CSci 1133, Spring 2015
Lab Exercise 2

Sequential Control

In this exercise, you will explore the basic elements of programming using Python. Take advantage of all the help around you and be sure to explore on your own!

This and future lab exercises will include 4 categories of problems: *warm-up*, *stretch*, *workout* and *challenge*. The first three must be completed during the lab session. The "*challenge*" problem(s) are optional for those of you who just can't get enough.

Warm-up

Functions and Procedural Abstraction

As discussed in lecture, a *computational procedure* is a series of steps that direct a (mindless) machine to manipulate data and produce some desired *result*. We generally think of a *computational procedure* as a mapping from a set of input data to some set of output data:

input → *process* → *output*

A *program* is simply a computational procedure that has been translated into a specific *programming language.* A computer equipped with the necessary software to interpret that language is then able to execute the program and accomplish the specified task.

Some computational procedures perform tasks that we need to repeat frequently, such as "*compute the square root of some number*", or "*convert some value to characters and display them on the computer terminal*". These are, in their own right, self-contained computational procedures that take some input (i.e., number, value) and produce output (square root of the value, character-stream).

We can create a powerful and useful *abstraction* by giving a computational procedure a *name.* For example, we might assign the name `sqrt` to the procedure, "*compute the square root of some number*". Now, instead of re-typing all the steps that define this procedure each time we need it, we simply *invoke* it by providing it's *name*: `sqrt`. This is the idea of *procedural abstraction*. A *procedural abstraction* is simply a means of giving some detailed process a name and then subsequently forgetting about all the details (which we really didn't care about anyway!) When we use abstractions (named procedures) in our program, the computer takes care of invoking the necessary steps for us, freeing us to think and develop programs at a much higher level.

In Python, a procedural abstraction is referred to as a *function*. A *function-call* consists of giving the name of the function followed by parentheses. Some (but not all) functions such as `sqrt` require input values, or *arguments*. For example, `sqrt` requires the *value* for which it is supposed to compute the square root. When a function requires input values, the argument (or arguments) are provided inside the parentheses and separated by commas. For example, `sqrt(x)` will return the value of the square root of the value currently referenced by the variable `x`.

A huge advantage of using procedural abstractions is that we can leverage a large body of programming effort that other people have already expended. We simply get to *use* it. For example, in last week's lab exercise, you learned how to display the value of an object using Python's built-in function `print()` without understanding the details of what it does (and not really caring!)

The basic syntax of Python is quite simple. Much of the power of Python, and any programming language for that matter, derives from the large number of procedural abstractions (functions) that are provided with programming language. We will be exploring many of these as we go along.

The input Function

Useful computational procedures require the ability to input and output data from/to an external device. Every programming language will provide facilities to gather externally supplied data (input) and provide computed results (output). In the previous lab you were introduced to the Python `print()` function that outputs values to the terminal display. In addition, Python provides the ability to *input* external data using the `input()` function.

Using the input function to solicit keyboard input is simple. You simply name the object returned from the function. Start the Python3 interpreter in command-line mode. If you need help, refer to last week's Lab Exercise. Now try this:

```
>>> ival = input() <ENTER>
```

You should notice that Python has not provided another >>> prompt… In fact Python is still executing the `input()` function, waiting for you to type something. Now type:

```
3.2 <ENTER>
```

When you press ENTER, the characters you just entered from the keyboard are read by the input() function and returned. The *assignment* operation (=) stores this value as an object in memory and associates it with the name, `ival`.

In Python, every unique value has a specific *type*. For example, numeric values are either *integers*, *floating-point*, or *complex*. Individual letters and digits are considered *characters*, and groups of contiguous characters are *strings*. Python provides a built-in function that give us the *type* of any value (object). Let's see what *type* of object the `input()` function returned. To do this, use the `type()` function:

```
>>> type(ival) <ENTER>
```

You should see something like this:

```
>>> <class 'str'>
>>>
```

The `input()` function always returns a *string* object! Even though you typed a *number*, Python treats it as a *string*. Try this:

```
>>> ival+36 <ENTER>
```

What happened? You get a *syntax* error telling you something about converting int and string. Python is telling you that it doesn't make sense to add 36 to the string "3.2". Recall that strings are collections of characters, *not* values. *Digits* typed on a keyboard are actually *characters*. To obtain the value that the collection of symbols *represents*, you need to *convert* the string to a numeric value. You can use the built-in function `float()` to convert a string to a *floating-point* value. Try it:

```
>>>  float(ival) <ENTER>
```

Python returns a numeric object whose *value* is equal to the number represented by the string you input using the `input()` function.

Python also provides another function, `int()` that will convert a *string* to an *integer* value. Let's try it:

```
>>>  int(ival) <ENTER>
```

Oops… what happened? Python is telling you that "3.2" is *not* an integer. This is an example of a *runtime* error. If you had originally input the string "42", the conversion would have worked just fine. "But", you might argue, "how is my program supposed to know in advance what *type* of number someone enters? I have no control over what they *might* do!" That is *correct*! So a better way of converting input strings to values is to use another built-in Python function that *automatically* determines what the appropriate type is and simply converts it without complaining. This function is `eval()` and it works much like `int()` and `float()`.

```
>>>  realval = eval(ival) <ENTER>
>>>   type(realval)  <ENTER>
```

Generally, it is simpler to combine both *input* and *evaluation* in a single statement:

```
>>> ival = eval(input()) <ENTER>
42  <ENTER>
>>> print(ival) <ENTER>
>>> type(ival)  <ENTER>
```

Think carefully about what just happened. `eval()` is evaluating *its* argument which happens to be the string that `input()` returns. You will see this *nested* use of functions over and over in computer science and computer programming.

Finally, we've been using `input()` without any arguments. In fact, it is quite useful to provide a *prompt* to the user so they have some idea what the program is asking them to do. Otherwise when `input()` is executed, the cursor just blinks, waiting for the user to do something without any instruction. To this end, you can optionally provide a string object as an argument to the `input()` function. A *string* is simply a collection of alphanumeric characters enclosed in double-quotes (" "). The string will be displayed as a prompt prior to waiting for input:

```
>>> ival = eval(input("Enter the distance: ")) <ENTER>
```

Throughout this lab, your programs will need to solicit values from the keyboard for various computations. If you are still uncomfortable with `input()` and `eval()`, consult one of your TAs before beginning the following exercises.

Python's Integrated Development Environment (IDLE)

Recall that Python is an *interpreter* that can be run in either "command-line" or "script" mode. Although command-line mode is useful for exploring various constructs and testing out ideas, in general, most Python programming involves first constructing a program (*script*) using a text editor, then subsequently running that program using Python in *script* mode. In last week's lab, you used a text editor to enter Python commands

into a script file named `decay.py` Note that Python programs (scripts) have the file extension `.py`. In order to "run" the program, you invoked the Python interpreter in script mode by using the `python3` command followed by the name of the script file:

```
% python3 decay.py
```

Nearly all of the programming you will do from now on will involve writing script files and then executing them in script mode. Generally, you will find that they don't actually do what you intended when you first try to run them. Either you will make unintended typing mistakes (*syntax* errors), or something crazy will happen which results in the interpreter halting the execution of your program with a dire-sounding warning message (*runtime* errors). As you get better at writing Python programs, you will encounter fewer *syntax* and *runtime* errors, however you will often discover that your script runs fine yet produces erroneous results. This, more common problem, is known as a *logic* error and you will subsequently need to deduce why your program isn't doing what you intended. We refer to this as *debugging*, and it involves isolating potential problems, editing your source file to remedy the problem, and then re-running your script. You may have to repeat this cycle many times to get your program to produce the correct results. This is actually a very common scenario and, in fact, becomes one of the primary challenges for anyone who writes computer programs!

To facilitate this *edit-run-debug* cycle, it is quite useful to employ an *Integrated Development Environment* or IDE. An IDE is basically a text-editor combined with a Python interpreter that allows you to run your program directly from the editor (without needing to invoke the interpreter each time). The standard Python distribution includes a basic, but very useful, integrated development environment named IDLE3. Let's try it out. From the terminal, enter the following:

```
% idle3 <ENTER>
```

You should see a window that looks like the terminal with the Python prompt:

```
>>>
```

This is telling you that the Python interpreter is running and you can enter Python commands in command-line mode as always.

The window has several "pull-down" menus. Under the *file* tab, you will see several options including *new window*. If you select this, you will open a second *editing* window. You can create your Python program in this window. When you want to try running your program, simply choose *run module* from the *run* pull-down menu.

Experiment a bit with IDLE. Make sure you are using `idle3` and not `idle` when you start the program, otherwise you will be running the wrong version of Python. If you are having trouble, ask one of your TAs to help you.

Stretch

For the following exercises, you should implement the programs by using Python in *script* mode. Create your .py scripts using the Python IDLE3 integrated development environment discussed in the last section of the warm-up exercises.

1) **Pounds and Kilos**

Write a Python program that will prompt the user to enter a weight in pounds, then convert it to kilograms and output the result. Note, one pound is .454 kilograms.

2) **Wind Chill Temperature**

Write a Python program that will prompt the user to enter a temperature *t* in degrees Fahrenheit and a wind velocity *v* in miles/hour. Compute and output the Wind Chill Temperature according the following formula:

$$windChill = 35.74 + 0.6215t - 35.75v^{0.16} + 0.4275tv^{0.16}$$

Note that this formula only works for temperatures in the range of [-58…41]

3) **Loan Payments**

Write a Python program that will prompt the user to enter a loan amount, an *annual* interest rate and a loan duration (in months). Then compute and output the monthly payment required to pay-off the loan in the indicated time using the following formula:

$$payment = \frac{r(loan\_amount)}{1-(1+r)^{-n}}$$

where *r* is the monthly interest rate (annual rate divided by 12) and *n* is the number of monthly payments.

Workout

1) **Reverse the Digits**

Write a Python program that will prompt the user to enter a four digit number as an integer and then output the digits in reverse order. (Hint: Use integer division and modulus to identify the 1's, 10's, 100's, and 1000's digits)

Challenge

This problem is for those of you who just can't get enough! You should be able to complete the warm-up, stretch and workout problems in the lab. Try this if you have extra time or would like additional practice outside of lab.

1) **What Time Is It?**

On UNIX systems, the current time of day is measured as the number of seconds transpired since Midnight on January 1, 1970 GMT. Python provides a `time` module that includes methods you can use to obtain date and time information.

Write a Python program that will use the `time` module to determine the current GMT time and display it in HH:MM (hours:minutes) in a 24-hour format (e.g., midnight is 00:00, 5:00pm is 17:00, etc.). Note the following:

- Import the `time` module using the following line in your program:

```
import time
```

- The `time.time()` method will return the number of (fractional) seconds since midnight on January 1, 1970 as a floating-point value. You will need to convert the value to an integer.

- Hint: You will need to use the modulus (%) operator to determine the actual hour and minute values and integer division (//) to determine the total hours & minutes.