Python Tutorial

Adapted from:

http://www.programiz.com

EBook prepared by:

Sadaqur Rahman

Department of Biochemistry & Molecular Biology,

Shahjalal University of Science & Technology, Sylhet

Table of Contents

Python Programming Tutorial

Python is a very powerful high-level, object-oriented programming language. Guido van Rossum is the creator of Python with its first implementation in 1989. Python has a very easy-to-use and simple syntax, making it the perfect language for someone trying to learn computer programming for the first time. Python is an interpreted language. Interpreter is a program that converts the high-level program we write into low-level program that the computer understands. This tutorial is based on Python 3 and all the examples in this tutorial have been tested and verified in Python 3.3.2 running on Windows 7.

Python is a cross-platform programming language, meaning, it runs on multiple platforms like Windows, Mac OS X, Linux, Unix and has even been ported to the Java and .NET virtual machines. It is free and open source. Even though most of today's Linux and Mac have Python preinstalled in it, the version might be out-of-date. So, it is always a good idea to install the most current version.

Starting The Interpreter

After installation, the python interpreter lives in the installed directory. By default it is `/usr/local/bin/pythonX.X` in Linux/Unix and `C:\PythonXX` in Windows, where the `'X'` denotes the version number. To invoke it from the shell or the command prompt we need to add this location in the search path. Search path is a list of directories (locations) where the operating system searches for executables. For example, in Windows command prompt, we can type `set path=%path%;c:\python33` (python33 means version 3.3, it might be different in your case) to add the location to path for that particular session. In Mac OS we need not worry about this as the installer takes care about the search path.

Now there are various ways to start Python.

1. Immediate mode:

Typing `python` in the command line will invoke the interpreter in immediate mode. We can directly type in Python expressions and press enter to get the output.

```
>>>
```

is the Python prompt. It tells us that the interpreter is ready for our input. Try typing in `1 + 1` and press enter. We get `2` as the output. This prompt can be used as a calculator. To exit this mode type `exit()` or `quit()` and press enter.

2. Script mode:

This mode is used to execute Python program written in a file. Such a file is called a **script**. Scripts can be saved to disk for future use. Python scripts have the extension `.py`, meaning that the filename ends with `.py`.

For example: `helloWorld.py`

To execute this file in script mode we simply write `python helloWorld.py` at the command prompt.

Integrated Development Environment (IDE)

We can use any text editing software to write a Python script file. We just need to save it with the `.py` extension. But using an IDE can make our life a lot easier. IDE is a piece of software that provides useful features like code hinting, syntax highlighting and checking, file explorers etc. to the programmer for application development. Using an IDE can get rid of redundant tasks and significantly decrease the time required for application development.

IDEL is a graphical user interface (GUI) that can be installed along with the Python programming language and is available from the official website. We can also use other commercial or free IDE according to our preference. We have used the PyScripter IDE for our testing and we recommend the same. It is free and open source.

Hello World Example

Now that we have Python up and running, we can continue on to write our first Python program. Type the following code in any text editor or an IDE and save it as `helloWorld.py`

```
print("Hello world!")
```

Now at the command window, go to the loaction of this file. You can use the `cd` command to change directory. To run the script, type `python helloWorld.py` in the command window. We should be able to see the output as follows

Output:

```
Hello world!
```

If you are using PyScripter, there is a green arrow button on top. Press that button or press `Ctrl+F9` on your keyboard to run the program.

Explanation:

In this program we have used the built-in function `print()`, to print out a string to the screen. String is the value inside the quotation marks, i.e. `Hello world!` . Now try printing out your name by modifying this program.

Congratulations! You just wrote your first program in Python. As we can see, it was pretty easy. This is the beauty of Python programming language.

Python Introduction

- Keywords and Identifier
- Statements & Comments
- Python Datatypes
- Python I/O and Import
- Python Operators
- Precedence & Associativity

Python Keywords and Identifier

Keywords

Keywords are the reserved words in Python. We cannot use a keyword as variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language. In Python, keywords are case sensitive.

There are 33 keywords in Python 3.3. This number can vary slightly in course of time. All the keywords except `True`, `False` and `None` are in lowercase and they must be written as it is. The list of all the keywords are given below.

Keywords in Python programming language

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

Identifiers

Identifier is the name given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another.

Rules for writing identifiers in Python

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like `myClass`, `var_1` and `print_this_to_screen`, all are valid example.
2. An identifier cannot start with a digit. `1variable` is invalid, but `variable1` is perfectly fine.
3. Keywords cannot be used as identifiers.

```
>>> global = 1
  File "<interactive input>", line 1
    global = 1
           ^
      SyntaxError: invalid syntax
```

4. We cannot use special symbols like !, @, #, $, % etc. in our identifier.

```
>>> a@ = 0
  File "<interactive input>", line 1
    a@ = 0
     ^
      SyntaxError: invalid syntax
```

5. Identifier can be of any length.

Things to care about

Python is a case-sensitive language. This means, `Variable` and `variable` are not the same. Always name identifiers that make sense. While, `c = 10` is valid. Writing `count = 10` would make more sense and it would be easier to figure out what it does even when you look at your code after a long gap. Multiple words can be separated using an underscore, `this_is_a_long_variable`. We can also use camel-case style of writing, i.e., capitalize every first letter of the word except the initial word without any spaces. For example: `camelCaseExample`

Python Statement, Indentation and Comments

Python Statement

Instructions that a Python interpreter can execute are called statements. For example, `a = 1` is an assignment statement. `if` statement, `for` statement, `while` statement etc. are other kinds of statements which will be discussed later.

Multi-line statement

In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\). For example:

```
a = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8 + 9
```

This is explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ] and braces { }. For instance, we can implement the above multi-line statement as

```
a = (1 + 2 + 3 +
    4 + 5 + 6 +
    7 + 8 + 9)
```

Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with [ ] and { }. For example:

```
colors = ['red',
          'blue',
          'green']
```

We could also put multiple statements in a single line using semicolons, as follows

```
a = 1; b = 2; c = 3
```

Python Indentation

Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation. A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block. Generally four whitespaces are used for indentation and is preferred over tabs. Here is an example.

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

The enforcement of indentation in Python makes the code look neat and clean. This results into Python programs that look similar and consistent. Indentation can be ignored in line continuation. But it's a good idea to always indent. It makes the code more readable. For example:

```
if True:
    print('Hello')
    a = 5
```

and

```
if True: print('Hello'); a = 5
```

both are valid and do the same thing. But the former style is clearer.

Incorrect indentation will result into `IndentationError`.

Python Comments

Comments are very important while writing a program. It describes what's going on inside a program so that a person looking at the source code does not have a hard time figuring it out. You might forget the key details of the program you just wrote in a month's time. So taking time to explain these concepts in form of comments is always fruitful.

In Python, we use the hash (#) symbol to start writing a comment. It extends up to the newline character. Comments are for programmers for better understanding of a program. Python Interpreter ignores comment.

```
#This is a comment
#print out Hello
print('Hello')
```

Multi-line comments

If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line. For example:

```
#This is a long comment
#and it extends
#to multiple lines
```

Another way of doing this is to use triple quotes, either `'''` or `"""`. These triple quotes are generally used for multi-line strings. But they can be used as multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

```
"""This is also a
perfect example of
multi-line comments"""
```

Docstring

Docstring is short for documentation string. It is a string that occurs as the first statement in a module, function, class, or method definition. We must write what a function/class does in the docstring. Triple quotes are used while writing docstrings. For example:

```
def double(num):
    """Function to double the value"""
    return 2*num
```

Docstring is available to us as the attribute `__doc__` of the function.

```
>>> print(double.__doc__)
Function to double the value
```

Python Variables and Datatypes

Python Variables

A variable is a location in memory used to store some data (value). They are given unique names to differentiate between different memory locations. The rules for writing a variable name is same as the rules for writing identifiers in Python.

We don't need to declare a variable before using it. In Python, we simply assign a value to a variable and it will exist. We don't even have to declare the type of the variable. This is handled internally according to the type of value we assign to the variable.

Variable assignment

We use the assignment operator (=) to assign values to a variable. Any type of value can be assigned to any valid variable.

```
a = 5
b = 3.2
c = "Hello"
```

Here, we have three assignment statements. `5` is an integer assigned to the variable *a*. Similarly, `3.2` is a floating point number and `"Hello"` is a string (sequence of characters) assigned to the variables *b* and *c* respectively.

Multiple assignments

In Python, multiple assignments can be made in a single statement as follows:

```
a, b, c = 5, 3.2, "Hello"
```

If we want to assign the same value to multiple variables at once, we can do this as

```
x = y = z = "same"
```

This assigns the "same" string to all the three variables.

Datatypes in Python

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

Python Numbers

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

```
>>> a = 5
>>> type(a)
<class 'int'>
>>> type(2.0)
<class 'float'>
>>> isinstance(1+2j,complex)
True
```

Integers can be of any length, it is only limited by the memory available. A floating point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. `1` is integer, `1.0` is floating point number. Complex numbers are written in the form, `x + yj`, where *x* is the real part and *y* is the imaginary part. Here are some examples.

```
>>> a = 1234567890123456789
>>> a
1234567890123456789
>>> b = 0.1234567890123456789
>>> b
0.12345678901234568
>>> c = 1+2j
>>> c
(1+2j)
```

Notice that the `float` variable *b* got truncated.

Python List

List is an ordered sequence of items. It is one of the most used datatype in Python and is very flexible. All the items in a list do not need to be of the same type. Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets [ ].

```
>>> a = [1, 2.2, 'python']
>>> type(a)
<class 'list'>
```

We can use the slicing operator [ ] to extract an item or a range of items from a list. Index starts form 0 in Python.

```
>>> a = [5,10,15,20,25,30,35,40]
>>> a[2]
15
>>> a[0:3]
[5, 10, 15]
```

```
>>> a[5:]
[30, 35, 40]
```

Lists are mutable, meaning, value of elements of a list can be altered.

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

Python Tuple

Tuple is an ordered sequence of items same as list. The only difference is that tuples are immutable. Tuples once created cannot be modified. They are used to write-protect data and are usually faster than list as it cannot change dynamically. Tuple is defined within parentheses () where items are separated by commas.

```
>>> t = (5,'program', 1+3j)
>>> type(t)
<class 'tuple'>
```

We can use the slicing operator [] to extract items but we cannot change its value.

```
>>> t[1]
'program'
>>> t[0:3]
(5, 'program', (1+3j))
>>> t[0] = 10
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Python Strings

String is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, `'''` or `"""`.

```
>>> s = "This is a string"
>>> type(s)
<class 'str'>
>>> s = '''a multiline
... string'''
```

Like list and tuple, slicing operator [ ] can be used with string. Strings are immutable.

```
>>> s = 'Hello world!'
>>> s[4]
'o'
>>> s[6:11]
'world'
```

```
>>> s[5] ='d'
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Python Set

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

```
>>> a = {5,2,3,1,4}
>>> a
{1, 2, 3, 4, 5}
>>> type(a)
<class 'set'>
```

We can perform set operations like union, intersection on two sets. Set have unique values. They eliminate duplicates.

```
>>> a = {1,2,2,3,3,3}
>>> a
{1, 2, 3}
```

Since, set are unordered collection, indexing has no meaning. Hence the slicing operator [] does not work.

```
>>> a = {1,2,3}
>>> a[1]
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

Python Dictionary

Dictionary is an unordered collection of key-value pairs. It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value. In Python, dictionaries are defined within braces {} with each item being a pair in the form `key:value`. Key and value can be of any type.

```
>>> d = {1:'value','key':2}
>>> type(d)
<class 'dict'>
```

We use key to retrieve the respective value. But not the other way around.

```
>>> d[1]
'value'
>>> d['key']
2
```

```
>>> d[2]
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
KeyError: 2
```

Conversion between datatypes

We can convert between different datatypes by using different type conversion functions like int(), float(), str() etc.

```
>>> float(5)
5.0
```

Conversion from float to int will truncate the value (make it closer to zero).

```
>>> int(10.6)
10
>>> int(-10.6)
-10
```

Conversion to and from string must contain compatible values.

```
>>> float('2.5')
2.5
>>> str(25)
'25'
>>> int('1p')
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1p'
```

We can even convert one sequence to another.

```
>>> set([1,2,3])
{1, 2, 3}
>>> tuple({5,6,7})
(5, 6, 7)
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

To convert to dictionary, each element must be a pair

```
>>> dict([[1,2],[3,4]])
{1: 2, 3: 4}
>>> dict([(3,26),(4,44)])
{3: 26, 4: 44}
```

Python Input, Output and Import

Python language provides numerous built-in functions that are readily available to us at the Python prompt. Some of the functions like `input()` and `print()` are widely used for standard input and output operations respectively. Let us see the output section first.

Output

We use the `print()` function to output data to the standard output device (screen). We can also output data to a file, but this will be discussed later. An example use is given below.

```
>>> print('This sentence is output to the screen')
This sentence is output to the screen
>>> a = 5
>>> print('The value of a is',a)
The value of a is 5
```

In the second example, we can notice that a space was added between the string and the value of variable *a*. This is by default, but we can change it. The actual syntax of the `print()` function is

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Here, `objects` is the value(s) to be printed. `sep` is the separator used between the values. It defaults into a space character. `end` is printed after printing all the values. It defaults into a new line. `file` is the object where the values are printed and its default value is `sys.stdout` (screen). Here are an example to illustrate this.

```
print(1,2,3,4)
print(1,2,3,4,sep='*')
print(1,2,3,4,sep='#',end='&')
```

Output

```
1 2 3 4
1*2*3*4
1#2#3#4&
```

Output formatting

Sometimes we would like to format our output to make it look attractive. This can be done by using the `str.format()` method. This method is visible to any string object.

```
>>> x = 5; y = 10
>>> print('The value of x is {} and y is {}'.format(x,y))
The value of x is 5 and y is 10
```

Here the curly braces {} are used as placeholders. We can specify the order in which it is printed by using numbers (tuple index).

```
>>> print('I love {0} and {1}'.format('bread','butter'))
I love bread and butter
>>> print('I love {1} and {0}'.format('bread','butter'))
I love butter and bread
```

We can even use keyword arguments to format the string.

```
>>> print('Hello {name},
{greeting}'.format(greeting='Goodmorning',name='John'))
Hello John, Goodmorning
```

We can even format strings like the old `sprintf()` style used in C programming language. We use the `%` operator to accomplish this.

```
>>> x = 12.3456789
>>> print('The value of x is %3.2f' %x)
The value of x is 12.35
>>> print('The value of x is %3.4f' %x)
The value of x is 12.3457
```

Input

Up till now, our programs were static. The value of variables were defined or hard coded into the source code. To allow flexibility we might want to take the input from the user. In Python, we have the `input()` function to allow this. The syntax for `input()` is

```
input([prompt])
```

where `prompt` is the string we wish to display on the screen. It is optional.

```
>>> num = input('Enter a number: ')
Enter a number: 10
>>> num
'10'
```

Here, we can see that the entered value `10` is a string, not a number. To convert this into a number we can use `int()` or `float()` functions.

```
>>> int('10')
10
>>> float('10')
10.0
```

This same operation can be performed using the `eval()` function. But it takes it further. It can evaluate even expressions, provided the input is a string

```
>>> int('2+3')
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '2+3'
```

```
>>> eval('2+3')
5
```

Import

When our program grows bigger, it is a good idea to break it into different modules. A module is a file containing Python definitions and statements. Python modules have a filename and end with the extension `.py`.

Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the `import` keyword to do this. For example, we can import the `math` module by typing in `import math`.

```
>>> import math
>>> math.pi
3.141592653589793
```

Now all the definitions inside `math` module are available in our scope. We can also import some specific attributes and functions only, using the `from` keyword. For example:

```
>>> from math import pi
>>> pi
3.141592653589793
```

While importing a module, Python looks at several places defined in `sys.path`. It is a list of directory locations.

```
>>> import sys
>>> sys.path
['',
 'C:\\Python33\\Lib\\idlelib',
 'C:\\Windows\\system32\\python33.zip',
 'C:\\Python33\\DLLs',
 'C:\\Python33\\lib',
 'C:\\Python33',
 'C:\\Python33\\lib\\site-packages']
```

We can add our own location to this list as well.

Python Operators

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand. For example:

```
>>> 2+3
5
```

Here, `+` is the operator that performs addition. `2` and `3` are the operands and `5` is the output of the operation. Python has a number of operators which are classified below.

Type of operators in Python

Arithmetic operators
Comparison (Relational) operators
Logical (Boolean) operators
Bitwise operators
Assignment operators
Special operators

Arithmetic operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

Arithmetic operators in Python

Operator	Meaning	Example
+	Add two operands or unary plus	x + y +2
-	Subtract right operand from the left or unary minus	x - y -2
*	Multiply two operands	x * y

/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

Here is an example.

```
x = 15
y = 4
print('x + y = ',x+y)
print('x - y = ',x-y)
print('x * y = ',x*y)
print('x / y = ',x/y)
print('x // y = ',x//y)
print('x ** y = ',x**y)
```

Output

```
x + y =  19
x - y =  11
x * y =  60
x / y =  3.75
x // y =  3
x ** y =  50625
```

Comparison operators

Comparison operators are used to compare values. It either returns `True` or `False` according to the condition.

Comparision operators in Python

Operator	**Meaning**	**Example**
>	Greater that - True if left operand is greater than the right	x > y
<	Less that - True if left operand is less than the right	x < y
==	Equal to - True if both operands are equal	x == y

!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to - True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y

Here is an example.

```
x = 10
y = 12
print('x > y  is',x>y)
print('x < y  is',x<y)
print('x == y is',x==y)
print('x != y is',x!=y)
print('x >= y is',x>=y)
print('x <= y is',x<=y)
```

Output

```
x > y  is False
x < y  is True
x == y is False
x != y is True
x >= y is False
x <= y is True
```

Logical operators

Logical operators are the `and`, `or`, `not` operators.

Logical operators in Python

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Here is an example.

```
x = True
y = False
print('x and y is',x and y)
print('x or y is',x or y)
print('not x is',not x)
```

Output

```
x and y is False
x or y is True
not x is False
```

Bitwise operators

Bitwise operators act on operands as if they were string of binary digits. It operates bit by bit, hence the name. For example, 2 is `10` in binary and 7 is `111`.

Let *x* = 10 (`0000 1010` in binary) and *y* = 4 (`0000 0100` in binary)

Bitwise operators in Python

Operator	Meaning	Example
&	Bitwise AND	x& y = 0 (`0000 0000`)
|	Bitwise OR	x | y = 14 (`0000 1110`)
~	Bitwise NOT	~x = -11 (`1111 0101`)
^	Bitwise XOR	x ^ y = 14 (`0000 1110`)
>>	Bitwise right shift	x>> 2 = 2 (`0000 0010`)
<<	Bitwise left shift	x<< 2 = 42 (`0010 1000`)

Assignment operators

Assignment operators are used in Python to assign values to variables. `a = 5` is a simple assignment operator that assigns the value 5 on the right to the variable *a* on the left. There are various compound operators in Python like `a += 5` that adds to the variable and later assigns the same. It is equivalent to `a = a + 5`.

Assignment operators in Python

Operator	Example	Equivatent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Special operators

Python language offers some special type of operators like the identity operator or the membership operator. They are described below with examples.

Identity operators

`is` and `is not` are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

Identity operators in Python

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Here is an example.

```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]
print(x1 is not y1)
print(x2 is y2)
print(x3 is y3)
```

Output

```
False
True
False
```

Here, we see that *x1* and *y1* are integers of same values, so they are equal as well as identical. Same is the case with *x2* and *y2* (strings). But *x3* and *y3* are list. They are equal but not identical. Since list are mutable (can be changed), interpreter locates them separately in memory although they are equal.

Membership operators

`in` and `not in` are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary). In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Here is an example.

```
x = 'Hello world'
y = {1:'a',2:'b'}
print('H' in x)
print('hello' not in x)
print(1 in y)
print('a' in y)
```

Output

```
True
True
True
False
```

Here, `'H'` is in *x* but `'hello'` is not present in *x* (remember, Python is case sensitive). Similary, `1` is key and `'a'` is the value in dictionary *y*. Hence, `'a' in y` returns `False`.

Python Flow Control

- Python if...else
- Python for Loop
- Python while Loop
- Python break and continue
- Python Pass
- Looping Technique

Python if...elif...else and Nested if

Decision making is required when we want to execute a code only if a certain condition is satisfied. The `if…elif…else` statement is used in Python for decision making.

Python if Statement Syntax

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

Here, the program evaluates the `test expression` and will execute statement(s) only if the text expression is `True`. If the text expression is `False`, the statement(s) is not executed. In Python, the body of the `if` statement is indicated by the indentation. Body starts with an indentation and the first unindented line marks the end. Python interprets non-zero values as `True`. `None` and `0` are interpreted as `False`.

Python if Statement Flowchart

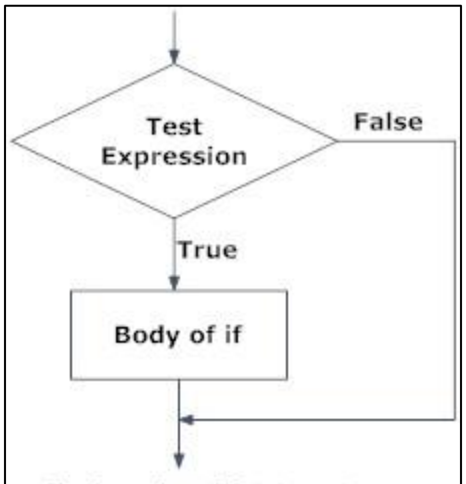


Fig: Operation of if statement

Example: Python if Statement

```
# In this program, user inputs a number.
# If the number is positive, we print an appropriate message

num = float(input("Enter a number: "))
if num > 0:
    print("Positive number")
print("This is always printed")
```

Output 1

```
Enter a number: 3
Positive number
This is always printed
```

Output 2

```
Enter a number: -1
This is always printed
```

In the above example, `num > 0` is the test expression. The body of `if` is executed only if this evaluates to `True`. When user enters 3, test expression is true and body inside body of `if` is executed. When user enters -1, test expression is false and body inside body of `if` is skipped. The `print()` statement falls outside of the `if` block (unindented). Hence, it is executed regardless of the test expression. We can see this in our two outputs above.

Python if...else

Syntax of if...else

```
if test expression:
    Body of if
else:
    Body of else
```

The `if..else` statement evaluates `test expression` and will execute body of `if` only when test condition is `True`. If the condition is `False`, body of `else` is executed. Indentation is used to separate the blocks.

Python if..else Flowchart

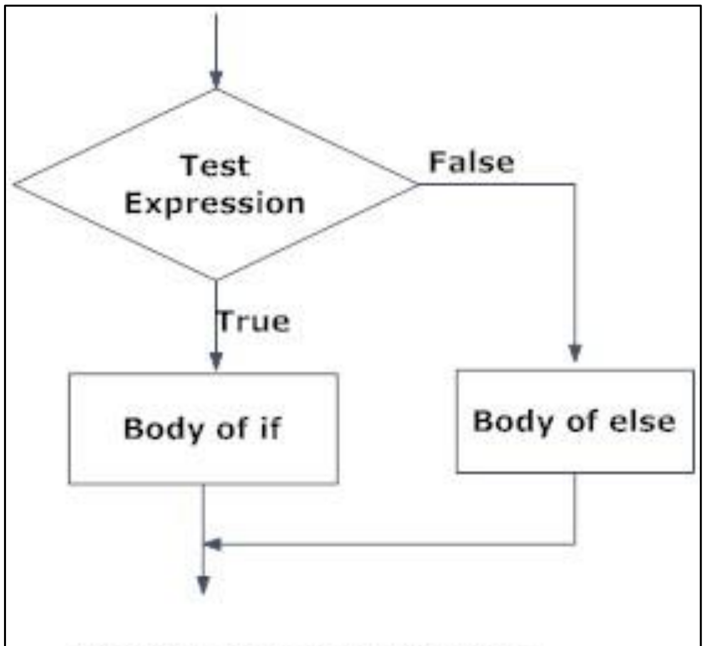


Fig: Operation of if...else statement

Example of if...else

```
# In this program, user input a number
# Program check if the number is positive or negative and display an
appropriate message

num = float(input("Enter a number: "))
if num >= 0:
    print("Positive or Zero")
else:
    print("Negative number")
```

Output 1

```
Enter a number: 2
Positive or Zero
```

Output 2

```
 Enter a number: -3
Negative number
```

In the above example, when user enters 2, the test epression is true and body of `if` is executed and `body` of else is skipped. When user enters -3, the test expression is false and body of `else` is executed and body of `if` is skipped.

Python if...elif...else

Syntax of if...elif...else

```
if test expression:
    Body of if
elif test expression:
    Body of elif
else:
    Body of else
```

The `elif` is short for else if. It allows us to check for multiple expressions. If the condition for `if` is `False`, it checks the condition of the next `elif` block and so on. If all the conditions are `False`, body of else is executed. Only one block among the several `if...elif...else` blocks is executed according to the condition. A `if` block can have only one `else` block. But it can have multiple `elif` blocks.

Flowchart of if...elif...else

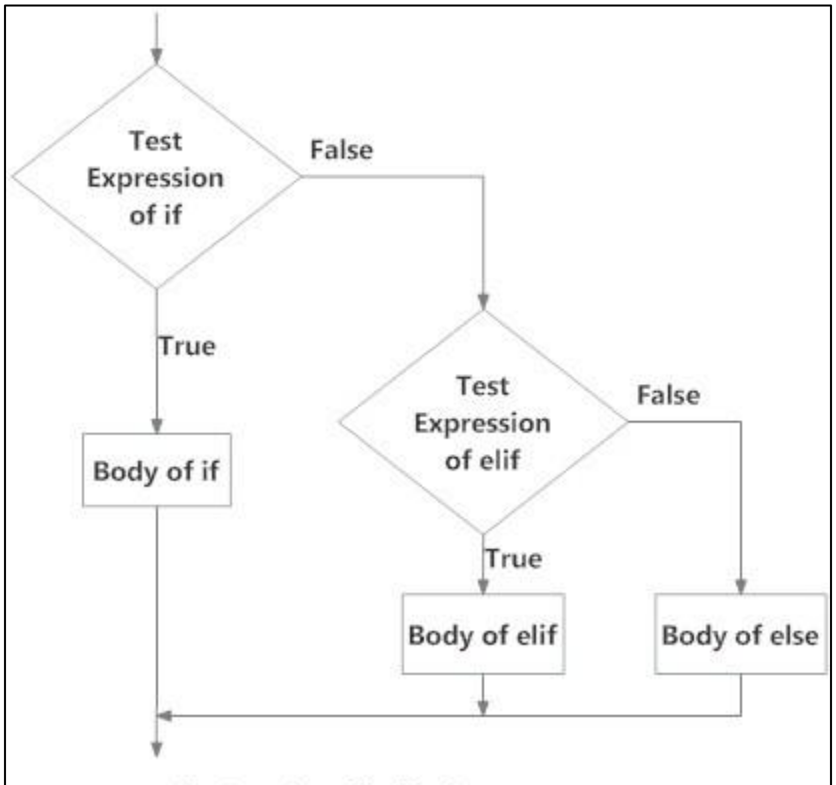


Fig: Operation of if...elif...else statement

Example of if...elif...else

```
# In this program, we input a number
# check if the number is positive or
# negative or zero and display
# an appropriate message

num = float(input("Enter a number: "))
if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

Output 1

```
Enter a number: 2
Positive number
```

Output 2

```
Enter a number: 0
Zero
```

Output 3

```
Enter a number: -2
Negative number
```

Python Nested if statements

We can have a `if...elif...else` statement inside another `if...elif...else` statement. This is called nesting in computer programming. In fact, any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. This can get confusing, so must be avoided if we can.

Python Nested if Example

```
# In this program, we input a number
# check if the number is positive or
# negative or zero and display
# an appropriate message
# This time we use nested if

num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

Output 1

```
Enter a number: 5
Positive number
```

Output 2

```
Enter a number: -1
Negative number
```

Output 3

```
Enter a number: 0
Zero
```

Python for Loop

The `for` loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

Syntax of for Loop

```
for val in sequence:
    Body of for
```

Here, `val` is the variable that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence. The body of `for` loop is separated from the rest of the code using indentation.

Flowchart of for Loop

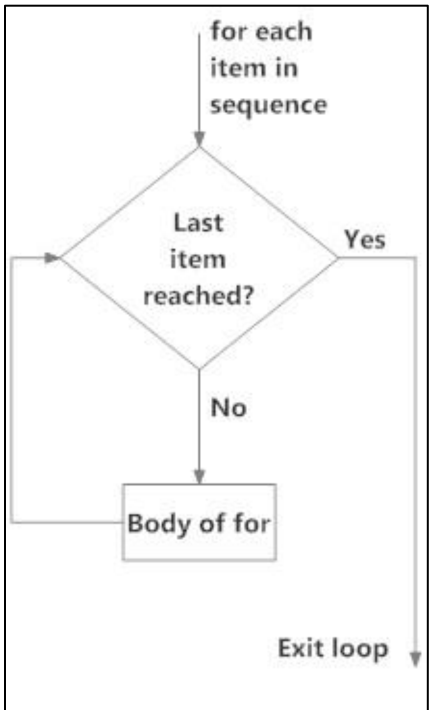


Fig: operation of for loop

Example: Python for Loop

```
# Program to find
# the sum of all numbers
# stored in a list

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

# variable to store the sum
sum = 0
```

```
# iterate over the list
for val in numbers:
    sum = sum+val

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

Output

```
The sum is 48
```

The range() function

We can generate a sequence of numbers using `range()` function. `range(10)` will generate numbers from 0 to 9 (10 numbers). We can also define the `start`, `stop` and `step size` as `range(start,stop,step size)`. `step size` defaults to 1 if not provided. This function does not store all the values in memory, it would be inefficient. So it remembers the `start`, `stop`, `step size` and generates the next number on the go. To force this function to output all the items, we can use the function `list()`.

The following example will clarify this.

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2,8))
[2, 3, 4, 5, 6, 7]
>>> list(range(2,20,3))
[2, 5, 8, 11, 14, 17]
```

We can use the `range()` function in `for` loops to iterate through a sequence of numbers. It can be combined with the `len()` function to iterate though a sequence using indexing. Here is an example.

```
# Program to iterate
# through a list
# using indexing

# List of genre
genre = ['pop','rock','jazz']

# iterate over the list using index
for i in range(len(genre)):
    print("I like",genre[i])
```

Output

```
I like pop
I like rock
I like jazz
```

for loop with else

A `for` loop can have an optional `else` block as well. The `else` part is executed if the items in the sequence used in `for` loop exhausts. `break` statement can be used to stop a `for` loop. In such case, the `else` part is ignored. Hence, a `for` loop's `else` part runs if no break occurs.

Here is an example to illustrate this.

```
# Program to show
# the control flow
# when using else block
# in a for loop

# a list of digit
list_of_digits = [0,1,2,3,4,5,6]

# take input from user
input_digit = int(input("Enter a digit: "))

# search the input digit in our list
for i in list_of_digits:
    if input_digit == i:
        print("Digit is in the list")
        break
else:
    print("Digit not found in list")
```

Output 1

```
Enter a digit: 3
Digit is in the list
```

Output 2

```
Enter a digit: 9
Digit not found in list
```

Here, we have a list of digits from 0 to 6. We ask the user to enter a digit and check if the digit is in our list or not. If the digit is present, `for` loop breaks prematurely. So, the `else` part does not run. But if the items in our list exhausts (digit not found in our list), the program enters the `else` part.

Python while Loop

The `while` loop in Python is used to iterate over a block of code as long as the test expression (condition) is true. We generally use this loop when we don't know beforehand, the number of times to iterate.

Syntax of while Loop

```
while test_expression:
    Body of while
```

In `while` loop, test expression is checked first. The body of the loop is entered only if the `test_expression` evaluates to `True`. After one iteration, the test expression is checked again. This process continues untill the `test_expression` evaluates to `False`.

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

Flowchart of while Loop

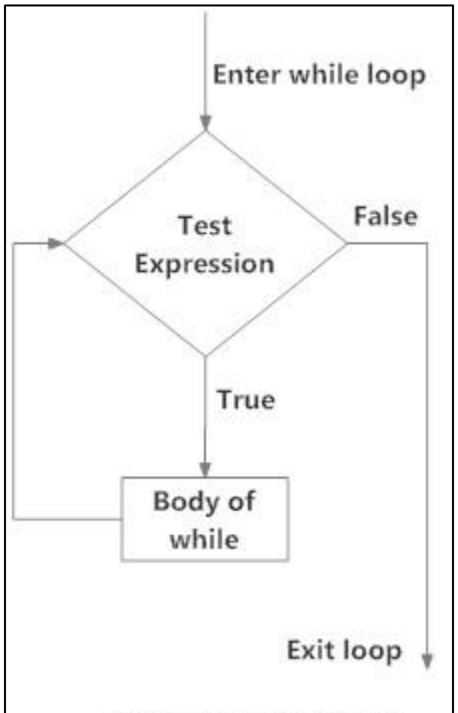


Fig: operation of while loop

Example: Python while Loop

```
# Program to add natural
# numbers upto n where
# n is provided by the user
# sum = 1+2+3+...+n

# take input from the user
n = int(input("Enter n: "))

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1    # update counter

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

Output

```
Enter n: 10
The sum is 55
```

In the above program, we asked the user to enter a number, *n*. `while` loop is used to sum from 1 to that number. The condition will be `True` as long as our counter variable *i* is less than or equal to *n*. We need to increase the value of counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never ending loop). Finally the result is displayed.

while loop with else

Same as that of `for` loop, we can have an optional `else` block with `while` loop as well. The `else` part is executed if the condition in the `while` loop evaluates to `False`. `while` loop can be terminated with a `break` statement. In such case, the `else` part is ignored. Hence, a `while` loop's `else` part runs if no break occurs and the condition is false.

Here is an example to illustrate this.

```
# Example to illustrate
# the use of else statement
# with the while loop

counter = 0

while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

Output

```
Inside loop
Inside loop
Inside loop
Inside else
```

Here, we use a counter variable to print the string `Inside loop` three times. On the forth iteration, the condition in `while` becomes `False`. Hence, the `else` part is executed.

Python break and continue Statement

In Python, `break` and `continue` statements can alter the flow of a normal loop. Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without cheking test expression. The `break` and `continue` statements are used in these cases.

break statement

The `break` statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop. If it is inside a nested loop (loop inside another loop), `break` will terminate the innermost loop.

Syntax of break

```
break
```

Flowchart of break

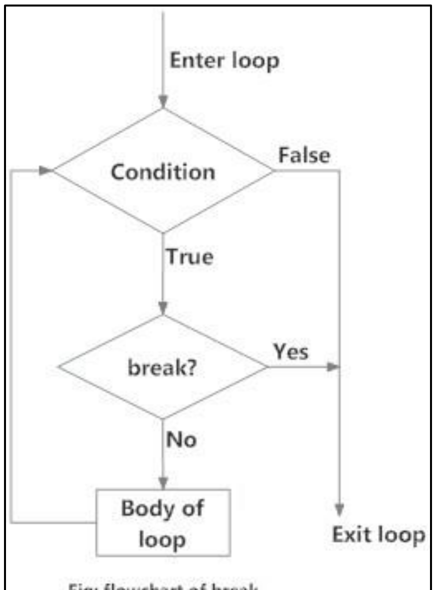
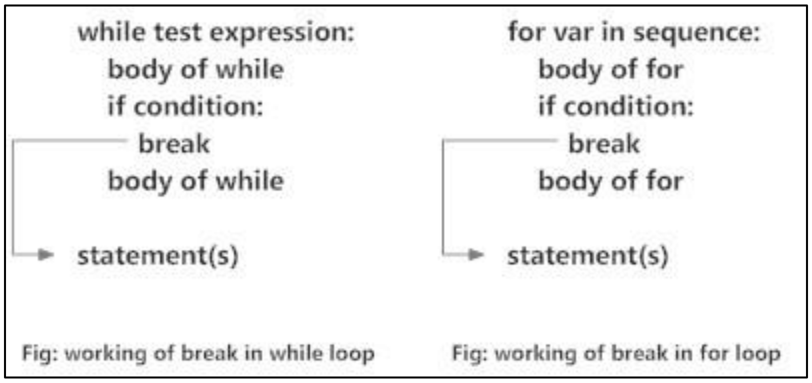


Fig: flowchart of break

The working of `break` statement in for loop and while loop is shown below.



Example: Python break

```
# Program to show the use of break statement inside loop

for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

Output

```
s
t
r
The end
```

In this program, we iterate through the `"string"` sequence. We check if the letter is `"i"`, upon which we break from the loop. Hence, we see in our output that all the letters up till `"i"` gets printed. After that, the loop terminates.

continue statement

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

Syntax of Continue

```
continue
```

Flowchart of continue

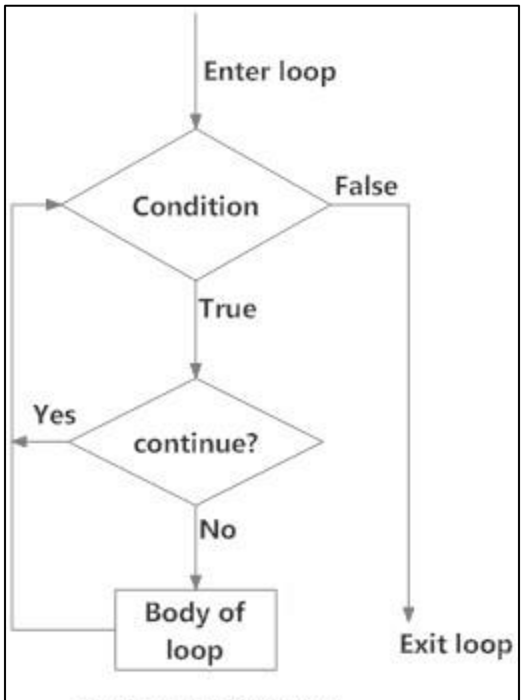
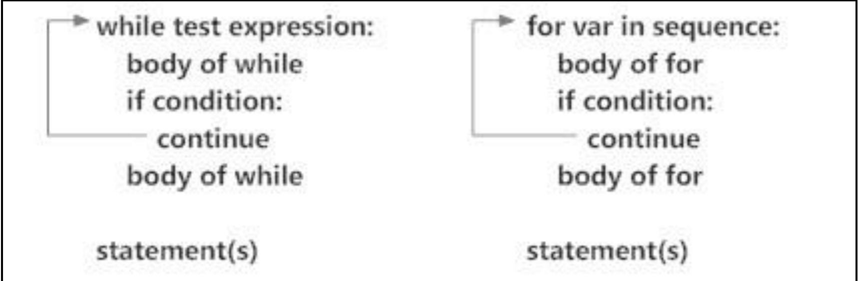


Fig: flowchart of continue

The working of `continue` statement in for and while loop is shown below.



```
while test expression:
    body of while
    if condition:
        continue
    body of while

statement(s)
```

Fig: working of continue in while loop

```
for var in sequence:
    body of for
    if condition:
        continue
    body of for

statement(s)
```

Fig: working of continue in for loop

Example: Python continue

```
# Program to show
# the use of continue
# statement inside loops

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

print("The end")
```

Output

```
s
t
r
n
g
The end
```

This program is same as the above example except the `break` statement has been replaced with `continue`. We continue with the loop, if the string is `"i"`, not executing the rest of the block. Hence, we see in our output that all the letters except `"i"` gets printed.

Python pass Statement

In Python programming, `pass` is a null statement. The difference between a comment and `pass` statement in Python is that, while the interpreter ignores a comment entirely, `pass` is not ignored. But nothing happens when it is executed. It results into no operation (NOP).

Syntax of pass

```
pass
```

We generally use it as a placeholder. Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would complain. So, we use the `pass` statement to construct a body that does nothing.

Example: pass Statement

```
for val in sequence:
    pass
```

We can do the same thing in an empty function or class as well.

```
def function(args):
    pass

class example:
    pass
```

Python Looping Techniques

Python programming offers two kinds of loop, the for loop and the while loop. Using these loops along with loop control statements like `break` and `continue`, we can create various forms of loop.

The infinite loop

We can create an infinite loop using `while` statement. If the condition of `while` loop is always `True`, we get an infinite loop.

Example of infinite loop

```
# An example of infinite loop
# press Ctrl + c to exit from the loop

while True:
   num = int(input("Enter an integer: "))
   print("The double of",num,"is",2 * num)
```

Output

```
Enter an integer: 3
The double of 3 is 6
Enter an integer: 5
The double of 5 is 10
Enter an integer: 6
The double of 6 is 12
Enter an integer:
Traceback (most recent call last):
```

Loop with condition at the top

This is a normal `while` loop without `break` statements. The condition of the `while` loop is at the top and the loop terminates when this condition is `False`.

Flowchart of Loop With Condition at Top

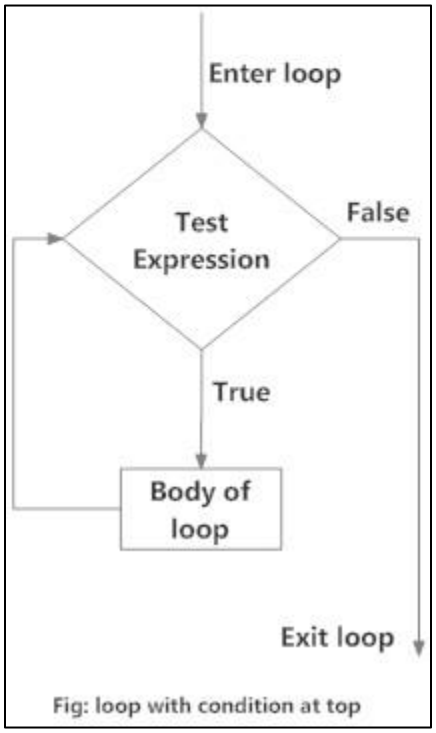


Fig: loop with condition at top

Example

```
# Program to illustrate a loop with condition at the top

n = int(input("Enter n: "))

# initialize sum and counter
sum = 0
i = 1

while i <= n:
   sum = sum + i
   i = i+1    # update counter

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

Output

```
Enter n: 10
The sum is 55
```

Loop with condition in the middle

This kind of loop can be implemented using an infinite loop along with a conditional break in between the body of the loop.

Flowchart of Loop with Condition in Middle

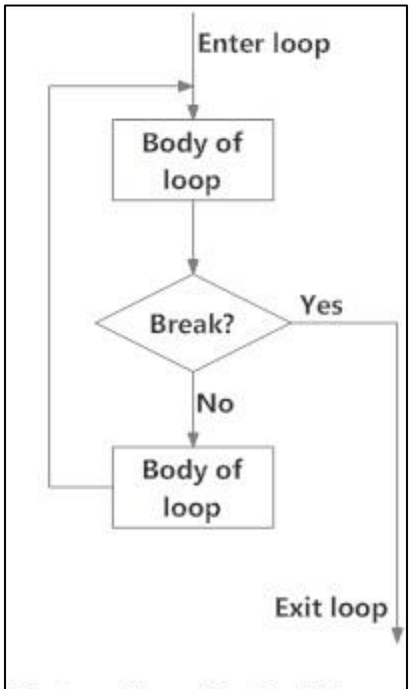


Fig: loop with condition in middle

Example

```
# Program to illustrate a loop with condition in the middle.
# Take input from the user untill a vowel is entered

vowels = "aeiouAEIOU"

# infinite loop
while True:
   v = input("Enter a vowel: ")
   # condition in the middle
   if v in vowels:
       break
   print("That is not a vowel. Try again!")

print("Thank you!")
```

Output

```
Enter a vowel: r
That is not a vowel. Try again!
Enter a vowel: 6
That is not a vowel. Try again!
Enter a vowel: ,
That is not a vowel. Try again!
Enter a vowel: u
Thank you!
```

Loop with condition at the bottom

This kind of loop ensures that the body of the loop is executed at least once. It can be implemented using an infinite loop along with a conditional break at the end. This is similar to the `do...while` loop in C.

Flowchart of Loop with Condition at Bottom

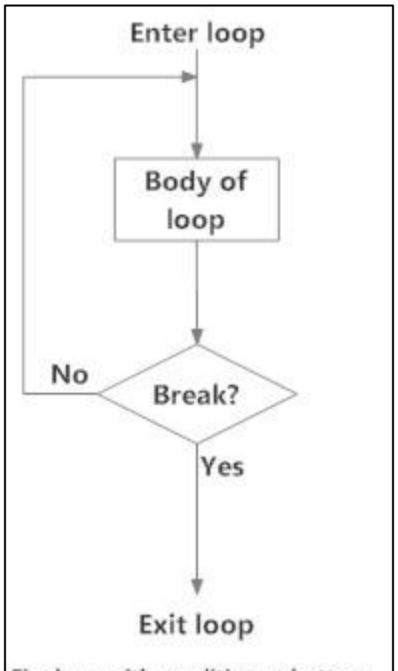


Fig: loop with condition at bottom

Example

```
# Python program to illustrate a loop with condition at the bottom
# Roll a dice untill user chooses to exit

# import random module
import random

while True:
   input("Press enter to roll the dice")

   # get a number between 1 to 6
   num = random.randint(1,6)
   print("You got",num)
   option = input("Roll again?(y/n) ")

   # contion
   if option == 'n':
       break
```

Output

```
Press enter to roll the dice
You got 1
Roll again?(y/n) y
Press enter to roll the dice
You got 5
Roll again?(y/n) n
```

Python Functions

Python Functions

In Python, function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular chucks. As our program grows larger and larger, functions make it more organized and manageable. Furthermore, it avoids repetition and makes code reusable.

Syntax of Function

```
def function_name(parameters):
   """docstring"""
   statement(s)
```

Above shown is a function definition which consists of following components.

1. Keyword `def` marks the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional `return` statement to return a value from the function.

Example of a function

```
def greet(name):
   """This function greets to
   the person passed in as
   parameter"""
   print("Hello, " + name + ". Good morning!")
```

Function Call

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Paul')
Hello, Paul. Good morning!
```

Docstring

The first string after the function header is called the docstring and is short for documentation string. It is used to explain in brief, what a function does. Although optional, documentation is a good programming practice. Unless you can remember what you had for dinner last week, always document your code.

In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as `__doc__` attribute of the function. For example:

```
>>> print(greet.__doc__)
This function greets to
   the person passed into the
   name parameter
```

The return statement

The `return` statement is used to exit a function and go back to the place from where it was called.

Syntax of return

```
return [expression_list]
```

This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the `return` statement itself is not present inside a function, then the function will return the `None` object. For example:

```
>>> print(greet("May"))
Hello, May. Good morning!
None
```

Here, `None` is the returned value.

Example of return

```
def absolute_value(num):
   """This function returns the absolute
      value of the entered number"""

   if num >= 0:
       return num
   else:
       return -num

print(absolute_value(2))
print(absolute_value(-4))
```

Output

```
2
4
```

Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

Lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes. They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Here is an example to illustrate the scope of a variable inside a function.

```
def my_func():
   x = 10
   print("Value inside function:",x)

x = 20
my_func()
print("Value outside function:",x)
```

Output

```
Value inside function: 10
Value outside function: 20
```

Here, we can see that the value of *x* is 20 initially. Even though the function `my_func()` changed the value of *x* to 10, it did not effect the value outside the function. This is because the variable *x* inside the function is different (local to the function) from the one outside. Although they have same names, they are two different variables with different scope.

On the other hand, variables outside of the function are visible from inside. They have a global scope. We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword `global`.

Types of Functions

Basically, we can divide functions into the following two types:

1. Built-in functions - Functions that are built into Python.
2. User-defined functions - Functions defined by the users themselves.

Python Programming Built-in Functions

Functions that come built into the Python language itself are called built-in functions and are readily available to us. Functions like print(), input(), eval() etc. that we have been using, are some examples of the built-in function. There are 68 built-in functions (may change with version) in Python. They are listed below alphabetically along with a brief description.

Python built-in functions

Built-in Function	**Description**
abs()	Return the absolute value of a number.
all()	Return `True` if all elements of the iterable are true (or if the iterable is empty).
any()	Return `True` if any element of the iterable is true. If the iterable is empty, return `False`.
ascii()	Return a string containing a printable representation of an object, but escape the non-ASCII characters.
bin()	Convert an integer number to a binary string.
bool()	Convert a value to a Boolean.
bytearray()	Return a new array of bytes.
bytes()	Return a new "bytes" object.
callable()	Return `True` if the object argument appears callable, `False` if not.
chr()	Return the string representing a character.
classmethod()	Return a class method for the function.
compile()	Compile the source into a code or AST object.
complex()	Create a complex number or convert a string or number to a complex number.
delattr()	Deletes the named attribute of an object.
dict()	Create a new dictionary.

dir()	Return the list of names in the current local scope.
divmod()	Return a pair of numbers consisting of quotient and remainder when using integer division.
enumerate()	Return an enumerate object.
eval()	The argument is parsed and evaluated as a Python expression.
exec()	Dynamic execution of Python code.
filter()	Construct an iterator from elements of iterable for which function returns true.
float()	Convert a string or a number to floating point.
format()	Convert a value to a "formatted" representation.
frozenset()	Return a new `frozenset` object.
getattr()	Return the value of the named attribute of an object.
globals()	Return a dictionary representing the current global symbol table.
hasattr()	Return `True` if the name is one of the object's attributes.
hash()	Return the hash value of the object.
help()	Invoke the built-in help system.
hex()	Convert an integer number to a hexadecimal string.
id()	Return the "identity" of an object.
input()	Reads a line from input, converts it to a string (stripping a trailing newline), and returns that.
int()	Convert a number or string to an integer.
isinstance()	Return `True` if the object argument is an instance.
issubclass()	Return `True` if class is a subclass.

iter()	Return an iterator object.
len()	Return the length (the number of items) of an object.
list()	Return a list.
locals()	Update and return a dictionary representing the current local symbol table.
map()	Return an iterator that applies function to every item of iterable, yielding the results.
max()	Return the largest item in an iterable.
memoryview()	Return a "memory view" object created from the given argument.
min()	Return the smallest item in an iterable.
next()	Retrieve the next item from the iterator.
object()	Return a new featureless object.
oct()	Convert an integer number to an octal string.
open()	Open file and return a corresponding file object.
ord()	Return an integer representing the Unicode.
pow()	Return power raised to a number.
print()	Print objects to the stream.
property()	Return a property attribute.
range()	Return an iterable sequence.
repr()	Return a string containing a printable representation of an object.
reversed()	Return a reverse iterator.
round()	Return the rounded floating point value.
set()	Return a new set object.

setattr()	Assigns the value to the attribute.
slice()	Return a slice object.
sorted()	Return a new sorted list.
staticmethod()	Return a static method for function.
str()	Return a str version of object.
sum()	Sums the items of an iterable from left to right and returns the total.
super()	Return a proxy object that delegates method calls to a parent or sibling class.
tuple()	Return a tuple
type()	Return the type of an object.
vars()	Return the `__dict__` attribute for a module, class, instance, or any other object.
zip()	Make an iterator that aggregates elements from each of the iterables.
__import__()	This function is invoked by the `import` statement.

Python Programming User-defined Functions

Functions that we define ourselves to do certain specific task are referred as user-defined functions. The way in which we define and call functions in Python are already discussed. Functions that readily come with Python are called built-in functions. If we use functions written by others in the form of library, it can be termed as library functions. All the other functions that we write on our own fall under user-defined functions. So, our user-defined function could be a library function to someone else.

Advantages of user-defined functions

1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
3. Programmars working on large project can divide the workload by making different functions.

Example of a user-defined function

```
# Program to illustrate
# the use of user-defined functions

def my_addition(x,y):
   """This function adds two
   numbers and return the result"""
   sum = x + y
   return sum

num1 = float(input("Enter a number: "))
num2 = float(input("Enter another number: "))
print("The sum is", my_addition(num1,num2))
```

Output

```
Enter a number: 2.4
Enter another number: 6.5
The sum is 8.9
```

Explanation

Here, we have defined the function `my_addition()` which adds two numbers and returns the result. This is our user-defined function. We could have multiplied the two numbers inside our function (it's all up to us). But this operation would not be consistent with the name of the function. It would create ambiguity. It is always a good idea to name functions according to the task they perform.

In the above example, `input()`, `print()` and `float()` are built-in functions of the Python programming language.

Python Function Arguments

In user-defined function topic, we learned about defining a function and calling it. Otherwise, the function call will result into an error. Here is an example.

```
def greet(name,msg):
   """This function greets to
   the person with the provided message"""
   print("Hello",name + ', ' + msg)


greet("Monica","Good morning!")
```

Output

```
Hello Monica, Good morning!
```

Here, the function `greet()` has two parameters. Since, we have called this function with two arguments, it runs smoothly and we do not get any error. But if we call it with different number of arguments, the interpreter will complain. Below is a call to this function with one and no arguments along with their respective error messages.

```
>>> greet("Monica")    # only one argument
TypeError: greet() missing 1 required positional argument: 'msg'
>>> greet()    # no arguments
TypeError: greet() missing 2 required positional arguments: 'name' and 'msg'
```

Variable Function Arguments

Up until now functions had fixed number of arguments. In Python there are other ways to define a function which can take variable number of arguments. Three different forms of this type are described below.

Default Arguments

Function arguments can have default values in Python. We can provide a default value to an argument by using the assignment operator (=). Here is an example.

```
def greet(name, msg = "Good morning!"):
   """This function greets to
   the person with the provided message.
   If message is not provided, it defaults
   to "Good morning!" """

   print("Hello",name + ', ' + msg)
```

In this function, the parameter `name` does not have a default value and is required (mandatory) during a call. On the other hand, the parameter `msg` has a default value of `"Good morning!"`. So, it is optional during a call. If a value is provided, it will overwrite the default value. Here are some valid calls to this function.

```
>>> greet("Kate")
Hello Kate, Good morning!

>>> greet("Bruce","How do you do?")
Hello Bruce, How do you do?
```

Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values. This means to say, non-default arguments cannot follow default arguments. For example, if we had defined the function header above as:

```
def greet(msg = "Good morning!", name):
```

We would get an error as:

```
SyntaxError: non-default argument follows default argument
```

Keyword Arguments

When we call a function with some values, these values get assigned to the arguments according to their position. For example, in the above function `greet()`, when we called it as `greet("Bruce","How do you do?")`, the value `"Bruce"` gets assigned to the argument *name* and similarly `"How do you do?"` to *msg*.

Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed. Following calls to the above function are all valid and produce the same result.

```
greet(name = "Bruce",msg = "How do you do?")     # 2 keyword arguments
greet(msg = "How do you do?",name = "Bruce")     # 2 keyword arguments (out of 
order)
greet("Bruce",msg = "How do you do?")            # 1 positional, 1 keyword 
argument
```

As we can see, we can mix positional arguments with keyword arguments during a function call. But we must keep in mind that keyword arguments must follow positional arguments. Having a positional argument after keyword arguments will result into errors. For example the function call as follows:

```
greet(name="Bruce","How do you do?")
```

Will result into error as:

```
SyntaxError: non-keyword arg after keyword arg
```

Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with arbitrary number of arguments. In the function definition we use an asterisk (*) before the parameter name to denote this kind of argument. Here is an example.

```
def greet(*names):
   """This function greets all
   the person in the names tuple."""

   # names is a tuple with arguments
   for name in names:
       print("Hello",name)


greet("Monica","Luke","Steve","John")
```

Output

```
Hello Monica
Hello Luke
Hello Steve
Hello John
```

Here, we have called the function with multiple arguments. These arguments get wrapped up into a tuple before being passed into the function. Inside the function, we use a `for` loop to retrieve all the arguments back.

Python Recursion

Recursion is the process of defining something in terms of itself. A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

Python Recursive Function

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

Following is an example of recursive function to find the factorial of an integer. Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

Example of recursive function

```
# An example of a recursive function to
# find the factorial of a number

def recur_fact(x):
   """This is a recursive function
   to find the factorial of an integer"""

   if x == 1:
       return 1
   else:
       return (x * recur_fact(x-1))


num = int(input("Enter a number: "))
if num >= 1:
   print("The factorial of", num, "is", recur_fact(num))
```

Output

```
Enter a number: 4
The factorial of 4 is 24
```

Explanation

In the above example, `recur_fact()` is a recursive functions as it calls itself. When we call this function with a positive integer, it will recursively call itself by decreasing the number. Each function call multiples the number with the factorial of number-1 until the number is equal to one. This recursive call can be explained in the following steps.

```
recur_fact(4)              # 1st call with 4
4 * recur_fact(3)          # 2nd call with 3
4 * 3 * recur_fact(2)      # 3rd call with 2
4 * 3 * 2 * recur_fact(1)  # 4th call with 1
```

```
4 * 3 * 2 * 1              # retrun from 4th call as number=1
4 * 3 * 2                  # return from 3rd call
4 * 6                      # return from 2nd call
24                         # return from 1st call
```

Our recursion ends when the number reduces to 1. This is called the base condition. Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely. We must avoid infinite recursion.

Advantages of recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

Python Anonymous/Lambda Function

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

Lambda Functions

A lambda function has the following syntax.

Syntax of Lambda Function

```
lambda arguments: expression
```

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Example of Lambda Function

Here is an example of lambda function that doubles the input value.

```
# Program to show the
# use of lambda functions

double = lambda x: x * 2

print(double(5))
```

Output

```
10
```

Explanation

In the above program, `lambda x: x * 2` is the lambda function. Here *x* is the argument and `x * 2` is the expression that gets evaluated and returned. This function has no name. It returns a function object which is assigned to the identifier `double`. We can now call it as a normal function. The statement

```
double = lambda x: x * 2
```

is nearly the same as

```
def double(x):
   return x * 2
```

Use of Lambda Function

We use lambda functions when we require a nameless function for a short period of time. In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like `filter()`, `map()` etc.

Example use with filter()

The `filter()` function in Python takes in a function and a list as arguments. The function is called with all the items in the list and a new list is returned which contains items for which the function evaluats to `True`.

Here is an example use of `filter()` function to filter out only even numbers from a list.

```
# Program to filter out
# only the even items from
# a list using filter() and
# lambda functions

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))
print(new_list)
```

Output

```
[4, 6, 8, 12]
```

Example use with map()

The `map()` function in Python takes in a function and a list. The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Here is an example use of `map()` function to double all the items in a list.

```
# Program to double each
# item in a list using map() and
# lambda functions

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))
print(new_list)
```

Output

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

Python Modules

Modules refer to a file containing Python statements and definitions. A file containing Python code, for e.g.: `example.py`, is called a module and its module name would be `example`. We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code. We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Let us create a module. Type the following and save it as `example.py`.

```
# Python Module example

def add(a, b):
   """This program adds two
   numbers and return the result"""

   result = a + b
   return result
```

Here, we have defined a function `add()` inside a module named `example`. The function takes in two numbers and returns their sum.

Importing modules

We can import the definitions inside a module to another module or the interactive interpreter in Python. We use the `import` keyword to do this. To import our previously defined module `example` we type the following in the Python prompt.

```
>>> import example
```

This does not enter the names of the functions defined in `example` directly in the current symbol table. It only enters the module name `example` there. Using the module name we can access the function using dot (.) operation. For example:

```
>>> example.add(4,5.5)
9.5
```

Python has a ton of standard modules available.. These files are in the Lib directory inside the location where you installed Python. Standard modules can be imported the same way as we import our user-defined modules.

There are various ways to import modules. They are listed as follows.

The import statement

We can import a module using `import` statement and access the definitions inside it using the dot operator as described above. Here is an example.

```
# import statement example
# to import standard module math

import math

print("The value of pi is", math.pi)
```

Output

```
The value of pi is 3.141592653589793
```

Import with renaming

We can import a module by renaming it as follows.

```
# import module by renaming it

import math as m

print("The value of pi is", m.pi)
```

The output of this is same as above. We have renamed the `math` module as `m`. This can save us typing time in some cases. Note that the name `math` is not recognized in our scope. Hence, `math.pi` is invalid, `m.pi` is the correct implementation.

The from...import statement

We can import specific names form a module without importing the module as a whole. Here is an example.

```
# import only pi from math module

from math import pi

print("The value of pi is", pi)
```

The output of this is same as above. We imported only the attribute pi form the module. In such case we don't use the dot operator. We could have imported multiple attributes as follows.

```
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
```

Import all names

We can import all names(definitions) form a module using the following construct.

```
# import all names form
# the standard module math

from math import *

print("The value of pi is", pi)
```

The output of this is same as above. We imported all the definitions from the math module. This makes all names except those beginnig with an underscore, visible in our scope.

Importing everything with the asterisk (*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

Python Module Search Path

While importing a module, Python looks at several places. Interpreter first looks for a built-in module then (if not found) into a list of directories defined in `sys.path`. The search is in this order.

- The current directory.
- `PYTHONPATH` (an environment variable with a list of directory).
- The installation-dependent default directory.

```
>>> import sys
>>> sys.path
['',
'C:\\Python33\\Lib\\idlelib',
'C:\\Windows\\system32\\python33.zip',
'C:\\Python33\\DLLs',
'C:\\Python33\\lib',
'C:\\Python33',
'C:\\Python33\\lib\\site-packages']
```

We can add modify this list to add our own path.

Reloading a module

The Python interpreter imports a module only once during a session. This makes things more efficient. Here is an example to show how this works.

Suppose we have the following code in a module named `my_module`.

```
# This module shows the effect of
#  multiple imports and reload

print("This code got executed")
```

Now we see the effect of multiple imports.

```
>>> import my_module
This code got executed
>>> import my_module
>>> import my_module
```

We can see that our code got executed only once. This goes to say that our module was imported only once.

Now if our module changed during the course of the program, we would have to reload it. One way to do this is to restart the interpreter. But this does not help much. Python provides a neat way of doing this. We can use the `reload()` function inside the `imp` module to reload a module. This is how its done.

```
>>> import imp
>>> import my_module
This code got executed
>>> import my_module
>>> imp.reload(my_module)
This code got executed
<module 'my_module' from '.\\my_module.py'>
```

The dir() built-in function

We can use the `dir()` function to find out names that are defined inside a module. For example, we have defined a function `add()` in the module `example` that we had in the beginning.

```
>>> dir(example)
['__builtins__',
'__cached__',
'__doc__',
'__file__',
'__initializing__',
'__loader__',
'__name__',
'__package__',
'add']
```

Here, we can see a sorted list of names (along with `add`). All other names that begin with an underscore are default Python attributes associated with the module (we did not define them ourself). For example, the `__name__` attribute contains the name of the module.

```
>>> import example
>>> example.__name__
'example'
```

All the names defined in our current namespace can be found out using the `dir()` function without any arguments.

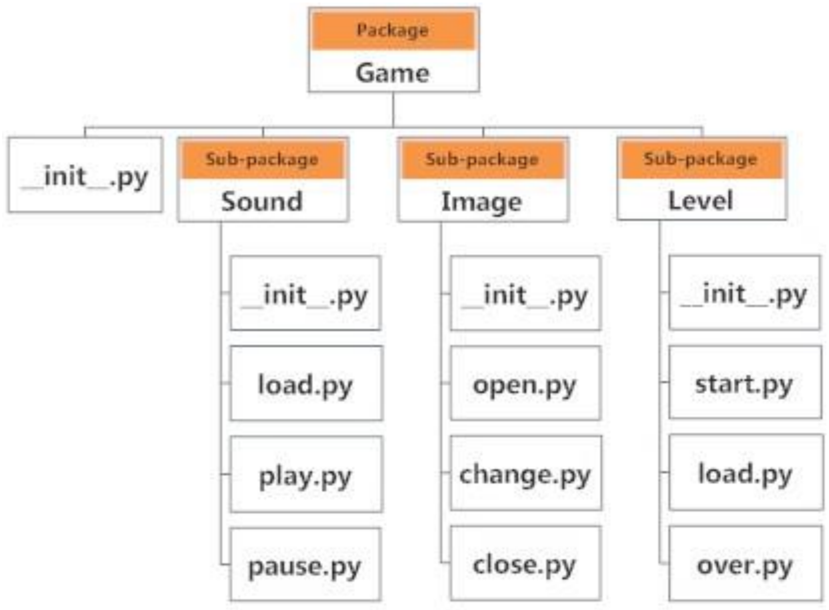
```
>>> a = 1
>>> b = "hello"
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'b', 'math', 'pyscripter']
```

Python Package

We don't usually store all of our files in our computer in the same location. We use a well-organized hierarchy of directories for easier access. Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.

As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear. Similar, as a directory can contain sub-directories and files, a Python package can have sub-packages and modules.

A directory must contain a file named `__init__.py` in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file. Here is an example. Suppose we are developing a game, one possible organization of packages and modules could be as shown in the figure below.



Importing module from a package

We can import modules from packages using the dot (.) operator. For example, if want to import the `start` module in the above example, it is done as follows.

```
import Game.Level.start
```

Now if this module contains a function named `select_difficulty()`, we must use the full name to reference it.

```
Game.Level.start.select_difficulty(2)
```

If this construct seems lengthy, we can import the module without the package prefix as follows.

```
from Game.Level import start
```

We can now call the function simply as follows.

```
start.select_difficulty(2)
```

Yet another way of importing just the required function (or class or variable) form a module within a package would be as follows.

```
from Game.Level.start import select_difficulty
```

Now we can directly call this function.

```
select_difficulty(2)
```

Although easier, this method is not recommended. Using the full namespace avoids confusion and prevents two same identifier names from colliding. While importing packages, Python looks in the list of directories defined in `sys.path`, similar as for module search path.

Python Native Datatypes

- Python Numbers
- Python List
- Python Tuple
- Python String
- Python Set
- Python Dictionary

Python Numbers, Type Conversion and Mathematics

Python supports integers, floating point numbers and complex numbers. They are defined as `int`, `float` and `complex` class in Python. Integers and floating points are separated by the presence or absence of a decimal point. 5 is integer whereas 5.0 is a floating point number. Complex numbers are written in the form, `x + yj`, where *x* is the real part and *y* is the imaginary part. We can use the `type()` function to know which class a variable or a value belongs to and `isinstance()` function to check if it belongs to a particular class.

```
>>> a = 5
>>> type(a)
<class 'int'>
>>> type(5.0)
<class 'float'>
>>> c = 5 + 3j
>>> c + 3
(8+3j)
>>> isinstance(c, complex)
True
>>> 1.1234567890123456789
1.1234567890123457
```

While integers can be of any length, a floating point number is accurate only up to 15 decimal places (the 16th place is inaccurate).

Numbers we deal with everyday are decimal (base 10) number system. But computer programmers (generally embedded programmer) need to work with binary (base 2), hexadecimal (base 16) and octal (base 8) number systems. In Python we can represent these numbers by appropriately placing a prefix before that number. Following table lists these prefix.

Number system prefix for Python numbers

Number System	Prefix
Binary	'0b' or '0B'
Octal	'0o' or '0O'
Hexadecimal	'0x' or '0X'

Here are some examples

```
>>> 0b1101011     # 107
107
>>> 0xFB + 0b10   # 251 + 2
253
```

```
>>> 0o15          # 13
13
```

Type Conversion

We can convert one type of number into another. This is also known as coercion. Operations like addition, subtraction coerce integer to float implicitly (automatically), if one of the operand is float.

```
>>> 1 + 2.0
3.0
```

We can see above that 1 (integer) is coerced into 1.0 (float) for addition and the result is also a floating point number.

We can use built-in functions like `int()`, `float()` and `complex()` to convert between types explicitly. These function can even convert from strings.

```
>>> int(2.3)
2
>>> int(-2.8)
-2
>>> float(5)
5.0
>>> complex('3+5j')
(3+5j)
```

When converting from float to integer, the number gets truncated (integer that is closer to zero).

Python Decimal

Python built-in class `float` performs some calculations that might amaze us. We all know that the sum of 1.1 and 2.2 is 3.3, but Python seems to disagree.

```
>>> (1.1 + 2.2) == 3.3
False
```

What is going on? It turns out that floating-point numbers are implemented in computer hardware as binary fractions, as computer only understands binary (0 and 1). Due to this reason, most of the decimal fractions we know, cannot be accurately stored in our computer. Let's take an example. We cannot represent the fraction 1/3 as a decimal number. This will give 0.33333333... which is infinitely long, and we can only approximate it. Turns out decimal fraction 0.1 will result into an infinitely long binary fraction of 0.000110011001100110011... and our computer only stores a finite number of it. This will only approximate 0.1 but never be equal. Hence, it is the limitation of our computer hardware and not an error in Python.

```
>>> 1.1 + 2.2
3.3000000000000003
```

To overcome this issue, we can use `decimal` module that comes with Python. While floating point numbers have precision up to 15 decimal places, the decimal module has user settable precision.

```
>>> import decimal
>>> 0.1
0.1
>>> decimal.Decimal(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

This module is used when we want to carry out decimal calculations like we learned in school. It also preserves significance. We know 25.50 kg is more accurate than 25.5 kg as it has two significant decimal places compared to one.

```
>>> from decimal import Decimal as D
>>> D('1.1') + D('2.2')
Decimal('3.3')
>>> D('1.2') * D('2.50')
Decimal('3.000')
```

Notice the trailing zeroes in the above example. We might ask, why not implement Decimal every time, instead of float? The main reason is efficiency. Floating point operations are carried out must faster than Decimal operations. We generally use Decimal in the following cases.

- When we are making financial applications that need exact decimal representation.
- When we want to control the level of precision required.
- When we want to implement the notion of significant decimal places.
- When we want the operations to be carried out like we did at school

Python Fractions

Python provides operations involving fractional numbers through its `fractions` module. A fraction has a numerator and a denominator, both of which are integers. This module has support for rational number arithmetic.

We can create Fraction objects in various ways.

```
>>> import fractions
>>> fractions.Fraction(1.5)
Fraction(3, 2)
>>> fractions.Fraction(5)
Fraction(5, 1)
>>> fractions.Fraction(1,3)
Fraction(1, 3)
```

While creating Fraction from `float`, we might get some unusual results. This is due to the imperfect binary floating point number representation as discussed in the previous section. Fortunately, this class allows us to instantiate with string as well. This is the preferred options when using decimal numbers.

```
>>> fractions.Fraction(1.1)   # as float
Fraction(2476979795053773, 2251799813685248)
>>> fractions.Fraction('1.1') # as string
Fraction(11, 10)
```

This datatype supports all basic operations. Here are few examples.

```
>>> from fractions import Fraction as F
>>> F(1,3) + F(1,3)
Fraction(2, 3)
>>> 1 / F(5,6)
Fraction(6, 5)
>>> F(-3,10) > 0
False
>>> F(-3,10) < 0
True
```

Python Mathematics

Python offers modules like `math` and `random` to carry out different mathematics like trigonometry, logarithms, probability and statistics, etc.

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.cos(math.pi)
-1.0
>>> math.exp(10)
22026.465794806718
>>> math.log10(1000)
3.0
>>> math.sinh(1)
1.1752011936438014
>>> math.factorial(6)
720
```

Full list functions and attributes available in Python math module.

```
>>> import random
>>> random.randrange(10,20)
16
>>> x = ['a', 'b', 'c', 'd', 'e']
>>> random.choice(x)
'd'
>>> random.shuffle(x)
>>> x
['a', 'b', 'd', 'c', 'e']
>>> random.random()
0.8025729372178537
```

Python List

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

Creating a List

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

```
# empty list
my_list = []

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

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

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

Accessing Elements in a List

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

Indexing

We can use the index operator [] to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4. Trying to access an element other that this will raise an `IndexError`. The index must be an integer. We can't use float or other types, this will result into `TypeError`. Nested list are accessed using nested indexing.

```
>>> my_list = ['p','r','o','b','e']
>>> my_list[0]
'p'
>>> my_list[2]
'o'
>>> my_list[4]
'e'
>>> my_list[4.0]
...
TypeError: list indices must be integers, not float
>>> my_list[5]
...
IndexError: list index out of range
```

```
>>> n_list = ["Happy", [2,0,1,5]]
>>> n_list[0][1]    # nested indexing
'a'
>>> n_list[1][3]    # nested indexing
5
```

Negative indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

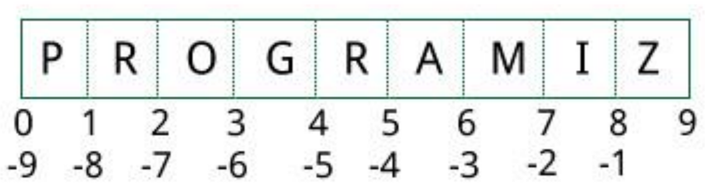
```
>>> my_list = ['p','r','o','b','e']
>>> my_list[-1]
'e'
>>> my_list[-5]
'p'
```

Slicing

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

```
>>> my_list = ['p','r','o','g','r','a','m','i','z']
>>> my_list[2:5]    # elements 3rd to 5th
['o', 'g', 'r']
>>> my_list[:-5]    # elements beginning to 4th
['p', 'r', 'o', 'g']
>>> my_list[5:]     # elements 6th to end
['a', 'm', 'i', 'z']
>>> my_list[:]      # elements beginning to end
['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']
```

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



Changing or Adding Elements to a List

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

```
>>> odd = [2, 4, 6, 8]     # mistake values
>>> odd[0] = 1             # change the 1st item
>>> odd
[1, 4, 6, 8]
>>> odd[1:4] = [3, 5, 7]  # change 2nd to 4th items
>>> odd                   # changed values
[1, 3, 5, 7]
```

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

```
>>> odd
[1, 3, 5]
>>> odd.append(7)
>>> odd
[1, 3, 5, 7]
>>> odd.extend([9, 11, 13])
>>> odd
[1, 3, 5, 7, 9, 11, 13]
```

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

```
>>> odd
[1, 3, 5]
>>> odd + [9, 7, 5]
[1, 3, 5, 9, 7, 5]
>>> ["re"] * 3
['re', 're', 're']
```

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

```
>>> odd
[1, 9]
>>> odd.insert(1,3)
>>> odd
[1, 3, 9]
>>> odd[2:2] = [5, 7]
>>> odd
[1, 3, 5, 7, 9]
```

Deleting or Removing Elements from a List

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

```
>>> my_list = ['p','r','o','b','l','e','m']
>>> del my_list[2]    # delete one item
>>> my_list
['p', 'r', 'b', 'l', 'e', 'm']
>>> del my_list[1:5]  # delete multiplt items
>>> my_list
['p', 'm']
```

```
>>> del my_list       # delete entire list
>>> my_list
...
NameError: name 'my_list' is not defined
```

We can use `remove()` method to remove the given item or `pop()` method to remove an item at the given index. The `pop()` method removes and returns the last item if index is not provided. This helps us implement lists as stacks (first in, last out data structure). We can also use the `clear()` method to empty a list.

```
>>> my_list = ['p','r','o','b','l','e','m']
>>> my_list.remove('p')
>>> my_list
['r', 'o', 'b', 'l', 'e', 'm']
>>> my_list.pop(1)
'o'
>>> my_list
['r', 'b', 'l', 'e', 'm']
>>> my_list.pop()
'm'
>>> my_list
['r', 'b', 'l', 'e']
>>> my_list.clear()
>>> my_list
[]
```

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

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

Python List Methods

Methods that are available with list object in Python programming are tabulated below. They are accessed as `list.method()`. Some of the methods have already been used above.

Python List Methods

Method	Description
append(*x*)	Add item *x* at the end of the list
extend(*L*)	Add all items in given list *L* to the end

insert(*i*, *x*)	Insert item *x* at position *i*
remove(*x*)	Remove first item that is equal to *x*, from the list
pop([*i*])	Remove and return item at position *i* (last item if *i* is not provided)
clear()	Remove all items and empty the list
index(*x*)	Return index of first item that is equal to *x*
count(*x*)	Return the number of items that is equal to *x*
sort()	Sort items in a list in ascending order
reverse()	Reverse the order of items in a list
copy()	Return a shallow copy of the list

```
>>> my_list = [3, 8, 1, 6, 0, 8, 4]
>>> my_list.index(8)
1
>>> my_list.count(8)
2
>>> my_list.sort()
>>> my_list
[0, 1, 3, 4, 6, 8, 8]
>>> my_list.reverse()
>>> my_list
[8, 8, 6, 4, 3, 1, 0]
```

Python List Comprehension

List comprehension is an elegant and concise way to create new list from an existing list in Python. List comprehension consists of an expression followed by `for` statement inside square brackets. Here is an example to make a list with each item being increasing power of 2.

```
>>> pow2 = [2 ** x for x in range(10)]
>>> pow2
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

This code is equivalent to

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

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

```
>>> pow2 = [2 ** x for x in range(10) if x > 5]
>>> pow2
[64, 128, 256, 512]
>>> odd = [x for x in range(20) if x % 2 == 1]
>>> odd
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> [x+y for x in ['Python ','C '] for y in ['Language','Programming']]
['Python Language', 'Python Programming', 'C Language', 'C Programming']
```

Other List Operations

List Membership Test

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

```
>>> my_list = ['p','r','o','b','l','e','m']
>>> 'p' in my_list
True
>>> 'a' in my_list
False
>>> 'c' not in my_list
True
```

Iterating Through a List

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

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

Built-in Functions with List

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

Built-in Functions with List

Function	**Description**
all()	Return `True` if all elements of the list are true (or if the list is empty).
any()	Return `True` if any element of the list is true. If the list is empty, return `False.`
enumerate()	Return an enumerate object. It contains the index and value of all the items of list as a tuple.
len()	Return the length (the number of items) in the list.
list()	Convert an iterable (tuple, string, set, dictionary) to a list.
max()	Return the largest item in the list.
min()	Return the smallest item in the list
sorted()	Return a new sorted list (does not sort the list itself).
sum()	Retrun the sum of all elements in the list.

Python Tuple

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

Creating a Tuple

A tuple is created by placing all the items (elements) inside a parentheses (), separated by comma. The parentheses are optional but is a good practice to write it. A tuple can have any number of items and they may be of different types (integer, float, list, string etc.).

```
# empty tuple
my_tuple = ()

# tuple having integers
my_tuple = (1, 2, 3)

# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)

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

# tuple can be created without parentheses
# also called tuple packing
my_tuple = 3, 4.6, "dog"
# tuple unpacking is also possible
a, b, c = my_tuple
```

Creating a tuple with one element is a bit tricky. Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is in fact a tuple.

```
>>> my_tuple = ("hello")   # only parentheses is not enough
>>> type(my_tuple)
<class 'str'>
>>> my_tuple = ("hello",)  # need a comma at the end
>>> type(my_tuple)
<class 'tuple'>
>>> my_tuple = "hello",    # parentheses is optional
>>> type(my_tuple)
<class 'tuple'>
```

Accessing Elements in a Tuple

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

Indexing

We can use the index operator [] to access an item in a tuple. Index starts from 0. So, a tuple having 6 elements will have index from 0 to 5. Trying to access an element other that this will raise an `IndexError`. The index must be an integer. We can't use float or other types, this will result into `TypeError`. Nested tuple are accessed using nested indexing.

```
>>> my_tuple = ['p','e','r','m','i','t']
>>> my_tuple[0]
'p'
>>> my_tuple[5]
't'
>>> my_tuple[6]   # index must be in range
...
IndexError: list index out of range
>>> my_tuple[2.0] # index must be an integer
...
TypeError: list indices must be integers, not float
>>> n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
>>> n_tuple[0][3]  # nested index
's'
>>> n_tuple[1][1]  # nested index
4
>>> n_tuple[2][0]  # nested index
1
```

Negative Indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

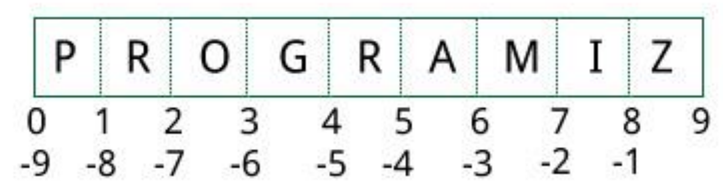
```
>>> my_tuple = ['p','e','r','m','i','t']
>>> my_tuple[-1]
't'
>>> my_tuple[-6]
'p'
```

Slicing

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

```
>>> my_tuple = ('p','r','o','g','r','a','m','i','z')
>>> my_tuple[1:4]  # elements 2nd to 4th
('r', 'o', 'g')
>>> my_tuple[:-7]  # elements beginning to 2nd
('p', 'r')
>>> my_tuple[7:]   # elements 8th to end
('i', 'z')
>>> my_tuple[:]    # elements beginning to end
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

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



Changing or Deleting a Tuple

Unlike lists, tuples are immutable. This means that elements of a tuple cannot be changed once it has been assigned. But if the element is itself a mutable datatype like list, its nested items can be changed. We can also assign a tuple to different values (reassignment).

```
>>> my_tuple = (4, 2, 3, [6, 5])
>>> my_tuple[1] = 9  # we cannot change an element
...
TypeError: 'tuple' object does not support item assignment
>>> my_tuple[3] = 9  # we cannot change an element
...
TypeError: 'tuple' object does not support item assignment
>>> my_tuple[3][0] = 9   # but item of mutable element can be changed
>>> my_tuple
(4, 2, 3, [9, 5])
>>> my_tuple = ('p','r','o','g','r','a','m','i','z') # tuples can be
reassigned
>>> my_tuple
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

We can use + operator to combine two tuples. This is also called concatenation. The * operator repeats a tuple for the given number of times. These operations result into a new tuple.

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> ("Repeat",) * 3
('Repeat', 'Repeat', 'Repeat')
```

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

```
>>> my_tuple = ('p','r','o','g','r','a','m','i','z')
>>> del my_tuple[3] # can't delete items
...
TypeError: 'tuple' object doesn't support item deletion
>>> del my_tuple    # can delete entire tuple
>>> my_tuple
...
```

```
NameError: name 'my_tuple' is not defined
```

Python Tuple Methods

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

Python Tuple Method

Method	Description
count(*x*)	Return the number of items that is equal to *x*
index(*x*)	Return index of first item that is equal to *x*

```
>>> my_tuple = ('a','p','p','l','e',)
>>> my_tuple.count('p')
2
>>> my_tuple.index('l')
3
```

Other Tuple Operations

Tuple Membership Test

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

```
>>> my_tuple = ('a','p','p','l','e',)
>>> 'a' in my_tuple
True
>>> 'b' in my_tuple
False
>>> 'g' not in my_tuple
True
```

Iterating Through a Tuple

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

```
>>> for name in ('John','Kate'):
...     print("Hello",name)
...
Hello John
Hello Kate
```

Built-in Functions with Tuple

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

Built-in Functions with Tuple

Function	Description
all()	Return `True` if all elements of the tuple are true (or if the tuple is empty).
any()	Return `True` if any element of the tuple is true. If the tuple is empty, return `False`.
enumerate()	Return an enumerate object. It contains the index and value of all the items of tuple as pairs.
len()	Return the length (the number of items) in the tuple.
max()	Return the largest item in the tuple.
min()	Return the smallest item in the tuple
sorted()	Take elements in the tuple and return a new sorted list (does not sort the tuple itself).
sum()	Retrun the sum of all elements in the tuple.
tuple()	Convert an iterable (list, string, set, dictionary) to a tuple.

Advantage of Tuple over List

Tuples and list look quite similar except the fact that one is immutable and the other is mutable. We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes. There are some advantages of implementing a tuple than a list. Here are a few of them.

- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Python Strings

String is a sequence of characters. A character is simply a symbol. For example, the English language has 26 characters. Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0's and 1's. This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encoding used.

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

Creating a String

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

```
# all of the following are equivalent
my_string = 'Hello'
my_string = "Hello"
my_string = '''Hello'''
my_string = """Hello"""

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

Accessing Characters in a String

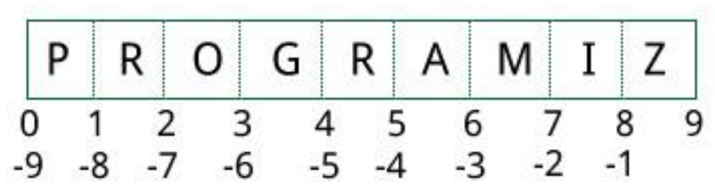
We can access individual characters using indexing and a range of characters using slicing. Index starts from 0. Trying to access a character out of index range will raise an `IndexError`. The index must be an integer. We can't use float or other types, this will result into `TypeError`.

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on. We can access a range of items in a string by using the slicing operator (colon).

```
>>> my_string = 'programiz'
>>> my_string[0]   # 1st character
'p'
>>> my_string[-1]  # last character
'z'
>>> my_string[15]  # index must be in range
...
IndexError: string index out of range
>>> my_string[1.5] # index must be an integer
...
TypeError: string indices must be integers
```

```
>>> my_string[1:5]  # slicing 2nd to 5th character
'rogr'
>>> my_string[5:-2] # slicing 6th to 7th character
'am'
```

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



Changing or Deleting a String

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

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

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

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

Python String Operations

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

Concatenation

Joining of two or more strings into a single one is called concatenation. The `+` operator does this in Python. Simply writing two string literals together also concatenates them. The `*` operator can be used to repeat the string for a given number of times. Finally, if we want to concatenate strings in different lines, we can use parentheses.

```
>>> # using +
>>> 'Hello ' + 'World!'
'Hello World!'

>>> # two string literals together
>>> 'Hello ''World!'
'Hello World!'

>>> # using *
>>> 'Hello ' * 3
'Hello Hello Hello '

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

Iterating Through String

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

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

String Membership Test

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

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

Built-in functions

Various built-in functions that work with sequence, works with string as well. Some of the commonly used ones are `enumerate()` and `len()`. The `enumerate()` function returns an enumerate object. It contains the index and value of all the items in the string as pairs. This can be useful for iteration. Similarly, `len()` returns the length (number of characters) of the string.

```
>>> list(enumerate('cold'))
[(0, 'c'), (1, 'o'), (2, 'l'), (3, 'd')]
>>> long_word = 'Pneumonoultramicroscopicsilicovolcanoconiosis'
>>> len(long_word)
45
```

Python String Formatting

Escape Sequence

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

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

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

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

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

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

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

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

Escape Sequence in Python

Escape Sequence	Description
\newline	Backslash and newline ignored
\\	Backslash
\'	Single quote
\"	Double quote
\a	ASCII Bell
\b	ASCII Backspace
\f	ASCII Formfeed
\n	ASCII Linefeed
\r	ASCII Carriage Return
\t	ASCII Horizontal Tab
\v	ASCII Vertical Tab
\ooo	Character with octal value ooo
\xHH	Character with hexadecimal value HH

Here are some examples

```
>>> print("C:\\Python32\\Lib")
C:\Python32\Lib

>>> print("This is printed\nin two lines")
This is printed
in two lines

>>> print("This is \x48\x45\x58 representation")
This is HEX representation
```

Raw String

Sometimes we may wish to ignore the escape sequences inside a string. To do this we can place `r` or `R` in front of the string. This will imply that it is a raw string and any escape sequence inside it will be ignored.

```
>>> print("This is \x61 \ngood example")
This is a
good example
>>> print(r"This is \x61 \ngood example")
This is \x61 \ngood example
```

The format() Method

The `format()` method that is available with the string object is very versatile and powerful in formatting strings. Format strings contains curly braces {} as placeholders or replacement fields which gets replaced. We can use positional arguments or keyword arguments to specify the order.

```
>>> # default(implicit) order
>>> "{}, {} and {}".format('John','Bill','Sean')
'John, Bill and Sean'

>>> # order using positional argument
>>> "{1}, {0} and {2}".format('John','Bill','Sean')
'Bill, John and Sean'

>>> # order using keyword argument
>>> "{s}, {b} and {j}".format(j='John',b='Bill',s='Sean')
'Sean, Bill and John'
```

The `format()` method can have optional format specifications. They are separated from field name using colon. For example, we can left-justify <, right-justify > or center ^ a string in the given space. We can also format integers as binary, hexadecimal etc. and floats can be rounded or displayed in the exponent format. There are a ton of formatting you can use. Visit here for all the string formatting available with the `format()` method.

```
>>> # formatting integers
>>> "Binary representation of {0} is {0:b}".format(12)
'Binary representation of 12 is 1100'

>>> # formatting floats
>>> "Exponent representation: {0:e}".format(1566.345)
'Exponent representation: 1.566345e+03'

>>> # round off
>>> "One third is: {0:.3f}".format(1/3)
'One third is: 0.333'

>>> # string alignment
>>> "|{:<10}|{:^10}|{:>10}|".format('butter','bread','ham')
'|butter    |  bread   |       ham|'
```

Old style formatting

We can even format strings like the old `sprintf()` style used in C programming language. We use the `%` operator to accomplish this.

```
>>> x = 12.3456789
>>> print('The value of x is %3.2f' %x)
The value of x is 12.35
>>> print('The value of x is %3.4f' %x)
The value of x is 12.3457
```

Python String Methods

There are numerous methods available with the string object. The `format()` method that we mentioned above is one of them. Some of the commonly used methods are `lower()`, `upper()`, `join()`, `split()`, `find()`, `replace()` etc.

```
>>> "PrOgRaMiZ".lower()
'programiz'
>>> "PrOgRaMiZ".upper()
'PROGRAMIZ'
>>> "This will split all words into a list".split()
['This', 'will', 'split', 'all', 'words', 'into', 'a', 'list']
>>> ' '.join(['This', 'will', 'join', 'all', 'words', 'into', 'a', 'string'])
'This will join all words into a string'
>>> 'Happy New Year'.find('ew')
7
>>> 'Happy New Year'.replace('Happy','Brilliant')
'Brilliant New Year'
```

Python Sets

Set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable. However, the set itself is mutable (we can add or remove items). Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

Creating a Set in Python

A set is created by placing all the items (elements) inside curly braces {}, separated by comma or by using the built-in function `set()`. It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have a mutable element, like list, set or dictionary, as its element.

```
>>> # set of integers
>>> my_set = {1, 2, 3}

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

>>> # set donot have duplicates
>>> {1,2,3,4,3,2}
{1, 2, 3, 4}

>>> # set cannot have mutable items
>>> my_set = {1, 2, [3, 4]}
...
TypeError: unhashable type: 'list'

>>> # but we can make set from a list
>>> set([1,2,3,2])
{1, 2, 3}
```

Creating an empty set is a bit tricky. Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements we use the `set()` function without any argument.

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

Changing a Set in Python

Sets are mutable. But since they are unordered, indexing have no meaning. We cannot access or change an element of set using indexing or slicing. Set does not support it. We can add single elements using the method `add()`. Multiple elements can be added using `update()` method. The `update()` method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

```
>>> my_set = {1,3}
>>> my_set[0]
...
TypeError: 'set' object does not support indexing
>>> my_set.add(2)
>>> my_set
{1, 2, 3}
>>> my_set.update([2,3,4])
>>> my_set
{1, 2, 3, 4}
>>> my_set.update([4,5], {1,6,8})
>>> my_set
{1, 2, 3, 4, 5, 6, 8}
```

Removing Elements from a Set

A particular item can be removed from set using methods like `discard()` and `remove()`. The only difference between the two is that, while using `discard()` if the item does not exist in the set, it remains unchanged. But `remove()` will raise an error in such condition. The following example will illustrate this.

```
>>> my_set = {1, 3, 4, 5, 6}
>>> my_set.discard(4)
>>> my_set
{1, 3, 5, 6}
>>> my_set.remove(6)
>>> my_set
{1, 3, 5}
>>> my_set.discard(2)
>>> my_set
{1, 3, 5}
>>> my_set.remove(2)
...
KeyError: 2
```

Similarly, we can remove and return an item using the `pop()` method. Set being unordered, there is no way of determining which item will be popped. It is completely arbitrary. We can also remove all items from a set using `clear()`.

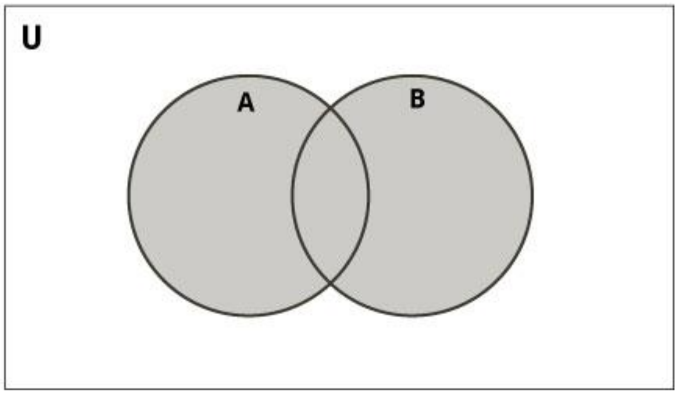
```
>>> my_set = set("HelloWorld")
>>> my_set.pop()
'r'
>>> my_set.pop()
'W'
>>> my_set
{'d', 'e', 'H', 'o', 'l'}
>>> my_set.clear()
>>> my_set
set()
```

Python Set Operation

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

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

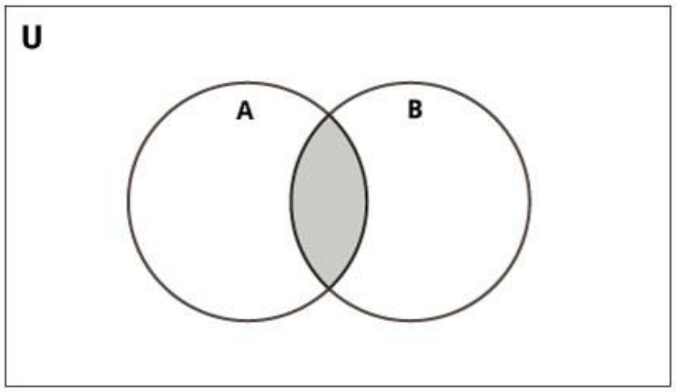
Set Union



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

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

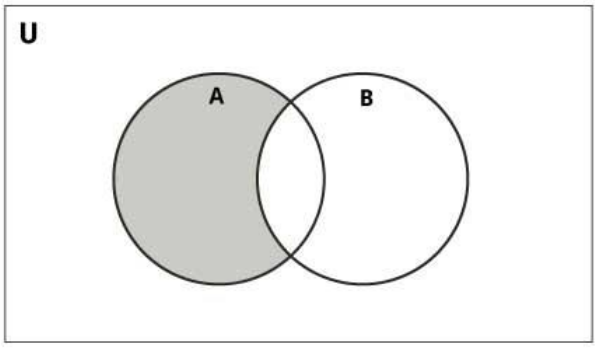
Set Intersection



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

```
>>> A & B
{4, 5}
>>> A.intersection(B)
{4, 5}
>>> B.intersection(A)
{4, 5}
```

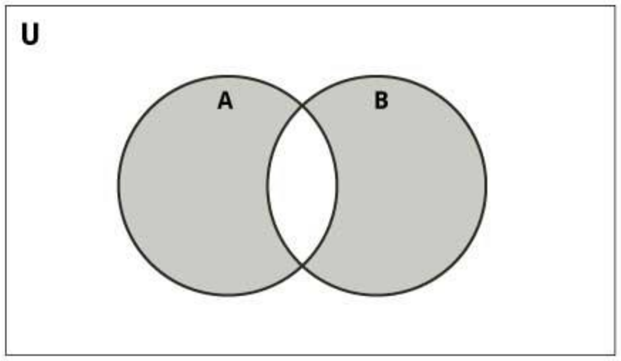
Set Difference



Difference of *A* and *B* (*A* - *B*) is a set of elements that are only in *A* but not in *B*. Similarly, *B* - *A* is a set of element in *B* but not in *A*. Difference is performed using – operator. Same can be accomplished using the method `difference()`.

```
>>> A - B
{1, 2, 3}
>>> A.difference(B)
{1, 2, 3}
>>> B - A
{8, 6, 7}
>>> B.difference(A)
{8, 6, 7}
```

Set Symmetric Difference



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

```
>>> A ^ B
{1, 2, 3, 6, 7, 8}
>>> A.symmetric_difference(B)
{1, 2, 3, 6, 7, 8}
>>> B.symmetric_difference(A)
{1, 2, 3, 6, 7, 8}
```

Python Set Methods

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

Python Set Methods

Method	**Description**
add()	Add an element to a set
clear()	Remove all elemets form a set
copy()	Return a shallow copy of a set
difference()	Return the difference of two or more sets as a new set
difference_update()	Remove all elements of another set from this set
discard()	Remove an element from set if it is a member. (Do nothing if the element is not in set)
intersection()	Return the intersection of two sets as a new set
intersection_update()	Update the set with the intersection of itself and another
isdisjoint()	Return `True` if two sets have a null intersection
issubset()	Return `True` if another set contains this set

issuperset()	Return `True` if this set contains another set
pop()	Remove and return an arbitary set element. Raise `KeyError` if the set is empty
remove()	Remove an element from a set. It the element is not a member, raise a `KeyError`
symmetric_difference()	Return the symmetric difference of two sets as a new set
symmetric_difference_update()	Update a set with the symmetric difference of itself and another
union()	Return the union of sets in a new set
update()	Update a set with the union of itself and others

Other Set Operations

Set Membership Test

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

```
>>> my_set = set("apple")
>>> 'a' in my_set
True
>>> 'p' not in my_set
False
```

Iterating Through a Set

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

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

Built-in Functions with Set

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

Built-in Functions with Set

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

Python Frozenset

Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets. Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets are hashable and can be used as keys to a dictionary.

Frozensets can be created using the function `frozenset()`. This datatype supports methods like `copy()`, `difference()`, `intersection()`, `isdisjoint()`, `issubset()`, `issuperset()`, `symmetric_difference()` and `union()`. Being immutable it does not have method that add or remove elements.

```
>>> A = frozenset([1, 2, 3, 4])
>>> B = frozenset([3, 4, 5, 6])
>>> A.isdisjoint(B)
False
```

```
>>> A.difference(B)
frozenset({1, 2})
>>> A | B
frozenset({1, 2, 3, 4, 5, 6})
>>> A.add(3)
...
AttributeError: 'frozenset' object has no attribute 'add'
```

Python Dictionary

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

Creating a Dictionary

Creating a dictionary is as simple as placing items inside curly braces {} separated by comma. An item has a key and the corresponding value expressed as a pair, key: value. While values can be of any datatype and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique. We can also create a dictionary using the built-in function `dict()`.

```
# empty dictionary
my_dict = {}

# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()
my_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
```

Accessing Elements in a Dictionary

While indexing is used with other container types to access values, dictionary uses keys. Key can be used either inside square brackets or with the `get()` method. The difference while using `get()` is that it returns `None` instead of `KeyError`, if the key is not found.

```
>>> my_dict = {'name':'Ranjit', 'age': 26}
>>> my_dict['name']
'Ranjit'

>>> my_dict.get('age')
26

>>> my_dict.get('address')

>>> my_dict['address']
...
KeyError: 'address'
```

Changing or Adding Elements in a Dictionary

Dictionary are mutable. We can add new items or change the value of existing items using assignment operator. If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

```
>>> my_dict
{'age': 26, 'name': 'Ranjit'}

>>> my_dict['age'] = 27  # update value
>>> my_dict
{'age': 27, 'name': 'Ranjit'}

>>> my_dict['address'] = 'Downtown'  # add item
>>> my_dict
{'address': 'Downtown', 'age': 27, 'name': 'Ranjit'}
```

Deleting or Removing Elements from a Dictionary

We can remove a particular item in a dictionary by using the method `pop()`. This method removes as item with the provided key and returns the value. The method, `popitem()` can be used to remove and return an arbitrary item (key, value) form the dictionary. All the items can be removed at once using the `clear()` method. We can also use the `del` keyword to remove individual items or the entire dictionary itself.

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

>>> squares.pop(4)  # remove a particular item
16
>>> squares
{1: 1, 2: 4, 3: 9, 5: 25}

>>> squares.popitem()  # remove an arbitrary item
(1, 1)
>>> squares
{2: 4, 3: 9, 5: 25}

>>> del squares[5]  # delete a particular item
>>> squares
{2: 4, 3: 9}

>>> squares.clear()  # remove all items
>>> squares
{}

>>> del squares  # delete the dictionary itself
>>> squares
...
NameError: name 'squares' is not defined
```

Python Dictionary Methods

Methods that are available with dictionary are tabulated below. Some of them have already been used in the above examples.

Python Dictionary Methods

Method	Description
clear()	Remove all items form the dictionary.
copy()	Return a shallow copy of the dictionary.
fromkeys(*seq*[, *v*])	Return a new dictionary with keys from *seq* and value equal to *v* (defaults to `None`).
get(*key*[,*d*])	Return the value of *key*. If *key* doesnot exit, return *d* (defaults to `None`).
items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.
pop(*key*[,*d*])	Remove the item with *key* and return its value or *d* if *key* is not found. If *d* is not provided and *key* is not found, raises `KeyError`.
popitem()	Remove and return an arbitary item (key, value). Raises `KeyError` if the dictionary is empty.
setdefault(*key*[,*d*])	If *key* is in the dictionary, return its value. If not, insert *key* with a value of *d* and return *d* (defaults to `None`).
update([*other*])	Update the dictionary with the key/value pairs from *other*, overwriting existing keys.
values()	Return a new view of the dictionary's values

Here are a few example use of these methods.

```
>>> marks = {}.fromkeys(['Math','English','Science'], 0)
>>> marks
{'English': 0, 'Math': 0, 'Science': 0}
```

```
>>> for item in marks.items():
...     print(item)
...
('English', 0)
('Math', 0)
('Science', 0)

>>> list(sorted(marks.keys()))
['English', 'Math', 'Science']
```

Python Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create new dictionary from an iterable in Python. Dictionary comprehension consists of an expression pair (key: value) followed by `for` statement inside curly braces {}. Here is an example to make a dictionary with each item being a pair of a number and its square.

```
>>> squares = {x: x*x for x in range(6)}
>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

This code is equivalent to

```
squares = {}
for x in range(6):
   squares[x] = x*x
```

A dictionary comprehension can optionally contain more `for` or `if` statements. An optional `if` statement can filter out items to form the new dictionary. Here are some examples to make dictionary with only odd items.

```
>>> odd_squares = {x: x*x for x in range(11) if x%2 == 1}
>>> odd_squares
{1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

Other Dictionary Operations

Dictionary Membership Test

We can test if a key is in a dictionary or not using the keyword `in`. Notice that membership test is for keys only, not for values.

```
>>> squares
{1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
>>> 1 in squares
True

>>> 2 not in squares
True

>>> # membership tests for key only not value
>>> 49 in squares
False
```

Iterating Through a Dictionary

Using a `for` loop we can iterate though each key in a dictionary.

```
>>> squares
{1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
>>> for i in squares:
...     print(squares[i])
...
1
9
81
25
49
```

Built-in Functions with Dictionary

Built-in functions like `all()`, `any()`, `len()`, `cmp()`, `sorted()` etc. are commonly used with dictionary to perform different tasks.

Built-in Functions with Dictionary

Function	**Description**
all()	Return `True` if all keys of the dictionary are true (or if the dictionary is empty).
any()	Return `True` if any key of the dictionary is true. If the dictionary is empty, return `False`.
len()	Return the length (the number of items) in the dictionary.
cmp()	Compares items of two dictionaries.
sorted()	Return a new sorted list of keys in the dictionary.

Here is a couple of example.

```
>>> squares
{1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
>>> len(squares)
5
>>> sorted(squares)
[1, 3, 5, 7, 9]
```

Python File Handling

Python File I/O

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk). Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed. Hence, in Python, a file operation takes place in the following order.

1. Open a file
2. Read or write (perform operation)
3. Close the file

Opening a File

Python has a built-in function `open()` to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
>>> f = open("test.txt")    # open file in current directory
>>> f = open("C:/Python33/README.txt")  # specifying full path
```

We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode. The default is reading in text mode. In this mode, we get strings when reading from the file. On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

Python File Modes

Mode	Description
'r'	Open a file for reading. (default)
'w'	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
'x'	Open a file for exclusive creation. If the file already exists, the operation fails.
'a'	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
't'	Open in text mode. (default)

'b'	Open in binary mode.
'+'	Open a file for updating (reading and writing)

```
f = open("test.txt")      # equivalent to 'r' or 'rt'
f = open("test.txt",'w')  # write in text mode
f = open("img.bmp",'r+b') # read and write in binary mode
```

Since the version 3.x, Python has made a clear distinction between `str` (text) and `bytes` (8-bits). Unlike other languages, the character 'a' does not imply the number 97 until it is encoded using `ASCII` (or other equivalent encodings). Hence, when working with files in text mode, it is recommended to specify the encoding type. Files are stored in bytes in the disk, we need to decode them into `str` when we read into Python. Similarly, encoding is performed while writing texts to the file.

The default encoding is platform dependent. In windows, it is 'cp1252' but 'utf-8' in Linux. Hence, we must not rely on the default encoding otherwise, our code will behave differently in different platforms. Thus, this is the preferred way to open a file for reading in text mode.

```
f = open("test.txt",mode = 'r',encoding = 'utf-8')
```

Closing a File

When we are done with operations to the file, we need to properly close it. Python has a garbage collector to clean up unreferenced objects. But we must not rely on it to close the file. Closing a file will free up the resources that were tied with the file and is done using the `close()` method.

```
f = open("test.txt",encoding = 'utf-8')
# perform file operations
f.close()
```

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file. A safer way is to use a `try...finally` block.

```
try:
   f = open("test.txt",encoding = 'utf-8')
   # perform file operations
finally:
   f.close()
```

This way, we are guaranteed that the file is properly closed even if an exception is raised, causing program flow to stop.

The best way to do this is using the `with` statement. This ensures that the file is closed when the block inside `with` is exited. We don't need to explicitly call the `close()` method. It is done internally.

```
with open("test.txt",encoding = 'utf-8') as f:
   # perform file operations
```

Writing to a File

In order to write into a file we need to open it in write 'w', append 'a' or exclusive creation 'x' mode. We need to be careful with the 'w' mode as it will overwrite into the file if it already exists. All previous data are erased.

Writing a string or sequence of bytes (for binary files) is done using `write()` method. This method returns the number of characters written to the file.

```
with open("test.txt",'w',encoding = 'utf-8') as f:
   f.write("my first file\n")
   f.write("This file\n\n")
   f.write("contains three lines\n")
```

This program will create a new file named 'test.txt' if it does not exist. If it does exist, it is overwritten. We must include the newline characters ourselves to distinguish different lines.

Reading From a File

To read the content of a file, we must open the file in reading mode. There are various methods available for this purpose. We can use the `read(size)` method to read in *size* number of data. If *size* parameter is not specified, it reads and returns up to the end of the file.

```
>>> f = open("test.txt",'r',encoding = 'utf-8')
>>> f.read(4)    # read the first 4 data
'This'

>>> f.read(4)    # read the next 4 data
' is '

>>> f.read()     # read in the rest till end of file
'my first file\nThis file\ncontains three lines\n'

>>> f.read()  # further reading returns empty sting
''
```

We can see, that `read()` method returns newline as '\n'. Once the end of file is reached, we get empty string on further reading. We can change our current file cursor (position) using the `seek()` method. Similarly, the `tell()` method returns our current position (in number of bytes).

```
>>> f.tell()    # get the current file position
56

>>> f.seek(0)   # bring file cursor to initial position
0

>>> print(f.read())  # read the entire file
This is my first file
This file
contains three lines
```

We can read a file line-by-line using a `for` loop. This is both efficient and fast.

```
>>> for line in f:
...     print(line, end = '')
...
This is my first file
This file
contains three lines
```

The lines in file itself has a newline character '\n'. Moreover,the `print()` function also appends a newline by default. Hence, we specify the *end* parameter to avoid two newlines when printing.

Alternately, we can use `readline()` method to read individual lines of a file. This method reads a file till the newline, including the newline character.

```
>>> f.readline()
'This is my first file\n'

>>> f.readline()
'This file\n'

>>> f.readline()
'contains three lines\n'

>>> f.readline()
''
```

Lastly, the `readlines()` method returns a list of remaining lines of the entire file. All these reading method return empty values when end of file (EOF) is reached.

```
>>> f.readlines()
['This is my first file\n', 'This file\n', 'contains three lines\n']
```

Python File Methods

There are various methods available with the file object. Some of them have been used in above examples. Here is the complete list of methods in text mode with a brief description.

Python File Methods

Method	Description
close()	Close an open file. It has no effect if the file is already closed.
detach()	Separate the underlying binary buffer from the `TextIOBase` and return it.
fileno()	Return an integer number (file descriptor) of the file.
flush()	Flush the write buffer of the file stream.
isatty()	Return `True` if the file stream is interactive.
read(*n*)	Read atmost *n* characters form the file. Reads till end of file if it is negative or `None`.
readable()	Returns `True` if the file stream can be read from.
readline(*n*=-1)	Read and return one line from the file. Reads in at most *n* bytes if specified.
readlines(*n*=-1)	Read and return a list of lines from the file. Reads in at most *n* bytes/characters if specified.
seek(*offset*,*from*=`SEEK_SET`)	Change the file position to *offset* bytes, in reference to *from* (start, current, end).
seekable()	Returns `True` if the file stream supports random access.
tell()	Returns the current file location.
truncate(*size*=`None`)	Resize the file stream to *size* bytes. If *size* is not specified, resize to current location.
writable()	Returns `True` if the file stream can be written to.
write(*s*)	Write string *s* to the file and return the number of characters written.
writelines(*lines*)	Write a list of *lines* to the file.

Python Directory and Files Management

If there are large number of files in Python, we can place related files in different directories to make things more manageable. A directory or folder is a collection of files and sub directories. Python has the `os` module, which provides us with many useful methods to work with directories (and files as well).

Get Current Directory

We can get the present working directory using the `getcwd()` method. This method returns the current working directory in the form of a string. We can also use the `getcwdb()` method to get it as bytes object.

```
>>> import os

>>> os.getcwd()
'C:\\Program Files\\PyScripter'

>>> os.getcwdb()
b'C:\\Program Files\\PyScripter'
```

The extra backslash implies escape sequence. The `print()` function will render this properly.

```
>>> print(os.getcwd())
C:\Program Files\PyScripter
```

Changing Directory

We can change the current working directory using the `chdir()` method. The new path that we want to change to must be supplied as a string to this method. We can use both forward slash (/) or the backward slash (\) to separate path elements. It is safer to use escape sequence when using the backward slash.

```
>>> os.chdir('C:\\Python33')

>>> print(os.getcwd())
C:\Python33
```

List Directories and Files

All files and sub directories inside a directory can be known using the `listdir()` method. This method takes in a path and returns a list of sub directories and files in that path. If no path is specified, it returns from the current working directory.

```
>>> print(os.getcwd())
C:\Python33

>>> os.listdir()
```

```
['DLLs',
'Doc',
'include',
'Lib',
'libs',
'LICENSE.txt',
'NEWS.txt',
'python.exe',
'pythonw.exe',
'README.txt',
'Scripts',
'tcl',
'Tools']

>>> os.listdir('G:\\')
['$RECYCLE.BIN',
'Movies',
'Music',
'Photos',
'Series',
'System Volume Information']
```

Making a New Directory

We can make a new directory using the `mkdir()` method. This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.

```
>>> os.mkdir('test')

>>> os.listdir()
['test']
```

Renaming a Directory or a File

The `rename()` method can rename a directory or a file. The first argument is the old name and the new name must be supplies as the second argument.

```
>>> os.listdir()
['test']

>>> os.rename('test','new_one')

>>> os.listdir()
['new_one']
```

Removing Directory or File

A file can be removed (deleted) using the `remove()` method. Similarly, the `rmdir()` method removes an empty directory.

```
>>> os.listdir()
['new_one', 'old.txt']

>>> os.remove('old.txt')
>>> os.listdir()
['new_one']

>>> os.rmdir('new_one')
>>> os.listdir()
[]
```

However, note that `rmdir()` method can only remove empty directories. In order to remove a non-empty directory we can use the `rmtree()` method inside the `shutil` module.

```
>>> os.listdir()
['test']

>>> os.rmdir('test')
Traceback (most recent call last):
...
OSError: [WinError 145] The directory is not empty: 'test'

>>> import shutil

>>> shutil.rmtree('test')
>>> os.listdir()
[]
```

Python Built-in Exceptions

When writing a program, we, more often than not, will encounter errors. Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.

```
>>> if a < 3
  File "<interactive input>", line 1
    if a < 3
           ^
SyntaxError: invalid syntax
```

We can notice here that a colon is missing in the `if` statement.

Errors can also occur at runtime and these are called exceptions. They occur, for example, when a file we try to open does not exist (`FileNotFoundError`), dividing a number by zero (`ZeroDivisionError`), module we try to import is not found (`ImportError`) etc. Whenever these type of runtime error occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

```
>>> 1 / 0
Traceback (most recent call last):
 File "<string>", line 301, in runcode
 File "<interactive input>", line 1, in <module>
ZeroDivisionError: division by zero

>>> open("imaginary.txt")
Traceback (most recent call last):
 File "<string>", line 301, in runcode
 File "<interactive input>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'imaginary.txt'
```

Illegal operations can raise exceptions. There are plenty of built-in exceptions in Python that are raised when corresponding errors occur. We can view all the built-in exceptions using the `local()` built-in functions as follows.

```
>>> locals()['__builtins__']
```

This will return us a dictionary of built-in exceptions, functions and attributes. Some of the common built-in exceptions in Python programming along with the error that cause then are tabulated below.

Python Built-in Exceptions

Exception	Cause of Error
AssertionError	Raised when `assert` statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the `input()` functions hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's `close()` method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by `next()` function to indicate that there is no further item to be returned by iterator.
SyntaxError	Raised by parser when syntax error is encountered.

IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by `sys.exit()` function.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets argument of correct type but improper value.
ZeroDivisionError	Raised when second operand of division or modulo operation is zero.

Python Exception Handling - Try, Except and Finally

When an exception occurs in Python, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash. For example, if function `A` calls function `B` which in turn calls function `C` and an exception occurs in function `C`. If it is not handled in `C`, the exception passes to `B` and then to `A`. If never handled, an error message is spit out and our program come to a sudden, unexpected halt.

Catching Exceptions in Python

In Python, exceptions can be handled using a `try` statement. A critical operation which can raise exception is placed inside the `try` clause and the code that handles exception is written in `except` clause. It is up to us, what operations we perform once we have caught the exception. Here is a simple example.

```
# import module sys to get the type of exception
import sys

while True:
   try:
       x = int(input("Enter an integer: "))
       r = 1/x
       break
   except:
       print("Oops!",sys.exc_info()[0],"occured.")
       print("Please try again.")
       print()

print("The reciprocal of",x,"is",r)
```

Here is a sample run of this program.

```
Enter an integer: a
Oops! <class 'ValueError'> occured.
Please try again.

Enter an integer: 1.3
Oops! <class 'ValueError'> occured.
Please try again.

Enter an integer: 0
Oops! <class 'ZeroDivisionError'> occured.
Please try again.

Enter an integer: 2
The reciprocal of 2 is 0.5
```

In this program, we loop until the user enters an integer that has a valid reciprocal. The portion that can cause exception is placed inside `try` block. If no exception occurs, `except` block is skipped and normal flow continues. But if any exception occurs, it is caught by the `except` block. Here, we print the name of the exception using `ex_info()` function inside `sys` module and ask the user to try again. We can see that the values 'a' and '1.3' causes `ValueError` and '0' causes `ZeroDivisionError`.

Catching Specific Exceptions in Python

In the above example, we did not mention any exception in the `except` clause. This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an `except` clause will catch. A `try` clause can have any number of `except` clause to handle them differently but only one will be executed in case an exception occurs. We can use a tuple of values to specify multiple exceptions in an `except` clause. Here is an example pseudo code.

```
try:
   # do something
   pass

except ValueError:
   # handle ValueError exception
   pass

except (TypeError, ZeroDivisionError):
   # handle multiple exceptions
   # TypeError and ZeroDivisionError
   pass

except:
   # handle all other exceptions
   pass
```

Raising Exceptions

In Python programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using the keyword `raise`. We can also optionally pass in value to the exception to clarify why that exception was raised.

```
>>> raise KeyboardInterrupt
Traceback (most recent call last):
...
KeyboardInterrupt

>>> raise MemoryError("This is an argument")
Traceback (most recent call last):
...
MemoryError: This is an argument

>>> try:
...     a = int(input("Enter a positive integer: "))
...     if a <= 0:
...         raise ValueError("That is not a positive number!")
... except ValueError as ve:
...     print(ve)
...
```

```
Enter a positive integer: -2
That is not a positive number!
```

try...finally

The `try` statement in Python can have an optional `finally` clause. This clause is executed no matter what, and is generally used to release external resources. For example, we may be connected to a remote data center through the network or working with a file or working with a Graphical User Interface (GUI). In all these circumstances, we must clean up the resource once used, whether it was successful or not. These actions (closing a file, GUI or disconnecting from network) are performed in the `finally` clause to guarantee execution.

Here is an example to illustrate this.

```
try:
   f = open("test.txt",encoding = 'utf-8')
   # perform file operations
finally:
   f.close()
```

This type of construct makes sure the file is closed even if an exception occurs.

Python User-Defined Exception

Users can define their own exception by creating a new class in Python. This exception class has to be derived, either directly or indirectly, from `Exception` class. Most of the built-in exceptions are also derived from this class.

```
>>> class CustomError(Exception):
...     pass
...

>>> raise CustomError
Traceback (most recent call last):
...
__main__.CustomError

>>> raise CustomError("An error occurred")
Traceback (most recent call last):
...
__main__.CustomError: An error occurred
```

Here, we have created a user-defined exception called `CustomError` which is derived from the `Exception` class. This new exception can be raised, like other exceptions, using the `raise` statement with an optional error message.

When we are developing a large Python program, it is a good practice to place all the user-defined exceptions that our program raises in a separate file. Many standard modules do this. They define their exceptions separately as `exceptions.py` or `errors.py` (generally but not always).

User-defined exception class can implement everything a normal class can do, but we generally make them simple and concise. Most implementations declare a custom base class and derive others exception classes from this base class. This concept is made clearer in the following example.

Example of User Defined Exception

In this example, we will illustrate how user-defined exceptions can be used in a program to raise and catch errors. This program will ask the user to enter a number until they guess a stored number correctly. To help them figure it out, hint is provided whether their guess is greater than or less than the stored number.

```
# define Python user-defined exceptions
class Error(Exception):
   """Base class for other exceptions"""
   pass

class ValueTooSmallError(Error):
   """Raised when the input value is too small"""
   pass
```

```
class ValueTooLargeError(Error):
   """Raised when the input value is too large"""
   pass

# our main program
# user guesses a number until he/she gets it right

# you need to guess this number
number = 10

while True:
   try:
       i_num = int(input("Enter a number: "))
       if i_num < number:
           raise ValueTooSmallError
       elif i_num > number:
           raise ValueTooLargeError
       break
   except ValueTooSmallError:
       print("This value is too small, try again!")
       print()
   except ValueTooLargeError:
       print("This value is too large, try again!")
       print()

print("Congratulations! You guessed it correctly.")
```

Here is a sample run of this program.

```
Enter a number: 12
This value is too large, try again!

Enter a number: 0
This value is too small, try again!

Enter a number: 8
This value is too small, try again!

Enter a number: 10
Congratulations! You guessed it correctly.
```

Here, we have defined a base class called `Error`. The other two exceptions (`ValueTooSmallError` and `ValueTooLargeError`) that are actually raised by our program are derived from this class. This is the standard way to define user-defined exceptions in Python programming, but you are not limited to this way only.

Python Object and Class

Python Namespace and Scope

If you have ever read 'The Zen of Python' (type "import this" in Python interpreter), the last line states, **Namespaces are one honking great idea -- let's do more of those!** So what are these mysterious namespaces? Let us first look at what name is.

Name (also called identifier) is simply a name given to objects. Everything in Python is an object. Name is a way to access the underlying object. For example, when we do the assignment `a = 2`, here `2` is an object stored in memory and *a* is the name we associate it with. We can get the address (in RAM) of some object through the built-in function, `id()`. Let's check it.

```
>>> a = 2
>>> id(2)
507098816
>>> id(a)
507098816
```

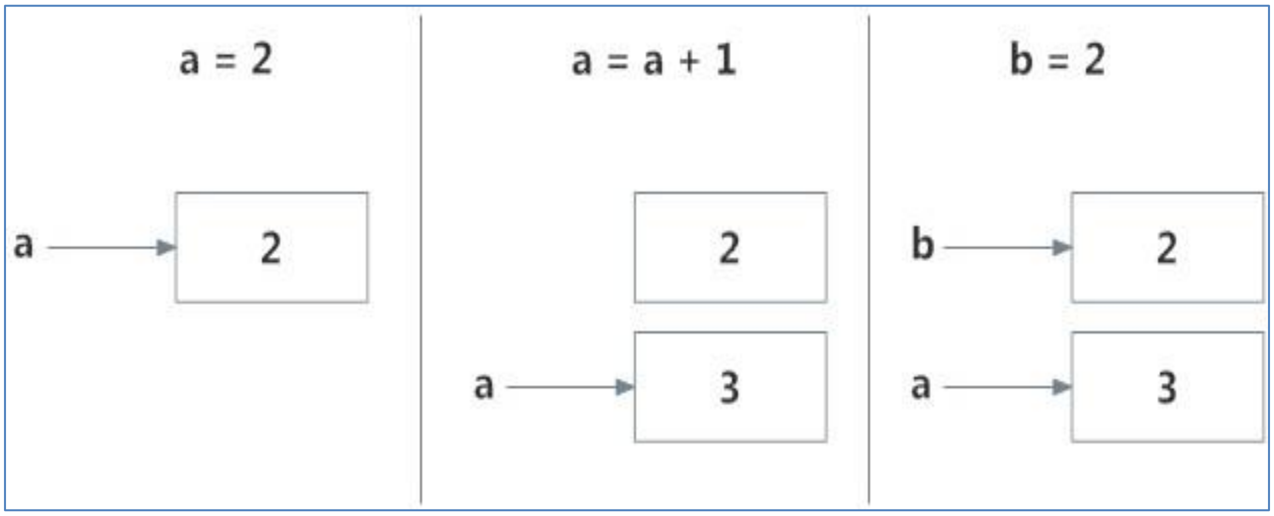
We can see that both refer to the same object. Let's make things a little more interesting.

```
>>> a = 2
>>> id(a)
507098816

>>> a = a+1
>>> id(a)
507098848
>>> id(3)
507098848

>>> b = 2
>>> id(b)
507098816
```

What is happening in the above sequence of steps? A diagram will help us explain this.



Initially, an object `2` is created and the name *a* is associated with it, when we do `a = a+1`, a new object `3` is created and now *a* associates with this object. Note that `id(a)` and `id(3)` have same values. Furthermore, when we do `b = 2`, the new name *b* gets associated with the previous object `2`. This is efficient as Python doesn't have to create a new duplicate object. This dynamic nature of name binding makes Python powerful; a name could refer to any type of object.

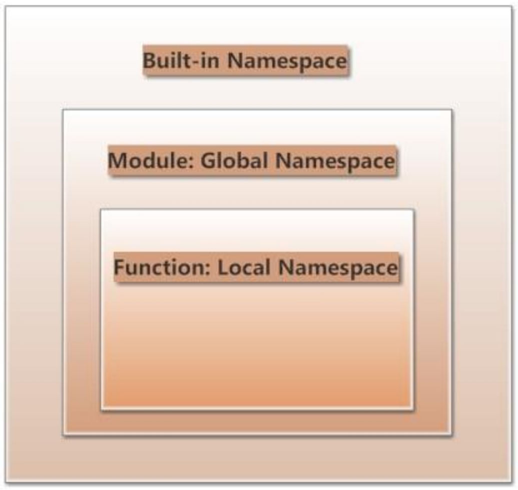
```
>>> a = 5
>>> a = 'Hello World!'
>>> a = [1,2,3]
```

All these are valid and *a* will refer to three different types of object at different instances. Functions are objects too, so a name can refer to them as well.

```
>>> def func():
...     print("Hello")
...
>>> a = func
>>> a()
Hello
```

Our same name *a* can refer to a function and we can call the function through it, pretty neat.

So now that we understand what names are, we can move on to the concept of namespaces. To simply put it, namespace is a collection of names. In Python, you can imagine a namespace as a mapping of every name, you have defined, to corresponding objects. Different namespaces can co-exist at a given time but are completely isolated. A namespace containing all the built-in names is created when we start the Python interpreter and exists as long we don't exit. This is the reason that built-in functions like `id()`, `print()` etc. are always available to us from any part of the program. Each module creates its own global namespace. Since, these different namespaces are isolated, same name that may exist in different modules do not collide. Modules can have various functions and classes. A local namespace is created when a functions is called, which has all the names defined in it. Similar, is the case with class. Following diagram may help to clarify this concept.



Python Scope

Although there are various unique namespaces defined, we may not be able to access all of them from every part of the program. The concept of scope comes into play. Scope is the portion of the program from where a namespace can be accessed directly without any prefix. At any given moment, there are at least three nested scopes.

1. Scope of the current function which has local names
2. Scope of the module which has global names
3. Outermost scope which has built-in names

When a reference is made inside a function, the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace. If there is a function inside another function, a new scope is nested inside the local scope.

Example of Scope and Namespace in Python

```
def outer_function():
    b = 20
    def inner_func():
        c = 30

a = 10
```

Here, the variable *a* is in the global namespace. Variable *b* is in the local namespace of `outer_function()` and *c* is in the nested local namespace of `inner_function()`. When we are in `inner_function()`, *c* is local to us, *b* is nonlocal and *a* is global. We can read as well as assign new values to *c* but can only read *b* and *c* from `inner_function()`. If we try to assign as a value to *b*, a new variable *b* is created in the local namespace which is different than the nonlocal *b*. Same thing happens when we assign a value to *a*.

However, if we declare *a* as global, all the reference and assignment go to the global *a*. Similarly, if we want to rebind the variable *b*, it must be declared as nonlocal. The following example will further clarify this.

```
def outer_function():
    a = 20
    def inner_function():
        a = 30
        print('a =',a)

    inner_function()
    print('a =',a)

a = 10
outer_function()
print('a =',a)
```

The output of this program is

```
a = 30
a = 20
a = 10
```

In this program, three different variables *a* are defined in separate namespaces and accessed accordingly. While in the following program,

```
def outer_function():
    global a
    a = 20
    def inner_function():
        global a
        a = 30
        print('a =',a)

    inner_function()
    print('a =',a)

a = 10
outer_function()
print('a =',a)

the output is.
a = 30
a = 30
a = 30
```

Here, all reference and assignment are to the global *a* due to the use of keyword `global`.

Python Objects and Class

Python is an object oriented programming language. Unlike procedure oriented programming, in which the main emphasis is on functions, object oriented programming stress on objects. Object is simply a collection of data (variables) and methods (functions) that act on those data.

Class is a blueprint for the object. We can think of class like a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object. As, many houses can be made from a description, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called instantiation.

Defining a Class in Python

Like function definitions begin with the keyword `def`, in Python, we define a class using the keyword `class`. The first string is called docstring and has a brief description about the class. Although not mandatory, this is recommended. Here is a simple class definition.

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    pass
```

A class creates a new local namespace where all its attributes are defines. Attributes may be data or functions. There are also special attributes in it that begins with double underscores (__). For example, __doc__ gives us the docstring of that class. As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
>>> class MyClass:
...     "This is my second class"
...     a = 10
...     def func(self):
...         print('Hello')
...
>>> MyClass.a
10

>>> MyClass.func
<function MyClass.func at 0x0000000003079BF8>

>>> MyClass.__doc__
'This is my second class'
```

Creating an Object in Python

We saw that the class object could be used to access different attributes. It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a function call.

```
>>> ob = MyClass()
```

This will create a new instance object named *ob*. We can access attributes of objects using the object name prefix. Attributes may be data or method. Method of an object are corresponding functions of that class. Any function object that is a class attribute defines a method for objects of that class. This means to say, since `MyClass.func` is a function object (attribute of class), `ob.func` will be a method object.

```
>>> ob = MyClass()
>>> MyClass.func
<function MyClass.func at 0x000000000335B0D0>
>>> ob.func
<bound method MyClass.func of <__main__.MyClass object at
0x000000000332DEF0>>

>>> ob.func()
Hello
```

You may have notices the *self* parameter in function definition inside the class. But we called the method simply as `ob.func()` without any arguments. It still worked. This is because, whenever an object calls its method, the object itself is pass as the first argument. So, `ob.func()` translates into `MyClass.func(ob)`. In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument. For these reasons, the first argument of the function in class must be the object itself. This is conventionally called *self*. It can be named otherwise but we highly recommend to follow the convention.

Now you must be familiar with class object, instance object, function object, method object and their differences.

Constructors in Python

Class functions that begins with double underscore (__) are called special functions as they have special meaning. Of one particular interest is the `__init__()` function. This special function gets called whenever a new object of that class is instantiated. This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

```
class ComplexNumber:
    def __init__(self,r = 0,i = 0):
        self.real = r
        self.imag = i

    def getData(self):
        print("{0}+{1}j".format(self.real,self.imag))
```

In the above example, we define a new class to represent complex numbers. It has two functions, `__init__()` to initialize the variables (defaults to zero) and `getData()` to display the number properly. Here, are some sample runs.

```
>>> c1 = ComplexNumber(2,3)
>>> c1.getData()
2+3j

>>> c2 = ComplexNumber(5)
>>> c2.attr = 10
>>> (c2.real, c2.imag, c2.attr)
(5, 0, 10)
>>> c1.attr
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'attr'
```

An interesting thing to note in the above step is that attributes of an object can be created on the fly. We created a new attribute *attr* for object *c2* and we read it as well. But this did not create that attribute for object *c1*.

Deleting Attributes and Objects

Any attribute of an object can be deleted anytime, using the `del` statement.

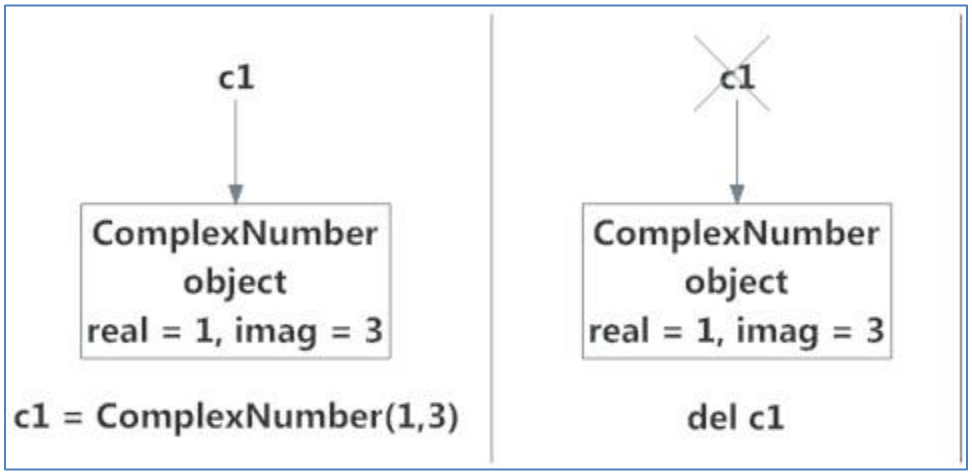
```
>>> c1 = ComplexNumber(2,3)
>>> del c1.imag
>>> c1.getData()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'imag'

>>> del ComplexNumber.getData
>>> c1.getData()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'getData'
```

We can even delete the object itself, using the `del` statement.

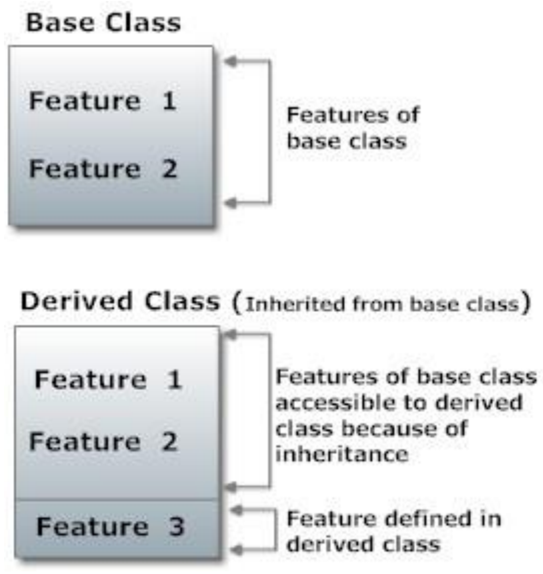
```
>>> c1 = ComplexNumber(1,3)
>>> del c1
>>> c1
Traceback (most recent call last):
...
NameError: name 'c1' is not defined
```

Actually, it is more complicated than that. When we do `c1 = ComplexNumber(1,3)`, a new instance object is created in memory and the name *c1* binds with it. On the command `del c1`, this binding is removed and the name *c1* is deleted from the corresponding namespace. The object however continues to exist in memory and if no other name is bound to it, it is later automatically destroyed. This automatic destruction of unreferenced objects in Python is also called garbage collection.


c1
ComplexNumber object
real = 1, imag = 3
c1 = ComplexNumber(1,3)
c1
ComplexNumber object
real = 1, imag = 3
del c1

Python Inheritance

Inheritance is a powerful feature in object oriented programming. It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class. Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.


Base Class
Feature 1
Feature 2
Features of base class
Derived Class (Inherited from base class)
Feature 1
Feature 2
Features of base class accessible to derived class because of inheritance
Feature 3
Feature defined in derived class

Python Inheritance Syntax

```
class DerivedClass(BaseClass):
    body_of_derived_class
```

Example of Inheritance in Python

To demonstrate the use of inheritance, let us take an example. A polygon is a closed figure with 3 or more sides. Say, we have a class called `Polygon` defined as follows.

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in 
range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
```

This class has data attributes to store the number of sides, *n* and magnitude of each side as a list, *sides*. Method `inputSides()` takes in magnitude of each side and similarly, `dispSides()` will display these properly.

A triangle is a polygon with 3 sides. So, we can created a class called `Triangle` which inherits from `Polygon`. This makes all the attributes available in class `Polygon` readily available in `Triangle`. We don't need to define them again (code re-usability). `Triangle` is defined as follows.

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

However, class `Triangle` has a new method `findArea()` to find and print the area of the triangle. Here is a sample run.

```
>>> t = Triangle()

>>> t.inputSides()
Enter side 1 : 3
Enter side 2 : 5
Enter side 3 : 4

>>> t.dispSides()
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0

>>> t.findArea()
The area of the triangle is 6.00
```

We can see that, even though we did not define methods like `inputSides()` or `dispSides()` for class `Triangle`, we were able to use them. If an attribute is not found in the class, search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

Method Overriding in Python

In the above example, notice that `__init__()` method was defined in both classes, `Triangle` as well `Polygon`. When this happens, the method in the derived class overrides that in the base class. This is to say, `__init__()` in `Triangle` gets preference over the same in `Polygon`. Generally when overriding a base method, we tend to extend the definition rather than simply replace it. The same is being done by calling the method in base class from the one in derived class (calling `Polygon.__init__()` from `__init__()` in `Triangle`). A better option would be to use the built-in function `super()`. So, `super().__init(3)` is equivalent to `Polygon.__init__(self,3)` and is preferred.

Two built-in functions `isinstance()` and `issubclass()` are used to check inheritances. Function `isinstance()` returns `True` if the object is an instance of the class or other classes derived from it. Each and every class in Python inherits from the base class `object`.

```
>>> isinstance(t,Triangle)
True

>>> isinstance(t,Polygon)
True

>>> isinstance(t,int)
False

>>> isinstance(t,object)
True
```

Similarly, `issubclass()` is used to check for class inheritance.

```
>>> issubclass(Polygon,Triangle)
False

>>> issubclass(Triangle,Polygon)
True

>>> issubclass(bool,int)
True
```

Python Multiple Inheritance

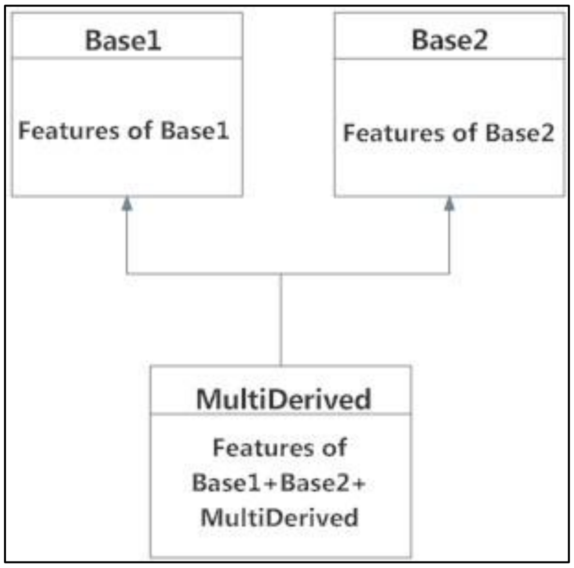
Multiple inheritance is possible in Python unlike other programming languages. A class can be derived from more than one base classes. The syntax for multiple inheritance is similar to single inheritance.

Python Multiple Inheritance Example

```
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass
```



The class `MultiDerived` inherits from both `Base1` and `Base2`.

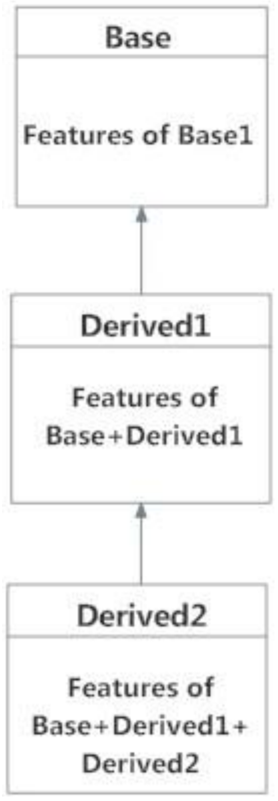
Multilevel Inheritance in Python

On the other hand, we can inherit form a derived class. This is also called multilevel inheritance. Multilevel inheritance can be of any depth in Python. An example with corresponding visualization is given below.

```
class Base:
    pass

class Derived1(Base):
    pass

class Derived2(Derived1):
    pass
```


Base
Features of Base1
Derived1
Features of Base+Derived1
Derived2
Features of Base+Derived1+ Derived2

Method Resolution Order in Python

Every class in Python is derived from the class `object`. It is the most base type in Python. So technically, all other class, either built-in or user-defines, are derived classes and all objects are instances of `object` class.

```
>>> issubclass(list,object)
True
>>> isinstance(5.5,object)
True
>>> isinstance("Hello",object)
True
```

In the multiple inheritance scenario, any specified attribute is searched first in the current class. If not found, the search continues into parent classes in depth-first, left-right fashion without searching same class twice. So, in the above example of `MultiDerived` class the search order is [`MultiDerived`, `Base1`, `Base2`, `object`]. This order is also called linearization of `MultiDerived` class and the set of rules used to find this order is called Method Resolution Order (MRO). MRO must prevent local precedence ordering and also provide monotonicity. It ensures that a class always appears before its parents and in case of multiple parents, the order is same as tuple of base classes.

MRO of a class can be viewed as the `__mro__` attribute or `mro()` method. The former returns a tuple while latter returns a list.

```
>>> MultiDerived.__mro__
(<class '__main__.MultiDerived'>,
 <class '__main__.Base1'>,
 <class '__main__.Base2'>,
 <class 'object'>)

>>> MultiDerived.mro()
[<class '__main__.MultiDerived'>,
 <class '__main__.Base1'>,
 <class '__main__.Base2'>,
 <class 'object'>]
```

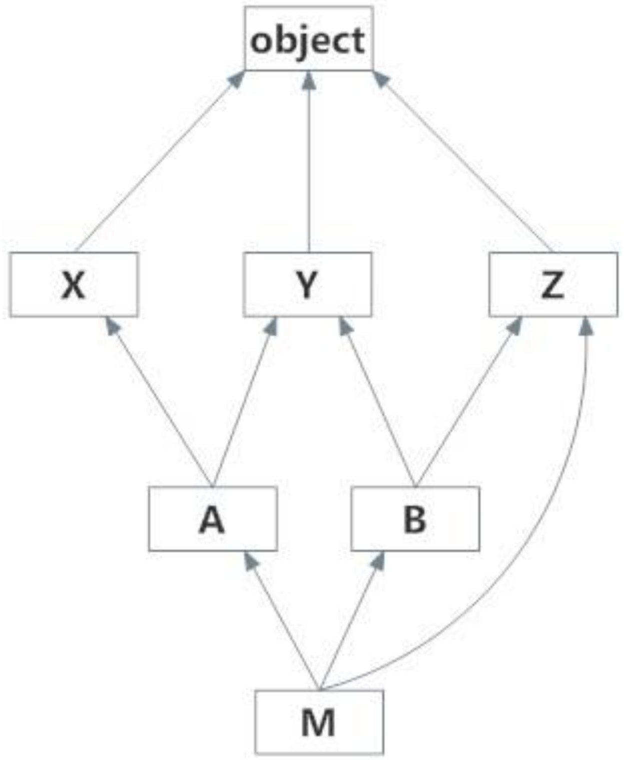
Here is a little more complex multiple inheritance example and its visualization along with the MRO.

```
class X: pass
class Y: pass
class Z: pass

class A(X,Y): pass
class B(Y,Z): pass

class M(B,A,Z): pass

print(M.mro())
```



Output

```
[<class '__main__.M'>, <class '__main__.B'>, <class '__main__.A'>, <class
'__main__.X'>, <class '__main__.Y'>, <class '__main__.Z'>, <class 'object'>]
```

Python Operator Overloading

Python operators work for built-in classes. But same operator behaves differently with different types. For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings. This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading.

So what happens when we use them with objects of a user-defined class? Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

```
class Point:
    def __init__(self,x = 0,y = 0):
        self.x = x
        self.y = y
```

Now when we try to add two points that we create as follows.

```
>>> p1 = Point(2,3)
>>> p2 = Point(-1,2)
>>> p1 + p2
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

Whoa! That's a lot of complains. `TypeError` was raised since Python didn't know how to add two `Point` objects together. However, the good news is that we can teach this to Python through operator overloading. But first, let's get a notion about special functions.

Special Functions in Python

Class functions that begins with double underscore (__) are called special functions in Python. This is because, well, they are not ordinary. The `__init__()` function we defined above, is one of them. It gets called every time we create a new object of that class. There are a ton of special functions in Python.

Using special functions, we can make our class compatible with built-in functions.

```
>>> p1 = Point(2,3)
>>> print(p1)
<__main__.Point object at 0x00000000031F8CC0>
```

That did not print well. But if we define `__str__()` method in our class, we can control how it gets printed. So, let's add this to our class.

```
class Point:
    # previous definitions...

    def __str__(self):
        return "({0},{1})".format(self.x,self.y)
```

Now let's try the `print()` function again.

```
>>> print(p1)
(2,3)
```

That's better. Turns out, that this same method is invoked when we use the built-in function `str()` or `format()`.

```
>>> str(p1)
'(2,3)'

>>> format(p1)
'(2,3)'
```

So, when you do `str(p1)` or `format(p1)`, Python is internally doing `p1.__str__()`. Hence the name, special functions.

Overloading the + Operator

To overload the `+` sign, we will need to implement `__add__()` function in the class. With great power comes great responsibility. We can do whatever we like, inside this function. But it is sensible to return a `Point` object of the coordinate sum.

```
class Point:
    # previous definitions...

    def __add__(self,other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x,y)
```

Now let's try that addition again.

```
>>> p1 = Point(2,3)
>>> p2 = Point(-1,2)
>>> print(p1 + p2)
(1,5)
```

What actually happens is that, when you do `p1 + p2`, Python will call `p1.__add__(p2)` which in turn is `Point.__add__(p1,p2)`. Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator Overloading Special Functions in Python

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()

Overloading Comparison Operators in Python

Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well. Suppose, we wanted to implement the less than symbol < symbol in our `Point` class. Let us compare the magnitude of these points from the origin and return the result for this purpose. It can be implemented as follows.

```
class Point:
    # previous definitions...

    def __lt__(self,other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag
```

Some sample runs.

```
>>> Point(1,1) < Point(-2,-3)
True

>>> Point(1,1) < Point(0.5,-0.2)
False

>>> Point(1,1) < Point(1,1)
False
```

Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

Comparision Operator Overloading in Python

Operator	**Expression**	**Internally**
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

Additional Tutorials

- Python Iterator
- Python Generator
- Python Closure
- Python Decorators
- Python Property
- Python Examples

Python Iterators

Iterators are everywhere in Python. They are elegantly implemented within `for` loops, comprehensions, generators etc. but hidden in plain sight. Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, Python **iterator object** must implement two special methods, `__iter__()` and `__next__()`, collectively called the **iterator protocol**.

An object is called **iterable** if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables. The `iter()` function (which in turn calls the `__iter__()` method) returns an iterator from them.

Iterating Through an Iterator in Python

We use the `next()` function to manually iterate through all the items of an iterator. When we reach the end and there is no more data to be returned, it will raise `StopIteration`. Following is an example.

```
>>> # define a list
>>> my_list = [4, 7, 0, 3]

>>> # get an iterator using iter() on that list
>>> my_iter = iter(my_list)
>>> my_iter
<list_iterator object at 0x00000000031AD9B0>

>>> # iterate through it using next()
>>> next(my_iter)
4
>>> next(my_iter)
7

>>> # next(obj) is same as obj.__next__()
>>> my_iter.__next__()
0
>>> my_iter.__next__()
3

>>> next(my_iter)
Traceback (most recent call last):
...
StopIteration
>>> # no more items left
```

A more elegant way of automatically iterating is by using the `for` loop. Using this, we can iterate over any object that can return an iterator, for example list, string, file etc.

```
>>> for element in my_list:
...     print(element)
...
```

```
4
7
0
3
```

How for Loop Actually Works

As we see in the above example, the `for` loop was able to iterate automatically through the list. In fact the `for` loop can iterate over any iterable. Let's take a closer look at how the `for` loop is actually implemented in Python.

```
for element in iterable:
    # do something with element
```

Is actually implemented as.

```
# create an iterator object from that iterable
iter_obj = iter(iterable)

# infinite loop
while True:
    try:
        # get the next item
        element = next(iter_obj)
        # do something with element
    except StopIteration:
        # if StopIteration is raised, break from loop
        break
```

So internally, the `for` loop creates an iterator object, `iter_obj` by calling `iter()` on the iterable. Ironically, this `for` loop is actually an infinite `while` loop. Inside the loop, it calls `next()` to get the next element and executes the body of the `for` loop with this value. After all the items exhaust, `StopIteration` is raised which is internally caught and the loop ends. Note that any other kind of exception will pass through.

Building Your Own Iterator in Python

Building an iterator from scratch is easy in Python. We just have to implement the methods `__iter__()` and `__next__()`. The `__iter__()` method returns the iterator object itself. If required, some initialization can be performed. The `__next__()` method must return the next item in the sequence. On reaching the end, and in subsequent calls, it must raise `StopIteration`.

Here, we show an example that will give us next power of 2 in each iteration. Power exponent starts from zero up to a user set number.

```
class PowTwo:
    """Class to implement an iterator
    of powers of two"""

    def __init__(self, max = 0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration
```

Now we can create an iterator and iterate through it as follows.

```
>>> a = PowTwo(4)
>>> i = iter(a)
>>> next(i)
1
>>> next(i)
2
>>> next(i)
4
>>> next(i)
8
>>> next(i)
16
>>> next(i)
Traceback (most recent call last):
...
StopIteration
```

We can also use a `for` loop to iterate over our iterator class.

```
>>> for i in PowTwo(5):
...     print(i)
...
1
2
4
8
16
32
```

Python Infinite Iterators

It is not necessary that the item in an iterator object has to exhaust. There can be infinite iterators (which never ends). We must be careful when handling such iterator.

Here is a simple example to demonstrate infinite iterators. The built-in function `iter()` can be called with two arguments where the first argument must be a callable object (function) and second is the sentinel. The iterator calls this function until the returned value is equal to the sentinel.

```
>>> int()
0

>>> inf = iter(int,1)
>>> next(inf)
0
>>> next(inf)
0
```

We can see that the `int()` function always returns 0. So passing it as `iter(int,1)` will return an iterator that calls `int()` until the returned value equals 1. This never happens and we get an infinite iterator.

We can also built our own infinite iterators. The following iterator will, theoretically, return all the odd numbers.

```
class InfIter:
    """Infinite iterator to return all
        odd numbers"""

    def __iter__(self):
        self.num = 1
        return self

    def __next__(self):
        num = self.num
        self.num += 2
        return num
```

A sample run would be as follows.

```
>>> a = iter(InfIter())
>>> next(a)
1
>>> next(a)
3
>>> next(a)
5
>>> next(a)
7
```

And so on...

Be careful to include a terminating condition, when iterating over these type of infinite iterators. The advantage of using iterators is that they save resources. Like shown above, we could get all the odd numbers without storing the entire number system in memory. We can have infinite items (theoretically) in finite memory. Iterator also makes a code look cool.

Python Generators

There was a lot of overhead in building an iterator in Python; we had to implement a class with `__iter__()` and `__next__()` method, keep track of internal states, raise `StopIteration` when there was no values to be returned etc. This is both lengthy and counter intuitive. Generator comes into rescue in such situations.

Python generators are a simple way of creating iterators. All the overhead that we mentioned above are automatically handled by generators in Python. Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

Creating a Generator in Python

It is fairly simple to create a generator in Python. It is as easy as defining a normal function with `yield` statement instead of a `return` statement. If a function contains at least one `yield` statement (it may contain other `yield` or `return` statements), it becomes a generator function. Both `yield` and `return` will return some value from a function. The difference is that, while a `return` statement terminates a function entirely, `yield` statement pauses the function saving all its states and later continues from there on successive calls. Here is how a generator function differs from a normal function.

- Generator function contains one or more `yield` statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like `__iter__()` and `__next__()` are implemented automatically. So we can iterate through the items using `next()`.
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and theirs states are remembered between successive calls.
- Finally, when the function terminates, `StopIteration` is raised automatically on further calls.

Here is an example to illustrate all of the points stated above. We have a generator function named `my_gen()` with several `yield` statements.

```
def my_gen():
    """a simple generator function"""
    n = 1
    print("This is printed first")
    # Generator function contains yield statements
    yield n

    n += 1
    print("This is printed second")
    yield n

    n += 1
    print("This is printed at last")
    yield n
```

An interactive run in the interpreter is given below.

```
>>> # It returns an object but does not start execution immediately.
>>> a = my_gen()

>>> # We can iterate through the items using next().
>>> next(a)
This is printed first
1
>>> # Once the function yields, the function is paused and the control is
transferred to the caller.

>>> # Local variables and theirs states are remembered between successive
calls.
>>> next(a)
This is printed second
2
>>> next(a)
This is printed at last
3

>>> # Finally, when the function terminates, StopIteration is raised
automatically on further calls.
>>> next(a)
Traceback (most recent call last):
...
StopIteration
>>> next(a)
Traceback (most recent call last):
...
StopIteration
```

One interesting thing to note in the above example is that, the value of variable *n* is remembered between each call. Unlike normal functions, the local variables are not destroyed when the function yields. Furthermore, the generator object can be iterated only once. To restart the process we need to create another generator object using something like `a = my_gen()`.

One final thing to note is that we can use generators with `for` loops directly. This is because, a `for` loop takes an iterator and iterates over it using `next()` function. It automatically ends when `StopIteration` is raised.

```
>>> for item in my_gen():
...     print(item)
...
This is printed first
1
This is printed second
2
This is printed at last
3
```

Python Generators with a Loop

The above example is of less use and we studied it just to get an idea of what was happening in the background. Normally, generator functions are implemented with a loop having a suitable terminating condition. Let's take an example of a generator that reverses a string.

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1,-1,-1):
        yield my_str[i]
```

In this example, we use `range()` function to get the index in reverse order. Here is a call to this function.

```
>>> for char in rev_str("hello"):
...     print(char)
...
o
l
l
e
h
```

It turns out that this generator function not only works with string, but also with other kind of iterables like list, tuple etc.

Python Generator Expression

Simple generators can be easily created on the fly using generator expressions. It makes building generators easy. Same as lambda function creates an anonymous function, generator expression creates an anonymous generator function. The syntax for generator expression is similar to that of a list comprehension in Python. But the square brackets are replaced with round parentheses.

The major difference between a list comprehension and a generator expression is that while list comprehension produces the entire list, generator expression produces one item at a time. They are kind of lazy, producing items only when asked for. For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

```
>>> my_list = [1, 3, 6, 10]

>>> # square each term using list comprehension
>>> [x**2 for x in my_list]
[1, 9, 36, 100]

>>> # same thing can be done using generator expression
>>> (x**2 for x in my_list)
<generator object <genexpr> at 0x0000000002EBDAF8>
```

We can see above that the generator expression did not produce the required result immediately. Instead, it returned a generator object with produces items on demand.

```
>>> a = (x**2 for x in my_list)
>>> next(a)
1
>>> next(a)
9
>>> next(a)
36
>>> next(a)
100
>>> next(a)
Traceback (most recent call last):
...
StopIteration
```

Generator expression can be used inside functions. When used in such a way, the round parentheses can be dropped.

```
>>> sum(x**2 for x in my_list)
146

>>> max(x**2 for x in my_list)
100
```

Why generators are used in Python?

There are several reasons which make generators an attractive implementation to go for.

1. **Easy to Implement**

   Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of 2's using iterator class.

```
class PowTwo:
    def __init__(self, max = 0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n > self.max:
            raise StopIteration

        result = 2 ** self.n
        self.n += 1
        return result
```

   This was lengthy. Now let's do the same using a generator function.

```
def PowTwoGen(max = 0):
    n = 0
    while n < max:
        yield 2 ** n
        n += 1
```

Since, generators keep track of details automatically, it was concise and much cleaner in implementation.

2. **Memory Efficient**

A normal function to return a sequence will make the entire sequence in memory before returning the result. This is an overkill if the number of items in the sequence is very large. Generator implementation of such sequence is memory friendly and is preferred since it only produces one item at a time.

3. **Represent Infinite Stream**

Generators are excellent medium to represent an infinite stream of data. Infinite streams cannot be stored in memory and since generators produce only one item at a time, it can represent infinite stream of data. The following example can generate all the even numbers (at least in theory).

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2
```

4. **Pipelining Generators**

Generators can be used to pipeline a series of operations. This is best illustrated using an example.

Suppose we have a log file from a famous fast food chain. The log file has a column (4th column) that keeps track of the number of pizza sold every hour and we want to sum it to find the total pizzas sold in 5 years. Assume everything is in string and numbers that are not available are marked as 'N/A'. A generator implementation of this could be as follows.

```
with open('sells.log') as file:
    pizza_col = (line[3] for line in file)
    per_hour = (int(x) for x in pizza_col if x != 'N/A')
    print("Total pizzas sold = ",sum(per_hour))
```

This pipelining is efficient and easy to read (and yes, a lot cooler!).

Python Closures

A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope. In Python, these non-local variables are read only by default and we must declare them explicitly as non-local (using `nonlocal` keyword) in order to modify them. Following is an example of a nested function accessing a non-local variable.

```
def print_msg(msg):
    """This is the outer enclosing function"""

    def printer():
        """This is the nested function"""
        print(msg)

    printer()
```

We execute the function as follows.

```
>>> print_msg("Hello")
Hello
```

We can see that the nested function `printer()` was able to access the non-local variable *msg* of the enclosing function. In the example above, what would happen if the last line of the function `print_msg()` returned the `printer()` function instead of calling it? This means the function was defined as follows.

```
def print_msg(msg):
    """This is the outer enclosing function"""

    def printer():
        """This is the nested function"""
        print(msg)

    return printer  # this got changed
```

Now let's try calling this function.

```
>>> another = print_msg("Hello")
>>> another()
Hello
```

That's unusual. The `print_msg()` function was called with the string `"Hello"` and the returned function was bound to the name *another*. On calling `another()`, the message was still remembered although we had already finished executing the `print_msg()` function. This technique by which some data (`"Hello"`) gets attached to the code is called closure in Python.

This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

```
>>> del print_msg
>>> another()
Hello
>>> print_msg("Hello")
Traceback (most recent call last):
...
NameError: name 'print_msg' is not defined
```

When Do We Have a Closure?

As seen from the above example, we have a closure in Python when a nested function references a value in its enclosing scope. The criteria that must be met to create closure in Python are summarized in the following points.

- We must have a nested function (function inside a function).
- The nested function must refer to a value defined in the enclosing function.
- The enclosing function must return the nested function.

When To Use Closures?

So what are closures good for? Closures can avoid the use of global values and provides some form of data hiding. It can also provide an object oriented solution to the problem. When there are few methods (one method in most cases) to be implemented in a class, closures can provide an alternate and more elegant solutions. But when the number of attributes and methods get larger, better implement a class.

Here is a simple example where a closure might be more preferable than defining a class and making objects. But the preference is all yours.

```
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier
```

Here is how we can use it.

```
>>> times3 = make_multiplier_of(3)
>>> times5 = make_multiplier_of(5)

>>> times3(9)
27
>>> times5(3)
15
>>> times5(times3(2))
30
```

Decorators in Python make an extensive use of closures as well.

On a concluding note, it is good to point out that the values that get enclosed in the closure function can be found out. All function objects have a `__closure__` attribute that returns a tuple of cell objects if it is a closure function. Referring to the example above, we know `times3` and `times5` are closure functions.

```
>>> make_multiplier_of.__closure__
>>> times3.__closure__
(<cell at 0x0000000002D155B8: int object at 0x000000001E39B6E0>,)
```

The cell object has the attribute cell_contents which stores the closed value.

```
>>> times3.__closure__[0].cell_contents
3
>>> times5.__closure__[0].cell_contents
5
```

Python Decorators

Python has an interesting feature called **decorators** to add functionality to an existing code. This is also called **metaprogramming** as a part of the program tries to modify another part of the program at compile time.

Preliminaries

In order to understand about decorators, we must first know a few basic things in Python. We must be comfortable with the fact that, everything in Python (Yes! Even classes), are objects. Names that we define are simply identifiers bound to these objects. Functions are no exceptions, they are objects too (with attributes). Various different names can be bound to the same function object. Here is an example.

```
>>> def first(msg):
...     print(msg)
...

>>> first("Hello")
Hello

>>> second = first
>>> second("Hello")
Hello
```

Here, the names `first` and `second` refer to the same function object.

Now things start getting weirder. Functions can be passed as arguments to another function. If you have used functions like `map`, `filter` and `reduce` in Python, then you already know about this. Such function that take other functions as arguments are also called **higher order functions**. Here is an example of such a function.

```
def inc(x):
    """Function to increase value by 1"""
    return x + 1

def dec(x):
    """Function to decrease value by 1"""
    return x - 1

def operate(func, x):
    """A higer order function to increase or decrease"""
    result = func(x)
    return result
```

We invoke the function as follows.

```
>>> operate(inc,3)
4
>>> operate(dec,3)
```

```
2
```

Furthermore, a function can return another function.

```
>>> def is_called():
...     def is_returned():
...         print("Hello")
...     return is_returned
...

>>> new = is_called()
>>> new()
Hello
```

Here, `is_returned()` is a nested function which is defined and returned, each time we call `is_called()`.

Finally, we must know about closures in Python.

Back To Decorators

Functions and methods are called **callable** as they can be called. In fact, any object which implements the special method `__call__()` is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable. Basically, a decorator takes in a function, adds some functionality and returns it.

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner

def ordinary():
    print("I am ordinary")
>>> ordinary()
I am ordinary

>>> # let's decorate this ordinary function
>>> pretty = make_pretty(ordinary)
>>> pretty()
I got decorated
I am ordinary
```

In the example shown above, `make_pretty()` is a decorator. In the assignment step.

```
pretty = make_pretty(ordinary)
```

The function `ordinary()` got decorated and the returned function was given the name `pretty`. We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

Generally, we decorate a function and reassign it as,

```
ordinary = make_pretty(ordinary).
```

This is a common construct and for this reason, Python has a syntax to simplify this. We can use the `@` symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

```
@make_pretty
def ordinary():
    print("I am ordinary")
```

is equivalent to

```
def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

This is just a syntactic sugar to implement decorators.

Decorating Functions with Parameters

The above decorator was simple and it only worked with functions that did not have any parameters. What if we had functions that took in parameters like below?

```
def divide(a, b):
    return a/b
```

This function has two parameters, *a* and *b*. We know, it will give error if we pass in *b* as 0.

```
>>> divide(2,5)
0.4
>>> divide(2,0)
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

Now let's make a decorator to check for this case that will cause the error.

```
def smart_divide(func):
    def inner(a,b):
        print("I am going to divide",a,"and",b)
        if b == 0:
            print("Whoops! cannot divide")
            return

        return func(a,b)
    return inner

@smart_divide
def divide(a,b):
    return a/b
```

This new implementation will return `None` if the error condition arises.

```
>>> divide(2,5)
I am going to divide 2 and 5
0.4

>>> divide(2,0)
I am going to divide 2 and 0
Whoops! cannot divide
```

In this manner we can decorate functions that take parameters. A keen observer will notice that parameters of the nested `inner()` function inside the decorator is same as the parameters of functions it decorates. Taking this into account, now we can make general decorators that work with any number of parameter. In Python, this magic is done as `function(*args, **kwargs)`. In this way, `args` will be the tuple of positional arguments and `kwargs` will be the dictionary of keyword arguments. An example of such decorator will be.

```
def works_for_all(func):
    def inner(*args, **kwargs):
        print("I can decorate any function")
        return func(*args, **kwargs)
    return inner
```

Chaining Decorators in Python

Multiple decorators can be chained in Python. This is to say, a function can be decorated multiple times with different (or same) decorators. We simply place the decorators above the desired function.

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner

def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    return inner

@star
@percent
def printer(msg):
    print(msg)
```

This will give the output.

```
>>> printer("Hello")

******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
```

The above syntax of,

```
@star
@percent
def printer(msg):
    print(msg)
```

is equivalent to

```
def printer(msg):
    print(msg)
printer = star(percent(printer))
```

The order in which we chain decorators matter. If we had reversed the order as,

```
@percent
@star
def printer(msg):
    print(msg)
```

The execution would take place as,

```
>>> printer("Hello")

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
Hello
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Python @property

Python has a great concept called property, which makes the life of an object oriented programmer much simpler. Before defining and going into details of what a property in Python is, let us first build an intuition on why it would be needed in the first place.

An Example To Begin With

Let us assume that one day you decide to make a class that could store the temperature in degree Celsius. It would also implement a method to convert the temperature into degree Fahrenheit. One way of doing this is as follows.

```
class Celsius:
    def __init__(self, temperature = 0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32
```

We could make objects out of this class and manipulate the attribute `temperature`, as we wished.

```
>>> # create new object
>>> man = Celsius()

>>> # set temperature
>>> man.temperature = 37

>>> # get temperature
>>> man.temperature
37

>>> # get degrees Fahrenheit
>>> man.to_fahrenheit()
98.60000000000001
```

The extra decimal places when converting into Fahrenheit is due to the floating point arithmetic error (try 1.1 + 2.2 in the Python interpreter). Whenever we assign or retrieve any object attribute like `temperature`, as show above, Python searches it in the object's `__dict__` dictionary.

```
>>> man.__dict__
{'temperature': 37}
```

Therefore, `man.temperature` internally becomes `man.__dict__['temperature']`.

Now, let's further assume that our class got popular among clients and they started using it in their programs. They did all kinds of assignments to the object. One faithful day, a trusted client came to us and suggested that temperatures cannot go below -273 degree Celsius (students of thermodynamics might argue that it's actually -273.15), also called the absolute zero. He further asked us to implement this value constraint. Being a company that strive for customer satisfaction, we happily heeded the suggestion and released version 1.01, an upgrade of our existing class.

Using Getters and Setters

An obvious solution to the above constraint will be to hide the attribute `temperature` (make it private) and define new getter and setter interfaces to manipulate it. This can be done as follows.

```
class Celsius:
    def __init__(self, temperature = 0):
        self.set_temperature(temperature)

    def to_fahrenheit(self):
        return (self.get_temperature() * 1.8) + 32

    # new update
    def get_temperature(self):
        return self._temperature

    def set_temperature(self, value):
        if value < -273:
            raise ValueError("Temperature below -273 is not possible")
        self._temperature = value
```

We can see above that new methods `get_temperature()` and `set_temperature()` were defined and furthermore, `temperature` was replaced with `_temperature`. An underscore (_) at the beginning is used to denote private variables in Python.

```
>>> c = Celsius(-277)
Traceback (most recent call last):
...
ValueError: Temperature below -273 is not possible

>>> c = Celsius(37)
>>> c.get_temperature()
37
>>> c.set_temperature(10)

>>> c.set_temperature(-300)
Traceback (most recent call last):
...
ValueError: Temperature below -273 is not possible
```

This update successfully implemented the new restriction. We are no longer allowed to set temperature below -273.
Please note that private variables don't exist in Python. There are simply norms to be followed. The language itself don't apply any restrictions.

```
>>> c._temperature = -300
>>> c.get_temperature()
-300
```

But this is not of great concern. The big problem with the above update is that, all the clients who implemented our previous class in their program have to modify their code from `obj.temperature` to `obj.get_temperature()` and all assignments like `obj.temperature = val` to `obj.set_temperature(val)`. This refactoring can cause headaches to the clients with hundreds of thousands of lines of codes.

All in all, our new update was not backward compatible. This is where property comes to rescue.

The Power of Property

The pythonic way to deal with the above problem is to use property. Here is how we could have achieved it.

```
class Celsius:
    def __init__(self, temperature = 0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    def get_temperature(self):
        print("Getting value")
        return self._temperature

    def set_temperature(self, value):
        if value < -273:
            raise ValueError("Temperature below -273 is not possible")
        print("Setting value")
        self._temperature = value

    temperature = property(get_temperature,set_temperature)
```

We added a `print()` function inside `get_temperature()` and `set_temperature()` to clearly observe that they are being executed. The last line of the code, makes a property object `temperature`. Simply put, property attaches some code (`get_temperature` and `set_temperature`) to the member attribute accesses (`temperature`). Any code that retrieves the value of `temperature` will automatically call `get_temperature()` instead of a dictionary (__dict__) look-up. Similarly, any code that assigns a value to `temperature` will automatically call `set_temperature()`. This is one cool feature in Python. Let's see it in action.

```
>>> c = Celsius()
Setting value
```

We can see above that `set_temperature()` was called even when we created an object. Can you guess why? The reason is that when an object is created, `__init__()` method gets called. This method has the line `self.temperature = temperature`. This assignment automatically called `set_temperature()`.

```
>>> c.temperature
Getting value
0
```

Similarly, any access like `c.temperature` automatically calls `get_temperature()`. This is what property does. Here are a few more examples.

```
>>> c.temperature = 37
Setting value

>>> c.to_fahrenheit()
Getting value
98.60000000000001
```

By using property, we can see that, we modified our class and implemented the value constraint without any change required to the client code. Thus our implementation was backward compatible and everybody is happy.

Finally note that, the actual temperature value is stored in the private variable `_temperature`. The attribute `temperature` is a property object which provides interface to this private variable.

Digging Deeper into Property

In Python, `property()` is a built-in function that creates and returns a property object. The signature of this function is

```
property(fget=None, fset=None, fdel=None, doc=None)
```

where, `fget` is function to get value of the attribute, `fset` is function to set value of the attribute, `fdel` is function to delete the attribute and `doc` is a string (like a comment). As seen from the implementation, these function arguments are optional. So, a property object can simply be created as follows.

```
>>> property()
<property object at 0x0000000003239B38>
```

A property object has three methods, `getter()`, `setter()`, and `delete()` to specify `fget`, `fset` and `fdel` at a later point. This means, the line

```
temperature = property(get_temperature,set_temperature)
```

could have been broken down as

```
# make empty property
temperature = property()
# assign fget
temperature = temperature.getter(get_temperature)
# assign fset
temperature = temperature.setter(set_temperature)
```

These two pieces of codes are equivalent.

Programmers familiar with decorators in Python can recognize that the above construct can be implemented as decorators. We can further go on and not define names `get_temperature` and `set_temperature` as they are unnecessary and pollute the class namespace. For this, we reuse the name `temperature` while defining our getter and setter functions. This is how it can be done.

```
class Celsius:
    def __init__(self, temperature = 0):
        self._temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    @property
    def temperature(self):
        print("Getting value")
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        if value < -273:
            raise ValueError("Temperature below -273 is not possible")
        print("Setting value")
        self._temperature = value
```

The above implementation is both, simple and recommended way to make properties. You will most likely encounter these types of constructs when looking for property in Python.

Python Programming Examples

Python Program to Multiply Two Matrices
Python Program to Check Whether a String is Palindrome or Not
Python Program to Remove Punctuations From a String
Python Program to Sort Words in Alphabetic Order
Python Program to Illustrate Different Set Operations
Python Program to Count the Number of Each Vowel
Python Program to Merge Mails
Python Program to Find the Size (Resolution) of Image
Python Program to Find Hash of File

Python Program to Print Hello world!

To understand this example, you should have knowledge of following Python programming topics:

- Getting Started in Python
- Python Input, Output and Import

Source Code

```
# This program prints Hello, world!

print('Hello, world!')
```

Output

```
Hello, world!
```

In this program, we have used the built-in `print()` function to print the string `Hello, world!` on our screen. String is a sequence of characters. In Python, strings are enclosed inside single quotes, double quotes or triple quotes (''', """).

Python Program to Add Two Numbers

- Python Input, Output and Import
- Python Variables and Datatypes
- Python Operators

Source Code

```
# This program adds two numbers provided by the user

# Store input numbers
num1 = input('Enter first number: ')
```

```
num2 = input('Enter second number: ')

# Add two numbers
sum = float(num1) + float(num2)

# Display the sum
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

Output

```
Enter first number: 1.5
Enter second number: 6.3
The sum of 1.5 and 6.3 is 7.8
```

Explanation

In this program, we asked user to enter two numbers and this program displays the sum of tow numbers entered by user. We use the built-in function `input()` to take the input. `input()` returns a string, so we convert it into number using the `float()` function.

We add the two numbers using the + arithmetic operator. Changing this operator, we can subtract (-), multiply (*), divide (/), floor divide (//) or find the remainder (%) of two numbers. Find out more about arithmetic operators and input in Python.

Alternative to this, we can perform this addition in a single statement without using any variables as follows.

```
print('The sum is %.1f' %(float(input('Enter first number: '))+float(input('Enter second number: '))))
```

Although this program uses no variable (memory efficient), it is not quite readable. Some people will have difficulty understanding it. It is better to write clear codes. So, there is always a compromise between clarity and efficiency. We need to strike a balance.

Python Program to Find the Square Root

- Python Input, Output and Import
- Python Variables and Datatypes
- Python Operators

Source Code

```
# Python Program to calculate the square root

num = float(input('Enter a number: '))
num_sqrt = num ** 0.5
print('The square root of %0.3f is %0.3f'%(num ,num_sqrt))
```

Output

```
Enter a number: 8
The square root of 8.000 is 2.828
```

In this program, we ask the user for a number and find the square root using the `**` exponent operator. This program works for all positive real numbers. But for negative or complex numbers, it can be done as follows.

```
# Find square root of real or complex numbers
# Import the complex math module
import cmath

num = eval(input('Enter a number: '))
num_sqrt = cmath.sqrt(num)
print('The square root of {0} is {1:0.3f}+{2:0.3f}j'.format(num
,num_sqrt.real,num_sqrt.imag))
```

Output

```
Enter a number: 1+2j
The square root of (1+2j) is 1.272+0.786j
```

In this program, we use the `sqrt()` function in the `cmath` (complex math) module. Notice that we have used the `eval()` function instead of `float()` to convert complex number as well. Also notice the way in which the output is formatted.

Python Program to Calculate the Area of a Triangle

- Python Input, Output and Import
- Python Variables and Datatypes
- Python Operators

Source Code

```
# Python Program to find the area of triangle
# Three sides of the triangle a, b and c are provided by the user

a = float(input('Enter first side: '))
b = float(input('Enter second side: '))
c = float(input('Enter third side: '))

# calculate the semi-perimeter
s = (a + b + c) / 2

# calculate the area
area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
print('The area of the triangle is %0.2f' %area)
```

Output

```
Enter first side: 5
Enter second side: 6
Enter third side: 7
The area of the triangle is 14.70
```

In this program, we asked users to enter the length of three sides of a triangle. We used the Heron's Formula to calculate the semi-perimeter and hence the area of the triangle.

Python Program to Solve Quadratic Equation

- Python Variables and Datatypes
- Python Input, Output and Import
- Python Operators

Source Code

```
# Solve the quadratic equation ax**2 + bx + c = 0
# Coeffients a, b and c are provided by the user

# import complex math module
import cmath

a = float(input('Enter a: '))
b = float(input('Enter b: '))
c = float(input('Enter c: '))

# calculate the discriminant
d = (b**2) - (4*a*c)

# find two solutions
sol1 = (-b-cmath.sqrt(d))/(2*a)
sol2 = (-b+cmath.sqrt(d))/(2*a)

print('The solution are {0} and {1}'.format(sol1,sol2))
```

Output

```
Enter a: 1
Enter b: 5
Enter c: 6
The solutions are (-3+0j) and (-2+0j)
```

In this program, we ask the user for the coefficients of the quadratic equation. We have imported the `cmath` module to perform complex square root. First we calculate the discriminant and then find the two solutions of the quadratic equation.

Python Program to Swap Two Variables

- Python Variables and Datatypes
- Python Input, Output and Import
- Python Operators

Source Code

```
# Python program to swap two variables provided by the user

x = input('Enter value of x: ')
y = input('Enter value of y: ')

# create a temporary variable and swap the values
temp = x
x = y
y = temp

print('The value of x after swapping: {}'.format(x))
print('The value of y after swapping: {}'.format(y))
```

Output

```
Enter value of x: 5
Enter value of y: 10
The value of x after swapping: 10
The value of y after swapping: 5
```

In this program, we use the *temp* variable to temporarily hold the value of *x*. We then put the value of *y* in *x* and later *temp* in *y*. In this way, the values get exchanged.

Python Program to Swap Variables Without Temporary Variable

In python programming, there is a simple construct to swap variables. The following code does the same as above but without the use of any temporary variable.

```
x,y = y,x
```

If the variables are both numbers, we can use arithmetic operations to do the same. It might not look intuitive at the first sight. But if you think about it, its pretty easy to figure it out.Here are a few example

Addition and Subtraction

```
x = x + y
y = x - y
x = x - y
```

Multiplication and Division

```
x = x * y
y = x / y
x = x / y
```

XOR swap

This algorithm works for integers only

```
x = x ^ y
y = x ^ y
x = x ^ y
```

Python Program to Generate a Random Number

- Python Input, Output and Import
- Python Random Module

Source Code

```
# Program to generate a random number between 0 and 9

# import the random module
import random

print(random.randint(0,9))
```

Output

```
5
```

In this program, we use the `randint()` function inside the random module. Note that, we may get different output because this program generates random number in range 0 and 9. The syntax of this function is:

```
random.randint(a,b)
```

This returns a number *N* in the inclusive range `[a,b]`, meaning `a <= N <= b`, where the endpoints are included in the range.

Python Program to Convert Kilometers to Miles

- Python Variables and Datatypes
- Python Input, Output and Import
- Python Operators

Source Code

```
# Program to convert kilometers into miles
# Input is provided by the user in kilometers

# take input from the user
kilometers = float(input('How many kilometers?: '))

# conversion factor
conv_fac = 0.621371

# calculate miles
miles = kilometers * conv_fac
print('%0.3f kilometers is equal to %0.3f miles' %(kilometers,miles))
```

Output

```
How many kilometers?: 5.5
5.500 kilometers is equal to 3.418 miles
```

Explanation

In this program, we use the ask the user for kilometers and convert it to miles by multiplying it with the conversion factor. With a slight modification, we can convert miles to kilometers. We ask for miles and use the following formula to convert it into kilometers.

```
kilometers = miles / conv_fac
```

Python Program to Convert Celsius To Fahrenheit

- Python Variables and Datatypes
- Python Input, Output and Import
- Python Operators

Source Code

```
# Python Program to convert temperature in celsius to fahrenheit
# Input is provided by the user in degree celsius

# take input from the user
celsius = float(input('Enter degree Celsius: '))

# calculate fahrenheit
fahrenheit = (celsius * 1.8) + 32
print('%0.1f degree Celsius is equal to %0.1f degree Fahrenheit'
%(celsius,fahrenheit))
```

Output

```
Enter degree Celsius: 37.5
37.5 degree Celsius is equal to 99.5 degree Fahrenheit
```

In this program, we ask the user for temperature in degree Celsius and convert it into degree Fahrenheit. They are related by the formula `celsius * 1.8 = fahrenheit - 32`. With a simple modification to this program, we can convert Fahrenheit into Celsius. We ask the user for temperature in Fahrenheit and use the following formula to convert it into Celsius.

```
celsius = (fahrenheit - 32) / 1.8
```

Python Program to Check if a Number is Positive, Negative or Zero

- Python if...elif...else and Nested if

Source Code

```
# In this python program, user enters a number and checked if the number is
positive or negative or zero

num = float(input("Enter a number: "))
if num > 0:
   print("Positive number")
elif num == 0:
   print("Zero")
else:
   print("Negative number")
```

Here, we have used the `if...elif...else` statement. We can do the same thing using nested `if` statements as follows.

```
# This time use nested if to solve the problem

num = float(input("Enter a number: "))
if num >= 0:
   if num == 0:
       print("Zero")
   else:
       print("Positive number")
else:
   print("Negative number")
```

Output 1

```
Enter a number: 2
Positive number
```

Output 2

```
Enter a number: 0
Zero
```

A number is positive if it is greater than zero. We check this in the expression of `if`. If it is `False`, the number will either be zero or negative. This is also tested in subsequent expression.

Python Program to Check if a Number is Odd or Even

- Python Operators
- Python if...elif...else and Nested if

Source Code

```
# Python program to check if the input number is odd or even.
# A number is even if division by 2 give a remainder of 0.
# If remainder is 1, it is odd number.

num = int(input("Enter a number: "))
if (num % 2) == 0:
   print("{0} is Even".format(num))
else:
   print("{0} is Odd".format(num))
```

Output 1

```
Enter a number: 43
43 is Odd
```

Output 2

```
Enter a number: 18
18 is Even
```

In this program, we ask the user for the input and check if the number is odd or even. A number is even if it is perfectly divisible by 2. When the number is divided by 2, we use the remainder operator `%` to compute the remainder. If the remainder is not zero, the number is odd.

Python Program to Check Leap Year

- Python Operators
- Python if...elif...else and Nested if

Source Code

```
# Python program to check if the input year is a leap year or not

year = int(input("Enter a year: "))
```

```
if (year % 4) == 0:
   if (year % 100) == 0:
       if (year % 400) == 0:
           print("{0} is a leap year".format(year))
       else:
           print("{0} is not a leap year".format(year))
   else:
       print("{0} is a leap year".format(year))
else:
   print("{0} is not a leap year".format(year))
```

Output 1

```
Enter a year: 2000
2000 is a leap year
```

Output 2

```
Enter a year: 1775
1775 is not a leap year
```

In this program, we ask the user to input a year and check if it is a leap year or not. Leap years are those divisible by 4. Except those that are divisible by 100 but not by 400. Thus 1900 is not a leap year as it is divisible by 100. But 2000 is a leap year because it if divisible by 400 as well.

Python Program to Find the Largest Among Three Numbers

- Python if...elif...else and Nested if

Source Code

```
# Python program to find the largest number among the three input numbers

# take three numbers from user
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
num3 = float(input("Enter third number: "))

if (num1 > num2) and (num1 > num3):
   largest = num1
elif (num2 > num1) and (num2 > num3):
   largest = num2
else:
   largest = num3

print("The largest number is",largest)
```

Output 1

```
Enter first number: 10
Enter second number: 12
Enter third number: 14
The largest number is 14.0
```

Output 2

```
Enter first number: -1
Enter second number: 0
Enter third number: -3
The largest number is 0.0
```

In this program, we ask the user to input three numbers. We use the `if...elif...else` ladder to find the largest among the three and display it.

Python Program to Check Prime Number

- Python if...elif...else and Nested if
- Python for Loop
- Python break and continue Statement

A positive integer greater than 1 which has no other factors except 1 and the number itself is called a prime number. 2, 3, 5, 7 etc. are prime numbers as they do not have any other factors. But 6 is not prime (it is composite) since, `2 x 3 = 6`.

Source Code

```
# Python program to check if the input number is prime or not

# take input from the user
num = int(input("Enter a number: "))

# prime numbers are greater than 1
if num > 1:
   # check for factors
   for i in range(2,num):
       if (num % i) == 0:
           print(num,"is not a prime number")
           print(i,"times",num//i,"is",num)
           break
   else:
       print(num,"is a prime number")

# if input number is less than
# or equal to 1, it is not prime
else:
   print(num,"is not a prime number")
```

Output 1

```
Enter a number: 407
407 is not a prime number
11 times 37 is 407
```

Output 2

```
Enter a number: 853
853 is a prime number
```

In this program, user is asked to enter a number and this program check whether that number is prime or not. Numbers less than or equal to 1 are not prime numbers. Hence, we only proceed if the *num* is greater than 1. We check if *num* is exactly divisible by any number from 2 to *num* - 1. If we find a factor in that range, the number is not prime. Else the number is prime.

We can decrease the range of numbers where we look for factors. In the above program, our search range is from 2 to *num* - 1. We could have used the range, [2, *num* / 2] or [2, *num* ** 0.5]. The later range is based on the fact that a composite number must have a factor less than square root of that number. Otherwise the number is prime.

Python Program to Print all Prime Numbers in an Interval

- Python if...elif...else and Nested if
- Python for Loop
- Python break and continue Statement

A positive integer greater than 1 which has no other factors except 1 and the number itself is called a prime number. 2, 3, 5, 7 etc. are prime numbers as they do not have any other factors. But 6 is not prime (it is composite) since, `2 x 3 = 6`. We ask the user for a range and display all the primes in that interval.

Source Code

```
# Python program to ask the user for a range and display all the prime
numbers in that interval

# take input from the user
lower = int(input("Enter lower range: "))
upper = int(input("Enter upper range: "))

for num in range(lower,upper + 1):
   # prime numbers are greater than 1
   if num > 1:
       for i in range(2,num):
           if (num % i) == 0:
               break
       else:
           print(num)
```

Output

```
Enter lower range: 900
Enter upper range: 1000
907
911
919
929
937
941
947
953
967
971
977
983
991
997
```

Here, we take an interval from the user and find prime numbers in that range.

Python Program to Find the Factorial of a Number

- Python if...elif...else and Nested if
- Python for Loop

The factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720. Factorial is not defined for negative numbers and the factorial of zero is one, 0! = 1.

Source Code

```
# Python program to find the factorial of a number provided by the user.

# take input from the user
num = int(input("Enter a number: "))
factorial = 1

# check if the number is negative, positive or zero
if num < 0:
   print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
else:
   for i in range(1,num + 1):
       factorial = factorial*i
   print("The factorial of",num,"is",factorial)
```

Output 1

```
Enter a number: -2
Sorry, factorial does not exist for negative numbers
```

Output 2

```
Enter a number: 7
The factorial of 7 is 5040
```

Here, we take input from the user and check if the number is negative, zero or positive using `if...elif...else` statement. If the number is positive, we use `for` loop and `range()` function to calculate the factorial.

Python Program to Display the multiplication Table

- Python for Loop

Source Code

```
# Python program to find the multiplication table (from 1 to 10) of a number
input by the user

# take input from the user
num = int(input("Display multiplication table of? "))

# use for loop to iterate 10 times
for i in range(1,11):
   print(num,'x',i,'=',num*i)
```

Output

```
Display multiplication table of? 12
12 x 1 = 12
12 x 2 = 24
12 x 3 = 36
12 x 4 = 48
12 x 5 = 60
12 x 6 = 72
12 x 7 = 84
12 x 8 = 96
12 x 9 = 108
12 x 10 = 120
```

Here, we ask the user for a number and display the multiplication table upto 10. We use `for` loop along with the `range()` function to iterate 10 times.

Python Program to Print the Fibonacci sequence

- Python if...elif...else and Nested if
- Python while Loop

A Fibonacci sequence is the integer sequence of 0, 1, 1, 2, 3, 5, 8.... The first two terms are 0 and 1. All other terms are obtained by adding the preceding two terms. This means to say the nth term is the sum of (n-1)th and (n-2)th term.

Source Code

```
# Program to display the Fibonacci sequence up to n-th term where n is
provided by the user

# take input from the user
nterms = int(input("How many terms? "))

# first two terms
n1 = 0
n2 = 1
count = 2

# check if the number of terms is valid
if nterms <= 0:
   print("Plese enter a positive integer")
elif nterms == 1:
   print("Fibonacci sequence:")
   print(n1)
else:
   print("Fibonacci sequence:")
   print(n1,",",n2,end=', ')
   while count < nterms:
       nth = n1 + n2
       print(nth,end=' , ')
       # update values
       n1 = n2
       n2 = nth
       count += 1
```

Output

```
How many terms? 10
Fibonacci sequence:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34,
```

Here, we ask the user for the number of terms in the sequence. We initialize the first term to 0 and the second term to 1. If the number of terms is more than 2, we use a `while` loop to find the next term in the sequence by adding the preceding two terms. We then interchange the variables (update it) and continue on with the process.

Python Program to Check Armstrong Number

- Python if...elif...else and Nested if
- Python while Loop

An Armstrong number, also known as narcissistic number, is a number that is equal to the sum of the cubes of its own digits. For example, 370 is an Armstrong number since 370 = 3*3*3 + 7*7*7 + 0*0*0.

Source Code

```
# Python program to check if the number provided by the user is an Armstrong
number or not

# take input from the user
num = int(input("Enter a number: "))

# initialise sum
sum = 0

# find the sum of the cube of each digit
temp = num
while temp > 0:
   digit = temp % 10
   sum += digit ** 3
   temp //= 10

# display the result
if num == sum:
   print(num,"is an Armstrong number")
else:
   print(num,"is not an Armstrong number")
```

Output 1

```
Enter a number: 663
663 is not an Armstrong number
```

Output 2

```
Enter a number: 407
407 is an Armstrong number
```

Here, we ask the user for a number and check if it is an Armstrong number. We need to calculate the sum of cube of each digit. So, we initialize the sum to 0 and obtain each digit number by using the modulus operator %. Remainder of a number when it is divide by 10 is the last digit of that number. We take the cubes using exponent operator. Finally, we compare the sum with the original number and conclude that it is Armstrong number if they are equal.

Python Program to Find Armstrong Number in an Interval

- Python if...elif...else and Nested if
- Python while Loop

An Armstrong number, also known as narcissistic number, is a number that is equal to the sum of the cubes of its own digits. For example, 371 is an Armstrong number since 371 = 3*3*3 + 7*7*7 + 1*1*1.

Source Code

```
# Program to ask the user for a range and display all Armstrong numbers in
that interval

# take input from the user
lower = int(input("Enter lower range: "))
upper = int(input("Enter upper range: "))

for num in range(lower,upper + 1):
   # initialize sum
   sum = 0

   # find the sum of the cube of each digit
   temp = num
   while temp > 0:
       digit = temp % 10
       sum += digit ** 3
       temp //= 10

   if num == sum:
       print(num)
```

Output

```
Enter lower range: 100
Enter upper range: 1000
153
370
371
407
```

Here, we ask the user for the interval in which we want to search for Armstrong numbers. We scan through the interval and display all the numbers that meet the condition. We can see that there are 4 three digit Armstrong numbers.

Python Program to Find the Sum of Natural Numbers

- Python if...elif...else and Nested if
- Python while Loop

Source Code

```
# Python program to find the sum of natural numbers up to n where n is
provided by user

# take input from the user
num = int(input("Enter a number: "))

if num < 0:
   print("Enter a positive number")
else:
   sum = 0
   # use while loop to iterate un till zero
   while(num > 0):
       sum += num
       num -= 1
   print("The sum is",sum)
```

Output

```
Enter a number: 16
The sum is 136
```

Here, we ask the user for a number and display the sum of natural numbers up to that number. We use `while` loop to iterate until the number becomes zero.

We could have solved the above problem without using any loops. From mathematics, we know that sum of natural numbers is given by `n*(n+1)/2`. We could have used this formula directly. For example, if **n = 16**, the sum would be **(16*17)/2 = 136**.

Python Program To Display Powers of 2 Using Anonymous Function

- Python for Loop
- Python Anonymous/Lambda Function

Source Code

```
# Python Program to display the powers of 2 using anonymous function

# Take number of terms from user
terms = int(input("How many terms? "))

# use anonymous function
result = list(map(lambda x: 2 ** x, range(terms)))

# display the result
for i in range(terms):
   print("2 raised to power",i,"is",result[i])
```

Output

```
How many terms? 10
2 raised to power 0 is 1
2 raised to power 1 is 2
2 raised to power 2 is 4
2 raised to power 3 is 8
2 raised to power 4 is 16
2 raised to power 5 is 32
2 raised to power 6 is 64
2 raised to power 7 is 128
2 raised to power 8 is 256
2 raised to power 9 is 512
```

In this program, we have used anonymous (lambda) function inside the `map()` built-in function to find the powers of 2.

Python Program to Find Numbers Divisible by Another Number

- Python Anonymous/Lambda Function
- Python List

Source Code

```
# Python Program to find numbers divisible by thirteen from a list using
anonymous function

# Take a list of numbers
my_list = [12, 65, 54, 39, 102, 339, 221,]

# use anonymous function to filter
result = list(filter(lambda x: (x % 13 == 0), my_list))

# display the result
print("Numbers divisible by 13 are",result)
```

Output

```
Numbers divisible by 13 are [65, 39, 221]
```

In this program, we have used anonymous (lambda) function inside the `filter()` built-in function to find all the numbers divisible by 13 in the list.

Python Program to Convert Decimal to Binary, Octal and Hexadecimal

- Python Programming Built-in Functions

Decimal system is the most widely used number system. But computer only understands binary. Binary, octal and hexadecimal number systems are closely related and we may require to convert decimal into these systems. Decimal system is base 10 (ten symbols, 0-9, are used to represent a number) and similarly, binary is base 2, octal is base 8 and hexadecimal is base 16.

A number with the prefix '0b' is considered binary, '0o' is considered octal and '0x' as hexadecimal. For example:

```
60 = 0b11100 = 0o74 = 0x3c
```

Source Code

```
# Python program to convert decimal number into binary, octal and hexadecimal
number system

# Take decimal number from user
dec = int(input("Enter an integer: "))
```

```
print("The decimal value of",dec,"is:")
print(bin(dec),"in binary.")
print(oct(dec),"in octal.")
print(hex(dec),"in hexadecimal.")
```

Output

```
Enter an integer: 344
The decimal value of 344 is:
0b101011000 in binary.
0o530 in octal.
0x158 in hexadecimal.
```

In this program, we have used built-in functions `bin()`, `oct()` and `hex()` to convert the given decimal number into respective number systems. These functions take an integer (in decimal) and return a string.

Python Program to Find ASCII Value of Character

- Python Programming Built-in Functions

ASCII stands for American Standard Code for Information Interchange. It is a numeric value given to different characters and symbols, for computers to store and manipulate. For example: ASCII value of the letter 'A' is 65.

Source Code

```
# Program to find the ASCII value of the given character

# Take character from user
c = input("Enter a character: ")

print("The ASCII value of '" + c + "' is",ord(c))
```

Output 1

```
Enter a character: p
The ASCII value of 'p' is 112
```

Here we have used `ord()` function to convert a character to an integer (ASCII value). This function actually returns the Unicode code point of that character. Unicode is also an encoding technique that provides a unique number to a character. While ASCII only encodes 128 characters, current Unicode has more than 100,000 characters from hundreds of scripts.

We can use `chr()` function to inverse this process, meaning, return a character for the input integer.

```
>>> chr(65)
'A'
```

```
>>> chr(120)
'x'
>>> chr(ord('S') + 1)
'T'
```

Here, `ord()` and `chr()` are built-in functions.

Python Program to Find HCF or GCD

- Python while Loop
- Python for Loop
- Python Functions
- Python Function Arguments
- Python Programming User-defined Functions

The highest common factor (H.C.F) or greatest common divisor (G.C.D) of two numbers is the largest positive integer that perfectly divides the two given numbers. For example, the H.C.F of 12 and 14 is 2.

Source Code

```
# Python program to find the H.C.F of two input number

# define a function
def hcf(x, y):
   """This function takes two
   integers and returns the H.C.F"""

   # choose the smaller number
   if x > y:
       smaller = y
   else:
       smaller = x

   for i in range(1,smaller + 1):
       if((x % i == 0) and (y % i == 0)):
           hcf = i

   return

# take input from the user
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

print("The H.C.F. of", num1,"and", num2,"is", hcf(num1, num2))
```

Output

```
Enter first number: 54
Enter second number: 24
The H.C.F. of 54 and 24 is 6
```

This program asks for two integers and passes them to a function which returns the H.C.F. In the function, we first determine the smaller of the two number since the H.C.F can only be less than or equal to the smallest number. We then use a `for` loop to go from 1 to that number. In each iteration we check if our number perfectly divides both the input numbers. If so, we store the number as H.C.F. At the completion of the loop we end up with the largest number that perfectly divides both the numbers.

The above method is easy to understand and implement but not efficient. A much more efficient method to find the H.C.F. is the Euclidean algorithm.

Euclidean algorithm

This algorithm is based on the fact that H.C.F. of two numbers divides their difference as well. In this algorithm, we divide the greater by smaller and take the remainder. Now, divide the smaller by this remainder. Repeat until the remainder is 0.

For example, if we want to find the H.C.F. of 54 and 24, we divide 54 by 24. The remainder is 6. Now, we divide 24 by 6 and the remainder is 0. Hence, 6 is the required H.C.F. We can do this in Python as follows.

Source Code

```
def hcf(x, y):
   """This function implements the Euclidian algorithm
   to find H.C.F. of two numbers"""

   while(y):
       x, y = y, x % y

   return x
```

Here we loop until *y* becomes zero. The statement `x, y = y, x % y` does swapping of values in Python. Click here to learn more about swapping variables in Python. In each iteration we place the value of *y* in *x* and the remainder `(x % y)` in *y*, simultaneously. When *y* becomes zero, we have H.C.F. in *x*.

Python Program to Find LCM

- Python while Loop
- Python Functions
- Python Function Arguments
- Python Programming User-defined Functions

The least common multiple (L.C.M.) of two numbers is the smallest positive integer that is perfectly divisible by the two given numbers. For example, the L.C.M. of 12 and 14 is 84.

Source Code to find LCM

```
# Python Program to find the L.C.M. of two input number

# define a function
def lcm(x, y):
   """This function takes two
   integers and returns the L.C.M."""

   # choose the greater number
   if x > y:
       greater = x
   else:
       greater = y

   while(True):
       if((greater % x == 0) and (greater % y == 0)):
           lcm = greater
           break
       greater += 1

   return lcm


# take input from the user
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

print("The L.C.M. of", num1,"and", num2,"is", lcm(num1, num2))
```

Output

```
Enter first number: 54
Enter second number: 24
The L.C.M. of 54 and 24 is 216
```

This program asks for two integers and passes them to a function which returns the L.C.M. In the function, we first determine the greater of the two number since the L.C.M. can only be greater than or equal to the largest number. We then use an infinite `while` loop to go from that number and beyond. In each iteration, we check if both the input numbers perfectly divides our number. If so, we store the number as L.C.M. and break from the loop. Otherwise, the number is incremented by 1 and the loop continues.

The above program is slower to run. We can make it more efficient by using the fact that the product of two numbers is equal to the product of least common multiple and greatest common divisor of those two numbers.

```
Number1 * Number2 = L.C.M. * G.C.D.
```

Here is a Python program to implement this.

Source Code

```
# Python program to find the L.C.M. of two input number

# define gcd function
def gcd(x, y):
   """This function implements the Euclidian algorithm
   to find G.C.D. of two numbers"""

   while(y):
       x, y = y, x % y

   return x

# define lcm function
def lcm(x, y):
   """This function takes two
   integers and returns the L.C.M."""

   lcm = (x*y)//gcd(x,y)
   return lcm


# take input from the user
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

print("The L.C.M. of", num1,"and", num2,"is", lcm(num1, num2))
```

The output of this program is same as before. We have two functions `gcd()` and `lcm()`. We require G.C.D. of the numbers to calculate its L.C.M. So, `lcm()` calls the function `gcd()` to accomplish this. G.C.D. of two numbers can be calculated efficiently using the Euclidean algorithm.

Python Program to Find Factors of Number

- Python if...elif...else and Nested if
- Python while Loop
- Python Programming User-defined Functions

Source Code

```
# Python Program to find the factors of a number

# define a function
def print_factors(x):
   """This function takes a
   number and prints the factors"""

   print("The factors of",x,"are:")
   for i in range(1, x + 1):
       if x % i == 0:
           print(i)
```

```
# take input from the user
num = int(input("Enter a number: "))

print_factors(num)
```

Output

```
Enter a number: 320
The factors of 320 are:
1
2
4
5
8
10
16
20
32
40
64
80
160
320
```

In this program we take a number from the user and display its factors using the function `print_factors()`. In the function, we use a `for` loop to iterate from 1 to that number and only print it if, it perfectly divides our number. Here, `print_factors()` is a user-defined function.

Python Program to Make a Simple Calculator

- Python Functions
- Python Function Arguments
- Python Programming User-defined Functions

Source Code

```
# Program make a simple calculator that can add, subtract, multiply and
divide using functions

# define functions
def add(x, y):
   """This function adds two numbers"""

   return x + y

def subtract(x, y):
   """This function subtracts two numbers"""

   return x - y
```

```
def multiply(x, y):
   """This function multiplies two numbers"""

   return x * y

def divide(x, y):
   """This function divides two numbers"""

   return x / y

# take input from the user
print("Select operation.")
print("1.Add")
print("2.Subtract")
print("3.Multiply")
print("4.Divide")

choice = input("Enter choice(1/2/3/4):")

num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

if choice == '1':
   print(num1,"+",num2,"=", add(num1,num2))

elif choice == '2':
   print(num1,"-",num2,"=", subtract(num1,num2))

elif choice == '3':
   print(num1,"*",num2,"=", multiply(num1,num2))

elif choice == '4':
   print(num1,"/",num2,"=", divide(num1,num2))
else:
   print("Invalid input")
```

Output

```
Select operation.
1.Add
2.Subtract
3.Multiply
4.Divide
Enter choice(1/2/3/4): 3
Enter first number: 15
Enter second number: 14
15 * 14 = 210
```

In this program, we ask the user to choose the desired operation. Options 1,2,3 and 4 are valid. Two numbers are taken and an `if...elif...else` branching is used to execute a particular section. User-defined functions `add()`, `subtract()`, `multiply()` and `divide()` evaluate respective operations.

Python Program to Shuffle Deck of Cards

- Python for Loop
- Python Modules
- Python Random Module
- Python Programming Built-in Functions

Source Code

```
# Python program to shuffle a deck of card using the module random and draw 5
cards

# import modules
import itertools, random

# make a deck of cards
deck =
list(itertools.product(range(1,14),['Spade','Heart','Diamond','Club']))

# shuffle the cards
random.shuffle(deck)

# draw five cards
print("You got:")
for i in range(5):
   print(deck[i][0], "of", deck[i][1])
```

Output 1

```
You got:
5 of Heart
1 of Heart
8 of Spade
12 of Spade
4 of Spade
```

Output 2

```
You got:
10 of Club
1 of Heart
3 of Diamond
2 of Club
3 of Club
```

In program, we used the `product()` function in `itertools` module to create a deck of cards. This function performs the Cartesian product of the two sequence. The two sequence are, numbers from 1 to 13 and the four suits. So, altogether we have 13 * 4 = 52 items in the deck with each card as a tuple. For e.g. `deck[0] = (1, 'Spade')`. Our deck is ordered, so we shuffle it using the function `shuffle()` in `random` module. Finally, we draw the first five cards and display it to the user. We will get different output each time you run this program as shown in our two outputs.

Here we have used the standard modules `itertools` and `random` that comes with Python.

Python Program to Display Calendar

- Python Modules
- Python Programming Built-in Functions

Source Code

```
# Python program to display calendar of given month of the year

# import module
import calendar

# ask of month and year
yy = int(input("Enter year: "))
mm = int(input("Enter month: "))

# display the calendar
print(calendar.month(yy,mm))
```

Output

```
Enter year: 2014
Enter month: 11
  November 2014
Mo Tu We Th Fr Sa Su
               1  2
3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

In this program we import the `calendar` module. We ask the user for a year and month. The `month()` function inside the module takes in the year and the month and displays the calendar for that month of the year.

Here we have used the standard module `calendar` that comes with Python.

Python Program to Display Fibonacci Sequence Using Recursion

- Python for Loop
- Python Functions
- Python Recursion

A Fibonacci sequence is the integer sequence of 0, 1, 1, 2, 3, 5, 8.... The first two terms are 0 and 1. All other terms are obtained by adding the preceding two terms. This means to say the nth term is the sum of (n-1)th and (n-2)th term.

Source Code

```
# Python program to display the Fibonacci sequence up to n-th term using
recursive functions

def recur_fibo(n):
   """Recursive function to
   print Fibonacci sequence"""
   if n <= 1:
       return n
   else:
       return(recur_fibo(n-1) + recur_fibo(n-2))


# take input from the user
nterms = int(input("How many terms? "))

# check if the number of terms is valid
if nterms <= 0:
   print("Plese enter a positive integer")
else:
   print("Fibonacci sequence:")
   for i in range(nterms):
       print(recur_fibo(i))
```

Output

```
How many terms? 10
Fibonacci sequence:
0
1
1
2
3
5
8
13
21
34
```

In this program, we ask the user for the number of terms in the sequence. A recursive function `recur_fibo()` is used to calculate the nth term of the sequence. We use a `for` loop to iterate and calculate each term recursively.

Python Program to Find Sum of Natural Numbers Using Recursion

- Python if...elif...else and Nested if
- Python Functions
- Python Recursion

Source Code

```
# Python program to find the sum of natural numbers up to n using recursive
function

def recur_sum(n):
   """Function to return the sum
   of natural numbers using recursion"""
   if n <= 1:
       return n
   else:
       return n + recur_sum(n-1)

# take input from the user
num = int(input("Enter a number: "))

if num < 0:
   print("Enter a positive number")
else:
   print("The sum is",recur_sum(num))
```

Output

```
Enter a number: 16
The sum is 136
```

In this program, we ask the user for a number and use recursive function `recur_sum()` to compute the sum up to that number.

Python Program to Find Factorial of Number Using Recursion

- Python if...elif...else and Nested if
- Python Functions
- Python Recursion

The factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720. Factorial is not defined for negative numbers and the factorial of zero is one, 0! = 1.

Source Code

```
# Python program to find the factorial of a number using recursion

def recur_factorial(n):
   """Function to return the factorial
   of a number using recursion"""
   if n == 1:
       return n
   else:
       return n*recur_factorial(n-1)


# take input from the user
num = int(input("Enter a number: "))

# check is the number is negative
if num < 0:
   print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
else:
   print("The factorial of",num,"is",recur_factorial(num))
```

Output 1

```
Enter a number: -2
Sorry, factorial does not exist for negative numbers
```

Output 2

```
Enter a number: 7
The factorial of 7 is 5040
```

Here, we ask the user for a number and use recursive function `recur_factorial()` to compute the product up to that number.

Python Program to Convert Decimal to Binary Using Recursion

- Python if...elif...else and Nested if
- Python Functions
- Python Recursion

Source Code

```
# Python program to convert decimal number into binary number using recursive
function

def binary(n):
   """Function to print binary number
   for the input decimal using recursion"""
   if n > 1:
        binary(n//2)
    print(n % 2,end = '')

# Take decimal number from user
dec = int(input("Enter an integer: "))
binary(dec)
```

Output

```
Enter an integer: 52
110100
```

In this program, we convert decimal number entered by the user into binary using a recursive function. Decimal number is converted into binary by dividing the number successively by 2 and printing the remainder in reverse order.

Python Program to Add Two Matrices

- Python for Loop
- Python List

In Python, we can implement a matrix as nested list (list inside a list). We can treat each element as a row of the matrix. For example `X = [[1, 2], [4, 5], [3, 6]]` would represent a 3x2 matrix. First row can be selected as `X[0]` and the element in first row, first column can be selected as `X[0][0]`.

We can perform matrix addition in various ways in Python. Here are a couple of them.

Matrix Addition using Nested Loop

Source Code

```
# Program to add two matrices using nested loop

X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]

Y = [[5,8,1],
    [6,7,3],
    [4,5,9]]

result = [[0,0,0],
         [0,0,0],
         [0,0,0]]

# iterate through rows
for i in range(len(X)):
   # iterate through columns
   for j in range(len(X[0])):
       result[i][j] = X[i][j] + Y[i][j]

for r in result:
   print(r)
```

Output

```
[17, 15, 4]
[10, 12, 9]
[11, 13, 18]
```

Explanation

In this program we have used nested `for` loops to iterate through each row and each column. At each point we add the corresponding elements in the two matrices and store it in the result.

Matrix Addition using Nested List Comprehension

Source Code

```
# Program to add two matrices
# using list comprehension

X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]

Y = [[5,8,1],
    [6,7,3],
    [4,5,9]]

result = [[X[i][j] + Y[i][j]  for j in range(len(X[0]))] for i in
range(len(X))]

for r in result:
   print(r)
```

The output of this program is the same as above. We have used nested list comprehension to iterate through each element in the matrix. List comprehension allows us to write concise codes and we must try to use them frequently in Python. They are very helpful.

Python Program to Transpose a Matrix

- Python for Loop
- Python List

In Python, we can implement a matrix as nested list (list inside a list). We can treat each element as a row of the matrix. For example `X = [[1, 2], [4, 5], [3, 6]]` would represent a 3x2 matrix. First row can be selected as `X[0]` and the element in first row, first column can be selected as `X[0][0]`.

Transpose of a matrix is the interchanging of rows and columns. It is denoted as *X'*. The element at *ith* row and *jth* column in *X* will be placed at *jth* row and *ith* column in *X'*. So if *X* is a 3x2 matrix, *X'* will be a 2x3 matrix. Here are a couple of ways to accomplish this in Python.

Matrix Transpose using Nested Loop

Source Code

```
# Program to transpose a matrix using nested loop

X = [[12,7],
    [4 ,5],
    [3 ,8]]

result = [[0,0,0],
         [0,0,0]]

# iterate through rows
for i in range(len(X)):
   # iterate through columns
   for j in range(len(X[0])):
       result[j][i] = X[i][j]

for r in result:
   print(r)
```

Output

```
[12, 4, 3]
[7, 5, 8]
```

In this program we have used nested `for` loops to iterate through each row and each column. At each point we place the *X[i][j]* element into *result[j][i]*.

Matrix Transpose using Nested List Comprehension

Source Code

```
# Program to transpose a matrix
# using list comprehension

X = [[12,7],
    [4 ,5],
    [3 ,8]]

result = [[X[j][i] for j in range(len(X))] for i in range(len(X[0]))]

for r in result:
   print(r)
```

Explanation

The output of this program is the same as above. We have used nested list comprehension to iterate through each element in the matrix. List comprehension allows us to write concise codes and we must try to use them frequently in Python. They are very helpful.

Python Program to Multiply Two Matrices

- Python for Loop
- Python List

In Python we can implement a matrix as nested list (list inside a list). We can treat each element as a row of the matrix. For example `X = [[1, 2], [4, 5], [3, 6]]` would represent a 3x2 matrix. First row can be selected as `X[0]` and the element in first row, first column can be selected as `X[0][0]`.

Multiplication of two matrices *X* and *Y* is defined only if the number of columns in *X* is equal to the number of rows *Y*. If *X* is a `n x m` matrix and *Y* is a `m x l` matrix then, *XY* is defined and has the dimension `n x l` (but *YX* is not defined). Here are a couple of ways to implement matrix multiplication in Python.

Matrix Multiplication using Nested Loop

Source Code

```
# Program to multiply two matrices using nested loops

# 3x3 matrix
X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]
# 3x4 matrix
Y = [[5,8,1,2],
    [6,7,3,0],
    [4,5,9,1]]
# result is 3x4
result = [[0,0,0,0],
         [0,0,0,0],
         [0,0,0,0]]

# iterate through rows of X
for i in range(len(X)):
   # iterate through columns of Y
   for j in range(len(Y[0])):
       # iterate through rows of Y
       for k in range(len(Y)):
           result[i][j] += X[i][k] * Y[k][j]

for r in result:
   print(r)
```

Output

```
[114, 160, 60, 27]
[74, 97, 73, 14]
[119, 157, 112, 23]
```

In this program, we have used nested `for` loops to iterate through each row and each column. We accumulate the sum of products in the result. This technique is simple but computationally expensive as we increase the order of matrix. For larger matrix operations we recommend optimized software packages like NumPy which is several (in the order of 1000) times faster than the above code.

Matrix Multiplication using Nested List Comprehension

Source Code

```
# Program to multiply two matrices using list comprehension

# 3x3 matrix
X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]
# 3x4 matrix
Y = [[5,8,1,2],
    [6,7,3,0],
    [4,5,9,1]]

# result is 3x4
result = [[sum(a*b for a,b in zip(X_row,Y_col)) for Y_col in zip(*Y)] for
X_row in X]

for r in result:
   print(r)
```

The output of this program is the same as above. To understand the above code we must first know about built-in function `zip()` and unpacking argumnet list using * operator. We have used nested list comprehension to iterate through each element in the matrix. The code looks complicated and unreadable at first. But once you get the hang of list comprehensions, you will probably not go back to nested loops.

Python Program to Check Whether a String is Palindrome or Not

- Python if...elif...else and Nested if
- Python Strings
- Python String Methods

A palindrome is a string which is same read forward or backwards. For example: "dad" is the same in forward or reverse direction. Another example is "aibohphobia" which literally means, an irritable fear of palindromes.

Source Code

```
# Program to check if a string
#  is palindrome or not

# take input from the user
my_str = input("Enter a string: ")

# make it suitable for caseless comparison
my_str = my_str.casefold()

# reverse the string
rev_str = reversed(my_str)

# check if the string is equal to its reverse
if list(my_str) == list(rev_str):
   print("It is palindrome")
else:
   print("It is not palindrome")
```

Output 1

```
Enter a string: aIbohPhoBiA
It is palindrome
```

Output 2

```
Enter a string: palindrome
It is not palindrome
```

In this program, we have taken a string from the user. Using the method `casefold()` we make it suitable for caseless comparisons. Basically, this method returns a lowercased version of the string. We reverse the string using the built-in function `reversed()`. Since this function returns a reversed object, we use the `list()` function to convert them into a list before comparing.

Python Program to Remove Punctuations From a String

- Python for Loop
- Python Strings

Sometimes, we may wish to break a sentence into a list of words. In such cases, we may first want to clean up the string and remove all the punctuation marks. Here is an example of how it is done.

Source Code

```
# Program to all punctuation from the string provided by the user

# define punctuation
```

```
punctuations = '''!()-[]{};:'"\,<>./?@#$%^&*_~'''

# take input from the user
my_str = input("Enter a string: ")

# remove punctuation from the string
no_punct = ""
for char in my_str:
   if char not in punctuations:
       no_punct = no_punct + char

# display the unpunctuated string
print(no_punct)
```

Output

```
Enter a string: "Hello!!!", he said ---and went.
Hello he said and went
```

In this program, we first define a string of punctuations. Then, we iterate over the provided string using a `for` loop. In each iteration, we check if the character is a punctuation mark or not using the membership test. We have an empty string to which we add (concatenate) the character if it is not a punctuation. Finally, we display the cleaned up string.

Python Program to Sort Words in Alphabetic Order

- Python for Loop
- Python Strings
- Python String Methods

In this example, we illustrate how words can be sorted lexicographically (alphabetic order).

Source Code

```
# Program to sort alphabetically the words form a string provided by the user

# take input from the user
my_str = input("Enter a string: ")

# breakdown the string into a list of words
words = my_str.split()

# sort the list
words.sort()

# display the sorted words
for word in words:
   print(word)
```

Output

```
Enter a string: Hello this Is an Example With cased letters
Example
Hello
Is
With
an
cased
letters
this
```

In this program, we take a string form the user. Using the split() method the string is converted into a list of words. The split() method splits the string at whitespaces. The list of words is then sorted using the sort() method and all the words are displayed.

Python Program to Illustrate Different Set Operations

- Python Sets

Python offers a datatype called set whose elements must be unique. It can be used to perform different set operations like union, intersection, difference and symmetric difference.

Source Code

```
# Program to perform different set operations like in mathematics

# define three sets
E = {0, 2, 4, 6, 8};
N = {1, 2, 3, 4, 5};

# set union
print("Union of E and N is",E | N)

# set intersection
print("Intersection of E and N is",E & N)

# set difference
print("Difference of E and N is",E - N)

# set symmetric difference
print("Symmetric difference of E and N is",E ^ N)
```

Output

```
Union of E and N is {0, 1, 2, 3, 4, 5, 6, 8}
Intersection of E and N is {2, 4}
Difference of E and N is {8, 0, 6}
Symmetric difference of E and N is {0, 1, 3, 5, 6, 8}
```

In this program, we take two different sets and perform different set operations on them. This can equivalently done by using set methods.

Python Program to Count the Number of Each Vowel

- Python for Loop
- Python Strings
- Python String Methods

Source Code

```
# Program to count the number of each vowel in a string

# string of vowels
vowels = 'aeiou'

# take input from the user
ip_str = input("Enter a string: ")

# make it suitable for caseless comparisions
ip_str = ip_str.casefold()

# make a dictionary with each vowel a key and value 0
count = {}.fromkeys(vowels,0)

# count the vowels
for char in ip_str:
   if char in count:
       count[char] += 1

print(count)
```

Output

```
Enter a string: Hello, have you tried our turorial section yet?
{'e': 5, 'u': 3, 'o': 5, 'a': 2, 'i': 3}
```

In this program we have taken a string from the user. Using the method `casefold()` we make it suitable for caseless comparisions. Basically, this method returns a lowercased version of the string. We use the dictionary method `fromkeys()` to construct a new dictionary with each vowel as its key and all values equal to 0. This is initialization of the count. Next we iterate over the input string using a `for` loop. In each iteration we check if the character is in the dictionary keys (`True` if it is a vowel) and increment the value by 1 if true.

We can do the same thing using a dictionary comprehension.

Source Code

```
# Program to count the number of
# each vowel in a string using
# dictionary and list comprehension

# take input from the user
ip_str = input("Enter a string: ")
```

```
# make it suitable for caseless comparisions
ip_str = ip_str.casefold()

# count the vowels
count = {x:sum([1 for char in ip_str if char == x]) for x in 'aeiou'}

print(count)
```

Explanation

The ouput of this program is the same as above. Here we have nested a list comprehension inside a dictionary comprehension to count the vowels in a single line. However, this program is slower as we iterate over the entire input string for each vowel.

Python Program to Merge Mails

- Python String Methods
- Python File I/O

When we want to send the same invitations to many people, the body of the mail does not change. Only the name (and maybe address) needs to be changed. Mail merge is a process of doing this. Instead of writing each mail separately, we have a template for body of the mail and a list of names that we merge together to form all the mails.

Source Code to Merge Mails

```
# Python program to mail merger
# Names are in the file names.txt
# Body of the mail is in body.txt

# open names.txt for reading
with open("names.txt",'r',encoding = 'utf-8') as names_file:

   # open body.txt for reading
   with open("body.txt",'r',encoding = 'utf-8') as body_file:

       # read entire content of the body
       body = body_file.read()

       # iterate over names
       for name in names_file:
           mail = "Hello "+name+body

           # write the mails to individual files
           with open(name.strip()+".txt",'w',encoding = 'utf-8') as
mail_file:
               mail_file.write(mail)
```

For this program, we have written all the names in separate lines in the file "names.txt". The body is in the "body.txt" file. We open both the files in reading mode and iterate over each name using a `for` loop. A new file with the name "[*name*].txt" is created, where *name* is the name of that person. We use `strip()` method to clean up leading and trailing whitespaces (reading a line from the file also reads the newline '\n' character). Finally, we write the content of the mail into this file using the `write()` method.

Python Program to Find the Size (Resolution) of Image

- Python Functions
- Python Programming User-defined Functions
- Python File I/O

JPEG (pronounced "jay-peg") stands for Joint Photographic Experts Group. It is one of the most widely used compression techniques for image compression.

Most of the file formats have headers (initial few bytes) which contain useful information about the file. For example, jpeg headers contain information like height, width, number of color (grayscale or RGB) etc. In this program, we find the resolution of a jpeg image reading these headers, without using any external library.

Source Code of Find Resolution of JPEG Image

```
# Python Program to find the resolution of a jpeg image without using
external libraries

def jpeg_res(filename):
   """"This function prints the resolution
   of the jpeg image file passed into it"""

   # open image for reading in binary mode
   with open(filename,'rb') as img_file:

       # height of image (in 2 bytes) is at 164th position
       img_file.seek(163)

       # read the 2 bytes
       a = img_file.read(2)

       # calculate height
       height = (a[0] << 8) + a[1]

       # next 2 bytes is width
       a = img_file.read(2)

       # calculate width
       width = (a[0] << 8) + a[1]

   print("The resolution of the image is",width,"x",height)


jpeg_res("img1.jpg")
```

Output

```
The resolution of the image is 280 x 280
```

In this program, we opened the image in binary mode. Non-text files must be open in this mode. The height of the image is at 164th position followed by width of the image. Both are 2 bytes long. Note that this is true only for JPEG File Interchange Format (JFIF) standard. If your image is encode using other standard (like EXIF), the code will not work. We convert the 2 bytes into a number using bitwise shifting operator <<. Finally, the resolution is displayed.

Python Program to Find Hash of File

- Python Functions
- Python Programming User-defined Functions
- Python File I/O

Hash functions take an arbitrary amount of data and return a fixed-length bit string. They are widely used in cryptography for authentication purposes. There are many hashing functions like MD5, SHA-1 etc. The output of the function is called the digest message. Refer this page to know more about hash functions in cryptography.

In this example, we will illustrate how to hash a file. We will use the SHA-1 hashing algorithm. The digest of SHA-1 is 160 bits long. We do not feed the data from the file all at once, because some files are very large to fit in memory all at once. Breaking the file into small chunks will make the process memory efficient.

Source Code to Find Hash

```
# Python rogram to find the SHA-1 message digest of a file

# import hashlib module
import hashlib

def hash_file(filename):
   """"This function returns the SHA-1 hash
   of the file passed into it"""

   # make a hash object
   h = hashlib.sha1()

   # open file for reading in binary mode
   with open(filename,'rb') as file:

       # loop till the end of the file
       chunk = 0
       while chunk != b'':
           # read only 1024 bytes at a time
           chunk = file.read(1024)
           h.update(chunk)

   # return the hex representation of digest
   return h.hexdigest()

message = hash_file("track1.mp3")
print(message)
```

Output

```
633d7356947eec543c50b76a1852f92427f4dca9
```

In this program, we open the file in binary mode. Hash functions are available in the `hashlib` module. We loop till the end of the file using a `while` loop. On reaching the end, we get empty bytes object. In each iteration we only read 1024 bytes (this value can be changed according to our wish) from the file and update the hashing function. Finally, we return the digest message in hexadecimal representation using the `hexdigest()` method.