Web Front End Technology

Python Part-1

Name……

Class……

Group……..

Contents

1 Python 3 Introduction

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently whereas the other languages use punctuations. It has fewer syntactical constructions than other languages.

Python is Interpreted − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.

Python is Interactive − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

Python is Object-Oriented − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

Python is a Beginner's Language − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

1.1 History of Python

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

Python 1.0 was released in November 1994. In 2000, Python 2.0 was released. Python 2.7.11 is the latest edition of Python 2.

Meanwhile, **Python 3.0** was released in 2008. Python 3 is not backward compatible with Python 2. The emphasis in Python 3 had been on the removal of duplicate programming constructs and modules so that "There should be one -- and preferably only one -- obvious way to do it." Python 3.5.1 is the latest version of Python 3.

1.2 Python 3 Features

Python's features include −

Easy-to-learn − Python has few keywords, simple structure, and a clearly defined syntax. This allows a student to pick up the language quickly.

Easy-to-read − Python code is more clearly defined and visible to the eyes.

Easy-to-maintain − Python's source code is fairly easy-to-maintain.

A broad standard library − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.

Interactive Mode − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.

Portable − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.

Extendable − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.

Databases − Python provides interfaces to all major commercial databases.

GUI Programming − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.

Scalable − Python provides a better structure and support for large programs than shell scripting.
Apart from the above-mentioned features, Python has a big list of good features. A, few are listed below −

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

2 Python 3 - Basic Syntax

The Python language has many similarities to Perl, C, and Java. However, there are some definite differences between the languages.

First Python Program
Let us execute the programs in different modes of programming.

Interactive Mode Programming
Invoking the interpreter without passing a script file as a parameter brings up the following prompt −

Type the following text at the Python prompt and press Enter −
>>> print ("Hello, Python!")
This produces the following result −

Hello, Python!

Script Mode Programming
Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.
Let us write a simple Python program in a script. Python files have the extension **.py**. Type the following source code in a test.py file −
print ("Hello, Python!")

3 Python 3 Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers −

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strong private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

4 Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constants or variables or any other identifier names. All the Python keywords contain lowercase letters only.

And	exec	not
As	finally	or
assert	for	pass
break	from	print
class	global	raise
continue	if	return
Def	import	try
Del	in	while

Elif	is	with
else	lambda	yield
except		

5 Lines and Indentation

Python does not use braces({}) to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example −

```
if True:
   print ("True")

else:
   print ("False")
```

However, the following block generates an error −

```
if True:
   print ("Answer")
   print ("True")

else:
   print ("Answer")
   print ("False")
```

Thus, in Python all the continuous lines indented with the same number of spaces would form a block. The following example has various statement blocks −

Note − Do not try to understand the logic at this point of time. Just make sure you understood the various blocks even if they are without braces.

Multi-Line Statements

Statements in Python typically end with a new line. Python, however, allows the use of the line continuation character (\) to denote that the line should continue. For example −

```
total = item_one + \
        item_two + \
        item_three
```

The statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example −

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Quotation in Python

Python accepts single ('), double (") and triple ('" or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal −

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

6 Python 3 Comments

Python supports two types of comments:

1) Single lined comment:
In case user wants to specify a single line comment, then comment must start with ?#?
Eg:

```
# This is single line comment.
```

2) Multi lined Comment:
Multi lined comment can be given inside triple quotes.
eg:

```
""" This
   Is
   Multipline comment"'
```

eg:

```
#single line comment
print("Hello Python")
"""This is
multiline comment"'
```

7 Python 3 - Variable Types

Variables are nothing but reserved memory locations to store values. It means that when you create a variable, you reserve some space in the memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to the variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables
Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example −

```
#!/usr/bin/python3
```

```
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string

print (counter)
print (miles)
print (name)
```

Here, 100, 1000.0 and "John" are the values assigned to counter, miles, and name variables, respectively. This produces the following result −

```
100
1000.0
John
```

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously.

For example −

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all the three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example −

```
a, b, c = 1, 2, "john"
```

Here, two integer objects with values 1 and 2 are assigned to the variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

8 Python 3 - Basic Operators

Operators are the constructs, which can manipulate the value of operands. Consider the expression 4 + 5 = 9. Here, 4 and 5 are called the operands and + is called the operator.

Types of Operator

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

Let us have a look at all the operators one by one.

8.1 Python 3 Arithmetic Operators

Assume variable **a** holds the value 10 and variable **b** holds the value 21, then −

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 31
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -11

* Multiplication	Multiplies values on either side of the operator	a * b = 210
/ Division	Divides left hand operand by right hand operand	b / a = 2.1
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 1
** 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

8.2 Python 3 Comparison Operators

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

Assume variable **a** holds the value 10 and variable **b** holds the value 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.	(a!= b) is 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.

8.3 Python 3 Assignment Operators

Assume variable **a** holds the value 10 and variable **b** holds the value 20, then −

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

8.4 Python 3 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

Python's built-in function bin() can be used to obtain binary representation of an integer number. The following Bitwise operators are 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 operand's value is moved left by the number of bits specified by the right operand.	a << = 240 (means 1111 0000)
>> Binary Right Shift	The left operand's value is moved right by the number of bits specified by the right operand.	a >> = 15 (means 0000 1111)

8.5 Python 3 Logical Operators

The following logical operators are supported by Python language. Assume variable **a** holds True and variable **b** holds False then −

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is False.
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 True.

8.6 Python 3 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.

8.7 Python 3 Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators as 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).

8.8 Python 3 Operators Precedence

The following table lists all operators from highest precedence to the lowest.

S.No.	Operator & Description
1	****** Exponentiation (raise to the power)
2	**~ + -** Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
3	*** / % //** Multiply, divide, modulo and floor division
4	**+ -** Addition and subtraction
5	**>> <<** Right and left bitwise shift
6	**&** Bitwise 'AND'

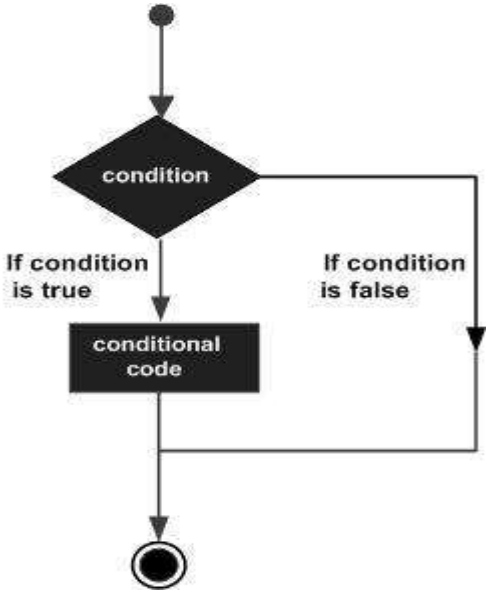
7	**^ |** Bitwise exclusive `OR' and regular `OR'
8	**<= < > >=** Comparison operators
9	**<> == !=** Equality operators
10	**= %= /= //= -= += *= **=** Assignment operators
11	**is is not** Identity operators
12	**in not in** Membership operators
13	**not or and** Logical operators

9 Python 3 - Decision Making

Decision-making is the anticipation of conditions occurring during the execution of a program and specified actions taken according to the conditions.

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

Following is the general form of a typical decision making structure found in most of the programming languages −



Python programming language assumes any **non-zero** and **non-null** values as TRUE, and any **zero** or **null values** as FALSE value.
Python programming language provides the following types of decision-making statements.

S.No.	Statement & Description
1	**if statements** An **if statement** consists of a boolean expression followed by one or more statements.
2	**if…else statements** An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE.
3	**nested if statements** You can use one **if** or **else if** statement inside another **if** or **else if**statement(s).

Let us go through each decision-making statement quickly.

Single Statement Suites
If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.

Example
```
var = 100
if ( var  == 100 ) : print ("Value of expression is 100")
print ("Good bye!")
```

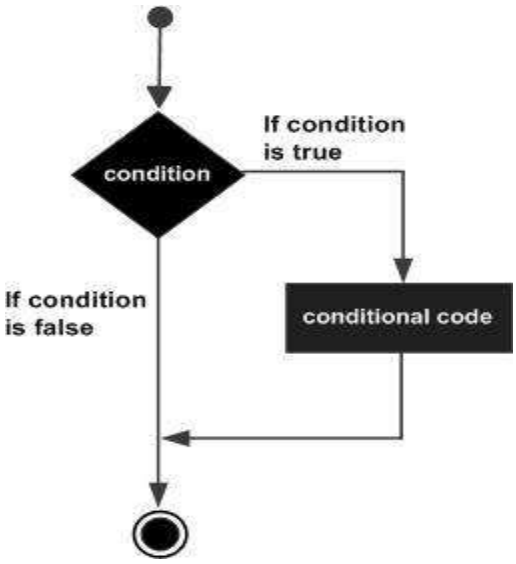
Output
When the above code is executed, it produces the following result −
```
Value of expression is 100
Good bye!
```

9.1 Python 3 - IF Statement

The IF statement is similar to that of other languages. The **if** statement contains a logical expression using which the data is compared and a decision is made based on the result of the comparison.

Syntax
```
if expression:
   statement(s)
```
If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. In Python, statements in a block are uniformly indented after the : symbol. If boolean expression evaluates to FALSE, then the first set of code after the end of block is executed.

Flow Diagram



Example

```
var1 = 100
if var1:
   print ("1 - Got a true expression value")
   print (var1)

var2 = 0
if var2:
   print ("2 - Got a true expression value")
   print (var2)
print ("Good bye!")
```

Output

When the above code is executed, it produces the following result −

```
1 - Got a true expression value
100
Good bye!
```

9.2 Python 3 - IF…ELIF…ELSE Statements

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

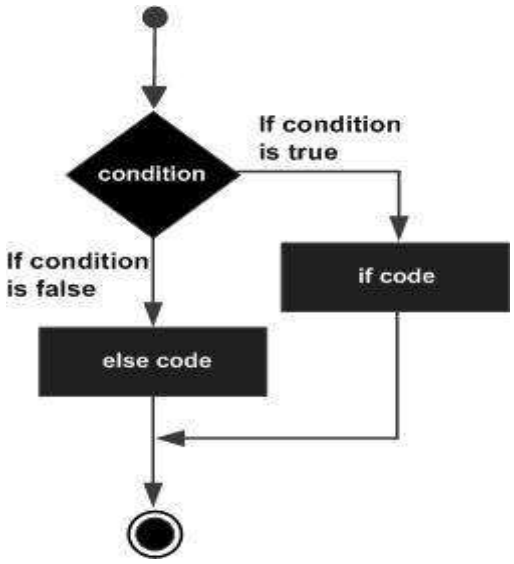
The else statement is an optional statement and there could be at the most only one **else** statement following **if**.

Syntax

The syntax of the **if…else** statement is −

```
if expression:
   statement(s)

else:
   statement(s)
```

Flow Diagram



Example

```
amount = int(input("Enter amount: "))

if amount<1000:
   discount = amount*0.05
   print ("Discount",discount)
else:
   discount = amount*0.10
   print ("Discount",discount)

print ("Net payable:",amount-discount)
```

Output

In the above example, discount is calculated on the input amount. Rate of discount is 5%, if the amount is less than 1000, and 10% if it is above 10000. When the above code is executed, it produces the following result −

```
Enter amount: 600
Discount 30.0
Net payable: 570.0

Enter amount: 1200
Discount 120.0
Net payable: 1080.0
```

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 the 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)
```

Core Python does not provide switch or case statements as in other languages, but we can use if..elif...statements to simulate switch case as follows −

Example

```
amount = int(input("Enter amount: "))

if amount<1000:
   discount = amount*0.05
   print ("Discount",discount)
elif amount<5000:
   discount = amount*0.10
   print ("Discount",discount)
else:
   discount = amount*0.15
   print ("Discount",discount)

print ("Net payable:",amount-discount)
```

When the above code is executed, it produces the following result −

```
Enter amount: 600
Discount 30.0
Net payable: 570.0

Enter amount: 3000
Discount 300.0
Net payable: 2700.0

Enter amount: 6000
Discount 900.0
Net payable: 5100.0
```

9.3 Python 3 - 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.

In a nested **if** construct, you can have an **if…elif…else** construct inside another **if…elif…else** construct.

Syntax

The syntax of the nested if...elif...else construct may be −

```
if expression1:
   statement(s)
   if expression2:
```

```
    statement(s)
  elif expression3:
    statement(s)
  else
    statement(s)
elif expression4:
  statement(s)
else:
  statement(s)
```

Example

```
num = int(input("enter number"))
if num%2 == 0:
  if num%3 == 0:
    print ("Divisible by 3 and 2")
  else:
    print ("divisible by 2 not divisible by 3")
else:
  if num%3 == 0:
    print ("divisible by 3 not divisible by 2")
  else:
    print  ("not Divisible by 2 not divisible by 3")
```

Output

When the above code is executed, it produces the following result −

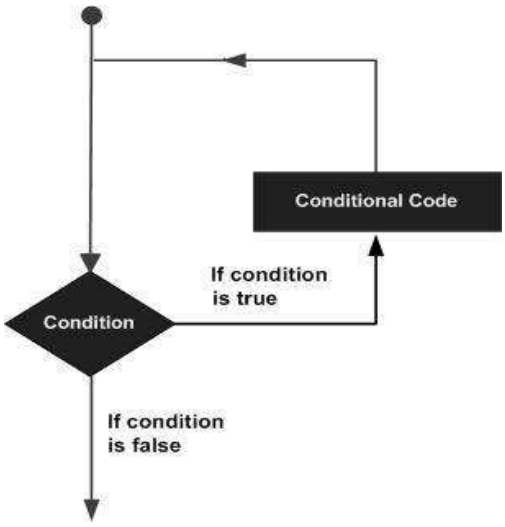
```
enter number8
divisible by 2 not divisible by 3

enter number15
divisible by 3 not divisible by 2

enter number12
Divisible by 3 and 2

enter number5
not Divisible by 2 not divisible by 3
```

10 Python 3 - Loops

In general, statements are executed sequentially − The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −



Python programming language provides the following types of loops to handle looping requirements.

S.No.	Loop Type & Description
1	**while loop** Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	**for loop** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	**nested loops** You can use one or more loop inside any another while, or for loop.

10.1 Python 3 - while Loop Statements

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

Syntax
The syntax of a **while** loop in Python programming language is −

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

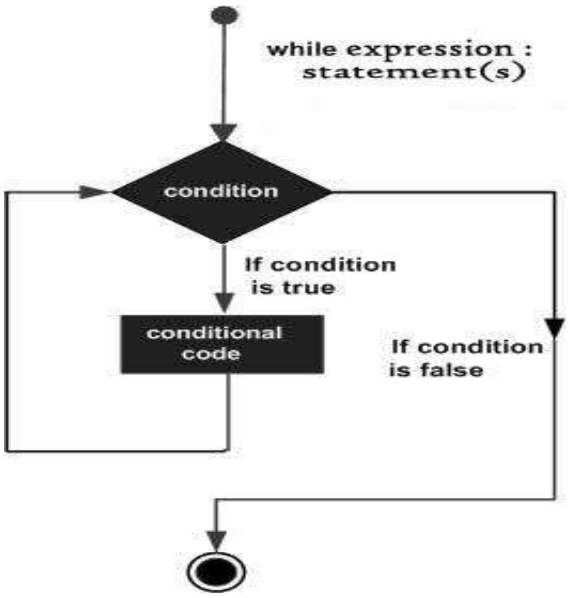
Here, **statement(s)** may be a single statement or a block of statements with uniform indent.

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.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Flow Diagram



Here, a key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

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

print ("Good bye!")
```

Output

When the above code is executed, it produces the following result −

```
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
Good bye!
```

The block here, consisting of the print and increment statements, is executed repeatedly until count is no longer less than 9. With each iteration, the current value of the index count is displayed and then increased by 1.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes FALSE. You must be cautious when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.
An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

Example

```
var = 1
while var == 1 :  # This constructs an infinite loop
   num = int(input("Enter a number  :"))
   print ("You entered: ", num)
print ("Good bye!")
```

Output

When the above code is executed, it produces the following result −

```
Enter a number  :20
You entered:  20
Enter a number  :29
You entered:  29
Enter a number  :3
You entered:  3
Enter a number  :11
You entered:  11
Enter a number  :22
You entered:  22
Enter a number  :Traceback (most recent call last):
   File "examples\test.py", line 5, in
      num = int(input("Enter a number  :"))
KeyboardInterrupt
```

The above example goes in an infinite loop and you need to use CTRL+C to exit the program.

Using else Statement with Loops

Python supports having 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.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise the else statement gets executed.

Example

```
count = 0
while count < 5:
   print (count, " is  less than 5")
```

```
   count = count + 1
else:
   print (count, " is not less than 5")
```

Output
When the above code is executed, it produces the following result −
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

Single Statement Suites
Similar to the **if** statement syntax, if your **while** clause consists only of a single statement, it may be placed on the same line as the while header.

Example
Here is the syntax and example of a **one-line while** clause −

```
flag = 1

while (flag): print ('Given flag is really true!')

print ("Good bye!")
```

The above example goes into an infinite loop and you need to press CTRL+C keys to exit.

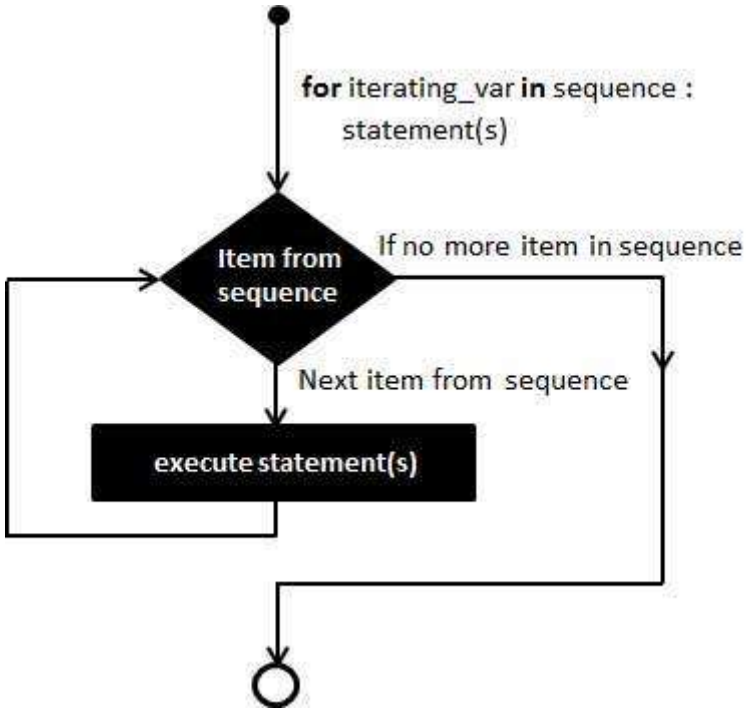
10.2 Python 3 - for Loop Statements

The for statement 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.
Flow Diagram



The range() function

The built-in function range() is the right function to iterate over a sequence of numbers. It generates an iterator of arithmetic progressions.

Example

```
>>> range(5)
range(0, 5)
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Example

range() generates an iterator to progress integers starting with 0 upto n-1. To obtain a list object of the sequence, it is typecasted to list(). Now this list can be iterated using the for statement.

```
>>> for var in list(range(5)):
  print (var)
```

Output

This will produce the following output.

```
0
1
2
3
4
```

Example

```
for letter in 'Python':     # traversal of a string sequence
  print ('Current Letter :', letter)
print()
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # traversal of List sequence
  print ('Current fruit :', fruit)
```

```
print ("Good bye!")
```

Output
When the above code is executed, it produces the following result −

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

Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Iterating by Sequence Index
An alternative way of iterating through each item is by index offset into the sequence itself. Following is a simple example −

Example

```
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print ('Current fruit :', fruits[index])

print ("Good bye!")
```

Output
When the above code is executed, it produces the following result −

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Here, we took the assistance of the len() built-in function, which provides the total number of elements in the tuple as well as the range() built-in function to give us the actual sequence to iterate over.

Using else Statement with Loops
Python supports having an else statement associated with a loop statement.

If the **else** statement is used with a **for** loop, the **else** block is executed only if for loops terminates normally (and not by encountering break statement).

If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

Example
The following example illustrates the combination of an else statement with a **for** statement that searches for even number in given list.

```
numbers = [11,33,55,39,55,75,37,21,23,41,13]

for num in numbers:
  if num%2 == 0:
    print ('the list contains an even number')
    break
else:
  print ('the list doesnot contain even number')
```

Output
When the above code is executed, it produces the following result −
the list does not contain even number

10.3 Python 3 - Nested loops

Python programming language allows the usage of one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

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

The syntax for a nested while loop statement in Python programming language is as follows −

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

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example a **for** loop can be inside a while loop or vice versa.

Example
The following program uses a nested-for loop to display multiplication tables from 1-10.

```
import sys
for i in range(1,11):
  for j in range(1,11):
    k=i*j
    print (k, end=' ')
  print()
```

The print() function inner loop has **end=' '** which appends a space instead of default newline. Hence, the numbers will appear in one row.
Last print() will be executed at the end of inner for loop.

Output
When the above code is executed, it produces the following result −
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100

11 Loop Control Statements

The Loop control statements change the execution from its normal sequence. When the execution leaves a scope, all automatic objects that were created in that scope are destroyed.
Python supports the following control statements.

S.No.	Control Statement & Description
1	**break statement** Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	**continue statement** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	**pass statement** The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Let us go through the loop control statements briefly.

Iterator and Generator

Iterator is an object which allows a programmer to traverse through all the elements of a collection, regardless of its specific implementation. In Python, an iterator object implements two methods, **iter()** and **next()**.
String, List or Tuple objects can be used to create an Iterator.

```
list = [1,2,3,4]
it = iter(list) # this builds an iterator object
print (next(it)) #prints next available element in iterator
Iterator object can be traversed using regular for statement
for x in it:
   print (x, end=" ")
or using next() function
while True:
```

```
  try:
    print (next(it))
  except StopIteration:
    sys.exit() #you have to import sys module for this
```

A **generator** is a function that produces or yields a sequence of values using yield method.
When a generator function is called, it returns a generator object without even beginning execution of the function.

When the next() method is called for the first time, the function starts executing until it reaches the yield statement, which returns the yielded value. The yield keeps track i.e. remembers the last execution and the second next() call continues from previous value.

Example
The following example defines a generator, which generates an iterator for all the Fibonacci numbers.

```
import sys
def fibonacci(n): #generator function
  a, b, counter = 0, 1, 0
  while True:
    if (counter > n):
      return
    yield a
    a, b = b, a + b
    counter += 1
f = fibonacci(5) #f is iterator object

while True:
  try:
    print (next(f), end=" ")
  except StopIteration:
    sys.exit()
```

11.1 Python 3 - break statement

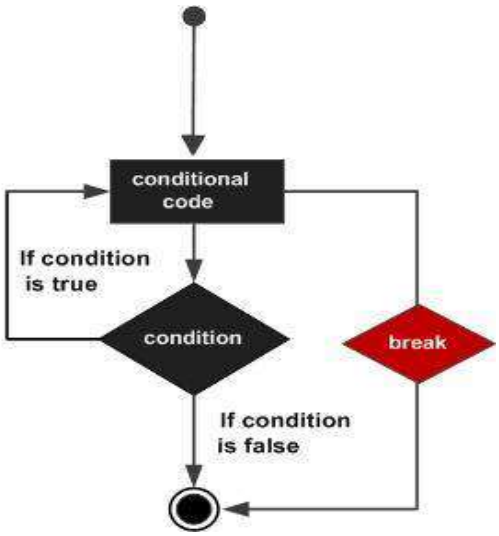
The **break** statement is used for premature termination of the current loop. After abandoning the loop, execution at the next statement is resumed, just like the traditional break statement in C.
The most common use of break is when some external condition is triggered requiring a hasty exit from a loop. The **break** statement can be used in both *while* and *for* loops.

If you are using nested loops, the break statement stops the execution of the innermost loop and starts executing the next line of the code after the block.

Syntax
The syntax for a **break** statement in Python is as follows −

```
break
```

Flow Diagram



Example

```
for letter in 'Python':     # First Example
   if letter == 'h':
      break
   print ('Current Letter :', letter)

var = 10                    # Second Example
while var > 0:
   print ('Current variable value :', var)
   var = var -1
   if var == 5:
      break

print ("Good bye!")
```

Output

When the above code is executed, it produces the following result −

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

The following program demonstrates use of the break in a for loop iterating over a list. User inputs a number, which is searched in the list. If it is found, then the loop terminates with the 'found' message.

Example

```
no = int(input('any number: '))
numbers = [11,33,55,39,55,75,37,21,23,41,13]

for num in numbers:
   if num == no:
```

```
    print ('number found in list')
    break
else:
  print ('number not found in list')
```

Output
The above program will produce the following output −

```
any number: 33
number found in list

any number: 5
number not found in list
```

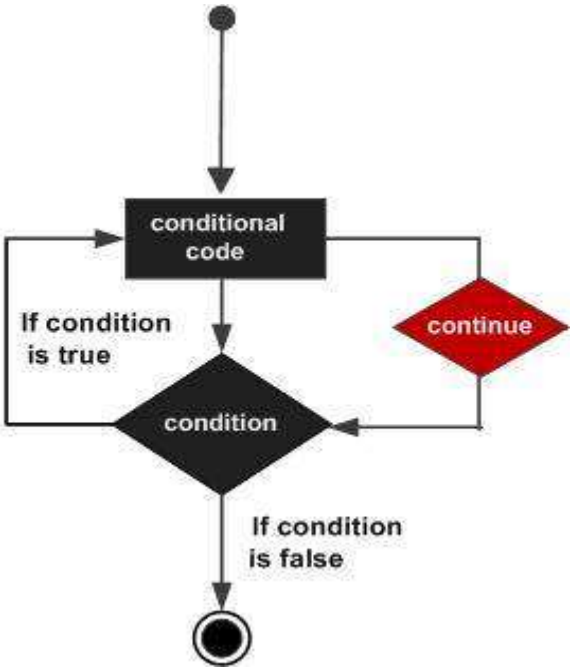
11.2 Python 3 - continue statement

The **continue** statement in Python returns the control to the beginning of the current loop. When encountered, the loop starts next iteration without executing the remaining statements in the current iteration.
The **continue** statement can be used in both *while* and *for* loops.

Syntax

```
continue
```

Flow Diagram



Example

```
for letter in 'Python':     # First Example
  if letter == 'h':
    continue
  print ('Current Letter :', letter)

var = 10                    # Second Example
while var > 0:
  var = var -1
```

```
   if var == 5:
      continue
   print ('Current variable value :', var)
print ("Good bye!")
```

Output
When the above code is executed, it produces the following result −

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
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
Good bye!
```

11.3 Python 3 - pass Statement

It is used when a statement is required syntactically but you do not want any command or code to execute.

The **pass** statement is a *null* operation; nothing happens when it executes. The **pass** statement is also useful in places where your code will eventually go, but has not been written yet i.e. in stubs.

Syntax

```
pass
```

Example

```
#!/usr/bin/python3

for letter in 'Python':
   if letter == 'h':
      pass
      print ('This is pass block')
   print ('Current Letter :', letter)

print ("Good bye!")
```

Output
When the above code is executed, it produces the following result −

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

```
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

12 Python 3 Data Types

Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.
There are various data types in Python. Some of the important types are listed below.

- Python Numbers
- Python List
- Python Tuple
- Python Strings
- Python Set
- Python Dictionary
- Conversion between data types

12.1 Python Numbers

Integers, floating point numbers and complex numbers falls under Python numbers category. They are defined as `int`, `float` and `complex` class in Python.
We can use the `type()` function to know which class a variable or a value belongs to and the `isinstance()` function to check if an object belongs to a particular class.

```
a = 5
print(a, "is of type", type(a))

a = 2.0
print(a, "is of type", type(a))

a = 1+2j
print(a, "is complex number?", isinstance(1+2j,complex))
```

Output

```
5 is of type <class 'int'>
2.0 is of type <class 'float'>
(1+2j) is complex number? True
```

Integers can be of any length, it is only limited by the memory available.
A floating point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. `1` is integer, `1.0` is floating point number.

Complex numbers are written in the form, `x + yj`, where `x` is the real part and `y` is the imaginary part.

12.2 Python List

Python offers a range of compound datatypes often referred to as sequences. List is one of the most frequently used and very versatile datatype used in Python.

12.2.1 How to create a list?

In Python programming, a list is created by placing all the items (elements) inside a square bracket [ ], separated by commas.
It can have any number of items and they may be of different types (integer, float, string etc.).

```
# empty list
my_list = []

# list of integers
my_list = [1, 2, 3]

# list with mixed datatypes
my_list = [1, "Hello", 3.4]
```

Also, a list can even have another list as an item. This is called nested list.

```
# nested list
my_list = ["mouse", [8, 4, 6], ['a']]
```

12.2.2 How to access elements from a list?

There are various ways in which we can access the elements of a list.

List Index

We can use the index operator [] to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4.
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 list are accessed using nested indexing.

```
my_list = ['p','r','o','b','e']
# Output: p
print(my_list[0])

# Output: o
print(my_list[2])

# Output: e
print(my_list[4])

# Error! Only integer can be used for indexing
# my_list[4.0]

# Nested List
n_list = ["Happy", [2,0,1,5]]

# Nested indexing
```

```
# Output: a
print(n_list[0][1])

# Output: 5
print(n_list[1][3])
```

Output

```
p
o
e
a
5
```

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_list = ['p','r','o','b','e']

# Output: e
print(my_list[-1])

# Output: p
print(my_list[-5])
```

Output

```
e
p
```

12.2.3 How to slice lists in Python?

We can access a range of items in a list by using the slicing operator (colon).

```
my_list = ['p','r','o','g','r','a','m','i','z']
# elements 3rd to 5th
print(my_list[2:5])

# elements beginning to 4th
print(my_list[:-5])

# elements 6th to end
print(my_list[5:])

# elements beginning to end
print(my_list[:])
```

Output

```
['o', 'g', 'r']
['p', 'r', 'o', 'g']
['a', 'm', 'i', 'z']
['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need two index that will slice that portion from the list.

How to change or add elements to a list?
List are mutable, meaning, their elements can be changed unlike string or tuple.
We can use assignment operator (=) to change an item or a range of items.

```
# mistake values
odd = [2, 4, 6, 8]

# change the 1st item
odd[0] = 1

# Output: [1, 4, 6, 8]
print(odd)

# change 2nd to 4th items
odd[1:4] = [3, 5, 7]

# Output: [1, 3, 5, 7]
print(odd)
```

Output

```
[1, 4, 6, 8]
[1, 3, 5, 7]
```

We can add one item to a list using append() method or add several items using extend()method.

```
odd = [1, 3, 5]

odd.append(7)

# Output: [1, 3, 5, 7]
print(odd)

odd.extend([9, 11, 13])

# Output: [1, 3, 5, 7, 9, 11, 13]
print(odd)
```

We can also use + operator to combine two lists. This is also called concatenation.
The * operator repeats a list for the given number of times.

```
odd = [1, 3, 5]

# Output: [1, 3, 5, 9, 7, 5]
print(odd + [9, 7, 5])

#Output: ["re", "re", "re"]
print(["re"] * 3)
```

Furthermore, we can insert one item at a desired location by using the method insert() or insert multiple items by squeezing it into an empty slice of a list.

```
odd = [1, 9]
odd.insert(1,3)

# Output: [1, 3, 9]
```

```
print(odd)

odd[2:2] = [5, 7]

# Output: [1, 3, 5, 7, 9]
print(odd)
```

Output

```
[1, 3, 9]
[1, 3, 5, 7, 9]
```

12.2.4 How to delete or remove elements from a list?

We can delete one or more items from a list using the keyword del. It can even delete the list entirely.

```
my_list = ['p','r','o','b','l','e','m']

# delete one item
del my_list[2]

# Output: ['p', 'r', 'b', 'l', 'e', 'm']
print(my_list)

# delete multiple items
del my_list[1:5]

# Output: ['p', 'm']
print(my_list)

# delete entire list
del my_list

# Error: List not defined
print(my_list)
```

Output

```
['p', 'r', 'b', 'l', 'e', 'm']
['p', 'm']

Traceback (most recent call last):
  File "<stdin>", line 19, in <module>
    print(my_list)
NameError: name 'my_list' is not defined
```

We can use remove() method to remove the given item or pop() method to remove an item at the given index.

The pop() method removes and returns the last item if index is not provided. This helps us implement lists as stacks (first in, last out data structure).

We can also use the clear() method to empty a list.

```
my_list = ['p','r','o','b','l','e','m']
my_list.remove('p')

# Output: ['r', 'o', 'b', 'l', 'e', 'm']
print(my_list)

# Output: 'o'
print(my_list.pop(1))

# Output: ['r', 'b', 'l', 'e', 'm']
print(my_list)

# Output: 'm'
print(my_list.pop())

# Output: ['r', 'b', 'l', 'e']
print(my_list)

my_list.clear()

# Output: []
print(my_list)
```

```
['r', 'o', 'b', 'l', 'e', 'm']
o
['r', 'b', 'l', 'e', 'm']
m
['r', 'b', 'l', 'e']
[]
```

Finally, we can also delete items in a list by assigning an empty list to a slice of elements.

```
>>> my_list = ['p','r','o','b','l','e','m']
>>> my_list[2:3] = []
>>> my_list
['p', 'r', 'b', 'l', 'e', 'm']
>>> my_list[2:5] = []
>>> my_list
['p', 'r', 'm']
```

12.2.5 Python List Methods

Methods that are available with list object in Python programming are tabulated below. They are accessed as list.method().

List append()

The append() method adds an item to the end of the list.

The append() method adds a single item to the existing list. It doesn't return a new list; rather it modifies the original list.

The syntax of append() method is:
list.append(item)

---

append() Parameters
The append() method takes a single item and adds it to the end of the list.
The item can be numbers, strings, another list, dictionary etc.

Return Value from append()
As mentioned, the append() method only modifies the original list. It doesn't return any value.

Example 1: Adding Element to a List
```
# animal list
animal = ['cat', 'dog', 'rabbit']

# an element is added
animal.append('guinea pig')

#Updated Animal List
print('Updated animal list: ', animal)
```

When you run the program, the output will be:
Updated animal list: ['cat', 'dog', 'rabbit', 'guinea pig']

Example 2: Adding List to a List
```
# animal list
animal = ['cat', 'dog', 'rabbit']

# another list of wild animals
wild_animal = ['tiger', 'fox']

# adding wild_animal list to animal list
animal.append(wild_animal)

#Updated List
print('Updated animal list: ', animal)
```

When you run the program, the output will be:
Updated animal list: ['cat', 'dog', 'rabbit', ['tiger', 'fox']]
It's important to notice that, a single item (wild_animal list) is added to the animal list in the above program.

List extend()
The extend() extends the list by adding all items of a list (passed as an argument) to the end.

The syntax of extend() method is:
list1.extend(list2)

Here, the elements of list2 are added to the end of list1.

---

extend() Parameters

As mentioned, the extend() method takes a single argument (a list) and adds it to the end.

If you need to add elements of other native datatypes (like tuple and set) to the list, you can simply use:

```
# add elements of a tuple to list
list.extend(list(tuple_type))
or even easier
list.extend(tuple_type)
```

Return Value from extend()

The extend() method only modifies the original list. It doesn't return any value.

Example 1: Using extend() Method

```
# language list
language = ['French', 'English', 'German']

# another list of language
language1 = ['Spanish', 'Portuguese']

language.extend(language1)

# Extended List
print('Language List: ', language)
```

When you run the program, the output will be:

```
Language List:  ['French', 'English', 'German', 'Spanish', 'Portuguese']
```

Example 2: Add Elements of Tuple and Set to List

```
# language list
language = ['French', 'English', 'German']

# language tuple
language_tuple = ('Spanish', 'Portuguese')

# language set
language_set = {'Chinese', 'Japanese'}

# appending element of language tuple
language.extend(language_tuple)

print('New Language List: ', language)

# appending element of language set
language.extend(language_set)

print('Newest Language List: ', language)
```

When you run the program, the output will be:
New Language List: ['French', 'English', 'German', 'Spanish', 'Portuguese']
Newest Language List: ['French', 'English', 'German', 'Spanish', 'Portuguese', 'Japanese', 'Chinese']

The native datatypes like tuple and set passed to extend() method is automatically converted to list. And, the elements of the list are appended to the end.

You can also add items of a list to another list using `+` or `+=` operator. For example,

```
a = [1, 2]
b = [3, 4]

a += b

# Output: a = [1, 2, 3, 4]
print('a = ', a)
```

List insert()

The insert() method inserts the element to the list at the given index.

The syntax of insert() method is

```
list.insert(index, element)
```

---

insert() Parameters

The insert() function takes two parameters:
index - position where element needs to be inserted
element - this is the element to be inserted in the list

Return Value from insert()

The insert() method only inserts the element to the list. It doesn't return any value.

Example 1: Inserting Element to List

```
vowel = ['a', 'e', 'i', 'u']

# inserting element to list at 4th position
vowel.insert(3, 'o')

print('Updated List: ', vowel)
```

When you run the program, the output will be:
Updated List: ['a', 'e', 'i', 'u', 'o']

Example 2: Inserting a Tuple (as an Element) to the List

```
mixed_list = [{1, 2}, [5, 6, 7]]

# number tuple
number_tuple = (3, 4)

# inserting tuple to the list
```

```
mixed_list.insert(1, number_tuple)

print('Updated List: ', mixed_list)
```

When you run the program, the output will be:
Updated List: [{1, 2}, (3, 4), [5, 6, 7]]
It is important to note that the index in Python starts from 0 not 1.

If you have to insert element in 4th place, you have to pass 3 as an index. Similarly, if you have to insert element in 2nd place, you have to use 1 as an index.

List pop()

The pop() method removes and returns the element at the given index (passed as an argument) from the list.

The syntax of pop() method is:
list.pop(index)

---

pop() parameter
The pop() method takes a single argument (index) and removes the element present at that index from the list.

If the index passed to the pop() method is not in the range, it throws **IndexError: pop index out of range** exception.
The parameter passed to the pop() method is optional. If no parameter is passed, the default index **-1** is passed as an argument which returns the last element.

Return Value from pop()
The pop() method returns the element present at the given index.
Also, the pop() method removes the element at the given index and updates the list.

Example 1: Print Element Present at the Given Index from the List

```
# programming language list
language = ['Python', 'Java', 'C++', 'French', 'C']

# Return value from pop()
# When 3 is passed
return_value = language.pop(3)
print('Return Value: ', return_value)

# Updated List
print('Updated List: ', language)
```

When you run the program, the output will be:
Return Value: French
Updated List: ['Python', 'Java', 'C++', 'C']
It is important to note that the index in Python starts from 0 not 1.
So, if you need to pop 4th element, you need to pass **3** to the pop() method.

Example 2: pop() when index is not passed and for negative indices

```
programming language list
language = ['Python', 'Java', 'C++', 'Ruby', 'C']

# When index is not passed
print('When index is not passed:')
print('Return Value: ', language.pop())
print('Updated List: ', language)

# When -1 is passed
# Pops Last Element
print('\nWhen -1 is passed:')
print('Return Value: ', language.pop(-1))
print('Updated List: ', language)

# When -3 is passed
# Pops Third Last Element
print('\nWhen -3 is passed:')
print('Return Value: ', language.pop(-3))
print('Updated List: ', language)
```

When you run the program, the output will be:

```
When index is not passed:
Return Value:  C
Updated List:  ['Python', 'Java', 'C++', 'Ruby']

When -1 is passed:
Return Value:  Ruby
Updated List:  ['Python', 'Java', 'C++']

When -3 is passed:
Return Value:  Python
Updated List:  ['Java', 'C++']
```

The pop() method returns and removes the element at the given index.

List count()

The count() method returns the number of occurrences of an element in a list.

In simple terms, count() method counts how many times an element has occurred in a list and returns it.

The syntax of count() method is:

```
list.count(element)
```

---

count() Parameters

The count() method takes a single argument:

element - element whose count is to be found.

Return value from count()

The count() method returns the number of occurrences of an element in a list.

Example 1: Count the occurrence of an element in the list

```
# vowels list
vowels = ['a', 'e', 'i', 'o', 'i', 'u']

# count element 'i'
count = vowels.count('i')

# print count
print('The count of i is:', count)
# count element 'p'
count = vowels.count('p')

# print count
print('The count of p is:', count)
```

When you run the program, the output will be:

```
The count of i is: 2
The count of p is: 0
```

Example 2: Count the occurrence of tuple and list inside the list

```
# random list
random = ['a', ('a', 'b'), ('a', 'b'), [3, 4]]

# count element ('a', 'b')
count = random.count(('a', 'b'))

# print count
print("The count of ('a', 'b') is:", count)

# count element [3, 4]
count = random.count([3, 4])

# print count
print("The count of [3, 4] is:", count)
```

When you run the program, the output will be:

```
The count of ('a', 'b') is: 2
The count of [3, 4] is: 1
```

List sort()

The sort() method sorts the elements of a given list.

The sort() method sorts the elements of a given list in a specific order - Ascending or Descending.

The syntax of sort() method is:

```
list.sort(key=..., reverse=...)
```

Alternatively, you can also use Python's in-built function sorted() for the same purpose.

```
sorted(list, key=..., reverse=...)
```

Note: Simplest difference between sort() and sorted() is: sort() doesn't return any value while, sorted() returns an iterable list.

sort() Parameters
By default, sort() doesn't require any extra parameters. However, it has two optional parameters:
reverse - If true, the sorted list is reversed (or sorted in Descending order)
key - function that serves as a key for the sort comparison

Return value from sort()
sort() method doesn't return any value. Rather, it changes the original list.
If you want the original list, use sorted().

Example 1: Sort a given list
```
# vowels list
vowels = ['e', 'a', 'u', 'o', 'i']

# sort the vowels
vowels.sort()

# print vowels
print('Sorted list:', vowels)
```

When you run the program, the output will be:
Sorted list: ['a', 'e', 'i', 'o', 'u']

How to sort in Descending order?
sort() method accepts a reverse parameter as an optional argument.
Setting reverse=True sorts the list in the descending order.
list.sort(reverse=True)
Alternately for sorted(), you can use the following code.
sorted(list, reverse=True)

Example 2: Sort the list in Descending order
```
# vowels list
vowels = ['e', 'a', 'u', 'o', 'i']

# sort the vowels
vowels.sort(reverse=True)

# print vowels
print('Sorted list (in Descending):', vowels)
```

When you run the program, the output will be:
Sorted list (in Descending): ['u', 'o', 'i', 'e', 'a']

How to sort using your own function with key parameter?
If you want your own implementation for sorting, sort() also accepts a keyfunction as an optional parameter.
Based on the results of the key function, you can sort the given list.
list.sort(key=len)
Alternatively for sorted

sorted(list, key=len)
Here, len is the Python's in-built function to count the length of an element.
The list is sorted based on the length of its each element, from lowest count to highest.

Example 3: Sort the list using key

```
# take second element for sort
def takeSecond(elem):
    return elem[1]

# random list
random = [(2, 2), (3, 4), (4, 1), (1, 3)]

# sort list with key
random.sort(key=takeSecond)
```

When you run the program, the output will be:
Sorted list: [(4, 1), (2, 2), (1, 3), (3, 4)]

List reverse()

The reverse() method reverses the elements of a given list.
The syntax of reverse() method is:

```
list.reverse()
```

---

reverse() parameter
The reverse() function doesn't take any argument.

Return Value from reverse()
The reverse() function doesn't return any value. It only reverses the elements and updates the list.

Example 1: Reverse a List

```
# Operating System List
os = ['Windows', 'macOS', 'Linux']
print('Original List:', os)

# List Reverse
os.reverse()

# updated list
print('Updated List:', os)
```

When you run the program, the output will be:
Original List: ['Windows', 'macOS', 'Linux']
Updated List: ['Linux', 'macOS', 'Windows']
There are other several ways to reverse a list.

Example 2: Reverse a List Using Slicing Operator

```
# Operating System List
os = ['Windows', 'macOS', 'Linux']
print('Original List:', os)
```

```
# Reversing a list
#Syntax: reversed_list = os[start:stop:step]
reversed_list = os[::-1]

# updated list
print('Updated List:', reversed_list)
```

When you run the program, the output will be:

```
Original List: ['Windows', 'macOS', 'Linux']
Updated List: ['Linux', 'macOS', 'Windows']
```

Example 3: Accessing Individual Elements in Reversed Order

If you need to access individual elements of a list in reverse order, it's better to use `reversed()` method.

```
# Operating System List
os = ['Windows', 'macOS', 'Linux']

# Printing Elements in Reversed Order
for o in reversed(os):
    print(o)
```

When you run the program, the output will be:

```
Linux
macOS
Windows
```

Python List Methods
append() - Add an element to the end of the list
extend() - Add all elements of a list to the another list
insert() - Insert an item at the defined index
remove() - Removes an item from the list
pop() - Removes and returns an element at the given index
clear() - Removes all items from the list
index() - Returns the index of the first matched item
count() - Returns the count of number of items passed as an argument
sort() - Sort items in a list in ascending order

reverse() - Reverse the order of items in the list
copy() - Returns a shallow copy of the list

Some examples of Python list methods:

```
my_list = [3, 8, 1, 6, 0, 8, 4]

# Output: 1
print(my_list.index(8))

# Output: 2
print(my_list.count(8))

my_list.sort()

# Output: [0, 1, 3, 4, 6, 8, 8]
print(my_list)

my_list.reverse()

# Output: [8, 8, 6, 4, 3, 1, 0]
print(my_list)
```

Output

```
1
2
[0, 1, 3, 4, 6, 8, 8]
[8, 8, 6, 4, 3, 1, 0]
```

12.2.6 List Comprehension: Elegant way to create new List

List comprehension is an elegant and concise way to create new list from an existing list in Python.
List comprehension consists of an expression followed by for statement inside square brackets.

Here is an example to make a list with each item being increasing power of 2.

```
pow2 = [2 ** x for x in range(10)]

# Output: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
print(pow2)
```

This code is equivalent to

```
pow2 = []
for x in range(10):
   pow2.append(2 ** x)
```

A list comprehension can optionally contain more for or if statements. An optional ifstatement can filter out items for the new list. Here are some examples.

```
>>> pow2 = [2 ** x for x in range(10) if x > 5]
>>> pow2
```

```
[64, 128, 256, 512]
>>> odd = [x for x in range(20) if x % 2 == 1]
>>> odd
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> [x+y for x in ['Python ','C '] for y in ['Language','Programming']]
['Python Language', 'Python Programming', 'C Language', 'C Programming']
```

12.2.7 Other List Operations in Python

List Membership Test
We can test if an item exists in a list or not, using the keyword in.

```
my_list = ['p','r','o','b','l','e','m']

# Output: True
print('p' in my_list)

# Output: False
print('a' in my_list)

# Output: True
print('c' not in my_list)
```

Iterating Through a List
Using a for loop we can iterate though each item in a list.

```
for fruit in ['apple','banana','mango']:
    print("I like",fruit)
I like apple
I like banana
I like mango
```

12.2.8 Built-in Functions with List

Built-in functions like all(), any(), enumerate(), len(), max(), min(), list(), sorted() etc. are commonly used with list to perform different tasks.

all()

The all() method returns True when all elements in the given iterable are true. If not, it returns False.

The syntax of all() method is:

```
all(iterable)
```

---

all() Parameters
The all() method takes a single parameter:
iterable - any iterable (list, tuple, dictionary, etc.) which contains the elements

---

Return Value from all()
The all() method returns:
True - If all elements in an iterable are true
False - If any element in an iterable is false

Truth table for all()	
When	**Return Value**
All values are true	True
All values are false	False
One value is true (others are false)	False
One value is false (others are true)	False
Empty Iterable	True

Example 1: How all() works for lists?

```
# all values true
l = [1, 3, 4, 5]
print(all(l))

# all values false
l = [0, False]
print(all(l))

# one false value
l = [1, 3, 4, 0]
print(all(l))

# one true value
l = [0, False, 5]
print(all(l))

# empty iterable
l = []
print(all(l))
```

When you run the program, the output will be:

```
True
False
False
False
True
```

Example 2: How all() works for strings?

```
s = "This is good"
print(all(s))

# 0 is False
# '0' is True
s = '000'
print(all(s))

s = ''
print(all(s))
```

When you run the program, the output will be:

```
True
True
True
```

Example 3: How all() works with Python dictionaries?

In case of dictionaries, if all keys (not values) are true or the dictionary is empty, all() returns True. Else, it returns false for all other cases..

```
s = {0: 'False', 1: 'False'}
print(all(s))

s = {1: 'True', 2: 'True'}
print(all(s))

s = {1: 'True', False: 0}
print(all(s))

s = {}
print(all(s))

# 0 is False
# '0' is True
s = {'0': 'True'}
print(all(s))
```

When you run the program, the output will be:

```
False
True
False
True
True
```

any()

The any() method returns True if any element of an iterable is true. If not, this method returns False.

The syntax of any() is:

```
any(iterable)
```

any() Parameters
They any() method takes an iterable (list, string, dictionary etc.) in Python.

Return Value from any()
The any method returns:
True if at least one element of an iterable is true
False if all elements are false or if an iterable is empty

Truth table for all()	
When	**Return Value**
All values are true	True
All values are false	False
One value is true (others are false)	True
One value is false (others are true)	True
Empty Iterable	False

Example 1: How any() works with Python List?

```
l = [1, 3, 4, 0]
print(any(l))

l = [0, False]
print(any(l))

l = [0, False, 5]
print(any(l))

l = []
print(any(l))
```

When you run the program, the output will be:
True
False
True
False
The any() method works in similar way for tuples and sets like lists.

Example 2: How any() works with Python Strings?

```
s = "This is good"
print(any(s))
```

```
# 0 is False
# '0' is True
s = '000'
print(any(s))

s = ''
print(any(s))
```

When you run the program, the output will be:

```
True
True
False
```

Example 3: How any() works with Python Dictionaries?

In case of dictionaries, if all keys (not values) are false, any() returns `False`. If at least one key is true, any() returns `True`.

```
d = {0: 'False'}
print(any(d))

d = {0: 'False', 1: 'True'}
print(any(d))

d = {0: 'False', False: 0}
print(any(d))

d = {}
print(any(d))

# 0 is False
# '0' is True
d = {'0': 'False'}
print(any(d))
```

When you run the program, the output will be:

```
False
True
False
False
True
```

enumerate()

The enumerate() method adds counter to an iterable and returns it (the enumerate object).

The syntax of enumerate() is:

```
enumerate(iterable, start=0)
```

---

enumerate() Parameters

The enumerate() method takes two parameters:

iterable - a sequence, an iterator, or objects that supports iteration

start (optional) - enumerate() starts counting from this number. If start is omitted, 0 is taken as start.

Return Value from enumerate()

The enumerate() method adds counter to an iterable and returns it. The returned object is a enumerate object.

You can convert enumerate objects to list and tuple using list() and tuple()method respectively.

Example 1: How enumerate() works in Python?

```
grocery = ['bread', 'milk', 'butter']
enumerateGrocery = enumerate(grocery)

print(type(enumerateGrocery))

# converting to list
print(list(enumerateGrocery))

# changing the default counter
enumerateGrocery = enumerate(grocery, 10)
print(list(enumerateGrocery))
```

When you run the program, the output will be:

```
<class 'enumerate'>
[(0, 'bread'), (1, 'milk'), (2, 'butter')]
[(10, 'bread'), (11, 'milk'), (12, 'butter')]
```

Example 2: Looping Over an Enumerate object

```
grocery = ['bread', 'milk', 'butter']

for item in enumerate(grocery):
  print(item)

print('\n')
for count, item in enumerate(grocery):
  print(count, item)

print('\n')
# changing default start value
for count, item in enumerate(grocery, 100):
  print(count, item)
```

When you run the program, the output will be:

```
(0, 'bread')
(1, 'milk')
(2, 'butter')

0 bread
1 milk
2 butter
```

100 bread
101 milk
102 butter

len()

The len() function returns the number of items (length) of an object.

The syntax of len() is:

len(s)

len() Parameters

s - a sequence (string, bytes, tuple, list, or range) or a collection (dictionary, set or frozen set)

Return Value from len()

The len() function returns the number of items of an object.
Failing to pass an argument or passing an invalid argument will raise a `TypeError` exception.

Example 1: How len() works with tuples, lists and range?

```
testList = []
print(testList, 'length is', len(testList))

testList = [1, 2, 3]
print(testList, 'length is', len(testList))

testTuple = (1, 2, 3)
print(testTuple, 'length is', len(testTuple))

testRange = range(1, 10)
print('Length of', testRange, 'is', len(testRange))
```

When you run the program, the output will be:

```
[] length is 0
[1, 2, 3] length is 3
(1, 2, 3) length is 3
Length of range(1, 10) is 9
```

Example 2: How len() works with strings and bytes?

```
testString = ''
print('Length of', testString, 'is', len(testString))

testString = 'Python'
print('Length of', testString, 'is', len(testString))

# byte object
testByte = b'Python'
print('Length of', testByte, 'is', len(testByte))
```

```
testList = [1, 2, 3]

# converting to bytes object
testByte = bytes(testList)
print('Length of', testByte, 'is', len(testByte))
```

When you run the program, the output will be:

```
Length of  is 0
Length of Python is 6
Length of b'Python' is 6
Length of b'\x01\x02\x03' is 3
```

list() Function

The list() constructor creates a list in Python.

The syntax of list() constructor is:

```
list([iterable])
```

---

list() Parameters

Python list() constructor takes a single argument:

iterable (Optional) - an object that could be a sequence (string, tuples) or collection (set, dictionary) or iterator object

---

Return value from list()

The list() constructor returns a mutable sequence list of elements.

If no parameters are passed, it creates an empty list

If iterable is passed as parameter, it creates a list of elements in the iterable

---

Example 1: Create list from sequence: string, tuple and list

```
# empty list
print(list())

# vowel string
vowelString = 'aeiou'
print(list(vowelString))

# vowel tuple
vowelTuple = ('a', 'e', 'i', 'o', 'u')
print(list(vowelTuple))

# vowel list
vowelList = ['a', 'e', 'i', 'o', 'u']
print(list(vowelList))
```

When you run the program, the output will be:

```
[]
['a', 'e', 'i', 'o', 'u']
['a', 'e', 'i', 'o', 'u']
```

['a', 'e', 'i', 'o', 'u']

Example 2: Create list from collection: set and dictionary

```
# vowel set
vowelSet = {'a', 'e', 'i', 'o', 'u'}
print(list(vowelSet))

# vowel dictionary
vowelDictionary = {'a': 1, 'e': 2, 'i': 3, 'o':4, 'u':5}
print(list(vowelDictionary))
```

When you run the program, the output will be:
['a', 'o', 'u', 'e', 'i']
['o', 'e', 'a', 'u', 'i']
Note: Keys in the dictionary are used as elements of the returned list. Also, the order in the list is not defined as a sequence.

Python max()

The max() method returns the largest element in an iterable or largest of two or more parameters.
Differnt syntaxes of max() are:
max(iterable, *iterables[,key, default])
max(arg1, arg2, *args[, key])
If you want to find the smallest element, use min() method.

---

max() Parameters
max() has two forms of arguments it can work with.

1. max(iterable, *iterables[, key, default])
iterable - sequence (tuple, string), collection (set, dictionary) or an iterator object whose largest element is to be found
***iterables (Optional)** - any number of iterables whose largest is to be found
key (Optional) - key function where the iterables are passed and comparison is performed based on its return value
default (Optional) - default value if the given iterable is empty

2. max(arg1, arg2, *args[, key])
arg1 - mandatory first object for comparison (could be number, string or other object)
arg2 - mandatory second object for comparison (could be number, string or other object)
***args (Optional)** - other objects for comparison
key - key function where each argument is passed, and comparison is performed based on its return value

Example 1: Find maximum among the given numbers

```
# using max(arg1, arg2, *args)
print('Maximum is:', max(1, 3, 2, 5, 4))

# using max(iterable)
num = [1, 3, 2, 8, 5, 10, 6]
```

```
print('Maximum is:', max(num))
```

When you run the program, the output will be:
Maximum is: 5
Maximum is: 10

Example 2: Find number whose sum of digits is largest using key function

```
def sumDigit(num):
    sum = 0
    while(num):
        sum += num % 10
        num = int(num / 10)
    return sum

# using max(arg1, arg2, *args, key)
print('Maximum is:', max(100, 321, 267, 59, 40, key=sumDigit))

# using max(iterable, key)
num = [15, 300, 2700, 821, 52, 10, 6]
print('Maximum is:', max(num, key=sumDigit))
```

When you run the program, the output will be:
Maximum is: 267
Maximum is: 821
Here, each element in the passed argument (list or argument) is passed to the same function sumDigit().
Based on the return value of the sumDigit(), i.e. sum of the digits, the largest is returned.

Example 3: Find list with maximum length using key function

```
num = [15, 300, 2700, 821]
num1 = [12, 2]
num2 = [34, 567, 78]

# using max(iterable, *iterables, key)
print('Maximum is:', max(num, num1, num2, key=len))
```

When you run the program, the output will be:
Maximum is: [15, 300, 2700, 821]

In this program, each iterable num, num1 and num2 is passed to the built-in method len().
Based on the result, i.e. length of each list, the list with maximum length is returned.

Python min()

The min() method returns the smallest element in an iterable or smallest of two or more parameters.

Differnt syntaxes of min() are:

```
min(iterable, *iterables[,key, default])
min(arg1, arg2, *args[, key])
```

If you want to find the largest element, use max() method.

---

min() Parameters
min() has two forms of arguments it can work with.

1. min(iterable, *iterables[, key, default])
iterable - sequence (tuple, string), collection (set, dictionary) or an iterator object whose smallest element is to be found
***iterables (Optional)** - any number of iterables whose smallest is to be found
key (Optional) - key function where the iterables are passed and comparison is performed based on its return value
default (Optional) - default value if the given iterable is empty

2. min(arg1, arg2, *args[, key])
arg1 - mandatory first object for comparison (could be number, string or other object)
arg2 - mandatory second object for comparison (could be number, string or other object)
***args (Optional)** - other objects for comparison
key - key function where each argument is passed, and comparison is performed based on its return value

Python sorted()

The sorted() method returns a sorted list from the given iterable.
The sorted() method sorts the elements of a given iterable in a specific order - Ascending or Descending.

The syntax of sorted() method is:
sorted(iterable[, key][, reverse])

---

sorted() Parameters
sorted() takes two three parameters:
iterable - sequence (string, tuple, list) or collection (set, dictionary, frozen set) or any iterator
reverse (Optional) - If true, the sorted list is reversed (or sorted in Descending order)
key (Optional) - function that serves as a key for the sort comparison

---

Return value from sorted()
sorted() method returns a sorted list from the given iterable.

Example 1: Sort a given sequence: string, list and tuple

```
# vowels list
pyList = ['e', 'a', 'u', 'o', 'i']
print(sorted(pyList))

# string
pyString = 'Python'
print(sorted(pyString))

# vowels tuple
pyTuple = ('e', 'a', 'u', 'o', 'i')
print(sorted(pyTuple))
```

When you run the program, the output will be:
['a', 'e', 'i', 'o', 'u']
['P', 'h', 'n', 'o', 't', 'y']
['a', 'e', 'i', 'o', 'u']
Note: A list also has sort() method which performs the same way as sorted(). Only difference being, sort() method doesn't return any value and changes the original list itself.

Example 2: Sort a given collection in descending order: set, dictionary and frozen set
sorted() method accepts a reverse parameter as an optional argument.
Setting reverse=True sorts the iterable in the descending order.

```
# set
pySet = {'e', 'a', 'u', 'o', 'i'}
print(sorted(pySet, reverse=True))

# dictionary
pyDict = {'e': 1, 'a': 2, 'u': 3, 'o': 4, 'i': 5}
print(sorted(pyDict, reverse=True))

# frozen set
pyFSet = frozenset(('e', 'a', 'u', 'o', 'i'))
print(sorted(pyFSet, reverse=True))
```

When you run the program, the output will be:
['u', 'o', 'i', 'e', 'a']
['u', 'o', 'i', 'e', 'a']
['u', 'o', 'i', 'e', 'a']

How to sort using your own function with key parameter?
If you want your own implementation for sorting, sorted() also accepts a keyfunction as an optional parameter.
Based on the results of the key function, you can sort the given iterable.
sorted(iterable, key=len)
Here, len is the Python's in-built function to count the length of an element.
The list is sorted based on the length of its each element, from lowest count to highest.

Example 3: Sort the list using sorted() having a key function

```
# take second element for sort
def takeSecond(elem):
    return elem[1]

# random list
random = [(2, 2), (3, 4), (4, 1), (1, 3)]

# sort list with key
sortedList = sorted(random, key=takeSecond)

# print list
print('Sorted list:', sortedList)
```

When you run the program, the output will be:

Sorted list: [(4, 1), (2, 2), (1, 3), (3, 4)]

Python sum()

The sum() function adds the items of an iterable and returns the sum.

The syntax of sum() is:

sum(iterable, start)

The sum() function adds start and items of the given iterable from left to right.

---

sum() Parameters

iterable - iterable (list, tuple, dict etc) whose item's sum is to be found. Normally, items of the iterable should be numbers.

start (optional) - this value is added to the sum of items of the iterable. The default value of start is 0 (if omitted)

Return Value from sum()

The sum() function returns the sum of start and items of the given iterable.

Example: How sum() function works in Python?

```
numbers = [2.5, 3, 4, -5]

# start parameter is not provided
numbersSum = sum(numbers)
print(numbersSum)

# start = 10
numbersSum = sum(numbers, 10)
print(numbersSum)
```

When you run the program, the output will be:

4.5

14.5

If you need to add floating point numbers with exact precision then, you should use math.fsum(iterable) instead

If you need to concatenate items of the given iterable (items must be string), then you can use

Function	Description
all()	Return True if all elements of the list are true (or if the list is empty).
any()	Return True if any element of the list is true. If the list is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of list as a tuple.
len()	Return the length (the number of items) in the list.
list()	Convert an iterable (tuple, string, set, dictionary) to a list.

max()	Return the largest item in the list.
min()	Return the smallest item in the list
sorted()	Return a new sorted list (does not sort the list itself).
sum()	Return the sum of all elements in the list.

12.3 Python Tuple

In Python programming, a tuple is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas in a list, elements can be changed.

Advantages of Tuple over List

Since, tuples are quite similar to lists, both of them are used in similar situations as well. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

We generally use tuple for heterogeneous (different) data types and list for homogeneous (similar) data types.

Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.

Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.

If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

12.3.1 Creating a Tuple

A tuple is created by placing all the items (elements) inside a parentheses (), separated by comma. 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.).

```
# empty tuple
# Output: ()
my_tuple = ()
print(my_tuple)

# tuple having integers
# Output: (1, 2, 3)
my_tuple = (1, 2, 3)
print(my_tuple)
```

```
# tuple with mixed datatypes
# Output: (1, "Hello", 3.4)
my_tuple = (1, "Hello", 3.4)
print(my_tuple)

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

# tuple can be created without parentheses
# also called tuple packing
# Output: 3, 4.6, "dog"

my_tuple = 3, 4.6, "dog"
print(my_tuple)

# tuple unpacking is also possible
# Output:
# 3
# 4.6
# dog
a, b, c = my_tuple
print(a)
print(b)
print(c)
```

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.

```
# only parentheses is not enough
# Output: <class 'str'>
my_tuple = ("hello")
print(type(my_tuple))

# need a comma at the end
# Output: <class 'tuple'>
my_tuple = ("hello",)
print(type(my_tuple))

# parentheses is optional
# Output: <class 'tuple'>
my_tuple = "hello",
print(type(my_tuple))
```

Output

```
<class 'str'>
<class 'tuple'>
<class 'tuple'>
```

12.3.2 Accessing Elements in a Tuple

There are various ways in which we can access the elements of a tuple.

1. Indexing

We can use the index operator [] to access an item in a tuple where the index starts from 0. So, a tuple having 6 elements will have index from 0 to 5. Trying to access an element other that (6, 7,...) will raise an IndexError.

The index must be an integer, so we cannot use float or other types. This will result into TypeError.

Likewise, nested tuple are accessed using nested indexing, as shown in the example below.

```
my_tuple = ('p','e','r','m','i','t')

# Output: 'p'
print(my_tuple[0])

# Output: 't'
print(my_tuple[5])

# index must be in range
# If you uncomment line 14,
# you will get an error.
# IndexError: list index out of range

#print(my_tuple[6])

# index must be an integer
# If you uncomment line 21,
# you will get an error.
# TypeError: list indices must be integers, not float

#my_tuple[2.0]

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

# nested index
# Output: 's'
print(n_tuple[0][3])

# nested index
# Output: 4
print(n_tuple[1][1])
```

Output

```
p
t
```

s
4

2. 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')

# Output: 't'
print(my_tuple[-1])

# Output: 'p'
print(my_tuple[-6])
```

Output
t
p

3. 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')

# elements 2nd to 4th
# Output: ('r', 'o', 'g')
print(my_tuple[1:4])

# elements beginning to 2nd
# Output: ('p', 'r')
print(my_tuple[:-7])

# elements 8th to end
# Output: ('i', 'z')
print(my_tuple[7:])

# elements beginning to end
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple[:])
```

Output

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the tuple.

12.3.3 Changing 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).

```
my_tuple = (4, 2, 3, [6, 5])

# we cannot change an element
# If you uncomment line 8
# you will get an error:
# TypeError: 'tuple' object does not support item assignment

#my_tuple[1] = 9

# but item of mutable element can be changed
# Output: (4, 2, 3, [9, 5])
my_tuple[3][0] = 9
print(my_tuple)

# tuples can be reassigned
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
my_tuple = ('p','r','o','g','r','a','m','i','z')
print(my_tuple)
```

We can use + operator to combine two tuples. This is also called **concatenation**.
We can also **repeat** the elements in a tuple for a given number of times using the * operator.
Both + and * operations result into a new tuple.

```
# Concatenation
# Output: (1, 2, 3, 4, 5, 6)
print((1, 2, 3) + (4, 5, 6))

# Repeat
# Output: ('Repeat', 'Repeat', 'Repeat')
print(("Repeat",) * 3)
```

12.3.4 Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. That also means we cannot delete or remove items from a tuple.
But deleting a tuple entirely is possible using the keyword del.

```
my_tuple = ('p','r','o','g','r','a','m','i','z')

# can't delete items
# if you uncomment line 8,
```

```
# you will get an error:
# TypeError: 'tuple' object doesn't support item deletion

#del my_tuple[3]

# can delete entire tuple
# NameError: name 'my_tuple' is not defined
del my_tuple
my_tuple
```

12.3.5 Python Tuple Methods

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

Python Tuple Method	
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

Some examples of Python tuple methods:

```
my_tuple = ('a','p','p','l','e',)

# Count
# Output: 2
print(my_tuple.count('p'))

# Index
# Output: 3
print(my_tuple.index('l'))
```

12.3.6 Other Tuple Operations

1. Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword in.

```
my_tuple = ('a','p','p','l','e',)

# In operation
# Output: True
print('a' in my_tuple)

# Output: False
print('b' in my_tuple)

# Not in operation
# Output: True
print('g' not in my_tuple)
```

2. Iterating Through a Tuple

Using a for loop we can iterate though each item in a tuple.

```
# Output:
# Hello John
# Hello Kate
for name in ('John','Kate'):
    print("Hello",name)
```

3. Built-in Functions with Tuple

Built-in functions like all(), any(), enumerate(), len(), max(), min(), sorted(), tuple() etc. are commonly used with tuple to perform different tasks.

Built-in Functions with Tuple	
Function	**Description**
all()	Return True if all elements of the tuple are true (or if the tuple is empty).
any()	Return True if any element of the tuple is true. If the tuple is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of tuple as pairs.
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.
tuple()	Convert an iterable (list, string, set, dictionary) to a tuple.

Python any()

The syntax of any() is:

```
any(iterable)
```

any() Parameters

They any() method takes an iterable (list, string, dictionary etc.) in Python.

The any method returns:

True if at least one element of an iterable is true

False if all elements are false or if an iterable is empty

Truth table for any()	
When	**Return Value**
All values are true	True
All values are false	False
One value is true (others are false)	True
One value is false (others are true)	True
Empty Iterable	False

Python all()

The syntax of any() is:
all(iterable)

all() Parameters
The all() method takes a single parameter:
iterable - any iterable (list, tuple, dictionary, etc.) which contains the elements

The all() method returns:
True - If all elements in an iterable are true
False - If any element in an iterable is false

Truth table for all()	
When	**Return Value**
All values are true	True
All values are false	False
One value is true (others are false)	False
One value is false (others are true)	False
Empty Iterable	True

Example

Any	All
l = [1, 3, 4, 0] print(any(l)) #output - True l = [0, False] print(any(l)) #output - False l = [0, False, 5] print(any(l)) #output - True l = [] print(any(l)) #output - False	l = [1, 3, 4, 0] print(all(l)) #output - False l = [0, False] print(all(l)) #output - False l = [0, False, 5] print(all(l)) #output - False l = [] print(all(l)) #output – True l = [2,5,8,9] print(all(l)) #output - True

12.4 Python Strings

A string is a sequence of characters.

A character is simply a symbol. For example, the English language has 26 characters. Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0's and 1's.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encoding used.

In Python, string is a sequence of Unicode character. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

12.4.1 How to create a string in Python?

Strings can be created by enclosing characters inside a single quote or double quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

```
# all of the following are equivalent
my_string = 'Hello'
print(my_string)

my_string = "Hello"
print(my_string)
```

```
my_string = '''Hello'''
print(my_string)

# triple quotes string can extend multiple lines
my_string = """Hello, welcome to
        the world of Python"""
print(my_string)
```

When you run the program, the output will be:

```
Hello
Hello
Hello
Hello, welcome to
        the world of Python
```

12.4.2 How to access characters in a string?

We can access individual characters using indexing and a range of characters using slicing. Index starts from 0. Trying to access a character out of index range will raise an `IndexError`. The index must be an integer.

We can't use float or other types, this will result into `TypeError`.
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. We can access a range of items in a string by using the slicing operator (colon).

```
str = 'programiz'
print('str = ', str)

#first character
print('str[0] = ', str[0])

#last character
print('str[-1] = ', str[-1])

#slicing 2nd to 5th character
print('str[1:5] = ', str[1:5])

#slicing 6th to 2nd last character
print('str[5:-2] = ', str[5:-2])
```

If we try to access index out of the range or use decimal number, we will get errors.

```
# index must be in range
>>> my_string[15]
...
IndexError: string index out of range

# index must be an integer
>>> my_string[1.5]
...
```

TypeError: string indices must be integers
Slicing can be best visualized by considering the index to be between the elements as shown below.
If we want to access a range, we need the index that will slice the portion from the string.

12.4.3 How to change or delete a string?

Strings are immutable. This means that elements of a string cannot be changed once it has been assigned. We can simply reassign different strings to the same name.

```
>>> my_string = 'programiz'
>>> my_string[5] = 'a'
...
TypeError: 'str' object does not support item assignment
>>> my_string = 'Python'
>>> my_string
'Python'
```

We cannot delete or remove characters from a string. But deleting the string entirely is possible using the keyword `del`.

```
>>> del my_string[1]
...
TypeError: 'str' object doesn't support item deletion
>>> del my_string
>>> my_string
...
NameError: name 'my_string' is not defined
```

12.4.4 Python String Operations

There are many operations that can be performed with string which makes it one of the most used datatypes in Python.

1. Concatenation of Two or More Strings
Joining of two or more strings into a single one is called concatenation.
The **+** operator does this in Python. Simply writing two string literals together also concatenates them.
The ***** operator can be used to repeat the string for a given number of times.

```
str1 = 'Hello'
str2 ='World!'

# using +
print('str1 + str2 = ', str1 + str2)

# using *
print('str1 * 3 =', str1 * 3)
```

Writing two string literals together also concatenates them like **+** operator.
If we want to concatenate strings in different lines, we can use parentheses.

```
>>> # two string literals together
```

```
>>> 'Hello "World!'
'Hello World!'

>>> # using parentheses
>>> s = ('Hello '
...      'World')
>>> s
'Hello World'
```

2. Iterating Through String

Using for loop we can iterate through a string. Here is an example to count the number of 'l' in a string.

```
count = 0
for letter in 'Hello World':
    if(letter == 'l'):
        count += 1
print(count,'letters found')
```

3. String Membership Test

We can test if a sub string exists within a string or not, using the keyword in.

```
>>> 'a' in 'program'
True
>>> 'at' not in 'battle'
False
```

12.4.5 Built-in functions to Work with Python

Various built-in functions that work with sequence, works with string as well.
Some of the commonly used ones are enumerate() and len(). The enumerate()function returns an enumerate object.

It contains the index and value of all the items in the string as pairs. This can be useful for iteration.

Similarly, len() returns the length (number of characters) of the string.

```
str = 'cold'

# enumerate()
list_enumerate = list(enumerate(str))
print('list(enumerate(str) = ', list_enumerate)

#character count
print('len(str) = ', len(str))
```

Python String Formatting

Escape Sequence

If we want to print a text like -He said, "What's there?"- we can neither use single quote or double quotes. This will result into SyntaxError as the text itself contains both single and double quotes.

```
>>> print("He said, "What's there?"")
...
SyntaxError: invalid syntax
>>> print('He said, "What's there?"')
...
SyntaxError: invalid syntax
```

One way to get around this problem is to use triple quotes. Alternatively, we can use escape sequences.

An escape sequence starts with a backslash and is interpreted differently. If we use single quote to represent a string, all the single quotes inside the string must be escaped. Similar is the case with double quotes. Here is how it can be done to represent the above text.

```
# using triple quotes
print('''He said, "What's there?"''')

# escaping single quotes
print('He said, "What\'s there?"')

# escaping double quotes
print("He said, \"What's there?\"")
```

Here is a list of all the escape sequence supported by Python.

Escape Sequence in Python	
Escape Sequence	**Description**
\newline	Backslash and newline ignored
\\	Backslash
\'	Single quote
\"	Double quote
\b	ASCII Backspace
\n	ASCII Linefeed
\t	ASCII Horizontal Tab
\v	ASCII Vertical Tab

12.5 Python Sets

A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed).
However, the set itself is mutable. We can add or remove items from it.
Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

12.5.1 How to create a set?

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, or dictionary, as its element.

```
# set of integers
my_set = {1, 2, 3}
print(my_set)

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

Exp-2

```
# set do not have duplicates
# Output: {1, 2, 3, 4}
my_set = {1,2,3,4,3,2}
print(my_set)

# set cannot have mutable items
# here [3, 4] is a mutable list
# this will cause an error.
# TypeError: unhashable type: 'list'

#my_set = {1, 2, [3, 4]}

# we can make set from a list
# Output: {1, 2, 3}
my_set = set([1,2,3,2])
print(my_set)
```

12.5.2 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.

```
# initialize a with {}
a = {}

# check data type of a
```

```
# Output: <class 'dict'>
print(type(a))

# initialize a with set()
a = set()

# check data type of a
# Output: <class 'set'>
print(type(a))
```

12.5.3 How to change 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 element using the add() method and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

```
# initialize my_set
my_set = {1,3}
print(my_set)

# if you uncomment line 9,
# you will get an error
# TypeError: 'set' object does not support indexing

#my_set[0]

# add an element
# Output: {1, 2, 3}
my_set.add(2)
print(my_set)

# add multiple elements
# Output: {1, 2, 3, 4}
my_set.update([2,3,4])
print(my_set)

# add list and set
# Output: {1, 2, 3, 4, 5, 6, 8}
my_set.update([4,5], {1,6,8})
print(my_set)
```

When you run the program, the output will be:

```
{1, 3}
{1, 2, 3}
{1, 2, 3, 4}
{1, 2, 3, 4, 5, 6, 8}
```

12.5.4 How to remove elements from a set?

A particular item can be removed from set using methods, 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.
The following example will illustrate this.

```
# initialize my_set
my_set = {1, 3, 4, 5, 6}
print(my_set)

# discard an element
# Output: {1, 3, 5, 6}
my_set.discard(4)
print(my_set)

# remove an element
# Output: {1, 3, 5}
my_set.remove(6)
print(my_set)

# discard an element
# not present in my_set
# Output: {1, 3, 5}
my_set.discard(2)
print(my_set)

# remove an element
# not present in my_set
# If you uncomment line 27,
# you will get an error.
# Output: KeyError: 2

#my_set.remove(2)
```

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().

```
# initialize my_set
# Output: set of unique elements
my_set = set("HelloWorld")
print(my_set)

# pop an element
# Output: random element
print(my_set.pop())
```

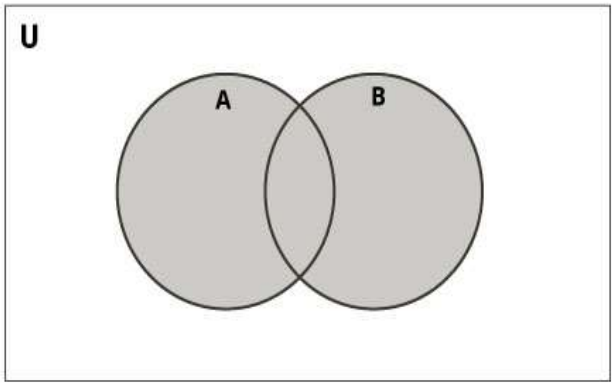
```
# pop another element
# Output: random element
my_set.pop()
print(my_set)

# clear my_set
#Output: set()
my_set.clear()
print(my_set)
```

12.5.5 Python Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.
Let us consider the following two sets for the following operations.

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

Set Union



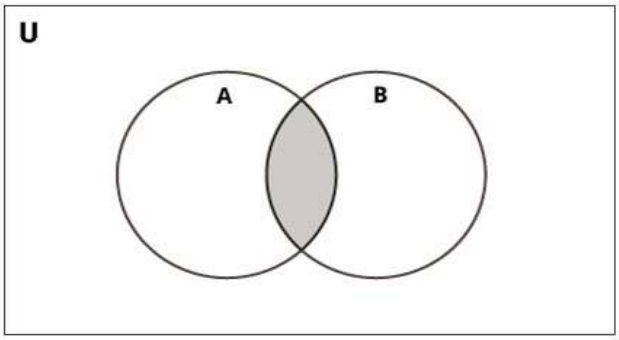
Union of A and B is a set of all elements from both sets.
Union is performed using `|` operator. Same can be accomplished using the method `union()`

```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use | operator
# Output: {1, 2, 3, 4, 5, 6, 7, 8}
print(A | B)
```

Try the following examples on Python shell.

```
# use union function
>>> A.union(B)
{1, 2, 3, 4, 5, 6, 7, 8}

# use union function on B
>>> B.union(A)
{1, 2, 3, 4, 5, 6, 7, 8}
```

Set Intersection



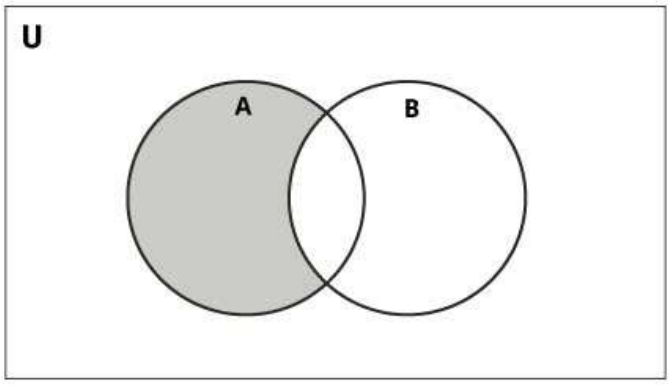
Intersection of *A* and *B* is a set of elements that are common in both sets.
Intersection is performed using & operator. Same can be accomplished using the method intersection().

```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use & operator
# Output: {4, 5}
print(A & B)
```

Try the following examples on Python shell.

```
# use intersection function on A
>>> A.intersection(B)
{4, 5}

# use intersection function on B
>>> 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().

```
# initialize A and B
A = {1, 2, 3, 4, 5}
```

```
B = {4, 5, 6, 7, 8}

# use - operator on A
# Output: {1, 2, 3}
print(A - B)
```

Try the following examples on Python shell.

```
# use difference function on A
>>> A.difference(B)
{1, 2, 3}

# use - operator on B
>>> B - A
{8, 6, 7}

# use difference function on B
>>> B.difference(A)
{8, 6, 7}
```

Set difference_update()

If *A* and *B* are two sets. The set difference of *A* and *B* is a set of elements that exists only in set *A* but not in *B*.
To learn more, visit Python set difference.

The syntax of difference_update() is:

```
A.difference_update(B)
```

Here, *A* and *B* are two sets. The difference_update() updates set *A* with the set difference of A-B.

Return Value from difference_update()
The difference_update() returns None indicating the object (set) is mutated.
Suppose,

```
result = A.difference_update(B)
```

When you run the code,
result will be None
A will be equal to *A-B*
B will be unchanged

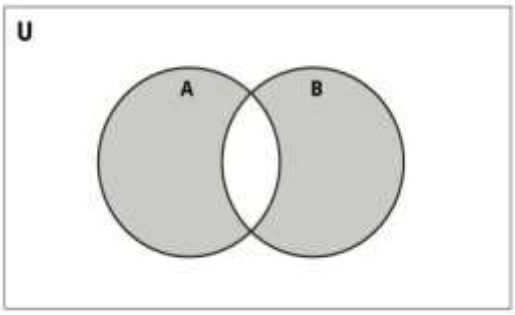
```
A = {'a', 'c', 'g', 'd'}
B = {'c', 'f', 'g'}

result = A.difference_update(B)

print('A = ', A)
print('B = ', B)
print('result = ', result)
```

Output

```
A =  {'a', 'd'}
```

B = {'c', 'g', 'f'}
result = None

Set Symmetric Difference



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

```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use ^ operator
# Output: {1, 2, 3, 6, 7, 8}
print(A ^ B)
```

Try the following examples on Python shell.

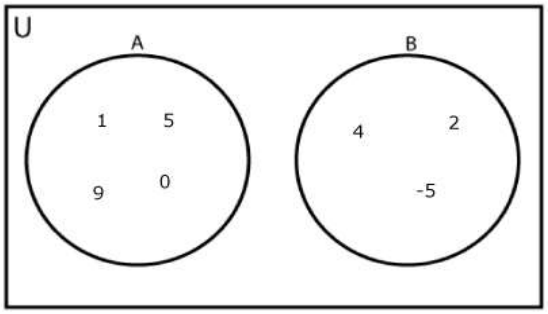
```
# use symmetric_difference function on A
>>> A.symmetric_difference(B)
{1, 2, 3, 6, 7, 8}

# use symmetric_difference function on B
>>> B.symmetric_difference(A)
{1, 2, 3, 6, 7, 8}
```

Set isdisjoint()

The isdisjoint() method returns True if two sets are disjoint sets. If not, it returns False.
Two sets are said to be disjoint sets if they have no common elements. For example:

```
A = {1, 5, 9, 0}
B = {2, 4, -5}
```

Here, sets *A* and *B* are disjoint sets.



The syntax of isdisjoint() is:
set_a.isdisjoint(set_b)

---

isdisjoint() Parameters
The isdisjoint() method takes a single argument (a set).
You can also pass an iterable (list, tuple, dictionary and string) to disjoint(). The isdisjoint() method will automatically convert iterables to set and checks whether the sets are disjoint or not.

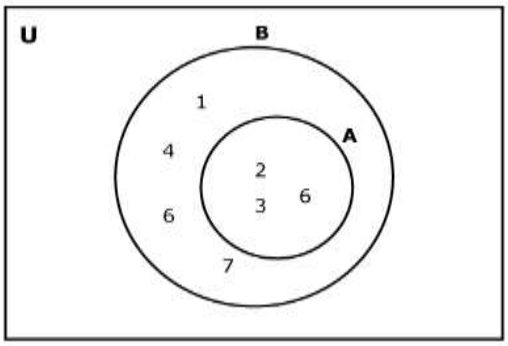
Return Value from isdisjoint()
The isdisjoint() method returns
True if two sets are disjoint sets (if *set_a* and *set_b* are disjoint sets in above syntax)
False if two sets are not disjoint sets

Example 1: How isdisjoint() works?
A = {1, 2, 3, 4}
B = {5, 6, 7}
C = {4, 5, 6}

print('Are A and B disjoint?', A.isdisjoint(B))
print('Are A and C disjoint?', A.isdisjoint(C))
Are A and B disjoint? True
Are A and C disjoint? False

Set issubset()

Set *A* is said to be the subset of set *B* if all elements of *A* are in *B* .



Here, set *A* is a subset of *B*.

The syntax of issubset() is:
A.issubset(B)
The above code checks if *A* is a subset of *B*.

Return Value from issubset()
The issubset() returns
True if A is a subset of B
False if A is not a subset of B

Example: How issubset() works?
A = {1, 2, 3}
B = {1, 2, 3, 4, 5}

C = {1, 2, 4, 5}

# Returns True
print(A.issubset(B))

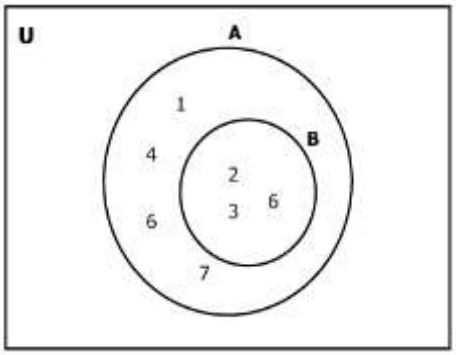
# Returns False
# B is not subset of A
print(B.issubset(A))

# Returns False
print(A.issubset(C))

# Returns True
print(C.issubset(B))

Set issuperset()

Set *A* is said to be the superset of set *B* if all elements of *B* are in *A*.



Here, set *A* is a superset of set *B* and *B* is a subset of set *A*.

The syntax of issuperset() is:

A.issuperset(B)

The following code checks if *A* is a superset of *B*.

Return Value from issuperset()

The issuperset() returns

True if A is a superset of B

False if A is not a superset of B

Example: How issuperset() works?

A = {1, 2, 3, 4, 5}
B = {1, 2, 3}
C = {1, 2, 3}

# Returns True
print(A.issuperset(B))

# Returns False
print(B.issuperset(A))

Returns True

```
print(C.issuperset(B))
```

12.5.6 Different Python Set Methods

There are many set methods, some of which we have already used above. Here is a list of all the methods that are available with set objects.

Python Set Methods	
Method	**Description**
add()	Add an element to a set
clear()	Remove all elements form a set
copy()	Return a shallow copy of a set
difference()	Return the difference of two or more sets as a new set
difference_update()	Remove all elements of another set from this set
discard()	Remove an element from set if it is a member. (Do nothing if the element is not in set)
intersection()	Return the intersection of two sets as a new set
intersection_update()	Update the set with the intersection of itself and another
isdisjoint()	Return True if two sets have a null intersection
issubset()	Return True if another set contains this set
issuperset()	Return True if this set contains another set
pop()	Remove and return an arbitary set element. Raise KeyError if the set is empty
remove()	Remove an element from a set. If the element is not a member, raise a KeyError
symmetric_difference()	Return the symmetric difference of two sets as a new set
symmetric_difference_update()	Update a set with the symmetric difference of itself and another
union()	Return the union of sets in a new set
update()	Update a set with the union of itself and others

12.5.7 Other Set Operations

Set Membership Test
We can test if an item exists in a set or not, using the keyword in.

```
# initialize my_set
my_set = set("apple")

# check if 'a' is present
# Output: True
print('a' in my_set)

# check if 'p' is present
# Output: False
print('p' not in my_set)
```

Iterating Through a Set
Using a for loop, we can iterate though each item in a set.

```
>>> for letter in set("apple"):
...     print(letter)
...
a
p
e
l
```

12.5.8 Built-in Functions with Set

Built-in functions like all(), any(), enumerate(), len(), max(), min(), sorted(), sum() etc. are commonly used with set to perform different tasks.

Built-in Functions with Set	
Function	Description
all()	Return True if all elements of the set are true (or if the set is empty).
any()	Return True if any element of the set is true. If the set is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of set as a pair.
len()	Return the length (the number of items) in the set.
max()	Return the largest item in the set.
min()	Return the smallest item in the set.
sorted()	Return a new sorted list from elements in the set(does not sort the set itself).

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

12.5.9 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.
Frozensets can be created using the function frozenset().

This datatype supports methods like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric_difference() and union(). Being immutable it does not have method that add or remove elements.

```
# initialize A and B
A = frozenset([1, 2, 3, 4])
B = frozenset([3, 4, 5, 6])
```
Try these examples on Python shell.
```
>>> A.isdisjoint(B)
False
>>> 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'
```

12.6 Python Dictionary

Python dictionary is an **unordered collection** of items. While other compound data types have only value as an element, a dictionary has a **key: value** pair.
Dictionaries are optimized to retrieve values when the key is known.

12.6.1 How to create a dictionary?

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 data type** and **can repeat**, **keys must be of immutable** type (string, number or tuple with immutable elements) and **must be unique**.

```
# 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')])
```

As you can see above, we can also create a dictionary using the built-in function dict().

12.6.2 How to access elements from 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':'Jack', 'age': 26}

# Output: Jack
print(my_dict['name'])

# Output: 26
print(my_dict.get('age'))

# Trying to access keys which doesn't exist throws error
# my_dict.get('address')
# my_dict['address']
```

When you run the program, the output will be:

```
Jack
26
```

How to change or add 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 = {'name':'Jack', 'age': 26}

# update value
my_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}
print(my_dict)
```

```
# add item
my_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}
print(my_dict)
```

When you run the program, the output will be:

```
{'name': 'Jack', 'age': 27}
{'name': 'Jack', 'age': 27, 'address': 'Downtown'}
```

12.6.3 How to delete or remove 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. 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.

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

# remove a particular item
# Output: 16
print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}
print(squares)

# remove an arbitrary item
# Output: (1, 1)
print(squares.popitem())

# Output: {2: 4, 3: 9, 5: 25}
print(squares)

# delete a particular item
del squares[5]

# Output: {2: 4, 3: 9}
print(squares)

# remove all items
squares.clear()

# Output: {}
print(squares)

# delete the dictionary itself
del squares
```

```
# Throws Error
# print(squares)
```
When you run the program, the output will be:
```
16
{1: 1, 2: 4, 3: 9, 5: 25}
(1, 1)
{2: 4, 3: 9, 5: 25}
{2: 4, 3: 9}
{}
```

12.6.4 Python Dictionary Methods

Methods that are available with dictionary are explained below.

Dictionary get()

The get() method returns the value for the specified key if key is in dictionary.
The syntax of get() is:
dict.get(key[, value])

---

get() Parameters
The get() method takes maximum of two parameters:
key - key to be searched in the dictionary
value (optional) - Value to be returned if the *key* is not found. The default value is None.

Return Value from get()
The get() method returns:
the value for the specified *key* if *key* is in dictionary.
None if the *key* is not found and *value* is not specified.
value if the *key* is not found and *value* is specified.

Example 1: How get() works for dictionaries?

```
person = {'name': 'Phill', 'age': 22}

print('Name: ', person.get('name'))
print('Age: ', person.get('age'))

# value is not provided
print('Salary: ', person.get('salary'))

# value is provided
print('Salary: ', person.get('salary', 0.0))
```

When you run the program, the output will be:
```
Name:  Phill
Age:  22
Salary:  None
Salary:  0.0
```

Python get() method Vs dict[key] to Access Elements
The get() method returns a default value if the key is missing.
However, if the *key* is not found when you use dict[key], KeyError exception is raised.
print('Salary: ', person.get('salary'))
print(person['salary'])

Dictionary keys()

The keys() method returns a view object that displays a list of all the keys in the dictionary
The syntax of keys() is:
dict.keys()

---

keys() Parameters
The keys() doesn't take any parameters.

Return Value from keys()
The keys() returns a view object that displays a list of all the keys.
When the dictionary is changed, the view object also reflect these changes.

Example 1: How keys() works?

```
person = {'name': 'Phill', 'age': 22, 'salary': 3500.0}
print(person.keys())

empty_dict = {}
print(empty_dict.keys())
```

When you run the program, the output will be:

```
dict_keys(['name', 'salary', 'age'])
dict_keys([])
```

Example 2: How keys() works when dictionary is updated?

```
person = {'name': 'Phill', 'age': 22, }

print('Before dictionary is updated')
keys = person.keys()
print(keys)

# adding an element to the dictionary
person.update({'salary': 3500.0})
print('\nAfter dictionary is updated')
print(keys)
```

When you run the program, the output will be:

```
Before dictionary is updated
dict_keys(['name', 'age'])

After dictionary is updated
dict_keys(['name', 'age', 'salary'])
```

Here, when the dictionary is updated, keys is also automatically updated to reflect changes.

Dictionary values()

The values() method returns a view object that displays a list of all the values in the dictionary.
The syntax of values() is:
dictionary.values()

---

values() Parameters
The values() method doesn't take any parameters.

Return value from values()
The values() method returns a view object that displays a list of all values in a given dictionary.

Example 1: Get all values from the dictionary

```
# random sales dictionary
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

print(sales.values())
```

When you run the program, the output will be:
dict_values([2, 4, 3])

Example 2: How values() works when dictionary is modified?

```
# random sales dictionary
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

values = sales.values()
print('Original items:', values)

# delete an item from dictionary
del[sales['apple']]
print('Updated items:', values)
```

When you run the program, the output will be:
Original items: dict_values([2, 4, 3])
Updated items: dict_values([4, 3])

The view object `values` doesn't itself return a list of `sales` item values but it returns a view of all values of the dictionary.
If the list is updated at any time, the changes are reflected on to the view object itself, as shown in the above program.

Dictionary items()

The items() method returns a view object that displays a list of dictionary's (key, value) tuple pairs.

The syntax of items() method is:
dictionary.items()

---

items() Parameters

The items() method doesn't take any parameters.

Return value from items()
The items() method returns a view object that displays a list of a given dictionary's (key, value) tuple pair.

Example 1: Get all items of a dictionary with items()

```
# random sales dictionary
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

print(sales.items())
```

When you run the program, the output will be:

```
dict_items([('apple', 2), ('orange', 3), ('grapes', 4)])
```

Example 2: How items() works when a dictionary is modified?

```
# random sales dictionary
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

items = sales.items()
print('Original items:', items)

# delete an item from dictionary
del[sales['apple']]
print('Updated items:', items)
```

When you run the program, the output will be:

```
Original items: dict_items([('apple', 2), ('orange', 3), ('grapes', 4)])
Updated items: dict_items([('orange', 3), ('grapes', 4)])
```

The view object items doesn't itself return a list of sales items but it returns a view of sales's (key, value) pair.
If the list is updated at any time, the changes are reflected on to the view object itself, as shown in the above program.

Dictionary update()

The update() method updates the dictionary with the elements from the another dictionary object or from an iterable of key/value pairs.
The update() method adds element(s) to the dictionary if the key is not in the dictionary. If the key is in the dictionary, it updates the key with the new value.

The syntax of update() is:

```
dict.update([other])
```

---

update() Parameters
The update() method takes either a dictionary or an iterable object of key/value pairs (generally tuples).
If update() is called without passing parameters, the dictionary remains unchanged.

Return Value from update()
They update() method updates the dictionary with elements from a dictionary object or an iterable object of key/value pairs.
It doesn't return any value (returns None).

Example 1: How update() works in Python?

```
d = {1: "one", 2: "three"}
d1 = {2: "two"}

# updates the value of key 2
d.update(d1)
print(d)

d1 = {3: "three"}

# adds element with key 3
d.update(d1)
print(d)
```

When you run the program, the output will be:

```
{1: 'one', 2: 'two'}
{1: 'one', 2: 'two', 3: 'three'}
```

Example 2: How update() Works With an Iterable?

```
d = {'x': 2}

d.update(y = 3, z = 0)
print(d)
```

When you run the program, the output will be:

```
{'x': 2, 'y': 3, 'z': 0}
```

Dictionary setdefault()

The setdefault() method returns the value of a key (if the key is in dictionary). If not, it inserts key with a value to the dictionary.
The syntax of setdefault() is:
dict.setdefault(key[, default_value])

---

setdefault() Parameters
The setdefault() takes maximum of two parameters:
key - key to be searched in the dictionary
default_value (optional) - *key* with a value *default_value* is inserted to the dictionary if key is not in the dictionary.
If not provided, the *default_value* will be None.

Return Value from setdefault()
The setdefault() returns:
value of the *key* if it is in the dictionary

None if key is not in the dictionary and default_value is not specified
default_value if *key* is not in the dictionary and *default_value* is specified

---

Example 1: How setdefault() works when key is in the dictionary?

```
person = {'name': 'Phill', 'age': 22}
age = person.setdefault('age')
print('person = ',person)
print('Age = ',age)
```

When you run the program, the output will be:

```
person =  {'name': 'Phill', 'age': 22}
Age =  22
```

Example 2: How setdefault() works when key is not in the dictionary?

```
person = {'name': 'Phill'}

# key is not in the dictionary
salary = person.setdefault('salary')
print('person = ',person)
print('salary = ',salary)

# key is not in the dictionary
# default_value is provided
age = person.setdefault('age', 22)
print('person = ',person)
print('age = ',age)
```

When you run the program, the output will be:

```
person =  {'name': 'Phill', 'salary': None}
salary =  None
person =  {'name': 'Phill', 'age': 22, 'salary': None}
age =  22
```

Dictionary pop()

The pop() method removes and returns an element from a dictionary having the given key.

The syntax of pop() method is

```
dictionary.pop(key[, default])
```

---

pop() Parameters

The pop() method takes two parameters:

key - key which is to be searched for removal
default - value which is to be returned when the key is not in the dictionary

---

Return value from pop()

The pop() method returns:
If key is found - removed/popped element from the dictionary
If key is not found - value specified as the second argument (default)
If key is not found and default argument is not specified - KeyErrorexception is raised

Example 1: Pop an element from the dictionary

```
# random sales dictionary
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

element = sales.pop('apple')
print('The popped element is:', element)
print('The dictionary is:', sales)
```

When you run the program, the output will be:

```
The popped element is: 2
The dictionary is: {'orange': 3, 'grapes': 4}
```

Example 2: Pop an element not present from the dictionary

```
# random sales dictionary
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

element = sales.pop('guava')
When you run the program, the output will be:
KeyError: 'guava'
```

Example 3: Pop an element not present from the dictionary, provided a default value

```
# random sales dictionary
sales = { 'apple': 2, 'orange': 3, 'grapes': 4 }

element = sales.pop('guava', 'banana')
print('The popped element is:', element)
print('The dictionary is:', sales)
```

When you run the program, the output will be:

```
The popped element is: banana
The dictionary is: {'orange': 3, 'apple': 2, 'grapes': 4}
```

Python Dictionary popitem()

The popitem() returns and removes an arbitrary element (key, value) pair from the dictionary.

The syntax of popitem() is:

```
dict.popitem()
```

popitem() Parameters

The popitem() doesn't take any parameters.

Return Value from popitem()

The popitem()
returns an arbitrary element (key, value) pair from the dictionary
removes an arbitrary element (the same element which is returned) from the dictionary.

Note: Arbitrary elements and random elements are not same. The popitem() doesn't return a random element.

Example: How popitem() works?

```
person = {'name': 'Phill', 'age': 22, 'salary': 3500.0}

result = person.popitem()
print('person = ',person)
print('Return Value = ',result)
```

When you run the program, the output will be:

```
person =  {'name': 'Phill', 'salary': 3500.0}
result =  ('age', 22)
```

The popitem() raises a KeyError error if the dictionary is empty.

Dictionary fromkeys()

The method **fromkeys()** creates a new dictionary with keys from *seq* and *values*set to value.

The syntax of fromkeys() method is:

```
dictionary.fromkeys(sequence[, value])
```

---

fromkeys() Parameters

The fromkeys() method takes two parameters:

sequence - sequence of elements which is to be used as keys for the new dictionary

value (Optional) - value which is set to each each element of the dictionary

---

Return value from fromkeys()

The fromkeys() method returns a new dictionary with the given sequence of elements as the keys of the dictionary.

If the value argument is set, each element of the newly created dictionary is set to the provided value.

Example 1: Create a dictionary from a sequence of keys

```
# vowels keys
keys = {'a', 'e', 'i', 'o', 'u' }

vowels = dict.fromkeys(keys)
print(vowels)
```

When you run the program, the output will be:

```
{'a': None, 'u': None, 'o': None, 'e': None, 'i': None}
```

Python Dictionary Methods	
Method	**Description**
clear()	Remove all items form the dictionary.
copy()	Return a shallow copy of the dictionary.

fromkeys(*seq*[, *v*])	Return a new dictionary with keys from *seq*and value equal to *v* (defaults to None).
get(*key*[,*d*])	Return the value of *key*. If *key* doesnot exit, return *d* (defaults to None).
items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.
pop(*key*[,*d*])	Remove the item with *key* and return its value or *d* if *key* is not found. If *d* is not provided and *key* is not found, raises KeyError.
popitem()	Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(*key*[,*d*])	If *key* is in the dictionary, return its value. If not, insert *key* with a value of *d* and return *d* (defaults to None).
update([*other*])	Update the dictionary with the key/value pairs from *other*, overwriting existing keys.
values()	Return a new view of the dictionary's values

Here are a few example use of these methods.

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

# Output: {'English': 0, 'Math': 0, 'Science': 0}
print(marks)

for item in marks.items():
   print(item)

# Output: ['English', 'Math', 'Science']
list(sorted(marks.keys()))
```

12.6.5 Other Dictionary Operations

Dictionary Membership Test

We can test if a key is in a dictionary or not using the keyword in. Notice that membership test is for keys only, not for values.

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: True
print(1 in squares)

# Output: True
print(2 not in squares)
```

```
# membership tests for key only not value
# Output: False
print(49 in squares)
```

Iterating Through a Dictionary

Using a for loop we can iterate though each key in a dictionary.

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
for i in squares:
   print(squares[i])
```

12.6.6 Built-in Functions with Dictionary

Built-in functions like all(), any(), len(), cmp(), sorted() etc. are commonly used with dictionary to perform different tasks.

Built-in Functions with Dictionary	
Function	**Description**
all()	Return True if all keys of the dictionary are true (or if the dictionary is empty).
any()	Return True if any key of the dictionary is true. If the dictionary is empty, return False.
len()	Return the length (the number of items) in the dictionary.
cmp()	Compares items of two dictionaries.
sorted()	Return a new sorted list of keys in the dictionary.

Here are some examples that uses built-in functions to work with dictionary.

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: 5
print(len(squares))

# Output: [1, 3, 5, 7, 9]
print(sorted(squares))
```

12.7 Conversion between data types

We can convert between different data types by using different type conversion functions like int(), float(), str() etc.

```
>>> float(5)
5.0
```

Conversion from float to int will truncate the value (make it closer to zero).

```
>>> int(10.6)
10
>>> int(-10.6)
```

```
-10
Conversion to and from string must contain compatible values.
>>> float('2.5')
2.5
>>> str(25)
'25'
>>> int('1p')
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1p'
We can even convert one sequence to another.
>>> set([1,2,3])
{1, 2, 3}
>>> tuple({5,6,7})
(5, 6, 7)
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
To convert to dictionary, each element must be a pair
>>> dict([[1,2],[3,4]])
{1: 2, 3: 4}
>>> dict([(3,26),(4,44)])
{3: 26, 4: 44}
```