We may make statements regarding planned or future development efforts for our existing or new products and services. These statements are not intended to be a promise or guarantee of future delivery of products, services or features but merely reflect our current plans, which may change. Purchasing decisions should not be made based upon reliance on these statements.

The Company assumes no obligation to update these forward-looking statements to reflect events that occur or circumstances that exist or change after the date on which they were made.

Please note that at the time of printing the Maya portion of this session's documentation was unavailable. Please see the insert for this material.

A Python Introduction to the Maya API.
Habib Chahin, Autodesk Consulting,

The goal of the seminar is to get people up-to-speed in Python and start cranking Maya plug-ins right away. :-) Ok may be not that fast, but I want people to go home with a road map and a good starting point to delve into the Maya API using Python programming.

The intended audience, are people who know at least one programming language and are familiar with Maya.

1. I will introduce python with quick overview of its syntax, and highlight class structure and exception handling.
I will also be talking about the import statement and how to find what modules to use. I will focus on the
xml module in python since it is a popular module and many people would like to understand it. I will be talking
about an open source tool to make python a little more secure.

The addition of Python to Maya is a natural progression of Maya, given the openness and flexibility of its API. The addition of Python to Maya opens the scope of the Maya API development to almost limitless possibilities, within a reasonable timeframe.

Part 1. Python Basics

Overview

In order to understand Python, I listed below the important concepts and ideas that related to Maya development. It is meant to guide in learning the language, for people who already know another language. It is by no means a comprehensive primer of the language, however, I wanted to highlight the similarities and differences with other languages. So, I provided detail where I thought it was needed. Again the assumption is to get somebody who already knows how to program up to speed into learning python.

Python is a great (scripting) programming language. It is easy to learn, has a broad and powerful standard library, and benefits from an active community of developers who maintain a range of XML and database tools, as well as servers, and application frameworks. Python is an interpreted, interactive, and object-oriented programming language. It runs on Windows, Linux/Unix, Mac OS X, OS/2, Palm Handhelds, and Nokia mobile phones.

The main resource for python on the web is http://www.python.org

Python is loaded with "batteries included": Describes the standard library (http://docs.python.org/lib/lib.html), which covers everything from asynchronous processing to zip files. The language itself is a flexible powerhouse that can handle practically any problem domain. Build your own web server in three lines of code. Build flexible data-driven code using Python's powerful and dynamic introspection capabilities and advanced language features such as meta-classes, duck typing and decorators.

Installation

If you are unfamiliar with Python or even unfamiliar with programming take a look at

http://wiki.python.org/moin/BeginnersGuide/Programmers

However the best introduction to the language is written by its Guido Van Rossum, the person who developed Python. His tutorial http://docs.python.org/tut/tut.html is a must for any person wishing to learn the language.

For an advanced treatment take a look at http://diveintopython.org/

The only book I would recommend is "Python in a Nutshell" from O'reilly publishing.

Prior to using Python within Maya, I highly recommend downloading a Python interpreter with the same version as the one that is embedded in Maya. That way you can eliminate incompatibilities from the start. To find the correct version of the interpreter, launch Maya, click on the script editor, and then click on the python tab.

Type the following lines in the command window:

```
import sys

print sys.version
```

highlight them , the hit CTRL-ENTER.

You should get something like

```
2.5.1 (r251:54863, Jun  5 2007, 22:56:07) [MSC v.1400 32 bit (Intel)]
```

In this case the version of the python interpreter is 2.5.1 32 bit interpreter.

In order to download the interpreter, visit the http://www.python.org/download

Here is a partial list from that page of the available versions.

This is a list of the standard releases, both source and Windows installers. Consider your needs carefully before using a version other than the current production version:

- Python 2.5.1 (April 18, 2007)
- Python 2.4.5 (March 11, 2008)
- Python 2.3.7 (March 11, 2008

Clicking on Python 2.5.1 will load the download page for that interprer

Here is an excerpt :

Windows

- For x86 processors: python-2.5.1.msi
- For Win64-Itanium users: python-2.5.1.ia64.msi
- For Win64-AMD64 users: python-2.5.1.amd64.msi
- This installer allows for automated installation and many other new features

Download and Install the msi that matches your platform.

For the remainder of this document, I will be talking about Python 2.5.1 and Maya 2008 extension 2 on a windows platform.

Python comes with a GUI editor called IDLE. It offers syntax highlighting and very well integrated with the interpreter. Another feature of the interpreter is that you can start a Python console shell, so that you can enter commands via the shell. The shell is useful for development, debugging, as well as general control of your system. Note that there is plethora of IDEs that are available to write python programs, however the IDLE seems to be good enough for all you need to write Python Maya plugins.

In order to configure the Python shell so that it can be invoked from the command shell in windows. Append the system path environment variable with the path to the Python interpreter directory. For example append c:\python25 to the path variable.

Another path you want to set is the PYTHONDOCS path, but prior to that, you to decompile python25.chm file to create the html docs. Go to c:\python25\doc directory and type hh –decompile . python25.chm and hit return. It will create html version of the Python documentation. Now set the PYTHONDOCS path to c:\python25\doc.

It goes without saying that the IDLE will be used to write programs as well as to debug them. Note that Python programs are usually written with ".py" extension. That way windows will identify it as a python executable. On a Linux system, you have to change to permission using chmod command to make it executable.

In order to understand the basics of Python it will be easier to start by using the shell in order to go over basics of the language.

Start a command prompt, and type python.

Type print “Welcome to Maya API conference” and hit enter

Note the output of the command appears below the line

The screen will look as follows:

```
>>> print "Welcome to Maya API conference"

Welcome to Maya API conference

>>>
```

To exit the python prompt, Press CTRL Z,followed by Enter.

You are back at the C: prompt. Type Python and enter to go back to the python environment.

The Elements and Data structures

Comments can be written by prefixing the statements with a ‘#’ sign. To find help on any function use the help command. Enter help(“command”) at the prompt sign.
A python program consists of a sequence of lines.

Python Tokens: or the elements of syntax: Identifiers, Keywords, Operators, Delimiters, and Literals. White spaces have special purpose when used to identify nested blocks. They are the equivalent of curly braces “{}” in C, C++, Perl, Java, C#, and Javascript etc…
So, the indentation at the beginning of every line is important. If a line has deeper indentation than the line above is the equivalent of a left brace '{' at the beginning of that line, which signals the beginning of a block. If a line has shallower indentation than the line which precedes it, is equivalent to right brace '}' at the beginning of that line and hence the ending of a block.

Identifiers are the names used to define a variable. Their syntax follows closely that of Java, C++, or C#. They are case sensitive. Here is a list of examples: _Node, sumX, total, first_name.

Keywords are identifiers which python reserves for special syntactic purposes. There are 30 keywords, all lower case. Here is the list of keywords:

and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for,

from, global, if, import, in, is, lambda, not ,or, pass, print, raise, return, try, while,

with, yield.

Operators are non alphanumeric characters used to define mathematical, grouping, and logical operations. Here is a list of the operators:

+ - * / % ** // << >> & | ^ ~ < <= > >= <> != ==

Delimeters used in expressions, lists, dictionaries and various statements.

( ) [ ] { } , : . ' = ; += -= *= /= //= %= &= |= ^= >>= <<= **=

Special Character such as ` "

#	(comment)
\	(escape character)
$	(template: advanced)
?	(ternary condition)
Whitespace	
@	(decorators)

Literals
are numbers and strings that appear directly in a program. For example: 32 (integers) , 12.19 (floating-point), 3 + 5j (complex numbers). 'hello world' (string) , "hello api" (string),

```
""" The is a multi line
    String in python. Note
    The triple double quotes at the beginning
     And the end of the string
"""
```

Data Types
Python uses dynamic typing to determine the type of the variables. Therefore you don't have to declare the type of a variable prior to using it. Types are determined by the context within which the variables are used.
for example, x = 5 will create a variable of type integer
where as x = [5] will create a list object, where one of its elements is an integer.
All data values in Python are objects, and alternately each object has a type. Each type determines the operation the object supports. Refer to the function type(obj) to determine the type of objects or data values. Here is a list of the basic types in Python:

Integer numbers, Floating-point numbers, Complex numbers, Boolean, and Iterators (Sequences). The first 4 types should be familiar, so we will focus on the fourth type.

Iterators
Here is where Python diverges from the traditional programming constructs and adds its own power and flavor. Even though other languages such as Perl may have implemented similar constructs, yet Python took that concept to newer heights by providing the developer with a programmatic construct called iterators.
An iterator is an abstract type to which all sequences belong, which provides a method next() in order to get to the consecutive element in the sequence. For example if you
Have a list I, you can write a while loop to iterate through the elements of the list.
I[0], I.next() = I[1], I.next() = [2] , …., I.next[m-1], I.next() = I[m]. If the size of the list is m+1 and you are at element I[m], then I.next() will raise an exception StopIteration .
This capability to iterate over objects, gives rise to more elegant coding and helps the developer focus on the problem domain of the project.

Sequences
Are ordered containers of items, indexed by nonnegative integers which allows them to be iterable. Python provides built-in sequences in the data types such as strings, tuples, lists, sets, and dictionaries. What is interesting is the commonality of operations (Thanks to polymorphism). Note that to find the number of elements in a sequence use the len function. Hence, len(S) returns the number of elements of sequence S.

Mutable/Immutable sequences
In order to understand Python better we need to understand the concept of mutable and immutable objects. Immutable objects are those attributes cannot be altered. While mutable objects allow such alterations. Therefore, when ever operate on immutable objects you create new objects, while operations on immutable objects will change the contents of that object. For example if you have a string object
x = "abc", which is also a sequence it cannot be altered by using an statement such as x[0] = d, however if you have a list l=[a,b,c], then you can use l[0] = d. So the only way to get a string "dabc" based on x is to create a new string x = "d" + x[1:]

Sequence Operations

Concatenation: you can concatenate two sequences of the same type by the + operator. For example if x = "abc" and y = "def" then x + y is "abcdef"

Indexing: same as array indexing in C, C#, Java. All sequences start at 0, and the last item is L-1, where L is the number of items in the list. To get the mth element of sequence you use same notation as other languages, S[m] where m is a positive ineger between 0, and L-1. Note that you can use negative indexes, such that if m is negative then S[m] equivalent to S[L+m]. For example S[-2] = S[L-2]. Note that if m >= L then IndexError exception will be generated.
Slicing: is a way to determine subsequence. To get subsequence S' of S you can use a statement like S'= S[i:j]. S' is a subsequence of S which has S[i] as its first element and S[j-1] as its last element. The rule in Python for slices is that it includes the lower bound and excludes the upper bound for the number. If the lower bound is not available then it is the same as i=0, if the upper bound is not available then it is the same as j >=L. Hence S' = S[:] is equivalent to copying S into S'.
Here are some examples of sequences: S = ['a','b','c','d','e','f']
S[2:4] = ['c','d']
S[:4] = ['a','b','c','d']

Membership: To test if an object x belongs to a sequence S, you can use the operator 'in' to test for membership. For example, ('a' in S) returns True while ('j' in S) returns False.

Built-in Sequences

Strings: are Immutable sequences of characters used to represent text. So,if you try to delete an item, replace, or delete a slice and exception will be raised.
Here are some examples: "Arizona is beautiful" 'I like Maya' "Michigan is the capital of the world!"

“Python is the coolest language, and I quote \”Python is the coolest language\””
‘Welcome to the world of Python It\’s a wonderful world’

Here is a list of the most common string functions:

s.count(sub,start,end) returns the number of non-overlapping occurrences of substring sub in s[start:end].

s.find(sub,start,end) returns the lowest index in s where substring sub is found, such that sub is entirely contained in s[start:end]. For example,'banana'.find('na') is 2, as is 'banana'.find('na',1), while 'banana'.find('na',3) is 4, as is 'banana'.find('na',-2). find returns -1 if sub is not found.

s.index(sub,start,end)Like find, but raises ValueError when sub is not found.

s.isalnum( ) returns true when len(s) is greater than 0 and all characters in s are letters or decimal digits. When s is empty, or when at least one character of s is neither a letter nor a decimal digit, isalnum returns False.

s.isalpha( )returns true when len(s) is greater than 0 and all characters in s are letters. When s is empty, or when at least one character of s is not a letter, isalpha returns False.

s.isdigit( ) returns true when len(s) is greater than 0 and all characters in s are digits. When s is empty, or when at least one character of s is not a digit, isdigit returns False.

s.islower( ) returns TRue when all letters in s are lowercase. When s contains no letters, or when at least one letter of s is uppercase, islower returns False.

s.isspace( ) returns true when len(s) is greater than 0 and all characters in s are whitespace. When s is empty, or when at least one character of s is not whitespace, isspace returns False.

s.isupper( ) returns true when all letters in s are uppercase. When s contains no letters, or when at least one letter of s is lowercase, isupper returns False.

s.join(seq) returns the string obtained by concatenating the items of seq, which must be a sequence or other iterable whose items are strings, and interposing a copy of s between each pair of items (e.g., ''.join(str(x) for x in range(7)) is '0123456' and 'x'.join('aeiou') is 'axexixoxu').

s.replace(old,new,maxsplit) returns a copy of s with the first maxsplit (or fewer, if there are fewer) nonoverlapping occurrences of substring old replaced by string new (e.g., 'banana'.replace('a','e',2) is 'benena').

s.split(sep,maxsplit) returns a list L of up to maxsplit+1 strings. Each item of L is a "word" from s, where string sep separates words. When s has more than maxsplit words, the last item of L is the substring of s that follows the first maxsplit words. When sep is None, any string of whitespace separates words (e.g., 'four score and seven years ago'.split(None,3) is ['four','score','and','seven years ago']).

Note the difference between splitting on None (any string of whitespace is a separator) and splitting on ' ' (each single space character, not other whitespace such as tabs and newlines, and not strings, is a separator). For example:

x = 'a b' # two spaces between a and b
x.split( ) will return ['a', 'b']
x.split(' ') will return ['a', '', 'b']

In the first case, the two-spaces string in the middle is a single separator; in the second case, each single space is a separator so that there is an empty string between the spaces.

Like s.split('\n'). When keepends is true, however, the trailing '\n' is included in each item of the resulting list.

s.strip(x) returns a copy of s, removing both leading and trailing characters that are found in string x.

Lists: are mutable ordered sequences of items. Lists are specified as comma separated list of items enclosed between two square brackets. For example: L = [1,'a',4,2]
An empty list is defined as follows: []

Here some of the common functions on List L:

L.count(x) returns the number of items of L that are equal to x.

L.index(x)returns the index of the first occurrence of an item in L that is equal to x, or raises an exception if L has no such item.

L.append(x) Appends item x to the end of L ; e.g., L[len(L):]=[x].

L.extend(s) appends all the items of iterable s to the end of L; e.g., L[len(L):]=s.

L.insert(i, x) Inserts item x in L before the item at index i, moving following items of L (if any) "rightward" to make space (increases len(L) by one, does not replace any item, does not raise exceptions: acts just like L[i:i]=[x]).

L.remove(x) removes from L the first occurrence of an item in L that is equal to x, or raises an exception if L has no such item.

L.pop([i]) returns the value of the item at index i and removes it from L; if i is omitted, removes and returns the last item; raises an exception if L is empty or i is an invalid index in L.

L.reverse() reverses,the items of L.

Tuples: are the immutable version of lists. Therefore, any operations which affects the contents of a tuple will raise exceptions. Tuple are defined using parentheses. For example x = (1,2,3,4,5), x[0] = 6 is going to raise an exception. Also, note that to a tuple of one element is defined as follows x = (1,) where a comma is used after a comma, other x = (1) is the same as x = 1.

Sets: (Added in Python 2.4) are unordered collections of unique elements. Common uses include membership testing, removing duplicates from a sequence, and computing standard math operations on sets such as intersection, union, difference, and symmetric difference. To create an empty set you can call the function set() without arguments, or you can call it with an iterable argument. For example: x = set([1,2,3,4,3,4,6]) and the result would be set([1,23,4,6])

Another example with a string:

x = set ("asdfasdfasdFasdf") and the result would be set(['a', 's', 'F', 'd', 'f'])

Here is a list of the most common set functions:

S.difference(S1) returns the set of all items of S that aren't in S1

S.intersection(S1) returns the set of all items of S that are also in S1

S.issubset(S1) returns true if all items of S are also in S1; otherwise, returns False

S.issuperset(S1) returns true if all items of S1 are also in S; otherwise, returns False (like S1.issubset(S))

S.symmetric_difference(S1) returns the set of all items that are in either S or S1, but not in both sets

S.union(S1) returns the set of all items that are in S, S1, or in both sets

S.add(x) adds x as an item to S; no effect if x was already an item in S

S.clear( ) removes all items from S, leaving S empty

S.discard(x) removes x as an item of S; no effect if x was not an item of S

S.pop( ) removes and returns an arbitrary item of S

S.remove(x) removes x as an item of S; raises a KeyError exception if x is not an item of S

Dictionaries: or a mapping or a hash as described by other languages. Are mutable sequences similar to lists however, they are not ordered, and can be indexed by arbitrary object types. Indexes in dictionaries are known as keys and each item in a dictionary is identified by a key value pair. To specify a dictionary, you can use a series of pairs expressions, using key:value format. For example D = {'a':3,1:"Maya"} is a dictionary. The keys are 'a' and 1 the key value pairs are 'a':3 and 1:"Maya"
Also, D['a'] is 3 and D[1] is "Maya"

Here is a list of the most common functions on a dictionary D:

D.copy() returns a shallow copy of the dictionary (a copy whose items are the same objects as D's, not copies thereof)

D.has_key(k) returns true if k is a key in D; otherwise, returns False, just like k in D

D.items( ) returns a new list with all items (key/value pairs) in D

D.keys( ) returns a new list with all keys in D

D.values( ) returns a new list with all values in D

D.iteritems( ) returns an iterator on all items (key/value pairs) in D

D.iterkeys( ) returns an iterator on all keys in D

D.itervalues( ) returns an iterator on all values in D

D.get(k[, x]) returns D[k] if k is a key in D; otherwise, returns x (or None, if x is not given)

D.clear( ) removes all items from D, leaving D empty

D.update(D1) for each k in D1, sets D[k] equal to D1[k]

D.pop(k[, x]) removes and returns D[k] if k is a key in D; otherwise, returns x (or raises an exception if x is not given)

D.popitem( ) removes and returns an arbitrary item (key/value pair)

Control Flow: Python supports the standard control flow statement that you find in other languages.

If statement:

```
if Boolean expression: #the ':' signals the start of a new block beneath it
   statement 1 #Note the indentation
        ...
   statement M #Note the indentation is the same and the previous one
elif Boolean expression: # elif same as else if
```

```
        statement 1
            …
        statement M'
    else:
        statement 1
            …
        statement M"
```

An example of an if statement:

```
if x > 5 and y <18:
        print "Hello World\n"
elif x<5 and y<18:
        print "Hello Arizona\n"
else:
        print "Hello Michigan\n"

print "We are out of the if statement\n"
```

while loop statement:

```
        while Boolean expression: #the ':' signals the start of a new block
            statement 1
                        …
            statement M
```

Here is an example:

```
        X = 5
        Y = 0
        while X >0 :
                print X ,"\n"
                Y += 1
                X -= 1
```

Here is another example of a while loop:

```
num = 24
trials = 0
while trials < 10:
        pick = int(raw_input('Enter an integer : '))
        trials += 1
        if pick == num:
```

```
            print 'Good guess.'
            break
    elif pick < num:
            print 'pick a bigger number.'
    else:
            print 'pick a lower number.'
else:
    print "You will get here if you ran out of trials"

print 'You get here directly if you guessed correctly"
```

for loop statement: The for statement is somewhat different than the typical for statement in terms of how to specify loop condition. Here is the general form of the loop statement:

```
            for item in iterableEntity: #the ':' signals the start of a new block
                statement 1
                    …
                statement M
```

The iterableEntity can be any iterable sequence.
For example to loop over a list
:

```
L = [2,5,6,8]
for l in L:
    print l
```

another example to loop over a string

```
S = "asdf"
for c in S:
    print c
```

Another example to iterate over a dictionary would be:

```
for key, value in D.items():
    print key, " : ", value , "\n"
```

to loop over a set:

```
x = set('Maya is the coolest tool')
for c in x:
    print c
```

The output of the loop will be
a

c
e
i
h
M

l
o
s
t
y

Ranges: are used to allow iteration over a sequence of integers. Similar to the standard for loop statements in other languages. Python has two function to specify ranges: **range**(start, end, step) the step is optional default to 1.
The range function create a sequence which starts at 'start', where start is included and ends at 'end' where 'end' is exluded. For example range(1,4) returns the sequence
[1,2,3]
Another example of a for loop with a range:

```
for i in range(3,20):
    print i
```

The **break** and **continue** statements are allowed in the loop bodies. **break** will transfer execution to the outer block of the loop, and therefore exits the loop. While **continue** will transfer the execution to the Boolean expression of the loop, and therefore to the next iteration.

List comprehensions: are lists created a natural way based on other lists. It is a compact way to iterate over the elements of a list under an optional condition. The notation is similar to the way mathematicians specify a set in set theory.
S' = {f(x) such that x belongs to D}
Or
S' = `{x*x | x in S and x even}`

```
S = range(2,12,2)
M = [2+x for x in S if x % 2 == 0] # % is modulo funtion
```

M will end up being [4,6,10,12]

Functions: You can define functions in Python using the **def** keyword as follow:

```
def function-name(param1, param2, …, paramM):
    statement 1
    statement 2
        ….
    Statement N
```

If the function is supposed to return a value, use the **return** keyword.

An example:

```
def pay(x , y , z):

    if  x>y and y<z:
```

```
        return y + z
    else:
        return z
```

The function name is a variable which can be assigned to other functions.

For example foo = pay

X = foo(3,4,5) will set X to 9

Parameters of a functions can have a default value specified in the function definition

For example:

```
def register(course, university, state="Michigan")
```

In this case the function can be called as register("math101","MSU") or

```
register("math101", "ASU", "AZ")
```

All mutable types are passed by **reference**. So, if a list is passed as a parameter, and elements of that parameter where manipulated by the function, then the change is going to affect the list that was passed to the function.

For example:

```
def foo (s):
    S[0] = 3
X = [1,2,4]
foo(X)
```

X is now [3,2,4]

Variables defined within functions are considered local and they exist for the duration of the function. You can you the **global** keyword to refer to a pre-existing global variable ie outside the scope of the function or make a local variable global.

Exception handling:

Exception handling enables programmers to create applications that can resolve exception. Python creates **traceback** objects when it encounters exceptions.

Python implements exception handling via try...except blocks.
Python uses try...except to handle exceptions and raise to generate them.

Exceptions are everywhere in Python. Virtually every module in the standard Python library uses them, and Python
itself will raise them in a lot of different circumstances. You've already seen them repeatedly throughout this book.

· Accessing a non−existent dictionary key will raise a KeyError exception.
· Searching a list for a non−existent value will raise a ValueError exception.
· Calling a non−existent method will raise an AttributeError exception.
· Referencing a non−existent variable will raise a NameError exception.

```
try:
        f = open('myfile.txt')
        s = f.readline()
        i = int(s.strip())
except IOError, (errno, strerror):
        print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
        print "Could not convert data to an integer."
except:
        print "Unexpected error:", sys.exc_info()[0]

        raise
```

The try . . . except statement has an optional *else clause*, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

```
for arg in sys.argv[1:]:
try:
        f = open(arg, 'r')
except IOError:
        print 'cannot open', arg
else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

The try statement has another optional clause which is intended to define clean-up actions that must be executed
under all circumstances. For example:

```
try:
```

```
        raise KeyboardInterrupt
finally:
        print 'Goodbye, world!'
```

A *finally clause* is executed whether or not an exception has occurred in the try clause. When an exception has occurred, it is re-raised after the finally clause is executed. The finally clause is also executed "on the way out", when the try statement is left via a break or return statement. The code in the finally clause is useful for releasing external resources (such as files or network connections), regardless of whether or not the use of the resource was successful.
A try statement must either have one or more except clauses or one finally clause, but not both.

Modules (adapted from the Python Tutorial): A module is a file containing Python definitions and statements. The file name is the module name with the suffix '.py' appended. Within a module, the module's name (as a string) is available as the value of the global variable __name__.

For instance, use your favorite text editor to create a file called 'fibo.py' in the current directory with
the following contents:

```
# Fibonacci numbers module
        def fib(n): # write Fibonacci series up to n
                a, b = 0, 1
                while b < n:
                print b,
                a, b = b, a+b
                def fib2(n): # return Fibonacci series up to n
                result = []
                a, b = 0, 1
                while b < n:
                result.append(b)
                a, b = b, a+b
                return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters
the module name fibo there. Using the module name you can access the functions:


```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

```
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, modname.itemname.
Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing
module's global symbol table.
There is a variant of the import statement that imports names from a module directly into the importing module's
symbol table. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, fibo is not defined).
There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (_).

The Module Search Path: When a module named spam is imported, the interpreter searches for a file named 'spam.py' in the current directory, and then in the list of directories specified by the environment variable PYTHONPATH. This has the
same syntax as the shell variable PATH, that is, a list of directory names. When PYTHONPATH is not set, or when the file is not found there, the search continues in an installation-dependent default path; on UNIX, this is usually '. C:\Python25\lib\Python'.

Actually, modules are searched in the list of directories given by the variable sys.path which is initialized from the directory containing the input script (or the current directory), PYTHONPATH and the installation-dependent default.

Standard Modules

Python comes with a library of standard modules, described in the *Python Library Reference.* One particular module deserves some attention: sys, which is built into every Python interpreter.
The variable sys.path is a list of strings that determine the interpreter's search path for modules. It is initialized to a default path taken from the environment variable PYTHONPATH, or from a built-in default if PYTHONPATH is not set. You can modify it using standard list operations:

```
>>> import sys
>>> sys.path.append('c:\myApps\pyApps')
```

The built-in function dir() is used to find out which names a module defines. It returns a sorted list of strings. Without arguments, dir() lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
```

Packages : Packages are a way of structuring Python's module namespace by using "dotted module names". For example,the module name A.B designates a submodule named 'B' in a package named 'A'.

A package is a module that contains other modules or packages. Some or all of the modules in a package may be subpackages, resulting in a hierarchical tree-like structure. A package named P resides in a subdirectory, also called P, of some directory in sys.path. Packages can also live in zip files; as in the site-packages directory in Maya.

Suppose you want to design a package for handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: .wav, .aiff, .au), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

(EXAMPLE take from the python tutorial)

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
```

```
        auread.py
        auwrite.py
        ...
    effects/                Subpackage for sound effects
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
    filters/                Subpackage for filters
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
        ...
```

When importing the package, Python searches through the directories on sys.path looking for the package subdirectory.

The __init__.py files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as "string", from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, __init__.py can just be an empty file, but it can also execute initialization code for the packag.

Users of the package can import individual modules from the package, for example:

import sound.effects.echo

This loads the submodule sound.effects.echo. It must be referenced with its full name.

sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)

An alternative way of importing the submodule is:

from sound.effects import echo

This also loads the submodule echo, and makes it available without its package prefix, so it can be used as follows:

echo.echofilter(input, output, delay=0.7, atten=4)

Yet another variation is to import the desired function or variable directly:

from sound.effects.echo import echofilter

Again, this loads the submodule echo, but this makes its function echofilter() directly available:

echofilter(input, output, delay=0.7, atten=4)

Note that when using from *package* import *item*, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The import statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an ImportError exception is raised.

Classes: Similar to other languages, a class is a user-defined type, which you can instantiate to obtain instances, meaning objects of that type. Python supports these concepts through its class and instance objects. The concepts Encapsulation, Polymorphism, ie the ability to overload standard operators and functions so that they have appropriate behavior based on their context, and Inheritance, apply.

The general form to define a class is as follows:

```
class ClassName(inheritedClasses):
    "”” Describe class functionality.
       This string is optional, however it is useful, to
       Document the purpose of the class. It is displayed,
       By, the help() function. Note that functions can have such
       String as well.
    "”"
    <statement-1>
            .
            .
            .
    <statement-N>
```

The class constructor __init__() function:

When a class defines or inherits a method named __init__, calling the class object implicitly executes __init__ on the new instance to perform any needed instance-specific initialization. Arguments passed in the call must correspond to the parameters of _ _init_ _, except for parameter self. For example, consider the following class:

```
class C:
   def __init__(self, n):
      self.x = n
```

Each object created from a class is an instance of a class. They all look alike and exhibit a similar behavior.
A class stores object attributes*,* data members, and the behavior of objects, methods. This behavior can be inherited from other (*base*) classes. The non-method attributes of the class are usually referred to as class members or class attributes so that they are not confused with instance attributes.
Each class has its own namespace in which all the assignments and function definitions occur.

All methods have the additional argument self as the first argument in the method header—it is conventional to call it self, however it can be any other name. Python's self argument is similar to 'this' keyword in C++. Its function is to transport a reference of the object in a way that when a method is called, it knows which object should be used.

```
class Vehicle:
        def __init__(self, name):
                self.name = name
        def  printModel(self):
                print 'Car model', self.name
```

Note in the example is variable name, is considered an instance variable.

```
class molecule:
        def __init__(self,name='Generic'):
                self.name = name
                self.atomlist = []
        def addatom(self,atom):
                self.atomlist.append(atom)
                def __repr__(self):
                str = 'This is a molecule named %s\n' % self.name
                str = str+'It has %d atoms\n' % len(self.atomlist)
                for atom in self.atomlist:
                str = str + `atom` + '\n'
                return str
```

There are two types of variables or attributes that can be associated with a class. Class attributes and Object or Instance attributes. Class attributes are accessed directly from the class definition, where as instance attributes are accessed via an object instance of the class. Object attributes are defined within the instance methods of the class, Usually within the __init__() constructor function.
Class attributes are defined at the class level.

Here is a good description of the various components of a class.

```
class class_name [(baseclass1, baseclass2, …)]: # [ ] mean optional
        ["doc text"]                    #Optional doc text, it can be multi line.
        classVar1 = None                #Class variable are defined at the class level
        classVar2 = []          #Another class variable.
        classVar3 = ""

        def staticFunction(param1, param2, … ): #Note there is no self parameter
                Statement1
                Statement2
                Statement3

        def __init__(self,instanceParam1, instanceParam2, …):
```

```
        self.instanceVar1 = instanceParam2
        self.instanceVar1 = instanceParam2
        statement1
        statement2

    def instFunction(self, p1, p2, …): #
        self.instanceVar2 = p1
        self.instanceVar2.append(p2)
        statement 1
        statement 2
```

Class variables can be set directly as follows:

class_name.classVar1 = "Hi There!"

Class static functions can be called directly as follows:

class_name.staticFunction("a","b","c", …)

In order to access instance attributes and methods, you have to create an instance of the class:

instance = class_name('a','b',…)
hence instanceVar1 is now being set to 'a' and instanceVar2 is being set to 'b' as per the __init__ function.

Methods may call other methods by using method attributes of the self argument:

```
        class Bag:
            def __init__(self):
                self.data = []
            def add(self, x):
                self.data.append(x)
            def addtwice(self, x):
                self.add(x)
                self.add(x)
```

MySQL database interaction: In order to use MySQL with Python, download the MySQLdb module for python from sourceforge and install it. You also have to have access to a MySQL database either locally or on a reachable system. The author of the module is Andy Dustman, http://dustman.net/andy/python/
His website has more elaborate description of the module. Also, refer to Python Database API Specification v2.0 at http://www.python.org/dev/peps/pep-0249/ for a description of all the MySQLdb API objects and their methods.
After you install the module. Start the Python console in order to get familiar with how use the module.

Scripts that access MySQL through DB-API using MySQLdb generally perform the following steps:

- Import the MySQLdb module
- Open a connection to the MySQL server
- Issue statements and retrieve their results
- Close the server connection

In order to use a MySQL database you have to connect to the server where the database is located.
In order to do that you use the connect function of the MySQLdb module. It will created a connection object which will be your conduit to access the database.

Use IDLE to write the following script. This script uses MySQLdb to interact with the MySQL server, albeit in relatively rudimentary fashion--all it does is ask the server for its version string:

```
import MySQLdb

 try:
          connection = MySQLdb.connect (host="localhost",user = "maya_user",
          passwd = "myPassword", db = "mayaTexturesDB")
         cursor.execute( "SELECT * FROM MAYA_TEXTURES;")

         tableHeader = cursor.description()
         allRows = cursor.fetchall()
except MySQLdb.Error, e:
          print "Error %d: %s" % (e.args[0], e.args[1])
          sys.exit (1)
else:
     cursor.close ()

     connection.close ()
```

The assumption here is that you have created a database in MySQL called mayaTexturesDB with a user maya_user and password myPassword. For more information on how to do that I highly recommend www.mysql.org, of course you have to also be familiar with SQL language in order to create the table.

```
cursor = conn.cursor ()
```

Then you have to create a cursor over the connection in order to supply the SQL statements you want to execute, and hence process these statements.

```
cursor.execute( "SELECT * FROM MAYA_TEXTURES;")
```

```
tableHeader = cursor.description() #get field names, table Header is a
tuple.
allRows = cursor.fetchall() #allRows will be in the form of tuple of
tuples.
```

The cursor object's `execute()` method sends the statement to the server and `fetchall()` retrieves a row as a tuple, and hence it will be a tuple of tuples for the where each tuple is a row in the table. For the statement shown here, each row or tuple tuple contains multiple items. (If no rows, ie empty table then `fetchall()` will return the value `None`; Cursor objects can be used to issue multiple statements.

Finally, the script invokes the connection object's `close()` method to disconnect from the server.

Here is another example:

```
cursor = connection.cursor ()
  cursor.execute ("""
      CREATE TABLE MAYA_TEXTURES
      (
        id              INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
        file_name        VARCHAR(100),
        category        VAR CHAR(100),
        VehicleId       VARCHAR(20),
        Dept            VARCHAR20),
        AuthorID        VARCHAR(40),
        RevID           VARCHAR(20)

      )
    """)
cursor.execute ("""
      INSERT INTO animal (name, category)
      VALUES
        (
'Cosmo_Brand_Thread',
'Tires',"TRANSACT_CAR","CONCEPT","ST100","OPERATIONS","JWYMAC","v2.1"),

    """)

print "Number of rows inserted: %d" % cursor.rowcount
```

Parsing XML with xml.sax package: SAX stands for the Simple API for XML. It is a popular interface due to its simplicity and ease of use. Originally a Java-only API. SAX was the first widely adopted API for XML in Java, and is a "de facto" standard. The current version is SAX 2.0.1, and there are versions for several programming language environments other than Java, including Python. The first job of using SAX is to design and implement a (event) handler that works with your specific XML documents. SAX is a callback-based API in which you implement handler objects to process XML. When parsing begins, the parser calls the methods on your handler objects whenever it encounter an xml element interest and hence an event is generated

which allows you to process the XML, so that you can do something useful with it in your applications and distributed system. SAX allows for faster processing of documents, as well as handling of documents that are simply too large to load into memory.
Additionally, the event-based API allows you to react to parsing events and errors in "real-time," as they occur, while parsing the document, rather than waiting for the entire document to load. Another huge win for many applications is the lower memory consumption when compared to DOM-based code.
SAX refers to the parser object as a reader. It reads input from some source and generates calls
to the handler methods for particular events in the input. As the parser encounters markup, events are generated and the parser calls the event handlers to handle those tags. For example if it encounters a start tag, the startElement event handler is called, or when it counters data or text between tags the characters event handler is called. These handlers are specified by the developer based on the desired response when encountering that tag.
The main event handlers which are methods of the xml.sax.ContentHandler module are:
character(data): called when the parser encounters character data which is captured by the data parameter of the character method.
endDocument() method: Called when the parser encounters the end of the document.
endElement(tag): Called when the parser encounters an end tag. The tag name is passed as a parameter to the event handler.
startDocument(): called when the parser encounters the beginning of the document.
The main modules of xml.sax are ContentHandler which encapsulates the callback or event handler functions. The parse module, the reader module, and the SAXParseException module.