

Unit –II Python Basics, Syntax and Style

Variable Assignment, Identifiers.

Numbers: Introduction to Numbers, Integers, Floating Point Real Numbers, Complex Numbers, Operators, Built-in Functions.

Strings: Strings and Operators, String-only Operators, Built-in Functions, String Built-in Methods, Special Features of Strings.

Lists: Operators, Built-in Functions, List Type Built-in Methods, Special Features of Lists.

Tuples: Tuple Operators and Built-in Functions, Special Features of Tuples

Variable Assignment:

Assignment Operator:

The equal sign (=) is the main Python assignment operator. (The others are augmented assignment operator.)

```
anInt = -12
aString = 'cart'
aFloat = -3.1415 * (5.0 ** 2)
anotherString = 'shop' + 'ping'
aList = [3.14e10, '2nd elmt of a list', 8.82-4.371j]
```

Chaining together assignments is:

```
>>> y = x = x + 1
>>> x, y
(2, 2)
```

Augmented Assignment:

the equal sign can be combined with an arithmetic operation and the resulting value reassigned to the existing variable. Known as augmented assignment:

```
x = x + 1
```

... can now be written as ...

```
x += 1
```

Multiple Assignment:

```
>>> x = y = z = 1
>>> x
1
>>> y
1
>>> z
1
```

"Multiple" Assignment:

```
>>> x, y, z = 1, 2, 'a string'
>>> x
1
>>> y
2
>>> z
'a string'
```

Identifiers:

Identifiers are the set of valid strings that are allowed as names in a computer language. names that form a construct of the language. Such identifiers are reserved words that may not be used for any other purpose, or else a syntax error (SyntaxError exception) will occur.

Valid Python Identifiers:

The rules for Python identifier strings are like most other high-level programming languages that come from the C world:

- First character must be a letter or underscore (_)
- Any additional characters can be alphanumeric or underscore
- Case-sensitive

No identifiers can begin with a number, and no symbols other than the underscore are ever allowed. The easiest way to deal with underscores is to consider them as alphabetic characters. Case-sensitivity means that identifier foo is different from Foo, and both of those are different from FOO.

Introduction to Numbers:

Numbers provide literal or scalar storage and direct access. A number is also an **immutable** type, meaning that changing or updating its value results in a newly allocated object

Python has several numeric types: "plain" integers, long integers, Boolean, double-precision floating point real numbers, decimal floating point numbers, and complex numbers.

Creating Assigning Numbers:

Creating numbers is as simple as assigning a value to a variable:

```
anInt = 1
aLong = -9999999999999999L
aFloat = 3.1415926535897932384626433832795
aComplex = 1.23+4.56J
```

Updating Numbers:

You can "update" an existing number by (re)assigning a variable to another number. The new value can be related to its previous value or to a completely different number altogether. We put quotes around update because you are not really changing the value of the original variable. Because numbers are immutable, you are just making a new number and reassigning the reference.

In Python, variables act like pointers that point to boxes. For immutable types, you do not change the contents of the box, you just point your pointer at a new box. Every time you assign another number to a variable, you are creating a new object and assigning it. (This is true for all immutable types, not just numbers.)

```
anInt += 1
aFloat = 2.718281828
```

Removing Numbers:

Under normal circumstances, you do not really "remove" a number; you just stop using it! If you really want to delete a reference to a number object, just use the del statement (introduced in Section 3.5.6). You can no longer use the variable name, once removed, unless you assign it to a new object; otherwise, you will cause a NameError exception to occur.

```
del anInt
del aLong, aFloat, aComplex
```

Integers:

Python has several types of integers. There is the Boolean type with two possible values. There are the regular or plain integers: generic vanilla integers recognized on most systems today. Python also has a long integer size; however, these far exceed the size provided by C longs.

Boolean: Objects of this type have two possible values, Boolean TRue and False.

Standard (Regular or Plain) Integers:

Python's "plain" integers are the universal numeric type. Most machines (32-bit) running Python will provide a range of -231 to 231-1, that is -2, 147,483,648 to 2,147,483,647. If Python is compiled on a 64-bit system with a 64-bit compiler, then the integers for that system will be 64-bit. Here are some examples of Python integers:

```
0101 84 -237 0x80 017 -680 -0X92
```

Python integers are implemented as (signed) longs in C.

Long Integers:

Longs are a superset of integers and are useful when your application requires integers that exceed the range of plain integers, meaning less than -231 or greater than 231-1. Use of longs is denoted by the letter "L", uppercase (L) or lowercase (l), appended to the integer's numeric value. Values can be expressed in decimal, octal, or hexadecimal. The following are examples of longs:

```
16384L -0x4E8L 017L -2147483648l 052144364L
```

Double Precision Floating Point Numbers:

Floats in Python are implemented as C doubles, double precision floating point real numbers, values that can be represented in straightforward decimal or scientific notations. These 8-byte (64-bit) values conform to the IEEE 754 definition (52M/11E/1S) where 52 bits are allocated to the mantissa, 11 bits to the exponent (this gives you about ± 10308.25 in range), and the final bit to the sign. That all sounds fine and dandy; however, the actual degree of precision you will receive (along with the range and overflow handling) depends completely on the architecture of the machine as well as the implementation of the compiler that built your Python interpreter.

Floating point values are denoted by a decimal point (.) in the appropriate place and an optional "e" suffix representing scientific notation. We can use either lowercase (e) or uppercase (E). Positive (+) or negative (-) signs between the "e" and the exponent indicate the sign of the exponent. Absence of such a sign indicates a positive exponent. Here are some floating point values:

```
0.0 -777. 1.6 -5.555567119 96e3 * 1.0
4.3e25 9.384e-23 -2.172818 float (12) 1.000000001
3.1416 4.2E-10 -90. 6.022e23 -1.609E-19
```

Complex Numbers:

Not only real numbers, Python can also handle complex numbers and its associated functions using the file "cmath". Complex numbers have their uses in many applications related to mathematics and python provides useful tools to handle and manipulate them.

Converting real numbers to complex number

An complex number is represented by " $x + yi$ ". Python converts the real numbers x and y into complex using the function **complex(x,y)**. The real part can be accessed using the function **real()** and imaginary part can be represented by **imag()**.

The following are examples of complex numbers:

```
64.375+1j 4.23-8.5j 0.23-8.55j 1.23e-045+6.7e+089j
6.23+1.5j -1.23-875j 0+1j 9.80665-8.31441j -.0224+0j
```

Example:

```
>>> aComplex = -8.333-1.47j
>>> aComplex
(-8.333-1.47j)
```

```
# Python code to demonstrate the working of
# complex(), real() and imag()

# importing "cmath" for complex number operations
import cmath

# Initializing real numbers
x = 5
y = 3

# converting x and y into complex number
z = complex(x,y);

# printing real and imaginary part of complex number
print ("The real part of complex number is : ",end="")
```

```
print (z.real)

print ("The imaginary part of complex number is : ",end="")
print (z.imag)
```

Operators:

Numeric types support a wide variety of operators, ranging from the standard type of operators to operators created specifically for numbers, and even some that apply to integer types only.

Standard Type Operators:

Here are some examples of the standard type operators in action with numbers:

```
>>> 5.2 == 5.2
True
>>> -719 >= 833
False
>>> 5+4e >= 2-3e
True
>>> 2 < 5 < 9 # same as ( 2 < 5 )and ( 5 < 9 )
True
>>> 77 > 66 == 66 # same as ( 77 > 66 )and ( 66 == 66 )
True
>>> 0. < -90.4 < 55.3e2 != 3 < 181
False
>>> (-1 < 1) or (1 < -1)
True
```

Numeric Type (Arithmetic) Operators

Python supports unary operators for no change and negation, + and -, respectively; and binary arithmetic operators +, -, *, /, %, and **, for addition, subtraction, multiplication, division, modulo, and exponentiation, respectively. In addition, there is a new division operator, //.

Division

Those of you coming from the C world are intimately familiar with *classic division* that is, for integer operands, *floor division* is performed, while for floating point numbers, real or *true division* is the operation. However, for those who are learning programming for the first time, or for those who rely on accurate calculations, code must be tweaked in a way to obtain the desired results. This includes casting or converting all values to floats before performing the division.

The decision has been made to change the division operator in some future version of Python from classic to true division and add another operator to perform floor division. We now summarize the various division types and show you what Python currently does, and what it will do in the future.

Classic Division

When presented with integer operands, classic division truncates the fraction, returning an integer (floor division). Given a pair of floating-point operands, it returns the actual floating-point quotient (true division). This functionality is standard among many programming languages, including Python.

Example:

```
>>> 1 / 2 # perform integer result (floor)
0
>>> 1.0 / 2.0 # returns actual quotient
0.5
```

True Division

This is where division always returns the actual quotient, regardless of the type of the operands. In a future version of Python, this will be the algorithm of the division operator. For now, to take advantage of true division, one must give the **from __future__ import** division directive. Once that happens, the division operator (/) performs only true division:

```
>>> from __future__ import division
>>>
>>> 1 / 2 # returns real quotient
0.5
>>> 1.0 / 2.0 # returns real quotient
```

0.5

Floor Division

A new division operator (`//`) has been created that carries out floor division: it always truncates the fraction and rounds it to the next smallest whole number toward the left on the number line, regardless of the operands' numeric types. This operator works starting in 2.2 and does not require the `__future__` directive above.

```
>>> 1 // 2 # floors result, returns integer
```

```
0
```

```
>>> 1.0 // 2.0 # floors result, returns float
```

```
0.0
```

```
>>> -1 // 2 # move left on number line
```

```
-1
```

There were strong arguments for as well as against this change, with the former from those who want or need true division versus those who either do not want to change their code or feel that altering the division operation from classic division is wrong.

Exponentiation

The exponentiation operator has a peculiar precedence rule in its relationship with the unary operators: it binds more tightly than unary operators to its left, but less tightly than unary operators to its right.

Due to this characteristic, you will find the `**` operator twice in the numeric operator charts in this text.

Here are some examples:

```
>>> 3 ** 2
```

```
9
```

```
>>> -3 ** 2 # ** binds tighter than - to its left
```

```
-9
```

```
>>> (-3) ** 2 # group to cause - to bind first
```

```
9
```

```
>>> 4.0 ** -1.0 # ** binds looser than - to its right
```

0.25

Numeric Type Arithmetic Operators:

Arithmetic Operator Function

<code>expr1 ** expr2</code>	<code>expr1</code> raised to the power of <code>expr2</code> [a]
<code>+expr</code>	(unary) <code>expr</code> sign unchanged
<code>-expr</code>	(unary) negation of <code>expr</code>
<code>expr1 ** expr2</code>	<code>expr1</code> raised to the power of <code>expr2</code> [a]
<code>expr1 * expr2</code>	<code>expr1</code> times <code>expr2</code>
<code>expr1 / expr2</code>	<code>expr1</code> divided by <code>expr2</code> (classic or true division)
<code>expr1 // expr2</code>	<code>expr1</code> divided by <code>expr2</code> (floor division [only])
<code>expr1 % expr2</code>	<code>expr1</code> modulo <code>expr2</code>
<code>expr1 + expr2</code>	<code>expr1</code> plus <code>expr2</code>
<code>expr1 - expr2</code>	<code>expr1</code> minus <code>expr2</code>

Here are a few more examples of Python's numeric operators:

```
>>> -442 - 77
```

```
-519
```

```
>>>
```

```
>>> 4 ** 3
```

```
64
```

```
>>>
```

```
>>> 4.2 ** 3.2
```

```
98.7183139527
```

```
>>> 8 / 3
```

```
2
```

```
>>> 8.0 / 3.0
```

```
2.66666666667
```

```
>>> 8 % 3
```

```
2
```

```

>>> (60. - 32.) * ( 5. / 9. )
15.5555555556
>>> 14 * 0x04
56
>>> 0170 / 4
30
>>> 0x80 + 0777
639
>>> 45L * 22L
990L
>>> 16399L + 0xA94E8L
709879L
>>> -2147483648L - 52147483648L
-54294967296L
>>> 64.375+1j + 4.23-8.5j
(68.605-7.5j)
>>> 0+1j ** 2 # same as 0+(1j**2)
(-1+0j)
>>> 1+1j ** 2 # same as 1+(1j**2)
0j
>>> (1+1j) ** 2
2j

```

Built-in and Factory Functions:

Standard Type Functions

In the last chapter, we introduced the `cmp()`, `str()`, and `type()` built-in functions that apply for all standard types. For numbers, these functions will compare two numbers, convert numbers into strings, and tell you a number's type, respectively. Here are some examples of using these functions:

```

>>> cmp(-6, 2)
-1
>>> cmp(-4.333333, -2.718281828)
-1
>>> cmp(0xFF, 255)
0
>>> str(0xFF)
'255'
>>> str(55.3e2)
'5530.0'
>>> type(0xFF)
<type 'int'>
>>> type(98765432109876543210L)
<type 'long'>
>>> type(2-1j)
<type 'complex'>

```

Numeric Type Functions

Python currently supports different sets of built-in functions for numeric types. Some convert from one numeric type to another while others are more operational, performing some type of calculation on their numeric arguments.

Conversion Factory Functions

The `int()`, `long()`, `float()`, and `complex()` functions are used to convert from any numeric type to another. Starting in Python 1.5, these functions will also take strings and return the numerical value represented by the string. Beginning in 1.6, `int()` and `long()` accepted a base parameter (see below) for proper string conversions; it does not work for numeric type conversion.

The following are some examples of using these functions:

```

>>> int(4.25555)
4

```

```
>>> long(42)
42L
>>> float(4)
4.0
>>> complex(4)
(4+0j)
>>>
>>> complex(2.4, -8)
(2.4-8j)
>>>
>>> complex(2.3e-10, 45.3e4)
(2.3e-10+453000j)
```

Strings:

Python string is one of the most popular types in Python. By definition, a string is a sequence of characters. You can create a string simply by enclosing characters in single or double quotes. If a string spans multiple lines, you can use triple single or double quotes to enclose characters.

Let's look at the following example:

```
str1 = "This is a string in Python"
str2 = 'This is another string in Python'
str3 = """ another string in Python with triple single quotes
that span multiple lines"""
str4 = """ another string in Python with triple double quotes
that span multiple lines"""
print(str1,str3,str3,str4)
```

In this example, we defined four strings.

Notice that a string must begin and ends with the same type of quotes.

Accessing string elements:

Because a string is a sequence of character, you can access its elements using index or slice notation. The following example demonstrates accessing elements using index and slice notation:

```
s = "Python String"
print(s[0]) # P
print(s[-1]) # g
print(s[0:6]) # Python
```

In this example, we defined a string `s`. Then we accessed element of string `s` via index using square `[]` brackets: `s[0]` and `s[-1]`. If you use negative index, Python will count from the end of string starting `-1`, `-2` and so forth. We also accessed a substring of `s` using slice notation `s[0:6]`.

0	1	2	3	4	5	6	7	8	9	10	11	12
P	y	t	h	o	n		S	t	r	i	n	g
-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Because string is immutable, the following assignment will issue an error:

```
s[0] = 'c'
```

Updating a string:

Update a string requires creating a new string. You can “update” an existing string by assigning or reassigning a new string to the old one. A new string may or may not relate to the old one. Take a look at the following example:

```
s = "Python String"
```

```
s1 = s[0:6]
```

```
s2 = s[7:13]
```

```
print(s1)
```

```
print(s2)
```

```
s = s[0]
```

```
print(s)
```

In this example, we assigned substring of `s` to new string variables `s1` and `s2`. We also assigned string `s` to its substring. It is important to note that `s` is pointing to a new string not the old one.

Python string operators

You can apply many operators on strings. The following example demonstrates how to use operators on strings:

```
s1 = "Python"
```

```
s2 = "String"
```

```
s3 = s1
```

```
print(s1 != s2) # True
```

```
print(s1 > s3) # False
```

The `in` and `not in` operators

The `in` and `not in` operators allow you to check if a string appears in another string. See the following example:

```
s1 = "python"
s2 = "python rock!"
s3 = "roll"
print(s1 in s2) # True
print(s3 not in s2) # True
print(s3 in s2) # False
```

Python string concatenating operator `+`

You can use operator `+` to create a new string from existing ones. The following example demonstrates the string concatenation:

```
s1 = 'String'
s2 = ' in Python'
s3 = s1 + s2
print(s3)
```

It is not recommended to use operator `+` for string concatenation because of the performance. Each time you use operator `+`, Python has to allocate memory for all string involved, including the result string.

Built-in Functions:

Python has several built-in functions associated with the [string data type](#). These functions let us easily modify and manipulate strings. We can think of functions as being actions that we perform on elements of our code. Built-in functions are those that are defined in the Python programming language and are readily available for us to use.

Making Strings Upper and Lower Case

The functions `str.upper()` and `str.lower()` will return a string with all the letters of an original string converted to upper- or lower-case letters. Because strings are immutable data types, the returned string will be a new string. Any characters in the string that are not letters will not be changed.

Let's convert the string Sammy Shark to be all upper case:

```
ss = "rsml"
print(ss.upper())
Output
RSML
```

Now, let's convert the string to be all lower case:

```
print(ss.lower())
```

Output

```
sammy shark
```

The `str.upper()` and `str.lower()` functions make it easier to evaluate and compare strings by making case consistent throughout. That way if a user writes their name all lower case, we can still determine whether their name is in our database by checking it against an all upper-case version, for example.

Boolean Methods

Python has some string methods that will evaluate to a Boolean value. These methods are useful when we are creating forms for users to fill in, for example. If we are asking for a post code we will only want to accept a numeric string, but when we are asking for a name, we will only want to accept an alphabetic string.

There are a number of string methods that will return Boolean values:

Method	True if
<code>str.isalnum()</code>	String consists of only alphanumeric characters (no symbols)
<code>str.isalpha()</code>	String consists of only alphabetic characters (no symbols)
<code>str.islower()</code>	String's alphabetic characters are all lower case
<code>str.isnumeric()</code>	String consists of only numeric characters
<code>str.isspace()</code>	String consists of only whitespace characters
<code>str.istitle()</code>	String is in title case
<code>str.isupper()</code>	String's alphabetic characters are all upper case

Let's look at a couple of these in action:

```
number = "5"
```

```
letters = "abcdef"
```

```
print(number.isnumeric())
```

```
print(letters.isnumeric())
```

Output

True

False

Using the `str.isnumeric()` method on the string `5` returns a value of `True`, while using the same method on the string `abcdef` returns a value of `False`.

Similarly, we can query whether a string's alphabetic characters are in title case, upper case, or lower case. Let's create a few strings:

```
movie = "2001: A SAMMY ODYSSEY"
```

```
book = "A Thousand Splendid Sharks"
```

```
poem = "sammy lived in a pretty how town"
```

Now let's try the Boolean methods that check for case:

```
print(movie.islower())
```

```
print(movie.isupper())
```

```
print(book.istitle())
```

```
print(book.isupper())
```

```
print(poem.istitle())
```

```
print(poem.islower())
```

Now we can run these small programs and see the output:

Output of movie string

False

True

Output of book string

True

False

Output of poem string

False

True

Checking whether characters are lower case, upper case, or title case, can help us to sort our data appropriately, as well as provide us with the opportunity to standardize data we collect by checking and then modifying strings as needed.

Boolean string methods are useful when we want to check whether something a user enters fits within given parameters.

Determining String Length

The string function `len()` returns the number of characters in a string. This method is useful for when you need to enforce minimum or maximum password lengths, for example, or to truncate larger strings to be within certain limits for use as abbreviations.

To demonstrate this method, we'll find the length of a sentence-long string:

```
open_source = "Sammy contributes to open source."  
print(len(open_source))
```

Output

33

We set the variable `open_source` equal to the string "Sammy contributes to open source." and then we passed that variable to the `len()` function with `len(open_source)`. We then passed the method into the `print()` method so that we could see the output on the screen from our program.

Keep in mind that any character bound by single or double quotation marks — including letters, numbers, whitespace characters, and symbols — will be counted by the `len()` function.

join(), split(), and replace() Methods

The `str.join()`, `str.split()`, and `str.replace()` methods are a few additional ways to manipulate strings in Python.

The `str.join()` method will concatenate two strings, but in a way that passes one string through another.

Let's create a string:

```
balloon = "Sammy has a balloon."
```

Now, let's use the `str.join()` method to add whitespace to that string, which we can do like so:

```
" ".join(balloon)
```

If we print this out:

```
print(" ".join(balloon))
```

We will see that in the new string that is returned there is added space throughout the first string:

Output

```
S a m m y   h a s   a   b a l l o o n .
```

We can also use the `str.join()` method to return a string that is a reversal from the original string:

```
print("".join(reversed(balloon)))
```

Output

```
.noollab a sah ymmaS
```

We did not want to add any part of another string to the first string, so we kept the quotation marks touching with no space in between.

The `str.join()` method is also useful to combine a list of strings into a new single string.

Let's create a comma-separated string from a list of strings:

```
print(", ".join(["sharks", "crustaceans", "plankton"]))
```

Ouput

```
sharks,crustaceans,plankton
```

If we want to add a comma and a space between string values in our new string, we can simply rewrite our expression with a whitespace after the comma: ", ".join(["sharks", "crustaceans", "plankton"]).

Just as we can join strings together, we can also split strings up. To do this, we will use the `str.split()` method:

```
print(balloon.split())
```

Ouput

```
['Sammy', 'has', 'a', 'balloon.']
```

The `str.split()` method returns a list of strings that are separated by whitespace if no other parameter is given.

We can also use `str.split()` to remove certain parts of an original string. For example, let's remove the letter a from the string:

```
print(balloon.split("a"))
```

Ouput

```
['S', 'mmy h', 's', ' b', 'lloon.']
```

Now the letter a has been removed and the strings have been separated where each instance of the letter a had been, with whitespace retained.

The `str.replace()` method can take an original string and return an updated string with some replacement.

Let's say that the balloon that Sammy had is lost. Since Sammy no longer has this balloon, we will change the substring "has" from the original string balloon to "had" in a new string:

```
print(balloon.replace("has", "had"))
```

Within the parentheses, the first substring is what we want to be replaced, and the second substring is what we are replacing that first substring with. Our output will look like this:

Ouput

```
Sammy had a balloon.
```

Using the string methods `str.join()`, `str.split()`, and `str.replace()` will provide you with greater control to manipulate strings in Python.

Python string built-in methods:

The following table illustrates the most important string's built-in methods:

Method	Description
--------	-------------

<code>string.capitalize()</code>	Converts the first character of a string into uppercase.
<code>string.center(width)</code>	Center justifies the original string into a given width size, padded with space for the rest.
<code>string.count(str, beg=0, end=len(string))</code>	Returns number of times that <code>str</code> occurs in <code>string</code> with index starts at <code>begs</code> position and ends at <code>end</code> position. The default value of <code>end</code> position is the last position of the string. The default value of the <code>beg</code> position is the first position of the string.
<code>string.endswith(str, beg=0, end=len(string))</code>	Check the end of a string for a match
<code>string.expandtabs(tabsize=8)</code>	Convert all tabs in string to multiple spaces, if <code>tabsize</code> is not passed, default to 8 spaces per tab.
<code>string.find(str, b=0 e=len(string))</code>	Checks if <code>str</code> appears in <code>string</code> or in substring of <code>string</code> specified by starting index <code>b</code> and ending index <code>e</code> . Return position of the first character of the first instance of <code>string str</code> in <code>string</code> , otherwise return -1 if <code>str</code> not found in <code>string</code> .
<code>string.format(*args, **kwargs)</code>	Format string based on given input parameters
<code>string.index(str, b=0, e=len(string))</code>	Similar to <code>find()</code> , but throw an exception if <code>str</code> not found
<code>string.isalnum()</code>	Returns True if <code>string</code> contains only alphabetic and/or numeric characters and <code>string</code> has at least 1 character, otherwise return False
<code>string.isalpha()</code>	Returns True if <code>string</code> has at least 1 character and all characters are alphabetic, and otherwise returns False
<code>string.isdecimal()</code>	Returns True if <code>string</code> has only decimal digits, and otherwise returns False
<code>string.isdigit()</code>	Returns True if <code>string</code> has only digits, and otherwise returns False
<code>string.islower()</code>	Returns True if at least 1 cased character exists and all characters in <code>string</code> are in lowercase, otherwise returns False
<code>string.isnumeric()</code>	Returns True if <code>string</code> has only numeric characters, otherwise returns False
<code>string.isspace()</code>	Returns True if <code>string</code> contains only whitespace characters, and otherwise returns False
<code>string.istitle()</code>	Returns True if the first character of each word in a <code>string</code> are in uppercase, otherwise returns False.
<code>string.isupper()</code>	Returns True if at least 1 cased character exists and all characters in <code>string</code>

	are in uppercase, otherwise returns False
<code>string.join()</code>	Concatenate a list of string to form a single string with the original string between each element.
<code>string.ljust(width)</code>	Left justifies the original string into a given width size, padded with space for the rest.
<code>string.lower()</code>	Converts all characters in the string into lowercase.
<code>string.lstrip()</code>	Returns original string except any whitespace at the beginning of the string has been removed
<code>string.replace(str1, str2, num=string.count(str1))</code>	Replaces <i>num</i> occurrences of str1 in string with str2. If num is not passed, replace all occurrences.
<code>string.rfind(str, beg=0, end=len(string))</code>	Similar to find(), but its search starts from the end of the string
<code>string.rindex(str, b=0, e=len(string))</code>	Similar to index(), but its search starts from the end of the string
<code>string.rjust(width)</code>	Right justifies the original string into a given width size, padded with space for the rest.
<code>string.rstrip()</code>	Returns the original string except any whitespace at the end of string has been removed
<code>string.split(str="", num=string.count(str))</code>	returns a list of substrings in the string
<code>string.splitlines(num=string.count('\n'))</code>	Returns a list of single line strings from the original multiple line string.
<code>string.startswith(str, b=0, e=len(string))</code>	Check the beginning of a string for a match
<code>string.strip([obj])</code>	Returns original string except any whitespace at the beginning or end of the string has been removed
<code>string.swapcase()</code>	For all characters of string, convert uppercase to lowercase and vice versa.
<code>string.title()</code>	Capitalizes the first character of each word in a string
<code>string.upper()</code>	Converts all characters of a string into uppercase.

Python string features:

Python string object has the following features:

- String is scalar type, meaning that the Python interpreter treats string as a single value not a container that holds other Python objects.
- String is immutable, for efficiency reason. It means string is not modifiable. If you change an element inside a string, a new string is created.
- String consists of individual characters so you can access sequentially using slicing.

Defining a Python list:

In Python, a list is an ordered collection of objects. A list can contain different types of objects, even other lists. A list is enclosed by [] brackets, where each element is separated by a comma.

For example, you can define a list of integers as follows:

```
list = [1,3,2,7,9,4]
```

In Python, the size of the list can grow or shrink when needed.

Example:

```
list1 = ['Python', 'list', 'is', 'rock'] # list of string
```

```
list2 = ['one',2,3,'four'] # mixed between integer and string
```

```
list3 = ['list',list1] # contain another list
```

```
list4 = [1,2,['one','two']] # contain another list
```

The list1 is a list of strings

The list2 is a list of strings and integers. List can have elements that have different data types

The list3 contains a string and another list. The list3 is called a nested list.

The list4 contains 2 integers and another list. The list4 is also a nested list.

The following code defines an empty list:

```
empty_list = []
```

Python list index

You can access elements of a list via list index. List index starts from zero. For example, you can access elements of a list as follows:

```
list = [1,2,'three','four']
```



```
print(list[0]) # 1
```

```
print(list[3]) # four
```

List slicing:

List slicing basically is another way to access list elements. You can access a sequential subset of list elements using the slice. The slice syntax as follows:

The operator (:) is used between start and end values. Both of these values are optional.

```
List[start:end]
```

If you omit the start value, it returns all elements from beginning of the list to the element of the end except the last element with the end index.

If you omit the end value, it returns all elements starting from start index to the end element.

If you omit both values, you get the all elements of the list.

The following example shows you how to use list slicing:

```
list = [0,1,2,3,4,5,6,7,8,9]
```

```
print(list[:3]) # [0, 1, 2]
```

```
print(list[5:]) # [5, 6, 7, 8, 9]
```

```
print(list[:]) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Modifying list elements

To modify list element, you access it via index or list slicing, and change its value, for example:

```
list = [0,'modify','a','list']
```

```
list[0] = 'how to'
```

```
print(list) # ['how to', 'modify', 'a', 'list']
```

Built-in Functions and Build-in Method:

Appending the lists:

Python supports two functions that allow you to append a list to another list.

Function	Meaning
Append	to append a list as an element of another list
Extend	to append all element of a list to another list

The following example demonstrates how to use the `append()` and `extend()` functions:

```
a = [1,2,3]
b = [4,5,6]
a.append(b)
print(a) #[1, 2, 3, [4, 5, 6]]
```

```
a = [1,2,3]
a.extend(b)
print(a) #[1, 2, 3, 4, 5, 6]
```

Besides the `extend()` function, you can use concatenate operator (+) to append all elements of a list to another. See the following example:

```
x = ['one','two','three']
y = ['four','five']
```

```
x = x + y
print(x) #['one', 'two', 'three', 'four', 'five']
```

Sorting the list:

You can sort a list by using the built-in function: `sort()`. The following example shows you how to use the `sort()` function to sort a list of integers:

```
# sort list by using sort function
```

```
list = [1,5,8,4,2,9]
list.sort()
```

```
print(list) #[1, 2, 4, 5, 8, 9]
```

Python allows you to define custom sorting rule by using a custom key function. Let's take a look at an example:

```
# sort list by using sort function with custom key function
def compare(str):
    return len(str)
list = ['A','B','AA','BB','C']
```

```
# sort list without using key function
```

```
list.sort()
```

```
print(list) #['A', 'AA', 'B', 'BB', 'C']
```

```
# sort list without using custom key function
```

```
list.sort(key=compare)
```

```
print(list) #['A', 'B', 'C', 'AA', 'BB']
```

Let's examine the code above in more detail:

First, we defined a simple function that returns the number of characters in a string.

Second, we sorted the list by using default key function. The list is sorted alphabetically in ascending order.

Third, we pass the custom function the key parameter of the sort function. The sort function uses the custom key function for comparing objects when sorting them.

Python list supports many other useful operations listed in the table below. You can practice with each of them to see how it work.

Operations	Meaning	Examples	Result
*	Use to replicated the list	a = ['A'] * 4	a = ['A', 'A', 'A', 'A']
min	Returns the smallest element in a list	list = [1,2,6] min(list)	1
max	Returns the largest element in a list	list = [1,2,6] max(list)	6
index	Returns the position of a value in a list	list = [1,2,6] list.index(2)	1
count	Counts the number of times a value occurs in a list	list=[1,2,2,3,2] list.count(2)	3
in	Returns whether an item is in a list	list = [1,2,6] a = 2 b = a in list print(b)	true

Special Features of list:

Creating Other Data Structures Using Lists. Because of their container and mutable features, lists are fairly flexible and it is not very difficult to build other kinds of data structures using lists. Two that we can come up with rather quickly are stacks and queues.

Stack:

Stack works on the principle of “Last-in, first-out”. Also, the inbuilt functions in Python make the code short and simple. To add an item to the top of the list, i.e., to push an item, we use **append()** function and to pop out an element we use **pop()** function. These functions work quiet efficiently and fast in end operations.

```
# Python code to demonstrate Implementing stack using list
```

```
stack = ["Amar", "Akbar", "Anthony"]
```

```
stack.append("Ram")
```

```
stack.append("Iqbal")
```

```
print(stack)
```

```
print(stack.pop())
```

```
print(stack)
```

```
print(stack.pop())
```

```
print(stack)
```

queue:

Implementing queue is a bit different. Queue works on the principle of “First-in, first-out”. Time plays an important factor here. We saw that during the implementation of stack we used `append()` and `pop()` function which was efficient and fast because we inserted and popped elements from the end of the list, but in queue when insertion and pops are made from the beginning of the list, it is slow. This occurs due to the properties of list, which is fast at the end operations but slow at the beginning operations, as all other elements have to be shifted one by one. So, we prefer the use of **collections.deque** over list, which was specially designed to have fast appends and pops from both the front and back end.

```
# Python code to demonstrate Implementing
```

```
# Queue using deque and list
```

```
from collections import deque
```

```
queue = deque(["Ram", "Tarun", "Asif", "John"])
```

```
print(queue)
```

```
queue.append("Akbar")
```

```
print(queue)
```

```
queue.append("Birbal")
```

```
print(queue)
```

```
print(queue.popleft())
```

```
print(queue.popleft())
```

```
print(queue)
```

Tuples: Tuple Operators and Built-in Functions, Special Features of Tuples

Tuples:

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use () parentheses, whereas lists use square brackets.

Creating Tuples:

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between () parentheses also.

Example:

```
tup1 = ('physics', 'chemistry', 1997, 2000);
```

```
tup2 = (1, 2, 3, 4, 5);
```

```
tup3 = "a", "b", "c", "d";
```

The empty tuple is written as two parentheses containing nothing –

```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value.

```
tup1 = (50,);
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuples:

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index.

Example:

```
tup1 = ('physics', 'chemistry', 1997, 2000);
```

```
tup2 = (1, 2, 3, 4, 5, 6, 7);
```

```
print "tup1[0]: ", tup1[0];
```

```
print "tup2[1:5]: ", tup2[1:5];
```

Updating Tuples:

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples.

Example.

```
tup1 = (12, 34.56);
```

```
tup2 = ('abc', 'xyz');
```

```
# Following action is not valid for tuples
```

```
# tup1[0] = 100;
```

```
# So let's create a new tuple as follows
```

```
tup3 = tup1 + tup2;
```

```
print tup3;
```

Delete Tuple Elements:

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

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

```
Example:
```

```
tup = ('physics', 'chemistry', 1997, 2000);
```

```
print tup;
```

```
del tup;
```

```
print "After deleting tup : ";
```

```
print tup;
```

Tuple Operators and Built-in Functions:

Standard and Sequence Type Operators and Built-in Functions:

Object and sequence operators and built-in functions act the exact same way toward tuples as they do with lists. You can still take slices of tuples, concatenate and make multiple copies of tuples, validate membership, and compare tuples.

Creation, Repetition, Concatenation

```
>>> t = (['xyz', 123], 23, -103.4)
```

```
>>> t
```

```
(['xyz', 123], 23, -103.4)
```

```
>>> t * 2
```

```
(['xyz', 123], 23, -103.4, ['xyz', 123], 23, -103.4)
```

```
>>> t = t + ('free', 'easy')
```

```
>>> t
```

```
(['xyz', 123], 23, -103.4, 'free', 'easy')
```

Membership, Slicing

```
>>> 23 in t
True
>>> 123 in t
False
>>> t[0][1]
123
>>> t[1:]
(23, -103.4, 'free', 'easy')
```

Built-in Functions:

```
>>> str(t)
(['xyz', 123], 23, -103.4, 'free', 'easy')
>>> len(t)
5
>>> max(t)
'free'
>>> min(t)
-103.4
>>> cmp(t, ('xyz', 123], 23, -103.4, 'free', 'easy'))
0
>>> list(t)
[['xyz', 123], 23, -103.4, 'free', 'easy']
```

Operators:

```
>>> (4, 2) < (3, 5)
file:///D:/1/0132269937/ch06lev1sec17.html (1 von 2) [13.11.2007 16:23:15]
Section 6.17. Tuple Operators and Built-in Functions
False
>>> (2, 4) < (3, -1)
True
>>> (2, 4) == (3, -1)
False
>>> (2, 4) == (2, 4)
True
```

Tuple Type Operators and Built-in Functions and Methods:

Like lists, tuples have no operators or built-in functions for themselves. All of the list methods described in the previous section were related to a list object's mutability, i.e., sorting, replacing, appending, etc.

Since tuples are immutable, those methods are rendered superfluous, thus unimplemented.

Special Features of Tuples:

Some of the characteristics features of Tuples are:

- Tuples are defined in the same way as lists.
- They are enclosed within parenthesis and not within square braces.
- Elements of the tuple must have a defined order.
- Negative indices are counted from the end of the tuple, just like lists.
- Tuple also has the same structure where the values are separated by commas.