1 Strings (Review)

Strings are a collection of characters. In python, strings can be delineated by either single or double quotes.

- `'this is a string'`
- `"this is also a string"`

If you use one type of quote marker to delineate the string, you can use the other type of quote marker as part of the string.

```
'you can put "quotes" around a word'
"in one of 'two' ways"
```

Each character in a string is typically represented as a byte (8 bits) of memory. Once upon a time in computer history someone came up with a mapping from numbers to characters. The most common mapping is called ASCII [American Standard Code for Information Interchange]. ASCII uses 8-bits, or one byte to represent each character. You can find out what the ASCII code is for a character by using the `ord()` function.

A string is simply a collection of characters that python has put in consecutive memory locations inside the computer. The name of the string is associated with the address of the first character in the string. When data is stored conceptually (or physically) as a sequence of elements we call that an array.

Python provides a notation using square brackets that allow us to look at the specific elements in an array. The bracket notation is general across various types of arrays, or sequential things in python.

```
>>> a = 'abcd'
>>> a[0]
'a'
>>> a[1]
'b'
>>> a[2]
'c'
>>> a[3]
'd'
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
>>> a[-1]
'd'
```

Note what happens when we use an index that goes beyond the end of the array. Python generates an error and tells us our index (4) is too big for this array. Array indexes follow three rules.

- Indices must be integers (a[1.0] generates an error)
- The first element in a string is at index 0
- Indices must not try to access elements beyond the length of the string
- Positive indices from 0 to length-1 are valid and index the string from left to right.
- Negative indices from -1 to -length are valid and index the string from right to left.

We can always discover the number of characters in a string using the `len()` function.

```
>>> a = 'lots of characters'
>>> len(a)
18
```

One thing to be aware of is that python strings are **immutable**. That means you can't change the value of a character in a string once it's created. You can, however, build a completely new string and put the new string into the old variable. It's an idiosyncrasy of python.

As noted previously in lecture, string are objects, and object classes in Python can have methods. A method is simply a function that is attached to the object (and defined within its symbol table entry). Some useful string methods include the following.

- `split(key)` - splits the string based on the key character and returns a list of strings
- `count(value)` - counts the number of times value (whatever it is) appears in the list
- `index(value)` - returns the index of the first occurrence of value (whatever it is)
- `lower()` - return an all lowercase version of the string

2 for (Review)

The `for` loop provides a convenient syntax and mechanism for iterating over an array or a list of elements.

Syntax:

```
for <variable> in <list or array>:
    <statements>
```

The variable can be any legal variable name in python. The variable has scope only within the for loop and where it has scope it will override any variable of the same name defined elsewhere.

The list or array is a the collection of things over which the loop should iterate. The first time through the loop, the variable gets the value of the first item in the list or array. The second time through the loop, the variables gets the value of the second item in the list or array, and so on, until all of the elements have been processed.

To loop over the elements of a string, we can simply do something like:

```
for curChar in aString:
    print curChar
    < other stuff with the character curChar >
```

If you want to have the loop variable take on a set of consecutive numbers, you can use the built-in `range()` function.

```
range(<start>, <end>, <step)
```

- If you give the range function a single parameter, it generates a list that contains elements from 0 to one less than the given number in increments of 1.
- If you give the range function two parameters, it produces a list starting at the first number and ending at one less than the second number in increments of 1.

- If you give the range function three parameters, it produces a list starting at the first number, incrementing by step, and ending no less than step from the ending number.

Key concept: for loops work through a sequence of items

- The sequence can be a string, in which case it works through the elements of the string
- The sequence can be a list, which is (conceptually) a linear sequence of things
- The range() function is your friend

Simple lists are a comma separated sequence of items, surrounded by brackets. You can index them using the bracket notation, just like strings.

```
listOfNumbers = [1, 2, 3, 4, 5]
listOfStrings = ['ab', 'cd', 'de']
listOfMixture = [1, 'ab', 2, 'bc', 3.0]
```

Example: Use a for loop to iterate over the elements of a list and print out the value and type of each element.

```
>>> def showlist(mylist):
...     for item in mylist:
...             print item, " : ", type(item)
...
>>> alist = [1, "45", 45, "thirty", 30.0]
>>> showlist(alist)
1  :  <type 'int'>
45  :  <type 'str'>
45  :  <type 'int'>
thirty  :  <type 'str'>
30.0  :  <type 'float'>
```

With for loops in python, it is important to remember that both strings and lists can be used as the foundation of the for loop. For example, each of the following for loops does the same thing.

```
a = "abcdef"
for char in a:
print char

b = ['a', 'b', 'c', 'd', 'e', 'f']
for char in b:
print a
```

3 Lists

As noted above, lists are sequential collections of objects. What is an object?

- A standard data type: int, long, float, str, bool, func
- A programmer defined data structure called a class
- Another list

2D lists (and higher) provide lots of opportunities for organizing information.

Example: organizing image and effect information

Your collage functions take a lot of parameters. How could we better organize those as a list?

Current:

```
def collage4( file1, file2, file3, file4, e1, e2, e3, e4, centerfile ):
```

Why not use a list for the filenames and a list for the effects?

```
def collage4( files, effects, centerfile ):
```

This tidies up our parameter list a bit. However, we then have to ask why the files and effects are separate lists. If we pass them in as separate lists, then we need an implicit rule inside the function that the image and effect with the same index are related.

An alternative, that makes the relationship between image and effect more explicit is to pass in a single list, where each entry in the list is a list that contains the filename and the effect.

```
def collage4( imageInfo, centerfile ):
    # create the background image...

    for entry in imageInfo:
        filename = entry[0]
        effect = entry[1]
        # read in the image and draw it into the background image
```

We can create the list before calling the collage function as follows.

```
def main():

    f1 = pickAFile()
    f2 = pickAFile()
    f3 = pickAFile()
    f4 = pickAFile()
    f5 = pickAFile()

    data = [ [f1, 'negative' ], [ f2, 'swap'], [ f3, 'normal'], [ f4, 'glow'] ]
    bkg = collage4( data, f5 )
    show(bkg)
```

Note that it also raises the issue of whether collage4 has to take only four files. Could we write the function so it takes any number of files and effects?

The key idea here is that an index is like an id number. In a list, we can store whatever we like at a particular id number. If we need to store all of the information about an image to go into a collage (filename, effect, location, removeBlue, mirror), we could do that. All we need to do is be consistent about how we store it.

Consider the collage function from project 4. What if we stored all of the information about each image in a list and then passed the list of lists into the function.

```
images = [ ['pic1.jpg', 'negative', True, 0, 0, 1.0],
           ['pic2.jpg', 'swap', True, 50, 100, 0.75] ]
```

In the above case, images[0] would provide the list of arguments for the first image. Each argument would be indexed as images[0][i] where i is the index of the argument.

To modify a particular argument, the same notation is used only on the left side of an assignment statement.

```
>>> images[0][1] = 'normal'
>>>images
[['pic1.jpg', 'normal', True, 0, 0, 1.0],
 ['pic2.jpg', 'swap', True, 50, 100, 0.75]]
```

In summary, both lists and strings are objects over which we can iterate using a for loop. The difference between lists and strings is twofold.

- The elements of a string cannot be changed, while the elements of a list are mutable.
- A string can hold only characters, while the elements of a list can be any object.

3.1 Lists as Objects

As noted above, a list is also an object. In python, an object can have methods, which are functions that are attached to the particular object type. For example, all lists have the method `append( val )`, which appends its argument to the end of the list.

Other list methods include the following:

- `append(object)` - appends object (whatever it is) to the end of the list
- `count(value)` - counts the number of times value (whatever it is) appears in the list
- `index(value)` - returns the index of the first occurrence of value (whatever it is)
- `pop()` - remove and return the last value from the list
- `reverse()` - reverse the elements of a list in place (modifies the ordering of the list)
- `sort()` - sort the items of the list in place (modifies the ordering of the list)

In each case, the function is called by a list object using dot-notation. In other words, if `mylist` is a list, then to append the number 1 to the end of the list we could use `mylist.append( 1 )` . Python will give you an error if you try to call a method that is not defined for that object type.

The concept of object methods makes our symbol table a bit more complex: we not only have type information, but also method information stored under each object. Methods are at the heart of object-oriented design, however, which is very successful methodology.

4 File I/O

Files are streams of data. The important thing is how our program interprets the data.

Text files are just big long strings, just stored on a hard drive instead of in a variable in memory. To access a file, we just need to know its name and where it is in the file tree. Just like we can navigate through the file tree of our computer using the terminal, we can use the same method of specifying paths in python. The directory from which you run your program is also the working directory inside the program. Any files in the working directory can be accessed by just their filename. All other files either need to be specified relative to the file tree root (/) or relative to the current working directory.

It's useful to think of text files as strings separated by newlines.

There is a built-in data type called file that lets us open, read, write, and close files.

The file data type is a class. In most respects it's no different than an int or a float. To create a data type of a particular class, you use the name of the class as a function.

Besides the method of creating classes, the main difference between classes and the basic data types is that you can give classes methods. A method is just a function that belongs to a class. Think of it as a trick that the data type knows how to do.

Conceptually, if we had a class called dog, we might be able to do the following:

```
fido = dog()
fido.shake()
fido.fetch()
```

fido is just a variable, a label for a memory location. The first line assigns the variable an object of type dog. The class dog has certain methods that someone has written that do things. We can have the object execute those functions using the dot notation and the method name.

The file class works the same way.

```
fp = file('myfilename', 'w')
fp.write('writing a string to the file\n')
fp.close()
```

Reading files works about the same way.

```
fp = file('myfilename', 'r')
lineOfText = fp.readline()
print 'The contents of the file are:'
print lineOfText
fp.close()
```

Files use something called a file pointer to keep track of where you are in the file. When you read a line from the file, if you call readline again, it reads the next line of the file. The file pointer always moves to point to the next thing that will be read from the file. When you open a file using the 'r' or 'w' mode, the file pointer starts at the beginning of the file. If you open a file using the 'a' mode, the file is opened for writing, but the file pointer starts at the end. Thus, the 'a' mode is for appending things to a file (although it does create the file if the file does not exist).

Trying to open a file for reading that doesn't exist generates an IOError exception. When opening a file, it's always a good idea to try and catch IOError exceptions and handle them gracefully. We can use the

try/except structure to catch errors.

```
try:
    fp = file(filename, 'w')
except IOError:
    print 'Unable to open file', filename
    return
```

When reading a file, there are three `read` methods from which to choose.

- read(): reads the rest of the file and returns it as a single string
- readline(): reads from the file up to, and including, the next newline character and returns the string
- readlines(): reads each line from the file and returns a list with each line as an entry in the list

Note that the first and last functions could return very large objects.

One way to think of files is as a stream of data. A text file is just a stream of characters, one after the other. Python lets us treat files like a string or a list when it comes to for loops. You can use a file object as the list or string and the loop variable then becomes the current line of the file.

```
fp = file('myfilename', 'r')

for line in fp:
    print line

fp.close()
```

This is an easy way to go through each line of a file and process it. If you wanted to go through each character of the file, you would need to go through each line with a second for loop nested inside the first.

```
fp = file('myfilename', 'r')

for line in fp:
    for char in line:
        print char,

fp.close()
```

The ability to read in a file as a series of lines tucked into a list (the `readlines()` function) provides an easy way to sort the lines of a file. How?

If we read the lines of a file into memory using the readlines() function, then we can apply the sort() method of the list and then write out each line back to a file using a for loop over the elements of the list.

How do you find this stuff out? Python has a help facility built into it. For any data type, you can type

```
help( type )
```

and it will print out information about the data type. Try it out with the `str`, `list`, or `file` data type and see what other functions are there. Any of these built-in methods you can apply to any object of that type.

Example: saving your work

In an interactive program you may be entering a series of commands. At the end you have a shape you like, but the only way to save the picture is to make a screen capture.

Why not store the commands the user enters and save the whole series of commands to a file when the user quits?

```
def main2():
    print 'starting'
    turtleSetup(500, 500, 200, 200)

    # initialize the variable in which to save the commands
    memory = ''

    while True:
        print 'Enter string to process: ',
        s = raw_input()

        # break out of the loop if the user entered nothing
        if len(s) == 0:
            break

        # process the string
        processString(s)

        # add the current command to memory
        memory += s

    print 'Writing list of commands to file memory.txt'

    # open a file, write the string, and close the file
    fp = file( 'memory.txt', 'w' )
    fp.write(memory)
    fp.close()

    print 'Terminating'
```

What if we wanted to start the whole process with a string read from a file?

4.1 Random Numbers

Random numbers play a big role in computer science, especially for simulating natural phenomena. The real world contains randomness, and in order to simulate the real world we also need to simulate these phenomena. In most computer systems, random numbers are generated using mathematical computations that generate sequences of what are called pseudo-random numbers. The number sequences have properties similar to true random numbers, but they are computed deterministically and, eventually, will start to repeat. The length of the sequence, however, tends to be very large.

Python's random package, for example, uses a method that has a period of $2^{19937} - 1$. So no simulation or computer program you can write will go through the entire period in your lifetime.

The fact that random number generation is deterministic, however, means we can actually repeat a sequence of random numbers if we reset the random number generator to a specific state. This ability can be important when testing a program so that we can compare different runs when looking for bugs.

It is also important to know this because, if you want your program to act differently each time it is run, you may want to seed the random number generator using something that will be different each time the program is run. The current time, for example, is commonly used to seed random number generators.

Example: use of the time package to set a random seed

- Note the use of the import <package> syntax
- Note that the package is treated as an object and functions in the package as methods
- The time package method `time()` returns time since the Epoch as a float
- The Epoch is the start of January 1st, 1970

```
import time

random.seed(time.time())
```

We've already used the ability of the random package to generate random numbers in the range $[0, 1)$.

- The use of a bracket in range notation means the range includes the number on that end of the range
- The use of a parenthesis in range notation means the range does not include the number on that end of the range
- In the case above, the function returns numbers from 0.0 to almost 1.0, but not including 1.0

There are many other functions available in the random package that are useful in various situations.

- `random.uniform(a, b)` - returns a floating point number in the range $[a, b)$
- `random.randint(a, b)` - returns an integer in the range $[a, b]$
- `random.gauss(mu, sigma)` - returns a floating point value drawn from a Gaussian, or normal distribution parameterized by its mean (mu) and standard deviation (sigma)
- `random.choice( list )` - returns a randomly selected element of the argument, which should be a list or string

- `random.shuffle( list )` - shuffles the elements of the list or string in place
- `random.sample( list, N )` - returns a randomly selected set of N elements from the list

Review

Variables are labels for locations where data can be stored. In python, any kind of data can be stored in any variable. The rules for variable names determine what kinds of labels you can use.

- A variable is created when you assign something to it.
  `agaboo = 15`
- If you try to use a variable before it has been created, it will cause an error
- To modify the value of a variable, use an assignment statement (=). Whatever is on the right side of an assignment gets put into the variable on the left side.
- Variables can hold data types that consist of sequential elements (lists, strings)
  - Use the bracket notation to access a particular element in the sequence
    ```
    abor = [1, 2, 3]
    print abor[1]
    ```
  - Lists and strings also have methods associated with them that can be called
  - You can use the help() function in python to learn about the methods
- Variables can hold data types that are objects, or classes
  - Classes can contain data of various types
  - Classes can have methods associated with them
  - Use the dot notation (variablename.methodname()) to call a method of a class
- Strings are different than variables
  - A string is denoted by quotes (single, or double)
  - A string is a sequence of characters, it is what it is and nothing else
  - A variable is a label for a memory location that can hold data
  - 'varname' is not the same thing as varname

Functions are one way to organize programs. They let you subdivide a program into smaller and smaller bits until the individual bits are simple to write. Functions can also return values.

- You can use a function that returns a value on the right side of an assignment statement.
- The return value of the function is used to evaluate the expression
- The function is called before the expression can be evaluated
- If you want to use the return value of a function, you have to store it in a variable using an assignment
  `huki = myfunction(5)`
- To return a value from a function, use the `return` keyword