TYPES

Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example –

```
var1 =1
var2 =10
```

Python supports four different numerical types –

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

STRING OPERATION:

The+ operator works with strings, but it is not addition in the mathematical sense.Instead it performs concatenation , which means joining the strings by linking
them end-to-end. For example:

```
>>>first = 10
>>>second = 15
>>>printfirst+second
25
>>>first ='100'
>>>second ='150'
>>>print first + second
100150
```

BOOLEAN EXPRESSIONS:

Aboolean expression is an expression that is either true or false. The following examples use the operator

`==` , which compares two operands and produces `True` if they are equal and `False`otherwise:
`True` and `False`are special values that belong to the type `bool` ; they are not strings:

```
>>>type(True)
<type 'bool'>
>>>type(False)
<type 'bool'>

x != y # x is not equal to y
x > y # x is greater than y
x < y # x is less than y
x >= y # x is greater than or equal to y
x <= y # x is less than or equal to y
x is y # x is the same as y
x is not y # x is not the same as Y.
```

OPERATORS

Operators are the constructs which can manipulate the value of operands.

Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

TYPES OF OPERATORS

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators

Membership Operators

- Identity Operators

Let us have a look on all operators one by one.

Python Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then −

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity):	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Python Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds 10 and variable b holds 20, then −

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Python Assignment Operators

Assume variable a holds 10 and variable b holds 20, then −

Operator	Description	Example
=	Assigns values from right side operands to left side	c = a + b assigns

	operand	value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

----

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

There are following Bitwise operators supported by Python language

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to

		a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> = 15 (means 0000 1111)

Python Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

Python Membership Operators

Python’s membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below.

Operator	Description	Example
In	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below:

Operator	Description	Example
Is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

EXPRESSIONS:

Anexpression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable x has been assigned a value):

17

x

x + 17

If you type an expression in interactive mode, the interpreter evaluates it and displays the result:

>>> 1 + 1

2

But in a script, an expression all by itself doesn't do anything! This is a common source of confusion for beginners.

Exercise 2 Type the following statements in the Python interpreter to see what they do:

5

x = 5

x + 1

Now put the same statements into a script and run it. What is the output? Modify the script by transforming each expression into a print statement and then run it again.

Order of operations:

When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. For mathematical operators, Python follows mathematical convention. The acronym PEMDAS is a useful way to remember the rules:

Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 * (3-1) is 4, and (1+1)**(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even if it doesn't change the result.

Exponentiation has the next highest precedence, so 2**1+1 is 3, not 4, and 3*1**3 is 3, not 27.

Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence. So 2*3-1 is 5, not 4, and 6+4/2 is 8, not 5.

Operators with the same precedence are evaluated from left to right (except exponentiation). So in the expression degrees / 2 * pi, the division happens first and the result is multiplied by pi. To divide by 2 π, you can use parentheses or write degrees / 2 / pi.

DECISION MAKING IN PYTHON

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

IF-STATEMENT:

The ifstatement is the simplest form of decision control statement that is frequently used in making decision .An if statement is a selection control statement based on the value of a given Boolean expression.

SYNTAX:

The syntax of if statement

If test_expression:

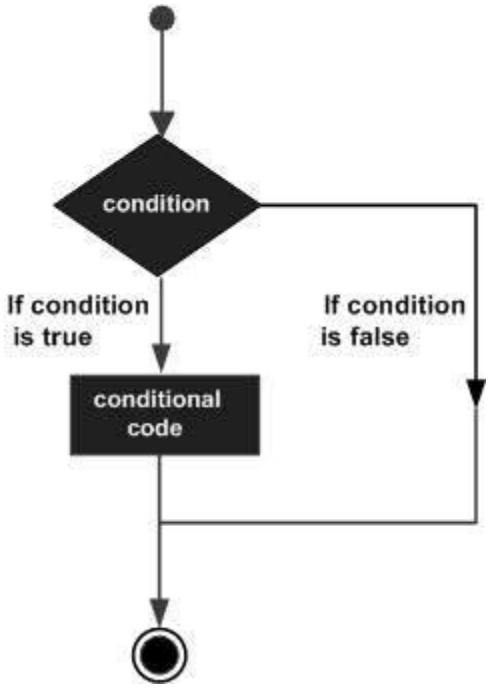
    Statement1

    ……

    Statement n

Statement x

FLOW CHART:



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.

Example:

if x > 0 :

print 'x is positive'

IF-ELSE:

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.

The *else* statement is an optional statement and there could be at most only one **else** statement following **if** .

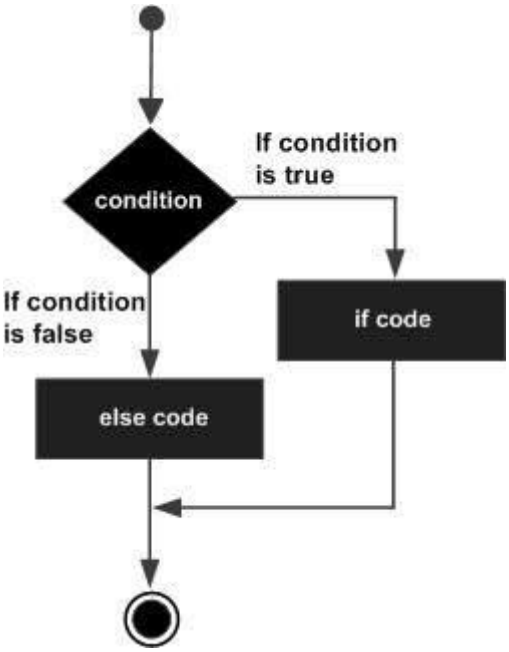
Syntax:

The syntax of the *if...else* statement is –

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

```
else:
statement(s)
```

Flow Diagram



Example:

if x%2 == 0 :

print 'x is even'

else :

print'x is odd'

The *elif* Statement:

The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at most one statement, there can be an arbitrary number of **elif** statements following an **if**.

Syntax:

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

Example:

if x < y:

print'x is less than y'

elif x > y:

print'x is greater than y'

else:

print'x and y are equal’

Nested conditionals:

One conditional can also be nested within another. We could have written the trichotomy example like this:

if x == y:

print'x and y are equal'

else:

if x < y:

print 'x is less than y'

else:

print'x is greater than y'

LOOPS IN PYTHON:

While loop:

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

We generally use this loop when we don't know beforehand, the number of times to iterate.

Syntax:

whiletest_expression:

    Body of while

In Python, the body of the while loop is determined through indentation.Body starts with indentation and the firstunindented line marks the end.Python interprets any non-zero value as True. None and 0 are interpreted as False.

Flow chart:

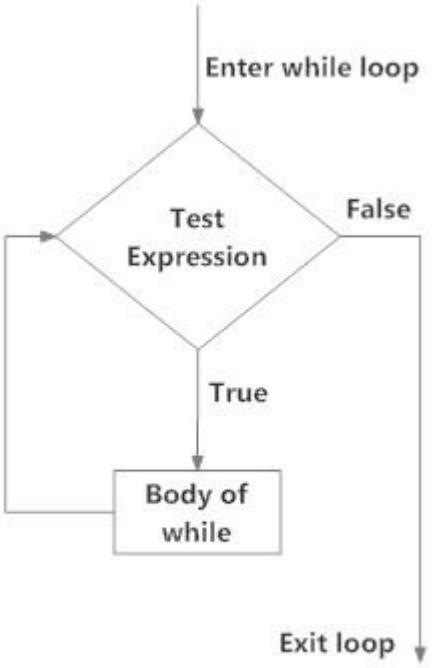


Fig: operation of while loop

Example:

```
n = 10

sum = 0

i = 1


while i <= n:

sum = sum + i

    i = i+1

print"The sum is", sum
```

Syntax of for Loop:

```
forval in sequence:

       Body of for
```

Here, `val` is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Flow chart:

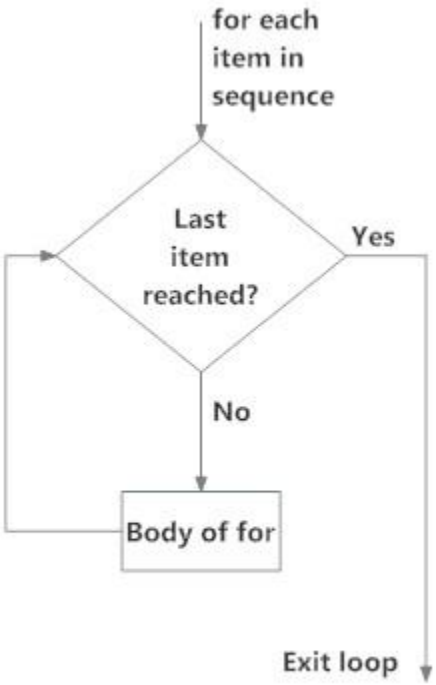


Fig: operation of for loop

Example:

numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

```
sum = 0

forval in numbers:

        sum = sum+val

print("The sum is", sum)
```

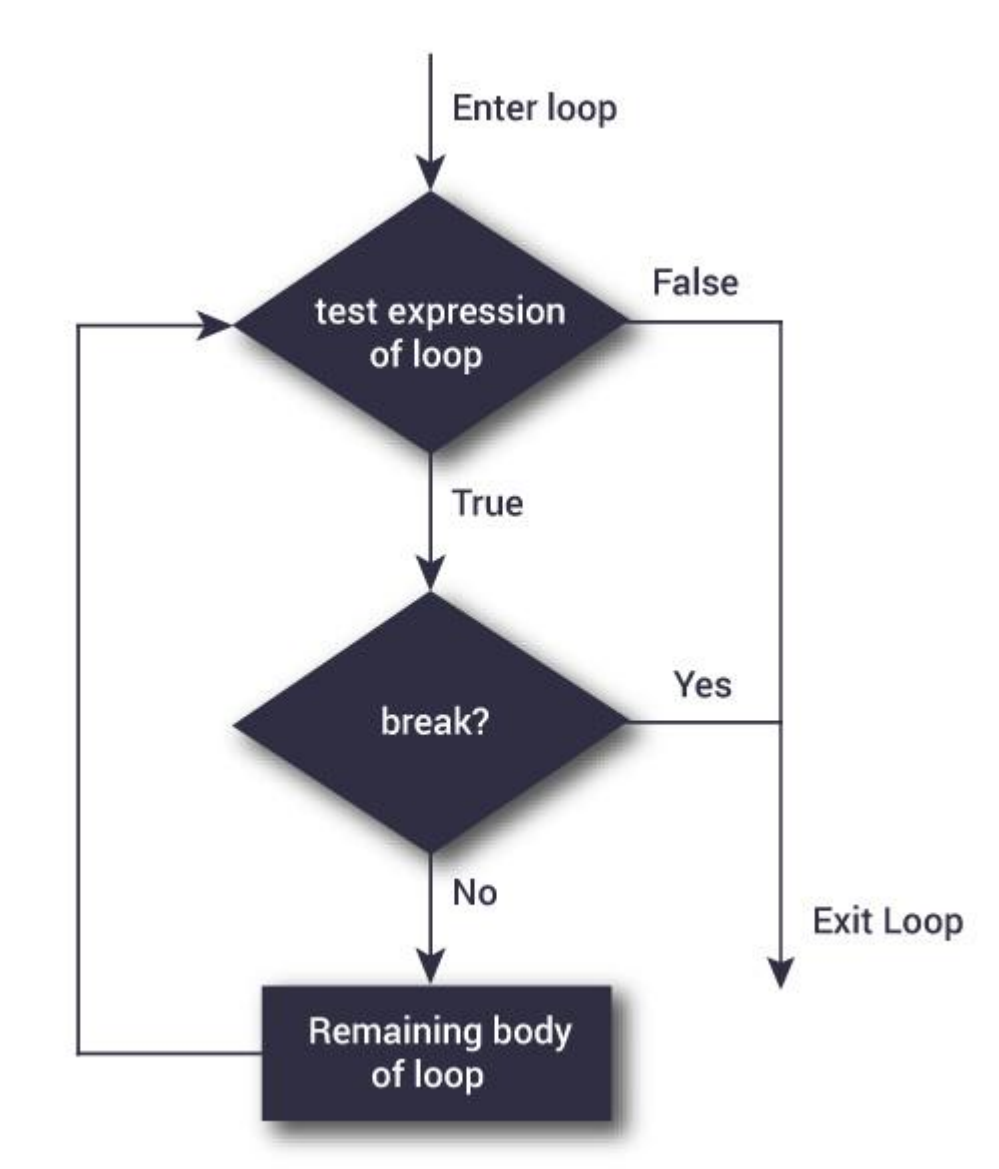
Python break statement

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

Syntax of break

```
Break
```



Example:

forval in "string":

ifval == "i":

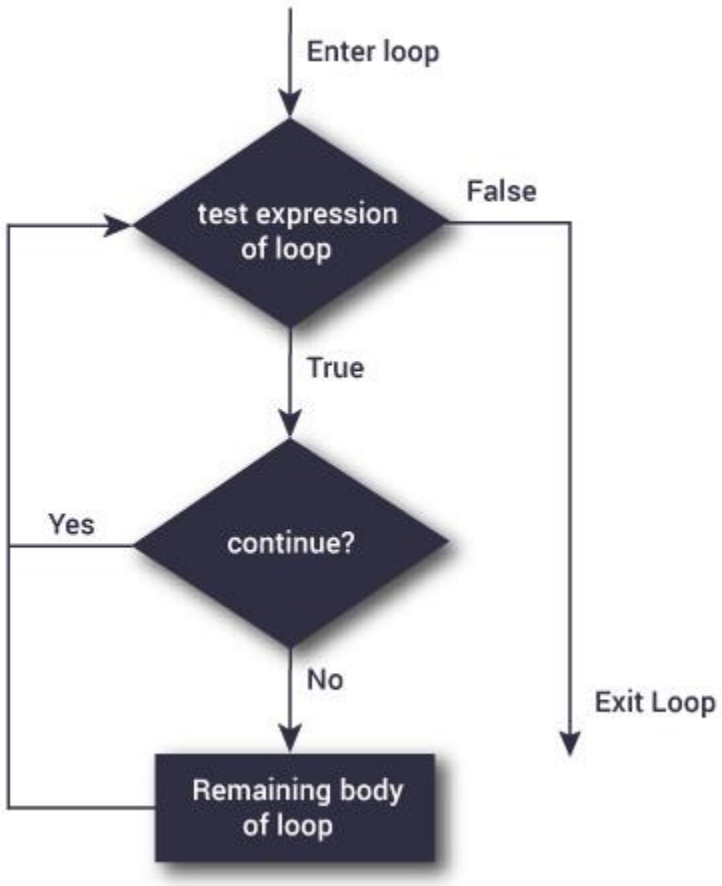
```
break

print(val)


print("The end")
```

Python continue statement:

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

Syntax of Continue

```
Continue
```



Example:

```
forval in "string":

ifval == "i":

continue

print(val)

print("The end")
```

Pass statement:

Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the `pass` statement, which does nothing.

```
if x < 0 :
pass
```

If you enter an if statement in the Python interpreter, the prompt will change from three chevrons to three dots to indicate you are in the middle of a block of statements as shown below:

```
>>> x = 3
>>>if x < 10:
... print'Small'
...
Small
>>>
```

Lists

A list is a sequence

Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called elements or sometimes items.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is nested.

A list that contains no elements is called an empty list; you can create one with empty brackets, [].

As you might expect, you can assign list values to variables:

```
>>>cheeses = ['Cheddar', 'Edam', 'Gouda']
>>>numbers = [17, 123]
>>>empty = []
>>>print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

Lists are mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>> print cheeses[0]
Cheddar
```

Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>>numbers = [17, 123]
>>>numbers[1] = 5
>>> print numbers
[17, 5]
```

list as a relationship between indices and elements. This relationship is called a mapping; each index "maps to" one of the elements.

The in operator also works on lists.

```
>>>cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

Traversing a list

The most common way to traverse the elements of a list is with a forloop. The syntax is the same as for strings:

```
for cheese in cheeses:
print cheese
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions range and len:

```
for i in range(len(numbers)):
numbers[i] = numbers[i] * 2
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

List operations

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

Similarly, the * operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

List slices

The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>>t[1:3]
['b', 'c']
>>>t[:4]
['a', 'b', 'c', 'd']
>>>t[3:]
['d', 'e', 'f']
```

```
>>>seq[:]                # [seq[0],   seq[1],          ..., seq[-1]    ]
```

```
>>>seq[low:]             # [seq[low], seq[low+1],      ..., seq[-1]    ]
>>>seq[:high]            # [seq[0],   seq[1],          ..., seq[high-1]]
>>>seq[low:high]        # [seq[low], seq[low+1],      ..., seq[high-1]]
>>>seq[::stride]        # [seq[0],   seq[stride],     ..., seq[-1]    ]
>>>seq[low::stride]     # [seq[low], seq[low+stride], ..., seq[-1]    ]
>>>seq[:high:stride]    # [seq[0],   seq[stride],     ..., seq[high-1]]
>>>seq[low:high:stride] # [seq[low], seq[low+stride], ..., seq[high-1]]
```

A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>>t[1:3] = ['x', 'y']
>>> print t
['a', 'x', 'y', 'd', 'e', 'f']
```

List methods

Python provides methods that operate on lists. For example, append adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>>t.append('d')
>>> print t
['a', 'b', 'c', 'd']
```

extendtakes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>>t1.extend(t2)
```

sortarranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>>t.sort()
>>> print t
['a', 'b', 'c', 'd', 'e']
```

Deleting elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use pop:

8.8. Lists and functions 95

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print t
['a', 'c']
>>>print x
b
```

popmodifies the list and returns the element that was removed. If you don’t provide an index, it deletes and returns the last element.

If you don’t need the removed value, you can use the deloperator:

```
>>> t = ['a', 'b', 'c']
>>>del t[1]
>>> print t
['a', 'c']
```

If you know the element you want to remove (but not the index), you can use

remove:

```
>>> t = ['a', 'b', 'c']
>>>t.remove('b')
>>> print t
['a', 'c']
```

To remove more than one element, you can use delwith a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>>del t[1:5]
>>> print t
['a', 'f']
```

```
numlist = list()
while ( True ) :
inp = raw_input('Enter a number: ')
ifinp == 'done' : break
value = float(inp)
numlist.append(value)
average = sum(numlist) / len(numlist)
print 'Average:', average
```

Lists and strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use list:

```
>>> s = 'spam'
>>> t = list(s)
>>> print t
['s', 'p', 'a', 'm']
```

The list function breaks a string into individual letters. If you want to break a string into words, you can use the split method:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
>>> print t[2]
the
```

You can call split with an optional argument called a delimiter specifies which characters to use as word boundaries.

```
>>> s = 'spam-spam-spam'
>>>delimiter = '-'
>>>s.split(delimiter)
['spam', 'spam', 'spam']
```

joinis the inverse of split. It takes a list of strings and concatenates the elements. joinis a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>>delimiter = ' '
>>>delimiter.join(t)
'pining for the fjords'
```

To concatenate strings without spaces, you can use the empty string, '', as a delimiter.

Objects and values

If we execute these assignment statements:

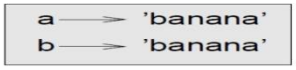
```
a = 'banana'
b = 'banana'
```

We know that a and b both refer to a string, but we don't know whether they refer to the *same* string. There are two possible states:

a='banana'

b='banana'

In one case, a andb refer to two different objects that have the same value. In the second case, they refer to the same object.



To check whether two variables refer to the same object, you can use the isoperator.

```
>>> a = 'banana'
>>> b = 'banana'
>>>a is b
True
```

In this example, Python only created one string object, and both a andb refer to it.

But when you create two lists, you get two objects:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>>a is b
False
```

In this case we would say that the two lists are equivalent, because they have the same elements, but not identical, because they are not the same object. If two

objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

Aliasing

If a refers to an object and you assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>>b is a
True
```

The association of a variable with an object is called a reference. In this example, there are two references to the same object.
An object with more than one reference has more than one name, so we say that the object is aliased.
If the aliased object is mutable, changes made with one alias affect the other

```
>>>b[0] = 17
>>> print a
[17, 2, 3]
```

Tuples

A tupleis a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are immutable.

Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'

>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include the final comma:

```
>>> t1 = ('a',)
```

Without the comma Python treats ('a') as an expression with a string in parentheses that evaluates to a string:

```
>>> t2 = ('a')
>>>type(t2)
<type 'str'>
```

Another way to construct a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

```
>>> t = tuple()
>>> print t
()
```

If the argument is a sequence (string, list or tuple), the result of the call to tuple is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's')
```

Most list operators also work on tuples. The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
'a'
```

And the slice operator selects a range of elements.

```
>>> print t[1:3]
('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

```
>>>t[0] = 'A'
```

Comparing tuples

The comparison operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ

```
>>> (0, 1, 2) < (0, 3, 4)
```

Tuple assignment

```
>>> m = [ 'have', 'fun' ]
>>>x, y = m
>>>x
'have'
>>>y
'fun'
```

Python *roughly* translates the tuple assignment syntax to be the following

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>>x
'have'
>>>y
'fun'
```

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>>x
'have'
>>>y
'fun'
```

tuple assignment allows us to swap the values of two variables in a single statement:
```
>>>a, b = b, a
```

For example, to split an email address into a user name and a domain, you could write:
```
>>>addr = 'monty@python.org'
>>>uname, domain = addr.split('@')
```

Multiple assignment with dictionaries

```
d = {'a':10, 'b':1, 'c':22}

fork,v in d.items()

printk,v
```

Using tuples as keys in dictionaries

Because tuples are hashableand lists are not, if we want to create a composite key to use in a dictionary we must use a tuple as the key. We would encounter a composite key if we wanted to create a telephone directory
that maps from last-name, first-name pairs to telephone numbers. Assuming that we have defined the variables last, first and number, we could write a dictionary assignment statement as follows:
```
directory[last,first] = number
```
The expression in brackets is a tuple. We could use tuple assignment in a for loop to traverse this dictionary.
```
for last, first in directory:
print first, last, directory[last,first]
```

+ and * on tuples

We can use sum(), max(), min(), len(),sorted() functions with tuple. Run the following and understand how these functions works with tuple.

```
X=(12,22,33,2)

print (sum(X))

print (max(X))

print (min(X))



X=("Rama", "Abhi", "Anuj")

print (max(X))

print (min(X))

print (sorted(X))
```

```
print (sorted(X, reverse=True))
```

Comparing tuples

Usually, The function cmp() compares the values of two arguments x and y:

```
cmp(x,y)
```

The return value is:

A negative number if x is less than y.

Zero if x is equal to y.

A positive number if x is greater than y.

The built-in cmp() function will typically return only the values -1, 0, or 1.

```
X,Y=(12,33),(22,44)

print(cmp(x,y))

print (cmp(y,x))


X,Y=(12,33,5),(22,44,2)

print(cmp(x,y))

print (cmp(y,x))


X,Y=(12,33),(12,33)

print(cmp(x,y))

print (cmp(y,x))
```

We can **use all(), any(), enumerate** functions with tuples. For example, all() function returns true if all the elements are true (or tuple is empty); otherwise returns False(if the the tuple is emply also it returns False). Similarly, any() method returns True if atleast one of its elements is True; otherwise falls.

enumerate prints all elements of the tuple along with their index.

```
x=(992,33,33,None)
print (any(x))
print (all(x))
for y in enumerate(x):
print (y)
```

```
x=(992,33,33,45,55,66)
for y in enumerate(x):
print (y)
for y in enumerate(x,3):
print (y)
```

```
X=[[i,j] for i,j in enumerate(x)]
print (type(X))
print (X)
```

```
X=[(i,j) for i,j in enumerate(x)]
print (type(X))
```

```
print (X)

X=((i,j) for i,j in enumerate(x))
print (type(X))
print (X)

X=[{i,j} for i,j in enumerate(x)]
print (type(X))
print (X)

X={i:j for i,j in enumerate(x)}
print (type(X))
print (X)
```

The following program illustrates the use of sorted(), reversed(), zip() methods with tuples.

```
albums = ('A', 'B', 'C', 'D')
years = (1976, 1987, 1990, 2003)

for album in sorted(albums):
print (album)

for album in reversed(albums):
print (album)
for i, album in enumerate(albums):
print (i, album)
```

```
for album, yr in zip(albums, years):

print (yr, album)
```

Convert a tuple to a string

```
t=(1,2,3,4,5)
str(t)
```

Sets and Frozen Sets

1. Sets do not contain duplicates; if we add a valute that already is in a set, the set remains unchanged; this means we can often add a value to a set without first checking if it is in the set: if it isn't in the set, it is added; if it is in the set, the set remains unchanged.
2. Sets are unordered: we cannot index different values, and when we iterate through them the order of the values produced is not fixed (like dictionaries)
3. All values in sets (like keys in dictionaries) must be immutable. So we can have sets of tuples, but not sets of lists.
4. a = set() is the empty set (no/0 values)
5. b = {'a', b', 'c'} is a set of str
6. c = { 1, 2, 4, 5} is a set of integers
7. d = {('NB','Venkat'), ('GV', 'Saradamba')} is a set of tuples

```
s=()

ss={}

sss={1}

ssss={'a':12}

sssss=set()

print type(s)

print type(ss)

print type(sss)

print type(ssss)

print type(sssss)
```

Set operations

1. len(): We can compute the length of a set (number of values at the top-level). For example, the with the sets defined above,

   len(a) is 0; len(b) is 3; len(c) is 4; len(d) is 2

2. We cannot index the sets as the values in sets are unordered.
3. Slicing is not possible with sets
4. Checking containment: the `in/notin` operators can be used with sets. These operators work on the values in a set. For example,

   'a' in set of a (above) is False

   'a' in set b is True;

5. We can notCatenate sets
6. We can not use Multiplication with sets
7. We can iterate over elements of a set using for loop. For example, the following program allows us to iterate over the set b.

```
b={'a', 'b', 'c'}

for i in b:

print(i)

alist=[1, 2, 3, 5,2,9,3]

aset  = set(alist)

printlen(aset), len(alist)

printalist

printaset



# Asaset has no duplicated values, its length will be smaller than or equal to alist



#We can generate a list from a set

blist = list(aset)

printlen(blist),len(aset)
```

printaset

printblist

#Length of aset and blist will be exactly same

Set Objects

Instances of `Set` and `ImmutableSet` both provide the following operations:

Operation	Equivalent	Result
`len(s)`		number of elements in set *s* (cardinality)
`xins`		test *x* for membership in *s*
`xnotins`		test *x* for non-membership in *s*
`s.issubset(t)`	`s<=t`	test whether every element in *s* is in *t*
`s.issuperset(t)`	`s>=t`	test whether every element in *t* is in *s*
`s.union(t)`	`s|t`	new set with elements from both *s* and *t*
`s.intersection(t)`	`s&t`	new set with elements common to *s* and *t*
`s.difference(t)`	`s-t`	new set with elements in *s* but not in *t*
`s.symmetric_difference(t)`	`s^t`	new set with elements in either *s* or *t* but not both
`s.copy()`		new set with a shallow copy of *s*

Frozen Sets Frozen sets are immutable objects that only support methods and operators that produce a result without a?ecting the frozen set or sets to which they are applied.

```
# Python program to demonstrate differences
# between normal and frozen set

# Same as {"a", "b","c"}
normal_set =set(["a", "b","c"])

# Adding an element to normal set is fine
normal_set.add("d")
```

```
print("Normal Set")
print(normal_set)

# A frozen set
frozen_set =frozenset(["e", "f", "g"])

print("Frozen Set")
print(frozen_set)

# Uncommenting below line would cause error as
# we are trying to add element to a frozen set
# frozen_set.add("h")
```

1. add(x) Method: Adds the item x to set if it is not already present in the set.

```
people = {"Jay", "Idrish", "Archil"}
people.add("Daxit")
```

union(s) Method: Returns a union of two set.Using the '|' operator between 2 sets is the same as writing set1.union(set2)

```
people = {"Jay", "Idrish", "Archil"}
vampires = {"Karan", "Arjun"}
population = people.union(vampires)
```

OR

```
population = people|vampires
```

3.**intersect(s) Method:** Returns an intersection of two sets.The '&' operator comes can also be used in this case.

```
victims = people.intersection(vampires)
```

-> Set victims will contain the common element of people and vampire

4. difference(s) Method: Returns a set containing all the elements of invoking set but not of the second set. We can use '-' operator here.

```
safe = people.difference(vampires)
```

OR

```
safe = people - vampires
```

-> Set safe will have all the elements that are in people but not vampire

5. clear() Method: Empties the whole set.

```
victims.clear()
```

-> Clears victim set

However there are two major pitfalls in Python sets:

Operators for Sets

Sets and frozen sets support the following operators:

key in s # containment check

key not in s # non-containment check

s1 == s2 # s1 is equivalent to s2

s1 != s2 # s1 is not equivalent to s2

s1 <= s2 # s1is subset of s2

s1 < s2 # s1 is proper subset of s2

s1 >= s2 # s1is superset of s2

s1 > s2 # s1 is proper superset of s2

s1 | s2 # the union of s1 and s2

s1 & s2 # the intersection of s1 and s2

s1 – s2 # the set of elements in s1 but not s2

s1 ˆ s2 # the set of elements in precisely one of s1 or s2

```
# Python program to demonstrate working# of
# Set in Python

# Creating two sets
set1 =set()
set2 =set()

# Adding elements to set1
fori inrange(1, 6):
    set1.add(i)

# Adding elements to set2
fori inrange(3, 8):
    set2.add(i)
```

```
print("Set1 = ", set1)
print("Set2 = ", set2)
print("\n")

# Union of set1 and set2
set3 =set1 | set2# set1.union(set2)
print("Union of Set1 & Set2: Set3 = ", set3)

# Intersection of set1 and set2
set4 =set1 & set2# set1.intersection(set2)
print("Intersection of Set1 & Set2: Set4 = ", set4)
print("\n")

# Checking relation between set3 and set4
ifset3 > set4: # set3.issuperset(set4)
    print("Set3 is superset of Set4")
elifset3 < set4: # set3.issubset(set4)
    print("Set3 is subset of Set4")
else: # set3 == set4
    print("Set3 is same as Set4")

# displaying relation between set4 and set3
ifset4 < set3: # set4.issubset(set3)
    print("Set4 is subset of Set3")
    print("\n")

# difference between set3 and set4
set5 =set3 -set4
print("Elements in Set3 and not in Set4: Set5 = ", set5)
print("\n")

# checkv if set4 and set5 are disjoint sets
ifset4.isdisjoint(set5):
    print("Set4 and Set5 have nothing in common\n")

# Removing all the values of set5
set5.clear()

print("After applying clear on sets Set5: ")
print("Set5 = ", set5)
```

Dictionaries

A dictionary is like a list, but more general. In a list, the positions (a.k.a. indices) have to be integers; in a dictionary the indices can be (almost) any type

a dictionary as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value.
The association of a key and a value is called a key-value pair or sometimes an item.

The function dictcreates a new dictionary with no items. Because dictis the

name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> print eng2sp
```

The squiggly-brackets, {}, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```
>>>eng2sp['one'] = 'uno'
```

If we
print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> print eng2sp
```

create a new dictionary with three items:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}

>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

The order of the key-value pairs is not the same.
In general, the order of items in a dictionary is unpredictable.
But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> print eng2sp['two']
```

The lenfunction works on dictionaries; it returns the number of key-value pairs:

```
>>>len(eng2sp)
3
```

The in operator works on dictionaries; it tells you whether something appears as a*key* in the dictionary.

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

To see whether something appears as a value in a dictionary, you can use the methodvalues, which returns the values as a list, and then use the in operator:

```
>>>vals = eng2sp.values()
>>> 'uno' in vals
True
```

The in operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As the list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python uses an algorithm called a hash table

Dictionary as a set of counters

Suppose you are given a string and you want to count how many times each letter appears

Dictionaries have a method called **get**that takes a key and a default value. If the key appears in the dictionary, get returns the corresponding value; otherwise it returns the default value

```
>>>h.get('a', 0)
1
>>>h.get('b', 0)
```

Looping and dictionaries

Again, the keys are in no particular order.
If you want to print the keys in alphabetical order, you first make a list of the keys in the dictionary using the keys method available in dictionary objects, and then sort that list and loop through the sorted list

We can convert a list of lists into a dictionary

```
grades=dict([['A',10],['B',8],['C',6], ['D',4],['E',2]])
print(grades)
print(type(grades))
```

Dictionary Methods

Dictionaries have a number of useful built-in methods. The following table provides a summary and more details can be found in the Python Documentation.

Method	Parameters	Description
keys	None	Returns a view of the keys in the dictionary
values	None	Returns a view of the values in the dictionary
items	None	Returns a view of the key-value pairs in the dictionary
get	Key	Returns the value associated with key; None otherwise
get	key,alt	Returns the value associated with key; alt otherwise

```
terse={}
terse['abscond']='Dis-appear without knowledge'
terse['mutable']='able to change'
terse['abstain']='stop from voting'
terse['abbey']='A place where people will pray for god'
terse['awkward']='Not in proper order'

for k in terse:
print ("Key",k)

kys=list(terse.keys())
print(kys)

for k in terse.keys():
print("Key",k)
```

Aliasing in Dictionaries

Whenever two variables refer to the same dictionary object, changes to one affect the other. This problem is known as aliasing problem which we did encounter in lists also. In order to avoid this, we can use copy() method of dictionary class to create a copy of it.

```
terse={'abscond':'Dis-appear without knowledge', 'mutable':'able to change','abstain':'stop from voting'}
compact=terse
print(compact is terse)
print(terse)
print(compact)
compact['abstain']='xxxxx'
print(terse)
print(compact)

concise=terse.copy()
print(concise is terse)
concise['abstain']='stop from voting'
print(terse)
print(concise)terse={'abscond':'Dis-appear without knowledge', 'mutable':'able to change','abstain':'stop
from voting'}
compact=terse
print(compact is terse)
print(terse)
print(compact)
compact['abstain']='xxxxx'
print(terse)
print(compact)

concise=terse.copy()
print(concise is terse)
concise['abstain']='stop from voting'
print(terse)
print(concise)
```

Another way of constructing

```
>>>dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>>
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
>>>
>>>dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `items()` method.

```
>>>
>>>knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>>for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

```
>>>
>>>for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe

questions=['name','quest','favoritecolor']
>>>answers=['lancelot','the holy grail','blue']
>>>forq,ainzip(questions,answers):
... print('What is your {0}?  It is {1}.'.format(q,a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favoritecolor?  It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the reversed() function.

```
>>>
>>>for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
```

```
3
1
```

`chr`(*i*)

Return a string of one character whose ASCII code is the integer *i*. For example, `chr(97)` returns the string `'a'`. This is the inverse of `ord()`

Dictionary Comprehensions

```
P = {x: x ** 2 for x in range(5)}
print(P)
Q = {x: 'A' + str(x) for x in range(10)}
print(Q)

grades = {x: chr(70-x//2) for x in range(2,11,2)}
print(grades)
#inverting the dictionary of grades
points={v: k for k, v in grades.items()}
print(points)
```

```
numbers = [i for i in xrange(1,11)] + [i for i in xrange(1,6)]
print(numbers)

unique_numbers = []
for n in numbers:
if n not in unique_numbers:
unique_numbers.append(n)

print(unique_numbers)

#using dictionary comprehension
unique_numbers1={ d:d for d in numbers }.values()
print(unique_numbers1)
```

Some times, we want to initialize all values of a dictionary with 0 or none. For this, we can use list comprehension or dict.fromkeys() method as shown below.

`fromkeys()` is a class method that returns a new dictionary. *value* defaults to `None`.

```
x={k:0 for k in range(10)}
y={k:Nonefor k in range(10)}


z=dict.fromkeys(range(10))
```

Dictionaries for Sparse Matrices

m={(0,3):1,(2,1):2,(3,2):1}

matrix = {(0, 3): 1, (2, 1): 2, (4, 3): 3}

print(matrix.get((0,3)))

print(matrix.get((1, 3), 0))

o/p

```
{(0, 0): 0, (0, 1): 1, (0, 2): 2, (0, 3): 3, (1, 0): 1, (1, 1): 2, (1, 2): 3,
(1, 3): 4, (2, 0): 2, (2, 1): 3, (2, 2): 4, (2, 3): 5, (3, 0): 3, (3, 1): 4,
(3, 2): 5, (3, 3): 6}
```

writedict for it

uses nested dictionary comprehension generates a dictionary of an identity matrix of size 4x4.

```
o/p {(0, 0): 1, (0, 1): 0, (0, 2): 0, (0, 3): 0, (1, 0): 0, (1, 1): 1, (1,
2): 0, (1, 3): 0, (2, 0): 0, (2, 1): 0, (2, 2): 1, (2, 3): 0, (3, 0): 0, (3,
1): 0, (3, 2): 0, (3, 3): 1}
```

Assume that you have a list a set of words and you want to display how many times each word has appeared. We use dictionary comprehension to achieve this

mylist = ['hello', 'hi', 'hello', 'today', 'morning', 'again', 'hello']

mydict = {k:mylist.count(k) for k in mylist}

print(mydict)