

# 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

```
>>> print "%d+%d=%d" %  
      (10,20,10+20)  
10+20=30
```

# File-oriented I/O

You open a file using

`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

```
>>> f=open("/tmp/f1", "w");
```

```
>>> f
```

```
<open file '/tmp/f1', mode 'w' at  
dec8>
```

opens a temp file for writing.

The command

```
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:

```
>>> 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:

```
>>> 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

```
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:

```
>>> 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:

```
>>> 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:

```
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

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

to `Example`,

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

```
>>> 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:

```
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

```
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:

```
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 exceptionName1, id1:  
    statement(s)  
...  
except exceptionNamen, idn:  
    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,

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

# Modules

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

```
import mydefs
```

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

```
dir(mydefs)
```

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

```
mydefs.fct
```

accesses `fct` which is defined in module `mydefs`.

To avoid explicit qualification you can use the command

```
from modulename import id1, id2,  
...
```

This makes `id1`, `id2`, ... 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.