Data Structures in Python

Module 2

Outline

1. Data structures in Python
    - List
    - Tuple
    - Dictionary
    - Set
    - Arrays
2. Conversions of Data Structures

Data Structures

- **Data Structures** are a way of organizing data so that it can be accessed more efficiently depending upon the situation.
- Data Structures are fundamentals of any programming language around which a program is built.
- Built in Data Structures in Python are:
  - List
  - Tuple
  - Dictionary
  - Set

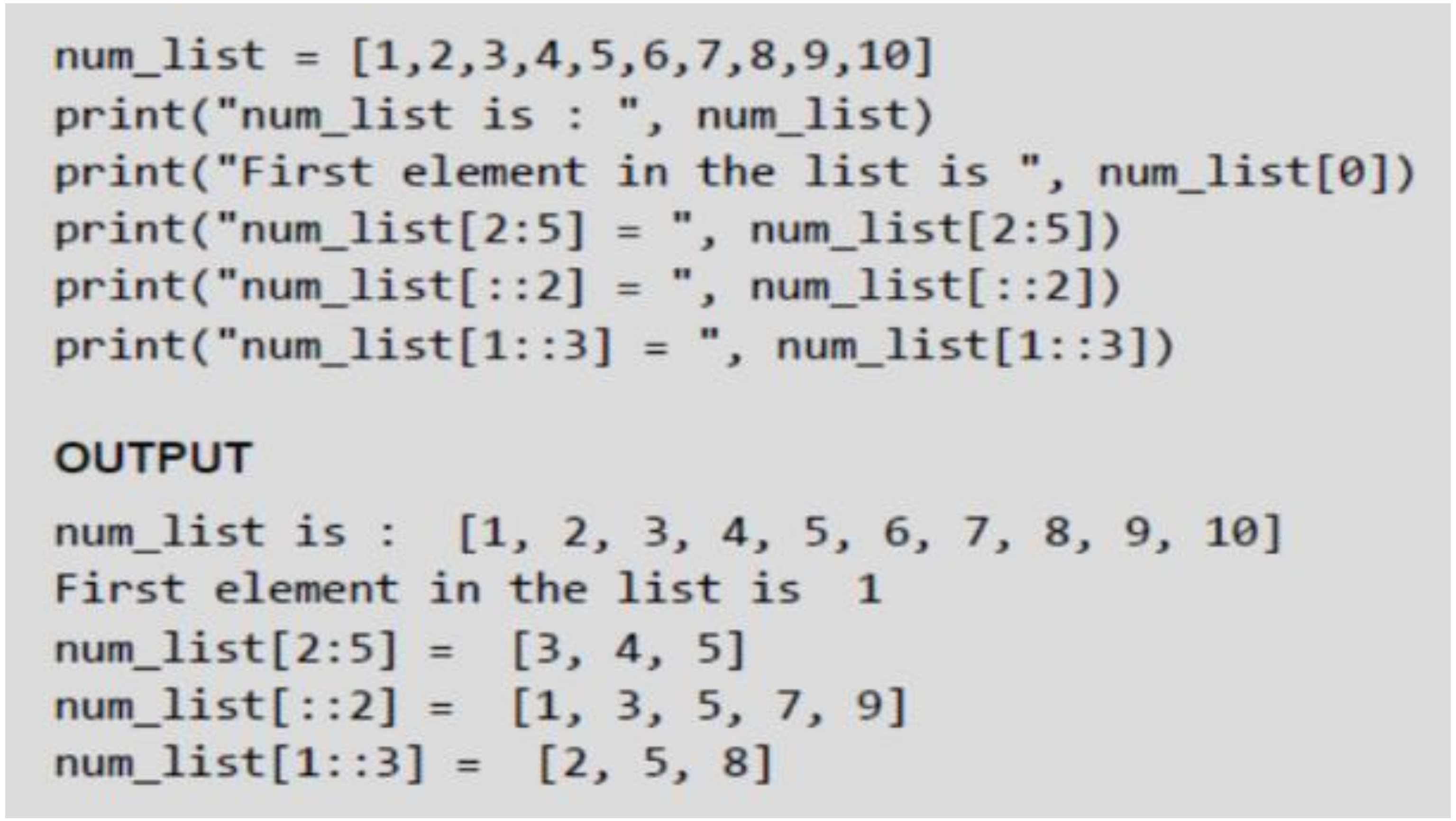
List

- The Python List is an **ordered collection (also known as a sequence) of elements**. List elements can be accessed, iterated, and removed according to the order they inserted at the creation time.
- We use the list data type to represent groups of the element as a single entity. For example: If we want to store all student's names, we can use list type.
- The list can contain data of all data types such as int, float, string
- Duplicates elements are allowed in the list
- The list is mutable which means we can modify the value of list elements
- We can create a list using the two ways:

1. By enclosing elements in the **square brackets []**.
2. Using a list() class.

Access Values in Lists

- Similar to strings, lists can also be sliced and concatenated.
- To access values in lists, square brackets are used to slice along with the index or indices to get value stored at that index.
- The syntax for the slice operation is given as, seq = List[start:stop:step]



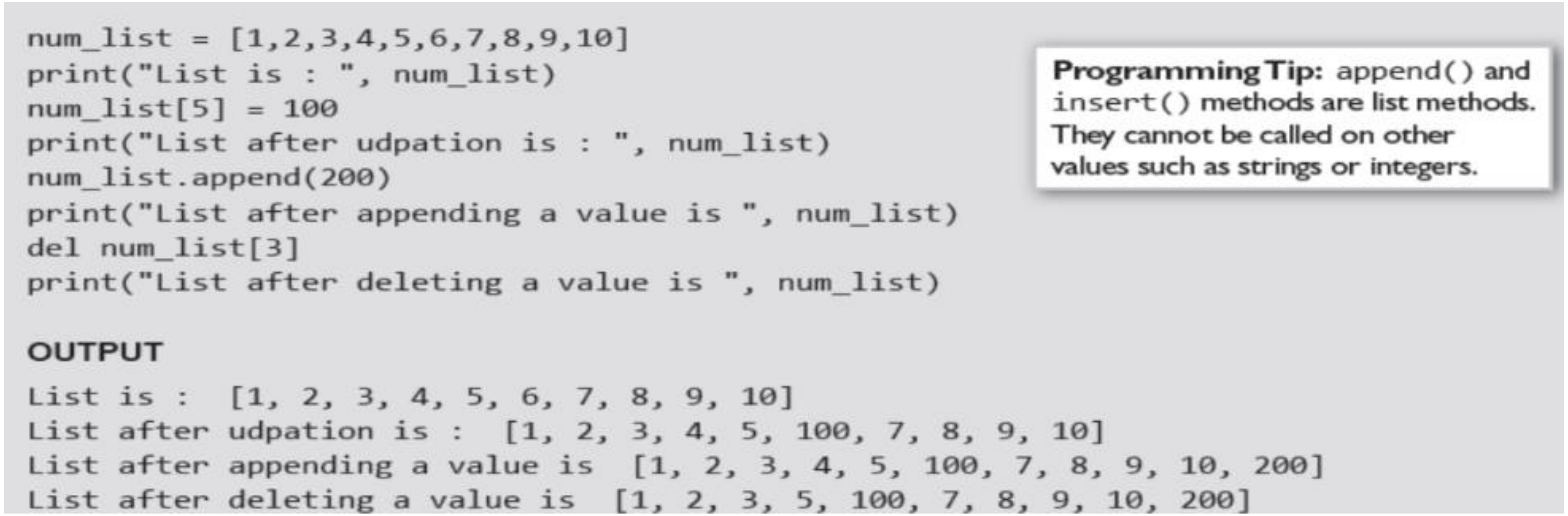
```
num_list = [1,2,3,4,5,6,7,8,9,10]
print("num_list is : ", num_list)
print("First element in the list is ", num_list[0])
print("num_list[2:5] = ", num_list[2:5])
print("num_list[::2] = ", num_list[::2])
print("num_list[1::3] = ", num_list[1::3])
```

OUTPUT

```
num_list is :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
First element in the list is  1
num_list[2:5] =  [3, 4, 5]
num_list[::2] =  [1, 3, 5, 7, 9]
num_list[1::3] =  [2, 5, 8]
```

Updating Values in Lists

- Once created, one or more elements of a list can be easily updated by giving the slice on the left-hand side of the assignment operator.
- You can also append new values in the list and remove existing value(s) from the list using the append() method and del statement respectively.



```
num_list = [1,2,3,4,5,6,7,8,9,10]
print("List is : ", num_list)
num_list[5] = 100
print("List after udpation is : ", num_list)
num_list.append(200)
print("List after appending a value is ", num_list)
del num_list[3]
print("List after deleting a value is ", num_list)
```

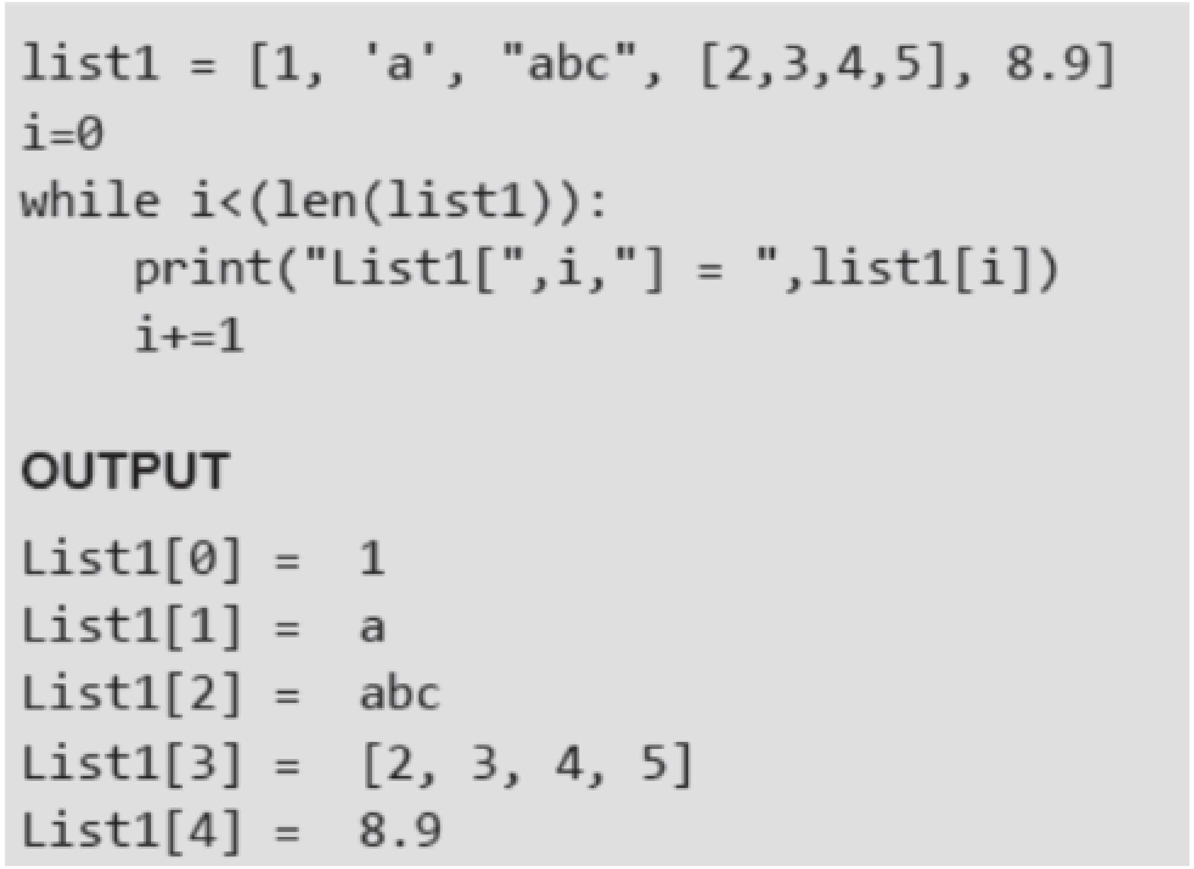
Programming Tip: append() and insert() methods are list methods. They cannot be called on other values such as strings or integers.

OUTPUT

```
List is :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
List after udpation is :  [1, 2, 3, 4, 5, 100, 7, 8, 9, 10]
List after appending a value is  [1, 2, 3, 4, 5, 100, 7, 8, 9, 10, 200]
List after deleting a value is  [1, 2, 3, 5, 100, 7, 8, 9, 10, 200]
```

Nested Lists

- Nested list means a list within another list.
- A list has elements of different data types which can include even a list.



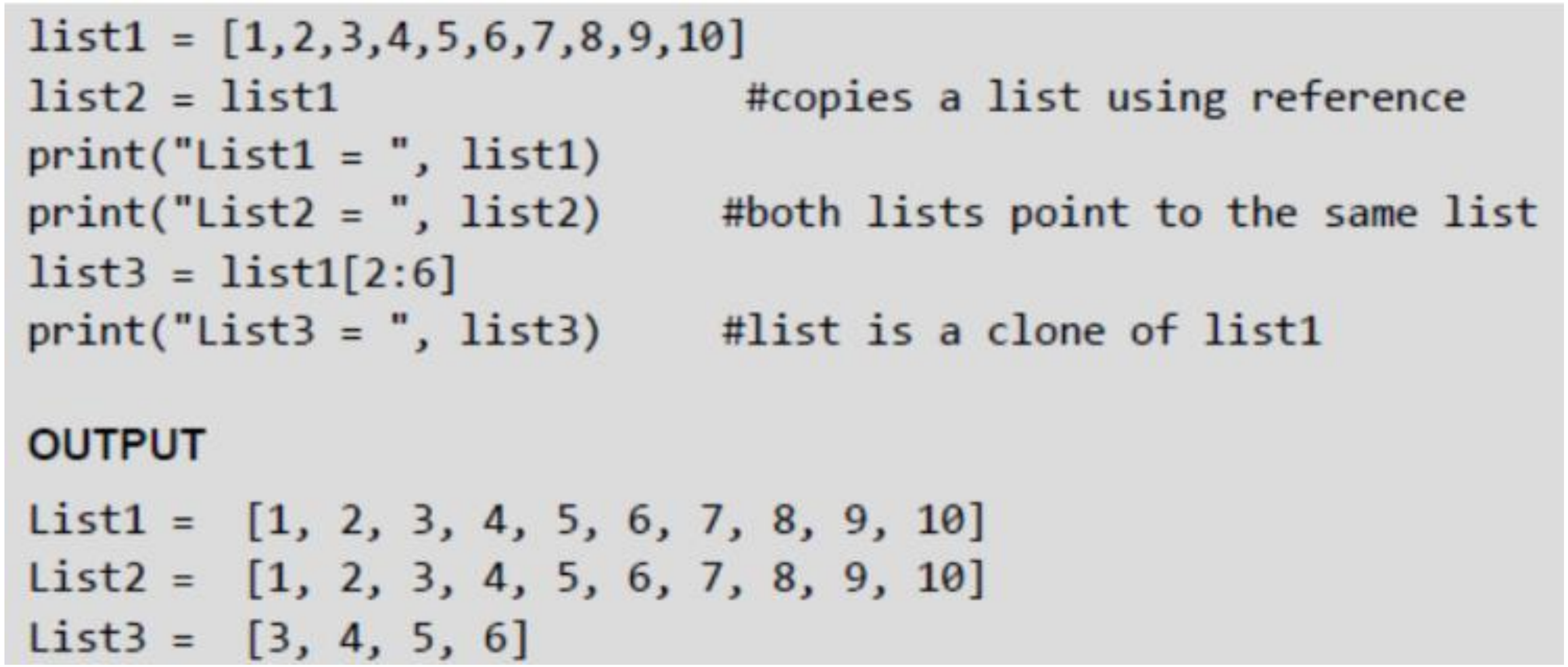
```
list1 = [1, 'a', "abc", [2,3,4,5], 8.9]
i=0
while i<(len(list1)):
    print("List1[",i,"] = ",list1[i])
    i+=1
```

OUTPUT

```
List1[0] =  1
List1[1] =  a
List1[2] =  abc
List1[3] =  [2, 3, 4, 5]
List1[4] =  8.9
```

Cloning Lists

- If you want to modify a list and also keep a copy of the original list, then you should create a separate copy of the list (not just the reference). This process is called cloning.
- The slice operation is used to clone a list.

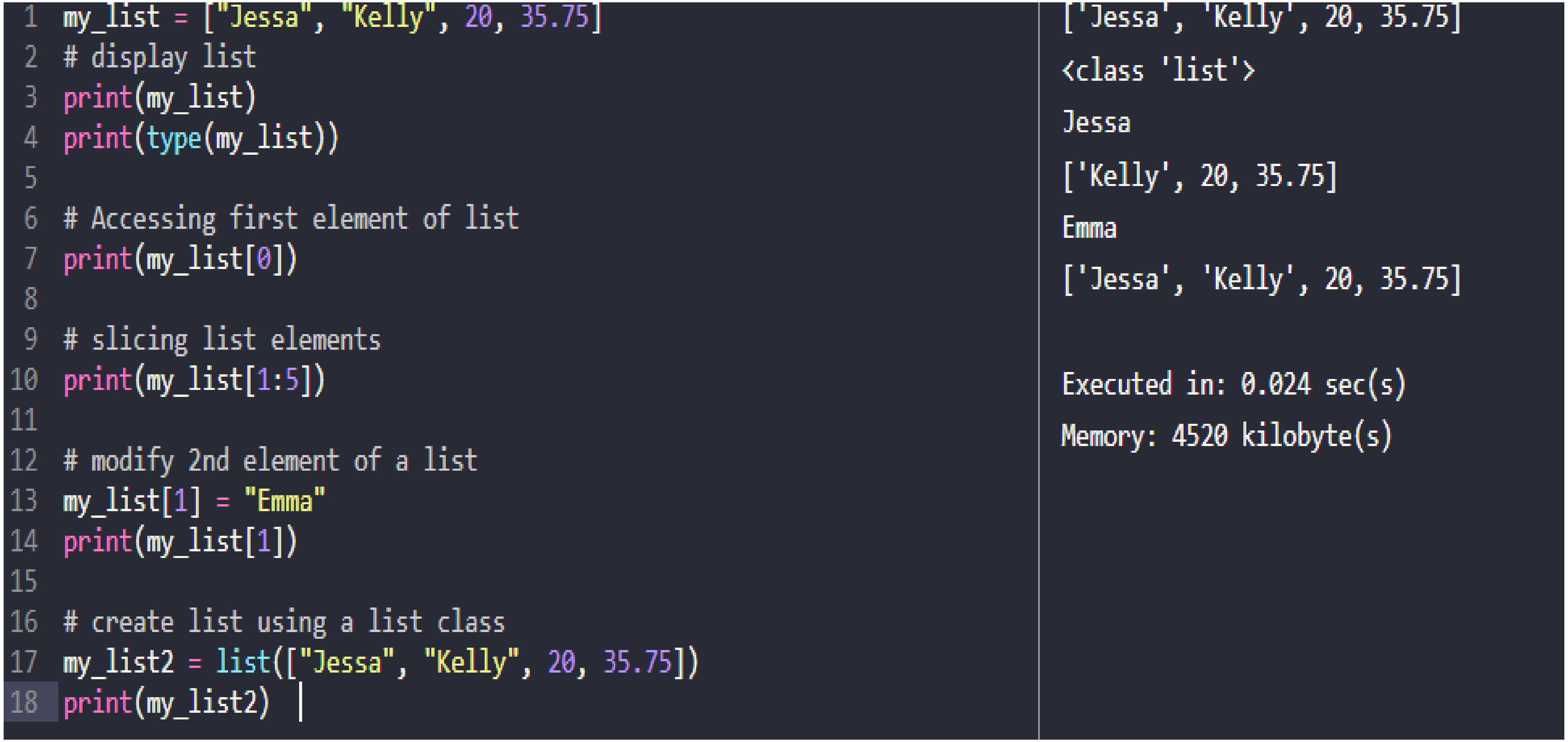


```
list1 = [1,2,3,4,5,6,7,8,9,10]
list2 = list1                      #copies a list using reference
print("List1 = ", list1)
print("List2 = ", list2)     #both lists point to the same list
list3 = list1[2:6]
print("List3 = ", list3)     #list is a clone of list1
```

OUTPUT

```
List1 =  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
List2 =  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
List3 =  [3, 4, 5, 6]
```

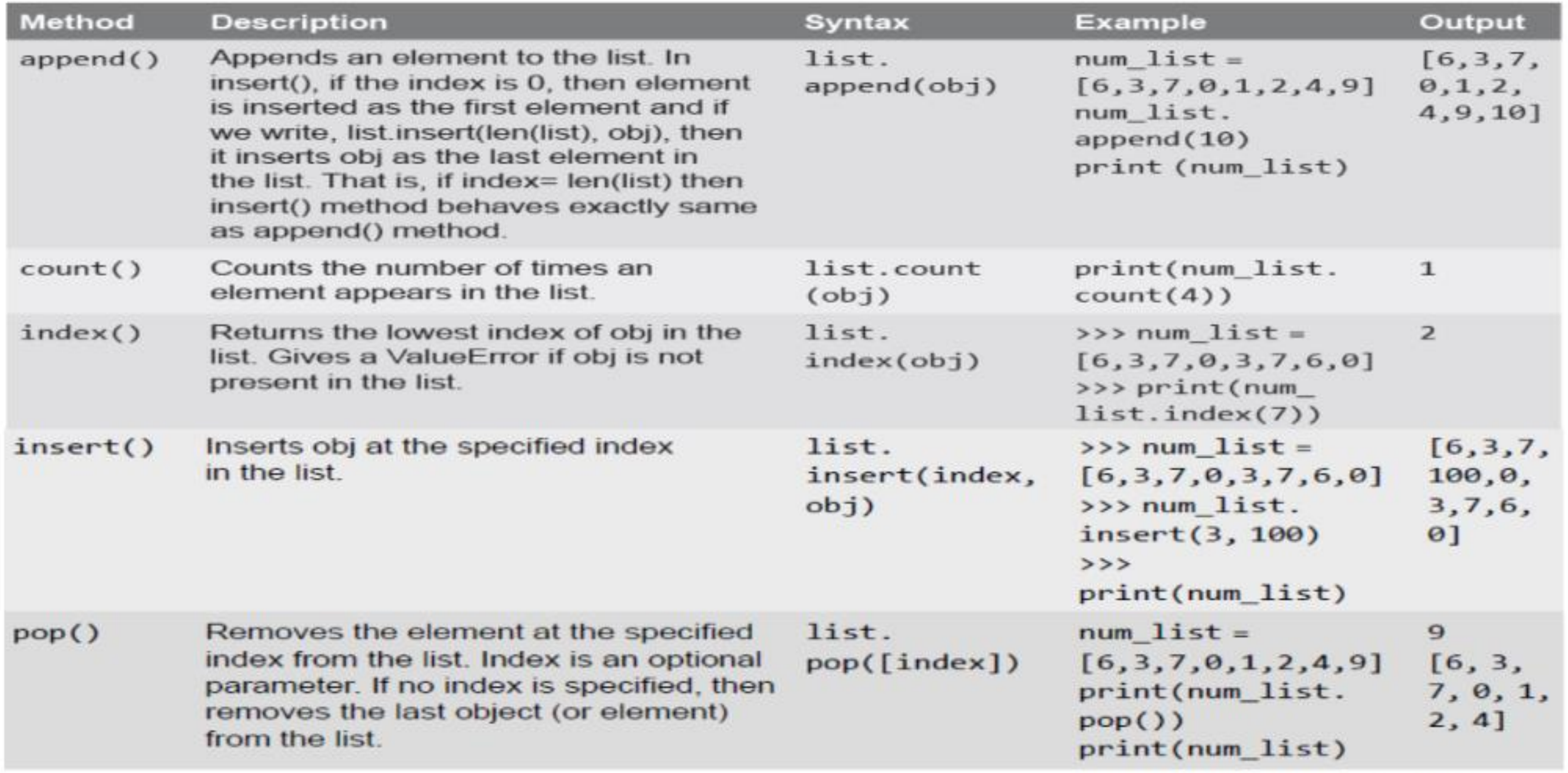
Example : List



```python
my_list = ["Jessa", "Kelly", 20, 35.75]
# display list
print(my_list)
print(type(my_list))

# Accessing first element of list
print(my_list[0])

# slicing list elements
print(my_list[1:5])

# modify 2nd element of a list
my_list[1] = "Emma"
print(my_list[1])

# create list using a list class
my_list2 = list(["Jessa", "Kelly", 20, 35.75])
print(my_list2)
```

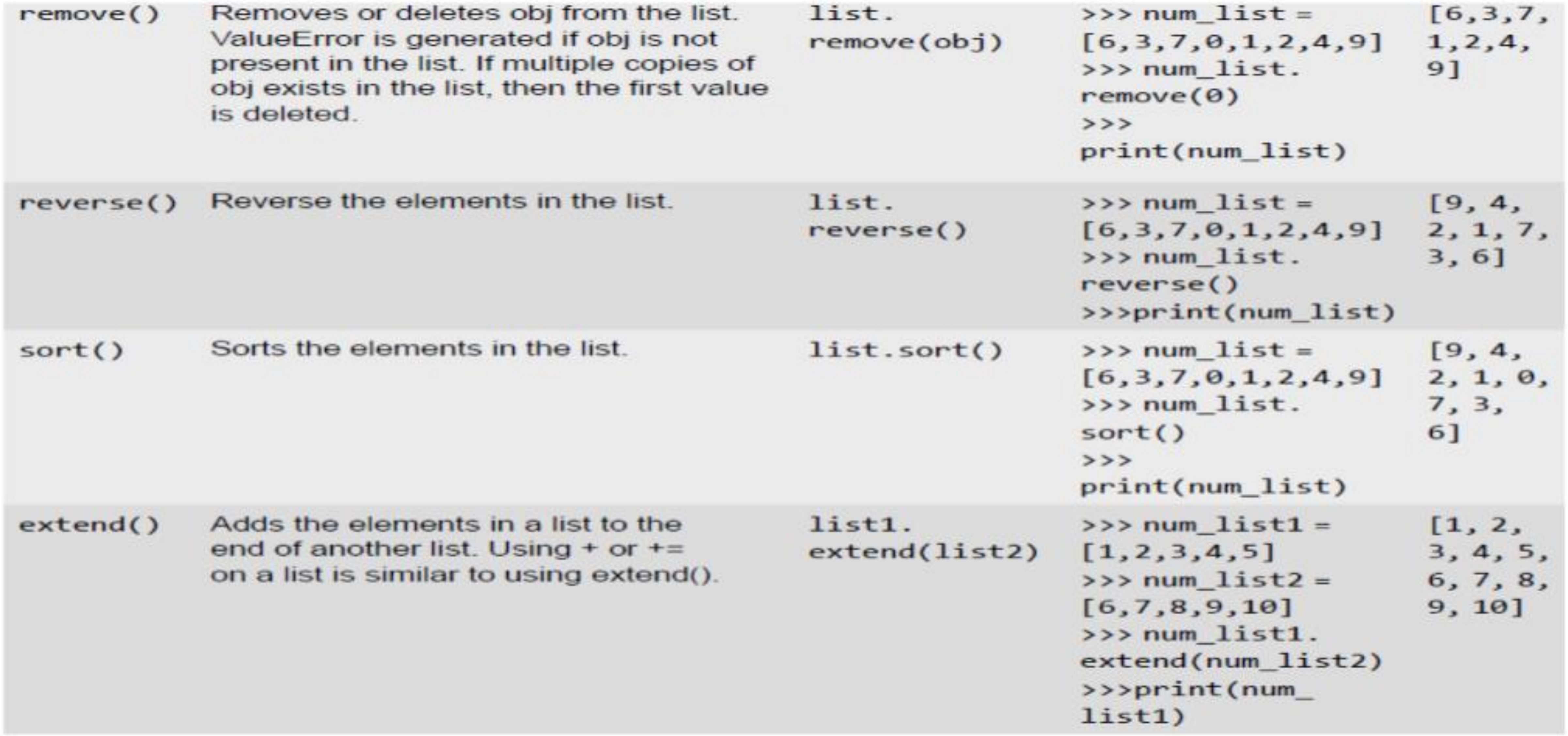
```
['Jessa', 'Kelly', 20, 35.75]
<class 'list'>
Jessa
['Kelly', 20, 35.75]
Emma
['Jessa', 'Kelly', 20, 35.75]


Executed in: 0.024 sec(s)
Memory: 4520 kilobyte(s)
```

Basic List Operations



Method	Description	Syntax	Example	Output
`append()`	Appends an element to the list. In insert(), if the index is 0, then element is inserted as the first element and if we write, list.insert(len(list), obj), then it inserts obj as the last element in the list. That is, if index= len(list) then insert() method behaves exactly same as append() method.	`list. append(obj)`	`num_list = [6,3,7,0,1,2,4,9] num_list. append(10) print (num_list)`	`[6,3,7, 0,1,2, 4,9,10]`
`count()`	Counts the number of times an element appears in the list.	`list.count (obj)`	`print(num_list. count(4))`	`1`
`index()`	Returns the lowest index of obj in the list. Gives a ValueError if obj is not present in the list.	`list. index(obj)`	`>>> num_list = [6,3,7,0,3,7,6,0] >>> print(num_ list.index(7))`	`2`
`insert()`	Inserts obj at the specified index in the list.	`list. insert(index, obj)`	`>>> num_list = [6,3,7,0,3,7,6,0] >>> num_list. insert(3, 100) >>> print(num_list)`	`[6,3,7, 100,0, 3,7,6, 0]`
`pop()`	Removes the element at the specified index from the list. Index is an optional parameter. If no index is specified, then removes the last object (or element) from the list.	`list. pop([index])`	`num_list = [6,3,7,0,1,2,4,9] print(num_list. pop()) print(num_list)`	`9 [6, 3, 7, 0, 1, 2, 4]`

List Methods



`remove()`	Removes or deletes obj from the list. ValueError is generated if obj is not present in the list. If multiple copies of obj exists in the list, then the first value is deleted.	`list. remove(obj)`	`>>> num_list = [6,3,7,0,1,2,4,9] >>> num_list. remove(0) >>> print(num_list)`	`[6,3,7, 1,2,4, 9]`
`reverse()`	Reverse the elements in the list.	`list. reverse()`	`>>> num_list = [6,3,7,0,1,2,4,9] >>> num_list. reverse() >>>print(num_list)`	`[9, 4, 2, 1, 7, 3, 6]`
`sort()`	Sorts the elements in the list.	`list.sort()`	`>>> num_list = [6,3,7,0,1,2,4,9] >>> num_list. sort() >>> print(num_list)`	`[9, 4, 2, 1, 0, 7, 3, 6]`
`extend()`	Adds the elements in a list to the end of another list. Using + or += on a list is similar to using extend().	`list1. extend(list2)`	`>>> num_list1 = [1,2,3,4,5] >>> num_list2 = [6,7,8,9,10] >>> num_list1. extend(num_list2) >>>print(num_ list1)`	`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

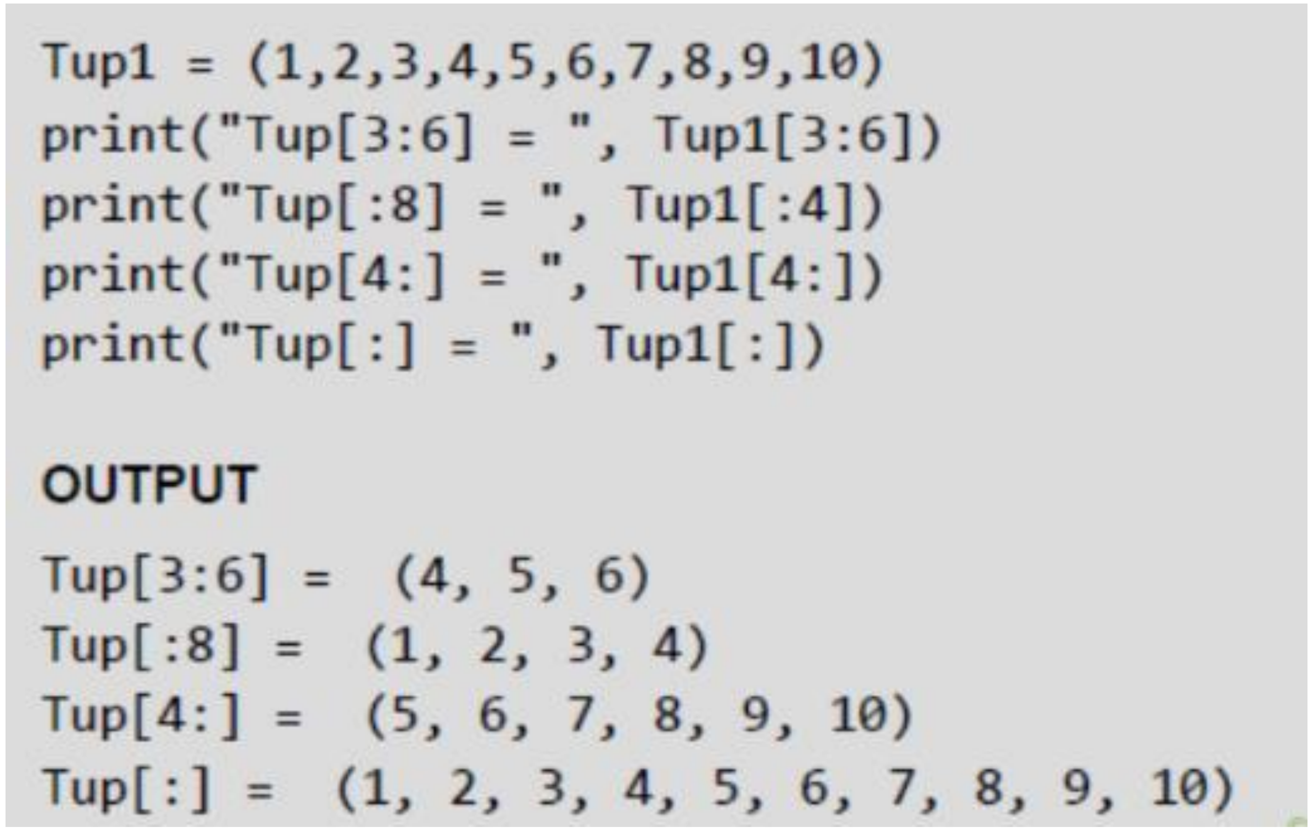
Tuple

- Tuples are **ordered collections of elements that are unchangeab**le. The tuple is the same as the list, except the tuple is immutable means we can’t modify the tuple once created.
- In other words, we can say a tuple is a read-only version of the list.
- For example: If you want to store the roll numbers of students that you don’t change, you can use the tuple data type.
- **Note**: Tuple maintains the insertion order and also, allows us to store duplicate elements.
- We can create a tuple using the two ways:

1. By enclosing elements in the parenthesis ()
2. Using a tuple() class.

Accessing Values in a Tuple

- Like other sequences (strings and lists) covered so far, indices in a tuple also starts at 0.
- You can even perform operations like slice, concatenate, etc. on a tuple. For example, to access values in tuple, slice operation is used along with the index or indices to obtain value stored at that index



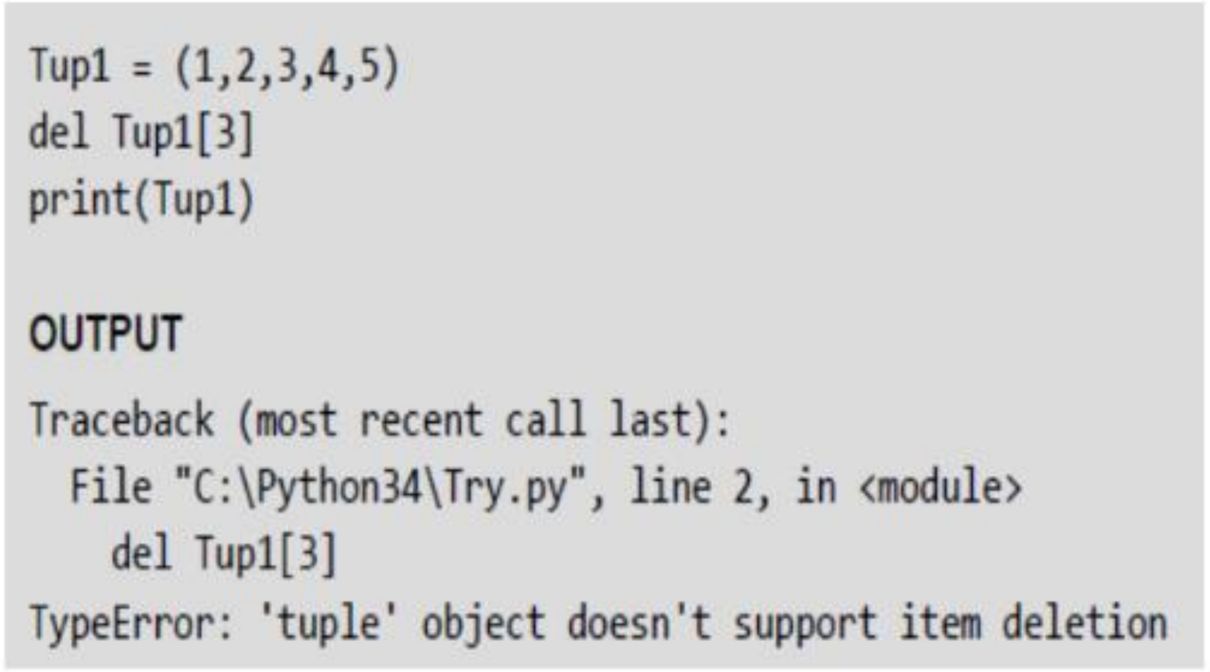
```
Tup1 = (1,2,3,4,5,6,7,8,9,10)
print("Tup[3:6] = ", Tup1[3:6])
print("Tup[:8] = ", Tup1[:4])
print("Tup[4:] = ", Tup1[4:])
print("Tup[:] = ", Tup1[:])
```

OUTPUT

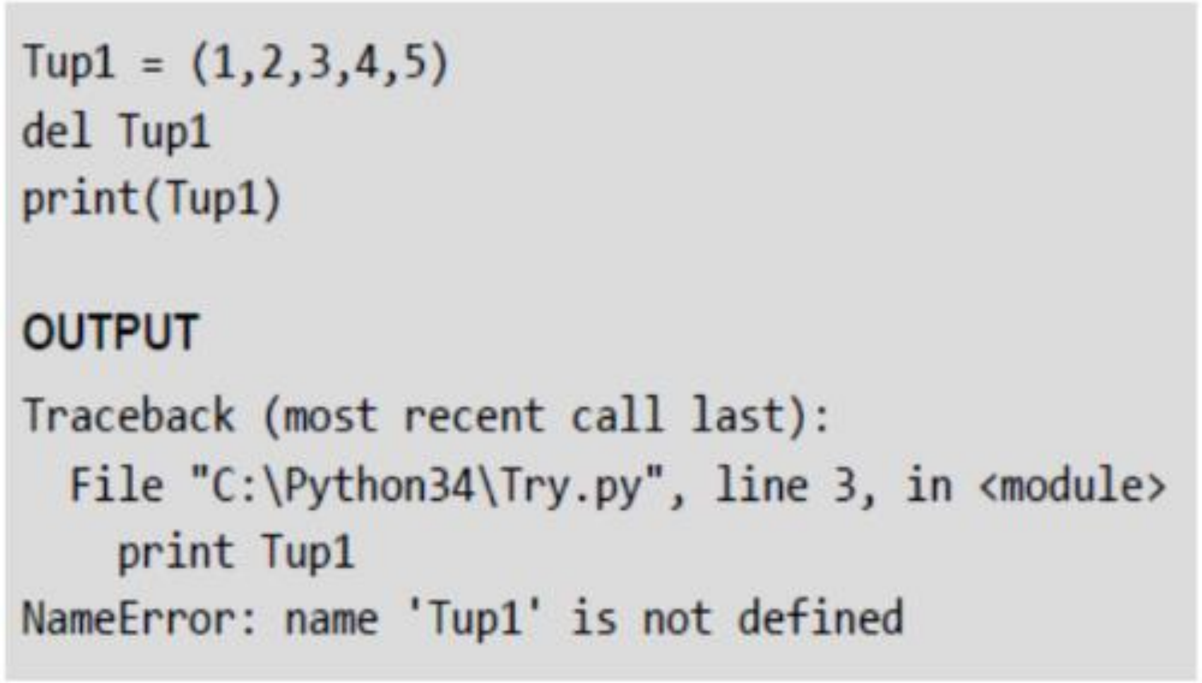
```
Tup[3:6] =  (4, 5, 6)
Tup[:8] =  (1, 2, 3, 4)
Tup[4:] =  (5, 6, 7, 8, 9, 10)
Tup[:] =  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Deleting Elements in Tuple

- Since tuple is an immutable data structure, you cannot delete value(s) from it.
- Of course, you can create a new tuple that has all elements in your tuple except the ones you don't want (those you wanted to be deleted).

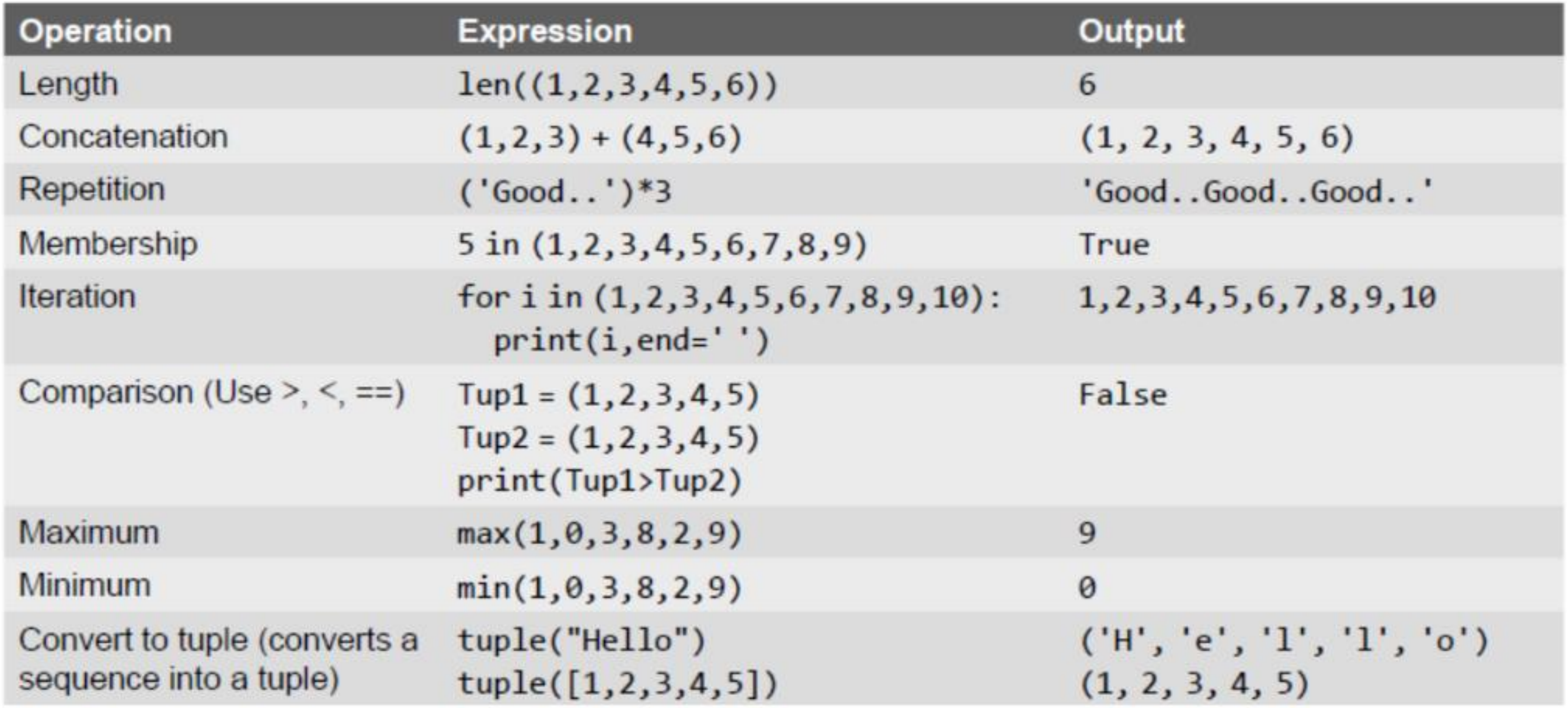


```
Tup1 = (1,2,3,4,5)
del Tup1[3]
print(Tup1)

OUTPUT
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 2, in <module>
    del Tup1[3]
TypeError: 'tuple' object doesn't support item deletion
```



```
Tup1 = (1,2,3,4,5)
del Tup1
print(Tup1)

OUTPUT
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 3, in <module>
    print Tup1
NameError: name 'Tup1' is not defined
```

Basic Tuple Operations



Operation	Expression	Output
Length	len((1,2,3,4,5,6))	6
Concatenation	(1,2,3) + (4,5,6)	(1, 2, 3, 4, 5, 6)
Repetition	('Good..')*3	'Good..Good..Good..'
Membership	5 in (1,2,3,4,5,6,7,8,9)	True

Iteration	for i in (1,2,3,4,5,6,7,8,9,10):
	  print(i,end=' ')
		1,2,3,4,5,6,7,8,9,10

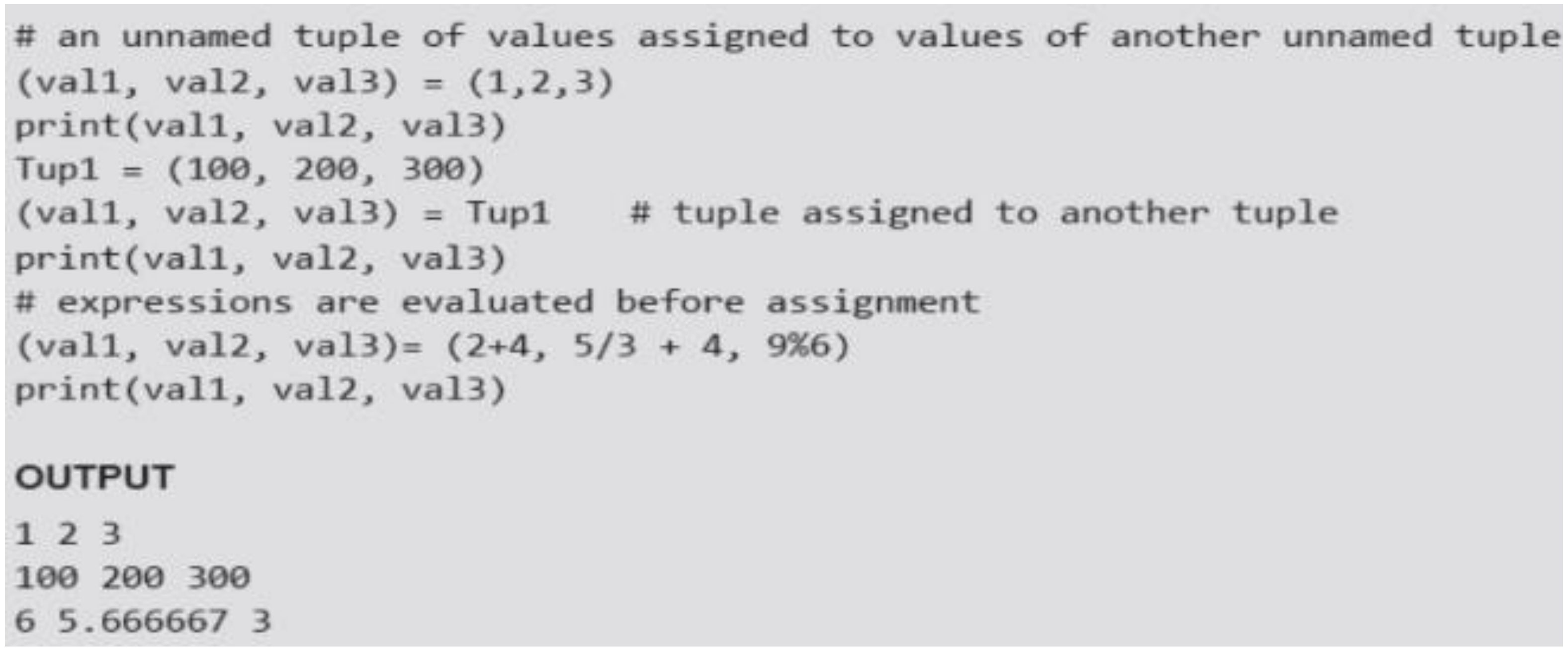
Comparison (Use >, <, ==)	Tup1 = (1,2,3,4,5)
	Tup2 = (1,2,3,4,5)
	print(Tup1>Tup2)
		False

Maximum	max(1,0,3,8,2,9)	9
Minimum	min(1,0,3,8,2,9)	0

Convert to tuple (converts a sequence into a tuple)	tuple("Hello")
	tuple([1,2,3,4,5])
		('H', 'e', 'l', 'l', 'o')
		(1, 2, 3, 4, 5)

Tuple Assignment

- Tuple assignment is a very powerful feature in Python.
- It allows a tuple of variables on the left side of the assignment operator to be assigned values from a tuple given on the right side of the assignment operator.
- Each value is assigned to its respective variable. In case, an expression is specified on the right side of the assignment operator, first that expression is evaluated and then assignment is done.

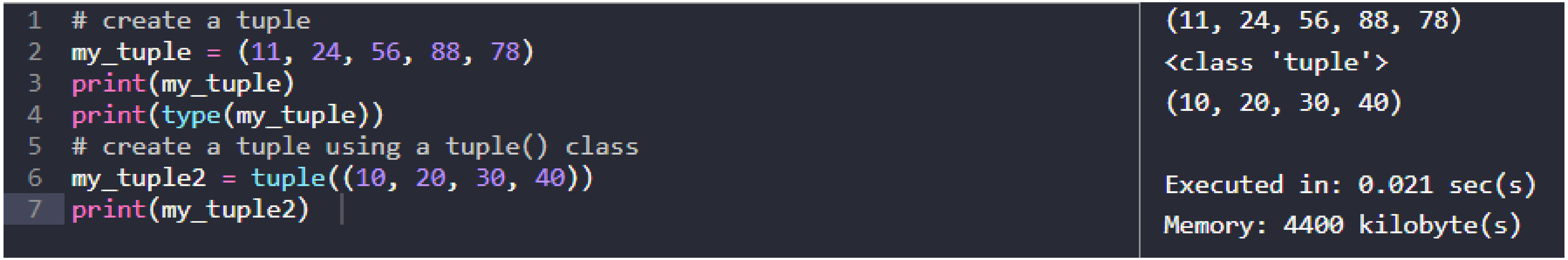


```
# an unnamed tuple of values assigned to values of another unnamed tuple
(val1, val2, val3) = (1,2,3)
print(val1, val2, val3)
Tup1 = (100, 200, 300)
(val1, val2, val3) = Tup1    # tuple assigned to another tuple
print(val1, val2, val3)
# expressions are evaluated before assignment
(val1, val2, val3)= (2+4, 5/3 + 4, 9%6)
print(val1, val2, val3)
```

OUTPUT

```
1 2 3
100 200 300
6 5.666667 3
```

Example: Tuple



```python
# create a tuple
my_tuple = (11, 24, 56, 88, 78)
print(my_tuple)
print(type(my_tuple))
# create a tuple using a tuple() class
my_tuple2 = tuple((10, 20, 30, 40))
print(my_tuple2)
```

```
(11, 24, 56, 88, 78)
<class 'tuple'>
(10, 20, 30, 40)


Executed in: 0.021 sec(s)
Memory: 4400 kilobyte(s)
```

Advantages of Tuple over List

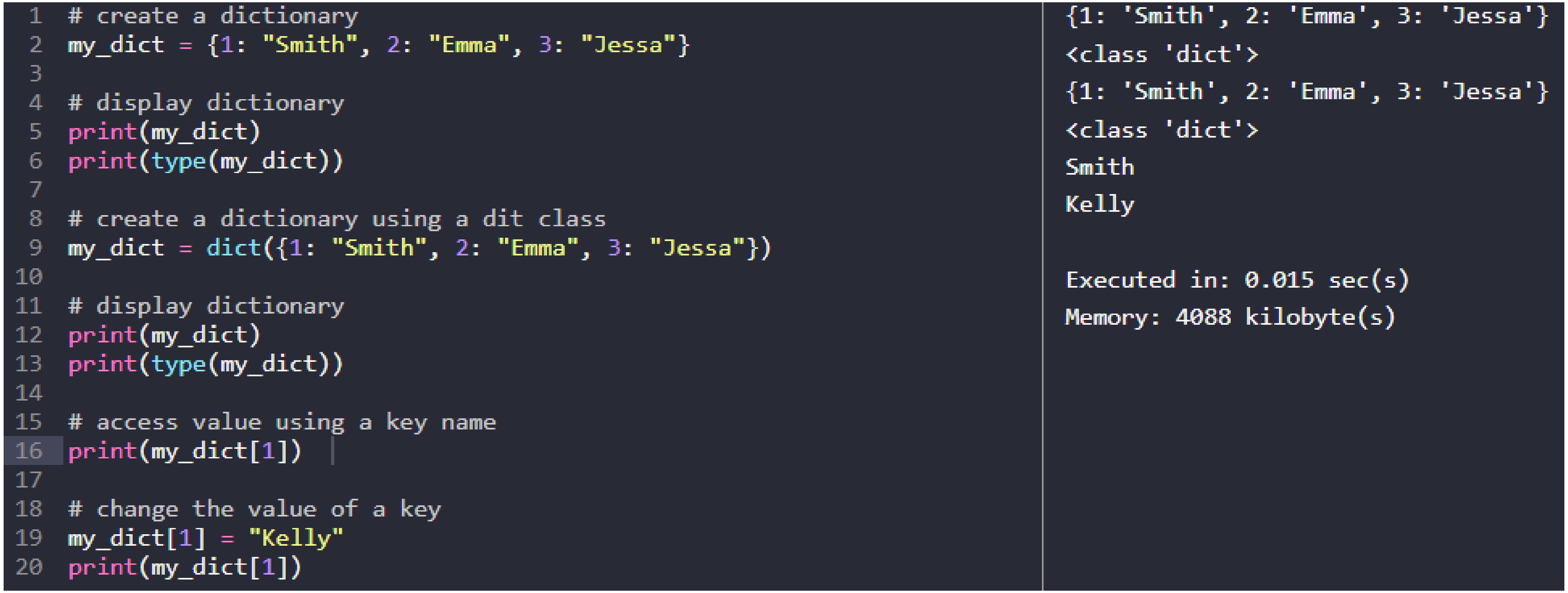
- Tuples are used to store values of different data types. Lists can however, store data of similar data types.
- Since tuples are immutable, iterating through tuples is faster than iterating over a list. This means that a tuple performs better than a list.
- Tuples can be used as key for a dictionary but lists cannot be used as keys.
- Tuples are best suited for storing data that is write-protected.
- Tuples can be used in place of lists where the number of values is known and small.
- If you are passing a tuple as an argument to a function, then the potential for unexpected behavior due to aliasing gets reduced.
- Multiple values from a function can be returned using a tuple.

Dictionary

- In Python, dictionaries are **unordered collections of unique values stored in (Key-Value) pairs**. Use a dictionary data type to store data as a key-value pair.
- The dictionary type is represented using a dict class. For example, If you want to store the name and roll number of all students, then you can use the dict type.
- In a dictionary, duplicate keys are not allowed, but the value can be duplicated. If we try to insert a value with a duplicate key, the old value will be replaced with the new value.
- Dictionary has some characteristics which are listed below:
- A heterogeneous (i.e., str, list, tuple) elements are allowed for both key and value in a dictionary. But An object can be a key in a dictionary if it is hashable.
- The dictionary is mutable which means we can modify its items
- Dictionary is unordered so we can’t perform indexing and slicing

- We can create a dictionary using the two ways
- By enclosing key and values in the curly brackets {}
- Using a dict() class.

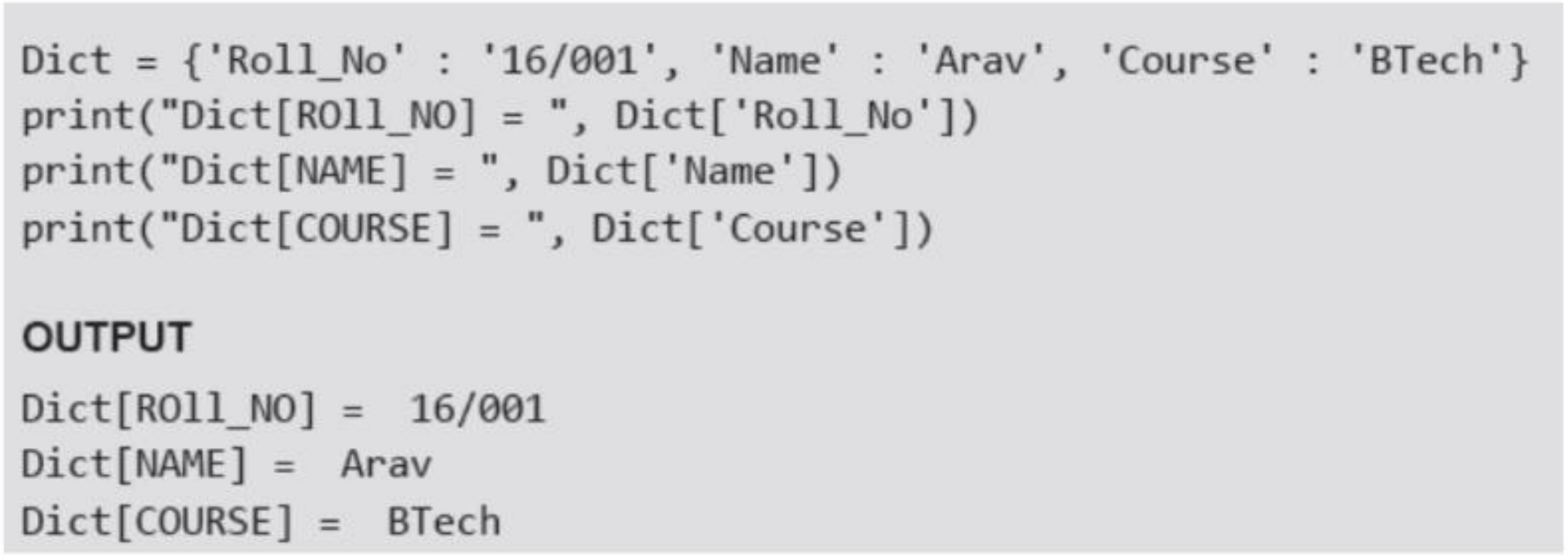
Example:



```python
# create a dictionary
my_dict = {1: "Smith", 2: "Emma", 3: "Jessa"}

# display dictionary
print(my_dict)
print(type(my_dict))

# create a dictionary using a dit class
my_dict = dict({1: "Smith", 2: "Emma", 3: "Jessa"})

# display dictionary
print(my_dict)
print(type(my_dict))

# access value using a key name
print(my_dict[1])

# change the value of a key
my_dict[1] = "Kelly"
print(my_dict[1])
```

```
{1: 'Smith', 2: 'Emma', 3: 'Jessa'}
<class 'dict'>
{1: 'Smith', 2: 'Emma', 3: 'Jessa'}
<class 'dict'>
Smith
Kelly

Executed in: 0.015 sec(s)
Memory: 4088 kilobyte(s)
```

Accessing Values of Dictionary

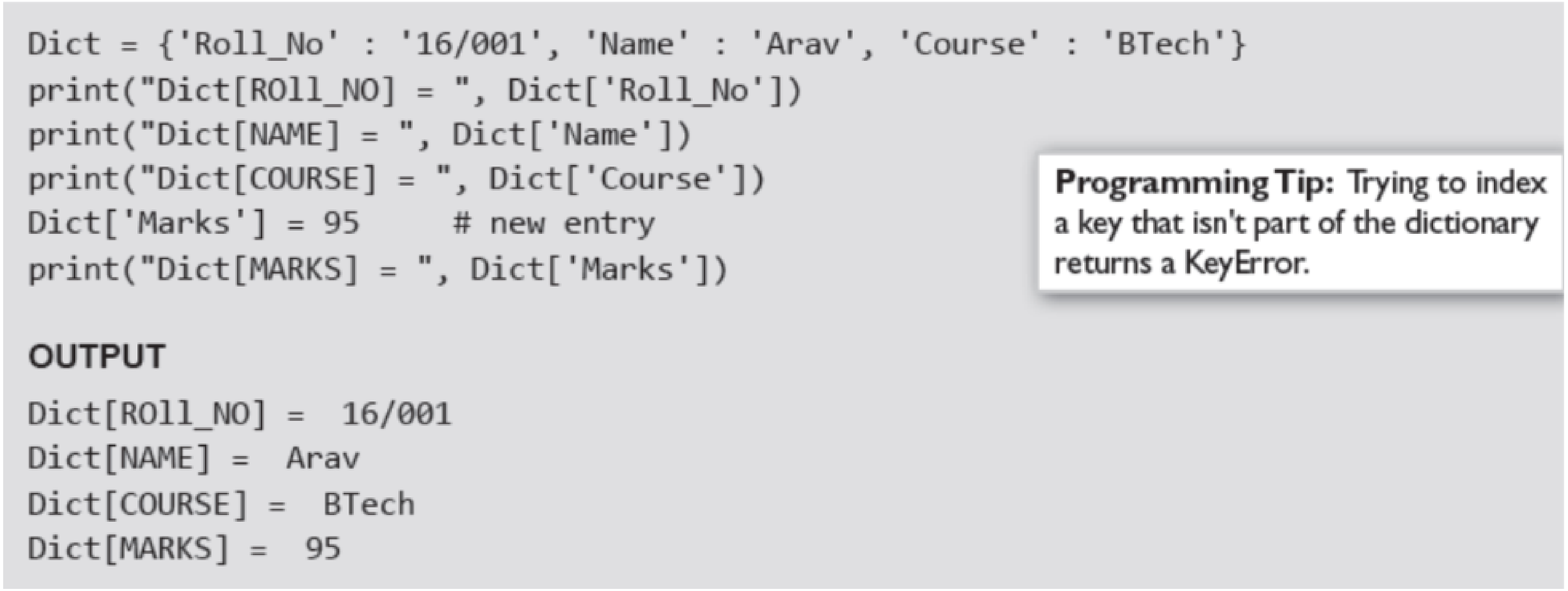


```
Dict = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}
print("Dict[ROll_NO] = ", Dict['Roll_No'])
print("Dict[NAME] = ", Dict['Name'])
print("Dict[COURSE] = ", Dict['Course'])
```

OUTPUT

```
Dict[ROll_NO] =  16/001
Dict[NAME] =  Arav
Dict[COURSE] =  BTech
```

Adding and Modifying an Item in a Dictionary



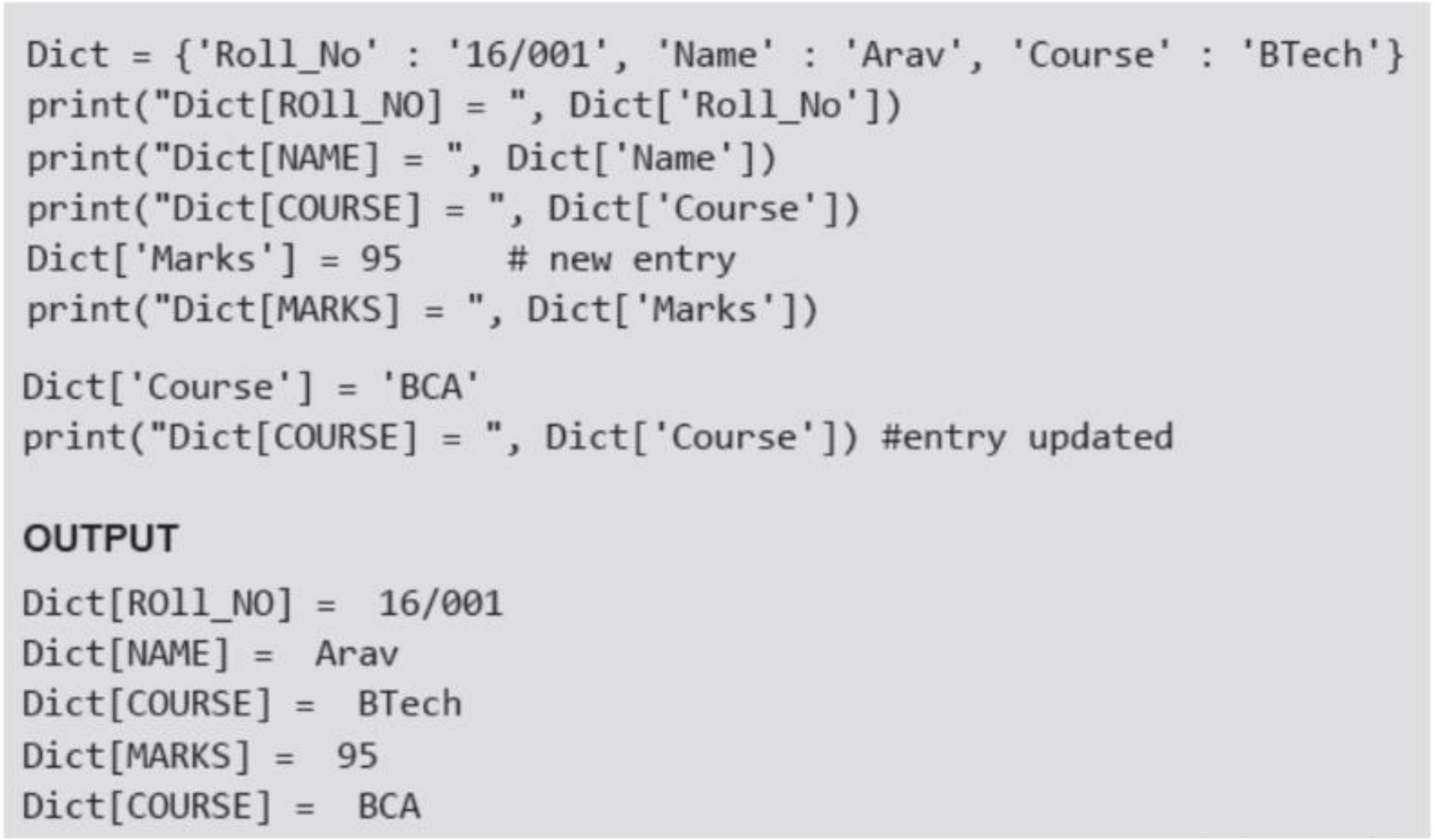
```
Dict = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}
print("Dict[ROll_NO] = ", Dict['Roll_No'])
print("Dict[NAME] = ", Dict['Name'])
print("Dict[COURSE] = ", Dict['Course'])
Dict['Marks'] = 95     # new entry
print("Dict[MARKS] = ", Dict['Marks'])
```

Programming Tip: Trying to index a key that isn't part of the dictionary returns a KeyError.

OUTPUT

```
Dict[ROll_NO] =  16/001
Dict[NAME] =  Arav
Dict[COURSE] =  BTech
Dict[MARKS] =  95
```

Modifying an Entry in Dictionary



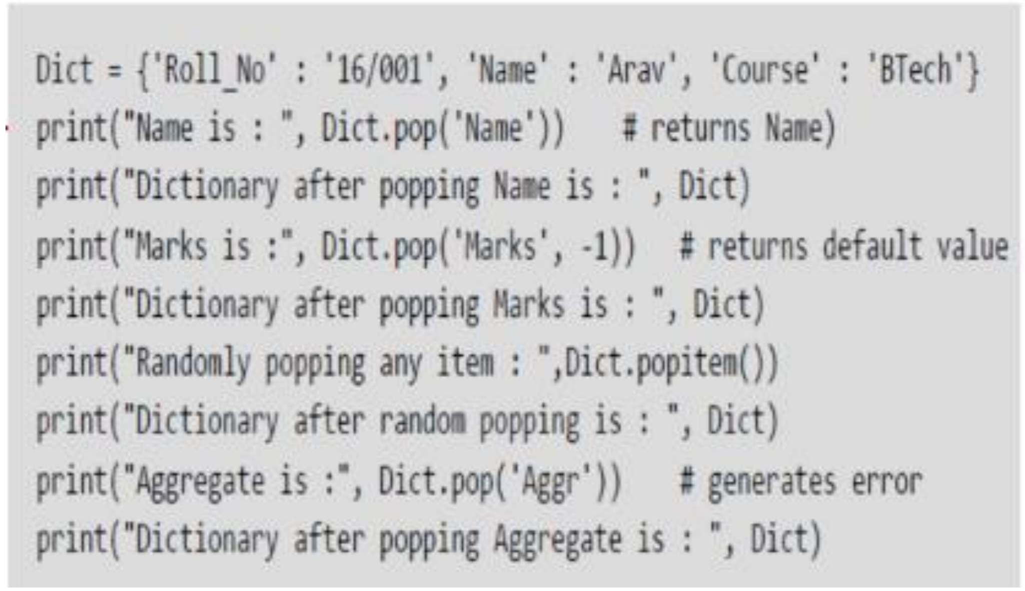
```
Dict = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}
print("Dict[ROll_NO] = ", Dict['Roll_No'])
print("Dict[NAME] = ", Dict['Name'])
print("Dict[COURSE] = ", Dict['Course'])
Dict['Marks'] = 95     # new entry
print("Dict[MARKS] = ", Dict['Marks'])

Dict['Course'] = 'BCA'
print("Dict[COURSE] = ", Dict['Course']) #entry updated
```

OUTPUT

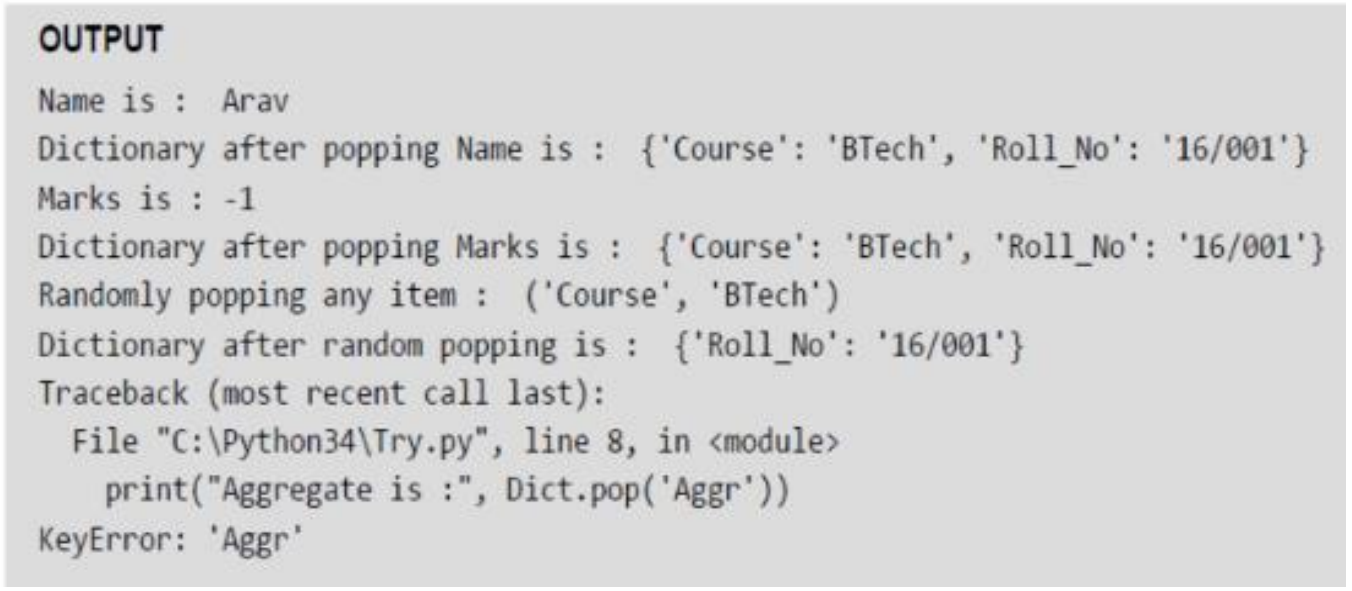
```
Dict[ROll_NO] =  16/001
Dict[NAME] =  Arav
Dict[COURSE] =  BTech
Dict[MARKS] =  95
Dict[COURSE] =  BCA
```

Deleting Items from Dictionary

- You can delete one or more items using the del keyword. To delete or remove all the items in just one statement, use the clear() function.
- Finally, to remove an entire dictionary from the memory, we can gain use the del statement as del Dict_name. The syntax to use the del statement can be given as,
- del dictionary_variable[key]



```
Dict = {'Roll_No' : '16/001', 'Name' : 'Arav', 'Course' : 'BTech'}
print("Name is : ", Dict.pop('Name'))    # returns Name)
print("Dictionary after popping Name is : ", Dict)
print("Marks is :", Dict.pop('Marks', -1))   # returns default value
print("Dictionary after popping Marks is : ", Dict)
print("Randomly popping any item : ",Dict.popitem())
print("Dictionary after random popping is : ", Dict)
print("Aggregate is :", Dict.pop('Aggr'))    # generates error
print("Dictionary after popping Aggregate is : ", Dict)
```



OUTPUT

```
Name is :  Arav
Dictionary after popping Name is :  {'Course': 'BTech', 'Roll_No': '16/001'}
Marks is : -1
Dictionary after popping Marks is :  {'Course': 'BTech', 'Roll_No': '16/001'}
Randomly popping any item :  ('Course', 'BTech')
Dictionary after random popping is :  {'Roll_No': '16/001'}
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 8, in <module>
    print("Aggregate is :", Dict.pop('Aggr'))
KeyError: 'Aggr'
```

Difference between a List and a Dictionary

- First, a list is an ordered set of items. But, a dictionary is a data structure that is used for matching one item (key) with another (value).
- Second, in lists, you can use indexing to access a particular item. But, these indexes should be a number. In dictionaries, you can use any type (immutable) of value as an index. For example, when we write Dict['Name'], Name acts as an index but it is not a number but a string.
- Third, lists are used to look up a value whereas a dictionary is used to take one value and look up another value. For this reason, dictionary is also known as a lookup table.
- Fourth, the key-value pair may not be displayed in the order in which it was specified while defining the dictionary. This is because Python uses complex algorithms (called hashing) to provide fast access to the items stored in the dictionary. This also makes dictionary preferable to use over a list of tuples.

Set

- In Python, a set is an **unordered collection of data items that are unique**. In other words, Python Set is a collection of elements (Or objects) that contains no duplicate elements.
- In Python, the Set data type used to represent a group of unique elements as a single entity. For example, If we want to store student ID numbers, we can use the set data type.
- The Set data type in Python is represented using a set class.
- We can create a Set using the two ways
  - By enclosing values in the curly brackets {}
  - Using a set() class.
- The set data type has the following characteristics.
- It is mutable which means we can change set items
- Duplicate elements are not allowed
- Heterogeneous (values of all data types) elements are allowed
- Insertion order of elements is not preserved, so we can’t perform indexing on a Set

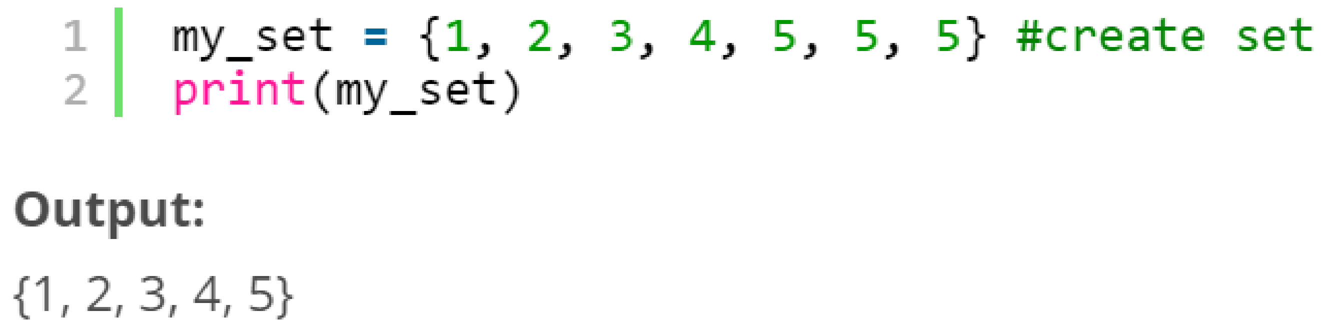
Some common useful set commands are as follow:

- # define an empty set
  a_set = set()
- # define a set
  a_set = {'one', 2}
- # adding item to a set
  a_set.add('c')
- # removing an item from a set, raise keyError if item does not exist
  a_set.remove('one')
- # removing an item from a set, no keyError if item does not exist
  a_set.discard('one')
- # remove the first item from a set and return it
  a_set.pop()
- # remove all items from a set
  a_set.clear()

Example:

- **Creating a set**

  Sets are created using the flower braces but instead of adding key-value pairs, you just pass values to it.



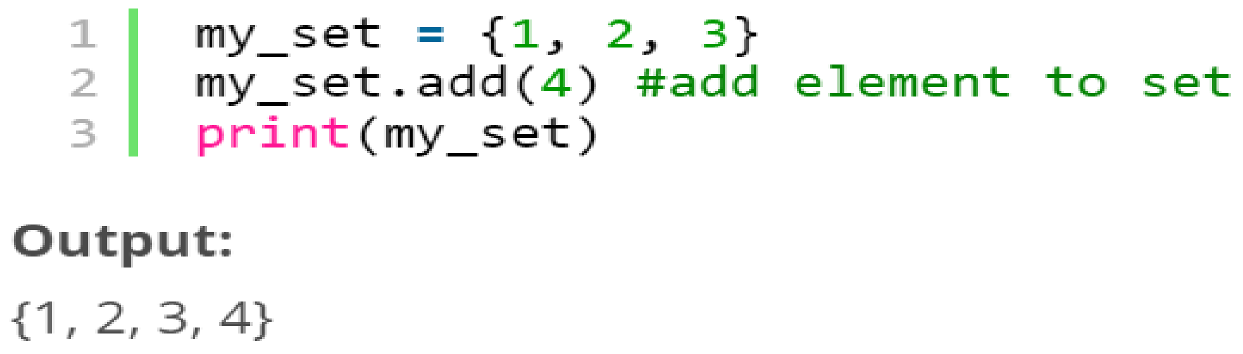
```python
my_set = {1, 2, 3, 4, 5, 5, 5} #create set
print(my_set)
```

Output:

```
{1, 2, 3, 4, 5}
```

- **Adding elements**

  To add elements, you use the add() function and pass the value to it.



```python
my_set = {1, 2, 3}
my_set.add(4) #add element to set
print(my_set)
```

Output:

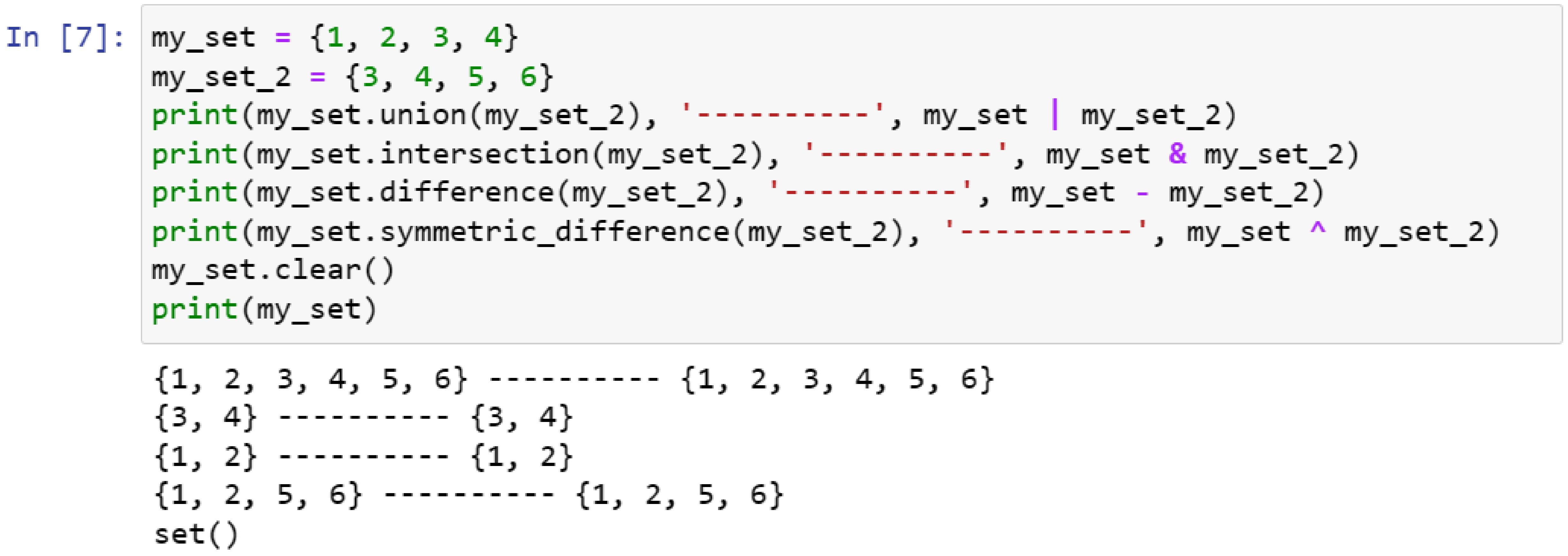
```
{1, 2, 3, 4}
```

Operations in sets

The different operations on set such as union, intersection and so on are shown below.

- The union() function combines the data present in both sets.
- The intersection() function finds the data present in both sets only.
- The difference() function deletes the data present in both and outputs data present only in the set passed.
- The symmetric_difference() does the same as the difference() function but outputs the data which is remaining in both sets.

Example:

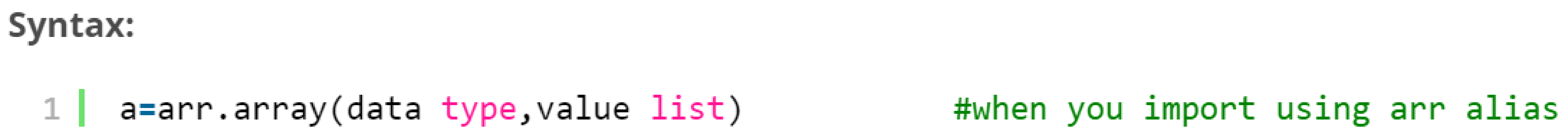


```
In [7]: my_set = {1, 2, 3, 4}
        my_set_2 = {3, 4, 5, 6}
        print(my_set.union(my_set_2), '----------', my_set | my_set_2)
        print(my_set.intersection(my_set_2), '----------', my_set & my_set_2)
        print(my_set.difference(my_set_2), '----------', my_set - my_set_2)
        print(my_set.symmetric_difference(my_set_2), '----------', my_set ^ my_set_2)
        my_set.clear()
        print(my_set)
```

```
{1, 2, 3, 4, 5, 6} ---------- {1, 2, 3, 4, 5, 6}
{3, 4} ---------- {3, 4}
{1, 2} ---------- {1, 2}
{1, 2, 5, 6} ---------- {1, 2, 5, 6}
set()
```

Arrays

- Arrays and lists are the same structure with one difference, Lists allow heterogeneous data element storage whereas Array allow only homogenous elements to be stored within them.
- Arrays in Python can be created after importing the array module as follows –
- →   import array as arr
- The array(data type, value list) function takes two parameters, the first being the data type of the value to be stored and the second is the value list.



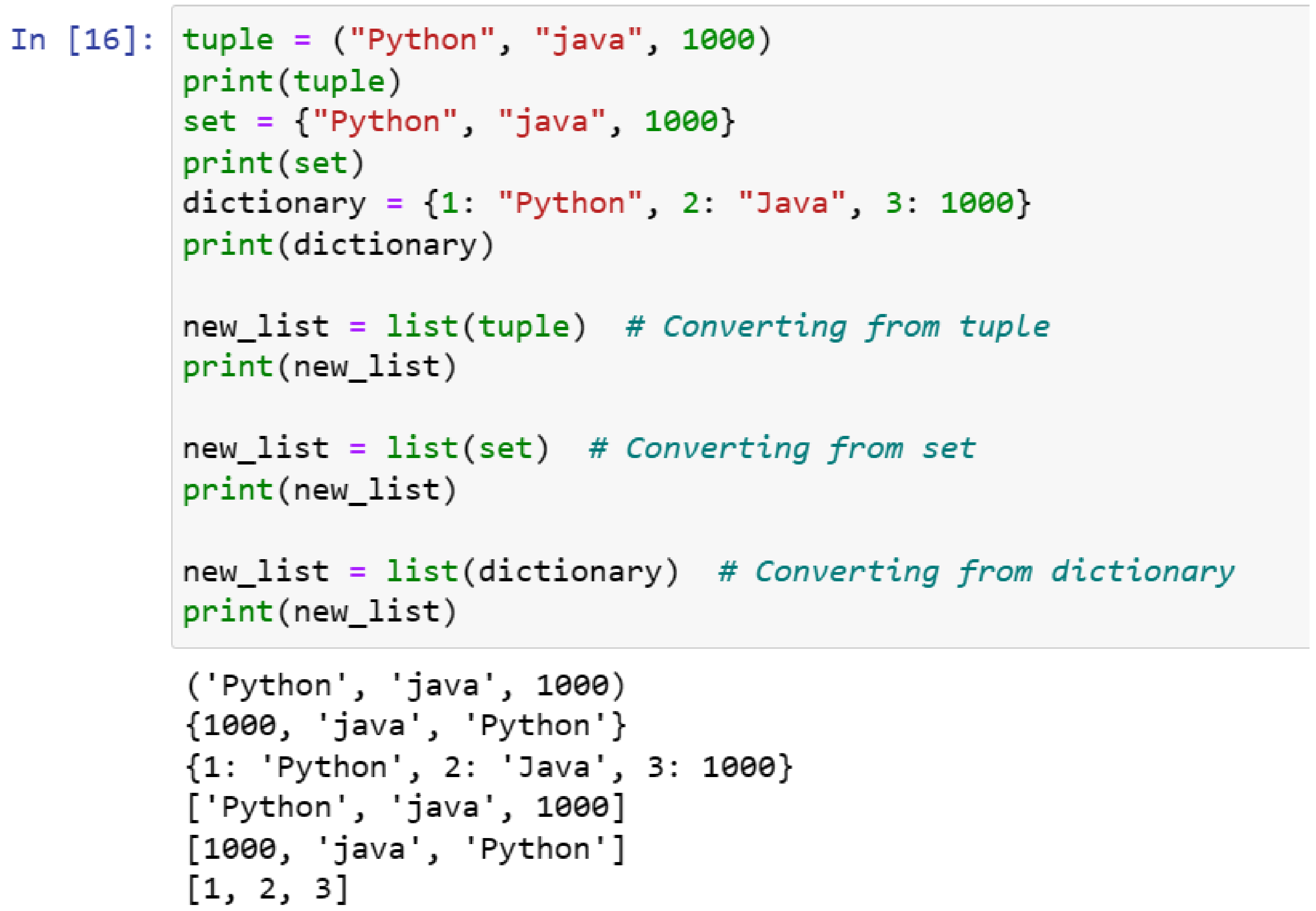
Syntax:

```
a=arr.array(data type,value list)          #when you import using arr alias
```

Conversions of Data Structures

Converting to a List

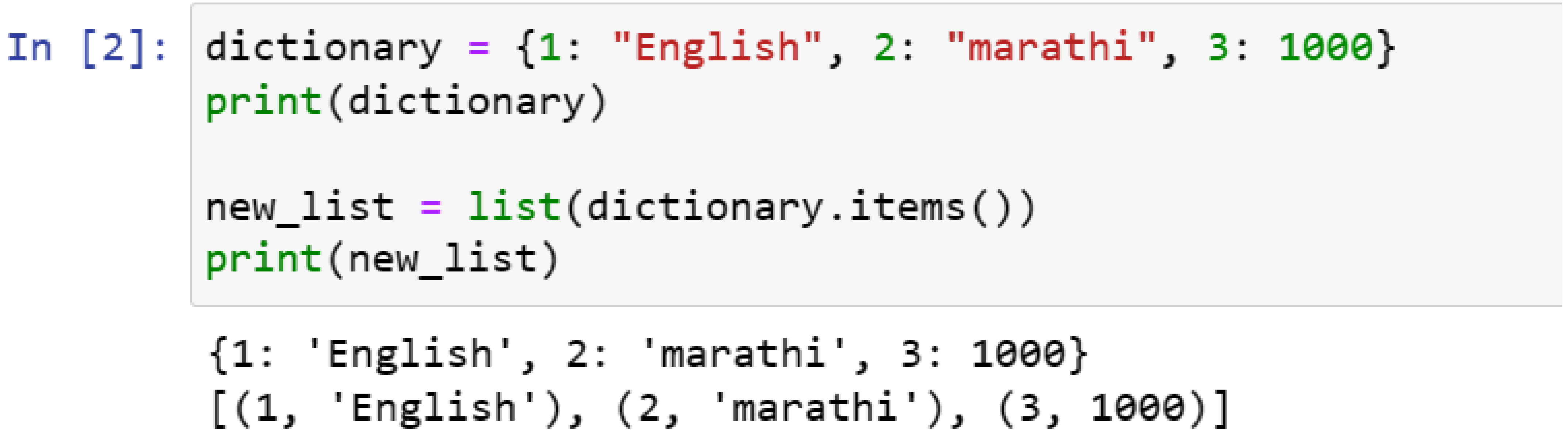
- a tuple, set, or dictionary can be converted to a list using the list() constructor.
- In the case of a dictionary, only the keys will be converted to a list.



```
In [16]: tuple = ("Python", "java", 1000)
         print(tuple)
         set = {"Python", "java", 1000}
         print(set)
         dictionary = {1: "Python", 2: "Java", 3: 1000}
         print(dictionary)

         new_list = list(tuple)  # Converting from tuple
         print(new_list)

         new_list = list(set)  # Converting from set
         print(new_list)

         new_list = list(dictionary)  # Converting from dictionary
         print(new_list)
```

```
('Python', 'java', 1000)
{1000, 'java', 'Python'}
{1: 'Python', 2: 'Java', 3: 1000}
['Python', 'java', 1000]
[1000, 'java', 'Python']
[1, 2, 3]
```

- We can use the dict.items() method of a dictionary to convert it into an iterable of (key, value) tuples. This can further be cast into a list of tuples using list()

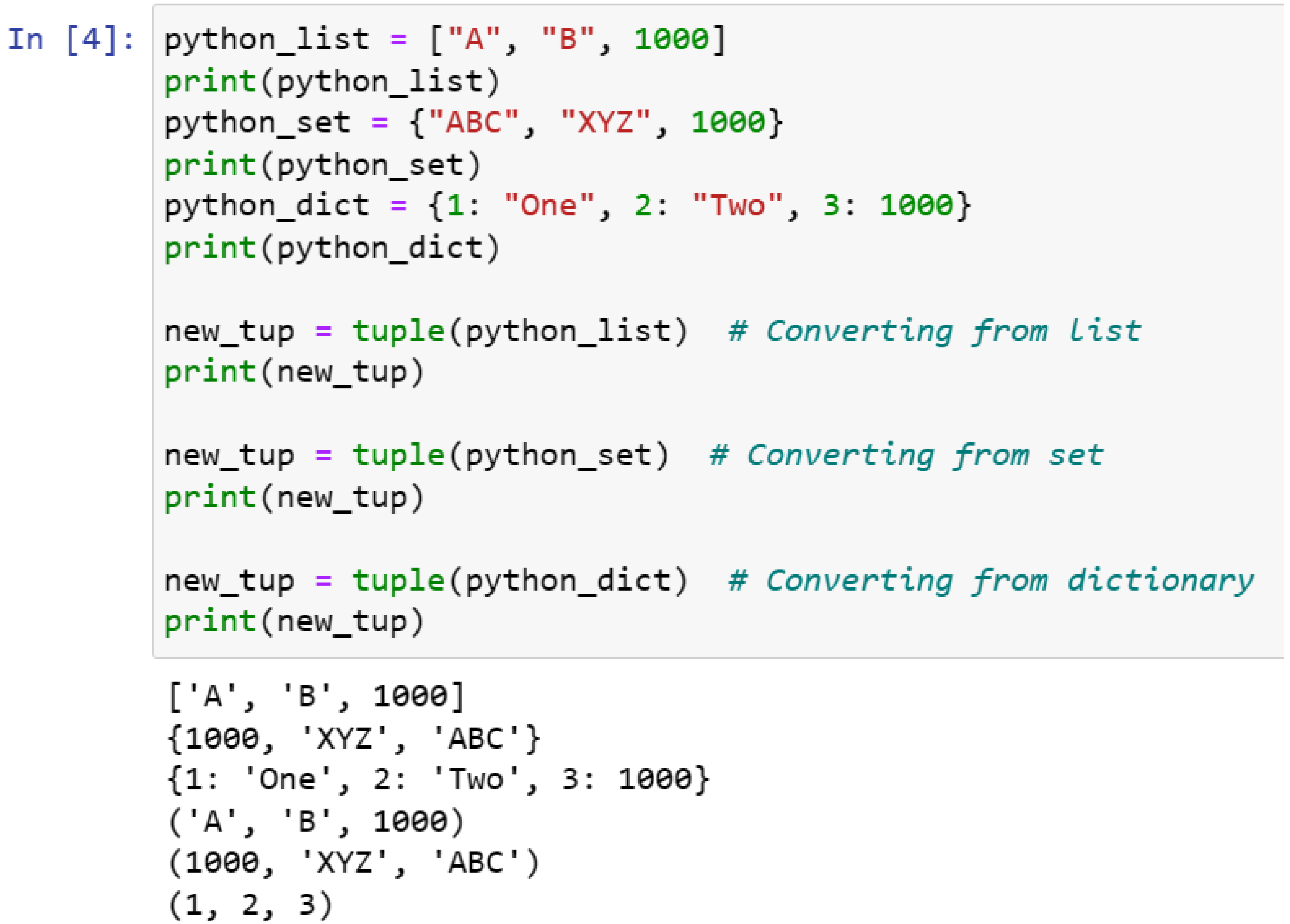


```
In [2]: dictionary = {1: "English", 2: "marathi", 3: 1000}
        print(dictionary)

        new_list = list(dictionary.items())
        print(new_list)
```

```
{1: 'English', 2: 'marathi', 3: 1000}
[(1, 'English'), (2, 'marathi'), (3, 1000)]
```

Converting to a Tuple

- Any data structure can be converted to a tuple using the tuple() constructor. In the case of a dictionary, only the keys will be converted to a tuple

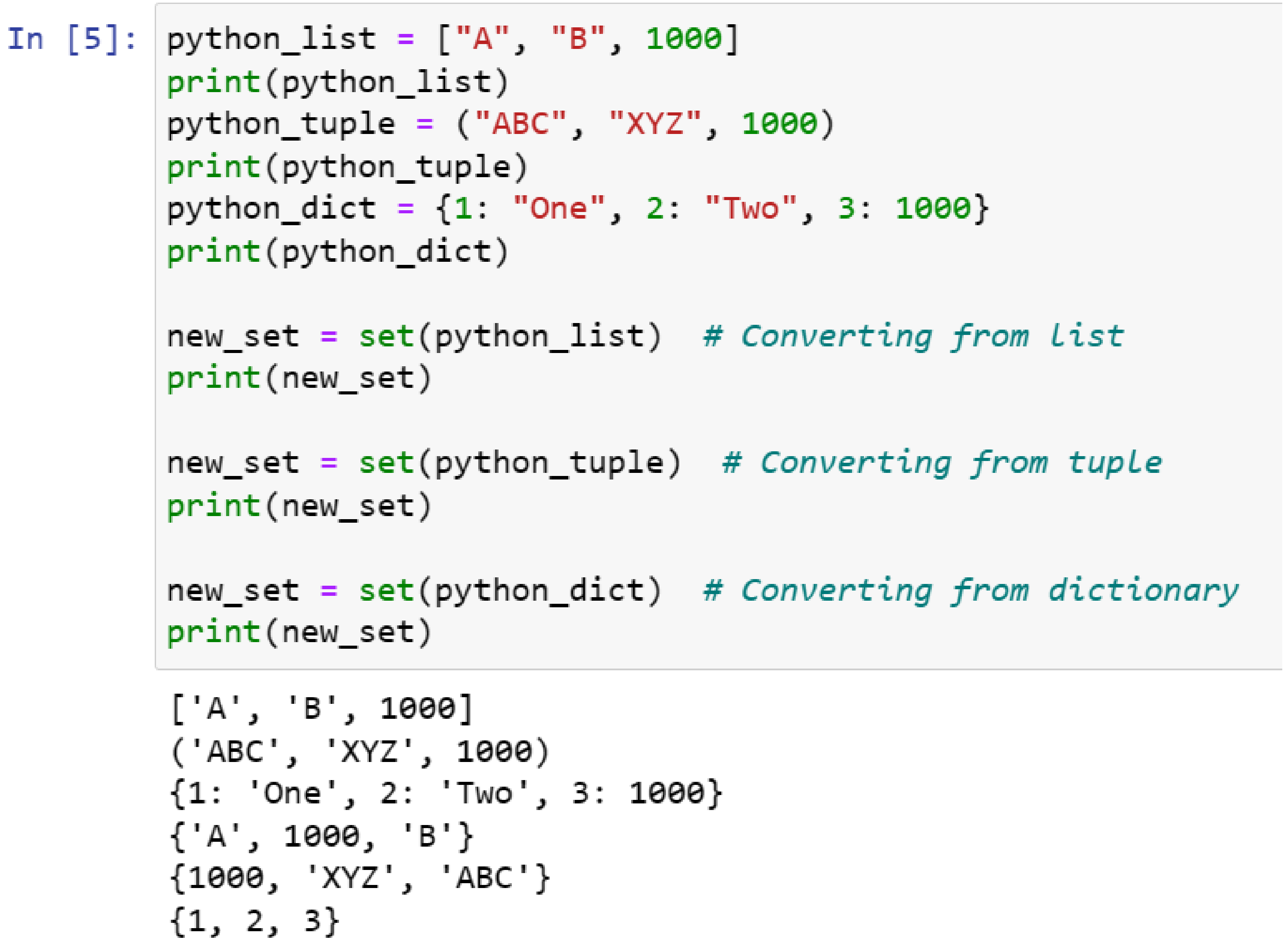


```
In [4]: python_list = ["A", "B", 1000]
        print(python_list)
        python_set = {"ABC", "XYZ", 1000}
        print(python_set)
        python_dict = {1: "One", 2: "Two", 3: 1000}
        print(python_dict)

        new_tup = tuple(python_list)  # Converting from list
        print(new_tup)

        new_tup = tuple(python_set)  # Converting from set
        print(new_tup)

        new_tup = tuple(python_dict)  # Converting from dictionary
        print(new_tup)
```

```
['A', 'B', 1000]
{1000, 'XYZ', 'ABC'}
{1: 'One', 2: 'Two', 3: 1000}
('A', 'B', 1000)
(1000, 'XYZ', 'ABC')
(1, 2, 3)
```

Converting to a Set

- The set() constructor can be used to create a set out of any other data structure. In the case of a dictionary, only the keys will be converted to a set



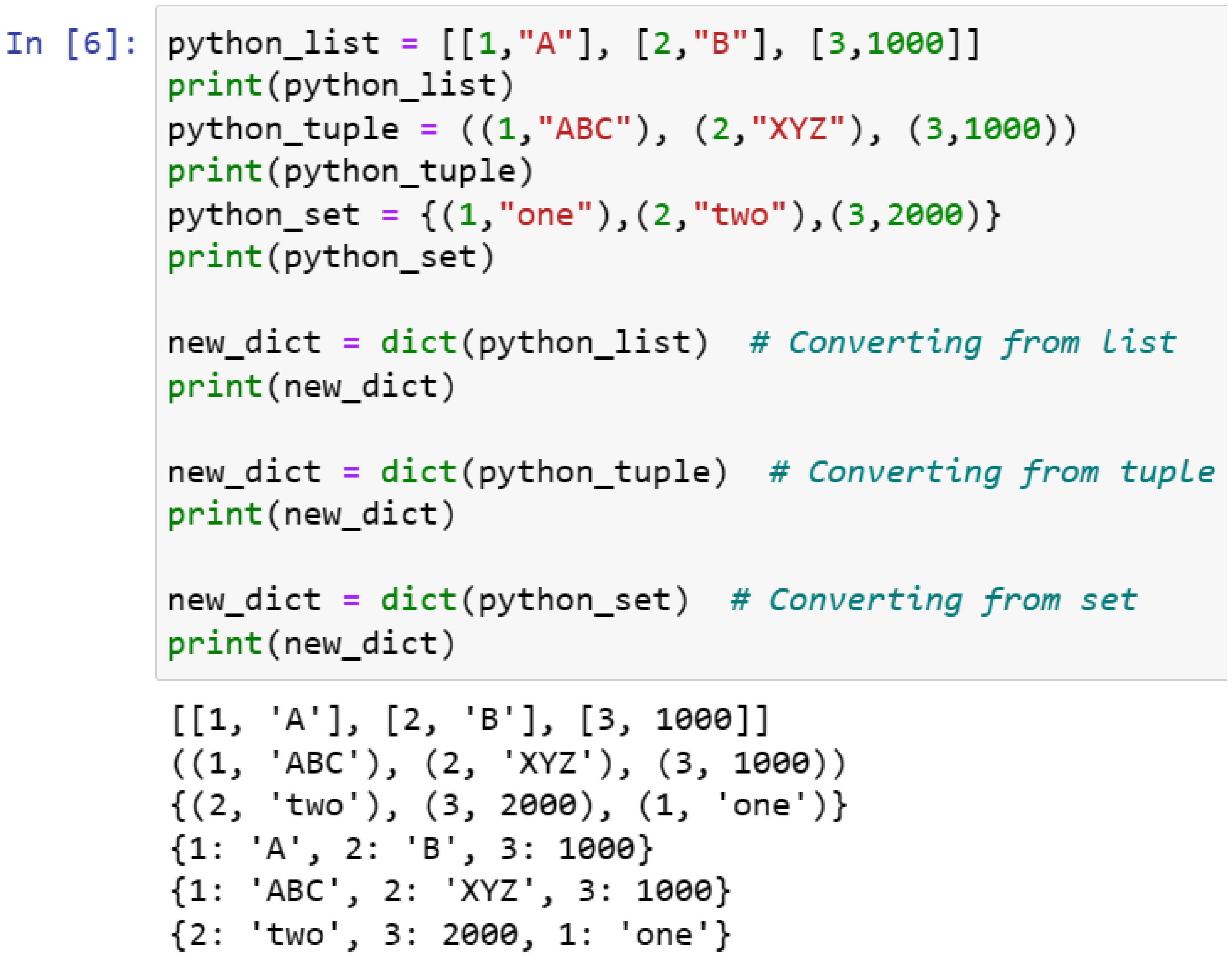
```
In [5]: python_list = ["A", "B", 1000]
        print(python_list)
        python_tuple = ("ABC", "XYZ", 1000)
        print(python_tuple)
        python_dict = {1: "One", 2: "Two", 3: 1000}
        print(python_dict)

        new_set = set(python_list)  # Converting from list
        print(new_set)

        new_set = set(python_tuple)  # Converting from tuple
        print(new_set)

        new_set = set(python_dict)  # Converting from dictionary
        print(new_set)
```

```
['A', 'B', 1000]
('ABC', 'XYZ', 1000)
{1: 'One', 2: 'Two', 3: 1000}
{'A', 1000, 'B'}
{1000, 'XYZ', 'ABC'}
{1, 2, 3}
```

Converting to a dictionary

- The dict() constructor cannot be used in the same way as the others because it requires key-value pairs instead of just values. Hence, the data must be stored in a format where **pairs** exist.
- For example, a list of tuples where the length of each tuple is 2 can be converted into a dictionary.
- Those pairs will then be converted into key-value pairs

- Example



```
In [6]: python_list = [[1,"A"], [2,"B"], [3,1000]]
        print(python_list)
        python_tuple = ((1,"ABC"), (2,"XYZ"), (3,1000))
        print(python_tuple)
        python_set = {(1,"one"),(2,"two"),(3,2000)}
        print(python_set)

        new_dict = dict(python_list)  # Converting from list
        print(new_dict)

        new_dict = dict(python_tuple)  # Converting from tuple
        print(new_dict)

        new_dict = dict(python_set)  # Converting from set
        print(new_dict)
```

```
[[1, 'A'], [2, 'B'], [3, 1000]]
((1, 'ABC'), (2, 'XYZ'), (3, 1000))
{(2, 'two'), (3, 2000), (1, 'one')}
{1: 'A', 2: 'B', 3: 1000}
{1: 'ABC', 2: 'XYZ', 3: 1000}
{2: 'two', 3: 2000, 1: 'one'}
```

When to use which Data Structure?

- Use lists to store a collection of data that does not need random access.
- Use lists if the data has to be modified frequently.
- Use a set if you want to ensure that every element in the data structure must be unique.
- Use tuples when you want that your data should not be altered.