>>> '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):
  >>> [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'
```

```python
>>> [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
```

```python
>>> 'c' in 'abcde'
1
```

• max and min:

```python
>>> max([3,8,-9,22,4])
22
```

```python
>>> 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):

  >>> [1,2,3,3,21].count(3)
  2

- `index` (Find first occurrence of an item in a list):

  >>> [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:

```python
>>> a=[1,2,3,4,5]
>>> a.reverse()
>>> a
[5, 4, 3, 2, 1]
```

• sort a list:

```python
>>> a=[5,1,4,2,3]
>>> a.sort()
>>> a
[1, 2, 3, 4, 5]
```

• Create a range of values:

```python
>>> 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

\{ key_1:value_1, key_2:value_2, ... \}

For example,

\c = \{ 'bmw':650, 'lexus':'sc 430', 'mercedes':'SL 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']
650
>>> c['bmw'] = 'z4 coupe'
>>> 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()
['z4 coupe', 'SL 500', 'sc 430']
>>> c.items()
[('bmw', 'z4 coupe'),
 ('mercedes', 'SL 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'
```
Function Definitions

Function definitions are of the form
\[
def \text{name}(\text{args}): \\
\text{body}
\]
The symbol \text{def} tells Python that a function is to be defined. The function is called \text{name} and \text{args} is a tuple defining the names of the function’s arguments. The \text{body} of the function is delimited using indentation. For example,

\[
def \text{fact}(n):
    \text{if } n \leq 1:
        \text{return } 1
    \text{else:}
        \text{return } n * \text{fact}(n-1)
\]

\[
>>> \text{fact}(5)
120
>>> \text{fact}(20L)
\]
Scalar parameters are passed by value; mutable objects are allocated in the heap and hence are passed (in effect) by reference:

```python
>>> def asg(ar):
...     a[1]=0
...     print ar
...     ...

>>> a=[1,2,3,4.5]
>>> asg(a)
[1, 0, 3, 4.5]
```
Arguments may be given a default value, making them optional in a call. Optional parameters must follow required parameters in definitions. For example,

```python
>>> def expo(val, exp=2):
...     return val**exp
... 
>>> expo(3, 3)
27
>>> expo(3)
9
>>> expo()
TypeError: not enough arguments; expected 1, got 0
```
A variable number of arguments is allowed; you prefix the last formal parameter with a *; this parameter is bound to a tuple containing all the actual parameters provided by the caller:

```python
>>> def sum(*args):
...     sum=0
...     for i in args:
...         sum=sum+i
...     return sum
... 
>>> sum(1,2,3)
6
>>> sum(2)
2
>>> sum()
0
```
You may also use the name of formal parameters in a call, making the order of parameters less important:

```python
>>> def cat(left="[",body="", right="]"):
...     return left+body+right
...

>>> cat(body='xyz');
' [xyz]

>>> cat(body='hi there!', left='--[')
'--[hi there!]
```
Scoping Rules in Functions

Each function body has its own local namespace during execution. An identifier is resolved (if possible) in the local namespace, then (if necessary) in the global namespace.

Thus

```python
>>> def f():
...     a=11
...     return a+b
...

>>> b=2; f()
13
>>> a=22; f()
13
>>> b=33; f()
44
```
Assignments are to local variables, even if a global exists. To force an assignment to refer to a global identifier, you use the declaration

```python
global id
```

which tells Python that in this function `id` should be considered global rather than local. For example,

```python
>>> a=1;b=2
>>> def f():
... global a
... a=111;b=222
... >>> f();print a,b
111 2
```
Other Operations on Functions

Since Python is interpreted, you can dynamically create and execute Python code.

The function `eval(string)` interprets `string` as a Python expression (in the current execution environment) and returns what is computed. For example,

```python
>>> a=1; b=2
>>> eval('a+b')
3
```
exec(string) executes string as arbitrary Python code (in the current environment):

```python
>>> a=1; b=2
>>> exec('for op in "+-*/": print(eval("a"+op+"b"))')
3
-1
2
0
```

eexecfile(string) executes the contents of the file whose pathname is specified by string. This can be useful in loading an existing set of Python definitions.
The expression

\texttt{lambda \text{args}: \text{expression}}

creates an anonymous function with \text{args} as its argument list and \text{expression} as its body. For example,

\begin{verbatim}
>>> (lambda \text{a}:\text{a+1})(2)
3
\end{verbatim}

And there are definitions of \texttt{map}, \texttt{reduce} and \texttt{filter} to map a function to a list of values, to reduce a list (using a binary function) and to select values from a list (using a predicate):

\begin{verbatim}
>>> def double(a):
...     return 2*a;
...     
>>> map(double, [1,2,3,4])
[2, 4, 6, 8]
\end{verbatim}
```python
>>> def sum(a, b):
...     return a + b
...

>>> reduce(sum, [1, 2, 3, 4, 5])
15

>>> def even(a):
...     return not(a % 2)
...

>>> filter(even, [1, 2, 3, 4, 5])
[2, 4]
```
I/O in Python

The easiest way to print information in Python is the print statement. You supply a list of values separated by commas. Values are converted to strings (using the str() function) and printed to standard out, with a terminating new line automatically included. For example,

```
>>> print "1+1="",1+1
1+1= 2
```

If you don’t want the automatic end of line, add a comma to the end of the print list:

```
>>> for i in range(1,11):
...    print i,
...
1 2 3 4 5 6 7 8 9 10
```
For those who love C’s printf, Python provides a nice formatting capability using a printf-like notation. The expression

```
format % tuple
```

formats a tuple of values using a format string. The detailed formatting rules are those of C’s printf. Thus

```python
>>> print "%d+%d=%d" % (10, 20, 10+20)
10+20=30
```
**File-oriented I/O**

You open a file using

```python
open(name, mode)
```

which returns a “file object.”

*name* is a string representing the file’s path name; *mode* is a string representing the desired access mode (‘r’ for read, ‘w’ for write, etc.).

Thus

```python
>>> f=open("/tmp/f1","w");
>>> f
<open file '/tmp/f1', mode 'w' at decd8>
```

opens a temp file for writing.

The command

```python
f.read(n)
```

reads n bytes (as a string).
f.read() reads the whole file into a string. At end-of-file, f.read returns the null string:

```python
>>> f = open("/tmp/ttt", "r")
>>> f.read(3)
'aaa'
>>> f.read(5)
' bbb'
>>> f.read()
'ccc\012ddd eee fff\012g h i\012'
>>> f.read()
''
```

f.readline() reads a whole line of input, and f.readlines() reads the whole input file into a list of strings:

```python
>>> f = open("/tmp/ttt", "r")
>>> f.readline()
'aaa bbb ccc\012'
>>> f.readline()
''
```
'ddd eee fff\012'
>>> f.readline()
'g h i\012'
>>> f.readline()
''

>>> f = open("/tmp/ttt","r")
>>> f.readlines()
['aaa bbb ccc\012', 'ddd eee fff\012', 'g h i\012']

f.write(string) writes a string to file object f; f.close() closes a file object:

>>> f = open("/tmp/ttt","w")
>>> f.write("abcd")
>>> f.write("%d %d"%(1,-1))
>>> f.close()

>>> f = open("/tmp/ttt","r")
>>> f.readlines()
['abcd1   -1']
Classes in Python

Python contains a class creation mechanism that’s fairly similar to what’s found in C++ or Java. There are significant differences though:

- All class members are public.

- Instance fields aren’t declared. Rather, you just create fields as needed by assignment (often in constructors).

- There are class fields (shared by all class instances), but there are no class methods. That is, all methods are instance methods.
All instance methods (including constructors) must explicitly provide an initial parameter that represents the object instance. This parameter is typically called `self`. It’s roughly the equivalent of `this` in C++ or Java.
Defining Classes

You define a class by executing a class definition of the form

```plaintext
class name:
    statement(s)
```

A class definition creates a class object from which class instances may be created (just like in Java). The statements within a class definition may be data members (to be shared among all class instances) as well as function definitions (prefixed by a `def` command). Each function must take (at least) an initial parameter that represents the class instance within which the function (instance method) will operate. For example,
class Example:
    cnt=1
    def msg(self):
        print "Bo"+"o"*Example.cnt+"!"*self.n

>>> Example.cnt
1
>>> Example.msg
<unbound method Example.msg>

Example.msg is unbound because we haven’t created any instances of the Example class yet.

We create class instances by using the class name as a function:
>>> e=Example()
>>> e.msg()
AttributeError: n

We get the AttributeError message regarding n because we haven’t defined n yet! One way to do
this is to just assign to it, using the usual field notation:

```python
>>> e.n=1
>>> e.msg()
Boo!
>>> e.n=2;Example.cnt=2
>>> e.msg()
Booo!!
```

We can also call an instance method by making the class object an explicit parameter:

```python
>>> Example.msg(e)
Booo!!
```

It's nice to have data members initialized when an object is created. This is usually done with a constructor, and Python allows this too.
A special method named `__init__` is called whenever an object is created. This method takes `self` as its first parameter; other parameters (possibly made optional) are allowed.

We can therefore extend our `Example` class with a constructor:

```python
class Example:
    cnt=1
    def __init__(self, nval=1):
        self.n=nval
    def msg(self):
        print "Bo"+"o"*Example.cnt+"!"*self.n

>>> e=Example()
>>> e.n
1
>>> f=Example(2)
>>> f.n
2
```
You can also define the equivalent of Java’s `toString` method by defining a member function named `__str__(self)`. For example, if we add

```python
def __str__(self):
    return "<%d>"%self.n
```

to `Example`, then we can include `Example` objects in `print` statements:

```python
>>> e=Example(2)
>>> print e
<2>
```
Inheritance

Like any language that supports classes, Python allows inheritance from a parent (or base) class. In fact, Python allows multiple inheritance in which a class inherits definitions from more than one parent.

When defining a class you specify parents classes as follows:

```python
class name(parent classes):
    statement(s)
```

The subclass has access to its own definitions as well as those available to its parents. All methods are virtual, so the most recent definition of a method is always used.
class C:
    def DoIt(self):
        self.PrintIt()
    def PrintIt(self):
        print "C rules!"

class D(C):
    def PrintIt(self):
        print "D rules!"
    def TestIt(self):
        self.DoIt()

dvar = D()
dvar.TestIt()

D rules!
If you specify more than one parent for a class, lookup is depth-first, left to right, in the list of parents provided. For example, given

```python
class A(B, C): ...
```

we first look for a non-local definition in \( B \) (and its parents), then in \( C \) (and its parents).
Operator Overloading

You can overload definitions of all of Python’s operators to apply to newly defined classes. Each operator has a corresponding method name assigned to it. For example, + uses __add__, - uses __sub__, etc.
Given

class Triple:
    def __init__(self, A=0, B=0, C=0):
        self.a = A
        self.b = B
        self.c = C
    def __str__(self):
        return "(%d, %d, %d)" % (self.a, self.b, self.c)
    def __add__(self, other):
        return Triple(self.a + other.a,
                      self.b + other.b,
                      self.c + other.c)

the following code

t1 = Triple(1, 2, 3)
t2 = Triple(4, 5, 6)
print t1 + t2

produces

(5, 7, 9)
Exceptions

Python provides an exception mechanism that’s quite similar to the one used by Java.

You “throw” an exception by using a `raise` statement:

```python
raise exceptionValue
```

There are numerous predefined exceptions, including `OverflowError` (arithmetic overflow), `EOFError` (when end-of-file is hit), `NameError` (when an undeclared identifier is referenced), etc.
You may define your own exceptions as subclasses of the predefined class Exception:

class badValue(Exception):
    def __init__(self, val):
        self.value = val

You catch exceptions in Python’s version of a try statement:

try:
    statement(s)
except exceptName$_1$, id$_1$:
    statement(s)
...
except exceptName$_n$, id$_n$:
    statement(s)

As was the case in Java, an exception raised within the try body is handled by an except clause if the raised exception matches the class named in
the except clause. If the raised exception is not matched by any except clause, the next enclosing try is considered, or the exception is reraised at the point of call.

For example, using our badValue exception class,

```python
def sqrt(val):
    if val < 0.0:
        raise badValue(val)
    else:
        return cmath.sqrt(val)
```

```python
try:
    print "Ans =", sqrt(-123.0)
except badValue,b:
    print "Can’t take sqrt of ",
    b.value
```

**When executed, we get**

Ans = Can’t take sqrt of -123.0
Python contains a module feature that allows you to access Python code stored in files or libraries. If you have a source file `mydefs.py` the command

```python
import mydefs
```

will read in all the definitions stored in the file. What’s read in can be seen by executing

```python
dir(mydefs)
```

To access an imported definition, you qualify it with the name of the module. For example,

```python
mydefs.fct
```

accesses `fct` which is defined in module `mydefs`. 
To avoid explicit qualification you can use the command

```
from modulename import id_1, id_2, ...
```

This makes \( id_1, id_2, \ldots \) available without qualification. For example,

```
>>> from test import sqrt
>>> sqrt(123)
(11.0905365064+0j)
```

You can use the command

```
from modulename import *
```

to import (without qualification) all the definitions in modulename.
The Python Library

One of the great strengths of Python is that it contains a vast number of modules (at least several hundred) known collectively as the Python Library. What makes Python really useful is the range of prewritten modules you can access. Included are network access modules, multimedia utilities, data base access, and much more.

See

www.python.org/doc/lib

for an up-to-date listing of what’s available.