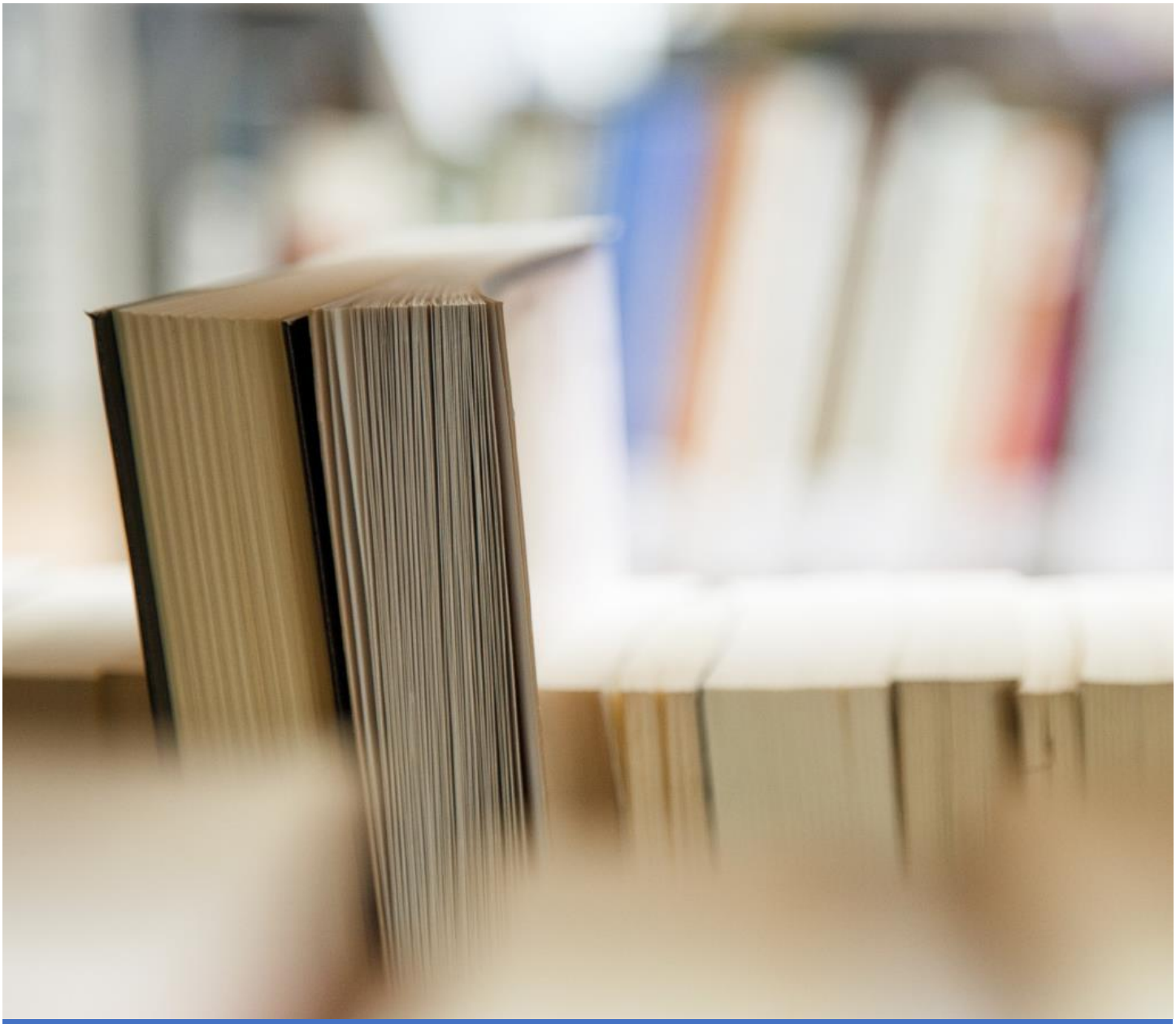

PYTHON CORE

BY: AMAR PANCHAL

Contents

PYTHON MASTER CLASS

BY
AMAR PANCHAL
9821601163

1. INTRODUCTION

A programming language is an artificial language designed to communicate instructions to a machine, usually a computer. Programming language is used to create programs (i.e. set of instructions) that control the behavior of a machine and/or to express algorithms precisely. Programming languages uses the same general principles, so after learning any one language, it is easy to grasp another.

Python is a **high-level general purpose programming language,** created by **Guido van Rossum** and first released in **1991**, that is used in a wide variety of application domains. Python has the right combination of performance and features that demystify program writing.

Features

1. It is **simple** and **easy to learn**.
2. Python implementation is under an **open source license** that makes it freely usable and distributable, even for commercial use.
3. It works on **many platforms** such as Windows, Linux, etc.
4. It is an **interpreted language**.
5. It is an **object-oriented language**.
6. **Embeddable** within applications as a scripting interface.
7. Python has a **comprehensive set** of packages to accomplish various tasks.

Python is an **interpreted language**, as opposed to a compiled one, though the distinction is blurry because of the presence of the bytecode compiler. Python source code is compiled into bytecode, so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). Interpreted languages typically have a shorter development/debug cycle than compiled ones, and also their programs generally also run slowly. Please note that Python uses a **7-bit ASCII** character set for program text.

The latest stable releases can always be found on Python's website **(http://www.python.org/)**. There are two recommended production-ready Python versions at this point in time, because at the moment there are two branches of stable releases: 2.x and 3.x. Python follows a modular programming approach, which is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality. Conceptually, modules represent a separation of concerns, and improve maintainability by enforcing logical boundaries between components.

Pythonic

"Pythonic" is a **bit different idea/approach of writing programs**, which is usually not followed in other programming languages. For example, to loop all elements of an iterable using for statement, usually the following approach is followed:

```
>>> food = ['pizza','burger','noodles']
>>> for i in range(len(food)):
        print(food[i])

# Output
'pizza'
'burger'
'noodles'
```

A cleaner Pythonic approach is:


```
>>> food = ['pizza','burger','noodles']
>>> for piece in food:
        print(piece)
# Output
'pizza'
'burger'
'noodles'
```

HISTORY

Python was created in the early 1990s by **Guido van Rossum** at Centrum Wiskunde & Informatica in the Netherlands as a successor of a language called "ABC". Guido remains Python's principal author, although it includes many contributions from others. When he began implementing Python, Guido van Rossum was also reading the published scripts from **"Monty Python's Flying Circus"**, a BBC comedy series from the 1970s. Van Rossum thought he needed a name that was short, unique, and slightly mysterious, so he decided to call the language "Python". In 1995, Guido continued his work on Python at the **Corporation for National Research Initiatives** in Reston, Virginia, where he released several versions of the software. In May 2000, Guido and the Python core development team moved to "BeOpen.com" to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations. In 2001, the Python Software Foundation was formed, a non-profit organization created specifically to own Python-related intellectual property. Zope Corporation is a sponsoring member of the PSF.

Integrated Development Environment

An Integrated Development Environment (IDE) is an application that provides comprehensive facilities for software development. An IDE normally consists of a **source code editor, compiler and/or interpreter, and a debugger.**

IDLE

IDLE is an **IDE**, and it is the basic editor and interpreter environment which ships with the standard distribution of Python. IDLE is built using the **"Tkinter" GUI toolkit**, and has the following features: Coded in Python, using the Tkinter GUI toolkit. Cross-platform i.e. works on Windows and Unix. Has source code editor with multiple undo, text highlighting, smart indent, call tips and many other features (shown in figure 1-2). Has a Python shell window, also known as "interactive interpreter" (shown in figure 1-1).

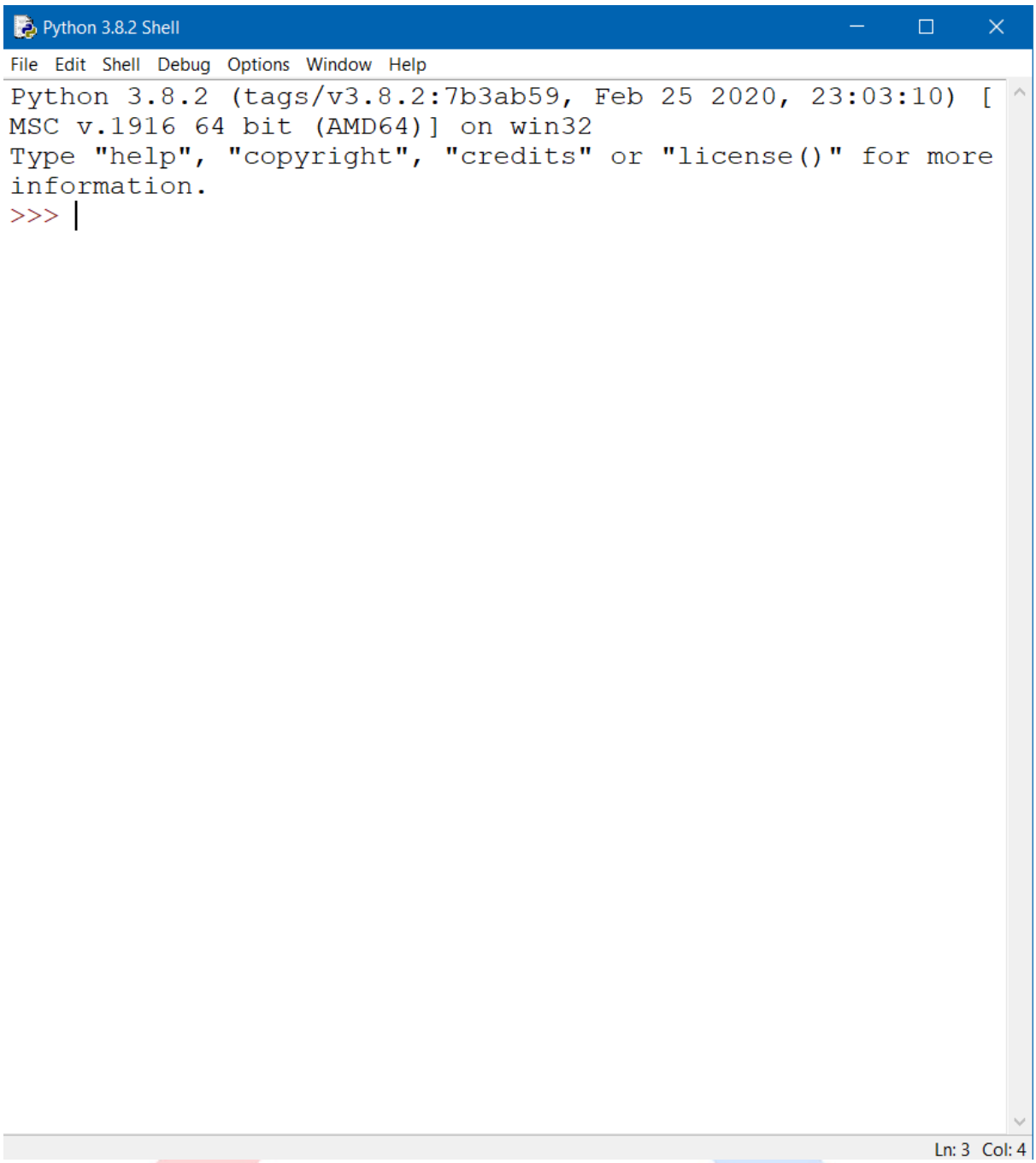

Figure 1-1: IDLE's Python Shell

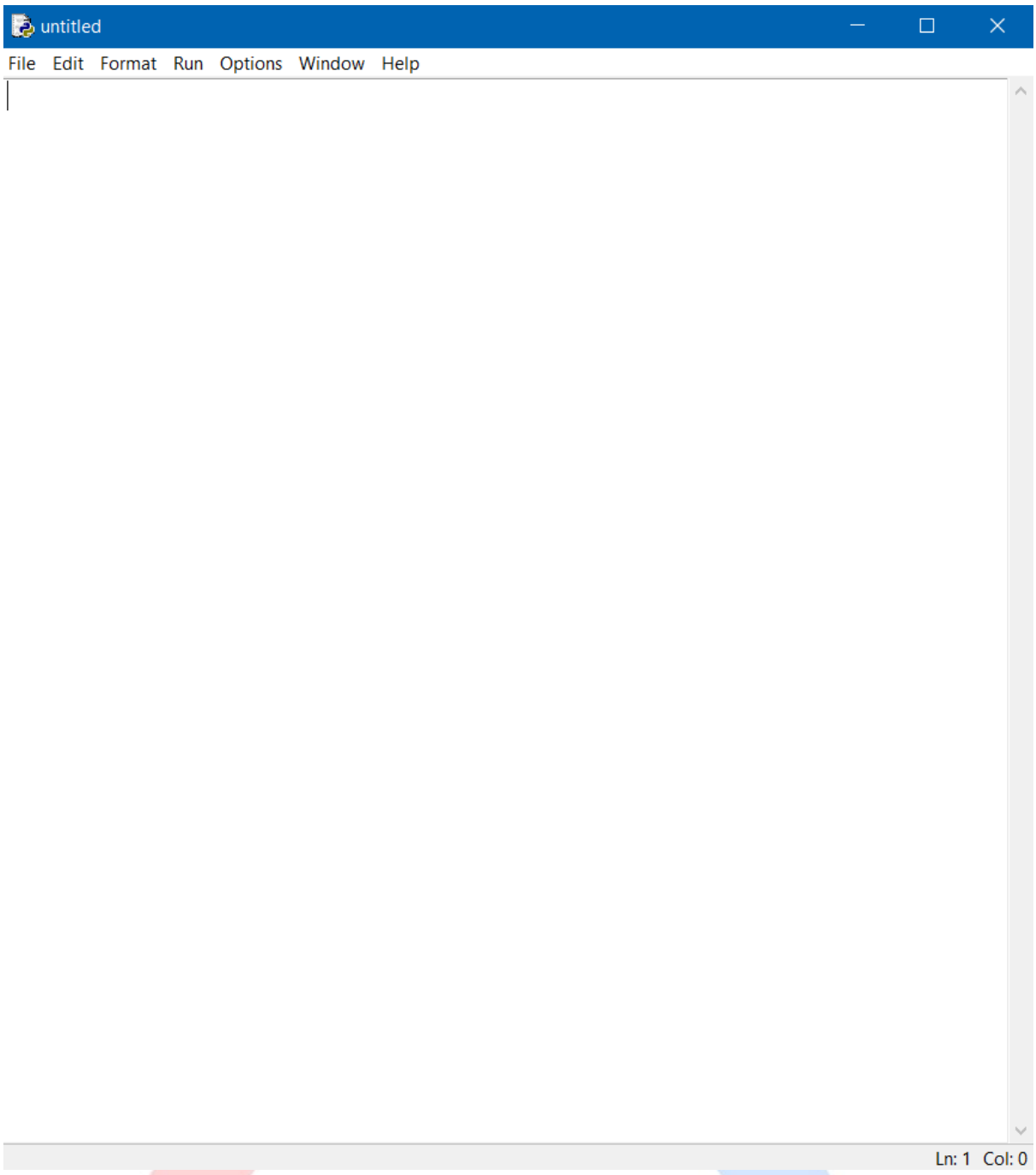

Figure 1-2: IDLE's Source Code Editor

Check version in command Line:

Python --version

>>> Python 3.0

Invoking Python interpreter in command line:

python (Hit Enter)

>>>

In command line, run a file:

python hello.py OR py hello.py

>>> "Hello World"

There are multiple ways to close the Python shell:

>>> exit()

Or

>>> quit()

2. Variable, Identifier and Literal

A variable is a **storage location** that has an associated symbolic name (called "identifier"), which contains some **value** (can be literal or other data) that can change. An identifier is a name used to **identify a variable, function, class, module or other object**. Literal is a notation for **constant values** of some **built-in type**. Literal can be **string, plain integer, long integer, floating point number, imaginary number**.

<variable name**> = <value>**

For e.g., in the expressions
>>> var1 = 5
>>> var2 = 'Tom'

var1 and var2 are **identifiers**, while 5 and ' Tom' are **integer** and **string** literals, respectively.

There are some **rules** that need to be followed for valid identifier naming:

1. Variables names must **start with** a **letter** or an **underscore**.

   x = True # valid

   _y = True # valid

   9x = False # starts with numeral

   => SyntaxError: invalid syntax

   $y = False # starts with symbol

   => SyntaxError: invalid syntax

2. The remainder of your variable name may consist of **letters**(uppercase or lowercase character), **numbers**(0-9) and **underscores**('_').

   has_0_in_it = "Still Valid"

3. Names are **case sensitive**.

   x = 9

   y = X * 5

=>NameError: name 'X' is not defined

4. Identifiers can be of **unlimited length**.

Token

A token is a string of one or more characters that is significant as a group. Consider an expression:
sum=6+2

Token	Token type
sum	Identifier
=	Assignment operator
6	Integer literal
+	Addition operator
2	Integer literal

The process of converting a sequence of characters into a sequence of tokens is called **"lexical analysis"**. A program or function that performs lexical analysis is called a lexical analyzer, lexer, or tokenizer. A lexer is generally combined with a parser (beyond the scope of this book), which together analyze the syntax of computer language. Python supports the following categories of tokens: NEWLINE, INDENT, DEDENT, identifiers, keywords, literals, operators, and delimiters.

Keywords

The following identifiers (as shown as output in the following code) are used as **reserved words** (or "keywords") of the language, and **cannot** be used as ordinary identifiers.

`False`	`await`	`else`	`import`	`pass`
`None`	`break`	`except`	`in`	`raise`
`True`	`class`	`finally`	`is`	`return`
`and`	`continue`	`for`	`lambda`	`try`
`as`	`def`	`from`	`nonlocal`	`while`
`assert`	`del`	`global`	`not`	`with`
`async`	`elif`	`if`	`or`	`yield`


```
>>> import keyword
>>> for kwd in keyword.kwlist:
        print(kwd, end = " ")
#Output
```

and as assert break class continue def del elif else except exec finally for from global if import in is lambda not or pass print raise return try while with yield

One can also check if an identifier is a keyword or not using iskeyword () function.

```
>>> import keyword

>>> keyword.iskeyword('hi')

False

>>> keyword.iskeyword('print')

True
```

3. Operators and Operands

An operator is a **symbol** (such as +, x, etc.) that **represents an operation**. An operation is an **action or procedure** which produces a new value from **one or more input** values called **operands**. There are two types of operators: **unary and binary**. Unary operator operates only on one operand, such as negation. On the other hand, binary operators operate on two operands, which includes addition, subtraction, multiplication, division, exponentiation operators etc.

Consider an expression 3 + 8, here 3 and 8 are called operands, while V is called operator. The operators can also be categorized into:

1. Arithmetic operators.
2. Comparison (or Relational) operators.
3. Assignment operators.
4. Logical operators.
5. Bitwise operators.
6. Membership operators.
7. Identity operators.

Arithmetic operator

The arithmetic operators with a short note on the operators.

Operator	Description
+	**Addition operator** - Add operands on either side of the operator.
–	**Subtraction operator** – Subtract right hand operand from left hand operand.
*	**Multiplication operator** – Multiply operands on either side of the operator.
/	**Division operator** – Divide left hand operand by right hand operand.
%	**Modulus operator** – Divide left hand operand by right hand operand and return remainder.
* *	**Exponent operator** – Perform exponential (power) calculation on operands.
//	**Floor Division operator** – The division of operands where the result is the quotient in which the digits after the decimal point are removed.

The following example illustrates the use of the above discussed operators.

```
>>> a = 20
>>> b = 45.0
```

```
>>> a + b
65.0
>>> a - b
- 25.0
>>> a * b
900.0
>>> b / a
2.25
>>> b % a
5.0
>>> a ** b
3.5184372088832e+58
>>> b // a
2.0
```


Relational operators

A relational operator is an operator that tests some kind of relation between two operands.

Operator	Description
==	Check if the values of two operands are equal.
!=	Check if the values of two operands are not equal.
<>	Check if the value of two operands are not equal (same as != operator).
>	Check if the value of left operand is greater than the value of right operand.

<	Check if the value of left operand is less than the value of right operand.
>=	Check if the value of left operand is greater than or equal to the value of right operand.
<=	Check if the value of left operand is less than or equal to the value of right operand.

The following example illustrates the use of the above discussed operators.


```
>>> a, b = 20, 40
>>> a == b
False
>>> a != b
True
>>> a <> b
True
>>> a > b
False
>>> a >> a>=b
False
>>> a<=b
True
```

Assignment operators

Assignment operator is an operator which is used to bind or rebind names to values. Augmented assignment is the combination, in a single statement, of a binary operation and an assignment statement. An augmented assignment expression like x+=l can be rewritten as x=x+1.

Operator	Description
=	**Assignment operator** - Assigns values from right side operand to left side operand.
+=	**Augmented assignment operator** - It adds right side operand to the left side operand and assign the result to left side operand.
-=	**Augmented assignment operator** - It subtracts right side operand from the left side operand and assign the result to left side operand.
*=	**Augmented assignment operator** - It multiplies right side operand with the left side operand and assign the result to left side operand.
/=	**Augmented assignment operator** - It divides left side operand with the right side operand and assign the result to left side operand.
%=	**Augmented assignment operator** - It takes modulus using two operands and assign the result to the left side operand.

**=	**Augmented assignment operator** - Performs exponential (power) calculation on operands and assigns value to the left side operand.
//=	**Augmented assignment operator** - Performs floor division on operators and assigns value to the left side operand

The following example illustrates the use of the above discussed operators.

```
>>> a,b = 20, 40
>>> c=a+b
>>> c
60
>>> a, b = 2.0, 4.5
>>> c = a + b
>>> c
6.5
>>> c += a
>>> c
8.5
>>> c -= a
>>> c
6.5
>>> c *= a
>>> c
13.0
>>> c /= a
>>> c
```

6.5

>>> c %= a

>>> c

0.5

>>> c **= a

>>> c

0.25

>>> c //= a

>>> c

0.0

Bitwise operators

A bitwise operator operates on one or more bit patterns or binary numerals at the level of their individual bits. Table 2-5 enlist the bitwise operators with description.

Operator	Description
&	**Binary AND operator** - Copies corresponding binary 1 to the result, if it exists in both operands.
I	**Binary OR operator** - Copies corresponding binary 1 to the result, if it exists in either operand.
^	**Binary XOR operator** - Copies corresponding binary 1 to the result, if it is set in one operand, but not both.

~	**Binary ones complement operator** - It is unary and has the effect of flipping bits.
<<	**Binary left shift operator** - The left side operand bits are moved to the left side by the number on right side operand.
>>	**Binary right shift operator** - The left side operand bits are moved to the right side by the number on right side operand.

The following example illustrates the use of the above discussed operators.


```
>>> a,b = 60, 13
>>> a&b 12
>>> a|b 61
>>> a^b 49
>>> ~a
-61
>>> a<<2 240
>>> a>>2
15
```

In the above example, the binary representation of variables a and b are 00111100 and 00001101, respectively. The above binary operations example is tabulated in table.

Bitwise operation	Binary representation	Decimal representation
a&b	00001100	12

a | b	00111101	61
a^b	00110001	49
~a	11000011	-61
a<<2	11110000	240
a>>2	00001111	15

Logical operators

Logical operators compare boolean expressions and return a boolean result.

Operator	Description
and	**Logical AND operator** - If both the operands are true (or non-zero), then condition becomes true.
or	**Logical OR operator** - If any of the two operands is true (or non-zero), then condition becomes true.
not	**Logical NOT operator** - The result is reverse of the logical state of its operand. If the operand is true (or non-zero), then the condition becomes false.

The following example illustrates the use of the above discussed operators.

```
>>> 5>2 and 4<8
True
>>> 5>2 or 4>8
True
>>> not(5>2)
False
```

Membership operators

Membership operator is an operator which tests for membership in a sequence, such as string, list, tuple etc.

Operator	Description
in	Evaluate to true, if it find a variable in the specified sequence; otherwise false.
not in	Evaluate to true, if it does not find a variable in the specified sequence; otherwise false.

The following example illustrates the use of the above discussed operators.

```
>>> 5 in [0,5,10,15]
True
>>> 6 in [0,5,10,15]
```

False

>>> 5 not in [0,5,10,15]

False

>>> 6 not in [0,5,10,15]

True

Identity operators

Identity operators compare the memory locations of two objects.

Operator	Description
is	Evaluates to true, if the operands on either side of the operator point to the same object, and false otherwise.
is not	Evaluates to false, if the operands on either side of the operator point to the same object, and true otherwise.

The following example illustrates the use of the above discussed operators.

>>> a=b=3.1

>>> a is b True

>>> id(a)

30984528

>>> id(b) 30984528

>>> c,d=3.1,3.1

>>> c is d False

```
>>> id(c)

35058472

>>> id (d)

30984592

>>> c is not d

True

>>> a is not b

False
```

Operator precedence

Operator precedence determines how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator. In the expression x=7+3*2, x is assigned 13, not 20, because operator * has higher precedence than +, so it first multiplies 3*2 and then adds into 7.
Table summarizes the operator's precedence in Python, from lowest precedence to highest precedence (from top to bottom). Operators in the same box have the same precedence.

Operator
not, or, and
in, not in
is, is not
=, %, =/, =//, -=, +=,*=, **=

<>, ==, !=
<=, <, >, >=
^, /
&
>>,<<
+/ –
*, /, %, //
~,+, –
**

4. INPUT / OUTPUT

To get input from the user, use the ***"input"*** function.

```
>>> name = input("What is your name? ")
What is your name?
```

If the user types **"Bob"** and hits enter, the variable name will be **assigned** to the string "Bob":

```
>>> name = input("What is your name? ")
#Output : What is your name? Bob
print(name)
#Output : Bob
```

Note that the **input is always of type str**, which is important if you want the user to enter numbers. Therefore, you need to convert the str before trying to use it as a number:

```
x = input("Write a number:")
# Output : Write a number: 10
x / 2
# Output : TypeError: unsupported operand type(s) for /: 'str' and 'int'
float(x) / 2
# Output : 5.0
```

5. CONDITIONAL STATEMENTS

IF

It consists of a **Boolean expression** which results is either **TRUE or FALSE** followed by one or more statements.

SYNTAX:

```
if condition:
        # code
```

Example :

```
>>> var = 100
>>> if var == 100:
        print("var has a value", var)
```

#Output : var has a value 100

IF ELSE

SYNTAX

```
if condition:
        #code
else:
        #code
```

In this **"if"** statement is **True, "if block"** will executed. Otherwise, **"if"** condition is **"false"** then **"else block"** will be executed respectively.

Example:

```
>>> var = 100
>>> if var != 100:
        print('var does not have value 100')
    else:
        print('var has value 100')
```

#Output : var has value 100

IF ELIF ELSE

SYNTAX

if condition:
    #code

elif condition:
    #code

else:
    #code

Example:

```
>>> var=100
>>> if var<100:
        print('var has value less than 100')
    elif var>100:
        print('var has value greater than 100')
    else:
        print('var has value 100')
```

#Output : var has value 100

6. LOOPS

A loop statement allows us to execute a statement or group of statements mutiple times.

WHILE LOOP

The while statement is used for repeated execution of group of statements as long as an expression is true

SYNTAX
while condition:
    #code

Example
```
>>> var=1
>>> while var <= 5:
        print('Count ' , var)
        var = var + 1
Output :
Count 1
Count 2
Count 3
Count 4
Count 5
>>> print('The loop ends')
#Output : The loop ends
```


FOR LOOP

The for statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object

SYNTAX
for iterating_var in sequence:
    #code

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

```
>>> for i in [1,2,3,4,5]:
        print('Count ',i)
#Output :
Count 1
Count 2
Count 3
Count 4
Count 5
>>> print('The loop ends')
#Output : The loop ends
```

LOOP CONTROL STATEMENTS

BREAK STATEMENT

The break statement terminates the nearest enclosing loop, skipping the optional else clause if the loop has one.

Example

```
a=5
for i in range(10):
        if i==a:
                break
        else :
                print(i, end = " ")
print("value of i outside the loop is ",i)
#Output :
0 1 2 3 4
value of i outside the loop is 5
```

CONTINUE STATEMENT

The continue statement makes the current nearest enclosing loop to skip one iteration and executes the remaining ones.

Example

```
for val in "string":
        if val == "i":
                continue
        print(val)
print("The end")
```

#Output :
s
t
i
r
n
g
The end

PASS STATEMENT

The pass statement is a null operation, nothing happens when it is executed. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed.

Example

```
for letter in 'Python':
        if letter == 'h':
        pass
        print('This is pass block')
print(letter)
print("Good bye!")
```

#Output
P
y
t
This is pass block
h
o
n
Good bye!

7. Standard Data Types

Numbers

Number data types store numeric values. They are immutable data types, means that changing the value of a number data type results in a newly allocated object.

Number objects are created when you assign a value to them. For example −

var1 = 1
var2 = 10

You can also delete the reference to a number object by using the del statement. The syntax of the del statement is −
del var1[,var2[,var3[....,varN]]]]

You can delete a single object or multiple objects by using the del statement.
For example −
del var del var_a, var_b

Python supports four different numerical types −

- **int (signed integers)** − They are often called just integers or ints, are positive or negative whole numbers with no decimal point.

- **long (long integers)** − Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.

- **float (floating point real values)** − Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).

- **complex (complex numbers)** − are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

Example

int	**long**	**float**	**complex**
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEL	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- Python allows you to use a lowercase L with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
- A complex number consists of an ordered pair of real floating point numbers denoted by a + bj, where a is the real part and b is the imaginary part of the complex number.

Number Type Conversion

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

- Type **int(x)** to convert x to a plain integer.
- Type **long(x)** to convert x to a long integer.
- Type **float(x)** to convert x to a floating-point number.
- Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.
- Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

Mathematical Functions

Function	Returns
abs(x)	The absolute value of x: the (positive) distance between x and zero.
ceil(x)	The ceiling of x: the smallest integer not less than x
cmp(x, y)	-1 if x < y, 0 if x == y, or 1 if x > y
exp(x)	The exponential of x: e**x
fabs(x)	The absolute value of x.
floor(x)	The floor of x: the largest integer not greater than x
log(x)	The natural logarithm of x, for x> 0
log10(x)	The base-10 logarithm of x for x> 0.
max(x1, x2,...)	The largest of its arguments: the value closest to positive infinity
min(x1, x2,...)	The smallest of its arguments: the value closest to negative infinity
modf(x)	The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
pow(x, y)	The value of x**y.

round(x [,n])	x rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0.
sqrt(x)	The square root of x for x > 0

8. String

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable.
For example −
var1 = 'Hello World!'
var2 = "Python Programming"
Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

For example −

```
var1 = 'Hello World!'

var2 = "Python Programming"

print("var1[0]: ", var1[0])

print("var2[1:5]: ", var2[1:5])

#Output

var1[0]:  H

var2[1:5]:  ytho
```

Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether.

For example −

```
var1 = 'Hello World!'
```

```
print("Updated String :- ", var1[:6] + 'Python')
#Output
Updated String :-  Hello Python
```

String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then −

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*****	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[ ]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n
%	Format - Performs String formatting	See at next section

Built-in String Methods

capitalize() Capitalizes first letter of string
center(width, fillchar) Returns a space-padded string with the original string centered to a total of width columns.
count(str, beg= 0,end=len(string)) Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
decode(encoding='UTF-8',errors='strict') Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
encode(encoding='UTF-8',errors='strict') Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
endswith(suffix, beg=0, end=len(string)) Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
expandtabs(tabsize=8) Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
find(str, beg=0 end=len(string)) Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
index(str, beg=0, end=len(string)) Same as find(), but raises an exception if str not found.
isalnum()

Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
isalpha() Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
isdigit() Returns true if string contains only digits and false otherwise.
islower() Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
isnumeric() Returns true if a unicode string contains only numeric characters and false otherwise.
isspace() Returns true if string contains only whitespace characters and false otherwise.
istitle() Returns true if string is properly "titlecased" and false otherwise.
isupper() Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
join(seq) Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
len(string) Returns the length of the string

ljust(width[, fillchar]) Returns a space-padded string with the original string left-justified to a total of width columns.
lower() Converts all uppercase letters in string to lowercase.
lstrip() Removes all leading whitespace in string.
maketrans() Returns a translation table to be used in translate function.
max(str) Returns the max alphabetical character from the string str.
min(str) Returns the min alphabetical character from the string str.
replace(old, new [, max]) Replaces all occurrences of old in string with new or at most max occurrences if max given.
rfind(str, beg=0,end=len(string)) Same as find(), but search backwards in string.
rindex(str, beg=0, end=len(string)) Same as index(), but search backwards in string.
rjust(width,[, fillchar]) Returns a space-padded string with the original string right-justified to a total of width columns.
rstrip() Removes all trailing whitespace of string.

split(str="", num=string.count(str)) Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
splitlines(num=string.count('\n')) Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
startswith(str, beg=0,end=len(string)) Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
strip([chars]) Performs both lstrip() and rstrip() on string.
swapcase() Inverts case for all letters in string.
title() Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
translate(table, deletechars="") Translates string according to translation table str(256 chars), removing those in the del string.
upper() Converts lowercase letters in string to uppercase.
zfill (width) Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).

isdecimal()

Returns true if a unicode string contains only decimal characters and false otherwise.

9. List

Python has a number of built-in types to group together data items. The most versatile is the list, which is a group of comma-separated values (items) between square brackets. List items need not to be of same data type.

```
>>> a=[1 spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```


Python allows adding a trailing comma after last item of list, tuple, and dictionary, . There are several reasons to allow this:
When the list, tuple, or dictionary elements spread across multiple lines, one needs to remember to add a comma to the previous line. Accidentally omitting the comma can lead to error that might be hard to diagnose. Always adding the comma avoids this source of error (example is given below).
If comma is placed in each line, they can be reordered without creating a syntax error. The following example shows the consequence of missing a comma while creating list.

```
>>> a=[ 'hi', 'hello' 'bye', 'tata', ]
>>> a
['hi', 'hellobye', 'tata']
```

This list looks like it has four elements, but it actually contains three: ‘hi’, ‘hellobye’ and ‘ tata’. Always adding the comma avoids this source of error.

List creation
List can be created in many ways.

Using square brackets
As discussed before, most common way of creating a list is by enclosing comma-separated values (items) between square brackets. Simply assigning square bracket to a variable creates an empty list.

```
>>> a=[]
>>> a []
```

```
>>> type(a)
<type 'list'>
```

Using other lists

A list can also be created by copying a list or slicing a list.

```
>>> a=['spam', 'eggs', 100, 1234]
>>> b=a[:]
>>> b
['spam', 'eggs', 100, 1234]
>>> c=a[l:3]
>>> c
['eggs', 100]
```

List comprehension

List comprehension provides a concise way to create list. A list comprehension consist of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a sub-sequence of those elements that satisfy a certain condition. For example, creating a list of squares using an elaborate approach is as follows:


```
>>> squares=[]
>>> for x in range(10):
        squares.append(x**2)
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The same can be obtained using list comprehension as:

```
>>> squares=[x**2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Alternatively, it can also be obtained using map () built-in function:

```
>>> squares=map(lambda x: x**2,range(10))
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The following list comprehension combines the elements of two lists if they are not equal:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x!=y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it's equivalent to:

```
>>> combs=[]
>>> for x in [1,2,3]:
```

```
        for y in [3,1,4]:
                if x!=y:
                        combs.append((x,y))
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Using built-in function

List can also be created using built-in function list (). The list ([iterable]) function return a list whose items are the same and in the same order as iterable items. The iterable can be either a sequence, a container that supports iteration, or an iterator object. If iterable is already a list, a copy is made and returned. If no argument is given, a new empty list is returned.

```
>>> list(('hi', 'hello','bye'))
['hi', 'hello', 'bye']
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
>>> list ( (10,50,))
[10, 50]
>>> list()
[]
```


Accessing list elements

Like string indices, list indices start at 0. To access values in list, use the square brackets with the index or indices to obtain a slice of the list.

```
>>> a=['spam','eggs',100,1234]
>>> a [ 0 ]
'spam'
>>> a [ 0 ] [1]
'p'
>>> a[1:3]
[ ' eggs ' , 100 ]
```

Updating list elements

It is possible to change individual or multiple elements of a list:

```
>>> a= [' spam', 'eggs', 100, 1234]
>>> a[0]='filter'
>>> a
['filter', 'eggs', 100, 1234]
>>> a[2:4]=455,56858
>>> a
```

['filter', 'eggs', 455, 56858]

The items of list can be updated by the elements of another iterable (list, tuple).

```
>>> a=[66.25,333,333,1,1234.5]
>>> b=[ 'hi', 'bye' ]
>>> a [ 2 : 4 ] =b
>>> a
[66.25, 333, 'hi', 'bye', 1234.5]
>>> a= [66.25,333,333, 1, 1234.5]
>>> b=('hi','bye')
>>> a [ 2 : 4 ] =b >>> a
[66.25, 333, 'hi', 'bye', 1234.5]
>>> a=[66.25,333,333,1,1234.5]
>>> b=['hi','bye']
>>> a[1:4:2]=b >>> a
[66.25, 'hi', 333, 'bye', 1234.5]
```


It is also possible to insert elements in a list.

```
>>> a=['spam','eggs',100,1234]
>>> a[1:1] = [ 'chair']
>>> a ['spam', 'chair', 'eggs', 100, 1234]
>>> a[1:1]=['hello','bye']
>>> a
['spam', 'hello', 'bye', 'chair', 'eggs', 100, 1234]
```

To insert a copy of list at the beginning of itself:

```
>>> a=['spam','eggs',100,1234]
>>> a[:0]=a
>>> a
['spam', 'eggs', 100, 1234, 'spam', 'eggs', 100, 1234]
```

Deleting list elements

To remove a list element, one can use either the del statement (if you know the element index to delete) or remove () method. The following example depicts deletion of an element using del statement.

```
>>> a=['spam', 'eggs100,1234]
>>> del a[1]
>>> a
```

['spam', 100, 1234]

The del statement can also be used to explicitly remove the entire list.

```
>>> a=['spam','eggs',100,1234]
 >>> del a
>>> a
Traceback (most recent call last):
File "", line 1, in
NameError: name 'a' is not defined
```

The following is an interesting case of del statement:

```
>>> a=['spam','eggs',100,1234]
>>> del a[1],a[1]
>>> a
['spam', 1234]
```


It should be noted that the same index is deleted twice. As soon as an element is deleted, the indices of succeeding elements are changed. So deleting an index element n times, would actually delete n elements.
It is also possible to delete multiple items of a list.

```
>>> a=['spam','eggs',100,1234]
>>> a [1:3] = []
>>> a
['spam', 1234 ]
```

Swapping lists

There might be a scenario in which multiple lists needs to be swapped among themselves. This is can done easily using multiple assignments expression.

```
>>> a=[10,20,30]
>>> b=[40,50,60]
>>> c= [70, 80, 90]
>>> a,b,c=c,a,b
>>> a
[70, 80, 90]
>>> b
[10, 20, 30]
>>> c
[40, 50, 60]
```

Looping techniques

List can also be used in iteration operation, for example:

```
>>> for a in [4,6,9,2]:
        print(a)
4
6
9
2
>>> for a in [4,6,9,2]:
        print(a)
4
6
9
2
```

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


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

To loop over two or more sequences at the same time, the entries can be paired with the zip () function.

```
>>> questions=['name','quest','favorite color']
>>> answers= [ ' lancelot' , ' the holy grailblue ' ] * .
>>> for q,a in zip(questions,answers):
        print('What is your {0}? It is {1}.'.format(q,a))
What is your name?
It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

While iterating a mutable sequence, if there is a need to change the sequence (for example to duplicate certain items), it is recommended to make a copy of the sequence before starting iteration. Looping over a sequence does not implicitly make a copy. The slice notation makes this especially convenient.

```
>>> words=['cat','window','defenestrate']
>>> for w in words[:]:
        if len(w)>6:
                words.insert(0,w)
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

Nested list

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

```
>>> q= [2,3]
>>> p=[1,q,4]
>>> len(p)
3
>>> P [ 1 ]
[2, 3]
>>> p [1] [0]
2
>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra']', 4]
>>> q
[2, 3, 'xtra']
```


Note that in the last example, p [1] and q refer to the same object, which can cross-checked using id () built-in function.

```
>>> id(q)
104386192
>>> id(p[l] )
104386192
```

Some list operations

Some of the list operations supported by Python are given below.

Concatenation

List concatenation can be carried out using + operator.

```
>>> [1,2,3]+['hi','hello']
[1, 2, 3, 'hi', 'hello']
```

Repetition

The * operator can be used to carry out repetition operation.

```
>>> ['Hey']*3
['Hey', 'Hey', 'Hey']
```

Membership operation

List also supports membership operation i.e. checking the existence of an element in a list.

```
>>> 4 in [6,8, 1,3,5,0]
False
>>> 1 in [6,8,1,3,5,0]
True
```

Slicing operation

As list is a sequence, so indexing and slicing work the same way for list as they do for strings. All slice operations return a new list containing the requested elements:


```
>>> a=['spam','eggs100,1234]
>>> a[2]
100
>>> a[-2]
100
>>> a [ 1: 3 ]
[ 'eggs' , 100]
>>> a[:]
['spam', 'eggs', 100, 1234]
```

List slicing can be in form of steps, the operation s [i : j : k] slices the list s from i to j with step k.

```
>>> squares=[x**2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
>>> squares[2:12:3 ]
[4, 25, 64, 121]
```

Slice indices have useful defaults, an omitted first index defaults to zero, an omitted second index defaults to the size of the list being sliced.

```
>>> squares=[x**2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144,,.169, 196]
```

```
>>> squares[5:]
[25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
>>> squares[:5]
[0, 1, 4, 9, 16]
```

List methods

Below are the methods of list objects.

```
>>> list.append(x)
```

Add an item to the end of the list. It is same as list [len (list) : len (list)] = [x].

```
>>> a=[66.25,333,333, 1, 1234.5]
>>> a.append(45)
>>> a
[66.25, 333, 333, 1, 1234.5, 45]
>>> a=[66.25,333,333,1, 1234.5]
>>> a[len(a):len(a)]=[45]
>>> a
[66.25, 333, 333, 1, 1234.5, 45]
```


list.extend(L)

Extend a list by appending all the items of a given list. It is same as list [len (list) : len (list)] =L.

```
>>> a=[66.25,333,333,1,1234.5]
>>> b= [ 7.3,6.8 ]
>>> a.extend(b)
>>> a
[66.25, 333, 333, 1, 1234.5, 7.3, 6.8]
>>> a= [66.25,333,333, 1, 1234.5]
>>> b= [ 7.3,6.8 ]
>>> a[len(a):len(a)]=b
>>> a
[66.25, 333, 333, -1, 1234.5, 7.3, 6.8]
```

list.insert(i,x)

Insert an item at a given position in the list. The first argument i is the index before which an item x need to be inserted. It is same as list [i : i] = [x].

```
>>> a=[66.25,333,333,1,1234.5]
>>> a.insert(2,5.7)
>>> a [66.25, 333, 5.7, 333, 1, 1234.5]
```

```
>>> a=[66.25,333,333, 1,1234.5]
>>> a [2 : 2] = [5.7]
>>> a
[66.25, 333, 5.7, 333, 1, 1234.5]
```

list.index(x [,i [,j]])
Return the index in the list of the first occurrence of item x. In other words, it returns the smallest index k such that list [k] ==x and i<=k< j. A ValueError exception is raised in absence of item x.

```
>>> a=[66.25,333,333,1,1234.5]
>>> a.index(333)
1
```

list.remove(x)
Remove the first item from the list whose value is x. An error (ValueError exception) occur in absence of item x. It is same as del list [list. index (x)].


```
>>> a=[66.25,333,333,1,1234.5]
>>> a.remove(333)
>>> a
[66.25, 333, 1, 1234.5]
>>> a=[66.25,333,333, 1,1234.5]
>>> del a[a.index(333)]
>>> a
[66.25, 333, 1, 1234.5]
```

list.pop([i])
Remove the item at the given position i in the list, and return it. If no index is specified (defaults to -1), pop () removes and returns the last item in the list.

```
>>> a=[66.25, 333,333, 1, 1234.5]
>>> a.pop(3)
1
>>> a [66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[66.25, 333, 333]
```

list.count(x)
Return the number of times item x appears in the list.

```
>>> a=[66.25,333,333, 1, 1234.5]
>>> a.count(333)
2
```

list.reverse()
Reverse the element's position in the list; no new list is returned. It is same as list=list [: : -1].

```
>>> a=[66.25,333,333,1,1234.5]
>>> a.reverse()
>>> a
[1234.5, 1, 333, 333, 66.25]
>>> a=[66.25,333,333, 1,1234.5]
>>> a=a[::-1]
[1234.5, 1, 333, 333, 66.25]
```

list.sort([cmp[,key[,reverse]]])
Sort the items of the list; no new list is returned. The optional arguments have same meaning as given in sorted () built-in function.


```
>>> a=[66.25, 333,333,1,1234.5]
>>> a.sort()
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> a= [66.25,333, ' abc ', 333,1, 'ab', 1234.5]
>>> a . sort ( )
>>> a
[1, 66.25, 333, 333, 1234.5, 'ab', 'abc']
>>> a=[66.25,333,'abc',333,1,'ab',1234.5]
>>> a.sort(reverse=True)
>>> a
['abc', 'ab', 1234.5, 333, 333, 66.25, 1]
```

Using list as Stack
The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out" approach). To add an item to the top of the stack, use append (). To retrieve an item from the top of the stack, use pop () without an explicit index.

For example:

```
>>> stack=[3,4,5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack [3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack [3, 4]
```

Using list as queue

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, list is not efficient for this purpose. While appending and popping of elements from the end of list are fast, doing inserting and popping from the beginning of a list is slow (because all of the other elements have to be shifted by one). To implement a queue, use collections . deque which was designed to have fast appends and pops from both ends.

For example:

```
>>> from collections import deque
>>> queue=deque(["Eric","John","Michael"])
>>> queue.append("Terry")
>>> queue.append("Graham")
>>> queue
deque(['Eric', 'John', 'Michael', 'Terry', 'Graham'])
>>> queue.popleft()
'Eric'
>>> queue.popleft()
'John'
>>> queue
deque(['Michael', 'Terry', 'Graham'])
```

10. Tuple

There is also another sequence type- tuple. Tuples is a sequence just like list. The differences are that tuple cannot be changed i.e. tuple is immutable, while list is mutable, and tuple use parentheses, while list use square brackets. Tuple items need not to be of same data type. Also, as mentioned previously, Python allow adding a trailing comma after last item of tuple.

```
>>> a=('spam','eggs',100,1234)
>>> a
('spam', 'eggs', 100, 1234)
>>> a=('spam','eggs',100,1234,)
>>> a
('spam', 'eggs', 100, 1234)
```

The above expressions are examples of “tuple packing” operation i.e. the values ‘spam’, ‘eggs’, 100 and 1234 are packed together in a tuple. The reverse operation is also possible,

For example:

```
>>> a1, a2 , a3, a4=a >>> al 'spam'
>>> a2 'eggs'
>>> a3 100
>>> a4 1234
```

This is called “sequence unpacking”, and works for any sequence on the right-hand side. Sequence unpacking requires the group of variables on the left to have the same number of elements as the length of the sequence. Note that multiple assignments are really just a combination of tuple packing and sequence unpacking.

```
>>> a,b=10,20
>>> a
10
>>> b
20
```

Tuple creation

Tuple can be created in many ways.

Using parenthesis

Creating a tuple is as simple as grouping various comma-separated values, and optionally these comma-separated values between parentheses.

```
>>> a= ('spameggs100,1234)
```

```
>>> a
('spam', 'eggs', 100, 1234)
>>> a=('spam','eggs',100,1234,)
>>> a
('spam', 'eggs', 100, 1234)
>>> a='spam','eggs',100,1234
>>> a
('spam', 'eggs', 100, 1234)
>>> a='spam','eggs',100,1234,
>>> a
('spam', 'eggs', 100, 1234)
```

A tuple with one item is created by a comma after the item (it is not necessary to enclose a single item in parentheses).

```
>>> a=(10)
>>> a
10
>>> type(a)
<type 'int'>
>>> b= (10, )
>>> b
(10, )
>>> type(b)
<type 'tuple'>
>>> c=10
>>> c (10, )
>>> type(c)
<type 'tuple'>
```

It is also possible to create an empty tuple by assigning parenthesis to a variable. >>> a=()
>>> a
0

Using other tuples

A tuple can also be created by copying a tuple or slicing a tuple.

```
>>> a=('spam','eggs',100,1234)
>>> b=a[:]
>>> b
('spam', 'eggs', 100, 1234)
>>> c=a[l:3]
```

```
>>> c
('eggs', 100)
```

Using built-in function

Tuple can also be created using built-in function tuple (). The tuple ([iterable]) function return a tuple whose items are same and in the same order as items of the iterable. The iterable may be a sequence, a container that supports iteration, or an iterator object. If iterable is already a tuple, it is returned unchanged. If no argument is given, an empty tuple is returned.

```
>>> tuple(['apple','pineapple','banana'])
('apple', 'pineapple', 'banana')
>>> tuple('apple')
('a', 'p', 'p', '1', 'e')
>>> tuple(['apple'])
('apple',)
```

Accessing tuple elements

The tuple index start with 0, and to access values in tuple, use the square brackets with the index or indices to obtain a slice of the tuple.

```
>>> a=('spam','eggs',100,1234)
>>> a[0]
'spam'
>>> a [ 1 : 3 ]
( 'eggs', 100)
```

Update tuple

Tuple is immutable, so changing element value or adding new elements is not possible. But one can take portions of existing tuples to create a new tuples.

```
>>> tuplel=(1,2,3)
>>> tuple2=('a','b','c')
>>> tuple3=tuplel+tuple2
>>> tuple3
(1, 2, 3, 'a', 'b', 'c')
>>> tuple3=tuple1[0:2]+tuple2[l:3]
>>> tuple3
(1, 2, 'b', 'c')
```

It is also possible to create tuple which contain mutable objects, such as lists.

```
>>> a= [1, 2,3] , [4,5,6, 7]
>>> a
(fl, 2, 3], [4, 5, 6, 7])
>>> a [0] [1]=200
>>> a
([1, 200, 3], [4, 5, 6, 7])
```

Deleting tuple

Removing individual tuple element is not possible. To explicitly remove an entire tuple, just use the del statement.

```
>>> a=('spam', 'eggs100,1234)
>>> del a
```

Swapping tuples

There might be a scenario in which multiple tuples needs to be swapped among themselves. This is can done easily using multiple assignments expression.


```
>>> a=(10,20,30)
>>> b= (40,50,60)
>>> c=(70,80,90)
>>> a,b,c=c,a,b
>>> a
(70, 80, 90)
>>> b
(10, 20, 30)
>>> c
(40, 50, 60)
```

Looping techniques

Tuple can also be used in iteration operation, for example:

```
>>> for a in (4,6,9,2):
        print(a)
4
6
9
2
>>> for a in (4,6,9,2):
        print(a)
4
```

```
6
9
2
```

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

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

To loop over two or more sequences at the same time, the entries can be paired with the zip () function.

```
>>> questions=('name','quest','favorite color')
>>> answers=('lancelot', 'the holy grail', 'blue')
>>> for q,a in zip(questions,answers):
        print('What is your {0}? It is {1}.'.format(q,a))

What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

Nested tuple

It is possible to create nested tuples.

```
>>> q=(2,3)
>>> r=[5,6]
>>> p= (1, q, 4, r)
>>> len(p)
4
 >>> p[1]
(2, 3)
>>> p [1] [0]
2
>>> p [ 3 ]
[5, 6]
>>> p
(1, (2, 3), 4, [5, 6])
```

Note that in the last example, p [1] and q refer to the same object, also p [3] and r refer to the same object.

Some tuple operations

Some of the tuple operations supported by Python are given below.

Concatenation
Tuple concatenation can be carried out using + operator.

```
>>> (1, 2,3) + ( 'hi', 'hello' )
(1, 2, 3, 'hi', 'hello')
```

Repetition
The * operator can be used to carry out repetition operation.

```
>>> ('Hey',)*3
('Hey', 'Hey', 'Hey')
```


Membership operation
Tuple also supports membership operation i.e. checking the existence of an element in a tuple.

```
>>> 4 in (6,8,1,3,5,0)
False
>>> 1 in (6, 8,1,3,5,0 )
True
```

Slicing operation
As tuple is a sequence, so indexing and slicing work the same way for tuple as they do for strings. All slice operations returns a new tuple containing the requested elements:

```
>>> a=('spam', 'eggs 1, 100, 1234)
>>> a[2]
100
>>> a[-2]
100
>>> a [ 1: 3 ]
('eggs', 100)
>>> a[:]
('spam', 'eggs', 100, 1234)
```

Tuple slicing can be in form of steps, the operation s [i : j : k] slices the tuple s from i to j with step k.

```
>>> squares= ( 0, 1,4,9,16,25,36,49,64,81,100,121,144,169,196)
>>> squares[2:12:3]
(4, 25, 64, 121)
```

Slice indices have useful defaults, an omitted first index defaults to zero, an omitted second index defaults to the size of the tuple being sliced.

```
>>> squares= (0, 1, 4,9,16,25,36,49,64, 81,100,121,144,169,196)
>>> squares[5:]
(25, 36, 49, 64, 81, 100, 121, 144, 169, 196)
>>> squares [ : 5 ]
(0, 1, 4, 9, 16)
```


11. Set

A set is an unordered collection with no duplicate elements. Basic uses include membership testing, eliminating duplicate entries from sequence, mathematical operations like union, intersection, difference, symmetric difference etc. As mentioned in chapter 2, sets are of two types: set (mutable set) and frozenset (immutable set).

```
>>> a=set(['spam','eggs',100,1234])
>>> a
set(['eggs', 100, 1234, 'spam'])
>>> a=set('abracadabra')
>>> a
set(['a', 'r', 'b', 'c', 'd'])
>>> a=frozenset(['spam','eggs’,100,1234])
>>> a
frozenset(['eggs', 100, 1234, 'spam'])
```

Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

Set creation

A set can be created in many ways.

Using curly braces

Non-empty set (not frozenset) can be created by placing a comma-separated list of elements within braces. Curly braces cannot be used to create an empty set, because it will create an empty dictionary that will be discussed in the next section.

```
>>> {'spam','eggs',100, 1234}
set([1234, 100, 'eggs', 'spam'])
```

Set comprehension

Python also supports set comprehension:

```
>>> a={x for x in 'abracadabra' if x not in 'abc'}
>>> a
set(['r', 'd'])
```

Using built-in function The built-in functions set () and frozenset () are used to create set and frozenset, respectively, whose elements are taken from iterable. If iterable is not specified, a new empty set is returned.


```
>>> a=set(('spam','eggs',100,1234))
>>> a
set(['eggs', 100, 1234, 'spam'])
>>> set ()
set ( [ ] )
>>> a=frozenset(('spam','eggs',100,1234))
>>> a
frozenset(['eggs', 100, 1234, 'spam'])
>>> frozenset()
frozenset([])
>>> set('abc')==frozenset('abc')
True
```

Deleting set

To explicitly remove an entire set, just use the del statement.

```
>>> ss=set('abracadabra')
>>> ss
set(['a', 'r', 'b', 'c', 'd'])
>>> del ss
>>> ss
Traceback (most recent call last): File "", line 1, in
```

NameError: name 'ss' is not defined

Looping techniques

It is also possible to iterate over each element of a set. However, since set is unordered, it is not known which order the iteration will follow.

```
>>> ss=set('abracadabra')
>>> ss
set(['a', 'r', 'b', 'c', 'd'])
>>> for item in ss:
        print(item)
a
r
b
c
d
```

Membership operation

Set also support membership operation.

```
a=set('abracadabra')
>>> a
set(['a', 'r', 'b', 'c', 'd'])
>>> 'r' in a
True
>>> 'rrr' not in a
True
>>> a^frozenset ( ' abracadabra ' )
>>> a
frozenset(['a', 'r', 'b', 'c', 'd"])
>>> 'r' in a
True
>>> 'rrr' not in a
True
```

Set methods

Below are the methods of both set and f rozenset objects. Note that the non-operator versions of these methods accepts any iterable as an argument, while their operator based counterparts require their arguments to be sets (set and frozenset).

isdisjoint(other)

Return True, if the set has no elements in common with other. Sets are disjoint, if and only if their intersection is the empty set.

```
>>> s1=set([5,10,15,20])
>>> s2=set([30,35,40])
>>> si.isdisjoint(s2)
True
>>> sl=frozenset ( [5,10,15,20])
>>> s2=frozenset([30,35,40])
>>> si.isdisjoint(s2)
True
>>> sl=set([5,10,15,20])
>>> s2=frozenset([30,35,40])
>>> si.isdisjoint(s2)
True
```

issubset(other)

Test whether every element in the set is in other.

```
>>> s1=set ( [5,15])
>>> s2=set([5,10,15,20])
>>> si.issubset(s2)
True
>>> si.issubset((5,10,15,20))
True
>>> sl=frozenset([5,15])
>>> s2=frozenset([5,10,15,20])
>>> si.issubset(s2)
True
>>> si.issubset((5,10,15,20))
True
```

The operator based version of the the above method is

set <=other. >>> s1=set ([5,15])

```
>>> s2=frozenset([5,10,15,20])
>>> s1<=s2
True
```

The operator based version

set>> s1=set([5,15])

```
>>> s2=frozenset([5,10,15,20])
>>> sl>> s1=set ( [5,15])
```

```
>>> s2=set( [5,10,15,20])
>>> s2.issuperset(s1)
True
>>> s2.issuperset((5,15))
True
>>> s1=frozenset([5,15])
>>> s2=frozenset([5,10,15,20])
>>> s2.issuperset(sl)
True
>>> s2.issuperset((5,15))
```

The operator based version of the the above method is set>=other.

```
>>> s1=set([5,15])
>>> s2=frozenset([5,10,15,20])
>>> s1>=s2
False
```


The operator based version set>other test whether the set is a proper superset of other, that is, set>=other and set!-other.

```
>>> s1=set([5,15])
>>> s2=frozenset( [5,10,15,20])
>>> s1>> s1=set ( [5,15])
>>> s2=[15, 20,25]
>>> s3=frozenset([30,35,40])
>>> s1.union(s2,s3)
set ( [35, 20, 5, 40, 25, 30, 15])
```

The operator based version of the the above method is set | other.

```
>>> s1=set([5,15])
>>> s2=set([15,20,25])
>>> s3=frozenset([30,35,40])
>>>s 1 | s2 | s3
set ([ 35, 20, 5, 40, 25, 30, 15])
```

intersection(other, …)
Return a new set with elements common to the set and all others .

```
>>> s1=set([5,10,15])
```

```
>>> s2=[15,20,25,10]
>>> s3=frozenset([30,15,35,40,10])
>>> s4=(40,50,10,15,20)
>>> si.intersection(s2,s3,s4)
set ([10, 15])
```

The operator based version of the the above method is set&other .

```
>>> s1=set([5,10,15])
>>> s2=set([15,20,25,10])
>>> s3=frozenset([30,15,35,40,10])
>>> s4=frozenset([40,50,10,15,20])
>>> s1&s2&s3&s4
set([10, 15])
```

difference(other, …)
Return a new set with elements in the set that are not in the others.


```
>>> s1=set([5,10,15])
>>> s2=[15,20,25,10]
>>> s3=frozenset( [30,-15, 35, 40, 10] )
>>> S4=(40,50,10,15,20)
>>> s3.difference(si,s2,s4)
frozenset([35, 30])
```

The operator based version of the the above method is set-other-. ..

```
>>> s1=set([5,10,15])
>>> s2=set([15,20,25,10])
>>> s3=frozenset([30,15,35,40,10])
>>> s4=frozenset([40,50,10,15,20])
>>> sl-s2-s3-s4 set( [5] )
>>> s3-sl-s2-s4 frozenset([35, 30])
```

symmetric_difference(other)
Return a new set with elements in either the set or other but not both.

```
>>> s1=set([5,10,15])
>>> s2=[15, 20,25, 10]
>>> si.symmetric_difference(s2)
set([25, 20, 5])
```

The operator based version of the the above method is set Another. \

```
>>> s1=set([5,10,15])
>>> s2=frozenset([15,20,25,10])
>>> s1^s2
set([25, 20, 5])
>>>s2^s1
frozenset([25, 20, 5])
```

copy()
Return a copy of the set.

```
>>> s=set([5,10,15,20])
>>> a=s.copy()
>>> a set ([10, 20, 5, 15])
>>> s=frozenset( [5,10,15,20])
>>> a=s.copy()
>>> a
frozenset([ 10, 20, 5, 15])
```


The following methods are available for set and do not apply to immutable instances of f rozenset.

update(other, …)

Update the set, adding elements from all others.

```
>>> s1=set([5,15])
>>> s2= (15, 20, 25)
>>> s3=frozenset([30,35,40])
>>> si.update(s2,s3)
>>> si
set ( [35, 20, 5, 40, 25, 30, 15])
```

The operator based version of the the above method is set | =other | .

```
 >>> s1=set([5,15])
>>> s2=set([15,20,25])
>>> s3=frozenset([30,35,40])
>>> si | =s2 | s3
>>> si
set( [35, 5, 40, 15, 20, 25, 30])
```

intersection_update(other, …)
Update the set, keeping only elements found in it and all others.

```
>>> s1=set([5,10,15])
>>> s2= [15,20,25, 10]
>>> s3=set([30,15,35,40,10])
>>> s4=(40,50, 10, 15,20)
>>> si.intersection_update(s2,s3,s4)
>>>                                                                    si
set( [10, 15])
```

The operator based version of the the above method is set&=other& . ..

```
>>> sl = set( [5,10,15] )
>>> s2 = set( [ 15, 20,25,10])
>>> s3=frozenset([30,15,35,40,10])
>>> s4=frozenset([40,50,10,15,20])
>>> sl&=s2&s3&s4
>>> si
set ( [ 10, 15])
```


difference_update(other, …)
Update the set, removing elements found in others.

```
>>> sl=frozenset([5,10,15])
>>> s2=[15, 20,25, 10]
>>> s3=set([30,15,35,40,10])
>>> s4=(40, 50, 10, 15,20)
>>> s3.difference_update(si, s2, s4)
>>> s3 set( [35, 30])
```

The operator based version of the the above method is set-=other |

```
>>> sl=frozenset([5,10,15])
>>> s2=frozenset([15,20,25,10])
>>> s3=set ( [30, 15, 35, 40, 10] )
>>> s4=frozenset([40,50,10,15,20])
>>> s3-=sl|s2|s4
>>> s3
set( [35, 30])
```

symmetric_difference_update(other)
Update the set, keeping only elements found in either set, but not in both.

```
>>> sl=set([5,10,15])
>>> s2=[15,20,25,10]
>>> si.symmetric_difference_update(s2)
>>> si
set( [25, 20, 5])
```

The operator based version of the the above method is set A=other.

```
>>> sl=set([5,10,15])
>>> s2=frozenset ([15,20,25, 10'] )
>>> slA=s2
>>> si
set([25, 20, 5])
```

add(elem)
The method adds element elem to the set.

```
>>> s=set([5,10,15,20])
>>> s.add(25)
>>> s
set ( [25, 10, 20, 5, 15])
```


remove(elem)
Remove element elem from the set. Raises KeyError, if elem is not contained in the set.

```
>>> s=set([5,10,15,20])
>>> s.remove(15)
>>> s
set( [10, 20, 5])
>>> s=set([5,10,15,20])
>>> s.remove(100)
Traceback (most recent call last): File "", line 1, in
KeyError: 100
```

discard(elem)
Remove element elem from the set if it is present. It is difference from remove () in a way that it does not raise KeyError if elem is not present in the set.

```
>>> s=set([5,10,15,20])
>>> s.discard(15)
>>> s set( [ 10, 20, 5])
>>> s.discard(100)
>>> s
set([10, 20, 5])
```

pop ()
Remove and return an arbitrary element from the set. Raises KeyErro^r, if the set is empty.

```
>>> s=set([5,10,15,20])
>>> s.pop()
10
>>> s
set ( [20, 5, 15])
>>> s.pop()
20
>>> s
set ( [5, 15])
```

clear()
Remove all elements from the set.

```
>>> s=set([5,10,15,20])
>>> s.clear()
>>> s
set ( [ ] )
```


12. Dictionary

Another useful mutable built-in type is “dictionary”. A dictionary as an unordered group of comma separated “key : value” pairs enclosed within braces, with the requirement that the keys are unique within a dictionary. The main operation on a dictionary is storing a value corresponding to a given key and extracting the value for that given key. Unlike sequences, which are indexed by a range of numbers, dictionary is indexed by key (key should be of immutable type, strings and numbers can always be keys). Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. List cannot be used as keys, since lists are of mutable type. Also, as mentioned previously, Python allow adding a trailing comma after last item of the dictionary.

```
>>> a={'sape':4139,'guido':4127,'jack':4098}
>>> a
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> a [ 'jack']
4098
>>> a={'sape':4139,'guido':4127,'jack':4098,}
>>> a
```

{'sape': 4139, 'jack': 4098, 'guido': 4127}

Dictionary creation

Dictionary can be created in many ways.

Using curly braces

Placing a comma-separated record of key-value pairs within the braces adds initial key-value pairs to the dictionary; this is also the way dictionaries are written as output.

```
>>> a={'sape':413 9, 'guido':4127, 'jack':4098}
>>> a
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

A pair of braces creates an empty dictionary.

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


Dictionary comprehension

Dictionary comprehension provides a concise way to create dictionary.

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

Using built-in function

Dictionary can also be created using built-in function diet (). Consider the following example using diet (), which returns the same dictionary { "one" : 1, "two" : 2, "three" : 3 }:

```
>>> a=dict(one=l,two=2,three=3)
>>> b={'one':1,'two':2,'three':3}
>>> c=dict(zip(['one'two', 'three'], [1,2,3]))
>>> d=dict([('two',2),('one',1),('three',3)])
>>> e=dict({'three':3,'one':1,'two':2})
>>> a==b==c==d==e
True
```

Accessing dictionary elements

To access a dictionary value, use key enclosed within square bracket. A KeyError exception is raised if the key is not present in the dictionary.

```
>>> a={'sape' :413 9, 'guido' :412 7, 'jack' :4098}
>>> a['guido']
4127
```

Updating dictionary elements

It is possible to add new item in a dictionary and can also change value for a given key.

```
>>> a={'sape':4139,'guido':4127,'jack':4098}
>>> a['mike']=2299 # Add new item
>>> a['guido']=1000 # Update existing item
>>> a \
{'sape': 4139, 'mike': 2299, 'jack': 4098, 'guido': 1000}
```

It is also possible to update a dictionary with another dictionary using update () method.

Deleting dictionary elements

To remove a dictionary item (key-value pair) or the entire dictionary, one can use the del statement. To remove all items (resulting an empty dictionary), clear () method can be used.

```
>>> a={'sape':4139,'guido':4127,'jack':4098}
>>> del a['guido']
>>> a
{'sape': 4139, 'jack': 4098}
>>> del a
>>> a
Traceback (most recent call last): File "", line 1, in
NameError: name 'a' is not defined
```

Membership operation

Dictionary support membership operation i.e. checking the existence of a key in the dictinary.

```
>>> a={'sape':4139,'guido':4127,'jack':4098}
>>> 'jack' in a
True
>>> 'tom' not in a
```

True
>>> 4127 in a
False

Looping techniques
When looping through dictionary, the key and corresponding value can be retrieved at the same time using the iteritems () method.

```
>>> a={'sape' :413 9, 'guido' :412 7, 'jack' :4 0 9 8 }
>>> for k,v in a.iteritems():
        print(k,v)
```
sape 4139 jack 4098- guido 4127

Dictionary methods
The following are some dictionary methods supported by Python.

diet.clear()
Removes all items from the dictionary.

```
>>> a={'Name':'Zara','Age':7}
>>> len(a)
2
>>> a.clear()
>>> len(a)
0
```

dict.copy()
Returns a copy of dictionary.

```
>>> dictl={'Name':'Zara','Age':7}
>>> dict2=dict1.copy()
>>> dict2
{'Age': 7, 'Name': 'Zara'} .
```

diet.fromkeys(seq[,value])
Create a new dictionary with keys from seq and values set to value (default as None).

```
>>> seq=('name','age','gender')
>>> a=dict.fromkeys(seq)
>>> a
{'gender': None, 'age': None, 'name': None}
>>> a=a.fromkeys(seq,10)
>>> a
```

{'gender': 10, 'age': 10, 'name': 10}

dict.get(key[,default])
Return the value for key, if key is in the dictionary, else default. If default is not given, the default is None.

```
>>> a={'Name':'Zara','Age':7}
>>> a.get('Age')
7
>>> a.get('Gender','Never')
'Never'
>>> print(a.get('Gender'))
None
```

dict.has_key(key)
Test for the presence of key in the dictionary. Returns True, if key is present in dictionary, otherwise False.


```
>>> a={'Name':'Zara','Age':7}
>>> a.has_key('Age')
True
>>> a.has_key('Gender')
False
```

diet.items()
Returns a list of dictionary's key-value tuple pairs.

```
>>> a= { ' Name ' : ' Zara ' , ' Age ' : 7 }
>>> a.items()
[('Age', 7), ('Name', 'Zara')]
```

diet.iteritems()
Returns an iterator over the dictionary's key-value pairs.

```
>>> a={ ' Gender ' : ' Female ' , 'Age ' : 7, ' Hair color ' :None, ' Name ' : ' Zara ' }
>>> for b in a.iteritems():
        print('{0} {1}'.format(b[0],b[1]))
Gender....Female
Age......7
Hair color.......None
Name......Zara
```

dict.iterkeys()

Returns an iterator over the dictionary's keys.

```
>>> a={'Gender':'Female','Age':7,'Hair color':None,'Name':'Zara'}
>>> for b in a.iterkeys():
        print(b)
Gender
Age
Hair color
Name
```

dict.itervalues()
Returns an iterator over the dictionary's values.

```
>>> a={'Gender':'Female','Age':7,'Hair color':None,'Name':'Zara'}
>>> for b in a.itervalues():
        print(b)
Female
7
None
Zara
```


dict.keys()
Returns list of dictionary keys.

```
>>> a={'Name':'Zara','Age':7}
>>> a.keys()
['Age', 'Name']
```

dict.pop(key[,default])
If key is in the dictionary, remove it and return its value, else return default. If default is not it given and key is not in the dictionary, a KeyError is raised.

```
>>> a={'Gender':'Female','Age':7,'Hair color':None,'Name':'Zara'}
>>> a.pop('Age',15)
7
>>> a.pop('Age',15)
15
>>> a
{'Gender': 'Female', 'Hair color': None, 'Name': 'Zara'}
```

dict.popitem() .
Remove and return an arbitrary key-value pair from the dictionary. If the dictionary is empty, calling popitem () raises an KeyError exception.

```
>>> a={'Gender':'Female','Age':7,'Hair color':None,'Name':'Zara'}
```

```
>>> a.popitem()
('Gender', 'Female')
>>> a.popitem()
('Age', 7)
>>> a
{'Hair color': None, 'Name': 'Zara'}
```

diet.setdefault(key[,default])
If key is in the dictionary, return its value. If not, insert key with a value of default and return default. The default defaults to None.

```
>>> a={'Name':'Zara','Age':7}
>>> a.setdefault('Age',15)
7
>>> a.setdefault('Gender','Female')
'Female'
>>> a.setdefault('Hair color')
>>> a
{'Gender': 'Female', 'Age': 7, 'Hair color': None, 'Name': 'Zara'}
```


dict.update([other])
Update the dictionary with the key-value pairs from other, overwriting existing keys, and returns None. The method accepts either another dictionary object or an iterable of key-value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key-value pairs.

```
>>> diet1={'Name':'Zara','Age':7}
>>> dict2={'Gender':'female'}
>>> diet1.update(dict2)
>>> dictl.update(hair_color='black',eye_color='blue')
>>> dictl
{'eye_color': 'blue', 'Gender': 'female', 'Age': 7, 'Name': 'Zara', 'hair color': 'black'}
```

dict.values()
Return list of dictionary values.

```
>>> a={'Name':'Zara','Age':7}
>>> a.values()
[7, 'Zara']
```

13. FUNCTION

Function is a compound statement which returns some value to the caller. The keyword def introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented. The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or "docstring". Docstring is a string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the doc attribute of the enclosing class, function or module.

Example

```
>>> def summation(a,b):
        . """ Gives sum of two numbers""" # docstring
        sum = a+b
        return sum
>>> summation(5,10)
#Output : 15
```

FUNCTION SCOPE

1. Global Variables
2. Local Variables

1. Global Variables

A variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

2. Local Variables

A variable that is defined in a block is available in that block only. It is not accessible outside the block.

Example

```
Total = 0
def sum(arg1, arg2):
        Total = arg1 + arg2
        print("Inside the function local total :", Total)
        return Total
sum(10, 20)
print("Outside the function global total :", Total)
```

#Output

Inside the function local total : 30
Outside the function global total : 0

PASS BY VALUE V/S REFERENCE

In python, **basic variables** are **passed by value** whereas **structured variables** like **list, tuple, dictionary, set** are **passed by reference**. In other words, changes made to local variables will not be seen in main in pass by value. Changes made in function will be seen in main in pass by reference.

Pass by value

```
def plus_1(x) :
    print("In function before change", x)
    x=x+1
    print("In function after change", x)

x=5
print("In main before change", x)
plus_1(x)
print("In main after change", x)
```

#Output
In main before change 5
In function before change 5
In function after change 6
In main after change 5

Pass by reference

```
def change_value(x) :
    print("In function before change", x)
    x[3]=300
    print("In function after change", x)

x=[1,2,3,4,5]
print("In main before change", x)
change_value(x)
print("In main after change", x)
```

#Output
In main before change [1, 2, 3, 4, 5]
In function before change [1, 2, 3, 4, 5]
In function after change [1, 2, 3, 300, 5]

In main after change [1, 2, 3, 300, 5]

Argument

Argument is the value passed to a function (or method) while calling the function. There are two types of arguments:

1. **Keyword argument**
2. **Positional argument**
3. **Default argument**

1. Keyword argument

When we call a function with some values, these values get assigned to the arguments according to their position.

Example

```
>>> def summation(aa,bb,cc,dd):
        print('aa:',aa)
        print('bb: ',bb)
        print('cc:',cc)
        print('dd:',dd)
        total = aa+bb+cc+dd
        return total
>>> sumup1 = summation(bb=20,dd=40,aa=10, cc=30)
#Output
aa: 10
bb: 20
cc: 30
dd: 40
>>> print('Sum is:',sumup1)
#Output : Sum is: 100
```

2. Positional argument

Positional arguments are arguments that need to be included in the proper position or order.

Example

```
>>> def summation(aa,bb,cc,dd):
```

```
        print('aa:' , aa)
        print('bb:',bb)
        print(' cc : ' , cc)
        print('dd:',dd)
        total = aa+bb+cc+dd
        return total
>>> sumup3 = summation(10,20,30,40)
#Output
aa:10
bb:20
cc:30
dd:40
>>> print('Sum is:, sumup3)
#Output : Sum is: 100
```

3. Default argument

Python has a different way of representing syntax and default values for function arguments. Default values indicate that the function argument will take that value if no argument value is passed during function call.

Example

```
def division(x, y=1):
        return x//y

print("Divide:", division(20, 10))

print("Divide:", division(20))

#Output

Divide: 2

Divide: 20
```

ANONYMOUS FUNCTION

An **anonymous function** is a **function** that is defined without a name. While normal **functions** are defined using the **def** keyword in **Python**, **anonymous functions** are defined using the **lambda** keyword. Hence, **anonymous functions** are also called **lambda functions.**

SYNTAX

lambda arguments : expression

Example 1

```
add = lambda x,y: x + y
print("Addition:", add(10, 20))
print("Addition:", add("he", "llo"))
print("Addition:", add(1.22, 4.66))
```

#Output
Addition: 30
Addition: hello
Addition: 5.88

Example 2

```
add = lambda x,y: x + y
sub = lambda x,y: x - y
mul = lambda x,y: x * y
div = lambda x,y: x // y
print("Addition:", add(20, 10))
print("Subtraction:", sub(20, 10))
print("Multiplication:", mul(20, 10))
print("Divide:", div(20, 10))
```

#Output
Addition: 30
Subtraction: 10
Multiplication: 200
Divide: 2