Installation: The Python interpreter is freely available at www.python.org. In addition to the basic Python interpreter, we will be using matplotlib (also know as pylab), a Python library for creating 2D plots. Matplotlib uses the NumPy library for numerical functions, so that library also needs to be installed.

Windows users: The Windows versions of Python, matplotlib and NumPy are posted on the course website. Download the following programs at run them in the listed order. Use default values for all installation parameters.
- python-2.5.2.msi
- numpy-1.1.1-win32-superpack-python2.5.exe
- matplotlib-0.98.3.win32-py2.5.exe

Mac and Linux users: Locate the appropriate installers from the following sites and install them in the order listed.
- Python (v2.5.2): http://www.python.org/download/
- NumPy (v1.1.1): http://numpy.scipy.org/
- Matplotlib (v0.98.3): http://matplotlib.sourceforge.net/

Writing Python code: You can write Python code with any ASCII text editor. The Windows installation also comes with IDLE, the Python development environment, which can be used to both write and run Python code. Your Python code files should always end in a .py suffix.

Running Python code: Since Python is an interpreted language, there is no separate compilation process (as there is in Java or C++). Simply call the interpreter and pass it the name of the Python file containing your main program.

If you used the default installation path for Windows, the interpreter will be C:\Python25\python.exe. If you are running form the Windows command line, you may want to add C:\Python25 to your Path environment variable (My Computer → right-click → Properties → Advanced → Environment Variables).

If you are using IDLE, you can run the code in the editor by pressing F5 or Run → Run Module.
Essential Documentation:

Installing Python also installs a variety of useful documents. On Windows, the documents are accessible from START → Programs → Python 2.5 → Python Manuals. These documents are also available online at http://docs.python.org/.

The primary documents of interest are “Tutorial” and “Library Reference”.

If you are new to Python, you should become familiar with the following chapters in the Tutorial:
   3. An Informal Introduction to Python: numbers, strings and lists
   4. More Control Flow: conditionals and loops
   5. Data Structures: lists, tuples and dictionaries are essential – sequences and sets are less important
   6. Modules: modules are how Python deals with multiple files and libraries
   9. Classes: object-oriented concepts

The following are less critical, but still valuable:
   7. Input and Output
   8. Errors and Exceptions
Essential Data Structures:

**Lists:** Lists are mutable ordered sequences. This makes them very flexible as they can also behave like many common data structures such as queues and arrays. Lists are constructed with square brackets `[]`.

**Tuples:** Tuples are immutable ordered sequences. Tuples are best used for packing values into a single structure, rather than for dynamic structures. Tuples are constructed as comma-separated values. They may be enclosed in parentheses `()`, which may be required in some contexts.

**Dictionaries:** Dictionaries define a mapping from keys to values. Other languages call these maps or associative arrays. Dictionaries are constructed with braces `{}`.

**Strings:** Strings are sequences of characters. Strings are immutable. Strings can be defined with double or single quotes (e.g. “foo” or ‘bar’).

Examples:

```python
>>> s = "a string"  # a string of length 8
>>> t = 'a string'  # a string of length 8
>>> a = ['spam', 'eggs', 100, 1234]  # a list of 4 elements
>>> b = []  # an empty list
>>> c = 1, 2, 3  # a tuple of three elements
>>> d = (1, 2, 3)  # a tuple of three elements
>>> e = ()  # an empty tuples
>>> f = 'spam',  # a tuple with one element (comma is required!)
>>> g = ('spam',)  # a tuple with one element (comma is required!)
>>> h = {'x':1, 'y':2, 'z':2}  # a dictionary mapping strings to integers
>>> i = {1:'x', 2:'y', 3:'z'}  # a dictionary mapping integers to strings
>>> j = {}  # a empty dictionary
```
Indexing: Lists, tuples and strings can be indexed – the first index is 0.

```python
>>> 'a string'
'a string'

>>> s = 'a string'
>>> s[0]
'a'
>>> s[3]
't'

Negative indexes begin counting from the back.

```python
>>> v = [10, 11, 12, 13, 14]
>>> v[-2]
13
```

Since lists are mutable, indexed list elements can appear on the left hand side of an assignment.

```python
>>> v
[10, 11, 12, 13, 14]
>>> v[3] = 999
>>> v
[10, 11, 12, 999, 14]
```

Slicing: Slicing specifies a subsequence, with start and end indexes. You can also slice with step size as a third argument.

```python
>>> v = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> v[3:4]
[4]

>>> v[3:9:2]
[4, 6, 8]

>>> v[-7:-1]
[4, 5, 6, 7, 8, 9]

>>> v[3:5] = [99, 100] # mutable objects only
>>> v
[1, 2, 3, 99, 100, 6, 7, 8, 9, 10]
```

```python
>>> v[3:5] = [99, 100, 101, 102]
>>> v
[1, 2, 3, 99, 100, 101, 102, 6, 7, 8, 9, 10]

>>> x = [1, 2, 3]
>>> x[:0] = x
>>> x
[1, 2, 3, 1, 2, 3]
```
Length function:  `len(x)` returns the number of elements in structure `x`. It can be applied to lists, strings, tuples and dictionaries.

Useful list methods: These are methods (rather than free functions) and require `x.f()` notation, where `x` is the list.

- `a.append(x)`: Add item `x` to the end of the list, equivalent to `a[len(a):] = [x]`.
- `a.extend(L)`: Extend the list `a` by appending all the items in the given list; equivalent to `a[len(a):] = L`.
- `a.insert(i, x)`: Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
- `a.remove(x)`: Remove the first item from the list whose value is `x`. It is an error if there is no such item.
- `a.pop([i])`: Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the `i` in the method signature denote that the parameter is optional)
- `a.index(x)`: Return the index in the list of the first item whose value is `x`. It is an error if there is no such item.
- `a.count(x)`: Return the number of times `x` appears in the list.
- `a.sort()`: Sort the items of the list, in place.
- `a.reverse()`: Reverse the elements of the list, in place.

Using Lists as Stacks or Queues:

```python
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

```python
>>> queue = ['Eric', 'John', 'Michael']
>>> queue.append('Terry')  # Terry arrives
>>> queue.append('Graham')  # Graham arrives
>>> queue.pop(0) 'Eric'
>>> queue.pop(0) 'John'
>>> queue
['Michael', 'Terry', 'Graham']
```
**Dictionaries:** Dictionaries are indexed by keys, rather than positions. Dictionaries are mutable, in that the value associated with a key may be changed.

```python
>>> d = { 'COMP51': 'Cliburn', 'COMP53': 'Carman', 'COMP151': 'Doherty' }
>>> d
{'COMP51': 'Cliburn', 'COMP53': 'Carman', 'COMP151': 'Doherty'}
>>> d['COMP51']
'Cliburn'
>>> d['COMP51'] = 'Topp'
>>> d['COMP51']
'Topp'
```

**Operator precedence:** Operators higher in the table have higher precedence.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>number of operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>logical OR</td>
<td>binary</td>
</tr>
<tr>
<td>and</td>
<td>logical AND</td>
<td>binary</td>
</tr>
<tr>
<td>not x</td>
<td>logical NOT</td>
<td>unary</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=, &lt;&gt;, !=, ==</td>
<td>relational (comparison)</td>
<td>binary</td>
</tr>
<tr>
<td>+, -</td>
<td>addition/concatenation and subtraction</td>
<td>binary</td>
</tr>
<tr>
<td>* / %</td>
<td>multiplication, division, remainder</td>
<td>binary</td>
</tr>
<tr>
<td>+x, -x</td>
<td>positive, negative</td>
<td>unary</td>
</tr>
<tr>
<td>**</td>
<td>exponentiation</td>
<td>binary</td>
</tr>
<tr>
<td>x[index]</td>
<td>indexing</td>
<td>binary</td>
</tr>
<tr>
<td>x[index:index]</td>
<td>slicing</td>
<td>trinary</td>
</tr>
</tbody>
</table>
### String Operators and Functions:

In column 1, s and t represent string values; i and j represent integer values.

<table>
<thead>
<tr>
<th>operator or function</th>
<th>return type</th>
<th>description</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>s + t</td>
<td>string</td>
<td>concatenation: combine two strings to make a new string</td>
<td>&gt;&gt;&gt; s = 'break' &gt;&gt;&gt; t = 'fast' &gt;&gt;&gt; s+t 'breakfast'</td>
</tr>
<tr>
<td>s *</td>
<td>string</td>
<td>repetition: concatenate multiple copies of the same string</td>
<td>&gt;&gt;&gt; s = 'yao' &gt;&gt;&gt; s*3 'yaoyaoyao'</td>
</tr>
<tr>
<td>len(s)</td>
<td>integer</td>
<td>length: returns number of characters in a string</td>
<td>&gt;&gt;&gt; a = 'abcdefg' &gt;&gt;&gt; len(a) 7</td>
</tr>
<tr>
<td>min(s)</td>
<td>integer</td>
<td>minimum: finds smallest character in a string (compared by ASCII codes)</td>
<td>&gt;&gt;&gt; z = 'June' &gt;&gt;&gt; min(z) 'J'</td>
</tr>
<tr>
<td>max(s)</td>
<td>integer</td>
<td>maximum: finds largest character in a string (compared by ASCII codes)</td>
<td>&gt;&gt;&gt; z = 'December' &gt;&gt;&gt; max(z) 'r'</td>
</tr>
<tr>
<td>s[i]</td>
<td>string</td>
<td>indexing: extract a single character from a string by position</td>
<td>&gt;&gt;&gt; t = 'particular' &gt;&gt;&gt; t[4] 'i' &gt;&gt;&gt; t[0] 'p'</td>
</tr>
<tr>
<td>s[i:j]</td>
<td>string</td>
<td>slicing: extract a range of characters from a string by position</td>
<td>&gt;&gt;&gt; t = 'particular' &gt;&gt;&gt; t[4:8] 'icul' &gt;&gt;&gt; t[:3] 'par' &gt;&gt;&gt; t[7:] 'lar'</td>
</tr>
</tbody>
</table>
Functions provided by the string library (requires `import string`):

In first column: `s, t` represent string values, `n` represents integer values and `list` represents a list of strings.

<table>
<thead>
<tr>
<th>function</th>
<th>return type</th>
<th>description</th>
<th>example</th>
</tr>
</thead>
</table>
| center(s, n) | string      | Create a new string of `n` characters with `s` in the center. Remaining characters are spaces. | `>>> string.center('dog', 8)`
|              |             |                                                                            | ' dog '                                                                                     |
| ljust(s, n)  | string      | Create a new string of `n` characters with `s` at the left. Remaining characters are spaces. | `>>> string.ljust('dog', 8)`
|              |             |                                                                            | 'dog                                                                       |
| rjust(s, n)  | string      | Create a new string of `n` characters with `s` at the right. Remaining characters are spaces. | `>>> string.rjust('dog', 8)`
|              |             |                                                                            | '     dog'                                                                   |
| strip(s)     | string      | Create a copy of `s` with leading and trailing whitespace removed.           | `>>> string.strip('   cat   ')`                                                            |
|              |             |                                                                            | 'cat'                                                                             |
| lstrip(s)    | string      | Create a copy of `s` with leading whitespace removed.                       | `>>> string.lstrip('   cat   ')`                                                            |
|              |             |                                                                            | 'cat   '                                                                      |
| rstrip(s)    | string      | Create a copy of `s` with trailing whitespace removed.                      | `>>> string.lstrip('   cat   ')`                                                            |
|              |             |                                                                            | '   cat'                                                                      |
| capitalize(s)| string      | Create a copy of `s` with the first character capitalized.                  | `>>> string.capitalize('driver disk')`                                                      |
|              |             |                                                                            | 'Driver disk'                                                                |
| capwords(s)  | string      | Create a copy of `s` with the first character in each word capitalized. (Words are separated by whitespace) | `>>> string.capwords('three short words')`                                                |
|              |             |                                                                            | 'Three Short Words'                                                             |
| upper(s)     | string      | Create a copy of `s` with all letters converted to upper-case.             | `>>> string.upper('this is loud')`                                                        |
|              |             |                                                                            | 'THIS IS LOUD'                                                               |
| lower(s)     | string      | Create a copy of `s` with all letters converted to lower-case.             | `>>> string.lower('A funKY StriNG')`                                                      |
|              |             |                                                                            | 'a funky string'                                                              |
| split(s)     | list of strings | Split a string into a list of substrings. Strings are split at whitespace (word boundaries). | `>>> string.split('Break this apart.')`                                                  |
|              |             |                                                                            | ['Break', 'this', 'apart.']                                                           |
| split(s, t)  | list of strings | Split a string into a list of substrings. Strings are split at occurrences of the string `t`. | `>>> string.split('Break this apart.', 'a')`                                           |
|              |             |                                                                            | ['Bre', 'k this ', 'p', 'rt.']                                                          |
| join(list)   | string      | Concatenate all string in a list to create a single string.                | `>>> list = ['One', 'Two', 'Three']`                                                      |
|              |             |                                                                            | `>>> string.join(list)`
|              |             |                                                                            | 'One Two Three'                                                                 |
Functions provided by the string library (requires `import string`):

In first column: s, t and u represent string values.

<table>
<thead>
<tr>
<th>function</th>
<th>return type</th>
<th>description</th>
<th>example</th>
</tr>
</thead>
</table>
| count(s, t)  | integer     | Determine the number of times that t appears in s.                          | >>> s = 'Mississippi'  
>>> string.count(s, 'iss') 2  
>>> string.count(s, 'issi') 1 |
| find(s, t)   | integer     | Determine the position (index) of the first occurrence of t in s, looking from the left. | >>> s = 'Mississippi'  
>>> string.find(s, 'iss') 1 |
| rfind(s, t)  | integer     | Determine the position (index) of the first occurrence of t in s, looking from the right. | >>> s = 'Mississippi'  
>>> string.rfind(s, 'iss') 4 |
| replace(s, t, u) | string     | Make a copy of s in which all occurrences of t have been replaced by u.      | >>> x = 'banana'  
>>> string.replace(x, 'an', '-an')  
'b-an-ana' |
CLASSES AND OBJECTS

An object is an encapsulation of data and the functions that operate on that data.

The data that stored in an object is stored as instance variables.

The functions that operate on that data are called methods.

A class defines a new data type that specifies the methods that operate on all instances of that class.

The instance variables in an object are identified by the keyword self. Instance variables are created the first time that an object method makes an assignment to that variable.

Here is an example of a simple class declaration:

```python
class Counter:
    def __init__(self):
        self.count = 0
    def increment(self):
        self.count = self.count + 1
    def getCount(self):
        return self.count

my_counter = Counter()
my_counter.increment()
my_counter.increment()
my_counter.increment()
print my_counter.getCount()  # prints 3
```

Here is an example of the usage of that class to create and manipulate an object:

```python
my_counter = Counter()
my_counter.increment()
my_counter.increment()
my_counter.increment()
print my_counter.getCount()  # prints 3
```

Objects are created by calling the class constructor. Constructors always have the same name as the class.

The implementation of a constructor is identified by the special method name __init__, (that’s underbar-underbar-init-underbar-underbar). The first argument to a constructor is always the keyword self, which is used to identify the instance variables of the object that is being created.
In our example, the constructor creates and initializes a single instance variable called `self.count`. Thus, the statement

```python
my_counter = Counter(),
```

creates an object that looks like:

```
my_counter

|self.count| 0 |
```

Notice how the variable `self.count` is encapsulated inside the object named `counter`.

We manipulate objects by calling the methods on the object, with the object named prepended with a dot. The object name actually gets used as the first parameter to the method, thus the object name becomes associated with the `self` argument. Thus, the statement

```python
my_counter.increment()
```

adds 1 to the `count` variable inside the object named `my_counter`.

```
my_counter

|self.count| 1 |
```

We can create multiple objects of the same type, and each has its own copy of the instance variables. For example:

```python
c1 = Counter()
c2 = Counter()
c1.increment()
c1.increment()
c2.increment()
p = c1.getCount(), c2.getCount # prints 2 1
```
produces the following objects:

Here's an example of a class that defines objects with two instance variables.

```python
class Scoreboard:
    def __init__(self):
        self.home = 0
        self.away = 0

    def addHome(self, points):
        self.home = self.home + points

    def addAway(self, points):
        self.away = self.away + points

    def homeScore(self):
        return self.home

    def awayScore(self):
        return self.away

    def printScore(self):
        print "Home:", self.home, "Away:", self.away
```

Notice that two of the methods have arguments besides `self`. Since the object name, which appears in front when calling a method, is used as the first argument, the additional arguments move forward one position in the call.

```python
board = Scoreboard()
board.addHome(3)
board.addAway(2)
board.printScore()
```

**OUTPUT:**
Home: 3 Away: 2

The following is a simplified version of the `Point` class from the graphics library. Notice that the constructor can also have additional arguments:

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def _move(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy

    def getX(self):
        return self.x

    def getY(self):
        return self.y
```

Here's how the parameter assignment works:

```python
board.addHome(3)
def addHome(self, points):
    self.home = self.home + points
```

The following is a simplified version of the Point class from the graphics library. Notice that the constructor can also have additional arguments:
Consider what will happen if we run the following code:

```python
p1 = Point(10, 20)
p2 = Point(100, 112)
p1.move(200, 0)
p2.move(50, -50)

print p1.getX(), p1.getY()
print p2.getX(), p2.getY()
```

At the end, we will have the following objects:

```
p1
   self.x ----> 210
   self.y ----> 20

p2
   self.x ----> 150
   self.y ----> 62
```

**Inheritance:** A class can be derived from another class through inheritance. The base class is indicated in parentheses in the declaration of the derived class. A derived class automatically has all methods defined for the base class.

```python
class ColoredPoint(Point):
    def __init__(self, x, y, color):
        Point.__init__(self, x, y)
        self.color = color

    def getColor(self):
        return self.color

c = ColoredPoint(1,2,'red')
print c.getX(), c.getY(), c.getColor()
```