Quick Review of Last Class

- **String** data type: str = “hello Bob”

Operator	Meaning	Eg. ?	
+	Concatenation	“hi”+” Bob”	'hi Bob'
*	Repetition	2*”hi”	'hihi'
<string>[]	Indexing	(start from ?)	0
<string>[:]	Slicing	str[0:-1] ?	"hello Bo"
len(<string>)	Length	len(str)	‘9'
for <var> in <string>	Iteration through characters		

Quick Review of Last Class

- **List:**
- Lists are more general than strings
- Strings are always sequences of characters, but ***lists*** can be sequences of arbitrary values.
  - **Lists** can have **numbers, strings, or both**!
    - myList = [1, "Spam ", 4, "U"]
- Lists are ***mutable***, meaning they can be changed. **Strings can not be changed**.

Quick Review of Last Class

- **Split()**
- **Split** can be used on characters other than space, by supplying the character as a parameter.

```
>>> "32,24,25,57".split(",")
['32', '24', '25', '57']
```

Quick Review of Last Class

- **Append** method can be used to add an item at the end of a list
- squares = []
- For x in range (1,5):
  - squares.append(x*x)

  ```
  >>> squares
  [1, 4, 9, 16]
  ```

```
For x in range (1,5):
        squares.append(x*x)
        Print(squares)
        >>>
        [1]
        [1, 4]
        [1, 4, 9]
        [1, 4, 9, 16]
```

String Representation and Message Encoding

- The ***ord* function** returns the numeric (**ordinal**) code of a single character.
- The ***chr* function** converts a numeric code to the corresponding character.

```
>>> ord("A")
65
>>> ord("a")
97
>>> chr(97)
'a'
>>> chr(65)
'A'
```

String Representation and Message Encoding

- Using **ord** and **char** we can convert a string into and out of numeric form.

- **The encoding algorithm is simple**:

```
get the message to encode
for each character in the message:
   print the letter number of the character
```

- A for loop iterates over a sequence of objects, so the for loop looks like:

```
for ch in <string>
```

- **Input/Output as String Manipulation**

Input/Output as String Manipulation

- Often we will need to do some string operations to prepare our string data for output ("pretty it up")
- Let's say we want to enter a date in the format "09/24/2014" and output "Sep 24, 2014." How could we do that?

Input/Output as String Manipulation

- Input the date in mm/dd/yyyy format (dateStr)
- Split dateStr into month, day, and year strings

- Convert the month string into a month number
- Use the month number to lookup the month name

- Create a new date string in the form "Month Day, Year"
- Output the new date string

Input/Output as String Manipulation

- The first two lines are easily implemented!
- dateStr = input("Enter a date (mm/dd/yyyy): ")
  monthStr, dayStr, yearStr = dateStr.split("/")
- The date is input as a string, and then "unpacked" into the three variables by splitting it at the slashes and using simultaneous assignment.

Input/Output as String Manipulation

- Next step: Convert monthStr into a number

- We can use the *int* function on monthStr to convert "05", for example, into the integer 5. (int("05") = 5)

Input/Output as String Manipulation

- Note: eval would work, but for the leading 0

```
>>> int("05")
5
>>> eval("05")
 Traceback (most recent call last):
File "<pyshell#9>", line 1, in <module>
eval("05")
File "<string>", line 1
05
^
SyntaxError: invalid token
```

Input/Output as String Manipulation

```
months = [“January”, “February”, …, “December”]
monthStr = months[int(monthStr) – 1]
```

- Remember that since we start counting at 0, we need to subtract one from the month.
- Now let’s concatenate the output string together!

Input/Output as String Manipulation

```
print ("The converted date is:", monthStr, dayStr+",", yearStr)
```

- Notice how the comma is appended to dayStr with concatenation!

-
```
>>> main()
Enter a date (mm/dd/yyyy): 01/23/2014
The converted date is: January 23, 2014
```

Input/Output as String Manipulation

- Sometimes we want to convert a number into a string.

- We can use the ***str*** function.

```
>>> str(500)
'500'
>>> value = 3.14
>>> str(value)
'3.14'
>>> print("The value is", str(value) + ".")
The value is 3.14.
```

Input/Output as String Manipulation

- If value is a string, we can concatenate a period onto the end of it.

- If value is an int, what happens?

```
>>> value = 3.14
>>> print("The value is", value + ".")
The value is

Traceback (most recent call last):
  File "<pyshell#10>", line 1, in -toplevel-
    print "The value is", value + "."
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Input/Output as String Manipulation

- We now have a complete set of type conversion operations:

Function	Meaning
float(<expr>)	Convert expr to a floating point value
int(<expr>)	Convert expr to an integer value
str(<expr>)	Return a string representation of expr
eval(<string>)	Evaluate string as an expression

- String Formatting

String Formatting

- String formatting is an easy way to get beautiful output!

```
Change Counter

Please enter the count of each coin type.
Quarters: 6
Dimes: 0
Nickels: 0
Pennies: 0

The total value of your change is 1.5
```

- Shouldn't that be more like $**1.50**??

String Formatting

- We can format our output by modifying the print statement as follows:

```
print("The total value of your change is ${0:0.2f}".format(total))
```

- Now we get something like:

```
The total value of your change is $1.50
```

- Key is the string format method.

String Formatting

- <template-string>.format(<values>)
- {} within the template-string mark “slots” into which the values are inserted.
- Each slot has description that includes *format specifier* telling Python how the value for the slot should appear.

String Formatting

```
print("The total value of your change is ${0:0.2f}".format(total)
```

- The template contains a single slot with the description: 0:0.2f
- Form of description:
  <index>:<format-specifier>
- Index tells which parameter to insert into the slot. In this case, total.

String Formatting

- <index>:<format-specifier>
- The formatting specifier has the form: <width>.<precision><type>
- f means "fixed point" number
- <width> tells us how many spaces to use to display the value. 0 means to use as much space as necessary.
- <precision> is the number of decimal places.

String Formatting

```
>>> "Hello {0} {1}, you may have won ${2}" .format("Mr.", "Smith",
    10000)
'Hello Mr. Smith, you may have won $10000'

>>> 'This int, {0:5}, was placed in a field of width 5'.format(7)
'This int,     7, was placed in a field of width 5'

>>> 'This int, {0:10}, was placed in a field of witdh 10'.format(10)
'This int,         10, was placed in a field of witdh 10'

>>> 'This float, {0:10.5}, has width 10 and precision 5.'.format(3.1415926)
'This float,     3.1416, has width 10 and precision 5.'

>>> 'This float, {0:10.5f},  is fixed at 5 decimal places.'.format(3.1415926)
'This float,     3.14159, has width 10 and precision 5.'
```

String Formatting

- If the width is wider than needed, numeric values are right-justified and strings are left- justified, by default.

- **You can also specify a justification before the width.**

```
>>> "left justification: {0:<5}.format("Hi!")
'left justification: Hi!  '
>>> "right justification: {0:>5}.format("Hi!")
'right justification:   Hi!'
>>> "centered: {0:^5}".format("Hi!")
'centered:  Hi! '
```

String Formatting

- The formatting specifier has the form: <width>.<precision><type>
- **{0:0.2f} .format(1.5)**
- f means "fixed point" number
- <width> tells us how many spaces to use to display the value. 0 means to use as much space as necessary.
- <precision> is the number of decimal places.

- Multi-Line Strings

Files: Multi-line Strings

- A *file* is a sequence of data that is stored in secondary memory (disk drive).
- Files can contain any data type, but the easiest to work with are text.
- A file usually contains more than one line of text.
- Python uses the standard newline character (\n) to mark line breaks.

Multi-Line Strings

- Hello
  World

  Goodbye 32

- When stored in a file:
  Hello\nWorld\n\nGoodbye 32\n

Multi-Line Strings

- This is exactly the same thing as embedding \n in print statements.
- Remember, these special characters only affect things when printed. They don't do anything during evaluation.

File Processing

- The process of *opening* a file involves associating a file on disk with an object in memory.

- We can manipulate the file by manipulating this object.
  - Read from the file
  - Write to the file

File Processing

- When done with the file, it needs to be *closed*. Closing the file causes any outstanding operations and other bookkeeping for the file to be completed.
- In some cases, not properly closing a file could result in data loss.

File Processing

- Reading a file into a word processor
  - File opened
  - Contents read into RAM
  - File closed
  - Changes to the file are made to the copy stored in memory, not on the disk.

File Processing

- Saving a word processing file
  - The original file on the disk is reopened in a mode that will allow writing (this actually erases the old contents)
  - File writing operations copy the version of the document in memory to the disk
  - The file is closed

File Processing

- Working with text files in Python
  - Associate a disk file with a file object using the open function
    <filevar> = open(<name>, <mode>)
  - Name is a string with the actual file name on the disk. Sometimes you need to specify the path of this file. The mode is either ‘r’ or ‘w’ depending on whether we are reading or writing the file.
  - Infile = open("numbers.dat", "r")

File Methods

- <file>.read() – returns the entire remaining contents of the file as a single (possibly large, multi-line) string
  - Eg. Infile.read()
- <file>.readline() – returns the next line of the file. This is all text up to *and including* the next newline character

- <file>.readlines() – returns a list of the remaining lines in the file. Each list item is a single line including the newline characters.

File Processing

```
# printfile.py
#     Prints a file to the screen.

def main():
    fname = input("Enter filename: ")
    infile = open(fname,'r')
    data = infile.read()
    print(data)

main()
```

- First, prompt the user for a file name
- Open the file for reading
- The file is read as one string and stored in the variable data

File Processing

- readline can be used to read the next line from a file, including the trailing newline character

-
  ```
  infile = open(someFile, "r")
  for i in range(5):
          line = infile.readline()
          print (line[:-1])
  ```
- This reads the first 5 lines of a file

- Slicing is used to strip out the newline characters at the ends of the lines

File Processing

- Another way to loop through the contents of a file is to read it in with readlines and then loop through the resulting list.

-
  ```
  infile = open(someFile, "r")
  for line in infile.readlines():
      # Line processing here
  infile.close()
  ```

File Processing

- Python treats the file itself as a sequence of lines!

-
  ```
  Infile = open(someFile, "r")
  for line in infile:
  	# process the line here
  infile.close()
  ```

File Processing

- Opening a file for writing prepares the file to receive data
- If you open an existing file for writing, you wipe out the file's contents. If the named file does not exist, a new one is created.
- Outfile = open("mydata.out", "w")
- print(<expressions>, file=Outfile)