MODULE-1

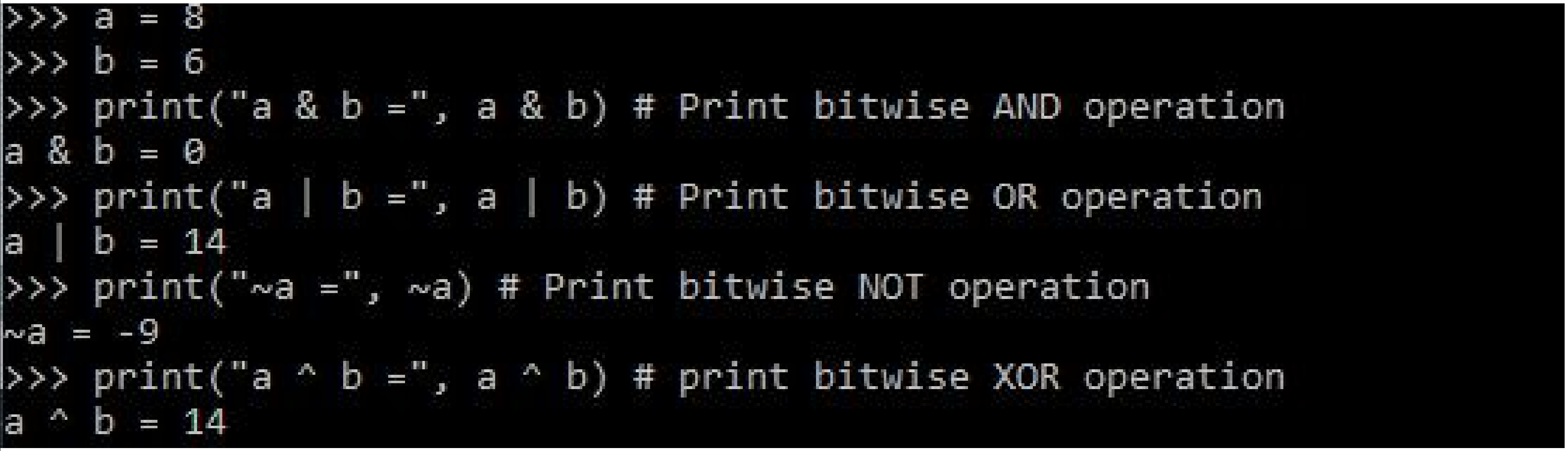
PPT-5

Python Bitwise Operators

- Bitwise operators are used to compare (binary) numbers.

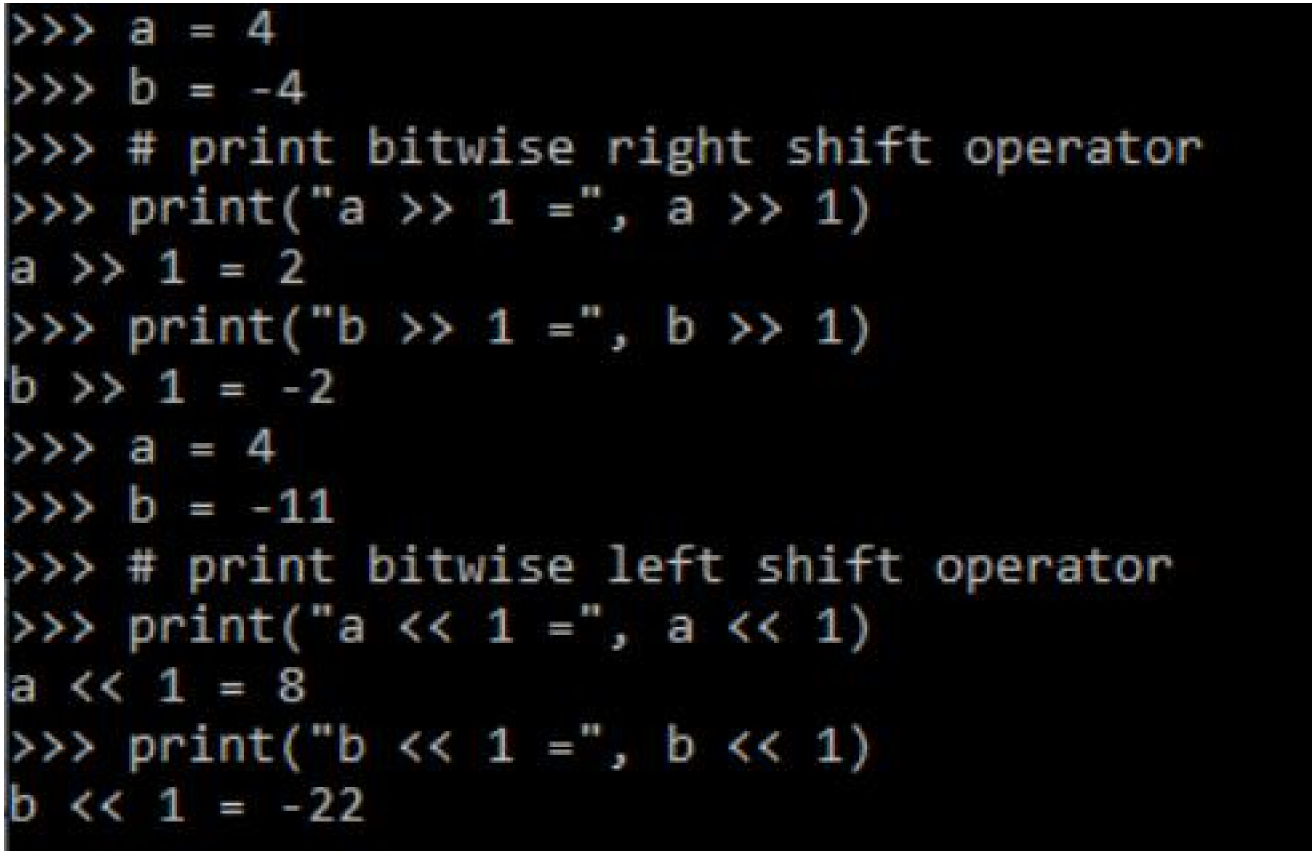
Operator	Name	Description	Syntax
&	AND	Sets each bit to 1 if both bits are 1	x & y
|	OR	Sets each bit to 1 if one of two bits is 1	x | y
^	XOR	Sets each bit to 1 if only one of two bits is 1	~x
~	NOT	Inverts all the bits () Returns one's complement of the number.	x ^ y
>>	Bitwise right shift	Shifts the bits of the number to the right and fills 0 on voids left as a result.	x>>
<<	Bitwise left shift	Shifts the bits of the number to the left and fills 0 on voids left as a result.	x<<

```
a = 8
b = 6
print("a & b =", a & b) # Print bitwise AND operation
print("a | b =", a | b) # Print bitwise OR operation
print("~a =", ~a) # Print bitwise NOT operation
print("a ^ b =", a ^ b) # print bitwise XOR operation
```



```
>>> a = 8
>>> b = 6
>>> print("a & b =", a & b) # Print bitwise AND operation
a & b = 0
>>> print("a | b =", a | b) # Print bitwise OR operation
a | b = 14
>>> print("~a =", ~a) # Print bitwise NOT operation
~a = -9
>>> print("a ^ b =", a ^ b) # print bitwise XOR operation
a ^ b = 14
```

```
a = 4
b = -4
# print bitwise right shift operator
print("a >> 1 =", a >> 1)
print("b >> 1 =", b >> 1)
a = 4
b = -11
# print bitwise left shift operator
print("a << 1 =", a << 1)
print("b << 1 =", b << 1)
```



```
>>> a = 4
>>> b = -4
>>> # print bitwise right shift operator
>>> print("a >> 1 =", a >> 1)
a >> 1 = 2
>>> print("b >> 1 =", b >> 1)
b >> 1 = -2
>>> a = 4
>>> b = -11
>>> # print bitwise left shift operator
>>> print("a << 1 =", a << 1)
a << 1 = 8
>>> print("b << 1 =", b << 1)
b << 1 = -22
```

Python Collections (Arrays)

- There are four collection data types in the Python programming language:
  1. **List** is a collection which is ordered and changeable. Allows duplicate members.
  2. **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
  3. **Set** is a collection which is unordered and unindexed. No duplicate members.
  4. **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.
- When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

List

- A list is a collection which is ordered and changeable. In Python lists are written with square brackets.
- Python knows a number of *compound* data types, used to group together other values.
- The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets.
- Lists might contain items of different types, but usually the items all have the same type.

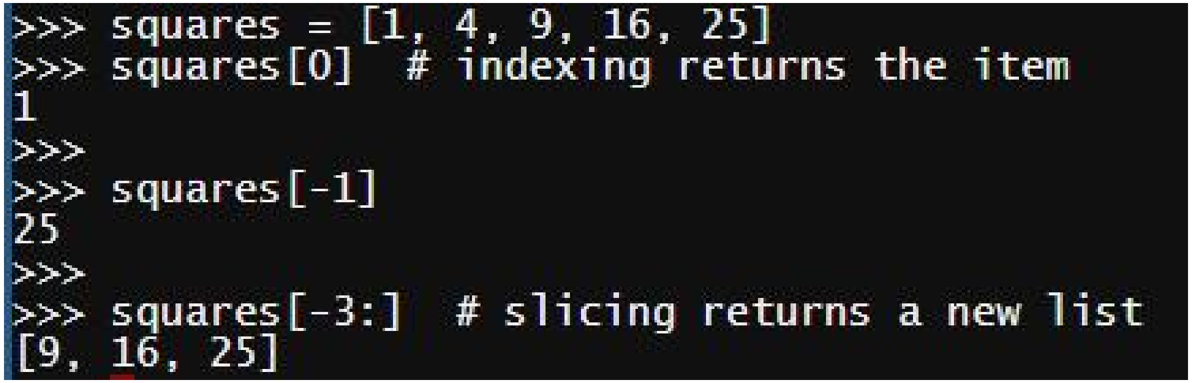
```
my_list= [“a", “b", “c"]
print(my_list)
squares = [1, 4, 9, 16, 25]
squares
```

- Like strings, lists can be indexed and sliced:

squares[0] *# indexing returns the item*

squares[-1]

squares[-3:] *# slicing returns a new list*



```
>>> squares = [1, 4, 9, 16, 25]
>>> squares[0]  # indexing returns the item
1
>>>
>>> squares[-1]
25
>>>
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

- Lists also support operations like concatenation:

squares + [36, 49, 64, 81, 100]

- Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

cubes = [1, 8, 27, 65, 125] *# cube of 4 is 64, not 65!*

cubes[3] = 64 *# replace the wrong value*

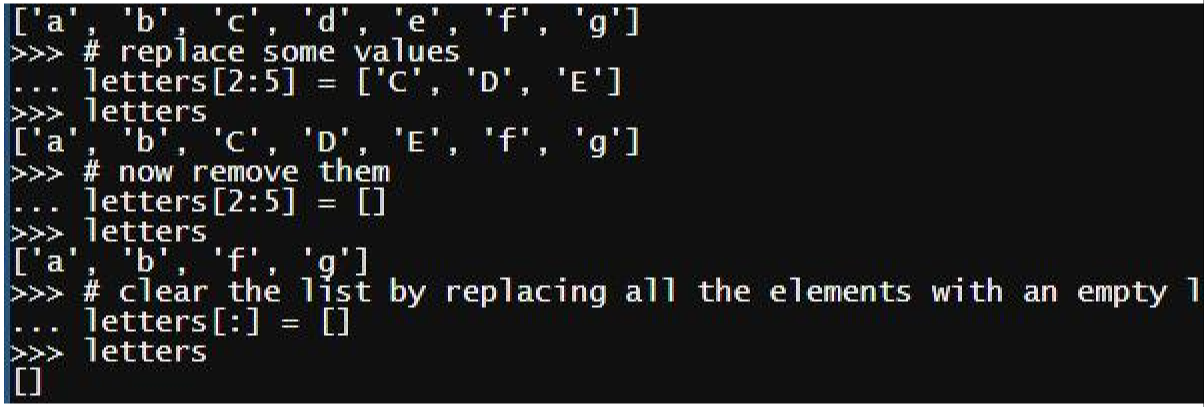
cubes

- You can also add new items at the end of the list, by using the append() *method* (we will see more about methods later):

```
cubes.append(216) # add the cube of 6
cubes.append(7 ** 3) # and the cube of 7
cubes
```

- Assignment to slices is also possible, and this can even change the size of the list or clear it entirely.

```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
letters
# replace some values
letters[2:5] = ['C', 'D', 'E']
letters
# now remove them
letters[2:5] = []
letters
# clear the list by replacing all the elements with an empty list
letters[:] = []
letters
```



```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
... letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
... letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty l
... letters[:] = []
>>> letters
[]
```

- The built-in function len() also applies to lists:

letters = ['a', 'b', 'c', 'd']

len(letters)

It is possible to nest lists (create lists containing other lists), for example:

a = ['a', 'b', 'c']

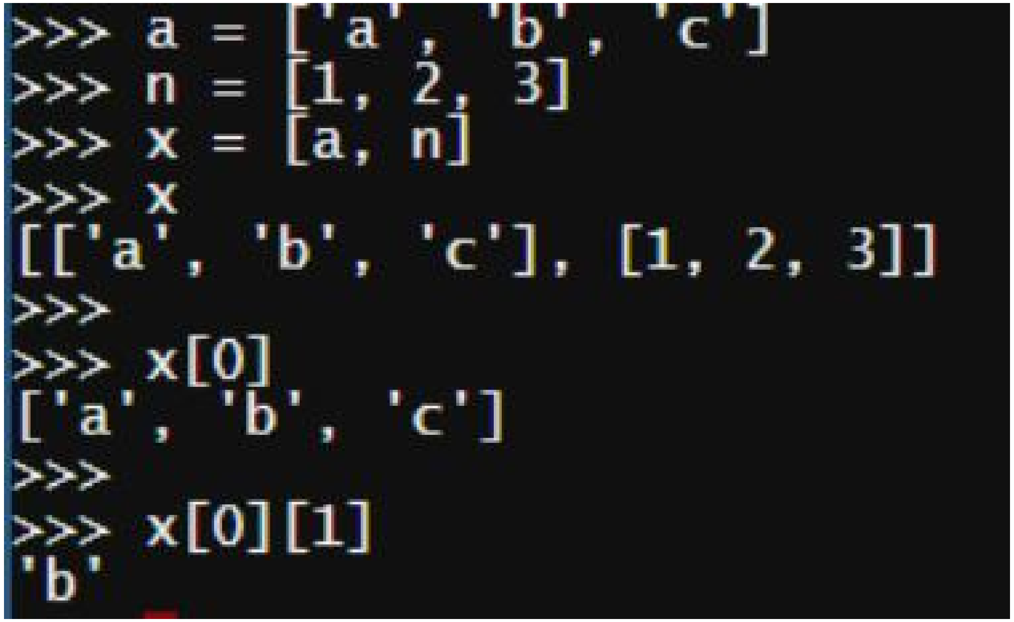
n = [1, 2, 3]

x = [a, n]

x

x[0]

x[0][1]



```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>>
>>> x[0]
['a', 'b', 'c']
>>>
>>> x[0][1]
'b'
```

String split() Method

Split a string into a list where each word is a list item:

```
txt = "welcome to the jungle"
x = txt.split()
print(x)
```

Tuple

- We saw that lists and strings have many common properties, such as indexing and slicing operations.
- Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.
- A tuple consists of a number of values separated by commas, for instance:
- A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

You can access tuple items by referring to the index number, inside square brackets:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

- Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

- You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new tuple with the specified items.

```
thistuple = "apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

- Specify negative indexes if you want to start the search from the end of the tuple:

```
print(thistuple[-4:-1])
```

- Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.
- But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

- To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

```
thistuple = ("apple",)
print(type(thistuple))
thistuple = ("apple") #NOT a tuple
print(type(thistuple))
```

- To determine how many items a tuple has, use the len() method.

Set

- A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

- **Note:** Sets are unordered, so you cannot be sure in which order the items will appear.
- You cannot access items in a set by referring to an index, since sets are unordered the items has no index.
- But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x) #Loop through the set, and print the values:
```

- Once a set is created, you cannot change its items, but you can add new items.
- To add one item to a set use the add() method.
- To add more than one item to a set use the update() method.

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

```
thisset = {"apple", "banana", "cherry"}
thisset.update(["orange", "mango", "grapes"])
print(thisset)
```

- To determine how many items a set has, use the len() method.
- To remove an item in a set, use the remove(), or the discard() method.

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

Note- If the item to remove does not exist, remove() will raise an error.

- Remove an item by using the discard() method:

```
thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
print(thisset)
```

Note- If the item to remove does not exist, discard() will **NOT** raise an error.

- You can also use the pop(), method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed. The return value of the pop() method is the removed item.

```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)
```

- **Note:** Sets are *unordered*, so when using the pop() method, you will not know which item that gets removed.

- The clear() method empties the set:

```
thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)
```

- The del keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}
del thisset
print(thisset)
```

- You can use the union() method that returns a new set containing all items from both sets, or the update() method that inserts all the items from one set into another:

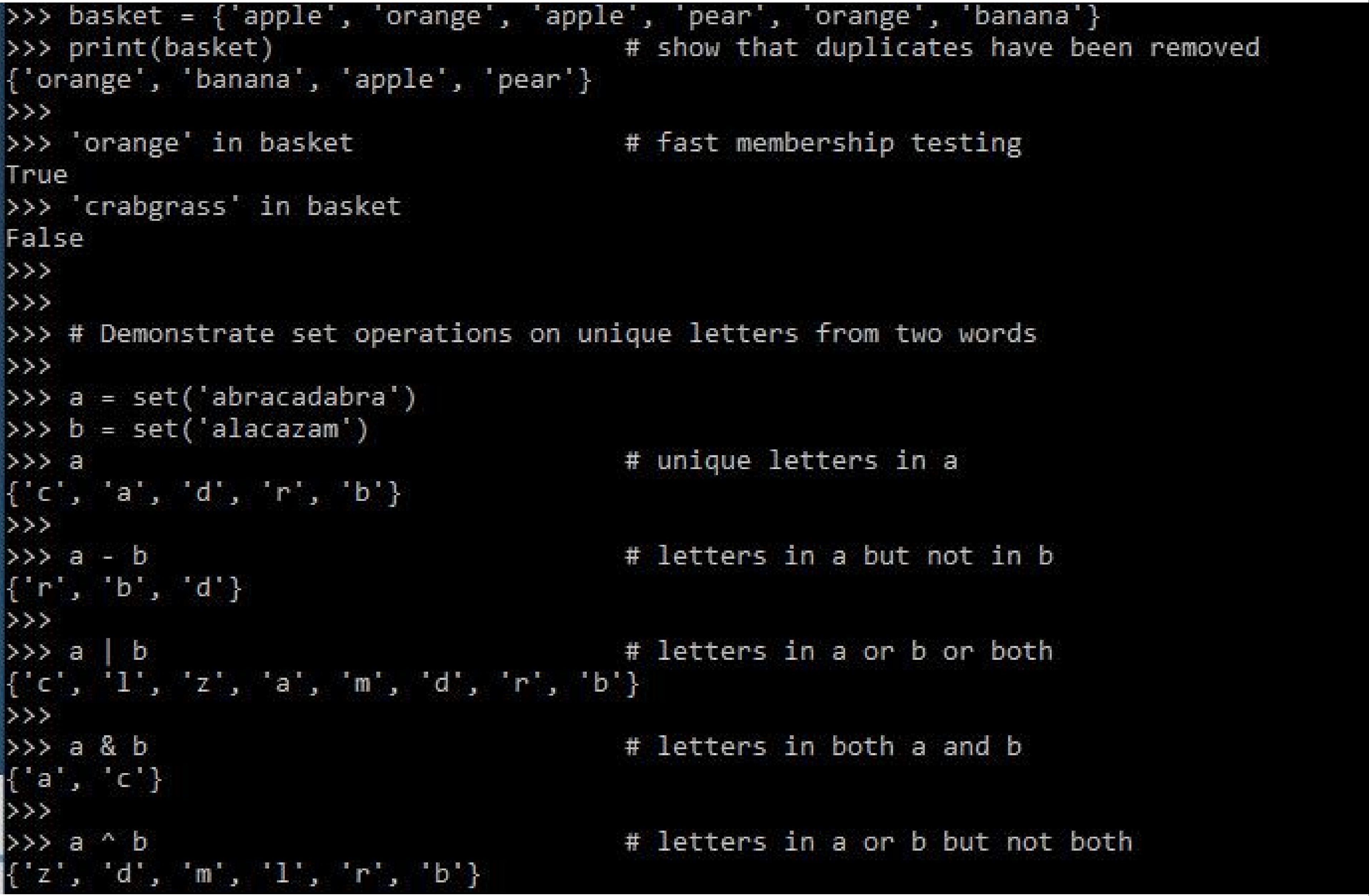
```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)
```

- The update() method inserts the items in set2 into set1:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set1.update(set2)
print(set1)
```

- **Note:** Both union() and update() will exclude any duplicate items.

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)     # show that duplicates have been removed
'orange' in basket          # fast membership testing
'crabgrass' in basket
# Demonstrate set operations on unique letters from two words
a = set('abracadabra')
b = set('alacazam')
a                                 # unique letters in a
a - b                             # letters in a but not in b
a | b                             # letters in a or b or both
a & b                             # letters in both a and b
a ^ b                             # letters in a or b but not both
```



```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'apple', 'pear'}
>>>
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False
>>>
>>>
>>> # Demonstrate set operations on unique letters from two words
>>>
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'c', 'a', 'd', 'r', 'b'}
>>>
>>> a - b                              # letters in a but not in b
{'r', 'b', 'd'}
>>>
>>> a | b                              # letters in a or b or both
{'c', 'l', 'z', 'a', 'm', 'd', 'r', 'b'}
>>>
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>>
>>> a ^ b                              # letters in a or b but not both
{'z', 'd', 'm', 'l', 'r', 'b'}
```

Set Methods

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

Dictionary

- A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

```
thisdict =        {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

- You can access the items of a dictionary by referring to its key name, inside square brackets:

```
x = thisdict["model"]
print(x)
```

- There is also a method called get() that will give you the same result.

```
x = thisdict.get("model")
```

- You can change the value of a specific item by referring to its key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

- Adding an item to the dictionary is done by using a new index key and assigning a value to it:

```
thisdict["color"] = "red"
print(thisdict)
```

- The pop() method removes the item with the specified key name:

```
thisdict.pop("model")
print(thisdict)
```

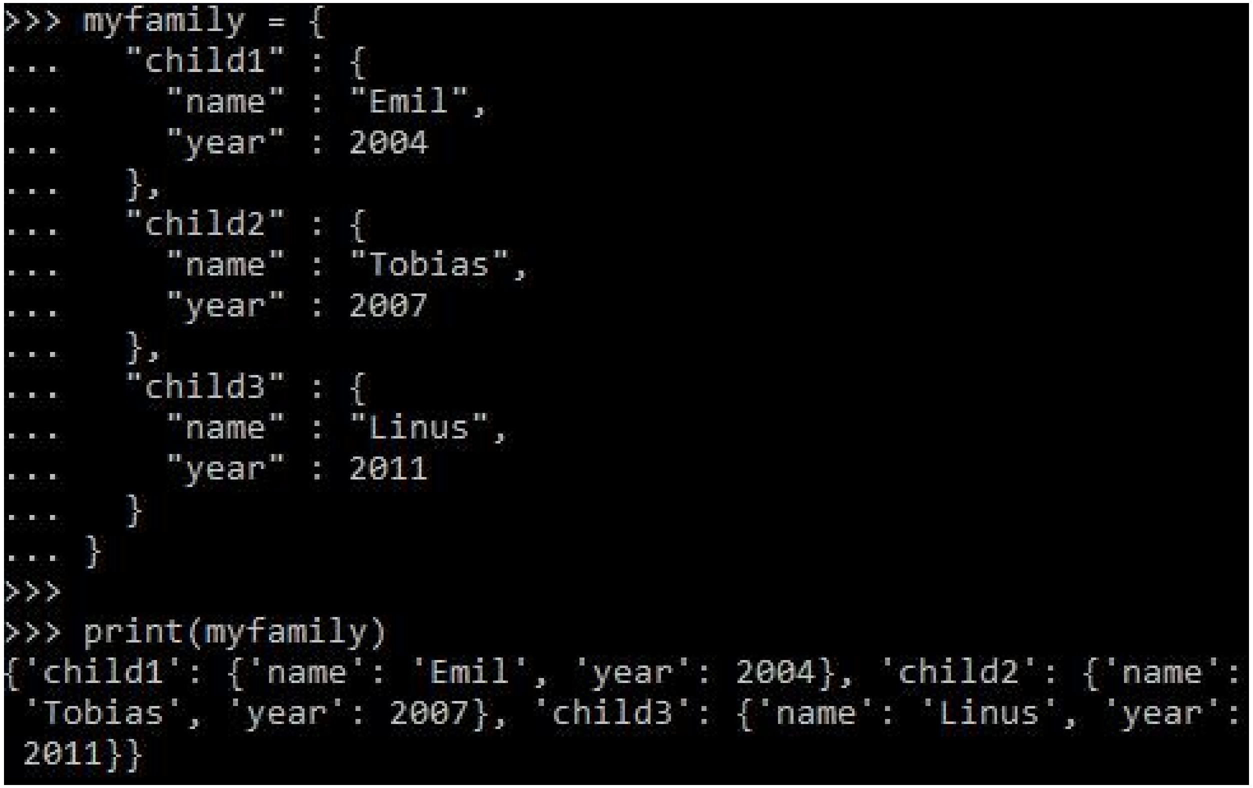
- The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead):

- The del keyword removes the item with the specified key name:

```
del thisdict["model"]
print(thisdict)
del thisdict #delete the dictionary completely:
print(thisdict)##this will cause an error because "thisdict" no longer exists.

thisdict.clear() #clear() method empties the dictionary
print(thisdict)

mydict = thisdict.copy() #Make a copy of a dictionary with the copy() method
print(mydict)

#Another way to make a copy is to use the built-in function dict()
mydict = dict(thisdict)
print(mydict)
```

- A dictionary can also contain many dictionaries, this is called nested dictionaries.

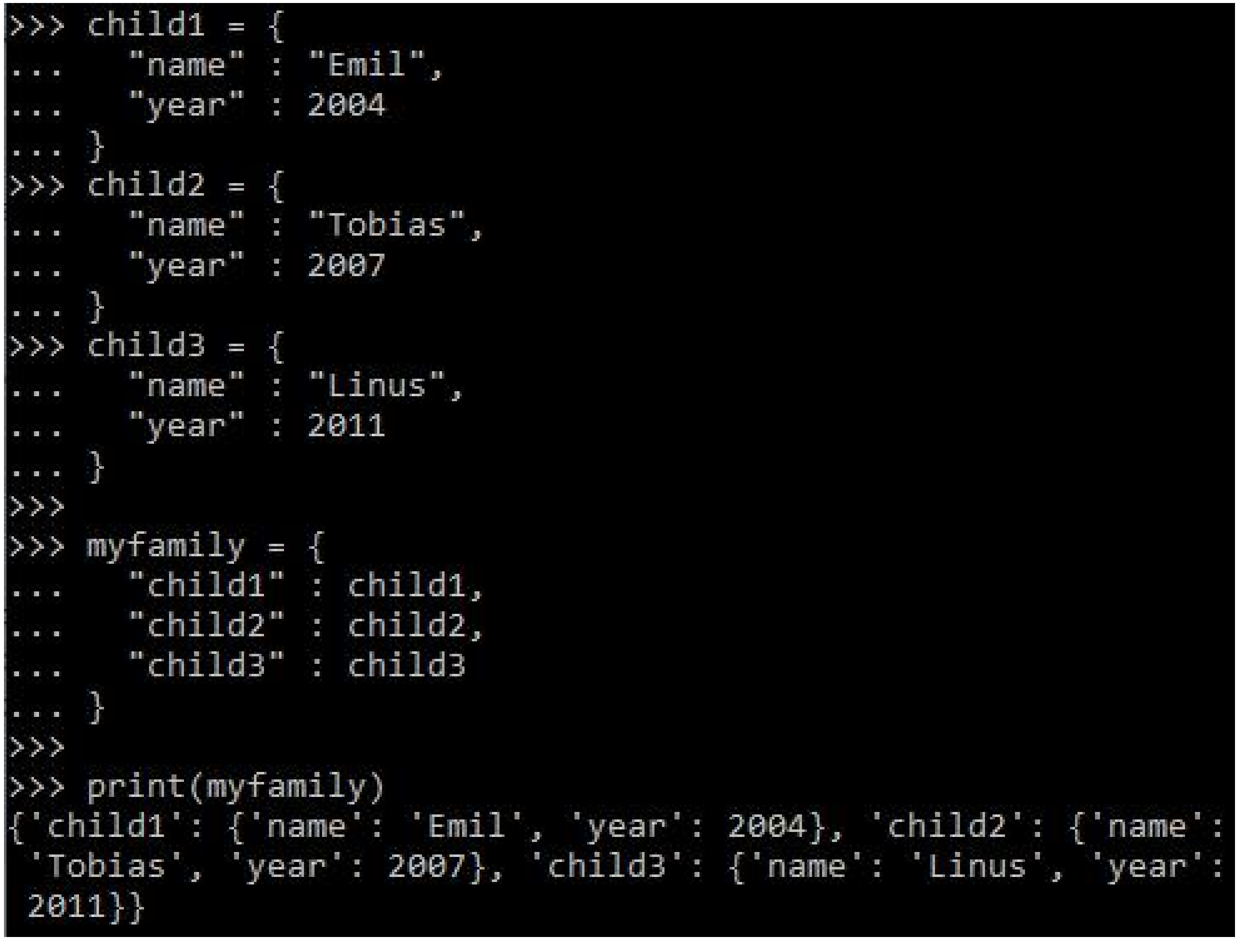
```
myfamily = {
 "child1" : {
  "name" : "Emil",
  "year" : 2004
 },
 "child2" : {
  "name" : "Tobias",
  "year" : 2007
 },
 "child3" : {
  "name" : "Linus",
  "year" : 2011
 }
}
print(myfamily)
```



```
>>> myfamily = {
...     "child1" : {
...       "name" : "Emil",
...       "year" : 2004
...     },
...     "child2" : {
...       "name" : "Tobias",
...       "year" : 2007
...     },
...     "child3" : {
...       "name" : "Linus",
...       "year" : 2011
...     }
... }
>>>
>>> print(myfamily)
{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name':
 'Tobias', 'year': 2007}, 'child3': {'name': 'Linus', 'year':
 2011}}
```

- Create three dictionaries, then create one dictionary that will contain the other three dictionaries:

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}
myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
print(myfamily)
```



```
>>> child1 = {
...   "name" : "Emil",
...   "year" : 2004
... }
>>> child2 = {
...   "name" : "Tobias",
...   "year" : 2007
... }
>>> child3 = {
...   "name" : "Linus",
...   "year" : 2011
... }
>>>
>>> myfamily = {
...   "child1" : child1,
...   "child2" : child2,
...   "child3" : child3
... }
>>>
>>> print(myfamily)
{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name':
 'Tobias', 'year': 2007}, 'child3': {'name': 'Linus', 'year':
 2011}}
```

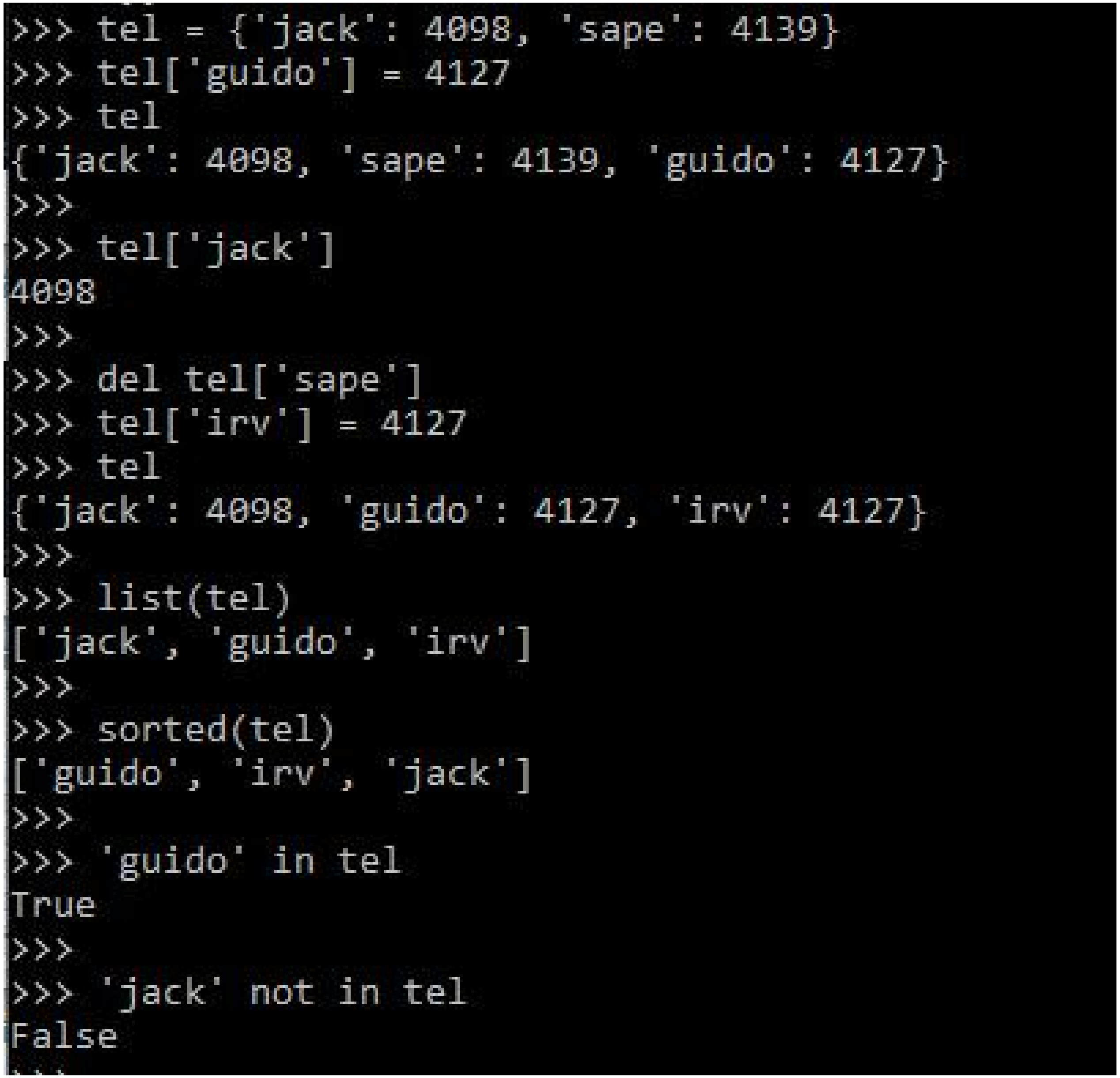
- It is also possible to use the dict() constructor to make a new dictionary:

```
thisdict = dict(brand="Ford", model="Mustang", year=1964)
# note that keywords are not string literals
# note the use of equals rather than colon for the assignment
print(thisdict)
```

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Example:

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
tel
tel['jack']
del tel['sape']
tel['irv'] = 4127
tel
list(tel)
sorted(tel)
'guido' in tel
'jack' not in tel
```



```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>>
>>> tel['jack']
4098
>>>
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>>
>>> list(tel)
['jack', 'guido', 'irv']
>>>
>>> sorted(tel)
['guido', 'irv', 'jack']
>>>
>>> 'guido' in tel
True
>>>
>>> 'jack' not in tel
False
>>>
```