total = item\_one + \  item\_two + \  item\_three* |
| --- |

* Statements contained within the [], {}, or () brackets do not need to use the line continuation character.

| days = ['Monday', 'Tuesday', 'Wednesday',  'Thursday', 'Friday'] |
| --- |

## 

## **Quotations**

* Python accepts single('), double(") and triple(''' or """) quotes to denote string literals.
* Triple quotes are used to span the string across multiple lines.

| word = 'word' sentence = "This is a sentence." paragraph = """This is a paragraph. It is made up of multiple lines and sentences.""" |
| --- |

## **Comments**

* # indicates single line comment.
* Characters after # are ignored by Python Interpreter.

| *# This is a comment.* *# This is a comment, too.* *# This is a comment, too.* *# I said that already.* |
| --- |

* Following triple-quoted string is also ignored by Python interpreter and can be used as a multiline comments:

| *''' This is a multiline comment. '''* |
| --- |

## **Multiple Statements on a Single Line**

* The semicolon ( ; ) allows multiple statements on the single line given that neither statement starts a new code block.

| *import sys; x = 'foo'; print(x + '\\n')* |
| --- |

# **VARIABLES**

1. It is a memory location which holds the actual value.
2. We retrieve the value by assigning a label to that memory location.
3. We call that label a variable.
4. Variables don’t require declaration. However, you must initialize them before use.

| *test = 10* |
| --- |

* + *Creation of an object to represent the value* ***10****.*
  + *If the variable* ***(test)*** *doesn’t exist, then it’ll get created.*
  + *Association of the variable with the object, so that it can refer to the value.*

1. Whenever the expression changes, Python associates a new object (a chunk of memory) to the variable for referencing that value. And the old one goes to the garbage collector.
2. An object is just a region of memory which can hold the following:
   * The actual object values.
   * A type designator to reflect the object type.
   * The reference counter which determines when it’s OK to reclaim the object.
3. It’s the object which has a type, not the variable. However, a variable can hold objects of different types as and when required.

| ***>>>*** *test = 10* ***>>>*** *type(test) <class '****int****'>*  *>>>* ***test*** *= '****techbeamers****' >>>* ***type****(test) <****class*** *'****str****'>*  *>>>* ***test*** *= {'****Python****', '****C****', '****C****++'} >>>* ***type****(test) <****class*** *'****set****'>* |
| --- |

# 

# 

# 

# **DATA TYPES**

1. In general, data type defines the format of data which can be used by the program appropriately.
2. In Python, we don’t need to explicitly mention the data type when declaring a variable.This is known as Dynamic Typing.
3. Python identifies the type of literal directly from the syntax at the runtime.
4. Variables don’t have types instead they are just labels in Python.
5. It is the value which gets associated with a type.
6. Hence, the same label can refer to values of different data types.

Below is the **list of important data types** that are commonly used in Python:

1. ***Booleans***
2. ***Numbers***
3. ***Strings***
4. ***Bytes***
5. ***Lists***
6. ***Tuples***
7. ***Sets***
8. ***Dictionaries***

## **Booleans**

1. It has only two values.
2. **True** and **False**
3. These values are constants and can be used to assign or compare boolean values.

Example:

| condition = False if condition == True:  print("You can continue with the program.") else:  print("The program will end here.") |
| --- |

1. While making boolean conditions in Python, we can skip the explicit comparison in our code. And we’ll still get the same behavior.

Example:

| condition = False if condition:  print("You can continue with the program.") else:  print("The program will end here.") |
| --- |

| if condition: → is equivalent to → if condition == True: |
| --- |

1. Next, an expression in Python can also produce a boolean result.

| **>>>** str = "Learn Python"  **>>>** len(str) 12  **>>>** len(str) == 12 True  **>>>** len(str) != 12 False |
| --- |

1. In some cases, the boolean constants **“True”** and **“False”** might also act as numbers.

| **>>>** A, B = True + 0, False + 0 **>>>** print(A, B) 1 0 **>>>** type(A), type(B) (<class '**int**'>, <**class** '**int**'>) |
| --- |

It is evident from the above example that **True is 1** and the value of **False is 0**.

## 

## 

## **Numbers**

1. Python has 3 numerical data types.
2. Integer | Float | Complex
3. type() can be used to check the data type of any variable or the value.
4. isinstance() can be used to check the type of object.

| print(type(test)) = <class 'int'> print(isinstance(test, int)) = True |
| --- |

| num = 5 print(f'Type of num = {num} is : ', type(num))  num = 5.0 print(f'Type of num = {num} is : ', type(num))  num = 5+0j print(f'Type of num = {num} is : ', type(num)) |
| --- |

Output:

| **Type of nu**m = 5 is : <class '**int**'> **Type** **of** **num** = 5.0 **is** : <class '**float**'> **Type** **of** **num** = (5+0j) **is** : <class '**complex**'> |
| --- |

1. Integers in Python don’t have any size limitation as long as the required memory is available.

| **>>>** num = 1234567890123456789 **>>>** num.bit\_length() 61 **>>>** num 1234567890123456789 **>>>** num = 1234567890123456789123456789012345678912345678901234567891234567890123456789 **>>>** num.bit\_length() 250 |
| --- |

1. A float type number can have precision up to 15 decimal places.

| **>>>** import sys **>>>** sys.float\_info sys.float\_info(max=1.7976931348623157e+308, max\_exp=1024, max\_10\_exp=308, min=2.2250738585072014e-308, min\_exp=-1021, min\_10\_exp=-307, dig=15, mant\_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1) **>>>** sys.float\_info.dig 15 |
| --- |

The dig in above example is the maximum number of decimal digits in a float.

# 

## **Strings**

1. A sequence of one or more characters enclosed within either single or double quotes.

| **>>>** str = 'A string wrapped in single quotes' |
| --- |

1. Any letter, number or symbol can be made into a string.

| **>>>** str = "A string enclosed within double quotes" |
| --- |

1. Python also supports multi-line strings which require a triple quotation mark at the start and one at the end.

| **>>>** str = """A multiline string starts and ends with a triple quotation mark.""" |
| --- |

1. Strings in Python are immutable. It means the memory will be allocated once and re-used thereafter.

| **>>>** A = 'Python3' **>>>** id(A) 56272968 **>>>** B = A **>>>** id(B) 56272968 |
| --- |

You can see the second string shares the same address as the first one does.

1. Python allows slicing strings using a special square-bracket syntax to extract a substring.

Index starts at 0 and is inclusive

Index ends at -1 and is exclusive

| **>>>** str = "Learn Python" **>>>** first\_5\_chars = str[0:5] **>>>** print(first\_5\_chars) Learn **>>>** substr\_from\_2\_to\_5 = str[1:5] **>>>** print(substr\_from\_2\_to\_5) earn **>>>** substr\_from\_6\_to\_end = str[6:] **>>>** print(substr\_from\_6\_to\_end) Python **>>>** last\_2\_chars = str[-2:] **>>>** print(last\_2\_chars) on **>>>** first\_2\_chars = str[:2] **>>>** print(first\_2\_chars) Le **>>>** two\_chars\_before\_last = str[-3:-1] **>>>** print(two\_chars\_before\_last) ho |
| --- |

## **Bytes**

1. Byte is an immutable type in Python.
2. It can store sequences of bytes(each 8-bits) ranging from 0-255.
3. Similar to an Array, we can fetch the value using the index, but we can not change the value.

Few differences b/w Bytes & Strings:

* Byte objects contain a sequence of bytes whereas the strings store sequences of characters.
* The bytes are machine-readable objects whereas the strings are just in human-readable form.
* Since the byte is machine-readable, so they can be directly stored to the disk. Whereas, the strings first need to be encoded before getting on to the disk.

| **>>>** *# Make an empty bytes object (8-bit bytes)* **>>>** empty\_object = bytes(16) **>>>** print(type(empty\_object)) <class '**bytes**'> >>> **print**(empty\_object) **b**'\**x00**\**x00**\**x00**\**x00**\**x00**\**x00**\**x00**\**x00**\**x00**\**x00**\**x00**\**x00**\**x00**\**x00**\**x00**\**x00**' |
| --- |

# 

# **OPERATORS**

## What is an operator in Python?

1. Like other programming languages, Python reserves some special characters for acting as Operators.
2. Every Operator carries out some operation like addition, multiplication to manipulate data and variables.
3. The variables passed as input to an Operator known as operands.

| **>>>** 7 % 4 3 |
| --- |

In this case, ‘%’ is the modulus operator that calculates the remainder of the division. The numbers ‘7’ and ‘4’ passed as input are the operands, whereas the numbers ‘3’ is the result of the action performed.

1. There are different types of operators available in Python:
   * ***Arithmetic***
   * ***Comparison***
   * ***Assignment***
   * ***Logical***
   * ***Bitwise***
   * ***Identity***
   * ***Membership***

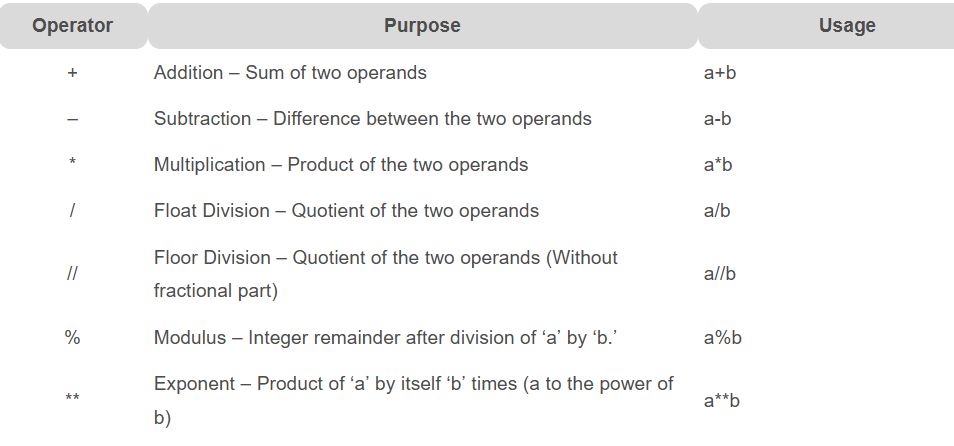
## 

## 

## 

## Arithmetic Operators

1. We can do various arithmetic operations like addition, subtraction, multiplication, division, modulus, exponent, etc.
2. The table below outlines the built-in arithmetic operators in Python:



Example:

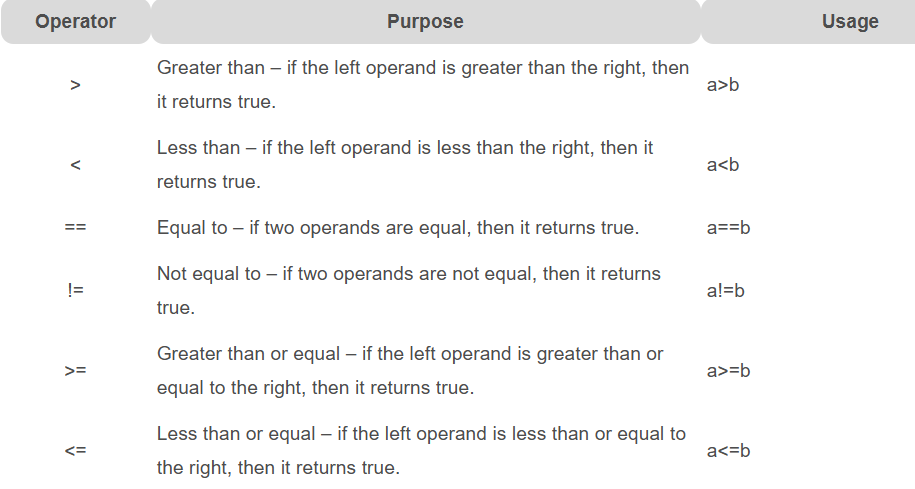
| a=7 b=4  print('Sum : ', a+b) print('Subtraction : ', a-b) print('Multiplication : ', a\*b) print('Division (float) : ', a/b) print('Division (floor) : ', a//b) print('Modulus : ', a%b) print('Exponent : ', a\*\*b) |
| --- |

Output:

| Sum : 11 Subtraction : 3 Multiplication : 28 Division (float) : 1.75 Division (floor) : 1 Modulus : 3 Exponent : 2401 |
| --- |

## Comparison Operators

1. Comparison Operators allow us to determine whether two values are equal.
2. If one is higher than the other and then make a decision based on the result.
3. The table below outlines the built-in comparison operators in Python:



Example:

| a=7 b=4  print('a > b is',a>b) print('a < b is',a<b) print('a == b is',a==b) print('a != b is',a!=b) print('a >= b is',a>=b) print('a <= b is',a<=b) |
| --- |

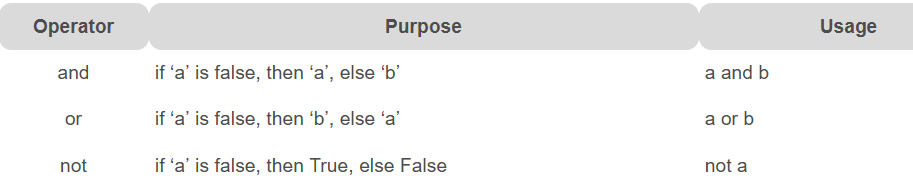
Output:

| a > b is True a < b is False a == b is False a != b is True a >= b is True a <= b is False |
| --- |

## Logical Operators

1. Logical Python operators enable us to make decisions based on multiple conditions.
2. The outcome of such an operation is either true or false (i.e., a Boolean value).
3. However, not all of these operators return a boolean result.

* The ‘and’ and ‘or’ operators do return one of their operands instead of pure boolean value.
* Whereas the ‘not’ operator always gives a real boolean outcome.



Example:

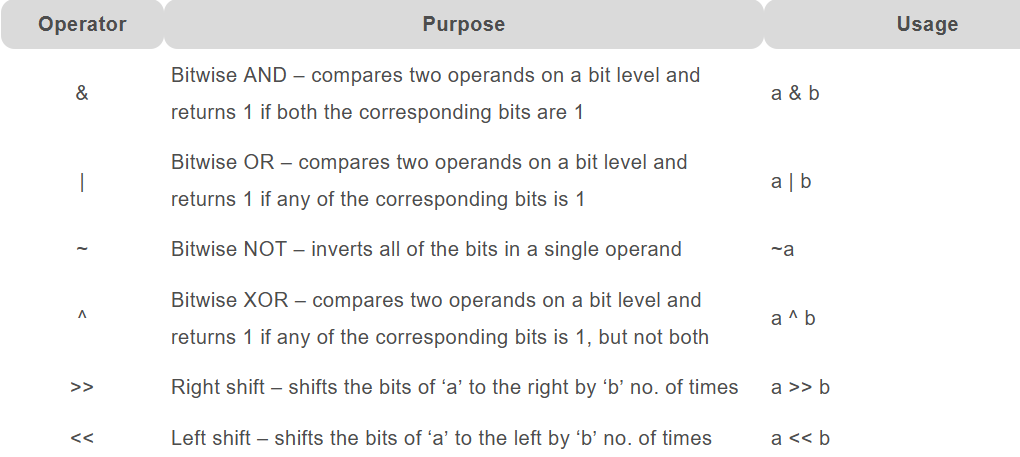
| a = 7 b = 4  *# Result: a and b is 4* print('a and b is', a and b)  *# Result: a or b is 7* print('a or b is', a or b)  *# Result: not a is False* print('not a is', not a) |
| --- |

Output:

| a and b is 4 a or b is 7 not a is False |
| --- |

## Bitwise Operators

1. Bitwise Python operators process the individual bits of integer values.
2. They treat them as sequences of binary bits.
3. We can use bitwise operators to check whether a particular bit is set.
4. For example, IoT applications read data from the sensors based on whether a specific bit is set or not. In such a situation, these operators can help.



Example:

Let’s consider the numbers 4 and 6 whose binary representations are ‘00000100’ and ‘00000110’. Now, we’ll perform the AND operation on these numbers.

| a=4 b=6  *#Bitwise AND: The result of 'a & b' is 4* print('a & b is',a & b) |
| --- |

Output:

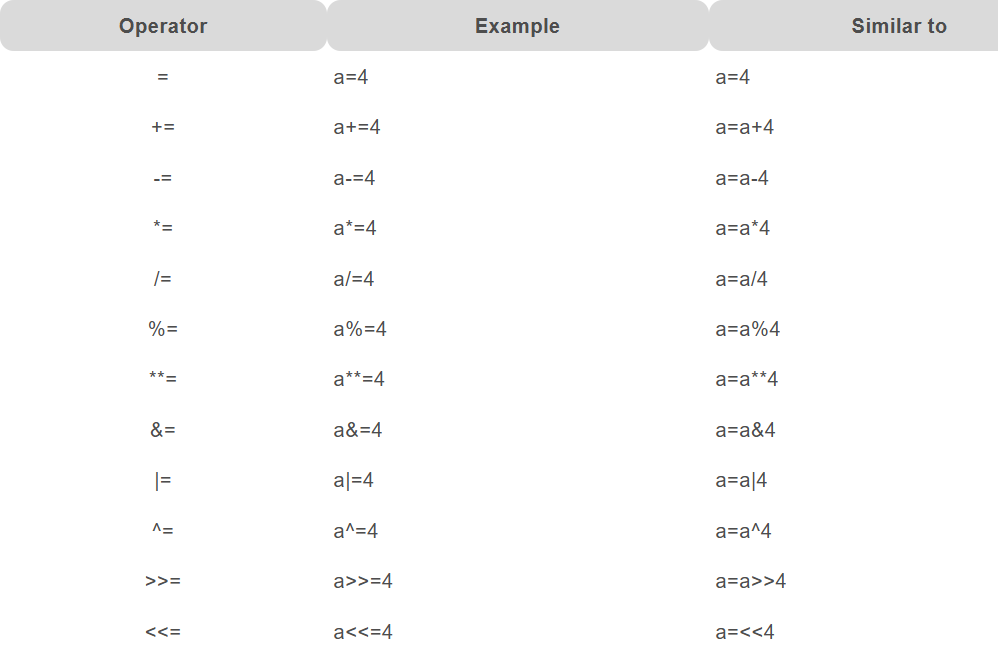
| a & b is 4 |
| --- |

The above result is the outcome of the following AND (‘&’) operation.

| 0 0 0 0 0 1 0 0 & 0 0 0 0 0 1 1 0 ------------------ 0 0 0 0 0 1 0 0 (the binary representation of the number 4) |
| --- |

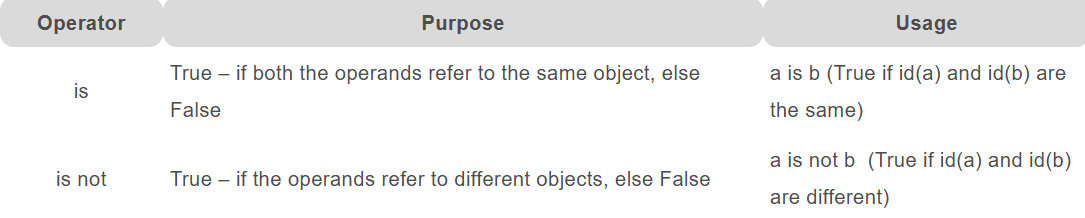
## Assignment Operators

1. We can use assignment operators to set values into variables.
2. The instruction a = 4 uses a primitive assignment operator that assigns the value 4 to the left operand.
3. Below is the list of available compound operators in Python:



## Identity Operators (is | is not)

1. These operators enable us to compare the memory locations of two Python objects/variables.
2. They can let us find if the objects share the same memory address.
3. The variables holding equal values are not necessarily identical.
4. Alternatively, we can use these operators to determine whether a value is of a specific class or type.



Example:

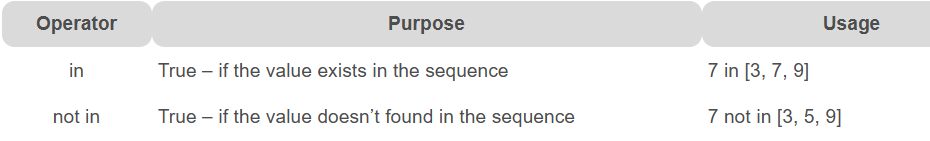
| *# Using 'is' identity operator* a = 7 if (type(a) is int):  print("true") else:  print("false")  *# Using 'is not' identity operator* b = 7.5 if (type(b) is not int):  print("true") else:  print("false") |
| --- |

Output:

| true true |
| --- |

## Membership Operators(in | not in)

1. Membership operators enable us to test whether a value is a member of other Python objects such as strings, lists, or tuples.
2. Also, note that this operator can also test against a dictionary but only for the key, not the value.



Example:

| *# Using Membership operator* str = 'Python operators' dict = {6:'June',12:'Dec'}  print('P' in str)  print('Python' in str) print('python' not in str) print(6 in dict)  print('Dec' in dict) |
| --- |

Output:

| True True True True False |
| --- |

# **INPUT/OUTPUT**

1. Developers often have a need to interact with users, either to get data or to provide some sort of result.
2. Python provides us with two inbuilt functions to read the input from the keyboard.

* input ( prompt )
* raw\_input ( prompt ) - works only in Python 2.x

**input()**

* This function first takes the input from the user and then evaluates the expression, which means Python automatically identifies whether the user entered a string or a number or list.
* If the input provided is not correct then either syntax error or exception is raised by python.

Example:

| val = input('Enter your Value : ') print(val) |
| --- |

**How the input function works in Python :**

* When input() function executes program flow will be stopped until the user has given an input.
* The text or message display on the output screen to ask a user to enter input value is optional i.e. the prompt, will be printed on the screen is optional.
* **Whatever you enter as input, *the input* function converts it into a string. if you enter an integer value still input() function convert it into a string. You need to explicitly convert it into an integer in your code using** [**typecasting**](https://www.geeksforgeeks.org/taking-input-from-console-in-python/)**.**

# **PYTHON NAMESPACE & SCOPE**

1. The concept of namespaces is not limited to any particular programming language. C/C++ and Java also have it where it works as a means to distinguish between different sections of a program.
2. The body of a section may consist of a method, or a function, or all the methods of a class. So, a namespace is a practical approach to define the scope, and it helps to avoid name conflicts.
3. While in Python, the namespace is a fundamental idea to structure and organize the code, especially more useful in large projects.

### 

### 

## What are names in Python?

1. Before getting on to namespaces, first, let’s understand what Python means by a name.
2. A name in Python is just a way to access a variable like in any other languages.
3. However, Python is more flexible when it comes to the variable declaration.
4. You can declare a variable by just assigning a name to it.
5. You can use names to reference values:

| num = 5 str = 'Z' seq = [0, 1, 1, 2, 3, 5] |
| --- |

1. You can even assign a name to a function.

| def **function**():  print('It is a function.')   foo = function foo() |
| --- |

1. You can also assign a name and then reuse it. Check the below example; it is alright for a name to point to different values.

| test = -1 print("type <test> :=", type(test)) test = "Pointing to a string now" print("type <test> :=", type(test)) test = [0, 1, 1, 2, 3, 5, 8] print("type <test> :=", type(test)) |
| --- |

And here is the output follows.

| type <test> := <class '**int**'> type <test> := <class '**str**'> type <test> := <class '**list**'> |
| --- |

So, you can see that one name is working perfectly fine to hold data of different types.

1. The naming mechanism works inline with Python’s object system, i.e., everything in Python is an object.
2. All the data types such as numbers, strings, functions, classes are all objects.

### 

## What are namespaces in Python?

1. A namespace is a simple system to control the names in a program.
2. It ensures that names are unique and won’t lead to any conflict.
3. Also, add to your knowledge that Python implements namespaces in the form of dictionaries.
4. It maintains a name-to-object mapping where names act as keys and the objects as values.
5. Multiple namespaces may have the same name but pointing to a different variable.

**Local Namespace:**

* This namespace covers the local names inside a function.
* Python creates this namespace for every function called in a program.
* It remains active until the function returns.

**Global Namespace:**

* This namespace covers the names from various imported modules used in a project.
* Python creates this namespace for every module included in your program.
* It’ll last until the program ends.

**Built-in Namespace:**

* This namespace covers the built-in functions and built-in exception names.
* Python creates it as the interpreter starts and keeps it until you exit.

## What is Scope in Python?

1. Namespaces make our programs immune from name conflicts.
2. However, it doesn’t give us a free ride to use a variable name anywhere we want.
3. Python restricts names to be bound by specific rules known as a scope.
4. The scope determines the parts of the program where you could use that name without any prefix.

**Scopes**:

Python outlines different scopes for locals, function, modules, and built-ins. Check out from the below list.

* A local scope, also known as the innermost scope, holds the list of all local names available in the current function.
* A scope for all the enclosing functions, it finds a name from the nearest enclosing scope and goes outwards.
* A module level scope, it takes care of all the global names from the current module.
* The outermost scope which manages the list of all the built-in names. It is the last place to search for a name that you cited in the program.

<https://www.techbeamers.com/python-namespace-scope/>

\*\*\*\*\*\*\*\* TO BE COMPLETED \*\*\*\*\*\*\*\*

# 

# 

# 

# 

# 

# 

# 

# 

# 

# **FORMATTING**

1. The format() method formats the specified value(s) and inserts them inside the string's placeholder.
2. The placeholder is defined using curly brackets: {}.
3. The format() method returns the formatted string.

## Single Argument Formatting

* syntax : **'{}'.format(param)**

\*\* '**{}**': It is the format target, i.e., the placeholder.

\*\* **param**: It can be a string, integer, float, or any of the collection types.

\*\* **Return**: It returns a formatted output with the input argument substituting the placeholder.

Example:

| **>>>** '{}'.format('Formatting a String in Python') 'Formatting a String in Python' |
| --- |

## Multiple Arguments Formatting

* syntax : **'{} {}'.format(arg1, arg2)**

\*\* '**{} {}**': We can have multiple placeholders.

\*\* **arg1, arg2**: They can be a string, integer, float, or any of the collection types.

\*\* **Return**: It returns a formatted output with the input argument substituting the placeholder.

Example:

| **>>>** '{} {}'.format('Python', 'Format') 'Python Format' |
| --- |

## 

## Basic String Formatting

* Print a single string using the format function.

| **>>>** test = "Hello" **>>>** print('{}'.format(test)) O/P: Hello |
| --- |

* Combine two string literals and print them using the format function.

| first = "Hello" second = "World" print('{} {}'.format(first, second))  O/P: Hello World |
| --- |

## Padding and Align Strings

* We can also use the Python format function to allow padding.
* Its purpose is to align a string either using a space or some value. You need to provide the length for the alignment which should be higher than the size of the target string.
* It is because the alignment can only occur in a box of fixed length.

Let’s align a string to the right with spaces.

| **>>>** print(format("Hello", ">10s"))  Hello |
| --- |

Now, let’s do the same operation using the ‘#’ character.

| **>>>** print(format("Hello", "#>10s")) *#####Hello* |
| --- |

Here is a brief description of the above steps.

* You can see that the box length for printing the string is 10.
* It means the max we can accommodate the ten characters.
* The word “Hello” itself has five letters and the ‘#’ gets filled at the remaining five places.
* You might have observed the “>” symbol in the format syntax. It makes our target string move to the right side.

If you like to align the target string from the left, then use the ‘<‘ symbol.

| **>>>** print(format("Hello", "#<10s")) Hello*#####* |
| --- |

You can even make a string center-aligned inside a fixed-length box.

| **>>>** print(format("Hello", "#^15s")) *#####Hello#####* |
| --- |

The carat ‘^’ sign makes the string format to the center.

## Format Integers

#### **Using basic number formatting**

| print("I've <{}> years of experience and my salary is <{}> USD per annum.".format(10, 75000))  O/P:  I've <10> years of experience and my salary is <75000> USD per annum. |
| --- |

#### **Using separator number formatting**

* It is a standard convention to display the salary with commas. Python format function supports this representation and requires a pair of ‘:’ and ‘,’ inside the parenthesis.

| print("I've <{}> years of experience and my salary is <{:,}> USD per annum.".format(10, 75000))  O/P:  I've <10> years of experience and my salary is <75,000> USD per annum. |
| --- |

#### **Specify field width for numbers**

* Same as we did for strings is applicable for the integers. For integers, we can’t use precision.

| print("I've <{:5}> years of experience and my salary is <{:15,}> USD per annum.".format(10, 75000))  O/P:  I've < 10> years of experience and my salary is < 75,000> USD per annum. |
| --- |

#### 

#### 

#### **Format a number as binary**

* The symbol ‘b’ after the colon inside the parenthesis notifies to display a number in binary format.

| **>>>** print('{0:b}'.format(10)) 1010 |
| --- |

#### **Format a number as octal**

* The symbol ‘o’ after the colon inside the parenthesis notifies to display a number in octal format.

| **>>>** print('{0:o}'.format(10)) 12 |
| --- |

#### **Format a number as hex**

* The symbol ‘x’ or ‘X’ after the colon inside the parenthesis notifies to display a number in hex format.

| **>>>** print('{0:x}'.format(10)) a  **>>>** print('{0:X}'.format(10)) A |
| --- |

**Float number**

Let’s print a full floating point number.

| **>>>** print("{0:.3f}".format(1.123456)) 1.123 |
| --- |

Now, let’s print a floating point number truncating after three decimal points.

| **>>>** print("{0:.3f}".format(1.123456)) 1.123 |
| --- |

Finally, let’s print a floating point number which truncates after three decimal places but does round the final value.

| **>>>** print("{0:.3f}".format(1.123556)) 1.124 |
| --- |

**Format Lists**

* The Python format method accepts a sequence of positional parameters. If we pass an array or a List, then let’s find out the result.

| **>>>** langs = ['C', 'C++', 'CSharp'] **>>>** print('Skillset: {}'.format(langs)) Skillset: ['C', 'C++', 'CSharp'] |
| --- |

* The whole list gets displayed. You can also decide to print one item from the sequence by providing its index.

| **>>>** print('Skillset: {0[1]}'.format(langs)) Skillset: C++ |
| --- |

* You can even send the list item as the positional parameters. To achieve this, unpack them using the \* operator.

| **>>>** print('Skillset: {}'.format(\*langs)) Skillset: C |
| --- |

#### **Formatting a dict in Python**

* Format function allows using a dictionary as a parameter. See the below example.

| **>>>** print(" Jake's salary is {0[jake]} \n Anand's salary is {0[anand]}".format({'jake': '$100K', 'anand': '$120K'})) Jake's salary is $100K  Anand's salary is $120K |
| --- |

* You can format dictionary data using the keyword arguments.

| **>>>** print(" Jake's salary is {sal[jake]} \n Anand's salary is {sal[anand]}".format(sal={'jake': '$100K', 'anand': '$120K'})) Jake's salary is $100K  Anand's salary is $120K |
| --- |

# **MODULES**

1. Modules are primarily the (.py) files which contain Python code defining functions, class, variables, etc. with a suffix .py appended in its file name.
2. They can have different functions, variables, and classes in one file.
3. We can also call them libraries.
4. A Python module brings certain benefits such as we can reduce redundancy in the code. It can let us maintain uniformity in the coding style.
5. Generally, it is a good practice to create modules which have a fixed purpose.
6. It increases readability and increases productivity and bug reporting.

**Examples:**

From Python Standard Library:

* OS, Time, Math, MatPlotlib, etc.

From Online Sources:

* Keras(for deep learning), Numpy(for number manipulation), Pandas(for array manipulation),etc.

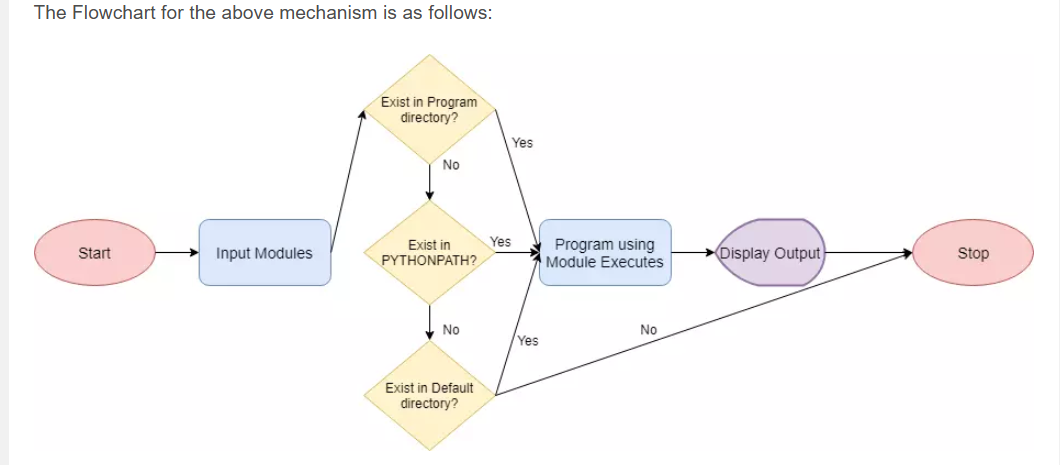
#### 

#### 

#### **Mechanism**

When we import modules, the python interpreter locates them from three locations:

1. The directory from the program is getting executed.
2. The directory specified in the PYTHONPATH variable (A shell variable or an environment variable).
3. The default directory (It depends on the OS distribution).



#### **Modules Listing**

* To find out the list of modules present in python, we can issue the command: **help(“modules”)** in Python interpreter shell.

### Implementation

1. Using Full Name

import os

1. Using a Short Name

import math as m

# **CONTROL FLOW**

## **CONDITIONAL STATEMENT**

1. A conditional statement (like if X > Y:) envelopes around a block of code.
2. It lets Python evaluate the logical expression (X > Y) first.
3. If the result is True, then only the code indented under the conditional block gets executed.

## **IF STATEMENT**

1. A bare Python if statement evaluates whether an expression is True or False.
2. It executes the underlying code only if the result is True.

**Syntax**

| if Logical\_Expression :  Indented Code Block |
| --- |

**Example**

| days = int(input("How many days in a leap year? ")) if days == 366:  print("You have cleared the test.") print("Congrats!")  Output:  How many days in a leap year? 366  You have cleared the test.  Congrats! |
| --- |

## **IF-ELSE STATEMENT**

1. A Python if else statement takes action irrespective of what the value of the expression is.
2. If the result is True, then the code block following the expression would run.
3. Otherwise, the code indented under the else clause would execute.

**Syntax**:

| if Logical\_Expression :  Indented Code Block 1 else :  Indented Code Block 2 |
| --- |

**Example:**

| answer = input("Is Python an interpreted language? Yes or No >> ").lower()  if answer == "yes" :  print("You have cleared the test.") else :  print("You have failed the test.")  print("Thanks!")  Output:  Is Python an interpreted language? Yes or No >> **yes**  You have cleared the test.  Thanks!  Is Python an interpreted language? Yes or No >> **no**  You have failed the test.  Thanks! |
| --- |

*Steps to check for student understanding*

1. Vivamus ultrices eros nec lectus interdum condimentum.
2. Cras vel felis a leo pretium pulvinar.
3. Curabitur tincidunt elementum ante

# **FUNCTIONS**

# **FILE HANDLING**

# **EXCEPTION HANDLING**

## **Difference between Syntax Error and Exceptions**

* **Syntax Error:** As the name suggests this error is caused by wrong syntax in the code. It leads to the termination of the program.
* **Exceptions:** Exceptions are raised when the program is syntactically correct but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

When a Python script raises an exception, it creates an exception object.

**Note:** Exception is the base class for all the exceptions in Python.

## **How to Handle Exceptions with Try-Except?**

1. We use the try-except statement to enable exception handling in Python programs.
2. Inside the try block, you write the code which can raise an exception.
3. And the code that handles or catches the exception, we place in the except clause.

Following is the syntax of a **Python try-except-else** block.

| try:  You do your operations here;  ...................... except ExceptionI:  If there is ExceptionI, then execute this block. except ExceptionII:  If there is ExceptionII, then execute this block.  ...................... else:  If there is no exception then execute this block. |
| --- |

Here is a checklist for using the Python try statement effectively :

* A single try statement can have multiple except statements depending on the requirement. In this case, a try block contains statements that can throw different types of exceptions.
* We can also add a generic except clause which can handle all possible types of exceptions.
* We can even include an else clause after the except clause. The instructions in the else block will execute if the code in the try block doesn’t raise an exception.

Example:

| try:  fob = open("test", "w")  fob.write("This is my test file for exception handling!!") except IOError:  print "Error: can\'t find the file or read data" else:  print "Write operation is performed successfully on the file"  fob.close() |
| --- |

## **Handling All Types of Exceptions with Except**

* If we use a bare **“except”** clause, then it would catch all types of exceptions.
* However, neither it’s a good programming practice nor does anyone recommend it.
* It is because such a Python try-except block can handle all types of exceptions. But it’ll not help the programmer to find what exception caused the issue.

You can go through the below code to see how to catch all exceptions.

| try:  You do your operations here;  ...................... except:  If there is any exception, then execute this block.  ...................... else:  If there is no exception then execute this block. |
| --- |

## **Handling Multiple Exceptions with Except**

* We can define multiple exceptions with the same except clause. It means that if the Python interpreter finds a matching exception, then it’ll execute the code written under the except clause.
* In short, when we define an except clause in this way, we expect the same piece of code to throw different exceptions. Also, we want to take the same action in each case.

Please refer to the below example:

| try:  You do your operations here;  ...................... except(Exception1[, Exception2[,...ExceptionN]]]):  If there is any exception from the given exception list,  then execute this block.  ...................... else:  If there is no exception then execute this block |
| --- |

## **How to handle Exceptions with Try-Finally?**

1. We can also enable Python exception handling with the help of try-finally statement.
2. With try block, we also have the option to define the “**finally**” block. This clause allows defining statements that we want to execute, no matters whether the try block has raised an exception or not.
3. This feature usually comes in the picture while releasing external resources.

Here is the coding snippet for help:

| try:  You do your operations here;  ......................  Due to any exception, this may be skipped. finally:  This would always be executed.  ...................... |
| --- |

Note : *One critical point is that we can either define an “****except****” or a “****finally****” clause with every try block. You can’t club these together. Also, you shouldn’t use the “****else****” clause along with a “****finally****” clause.*

| try:  fob = open('test', 'w')  fob.write("It's my test file to verify try-finally in exception handling!!"  )  print 'try block executed' finally:  fob.close()  print 'finally block executed' |
| --- |

If the exception doesn’t occur, then you’ll see the following output.

| >>try block executed >>finally block executed |
| --- |

When some code causes an exception in a try block, the execution immediately passes to the “**finally**” block. After all the statements in the “**finally**” block gets executed, the exception resumes to the “**except**” block for execution. But there must present a next higher layer of the “**try-except**” statement.

## 

## 

## 

## 

## 

## **Raise Exception with Arguments**

* We can forcefully raise an exception using the raise keyword.
* We can also optionally pass values to the exception and specify why it has occurred.

Syntax:

| raise [Exception [, args [, traceback]]] |
| --- |

Where,

* Under the **“Exception”** – specify its name.
* The “**args**” is optional and represents the value of the exception argument.
* The final argument, **“traceback,”** is also optional and if present, is the traceback object used for the exception.

## **Create Custom Exceptions in Python**

# 

# **MODULES**

# **OOP**

# **ADVANCED TOPICS**