Lab 1: Python as a foreign language

Overview

Like any language, Python has a grammar and vocabulary that you need to know to use it. In this lab, we'll start laying the foundations for speaking Python. The lab will focus on applying concepts from the Codecademy lessons online, as well as introducing some fundamental building blocks that you'll need for developing programs over the term and beyond.

Prerequisites

Make sure that you've gone through Codecademy sections on Python Syntax through Date and Time.

Get set up

In Codecademy, Python magic is embedded in the web page. In real life, we need to link:

a. Python.exe: The guts of Python -- the interpreter that converts your instructions into what you want
b. Your code: The instructions you write, saved as files that you can use over and over again (and turn in for grading!)

This requires that you know where to find everything on the file system of your computer, and tell Python where to find your code.

Instructions below are for the installation in the Digital Earth lab, but when you do this on your machine, you would need to adapt to find the correct paths to the appropriate files. It's quite likely that the python software package will be somewhere else on a different machine.

Checklist of deliverables

At the end of lab, use Zip or 7Zip to bundle these files and place the zipped file in the Geo578/Drop/Lab1 folder:

Lab 1 Journal: <yourlastname>_<yourfirstname>_lab1_journal.docx or .pdf
Part 1: <yourlastname>_<yourfirstname>_lab1_part1.py
Part 2: <yourlastname>_<yourfirstname>_lab1_part2.py
Part 3: <yourlastname>_<yourfirstname>_lab1_part3.py

Journaling:

First rule of programming: There will be problems, always! Anticipate this and keep good track of what you're doing, because without good notes you will have a very difficult time unraveling problems. Thus, you'll need to keep a journal/notebook/lab book -- whatever you prefer to call it -- that captures details of what you think and do, what problems you run into, and how you solved them.

Turn in a separate journal for each lab.

Components of journal

1. Use an outline form, with main goal as the outermost level, subtasks as next level in, and various tasks, findings, errors, etc. as next levels in.
2. Copy and paste into your journal all paths and filenames, and -- if appropriate -- any parameter values, important variables, etc.
3. Include notes to yourself about your logic about why you are trying things.
    a. For example: "Need to load shapefile. I'll try first using xxx, and then I'll have to check to make sure the attributes are as I expect. If not, then may need to check yyy." This provides you a roadmap, and helps as you're looking back in your journal to recall why you did things.
4. Include full notes about any errors you get -- best to copy and paste the full error message than to simply say "fail! didn't load", etc.
5. When you get to an error, stop and document the logic for what things might be causing it. This will help you systematically test things.
6. Include any fixes, aha moments, etc.

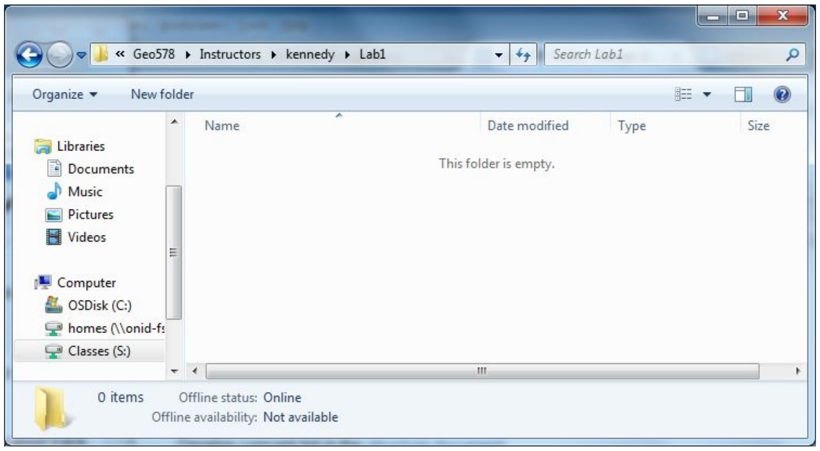
Grading: Up to 10 points for the journal.

Section 1:

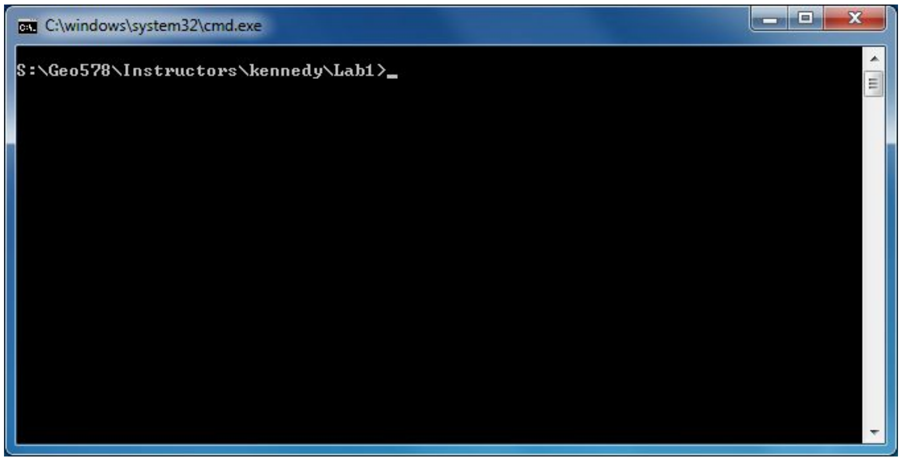
Set up your file structure to make a place for your own scripts and start a Python console.

Steps

1. Find your home folder on the Digital Earth server
    a. If you are in the lab, start a file explorer and find the "Classes (S:)" drive under "Computer", then navigate to your Students\<yournamehere>
2. Use the file explorer to **make a new folder within your directory called "Lab1"**, and open it so you see the "This folder is empty" note in the file explorer.



3. Start a journal file!
    a. Again, please keep a separate journal file for each lab. Name it: <yourlastname>_<yourfirstname>_lab1_journal.docx or .pdf.
    b. *If you want to use a format other than Word or PDF, be sure to clear it with me and Jane so we can read it!*
    c. NOTE: Again, make sure to document the name of the folder you've got here -- use the full pathname.
4. Place your cursor within that directory, HOLD THE SHIFT KEY DOWN and right click. You should see a dropdown window that includes "Open command window here". Select that!
    a. NOTE: If you don't hold the shift key down, you won't see that option.
5. After you select "Open command window here", you should see one of these:

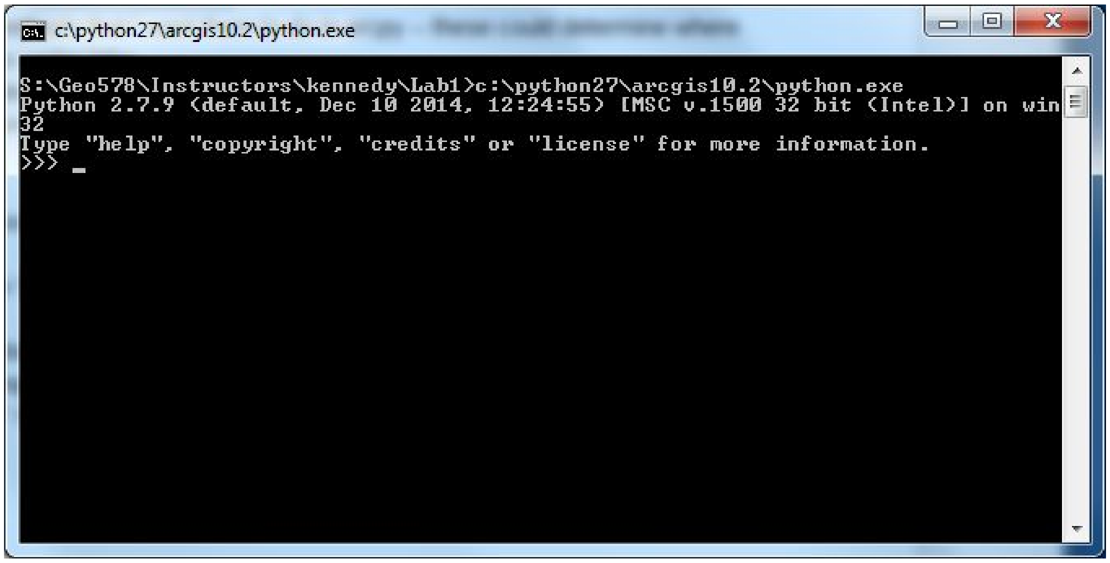


```
S:\Geo578\Instructors\kennedy\Lab1>_
```

6. What is this?
    a. If you've lived your life on graphical user interfaces, welcome to the world of the command line. They call it that because we can command the world from that line. Seriously. Will it feel clunky at first? Probably. You'll have to make sure you

type everything exactly right, no typos, etc. But all good things require a bit of pain to make them worthwhile, right? Plus, you get a lot of street cred with the programming geeks of the world.

7. The command line window shows a cursor with your current path in it. It should show something like: S:\Geo578\Students\<yournamehere>\Lab1
8. How does this work?
    a. When you type a command at the command line, the computer will try to make sense of it and do something. This is not Python yet -- it's the operating system command line for the kind of computer you're on. We're not really going to get into operating system commands in this class, but here are some things you can try typing here:
        i. dir
            1. This will list the contents of the directory you're in
        ii. echo hello
            1. This will spit back "hello" to you. Exciting.
9. Most importantly, you can type the name of programs in here and it will start them, as long as it knows where to find them. We're interested in running python. So let's try that:
    a. type: python
    b. What does it tell you? [no need to turn this in!]
10. To make this work, we need to tell the operating system where that python interpreter (python.exe) actually lives. We'll need to give the full path name to that file.
    a. On the lab computers, the python interpreter is at this location: C:\Python27\ArcGIS10.2\python.exe
        i. The ".exe" means it's a program that can be executed -- i.e. run by the computer!
    b. Thus, to run it, all you need to do in your terminal window is type the full pathname to the executable and press enter. In other words, at the prompt, type: C:\Python27\ArcGIS10.2\python.exe



```
S:\Geo578\Instructors\kennedy\Lab1>c:\python27\arcgis10.2\python.exe
Python 2.7.9 (default, Dec 10 2014, 12:24:55) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

11. What is in this window?
    a. This is Python's command line. It's as raw as it comes. We're going to get fancier next week, but for now it's important to see what this basic system looks like.
    b. When you type something at the cursor and hit enter, the python interpreter swings into action and tries to make sense of what you did. This is basically the same idea as what you have been doing in the Codecademy window, except now you're going to do it "for real"!!! Yeah! Later in the lab, we'll do some trials of the things you've learned in Codecademy (right?).

Section 2:
Make sure you can get things to respond at the Python console.

Steps
1. At the Python prompt (>>>), type:

   print "hello world"

    a. It should spit back "hello world" to you.
    b. Thought exercise: Why does "hello world" need to be in quotes? What kind of variable is it?
2. From here on out, I'll indicate things to type at the command line prompt with the ">>>" symbol, so in the example just now, I would ask you to do this:

   ```
   >>>  print "hello world"
   ```

And of course, you would know that I meant for you to only type the stuff *after* the >>> prompt, right?

3. If that works, it means your Python interpreter is going, and you can launch into replicating some of the stuff you've done in Codecademy.

Section 2 Interpreting the interpreter

1. At the interpreter prompt, try a few easy string examples to get started. No need to turn anything in here -- just want you to start getting a sense how to interact with the interpreter.
   i. assign the string "great power" to the variable named *third*
   ii. assign the string " has " (Note the spaces) to the variable named *second*
   iii. assign your own name to the variable *first*
   iv. see what you get when you: print first + second + third
   v. see what happens when you: print (first * 2) + second + third
2. Great! You're done with this section. There's an important piece of grammar we've just slipped in.
   a. In English, we think of sentences as having Subjects + Verbs that then act on Direct Objects. (see http://www.chompchomp.com/terms/directobject.htm if you want a reminder). Like: "Tommy, eat your porridge."
   b. When we type:

      ```
      >>> print first
      ```

      we are essentially writing a command-style sentence where "print" is the verb, "first" is the direct object, and the *subject is implied*. It's implied because it's always the same -- it's the Python interpreter! In other words, our full sentence would be: "Python Intepreter, print first".
   c. In Python-speak, we call the verbs "functions". And lo and behold, we call the direct objects "objects". Objects are everywhere in Python, and are more powerful than what we're using now, but for now it's handy analogue.
   d. Here's something to try -- the more formal syntax for functions. Instead of printing the way demonstrated above, try: print(first)
   e. Why? This makes it very explicit where the object of that verb is -- it's completely enclosed by parentheses. In the future, we'll want to use fancier functions that act on more than one object, and it will be important to know where all of the objects start and end!
   f. You could show all that you've learned from Codecademy by typing stuff into the console, but that would require doing loads of screen shots. Ted.I.Ous!!! Instead, we need to move on to a simpler way capture all of your coding brilliance

in a way that is easy for us to evaluate. Plus, it sets the stage for your further work.

Section 3:

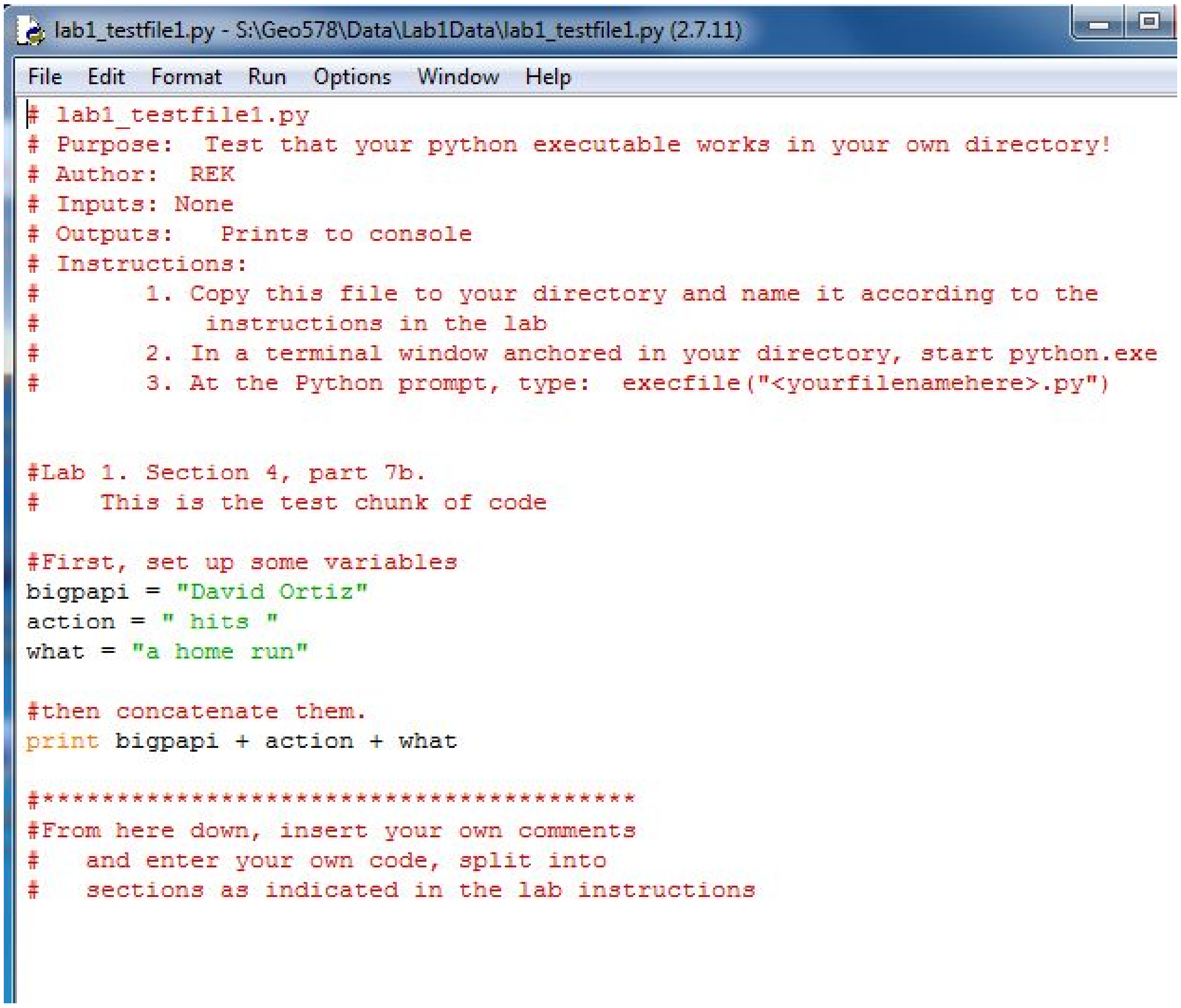
Have Python run the commands stored in a test file.

Steps

1. Using the file explorer, copy this test file
    a. S:\Geo578\Data\Lab1Data\lab1_testfile1.py

    to your own directory and rename as follows:

    b. **<yourlastname>_<yourfirstname>_lab1_part1.py**
    c. Note that you don't put the < and > signs in there -- those are just my shorthand to say: swap something else in for this entire thing
    d. You're going to turn this file in for credit!
    e. Note the name and what you did in your journal!
    f. You have to keep the ".py" at the end of the filename!!!
        i. Why? This is a secret code to the python interpreter that this is a file it should be happy to read. Secret codes. Cool!
2. Now, we want to open the file to look at the commands. What happens if you just double-click on it?
    a. Why did that happen? Because it's got the .py extension, Windows reads that secret code and knows that you want to run it in Python. The problem is that it starts a python console, runs the code, and then it disappears. It doesn't allow us to see what's in the file.
3. Instead, right-click and select: "Edit with IDLE"
    a. What is IDLE? It's a simple "Integrated Development Environment", or IDE, that gets bundled with the standard Python package. It is like a text editor, but adds more functionality to help you interpret your code. We're only going to officially use it for this first lab, but if you want to learn more about it, check it out on Wikipedia.


lab1_testfile1.py - S:\Geo578\Data\Lab1Data\lab1_testfile1.py (2.7.11)
File Edit Format Run Options Window Help

```
# lab1_testfile1.py
# Purpose:  Test that your python executable works in your own directory!
# Author:  REK
# Inputs: None
# Outputs:   Prints to console
# Instructions:
#       1. Copy this file to your directory and name it according to the
#           instructions in the lab
#       2. In a terminal window anchored in your directory, start python.exe
#       3. At the Python prompt, type:  execfile("<yourfilenamehere>.py")


#Lab 1. Section 4, part 7b.
#    This is the test chunk of code

#First, set up some variables
bigpapi = "David Ortiz"
action = " hits "
what = "a home run"

#then concatenate them.
print bigpapi + action + what

#*****************************************
#From here down, insert your own comments
#   and enter your own code, split into
#   sections as indicated in the lab instructions
```

b. You'll notice that text is colored differently depending on what role it has. IDLE and other IDEs evaluate your code and try to help separate the wheat from the chaff -- so that you can quickly visually scan the code to see where the action is, etc.
c. So, look at this code and try to predict what you expect to happen. It's probably pretty simple, but this habit is critical for later debugging: Predict what you think should happen at each step, and when there is a bug somewhere, you can step forward through your expectations until things don't match your expectations -- if you're lucky, just upstream of the bug!
d. To run the code, type the following in your Python interpreter console window.

>>> execfile("<yournewfilenamehere>")

i. NOTE: Do not put this in your <blahblahblah.py> file -- this command is meant to be run directly in the interactive Python interpreter window!

   ii. Also note: don't type this literally -- instead of <yournewfilenamehere>, use the filename you just assigned to your copy of the file!

e. In other words, if you named the copied file: thegreat_gonzo_lab1_part1.py then you would type in the interpreter

```
execfile("thegreat_gonzo_lab1_part1.py")
```

4. What does this do?
   a. This command tells the python interpreter to open the file, and read in the commands in the file one at a time just like you were typing them at the command line.
5. If this works, you're ready to go on to the next section. You will be modifying that file that you just made.

Section 4:

Add a bunch of commands to the file and run them. You'll turn this file in as part of your lab assignment!

Steps

3. **In your Python file you created in section 3,** add commands that respond to the pieces below.
   a. In other words, you're going to be adding on to the starter code in the file I just had you copy….
4. NOTE: Good practices alert!
   a. Comments: Recall using comments in Codecademy? Now's your chance. You need to comment heavily.   Documentation documentation documentation!
   b. To set up good habits, for these exercises do the following:
      i. Put a comment at the beginning of each sub-section below, where a sub-section is indicated by the lower-case letters. Indicate the sub-section and the overall goal of the section. Then, for each line in the code where you do something new, put a trailing comment at the end of the line to show what you expect that command to do. For example:

```
#  Lab 1. Section 4
#
#  Subsection 1:  String examples

print 5 * "Happy"        #make happy repeat five times

etc. etc. etc.
```

c. It is okay to put mini comment headings above a chunk of code where you're doing repetition of exactly the same thing. For example:

```
#Assign several strings to variables
dog = "fido"
cat = "fluffy"
rondo = "a la turk"
```

d. Grading: To encourage you to really comment, grading will take into account your commenting. In addition to the points listed for each command below, you'll get points for commenting. For each subsection where you put a comment, you get 1 point if you put in the comment. If you adequately comment the lines in the section (or mini headings, as appropriate), you get 1 point for that whole section. Easiest points you'll ever earn.
e. Logistics: You can run the execfile command a bunch of times, or you may want to test your solutions at the console until you get them to work, and only then put the working code into your growing .py file.

<u>Deliverables Section 4</u>

5. Deliverables
    a. Subsection 1: Do some string examples. For each of these, show the python commands and the results of implementing those commands.
        i. (1pt) Print 5 copies of the string "Happy"
        ii. (1pt) Without changing the "Happy" string, insert spaces between each of the "Happy"s.
            1. Hint: Use parentheses!
        iii. (1pt) Fix and then print this string: 'I can't believe I'm falling apart, said Humpty'
        iv. (1pt) Fix this statement:

            ```
            path = "c:\python27\arcgis10.2\"
            ```

            1. Note: In your comments, you may want to add a full comment line above the fixed command with a copy of the problematic statement -- this way you can see what you started with and what you ended with.
        v. (1pt) Set the variable *path* to your Lab1 directory on the S: drive, the variable *myfile* to "jujuhearts.py", and a third variable *fullfilename* to the

concatenation of *path* and *myfile*. Print *fullfilename.* Make sure there are no spaces in the final string!

b. Subsection 2. Accessing strings by index.
   i. (1 pt each) For your *path* variable, show:
      1. the fifth character
      2. the fifth character from the end
      3. the length of the string
      4. the uppercase version of the string

c. Subsection 3: Accessing strings by index: *More advanced use!*
   i. NOTES:
      1. Codecademy didn't cover this yet, but there are some important variants on indexing that you will use over and over, so we should start now!
      2. You can pull out more than one letter a time:
         a. Just for kicks, try this at the console: print "clever"[1:3]
            i. This tells the interpreter to pull out only the letters at the index starting at 1, go up to 3 (but not including it).
      3. You can pull all of the letters from or to a given starting point:
         a. Try at the console: print "clever"[1:]
      4. You can reference strings from the end of the string backwards!!!
         a. Try at the console: print "clever"[-3:]
      5. Habit forming: Do some searches online to learn more about this. You may end up on python.org, stackoverflow.org, or some other site. This is good practice -- when you hear about something new in Python, do a quick search. You'll find a huge community of people out there who have probably run into the same thing you want to do!
   ii. Okay, for points, do these:
   iii. (1 pt): Show the last three characters in your *fullfilename* variable from earlier.
   iv. (3pts): Make a new variable called *extension* and assign the value "arecool" to it. Then, make a new string called *longerfullfilename* that is similar to your *fullfilename*, but use insert variable *extension* just before the ".py", so the last part of the *longerfullfilename* reads "......eartsarecool.py"

d. Subsection 4: Do some math examples:
   i. Here's an example of what I want:
      1. My question:
         a. What is the value *c*, if *c* is the sum of *a* and *b*, where *a* is the product of 25 and 3, and *b* is 2 to the 6th power?
         b. You write the code in your <blahblahblah.py> file.
   ii. (1 pt each) Write the code to calculate these values.

1. What is the value of *z*, if *z* = *x* divided by *y*, where *x* is the product of 12 and 12, and *y* is 6?
2. What is the value of *c,* if *c* is the sum of *a* squared plus *b* squared, where *a* is 3, and *b* is 4?
3. What is the value of *d,* if *d* is the remainder of *a* divided by *b*, where *a* is 314, and *b* is 7?

e. Subsection 5: Decimals and more advanced calculations
   - i. As before, include all of these in your .py file.
   - ii. (1pt) Set the variable *mypi* to the approximation 22/7, but make sure your calculation results in a decimal value!!
   - iii. (2pts) Calculate the area of a circle with radius 5, using *mypi* as an approximation for *pi*.
   - iv. (3pts) You are building a budget for your proposal to NSF. You need to determine how much to budget to pay for six months (half a year) of faculty research assistant if you include salary, benefits, and OSU's indirect charges (the taxes on the entire proposal to pay for OSU upkeep!). First, calculate salary + benefits for six months, where salary is 50,000 / year, and benefits are 67% of the salary. This is your total "direct charge" costs. OSU charges 47% on that total direct charge, adding this cost to the direct charge to result in the final budget number. Show how you would use Python to calculate what that final budget number should be.

f. Subsection 6: String formatting
   - i. Set the string *town* to "Corvallis" and the string *precip* to "rains".
   - ii. (1pt) Show how you use the % and %s operators to substitute *town* and *precip* to make this statement: "It rains a lot in Corvallis."

g. Subsection 7: String formatting: The New Way
   - i. The % and %s approach for string substitution was developed in older Python versions, but the preferred way from version 2.6 forward (and we are using 2.7 here) involves the "format" method that is available for strings, much like the "len" and "uppercase" methods are available for strings (noted above).
     1. This is the grammar given in an example:
     2. "The substitutions happen {0} and {1}".format("here", "there")
        a. See https://mkaz.com/2012/10/10/python-string-format/ for a decent discussion of how to use the .format method.
   - ii. (2pts) Use the new method to achieve the same statement "It rains a lot in Corvallis"

h. Subsection 8: Converting formats
   - i. Often, we will have a number that we want to glue into a string. But numbers and strings are different types, and don't go together. We need to convert the number into a string using "str()" function, as you learned in Codecademy.

- ii. (1pt) Assign the number 12 to a variable, and then make a second variable that is a string version of the first variable.
- iii. (2pts) Use substitution and string conversion of the 12 to convert "There are twelve eggs in a dozen" to "There are 12 eggs in a dozen"

i. Subsection 9: Working with Date and Time

- i. There are many situations where you'll want to use the time or date. For example, you may want to incorporate the date you create a file to the filename, so you can keep track of different versions of your files.
- ii. This also gives us a chance to introduce another critical concept in Python: Extending functionality by using modules of code that someone else has pre-cooked for us! The **datetime** module covered in Codecademy is a good example. Someone has written a bunch of functions related to the date and time, and packaged them nicely for you to use.
  1. In Codecademy, the first part of the "Date and Time" lesson showed how to import the datetime module and use it. They had it set up to import the module, and then call functions within that module using the dot "." notation, e.g. datetime.now(), etc. Refer to that lesson for the following.
- iii. (1pt) Load the datetime module using import.
- iv. (1pt) Assign the current date and time to the variable *starttime.*
- v. (3pts) Add the day and month only to your *longerfullfilename* variable, inserting them just before the .py.

j. Subsection 10: New functions

- i. So far we have used several built-in functions to Python: execfile, print, str, len. There are more!!!
- ii. Go to https://docs.python.org/2/library/functions.html. You'll see a table of built-in functions.
- iii. Look at some of these. The descriptions of these functions are sometimes pretty heavy on computerese that will make more sense as we move through the quarter. But some should make some kind of sense. These are the base vocabulary of Python!
- iv. (4pts) Pick from the table two built-in functions we have not used yet. Show an example of using them, or if you can't get one to work, show what kind of error you get and speculate on what might be wrong.

Section 5:

Learn to spot syntax errors and exceptions in a file.

Steps

1. Load this file into your directory: S:\Geo578\Data\Lab1Data\lab1_debugfile2.py
2. Rename it to: **<yourlastname>_<yourfirstname>_lab1_part2.py**

3. There are at least five errors in that file. Try fixing them, then do an execfile("<yourfile>") to test. Repeat until you get them all fixed!
4. Use comments in the file to note your fix.

Deliverables Section 5

5. **Deliverables**
    a. (5pts) The renamed file with all errors fixed, and with comments to indicate where you fixed things.

Section 6:

Open-ended code writing. Here, you'll write a small piece of code to address a task I give to you. I won't provide any detail on how to get there.

Steps

1. Script files, as you have seen, are simply text files with python commands in them. Easy peasy. Here, we want you to write out your own file with some Python-ese in it.
2. Save your file as **"<yourlastname>_<yourfirstname>_lab1_part3.py"**
3. Choices: You can write files in your favorite text editor, or in IDLE. Next week, we'll move to a more powerful IDE.
    a. IF YOU USE A TEXT EDITOR, read this!!!
    b. Examples: Notepad, Wordpad on the Windows machine. **Do not use MS Word.**
    c. But there's a catch!!!! Many text editors will automatically attach ".txt" to the end of the file name when you write out a file. This is not good news, because you want your files to end with ".py" so the Python interpreter is happy. What's even worse, in its natural configuration, Windows won't even show you the .txt extension to your file name!!! So, you can write out the file name and manually name it something like "mycoolscript.py", and it will look fine, but when you try to run execfile("mycoolscript.py"), it will complain. You can try this if you don't believe me.
    d. So, you will want to make sure you see the full filename. Check out this how-to about how to turn off Windows' natural tendency to hide the ".txt" extension.
        i. http://windows.microsoft.com/en-us/windows/show-hide-file-name-extensions#show-hide-file-name-extensions=windows-7
        ii. Use those instructions to turn off the "Hide extensions for known file types".
    e. When you write and save your script, make sure the file name does not have the .txt extension.
        i. If it does, manually change it in the file explorer.
        ii. Alternatively, you can use double-quotes in the "save as" dialog to force your text editor to save the file name exactly as you want it.

Deliverables Section 6

1. Now, write a script to do something interesting, based on what you did at the command line earlier, or based on something more ambitious if you'd like. At a minimum, include these elements:
    a. (2pts):   Good comments at the beginning of the file to describe what the file is all about, who you are, and how to run the file.
    b. (2pts):   Do at least two string operations
    c. (2pts):   Do at least two math operations
    d. (2pts):   Do at least two date-time operations