Python – Introduction

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

- **Python is Interpreted** − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive** − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python

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

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

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

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

Python Features

Python's features include −

- **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** − Python code is more clearly defined and visible to the eyes.

- **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
- **A broad standard library** − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** − Python provides interfaces to all major commercial databases.
- **GUI Programming** − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable** − Python provides a better structure and support for large programs than shell scripting.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below −

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

Python- Variables

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

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

Assigning Values to Variables

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

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

```
#!/usr/bin/python

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

print counter
print miles
print name
```

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

```
100
1000.0
John
```

Multiple Assignment

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

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

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

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

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

Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types −

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

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

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

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

Examples

Here are some examples of numbers −

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 x + yj, where x and y are the real numbers and j is the imaginary unit.

Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example –

```
#!/usr/bin/python

str = 'Hello World!'

print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
print str * 2      # Prints string two times
print str + "TEST" # Prints concatenated string
```

This will produce the following result –

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example −

```
#!/usr/bin/python


list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tinylist = [123, 'john']


print list          # Prints complete list

print list[0]       # Prints first element of the list

print list[1:3]     # Prints elements starting from 2nd till 3rd

print list[2:]      # Prints elements starting from 3rd element

print tinylist * 2  # Prints list two times

print list + tinylist # Prints concatenated lists
```

This produce the following result −

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists. For example −

```
#!/usr/bin/python
```

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2   )

tinytuple = (123, 'john')


print tuple             # Prints complete list

print tuple[0]          # Prints first element of the list

print tuple[1:3]        # Prints elements starting from 2nd till 3rd

print tuple[2:]         # Prints elements starting from 3rd element

print tinytuple * 2     # Prints list two times

print tuple + tinytuple # Prints concatenated lists
```

This produce the following result –

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists –

```
#!/usr/bin/python


tuple = ( 'abcd', 786 , 2.23, 'john', 70.2   )

list = [ 'abcd', 786 , 2.23, 'john', 70.2   ]

tuple[2] = 1000     # Invalid syntax with tuple

list[2] = 1000      # Valid syntax with list
```

Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example –

```
#!/usr/bin/python

dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}


print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```

This produce the following result –

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Sr.No.	Function & Description
1	**int(x [,base])** Converts x to an integer. base specifies the base if x is a string.
2	**long(x [,base])** Converts x to a long integer. base specifies the base if x is a string.
3	**float(x)** Converts x to a floating-point number.
4	**complex(real [,imag])** Creates a complex number.
5	**str(x)** Converts object x to a string representation.
6	**repr(x)** Converts object x to an expression string.
7	**eval(str)** Evaluates a string and returns an object.
8	**tuple(s)** Converts s to a tuple.
9	**list(s)** Converts s to a list.
10	**set(s)**

	Converts s to a set.
11	**dict(d)** Creates a dictionary. d must be a sequence of (key,value) tuples.
12	**frozenset(s)** Converts s to a frozen set.
13	**chr(x)** Converts an integer to a character.
14	**unichr(x)** Converts an integer to a Unicode character.
15	**ord(x)** Converts a single character to its integer value.
16	**hex(x)** Converts an integer to a hexadecimal string.
17	**oct(x)** Converts an integer to an octal string.

Basic Operators

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

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

Types of Operator

Python language supports the following types of operators.

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

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

Python Arithmetic Operators

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

[Show Example]

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20

//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) −	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Python Comparison Operators

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

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

[Show Example]

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

>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Python Assignment Operators

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

[Show Example]

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c /

		a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python Bitwise Operators

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

a = 0011 1100

b = 0000 1101

----

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

There are following Bitwise operators supported by Python language

[Show Example]

Operator	Description	Example

& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Python Logical Operators

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

[Show Example]

Operator	Description	Example
and Logical	If both the operands are true then condition becomes	(a and b)

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

Used to reverse the logical state of its operand.

Python Membership Operators

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

[Show Example]

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

Python Identity Operators

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

[Show Example]

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

Python Operators Precedence

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

[Show Example]

Sr.No.	Operator & Description
1	****** Exponentiation (raise to the power)
2	**~ + -** Complement, unary plus and minus (method names for the last two are +@ and -@)
3	*** / % //**

	Multiply, divide, modulo and floor division
4	**+ -** Addition and subtraction
5	**>> <<** Right and left bitwise shift
6	**&** Bitwise 'AND'
7	**^ |** Bitwise exclusive `OR' and regular `OR'
8	**<= < > >=** Comparison operators
9	**<> == !=** Equality operators
10	**= %= /= //= -= += *= **=** Assignment operators
11	**is is not** Identity operators
12	**in not in** Membership operators
13	**not or and**

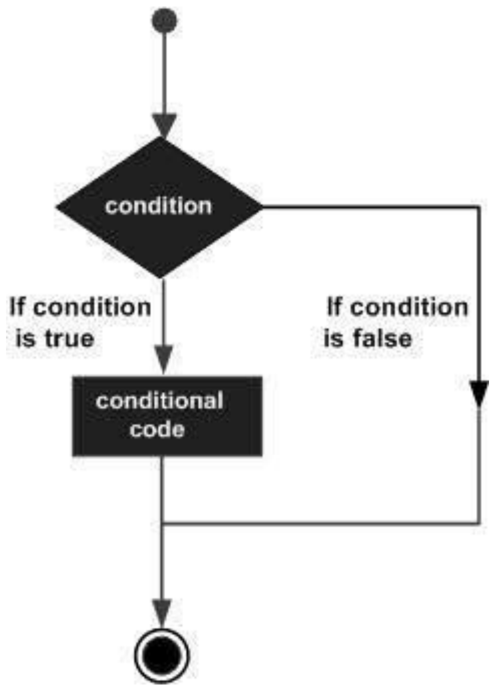
	Logical operators

Decision Making

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

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

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



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

Python programming language provides following types of decision making statements. Click the following links to check their detail.

Sr.No.	Statement & Description

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

Let us go through each decision making briefly −

Single Statement Suites

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

Here is an example of a **one-line if** clause −

```
#!/usr/bin/python


var = 100

if ( var == 100 ) : print "Value of expression is 100"

print "Good bye!"
```

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

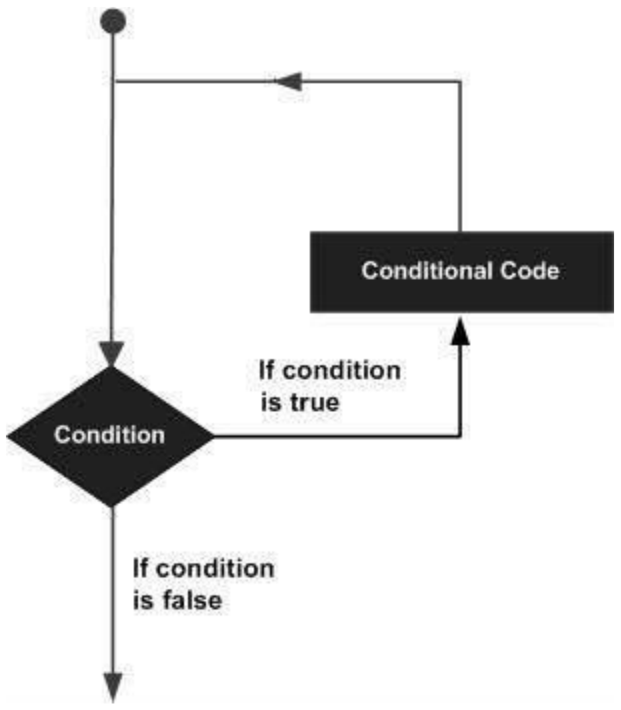
```
Value of expression is 100
Good bye!
```

Loops

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

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

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



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

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

	loop.

Loop Control Statements

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

Python supports the following control statements. Click the following links to check their detail.

Let us go through the loop control statements briefly

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

Lists

he most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Python Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Values in Lists

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

```
#!/usr/bin/python


list1 = ['physics', 'chemistry', 1997, 2000];

list2 = [1, 2, 3, 4, 5, 6, 7 ];

print "list1[0]: ", list1[0]

print "list2[1:5]: ", list2[1:5]
```

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

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example −

```
#!/usr/bin/python


list = ['physics', 'chemistry', 1997, 2000];

print "Value available at index 2 : "

print list[2]

list[2] = 2001;

print "New value available at index 2 : "

print list[2]
```

Note − append() method is discussed in subsequent section.

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

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example −

```
#!/usr/bin/python


list1 = ['physics', 'chemistry', 1997, 2000];

print list1

del list1[2];

print "After deleting value at index 2 : "
print list1
```

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

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

Note − remove() method is discussed in subsequent section.

Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input −

```
L = ['spam', 'Spam', 'SPAM!']
```

Python Expression	Results	Description
L[2]	SPAM!	Offsets start at zero
L[-2]	Spam	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Built-in List Functions & Methods

Python includes the following list functions −

Sr.No.	Function with Description
1	**cmp(list1, list2)** Compares elements of both lists.
2	**len(list)** Gives the total length of the list.
3	**max(list)** Returns item from the list with max value.
4	**min(list)** Returns item from the list with min value.
5	**list(seq)** Converts a tuple into list.

Python includes following list methods

Sr.No.	Methods with Description
1	**list.append(obj)**

	Appends object obj to list
2	**list.count(obj)** Returns count of how many times obj occurs in list
3	**list.extend(seq)** Appends the contents of seq to list
4	**list.index(obj)** Returns the lowest index in list that obj appears
5	**list.insert(index, obj)** Inserts object obj into list at offset index
6	**list.pop(obj=list[-1])** Removes and returns last object or obj from list
7	**list.remove(obj)** Removes object obj from list
8	**list.reverse()** Reverses objects of list in place
9	**list.sort([func])** Sorts objects of list, use compare func if given

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 a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For 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. For example −

```
#!/usr/bin/python


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];
```

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

```
tup1[0]:  physics
tup2[1:5]:  [2, 3, 4, 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 as the following example demonstrates −

```
#!/usr/bin/python


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;
```

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

```
(12, 34.56, 'abc', 'xyz')
```

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. For example −

```
#!/usr/bin/python


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

print tup;

del tup;

print "After deleting tup : ";

print tup;
```

This produces the following result. Note an exception raised, this is because after **del tup** tuple does not exist any more −

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
   File "test.py", line 9, in <module>
```

```
      print tup;
NameError: name 'tup' is not defined
```

Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter −

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input −

```
L = ('spam', 'Spam', 'SPAM!')
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero

L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples −

```
#!/usr/bin/python


print 'abc', -4.24e93, 18+6.6j, 'xyz';

x, y = 1, 2;

print "Value of x , y : ", x,y;
```

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

```
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

Built-in Tuple Functions

Python includes the following tuple functions −

Sr.No.	Function with Description
1	**cmp(tuple1, tuple2)** Compares elements of both tuples.
2	**len(tuple)** Gives the total length of the tuple.
3	**max(tuple)**

	Returns item from the tuple with max value.
4	**min(tuple)** Returns item from the tuple with min value.
5	**tuple(seq)** Converts a list into tuple.

Dictionary

ach key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example −

```
#!/usr/bin/python


dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

print "dict['Name']: ", dict['Name']

print "dict['Age']: ", dict['Age']
```

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

```
dict['Name']:  Zara
dict['Age']:  7
```

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows −

```
#!/usr/bin/python


dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

print "dict['Alice']: ", dict['Alice']
```

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

```
dict['Alice']:
Traceback (most recent call last):
   File "test.py", line 4, in <module>
      print "dict['Alice']: ", dict['Alice'];
KeyError: 'Alice'
```

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example −

```
#!/usr/bin/python


dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

dict['Age'] = 8; # update existing entry

dict['School'] = "DPS School"; # Add new entry


print "dict['Age']: ", dict['Age']

print "dict['School']: ", dict['School']
```

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

```
dict['Age']:  8
dict['School']:  DPS School
```

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example −

```
#!/usr/bin/python


dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

del dict['Name']; # remove entry with key 'Name'

dict.clear();     # remove all entries in dict

del dict ;        # delete entire dictionary


print "dict['Age']: ", dict['Age']

print "dict['School']: ", dict['School']
```

This produces the following result. Note that an exception is raised because after **del dict** dictionary does not exist any more −

```
dict['Age']:
Traceback (most recent call last):
   File "test.py", line 8, in <module>
      print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

Note − del() method is discussed in subsequent section.

Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

There are two important points to remember about dictionary keys −

(a) More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins. For example −

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
print "dict['Name']: ", dict['Name']
```

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

```
dict['Name']:  Manni
```

(b) Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example −

```
#!/usr/bin/python

dict = {['Name']: 'Zara', 'Age': 7}
print "dict['Name']: ", dict['Name']
```

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

```
Traceback (most recent call last):
   File "test.py", line 3, in <module>
      dict = {['Name']: 'Zara', 'Age': 7};
TypeError: unhashable type: 'list'
```

Built-in Dictionary Functions & Methods

Python includes the following dictionary functions −

Sr.No.	Function with Description
1	**cmp(dict1, dict2)** Compares elements of both dict.
2	**len(dict)** Gives the total length of the dictionary. This would be equal to the

	number of items in the dictionary.
3	**str(dict)** Produces a printable string representation of a dictionary
4	**type(variable)** Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Python includes following dictionary methods −

Sr.No.	Methods with Description
1	**dict.clear()** Removes all elements of dictionary *dict*
2	**dict.copy()** Returns a shallow copy of dictionary *dict*
3	**dict.fromkeys()** Create a new dictionary with keys from seq and values *set* to *value*.
4	**dict.get(key, default=None)** For *key* key, returns value or default if key not in dictionary
5	**dict.has_key(key)** Returns *true* if key in dictionary *dict*, *false* otherwise
6	**dict.items()** Returns a list of *dict*'s (key, value) tuple pairs
7	**dict.keys()** Returns list of dictionary dict's keys
8	**dict.setdefault(key, default=None)** Similar to get(), but will set dict[key]=default if *key* is not already in dict

9	**dict.update(dict2)** Adds dictionary *dict2*'s key-values pairs to *dict*
10	**dict.values()** Returns list of dictionary *dict*'s values

Basic Syntax

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

First Python Program

Let us execute programs in different modes of programming.

Interactive Mode Programming

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

```
$ python

Python 2.4.3 (#1, Nov 11 2010, 13:34:43)

[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>>
```

Type the following text at the Python prompt and press the Enter −

```
>>> print "Hello, Python!"
```

If you are running new version of Python, then you would need to use print statement with parenthesis as in **print ("Hello, Python!");**. However in Python version 2.4.3, this produces the following result −

```
Hello, Python!
```

Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. Python files have extension **.py**. Type the following source code in a test.py file –

```
print "Hello, Python!"
```

We assume that you have Python interpreter set in PATH variable. Now, try to run this program as follows –

```
$ python test.py
```

This produces the following result –

```
Hello, Python!
```

Let us try another way to execute a Python script. Here is the modified test.py file –

```
#!/usr/bin/python


print "Hello, Python!"
```

We assume that you have Python interpreter available in /usr/bin directory. Now, try to run this program as follows –

```
$ chmod +x test.py     # This is to make file executable

$./test.py
```

This produces the following result –

```
Hello, Python!
```

Python Identifiers

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

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

Here are naming conventions for Python identifiers −

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

Reserved Words

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

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return

del	import	try
elif	in	while
else	is	with
except	lambda	yield

Lines and Indentation

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

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

```
if True:
   print "True"
else:
   print "False"
```

However, the following block generates an error −

```
if True: print

"Answer" print

"True" else:

print "Answer"

print "False"
```

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

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

```
#!/usr/bin/python

import sys

try:
   # open file stream
   file = open(file_name, "w")
except IOError:
   print "There was an error writing to", file_name
   sys.exit()
print "Enter '", file_finish,
print "' When finished"
while file_text != file_finish:
   file_text = raw_input("Enter text: ")
   if file_text == file_finish:
      # close the file
      file.close
      break
   file.write(file_text)
   file.write("\n")
file.close()
file_name = raw_input("Enter filename: ")
if len(file_name) == 0:
   print "Next time please enter something"
   sys.exit()
try:
   file = open(file_name, "r")
except IOError:
   print "There was an error reading file"
   sys.exit()
file_text = file.read()
file.close()
print file_text
```

Multi-Line Statements

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

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

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

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

Quotation in Python

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

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

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

Comments in Python

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

```
#!/usr/bin/python
```

```
# First comment

print "Hello, Python!" # second comment
```

This produces the following result −

```
Hello, Python!
```

You can type a comment on the same line after a statement or expression −

```
name = "Madisetti" # This is again comment
```

You can comment multiple lines as follows −

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

Following triple-quoted string is also ignored by Python interpreter and can be used as a multiline comments:

```
'''
This is a multiline
comment.
'''
```

Using Blank Lines

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

Waiting for the User

The following line of the program displays the prompt, the statement saying “Press the enter key to exit”, and waits for the user to take action −

```
#!/usr/bin/python


raw_input("\n\nPress the enter key to exit.")
```

Here, "\n\n" is used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon –

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Multiple Statement Groups as Suites

A group of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines which make up the suite. For example –

```
if expression :
   suite
elif expression :
   suite
else :
   suite
```

Command Line Arguments

Many programs can be run to provide you with some basic information about how they should be run. Python enables you to do this with -h –

```
$ python -h

usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...

Options and arguments (and corresponding environment variables):

-c cmd : program passed in as string (terminates option list)

-d     : debug output from parser (also PYTHONDEBUG=x)

-E     : ignore environment variables (such as PYTHONPATH)

-h     : print this help message and exit


[ etc. ]
```

You can also program your script in such a way that it should accept various options. Command Line Arguments is an advanced topic and should be studied a bit later once you have gone through rest of the Python concepts

Regular Expressions

A *regular expression* is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The module **re** provides full support for Perl-like regular expressions in Python. The re module raises the exception re.error if an error occurs while compiling or using a regular expression.

We would cover two important functions, which would be used to handle regular expressions. But a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as **r'expression'**.

The *match* Function

This function attempts to match RE *pattern* to *string* with optional *flags*.

Here is the syntax for this function −

```
re.match(pattern, string, flags=0)
```

Here is the description of the parameters −

Sr.No.	Parameter & Description
1	**pattern** This is the regular expression to be matched.
2	**string** This is the string, which would be searched to match the pattern at the beginning of string.

3	**flags** You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.match* function returns a **match** object on success, **None** on failure. We use*group(num)* or *groups()* function of **match** object to get matched expression.

Sr.No.	Match Object Method & Description
1	**group(num=0)** This method returns entire match (or specific subgroup num)
2	**groups()** This method returns all matching subgroups in a tuple (empty if there weren't any)

Example

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs"

matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
   print "matchObj.group() : ", matchObj.group()
   print "matchObj.group(1) : ", matchObj.group(1)
   print "matchObj.group(2) : ", matchObj.group(2)
else:
   print "No match!!"
```

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

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

The *search* Function

This function searches for first occurrence of RE *pattern* within *string* with optional *flags*.

Here is the syntax for this function −

```
re.search(pattern, string, flags=0)
```

Here is the description of the parameters −

Sr.No.	Parameter & Description
1	**pattern** This is the regular expression to be matched.
2	**string** This is the string, which would be searched to match the pattern anywhere in the string.
3	**flags** You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The *re.search* function returns a **match** object on success, **none** on failure. We use *group(num)* or *groups()* function of **match** object to get matched expression.

Sr.No.	Match Object Methods & Description
1	**group(num=0)**

	This method returns entire match (or specific subgroup num)
2	**groups()** This method returns all matching subgroups in a tuple (empty if there weren't any)

Example

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

searchObj = re.search( r'(.*) are (.*?) .*', line, re.M|re.I)

if searchObj:
   print "searchObj.group() : ", searchObj.group()
   print "searchObj.group(1) : ", searchObj.group(1)
   print "searchObj.group(2) : ", searchObj.group(2)
else:
   print "Nothing found!!"
```

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

```
searchObj.group() :  Cats are smarter than dogs
searchObj.group(1) :  Cats
searchObj.group(2) :  smarter
```

Matching Versus Searching

Python offers two different primitive operations based on regular expressions: **match** checks for a match only at the beginning of the string, while **search**checks for a match anywhere in the string (this is what Perl does by default).

Example

```
#!/usr/bin/python
import re


line = "Cats are smarter than dogs";


matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
   print "match --> matchObj.group() : ", matchObj.group()
else:
   print "No match!!"


searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
   print "search --> searchObj.group() : ", searchObj.group()
else:
   print "Nothing found!!"
```

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

```
No match!!
search --> matchObj.group() :  dogs
```

Search and Replace

One of the most important **re** methods that use regular expressions is **sub**.

Syntax

```
re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided. This method returns modified string.

Example

```
#!/usr/bin/python
import re


phone = "2004-959-559 # This is Phone Number"


# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print "Phone Num : ", num


# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print "Phone Num : ", num
```

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

```
Phone Num :  2004-959-559
Phone Num :  2004959559
```

Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these −

Sr.No.	Modifier & Description
1	**re.I** Performs case-insensitive matching.
2	**re.L**

	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior(\b and \B).
3	**re.M** Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
4	**re.S** Makes a period (dot) match any character, including a newline.
5	**re.U** Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
6	**re.X** Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

Regular Expression Patterns

Except for control characters, **(+ ? . * ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python –

Sr.No.	Pattern & Description
1	**^** Matches beginning of line.

2	**$** Matches end of line.
3	**.** Matches any single character except newline. Using m option allows it to match newline as well.
4	**[...]** Matches any single character in brackets.
5	**[^...]** Matches any single character not in brackets
6	**re*** Matches 0 or more occurrences of preceding expression.
7	**re+** Matches 1 or more occurrence of preceding expression.
8	**re?** Matches 0 or 1 occurrence of preceding expression.
9	**re{ n}** Matches exactly n number of occurrences of preceding expression.
10	**re{ n,}** Matches n or more occurrences of preceding expression.
11	**re{ n, m}**

	Matches at least n and at most m occurrences of preceding expression.
12	**a| b** Matches either a or b.
13	**(re)** Groups regular expressions and remembers matched text.
14	**(?imx)** Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.
15	**(?-imx)** Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.
16	**(?: re)** Groups regular expressions without remembering matched text.
17	**(?imx: re)** Temporarily toggles on i, m, or x options within parentheses.
18	**(?-imx: re)** Temporarily toggles off i, m, or x options within parentheses.
19	**(?#...)** Comment.
20	**(?= re)** Specifies position using a pattern. Doesn't have a range.

21	**(?! re)** Specifies position using pattern negation. Doesn't have a range.
22	**(?> re)** Matches independent pattern without backtracking.
23	**\w** Matches word characters.
24	**\W** Matches nonword characters.
25	**\s** Matches whitespace. Equivalent to [\t\n\r\f].
26	**\S** Matches nonwhitespace.
27	**\d** Matches digits. Equivalent to [0-9].
28	**\D** Matches nondigits.
29	**\A** Matches beginning of string.
30	**\Z** Matches end of string. If a newline exists, it matches just before

	newline.
31	**\z** Matches end of string.
32	**\G** Matches point where last match finished.
33	**\b** Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
34	**\B** Matches nonword boundaries.
35	**\n, \t, etc.** Matches newlines, carriage returns, tabs, etc.
36	**\1...\9** Matches nth grouped subexpression.
37	**\10** Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

Regular Expression Examples

Literal characters

Sr.No.	Example & Description

1	**python** Match "python".

Character classes

Sr.No.	Example & Description
1	**[Pp]ython** Match "Python" or "python"
2	**rub[ye]** Match "ruby" or "rube"
3	**[aeiou]** Match any one lowercase vowel
4	**[0-9]** Match any digit; same as [0123456789]
5	**[a-z]** Match any lowercase ASCII letter
6	**[A-Z]** Match any uppercase ASCII letter
7	**[a-zA-Z0-9]** Match any of the above
8	**[^aeiou]** Match anything other than a lowercase vowel

9	**[^0-9]** Match anything other than a digit

Special Character Classes

Sr.No.	Example & Description
1	**.** Match any character except newline
2	**\d** Match a digit: [0-9]
3	**\D** Match a nondigit: [^0-9]
4	**\s** Match a whitespace character: [\t\r\n\f]
5	**\S** Match nonwhitespace: [^ \t\r\n\f]
6	**\w** Match a single word character: [A-Za-z0-9_]
7	**\W** Match a nonword character: [^A-Za-z0-9_]

Repetition Cases

Sr.No.	Example & Description

1	**ruby?** Match "rub" or "ruby": the y is optional
2	**ruby*** Match "rub" plus 0 or more ys
3	**ruby+** Match "rub" plus 1 or more ys
4	**\d{3}** Match exactly 3 digits
5	**\d{3,}** Match 3 or more digits
6	**\d{3,5}** Match 3, 4, or 5 digits

Nongreedy repetition

This matches the smallest number of repetitions −

Sr.No.	Example & Description
1	**<.*>** Greedy repetition: matches "<python>perl>"
2	**<.*?>** Nongreedy: matches "<python>" in "<python>perl>"

Grouping with Parentheses

Sr.No.	Example & Description
1	**\D\d+** No group: + repeats \d
2	**(\D\d)+** Grouped: + repeats \D\d pair
3	**([Pp]ython(,)?)+** Match "Python", "Python, python, python", etc.

Backreferences

This matches a previously matched group again −

Sr.No.	Example & Description
1	**([Pp])ython&\1ails** Match python&pails or Python&Pails
2	**(['"])[^\1]*\1** Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc.

Alternatives

Sr.No.	Example & Description
1	**python|perl** Match "python" or "perl"
2	**rub(y|le))**

	Match "ruby" or "ruble"
3	**Python(!+|\?)** "Python" followed by one or more ! or one ?

Anchors

This needs to specify match position.

Sr.No.	Example & Description
1	**^Python** Match "Python" at the start of a string or internal line
2	**Python$** Match "Python" at the end of a string or line
3	**\APython** Match "Python" at the start of a string
4	**Python\Z** Match "Python" at the end of a string
5	**\bPython\b** Match "Python" at a word boundary
6	**\brub\B** \B is nonword boundary: match "rub" in "rube" and "ruby" but not alone
7	**Python(?=!)** Match "Python", if followed by an exclamation point.

8	**Python(?!!)** Match "Python", if not followed by an exclamation point.

Special Syntax with Parentheses

Sr.No.	Example & Description
1	**R(?#comment)** Matches "R". All the rest is a comment
2	**R(?i)uby** Case-insensitive while matching "uby"
3	**R(?i:uby)** Same as above
4	**rub(?:y|le))** Group only without creating \1 backreference

Exception Handling

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them −

- **Exception Handling** − This would be covered in this tutorial. Here is a list standard Exceptions available in Python: Standard Exceptions.
- **Assertions** − This would be covered in Assertions in Pythontutorial.

List of Standard Exceptions −

Sr.No.	Exception Name & Description

1	**Exception** Base class for all exceptions
2	**StopIteration** Raised when the next() method of an iterator does not point to any object.
3	**SystemExit** Raised by the sys.exit() function.
4	**StandardError** Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError** Base class for all errors that occur for numeric calculation.
6	**OverflowError** Raised when a calculation exceeds maximum limit for a numeric type.
7	**FloatingPointError** Raised when a floating point calculation fails.
8	**ZeroDivisionError** Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError** Raised in case of failure of the Assert statement.

10	**AttributeError** Raised in case of failure of attribute reference or assignment.
11	**EOFError** Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError** Raised when an import statement fails.
13	**KeyboardInterrupt** Raised when the user interrupts program execution, usually by pressing Ctrl+c.
14	**LookupError** Base class for all lookup errors.
15	**IndexError** Raised when an index is not found in a sequence.
16	**KeyError** Raised when the specified key is not found in the dictionary.
17	**NameError** Raised when an identifier is not found in the local or global namespace.
18	**UnboundLocalError** Raised when trying to access a local variable in a function or method but no value has been assigned to it.

19	**EnvironmentError** Base class for all exceptions that occur outside the Python environment.
20	**IOError** Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
21	**IOError** Raised for operating system-related errors.
22	**SyntaxError** Raised when there is an error in Python syntax.
23	**IndentationError** Raised when indentation is not specified properly.
24	**SystemError** Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25	**SystemExit** Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
26	**TypeError** Raised when an operation or function is attempted that is invalid for the specified data type.
27	**ValueError**

	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	**RuntimeError** Raised when a generated error does not fall into any category.
29	**NotImplementedError** Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Assertions in Python

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.

Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.

Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

The *assert* Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError* exception.

The **syntax** for assert is −

```
assert Expression[, Arguments]
```

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a traceback.

Example

Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature −

```
#!/usr/bin/python

def KelvinToFahrenheit(Temperature):

   assert (Temperature >= 0),"Colder than absolute zero!"

   return ((Temperature-273)*1.8)+32

print KelvinToFahrenheit(273)

print int(KelvinToFahrenheit(505.78))

print KelvinToFahrenheit(-5)
```

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

```
32.0
451
Traceback (most recent call last):
File "test.py", line 9, in <module>
print KelvinToFahrenheit(-5)
File "test.py", line 4, in KelvinToFahrenheit
assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

What is Exception?

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Handling an exception

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax

Here is simple syntax of *try....except...else* blocks −

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Here are few important points about the above-mentioned syntax −

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

Example

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all −

```
#!/usr/bin/python


try:

   fh = open("testfile", "w")

   fh.write("This is my test file for exception handling!!")

except IOError:

   print "Error: can\'t find file or read data"

else:

   print "Written content in the file successfully"
   fh.close()
```

This produces the following result −

```
Written content in the file successfully
```

Example

This example tries to open a file where you do not have write permission, so it raises an exception −

```
#!/usr/bin/python


try:

   fh = open("testfile", "r")

   fh.write("This is my test file for exception handling!!")

except IOError:

   print "Error: can\'t find file or read data"

else:

   print "Written content in the file successfully"
```

This produces the following result −

```
Error: can't find file or read data
```

The *except* Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows −

```
try:
   You do your operations here;
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The *except* Clause with Multiple Exceptions

You can also use the same *except* statement to handle multiple exceptions as follows −

```
try:
   You do your operations here;
   ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception list,
   then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

The try-finally Clause

You can use a **finally:** block along with a **try:** block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this −

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

You cannot use *else* clause as well along with a finally clause.

Example

```
#!/usr/bin/python


try:

   fh = open("testfile", "w")

   fh.write("This is my test file for exception handling!!")

finally:

   print "Error: can\'t find file or read data"
```

If you do not have permission to open the file in writing mode, then this will produce the following result −

```
Error: can't find file or read data
```

Same example can be written more cleanly as follows −

```
#!/usr/bin/python


try:

   fh = open("testfile", "w")

   try:

      fh.write("This is my test file for exception handling!!")

   finally:

      print "Going to close the file"

      fh.close()

except IOError:

   print "Error: can\'t find file or read data"
```

When an exception is thrown in the *try* block, the execution immediately passes to the *finally* block. After all the statements in the *finally* block are executed, the exception is raised again and is handled in the *except*statements if present in the next higher layer of the *try-except* statement.

Argument of an Exception

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows −

```
try:
   You do your operations here;
   ......................
except ExceptionType, Argument:
   You can print value of Argument here...
```

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

Example

Following is an example for a single exception −

```
#!/usr/bin/python

# Define a function here.
def temp_convert(var):
   try:
      return int(var)
   except ValueError, Argument:

      print "The argument does not contain numbers\n", Argument


# Call above function here.

temp_convert("xyz");
```

This produces the following result −

```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```

Raising an Exceptions

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows.

Syntax

```
raise [Exception [, args [, traceback]]]
```

Here, *Exception* is the type of exception (for example, NameError) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

Example

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows −

```
def functionName( level ):

   if level < 1:

      raise "Invalid level!", level

      # The code below to this would not be executed

      # if we raise the exception
```

Note: In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write the except clause as follows −

```
try:

   Business Logic here...

except "Invalid level!":

   Exception handling here...

else:

   Rest of the code here...
```

User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to *RuntimeError*. Here, a class is created that is subclassed from *RuntimeError*. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class *Networkerror*.

```
class Networkerror(RuntimeError):

   def __init__(self, arg):

      self.args = arg
```

So once you defined above class, you can raise the exception as follows −

```
try:

   raise Networkerror("Bad hostname")

except Networkerror,e:

   print e.args
```

Python Data Types

Data type defines the type of the variable, whether it is an integer variable, string variable, tuple, dictionary, list etc. In this guide, you will learn about the data types and their usage in Python.

Python data types

Python data types are divided in two categories, mutable data types and immutable data types.

Immutable Data types in Python
1. Numeric
2. String
3. Tuple

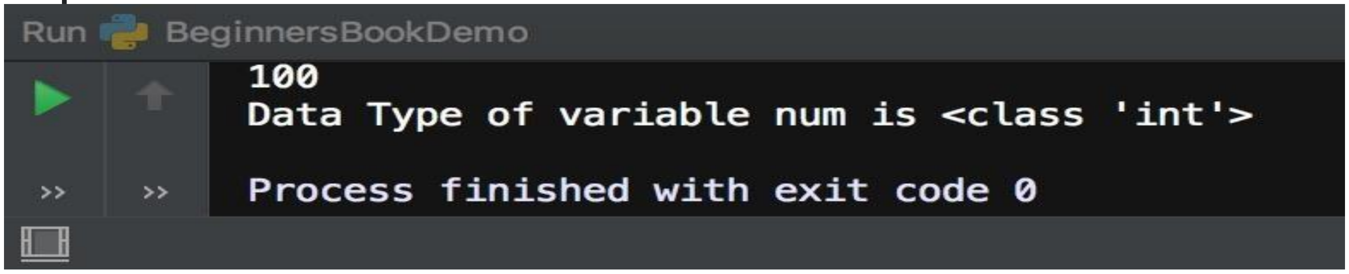
Mutable Data types in Python
1. List
2. Dictionary
3. Set

1. Numeric Data Type in Python

Integer – In Python 3, there is no upper bound on the integer number which means we can have the value as large as our system memory allows.

```
# Integer number
num = 100
print(num)
print("Data Type of variable num is", type(num))
```

Output:



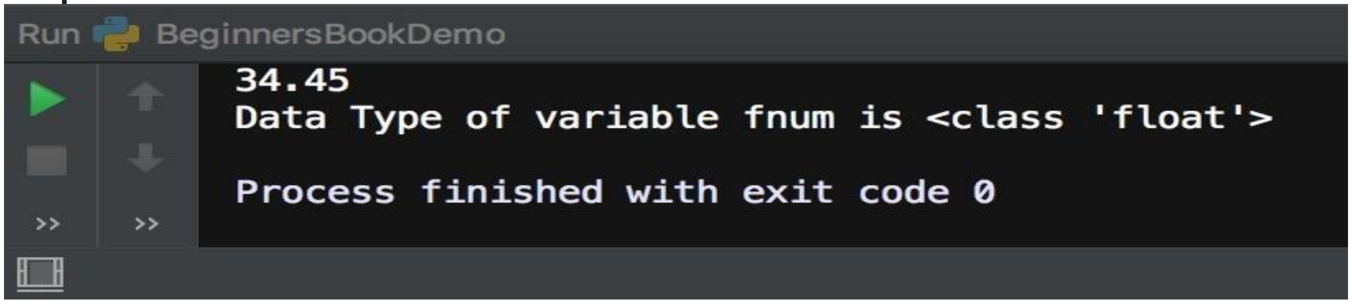
```
100
Data Type of variable num is <class 'int'>

Process finished with exit code 0
```

Long – Long data type is deprecated in Python 3 because there is no need for it, since the integer has no upper limit, there is no point in having a data type that allows larger upper limit than integers.

Float – Values with decimal points are the float values, there is no need to specify the data type in Python. It is automatically inferred based on the value we are assigning to a variable. For example here fnum is a float data type.

```
# float number
fnum = 34.45
print(fnum)
print("Data Type of variable fnum is", type(fnum))
```

Output:



```
34.45
Data Type of variable fnum is <class 'float'>

Process finished with exit code 0
```

Complex Number – Numbers with real and imaginary parts are known as complex numbers. Unlike other programming language such as Java, Python is able to identify these complex numbers with the values. In the following example when we print the type of the variable cnum, it prints as complex number.

```
# complex number
cnum = 3 + 4j
print(cnum)
print("Data Type of variable cnum is", type(cnum))
```

Binary, Octal and Hexadecimal numbers

In Python we can print decimal equivalent of binary, octal and hexadecimal numbers using the prefixes.
0b(zero + ‘b’) and 0B(zero + ‘B’) – **Binary Number**
0o(zero + ‘o’) and 0O(zero + ‘O’) – **Octal Number**
0x(zero + ‘x’) and 0X(zero + ‘X’) – **Hexadecimal Number**

```
# integer equivalent of binary number 101
num = 0b101
print(num)
```

```
# integer equivalent of Octal number 32
num2 = 0o32
print(num2)

# integer equivalent of Hexadecimal number FF
num3 = 0xFF
print(num3)
```

2. Python Data Type - String

String is a sequence of characters in Python. The data type of String in Python is called “str”.

Strings in Python are either enclosed with single quotes or double quotes. In the following example we have demonstrated two strings one with the double quotes and other string s2 with the single quotes. To read more about strings, refer this article: Python Strings.

```
# Python program to print strings and type

s = "This is a String"
s2 = 'This is also a String'

# displaying string s and its type
print(s)
print(type(s))

# displaying string s2 and its type
print(s2)
print(type(s2))
```

3. Python Data Type - Tuple

Tuple is immutable data type in Python which means it cannot be changed. It is an ordered collection of elements enclosed in round brackets and separated by commas. To read more about tuple, refer this tutorial: Python tuple.

```
# tuple of integers
t1 = (1, 2, 3, 4, 5)
# prints entire tuple
print(t1)

# tuple of strings
t2 = ("hi", "hello", "bye")
# loop through tuple elements
for s in t2:
    print (s)

#tuple of mixed type elements
t3 = (2, "Lucy", 45, "Steve")
'''
Print a specific element
indexes start with zero
'''
print(t3[2])
```

4. Python Data Type - List

List is similar to tuple, it is also an ordered collection of elements, however list is a mutable data type which means it can be changed unlike tuple which is an immutable data type.

A list is enclosed with square brackets and elements are separated by commas. To read more about Lists, refer this guide: Python Lists

```
# list of integers
lis1 = (1, 2, 3, 4, 5)
# prints entire list
print(lis1)

# list of strings
lis2 = ("Apple", "Orange", "Banana")
# loop through tuple elements
for x in lis2:
    print (x)

# List of mixed type elements
lis3 = (20, "Chaitanya", 15, "BeginnersBook")
'''
Print a specific element in list
indexes start with zero
'''
print("Element at index 3 is:",lis3[3])
```

5. Python Data Type - Dictionary

Dictionary is a collection of key and value pairs. A dictionary doesn't allow duplicate keys but the values can be duplicate. It is an ordered, indexed and mutable collection of elements. To read more about it refer: Python dictionary.

The keys in a dictionary doesn't necessarily to be a single data type, as you can see in the following example that we have 1 integer key and two string keys.

```
# Dictionary example

dict = {1:"Chaitanya","lastname":"Singh", "age":31}

# prints the value where key value is 1
print(dict[1])
# prints the value where key value is "lastname"
print(dict["lastname"])
# prints the value where key value is "age"
print(dict["age"])
```

6. Python Data Type - Set

A set is an unordered and unindexed collection of items. This means when we print the elements of a set they will appear in the random order and we cannot access the elements of set based on indexes because it is unindexed.

Elements of set are separated by commas and enclosed in curly braces. Lets take an example to understand the sets in Python.

```
# Set Example
myset = {"hi", 2, "bye", "Hello World"}

# loop through set
for a in myset:
    print(a)

# checking whether 2 exists in myset
print(2 in myset)

# adding new element
myset.add(99)
print(myset)
```

Python If Statement explained with examples

BY CHAITANYA SINGH | FILED UNDER: PYTHON TUTORIAL

If statements are control flow statements which helps us to run a particular code only when a certain condition is satisfied. For example, you want to print a message on the screen only when a condition is true then you can use if statement to accomplish this in programming. In this guide, we will learn how to use **if statements in Python programming** with the help of examples.

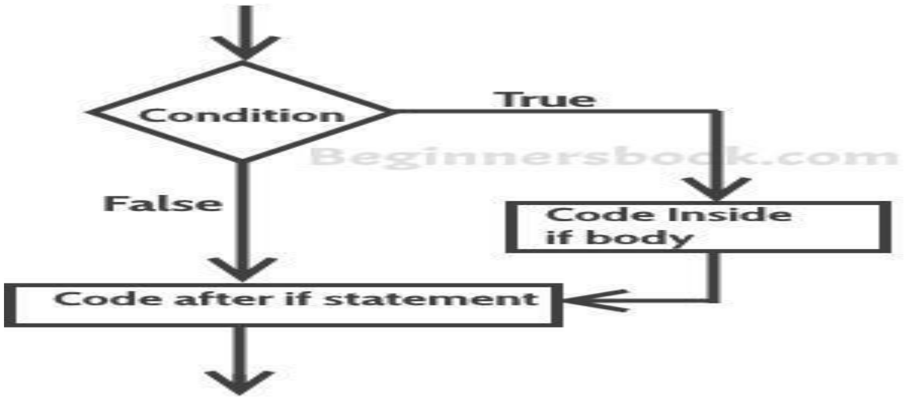
There are other control flow statements available in Python such as if..else, if..elif..else,
nested if etc. However in this guide, we will only cover the if statements, other control statements are covered in separate tutorials.

Syntax of If statement in Python

The syntax of if statement in Python is pretty simple.

```
if condition:
   block_of_code
```

If statement flow diagram



Python – If statement Example

```
flag = True
if flag==True:
    print("Welcome")
    print("To")
    print("BeginnersBook.com")
```

Output:

```
Welcome
To
BeginnersBook.com
```

In the above example we are checking the value of flag variable and if the value is True then we are executing few print statements. The important point to note here is that even if we do not compare the value of flag with the 'True' and simply put 'flag' in place of condition, the code would run just fine so the better way to write the above code would be:

```
flag = True
if flag:
    print("Welcome")
    print("To")
    print("BeginnersBook.com")
```

By seeing this we can understand how if statement works. The output of the condition would either be true or false. If the outcome of condition is true then the statements inside body of 'if' executes, however if the outcome of condition is false then the statements inside 'if' are **skipped**. Lets take another example to understand this:

```
flag = False
if flag:
    print("You Guys")
    print("are")
    print("Awesome")
```

The output of this code is none, it does not print anything because the outcome of condition is 'false'.

Python if example without boolean variables

In the above examples, we have used the boolean variables in place of conditions. However we can use any variables in our conditions. For example:

```
num = 100
if num < 200:
    print("num is less than 200")
```

Output:

```
num is less than 200
```

Python If else Statement Example

In the last tutorial we learned how to use if statements in Python. In this guide, we will learn another control statement 'if..else'.

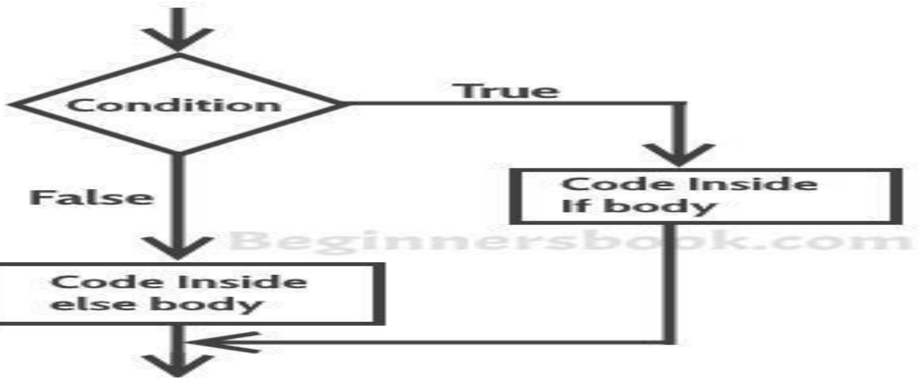
We use if statements when we need to execute a certain block of Python code when a particular condition is true. **If..else statements** are like extension of 'if' statements, with the help of if..else we can execute certain statements if condition is true and a different set of statements if condition is false. For example, you want to print 'even number' if the number is even and 'odd number' if the number is not even, we can accomplish this with the help of if..else statement.

Python – Syntax of if..else statement

```
if condition:
   block_of_code_1
else:
   block_of_code_2
```

block_of_code_1: This would execute if the given condition is true
block_of_code_2: This would execute if the given condition is false

If..else flow control



If-else example in Python

```
num = 22
if num % 2 == 0:
    print("Even Number")
else:
    print("Odd Number")
```

Output:

```
Even Number
```

Python If elif else statement example

In the previous tutorials we have seen if statement and if..else statement. In this tutorial, we will learn **if elif else statement in Python**. The if..elif..else statement is used when we need to check multiple conditions.

Syntax of if elif else statement in Python

This way we are checking multiple conditions.

```
if condition:
   block_of_code_1
elif condition_2:
   block_of_code_2
elif condition_3:
   block_of_code_3
..
..
..
else:
  block_of_code_n
```

Notes:
1. There can be multiple 'elif' blocks, however there is only 'else' block is allowed.
2. Out of all these blocks only one block_of_code gets executed. If the condition is true then the code inside 'if' gets executed, if condition is false then the next condition(associated with elif) is evaluated and so on. If none of the conditions is true then the code inside 'else' gets executed.

Python - if..elif..else statement example

In this example, we are **checking multiple conditions** using if..elif..else statement.

```
num = 1122
if 9 < num < 99:
    print("Two digit number")
elif 99 < num < 999:
    print("Three digit number")
elif 999 < num < 9999:
    print("Four digit number")
else:
    print("number is <= 9 or >= 9999")
```

Python Nested If else statement

In the previous tutorials, we have covered the if statement, if..else statement and if..elif..else statement. In this tutorial, we will learn the nesting of these control statements.

When there is an if statement (or if..else or if..elif..else) is present inside another if statement (or if..else or if..elif..else) then this is calling the **nesting of control statements**.

Nested if..else statement example

Here we have a if statement inside another if..else statement block. Nesting control statements makes us to check multiple conditions.

```
num = -99
if num > 0:
    print("Positive Number")
else:
    print("Negative Number")
    #nested if
    if -99<=num:
        print("Two digit Negative Number")
```

Output:

```
Negative Number
Two digit Negative Number
```

Python for Loop explained with examples

A loop is a used for iterating over a set of statements repeatedly. In Python we have three types of loops **for**, **while** and **do-while**. In this guide, we will learn **for loop** and the other two loops are covered in the separate tutorials.

Syntax of For loop in Python

```
for <variable> in <sequence>:
   # body_of_loop that has set of statements
   # which requires repeated execution
```

Here <variable> is a variable that is used for iterating over a <sequence>. On every iteration it takes the **next value** from <sequence> until the end of sequence is reached.

Lets take few examples of for loop to understand the usage.

Python - For loop example

The following example shows the use of for loop to iterate over a list of numbers. In the body of for loop we are calculating the square of each number present in list and displaying the same.

```
# Program to print squares of all numbers present in a list

# List of integer numbers
numbers = [1, 2, 4, 6, 11, 20]

# variable to store the  square  of  each  num  temporary
sq = 0

# iterating over the given list
for val in numbers:
    # calculating square of each number
    sq = val * val
    # displaying the squares
    print(sq)
```

Output:

```
1
4
16
36
121
400
```

Function range()

In the above example, we have iterated over a list using for loop. However we can also use a range() function in for loop to iterate over numbers defined by range().

range(n): generates a set of whole numbers starting from 0 to (n-1).
For example:
range(8) is equivalent to [0, 1, 2, 3, 4, 5, 6, 7]

range(start, stop): generates a set of whole numbers starting from start to stop-1.
For example:
range(5, 9) is equivalent to [5, 6, 7, 8]

range(start, stop, step_size): The default step_size is 1 which is why when we didn't specify the step_size, the numbers generated are having difference of 1. However by specifying step_size we can generate numbers having the difference of step_size.
For example:
range(1, 10, 2) is equivalent to [1, 3, 5, 7, 9]

Lets use the **range() function** in for loop:

Python for loop example using range() function

Here we are using **range() function** to calculate and display the sum of first 5 natural numbers.

```
# Program to print the sum of first 5 natural numbers

# variable to store the sum
sum = 0

# iterating over natural numbers using range()
for val in range(1, 6):
    # calculating sum
    sum = sum + val

# displaying sum of first 5 natural numbers
print(sum)
```

Output:

```
15
```

For loop with else block

Unlike Java, In Python we can have an optional 'else' block associated with the loop. The 'else' block executes only when the loop has completed all the iterations. Lets take an example:

```
for val in range(5):
        print(val)
else:
        print("The loop has completed execution")
```

Output:

```
0
1
2
3
4
The loop has completed execution
```

Note: The else block only executes when the loop is finished.

Nested For loop in Python

When a for loop is present inside another for loop then it is called a nested for loop. Lets take an example of nested for loop.

```
for num1 in range(3):
        for num2 in range(10, 14):
                print(num1, ",", num2)
```

Output:

```
0 , 10
0 , 11
0 , 12
0 , 13
1 , 10
1 , 11
1 , 12
1 , 13
2 , 10
2 , 11
2 , 12
2 , 13
```

Python While Loop

While loop is used to iterate over a block of code repeatedly until a given condition returns false. In the last tutorial, we have seen for loop in Python, which is also used for the same purpose. The main difference is that we use **while loop** when we are **not** certain of the number of times the loop requires execution, on the other hand when we exactly know how many times we need to run the loop, we use for loop.

Syntax of while loop

```
while condition:
    #body_of_while
```

The body_of_while is set of Python statements which requires repeated execution. These set of statements execute repeatedly until the given condition returns false.

Flow of while loop

1. First the given condition is checked, if the condition returns false, the loop is terminated and the control jumps to the next statement in the program after the loop.
2. If the condition returns true, the set of statements inside loop are executed and then the control jumps to the beginning of the loop for next iteration.

These two steps happen repeatedly as long as the condition specified in while loop remains true.

Python – While loop example

Here is an example of while loop. In this example, we have a variable `num` and we are displaying the value of `num` in a loop, the loop has a increment operation where we are increasing the value of `num`. This is very important step, the while loop must have a increment or decrement operation, else the loop will run indefinitely, we will cover this later in infinite while loop.

```
num = 1
# loop will repeat itself as long as
# num < 10 remains true
while num < 10:
    print(num)
    #incrementing the value of num
    num = num + 3
```

Output:

```
1
4
7
```

Infinite while loop

Example 1:
This will print the word ‘hello’ indefinitely because the condition will always be true.

```
while True:
   print("hello")
```

Example 2:

```
num = 1
while num<5:
    print(num)
```

This will print ‘1’ indefinitely because inside loop we are not updating the value of num, so the value of num will always remain 1 and the condition num < 5 will always return true.

Nested while loop in Python

When a while loop is present inside another while loop then it is called nested while loop. Lets take an example to understand this concept.

```
i = 1
j = 5
while i < 4:
    while j < 8:
        print(i, ",", j)
        j = j + 1
        i = i + 1
```

Output:

```
1 , 5
2 , 6
3 , 7
```

Python - while loop with else block

We can have a ‘else’ block associated with while loop. The ‘else’ block is optional. It executes only after the loop finished execution.

```
num = 10 while
num > 6:
    print(num)
    num = num-1
else:
    print("loop is finished")
```

Output:

```
10
9
8
7
loop is finished
```

Python break Statement

The **break statement** is used to terminate the loop when a certain condition is met. We already learned in previous tutorials (for loop and while loop) that a loop is used to iterate a set of statements repeatedly as long as the **loop condition** returns true. The break statement is generally used inside a loop along with a if statement so that when a particular condition (defined in if statement) returns true, the break statement is encountered and the loop terminates.

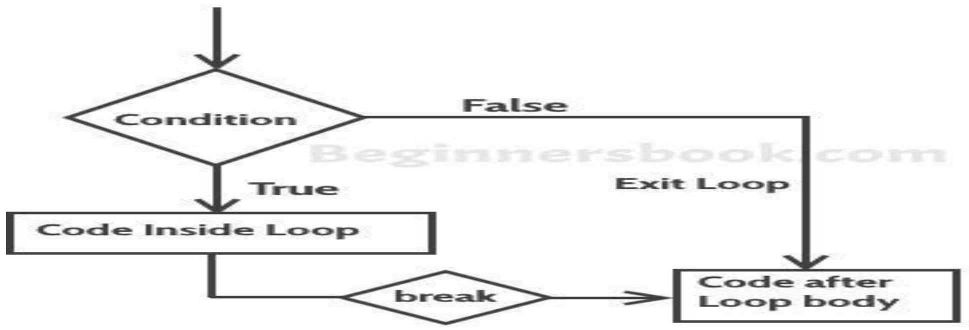
For example, lets say we are searching an element in a list, so for that we are running a loop starting from the first element of the list to the last element of the list. Using break statement, we can terminate the loop as soon as the element is found because why run the loop unnecessary till the end of list when our element is found. We can achieve this with the help of break statement (we will see this example programmatically in the example section below).

Syntax of break statement in Python

The syntax of **break statement in Python** is similar to what we have seen in Java.

```
break
```

Flow diagram of break



Example of break statement

In this example, we are searching a number '88' in the given list of numbers. The requirement is to display all the numbers till the number '88' is found and when it is found, terminate the loop and do not display the rest of the numbers.

```
# program to display all the elements before number 88
for num in [11, 9, 88, 10, 90, 3, 19]:
   print(num)
   if(num==88):
           print("The number 88 is found")
           print("Terminating the loop")
           break
```

Output:

```
11
9
88
The number 88 is found
Terminating the loop
```

Note: You would always want to use the break statement with a if statement so that only when the condition associated with ‘if’ is true then only break is encountered. If you do not use it with ‘if’ statement then the break statement would be encountered in the first iteration of loop and the loop would always terminate on the first iteration.

Python Continue Statement

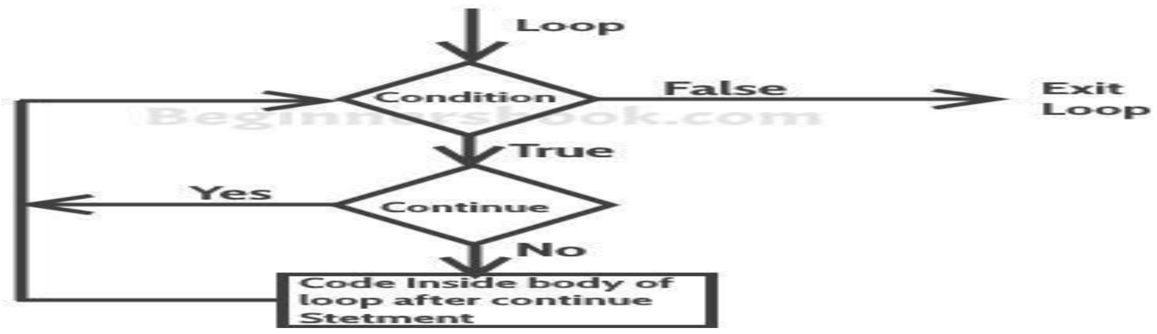
The **continue statement** is used inside a loop to skip the rest of the statements in the body of loop for the current iteration and jump to the beginning of the loop for next iteration. The break and continue statements are used to alter the flow of loop, break terminates the loop when a condition is met and continue skip the current iteration.

Syntax of continue statement in Python

The syntax of continue statement in Python is similar to what we have seen in Java(except the semicolon)

```
continue
```

Flow diagram of continue



Example of continue statement

Lets say we have a list of numbers and we want to print only the odd numbers out of that list. We can do this by using continue statement.
We are skipping the print statement inside loop by using continue statement when the number is even, this way all the even numbers are skipped and the print statement executed for all the odd numbers.

```
# program to display only odd numbers
for num in [20, 11, 9, 66, 4, 89, 44]:
    # Skipping the iteration when number is even
    if num%2 == 0:
        continue
    # This statement will be skipped for all even numbers
    print(num)
```

Output:

```
11
9
89
```

Python pass Statement

The **pass statement** acts as a **placeholder** and usually used when there is no need of code but a statement is still required to make a code syntactically correct. For example we want to declare a function in our code but we want to implement that function in future, which means we are not yet ready to write the body of the function. In this case we cannot leave the body of function empty as this would raise error because it is syntactically incorrect, in such cases we can use pass statement which **does nothing** but **makes the code syntactically correct**.

Pass statement vs comment

You may be wondering that a python comment works similar to the pass statement as it does nothing so we can use comment in place of pass statement. Well, it is not the case, a comment is not a placeholder and it is completely ignored by the Python interpreter while on the other hand `pass` is not ignored by interpreter, it says the interpreter to **do nothing**.

Python pass statement example

If the number is even we are doing nothing and if it is odd then we are displaying the number.

```
for num in [20, 11, 9, 66, 4, 89, 44]:
    if num%2 == 0:
        pass
    else:
        print(num)
```

Output:

```
11
9
89
```

Other examples:

A function that does nothing(yet), may be implemented in future.

```
def f(arg): pass    # a function that does nothing (yet)
```

A class that does not have any methods(yet), may have methods in future implementation.

```
class C: pass       # a class with no methods (yet)
```

Reference

Python docs – pass statement

Python Functions

In this guide, we will learn about **functions in Python**. A function is a block of code that contains one or more Python statements and used for performing a specific task.

Why use function in Python?

As I mentioned above, a function is a block of code that performs a specific task. Lets discuss what we can achieve in Python by using functions in our code:
1. **Code re-usability**: Lets say we are writing an application in Python where we need to perform a specific task in several places of our code, assume that we need to write 10 lines of code to do that specific task. It would be better to write those 10 lines of code in a function and just call the function wherever needed, because writing those 10 lines every time you perform that task is tedious, it would make your code lengthy, less-readable and increase the chances of human errors.

2. **Improves Readability**: By using functions for frequent tasks you make your code structured and readable. It would be easier for anyone to look at the code and be able to understand the flow and purpose of the code.

3. **Avoid redundancy**: When you no longer repeat the same lines of code throughout the code and use functions in places of those, you actually avoiding the redundancy that you may have created by not using functions.

Syntax of functions in Python

Function declaration:

```
def function_name(function_parameters):
        function_body # Set of Python statements
        return # optional return statement
```

Calling the function:

```
# when function doesn't return anything
function_name(parameters)
```

OR

```
# when function returns something
# variable is to store the returned value
```

```
variable = function_name(parameters)
```

Python Function example

Here we have a function `add()` that adds two numbers passed to it as parameters. Later after function declaration we are calling the function twice in our program to perform the addition.

```
def add(num1, num2):
    return num1 + num2


sum1 = add(100, 200)
sum2 = add(8, 9)
print(sum1)
print(sum2)
```

Output:

```
300
17
```

Default arguments in Function

Now that we know how to declare and call a function, lets see how can we use the **default arguments**. By using default arguments we can avoid the errors that may arise while calling a function without passing all the parameters. Lets take an example to understand this:

In this example we have provided the default argument for the second parameter, this default argument would be used when we do not provide the second parameter while calling this function.

```
# default argument for second parameter
def add(num1, num2=1):
    return num1 + num2


sum1 = add(100, 200)
sum2 = add(8)  # used default argument for second param
sum3 = add(100)# used default argument for second param
print(sum1)
print(sum2)
print(sum3)
```

Output:

```
300
9
101
```

Types of functions

There are two types of functions in Python:
1. **Built-in functions**: These functions are predefined in Python and we need not to declare these functions before calling them. We can freely invoke them as and when needed.
2. **User defined functions**: The functions which we create in our code are user-defined functions. The add() function that we have created in above examples is a user-defined function.

We will cover more about these function types in the separate guides.

Python Recursion

A function is said to be a **recursive** if it calls itself. For example, lets say we have a function abc()and in the body of abc() there is a call to the abc().

Python example of Recursion

In this example we are defining a **user-defined function** factorial(). This function finds the factorial of a number by calling itself repeatedly until the **base case**(We will discuss more about base case later, after this example) is reached.

```
# Example of recursion in Python to
# find the factorial of a given number

def factorial(num):
    """This function calls itself to find
    the factorial of a number"""

    if num == 1:
        return 1
    else:
        return (num * factorial(num - 1))


num = 5
print("Factorial of", num, "is: ", factorial(num))
```

Output:

```
Factorial of 5 is:  120
```

Lets see what happens in the above example:

```
factorial(5) returns 5 * factorial(5-1)
    i.e. 5 * factorial(4)
               |__5*4*factorial(3)
                      |__5*4*3*factorial(2)
                           |__5*4*3*2*factorial(1)
```

Note: factorial(1) is a base case for which we already know the value of factorial. The base case is defined in the body of function with this code:

```
if num == 1:
    return 1
```

What is a base case in recursion

When working with recursion, we should define a base case for which we already know the answer. In the above example we are finding factorial of an integer number and we already know that the factorial of 1 is 1 so this is our base case.

Each **successive recursive call** to the function should bring it closer to the base case, which is exactly what we are doing in above example.

We use base case in recursive function so that the function stops calling itself when the base case is reached. Without the base case, the function would keep calling itself indefinitely.

Why use recursion in programming?

We use recursion to break a big problem in small problems and those small problems into further smaller problems and so on. At the end the solutions of all the smaller subproblems are collectively helps in finding the solution of the big main problem.

Advantages of recursion

Recursion makes our program:
1. Easier to write.
2. Readable – Code is easier to read and understand.
3. Reduce the lines of code – It takes less lines of code to solve a problem using recursion.

Disadvantages of recursion

1. Not all problems can be solved using recursion.
2. If you don’t define the base case then the code would run indefinitely.
3. Debugging is difficult in recursive functions as the function is calling itself in a loop and it is hard to understand which call is causing the issue.
4. Memory overhead – Call to the recursive function is not memory efficient.

Python Numbers

In this guide, we will see how to work with **numbers in Python**. Python supports integers, floats and complex numbers.

An **integer** is a number without decimal point for example 5, 6, 10 etc.

A **float** is a number with decimal point for example 6.7, 6.0, 10.99 etc.

A **complex number** has a real and imaginary part for example 7+8j, 8+11j etc.

Example: Numbers in Python

```
# Python program to display numbers of
# different data types

#  int
num1 = 10
num2 = 100
print(num1+num2)

#  float
a  =  10.5
b  =  8.9
print(a-b)

# complex numbers
x = 3 +  4j
y = 9 + 8j
print(y-x)
```

Output:

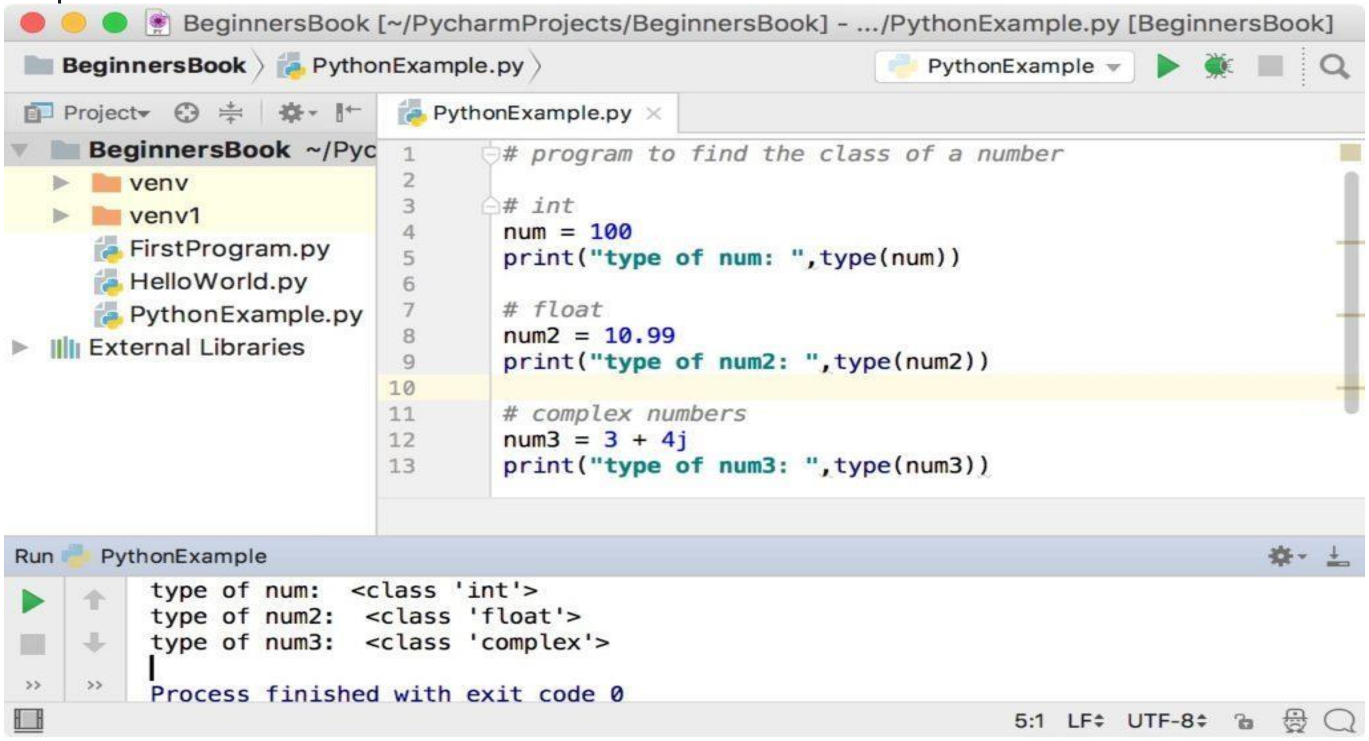
```
110
1.5999999999999996
(6+4j)
```

Python example to find the class(data type) of a number

We can use the type() function to find out the class of a number. An integer number belongs to intclass, a float number belongs to float class and a complex number belongs to complex class.

```
# program to find the class of a number

#  int
num = 100
print("type of num: ",type(num))

# float num2 =
10.99
print("type of num2: ",type(num2))

# complex numbers
num3 = 3 + 4j
print("type of num3: ",type(num3))
```

Output:



```
# program to find the class of a number

# int
num = 100
print("type of num: ",type(num))

# float
num2 = 10.99
print("type of num2: ",type(num2))

# complex numbers
num3 = 3 + 4j
print("type of num3: ",type(num3))
```

```
type of num:  <class 'int'>
type of num2:  <class 'float'>
type of num3:  <class 'complex'>

Process finished with exit code 0
```

The isinstance() function

The isinstance() functions checks whether a number belongs to a particular class and returns true or false based on the result.
For example:
isinstance(num, int) will return true if the number num is an integer number.
isinstance(num, int) will return false if the number num is not an integer number.

Example of isinstance() function

```
num = 100
# true because num is an integer
print(isinstance(num, int))

# false because num is not a float
print(isinstance(num, float))

# false because num is not a complex number
print(isinstance(num, complex))
```

Output:

```
True
False
False
```

Python List with examples

In this guide, we will discuss **lists in Python**. A list is a data type that allows you to store various types data in it. List is a compound data type which means you can have different-2 data types under a list, for example we can have integer, float and string items in a same list.

1. Create a List in Python

Lets see how to create a list in Python. To create a list all you have to do is to place the items inside a square bracket [] separated by comma ,.

```
# list of floats
num_list = [11.22, 9.9, 78.34, 12.0]

# list of int, float and strings
mix_list = [1.13, 2, 5, "beginnersbook", 100, "hi"]

# an empty list
nodata_list = []
```

As we have seen above, a list can have data items of same type or different types. This is the reason list comes under compound data type.

2. Accessing the items of a list

Syntax to access the list items:

```
list_name[index]
```

Example:

```
# a list of numbers
numbers = [11, 22, 33, 100, 200, 300]

# prints 11
print(numbers[0])

# prints 300
print(numbers[5])

# prints 22
print(numbers[1])
```

Output:

```
11
300
22
```

Points to Note:

1. The index cannot be a float number.

For example:

```
# a list of numbers
numbers = [11, 22, 33, 100, 200, 300]

# error
print(numbers[1.0])
```

Output:

```
TypeError: list indices must be integers or slices, not float
```

2. The index must be in range to avoid `IndexError`. The range of the index of a list having 10 elements is 0 to 9, if we go beyond 9 then we will get IndexError. However if we go below 0 then it would not cause issue in certain cases, we will discuss that in our next section.

For example:

```
# a list of numbers
numbers = [11, 22, 33, 100, 200, 300]

# error
print(numbers[6])
```

Output:

```
IndexError: list index out of range
```

3. Negative Index to access the list items from the end

Unlike other programming languages where negative index may cause issue, Python allows you to use negative indexes. The idea behind this to allow you to access the list elements starting from the end. For example an index of `-1` would access the last element of the list, `-2` second last, `-3` third last and so on.

Example of Negative indexes in Python

```
# a list of strings
my_list = ["hello", "world", "hi", "bye"]

# prints "bye"
print(my_list[-1])

# prints "world"
print(my_list[-3])

# prints "hello"
print(my_list[-4])
```

Output:

```
bye
world
hello
```

4. How to get a sublist in Python using slicing

We can get a sublist from a list in Python using slicing operation. Lets say we have a list `n_list`having 10 elements, then we can slice this list using colon `:` operator. Lets take an example to understand this:

Slicing example

```
# list of numbers
n_list = [1, 2, 3, 4, 5, 6, 7]

# list items from 2nd to 3rd
print(n_list[1:3])

# list items from beginning to 3rd
print(n_list[:3])

# list items from 4th to end of list
print(n_list[3:])
```

```
# Whole list
print(n_list[:])
```

Output:

```
[2, 3]
[1, 2, 3]
[4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
```

5. List Operations

There are various operations that we can perform on Lists.

Addition

There are several ways you can add elements to a list.

```
# list of numbers
n_list = [1, 2, 3, 4]

# 1. adding item at the desired location
# adding element 100 at the fourth location
n_list.insert(3, 100)

# list: [1, 2, 3, 100, 4]
print(n_list)

# 2. adding element at the end of the list
n_list.append(99)

# list: [1, 2, 3, 100, 4, 99]
print(n_list)

# 3. adding several elements at the end of list
# the following statement can also be written like this:
# n_list + [11, 22]
n_list.extend([11, 22])

# list: [1, 2, 3, 100, 4, 99, 11, 22]
print(n_list)
```

Output:

```
[1, 2, 3, 100, 4]
[1, 2, 3, 100, 4, 99]
[1, 2, 3, 100, 4, 99, 11, 22]
```

Update elements

We can change the values of elements in a List. Lets take an example to understand this:

```
# list of numbers
n_list = [1, 2, 3, 4]

# Changing the value of 3rd item
n_list[2] = 100

# list: [1, 2, 100, 4]
print(n_list)

# Changing the values of 2nd to fourth items
n_list[1:4] = [11, 22, 33]

# list: [1, 11, 22, 33]
print(n_list)
```

Output:

```
[1, 2, 100, 4]
[1, 11, 22, 33]
```

Delete elements

```
# list of numbers
n_list = [1, 2, 3, 4, 5, 6]

# Deleting 2nd element
del n_list[1]

# list: [1, 3, 4, 5, 6]
print(n_list)

# Deleting elements from 3rd to 4th
del n_list[2:4]

# list: [1, 3, 6]
print(n_list)

# Deleting the whole list
del n_list
```

Output:

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

Deleting elements using remove(), pop() and clear() methods

remove(item): Removes specified item from list.
pop(index): Removes the element from the given index.
pop(): Removes the last element.
clear(): Removes all the elements from the list.

```
# list of chars
ch_list = ['A', 'F', 'B', 'Z', 'O', 'L']

# Deleting the element with value 'B'
ch_list.remove('B')

# list: ['A', 'F', 'Z', 'O', 'L']
print(ch_list)

# Deleting 2nd element
ch_list.pop(1)

# list: ['A', 'Z', 'O', 'L']
print(ch_list)

# Deleting all the elements
ch_list.clear()

# list: []
print(ch_list)
```

Output:

```
['A', 'F', 'Z', 'O', 'L']
['A', 'Z', 'O', 'L']
[]
```

Python Strings

A string is usually a bit of text (sequence of characters). In Python we use ” (double quotes) or ‘ (single quotes) to represent a string. In this guide we will see how to create, access, use and manipulate strings in Python programming language.

1. How to create a String in Python

There are several ways to create strings in Python.
1. We can use ‘ (single quotes), see the string str in the following code.
2. We can use ” (double quotes), see the string str2 in the source code below.
3. Triple double quotes “”” and triple single quotes ”’ are used for creating multi-line strings in Python. See the strings str3 and str4 in the following example.

```
# lets see the ways to create strings in Python
str = 'beginnersbook'
print(str)

str2 = "Chaitanya"
print(str2)

# multi-line string
str3 = """Welcome to
   Beginnersbook.com"""
print(str3)

str4 = '''This is a tech
   blog'''
print(str4)
```

Output:

```
beginnersbook
Chaitanya
Welcome to
   Beginnersbook.com
This is a tech
   blog
```

2. How to access strings in Python

A string is nothing but an array of characters so we can use the indexes to access the characters of a it. Just like arrays, the indexes start from 0 to the length-1.

You will get **IndexError** if you try to access the character which is not in the range. For example,
if a string is of length 6 and you try to access the 8th char of it then you will get this error.

You will get **TypeError** if you do not use integers as indexes, for example if you use a float as an index then you will get this error.

```
str = "Kevin"

# displaying whole string
print(str)

# displaying first character of string
print(str[0])

# displaying third character of string
print(str[2])
```

```
# displaying the last character of the string
print(str[-1])

# displaying the second last char of string
print(str[-2])
```

Output:

```
Kevin
K
v
n
i
```

3. Python String Operations

Lets see the operations that can be performed on the strings.

Getting a substring in Python - Slicing operation

We can slice a string to get a **substring** out of it. To understand the concept of **slicing** we must understand the positive and negative indexes in Python (see the example above to understand this). Lets take a look at the few examples of slicing.

```
str = "Beginnersbook"

# displaying whole string
print("The original string is: ", str)

# slicing 10th to the last character
print("str[9:]: ", str[9:])

# slicing 3rd to 6th character
print("str[2:6]: ", str[2:6])

# slicing from start to the 9th character
print("str[:9]: ", str[:9])

# slicing from 10th to second last character
print("str[9:-1]: ", str[9:-1])
```

Output:

```
The original string is:  Beginnersbook
str[9:]:  book
str[2:6]:  ginn
str[:9]:  Beginners
str[9:-1]:  boo
```

Concatenation of strings in Python

The + operator is used for **string concatenation in Python**. Lets take an example to understand this:

```
str1 = "One"
str2 = "Two"
str3 = "Three"

# Concatenation of three strings
print(str1 + str2 + str3)
```

Output:

```
OneTwoThree
```

Note: When **+ operator** is used on numbers it adds them but when it used on strings it concatenates them. However if you try to use this between string and number then it will throw TypeError.

For example:

```
s = "one"
n = 2
print(s+n)
```

Output:

```
TypeError: must be str, not int
```

Repetition of string - Replication operator

We can use * operator to repeat a string by specified number of times.

```
str = "ABC"

# repeating the string str by 3 times
print(str*3)
```

Output:

```
ABCABCABC
```

Python Membership Operators in Strings

in: This checks whether a string is present in another string or not. It returns true if the entire string is found else it returns false.
not in: It works just opposite to what “in” operator does. It returns true if the string is not found in the specified string else it returns false.

```
str = "Welcome to beginnersbook.com"
str2 = "Welcome"
str3 = "Chaitanya"
str4 = "XYZ"

# str2 is in str? True
print(str2 in str)

# str3 is in str? False
print(str3 in str)

# str4 not in str? True
print(str4 not in str)
```

Output:

```
True
False
True
```

Python - Relational Operators on Strings

The **relational operators** works on strings based on the ASCII values of characters.
The ASCII value of a is 97, b is 98 and so on.
The ASCII value of A is 65, B is 66 and so on.

```
str = "ABC"
str2 = "aBC"
str3 = "XYZ"
str4 = "XYz"

# ASCII value of str2 is > str? True
print(str2 > str)

# ASCII value of str3 is > str4? False
print(str3 > str4)
```

Output:

```
True
False
```

Python Tuple with example

In Python, a tuple is similar to List except that the objects in tuple are immutable which means we cannot change the elements of a tuple once assigned. On the other hand, we can change the elements of a list.

1. Tuple vs List

1. The elements of a **list** are mutable whereas the elements of a **tuple** are immutable.
2. When we do not want to change the data over time, the **tuple** is a preferred data type whereas when we need to change the data in future, **list** would be a wise option.
3. Iterating over the elements of a **tuple** is faster compared to iterating over a **list**.
4. Elements of a **tuple** are enclosed in parenthesis whereas the elements of **list** are enclosed in square bracket.

2. How to create a tuple in Python

To create a tuple in Python, place all the elements in a () parenthesis, separated by commas. A tuple can have heterogeneous data items, a tuple can have string and list as data items as well.

Example - Creating tuple

In this example, we are creating few tuples. We can have tuple of same type of data items as well as mixed type of data items. This example also shows nested tuple (tuples as data items in another tuple).

```
# tuple of strings
my_data = ("hi", "hello", "bye")
print(my_data)

# tuple of int, float, string
my_data2 = (1, 2.8, "Hello World")
print(my_data2)

# tuple of string and list
my_data3 = ("Book", [1, 2, 3])
print(my_data3)

# tuples inside another tuple
# nested tuple
my_data4 = ((2, 3, 4), (1, 2, "hi"))
print(my_data4)
```

Output:

```
('hi', 'hello', 'bye')
(1, 2.8, 'Hello World')
('Book', [1, 2, 3])
((2, 3, 4), (1, 2, 'hi'))
```

Empty tuple:

```
# empty tuple
my_data = ()
```

Tuple with only single element:

Note: When a tuple has only one element, we must put a comma after the element, otherwise Python will not treat it as a tuple.

```
# a tuple with single data item
my_data = (99,)
```

If we do not put comma after 99 in the above example then python will treat my_data as an int variable rather than a tuple.

3. How to access tuple elements

We use indexes to access the elements of a tuple. Lets take few example to understand the working.

Accessing tuple elements using positive indexes

We can also have negative indexes in tuple, we have discussed that in the next section. Indexes starts with 0 that is why we use 0 to access the first element of tuple, 1 to access second element and so on.

```
# tuple of strings
my_data = ("hi", "hello", "bye")

# displaying all elements
print(my_data)

# accessing first element
# prints "hi"
print(my_data[0])

# accessing third element
# prints "bye"
print(my_data[2])
```

Output:

```
('hi', 'hello', 'bye')
hi
bye
```

Note:
1. **TypeError**: If you do not use integer indexes in the tuple. For example my_data[2.0] will raise this error. The index must always be an integer.
2. **IndexError**: Index out of range. This error occurs when we mention the index which is not in the range. For example, if a tuple has 5 elements and we try to access the 7th element then this error would occurr.

Negative indexes in tuples

Similar to list and strings we can use negative indexes to access the tuple elements from the end.
-1 to access last element, -2 to access second last and so on.

```
my_data = (1, 2, "Kevin", 8.9)

# accessing last element
# prints 8.9
print(my_data[-1])

# prints 2
print(my_data[-3])
```

Output:

```
8.9
2
```

Accessing elements from nested tuples

Lets understand how the double indexes are used to access the elements of nested tuple. The first index represents the element of main tuple and the second index represent the element of the nested tuple.

In the following example, when I used my_data[2][1], it accessed the second element of the nested tuple. Because 2 represented the third element of main tuple which is a tuple and the 1 represented the second element of that tuple.

```
my_data = (1, "Steve", (11, 22, 33))

# prints 'v'
print(my_data[1][3])

# prints 22
print(my_data[2][1])
```

Output:

```
v
22
```

4. Operations that can be performed on tuple in Python

Lets see the operations that can be performed on the tuples in Python.

Changing the elements of a tuple

We cannot change the elements of a tuple because elements of tuple are immutable. However we can change the elements of nested items that are mutable. For example, in the following code, we are changing the element of the list which is present inside the tuple. List items are mutable that’s why it is allowed.

```
my_data = (1, [9, 8, 7], "World")
print(my_data)

# changing the element of the list
# this is valid because list is mutable
my_data[1][2] = 99
print(my_data)

# changing the element of tuple
# This is not valid since tuple elements are immutable
# TypeError: 'tuple' object does not support item assignment
# my_data[0] = 101
# print(my_data)
```

Output:

```
(1, [9, 8, 7], 'World')
(1, [9, 8, 99], 'World')
```

Delete operation on tuple

We already discussed above that tuple elements are immutable which also means that we cannot delete the elements of a tuple. However deleting entire tuple is possible.

```
my_data = (1, 2, 3, 4, 5, 6)
print(my_data)

# not possible
#  error
# del my_data[2]
# deleting entire tuple is possible
del my_data

# not possible
#  error
# because my_data is deleted
# print(my_data)
```

Output:

```
(1, 2, 3, 4, 5, 6)
```

Slicing operation in tuples

```
my_data = (11, 22, 33, 44, 55, 66, 77, 88, 99)
print(my_data)

# elements from 3rd to 5th
# prints (33, 44, 55)
print(my_data[2:5])

# elements from start to 4th
# prints (11, 22, 33, 44)
print(my_data[:4])

# elements from 5th to end
# prints (55, 66, 77, 88, 99)
print(my_data[4:])

# elements from 5th to second last
# prints (55, 66, 77, 88)
print(my_data[4:-1])

# displaying entire tuple
print(my_data[:])
```

Output:

```
(11, 22, 33, 44, 55, 66, 77, 88, 99)
(33, 44, 55)
(11, 22, 33, 44)
(55, 66, 77, 88, 99)
(55, 66, 77, 88)
(11, 22, 33, 44, 55, 66, 77, 88, 99)
```

Membership Test in Tuples

in: Checks whether an element exists in the specified tuple.
not in: Checks whether an element does not exist in the specified tuple.

```
my_data = (11, 22, 33, 44, 55, 66, 77, 88, 99)
print(my_data)
```

```
# true
print(22 in my_data)

# false
print(2 in my_data)

# false
print(88 not in my_data)

# true
print(101 not in my_data)
```

Output:

```
(11, 22, 33, 44, 55, 66, 77, 88, 99)
True
False
False
True
```

Iterating a tuple

```
# tuple of fruits
my_tuple = ("Apple", "Orange", "Grapes", "Banana")

# iterating over tuple elements
for fruit in my_tuple:
     print(fruit)
```

Output:

```
Apple
Orange
Grapes
Banana
```

Python Dictionary with examples

Dictionary is a mutable data type in Python. A python dictionary is a collection of key and value pairs separated by a colon (:), enclosed in curly braces {}.

Python Dictionary

Here we have a dictionary. Left side of the colon(:) is the key and right side of the : is the value.

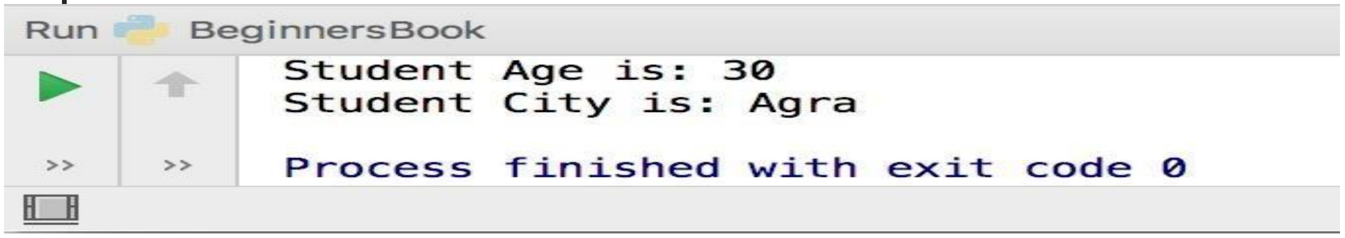
```
mydict = {'StuName': 'Ajeet', 'StuAge': 30, 'StuCity': 'Agra'}
```

Points to Note:
1. Keys must be unique in dictionary, duplicate values are allowed.
2. A dictionary is said to be empty if it has no key value pairs. An empty dictionary is denoted like this: {}.
3. The keys of dictionary must be of immutable data types such as String, numbers or tuples.

Accessing dictionary values using keys in Python

To access a value we can can use the corresponding key in the square brackets as shown in the following example. Dictionary name followed by square brackets and in the brackets we specify the key for which we want the value.

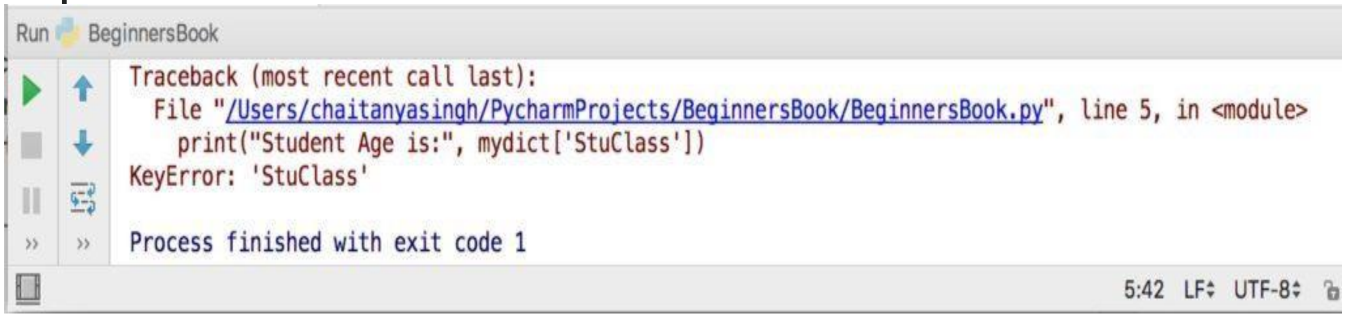
```
mydict = {'StuName': 'Ajeet', 'StuAge': 30, 'StuCity': 'Agra'}
print("Student Age is:", mydict['StuAge'])
print("Student City is:", mydict['StuCity'])
```
Output:



```
Student Age is: 30
Student City is: Agra

Process finished with exit code 0
```

If you specify a key which doesn't exist in the dictionary then you will get a compilation error. For example. Here we are trying to access the value for key 'StuClass' which does not exist in the dictionary `mydict`, thus we get a compilation error when we run this code.

```
mydict = {'StuName': 'Ajeet', 'StuAge': 30, 'StuCity': 'Agra'}
print("Student Age is:", mydict['StuClass'])
print("Student City is:", mydict['StuCity'])
```
Output:



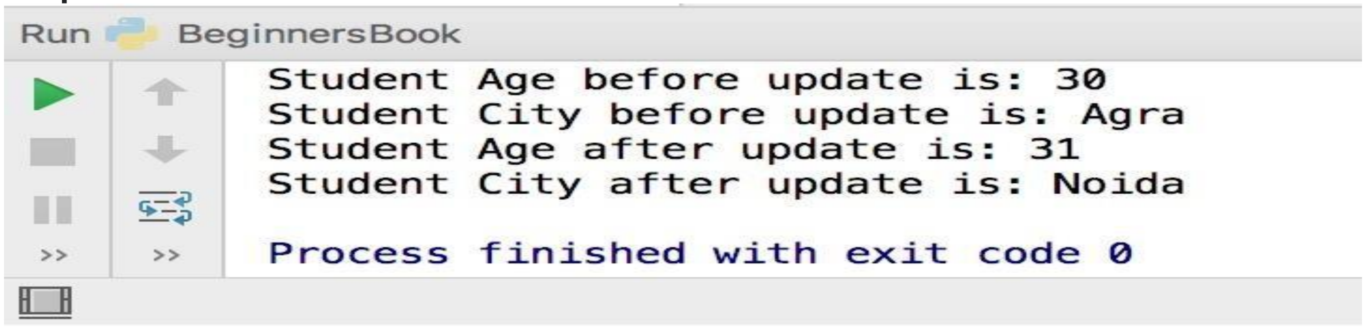
```
Traceback (most recent call last):
  File "/Users/chaitanyasingh/PycharmProjects/BeginnersBook/BeginnersBook.py", line 5, in <module>
    print("Student Age is:", mydict['StuClass'])
KeyError: 'StuClass'

Process finished with exit code 1
```

Change values in Dictionary

Here we are updating the values for the existing key-value pairs. To update a value in dictionary we are using the corresponding key.

```
mydict = {'StuName': 'Ajeet', 'StuAge': 30, 'StuCity': 'Agra'}
print("Student Age before update is:", mydict['StuAge'])
print("Student City before update is:", mydict['StuCity'])
mydict['StuAge'] = 31
mydict['StuCity'] = 'Noida'
print("Student Age after update is:", mydict['StuAge'])
print("Student City after update is:", mydict['StuCity'])
```

Output:



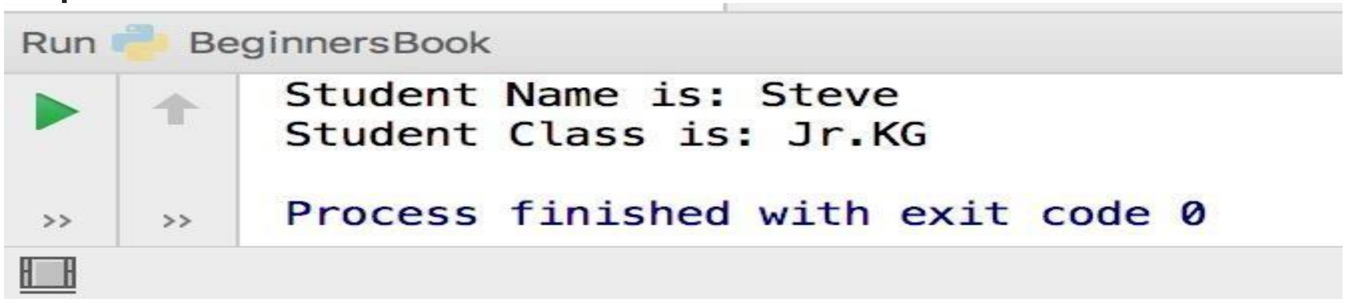
```
Student Age before update is: 30
Student City before update is: Agra
Student Age after update is: 31
Student City after update is: Noida

Process finished with exit code 0
```

Adding a new entry (key-value pair) in dictionary

We can also add a new key-value pair in an existing dictionary. Lets take an example to understand this.

```
mydict = {'StuName': 'Steve', 'StuAge': 4, 'StuCity': 'Agra'}
mydict['StuClass'] = 'Jr.KG'
print("Student Name is:", mydict['StuName'])
print("Student Class is:", mydict['StuClass'])
```

Output:



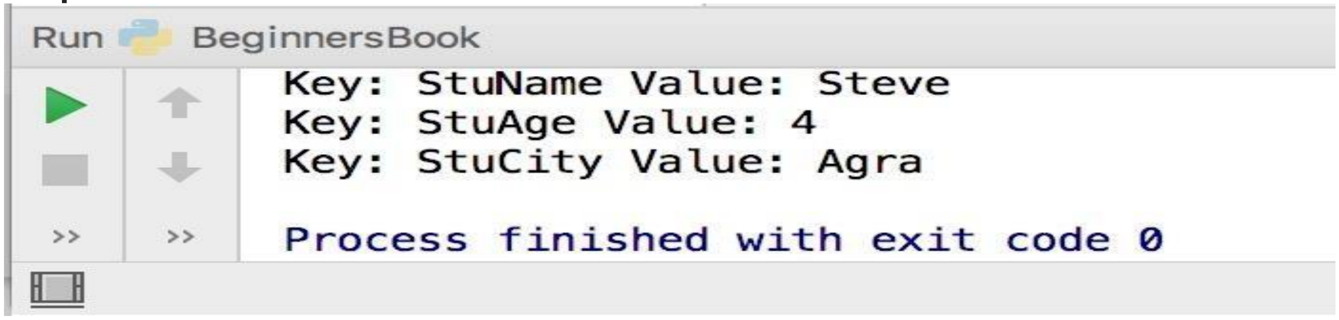
```
Student Name is: Steve
Student Class is: Jr.KG

Process finished with exit code 0
```

Loop through a dictionary

We can loop through a dictionary as shown in the following example. Here we are using for loop.

```
mydict = {'StuName': 'Steve', 'StuAge': 4, 'StuCity': 'Agra'}
for e in mydict:
  print("Key:",e,"Value:",mydict[e])
```

Output:



```
Key: StuName Value: Steve
Key: StuAge Value: 4
Key: StuCity Value: Agra

Process finished with exit code 0
```

Python delete operation on dictionary

We can delete key-value pairs as well as entire dictionary in python. Lets take an example. As you can see we can use del following by dictionary name and in square brackets we can specify the key to delete the specified key value pair from dictionary.

To delete all the entries (all key-value pairs) from dictionary we can use the clear() method.

To delete entire dictionary along with all the data use del keyword followed by dictionary name as shown in the following example.

```
mydict = {'StuName': 'Steve', 'StuAge': 4, 'StuCity': 'Agra'}
del mydict['StuCity']; # remove entry with key 'StuCity'
mydict.clear();     # remove all key-value pairs from mydict
del mydict ;        # delete entire dictionary mydict
```

Python Sets

BY CHAITANYA SINGH | FILED UNDER: PYTHON TUTORIAL

Set is an unordered and unindexed collection of items in Python. Unordered means when we display the elements of a set, it will come out in a random order.

Unindexed means, we cannot access the elements of a set using the indexes like we can do in list and tuples.

The elements of a set are defined inside square brackets and are separated by commas. For example –

```
myset = [1, 2, 3, 4, "hello"]
```

Python Set Example

```
# Set Example
myset = {"hi", 2, "bye", "Hello World"}
print(myset)
```

Checking whether an item is in the set

We can check whether an item exists in Set or not using “in” operator as shown in the following example. This returns the boolean value true or false. If the item is in the given set then it returns true, else it returns false.

```
# Set Example
myset = {"hi", 2, "bye", "Hello World"}

# checking whether 2 is in myset
print(2 in myset)

# checking whether "hi" is in myset
print("hi" in myset)

# checking whether "BeginnersBook" is in myset
print("BeginnersBook" in myset)
```

Loop through the elements of a Set in Python

We can loop through the elements of a set in Python as shown in the following elements. As you can see in the output that the elements will appear in random order each time you run the code.

```
# Set Example
myset = {"hi", 2, "bye", "Hello World"}

# loop through the elements of myset
for a in myset:
    print(a)
```

Python - Add or remove item from a Set

We can add an item in a Set using add() function and we can remove an item from a set using remove() function as shown in the following example.

```
# Set Example
myset = {"hi", 2, "bye", "Hello World"}
print("Original Set:", myset)

# adding an item
myset.add(99)
print("Set after adding 99:", myset)

# removing an item
myset.remove("bye")
print("Set after removing bye:", myset)
```

Set Methods

1. add(): This method adds an element to the Set.
2. remove(): This method removes a specified element from the Set
3. discard(): This method works same as remove() method, however it doesn't raise an error when the specified element doesn't exist.
4. clear(): Removes all the elements from the set.
5. copy(): Returns a shallow copy of the set.
6. difference(): This method returns a new set which is a difference between two given sets.
7. difference_update(): Updates the calling set with the Set difference of two given sets.
8. intersection(): Returns a new set which contains the elements that are common to all the sets.
9. intersection_update(): Updates the calling set with the Set intersection of two given sets.
10. isdisjoint(): Checks whether two sets are disjoint or not. Two sets are disjoint if they have no common elements.
11. issubset(): Checks whether a set is a subset of another given set.
12. pop(): Removes and returns a random element from the set.
13. union(): Returns a new set with the distinct elements of all the sets.
14. update(): Adds elements to a set from other passed iterable.
15. symmetric_difference(): Returns a new set which is a symmetric difference of two given sets.
16. symmetric_difference_update(): Updates the calling set with the symmetric difference of two given sets.

Python OOPs Concepts

Python is an **object-oriented programming language**. What this means is we can solve a problem in Python by creating objects in our programs. In this guide, we will discuss OOPs terms such as **class**, **objects**, **methods** etc. along with the Object oriented programming features such as **inheritance**, **polymorphism**, **abstraction**, **encapsulation**.

Object

An object is an entity that has attributes and behaviour. For example, `Ram` is an object who has attributes such as height, weight, color etc. and has certain behaviours such as walking, talking, eating etc.

Class

A class is a blueprint for the objects. For example, Ram, Shyam, Steve, Rick are all objects so we can define a template (blueprint) class `Human` for these objects. The class can define the common attributes and behaviours of all the objects.

Methods

As we discussed above, an object has attributes and behaviours. These behaviours are called methods in programming.

Example of Class and Objects

In this example, we have two objects `Ram` and `Steve` that belong to the class `Human`
Object attributes: name, height, weight
Object behaviour: eating()

Source code

```
class Human:
    # instance attributes
    def __init__(self, name, height, weight):
        self.name = name
        self.height = height
        self.weight = weight

    # instance methods (behaviours)
    def eating(self, food):
        return "{} is eating {}".format(self.name, food)


# creating objects of class Human
ram = Human("Ram", 6, 60)
steve = Human("Steve", 5.9, 56)

# accessing object information
print("Height of {} is {}".format(ram.name, ram.height))
print("Weight of {} is {}".format(ram.name, ram.weight))
print(ram.eating("Pizza"))
print("Weight of {} is {}".format(steve.name, steve.height))
print("Weight of {} is {}".format(steve.name, steve.weight))
print(steve.eating("Big Kahuna Burger"))
```

Output:

```
Height of Ram is 6
Weight of Ram is 60
Ram is eating Pizza
Weight of Steve is 5.9
Weight of Steve is 56
Steve is eating Big Kahuna Burger
```

How to create Class and Objects in Python

In the previous guide, we discussed Object-oriented programming in Python. In this tutorial, we will see how to create classes and objects in Python.

Define class in Python

A class is defined using the keyword `class`.

Example

In this example, we are creating an empty class `DemoClass`. This class has no attributes and methods.

The string that we mention in the triple quotes is a docstring which is an optional string that briefly explains the purpose of the class.

```
class DemoClass:
    """This is my docstring, this explains brief about the class"""

# this prints the docstring of the class
```

```
print(DemoClass.__doc__)
```

Output:

```
This is my docstring, this explains brief about the class
```

Creating Objects of class

In this example, we have a class `MyNewClass` that has an attribute `num` and a function `hello()`. We are creating an object `obj` of the class and accessing the attribute value of object and calling the method hello() using the object.

```
class MyNewClass:
    """This class demonstrates the creation of objects"""

    # instance attribute
    num = 100

    # instance method
    def hello(self):
        print("Hello World!")


# creating object of MyNewClass
obj = MyNewClass()

# prints attribute value
print(obj.num)

# calling method hello()
obj.hello()

# prints docstring
print(MyNewClass.__doc__)
```

Output:

```
100
Hello World!
This class demonstrates the creation of objects
```

Python Constructors – default and parameterized

A constructor is a special kind of method which is used for initializing the instance variables during object creation. In this guide, we will see what is a constructor, types of it and how to use them in the python programming with examples.

1. What is a Constructor in Python?

Constructor is used for initializing the instance members when we create the object of a class.

For example:
Here we have a instance variable `num` which we are initializing in the constructor. The constructor is being invoked when we create the object of the class (obj in the following example).

```
class DemoClass:
    # constructor
    def __init__(self):
        # initializing instance variable
        self.num=100

    # a method
    def read_number(self):
        print(self.num)


# creating object of the class. This invokes constructor
obj = DemoClass()

# calling the instance method using the object obj
obj.read_number()
```
Output:

```
100
```

Syntax of constructor declaration

As we have seen in the above example that a constructor always has a name `init` and the name init is prefixed and suffixed with a double underscore(__). We declare a constructor using `def` keyword, just like methods.

```
def __init__(self):
    # body of the constructor
```
2. Types of constructors in Python

We have two types of constructors in Python.
1. default constructor – this is the one, which we have seen in the above example. This constructor doesn’t accept any arguments.
2. parameterized constructor – constructor with parameters is known as parameterized constructor.

Python – default constructor example

Note: An object cannot be created if we don't have a constructor in our program. This is why when we do not declare a constructor in our program, python does it for us. Lets have a look at the example below.

Example: When we do not declare a constructor

In this example, we do not have a constructor but still we are able to create an object for the class. This is because there is a default constructor implicitly injected by python during program compilation, this is an empty default constructor that looks like this:

```
def __init__(self):
    # no body, does nothing.
```

Source Code:

```
class DemoClass:
    num = 101

    # a method
    def read_number(self):
        print(self.num)


# creating object of the class
obj = DemoClass()

# calling the instance method using the object obj
obj.read_number()
```

Output:

```
101
```

Example: When we declare a constructor

In this case, python does not create a constructor in our program.

```
class DemoClass:
    num = 101

    # non-parameterized constructor
    def __init__(self):
        self.num = 999

    # a method
    def read_number(self):
        print(self.num)


# creating object of the class
obj = DemoClass()
```

```
# calling the instance method using the object obj
obj.read_number()
```

Output:

```
999
```

Python - Parameterized constructor example

When we declare a constructor in such a way that it accepts the arguments during object creation then such type of constructors are known as Parameterized constructors. As you can see that with such type of constructors we can pass the values (data) during object creation, which is used by the constructor to initialize the instance members of that object.

```
class DemoClass:
    num = 101

    # parameterized constructor
    def __init__(self, data):
        self.num = data

    # a method
    def read_number(self):
        print(self.num)


# creating object of the class
# this will invoke parameterized constructor
obj = DemoClass(55)

# calling the instance method using the object obj
obj.read_number()

# creating another object of the class
obj2 = DemoClass(66)

# calling the instance method using the object obj
obj2.read_number()
```

Output:

Python Classes and Methods

Python is an “object-oriented programming language.” This means that almost all the code is implemented using a special construct called classes. Programmers use classes to keep related things together. This is done using the keyword “class,” which is a grouping of object-oriented constructs.

By the end of this tutorial you will be able to:

1. Define what is a class
2. Describe how to create a class
3. Define what is a method
4. Describe how to do object instantiation
5. Describe how to create instance attributes in Python

What is a class?

A class is a code template for creating objects. Objects have member variables and have behaviour associated with them. In python a class is created by the keyword `class`.

An object is created using the constructor of the class. This object will then be called the `instance` of the class. In Python we create instances in the following manner

```
Instance = class(arguments)
```

How to create a class

The simplest class can be created using the class keyword. For example, let's create a simple, empty class with no functionalities.

```
>>> class Snake:
...     pass
...
>>> snake = Snake()
>>> print(snake)
<_main_.Snake object at 0x7f315c573550>
```

Attributes and Methods in class:

A class by itself is of no use unless there is some functionality associated with it. Functionalities are defined by setting attributes, which act as containers for data and functions related to those attributes. Those functions are called methods.

Attributes:

You can define the following class with the name Snake. This class will have an attribute `name`.

```
>>> class Snake:
...     name = "python" # set an attribute `name` of the class
...
```

You can assign the class to a variable. This is called object instantiation. You will then be able to access the attributes that are present inside the class using the dot `.` operator. For example, in the Snake example, you can access the attribute `name` of the class `Snake`.

```
>>> # instantiate the class Snake and assign it to variable snake
>>> snake = Snake()

>>> # access the class attribute name inside the class Snake.
>>> print(snake.name)
python
```

Methods

Once there are attributes that “belong” to the class, you can define functions that will access the class attribute. These functions are called methods. When you define methods, you will need to always provide the first argument to the method with a self keyword.

For example, you can define a class `Snake`, which has one attribute `name` and one method `change_name`. The method change name will take in an argument `new_name` along with the keyword `self`.

```
>>> class Snake:
...     name = "python"
...
...     def change_name(self, new_name): # note that the first argument is
self
...         self.name = new_name # access the class attribute with the self
keyword
...
```

Now, you can instantiate this class `Snake` with a variable `snake` and then change the name with the method `change_name`.

```
>>> # instantiate the class
>>> snake = Snake()

>>> # print the current object name
>>> print(snake.name)
python

>>> # change the name using the change_name method
>>> snake.change_name("anaconda")
>>> print(snake.name)
anaconda
```

Instance attributes in python and the init method

You can also provide the values for the attributes at runtime. This is done by defining the attributes inside the init method. The following example illustrates this.

```
class Snake:

    def_init_(self, name):
        self.name = name

    def change_name(self, new_name):
        self.name = new_name
```

Now you can directly define separate attribute values for separate objects. For example,

```
>>> # two variables are instantiated
>>> python = Snake("python")
>>> anaconda = Snake("anaconda")

>>> # print the names of the two variables
>>> print(python.name)
python
>>> print(anaconda.name)
anaconda
```

Python classes and object object-oriented programming II

Classes are written to organize and structure code into meaningful blocks, which can then be used to implement the business logic. These implementations are used in such a way that more complex parts are abstracted away to provide for simpler interfaces which can then be used to build even simpler blocks. While doing this we will find that there are lots of times when we will need to establish relationships between the classes that we build. These relationships can then be established using either inheritance or composition. At this point it is best you take a look at our [Python Classes tutorial][1] to get in-depth knowledge on how classes are written in Python. Also, in case you are already doing object oriented programming in some other language, you may want to check out our notes on design patterns.

In this tutorial you will get to know how to build relationships between classes using inheritance and composition and the syntax that is needed.

Python inheritance

What is Inheritance

In inheritance an object is based on another object. When inheritance is implemented, the methods and attributes that were defined in the base class will also be present in the inherited class. This is generally done to abstract away similar code in multiple classes. The abstracted code will reside in the base class and the previous classes will now inherit from the base class.

How to achieve Inheritance in Python

Python allows the classes to inherit commonly used attributes and methods from other classes through inheritance. We can define a base class in the following manner:

```
class DerivedClassName(BaseClassName):
    pass
```

Let's look at an example of inheritance. In the following example, Rocket is the base class and MarsRover is the inherited class.

```
class Rocket:
    def_init_(self, name, distance):
        self.name = name
        self.distance = distance

    def launch(self):
        return "%s has reached %s" % (self.name, self.distance)


class MarsRover(Rocket): # inheriting from the base class
    def_init_(self, name, distance, maker):
        Rocket._init_(self, name, distance)
        self.maker = maker

    def get_maker(self):
        return "%s Launched by %s" % (self.name, self.maker)


if_name_== "_main__":
    x = Rocket("simple rocket", "till stratosphere")
    y = MarsRover("mars_rover", "till Mars", "ISRO")
    print(x.launch())
    print(y.launch())
    print(y.get_maker())
```

The output of the code above is shown below:

```
➜ Documents python rockets.py
simple rocket has reached till stratosphere
mars_rover has reached till Mars
mars_rover Launched by ISRO
```

Python Composition:

What is composition

In composition, we do not inherit from the base class but establish relationships between classes through the use of instance variables that are references to other objects. Talking in terms of pseudocode you may say that

```
class GenericClass:
    define some attributes and methods

class ASpecificClass:
    Instance_variable_of_generic_class = GenericClass

# use this instance somewhere in the class
    some_method(Instance_variable_of_generic_class)
```

So you will instantiate the base class and then use the instance variable for any business logic.

How to achieve composition in Python

To achieve composition you can instantiate other objects in the class and then use those instances. For example in the below example we instantiate the Rocket class using `self.rocket` and then using self.rocket in the method `get_maker`.

```
class MarsRoverComp():
    def_init_(self, name, distance, maker):
        self.rocket = Rocket(name, distance) # instantiating the base

        self.maker = maker

    def get_maker(self):
        return "%s Launched by %s" % (self.rocket.name, self.maker)


if___name___== "_main__":
    z = MarsRover("mars_rover2", "till Mars", "ISRO")
    print(z.launch())
    print(z.get_maker())
```

The output of the total code which has both inheritance and composition is shown below:

```
➜ Documents python rockets.py
simple rocket has reached till stratosphere
mars_rover has reached till Mars
mars_rover Launched by ISRO
mars_rover2 has reached till Mars
mars_rover2 Launched by ISRO
```

Errors and Exceptions in Python

Errors are problems in the program that the program should not recover from. If at any point in the program an error occurs, then the program should exit gracefully. On the other hand, Exceptions are raised when an external event occurs which in some way changes the normal flow of the program.

In this tutorial you will learn about common types of Errors and Exceptions in Python and common paradigms in handling them.

Handling Exceptions with Try/Except/Finally

Errors and Exceptions in Python are handled with the `Try: Except: Finally` construct. You put the unsafe code in the `try:` block. You put the fall-back code in the `Except:` block. The final code is kept in the `Finally:` block.

For example, look at the code below.

```
>>> try:
...     print("in the try block")
...     print(1/0)
... except:
...     print("In the except block")
... finally:
...     print("In the finally block")
...
in the try block
In the except block
In the finally block
```

Raising exceptions for a predefined condition

Exceptions can also be raised if you want the code to behave within specific parameters. For example, if you want to limit the user-input to only positive integers, raise an exception.

```
# exc.py
while True:
    try:
        user = int(input())
        if user < 0:
        raise ValueError("please give positive number")
        else:
        print("user input: %s" % user)
except ValueError as e:
        print(e)
```

So the output of the above program is:

```
➜  python exc.py
4
user input: 4
3
user input: 3
2
user input: 2
1
user input: 1
-1
please give positive number
5
user input: 5
2
user input: 2
-5
please give positive number
^C
Traceback (most recent call last):
File "exc.py", line 3, in <module>
  user = int(input())
KeyboardInterrupt
```

Python Iterators, generators, and the for loop

Iterators are containers for objects so that you can loop over the objects. In other words, you can run the "for" loop over the object. There are many iterators in the Python standard library. For example, list is an iterator and you can run a for loop over a list.

```
>>> for lib in popular_python_libs:
...     print(lib)
...
requests
scrapy
pillow
SQLAlchemy
NumPy
```

In this tutorial you will get to know:

1. How to create a custom iterator
2. How to create a generator
3. How to run for loops on iterators and generators

Python Iterators and the Iterator protocol

To create a Python iterator object, you will need to implement two methods in your iterator class.

`__iter_`: This returns the iterator object itself and is used while using the "for" and "in" keywords.

`__next_`: This returns the next value. This would return the StopIteration error once all the objects have been looped through.

Let us create a cool emoticon generator and l iterators.

```
# iterator_example.py
"""
This should give an iterator with a emoticon.
"""

import random

class CoolEmoticonGenerator(object):
    """docstring for CoolEmoticonGenerator."""

    strings = "!@#$^*_-=+?/,.:;~"
    grouped_strings = [("(", ")"), ("<", ">"), ("[", "]"), ("{", "}")]

    def create_emoticon(self, grp):
        """actual method that creates the emoticon"""
        face_strings_list = [random.choice(self.strings) for _ in range(3)]
        face_strings = "".join(face_strings_list)
        emoticon = (grp[0], face_strings, grp[1])
        emoticon = "".join(emoticon)
        return emoticon

    def __iter__(self):
        """returns the self object to be accessed by the for loop"""
        return self

    def_next__(self):
        """returns the next emoticon indefinitely"""
        grp = random.choice(self.grouped_strings)
        return self.create_emoticon(grp)
```

Now you can call the above class as an iterator. Which means you can run the next function on it.

```
from iterator_example import CoolEmoticonGenerator
g = CoolEmoticonGenerator()
print([next(g) for _ in range(5)])
```

Running the program above gives us the following output. The exact output may be different from what you get but it will be similar.

```
➜  python3.5 iterator_example.py
['<,~!>', '<;_~>', '<!;@>', '[~=#]', '{?^-}']
```

You can use the KeyboardInterrupt to stop the execution.

Python Generators

Python generator gives us an easier way to create python iterators. This is done by defining a function but instead of the return statement returning from the function, use the "yield" keyword. For example, see how you can get a simple vowel generator below.

```
>>> def vowels():
...     yield "a"
...     yield "e"
...     yield "i"
...     yield "o"
...     yield "u"
...
>>> for i in vowels():
...     print(i)
...
a
e
i
o
u
```

Now let's try and create the CoolEmoticonGenerator.

```
def create_emoticon_generator():
    while True:
        strings = "!@#$^*_-=+?/,.:;~"
        grouped_strings = [("(", ")"), ("<", ">"), ("[", "]"), ("{", "}")]
        grp = random.choice(grouped_strings)
        face_strings_list = [random.choice(strings) for _ in range(3)]
        face_strings = "".join(face_strings_list)
        emoticon = (grp[0], face_strings, grp[1])
        emoticon = "".join(emoticon)
        yield emoticon
```

Now, if you run the generator using the runner below

```
from iterator_example import CoolEmoticonGenerator
g = create_emoticon_generator()
print([next(g) for _ in range(5)])
```

You should get the following output

```
➜  python3.5 iterator_example.py
['(+~?)', '<**_>', '($?/)', '[#=+]', '{*=.}']
```

Functions

A function is a block of code that takes in some data and, either performs some kind of transformation and returns the transformed data, or performs some task on the data, or both. Functions are useful because they provide a high degree of modularity. Similar code can be easily grouped into functions and you can provide a name to the function that describes what the function is for. Functions are the simplest, and, sometimes the most useful, tool for writing modular code.

In this tutorial you will get to know:

- How to create a function
- How to call a function

How to create a function:

In Python to create a function, you need to write it in the following manner. Please note that the body of the function is indented by 4 spaces.

```
def name_of_the_function(arguments):
    '''
    doctring of the function
    note that the function block is indented by 4 spaces
    '''
    body of the function
    return the return value or expression
```

You can look at the example below where a function returns the sum of two numbers.

```
def add_two_numbers(num1, num2):
    '''returns the sum of num1 and num2'''
    result = num1 + num2
    return result
```

Here are all the parts of the function:

Keyword `def`: This is the keyword used to say that a function will be defined now, and the next word that is there, is the function name.

Function name: This is the name that is used to identify the function. The function name comes after the `def`keyword. Function names have to be a single word. PEP8, which is a style guide for Python, recommends that in case multiple words are used, they should be in lowercase and they should be separated with an underscore. In the example above, `add_two_numbers` is the parameter name.

Parameter list: Parameter list are place holders that define the parameters that go into the function. The parameters help to generalise the transformation/computation/task that is needed to be done. In Python, parameters are enclosed in parentheses. In the example above, the parameters are `num1`and `num2`. You can pass as many parameters as needed to a function.

Function docstrings: These are optional constructs that provide a convenient way for associated documentation to the corresponding function. Docstrings are enclosed by triple quotes `'''you will write the docstring here'''`

Function returns: Python functions returns a value. You can define what to return by the `return` keyword. In the example above, the function returns `result`. In case you do not define a return value, the function will return None.

How to call a function

Call a function with a return value

To call a function means that you are telling the program to execute the function. If there is a return value defined, the function would return the value, else the function would return None. To call the function, you write the name of the function followed by parentheses. In case you need to pass parameters/arguments to the function, you write them inside the parentheses.

For example, if you had a function that added two numbers

```
def add_two_numbers(num1, num2):
    '''returns the sum of num1 and num2'''
    result = num1 + num2
    return result
```

You would call the function like this:

```
add_two_numbers(1, 2)
```

Note that arguments 1 and 2 have been passed. Hence, the return value will be 3. You can put any two numbers in place of 1 and 2, and it will return the corresponding sum of the two numbers. But calling a function and not doing anything with the result is meaningless, isn’t it? So you can now assign it to a variable which may be used later on. In the following example, can just printing it.

```
>>> def add_two_numbers(num1, num2):
...     '''returns the sum of num1 and num2'''
...     result = num1 + num2
...     return result
...
>>> # call the function add_two_numbers with arguments 4 and 5 and assign it
>>> # to a variable sum_of_4_and_5
>>> sum_of_4_and_5 = add_two_numbers(4, 5)
>>>
>>> # show the value stored in sum of 4 and 5
>>> print(sum_of_4_and_5)
9
```

Call a function that performs a task and has no return value

In case the function is not meant to return anything and just performs some task, like committing something to a database or changing the text of some button in the user interface, then you do not need to assign the function to a variable. You can just call the function.

For example, if you had a function that prints a string

```
def printing_side_effects():
    '''a function with side effects'''
    print('this is a function with side effects and performs some task')
```

You can just call the function and it will get executed.

```
>>> printing_side_effects()
```

this is a function with side effects and performs some task

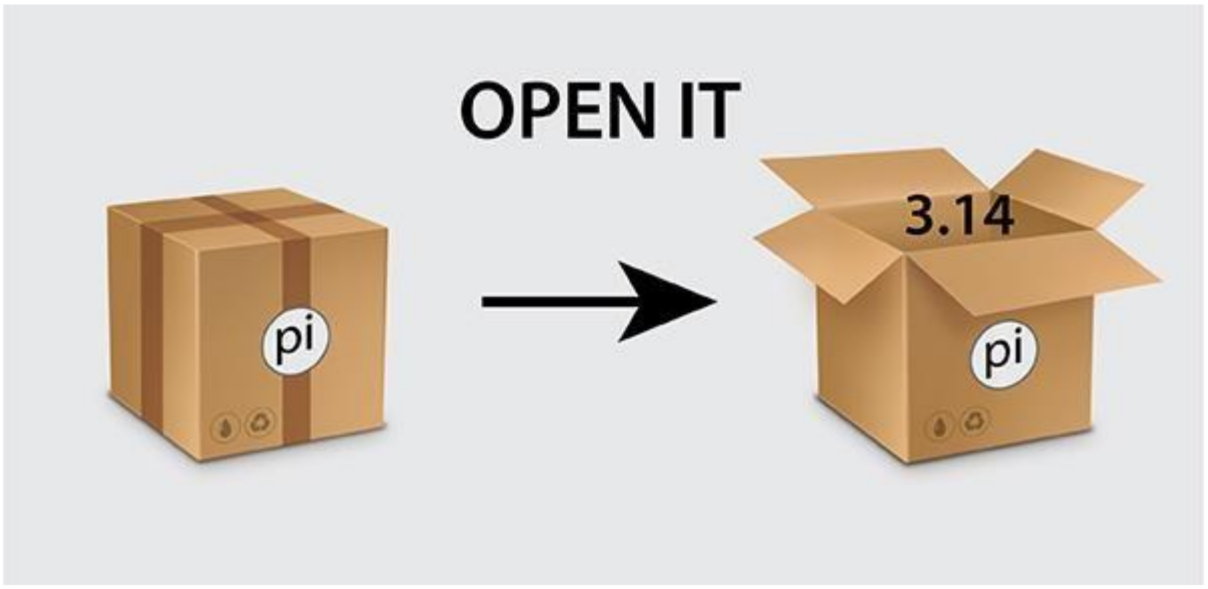
How to call a function with arguments

Note that in this case you pass parameters in the order in which they are supposed to be processed. For example, if you had a function that duplicates a string by the number of times, where both the string and the number needs to be provided by the function, such as:

```
>>> def string_multiplier(string_arg, number):
...     '''takes the string_arg and multiplies it with one more than the
number'''
...     return string_arg * (number + 1)
...

>>> # passing string_arg and number and in that order...
>>> print(string_multiplier('a', 5))aaaaaa

>>> # below code will return error as the arguments are not in order...
>>> print(string_multiplier(5, 'a'))Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in string_multiplier
TypeError: must be str, not int
```

Variables

A variable can be considered a storage container for data. Every variable will have a name. For example, we can have a variable named `speed_of_light`. A variable is a good way to store information while making it easy to refer to that information in our code later. A close analogy to variables may be a named box where you can store information.



For instance, instead of working with the number 3.14, we can assign it to a variable `pi`. You may forget that you need to use the number 3.14 when you will need to make relevant calculations later. On the other hand, it will be easier for you to remember to call `pi` when writing the code.

In this tutorial, you will learn how to name a variable and assign values. You will take a closer look at the methods that variables can support.

Assignment

In Python, assignment can be done by writing the variable name followed by the value separated by an equal `=`symbol. The skeleton or pseudo-code is

```
"Variable name" = " value or information "
```

In the following examples, you assign various numbers and strings to variables.

```
>>> # assign the value 299792458 to the variable speed_of_light
>>> speed_of_light = 299792458
>>> print(speed_of_light)
299792458

>>> # assign a decimal number 3.14 to the variable pi
>>> pi = 3.14
>>> print(pi)
3.14
```

```
>>> # assign a string
>>> fav_lang = "python"
>>> print(fav_lang)
'python'
```

Valid and invalid ways of assigning variables

Multiple words Assignment only works when the variable is a single word.

```
>>> multiple word = "multiple word"
  File "<stdin>", line 1
    multiple word = "multiple word"
                ^
SyntaxError: invalid syntax
```

So, if you want to have more than one word in the name, the convention is to use underscore "_" in the name.

```
>>> multiple_word = "multiple word" # note the variable name has an
underscore _
>>> print(multiple_word)
multiple word
```

Do not start with a number

You cannot start a variable name with a number. The rest of the variable name can contain a number.

For example, 1var is wrong.

```
>>> 1var = 1
  File "<stdin>", line 1
        1var = 1
       ^
SyntaxError: invalid syntax
```

But var1 is fine.

```
>>> var1 = 1
>>> print(var1)
1
```

More points to remember while deciding a variable name

You can only include a-z, A-Z, _, and 0-9 in your variable names. Other special characters are not permitted.

For example, you cannot have hash key `#` in your variable names.

```
>>> # a_var_containing_# will not work as it has # in the name
>>> a_var_containing_# = 1
 Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
NameError: name 'a_var_containing_' is not defined

>>> # but if we remove the # then it works
>>> a_var_containing_ = 1
>>> print(a_var_containing_)
1
```

Interestingly, you can have a variable name in your local language.

```
 >>> 零= 0 # chinese
>>> print(零)
0

 >>> ශුන්‍ය= 0 # sinhala
 >>> print(ශුන්‍ය)
 0
```

More on Assignments

Python supports assigning all data structures to variables.

For example, we can assign a list to a variable like you see in the following example, where we assign the list of names, denoted by `[...]`, to the variable `fav_writers`.

```
>>> # assigning list
>>> fav_writers = ["Mark Twain", "Fyodor Dostoyevsky"]
>>> print(fav_writers)
['Mark Twain', 'Fyodor Dostoyevsky']
```

Here is another example where you can assign dicts, shown by `{...}`, to a variable `birthdays`.

```
>>> # assign dicts
...
>>> birthdays = {"mom": "9Jan", "daughter": "24Dec"}
>>> print(birthdays)
{'mom': '9Jan', 'daughter': '24Dec'}
```

Data structures such as lists and dicts will be discussed in later tutorials.

You can also assign functions and classes to variables.

You will know more about functions and classes in later tutorials.

```
>>> # assigning functions
...
>>> import functools
>>> memoize = functools.lru_cache
>>> print(memoize)
<function lru_cache at 0x7fb2a6b42f28>

>>> # class assignment
...
>>> class MyClass:
...     pass
...
>>> give_me_more = MyClass()
>>> print(give_me_more)
<_main_.MyClass object at 0x7f512e65bfd0>
```

Working with variables

Variables will support any method the underlying type supports. For example, if an integer value is stored in a variable, then the variable will support integer functions such as addition.

In the following example, you assign the number 2 to the variable `var` and then add `3` to `var`. This will print `5`, the result of 3 being added to the value stored in `var` which is `2`.

```
>>> var = 2
>>> print(var + 3)
5
```

You can make a change in a variable and assign it to the same variable. This is done generally when some kind of data type change is done.

For example, you can take a number as input. This will take in the digit as a string. You can then take the string number and convert it to int and assign it to the same number.

```
>>> number = input()
2
>>> type(number)
<class 'str'>
>>> number = int(number)
>>> type(number)
<class 'int'>
```

We will use a function `range(3)` which returns three values.

```
>>> print(range(3))
[0, 1, 2]
```

Something that returns three values can be unpacked to three variables. This is like saying take whatever is in `range(3)` and instead of assigning it to a single variable, break it up and assign individual values to the three variables. This is done using a comma between the variables.

```
>>> id1, id2, id3 = range(3)
>>> print(id1)
0
>>> print(id2)
1
>>> print(id3)
2
```

Python String:

Strings are sequences of characters. Your name can be considered a string. Or, say you live in Zambia, then your country name is `"Zambia"`, which is a string.

In this tutorial you will see how strings are treated in Python, the different ways in which strings are represented in Python, and the ways in which you can use strings in your code.

How to create a string and assign it to a variable

To create a string, put the sequence of characters inside either single quotes, double quotes, or triple quotes and then assign it to a variable. You can look into how variables work in Python in the Python variables tutorial.

For example, you can assign a character 'a' to a variable `single_quote_character`. Note that the string is a single character and it is "enclosed" by single quotes.

```
>>> single_quote_character = 'a'
>>> print(single_quote_character)
a
>>> print(type(single_quote_character)) # check the type of the variable.
<class 'str'>
```

Similarly, you can assign a single character to a variable `double_quote_character`. Note that the string is a single character but it is "enclosed" by double quotes.

```
>>> double_quote_character = "b"
>>> print(double_quote_character)
b
>>> print(type(double_quote_character))
<class 'str'>
```

Also check out if you can assign a sequence of characters or multiple characters to a variable. You can assign both single quote sequences and double quote sequences.

```
>>> double_quote_multiple_characters = "aeiou"
>>> single_quote_multiple_characters = 'aeiou'
>>> print(type(double_quote_multiple_characters),
type(single_quote_multiple_characters))
<class 'str'> <class 'str'>
```

Interestingly if you check the equivalence of one to the other using the keyword `is`, it returns True.

```
>>> print(double_quote_multiple_characters is
double_quote_multiple_characters)
True
```

Take a look at assignment of strings using triple quotes and check if they belong to the class `str` as well.

```
>>> triple_quote_example = """this is a sentence written in triple quotes"""
>>> print(type(triple_quote_example))
<class 'str'>
```

In the examples above, the function `type` is used to show the underlying class that the object will belong to. Please note that all the variables that have been initiated with single, double, or triple quotes are taken as string. You can use single and double quotes for a single line of characters. Multiple lines are generally put in triple quotes.

String common methods

- Get the index of a substring in a string.

```
# find the index of a "c" in a string "abcde"
>>>  "abcde".index("c")
```

```
2
```

2 is returned because the position of the individual letters in the strings is 0-indexed. So, index of "a" in "abcde" is 0, that of "b" is 1, and so on.

- Test if a substring is a member of a larger string. This is done using the keyword `in` and writing the test. The skeleton is shown below.

  substring in string

```
>>> # for example, test if string "i" is present in string "pythonic"
at least once. "i" is present in the string. Therefore, the result
should be true.
>>>  "i" in "pythonic"
True
>>> # as "x" is not present in the string "pythonic" the below test
should return false
>>> "x" in "pythonic" # "x" is not present in "pythonic"
False
```

- Join a list of strings using the join method. A list of strings is written by delimiting the sequence with a comma `,`, and enclosing the whole group with brackets `[...]`. For a more detailed tutorial on lists head over to the python lists tutorial. You can join a list of strings by giving the delimiter as the object on which the method `join` will act and the list of strings as the argument.

```
• >>> # join a list of strings 1, 2, 3 with a space as a delimiter and
  1,2,3 as the list of strings. So, the result will be the strings with
  spaces between them.
• >>>  combined_string = " ".join(["1", "2", "3"])
```

```
'1 2 3'
```

- Break a string based on some rule. This takes in the string as the object on which the method `split` is passed using the dot operator. Splitting takes a space as the default parameter.

For example you can split a string based on the spaces between the individual values.

```
    >>> # split the string "1 2 3" and return a list of the numbers.
    >>>  "1 2 3".split() # splitting
    ['1', '2', '3']
```

Or you can split a string based on a delimiter like `:`.

```
     >>> “1:2:3”.split(“:”)
     [‘1’, ‘2’, ‘3’]
```

- Access individual characters in a string. Note the first element has index `0`. You access the first element with the index `0`, second element with the index `1`, and so on.

```
• >>>  lang = "python"
•   >>>  print(lang[0])
•   >>>  print(lang[2]) # access the 3rd letter
•   't'
•   >>>  print(lang[-3]) # access the third letter from the end.
```

```
    'h'
```

Formatting in String:

String object can be formatted. You can use `%s` as a formatter which will enable you to insert different values into a string at runtime and thus format the string. The `%s` symbol is replaced by whatever is passed to the string.

```
    >>>  print("I love %s in %s" % ("programming", "Python")) # templating
strings
    'I love programming in Python'
```

You can also use the keyword `format`. This will enable you to set your own formatters instead of `%s`.

```
>>> print("I love {programming} in
{python}".format(programming="programming", python="Python"))
'I love programming in Python'
```

Truth value testing of String

A string is considered to be true in Python if it is not an empty string. So, we get the following:

```
# Test truth value of empty string
>>>  print(bool(""))
False

# Test truth value of non-empty string "x"
>>>  print(bool("x"))
True
```

Python Control Structures - Loops and Conditionals

You can control the flow of logic in your code through various methods.

Basic control flows

- Selection (if statements)
- Iteration (for loops)

More advanced control flows

- Procedural Abstraction (functions)
- Recursion
- Concurrency
- Exception Handling and Speculation
- Nondeterminacy

In this tutorial you will come to know:

How to have sequential, selective and iterative flows in your code. This can be achieved using the for loop. How to achieve procedural abstraction. This can be done by the use of functions.

Other topics like Recursion, Exception Handling, Concurrency will be discussed in later tutorials.

Loops

Working on items of the iterable

If you want to run an operation on a collection of items, then you can do it using for loops. The skeleton for using the for loop is shown below.Note that the for statement line will end with a colon `:` and the rest of the code block must be indented with a spacing of 4 spaces. An iterable is any object that can be looped on such as list, tuple, string etc.

```
for item in iterable: # you can place any list or tuple or string in place of
iterable
    # write your code here.
    pass
```

If you want to print an element of a list of fruits, you can write the following code to achieve that.

```
>>> fruits = ["apples", "oranges", "mangoes"]
>>> for fruit in fruits:
...     print(fruit)
...
apples
oranges
mangoes
```

In the example above, note that items in the iterable (i.e fruits) will be assigned to the for loop variable (i.e fruit) during the iteration process. So, we can access the item directly.

```
>>> fruits = ["apples", "oranges", "mangoes"]
>>> for fruit in fruits:
```

```
...     string_size = 0
...     for alphabet in fruit:
...         string_size += 1
...     print("name of fruit: %s is has length %s" % (fruit, string_size))
...
name of fruit: apples is has length 6
name of fruit: oranges is has length 7
name of fruit: mangoes is has length 7
```

Looping on both indexes and items

In the previous section, index or the place value of the item in the iterable was not considered. However, if you are interested in working with the index, then you can call the `enumerate` function which returns a tuple of the index and the item. Taking the example above, you can print the name of the fruit and the index of the list of fruits.

```
>>> fruits = ["apples", "oranges", "mangoes"]
>>> for index, fruit in enumerate(fruits):
...     print("index is %s" % index)
...     print("fruit is %s" % fruit)
...     print("###########################")
...
index is 0
fruit is apples
###########################
index is 1
fruit is oranges
##########################
index is 2
fruit is mangoes
##########################
```

While statement

The while statement will execute a block of code as long as the condition is true. The skeleton of a while block is shown below.

```
while condition:
    code_block
```

Note that similar to the for loop, the `while` statement ends with a colon `:` and the remaining code block is indented by 4 spaces. We can implement the fruit example in the while block as well, although the logic becomes a bit complicated than the for block.

```
>>> fruits = ["apples", "oranges", "mangoes"] # get the list
>>> length = len(fruits) # get the length that will be needed for the while
condition
>>> i = 0 # initialise a counter
>>> while i < length: # give the condition
```

```
...         print(fruits[i]) # the code block
...         i += 1 # increment the counter
...
apples
oranges
mangoes
```

Nested for loops

You can have one or more nested for loops. For example, look at the following example where you can print the multiplication table. The table is shown only for 1 and 2 to save space. You can try for the remaining digits.

```
>>> for i in range(1,3):
...         for j in range(1,3):
...                 print('%d x %d = %d' % (i, j, i*j))
...
1 x 1 = 1
1 x 2 = 2
2 x 1 = 2
2 x 2 = 4
```

Selection and Python If statements

Creating if blocks

As a programmer, you will continually feel the need to control the flow of your program and let it make runtime decisions based on some condition. The is done using the if syntax. To implement this you can look at the if .. elif .. else syntax.

```
if condition1:
        code_block1
elif condition2:
        code_block2
else:
        code_block3
```

You can try the following example to understand better .

```
>>> num = 42
>>> if num == 42: # condition
...     print("number is 42") # direction 1
...
number is 42
```

Adding an else block:

```
>>> num = 43
>>> if num == 42:
...         print("number is 42")
...     else:
...         print("number if not 42")
...
number if not 42
```

Now, let us add an elif block to it as well and see what happens:

```
>>> num = 44
>>> if num == 42:
...         print("number is 42")
...     elif num == 44:
...         print("num is 44")
...     else:
...         print("num is neither 42 nor 44")
...
num is 44
```

Nested if statements

You can have one or more nested if blocks inside if statements.

```
>>> num = 42
>>> if num > 20:
...         if num < 50:
...                 print("num between 20 and 50")
...
num between 20 and 50
```

Lists

A list is a data-structure, or it can be considered a container that can be used to store multiple data at once. The list will be ordered and there will be a definite count of it. The elements are indexed according to a sequence and the indexing is done with 0 as the first index. Each element will have a distinct place in the sequence and if the same value occurs multiple times in the sequence, each will be considered separate and distinct element. A more detailed description on lists and associated data-types are covered in this tutorial.

In this tutorial you will come to know of the about how to create python lists and the common paradigms for a python list.

Lists are great if you want to preserve the sequence of the data and then iterate over them later for various purposes. We will cover iterations and for loops in our tutorials on for loops.

How to create a list:

To create a list, you separate the elements with a comma and enclose them with a bracket “[]”.

For example, you can create a list of company names containing “hackerearth”, “google”, “facebook”. This will preserve the order of the names.

```
>>> companies = ["hackerearth", "google", "facebook"]
>>> # get the first company name
>>> print(companies[0])
'hackerearth'
>>> # get the second company name
>>> print(companies[1])
'google'
>>> # get the third company name
>>> print(companies[2])
'facebook'
>>> # try to get the fourth company name
>>> # but this will return an error as only three names
>>> # have been defined.
>>> print(companies[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Trying to access elements outside the range will give an error. You can create a two-dimensional list. This is done by nesting a list inside another list. For example, you can group “hackerearth” and “paytm” into one list and “tcs” and “cts” into another and group both the lists into another “master” list.

```
>>> companies = [["hackerearth", "paytm"], ["tcs", "cts"]]
>>> print(companies)
[['hackerearth', 'paytm'], ['tcs', 'cts']]
```

Methods over Python Lists

Python lists support common methods that are commonly required while working with lists. The methods change the list in place. (More on methods in the classes and objects tutorial). In case you want to make some changes in the list and keep both the old list and the changed list, take a look at the functions that are described after the methods.

How to add elements to the list:

- list.append(elem) - will add another element to the list at the end.

```
• >>> # create an empty list
• >>> companies = []
•
```

```
• >>> # add “hackerearth” to companies
• >>> companies.append(“hackerearth”)
•
• >>> # add "google" to companies
• >>> companies.append("google")
•
• >>> # add "facebook" to companies
• >>> companies.append("facebook")
•
• >>> # print the items stored in companies
• >>> print(companies)
```

```
['hackerearth', 'google', 'facebook']
```

Note the items are printed in the order in which they youre inserted.

- list.insert(index, element) - will add another element to the list at the given index, shifting the elements greater than the index one step to the right. In other words, the elements with the index greater than the provided index will increase by one.

For example, you can create a list of companies `['hackerearth', 'google', 'facebook']` and insert “airbnb” in third position which is held by “facebook”.

```
>>> # initialise a preliminary list of companies
>>> companies = ['hackerearth', 'google', 'facebook']

>>> # check what is there in position 2
>>> print(companies[2])
facebook

>>> # insert “airbnb” at position 2
>>> companies.insert(2, "airbnb")
>>> # print the new companies list
>>> print(companies)
['hackerearth', 'google', 'airbnb', 'facebook']
>>> # print the company name at position 2
>>> print(companies[2])
airbnb
```

- list.extend(another_list) - will add the elements in list 2 at the end of list.

For example, you can concatenate two lists `["haskell", "clojure", "apl"]` and `["scala", "F#"]` to the same list `langs`.

```
>>> langs = ["haskell", "clojure", "apl"]
>>> langs.extend(["scala", "F#"])
>>> print(langs)
['haskell', 'clojure', 'apl', 'scala', 'F#']
```

- list.index(elem) - will give the index number of the element in the list.

For example, if you have a list of languages with elements `['haskell', 'clojure', 'apl', 'scala', 'F#']` and you want the index of `"scala"`, you can use the index method.

```
>>> index_of_scala = langs.index("scala")
>>> print(index_of_scala)
3
```

How to remove elements from the list:

- list.remove(elem) - will search for the first occurrence of the element in the list and will then remove it.

For example, if you have a list of languages with elements `['haskell', 'clojure', 'apl', 'scala', 'F#']` and you want to remove scala, you can use the remove method.

```
>>> langs.remove("scala")
>>> print(langs)
['haskell', 'clojure', 'apl', 'F#']
```

- list.pop() - will remove the last element of the list. If the index is provided, then it will remove the element at the particular index. For example, if you have a list `[5, 4, 3, 1]` and you apply the method `pop`, it will return the last element `1` and the resulting list will not have it.

```
>>> # assign a list to some_numbers
>>> some_numbers = [5, 4, 3, 1]

>>> # pop the list
>>> some_numbers.pop()
1

>>> # print the present
>>> print(some_numbers) list
[5, 4, 3]
```

Similarly, try to pop an element from a random index that exists in the list.

```
>>> # pop the element at index 1
>>> some_numbers.pop(1)
4
>>> # check the present list
>>> print(some_numbers)
[5, 3]
```

Other useful list methods

- list.sort() - will sort the list in-place.

For example, if you have an unsorted list `[4,3,5,1]`, you can sort it using the `sort` method.

```
>>> # initialise an unsorted list some_numbers
>>> some_numbers = [4,3,5,1]

>>> # sort the list
>>> some_numbers.sort()

>>> # print the list to see if it is sorted.
>>> some_numbers
[1, 3, 4, 5]
```

- list.reverse() - will reverse the list in place

For example, if you have a list `[1, 3, 4, 5]` and you need to reverse it, you can call the `reverse` method.

```
>>> # initialise a list of numbers that
>>> some_numbers = [1, 3, 4, 5]

>>> # Try to reverse the list now
>>> some_numbers.reverse()

>>> # print the list to check if it is really reversed.
>>> print(some_numbers)
[5, 4, 3, 1]
```

Functions over Python Lists:

- You use the function “len” to get the length of the list.

For example, if you have a list of companies `['hackerearth', 'google', 'facebook']` and you want the list length, you can use the `len` function.

```
>>> # you have a list of companies
>>> companies = ['hackerearth', 'google', 'facebook']

>>> # you want the length of the list
>>> print(len(companies))
3
```

- If you use another function “enumerate” over a list, it gives us a nice construct to get both the index and the value of the element in the list.

For example, you have the list of companies `['hackerearth', 'google', 'facebook']` and you want the index, along with the items in the list, you can use the `enumerate` function.

```
    >>> # loop over the companies and print both the index as youll as the
name.
    >>> for indx, name in enumerate(companies):
    ...     print("Index is %s for company: %s" % (indx, name))
    ...
    Index is 0 for company: hackerearth
    Index is 1 for company: google
    Index is 2 for company: facebook
```

In this example, you use the for loop. For loops are pretty common in all programming languages that support procedural constructs. You can head over to A complete theoretical reference to loops in C to have a deeper understanding of for loops. Also look at the tutorial on loops in Python in Python Control Structures tutorial.

- sorted function will sort over the list

Similar to the sort method, you can also use the sorted function which also sorts the list. The difference is that it returns the sorted list, while the sort method sorts the list in place. So this function can be used when you want to preserve the original list as well.

```
    >>> # initialise a list
    >>> some_numbers = [4,3,5,1]
    >>> # get the sorted list
    >>> print(sorted(some_numbers))
    [1, 3, 4, 5]
    >>> # the original list remains unchanged
    >>> print(some_numbers)
    [4, 3, 5, 1]
```

Python Dictionaries

A dictionary is a set of unordered key, value pairs. In a dictionary, the keys must be unique and they are stored in an unordered manner.

In this tutorial you will learn the basics of how to use the Python dictionary.

By the end of the tutorial you will be able to - Create Dictionaries - Get values in a Dictionary - Add and delete elements in a Dictionary - To and For Loops in a Dictionary

Creating a Dictionary

Let’s try to build a profile of three people using dictionaries. To do that you separate the key-value pairs by a colon(“:”). The keys would need to be of an immutable type, i.e., data-types for which the keys cannot be changed at runtime such as int, string, tuple, etc. The values can be of any type. Individual pairs will be separated by a comma(“,”) and the whole thing will be enclosed in curly braces(`{...}`).

For example, you can have the fields “city”, “name,” and “food” for keys in a dictionary and assign the key,value pairs to the dictionary variable person1_information.

```
>>> person_information = {'city': 'San Francisco', 'name': 'Sam', "food":
"shrimps"}
>>> type(person1_information)
<class 'dict'>
>>> print(person1_information)
{'city': 'San Francisco', 'name': 'Sam', 'food': 'shrimps'}
```

Get the values in a Dictionary

To get the values of a dictionary from the keys, you can directly reference the keys. To do this, you enclose the key in brackets `[...]` after writing the variable name of the dictionary.

So, in the following example, a dictionary is initialized with keys “city”, “name,” and “food” and you can retrieve the value corresponding to the key “city.”

```
>>> create a dictionary person1_information
>>> person1_information = {'city': 'San Francisco', 'name': 'Sam', "food":
"shrimps"}
>>> print the dictionary
>>> print(person1_information["city"])
San Francisco
```

You can also use the get method to retrieve the values in a dict. The only difference is that in the get method, you can set a default value. In direct referencing, if the key is not present, the interpreter throws KeyError.

```
>>> # create a small dictionary
>>> alphabets = {1: ‘a’}
>>> # get the value with key 1
>>> print(alphabets.get(1))
'a'
>>> # get the value with key 2. Pass “default” as default. Since key 2 does
not exist, you get “default” as the return value.
>>> print(alphabets.get(2, "default"))
'default'
>>> # get the value with key 2 through direct referencing
>>> print(alphabets[2])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

```
KeyError: 2
```

Looping over dictionary

Say, you got a dictionary, and you want to print the keys and values in it. Note that the key-words `for` and `in` are used which are used when you try to loop over something. To learn more about looping please look into tutorial on looping.

```
>>> person1_information = {'city': 'San Francisco', 'name': 'Sam', "food":
"shrimps"}
>>> for k, v in person1_information.items():
...     print("key is: %s" % k)
...     print("value is: %s" % v)
...     print("###########################")
...
key is: food
value is: shrimps
###########################
key is: city
value is: San Francisco
##########################
key is: name
value is: Sam
###########################
```

Add elements to a dictionary

You can add elements by updating the dictionary with a new key and then assigning the value to a new key.

```
>>> # initialize an empty dictionary
>>> person1_information = {}

>>> # add the key, value information with key “city”
>>> person1_information["city"] = "San Francisco"
>>> # print the present person1_information
>>> print(person1_information)
{'city': 'San Francisco'}

>>> # add another key, value information with key “name”
>>> person1_information["name"] = "Sam"
>>> # print the present dictionary
>>> print(person1_information)
{'city': 'San Francisco', 'name': 'Sam'}

>>> # add another key, value information with key “food”
>>> person1_information["food"] = "shrimps"
>>> # print the present dictionary
>>> print(person1_information)
```

```
{'city': 'San Francisco', 'name': 'Sam', 'food': 'shrimps'}
```

Or you can combine two dictionaries to get a larger dictionary using the update method.

```
>>> # create a small dictionary
>>> person1_information = {'city': 'San Francisco'}

>>> # print it and check the present elements in the dictionary
>>> print(person1_information)
{'city': 'San Francisco'}

>>> # have a different dictionary
>>> remaining_information = {'name': 'Sam', "food": "shrimps"}

>>> # add the second dictionary remaining_information to
personal1_information using the update method
>>> person1_information.update(remaining_information)

>>> # print the current dictionary
>>> print(person1_information)
{'city': 'San Francisco', 'name': 'Sam', 'food': 'shrimps'}
```

Delete elements of a dictionary

To delete a key, value pair in a dictionary, you can use the `del` method.

```
>>> # initialise a dictionary with the keys “city”, “name”, “food”
>>> person1_information = {'city': 'San Francisco', 'name': 'Sam', "food":
"shrimps"}

>>> # delete the key, value pair with the key “food”
>>> del person1_information["food"]

>>> # print the present personal1_information. Note that the key, value pair
“food”: “shrimps” is not there anymore.
>>> print(person1_information)
{'city': 'San Francisco', 'name': 'Sam'}
```

A disadvantage is that it gives KeyError if you try to delete a nonexistent key.

```
>>> # initialise a dictionary with the keys “city”, “name”, “food”
>>> person1_information = {'city': 'San Francisco', 'name': 'Sam', "food":
"shrimps"}

>>> # deleting a non existent key gives key error.
>>> del person1_information["non_existent_key"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'non_existent_key'
```

So, instead of the `del` statement you can use the `pop` method. This method takes in the key as the parameter. As a second argument, you can pass the default value if the key is not present.

```
>>> # initialise a dictionary with key, value pairs
>>> person1_information = {'city': 'San Francisco', 'name': 'Sam', "food":
"shrimps"}

>>> # remove a key, value pair with key “food” and default value None
>>> print(person1_information.pop("food", None))
'Shrimps'

>>> # print the updated dictionary. Note that the key “food” is not present
anymore
>>> print(person1_information)
{'city': 'San Francisco', 'name': 'Sam'}

>>> # try to delete a nonexistent key. This will return None as None is given
as the default value.
>>> print(person1_information.pop("food", None))
None
```

More facts about the Python dictionary

You can test the presence of a key using the `has_key` method.

```
>>> alphabets = {1: ‘a’}
>>> alphabets.has_key(1)
True
>>> alphabets.has_key(2)
False
```

A dictionary in Python doesn't preserve the order. Hence, we get the following:

```
>>> call = {'sachin': 4098, 'guido': 4139}
>>> call["snape"] = 7663
>>> call
{'snape': 7663, 'sachin': 4098, 'guido': 4139}
```

Sets

A set is an unordered collection data type with no duplicate elements. Sets are iterable and mutable. The elements appear in an arbitrary order when sets are iterated.

Sets are commonly used for membership testing, removing duplicates entries, and also for operations such as intersection, union, and set difference.

In this tutorial you will learn how to create a set and and the common paradigms for a set in Python.

How to create Sets

Sets can be created by calling the built-in set() function with a sequence or another iterable object.

```
>>> #creating an empty set
>>> setA = set()
>>> print(setA)
set()

>>> # creating a set with a string.
>>> # since a string is an iterable, this will succeed.
>>> setA = set("HackerEarth")
>>> print(setA)
{'h', 'H', 't', 'k', 'e', 'c', 'E', 'a', 'r'}

>>> # creating a set with a list
>>> setA = set(["C", “C++”, “Python”])
>>> print(setA)
{'C', 'Python', 'C++'}

>>> # creating a set with a list of numbers
>>> # there are some duplicates in it.
>>> setA = set([1, 2, 3, 4, 5, 6, 7, 7, 7])
>>> print(setA)
{1, 2, 3, 4, 5, 6, 7}

>>> # creating a set with a string. The string has some repeated characters.
>>> myString = 'foobar'
>>> setA = set(myString)
>>> print(setA)
{'r', 'a', 'b', 'f', 'o'}
```

set(object) iterates over the elements present in object and adds all the unique elements to the set.

Next you will learn about different operations available for Python Sets.

For all set operations, the set created below which is a set of integers. There are some integers that are repeated here. :

```
>>> setA = set([1, 2, 3, 4, 5, 6, 7, 7, 7])
>>> print(setA)
{1, 2, 3, 4, 5, 6, 7}
```

Methods to change a set

How to add elements to a set

- Python set add(element)

This will add element to a set:

For example, you can add the element 8 to the set 8

```
>>> setA.add(8)
>>> print(setA)
{1, 2, 3, 4, 5, 6, 7, 8}
```

Or you can add a tuple (9, 10) to the setA and the new set will consist of the tuple as well.

```
>>> setA.add((9, 10))
>>> print(setA)
{1, 2, 3, 4, 5, 6, 7, 8, (9, 10)}
```

- Python set update(element)

Adds element to list; it is an in-place set union operation.

For example you can pass a list to the update method and this will update the setA with the elements.

```
>>> # pass a list with elements 11 and 12
>>> setA.update([11, 12])
>>> # check if setA is updated with the elements.
>>> print(setA)
{1, 2, 3, 4, 5, 6, 7, 8, 11, 12, (9, 10)}
```

Similarly you can update with a list and a new set as shown below

```
>>> setA.update([12, 14], {15, 16})
>>> print(setA)
{1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 14, 15, (9, 10), 16}
```

Using add, elements can be added but not another iterable like set, list, or tuple. Update can be used to add iterable or iterables of hashable elements.

Methods to remove elements from a set

Python set discard(element) and remove(element) Used to remove element from the set

```
>>> # removes element 7 from set
>>> setA.discard(7)
>>> print(setA)
```

```
{1, 2, 3, 4, 5, 6, 8, 11, 12, 14, 15, (9, 10), 16}

>>> # removes element 8 from set
>>> setA.remove(8)
>>> print(setA)
{1, 2, 3, 4, 5, 6, 11, 12, 14, 15, (9, 10), 16}
```

Both discard and remove take a single argument, the element to be deleted from the set. If the value is not present, discard() does not do anything. Whereas, remove will raise a KeyError exception.

```
>>> # discard doesn't do anything is value to be discarded is not present
>>> setA.discard(19)
>>> print(setA)
{1, 2, 3, 4, 5, 6, 11, 12, 14, 15, (9, 10), 16}

>>> # this operation fails with an exception being raised
>>> setA.remove(19)
Traceback (most recent call last):
   File "python", line 1, in <module>
KeyError: 19
```

Other useful set methods

- Python set copy() Creates a shallow copy of the set with which it is called

```
>>> shallow_copy_of_setA = setA.copy()
>>> print(shallow_copy_of_setA)
```

```
{1, 2, 3, 4, 5, 6, 11, 12, 14, 15, (9, 10), 16}
```

Using assignment here instead of copy() will create a pointer to the already existing set.

- Python set clear() Will remove all elements from set

```
>>> # clear the set shallow_copy_of_setA created before using copy()
operation
>>> shallow_copy_of_setA.clear()
>>> print(shallow_copy_of_setA)
```

```
set()
```

- Python set pop() Removes an arbitrary set element

```
>>> # popping an element from setA
>>> setA.pop()
1
>>> # pop raises a KeyError exception if the set is empty
>>> shallow_copy_of_setA.pop()
```

```
Traceback (most recent call last):
  File "python", line 1, in <module>
```

```
KeyError: 'pop from an empty set'
```

Set Operations

- Set Intersection using intersection(s) Returns element present in both sets; this can also be achieved using the ampersand operator (&).

```
>>> # create a new set setB
>>> setB= set()

>>> # update setB with values
>>> setB.update([1, 2, 3, 4, 5, 10, 15, 22])
>>> print(setB)
{1, 2, 3, 4, 5, 10, 15, 22}

>>> # print a new set with the values present in both setA and setB
>>> print(setA & setB)
{2, 3, 4, 5, 15}

>>> # above operation and using method name intersection shows same
results
>>> setA.intersection(setB)
```

```
{2, 3, 4, 5, 15}
```

- Set Difference using difference() Returns the difference of two sets; “-” operator can also be used to find the set difference.

```
>>> # print a new set with values present in setA but not in setB
>>> setA.difference(setB)
{6, 11, 12, 14, (9, 10), 16}

>>> # this returns empty set
>>> setB.difference(setA)
```

```
set()
```

setB is a proper subset of setA to setB - setA is empty set.

Other Set Operations

- Python set isdisjoint() Returns true if intersection of sets is empty otherwise false

```
>>> # returns false as both have common elements
>>> setA.isdisjoint(setB)
False

>>> # create a new empty set setC
>>> setC = set()
>>> # update setC with values
>>> setC.update([100, 99])

>>> # returns true as setA and setC has no elements in common
>>> setA.disjoint(setC)
```

```
True
```

- Python set difference_update() setA.difference_update(setB) removes all elements of y from setA; '-=' can be used in place of the difference_update method.

```
>>> # update setA by removing elements present in setB from setA
>>> setA.difference_update(setB)
>>> # check the result set
>>> print(setA)
```

```
{6, 11, 12, 14, (9, 10), 16}
```

Similarly, setA.intersection_update(setB) removes elements from setA which are not present in the intersection set of setA and setB. '&=' can be used in place of the intersection_update method.

- Python set issubset() and issuperset() setA.issubset(setB) returns True if setA is subset of setB, False if not. "<=" operator can be used to test for issubset. To check for proper subset "<" is used.

```
>>> # check if setA is a subset of setB
>>> setA.issubset(setB)
False
>>> # check if set B is a subset of setA
>>> setB.issubset(setA)
```

```
False
```

Let's make setB a subset of setA by removing values 1, 10, and 22.

```
>>> # remove few elements to make setB a subset of setA
>>> setB.remove(1)
>>> setB.remove(10)
>>> setB.remove(22)

>>> # check the values present in setB now
>>> print(setB)
{2, 3, 4, 5, 15}
```

```
>>> # issubset now returns true
>>> setB.issubset(setA)
True
>>> setB < setA
True

>>> #setA now becomes a superset of setB
>>> setA.issuperset(setB)
True
```

Python Expressions:

Expressions are representations of value. They are different from statement in the fact that statements do something while expressions are representation of value. For example any string is also an expressions since it represents the value of the string as well.

Python has some advanced constructs through which you can represent values and hence these constructs are also called expressions.

In this tutorial you will get to know about:

1. What are expressions in Python
2. How to construct expressions.

How to create an expressions

Python expressions only contain identifiers, literals, and operators. So, what are these?

Identifiers: Any name that is used to define a class, function, variable module, or object is an identifier. **Literals**: These are language-independent terms in Python and should exist independently in any programming language. In Python, there are the string literals, byte literals, integer literals, floating point literals, and imaginary literals.**Operators**: In Python you can implement the following operations using the corresponding tokens.

Operator	Token
add	+
subtract	-

Operator	Token
multiply	*
power	**
Integer Division	/
remainder	%
decorator	@
Binary left shift	<<
Binary right shift	>>
and	&
or	\
Binary Xor	^
Binary ones complement	~
Less than	<
Greater than	>

Operator	Token
Less than or equal to	<=
Greater than or equal to	>=
Check equality	==
Check not equal	!=

Following are a few types of python expressions:

List comprehension

The syntax for list comprehension is shown below:

```
[ compute(var) for var in iterable ]
```

For example, the following code will get all the number within 10 and put them in a list.

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

Dictionary comprehension

This is the same as list comprehension but will use curly braces:

```
{ k, v for k in iterable }
```

For example, the following code will get all the numbers within 5 as the keys and will keep the corresponding squares of those numbers as the values.

```
>>> {x:x**2 for x in range(5)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Generator expression

The syntax for generator expression is shown below:

```
( compute(var) for var in iterable )
```

For example, the following code will initialize a generator object that returns the values within 10 when the object is called.

```
>>> (x for x in range(10))
<generator object <genexpr> at 0x7fec47aee870>
>>> list(x for x in range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Conditional Expressions

You can use the following construct for one-liner conditions:

```
true_value if Condition else false_value
```

Example:

```
>>> x = "1" if True else "2"
>>> x
'1'
```