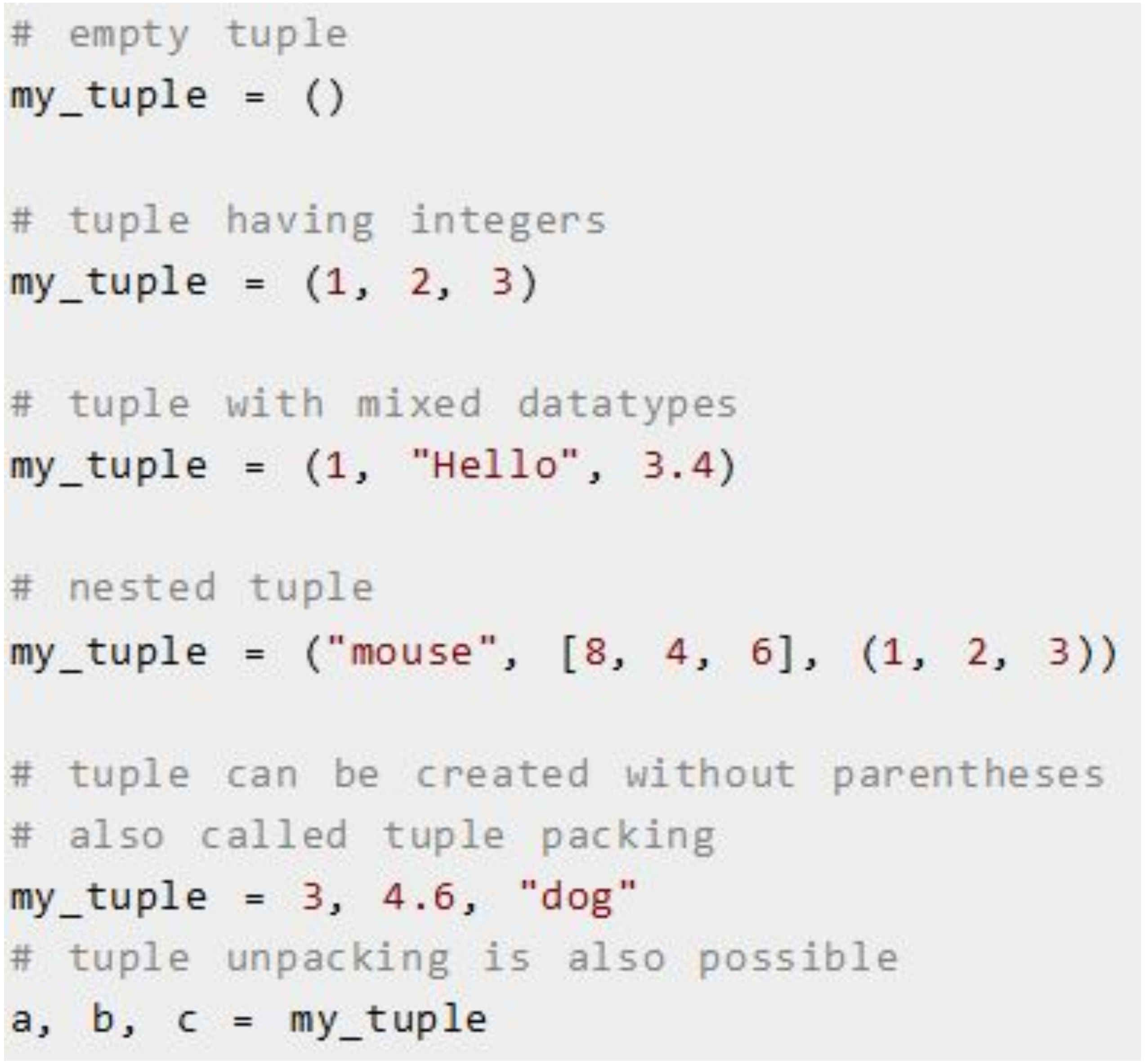
Python Programming - III

-By Nimesh Kumar Dagur, CDAC, India

Python Tuples

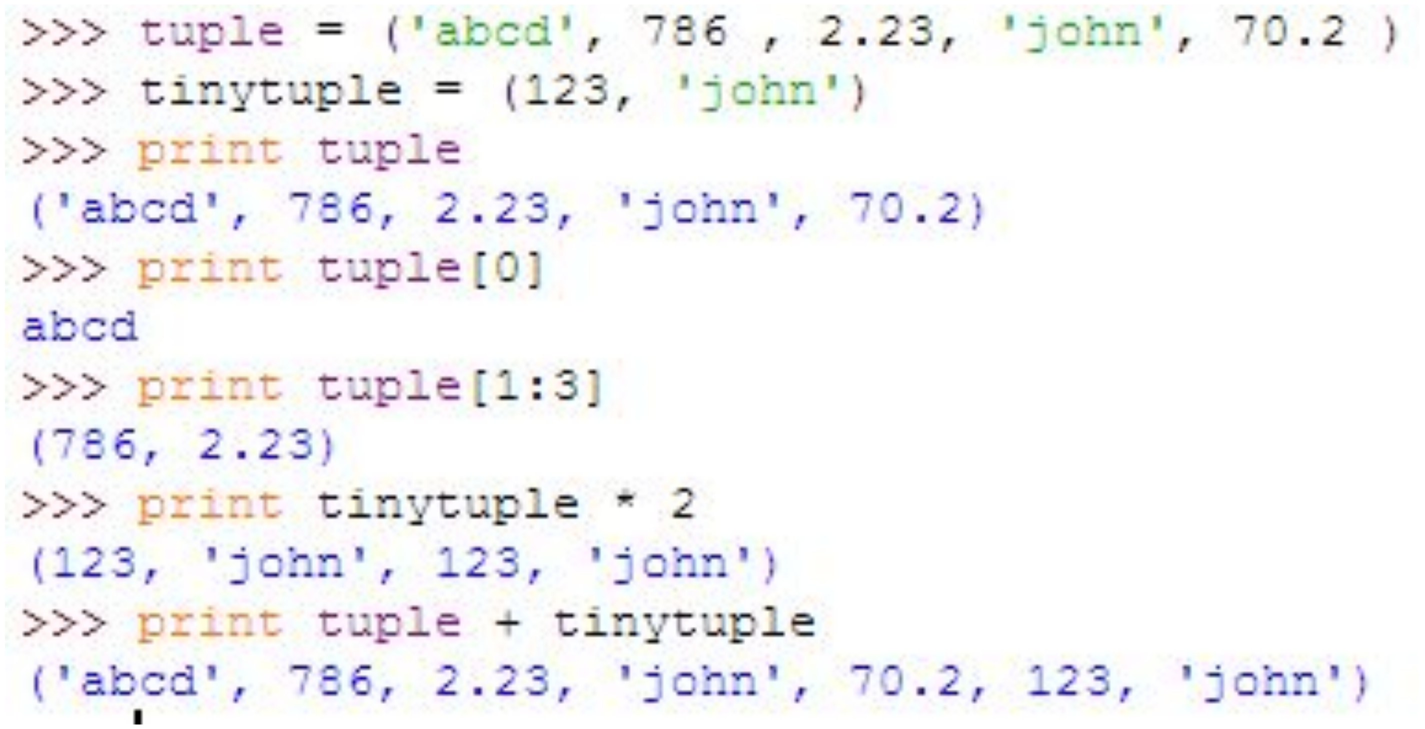
- A tuple is another sequence data type that is similar to the list.
- A tuple consists of a number of values separated by commas.
- Unlike lists, however, tuples are enclosed within parentheses.
- The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated.
- Tuples can be thought of as read-only lists.
- The parentheses are optional but is a good practice to write it.
- A tuple can have any number of items and they may be of different types (integer, float, list, string etc.).

Python Tuples



```
# empty tuple
my_tuple = ()

# tuple having integers
my_tuple = (1, 2, 3)

# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)

# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# tuple can be created without parentheses
# also called tuple packing
my_tuple = 3, 4.6, "dog"
# tuple unpacking is also possible
a, b, c = my_tuple
```

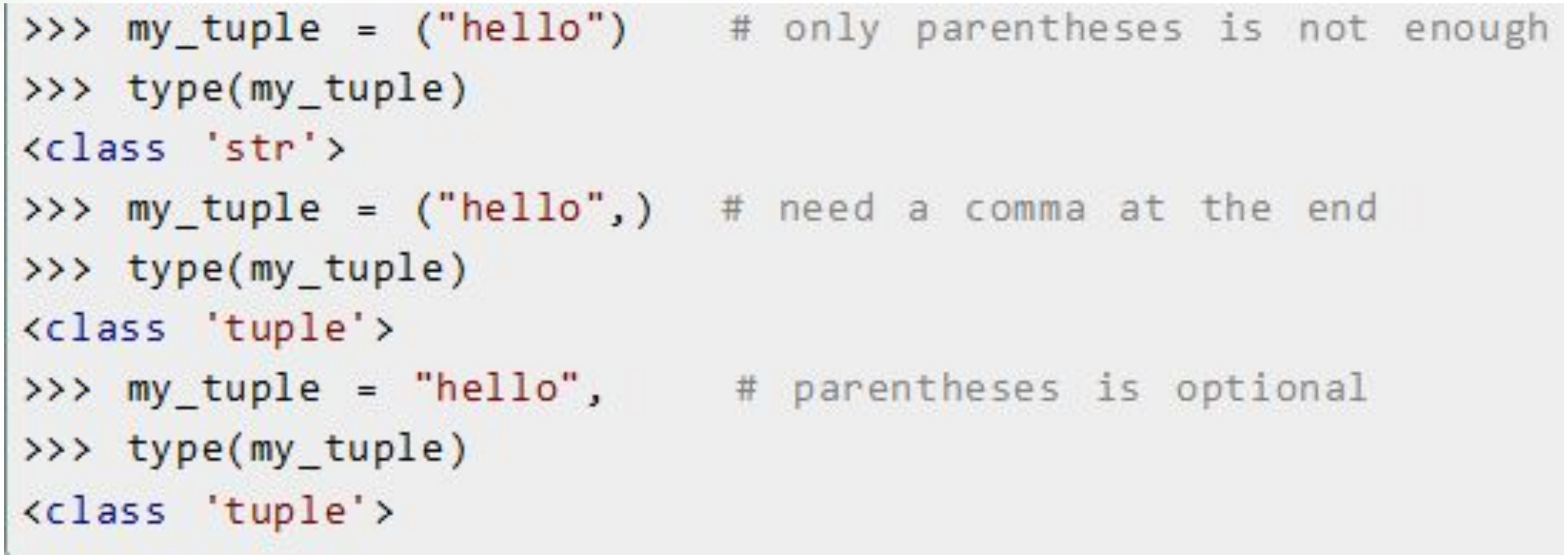
Python Tuples



```
>>> tuple = ('abcd', 786 , 2.23, 'john', 70.2 )
>>> tinytuple = (123, 'john')
>>> print tuple
('abcd', 786, 2.23, 'john', 70.2)
>>> print tuple[0]
abcd
>>> print tuple[1:3]
(786, 2.23)
>>> print tinytuple * 2
(123, 'john', 123, 'john')
>>> print tuple + tinytuple
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

Python Tuples

- Creating a tuple with one element is a bit tricky.
- Having one element within parentheses is not enough.
- We will need a trailing comma to indicate that it is in fact a tuple.

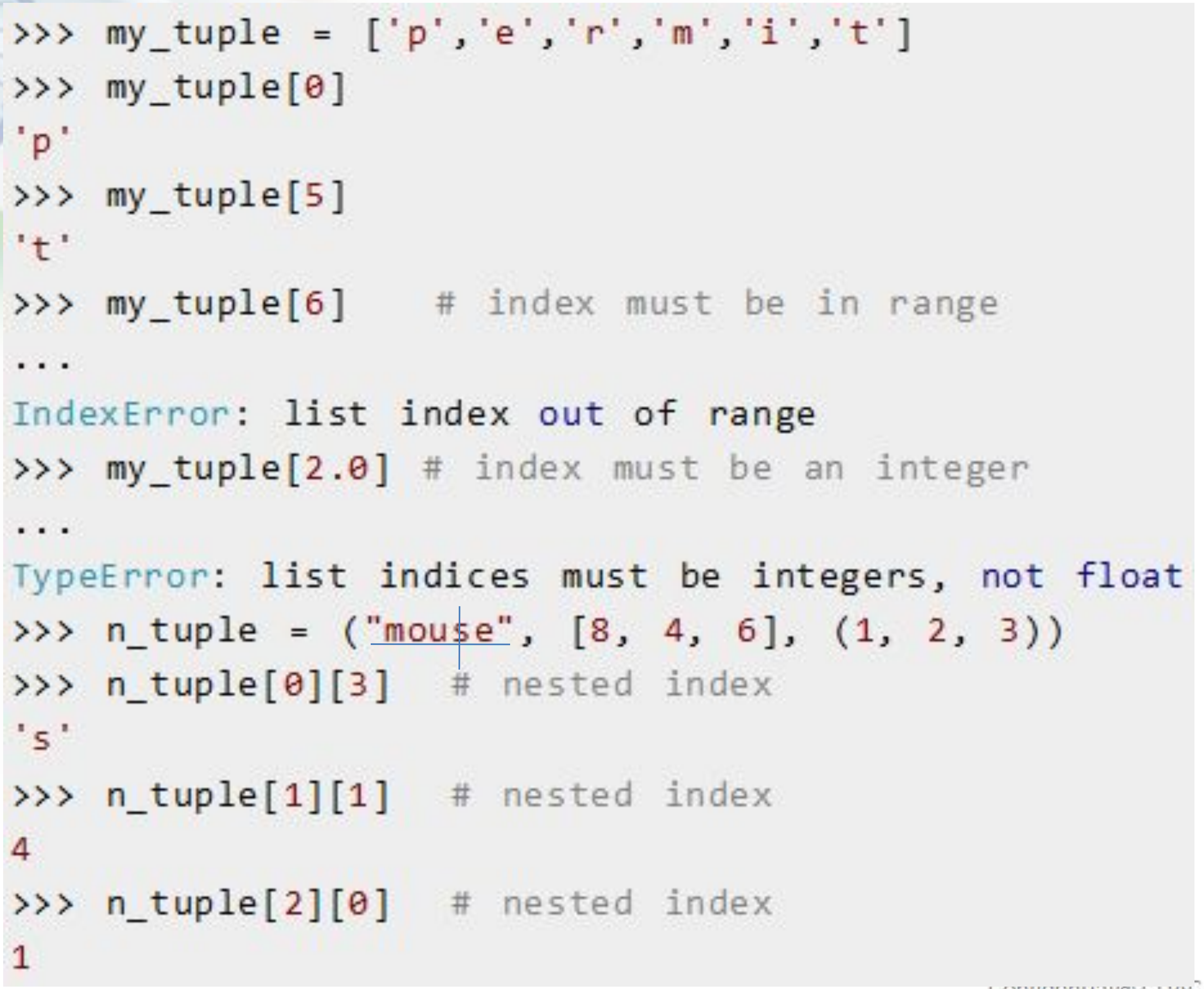


```
>>> my_tuple = ("hello")    # only parentheses is not enough
>>> type(my_tuple)
<class 'str'>
>>> my_tuple = ("hello",)   # need a comma at the end
>>> type(my_tuple)
<class 'tuple'>
>>> my_tuple = "hello",     # parentheses is optional
>>> type(my_tuple)
<class 'tuple'>
```

Accessing Elements in a Tuple

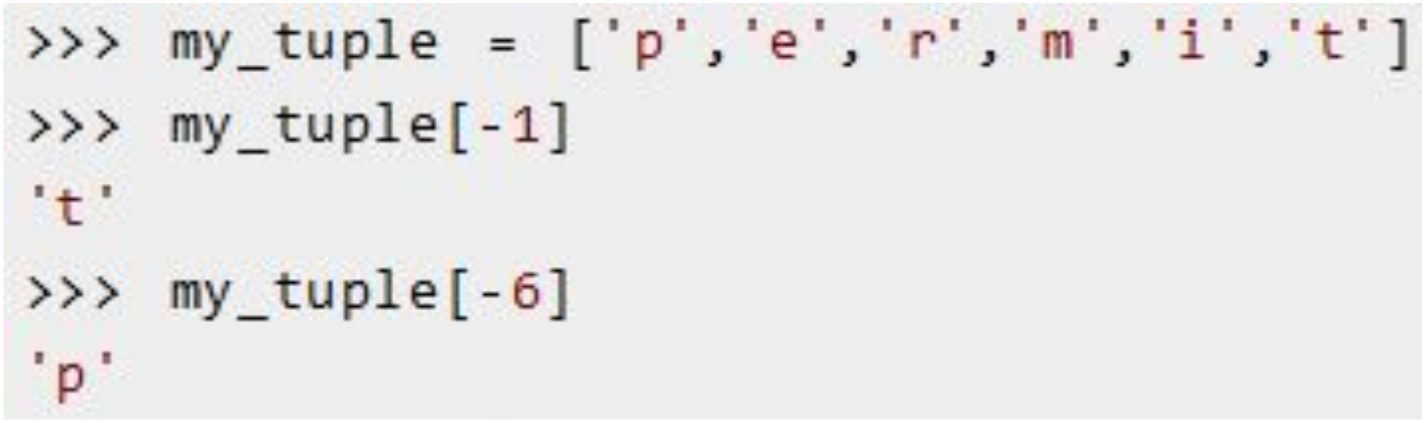
- There are various ways in which we can access the elements of a tuple.
- **Indexing:** We can use the index operator [] to access an item in a tuple.
- Index starts from 0. So, a tuple having 6 elements will have index from 0 to 5.
- Trying to access an element other that this will raise an IndexError.
- The index must be an integer. We can't use float or other types, this will result into TypeError.
- Nested tuple are accessed using nested indexing.

Accessing Elements in a Tuple



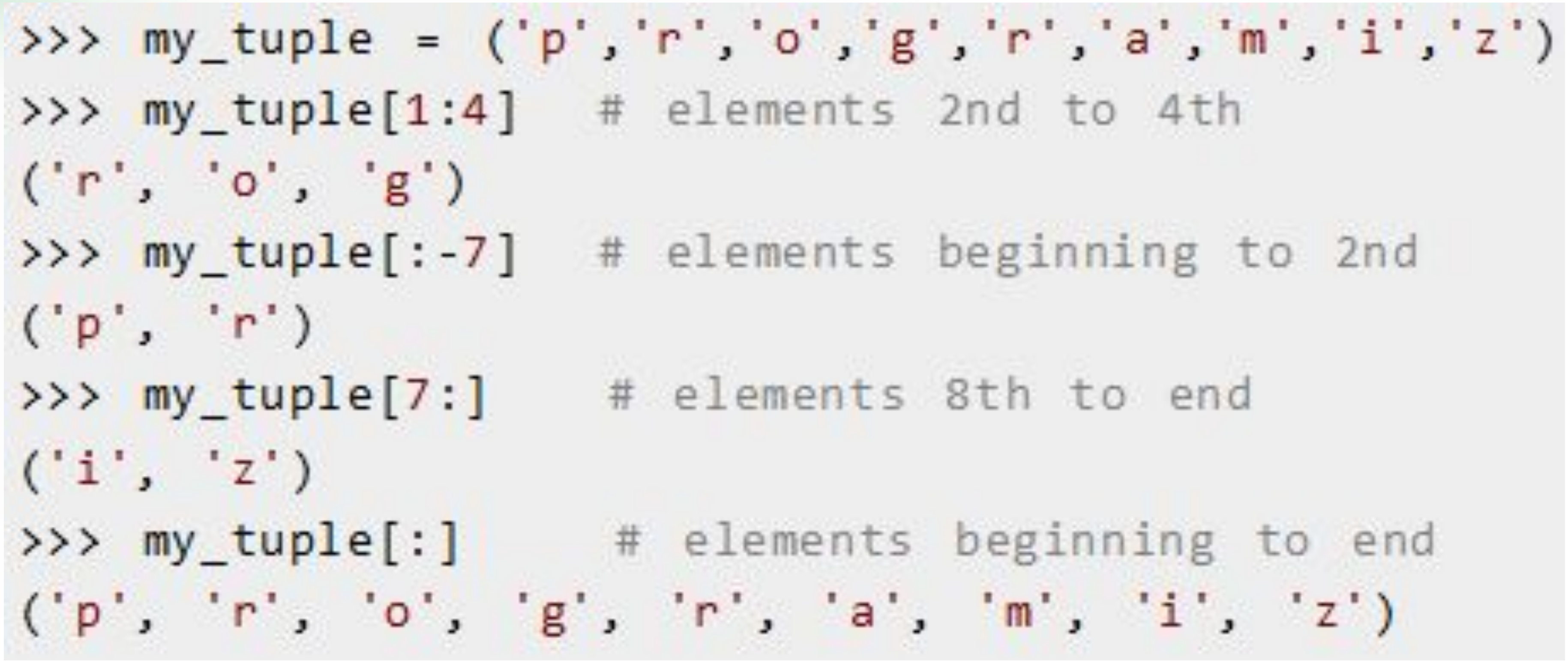
```
>>> my_tuple = ['p','e','r','m','i','t']
>>> my_tuple[0]
'p'
>>> my_tuple[5]
't'
>>> my_tuple[6]    # index must be in range
...
IndexError: list index out of range
>>> my_tuple[2.0] # index must be an integer
...
TypeError: list indices must be integers, not float
>>> n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
>>> n_tuple[0][3]   # nested index
's'
>>> n_tuple[1][1]   # nested index
4
>>> n_tuple[2][0]   # nested index
1
```

- **Negative Indexing:** Python allows negative indexing for its sequences.
- The index of -1 refers to the last item, -2 to the second last item and so on



```
>>> my_tuple = ['p','e','r','m','i','t']
>>> my_tuple[-1]
't'
>>> my_tuple[-6]
'p'
```

- **Slicing:** We can access a range of items in a tuple by using the slicing operator (colon).

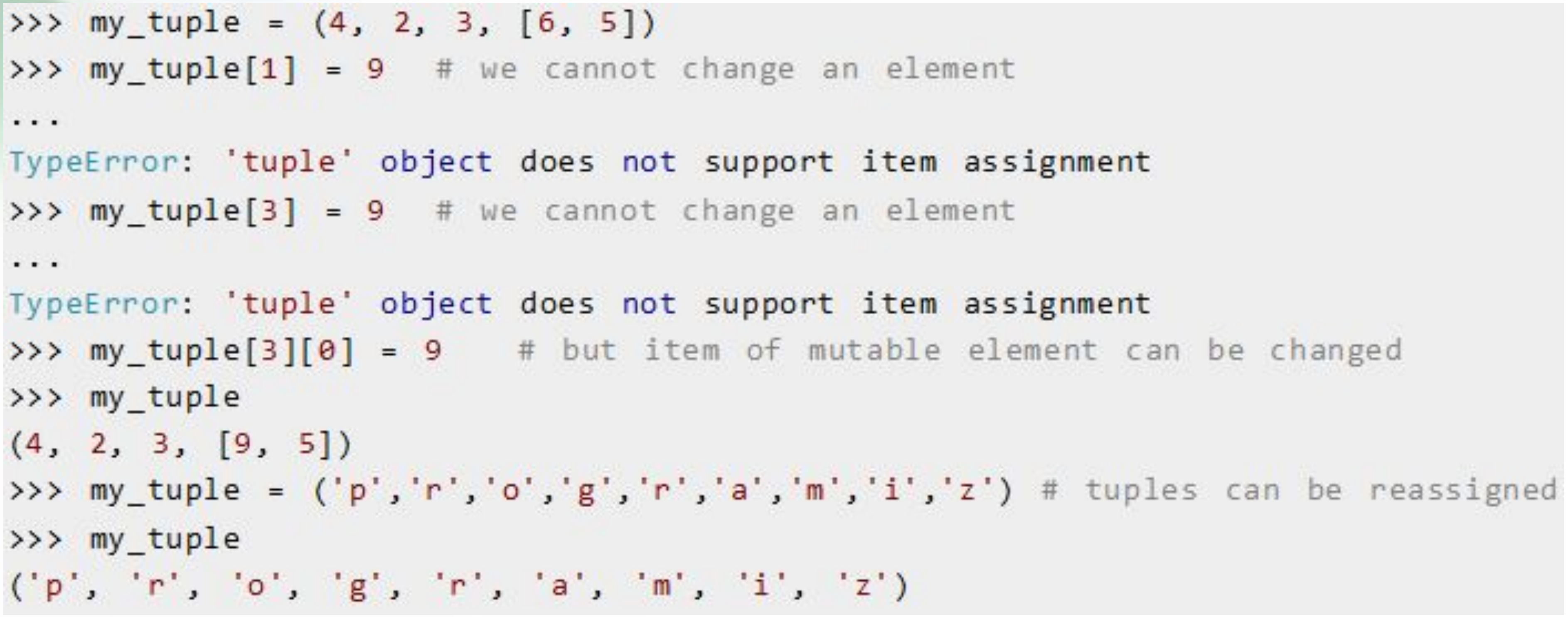


```
>>> my_tuple = ('p','r','o','g','r','a','m','i','z')
>>> my_tuple[1:4]   # elements 2nd to 4th
('r', 'o', 'g')
>>> my_tuple[:-7]   # elements beginning to 2nd
('p', 'r')
>>> my_tuple[7:]    # elements 8th to end
('i', 'z')
>>> my_tuple[:]     # elements beginning to end
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

Changing or Deleting a Tuple

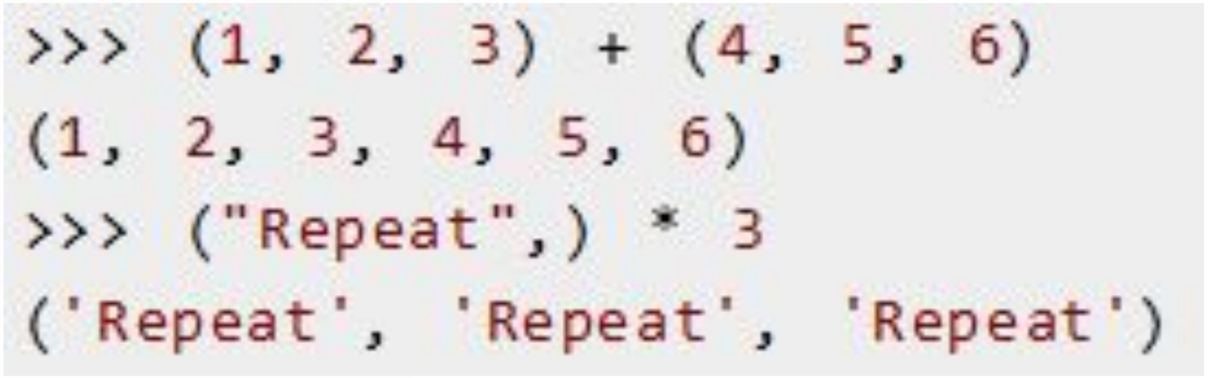
- Unlike lists, tuples are immutable.
- This means that elements of a tuple cannot be changed once it has been assigned.
- But if the element is itself a mutable datatype like list, its nested items can be changed.
- We can also assign a tuple to different values (reassignment).

Changing or Deleting a Tuple



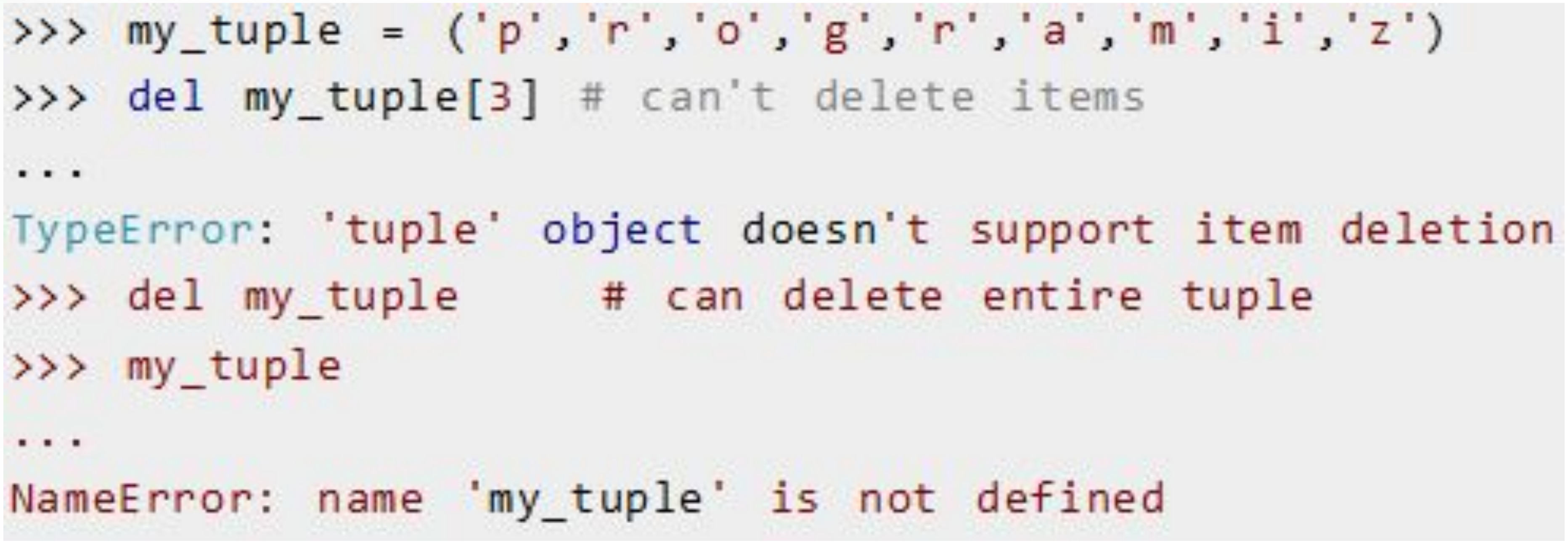
```
>>> my_tuple = (4, 2, 3, [6, 5])
>>> my_tuple[1] = 9  # we cannot change an element
...
TypeError: 'tuple' object does not support item assignment
>>> my_tuple[3] = 9  # we cannot change an element
...
TypeError: 'tuple' object does not support item assignment
>>> my_tuple[3][0] = 9   # but item of mutable element can be changed
>>> my_tuple
(4, 2, 3, [9, 5])
>>> my_tuple = ('p','r','o','g','r','a','m','i','z') # tuples can be reassigned
>>> my_tuple
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

- We can use + operator to combine two tuples. This is also called concatenation.
- The * operator repeats a tuple for the given number of times. These operations result into a new tuple.



```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> ("Repeat",) * 3
('Repeat', 'Repeat', 'Repeat')
```

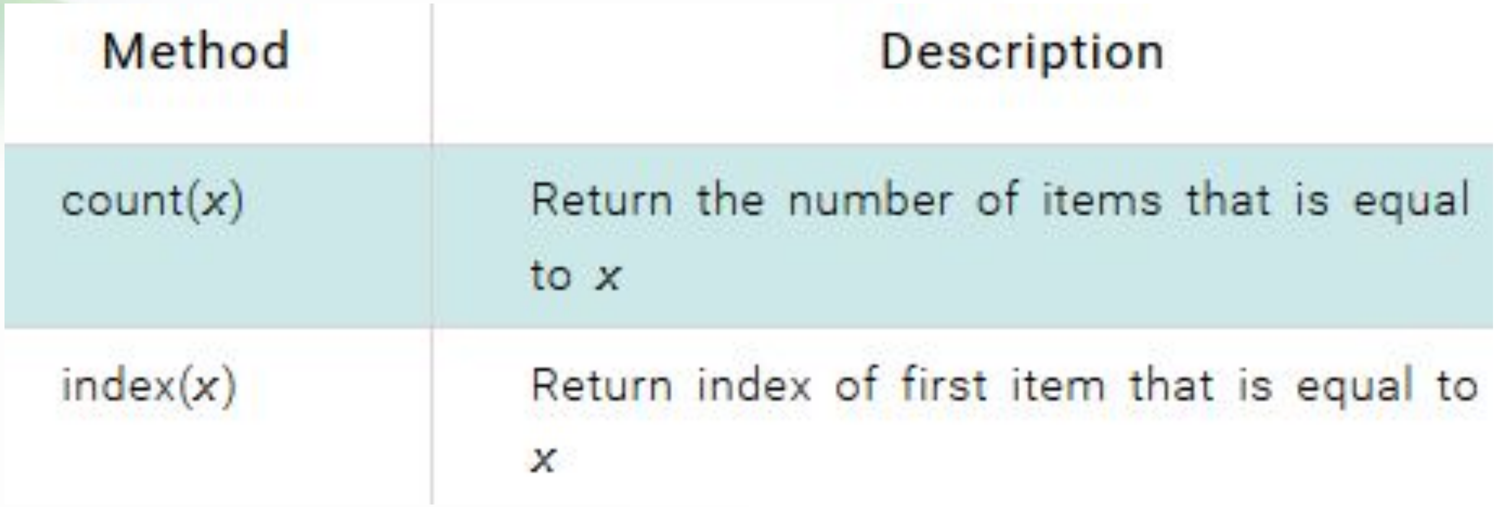
- We cannot delete or remove items from a tuple.
- But deleting the tuple entirely is possible using the keyword del.



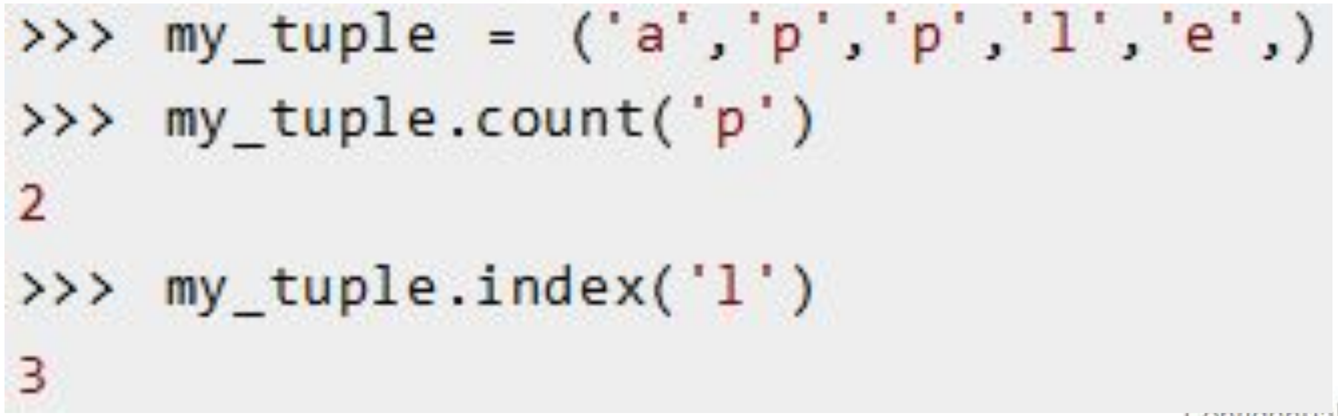
```
>>> my_tuple = ('p','r','o','g','r','a','m','i','z')
>>> del my_tuple[3] # can't delete items
...
TypeError: 'tuple' object doesn't support item deletion
>>> del my_tuple      # can delete entire tuple
>>> my_tuple
...
NameError: name 'my_tuple' is not defined
```

Python Tuple Methods

- Methods that add items or remove items are not available with tuple.

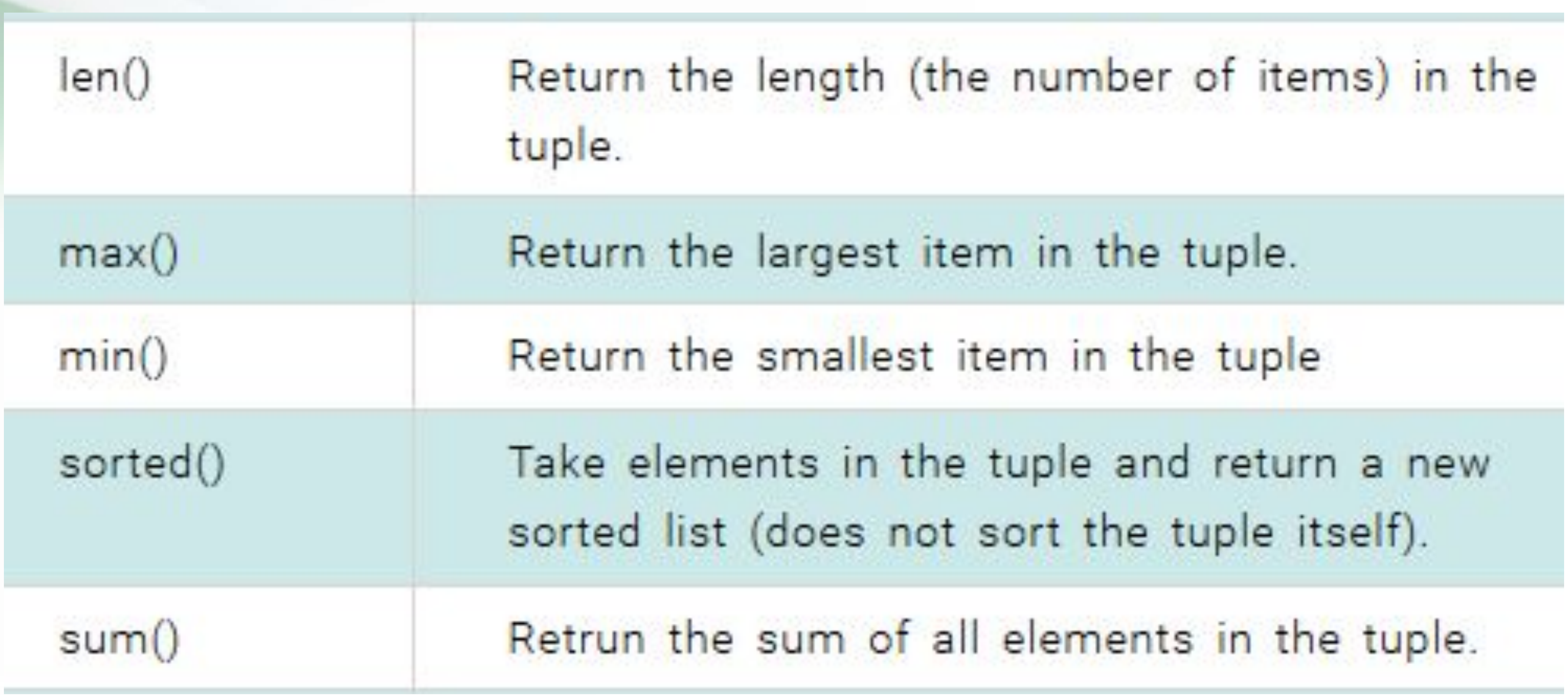


Method	Description
count(*x*)	Return the number of items that is equal to *x*
index(*x*)	Return index of first item that is equal to *x*



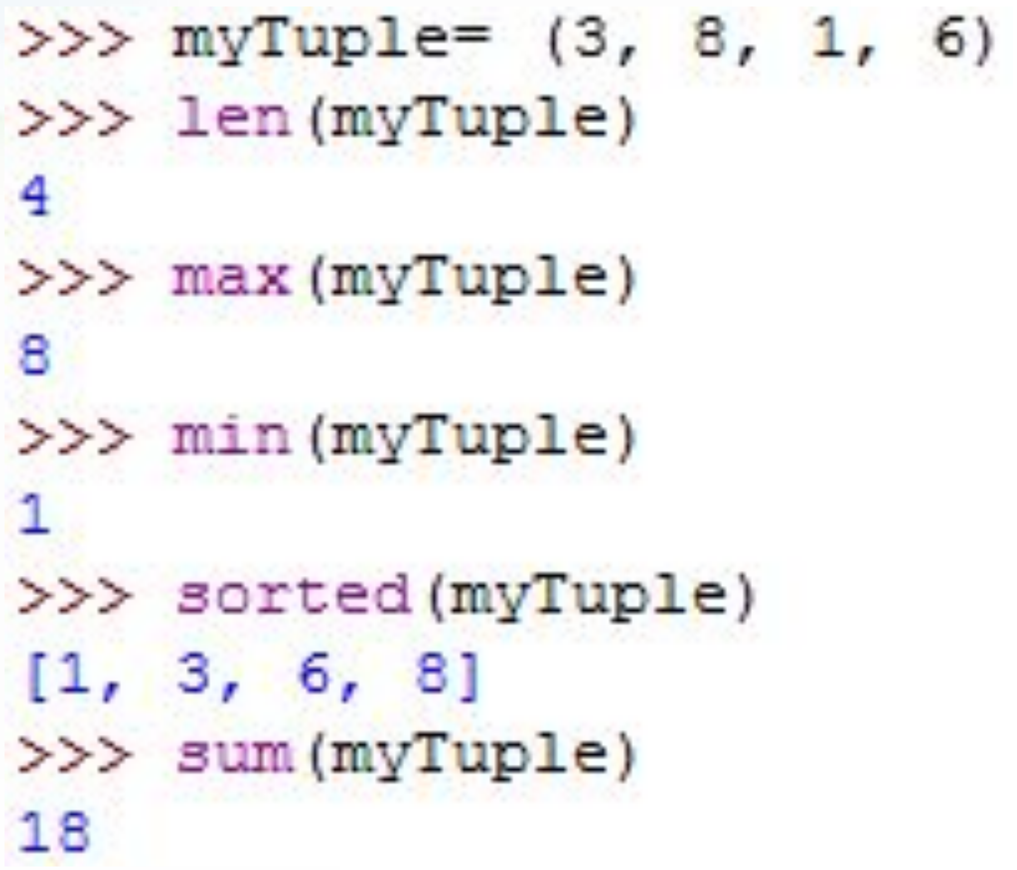
```
>>> my_tuple = ('a','p','p','l','e',)
>>> my_tuple.count('p')
2
>>> my_tuple.index('l')
3
```

Built-in Functions with Tuple



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()	Retrun the sum of all elements in the tuple.

Built-in Functions with Tuple



```
>>> myTuple= (3, 8, 1, 6)
>>> len(myTuple)
4
>>> max(myTuple)
8
>>> min(myTuple)
1
>>> sorted(myTuple)
[1, 3, 6, 8]
>>> sum(myTuple)
18
```

Advantage of Tuple over List

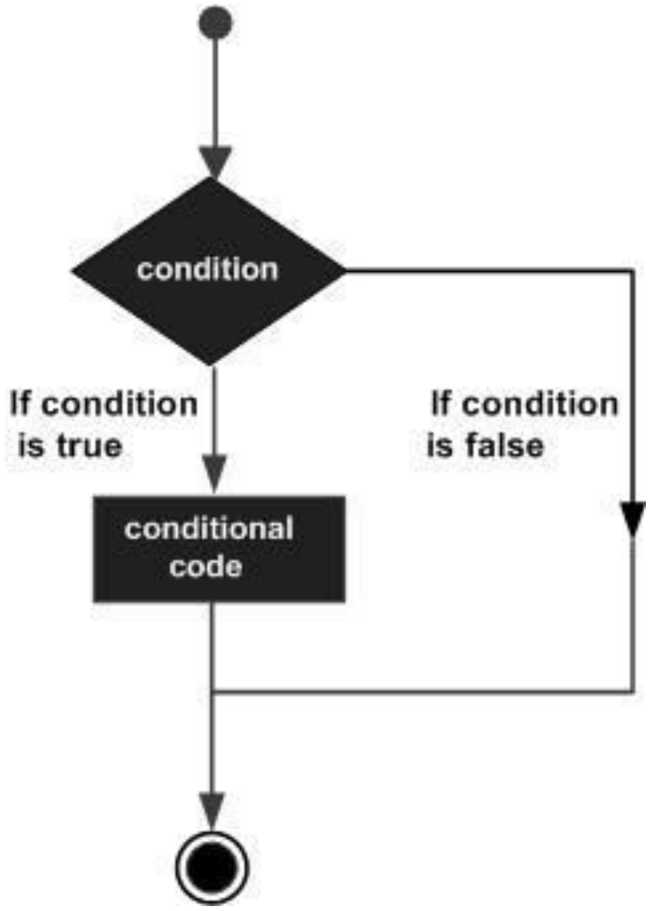
- Tuples and list look quite similar except the fact that one is immutable and the other is mutable.
- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- There are some advantages of implementing a tuple than a list. Here are a few of them:

1. Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
2. Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
3. If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Decision making in Python

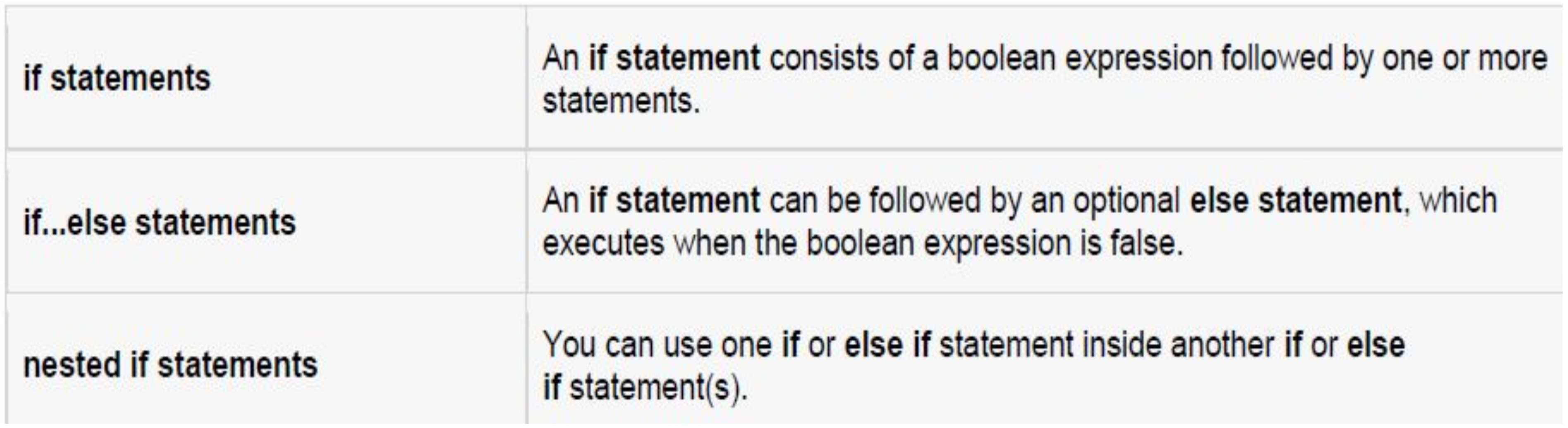
- Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

General form of a typical decision making structure



- Python programming language assumes any non-zero and non-null values as true, and if it is either zero or null, then it is assumed as false value.

Types of decision making statements

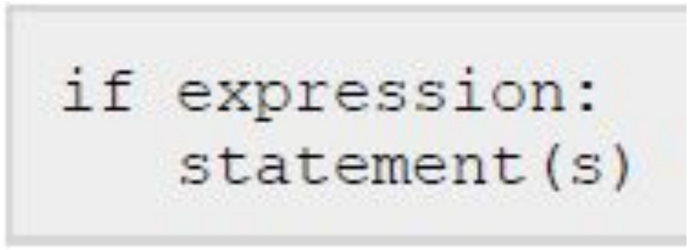


if statements	An **if statement** consists of a boolean expression followed by one or more statements.
if...else statements	An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.
nested if statements	You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

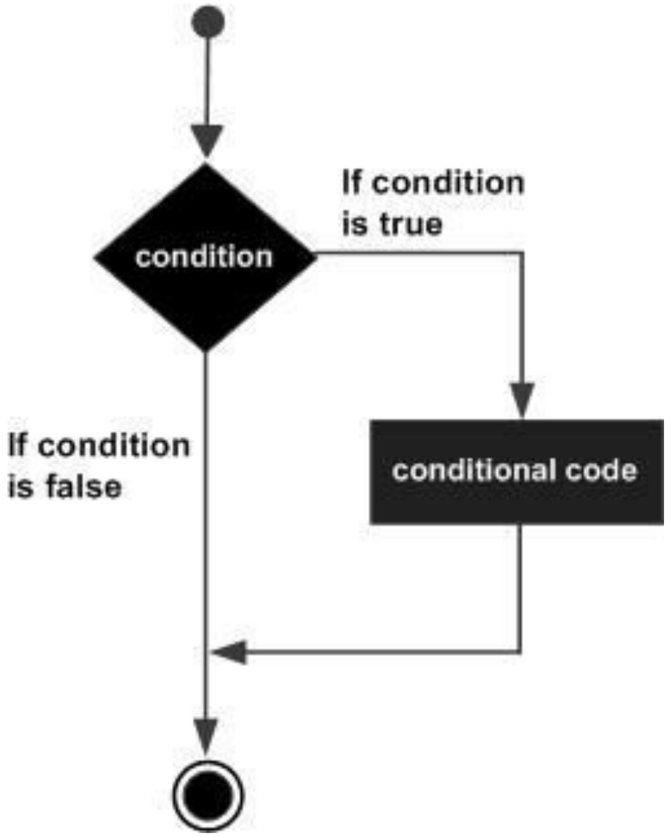
If statements

- The if statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

Syntax:



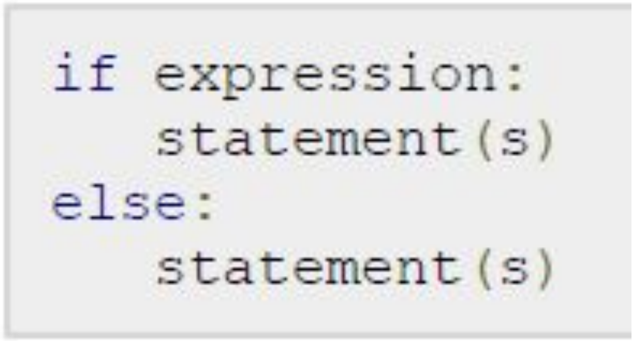
```
if expression:
   statement(s)
```



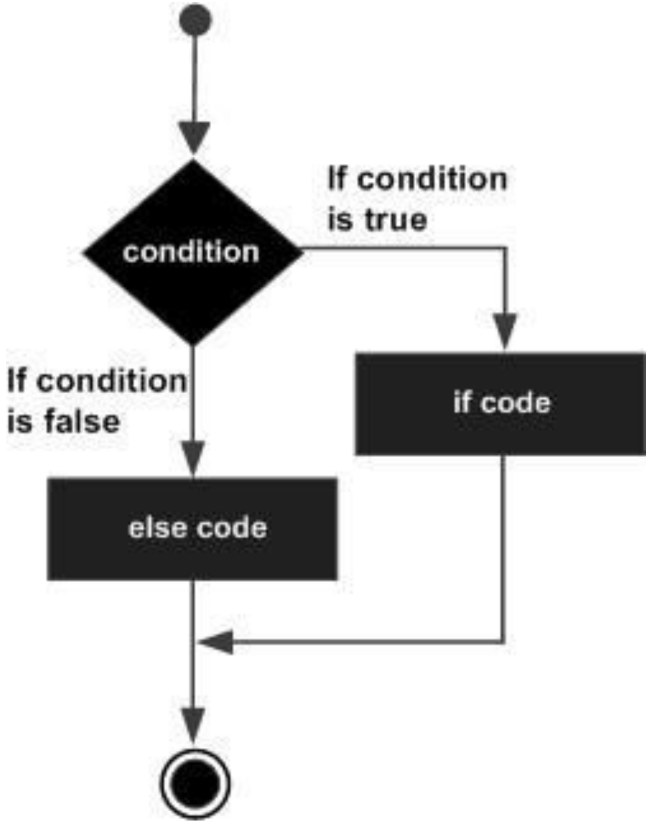
If-else statements

- An else statement can be combined with an if statement. An else statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a false value.

Syntax:



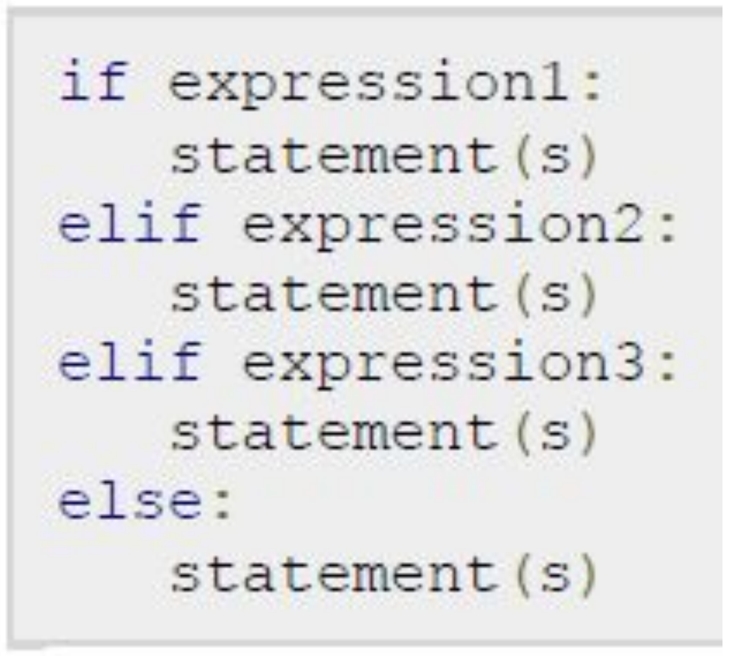
```
if expression:
   statement(s)
else:
   statement(s)
```



The *elif Statement*

- The elif statement allows you to check multiple expressions for truth value and execute a block of code as soon as one of the conditions evaluates to true.

Syntax:

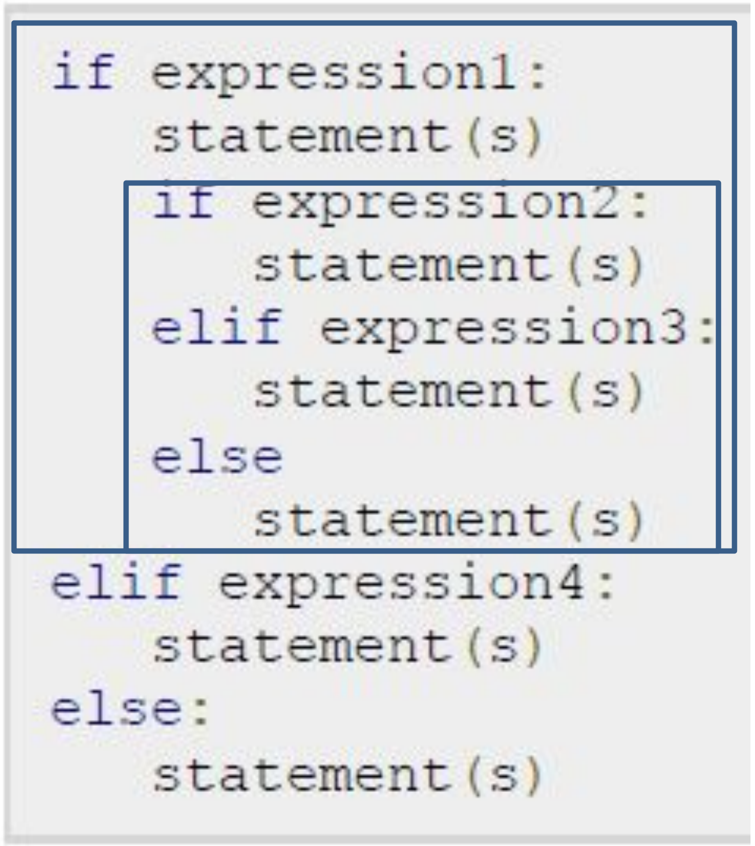


```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```

Nested if statements

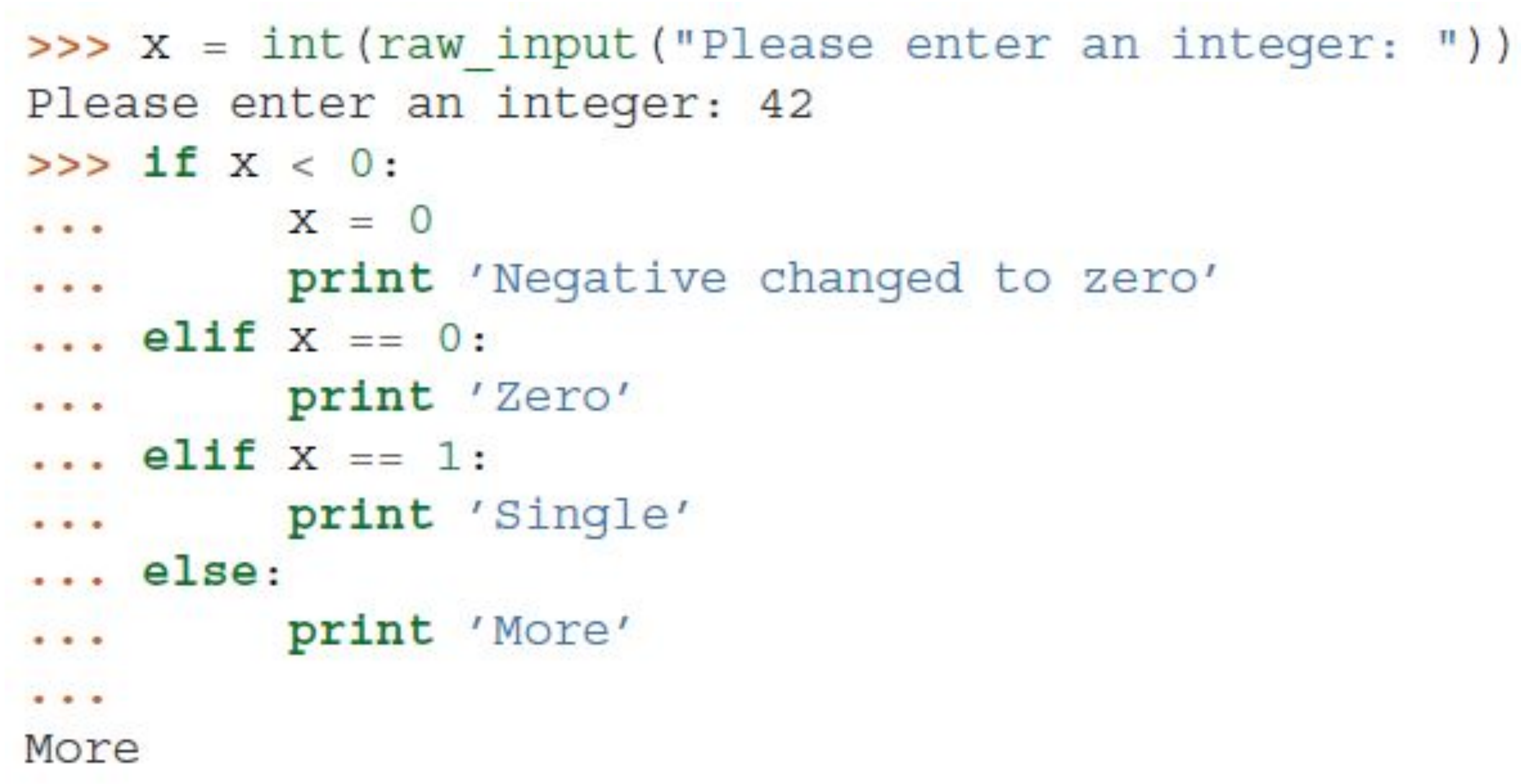
- There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested if construct.

Syntax:



```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)
   else
      statement(s)
elif expression4:
   statement(s)
else:
   statement(s)
```

Example

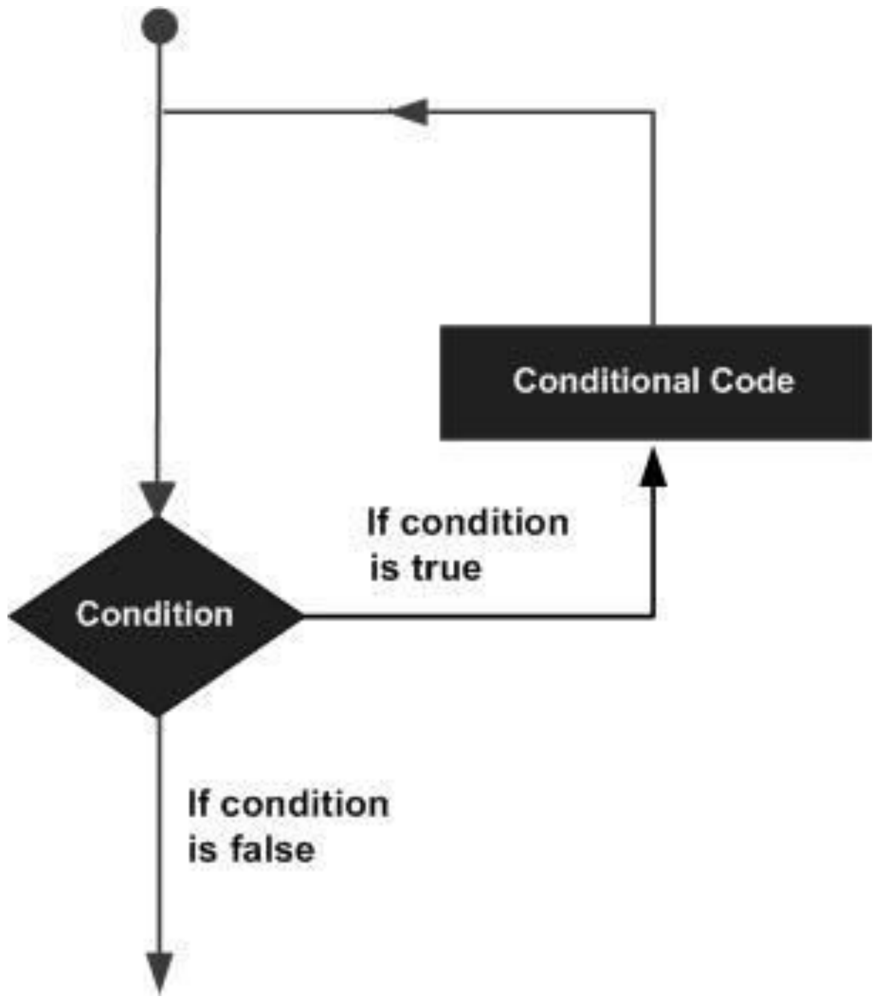


```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Single'
... else:
...      print 'More'
...
More
```

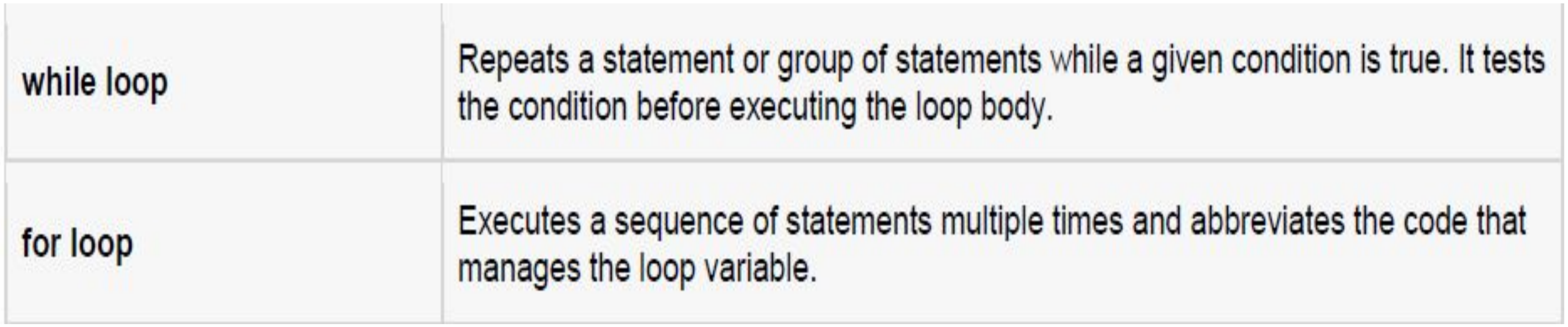
Loops in Python

- There may be a situation when you need to execute a block of code several number of times.
- In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.
- Programming languages provide various control structures that allow for more complicated execution paths.
- A loop statement allows us to execute a statement or group of statements multiple times

General form of a loop statement



Types of loops

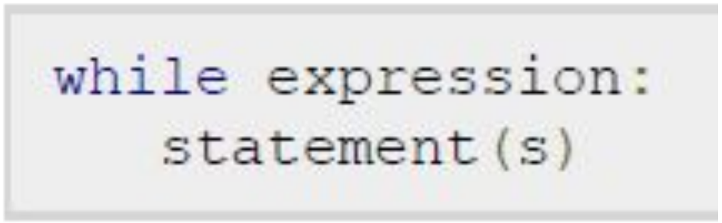


while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

while loop

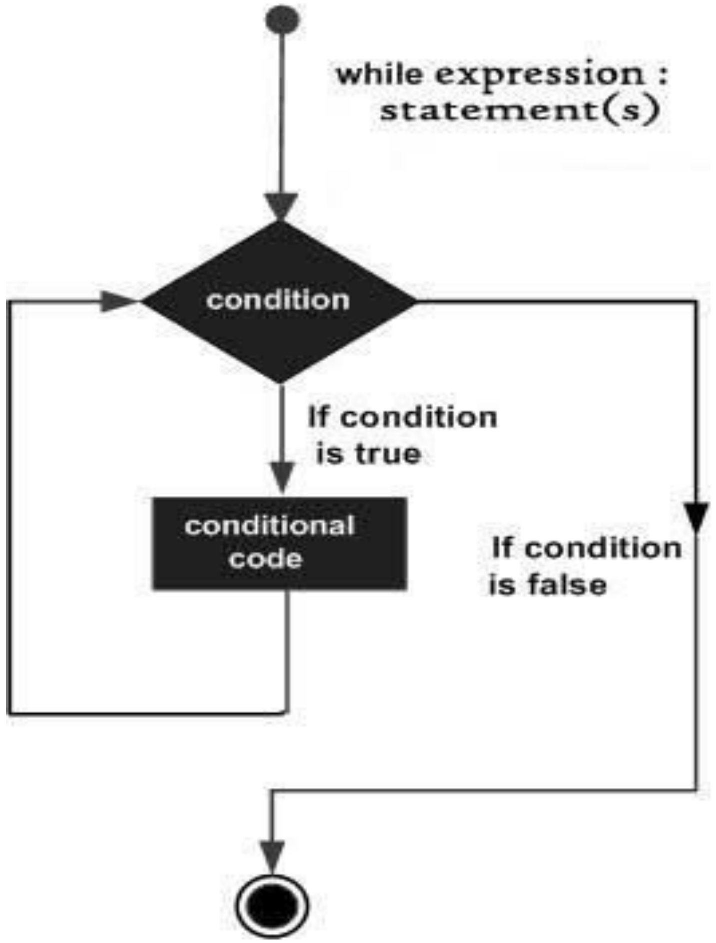
- A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true

Syntax:

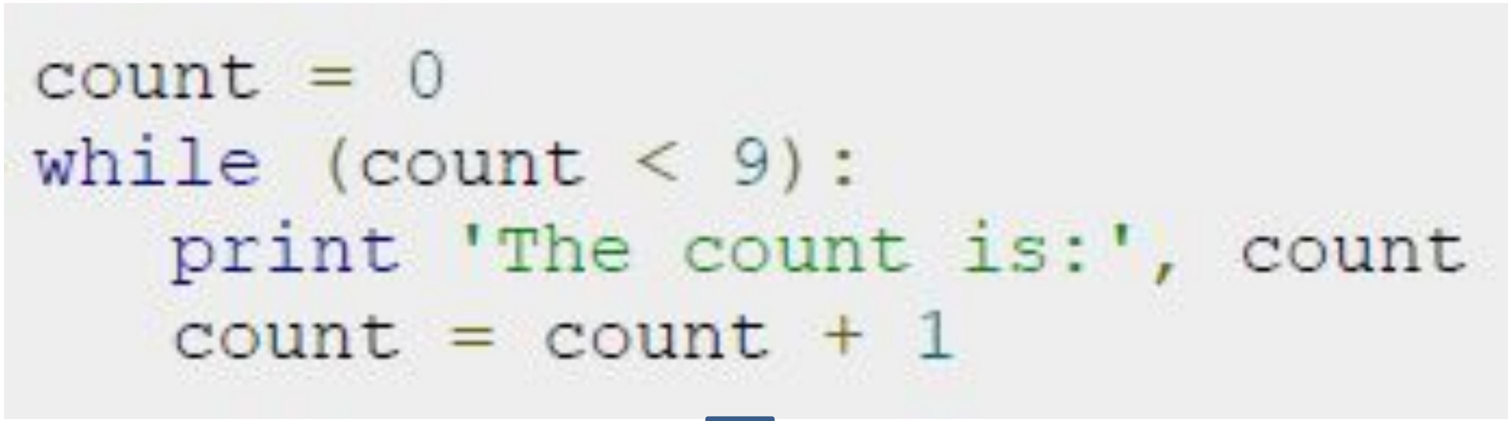


```
while expression:
    statement(s)
```

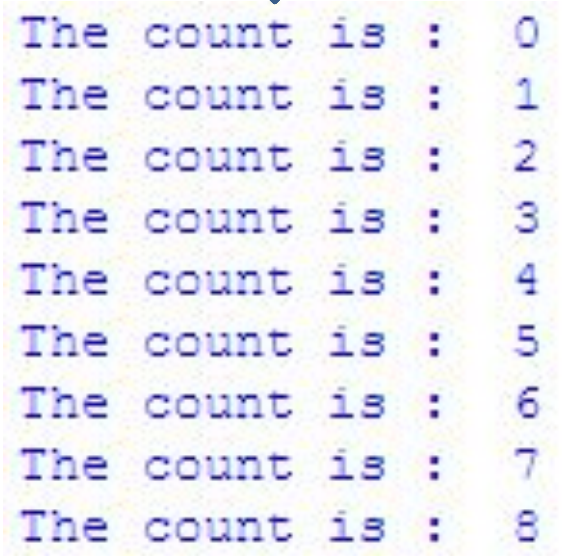
Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.



Example



```
count = 0
while (count < 9):
   print 'The count is:', count
   count = count + 1
```



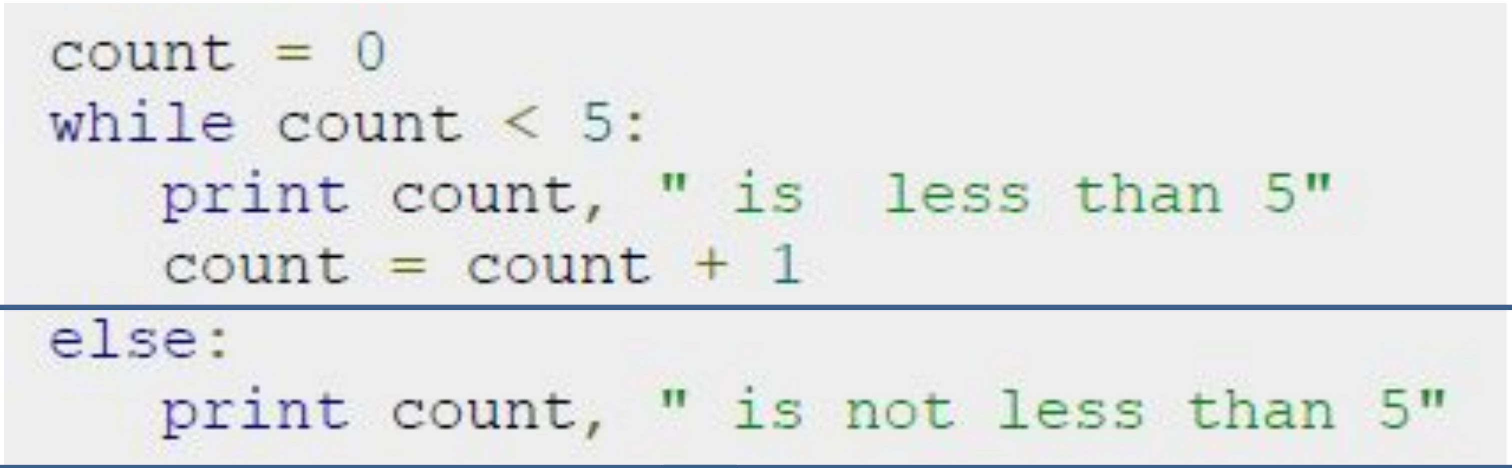
```
The count is :  0
The count is :  1
The count is :  2
The count is :  3
The count is :  4
The count is :  5
The count is :  6
The count is :  7
The count is :  8
```

The else Statement Used with Loops

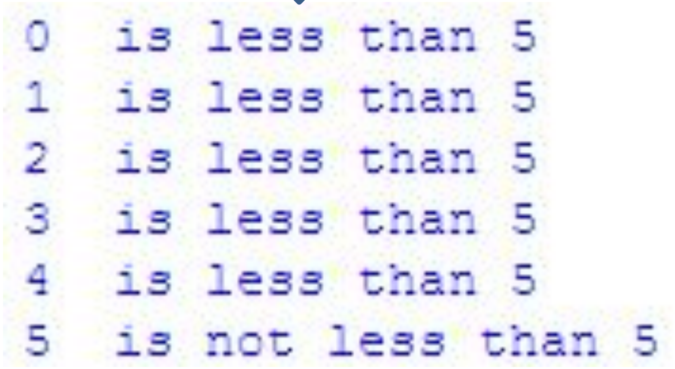
Python supports to have an else statement associated with a loop statement.

- If the else statement is used with a **for loop**, the else statement is executed when the loop has exhausted iterating the list.
- If the else statement is used with a **while loop**, the else statement is executed when the condition becomes false.

Example



```
count = 0
while count < 5:
   print count, " is  less than 5"
   count = count + 1
else:
   print count, " is not less than 5"
```

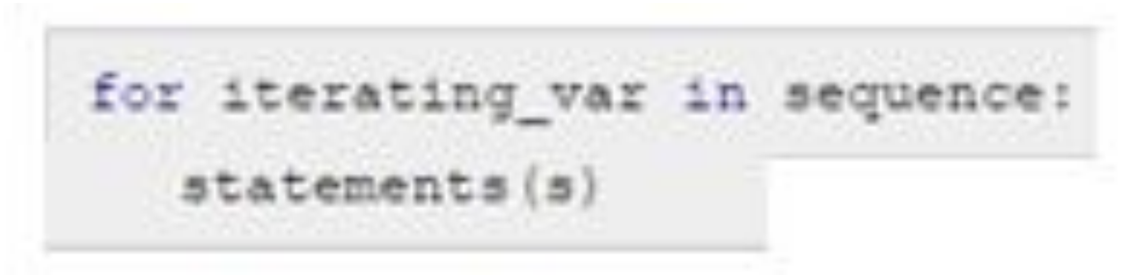


```
0  is less than 5
1  is less than 5
2  is less than 5
3  is less than 5
4  is less than 5
5  is not less than 5
```

for loop

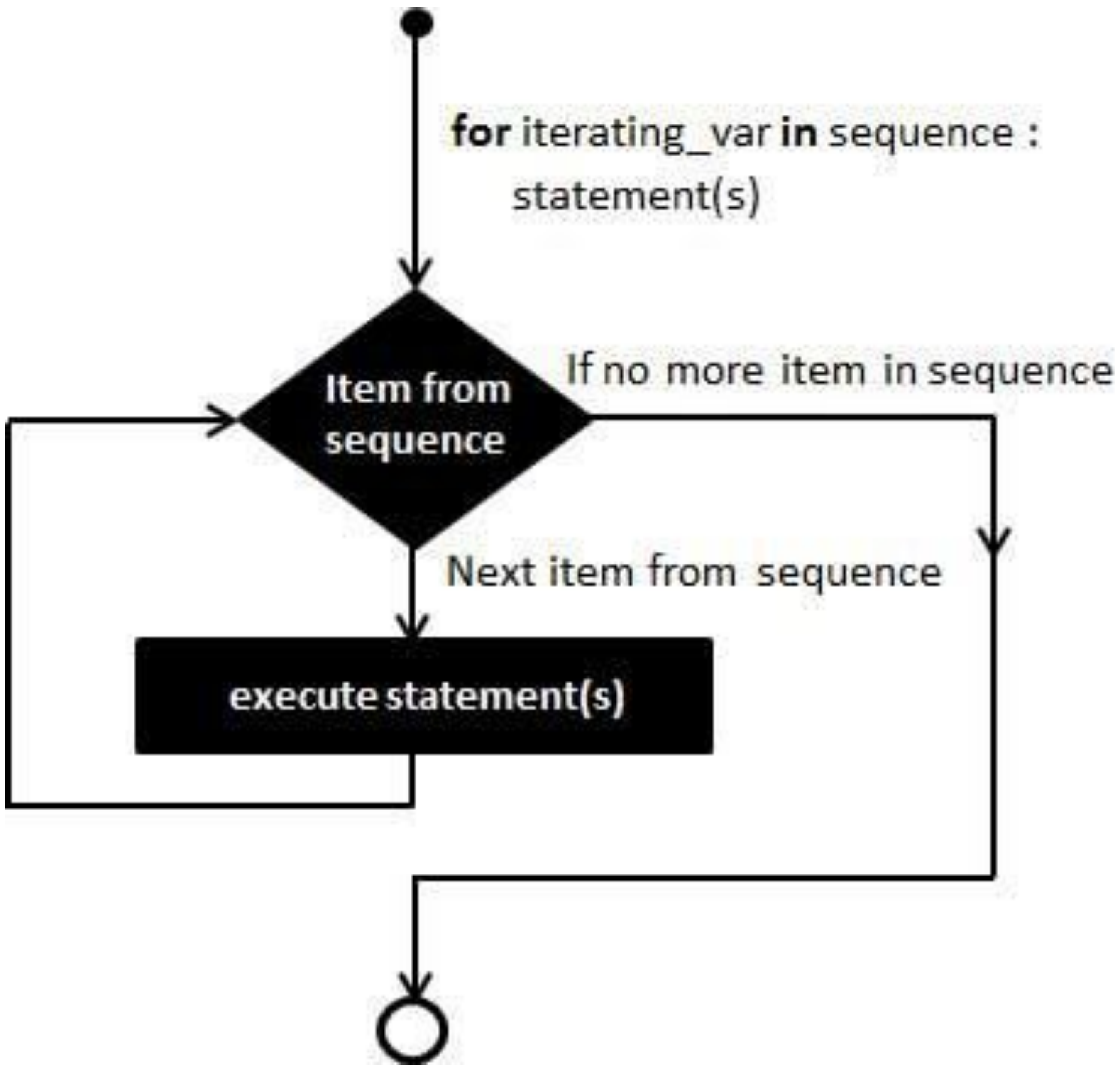
- The for loop in Python has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax:

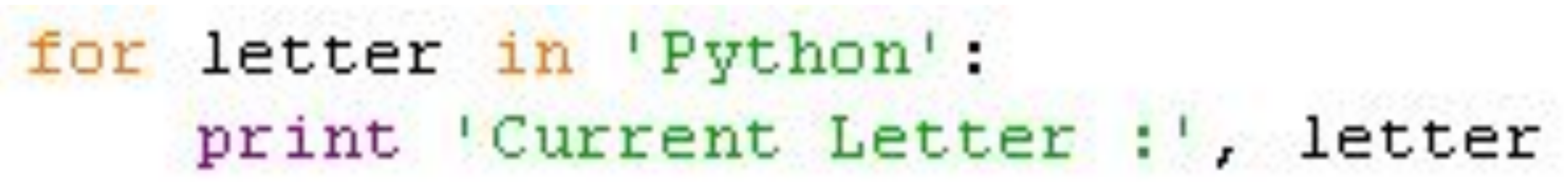


```
for iterating_var in sequence:
    statements(s)
```

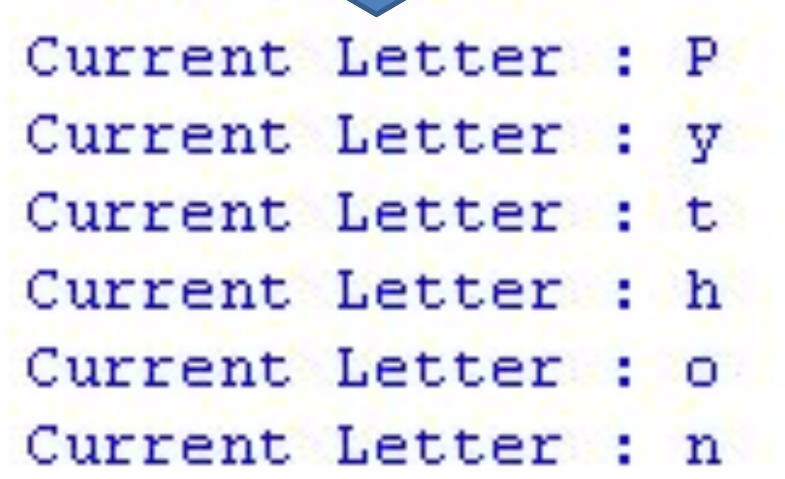
- If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable iterating_var. Next, the statements block is executed.
- Each item in the list is assigned to iterating_var, and the statement(s) block is executed until the entire sequence is exhausted.


for iterating_var in sequence :
statement(s)
Item from sequence
If no more item in sequence
Next item from sequence
execute statement(s)

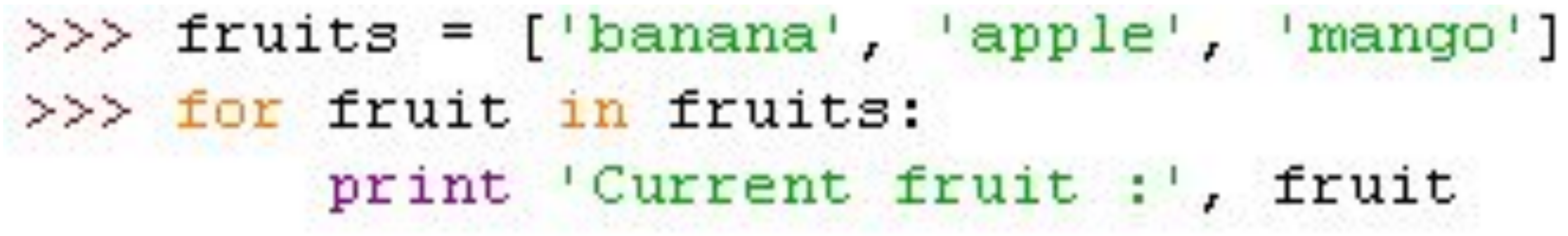
Example



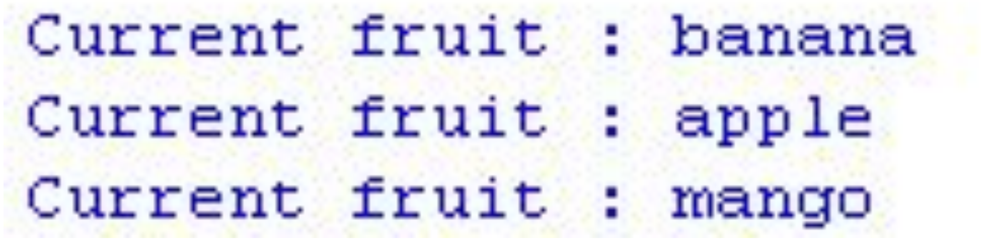
```
for letter in 'Python':
    print 'Current Letter :', letter
```



```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
```



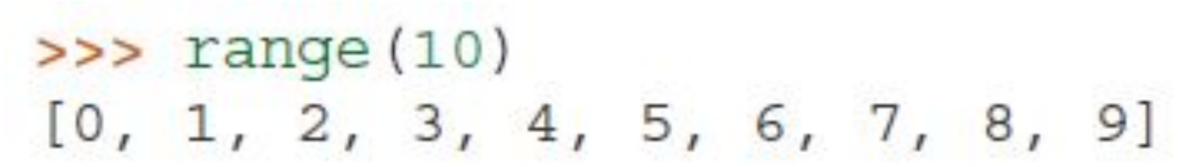
```
>>> fruits = ['banana', 'apple', 'mango']
>>> for fruit in fruits:
        print 'Current fruit :', fruit
```



```
Current fruit : banana
Current fruit : apple
Current fruit : mango
```

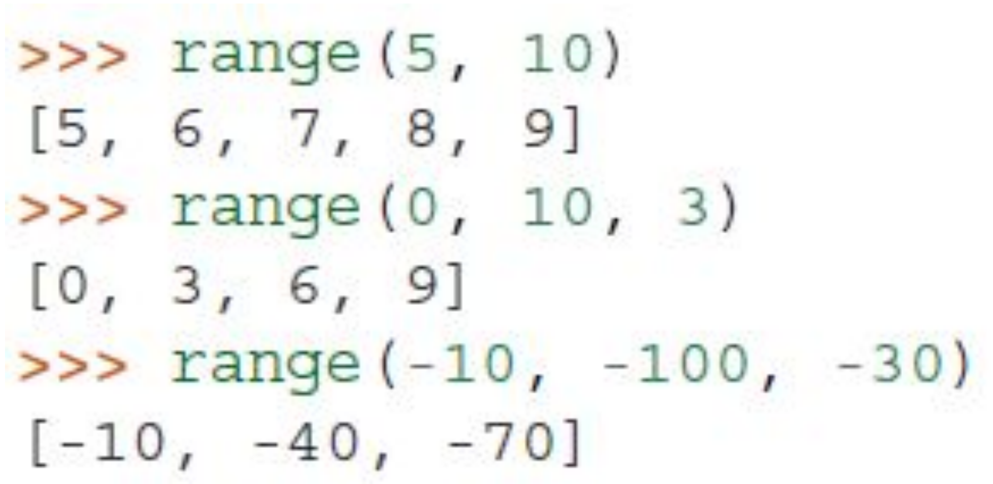
The range() Function

- The built-in function range() is used to iterate over a sequence of numbers

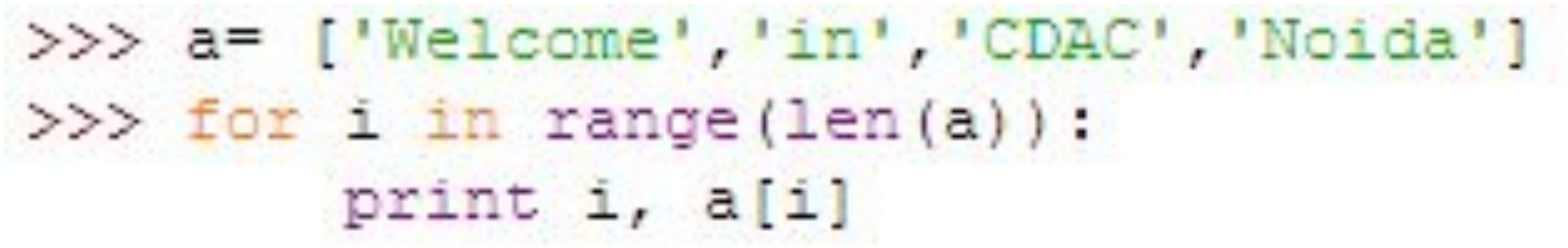


```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

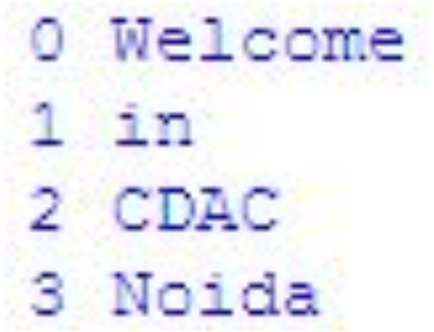
- It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):



```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```



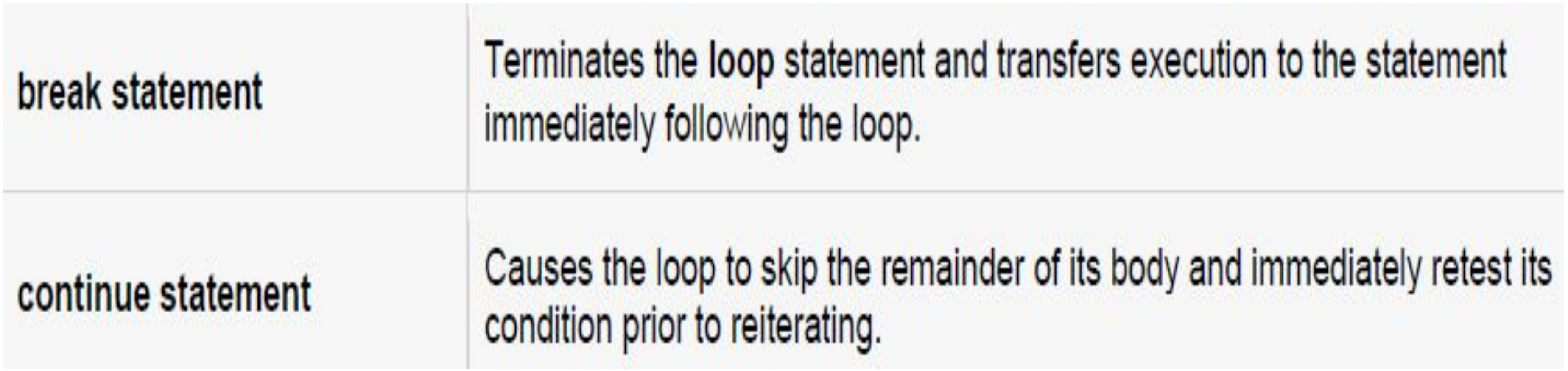
```
>>> a= ['Welcome','in','CDAC','Noida']
>>> for i in range(len(a)):
        print i, a[i]
```



```
0 Welcome
1 in
2 CDAC
3 Noida
```

Loop Control Statements

- Loop control statements change execution from its normal sequence.



break statement	Terminates the **loop** statement and transfers execution to the statement immediately following the loop.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

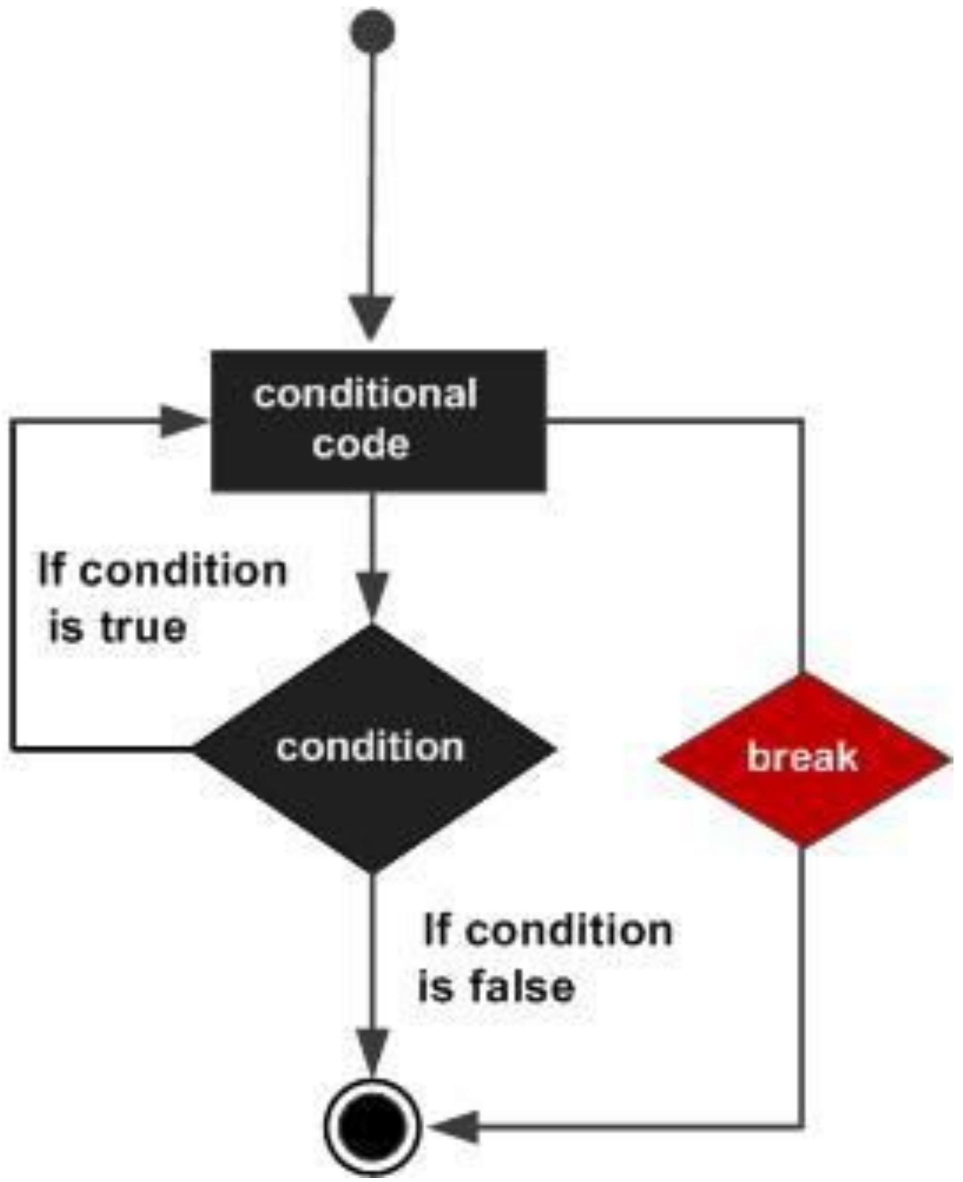
break statement

- The break statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C.
- The break statement can be used in both *while and for loops.*
- If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

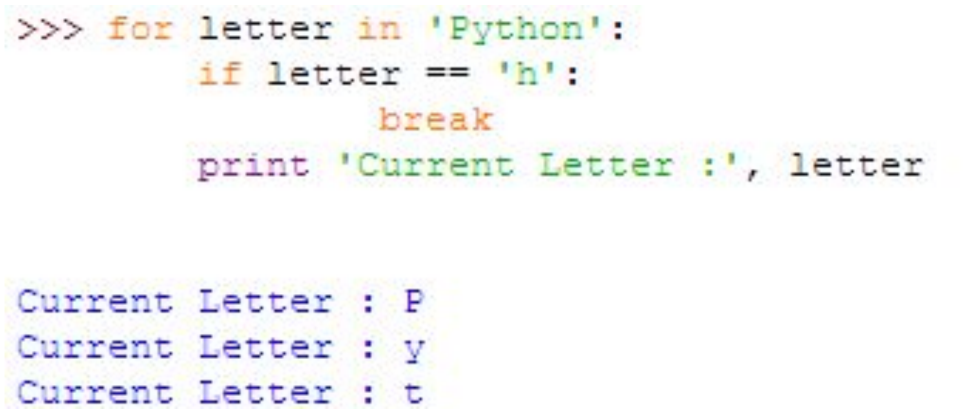
Syntax:

```
break
```

Flow Diagram of break

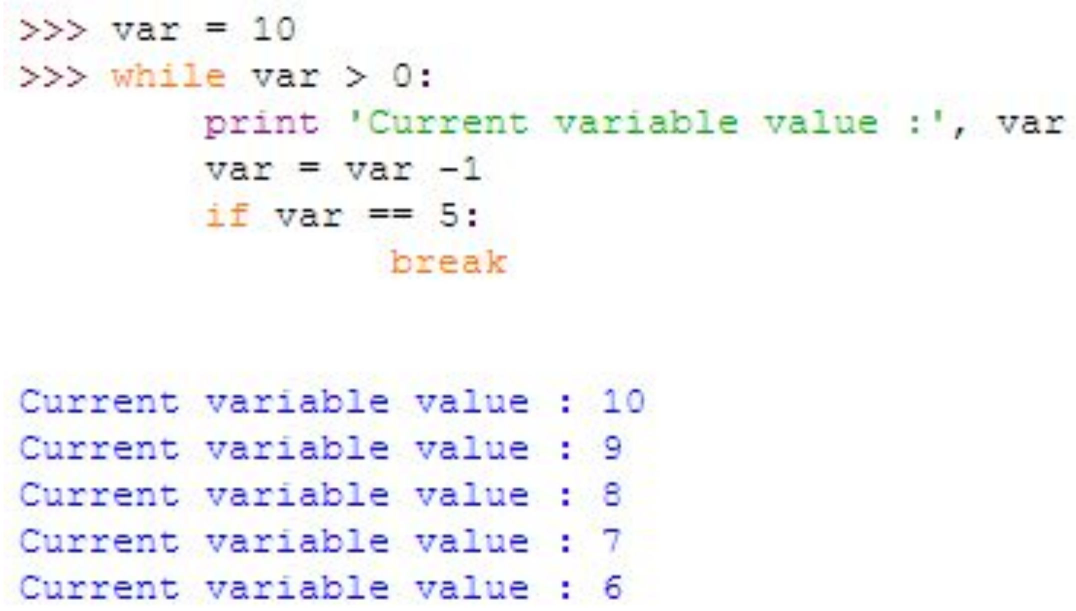


Example



```
>>> for letter in 'Python':
        if letter == 'h':
            break
        print 'Current Letter :', letter


Current Letter : P
Current Letter : y
Current Letter : t
```

continue

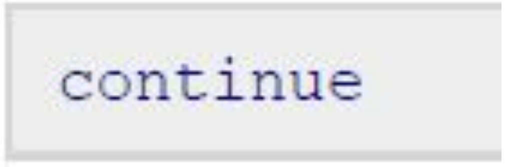


```
>>> var = 10
>>> while var > 0:
        print 'Current variable value :', var
        var = var -1
        if var == 5:
            break


Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
```

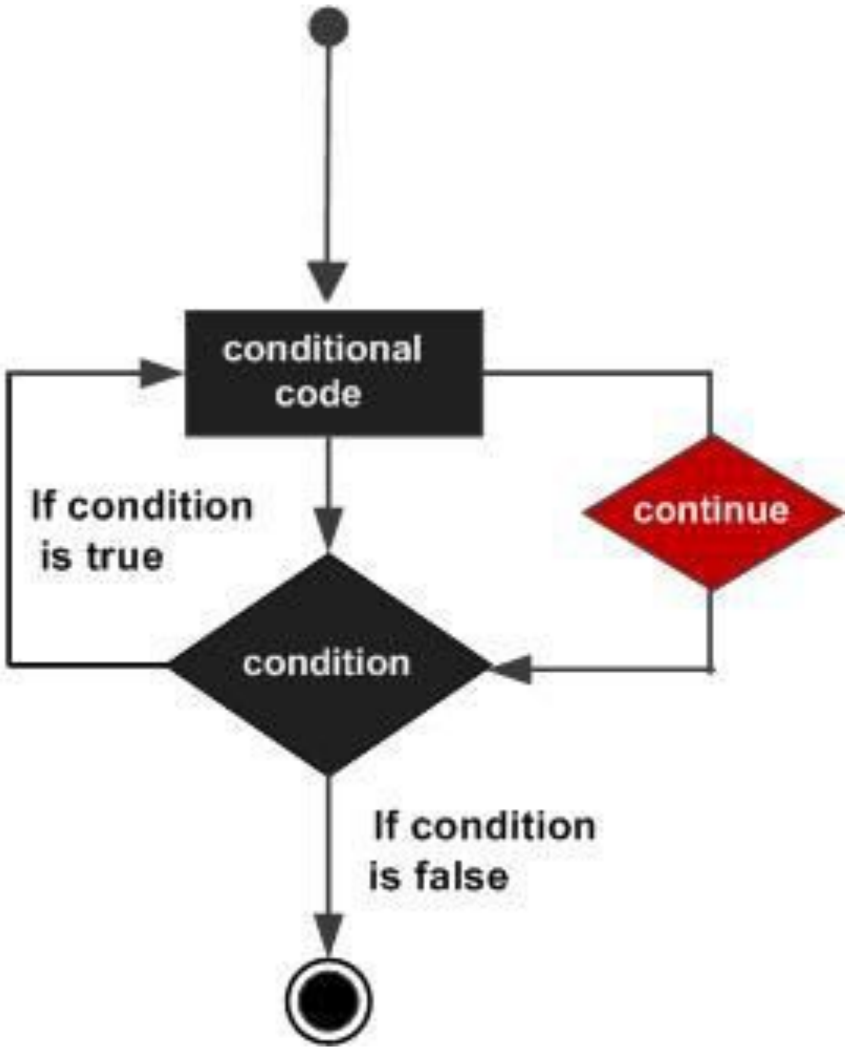
continue statement

- The continue statement in Python returns the control to the beginning of the while loop.
- The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.
- The continue statement can be used in both *while and for loops.*

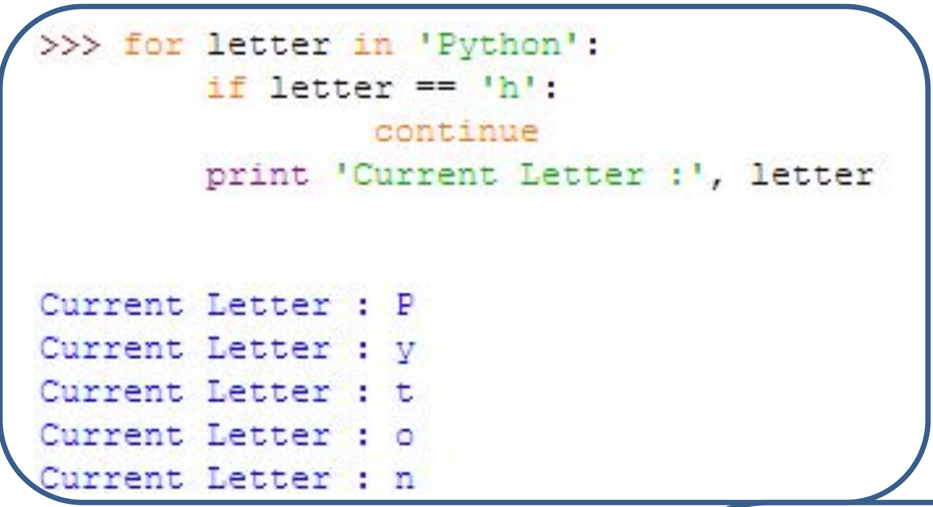
Syntax:



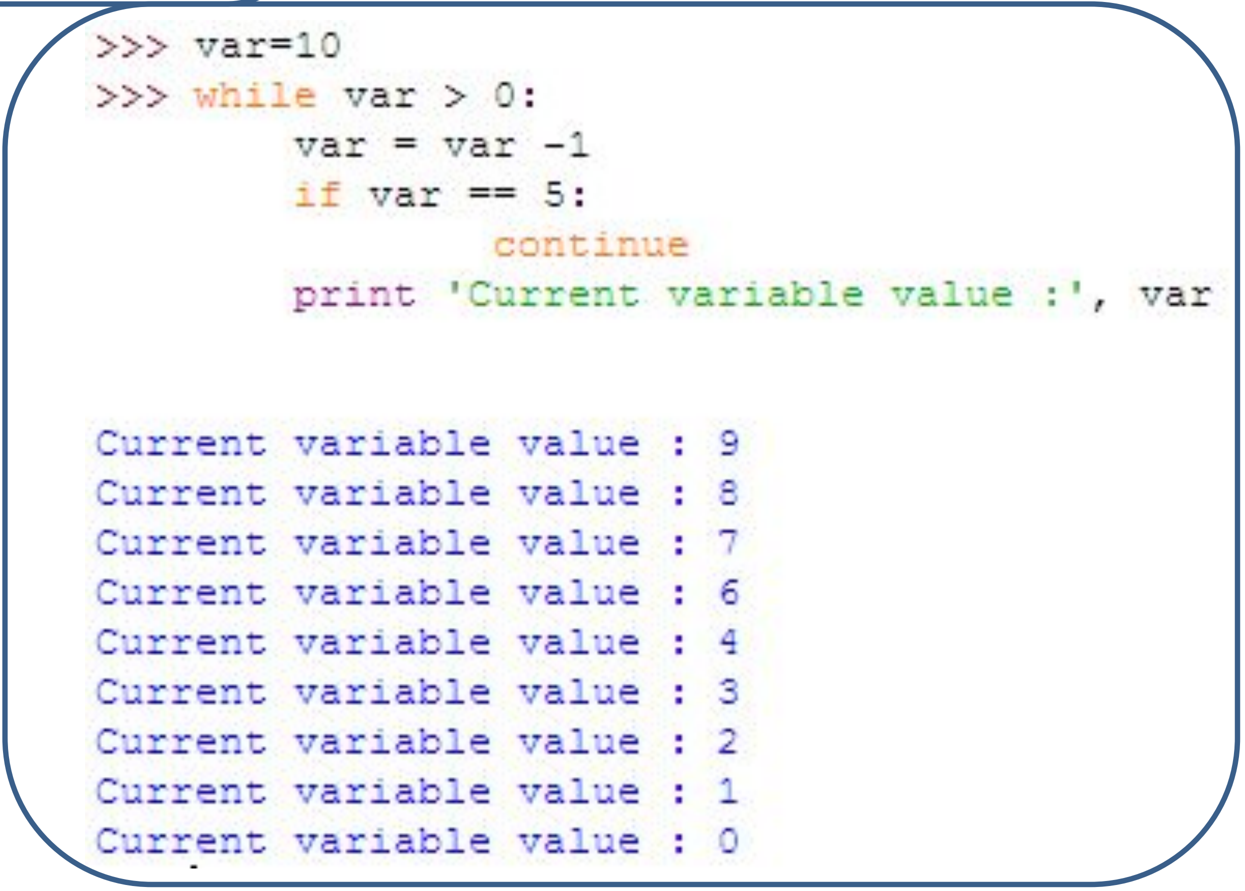
```
continue
```


conditional code
If condition is true
continue
condition
If condition is false

Example(s)



```
>>> for letter in 'Python':
        if letter == 'h':
                continue
        print 'Current Letter :', letter


Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
```



```
>>> var=10
>>> while var > 0:
        var = var -1
        if var == 5:
                continue
        print 'Current variable value :', var


Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
```

nested loops

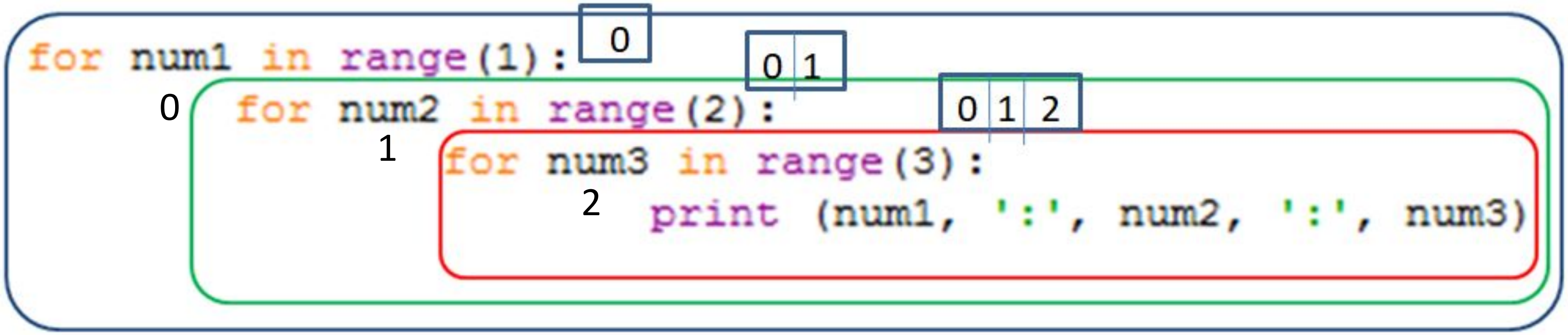
- Python programming language allows to use one loop inside another loop.

Syntax:

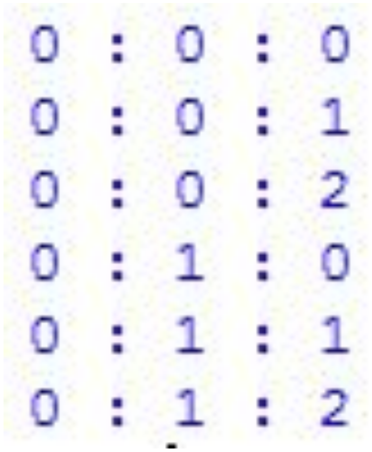
```
for iterating var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

Nested loop

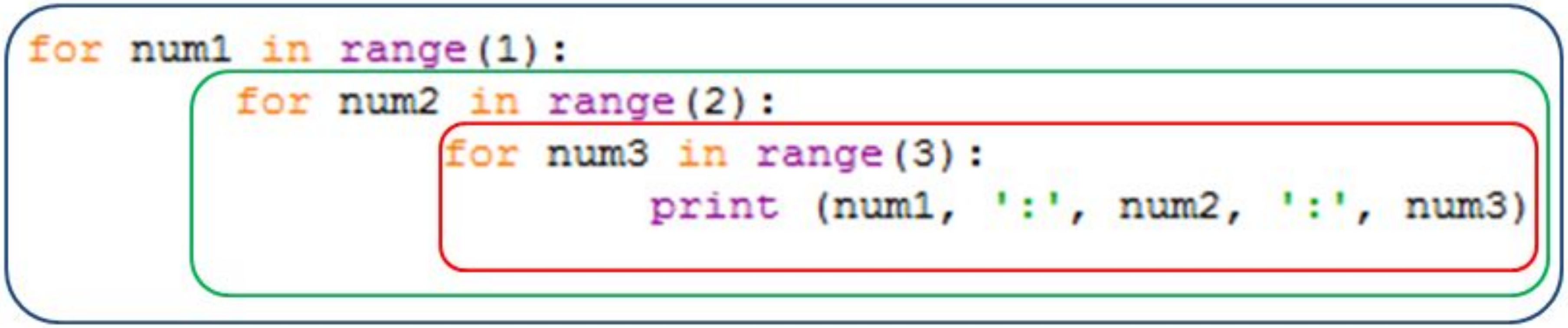


```
for num1 in range(1):
    for num2 in range(2):
        for num3 in range(3):
            print (num1, ':', num2, ':', num3)
```

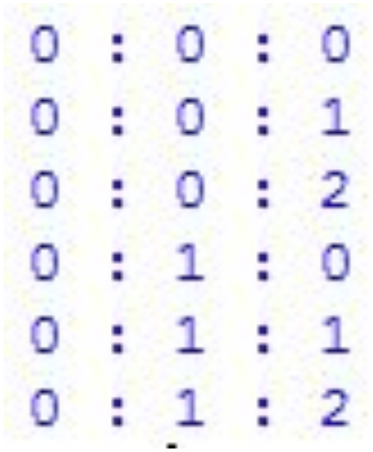


```
0 : 0 : 0
0 : 0 : 1
0 : 0 : 2
0 : 1 : 0
0 : 1 : 1
0 : 1 : 2
```

Nested loop



```
for num1 in range(1):
        for num2 in range(2):
                for num3 in range(3):
                        print (num1, ':', num2, ':', num3)
```



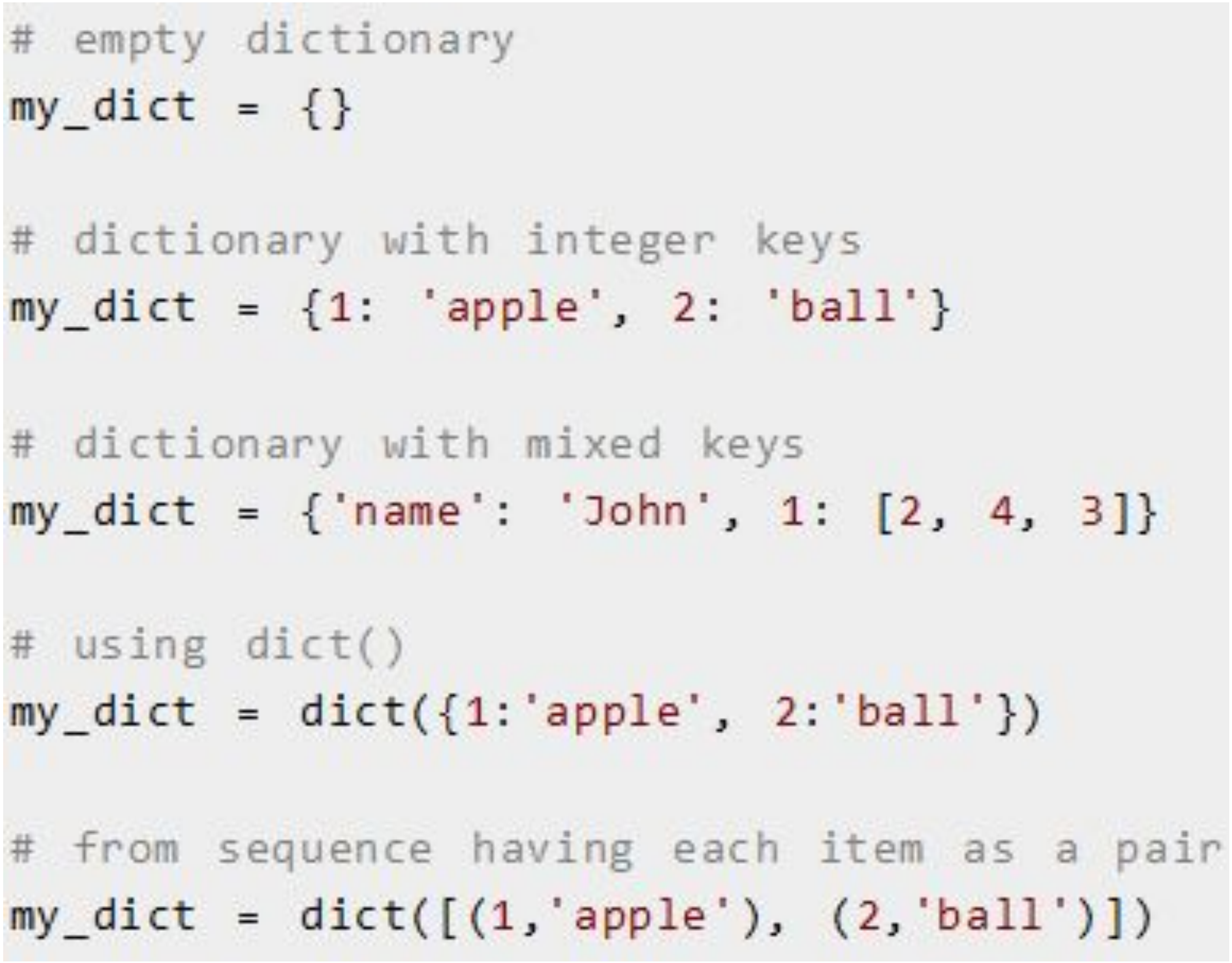
```
0 : 0 : 0
0 : 0 : 1
0 : 0 : 2
0 : 1 : 0
0 : 1 : 1
0 : 1 : 2
```

```
1
1  2
1  2  3
```

Python Dictionary

- Python dictionary is an unordered(upto 3.7) collection of items.
- While other compound datatypes have only value as an element, a dictionary has a ***key: value*** pair.
- Dictionaries are optimized to retrieve values when the key is known.
- Creating a dictionary is as simple as placing items inside curly braces {} separated by comma.
- An item has a key and the corresponding value expressed as a pair, key: value.
- While values can be of any datatype and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.
- We can also create a dictionary using the built-in function dict().

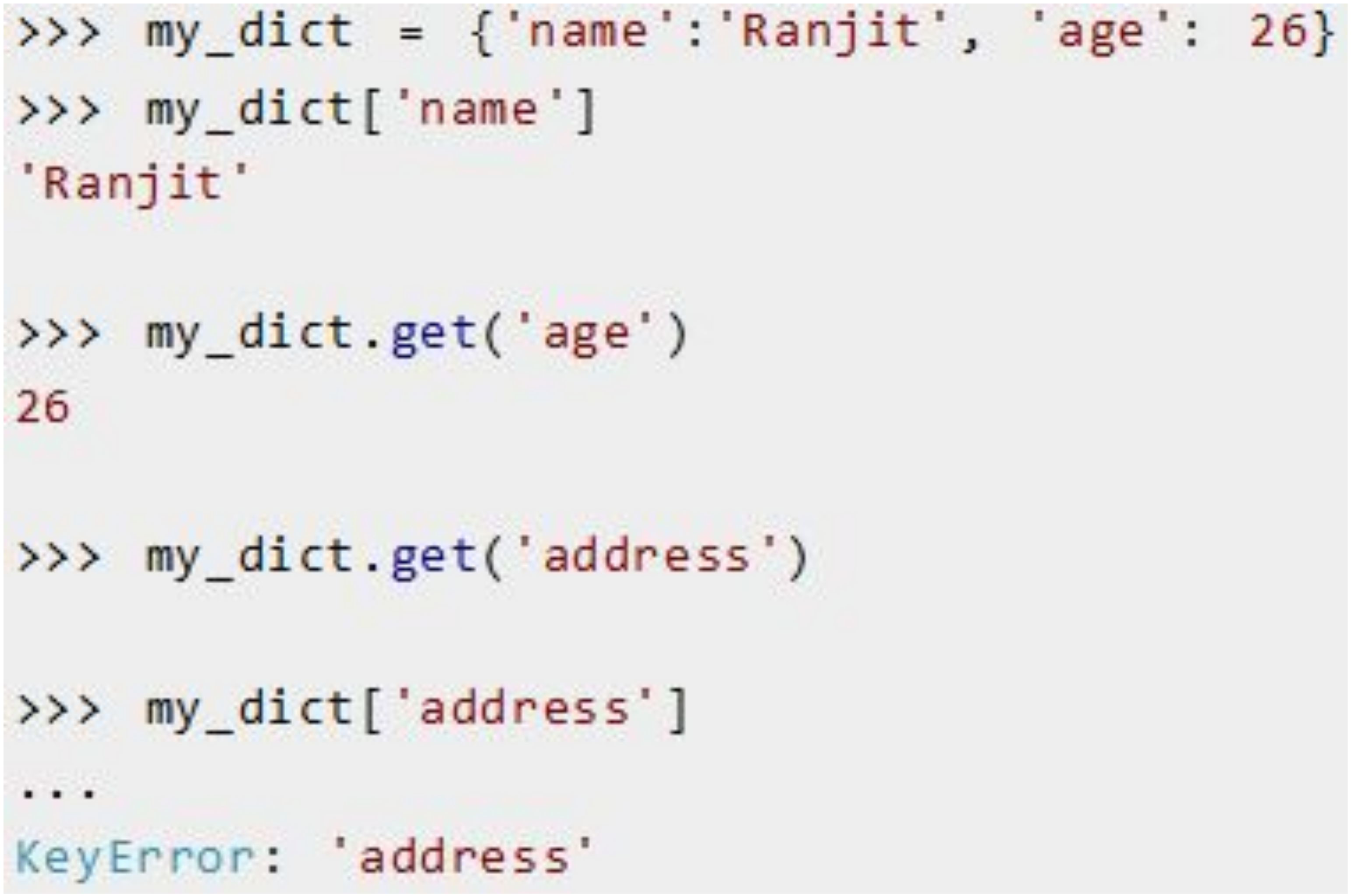
Python Dictionary



```
# 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')])
```

Accessing Elements in a Dictionary

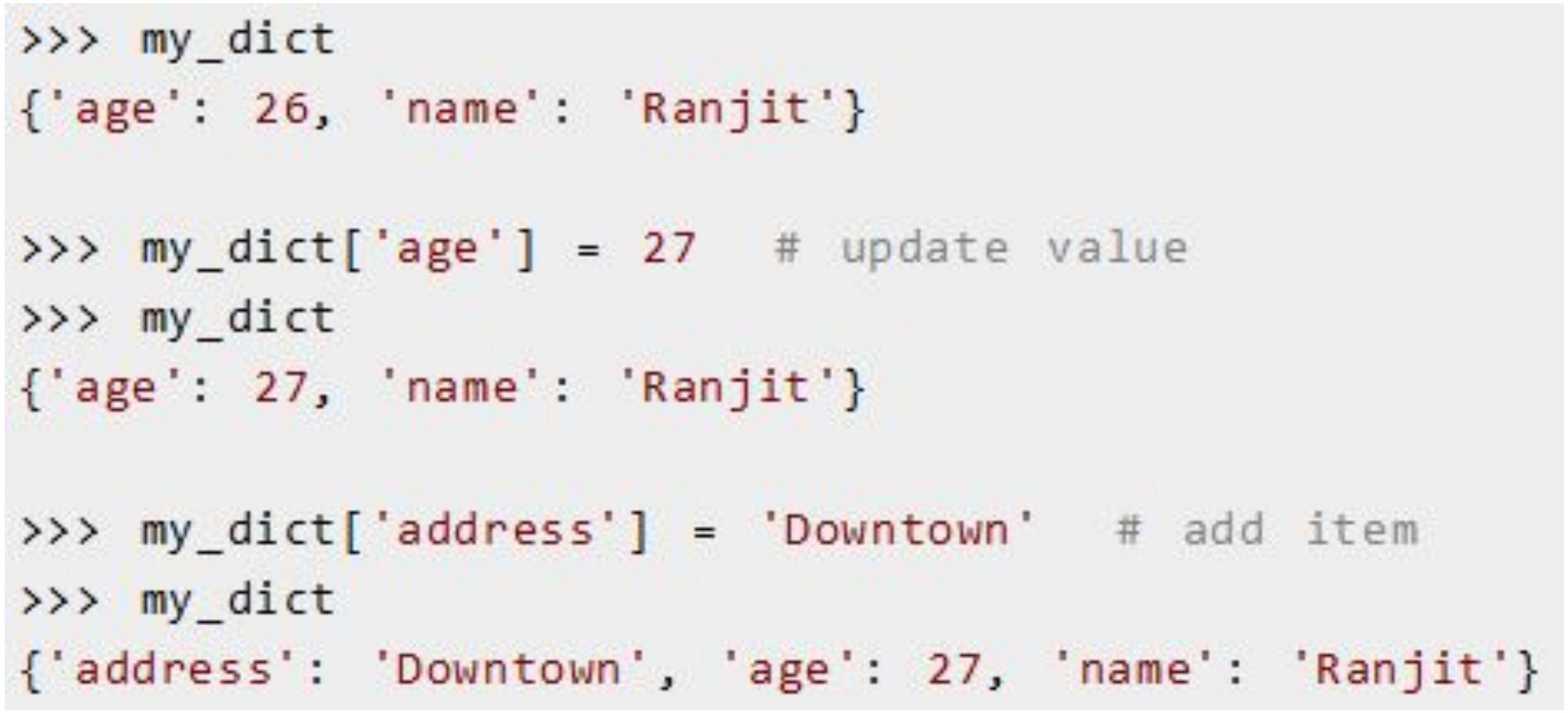
- While indexing is used with other container types to access values, dictionary uses keys.
- Key can be used either inside square brackets or with the get() method.
- The difference while using get() is that it returns None instead of KeyError, if the key is not found.



```
>>> my_dict = {'name':'Ranjit', 'age': 26}
>>> my_dict['name']
'Ranjit'

>>> my_dict.get('age')
26


>>> my_dict.get('address')


>>> my_dict['address']
...
KeyError: 'address'
```

Changing or Adding Elements in a Dictionary

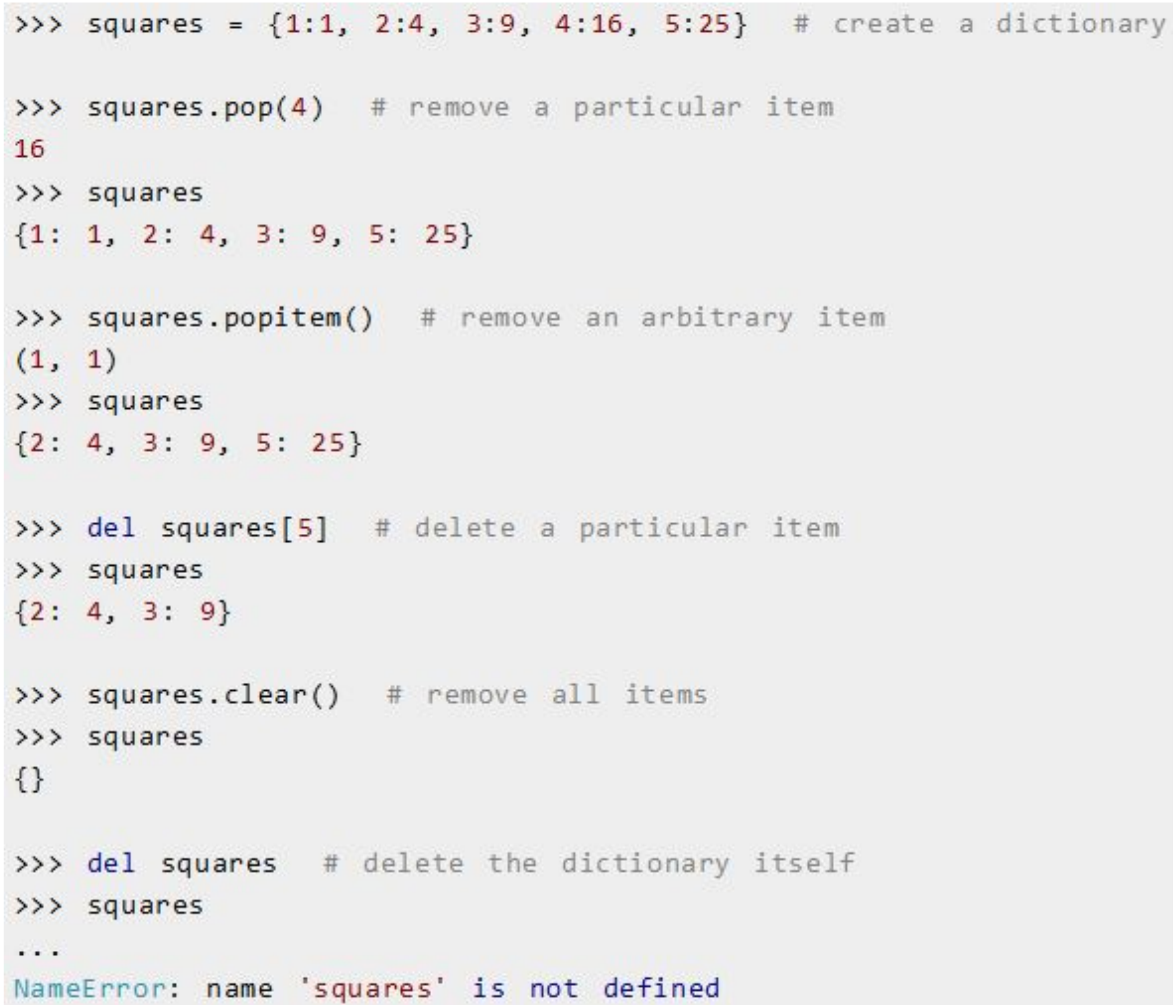
- Dictionary are mutable.
- We can add new items or change the value of existing items using assignment operator.
- If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.



```
>>> my_dict
{'age': 26, 'name': 'Ranjit'}

>>> my_dict['age'] = 27   # update value
>>> my_dict
{'age': 27, 'name': 'Ranjit'}

>>> my_dict['address'] = 'Downtown'   # add item
>>> my_dict
{'address': 'Downtown', 'age': 27, 'name': 'Ranjit'}
```

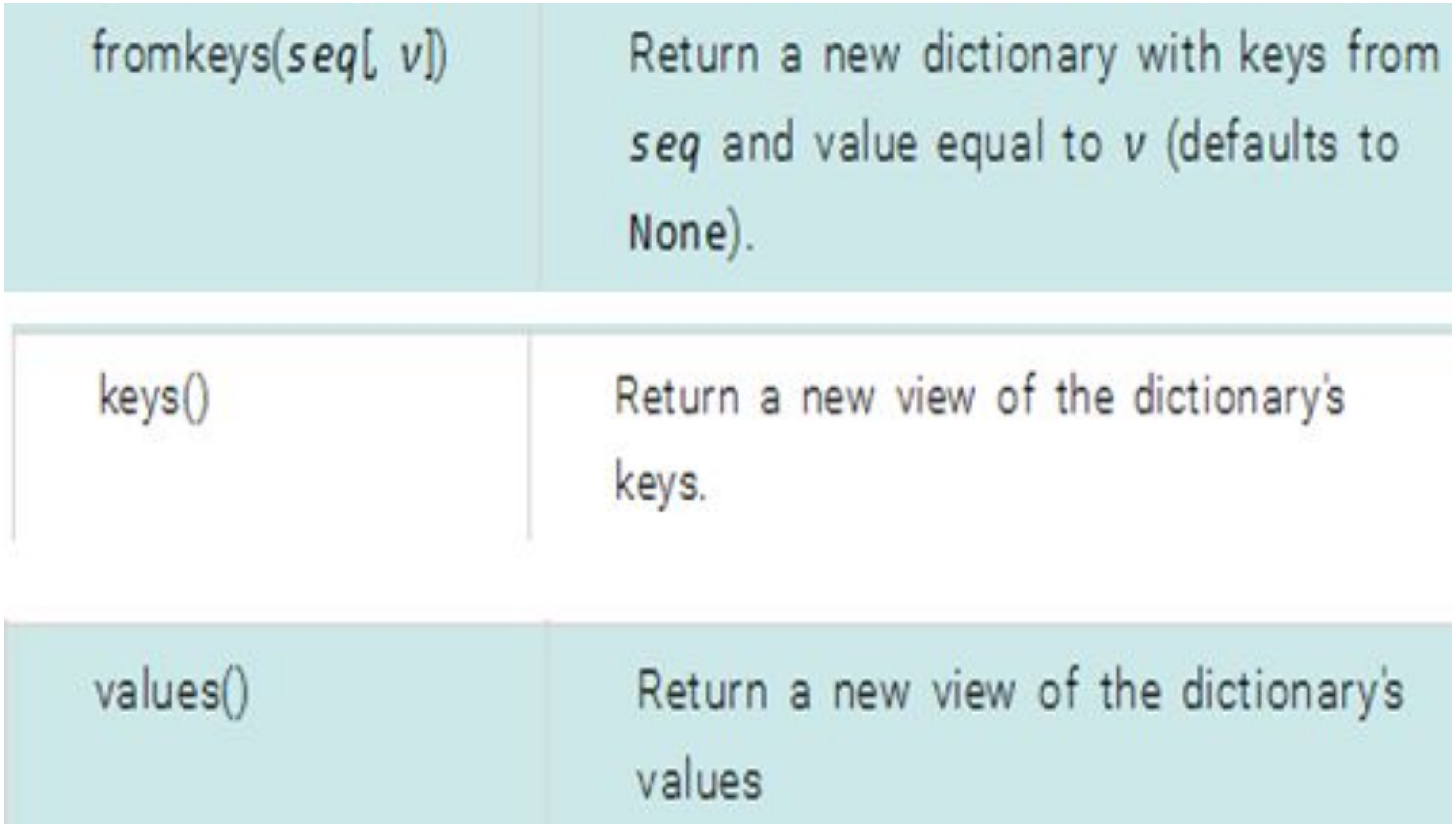
Deleting or Removing Elements from a Dictionary

- We can remove a particular item in a dictionary by using the method **pop().**
- This method removes as item with the provided key and returns the value.
- The method, **popitem()** can be used to remove and **return an arbitrary item** (key, value) form the dictionary.
- In python 3.0, **popitem() will be used to remove last key:value pair.**
- All the items can be removed at once using the **clear()** method.
- We can also use the **del** keyword to remove individual items or the entire dictionary itself.



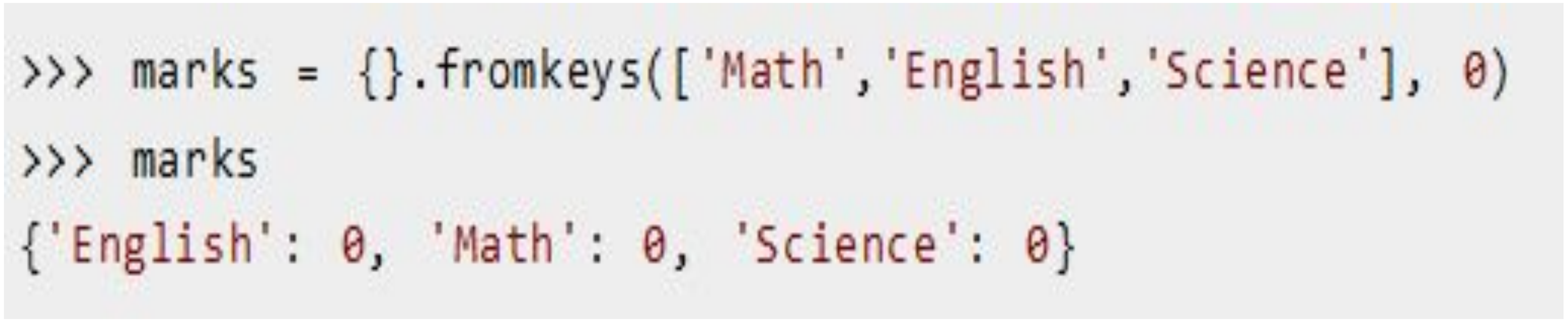
```
>>> squares = {1:1, 2:4, 3:9, 4:16, 5:25}  # create a dictionary

>>> squares.pop(4)  # remove a particular item
16
>>> squares
{1: 1, 2: 4, 3: 9, 5: 25}

>>> squares.popitem()  # remove an arbitrary item
(1, 1)
>>> squares
{2: 4, 3: 9, 5: 25}

>>> del squares[5]  # delete a particular item
>>> squares
{2: 4, 3: 9}

>>> squares.clear()  # remove all items
>>> squares
{}

>>> del squares  # delete the dictionary itself
>>> squares
...
NameError: name 'squares' is not defined
```

Python Dictionary Methods

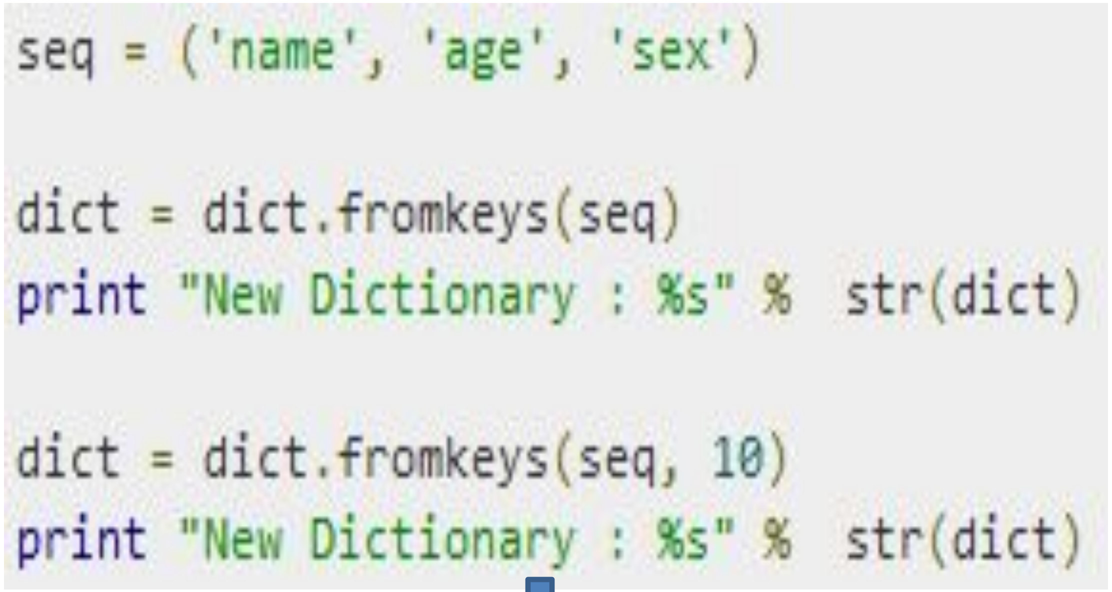


fromkeys(*seq*[, *v*])	Return a new dictionary with keys from *seq* and value equal to *v* (defaults to None).
keys()	Return a new view of the dictionary's keys.
values()	Return a new view of the dictionary's values

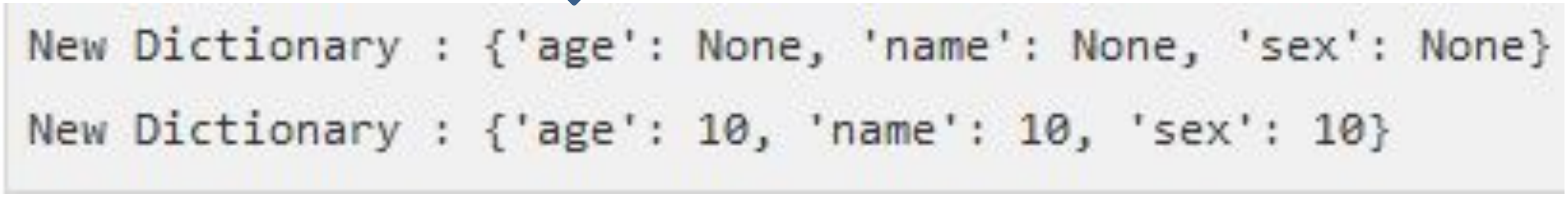
Python Dictionary Methods



```
>>> marks = {}.fromkeys(['Math','English','Science'], 0)
>>> marks
{'English': 0, 'Math': 0, 'Science': 0}
```

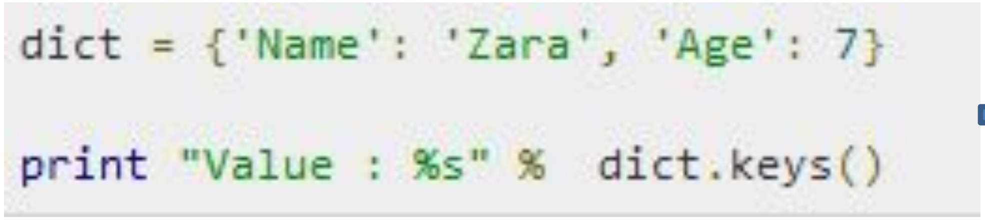


```
seq = ('name', 'age', 'sex')

dict = dict.fromkeys(seq)
print "New Dictionary : %s" %  str(dict)

dict = dict.fromkeys(seq, 10)
print "New Dictionary : %s" %  str(dict)
```

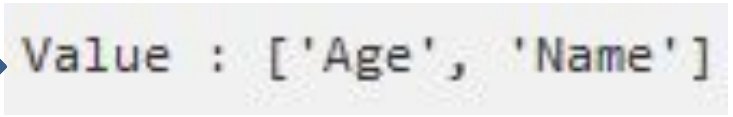


```
New Dictionary : {'age': None, 'name': None, 'sex': None}
New Dictionary : {'age': 10, 'name': 10, 'sex': 10}
```

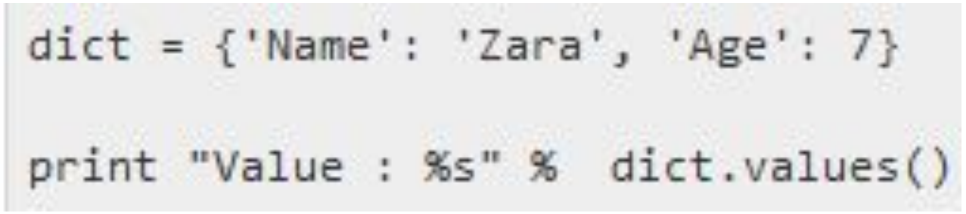
Python Dictionary Methods



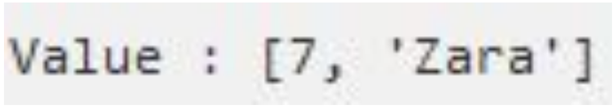
```
dict = {'Name': 'Zara', 'Age': 7}

print "Value : %s" %  dict.keys()
```



```
Value : ['Age', 'Name']
```



```
dict = {'Name': 'Zara', 'Age': 7}

print "Value : %s" %  dict.values()
```



```
Value : [7, 'Zara']
```

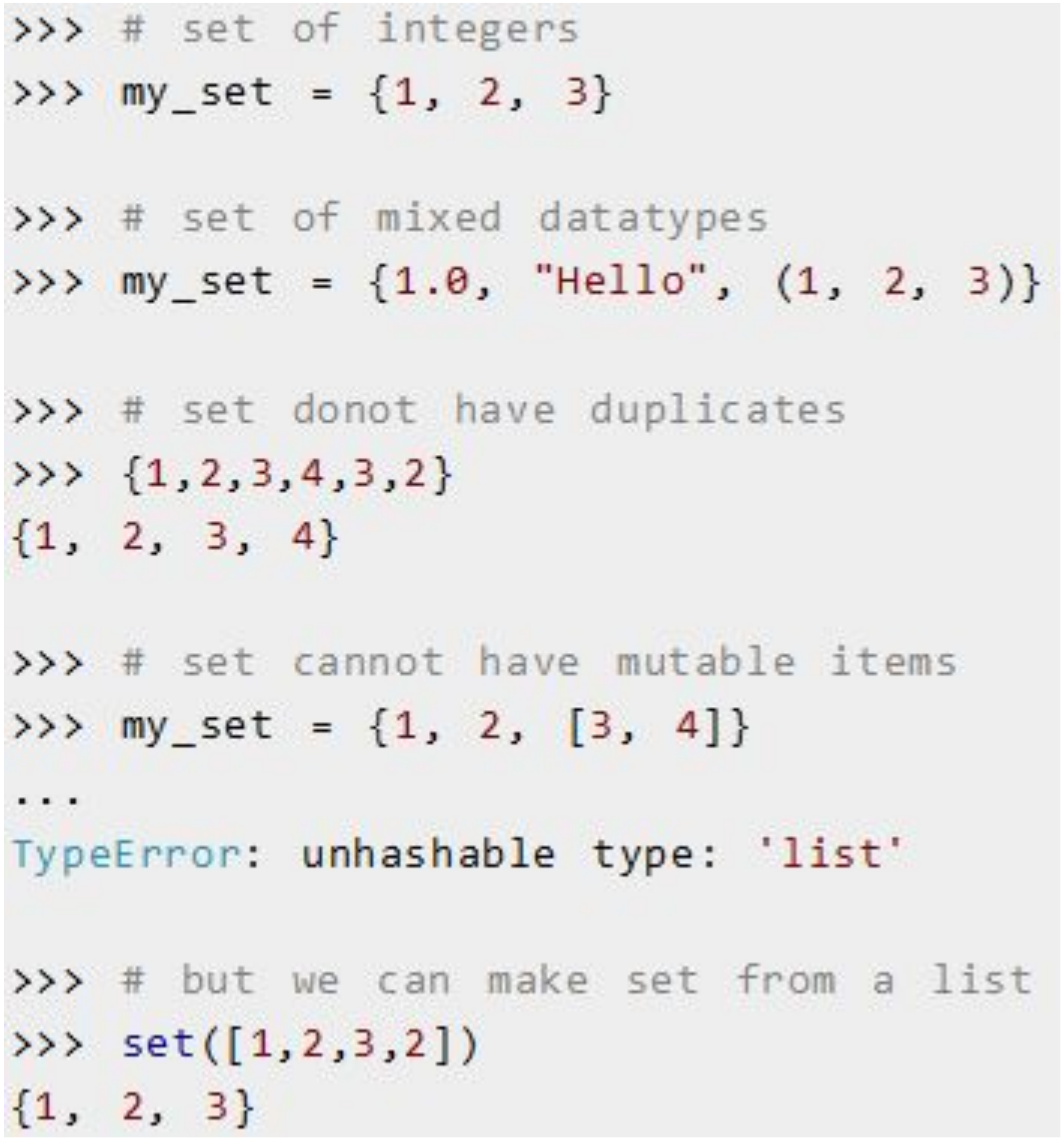
Python Sets

- Set is an unordered collection of items.
- Every element is unique (no duplicates) and must be immutable.
- However, the set itself is mutable (we can add or remove items).
- Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

Creating a Set in Python

- A set is created by placing all the items (elements) inside curly braces {}, separated by comma or by using the built-in function set().
- It can have any number of items and they may be of different types (integer, float, tuple, string etc.).
- But a set cannot have a mutable element, like list, set or dictionary, as its element.

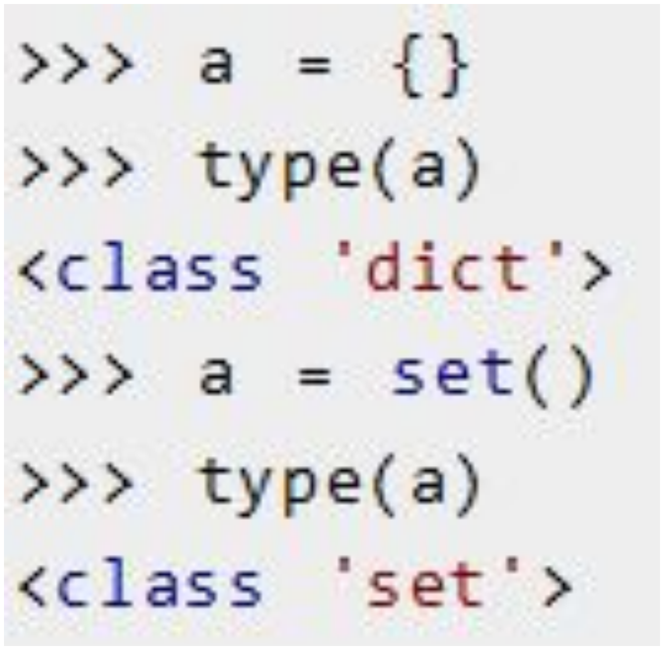
Creating a Set in Python



```
>>> # set of integers
>>> my_set = {1, 2, 3}

>>> # set of mixed datatypes
>>> my_set = {1.0, "Hello", (1, 2, 3)}

>>> # set donot have duplicates
>>> {1,2,3,4,3,2}
{1, 2, 3, 4}

>>> # set cannot have mutable items
>>> my_set = {1, 2, [3, 4]}
...
TypeError: unhashable type: 'list'

>>> # but we can make set from a list
>>> set([1,2,3,2])
{1, 2, 3}
```

Creating a Set in Python

- 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.

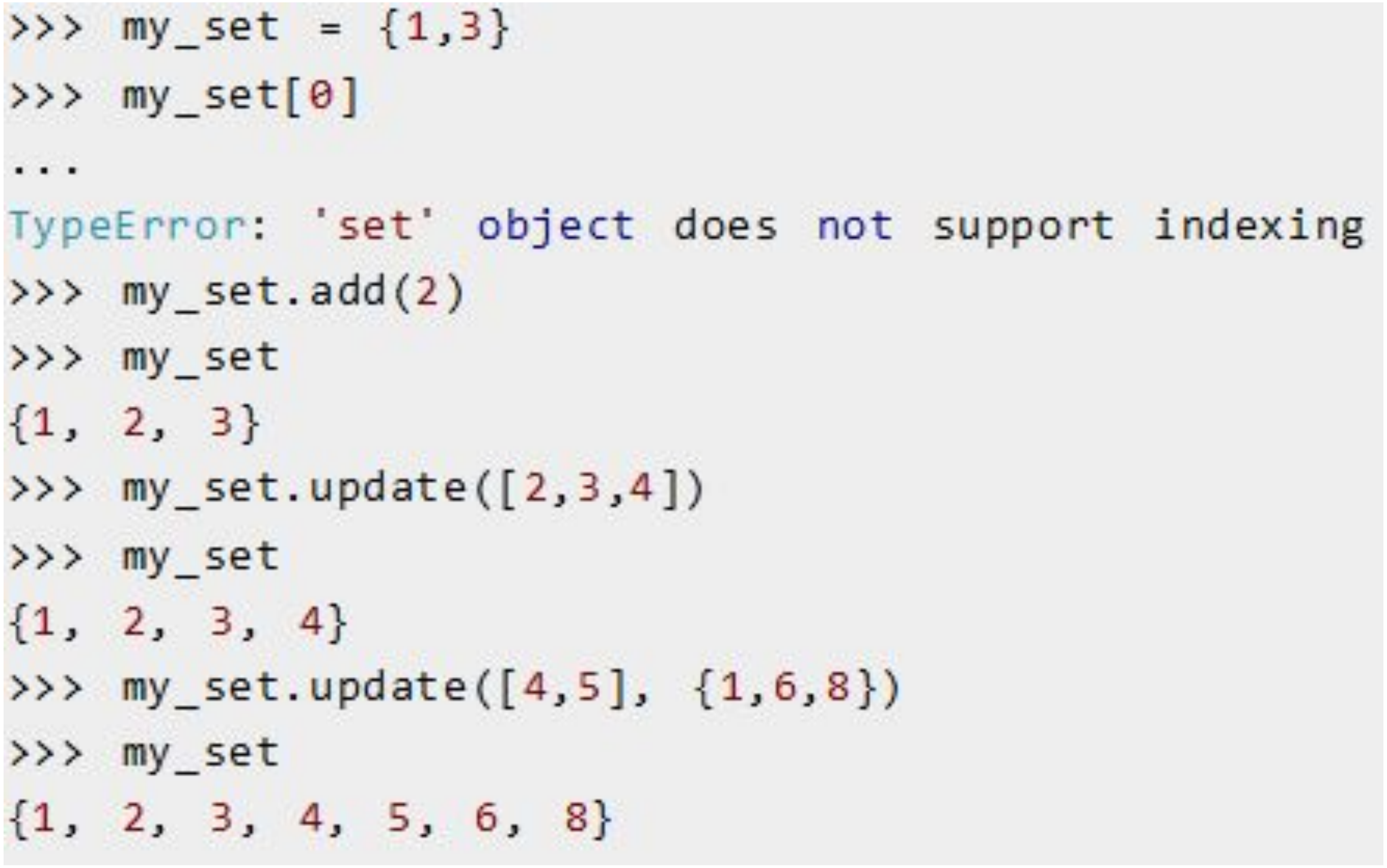


```
>>> a = {}
>>> type(a)
<class 'dict'>
>>> a = set()
>>> type(a)
<class 'set'>
```

Changing a Set in Python

- Sets are mutable.
- But since they are unordered, indexing have no meaning.
- We cannot access or change an element of set using indexing or slicing. Set does not support it.
- We can add single elements using the method add().
- Multiple elements can be added using update() method.
- The update() method can take tuples, lists, strings or other sets as its argument.
- In all cases, duplicates are avoided.

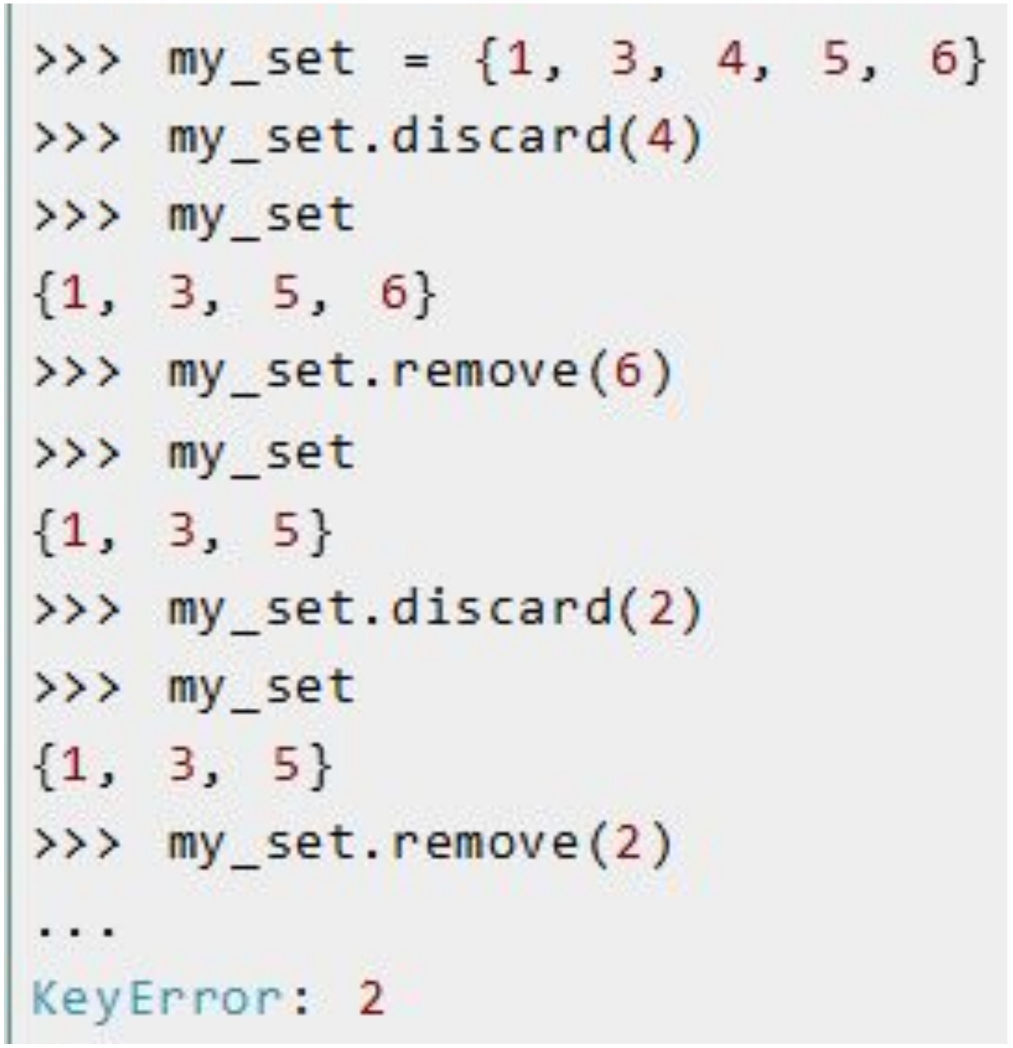
Changing a Set in Python



```
>>> my_set = {1,3}
>>> my_set[0]
...
TypeError: 'set' object does not support indexing
>>> my_set.add(2)
>>> my_set
{1, 2, 3}
>>> my_set.update([2,3,4])
>>> my_set
{1, 2, 3, 4}
>>> my_set.update([4,5], {1,6,8})
>>> my_set
{1, 2, 3, 4, 5, 6, 8}
```

Removing Elements from a Set

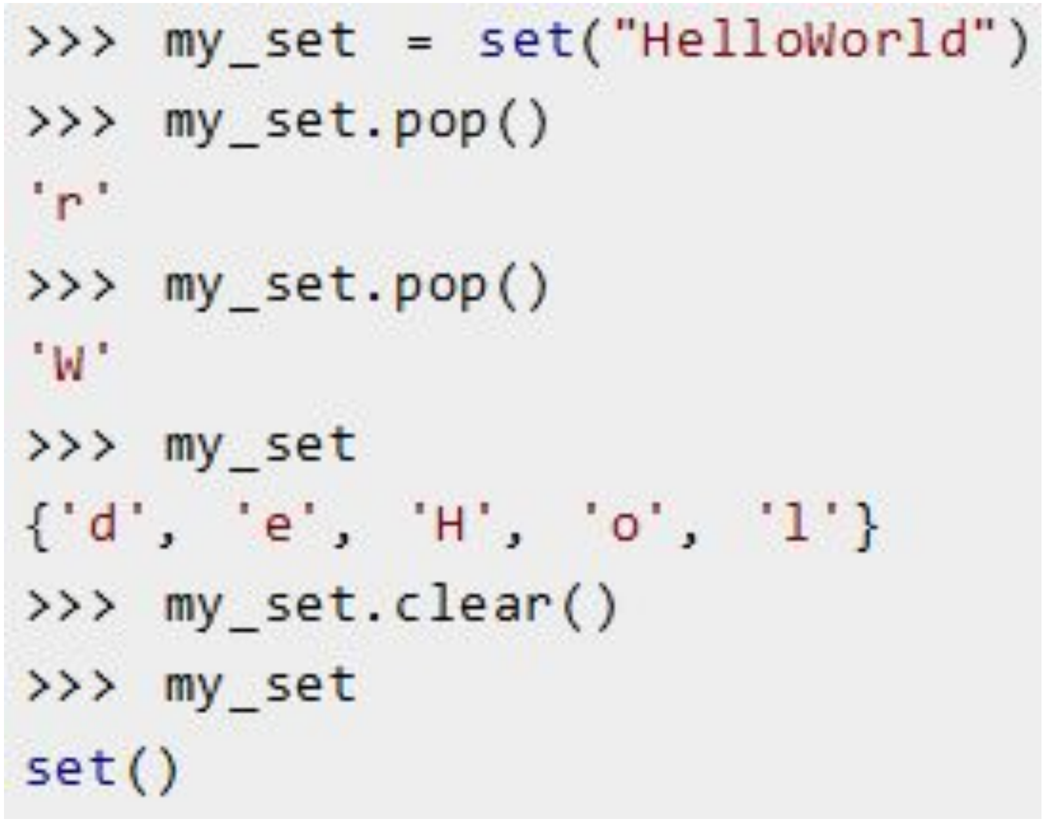
- A particular item can be removed from set using methods like **discard()** and **remove().**
- The only difference between the two is that, while using discard() if the item does not exist in the set, it remains unchanged.
- But remove() will raise an error in such condition.



```
>>> my_set = {1, 3, 4, 5, 6}
>>> my_set.discard(4)
>>> my_set
{1, 3, 5, 6}
>>> my_set.remove(6)
>>> my_set
{1, 3, 5}
>>> my_set.discard(2)
>>> my_set
{1, 3, 5}
>>> my_set.remove(2)
...
KeyError: 2
```

Removing Elements from a Set

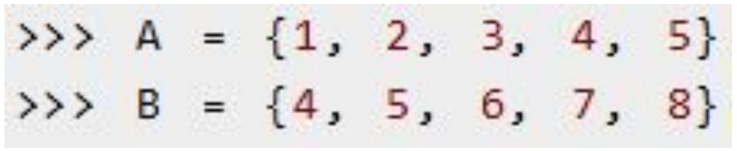
- Similarly, we can remove and return an item using the pop() method.
- Set being unordered, there is no way of determining which item will be popped.
- It is completely arbitrary. We can also remove all items from a set using clear().



```
>>> my_set = set("HelloWorld")
>>> my_set.pop()
'r'
>>> my_set.pop()
'W'
>>> my_set
{'d', 'e', 'H', 'o', 'l'}
>>> my_set.clear()
>>> my_set
set()
```

Python Set Operation

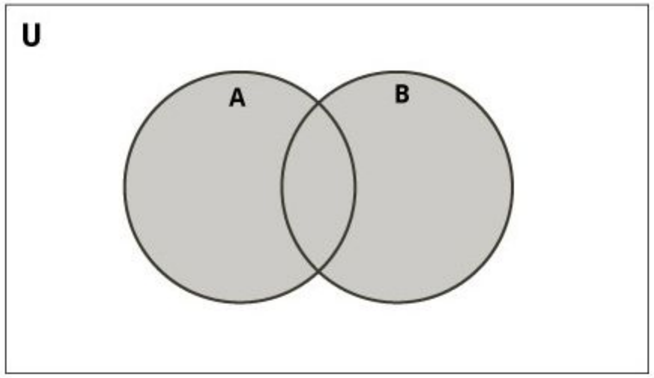
- Sets can be used to carry out following mathematical set operations
- **union,**
- **intersection,**
- **difference and**
- **symmetric difference**.
- We can do this with operators or methods.

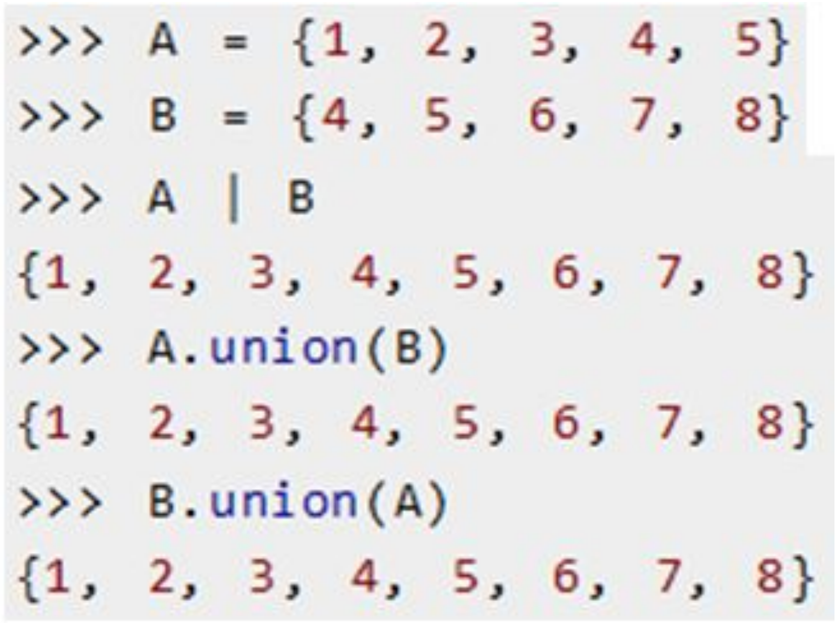


```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
```

Set Union

- Union of *A* and *B* is a set of all elements from both sets.
- Union is performed using **|** operator.
- Same can be accomplished using the method **union().**
- **Example:**

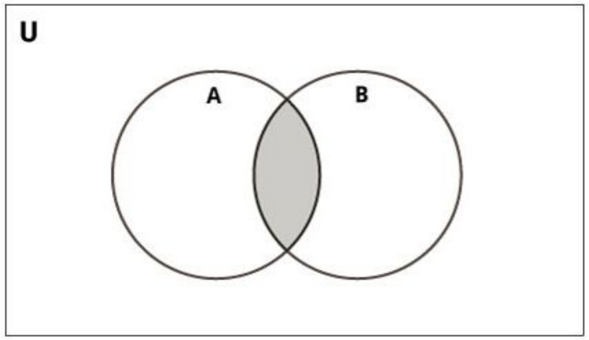


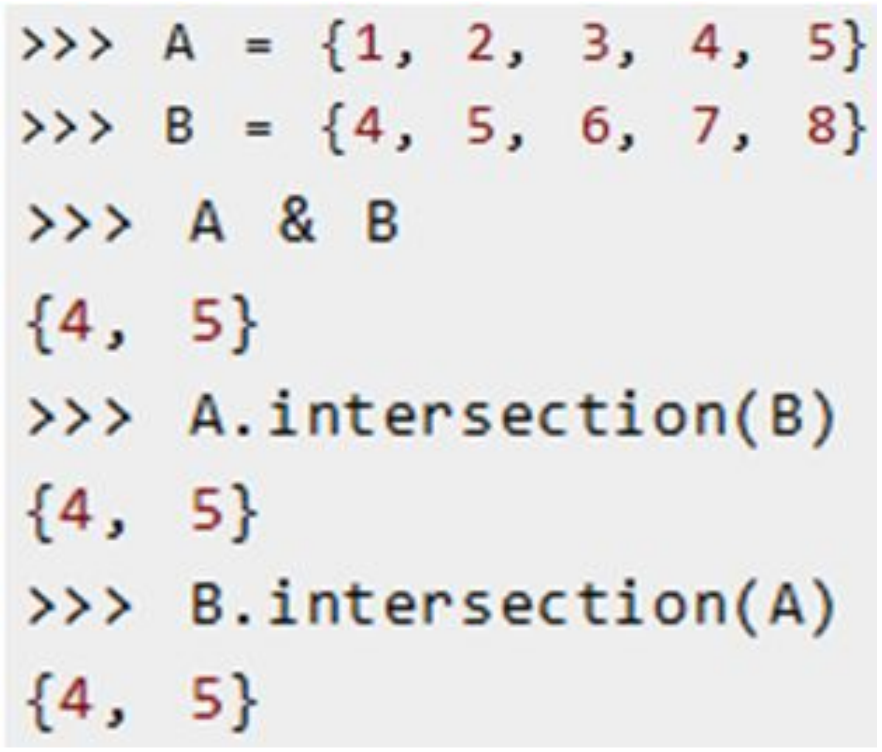


```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> A | B
{1, 2, 3, 4, 5, 6, 7, 8}
>>> A.union(B)
{1, 2, 3, 4, 5, 6, 7, 8}
>>> B.union(A)
{1, 2, 3, 4, 5, 6, 7, 8}
```

Set Intersection

- Intersection of *A* and *B* is a set of elements that are common in both sets.
- Intersection is performed using **&** operator.
- Same can be accomplished using the method **intersection().**
- **Example:**

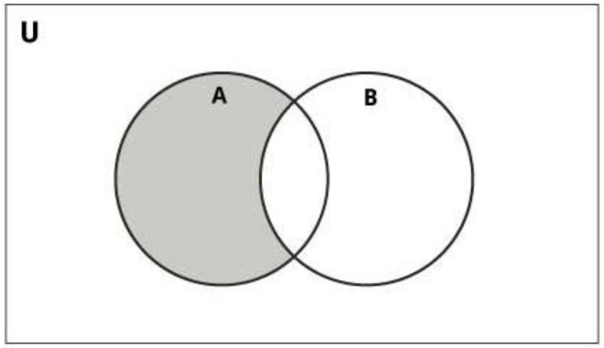


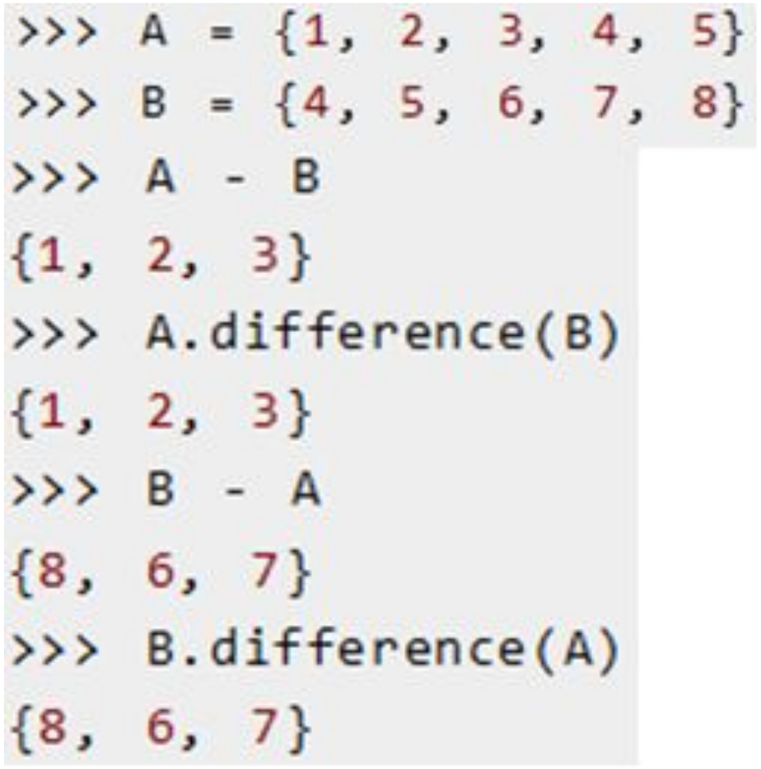


```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> A & B
{4, 5}
>>> A.intersection(B)
{4, 5}
>>> B.intersection(A)
{4, 5}
```

Set Difference

- Difference of *A* and *B* **(*A* - *B*)** is a set of elements that are only in *A* but not in *B*.
- Similarly, ***B* - *A*** is a set of element in *B* but not in *A*.
- Difference is performed using **-** operator.
- Same can be accomplished using the method **difference() .**
- **Example:**

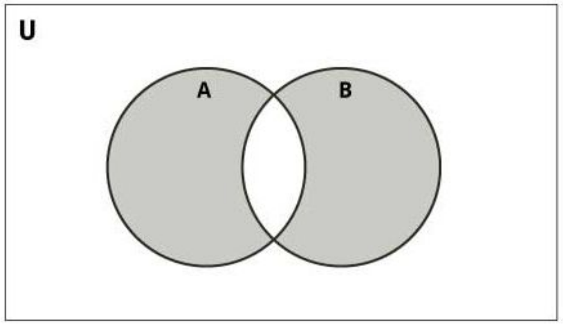


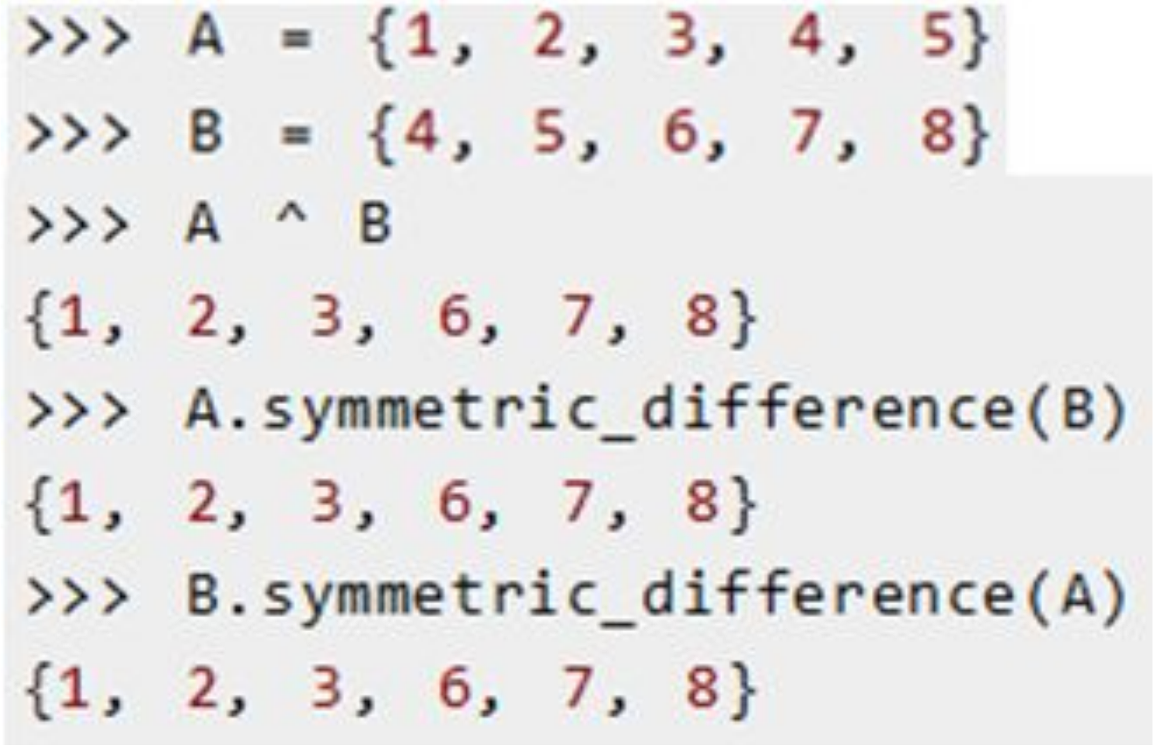


```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> A - B
{1, 2, 3}
>>> A.difference(B)
{1, 2, 3}
>>> B - A
{8, 6, 7}
>>> B.difference(A)
{8, 6, 7}
```

Set Symmetric Difference

- Symmetric Difference of *A* and *B* is a set of element in both *A* and *B* except those common in both.
- Symmetric difference is performed using **^** operator.
- Same can be accomplished using the method **symmetric_difference().**
- **Example:**



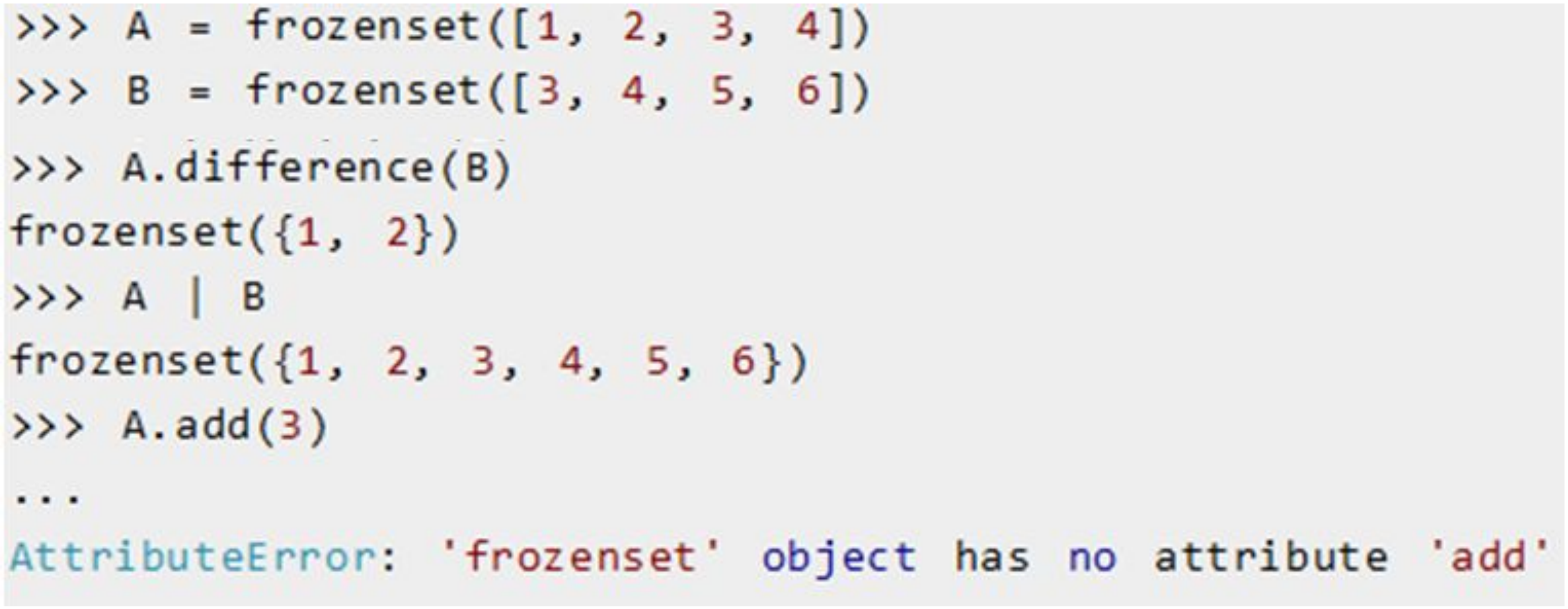


```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> A ^ B
{1, 2, 3, 6, 7, 8}
>>> A.symmetric_difference(B)
{1, 2, 3, 6, 7, 8}
>>> B.symmetric_difference(A)
{1, 2, 3, 6, 7, 8}
```

Python Frozenset

- Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned.
- While tuples are immutable lists, frozensets are immutable sets.
- Sets being mutable are unhashable, so they can't be used as dictionary keys.
- On the other hand, frozensets are hashable and can be used as keys to a dictionary.
- Being immutable it does not have method that add or remove elements.

Python Frozenset



```
>>> A = frozenset([1, 2, 3, 4])
>>> B = frozenset([3, 4, 5, 6])
>>> A.difference(B)
frozenset({1, 2})
>>> A | B
frozenset({1, 2, 3, 4, 5, 6})
>>> A.add(3)
...
AttributeError: 'frozenset' object has no attribute 'add'
```