Week 6 – Python Data Structures

There are four basic data structures in the Python programming language:

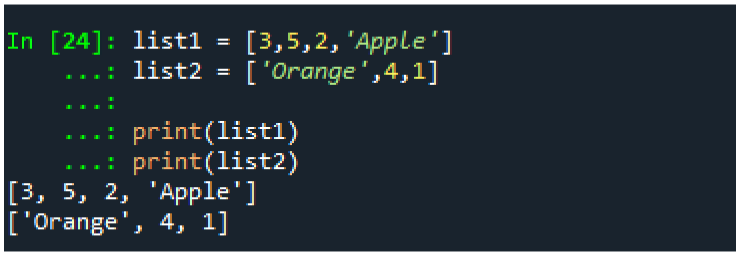
1. **List** is a collection which is ordered and changeable. Allows duplicate members.
2. **Dictionary** is a collection which is ordered** and changeable. No duplicate members.
3. **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
4. **Set** is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.

Lists

Lists in Python are dynamically sized arrays much like the vectors we used in R. This essentially a collection of things enclosed in brackets `[ ]` and separated by commas. Lists are the simplest containers that are an integral part of the Python language. Lists need not be homogenous always which makes it the most powerful tool in Python. A single list may contain DataTypes like integers, strings, as well as objects. Lists are mutable, and hence, they can be altered even after their creation.

Creating Lists

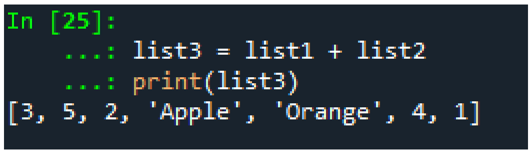
Creating lists in Python is straight forward. You can create an empty list using brackets or create a list with elements as shown in the below example.



```
In [24]: list1 = [3,5,2,'Apple']
    ...: list2 = ['Orange',4,1]
    ...:
    ...: print(list1)
    ...: print(list2)
[3, 5, 2, 'Apple']
['Orange', 4, 1]
```

Combining Lists

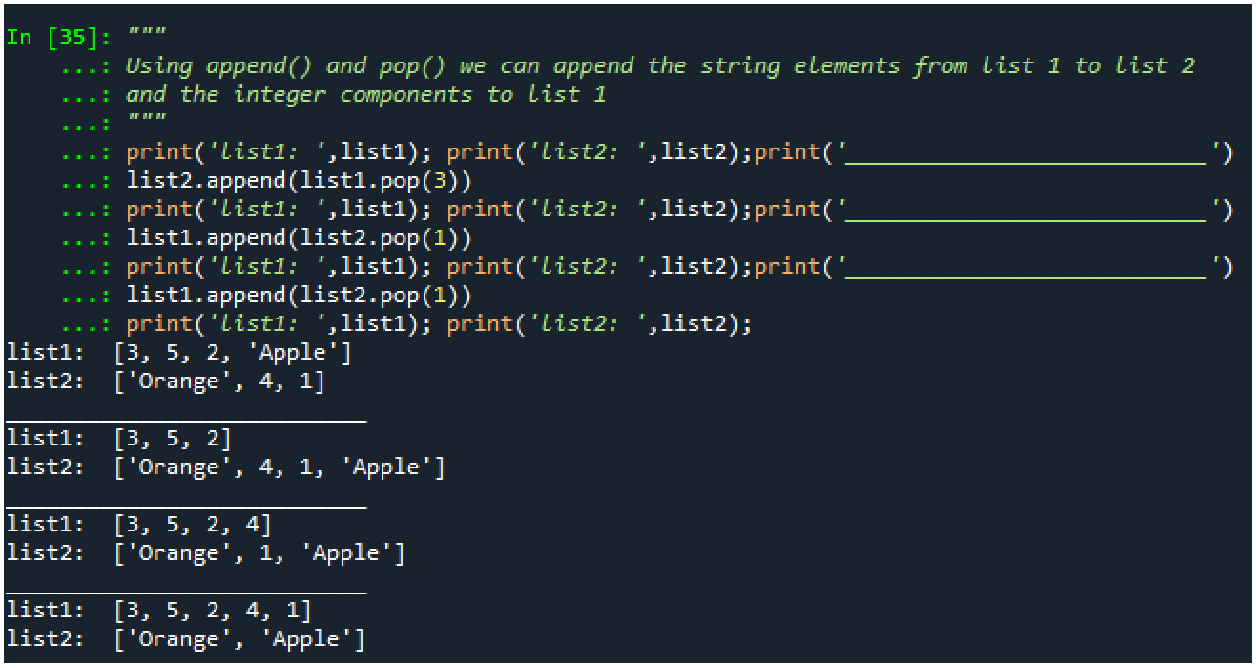
Python makes it easy to combine lists, and similarly other objects like strings with the `+` operator.



```
In [25]:
    ...: list3 = list1 + list2
    ...: print(list3)
[3, 5, 2, 'Apple', 'Orange', 4, 1]
```

Adding and Removing List Elements

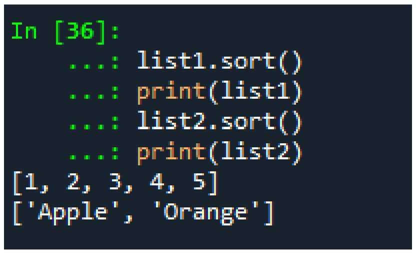
The append() method allows you to add elements to a list. The pop() method allows you to remove elements from a list by its index value and then returns the item in that index position. This is very useful for mixed object lists as shown in the below example.



```
In [35]: """
    ...: Using append() and pop() we can append the string elements from list 1 to list 2
    ...: and the integer components to list 1
    ...: """
    ...: print('list1: ',list1); print('list2: ',list2);print('___________________________')
    ...: list2.append(list1.pop(3))
    ...: print('list1: ',list1); print('list2: ',list2);print('___________________________')
    ...: list1.append(list2.pop(1))
    ...: print('list1: ',list1); print('list2: ',list2);print('___________________________')
    ...: list1.append(list2.pop(1))
    ...: print('list1: ',list1); print('list2: ',list2);
list1:  [3, 5, 2, 'Apple']
list2:  ['Orange', 4, 1]
___________________________
list1:  [3, 5, 2]
list2:  ['Orange', 4, 1, 'Apple']
___________________________
list1:  [3, 5, 2, 4]
list2:  ['Orange', 1, 'Apple']
___________________________
list1:  [3, 5, 2, 4, 1]
list2:  ['Orange', 'Apple']
```

Sorting Lists

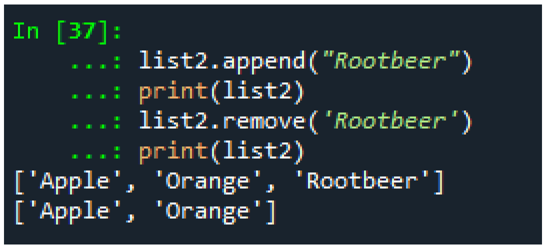
Once your lists are all of the same type you can use the sort method to re-arrange the values. By default, this works in ascending order but you can change that by specifying an argument. See the documentation for details.



```
In [36]:
    ...: list1.sort()
    ...: print(list1)
    ...: list2.sort()
    ...: print(list2)
[1, 2, 3, 4, 5]
['Apple', 'Orange']
```

Using Remove()

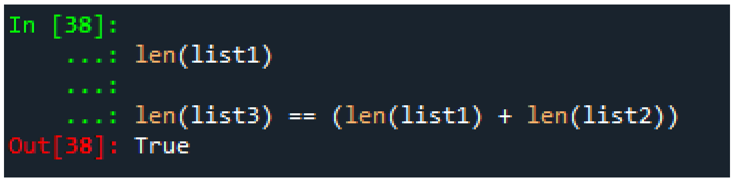
We’ve shown that you can remove list elements using pop() but sometimes you want to remove them by name because you may be multiple occurrences of an item.



```
In [37]:
    ...: list2.append("Rootbeer")
    ...: print(list2)
    ...: list2.remove('Rootbeer')
    ...: print(list2)
['Apple', 'Orange', 'Rootbeer']
['Apple', 'Orange']
```

List Length

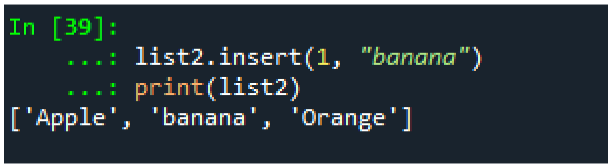
Finding the length of lists is especially useful for iterating over lists or comparing the size of lists for unit tests or some other logical expression.



```
In [38]:
    ...: len(list1)
    ...:
    ...: len(list3) == (len(list1) + len(list2))
Out[38]: True
```

Inserting Elements at a Specific Index

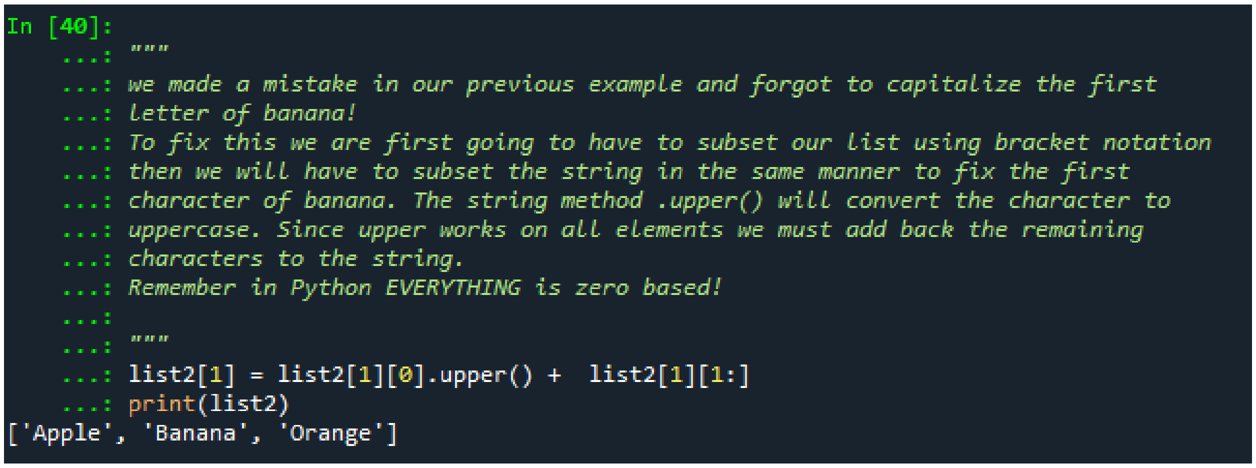
Append inserts list elements at the end or beginning of a list. But what if we wanted to insert an element in another location? The insert() method allows us to do that by providing an index, and the item to insert.



```
In [39]:
    ...: list2.insert(1, "banana")
    ...: print(list2)
['Apple', 'banana', 'Orange']
```

Subsetting Lists

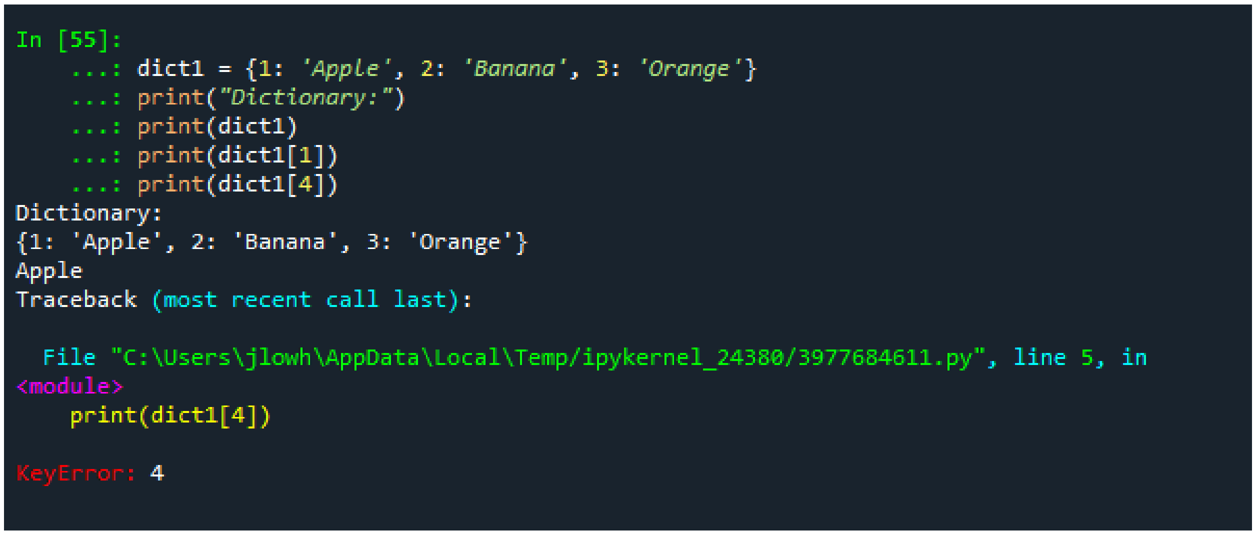
Subsetting lists in Python is identical to subsetting lists in R, the difference being the zero-based index. Python also uses the same bracket notation to allow you to subset strings which can be very powerful. In the below example we will correct the uncapitalize ‘b’ in the item we just added to our list in place.



```
In [40]:
    ...: """
    ...: we made a mistake in our previous example and forgot to capitalize the first
    ...: letter of banana!
    ...: To fix this we are first going to have to subset our list using bracket notation
    ...: then we will have to subset the string in the same manner to fix the first
    ...: character of banana. The string method .upper() will convert the character to
    ...: uppercase. Since upper works on all elements we must add back the remaining
    ...: characters to the string.
    ...: Remember in Python EVERYTHING is zero based!
    ...:
    ...: """
    ...: list2[1] = list2[1][0].upper() +  list2[1][1:]
    ...: print(list2)
['Apple', 'Banana', 'Orange']
```

Dictionaries & Defaultdict

Dictionaries are one of the most powerful data structures in Python. They are a key-value pair-based structure that is easily serializable to JSON (JavaScript Object Notation), a data format that is widely used in big data and web-based applications. In a Dictionary, the key must be unique and immutable. This means that a Python Tuple can be a key whereas a Python List cannot. A Dictionary can be created by placing a sequence of elements within curly {} braces, separated by ‘comma’.

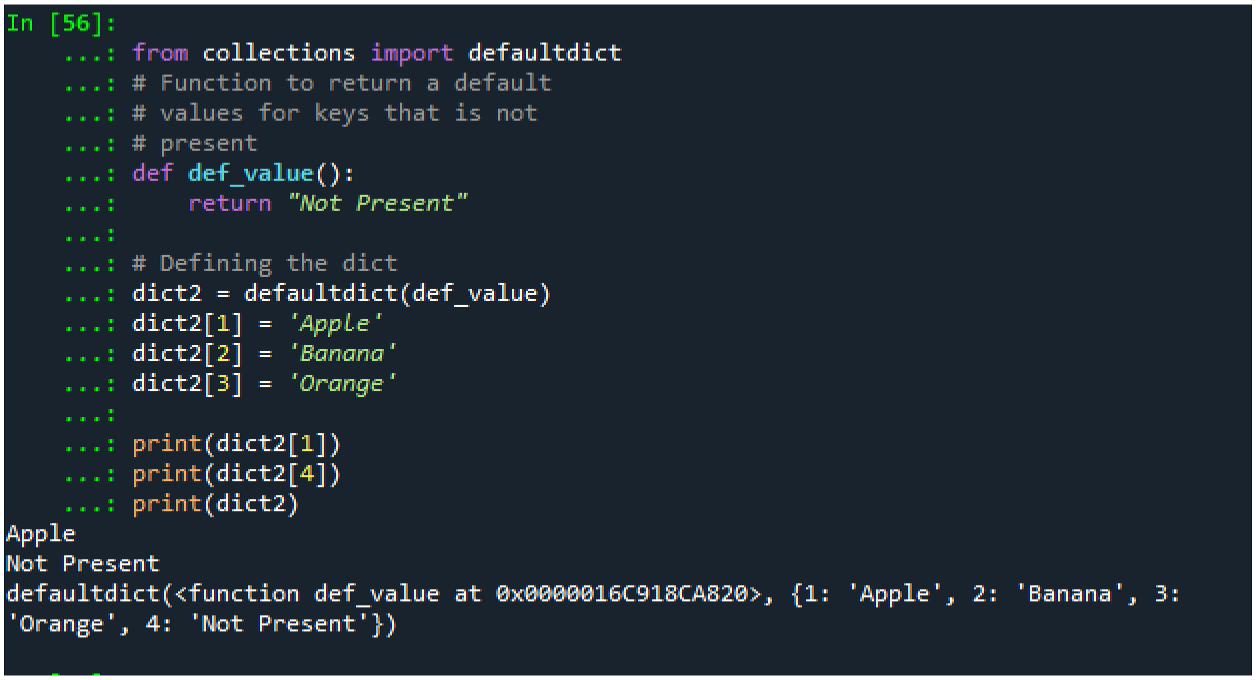


```
In [55]:
    ...: dict1 = {1: 'Apple', 2: 'Banana', 3: 'Orange'}
    ...: print("Dictionary:")
    ...: print(dict1)
    ...: print(dict1[1])
    ...: print(dict1[4])
Dictionary:
{1: 'Apple', 2: 'Banana', 3: 'Orange'}
Apple
Traceback (most recent call last):

  File "C:\Users\jlowh\AppData\Local\Temp/ipykernel_24380/3977684611.py", line 5, in
<module>
    print(dict1[4])

KeyError: 4
```

Notice in that example that we received an error that the key 4 was not present in our dictionary. For some scenarios that may become a problem (hashing). To overcome this Python introduces another dictionary like container known as Defaultdict which is present inside the collections module.

Defaultdict

Defaultdict is a container like dictionaries present in the module collections. Defaultdict is a sub-class of the dictionary class that returns a dictionary-like object. The functionality of both dictionaries and defaultdict are almost same except for the fact that defaultdict never raises a KeyError. It provides a default value for the key that does not exists which means that you can use defaultdict for hash tables.

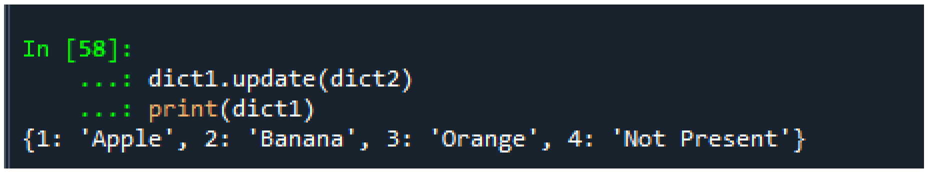

```
In [56]:
    ...: from collections import defaultdict
    ...: # Function to return a default
    ...: # values for keys that is not
    ...: # present
    ...: def def_value():
    ...:     return "Not Present"
    ...:
    ...: # Defining the dict
    ...: dict2 = defaultdict(def_value)
    ...: dict2[1] = 'Apple'
    ...: dict2[2] = 'Banana'
    ...: dict2[3] = 'Orange'
    ...:
    ...: print(dict2[1])
    ...: print(dict2[4])
    ...: print(dict2)
Apple
Not Present
defaultdict(<function def_value at 0x0000016C918CA820>, {1: 'Apple', 2: 'Banana', 3:
'Orange', 4: 'Not Present'})
```

Merging dictionaries with the .update() method in Python

Given two dictionaries that need to be combined, Python makes this easy with the .update() function.

For dict1.update(dict2), the key-value pairs of dict2 will be written into the dict1 dictionary.

For keys in both dict1 and dict2, the value in dict1 will be overwritten by the corresponding value in dict2.



```
In [58]:
    ...: dict1.update(dict2)
    ...: print(dict1)
{1: 'Apple', 2: 'Banana', 3: 'Orange', 4: 'Not Present'}
```

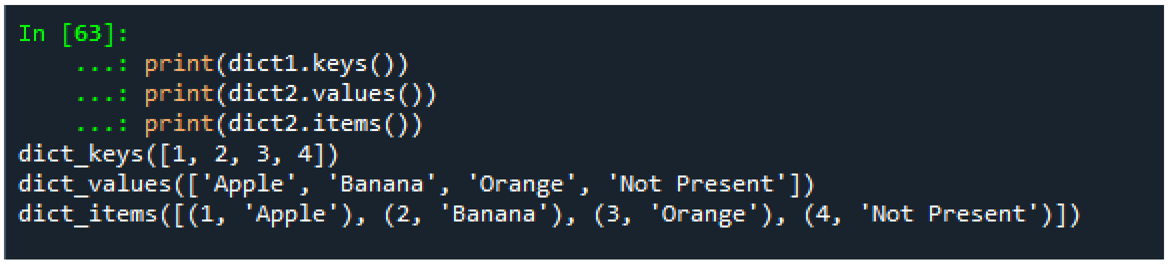
Dictionary Key-Value Methods

When trying to look at the information in a Python dictionary, there are multiple methods that return objects that contain the dictionary keys and values.

.keys() returns the keys through a dict_keys object.

.values() returns the values through a dict_values object.

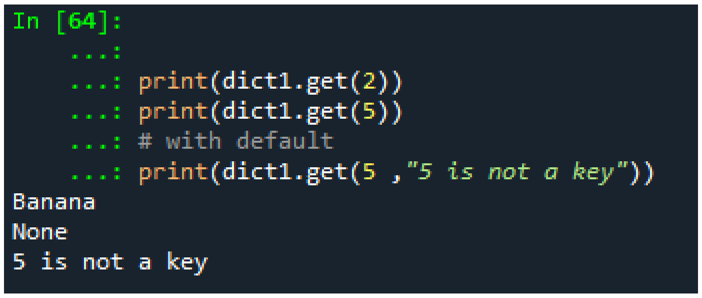
.items() returns both the keys and values through a dict_items object.



```
In [63]:
    ...: print(dict1.keys())
    ...: print(dict2.values())
    ...: print(dict2.items())
dict_keys([1, 2, 3, 4])
dict_values(['Apple', 'Banana', 'Orange', 'Not Present'])
dict_items([(1, 'Apple'), (2, 'Banana'), (3, 'Orange'), (4, 'Not Present')])
```

get() Method for Dictionary

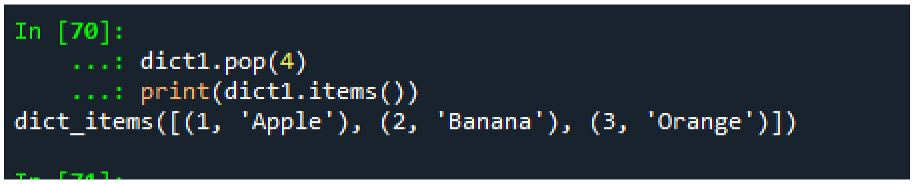
Python provides a **.get()** method to access a dictionary value if it exists. This method takes the key as the first argument and an optional default value as the second argument, and it returns the value for the specified key if key is in the dictionary. If the second argument is not specified and key is not found, then None is returned.



```
In [64]:
    ...:
    ...: print(dict1.get(2))
    ...: print(dict1.get(5))
    ...: # with default
    ...: print(dict1.get(5 ,"5 is not a key"))
Banana
None
5 is not a key
```

The .pop() Method for Dictionaries in Python

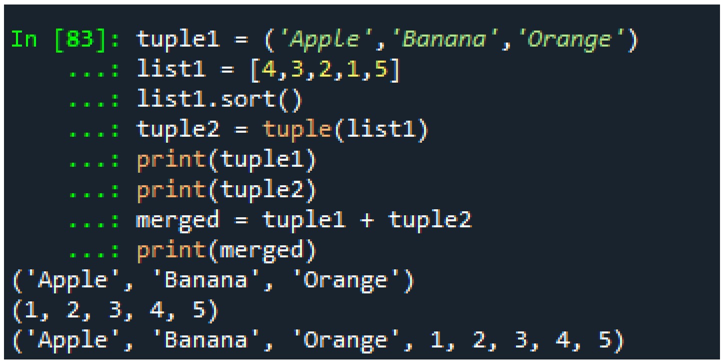
Python dictionaries can remove key-value pairs with the **.pop()** method. The method takes a key as an argument and removes it from the dictionary. At the same time, it also returns the value that it removes from the dictionary.



```
In [70]:
    ...: dict1.pop(4)
    ...: print(dict1.items())
dict_items([(1, 'Apple'), (2, 'Banana'), (3, 'Orange')])
```

Tuples

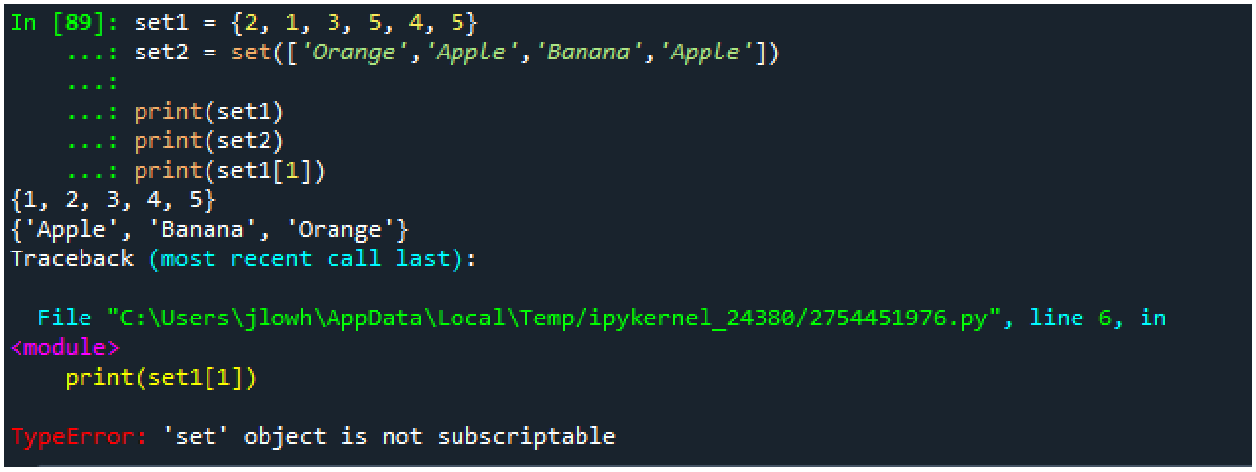
Tuples are used to store multiple items in a single variable. A tuple is a collection which is *ordered* and *immutable*. Tuples are written with round brackets. Since they are immutable there are no methods to be covered (.pop(), .append(), etc) aside from merging tuples. So why would you use a tuple and not a list? Well, if you want some data that isn't changeable or doesn't need to have its order changed, you could use a tuple. And when we talk about larger amounts of data, python executes tuples much faster.



```
In [83]: tuple1 = ('Apple','Banana','Orange')
    ...: list1 = [4,3,2,1,5]
    ...: list1.sort()
    ...: tuple2 = tuple(list1)
    ...: print(tuple1)
    ...: print(tuple2)
    ...: merged = tuple1 + tuple2
    ...: print(merged)
('Apple', 'Banana', 'Orange')
(1, 2, 3, 4, 5)
('Apple', 'Banana', 'Orange', 1, 2, 3, 4, 5)
```

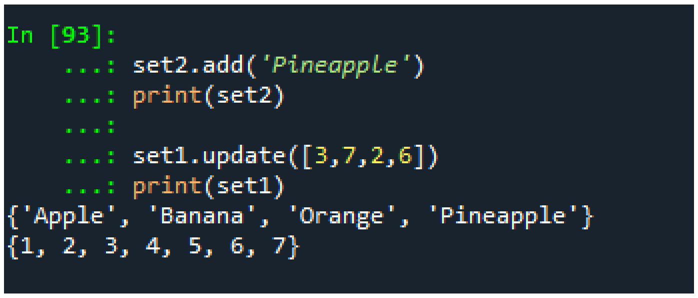
Sets

A set is an unordered collection with no duplicate elements. By unordered we mean it cannot be indexed. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference. A practical use for a set is to quickly find all of the unique items of a list.


```
In [89]: set1 = {2, 1, 3, 5, 4, 5}
    ...: set2 = set(['Orange','Apple','Banana','Apple'])
    ...:
    ...: print(set1)
    ...: print(set2)
    ...: print(set1[1])
{1, 2, 3, 4, 5}
{'Apple', 'Banana', 'Orange'}
Traceback (most recent call last):

  File "C:\Users\jlowh\AppData\Local\Temp/ipykernel_24380/2754451976.py", line 6, in
<module>
    print(set1[1])

TypeError: 'set' object is not subscriptable
```

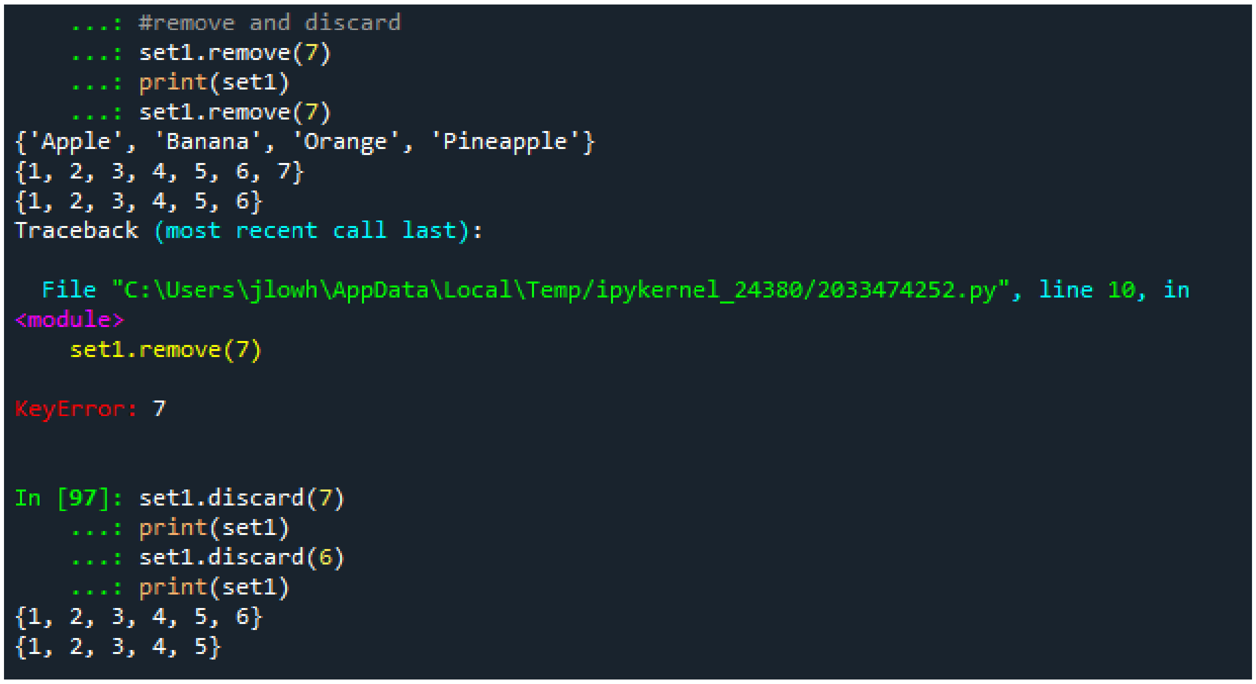
Set add() and update()

Using the add() method we can add a single element to the set. With update() we can add multiple elements.


```
In [93]:
    ...: set2.add('Pineapple')
    ...: print(set2)
    ...:
    ...: set1.update([3,7,2,6])
    ...: print(set1)
{'Apple', 'Banana', 'Orange', 'Pineapple'}
{1, 2, 3, 4, 5, 6, 7}
```

Set remove() and discard()

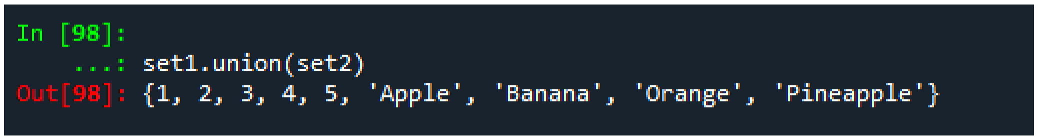
Both methods will remove an element from the set but remove() will raise a key error if the value doesn't exist. Discard() will not raise any errors.



```
    ...: #remove and discard
    ...: set1.remove(7)
    ...: print(set1)
    ...: set1.remove(7)
{'Apple', 'Banana', 'Orange', 'Pineapple'}
{1, 2, 3, 4, 5, 6, 7}
{1, 2, 3, 4, 5, 6}
Traceback (most recent call last):

  File "C:\Users\jlowh\AppData\Local\Temp/ipykernel_24380/2033474252.py", line 10, in
<module>
    set1.remove(7)

KeyError: 7


In [97]: set1.discard(7)
    ...: print(set1)
    ...: set1.discard(6)
    ...: print(set1)
{1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5}
```

set union()

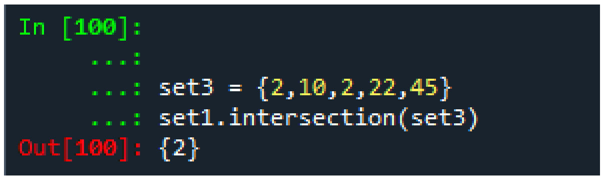
union() or | will create a new set that contains all the elements from the sets provided.



```
In [98]:
    ...: set1.union(set2)
Out[98]: {1, 2, 3, 4, 5, 'Apple', 'Banana', 'Orange', 'Pineapple'}
```

set intersection

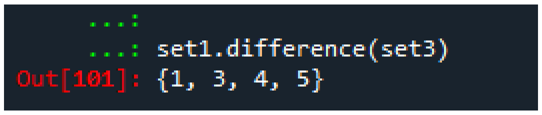
Intersection or & will return a set containing only the elements that are common to all of them.



```
In [100]:
     ...:
     ...: set3 = {2,10,2,22,45}
     ...: set1.intersection(set3)
Out[100]: {2}
```

set difference

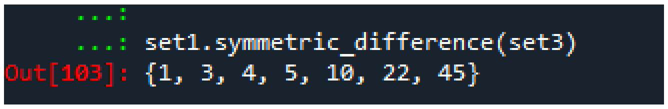
difference or - will return only the elements that are unique to the first set (invoked set).



```
      ...:
      ...: set1.difference(set3)
Out[101]: {1, 3, 4, 5}
```

set symmetric_difference

symmetric_difference or ^ will return all the elements that are not common between them.



```
      ...:
      ...: set1.symmetric_difference(set3)
Out[103]: {1, 3, 4, 5, 10, 22, 45}
```