The Python Programming Language

- Python is an interpreted language, and much like Ruby, it is ideal for prototyping because it can save a considerable amount of time during application development. The interpreter can be used interactively, which makes it easy to experiment with features of the language or to test code.

Interactive Mode

- In interactive mode, we open a python console and start entering commands. In this mode it prompts for the next command with the **primary prompt**, usually three greater-than signs (**>>>**); for continuation lines it prompts with the **secondary prompt**, by default three dots (**...**).

- The following are some examples of interactive mode (continuation lines are needed when entering a multi-line construct such as the ***if*** statement below):

```
# python
Python 2.7.10 (default, Sep 24 2015, 17:50:09)
[GCC 5.1.1 20150618 (Red Hat 5.1.1-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> Python_is_too_easy = 1
>>> if Python_is_too_easy:
...     print ("Why not use a civilized language instead?")
...
Why not use a civilized language instead?
>>>
```

- Note that indention using tabs is significant in Python.

- Python scripts can be made directly executable, like Ruby scripts, by adding the following line to the top of the file:

```
#! /usr/bin//python
```

- Or, if you wish invoke a specific version of Python:

```
#! /usr/bin//python3.4
```

- Python module is a single file with the same name (plus the .py extension)

Import and Modules

- Programs will often use classes and functions defined in another file or library.
- These are accessed using the "import" (like Java) keyword.
- The following are some examples of import with typical variations:

import time

  - Everything in time.py can be referred to by: ***time.Name.method("abc")*** or ***time.myFunction(100)***
  - For example: ***time.sleep(args.freq)***

from scapy.all import *

  - Everything in time.py can be referred to by: ***ClassName.method("abc")*** or ***myFunction***()
  - For example: ***sr (packet, timeout = 10)***

Code Structure and Basic Data Types

- Indentation matters to the meaning of the code; Block structure is indicated by indentation.
- Python figures out the variable types on its own, which means that variable types do not need to be declared. A variable is created when a value is first assigned to it.
- Variable assignment uses a "=", and comparison uses "==".
- Integers and floats use +, -, *, /, and % as operators. However, "+" is also used for string concatenation. Logical operators are words (and, or, not).
- String formatting uses % (similar to printf in C). Simple strings are printed using "print".
- **String examples:**

```
>>> 'foo bar'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"foo," foobar.'
'"foo," foobar.'
```

- We can make a string literal a "**raw**" string, **\n** sequences are not converted to newlines, but the backslash at the end of the line, and the newline character in the source, are both included in the string as data:

```
>>> print (hello)
Quis custodiet ipsos custodies? This is a latin phrase which means\n\
who will guard the guards?
```

- Strings can be concatenated with the **+** operator, and repeated with *:

```
>>> word = 'foo' + 'bar'
>>> word
'foobar'

>>> '<' + word*5 + '>'
'<foobarfoobarfoobarfoobarfoobar>'
```

- Strings can be subscripted (indexed); like an array, the first character of a string has subscript (index) 0:

```
>>> word[4]
'a'
>>> word[0:4]          # first four characters
'foob'
>>> word[:3]           # first three characters
'foo'
>>> word[3:]           # last three characters
'bar'
```

- Python represents all its data as objects. Some of these objects like ***lists*** and ***dictionaries*** are **mutable**, meaning their content can be changed without changing their identity.

- Other objects like integers, **floats**, **strings** and **tuples**, are **immutable;** objects that cannot be changed.

- As an example, we will use the ***id()*** method to examine the **identity** of objects:

```
>>> var = "foo"          # A string
>>> id(var)
140652893270984
>>> var += "bar"         # concatenate with another string
>>> id(var)
140652927334128         # ID has changed (mutable)
```

```
>>> foo = [1,2,3]          # A list
>>> id(foo)
140652927301320
>>> foo[0]
1
>>> foo[0] = 100           # change a list element
>>> id(foo)
140652927301320         # ID remains the same (immutable)
```

- **Lists**
    - Python provides a number of **compound** data types, used to group together other values. The most versatile is the **list**, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.
    - The following examples illustrate some list operations:

```
>>> bases = ['decimal', 'binary', 'hex', 10, 2, 16]
>>> bases[0]
'decimal'
>>> bases[:2]
['decimal', 'binary']
>>> bases[:2] + ['octal', 2*2]
['decimal', 'binary', 'octal', 4]
```

- **Tuples**
    - A **tuple** is a sequence of immutable Python **objects**. **Tuples** are sequences, just like lists. The differences between **tuples** and lists are, the **tuples** cannot be changed unlike lists and **tuples** use parentheses, whereas lists use square brackets.
    - A tuple consists of a number of values separated by commas, and to access values in tuple, we use the square brackets for slicing along with the index or indices to obtain value available at that index.

- The following are some examples:

```
>>> tup1 = ('fugu', 'awabi', 'shirako', 2000, 3000, 500)
>>> tup2 = (0, 1, 2, 3, 4, 5, 6, 7)
>>> tup1[0]
'fugu'
>>> tup2[0]
0
>>> print "tup1[0]: ", tup1[0]
tup1[0]:  fugu
>>> print "tup2[1:3]: ", tup2[1:3]          # slices between “1” and upto but including ”3”
tup2[1:3]:  (1, 2)
>>> print "tup2[1:6]: ", tup2[1:6]
tup2[1:6]:  (1, 2, 3, 4, 5)
>>> print "tup2[0:6]: ", tup2[0:6]
tup2[0:6]:  (0, 1, 2, 3, 4, 5)
```

Loops and Iterators

- Python provides all of the standard ways of performing repetitive operations using for, while, until, etc. These are summarized below.

- **while** loop

    - Executes its block of code as long as the condition (x < y) remains true. In Python, like in most other languages, any nonzero integer value is true; zero is false. For example:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, x, y = 0, 1, 10
>>> while x < y:
...     print('The value of x is', x)
...     a, x = x, a+x
...
('The value of x is', 1)
('The value of x is', 1)
('The value of x is', 2)
('The value of x is', 3)
('The value of x is', 5)
('The value of x is', 8)
>>>
```

    - The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false.

- The standard comparison operators are written the same as in C: < (less than), > (greater than), **==** (equal to), **<=** (less than or equal to), **>=** (greater than or equal to) and != (not equal to).

- The body of the loop is **indented**: indentation is Python's way of grouping statements.

- **if Statements**

  - This is similar to other languages:

```
>>> x = int(input("Enter an integer: "))
Enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('False')
... elif x == 1:
...     print('True')
... else:
...     print('You Entered', x)
...
('You Entered', 42)
>>>
```

  - There can be zero or more **elif** parts, and the **else** part is optional. The keyword '**elif**' is short for '**else if**', and is useful to avoid excessive indentation.

- **for Loops**

  - Python's **for** statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. The following examples illustrates this:

```
>>> languages = ["C", "C++", "Ruby", "Python"]
>>> for x in languages:
...     print x
...
C
C++
Ruby
Python
>>>
```

```
>>> languages = ['C', 'C++', 'Ruby', 'Python']
>>> for x in languages:
...     print(x, len(x))
...
('C', 1)
('C++', 3)
('Ruby', 4)
('Python', 6)
>>>
```

- It is not safe to modify the sequence being iterated over in the loop (this can only happen for mutable sequence types, such as lists).

- If you need to modify the list you are iterating over (for example, to duplicate selected items) you must iterate over a copy. The slice notation makes this particularly convenient:

```
>>> for x in languages[:]: # make a slice copy of the entire list
...     if len(x) > 4: languages.insert(0, x)
...
>>> languages
['Python', 'C', 'C++', 'Ruby', 'Python']
>>>
```

- The **range()**built-in function is very useful when iterating over a sequence of numbers. It generates arithmetic progressions:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>>
```

- The following example illustrates how to iterate over the indices of a sequence using **range()** and **len()** as follows:

```
>>> languages = ['C', 'C++', 'Ruby', 'Python']
>>> for i in range(len(languages)):
...     print(i, languages[i])
...
(0, 'C')
(1, 'C++')
(2, 'Ruby')
(3, 'Python')
>>>
```

Functions

- Python functions are a block of organized, reusable code that is used to perform a single, related action.
- Python provides many **built-in** functions like print(), etc. but we can also create our own functions. These functions are called **user-defined functions**.
- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The code block within every function starts with a **colon** (:) and is indented.
- The ***return*** [expression] statement exits a function, optionally passing back an expression to the caller.
- Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. The following example program illustrates this:

```
#!/usr/bin/python

# First define the function
def strprnt( str ):
   print str   # print the string argument
   return;

# Call the strprnt function, passing it a string argument
strprnt("Hello user-defined function!")
strprnt("print: foobar")
```

- The above code will produce the following output:

  # ./strprnt.py
  Hello user-defined function!
  print: foobar

- Another example program:

```
#!/usr/bin/python
from __future__ import print_function   # Need this for Python 2.x (print function)

def fib(n): # write Fibonacci series up to n
   """Print a Fibonacci series up to n."""
   a, b = 0, 1
   while a < n:
      print (a,end=' ')
      a, b = b, a+b
   print()

# Call the user-defined function with user argument
num = input('Enter a number: ')
fib(num)
```

- The above program will produce the following output:

  # ./fibo2.py
  Enter a number: 5000
  0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
  #

- **Python** uses a mechanism, which is known as "**Call-by-Object**", sometimes also called "**Call by Object Reference**" or "**Call by Sharing**".

- If you pass **immutable** arguments like **integers**, **strings** or **tuples** to a **function**, the passing acts like **call-by-value**. This means that the original argument will remain unchanged.

- The following example illustrates this:

```
#!/usr/bin/python

# User-defined function
def ArgsChange( FuncList ):
   "This changes a passed list into this function"
   FuncList = [1,2,3,4]; # This would assig new reference in mylist
   print "Values inside the function: ", FuncList
   return

# Now you can call function with an immutable argument
Origlist = [10,20,30];
ArgsChange( Origlist );
print "Values after the function call: ", Origlist
```

- The output:

```
# ./args.py
Values inside the function:  [1, 2, 3, 4]
Values after the function call:  [10, 20, 30]
```

- On the other hand, a **List** is a **mutable** type, which means that it is **passed by reference**. This means that the original argument will be changed by whatever operation the function performed on the **list**.

- In the following program a **mutable** argument is being passed by **reference** and the **list** is being overwritten inside the called function:

```
#!/usr/bin/python

# Pass by reference mutable example
def ChangeList(TheList):
   print 'Received', TheList
   TheList.append('Python')
   print 'Changed to', TheList

OrigList = ['C', 'C++', 'Ruby']

print 'Before function call: OrigList =', OrigList
ChangeList(OrigList)
print 'After function call, OrigList =', OrigList
```

- The output:

```
# ./list1.py
Before function call: OrigList = ['C', 'C++', 'Ruby']
Received ['C', 'C++', 'Ruby']
Changed to ['C', 'C++', 'Ruby', 'Python']
After function call, OrigList = ['C', 'C++', 'Ruby', 'Python']
```

lambda functions

- Python supports an interesting syntax that allows you to define one-line mini-functions. This a feature borrowed from Lisp; **lambda** functions can be used anywhere a function is required.

- The function has no name, but it can be called through the variable it is assigned to:

```
>>> square  = lambda x: x**2
>>> square(4)
16


>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> num = make_incrementor(42)
>>> num(1)
43
>>> num(10)
52
>>>
```

- A **lambda** function accomplishes the same thing as the normal functions already discussed. They have an abbreviated syntax, and there are no parentheses around the argument list, and the **return** keyword is omitted (it is implied, since the entire function can only be **one expression**).

Classes and Methods

- Before discussing classes it may be useful to summarize some the terminology that goes with Object-oriented-programming:

    - When we first describe a class, we are ***defining*** it (like with functions)
    - The ability to group similar **functions** and **variables** together is called ***encapsulation***
    - The word **'class'** can be used when describing the code where the class is defined; it can also refer to an **instance** of that class
    - A variable inside a class is known as an ***Attribute***
    - A function inside a class is known as a ***method***
    - A class is in the same category of things as variables, lists, dictionaries, etc. That is, they are ***objects***

- Class objects support two kinds of operations: **attribute references** and **instantiation**. Attribute references use the standard syntax used for all attribute references in Python: **obj.name**.

- Valid attribute names are all the names that were in the class's namespace when the class object was created.

- Consider the following class definition:

```
class MyClass:
        # A simple example class
        num = 12345
        def func(self):
                return 'hello world'
```

- Thus, **MyClass.num** and **MyClass.func** are valid attribute references, returning an integer and a function object, respectively. More about "**self**" later.

- Class **instantiation** uses function notation. Think of the class object as a function without parameters that returns a new **instance** of the **class**.

- For example (assuming the above class):

    c = MyClass()

    creates a new **instance** of the class and assigns this object to the local variable **c**.

- The instantiation operation ("calling" a class object) creates an empty object. Many classes create objects with instances customized to a specific **initial state**.

- Therefore a class may define a special method named __init__(), as follows:

```
def __init__(self):
        self.data = []
```

- When a class defines an **__init__()** method, class instantiation automatically invokes **__init__()** for the newly-created class instance.
- The **__init__()** method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to **__init__()**. (covered in a later example)

- In Python, each **method** is associated with a **class** and is intended to be invoked on **instances** of that class. A method is a function that takes a class instance as its first parameter. Methods are members of classes.

- **Methods** are just like **functions**, with two differences:

    - Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.

    - The syntax for invoking a method is different from the syntax for calling a function.
- The following is a basic illustration of a class with a method defined inside it:

```
#!/usr/bin/python

class C:
   def the_language(self):
      print "god intended us to code in C"

c = C()              # instantiate a new class "c" of class type C
c.the_language()    # execute the_language" method in the C class
```

- The class C contains a single method named "**the _language**", which simply prints a string.

- The "**c = C()**" creates a new object (instantiates) which is a class of type "**C**". "**c**" now has a method named "**the _language**".

- Now, calling **c.the_language()**; in python is the same as calling **C.the_language(c)**; thus **self** refers to **c**. (in other words, it just says, "print yourself")

- Finally, "**self**" is the instance object **automatically** passed to the class instance's method when called, to identify the instance that called it.

- "**self**" is used to access other attributes or methods of the object from inside the method. (methods are basically just functions that belong to a class)

- The following example covers the concepts presented so far:

```
>>> class FullName:
...     def __init__(self, first, last):
...             self.f = first
...             self.l = last
...
>>> name = FullName("Enoch", "Root")
>>> name.f, name.l
('Enoch', 'Root')
```

- The example provided (**shapes1.py**) is a more complete example that illustrates all of the concepts discussed so far.

- The code first creates an ***instance*** of a class by first giving its name (**Shape**) and then, in brackets, the values to pass to the **__init__** function. The ***init*** function runs (using the parameters you gave it in brackets) and then creates an ***instance*** of that class, which in this case is named **rectangle**.

- The class instance, **rectangle**, is a self-contained collection of variables and functions. In the same way that we used **self** to access functions and variables of the class instance from within itself, we use the name **rectangle** to access functions and variables of the class instance from ***outside*** of itself.

Inheritance

- One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.

- This also provides an opportunity to reuse the code functionality and fast implementation time. When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class.

- This existing class is called the **base** class, and the new class is referred to as the **derived** class. The syntax for a derived class definition is as follows:

  class DerivedClassName(BaseClassName):
  <**statement**-**1**>
  .
  .
  .
  <**statement**-**N**>

- Python makes inheritance relatively simple. We define a new class, based on another, **'parent'** class. The new class brings everything over (**inherits**) from the parent, and now we can also add other objects to it.

- If any new attributes or methods have the same name as an attribute or method in our parent class, it is used instead of the parent one.

- Using the previous example (**shapes1.py**) we will create a new class named "**Square**" which inherits from the "**Shape**" parent class.

- "**Square**" inherited everything from the "**Shape"** class, and changed only what needed to be changed. In this case we redefined the **__init__** function of Shape so that the X and Y values are the same.

- Now use all of the inherited methods to calculate the geometry of the square. The complete example is provided (**shapes2.py**).

Python Multithreading

- There are two modules which support the usage of threads in Python:
    - thread
    - threading
- The thread module is now considered as deprecated and is not available in Python 3.x)
- We will be using the threading module in this course. The newer threading module provides much more powerful, high-level support for threads.
- The following are some of the most commonly used methods provided by the ***Thread*** class:
    - **run():** This is the entry point for a thread.
    - **start():** Starts a thread by calling the run method.
    - **join([time]):** Waits for threads to terminate.
    - **isAlive():** Checks to see whether or not a thread is still executing.
    - **getName():** Returns the name of a thread.
    - **setName():** Sets the name of a thread.
- The following steps are required to create a new thread using the threading module:
    - Define a new subclass of the **Thread** class.
    - Override **the *__init__(self [,args])*** method to add additional arguments.
    - Then, override the ***run(self [,args])*** method to implement what the thread should do when started.
- Once you have created the new *Thread* subclass, you can create an instance of it and then start a new thread by invoking the ***start()***, which in turn calls ***run()*** method.
- The example provided (**mthreads1.py**) illustrates all of the steps above. The example creates a thread class (**myThread**) that the newly created threads will instantiate.
- Within the class both threads will execute the ***print_time*** function concurrently. Note that since there is no thread synchronization, the sequence of the time output will vary every time the program is executed.

- The following is a sample output:

```
# ./mthreads1.py
Starting Thread-1
 Exiting Main Thread
Starting Thread-2
Thread-1: Sat Jan 23 17:49:43 2016
Thread-2: Sat Jan 23 17:49:43 2016
Thread-1: Sat Jan 23 17:49:44 2016
Thread-2: Sat Jan 23 17:49:44 2016
Thread-1: Sat Jan 23 17:49:45 2016
Thread-2: Sat Jan 23 17:49:45 2016
Thread-1: Sat Jan 23 17:49:46 2016
Thread-2: Sat Jan 23 17:49:46 2016
Thread-1: Sat Jan 23 17:49:47 2016
Exiting Thread-1
Thread-2: Sat Jan 23 17:49:47 2016
Exiting Thread-2
```

Thread Synchronization

- The threading module provided with Python includes a simple-to-implement locking mechanism that allows us to synchronize threads.

- A new lock is created by calling the ***Lock()*** method, which returns the new lock (note that the **mutex** is now deprecated).

- Locks are the most fundamental synchronization mechanism provided by the **threading** module. At any time, a lock can only be held by a single thread. If a thread attempts to hold a lock that's already held by some other thread, execution of the first thread is blocked until the lock is released.

- The ***acquire*** method of the new lock object is used to enforce synchronization with multiple threads. The optional ***blocking*** parameter enables us to control whether the thread waits to acquire the lock.

- If ***blocking*** is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1 (default), the thread blocks and waits for the lock to be released.

- The ***release()*** method of the new lock object is used to release the lock when it is no longer required. This will release the lock and allow a blocked thread to acquire the lock and enter a critical section of code.

- Locks are typically used to synchronize access to a shared resource. A **Lock** object is created for each shared resource (critical code).

- When a thread needs to access the resource, it calls **acquire** to hold the lock (this will wait for the lock to be released, if necessary), and then calls **release** to release the lock. The following pseudocode illustrates this mechanism:

```
lock = Lock()

lock.acquire() # will block if lock is already held
        ... critical code shared by multiple threads
lock.release()
```

- Good coding practices dictate that it is important to release the lock in the event the current thread does not get to **lock.release** on its own. This prevents having the rest of the blocked threads from waiting indefinitely.

- The **try-finally** construct is very useful in implementing this:

```
lock.acquire()
try:
        ... critical code shared by multiple threads
finally:
   lock.release() # release lock unconditionally
```

- The example provided (**mthreads2.py**) illustrates all of the steps above. The example is basically the previous example with the synchronization code added in.

- The "**critical code**" in this example is the execution of the ***print_time*** function, only one thread will be able to execute that function at a time.

- This can be seen in the following sample output:

```
# ./mthreads2.py
Starting Thread-1
Starting Thread-2
Thread-1: Sat Jan 23 17:53:50 2016
Thread-1: Sat Jan 23 17:53:51 2016
Thread-1: Sat Jan 23 17:53:52 2016
Thread-2: Sat Jan 23 17:53:53 2016
Thread-2: Sat Jan 23 17:53:54 2016
Thread-2: Sat Jan 23 17:53:55 2016
Exiting Main Thread
```

File Input/Output

- The open() function returns a **file object**, and is usually used with two arguments:

  open(filename, mode).

- The first argument is a string containing the **filename**.

- The second argument is, **mode**, specifies the operation to be performed on the file:

  - '**r**' : open the file for reading only
  - '**w**': open the file for writing only (overwrites the existing file)
  - '**a**' : opens the file for appending (data is added to the end)
  - '**r+**': opens the file for both reading and writing.

- The **mode** argument is optional; 'r' will be assumed if it's omitted.

- By default, files are opened in **text mode**, which means that strings are read and written from and to the file, which are encoded in UTF-8 (character encoding capable of encoding all possible characters) by default.

- A 'b' appended to the mode opens the file in **binary mode** (useful for files such as JPEG or EXE)

- **File Object Methods**

  - There are a variety of methods that can be used on the file object. The following examples illustrate these.

  - To read a file's contents, call **fobj.read(size)**, where "**size**" specifies the amount of data to be read (**size** is an optional numeric argument; if omitted or negative, the entire contents of the file will be read and returned):

    ```
    fobj = open ('basho.txt', 'rw')
    >>> fobj.read()
    "A monk sips morning tea,\nit's quiet,\nthe chrysanthemum's flowering.\n\n"
    >>> fobj.read()
    ''
    ```

  - If the end of the file has been reached, **fobj.read()** will return an empty string ('').

- **f.readline()** reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline:

```
>>> fobj = open ('basho.txt', 'r')
>>> fobj.readline()
'A monk sips morning tea,\n'          # First line
>>> fobj.readline()
"it's quiet,\n"                       # Second line
>>> fobj.readline()
"the chrysanthemum's flowering.\n"    # Third line
>>> fobj.readline()
'\n'                                  # newline char
>>> fobj.readline()
''                                    # Empty string – End of line
>>>
```

- **f.readlines()** returns a **list** containing all the lines of data in the file. If given an optional parameter **sizehint**, it reads that many bytes from the file and enough more to complete a line, and returns the lines from that.

- This is often used to allow efficient reading of a large file by lines, but without having to load the entire file in memory. Only complete lines will be returned:

```
fobj = open ('basho.txt', 'r')
>>> fobj.readlines()
['A monk sips morning tea,\n', "it's quiet,\n", "the chrysanthemum's flowering.\n",
'\n']
>>>
```

- An alternative approach to reading lines is to loop over the file object. This is memory efficient, fast, and leads to simpler code:

```
>>> fobj = open ('basho.txt', 'r')
>>> for line in fobj:
...     print line
...
A monk sips morning tea,

it's quiet,

the chrysanthemum's flowering.
```

- If you want to read all the lines of a file in a list you can also use **list(fobj)** or **fobj.readlines().**

- **fobj.write(string)** writes the contents of **string** to the file:

```
>>> fobj = open ('foo.txt', 'w+')       # 'w+' creates the file if it does not exist
>>> fobj.write('Do\'nt panic\n')        # Must escape the ' character
>>> fobj.write('The answer is 42\n')
>>>
```

- To write something other than a string, it needs to be converted to a string first:

```
fobj = open('foo.txt','a')
>>> value = ('the answer is', 42)
>>> sval = str(value)
>>> fobj.write(sval)
>>>
```

- **fobj.tell()** returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file.

- To change the file object's position, we can use **fobj.seek(offset, from_what)**. The position is computed from adding **offset** to a reference point; the reference point is selected by the **from_what** argument.

- A **from_what** value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. **from_what** can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> fobj = open ('foo.txt', 'r+')
>>> fobj.read(1)
'D'
>>> fobj.tell()
1
>>> fobj.seek(6)          # Go to the 7th byte
>>> fobj.tell()
6
>>> fobj.read(2)          # Read 2 bytes at that position
'pa'
>>> fobj.seek(-3, 2)      # Go to the 3rd byte before EOF
>>> fobj.read(1)
'4'
>>>
```

- It is always important to close the file when all the operations are done, or at the end of the program:

```
fobj.close()
```