Course Name: Practical Python Programming

Week 2: Interactive and I/O

a. Let Python Asks The Questions.

Now we are going to pick up the pace. We did a lot of printing so that you get used to typing simple things, but those simple things are fairly boring. What we want to do now is get data into your programs. This is a little tricky because you have to learn to do two things that may not make sense right away, but trust me and do it anyway. It will make sense in a few exercises.

Most of what software does is the following:

1. Take some kind of input from a person.
2. Change it.
3. Print out something to show how it changed.

So far we only did printing, but we haven't been able to get any input from a person or change it. You may not even know what "input" means, so rather than talk about it, let's have you do some and see if you get it. In the next exercise, we'll do more to explain it.

```
print "How old are you?",
age = raw_input()
print "How tall are you?",
height = raw_input()
print "How much do you weigh (lbs)?",
weight = raw_input()
print "So, you're %r years old, %r tall and %r (lbs) heavy." % (
age, height, weight)
```

b. Prompting User Inputs.

When we typed raw_input(), you were typing the (and) characters, which are parenthesis characters. This is similar to when you used them to do a format with extra variables, as in "%s %s" % (x, y). For raw_input, you can also put in a prompt to show to a person so he knows what to type. Put a string that you want for the prompt inside the () so that it looks like this:

y = raw_input("Name? ")

This prompts the user with "Name?" and puts the result into the variable y. This is how you ask someone a question and get the answer.
This means we can completely rewrite our previous exercise using just raw_input to do all the prompting.

```
age = raw_input("How old are you? ")
height = raw_input("How tall are you? ")
weight = raw_input("How much do you weigh (lbs)? ")
print "So, you're %r years old, %r tall and %r (lbs) heavy." % (age,
```

```
height, weight)
```

c. Parameters, Unpacking, Variables.

In this section of exercise, we will cover one more input method you can use to pass variables to a script (script being another name for your .py files). You know how you type *python week02c.py* to run the *week02c.py* file? Well the *week02c.py* part of the command is called an “argument.” What we’ll do now is write a script that also accepts arguments. Type this program and I’ll explain it in detail:

```
from sys import argv

script, first, second, third = argv
print "The script is called:", script
print "Your first variable is:", first
print "Your second variable is:", second
print "Your third variable is:", third
```

d. Prompting And Passing.

We will do one more exercise session on using argv and raw_input together to ask the user something specific. You will need this for the next exercise, where we learn to read and write fi les. In this exercise, we’ll use raw_input slightly differently by having it just print a simple > prompt.

```
from sys import argv

script, user_name = argv
prompt = '> '
print "Hi %s, I'm the %s script." % (user_name, script)
print "I'd like to ask you a few questions."
print "Do you like me %s?" % user_name
likes = raw_input(prompt)
print "Where do you live %s?" % user_name
lives = raw_input(prompt)
print "What kind of computer do you have?"
computer = raw_input(prompt)
print """
Alright, so you said %r about liking me.
You live in %r. Not sure where that is.
And you have a %r computer. Nice.
""" % (likes, lives, computer)
```

e. Reading Files.

This exercise involves writing two files. One is our usual week02e.py file that we will run, but the other is named week02e_sample.txt. This second file isn’t a script but a plain text file we’ll be reading in our script. Here are the contents of that file:

```
You can document a Python function by giving it a doc string.
```

Example 2.2. Defining the buildConnectionString Function's doc string

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
```

Triple quotes signify a multi-line string. Everything between the start and end quotes is part of a single string, including carriage returns and other quote characters. You can use them anywhere, but you'll see them most often used when defining a doc string.

> Note: Python vs. Perl: Quoting
> Triple quotes are also an easy way to define a string with both single and double quotes, like qq/.../ in Perl.

Everything between the triple quotes is the function's doc string, which documents what the function does. A doc string, if it exists, must be the first thing defined in a function (that is, the first thing after the colon). You don't technically need to give your function a doc string, but you always should. I know you've heard this in every programming class you've ever taken, but Python gives you an added incentive: the doc string is available at runtime as an attribute of the function.

> Note: Why doc strings are a Good Thing
> Many Python IDEs use the doc string to provide context-sensitive documentation, so that when you type a function name, its doc string appears as a tooltip. This can be incredibly helpful, but it's only as good as the doc strings you write.

What we want to do is “open” that file in our script and print it out. However, we do not want to just “hard code” the name week02e_sample.txt into our script. “Hard coding” means putting some bit of information that should come from the user as a string right in our program. That’s bad because we want it to load other files later. The solution is to use argv and raw_input to ask the user what file the user wants instead of “hard coding” the file’s name.

```
from sys import argv

script, filename = argv
txt = open(filename)
print "Here's your file %r:" % filename
print "Anything between the asterisk line is the read out of the file"
print "*****************************************************"
print txt.read()
print "*****************************************************"
print "Type the filename again:"
file_again = raw_input("> ")
txt_again = open(file_again)
print txt_again.read()
```

Lines 1–3 should be a familiar use of argv to get a filename. Next we have line 5 where we use a new command open. Right now, run pydoc open and read the instructions. Notice how like your own scripts and raw_input, it takes a parameter and returns a value you can set to your own variable. You just opened a file.
Line 7 we print a little line, but on line 8 we have something very new and exciting. We call a function on txt. What you got back from open is a file, and it’s also got commands you can give it. You give a fi le a command by using the . (dot or period), the name of the command, and parameters. Just like with open and raw_input. The difference is that when you say txt.read() you are saying, “Hey txt! Do your read command with no parameters!”

f. Reading And Writing Files.

Recap of what we have learned so far:

• close—Closes the file. Like File- >Save.. in your editor.
• read—Reads the contents of the file. You can assign the result to a variable.
• readline—Reads just one line of a text file.
• truncate—Empties the fi le. Watch out if you care about the fi le.
• write(stuff)—Writes stuff to the file.

```
from sys import argv

script, filename = argv
print "We're going to erase %r." % filename
print "If you don't want that, hit CTRL- C (^C)."
print "If you do want that, hit RETURN."
raw_input("?")
print "Opening the file..."
target = open(filename, 'w')
print "Truncating the file. Goodbye!"
target.truncate()
print "Now I'm going to ask you for three lines."
line1 = raw_input("line 1: ")
line2 = raw_input("line 2: ")
line3 = raw_input("line 3: ")
print "I'm going to write these to the file."
target.write(line1)
target.write("\n")
target.write(line2)
target.write("\n")
target.write(line3)
target.write("\n")
print "And finally, we close it."
target.close()
```

g. More Files.

Let's do a few more things with files. We're going to actually write a Python script to copy one file to another. It'll be very short but will give you some ideas about other things you can do with files.

```
from sys import argv
from os.path import exists

script, from_file, to_file = argv
print "Copying from %s to %s" % (from_file, to_file)
# we could do these two on one line too, how?
in_file = open(from_file)
indata = in_file.read()
print "The input file is %d bytes long" % len(indata)
print "Does the output file exist? %r" % exists(to_file)
print "Ready, hit RETURN to continue, CTRL- C to abort."
raw_input()
out_file = open(to_file, 'w')
```

```
out_file.write(indata)
print "Alright, all done."
out_file.close()
in_file.close()
```

h. Names, Variables, Code, Functions.

We will learn the Python function!. Just about every computer language has the “function” feature. The explanation on “function” can go on and on. The different ideas about how they work and what they do are also varies based on which computer programing language. However, to summerize them all, they all have these three things in common:.

Functions do three things:

1. They name pieces of code the way variables name strings and numbers.
2. They take arguments the way your scripts take argv.
3. Using #1 and #2, they let you make your own “mini- scripts” or “tiny commands.”

You can create a function by using the word def in Python. I’m going to have you make four different functions that work like your scripts, and I’ll then show you how each one is related.

```
# this one is like your scripts with argv
def print_two(*args):
    arg1, arg2 = args
    print "arg1: %r, arg2: %r" % (arg1, arg2)

# ok, that *args is actually pointless, we can just do this
def print_two_again(arg1, arg2):
    print "arg1: %r, arg2: %r" % (arg1, arg2)

# this just takes one argument
def print_one(arg1):
    print "arg1: %r" % arg1

# this one takes no arguments
def print_none():
    print "This function does NOT have the argument(s)."

print_two("Jack","Tony")
print_two_again("John","Oakland")
print_one("First!")
print_none()
```

Let’s break down the first function, print_two, which is the most similar to what you already know from making scripts:

1. First we tell Python we want to make a function using def for “define.”

2. On the same line as def, we then give the function a name. In this case, we just called it print_two, but it could be peanuts too. It doesn’t matter, except that your function should have a short name that says what it does.

3. Then we tell it we want *args (asterisk args), which is a lot like your argv parameter but for functions. This has to go inside () parentheses to work.

4. Then we end this line with a : colon and start indenting.

5. After the colon all the lines that are indented four spaces will become attached to this name, print_two. Our first indented line is one that unpacks the arguments the same as with your scripts.

6. To demonstrate how it works, we print these arguments out, just like we would in a script.

i. Functions And Variables.

There is one important point that I want to emphasize in the previous section, which we'll reinforce rig now. The variables in your function are not connected to the variables in your script. Here's an exercise to get you thinking about this:

```
def cheese_and_crackers(cheese_count, boxes_of_crackers):
    print "You have %d cheeses!" % cheese_count
    print "You have %d boxes of crackers!" % boxes_of_crackers
    print "Man that's enough for a party!"
    print "Get a blanket.\n"

print "We can just give the function numbers directly:"
cheese_and_crackers(20, 30)

print "OR, we can use variables from our script:"
amount_of_cheese = 10
amount_of_crackers = 50

cheese_and_crackers(amount_of_cheese, amount_of_crackers)
print "We can even do math inside too:"

cheese_and_crackers(10 + 20, 5 + 6)
print "And we can combine the two, variables and math:"

cheese_and_crackers(amount_of_cheese + 100, amount_of_crackers + 1000)
```

This shows all the different ways we're able to give our function cheese_and_crackers the values it needs to print them. We can give it straight numbers. We can give it variables. We can give it math. We can even combine math and variables.

In a way, the arguments to a function are kind of like our = character when we make a variable. In fact, if you can use = to name something, you can usually pass it to a function as an argument.

j. Functions And Files.

This section we want to combine what we had learned on "file" with "function". The purpose is to see how "function" and "file" can work together to make useful stuff.

```
from sys import argv

script, input_file = argv
def print_all(f):
```

```
    print f.read()
def rewind(f):
    f.seek(0)
def print_a_line(line_count, f):
    print line_count, f.readline()

current_file = open(input_file)

print "First let's print the whole file:\n"
print_all(current_file)

print "Now let's rewind, kind of like a tape."

rewind(current_file)

print "Let's print three lines:"

current_line = 1
print_a_line(current_line, current_file)

current_line = current_line + 1
print_a_line(current_line, current_file)

current_line = current_line + 1
print_a_line(current_line, current_file)
```

k. Functions Can Return Something

We have been using the = character to name variables and set them to numbers or strings. We're now going to blow your mind again by showing you how to use = and a new Python word return to set variables to be a value from a function. There will be one thing to pay close attention to, but first type this in:

```
def add(a, b):
    print "ADDING %d + %d" % (a, b)
    return a + b
def subtract(a, b):
    print "SUBTRACTING %d - %d" % (a, b)
    return a - b
def multiply(a, b):
    print "MULTIPLYING %d * %d" % (a, b)
    return a * b
def divide(a, b):
    print "DIVIDING %d / %d" % (a, b)
    return a / b

print "Let's do some math with just functions!"

age = add(30, 5)
height = subtract(78, 4)
weight = multiply(90, 2)
iq = divide(100, 2)

print "Age: %d, Height: %d, Weight: %d, IQ: %d" % (age, height, weight,
iq)
```

```
# A puzzle for the extra credit, type it in anyway.
print "Here is a puzzle."

what = add(age, subtract(height, multiply(weight, divide(iq, 2))))

print "That becomes: ", what, "Can you do it by hand?"
```

We are now doing our own math functions for add, subtract, multiply, and divide. The important thing to notice is the last line where we say return a + b (in add). What this does is the following:

1. Our function is called with two arguments: a and b.
2. We print out what our function is doing, in this case ADDING.
3. Then we tell Python to do something kind of backward: we return the addition of a + b. You might say this as, “I add a and b, then return them.”
4. Python adds the two numbers. Then when the function ends, any line that runs it will be able to assign this a + b result to a variable.

I. More Practice

We should do some more practice. The first practice code is as follows:

```
print "Let's practice everything."
print 'You\'d need to know \'bout escapes with \\ that do \n newlines
and \t tabs.'

poem = """
\tThe lovely world
with logic so firmly planted
cannot discern \n the needs of love
nor comprehend passion from intuition
and requires an explanation
\n\t\twhere there is none.
"""

print "- - - - - - - - - - - - - - "
print poem
print "- - - - - - - - - - - - - - "

five = 10 - 2 + 3 - 6
print "This should be five: %s" % five

def secret_formula(started):
    jelly_beans = started * 500
    jars = jelly_beans / 1000
    crates = jars / 100
    return jelly_beans, jars, crates

start_point = 10000
beans, jars, crates = secret_formula(start_point)

print "With a starting point of: %d" % start_point
print "We'd have %d beans, %d jars, and %d crates." % (beans, jars,
```

```
crates)

start_point = start_point / 10

print "We can also do that this way:"
print "We'd have %d beans, %d jars, and %d crates." % secret_formu-
la(start_point)
```

The second practice code snippet Save it as *week2l2bimported.py*. Please try to run it. Observe the result. The goal for this code snippet is to learn how to reference (use) these functions from another module. In specific by using the "**import**" key word:

```
print "************To be imported from another module ***************"

def break_words(stuff):
    """This function will break up words for us."""
    words = stuff.split(' ')
    return words

def sort_words(words):
    """Sorts the words."""
    return sorted(words)

def print_first_word(words):
    """Prints the first word after popping it off."""
    word = words.pop(0)
    print word

def print_last_word(words):
    """Prints the last word after popping it off."""
    word = words.pop(- 1)
    print word

def sort_sentence(sentence):
    """Takes in a full sentence and returns the sorted words."""
    words = break_words(sentence)
    return sort_words(words)

def print_first_and_last(sentence):
    """Prints the first and last words of the sentence."""
    words = break_words(sentence)
    print_first_word(words)
    print_last_word(words)

def print_first_and_last_sorted(sentence):
    """Sorts the words then prints the first and last one."""
    words = sort_sentence(sentence)
    print_first_word(words)
    print_last_word(words)
```

m. More Functions And Files

This section we want to use the knowledge we have learned in week2j to create a very practical code snippet. We will make our Python code to read a specific type of file. This type of file is called: Python

configuration file. Which is similar to Microsoft's .ini file that supports the program running and initialization settings.

```
from sys import argv
import ConfigParser


#input_file = argv
scriptname, input_file = argv
# try:
config = ConfigParser.RawConfigParser()
config.read(input_file)
# except ConfigParser.Error as e:
#     print 'EXCEPTION: Program Configuration File - %s' % e.message

myTestToken = config.get('Avtech', 'TestToken')
print "'myTestToken' is: %s" %  myTestToken

myEndpointTest = config.get('Avtech', 'EndpointTest')
print "'myEndpointTest' is: %s" %  myEndpointTest

myUploadFileURL = config.get('Sharefile', 'url')
print "'myUploadFileURL' is: %s" %  myUploadFileURL

myUploadServerUserName = config.get('Sharefile', 'username')
print "'myUploadServerUserName' is: %s" %  myUploadServerUserName

myUploadServerPassword = config.get('Sharefile', 'password')
print "'myUploadServerPassword' is: %s" %  myUploadServerPassword

myUploadTargetDirectory = config.get('Sharefile', 'targetdirectory')
print "'myUploadTargetDirectory' is: %s" %  myUploadTargetDirectory
```

To fully understand the above code, please refer to the following URL:

https://docs.python.org/2/library/configparser.html

Now we will add the logging functionality to this module to complete the entire exercise.

```
from sys import argv
import ConfigParser
import logging

logging.basicConfig(format='[%(asctime)s] %(message)s')
logging.getLogger().setLevel(logging.INFO)


#input_file = argv
scriptname, input_file = argv
# try:
config = ConfigParser.RawConfigParser()
config.read(input_file)
# except ConfigParser.Error as e:
#     print 'EXCEPTION: Program Configuration File - %s' % e.message

myTestToken = config.get('Avtech', 'TestToken')
logging.INFO("'myTestToken' is: %s" %  myTestToken)
```

```
myEndpointTest = config.get('Avtech', 'EndpointTest')
print "'myEndpointTest' is: %s" %  myEndpointTest

myUploadFileURL = config.get('Sharefile', 'url')
print "'myUploadFileURL' is: %s" %  myUploadFileURL

myUploadServerUserName = config.get('Sharefile', 'username')
print "'myUploadServerUserName' is: %s" %  myUploadServerUserName

myUploadServerPassword = config.get('Sharefile', 'password')
print "'myUploadServerPassword' is: %s" %  myUploadServerPassword

myUploadTargetDirectory = config.get('Sharefile', 'targetdirectory')
print "'myUploadTargetDirectory' is: %s" %  myUploadTargetDirectory
```

NOTE 1. Python Logging
https://docs.python.org/2/howto/logging.html

NOTE 2. Python Log to a file

```
import logging

logging.basicConfig(filename='example.log',level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
```

n. Take Home Test.