One of the newest and most innovative scripting languages is Python, developed by Guido van Rossum in the mid-90s. Python is named after the BBC “Monty Python” television series.

Python blends the expressive power and flexibility of earlier scripting languages with the power of object-oriented programming languages.

It offers a lot to programmers:

• An interactive development mode as well as an executable “batch” mode for completed programs.

• Very reasonable execution speed. Like Java, Python programs are compiled. Also like Java, the compiled code is in
an intermediate language for which an interpreter is written. Like Java this insulates Python from many of the vagaries of the actual machines on which it runs, giving it portability of an equivalent level to that of Java. Unlike Java, Python retains the interactivity for which interpreters are highly prized.

- Python programs require no compilation or linking. Nevertheless, the semi-compiled Python program still runs much faster than its traditionally interpreted rivals such as the shells, awk and perl.

- Python is freely available on almost all platforms and operating systems (Unix, Linux, Windows, MacOs, etc.)
• Python is completely object oriented. It comes with a full set of objected oriented features.

• Python presents a first class object model with first class functions and multiple inheritance. Also included are classes, modules, exceptions and late (run-time) binding.

• Python allows a clean and open program layout. Python code is less cluttered with the syntactic “noise” of declarations and scope definitions. Scope in a Python program is defined by the indentation of the code in question. Python thus breaks with current language designs in that white space has now once again acquired significance.
• Like Java, Python offers automated memory management through runtime reference counting and garbage collection of unreferenced objects.

• Python can be embedded in other products and programs as a control language.

• Python’s interface is well exposed and is reasonably small and simple.

• Python’s license is truly public. Python programs can be used or sold without copyright restrictions.

• Python is extendable. You can dynamically load compiled Python, Python source, or even dynamically load new machine (object) code to provide new features and new facilities.
• Python allows low-level access to its interpreter. It exposes its internal plumbing to a significant degree to allow programs to make use of the way the plumbing works.

• Python has a rich set of external library services available. This includes, network services, a GUI API (based on tcl/Tk), Web support for the generation of HTML and the CGI interfaces, direct access to databases, etc.
Using Python

Python may be used in either interactive or batch mode.

In interactive mode you start up the Python interpreter and enter executable statements. Just naming a variable (a trivial expression) evaluates it and echoes its value.

For example (>>> is the Python interactive prompt):

```python
>>> 1
1
>>> a=1
>>> a
1
>>> b=2.5
>>> b
2.5
```
>>> a+b
3.5
>>> print a+b
3.5

You can also incorporate Python statements into a file and execute them in batch mode. One way to do this is to enter the command

define the file.py

where file.py contains the Python code you want executed. Be careful though; in batch mode you must use a \texttt{print} (or some other output statement) to force output to be printed. Thus

\begin{verbatim}
1
a=1
a
\end{verbatim}
b=2.5
b
a+b
print a+b
when run in batch mode prints only
3.5 (the output of the print statement).

You can also run Python programs as
Unix shell scripts by adding the line
#! /usr/bin/env python
to the head of your Python file.
(Since # begins Python comments,
you can also feed the same
augmented file directly to the Python interpreter)
**Python Command Format**

In Python, individual primitive commands and expressions must appear on a single line.

This means that

\[
a = 1 + b
\]

does not assign \(1 + b\) to \(a\)! Rather, it assigns \(1\) to \(a\), then evaluates \(+b\).

If you wish to span more than one line, you must use \\ to escape the line:

\[
a = 1 \ \backslash \ 
   +b
\]

is equivalent to

\[
a = 1 + b
\]
Compound statements, like *if* statements and *while* loops, can span multiple lines, but individual statements within an *if* or *while* (if they are primitive) must appear on a single line.

Why this restriction?
With it, ;’s are mostly unnecessary!
A ; at the end of a statement is legal but usually unnecessary, as the end-of-line forces the statement to end.
You can use a ; to squeeze more than one statement onto a line, if you wish:

```
a=1; b=2 ; c=3
```
Identifiers and Reserved Words

Identifiers look much the same as in most programming languages. They are composed of letters, digits and underscores. Identifiers must begin with a letter or underscore. Case is significant. As in C and C++, identifiers that begin with an underscore often have special meaning.

Python contains a fairly typical set of reserved words:

<table>
<thead>
<tr>
<th>and</th>
<th>del</th>
<th>for</th>
<th>is</th>
<th>raise</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert</td>
<td>elif</td>
<td>from</td>
<td>lambda</td>
<td>return</td>
</tr>
<tr>
<td>break</td>
<td>else</td>
<td>global</td>
<td>not</td>
<td>try</td>
</tr>
<tr>
<td>class</td>
<td>except</td>
<td>if</td>
<td>or</td>
<td>while</td>
</tr>
<tr>
<td>continue</td>
<td>exec</td>
<td>import</td>
<td>pass</td>
<td></td>
</tr>
<tr>
<td>def</td>
<td>finally</td>
<td>in</td>
<td>print</td>
<td></td>
</tr>
</tbody>
</table>
Numeric Types

There are four numeric types:

1. Integers, represented as a 32 bit (or longer) quantity. Digits sequences (possibly) signed are integer literals:
   1   -123   +456

2. Long integers, of unlimited precision. An integer literal followed by an \( l \) or \( L \) is a long integer literal:
   1234567890000000000000L

3. Floating point values, represented as a 64 bit floating point number. Literals are of fixed decimal or exponential form:
   123.456   1e10   6.0231023

4. Complex numbers, represented as a pair of floating point numbers. In complex literals \( j \) or \( J \) is used to
denote the imaginary part of the complex value:

\[ 1.0+2.0j \quad -22.1j \quad 10e10J+20.0 \]

There is no character type. A literal like 'a' or "c" denotes a string of length one.

There is no boolean type. A zero numeric value (any form), or None (the equivalent of void) or an empty string, list, tuple or dictionary is treated as false; other values are treated as true.

Hence

"abc" and "def"

is treated as true in an \texttt{if}, since both strings are non-empty.
### Arithmetic Operators

<table>
<thead>
<tr>
<th>Op</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>-</td>
<td>Unary plus</td>
</tr>
<tr>
<td>+</td>
<td>Unary minus</td>
</tr>
<tr>
<td>~</td>
<td>Bit-wise complement (int or long only)</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
</tr>
<tr>
<td>-</td>
<td>Binary plus</td>
</tr>
<tr>
<td>+</td>
<td>Binary minus</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Bit-wise left shift (int or long only)</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Bit-wise right shift (int or long only)</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bit-wise and (int or long only)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bit-wise Xor (int or long only)</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
</tbody>
</table>
\[\geq\quad \text{Greater than or equal}\]
\[\leq\quad \text{Less than or equal}\]
\[==\quad \text{Equal}\]
\[!=\quad \text{Not equal}\]
\[\text{and}\quad \text{Boolean and}\]
\[\text{or}\quad \text{Boolean or}\]
\[\text{not}\quad \text{Boolean not}\]
Operator Precedence Levels

Listed from lowest to highest:

or Boolean OR
and Boolean AND
not Boolean NOT
<, <=, >, >=, !, == Comparisons
| Bitwise OR
^ Bitwise XOR
& Bitwise AND
<<, >> Shifts
+, - Addition and subtraction
*, /, % Multiplication, division, remainder
** Exponentiation
+, - Positive, negative (unary)
~ Bitwise not
**Arithmetic Operator Use**

Arithmetic operators may be used with any arithmetic type, with conversions automatically applied. Bit-wise operations are restricted to integers and long integers. The result type is determined by the “generality” of the operands. (Long is more general than int, float is more general than both int and long, complex is the most general numeric type). Thus

```
>>> 1+2
3
>>> 1+111L
112L
>>> 1+1.1
2.1
>>> 1+2.0j
(1+2j)
```
Unlike almost all other programming languages, relational operators may be “chained” (as in standard mathematics).

Therefore

\[ a > b > c \]

means \((a > b) \) and \((b > c)\)
Assignment Statements

In Python assignment is a statement not an expression.

Thus

\[ a + (b=2) \]

is illegal.

Chained assignments are allowed:

\[ a = b = 3 \]

Since Python is dynamically typed, the type (and value) associated with an identifier can change because of an assignment:

```python
>>> a = 0
>>> print a
0
>>> a = a + 0L
>>> print a
```

In

```python
>>> a = a + 0.0
>>> print a
0.0
>>> a = a + 0.0j
>>> print a
0j
```
If and While Statements

Python contains *if* and *while* statements that are fairly similar to those found in C and Java.

There are some significant differences though.

A line that contains an *if, else* or *while* ends in a "": Thus we might write:

```python
if a > 0:
    b = 1
```

Moreover the indentation of the then part is significant! You don’t need `{ and } in Python because all statements indented at the same level are assumed to be part of the same block.
In the following Python statements

```python
if a>0:
    b=1
    c=2

d=3
```

the assignments to \( b \) and \( c \) constitute then part; the assignment to \( d \) follows the if statement, and is independent of it. In interactive mode a blank line is needed to allow the interpreter to determine where the `if` statement ends; this blank line is not needed in batch mode.
The if Statement

The full form of the `if` statement is

```python
if expression:
    statement(s)
elif expression:
    statement(s)
...
else:
    statement(s)
```

Note those pesky `:`'s at the end of the `if`, `elif` and `else` lines. The expressions following the `if` and optional `elif` lines are evaluated until one evaluates to true. Then the following statement(s), delimited by indentation, are executed. If no expression evaluates to true, the statements following the `else` are executed.
Use of `else` and `elif` are optional; a “bare” `if` may be used.

If any of the lists of statements is to be null, use `pass` to indicate that nothing is to be done.

For example

```python
if a>0:
    b=1
elif a < 0:
    pass
else
    b=0
```

This `if` sets `b` to 1 if `a` is > 0; it sets `b` to 0 if `a` == 0, and does nothing if `a` < 0.
**While Loops**

Python contains a fairly conventional **while** loop:

```python
while expression:
    body
```

Note the “:” that ends the header line. Also, indentation delimits the body of the loop; no braces are needed. For example,

```python
>>> a=0; b=0
>>> while a < 5:
...     b = b+a**2
...     a= a+1
...
>>> print a,b
5 30
```
Break, Continue and Else in Loops

Like C, C++ and Java, Python allows use of break within a loop to force loop termination. For example,

```python
>>> a=1
>>> while a < 10:
...   if a+a == a**2:
...     break
...   else:
...     a=a+1
... print a
2
```
A `continue` may be used to force the next loop iteration:

```python
>>> a=1
>>> while a < 100:
...     a=a+1
...     if a%2==0:
...         continue
...     a=3*a
...

>>> print a
105
```
Python also allows you to add an else clause to a while (or for) loop.
The syntax is

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

The else statements are executed when the termination condition becomes false, but not when the loop is terminated with a break. As a result, you can readily program “search loops” that need to handle the special case of search failure:
>>> a=1
>>> while a < 1000:
...     if a**2 == 3*a-1:
...         print "winner: ",a
...         break
...     a=a+1
... else:
...     print "No match"
...
No match
Sequence Types

Python includes three sequence types: strings, tuples and lists. All sequence types may be indexed, using a very general indexing system.

Strings are sequences of characters; tuples and lists may contain any type or combination of types (like Scheme lists).

Strings and tuples are immutable (their components may not be changed). Lists are mutable, and be updated, much like arrays.

Strings may be delimited by either a single quote (') or a double quote (") or even a triple quote (''') or """". A given string must start and stop with the same delimiter. Triply quoted strings may span multiple lines. There
is no character type or value; characters are simply strings of length 1. Legal strings include

'abc' "xyz" '''It's OK!'''

Lists are delimited by "[" and "]". Empty (or null lists) are allowed. Valid list literals include

[1,2,3] ["one",1]
[['a'],['b'],['c']] []

Tuples are a sequence of values separated by commas. A tuple may be enclosed within parentheses, but this isn’t required. A empty tuple is (). A singleton tuple ends with a comma (to distinguish it from a simple scalar value).

Thus (1,) or just 1, is a valid tuple of length one.
Indexing Sequence Types

Python provides a very general and powerful indexing mechanism. An index is enclosed in brackets, just like a subscript in C or Java. Indexing starts at 0.

Thus we may have

```python
>>> 'abcde'[2]
'c'
>>> [1,2,3,4,5][1]
2
>>> (1.1,2.2,3.3)[0]
1.1
```

Using an index that’s too big raises an `IndexError` exception:

```python
>>> 'abc'[3]
IndexError: string index out of range
```
Unlike most languages, you can use negative index values; these simply index from the right:

```python
>>> 'abc'[-1]
'c'
>>> [5, 4, 3, 2, 1][-2]
2
>>> (1, 2, 3, 4)[-4]
1
```

You may also access a slice of a sequence value by supplying a range of index values. The notation is

```
data[i:j]
```

which selects the values in `data` that are $\geq i$ and $< j$. Thus

```python
>>> 'abcde'[1:2]
'b'
>>> 'abcde'[0:3]
'abc'
```
You may omit a lower or upper bound on a range. A missing lower bound defaults to 0 and a missing upper bound defaults to the maximum legal index. For example,

```
>>> [1, 2, 3, 4, 5][2:]
[3, 4, 5]
>>> [1, 2, 3, 4, 5][:3]
[1, 2, 3]
```

An upper bound that’s too large in a range is interpreted as the maximum legal index:

```
>>> 'abcdef'[3:100]
'def'
```

You may use negative values in ranges too—they’re interpreted as being relative to the right end of the sequence:
>>> 'abcde'[0:-2]
'abc'
>>> 'abcdefg'[-5:-2]
'cde'
>>> 'abcde'[-3:]
'cde'
>>> 'abcde'[: -1]
'abcd'

Since arrays may be assigned to, you may assign a slice to change several values at once:

>>> a=[1,2,3,4]
>>> a[0:2]=[-1,-2]
>>> a
[-1, -2, 3, 4]
>>> a[2:]=[33,44]
>>> a
[-1, -2, 33, 44]
The length of the value assigned to a slice need not be the same size as the slice itself, so you can shrink or expand a list by assigning slices:

```python
>>> a = [1, 2, 3, 4, 5]
>>> a[2:3] = [3.1, 3.2]
>>> a
[1, 2, 3.1, 3.2, 4, 5]
>>> a[4:] = []
>>> a
[1, 2, 3.1, 3.2]
>>> a[:0] = [-3, -2, -1]
>>> a
[-3, -2, -1, 1, 2, 3.1, 3.2]
```
Other Operations on Sequences

Besides indexing and slicing, a number of other useful operations are provided for sequence types (strings, lists and tuples).

These include:

+ (catenation):

```python
>>> [1,2,3]+[4,5,6]
[1, 2, 3, 4, 5, 6]
>>> (1,2,3)+(4,5)
(1, 2, 3, 4, 5)
>>> (1,2,3)+[4,5]
TypeError: illegal argument type for built-in operation
>>> "abc"+"def"
'abcdef'
```
• * (Repetition):
  
  ```python
  >>> 'abc'*2
  'abcabc'
  >>> [3,4,5]*3
  [3, 4, 5, 3, 4, 5, 3, 4, 5]
  ```

• Membership (`in`, `not in`)
  
  ```python
  >>> 3 in [1,2,3,4]
  1
  >>> 'c' in 'abcde'
  1
  ```

• `max` and `min`:
  
  ```python
  >>> max([3,8,-9,22,4])
  22
  >>> min('aa','bb','abc')
  'aa'
  ```
Operations on Lists

As well as the operations available for all sequence types (including lists), there are many other useful operations available for lists. These include:

- **count** (Count occurrences of an item in a list):
  ```python
  >>> [1, 2, 3, 3, 21].count(3)
  2
  ```

- **index** (Find first occurrence of an item in a list):
  ```python
  >>> [1, 2, 3, 3, 21].index(3)
  2
  >>> [1, 2, 3, 3, 21].index(17)
  ValueError: list.index(x): x not in list
• **remove** (Find and remove an item from a list):
  ```python
  >>> a=[1,2,3,4,5]
  >>> a.remove(4)
  >>> a
  [1, 2, 3, 5]
  >>> a.remove(17)
  ValueError: list.remove(x): x not in list
  ```

• **pop** (Fetch and remove i-th element of a list):
  ```python
  >>> a=[1,2,3,4,5]
  >>> a.pop(3)
  4
  >>> a
  [1, 2, 3, 5]
  >>> a.pop()
  5
  >>> a
  [1, 2, 3]
  ```
• reverse a list:
  >>> a=[1,2,3,4,5]
  >>> a.reverse()
  >>> a
  [5, 4, 3, 2, 1]

• sort a list:
  >>> a=[5,1,4,2,3]
  >>> a.sort()
  >>> a
  [1, 2, 3, 4, 5]

• Create a range of values:
  >>> range(1,5)
  [1, 2, 3, 4]
  >>> range(1,10,2)
  [1, 3, 5, 7, 9]
Dictionaries

Python also provides a dictionary type (sometimes called an associative array). In a dictionary you can use a number (including a float or complex), string or tuple as an index. In fact any immutable type can be an index (this excludes lists and dictionaries).

An empty dictionary is denoted `{ }`. A non-empty dictionary may be written as

`{ key1:value1, key2:value2, ... }`

For example,

`c={ 'bmw':545, 'lexus':'sc 430', 'mercedes':'S 500'}`
You can use a dictionary much like an array, indexing it using keys, and updating it by assigning a new value to a key:

```python
>>> c['bmw']
545
>>> c['bmw']='m5'
>>> c['honda']='accord'

You can delete a value using `del`:

```python
>>> del c['honda']
>>> c['honda']
KeyError: honda
```
You can also check to see if a given key is valid, and also list all keys, values, or key-value pairs in use:

```python
>>> c.has_key('edsel')
0

>>> c.keys()
['bmw', 'mercedes', 'lexus']

>>> c.values()
['m5', 'S 500', 'sc 430']

>>> c.items()
[('bmw', 'm5'), ('mercedes', 'S 500'), ('lexus', 'sc 430')]
```
For Loops

In Python’s for loops, you don’t explicitly control the steps of an iteration. Instead, you provide a sequence type (a string, list or sequence), and Python automatically steps through the values.

Like a while loop, you must end the for loop header with a “:” and the body is delimited using indentation. For example,

```python
>>> for c in 'abc':
    ...   print c
... ...

a
b
c
```
The `range` function, which creates a list of values in a fixed range is useful in for loops:

```python
>>> a=[5,2,1,4]
>>> for i in range(0,len(a)):
...     a[i]=2*a[i]
...
>>> print a
[10, 4, 2, 8]
```
You can use an `else` with `for` loops too. Once the values in the specified sequence are exhausted, the `else` is executed unless the `for` is exited using a `break`. For example,

```python
for i in a:
    if i < 0:
        print 'Neg val:', i
        break
else:
    print 'No neg vals'
```