Python Introduction -1

Keywords

- Keywords are the reserved words in Python and we cannot use a keyword as variable name, function name or any other identifier.
- They are used to define the syntax and structure of the Python language.
- In Python, keywords are case sensitive.
- There are 33 keywords in Python 3.*.
- All the keywords accept True, False and None are in lowercase and they must be written as it is. The list of all the keywords are given below

Keywords

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

- Get the list of keywords in your current version by typing the following in the prompt.
- **>>>import keyword**
- **>>>print(keyword.kwlist)**

Get Started With Python Introduction -1

- Keywords
- Identifier
- Statements & Comments
- Datatypes
- I/O and Import

Identifiers

- Identifier is the name given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another.
- **Rules for writing identifiers**
  - Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like myClass, var_1 and print_this_to_screen, all are valid example.
  - An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
  - Keywords cannot be used as identifiers.

```
>>> global = 1  File "<interactive input>", line 1

global = 1

^SyntaxError: invalid syntax
```

- We cannot use special symbols like !, @, #, $, % etc. in our identifier.

```
>>> a@ = 0
 File "<interactive input>", line 1
   a@ = 0
    ^
SyntaxError: invalid syntax
```

- Identifier can be of any length.

Things to care about

- Python is a case-sensitive language. This means, Variable and variable are not the same. Always name identifiers that make sense.
- While, c = 10 is valid. Writing count = 10 would make more sense and it would be easier to figure out what it does even when you look at your code after a long gap.
- Multiple words can be separated using an underscore, this_is_a_long_variable.
- We can also use camel-case style of writing,
- i.e., capitalize every first letter of the word except the initial word without any spaces.
- For example: camelCaseExample.

Statements

- Instructions that a Python interpreter can execute are called statements.
- For example, a = 1 is an assignment statement. if statement, for statement, while statement etc.
- **Multi-line statement**
  - In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\).
  - For example:

```
a = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8 + 9
```

  - This is explicit line continuation.

- This is explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ] and braces { }.
- For instance, we can implement the above multi-line statement as

```
a = (1 + 2 + 3 +
    4 + 5 + 6 +
    7 + 8 + 9)
```

- Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with [ ] and { }. For example:

```
colors = ['red',
          'blue',
          'green']
```

- We could also put multiple statements in a single line using semicolons, as follows

```
a = 1; b = 2; c = 3
```

Indentation

- Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.
- A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.
- Generally four whitespaces are used for indentation and is preferred over tabs. Here is an example.

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

- Indentation can be ignored in line continuation. But it's a good idea to always indent. It makes the code more readable. For example:

```
if True:
    print('Hello')
    a = 5
```

and

```
if True: print('Hello'); a = 5
```

- both are valid and do the same thing. But the former style is clearer.
- Incorrect indentation will result into IndentationError

Comments

- Comments are very important while writing a program. It describes what's going on inside a program so that a person looking at the source code does not have a hard time figuring it out.
- You might forget the key details of the program you just wrote in a month's time. So taking time to explain these concepts in form of comments is always fruitful.
- In Python, we use the hash (#) symbol to start writing a comment. It extends up to the newline character.
- Comments are for programmers for better understanding of a program. Python Interpreter ignores comment.

```
#This is a comment
#print out Hello
print('Hello')
```

Multi-line comments

- If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line. For example:

  #This is a long comment

  #and it extends

  #to multiple lines

- Another way of doing this is to use triple quotes, either ''' or """".
- These triple quotes are generally used for multi-line strings. But they can be used as multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

  """"This is also a

  perfect example of

  multi-line comments""""

Variables

- A variable is a location in memory used to store some data (value).
- They are given unique names to differentiate between different memory locations.
- The rules for writing a variable name is same as the rules for writing identifiers in Python.
- We don't need to declare a variable before using it.
- In Python, we simply assign a value to a variable and it will exist.
- We don't even have to declare the type of the variable. This is handled internally according to the type of value we assign to the variable

Variable assignment

- We use the assignment operator (=) to assign values to a variable. Any type of value can be assigned to any valid variable.

```
a = 5
b = 3.2
c = "Hello"
```

- Here, we have three assignment statements. 5 is an integer assigned to the variable a.
- Similarly, 3.2 is a floating point number and "Hello" is a string (sequence of characters) assigned to the variables b and c respectively.

Multiple assignments

- In Python, multiple assignments can be made in a single statement as follows:

```
a, b, c = 5, 3.2, "Hello"
```

- If we want to assign the same value to multiple variables at once, we can do this as
  - x = y = z = "same"
- This assigns the "same" string to all the three variables.

Data types in Python

- Every value in Python has a datatype. Since everything is an object in Python programming.
- Datatypes are actually classes and variables are instance (object) of these classes.
- There are various datatypes in Python. Some of the important types are listed below.
  - Numbers
  - List
  - Tuple
  - Strings
  - Set
  - Dictionary

Numbers

- Integers, floating point numbers and complex numbers falls under Python numbers category. They are defined as int, float and complex class in Python.
- We can use the type() function to know which class a variable or a value belongs to and the isinstance() function to check if an object belongs to a particular class.

```
>>> a = 5
>>> type(a)
<class 'int'>
>>> type(2.0)
<class 'float'>
>>> isinstance(1+2j,complex)
True
```

- Integers can be of any length, it is only limited by the memory available.
- A floating point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points.
- 1 is integer, 1.0 is floating point number. Complex numbers are written in the form, x + yj, where x is the real part and y is the imaginary part.
- Here are some examples.

```
>>> a = 1234567890123456789
>>> a
1234567890123456789
>>> b = 0.1234567890123456789
>>> b
0.12345678901234568
>>> c = 1+2j
>>> c
(1+2j)
```

- Notice that the float variable b got truncated.

List

- **List** is an ordered sequence of items. It is one of the most used datatype in Python and is very flexible.
- All the items in a list do not need to be of the same type.
- Declaring a list is pretty straight forward.
- Items separated by commas are enclosed within brackets [ ].

```
>>> a = [1, 2.2, 'python']
>>> type(a)
<class 'list'>
```

- We can use the slicing operator [ ] to extract an item or a range of items from a list. Index starts from 0 in Python.

```
>>> a = [5,10,15,20,25,30,35,40]

>>> a[2]

15

>>> a[0:3]

[5, 10, 15]

>>> a[5:]

[30, 35, 40]
```

- Lists are mutable, meaning, value of elements of a list can be altered.

```
>>> a = [1,2,3]

>>> a[2]=4

>>> a

[1, 2, 4]
```

Tuple

- **Tuple** is an ordered sequence of items same as list. The only difference is that tuples are immutable.
- Tuples once created cannot be modified. They are used to write-protect data and are usually faster than list as it cannot change dynamically.
- Tuple is defined within parentheses () where items are separated by commas.

```
>>> t = (5,'program', 1+3j)
>>> type(t)
<class 'tuple'>
```

- We can use the slicing operator [] to extract items but we cannot change its value.

```
>>> t[1]
'program'
>>> t[0:3]
(5, 'program', (1+3j))
>>> t[0] = 10
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Strings

- **String** is sequence of Unicode characters.
- We can use single quotes or double quotes to represent strings.
- Multi-line strings can be denoted using triple quotes, ''' or """.

```
>>> s = "This is a string"
>>> type(s)                    <class 'str'>
>>> s = '''a multiline
               ... string'''
```

- Like list and tuple, slicing operator [ ] can be used with string. Strings are immutable.

```
>>> s = 'Hello world!'
>>> s[4]           'o'
>>> s[6:11]        'world'
>>> s[5] ='d'
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
```

- TypeError: 'str' object does not support item assignment

Set

- Set is an unordered collection of unique items.
- Set is defined by values separated by comma inside braces { }.
- Items in a set are not ordered.

```
>>> a = {5,2,3,1,4}
>>> a
{1, 2, 3, 4, 5}
>>> type(a)
<class 'set'>
```

- We can perform set operations like union, intersection on two sets.

- Set have unique values. They eliminate duplicates.

```
>>> a = {1,2,2,3,3,3}
>>> a
{1, 2, 3}
```

- Since, set are unordered collection, indexing has no meaning. Hence the slicing operator [] does not work.

```
>>> a = {1,2,3}
>>> a[1]
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
```

- TypeError: 'set' object does not support indexing

Dictionary

- **Dictionary** is an unordered collection of key-value pairs.
- It is generally used when we have a huge amount of data.
- Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.
- In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value.
- Key and value can be of any type.
  - >>> d = {1:'value','key':2}
  - >>> type(d)
  - <class 'dict'>

- We use key to retrieve the respective value. But not the other way around.

```
d = {1:'value','key':2}
print(type(d))
<class 'dict'>
print("d[1] = ", d[1]);
d[1] =  value
print("d['key'] = ", d['key']);
d['key'] =  2
# Generates error
print("d[2] = ", d[2]);
Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
    print("d[2] = ", d[2]);
KeyError: 2
```

Conversion between datatypes

- We can convert between different data types by using different type conversion functions like int(), float(), str() etc.

```
>>> float(5)          5.0
```

- Conversion from float to int will truncate the value (make it closer to zero).

```
>>> int(10.6)         10
>>> int(-10.6)        -10
```

- Conversion to and from string must contain compatible values.

```
>>> float('2.5')            2.5
>>> str(25)           '25'
>>> int('1p')
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1p'
```

- We can even convert one sequence to another.

```
>>> set([1,2,3])              {1, 2, 3}
>>> tuple({5,6,7})            (5, 6, 7)
>>> list('hello')             ['h', 'e', 'l', 'l', 'o']
```

- To convert to dictionary, each element must be a pair

```
>>> dict([[1,2],[3,4]])            {1: 2, 3: 4}
>>> dict([(3,26),(4,44)])          {3: 26, 4: 44}
```

I/O and Import

- Python language provides numerous built-in functions that are promptly (readily)available to us at the Python prompt.
- Some of the functions like input() and print() are widely used for standard input and output operations respectively
  - Input
  - Output
  - Import

Output Using print() Function

- We use the print() function to output data to the standard output device (screen).
    - print(“Hello RGUKT RK Valley”) Hello RGUKT RK Valley
    - >>> a=5 >>>print(“The Value of a is “,a) The value of a is 5
- we can notice that a space was added between the string and the value of variable *a*. This is by default, but we can change it.
- The actual syntax of the print() function is
    - print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
- Here, objects is the value(s) to be printed.
- The sep separator is used between the values. It defaults into a space character.
- After all values are printed, end is printed. It defaults into a new line.

- File is the object where the values are printed and its default value is sys.stdout (screen). Here are an example to illustrate this.

```
print(1,2,3,4)
print(1,2,3,4,sep='*')
print(1,2,3,4,sep='#',end='&')
```

- Output

```
1 2 3 4
1*2*3*4
1#2#3#4&
```

Output formatting

- We would like to format our output to make it look attractive. This can be done by using the **str.format()** method. This method is visible to any string object.

```
>>> x = 5; y = 10          >>> print('The value of x is {} and y is {}'.format(x,y))
The value of x is 5 and y is 10
```

- Here the curly braces {} are used as placeholders. We can specify the order in which it is printed by using numbers (tuple index).

```
>>> print('I love My {0} and {1}'.format(‘Mother', 'Father'))        I love My Mother and butter
>>> print('I love My {1} and {0}'.format(‘Father',' Mother'))        I love My Father and Mother
```

- We can even use keyword arguments to format the string.

```
>>> print('Hello {name}, {greeting}'.format(greeting='Goodmorning',name='John'))        Hello John, Goodmorning
```

- We can even format strings like the old sprintf() style used in C programming language. We use the % operator to accomplish this.

```
>>> x = 12.3456789        >>> print('The value of x is %3.2f' %x)        The value of x is 12.35
>>> print('The value of x is %3.4f' %x)                                   The value of x is 12.3457
```

Input Using input() Function

- As of now we wrote all programs in static method.
- The values of variables were fixed or defined into source code. This can only execute single output all the times.
- To overcome the single output execution , we need to allow flexible(dynamic) input values from the end-user(Keyboard).
- In Python, we have the input() function to allow this.
- The syntax for input() is **input([prompt])**
  - **Note:-** Here prompt will access String values from end-user

```
>>> num = input('Enter a number: ')          Enter a number: 10
>>> num                              '10'
```

How to Convert String to int or float

- Where prompt is the string and it is display on screen .It is optional
- Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use int() or float() functions

```
>>> int('10')          10
>>> float('10')        10.0
```

- This same operation can be performed using the eval() function. But it takes it further. It can evaluate even expressions, provided the input is a string

```
>>> int('2+3')
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '2+3'
>>> eval('2+3')        5
```

Next Python Operators