ME30_Lab2_01Aug18-Copy1

1 ME 30 Lab 2 - Variables, expressions and statements

ME 30 ReDev Team

Description and Summary: This lab reinforces to the material covered in Chapter 2 of *Think Python*, which covers the concepts of varaibles, expressions and statements in Python. You should have read and understood Chapters 1-2 in the textbook and have attended the in-class lecture (for the ME30-specific content) before embarking upon this lab.

It is assumed that you are continuing to use the environment that you set up in Lab 1, in particular, JupyterLab. ***

1.1 Classic uses of Python vs. JupyterLab

JupyterLab is based on a previous project called iPython. Although JupyterLab is fairly new, its predecessor iPython has been slowly evolving since 2001, and the Python language has been around for more than a decade before that. Point being that iPython/JupyterLab are hardly the only way to use Python.

For convenience, we will refer to Jupyterlab or iPython simply as JupyterLab, as the former has pretty much replaced the latter.

The following ways of working with Python are those that you'll often see unless you're specifically referencing a JupyterLab resource. Almost all books and videos dedicated to learning Python, the language, use either (and often both) *interactive mode* or *script mode*, as introduced in Chapter 2 of *Think Python*, rather than within JupyterLab. Therefore, you should be familiar with how they work, and to recognize how they're different from JupyterLab.

We're using JupyterLab because is particularly well suited to many multi-disciplinary applications of Python, in particular engineering and the new-ish field of "data science". Another practical reason to use it is that it behaves very similarly under both Windows and MacOS. This makes delivering the course easier for all of us since we'll (theoretically) be spending less time having to deal with operating system issues.

JupyterLab is attractive because offers a lot of extra functionality beyond both interactive and script mode, namely being able to integrate a web-browser-like interface for mixed/multi-media information, in addition to just hosting Python code.

"Interactive mode" uses what is called a *REPL* (read-evaluate-print loop), which comes will all installations of Python running on PC-class machines. It is particularly well suited to experimenting and exploring, which is particularly useful for those new to the language.

You will be required to use notebooks for most turned-in homeworks and labs, so you **absolutely** have to learn how to use JupyterLab notebooks. However, while exploring Python, it's up to you whether you choose to use notebooks, a REPL, or script mode. You may use whichever you'd like given the task at hand. This lab will help show you why you may want to use more than just one method.

1.1.1 What is a REPL?

The REPL is just a straight-up text console window in which you are directly interacting with a Python interpreter. REPLs exist for many languages similar to Python, but Python's REPL is specifically named *IDLE*, short for "Interactive Development Environment". When using other learning resources (books, YouTube videos, etc.), you are very likely to run into both terms, REPL and IDLE, which is why we want to introduce you to them here so that you are not lost when you encounter them.

The code cells in a JupyterLab notebook are all connected to a REPL, and the *Console* tab type in JupyterLab is "similar" to a bare REPL. You'll soon see the difference.

For every complete expression or statement that you enter into a REPL Console, you will recieve a response "right away", if there is one, and then a prompt for your next input. You can do this over and over again while you test out small pieces of Python that you'll later incorporate into a more complete program.

You should understand the basics of Python expressions and statements from your lecture and this week's homework.

1.1.2 Exercise 0: Create a JupyterLab notebook using the ME30 Lab template

All of your work will go into a new notebook which you will turn in. Start with the notebook template for the course, and annotate sub-sections for each of the exercises contained herein, starting with Exercise 1.

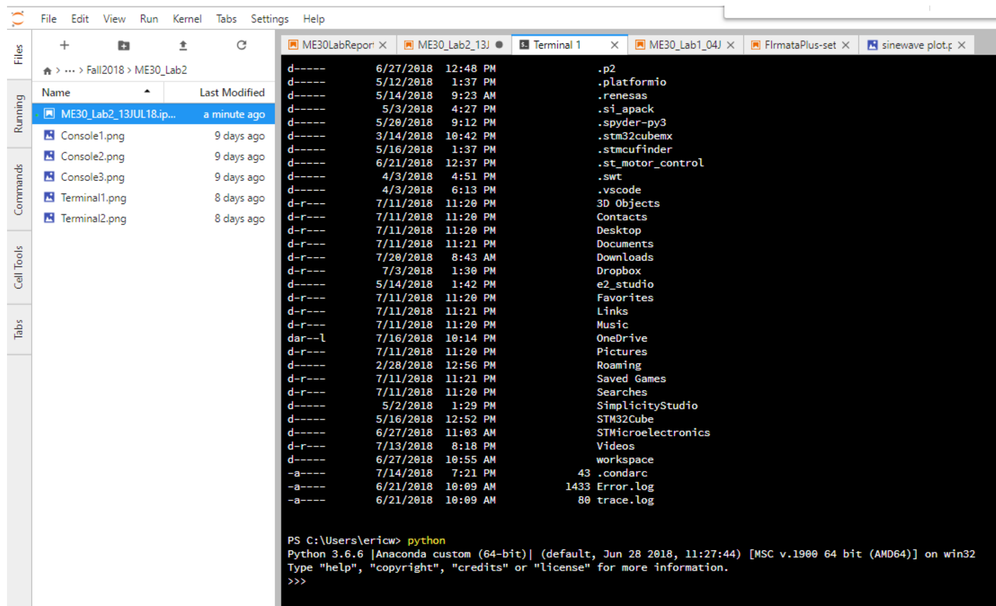
1.1.3 Exercise 1: Interacting with Python's "real" (classic) REPL

There are two ways to run Python's classic REPL, one is as a command-line program, and the other is slightly fancier, called *IDLE*. Functionally there is little difference between the two, and both are generically called "the REPL". Basic installations of Python (not Anaconda or JupyterLab) allow you to run the REPL directly from the operating system (via the Start menu or a Command Prompt on Windows, and similarly on MacOS). We are not going to demo that mechanism because there is extra software to install or configure that will only complicate things, for no real benefit since you already have access to a bare REPL inside of JupyterLab.

We will be running JupyterLab's embedded REPL using your operating system's "command window" within a JupyterLab tab. From the JupyterLab menu bar, select *File->Terminal*.

This gives you a regular-old command prompt window for your operating system. To demonstrate this, enter the command `dir` (Windows), or `ls` (MacOS), to list your files in the current working directory. The `date` command also works under both operating systems. Depending on how familiar you are working directly with the operating system, this may even be the first time that you've used your machine in this mode. Don't worry though -- this is pretty much all that we're using ancient "20th-century mode" for... :-).

Within this text window, run Python directly by entering `python`. The >>> prompt signifies that you're talking directly to Python.



2: JupyterLab Terminal window.

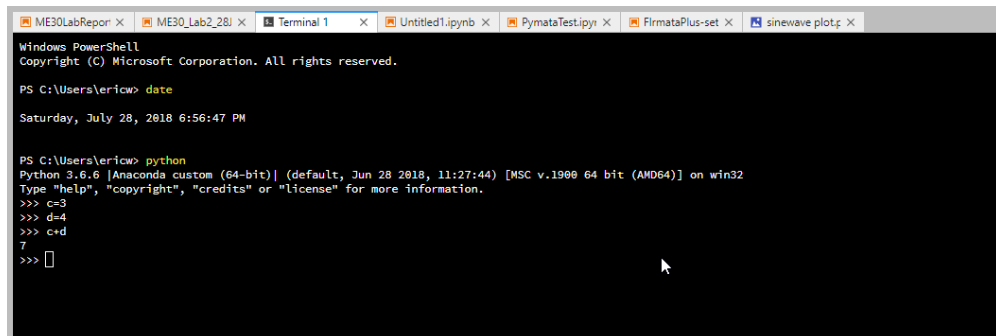


Figure 3: Terminal window REPL interaction.

Input the text as shown in the screenshot below. These statements set the value of the variable `c` to 3, `d` to 4 and evaluate the expression `c+d` based on the current values of those variables):

The REPL has a simple way of recalling and editing input that you've already entered, sometimes saving you a lot of unnecessary typing. Press your keyboard's *up-arrow* key and you will see that the last input line is recovered as the current one. Pressing the up (and down) arrow keys moves you farther backwards (and forward) in your input history. At this point your input history is only three statements deep -- the two assignments and the expression adding their values together.

Once you've recalled a previous line you want to re-enter or to edit, you can use the *right-arrow* and *left-arrow* key to navigate left-to-right, the *backspace* or *delete* key to delete characters as you'd expect. You can insert any new characters just by typing them. `<ENTER>` at any point will send this line to the REPL. Note that you do *not* have to be at the end of the line before the `<ENTER>`.

Recall the line containing the `c+d` expression (one statement back) using the up/down-arrow keys, then navigate left/right to remove the `c`, replace it with `(c*c)`, then `<ENTER>` to send it. The REPL immediately evaluates this new, edited expression and provides its response.

This method may seem a little old-timey, but many people prefer to interact with Python

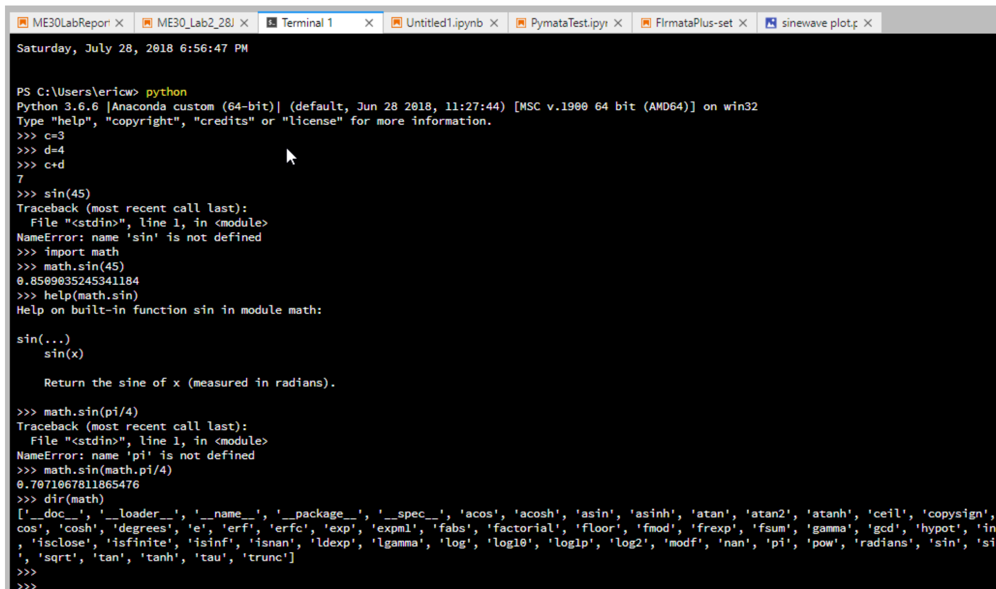


Figure 4: sin(x) REPL exploration.

this way because your hands don't have to go back-and-forth between the keyboard and the mouse/trackpad and buttons, as one has to do with a fancier, graphical editor.

Another example of something that you may want to try quickly is exploring a triginometric function like `sin(45 degrees)`, but you can't quite remember what it takes to use it.

Start by trying `sin(45)`. OK, it doesn't know what/where the `sin` function is. But you (OK, your future self!) remembers that many math functions are in the `math` library.

Try `math.sin(45)`. OK, no `math`... oh yeah, enter `import math` to suck in the `math` library.

Up-arrow twice and to re-try `math.sin(45)`. Hmmm, that doesn't look right.

Enter `help(math.sin)`. OK, `sin` requires radians... duh. Remembering again that up-arrow is your friend, try `math.sin(pi/4)`.

Fuh.... that's right, `pi` is also in the math library. Try `math.sin(math.pi/4)`. OK, that looks right. Now you remember how to use trig functions in your program.

Wondering what else is in the `math` library? Enter `dir(math)` for the directory of things in the math library.

Hey, the repl isn't so bad after all, once you know what you can do with it pretty quickly and easily.

The orderly way to exit the REPL is to enter `quit()` which returns you to the command prompt. When you leave the REPL all of the variables that you created are lost, so if you want to come back and continue where you left off, then don't quit out of it. However if you want to start fresh all over again, quitting and restarting will certainly toss everything out. Enter `quit()` now, but you do not need to close the Terminal window tab as we'll be using the window again later.

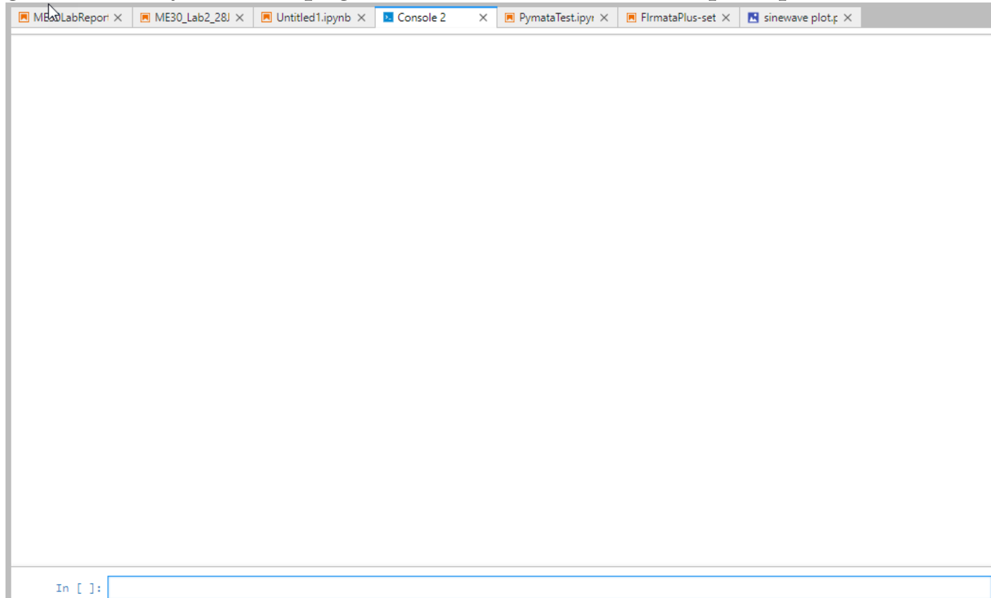
Once you get a little more experienced and comfortable with using Python, you may find that this way of accessing the REPL is the easiest way of exploring Python for quick, little tests. However, the downside is that it's difficult (maybe impossible) to cut and paste text contained in this window into another place, like a Jupyter notebook, at least on Windows (FIXME what about MacOS?).

Since you can't copy-and-paste proof of your work in this Console window into your turn-in notebook, use your operating system's screen-capturing feature to snapshot the relevant portion

of your screen for inclusion in your notebook. Use the syntax from Lab 1 into include these saved image(s).

1.1.4 Exercise 2: Interacting with JupyterLab's *Console* REPL vs. a JupyterLab notebook

JupyterLab has a *Console* feature that is basically the same as the REPL you were just introduced to, but with the user interface of a JupyterLab notebook. You can run it from the Anaconda menu bar by selecting *File->Console*. Do this now. You will get a nearly blank page with a notebook-like prompt at the bottom of the screen:



Enter the pair of lines below in the prompt field at the bottom of the page, then shift-Enter to send that batch of input to the REPL.

```
c=3
d=4
```

You'll see these two assignments echoed in the upper output area, but no output because no expressions were entered that generate a result.

Next, similar to the previous exercise, type in the next two lines in one batch, and again, *shift-Enter*.

```
c+d
(c*c)+d
```

and you should see something similar to below:

Well, isn't that curious...?! Only the value of the last expression was printed. Let's try entering just (followed by *shift-Enter*, of course):

```
c+d
```

OK, apparently when entering a batch of input statements, only the last expression's value is printed. In fact, the experiment below demonstrates that no expressions' results are printed

```
Python 3.6.6 |Anaconda custom (64-bit)| (default, Jun 28 2018, 11:27:44) [MSC v.1900 64 bit
(AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 6.4.0 -- An enhanced Interactive Python. Type '?' for help.


In [1]: c=3
        d=4

In [2]: c+d
        (c*c)+d

Out[2]: 13
```

Figure 5: Expression batch in Console window.

except for the last one, and only if it's the final statement in the batch. We'll spare you the effort to confirm this, but it turns out that notebook code cells behave the same way.

Perform a screen-capture of your work in the Console window and include it in your turn-in notebook.

As you can see, this Console window behaves very similar to a notebook (which you used in Lab 1), except that it *only* knows how to execute Python code. To be honest, the JupyterLab console appears to be not much more than a stripped-down notebook. It has none of the text formatting features, or the ability to change code that you already are working on in a Code cell, and easily re-execute it. Its only real advantage over the character-based REPL that was demonstrated above is that you can cut-and-paste lines from one part of the screen back into the input window for editing and re-execution. It only seems to exist because it clearly is a REPL, but has more modern style that you're used to than the conventional REPL.

Frankly, it seems to have no advantages for you (as ME30 students) over what you can accomplish in a notebook using code cells. Since most of your work this semester will need to be turned-in as a notebook, so you might as well just use a notebook. At this point you can probably say Good-bye to the JupyterLab Console because it offers no value that notebooks have, and is actually more difficult to use and less responsive to the classic REPL that you were first shown.

1.2 What is so different about script mode vs interactive mode?

Script mode is the non-interactive mode for running programs that's as old as Python. Before iPython notebooks, your only real choice to run programs in Python was to put your python code into a file (usually with the extension `.py`) and provide its name to the *python* program on the command line.

However, in JupyterLab, you have the ability to work on and execute on modest-sized programs right in a notebook, so it is kind of a one-stop-shop for doing both REPL-like exploration and program writing. With JupyterLab there's no reason to isolate your code into a separate file just so that you can run it. Again, you will likely read elsewhere about using a separate text editor to work on your code in a separate file and then executing it in a separate window -- which is basically what script mode is all about. Jupyterlab pretty much eliminates the distinction between interactive mode and script mode.

To demonstrate this we are going to write our first small program from scratch. A Python program is a series of Python statements that accomplish some task that you need to solve. How-

ever, just like any engineering task where you are building something of any non-trivial size, you start will smaller pieces which you assemble and test in sub-assemblies and then bring together in a final product. At that point, you're done programming the tool in Python and you switch to actually using the tool that you created.

1.2.1 Exercise: Building a small program

To make this exercise even the slightest bit interesting, we will have to introduce you to make use of some concepts that have not yet been discussed in lecture or the book. We will keep these excursions simple, so don't panic..!

Consider the generic form of the line-slope equation, from algebra:

$y = mx + b$.

The goal is to be able to interactively explore this equation for a bunch of x values to see how y changes. For the purposes of this example, we'll start off by fixing m and b to constant values, 7 and 4, respectively. We want the values for y for all integers x from *0* to *5*, inclusive.

Let's assume that you're *still* not super comfortable with your Python syntax, and you want to make sure that you know understand what all of the pieces need to be before putting them together. That's what interactive mode is for. You can choose whichever interactive mode you'd like to use, either a notebook, a Console window, or a Terminal window.

Since you're doing this task and it has to be turned-in as a JupyterLab notebook, we'll demonstrate how to do this there. You are free to use whichever method you like, as long as you show your work in all of your code cells, or with screenshots (to be embedded in your notebook), if using the classic REPL.

Setting up constant values It's normal to set any variables whose values to not change (that is, constants) first in a program. We've already decided to fix m and b as *7* and *4*, so let's do that first.

```
m=7
b=4
```

Knowing what you already know about the form of an algebraic expression in Python, and assigning it to a variable (in this case, y), you should hopefully be able to figure out that this looks like:

```
y = (m * x) + b     # in our case y = 7x+4
```

Enter these three statements into a code cell in your turn-in notebook and execute them.

Well... that's unfortunate -- an error! Why is that?

For a Python statement to be executed, all referenced variables need to be valid at the time that statement is executed. Even though these statements are syntactically perfectly valid Python, this last statement cannot be executed because `x` does not yet have a value. This is an example of a *runtime error*, as opposed to a *syntax error* which is (always) syntactically invalid.

Another example of a runtime error is:

```
a=3     # this is OK, but... wait for it...
a / (a-a)
```

Another runtime error would be trying to open a file on your computer to read data from, where that file does not exist. If the program does not check that the file exists before trying to open it, you'll get a runtime error for that also. You'll learn more about issues like these in future weeks.

The first x value we want to evaluate is 0, so let's start with that one. In a *new* code cell, enter the following updated code. Note that you would normally just add the new statement to the existing code cell when working on your own program, but we're requiring you to show your work progression.

```
m=7
b=4

x=0
y = (m * x) + b     # in our case y = 7x+4, and specifically for x=0
```

Evaluate the code cell and notice what happens to the output. You should see that there is *no output*. Remember what was said earlier about how expressions only display their value if: - the statement is an expression with a value - you're in interactive mode (in a REPL)

Referring back earlier in the lab, modify your code cell to display the value of `y`. To confirm that your mini-program is working correctly, manually compute what y should be and compare it what this Python statement comes up with. If these values don't match, don't proceed until you've figured out why there's a discrepancy.

In a new code cell: For *x=1*, edit the statement that sets `x` and recomputes the output for this new input.

If this was what programming was all about, we wouldn't have smart phones for another few hundred years. We're going to use one of the new Python statements we warned you about that you haven't been introduced to yet.

In a new code cell -- enter the following code and run it:

```
m=7
b=4

for x in [0, 1, 2, 3, 4, 5]:
    y = (m * x) + b
    print(y)
```

You may be able to figure out basically what's going on. Our program is starting to be useful by using this `for` statement. However, it's not super clear which output values correspond to which x value, so let's make the program do that work for us also.

In its simplest form, you can tell `print()` to print multiple values by handing it a comma-separated list of values to display, rather than just one item. **In a new code cell, use `print(x, y)` to display both the input and output values.**

1.2.2 Exercise: Let's be flexible

Again, working beyond our current means a little bit, let's suppose that we don't want to have to change the program every time we want to try new m and b values. We'll ask the user to enter them before we run the equation over 0...5, rather than forcing them (or us) to change the program every time in order to evaluate a different point-slope equation.

One gets keyboard input from the user with the `input()` function. The only problem is that input gives you a string, and this has to be converted to an integer value. You'll learn more about these and similar conversion functions in the next chapter, but we're going to play with just one right now.

In a new code cell, try this little test. You should see a little input mini-field open up below the cell that you type into, which was created by calling `input()`.

```
print('Enter a number:')
user_input = input()
print('OK, it looks like you input:')
print(user_input)

n = int(user_input)
print('It appears to be the number, half of which is:')
print(n/2)
```

Because `user_input` is a string, if you want to do math on it, you have to convert it to an integer before doing so. And as long as you enter something that can be interpreted as a number, calling `int()` to do the conversion on this string works nicely.

Just for "fun", re-run the code cell and give it your name as input. Yeah... not happy. We'll also see how to catch these errors gracefully in the coming weeks.

Your final task is now to create a new code cell by modifying the version of the point-slope code with the `for` statement above so that it asks for and uses both the m and b values, while still evaluating x from 0 to 5.