Learning Python Basic

08 May 2018 12:35

Learning Python Basic

First of all, what is Python? According to its creator, Python is a:
"*high-level programming language, and its core design philosophy is all about code readability and a syntax which allows programmers to express concepts in few lines of code.*"

The first reason to learn python was that it is, in fact, a beautiful programming language. It was really natural to code in it and express out thoughts.

Another reason was that we can use coding in Python in multiple ways: Data science, web development, and machine learning all shine here. Quora, Pinterest and Spotify all use Python for their backend web development.

The Basic

1. Variable and types:

Variables are nothing but reserved memory locations to store values. This means that when we create a variable we reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decided what can be stored in the reserved memory. Therefore, by assigning different data types to variables, we can store integers, decimals or characters in these variables.

Python is completely object oriented and not "statically typed". So we don't need to declare variables before using them, or declare their type. Every variable in Python is an object.

Python Support two types of numbers - integers and floating point numbers.
Strings are defined either with a single quote or a double quotes.

Besides integers, we can also use Booleans (True/False)

Besides integers, we can also use Booleans (True / False) for to represent truth values(although other values can also be considered false or true).

```
# Integers
myint = 7

# Float
book_price = 15.80

# Booleans
true_boolean = True
false_boolean = False

# String
country_name = "India"
```

2. Control Flow: conditional statements:

"**if**" uses an expression to evaluate whether as statement is True or False. If it is True, it executes what is inside the "if" statement. e.g.

```
if True:
    print("Hello Python if")

if 2 > 1:
    print("2 is greater than 1")
```

The "**else**" statement will be executed if the "if" expression is false.

```
if  1 > 2:
    print("1 is greater than 2")
else
    print("1 is not greater than 2")
```

We can also use an "**elif**" statement:

```
if 1 > 2:
    print("1 is greater than 2")
elif 2 > 1:
    print("1 is not greater than 2")
else:
    print("1 is equal to 2")
```

2. Looping / Iterator:

In Python, we can iterate in different forms: **while** and **for.**

while looping: while the statement is True, the code inside the block will be executed. e.g.

```
count = 0
while (count < 11):
   print(count)
   count = count + 1
```

The while loop needs a "loop condition". It stays True, it continues iterating until we set it to False.

for looping: for loop that is use to iterate over elements of a sequence, it is often used when we have a piece of code which we want to repeat "n" number of time.
It works like this: "for all elements in a list, do this".

This code will print the same as **while** code:

```
for count in range(1,11):
    print(count);
```

List: Collection | Array | Data Structure

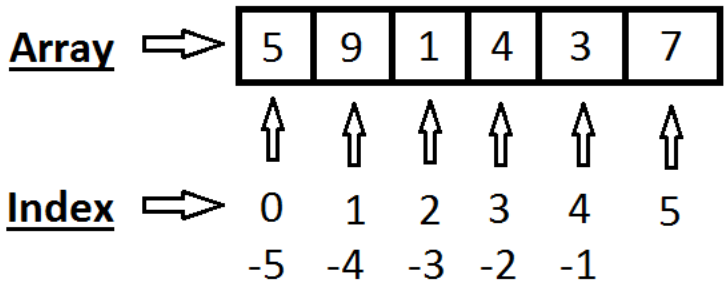
The list is a simplest data structure in Python and is used to store a list of values. Lists are collection of items (strings, integers, or even other lists. So let's use it:

```
my_integers = [1, 2, 3, 4, 5, 6]
```

It is really simple. We created an array and stored in **my_integers.**

A list is mutable, meaning we can change its contents

Lists are collection of items where each item in the list has an assigned index values. The first item gets index 0, the second get 1, and so on. To make it clearer, we can represent the array and each element with its index.



Using the Python syntax, it's also simple to understand:

```
my_integers = [5, 7, 1, 3, 4]
print(my_integers[0]) # 5
print(my_integers[1]) # 7
print(my_integers[4]) # 4
```

Lists are very easy to create, these are some of the ways to make lists.

```
list1 = ['one, two, three, four, five']
numlist = [1, 3, 5, 7, 9]
mixlist = ['yellow', 'red', 'blue', 'green', 'black']
```

An empty list is created using just square brackets:

```
list = []
```

We can get the length of a list using length function:

```
list = ["1", "hello", 2, "world"]
len(list)
>>4
```

With the help of append function we just add the item at the end of list.

```
list = ["Movies", "Music", "Pictures"]
list.append("Files")
print(list)
>>['Movies', 'Music', 'Pictures', 'Files']
```

We can use insert function for insert a item anywhere in the list.

```
list = ["Movies", "Music", "Pictures"]
list.insert(2,"Documents")
print(list)
>>['Movies', 'Music', 'Documents', 'Pictures', 'Files']
```

To remove an item first occurrence in a list, simply use remove function.

```
a = [1, 2, 3, 4]
a.remove(2)
print a
>>[1, 3, 4]
```

To remove an item based on index position use the del function.

```
a = [1, 2, 3, 4]
Del a[2]
print a
>>[1, 2, 4]
```

Python includes following list methods:

list.append(obj)	-> Appends object obj to list
list.count(obj)	-> Returns count of how many times obj occurs in list

list.extend(seq)	-> Appends the contents of seq to list
list.index(obj)	-> Returns the lowest index in list that obj appears
list.insert(index, obj)	-> Insert object obj into list at offset index
list.pop(obj=list[-1])	-> Removes and returns last object / obj from list
list.remove(obj)	-> Removes object obj from list
list.reverse()	-> Reverses objects of list in place
list.sort([func])	-> Sorts object of list, use compare func if given

Build-in list functions:

len(list)	-> Gives the total length of the list
max(list)	-> Returns item from the list with max value
min(list)	-> Returns item from the list with min value
list(seq)	-> Converts a tuple into list

Examples:

```
# Getting length of list
len([1, 2, 3, 4])
>> 2

# Concatenation
[1, 2, 3] + [4, 5, 6]
>> [1, 2, 3, 4, 5, 6]

# Repetition
['Hi!'] * 4
>>['Hi!', 'Hi!', 'Hi!', 'Hi!']

# Membership
3 in [1, 2, 3]
>> True

# Iteration
for x in [1, 2, 3]:
    print(x)
>>1 2 3

L = ['spam', 'Spam', 'SPAM!']
L[2]   # offest start at 0
>> 'SPAM!'

L[-2]  # Negative: count from the right
>>'Spam'

L[1:]  # Slicing fetches sections
>>['Spam', 'SPAM!']
```

Tuple: Immutable list Data Structure

Similar to python lists, but tuples are to store a sequence of items that is immutable (or unchangeable) and ordered. Tuples are initialized with () brackets rather than [] brackets as with lists. That means that, to create one, you simply have to do the following:

```
cake = ('c', 'a', 'k', 'e')
print(type(cake))
>><class 'tuple'>
```

The type() is a build-in function that allows you to check the data type of the parameter passed to it.

Tuples can hold both homogeneous as well as heterogeneous values. However, remember that once you declared those values, you cannot change them.

```
mixed_type = ('c', 0, 4, 'k', 'i', 'e')
mixed_type[1] = 'o'  # get an error
```

Tuples provide **read-only** access to the data values but they are also faster than lists.

According to the official Python documentation, immutable is 'an object with a fixed value', but 'value' is a rather vague term, the correct term for tuples would be 'id'. 'id' is the identity of the location of an object in memory.

Let's look a little more in-depth

```
# Tuple 'n_tuple' with a list as one of its item
n_tuple = (1,1,[3,4])
# Items with same values have the same id.
id(n_tuple[0]) == id(n_tuple[1])
>>True

# Items with different value have different id
id(n_tuple[0]) == id(n_tuple[2])
>>False

n_tuple.append(5)
>>AttributeError: 'tuple' object has no attribute 'append'

n_tuple[2].append(5)
n_tuple
>>(1, 1, [3, 4, 5])
```

Thus, allowing you to actually mutate the original tuple. This is because, the id of the list within the tuple still remains the same even though when we appended 5 to it.

Some tuples (that contain only immutable objects: string, integer etc) are immutable and some other tuples (that contain one or more mutable objects: lists, dictionary etc) are mutable.

You can't add elements to a tuple because of their immutable property. There's no append() or extend() method for tuples.
You can't remove elements from a tuple, also because of their immutability. Tuples have no remove() or pop() method.
You can find elements in a tuple since this doesn't change the tuple.
You can also use the in operator to check if an element exists in the tuple.

So, if you're defining a constant set of values and all you're going to do with it is iterate through it, use a tuple instead of a list. It will be faster than working with lists and also safer, as the tuples contain **write-protect** data.

Examples:

```
my_tuple = ('a', 'p',  'p', 'l', 'e',)
# count(x) : Return the number of items that is equal to x
print(my_tuple.count('p'))
>>2
# index(x) : return index of first item that is equal to x
print(my_tuple.index('l'))
>>3

# in operation
```

```
print('a' in my_tuple)
>>True
print('b' in my_tuple)
>>False

# not in operation
print('g' not in my_tuple)
>>True

# Iterating Through a Tuple
for name in ('John','Kate'):
     print("Hello ",name)
>>Hello John
  Hello Kate

x = (1, 2, 3, 4)
y = (5, 6, 7, 8)
# combining two tuples to form a new tuple
z = x + y
print(z)
>>(1, 2, 3, 4, 5, 6, 7, 8)
z = x * 2
print(z)
>>(1, 2, 3, 4, 1, 2, 3, 4)

# Assigning multiple values
a = (1, 2, 3)
(one, two, three) = a
print(one)
>>1
```

Built-in Tuple functions:

all()	-> Return *True* if all elements of the tuple are true (or if the tuple is empty).
any()	-> Return *True* if any element of the tuple is true. If the tuple is empty, return *False*
enumerate()	-> Return an enumerate object. It contains the index and value of all the items of tuple as pairs
len()	-> Return the length (the number of items) in the tuple
max()	-> Return the largest item in the tuple
min()	-> Return the smallest item in the tuple
sorted()	-> Take elements in the tuple and return a new sorted list(does not sort the tuple itself).
sum()	-> Return the sum of all elements in the tuple
tuple()	-> Convert an iterable (list, string, set, dictionary) to a tuple

Dictionary: key-Value Data Structure

Python dictionary is an unordered collection of items. While other compound data types (like list) have only value as an element, a dictionary is collection of key-value pairs

Creating a dictionary is an simple as placing items curly braces {} separated by comma.

While values can be of any data types and can repeat, keys must be of immutable type(like string, numbers or tuple with immutable elements) and must be unique

```
# empty dictionary
my_dict = {}

# dictionary with integer keys
```

```
my_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()
my_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
```

Dictionary **values** have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the **keys.**

More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during the assignment, the last assignment wins.

```
Dict = {'Name' : 'Zara', 'Age' : 7, 'Name' : 'Manni' }
Print(Dict['Name'])
>>Manni
```

Keys must be immutable. Which means you can use strings, numbers or tuple as dictionary keys but something like ['keys'] is not allowed,
Dict = {['Name'] : 'Zara', 'Age' : 7} # not allowed

Python includes the following functions & methods:

cmp(dict1, dict2)	-> Removes all elements of dictionary *dict*
len(dict)	-> Gives the total length of the dictionary. Thus would be equal to the number of items in the dictionary
str(dict)	-> Produces a printable string representation of a dictionary
type(variable)	-> Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Dictionary methods:

dict.copy()	-> Returns a shallow copy of dictionary *dict*
dict.fromkeys()	-> Create a new dictionary with keys from seq and values *set* to *value*.
dict.get(key, default=None)	-> For *key* key, returns value or default it key not in dictionary
dict.has_key(key)	-> Returns *true* if key in dictionary *dict*, *false* otherwise
dict.items()	-> Returns a list of dict's(key, value) tuple pairs
dict.keys()	-> Returns list of dictionary dict's keys
dict.setdefault(key, default=None)	-> Similar to get(), but will dict[key]=default if key is not already in dict
dict.update(dict2)	-> Adds dictionary *dict2*'s key-values pairs to *dict*.
dict.values()	-> Returns list of dictionary *dict*'s values

Sets:

Set in Python is a data structure equivalent to sets in mathematics that's perform union, intersection, symmetric difference etc.

A set is an ordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed). However, the set itself is mutable. We can add or remove items from it.

A set is created by placing all the elements inside {} braces, separated by comma or by using the build-in function set(). It can have any number of elements and they may be of different types(integer, float, tuple, string etc.).

```
my_set = {1, 2, 3}
my_set = {1.0, "Hello", (1, 2, 3)}
```

Creating an empty set is a bit tricky. Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements we use the set() function without any argument.

```
a  = {}
print(type(a))
>><class 'dict'>
a = set()
print(type(a))
>><class 'set'>
```

We cannot access or change an element of set using indexing or slicing. Set does not support it.

Sets ae implemented in a way, which doesn't allow mutable objects. e.g. we cannot include lists as elements

```
# valid set
cities = set((("Python","Perl"), ("Paris", "Berlin", "London")))
# invalid set because of its get an error while running
cities = set((["Python","Perl"], ["Paris", "Berlin", "London"]))
```

Though sets can't contain mutable objects, sets are mutable

```
cities = set(["Paris", "Berlin","London"])
cities.add("Strasbourg")
print(cities)
>>set(['Paris', 'Berlin', 'London', 'Strasbourg'])
```

Frozensets are like sets except that they cannot be changed, i.e. they are immutable

```
cities = set(["Paris", "Berlin","London"])
cities.add("Strasbourg")
>>
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Methods for sets

s.add(x)	-> Adds the element x to set s if it is not already present in the set. (Element x has to be immutable object)
s.clear()	-> All elements will removed from a set s.
s.copy()	-> Creates a shallow copy, which is returned
s.update(t) or s|=t	-> return set s with elements added from t
s.intersection(t) or s & t	-> returns the intersection of the instance set and the set t as a new set.(A set with all the elements which are contained in both sets is returned.)
s.intersection_update(t) or s &= t	-> return set s keeping only elements also found in t
s.difference(t) or s - t	-> returns the difference of two or more sets as new set
s.difference_update(t) or s -= t	-> return set s after removing elements from in t
s.symmetric_difference_update(t) or s ^= t	-> return set s with elements from s or t but not both add element x to set s
s.remove(x)	-> remove x from set s; raises KeyError if not present
s.discard(x)	-> removes x from set s if present

s.pop()	-> remove and return an element an arbitrary element from s; raises KeyError if empty
s.issubset(t) or s < t	-> return True if s is a subset of t
s.issuperset(t)	-> returns True if s is a superset of t

Examples:

```
colours = {"red","green"}
colours.add("yellow")
colours
>>set(['green', 'yellow', 'red'])
colours.add(["black","white"])
>>
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'

cities = {"Stuttgart", "Konstanz", "Freiburg"}
cities.clear()
cities
>>set([])

x = {"a","b","c","d","e"}
y = {"b","c"}
z = {"c","d"}
x.difference(y) # or x - y
>>set(['a', 'e', 'd'])
x.difference(y).difference(z) # or x - y -z
>>set(['a', 'e'])


x = {"a","b","c","d","e"}
y = {"b","c"}
x.difference_update(y) # or x = x - y or x -= y
>>set(['a', 'e', 'd'])


x = {"a","b","c","d","e"}
x.discard("a")
x
>>set(['c', 'b', 'e', 'd'])
x.discard("z")
x
>>set(['c', 'b', 'e', 'd'])

x = {"a","b","c","d","e"}
x.remove("a")
x
>>set(['c', 'b', 'e', 'd'])
x.remove("z")
>>
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'z'


x = {"a","b","c","d","e"}
y = {"c","d","e","f","g"}
x.intersection(y) # or x & y
>>set(['c', 'e', 'd'])
```

```
x = {"a","b","c","d","e"}
y = {"c","d"}
x.issubset(y)
>>False
y.issubset(x)
>>True
x < y
>>False
y < x # y is a proper subset of x
>>True
x < x # a set can never be a proper subset of oneself.
>>False
x <= x
>>True


x = {"a","b","c","d","e"}
y = {"c","d"}
x.issuperset(y)
>>True
x > y
>>True
x >= y
>>True
x >= x
>>True
x > x
>>False
x.issuperset(x)
>>True

x = {"a","b","c","d","e"}
x.pop()
>>'a'
x.pop()
>>'c'
```