Object Oriented Python - Data Structures

Python data structures are very intuitive from a syntax point of view and they offer a large choice of operations. You need to choose Python data structure depending on what the data involves, if it needs to be modified, or if it is a fixed data and what access type is required, such as at the beginning/end/random etc.

Lists

A List represents the most versatile type of data structure in Python. A list is a container which holds comma-separated values (items or elements) between square brackets. Lists are helpful when we want to work with multiple related values. As lists keep data together, we can perform the same methods and operations on multiple values at once. Lists indices start from zero and unlike strings, lists are mutable.

Data Structure - List

```
>>>
>>> # Any Empty List
>>> empty_list = []
>>>
>>> # A list of String
>>> str_list = ['Life', 'Is', 'Beautiful']
>>> # A list of Integers
>>> int_list = [1, 4, 5, 9, 18]
>>>
>>> #Mixed items list
>>> mixed_list = ['This', 9, 'is', 18, 45.9, 'a', 54, 'mixed', 99, 'list']
>>> # To print the list
>>>
>>> print(empty_list)
[]
>>> print(str_list)
['Life', 'Is', 'Beautiful']
>>> print(type(str_list))
<class 'list'>
```

```
>>> print(int_list)
[1, 4, 5, 9, 18]
>>> print(mixed_list)
['This', 9, 'is', 18, 45.9, 'a', 54, 'mixed', 99, 'list']
```

Accessing Items in Python List

Each item of a list is assigned a number – that is the index or position of that number.Indexing always start from zero, the second index is one and so forth. To access items in a list, we can use these index numbers within a square bracket. Observe the following code for example –

```
>>> mixed_list = ['This', 9, 'is', 18, 45.9, 'a', 54, 'mixed', 99, 'list']
>>>
>>> # To access the First Item of the list
>>> mixed_list[0]
'This'
>>> # To access the 4th item
>>> mixed_list[3]
18
>>> # To access the last item of the list
>>> mixed_list[-1]
'list'
```

Empty Objects

Empty Objects are the simplest and most basic Python built-in types. We have used them multiple times without noticing and have extended it to every class we have created. The main purpose to write an empty class is to block something for time being and later extend and add a behavior to it.

To add a behavior to a class means to replace a data structure with an object and change all references to it. So it is important to check the data, whether it is an object in disguise, before you create anything. Observe the following code for better understanding:

```
>>> #Empty objects
>>>
>>> obj = object()
>>> obj.x = 9
Traceback (most recent call last):
File "<pyshell#3>", line 1, in <module>
```

```
obj.x = 9
AttributeError: 'object' object has no attribute 'x'
```

So from above, we can see it's not possible to set any attributes on an object that was instantiated directly. When Python allows an object to have arbitrary attributes, it takes a certain amount of system memory to keep track of what attributes each object has, for storing both the attribute name and its value. Even if no attributes are stored, a certain amount of memory is allocated for potential new attributes.

So Python disables arbitrary properties on object and several other built-ins, by default.

```
>>> # Empty Objects
>>>
>>> class EmpObject:
    pass
>>> obj = EmpObject()
>>> obj.x = 'Hello, World!'
>>> obj.x
'Hello, World!'
```

Hence, if we want to group properties together, we could store them in an empty object as shown in the code above. However, this method is not always suggested. Remember that classes and objects should only be used when you want to specify both data and behaviors.

Tuples

Tuples are similar to lists and can store elements. However, they are immutable, so we cannot add, remove or replace objects. The primary benefits tuple provides because of its immutability is that we can use them as keys in dictionaries, or in other locations where an object requires a hash value.

Tuples are used to store data, and not behavior. In case you require behavior to manipulate a tuple, you need to pass the tuple into a function(or method on another object) that performs the action.

As tuple can act as a dictionary key, the stored values are different from each other. We can create a tuple by separating the values with a comma. Tuples are wrapped in parentheses but not mandatory. The following code shows two identical assignments .

```
>>> stock1 = 'MSFT', 95.00, 97.45, 92.45
>>> stock2 = ('MSFT', 95.00, 97.45, 92.45)
>>> type (stock1)
<class 'tuple'>
```

```
>>> type(stock2)
<class 'tuple'>
>>> stock1 == stock2
True
>>>
```

Defining a Tuple

Tuples are very similar to list except that the whole set of elements are enclosed in parentheses instead of square brackets.

Just like when you slice a list, you get a new list and when you slice a tuple, you get a new tuple.

```
>>> tupl = ('Tuple','is', 'an','IMMUTABLE', 'list')
>>> tupl
('Tuple', 'is', 'an', 'IMMUTABLE', 'list')
>>> tupl[0]
'Tuple'
>>> tupl[-1]
'list'
>>> tupl[1:3]
('is', 'an')
```

Python Tuple Methods

The following code shows the methods in Python tuples –

```
>>> tupl
('Tuple', 'is', 'an', 'IMMUTABLE', 'list')
>>> tupl.append('new')
Traceback (most recent call last):
   File "<pyshell#148>", line 1, in <module>
      tupl.append('new')
AttributeError: 'tuple' object has no attribute 'append'
>>> tupl.remove('is')
Traceback (most recent call last):
   File "<pyshell#149>", line 1, in <module>
      tupl.remove('is')
AttributeError: 'tuple' object has no attribute 'remove'
>>> tupl.index('list')
4
>>> tupl.index('new')
Traceback (most recent call last):
   File "<pyshell#151>", line 1, in <module>
```

```
      tupl.index('new')
ValueError: tuple.index(x): x not in tuple
>>> "is" in tupl
True
>>> tupl.count('is')
1
```

From the code shown above, we can understand that tuples are immutable and hence –

- You **cannot** add elements to a tuple.
- You **cannot** append or extend a method.
- You **cannot** remove elements from a tuple.
- Tuples have **no** remove or pop method.
- Count and index are the methods available in a tuple.

Dictionary

Dictionary is one of the Python's built-in data types and it defines one-to-one relationships between keys and values.

Defining Dictionaries

Observe the following code to understand about defining a dictionary –

```
>>> # empty dictionary
>>> my_dict = {}
>>>
>>> # dictionary with integer keys
>>> my_dict = { 1:'msft', 2: 'IT'}
>>>
>>> # dictionary with mixed keys
>>> my_dict = {'name': 'Aarav', 1: [ 2, 4, 10]}
>>>
>>> # using built-in function dict()
>>> my_dict = dict({1:'msft', 2:'IT'})
>>>
>>> # From sequence having each item as a pair
>>> my_dict = dict([(1,'msft'), (2,'IT')])
>>>
>>> # Accessing elements of a dictionary
>>> my_dict[1]
'msft'
```

```
>>> my_dict[2]
'IT'
>>> my_dict['IT']
Traceback (most recent call last):
   File "<pyshell#177>", line 1, in <module>
   my_dict['IT']
KeyError: 'IT'
>>>
```

From the above code we can observe that:

- First we create a dictionary with two elements and assign it to the variable **my_dict**. Each element is a key-value pair, and the whole set of elements is enclosed in curly braces.
- The number **1** is the key and **msft** is its value. Similarly, **2** is the key and **IT** is its value.
- You can get values by key, but not vice-versa. Thus when we try **my_dict['IT']** , it raises an exception, because **IT** is not a key.

Modifying Dictionaries

Observe the following code to understand about modifying a dictionary –

```
>>> # Modifying a Dictionary
>>>
>>> my_dict
{1: 'msft', 2: 'IT'}
>>> my_dict[2] = 'Software'
>>> my_dict
{1: 'msft', 2: 'Software'}
>>>
>>> my_dict[3] = 'Microsoft Technologies'
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies'}
```

From the above code we can observe that –

- You cannot have duplicate keys in a dictionary. Altering the value of an existing key will delete the old value.
- You can add new key-value pairs at any time.
- Dictionaries have no concept of order among elements. They are simple unordered collections.

Mixing Data types in a Dictionary

Observe the following code to understand about mixing data types in a dictionary –

```
>>> # Mixing Data Types in a Dictionary
>>>
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies'}
>>> my_dict[4] = 'Operating System'
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies', 4: 'Operating Syste
>>> my_dict['Bill Gates'] = 'Owner'
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies', 4: 'Operating Syste
'Bill Gates': 'Owner'}
```

From the above code we can observe that –

- Not just strings but dictionary value can be of any data type including strings, integers, including the dictionary itself.
- Unlike dictionary values, dictionary keys are more restricted, but can be of any type like strings, integers or any other.

Deleting Items from Dictionaries

Observe the following code to understand about deleting items from a dictionary –

```
>>> # Deleting Items from a Dictionary
>>>
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies', 4: 'Operating Syste
'Bill Gates': 'Owner'}
>>>
>>> del my_dict['Bill Gates']
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies', 4: 'Operating Syste
>>>
>>> my_dict.clear()
>>> my_dict
{}
```

From the above code we can observe that –

- **del** – lets you delete individual items from a dictionary by key.
- **clear** – deletes all items from a dictionary.

Sets

Set() is an unordered collection with no duplicate elements. Though individual items are immutable, set itself is mutable, that is we can add or remove elements/items from the set. We can perform mathematical operations like union, intersection etc. with set.

Though sets in general can be implemented using trees, set in Python can be implemented using a hash table. This allows it a highly optimized method for checking whether a specific element is contained in the set

Creating a set

A set is created by placing all the items (elements) inside curly braces **{}**, separated by comma or by using the built-in function **set()**. Observe the following lines of code –

```
>>> #set of integers
>>> my_set = {1,2,4,8}
>>> print(my_set)
{8, 1, 2, 4}
>>>
>>> #set of mixed datatypes
>>> my_set = {1.0, "Hello World!", (2, 4, 6)}
>>> print(my_set)
{1.0, (2, 4, 6), 'Hello World!'}
>>>
```

Methods for Sets

Observe the following code to understand about methods for sets –

```
>>> >>> #METHODS FOR SETS
>>>
>>> #add(x) Method
>>> topics = {'Python', 'Java', 'C#'}
>>> topics.add('C++')
>>> topics
{'C#', 'C++', 'Java', 'Python'}
>>>
>>> #union(s) Method, returns a union of two set.
>>> topics
{'C#', 'C++', 'Java', 'Python'}
```

```
>>> team = {'Developer', 'Content Writer', 'Editor','Tester'}
>>> group = topics.union(team)
>>> group
{'Tester', 'C#', 'Python', 'Editor', 'Developer', 'C++', 'Java', 'Content
Writer'}
>>> # intersets(s) method, returns an intersection of two sets
>>> inters = topics.intersection(team)
>>> inters
set()
>>>
>>> # difference(s) Method, returns a set containing all the elements of
invoking set but not of the second set.
>>>
>>> safe = topics.difference(team)
>>> safe
{'Python', 'C++', 'Java', 'C#'}
>>>
>>> diff = topics.difference(group)
>>> diff
set()
>>> #clear() Method, Empties the whole set.
>>> group.clear()
>>> group
set()
>>>
```

Operators for Sets

Observe the following code to understand about operators for sets –

```
>>> # PYTHON SET OPERATIONS
>>>
>>> #Creating two sets
>>> set1 = set()
>>> set2 = set()
>>>
>>> # Adding elements to set
>>> for i in range(1,5):
   set1.add(i)
>>> for j in range(4,9):
   set2.add(j)
>>> set1
{1, 2, 3, 4}
>>> set2
{4, 5, 6, 7, 8}
```

```
>>>
>>> #Union of set1 and set2
>>> set3 = set1 | set2 # same as set1.union(set2)
>>> print('Union of set1 & set2: set3 = ', set3)
Union of set1 & set2: set3 = {1, 2, 3, 4, 5, 6, 7, 8}
>>>
>>> #Intersection of set1 & set2
>>> set4 = set1 & set2 # same as set1.intersection(set2)
>>> print('Intersection of set1 and set2: set4 = ', set4)
Intersection of set1 and set2: set4 = {4}
>>>
>>> # Checking relation between set3 and set4
>>> if set3 > set4: # set3.issuperset(set4)
   print('Set3 is superset of set4')
elif set3 < set4: #set3.issubset(set4)
   print('Set3 is subset of set4')
else: #set3 == set4
   print('Set 3 is same as set4')
Set3 is superset of set4
>>>
>>> # Difference between set3 and set4
>>> set5 = set3 - set4
>>> print('Elements in set3 and not in set4: set5 = ', set5)
Elements in set3 and not in set4: set5 = {1, 2, 3, 5, 6, 7, 8}
>>>
>>> # Check if set4 and set5 are disjoint sets
>>> if set4.isdisjoint(set5):
   print('Set4 and set5 have nothing in common\n')
Set4 and set5 have nothing in common
>>> # Removing all the values of set5
>>> set5.clear()
>>> set5 set()
```