Week 1 Coding Lecture 1

How to read these notes

Lecture notes for this class will (mostly) be written in Jupyter notebook files through Google colab, which have a .ipynb extension. These files mix normal text (like this) with python code

```
2 + 3 * 4
```

```
14
```

The code block above displays the results of some arithmetic.

Text in a Jupyter notebook is divided into cells. If you click on any text in the document, the cell containing that text will be highlited. If you hover over the `[]` you will see a "play" button (or use the hot key `Ctrl+Enter`) to run the cell. If the cell just contains text, then probably nothing will happen, but if the cell contains python code, then Jupyter will run that code and display any output below (or possibly to the side of) the cell. Try to run the next cell several times.

```
import random
random.randint(0, 10)
```

```
9
```

The code in the previous cell produces a random integer between 0 and 10. Each time you run it, you should see a different output.

Writing your own code

You will probably never need to write a .ipynb file yourself. Instead, you should use a python console or a python script file.

Python Console

To open a python console in PyCharm, you can either click the button near the bottom of the screen labeled "Python Console" or go to the Tools menu and click the option "Python or Debug Console". You should see a tab with some information about the current version of python (hopefully 3.8.x), as well as a prompt labeled `In[1]:`. (The number may be different, but it is not important. It simply indicates what line of input you are on.) This prompt indicates where you can type your next line of code. Try typing the code from the previous block in the console. You should see the same output as before (i.e., a random integer between 0 and 10).

The console is useful for quick calculations, but it has some serious drawbacks. For one thing, you can't edit and rerun code you have already typed in the console. Instead, you have to retype it at the prompt. For another, nothing in the console is saved when you exit. You have to redo all work from the console every time you start PyCharm.

Scripts

Because the console is so limited, you should write most of your python code in scripts (files with a `.py` extension). You can make a new script by right-clicking in the project file explorer (typically in the top left of the window) and choosing New->Python File.

Scripts are essentially just the code blocks from a Jupyter notebook like this one without the text in between. Try opening a new script named `my_script.py` and typing the code:

```
import random
random.randint(0, 10)
```

```
2
```

(Side note: Python is pretty forgiving about file names. Any name that is valid on your operating system is allowed, as long as it ends in .py, but it's wise to be a little careful. In particular, I strongly recommend that you only use letters, numbers and underscores in your file names and that you start the file name with a letter. It is particularly unwise to use spaces in file names. Windows has a bad habit of including spaces in names by default, but other operating systems do not handle them well. If you expect anyone else to run your code - and at minimum you need Gradescope to run your homework - then it is a bad idea to use spaces in the file name.)

As you have probably noticed, nothing happens when you type this code. To run the code, you need go to the "Run" menu and click "Run". The first time you do this, you will need to choose the file `my_script.py` from a dropdown menu. You can also use the hotkey Alt+Shift+F10.

Note that there is a button with a green arrow at the top of your script. This button will run your script as well, but you have to be careful. By default, this button runs the last script you ran, not the script you are currently editing. You need to make sure that you are running the correct script.

A new tab should now appear in PyCharm (probably taking the place of the console you had open before) labeled "Run: my_script.py". There will not be much text in this window, except for the words

```
Process finished with exit code 0
```

Note that the number 0 here is not the result of our random integer generator. Instead, "exit code 0" means that the file ran successfully. (If you see any other exit code or any error messages, then you should double check that you copied the code above correctly.)

This is an important difference between the console and scripts. The console automatically prints results after every line of code, but scripts do not display output unless you specifically tell them to. (Jupyter notebooks are somewhere in between. Jupyter will display the output from the last line of each code cell by default, but not the output from other lines.) We can tell python to display the output of a line with the `print` command. We will talk about the `print` command later, but for now just know that wrapping a line in `print( )` will cause python to display output.

Try changing your script to the following:

```
import random
print(random.randint(0, 10))
```

```
2
```

and then run the script again. (You should be able to use the green arrow button now. Just make sure that the file name beside it is `my_script.py`.) You should now see a number between 0 and 10 before the "Process finished with exit code 0" line. Notice that each time you run the script you get a new number, demonstrating that PyCharm really is re-running your code.

This example is quite short, but in general your scripts will contain many lines of code. MATLAB runs each line in order from top to bottom (and, for the most part, only displays any output if the line includes a `print` command). Next week, we will learn how to make python repeat or skip lines of code or to run them out of order.

I highly recommend that you write all code longer than a single line in a script from now on. It will save you a lot of trouble.

Comments

It is often useful to leave explanatory text in a script. If you begin a line of code with a # sign, python will ignore that line. (Actually, the # doesn't have to be at the beginning, although it is good practice. Any text after the # will be ignored.) For example,

```
# This is a comment.  Python will ignore it
# when you run this code block.
```

Basic calculation

You can use python as a very complicated calculator. You are probably familiar with most of the syntax already, but here are some examples:

```
print(1 + 3)
print(4 - 5)
print(2 * 3.2)
print(4 / 3)
print(3 - 2 * 6)
print(8 * (1 + 2))
print(5 ** 3)
```

```
4
-1
6.4
1.3333333333333333
-9
```

```
24
125
```

As you can see, python uses most of the standard symbols for basic mathematical operations and follows the usual order of operations. The one symbol you might not be used to is the double asterisk `**`. This symbol is used for exponentiation, so `5**3` means 5 to the power of 3.

Importing Packages

Python also has all of the usual mathematical functions you would find on a scientific calculator, such as $\sin$, $\cos$, $\tan$, $\log$, etc. However, they are not defined in the core language. In other words, we can't just use the code `sin(3)`. If you try to add that code to one of these code cells, it may or may not work. This is a quirk of Jupyter notebooks - if you have already run some of the later cells then Jupyter will remember that you defined the function `sin`.

In order to use these functions, we have to tell python how to find them. These functions are defined in a package called `numpy`, which we installed when we installed Anaconda. We can tell python to use this package with the line

```
import numpy
```

We can now use any function defined in the numpy package. For instance, if we want to use the $\sin$ function to calculate $\sin(3)$, we can write

```
numpy.sin(3)
```

```
0.1411200080598672
```

The line `import numpy` tells python how to find any function definition supplied by the numpy package, and the prefix `numpy.` before `sin` tells python that it should look in the numpy package for the definition of a function called `sin`. We need both of these because it is entirely possible for two different packages to define functions with the same name. (Indeed, this is true for `sin`. Another package called `math` includes a `sin` function, but that version is often slower and so we do not want to use it.)

It quickly becomes tedious to type out `numpy.` every time we want to use one of these functions. There are two good ways (and several other bad ways that we will not discuss here) to avoid that issue. First, if we only need one specific function, we can import it directly. For example, we could get the `cos` function with the code

```
from numpy import cos
cos(10)
```

```
-0.8390715290764524
```

Notice that we do not have to prefix the `cos` function with `numpy.` anymore.

Alternatively, we can replace the package name (in this case "numpy") with something shorter. As an example, we could use

```
import numpy as np
np.tan(5)
```

```
-3.380515006246586
```

Now we can use the name "np" in place of "numpy". We will need the numpy package so often in this class that it is probably wise to start all of your scripts with the line `import numpy as np`.

Variables

So far, we have not been able to do anything with python that isn't already available (and much simpler) on a scientific calculator. One of the most important features of python, along with every other programming language, is the ability to save values for future computations. Python has pre-defined several variables for us and, more importantly, allows us to define our own variables whenever needed.

To start, let's look at a pre-defined variable. In many mathematical calculations we need to use the value of π. We could type it in as 3.14159, or however many decimal places we needed, but it is much more useful to have python remember the value for us. Fortunately, the numpy package includes just such a feature.

```
np.pi
```

```
3.141592653589793
```

(Notice that we used `np` instead of `numpy`. This works because of the import command from the previous cell.) The name `np.pi` is a stand in for the number 3.14159... We can use this name anywhere in our code that we could write a number. When python sees the text `np.pi`, it automatically replaces it with the appropriate number. For example, we could calculate

```
np.pi + 3
```

```
6.141592653589793
```

```
np.cos(3 * np.pi)
```

```
-1.0
```

```
3 * np.pi / 2
```

```
4.71238898038469
```

Assignment

Of course, we can't rely on numpy to define every variable we want. Instead, it is much more convenient to be able to make our own variables. To this end, we can use the assignment operator `=` to assign numbers to variables. The `=` symbol means "calculate the value on the right and assign it to the name on the left." For example,

```
x = 8
```

assigns the value `8` to a variable named `x`. After running this code, python will store the number 8 somewhere in memory. We can retrieve this value whenever we need it by using the name `x`. For example, we could print it out using

```
print(x)
```

```
8
```

or we could use it in other calculations such as

```
(x + 12) / 4
```

```
5.0
```

```
x - 2
```

```
6
```

```
x * 5.0 - x
```

```
32.0
```

Notice that in all of these calculations python simply replaces all instances of `x` with `8` and then proceeds with normal arithmetic.

If we want to change the value of `x`, we can just reassign it with the `=` operator. For example:

```
x = 3
```

The value of `x` is now `3` instead of `8`. To see this, we can print it out, or use it in a calculation:

```
print(x)
```

```
3
```

```
x + 10
```

```
13
```

This doesn't change any previous results (the output from the previous lines is the same), but all future occurrences of `x` will use the new value. (Actually, Jupyter notebooks behave a bit differently than scripts in this regard. The issue is that each line of a script is always run in order, so any use of `x` above the line `x = 3` will use the old value. In a notebook, however, you can run cells in any order you want. If you run the cell with `x = 3` and then scroll up and run the cell with `x - 2`, you will see the output 1 instead of 6. This is one of the reasons that Jupyter notebooks are not the preferred tool for computation: We generally want to specify the order in which code will be run and not let the user re-organize things.

There is no reason to restrict ourselves to just one variable. As long as they each have a different name.

(Side note: Python has a few rules for naming variables. Variable names can be as long as you want and can contain letters, numbers and underscores, but they must start with a letter or an underscore. Variables that start with an underscore have a special meaning and you should avoid them. This means that `x`, `answer_number_4` and `int2str` are all valid variable names, but `3test`, `x-y` and `answer number 4` are not. Names are also case sensitive, so `ans1` and `Ans1` and `ANS1` are all different names. That said, it is not a good idea to have different variables that differ only by case. There are also a few reserved words that python uses for other purposes and you cannot use as variable names. For example, `function`, `for`, `while`, `if` and `end` are all reserved words. Python will give an error if you try to use one of these as a variable name.)

As an example,

```
x = 8
y = 3
var27 = 0
```

We can now combine these variables just as if they were numbers.

```
z = x - y + 2 ** var27
print(z)
```

```
6
```

It is very important to remember that you can't use a variable until it has been assigned a value. If you try to do a calculation with a variable that you have not yet defined, python will give an error along the lines of "name is not defined". For instance,

```
3 * N
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-27-f66c9acabd59> in <module>
----> 1 3 * N
      2

NameError: name 'N' is not defined
```

SEARCH STACK OVERFLOW

You will undoubtedly see this error message many times in your coding career. There are three very common reasons such an error might arise.

1) You forgot to define the variable, or defined it in the wrong place (for instance, you defined it in a later line or in another file).

2) You misspelled a variable name. For instance, you already defined `my_variable` and then tried to use the code `myvariable`.

3) You are trying to use a pre-defined variable, but did not tell python to use the proper package. For example, you tried to use `pi` instead of `np.pi`.

It is important to remember that the assignment operator is not the same thing as mathematical equality. In particular, it treats variables on the left and right side differently. If a variable appears on the right side of an `=` operator, python replaces it with whatever value it finds in memory. For instance, the line `x - y + 2 ** var27` above is replaced with `8 - 3 + 2 ** 0`. On the other hand, if a variable appears on the left side then python does not look up its value. Instead, python stores the value on the right in the variable name on the left (overwriting any value that used to be stored under that name). One example that comes up a lot is

```
x = 8
x = x + 1
```

The last line doesn't make any sense in mathematics, but it is perfectly sensible code. Python evaluates this expression as follows. First, it sees an `x` on the right side of the `=` operator, so it looks up its value in memory. Since `x` has the value `8`, python replaces the right side with `8 + 1`, which is `9`. Next, python sees the name `x` on the left side of the `=` operator, so it stores the value `9` in the memory reserved for the variable `x`. If later lines of code use the variable `x`, it will have the value `9`.

There are a few things worth remembering about assignment and variables in python:

1) The left side of the `=` operator is the name of a variable. You can't have any calculations or other code - just a single, valid variable name.

2) If the name on the left side is already the name of a variable in python's memory, then python will overwrite the value of that variable with the result from the right hand side. This will not affect any previous calculations with the variable.

3) If the name on the left side is not already the name of a variable in python's memory, then python will reserve space for a new variable with that name and give it the value from the right hand side.

4) The right hand side can be as complicated as you want and can include code involving many different variables, but those variables must all be already defined.

Types

Variables in python have a property called a type. This type determines what sort of operations can be applied to the variable and what those operations do. For example, if we define

```
x = 8
```

then `x` has the type `int` (which stands for integer). Because `x` is an integer, it makes sense to add other integers to it, and so python can execute the code

```
x + 2
```

However, if we define

```
y = "This is a string of text"
```

then `y` has the type `str` (which stands for string). This is the type python uses to denote text. It does not make sense to add an integer to a string of text, and so the following code does not work:

```
x + y
```

You can determine the type of a variable with the function `type()`. For instance,

```
type(x)
```

```
type(y)
```

The other type that will come up very often is a float. Float variables are decimal numbers, such as `np.pi`. (Note that the type that python calls a float is called a double in many other languages, including MATLAB.)

For the most part, we will ignore the types of variables in this class. One of the nice features of MATLAB, as opposed to python, is that almost all variables have the same type and so you can get through the entire class without worrying about the concept. Unfortunately, there are a few cases in python where type issues will be unavoidable. We will talk about these issues more when they arise.

Displaying output

The vast majority of code you write for this class will be in python scripts. As we have already seen, when you run a python script it generally does not display very much output. This is not always very useful. As a general rule, we are writing code because we want the answer to a problem, and so it is important to be able to get python to display that answer in an informative way. Our primary tool for this purpose will be the `print` command. We have already seen that you can use the print command to display the value of a variable, such as

```
x = 8
print(x)
```

```
8
```

You can also use the print command to display text, like this:

```
print("This is some text that will get displayed")
```

```
This is some text that will get displayed
```

The quotation marks around the text are required. This is what tells python that the words are text (technically, these quotes define a string variable) and not code.

There are two very important ways you should use the print command. First, you should use lots of print statements to make sure that your code is working as expected while you write it. We will demonstrate this idea many times throughout the class, but it is good to get in the habit of running your code every time you add a line or two and printing out the values of different variables to make sure that they are correct.

Second, you should use print statements to display and format the final answers in your code. For example, you might write dozens of lines of code to calculate the velocity of an object. Once all of your code successfully runs, the final answer will be saved in a variable called `x`. Of course, at minimum you will need to include the code `print(x)` so that you can see the answer, but it is a much better practice to write

```
print("The velocity of the object (in meters per second) is: ")
print(x)
```

```
The velocity of the object (in meters per second) is:
8
```

(Actually, there is a better method involving a function called `format`. We will demonstrate `format` later, but if you are interested you can look it up now.)

When your code (and in particular, your homework) is finished, it should print out the relevant answers in an easily readable format like this and it should not print out anything else. In particular, you should remove any print statements that you added for debugging purposes and you should make sure that there are no error messages.