Lab 2: Functional challenges

Overview

We'll begin working with Wingware IDE, which is one of many different integrated development environments you could use. The lab will then focus on working from and adapting the codebook for this week: defining functions, using lists, passing arguments to functions, and calculating things within functions and returning the answers.

Prerequisites

Make sure you've gone through the next section of Codecademy through "A Coding Vacation."

Reminders

Journals: Keep a journal for this lab, separate from the prior lab. Use the naming convention from before: <yourlastname>_<yourfirstname>_lab2_journal.docx.

Checklist of deliverables

Lab is due at the beginning of the next lab (2pm) on Thursday January 21. When you complete the lab, use Zip or 7Zip to bundle these files and place the zipped file in the Geo578/Drop/Lab2 folder:

Lab 2 Journal: <yourlastname>_<yourfirstname>_lab2_journal.docx or .pdf

- This should include the deliverables described in Section 1 and Section 2, **clearly marked**

Section 3: <yourinitials>_lab2_functions.py
<yourinitials>_lab2_scripts.py

Section 1:

Get set up with Wingware.

Steps

1. Files:
    a. We'll be working from the code book. Make a new directory in your student directory called "week2", and **copy the three codebook files** from Classes\Geo578\Data\Codebooks\week2\ to your directory
2. Wingware!
    a. Start Wingware IDE -- search from the start button for "wing"
    b. It should start up a graphical user interface (GUI).
    c. If you want, you can try the tutorial to become familiar with the interface. The basic pieces are: the Python console

("Python shell") in the lower right, the Search and Stack data status windows in the lower left, and what will become your editor (where the big Wingware icon is initially).

3. Load some files!
    a. Load the three codebook files from your directory into the interface.
        i. Use the Folder icon and navigate, just as you would for any other program.
    b. When you load those files, they should show up in the top half of your window.
        i. Note that the different files are indicated by different tabs
        ii. Note the color-coding of the text! Cool.
4. Paths: Wingware updates Python's search path to include those files. But you should double check.
    a. Checking paths:
        i. In the console, TYPE THIS STUFF:

```
>>> import sys
>>> sys.path
```

    b. At this point, the returned line likely wraps off to the right of the screen. Select the "Options" button on the right just above the console, and select "Wrap lines"
    c. Make sure your path is in there. There is a long line of paths, probably, but look through and find that yours is there.
5. Try running the "class3_review.py" file.
    a. Use:

```
>>> execfile('class3_review.py')
```

    b. Did it work? Check the code to see what you expect to see it doing.
6. Now try running it a different way.
    a. In the Wingware icon panel, SELECT THE GREEN "PLAY" BUTTON. This will run the code that is in the front of the editor.

Section 1: Deliverables

Remember, put these in your lab2_journal.docx file, and make sure to label these as "Section 1 Deliverables". No screenshots necessary here -- just written answers to these very thought-provoking questions.

1. (1pt) What was happening internally in the interpreter's brains when you typed "import sys"?
2. (1pt) What type of variable is sys.path?
3. (2pt) Once you've run 'class3_review.py', use the "locals()" command in the console to SEE THE OBJECTS THAT ARE CURRENTLY IN MEMORY. Do you see the variables that I assigned in class3_review.py? Why or why not?
4. (2pt) Modify the code in class3_review.py to create A SYNTAX ERROR somewhere.

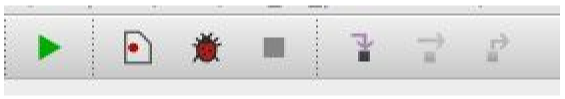
   a. Yes, I'm asking you to mess up your code
   b. What does Wingware do?
   c. UNDO YOUR SYNTAX ERROR for the next step

5. (2pt) In "class3_review.py", try to INSERT AN EXCEPTION ERROR of some sort (your choice!). Recall from class last week the difference between an exception-type error and a syntax error, and if necessary Google some examples. Once you've inserted the error, save the file, and then run it again using execfile. Briefly describe what you see. Does the line number where Wingware complains match where you put your exception? What error did you insert?
   a. Leave your error!!! We'll need it for the next step.

Section 2. Meet the debugger

7. Debugging: The most funnest bestest thing there is!
      i. Yes. Many grammatical errors in my sentence. See? You are a natural debugger!
   b. Debugging means finding bugs -- finding errors. How do you do it?
   c. Debugging logic:
      i. Debugging is essentially **hypothesis testing**. If something breaks, you:
         1. observe the kind of error
         2. then use your knowledge of the logic of the Python interpreter to develop some limited hypotheses about what could be happening
         3. and then you design tests to eliminate different causes.
      ii. Typically, if an error occurs at point "x" in the workflow, you'll need to understand the state of things *just before the error occurred* to see why Python went off the cliff. At that point, you can see the status of variables and functions and see which things do not match your expectations.
   d. So, how do we see the status of variables just before the error occurs? We re-run the program in "debugging" mode, and tell Python where we want it to stop. Here's how:

8. Debugging in Wingware:
   a. In the Wingware interface, you'll notice an icon near the top that looks like a ladybug. Pretty. That's the key to our debugging. You'll use it in a sec.



   b. I'm at my breaking point here!!!!
      i. In an IDE, you insert "breaking points" to halt the flow of program execution where you want to check on variables.
      ii. INSERT A BREAKPOINT:
         1. Make sure that you're looking at the "class3_review.py" still, and that your exception error is still in there.

2. INSERT A BREAKPOINT at the line where you created your exception error.
    a. How? You can insert them either by placing your cursor somewhere in your file and clicking the little red-dot-in-document icon next to the Ladybug, or you can just click in the grey border to the left of the text at the line you want. A red dot will show up there once you have a break point.
3. If you want to turn it off later, toggle your breakpoint off by clicking on the red dot.

c. Now, re-run the code, but instead of using the "execfile("")" or green play button approaches, use the debugger by doing the following:
    i. CLICK ON THE LADYBUG!
    ii. The debugger will run up to your breakpoint, then highlight the line where it is stopped.
    iii. LOOK IN THE "STACK DATA" tab on the lower left panel of Wingware. Notice how the variables in your "locals" list show the variables that I assigned in the class3_review.py program. This shows the status of the interpreter's brain just before the error you inserted.

d. You can "step through" the code one line at a time by clicking on the icon that looks like right-hand arrow above a black box (a few icons to the right of the ladybug).
    i. IF YOU CLICK THAT STEP-THROUGH ONCE, you should immediately hit your error!
    ii. On the right of your GUI, you should see a new Tab pop out with "Exceptions". This gives you an indicator of the error.

e. On to deliverables!

Section 2 Deliverables

Remember, put these in your journal.docx file, and make sure to label these as "Section 2 Deliverables". No screenshots necessary here -- just written answers to these very thought-provoking questions.

(2pt) Describe how the error you created can be linked logically to the condition of variables whose state you can see in the "Stack Data" tab.

(3pt) Pretend you did not know what caused the error you created. Look only at the error and the code (not the stack data), and develop a testable hypothesis about what caused the error that can be resolved by looking at the status of objects in the Stack Data window.

> Example using very generic language: "My error is about passing the wrong operand type to function X. As far as I know, that function X requires type Y. Test: If I look in the

stack data at the variable type I am passing to the function and it is not type Y, then I know I need to look earlier in my code to see why that variable is defined with the wrong type. If it is type Y, then there must be something else in the function syntax that I don't know about -- I'll have to look that up in the function syntax on Python.org."

Section 3

Load the codebook!

Steps

9. Start using the codebook.
    a. Here's how it works:
    b. The "week2_functions.py" is a file that defines a bunch of functions.
    c. The "week2_scripts.py" is a script file that illustrates how to call those functions.
10. Commenting
    a. Note that the "week2_scripts.py" file has many functions later on in the file that are "commented out", meaning I have put a comment symbol to force the interpreter to ignore them. This allows you to just focus on one function at a time. Initially, the only function that will run is week2_functions.sosimple()
11. What to do:
    a. UNDERSTAND ALL OF THE FUNCTIONS IN WEEK2_FUNCTIONS.PY
        i. In "week2_functions.py", read through the function's code first and see if you can predict what will happen when you run it.
        ii. Then, in "week2_scripts.py", uncomment the line that calls the function, and run it.
        iii. If necessary, put breakpoints in the code along the way to track how variables are evolving.

Section 3 Deliverables

Finally! You get to write some code. Based on what you learned looking through the codebook, develop your own codebook. Build two files analogous to my functions and scripts files, but with different names:

<yourinitials>_lab2_functions.py
<yourinitials>_lab2_scripts.py

Use a similar approach: Define the functions in the "functions.py" file, and develop variables in the "scripts.py" file and call the functions. Use the rules on good commenting as in prior lab:

Comment each logical section within your code and introduce each function well. Then, within each function, comment the lines where anything new happens.

In a moment, I'll describe what the functions should do. But first, what to deliver:

1. In your lab2_journal.docx file, document your trials and travails (if any) and the logic of what you're trying to do.
2. In your "functions.py" file, include all of the functions to meet the goals below. In the comments for each function, make it clear which directive that function is responding to from the list below.
3. In your "scripts.py" file, import your "functions.py" file, and call the functions (refer to the "week2_scripts.py" file for examples of how to do this).

What functions should you build? Points refer to the base points for each function. Appropriate commenting will get you an additional 2 points per function.

12. (3pts) Name math:
    a. Write a function that accepts your first and last name as a single string, uses the str.split() function (look it up) to separate your first and last name, and returns the product of the length of your first and last name.
13. (5pts) Latitude math:
    a. Write a function that accepts a three-element list of [degrees, minutes, seconds] and converts it to a single decimal degree value.
        i. Note that 60 minutes = 1 degree, and 60 seconds = 1 minute
        ii. Make sure the user does not enter a value greater than 90 degrees or less than 0 degrees. If they do, let them know gently.
14. (4pt) Filename iteration generator:
    a. You want to run a bunch of tests that will generate many variants on a starting file name. Assume the file name has a file extension with three characters following a ".", e.g. "filename.img". Write a function that accepts a single string, and returns a list variable with 10 different strings. Each string will be the starter string, but will have the number from the sequence [100m, 200m, 300m, etc. up to 1000m] interspersed before the filename extension, e.g. "filename100m.img", "filename200m.img", etc.
15. (4pts) Euclidean distance between two points
    a. You can pass more than one argument to a function. To pass more than one, you simply define the function as:

       ```
       def myfunction(arg1, arg2):
       ```
    b. Write a function that accepts as arg1 an [x1,y1] list of coordinates and arg2 as an [x2, y2] list of coordinates. Use the math.sqrt module to find the Euclidean distance between the two points.

16. (bonus 4pts) Random points

   a. Write a function that generates a list of random samples within a coordinate box defined by the user. The function accepts three arguments:
      i. arg1: number of samples desired
      ii. arg2: the [x,y] pair of the upper left corner of the box
      iii. arg3: the [x,y] pair of the lower right corner of the box
   b. The function returns a list of [x,y] coordinate list variables (a list of lists), or returns an array object (check out the “numpy” module, which we’ll be using later in the term).