**MEA 462 - Observational Methods and Data Analysis in Marine Physics**

Introduction to Python Programming

Joseph B. Zambon

*Session 2: 31 March 2021*

*Introduction to Modules, Variables, Calculations, Assignments and Arrays*

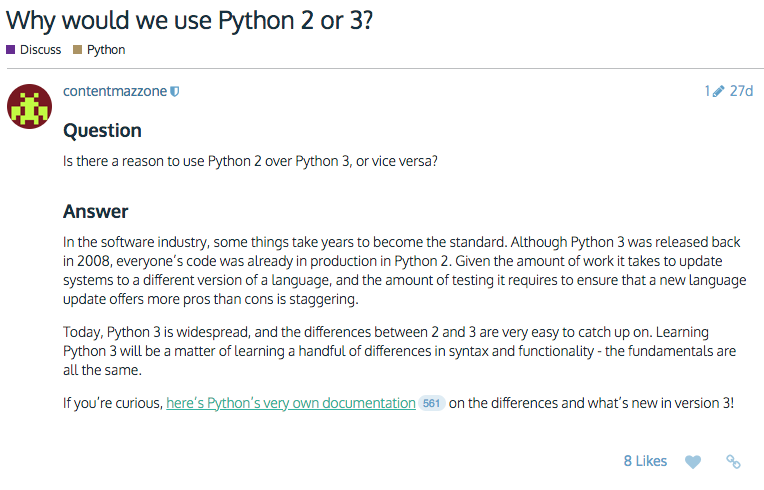
Note: The previous lesson demonstrated installation of Anaconda Navigator and iPython Notebook using Windows 10. Since the remaining lessons are independent of your chosen OS, I will be utilizing my native Mac environment. This should make minimal difference in executing code.

**Modules**

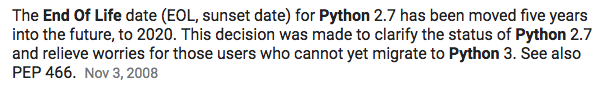
As I mentioned in the previous lesson, we will not be using Python 2.x codes at this point, but you may run into them in the future. Here are some notes on the subtle syntax changes

<http://python-notes.curiousefficiency.org/en/latest/python3/questions_and_answers.html>

An additional note regarding Python 2.x vs Python 3.x (<https://discuss.codecademy.com/t/why-would-we-use-python-2-or-3/297315>).

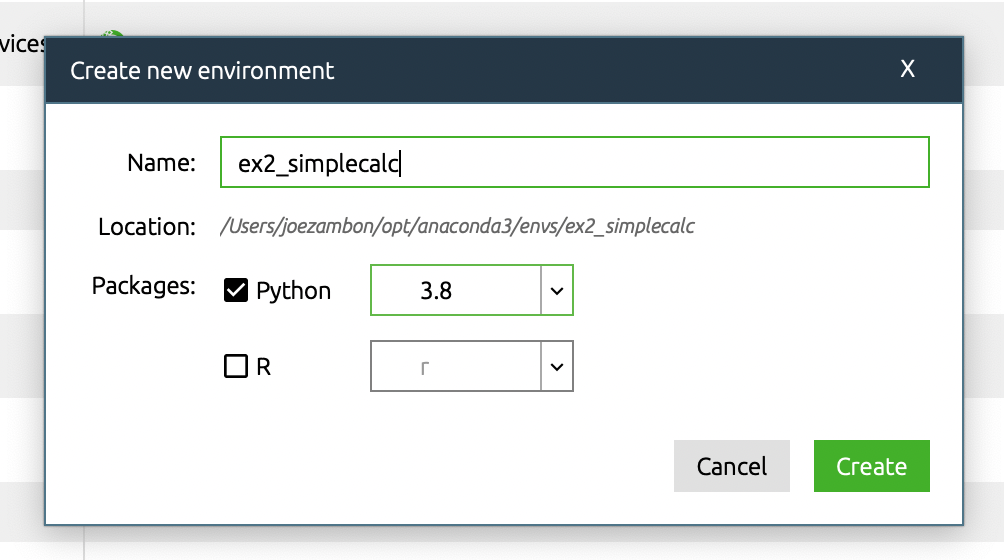


Python 2.7’s end of life (EOL) was extended several times but finally elapsed in 2020. There will be no more updates to Python 2.7 codes.

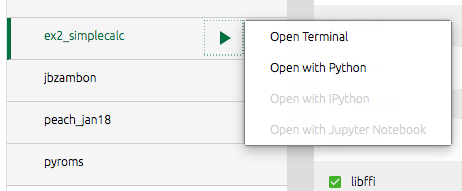


(Also see: <http://pythonclock.org/>)

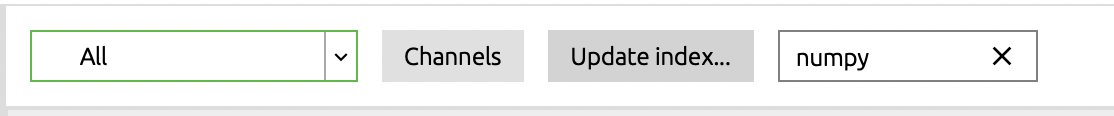
Start by creating a new environment. I’m simply calling this one ex2\_simplecalc. Again, make sure you’re installing for Python 3.x.



Since this is a new environment, it does not include a number of the required packages to run Notebook, and we will need to reinstall these.



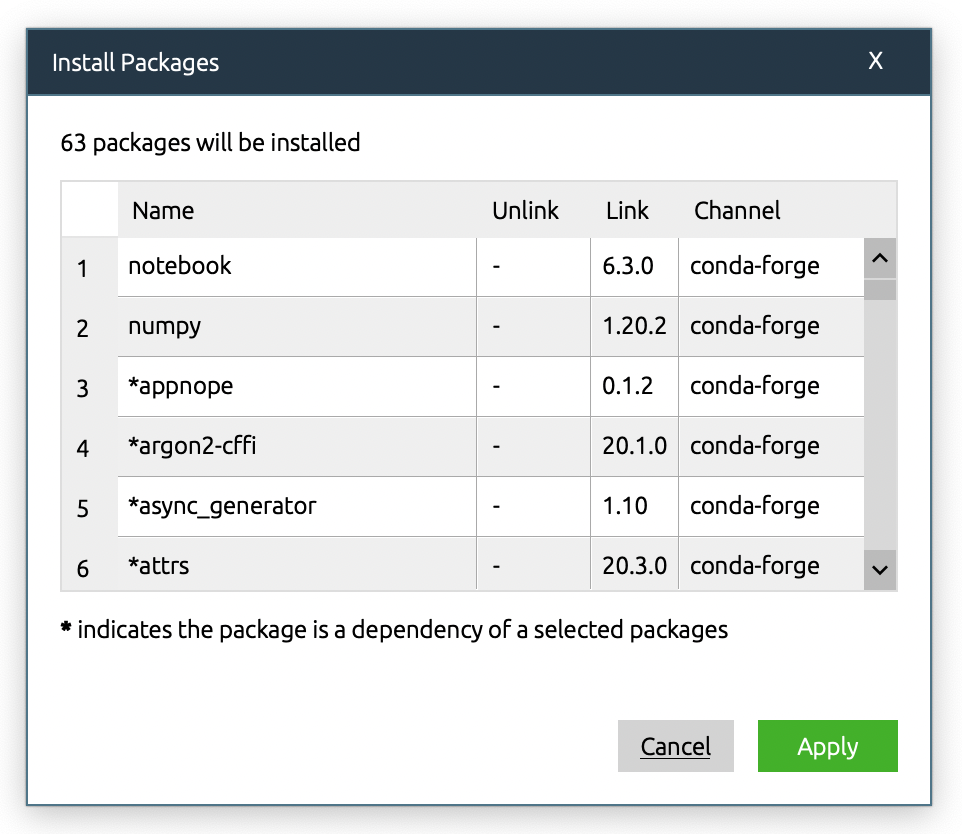
I had trouble finding the second package, “numpy” that we will be using. It did not get referenced by default so click “Update index…” if you’re unable to locate it.



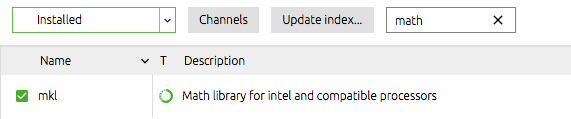
Install packages and related dependencies for “notebook” and “numpy”. Note that you can select multiple packages to install at the same time. This will save you some annoying wait time.

Graphical user interface, text, application, email

Description automatically generated



We will also be using the “math” and “sys” modules, but they are typically installed by default.



Anytime you’re not sure if you have the proper modules installed before you begin coding, a quick check is to open up a Python terminal and “import \_\_\_\_\_\_”. If no errors appear, you should be good to use. In the remaining part of this tutorial we will be using numpy, math, and sys.

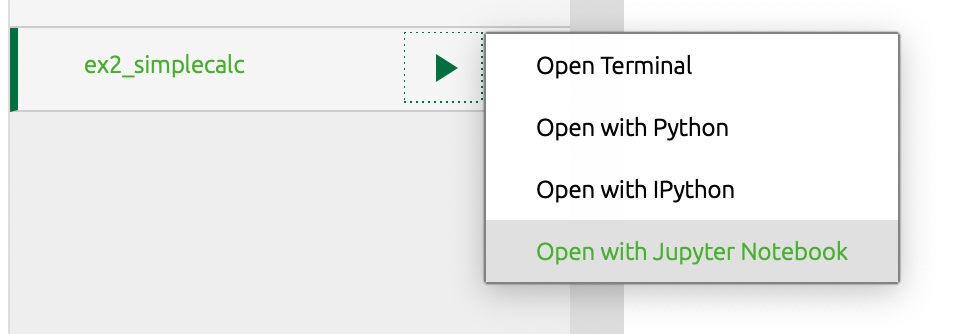
Graphical user interface, text, application, chat or text message

Description automatically generated

Text

Description automatically generated

After the packages are installed, open up iPython Notebook, create a new notebook, and we will begin coding.

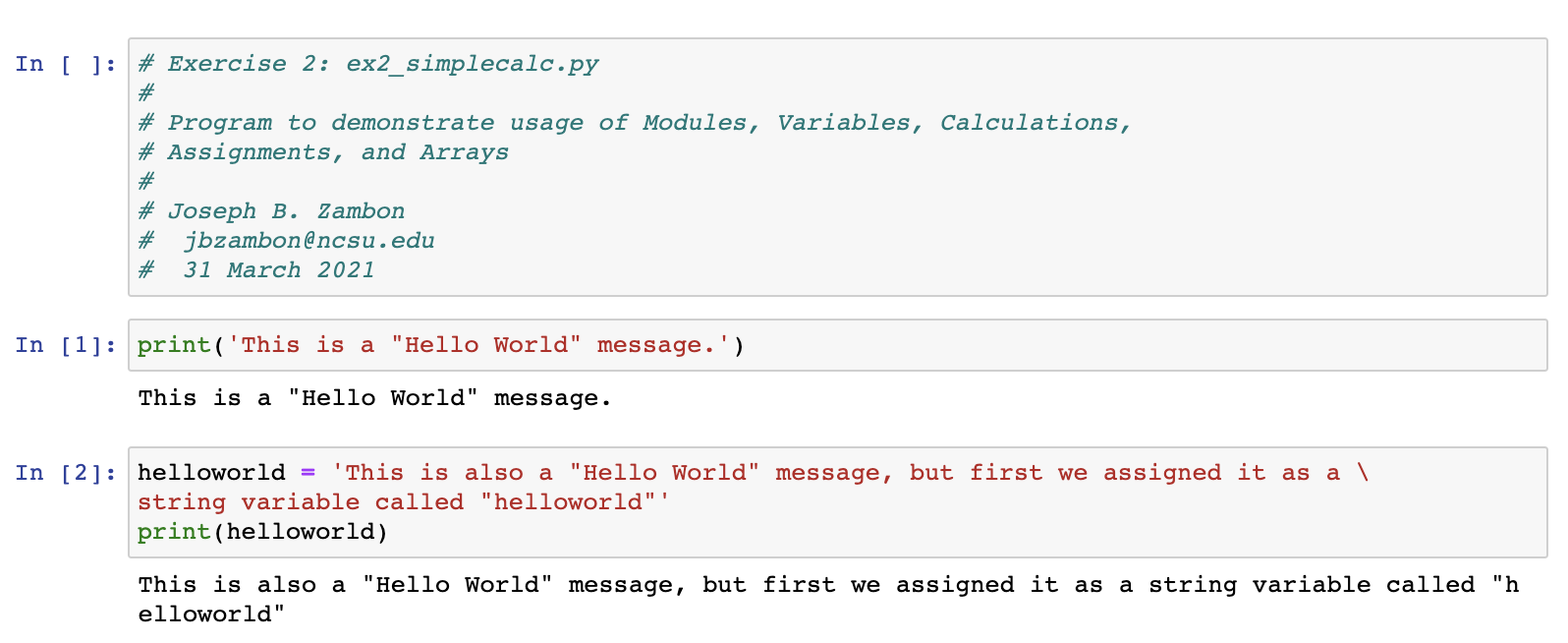


Graphical user interface, text, application

Description automatically generated

**Strings**

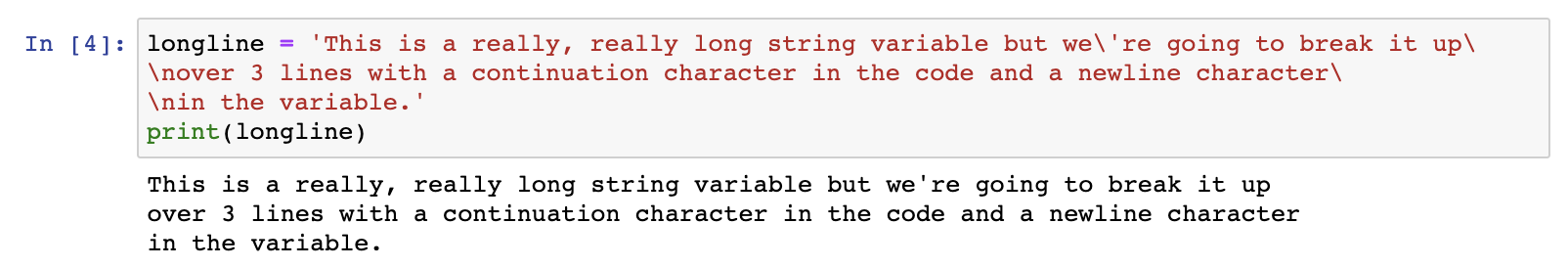
In the last lesson you output a simple message “Hello World.” You simply made a string and inserted it into a print statement. We can up the complexity a bit here, let’s create a variable and print that instead. Variable creation and assignment can be as simple as this example.



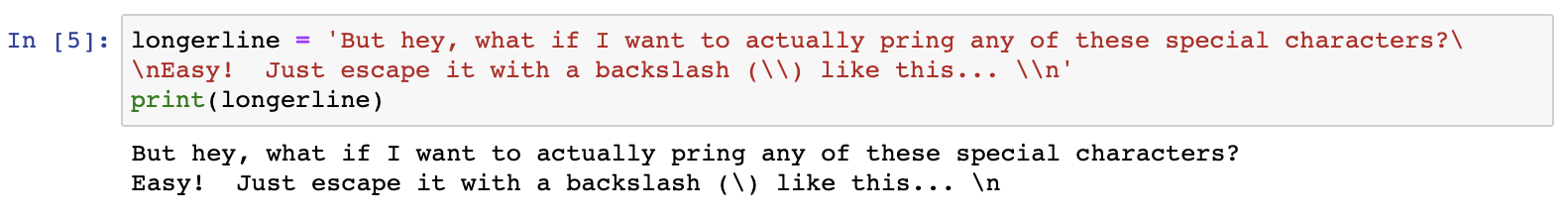
Notice that the string is broken across 2 lines arbitrarily by the size of the window? I am not thrilled with the way that looks. To clean up the code, we can use a continuation character “\” which allows you to continue the string to the next line. If you don’t use a continuation character and instead simply hit ENTER to get to the next line, you will produce an error as Python doesn’t understand where the end of the string is. Python, like Matlab and other programming languages, is very dependent on proper formatting!

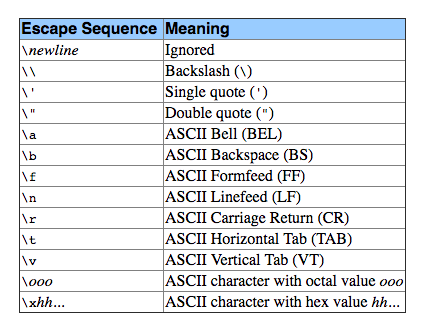


To get the string formatted correctly, use the continuation character (\) and newline character (\n) to break it up.



Notice that in order to get the apostrophe (‘) to print correctly, we had to “escape” it with a backslash (\)? This applies to any special characters you run into, and comes from the C programming language (reference below).





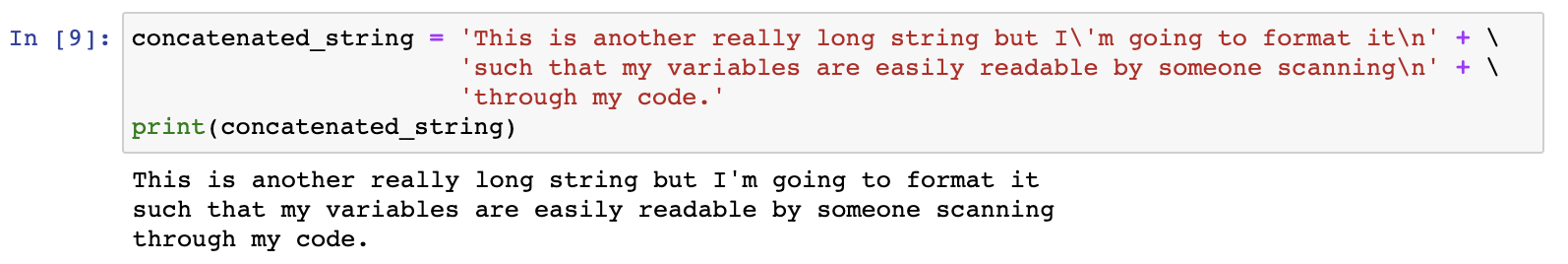
(Reference: <http://docs.python.org/2.0/ref/strings.html>)

I’m going to continue to be pedantic and demand that we properly format our code. Typically I like to indent variables so that the entire code is easily scannable. If you indent the string with a continuation character, these spaces will carryover to the actual print statement.

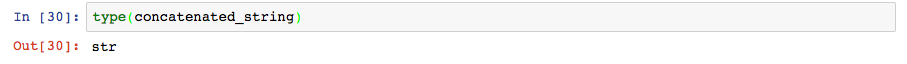
A picture containing text

Description automatically generated

By using a plus (+) between the different strings, we concatenated (coding lingo for “joined”) them together. This works for strings, using a + for numeric variables (integers, floats, etc) will literally add them together.

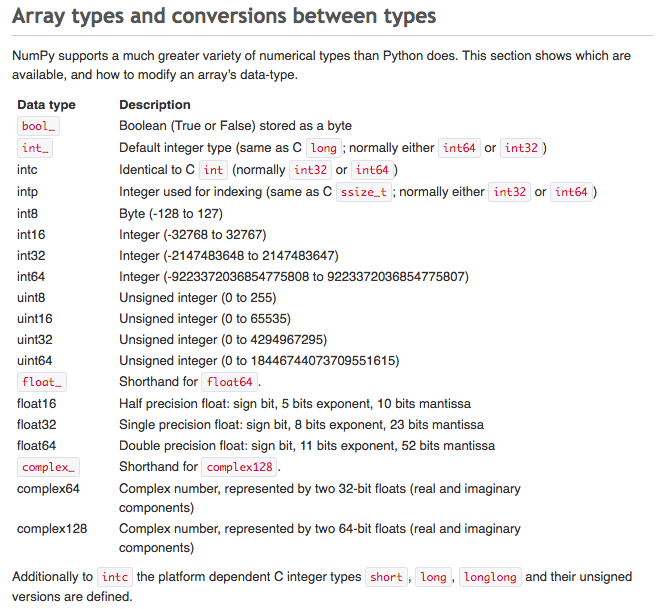


How do I know if my variable is a string? Easy! Use the type() function on any variable to find out what it is.



**Numbers**

So that covers the string type, what about numerical variables? Typically it is good practice to define variables that use the least amount of memory possible to deliver your desired result. Out-of-the-box Python doesn’t offer many choices. However, if you load the “numpy” module, you get lots of options to choose from!



Also, Python is more efficient than C in utilizing memory space since Python tags variables instead of creating memory space for every variable (Reference: <http://foobarnbaz.com/2012/07/08/understanding-python-variables/>).

Module – “out-of-the-box” Python is designed to be very lightweight, a big advantage over Matlab. Modules are additional packages that can be loaded into your program, on-demand, complete with definitions and functions.

While you can import modules anywhere in your code that you need them, good practice is to import all necessary modules at the beginning of your program. This offers a few advantages. First, if you are debugging your code, you can run the single cell and know that cells below it will have the requisite modules loaded. Second, if you later remove a section of code that doesn’t require the module, you can find it easily later. Third, if you’re running this code outside of your typical environment, you can see what modules you need to have installed right at the start. [As we discussed above you can run Python in the environment first to make sure everything is set up correctly.](#module_load)

To add a cell below your header (first) cell, just click to the left of the cell window (below “In [ ]:”). The entire cell is now highlighted in blue, meaning you’re editing the cell in relation to the rest of the program (rather than what is inside of the cell itself). This allows you to move the entire cell up/down as well as add cells above/below it.

Graphical user interface, text

Description automatically generated

Note that when you’re working within a cell, it is highlighted green.

Text

Description automatically generated

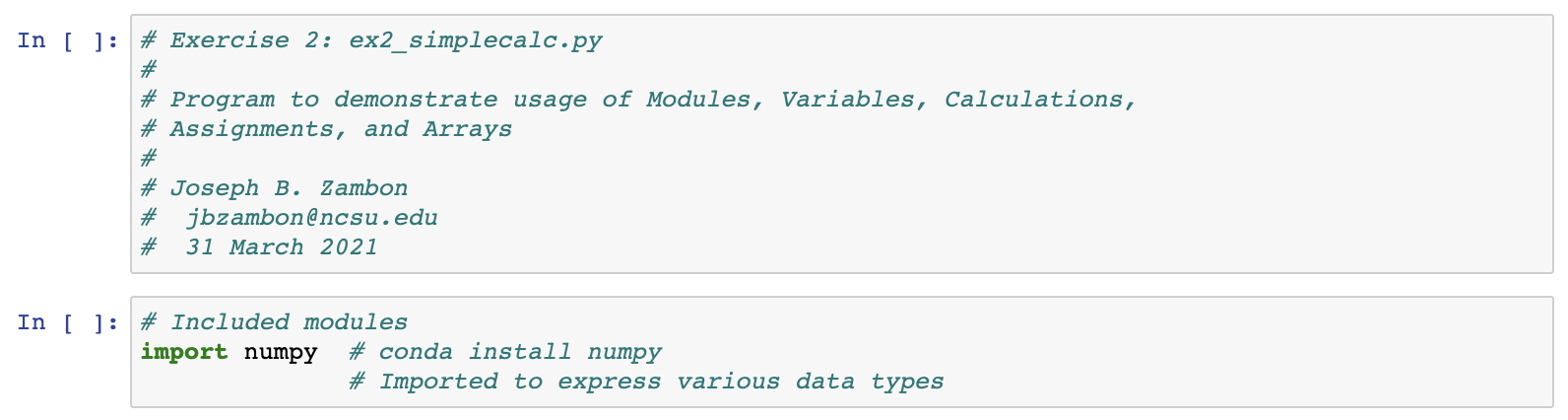
Now to actually add the cell, click Insert -> Insert Cell Above/Below

Graphical user interface, text, application

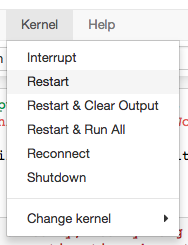
Description automatically generated

There is also a keyboard shortcut for Above (a) and Below (b). Just make sure you’re editing outside of the cell (blue highlight) otherwise you will just be adding a’s and b’s to your code within the cell.

It is also good practice to comment the package name that the module comes from to make other people easily run your code using Anaconda/conda, “conda install numpy”. Not all modules are named from their respective package, this one just happens to be).



Once you run the cell and import the module, it will remain in your memory space unless you shutdown or restart the kernel.



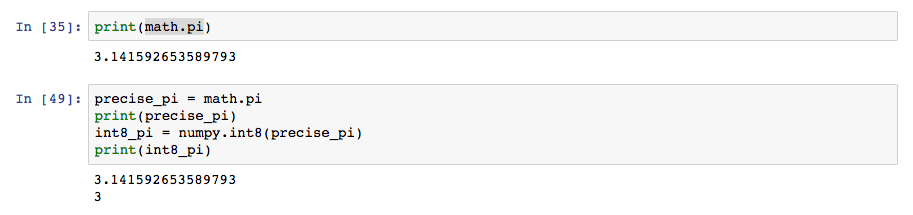
Now that we have numpy installed, let’s go over some easy mistakes in saving as much memory as possible. A big advantage of Python over Matlab is that it is extremely lightweight and extensible, you can even run code on a $30 Raspberry Pi! As a result, keeping your code as lightweight as possible is a good coding practice.

As you can see from [the table above](#array_types), int8 is the smallest possible variable type with minimal precision (-128 to 127) but only uses ONE BYTE (8 bits) of memory. Let’s intentionally give ourselves errors with precision. What’s a good number that requires lots of precision? Pi!

Import the “math” module, as it has pi to double precision. Define a variable “precise\_pi” and define it with pi from the math module, then convert the variable to int8 precision. Print the 2 different results.

Text

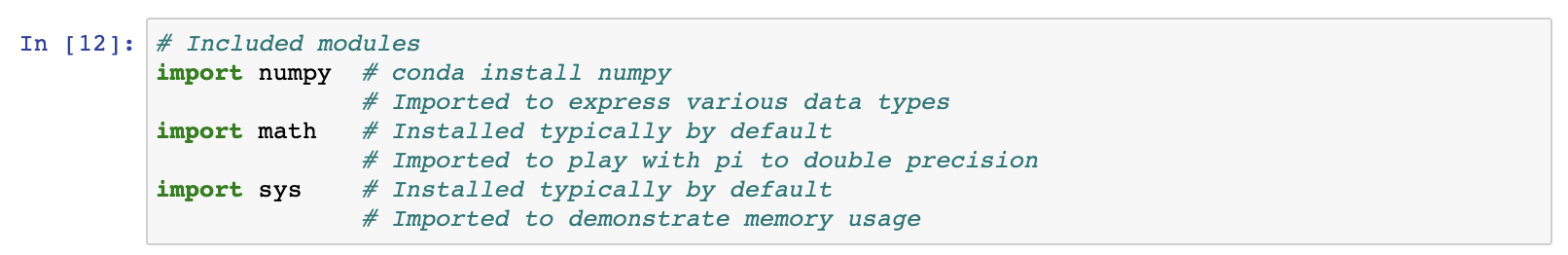
Description automatically generated



3.141592653589793 versus 3, that’s quite a difference in precision!

(We’ve been talking a lot about bits and bytes, here’s a reference video if you need a refresher on how binary works <http://www.youtube.com/watch?v=LpuPe81bc2w>)

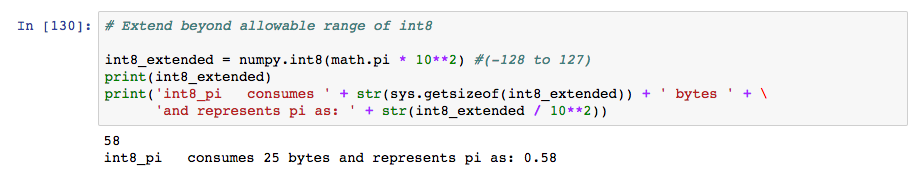
Let’s calculate the circumference of the Earth with differing precision. This will allow us to define an array, call a function, and do some very simple math, and print formatted statements. Import the “sys” module as we will be using the getsizeof() function to demonstrate memory usage. You should now be using 3 modules: numpy, math, and sys.



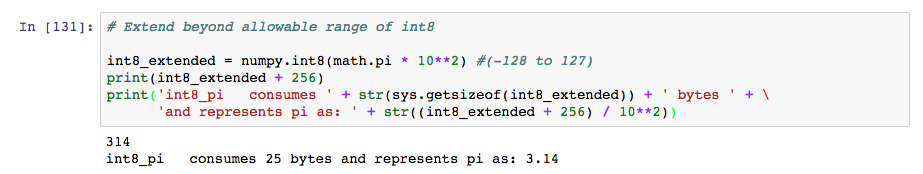
How can we represent pi with the most precision but least memory? A simple trick is to define it as an integer and move the decimal over by multiplying by factors of 10 while keeping it within the defined limits. To get the answer, you simply divide by that same power of 10.



What happens if you go beyond the allowable range? If we try to multiply Pi by 100 you get 314. This extends beyond the allowable range of a signed 8-bit integer (-128 to 127). Pi is now represented as the number 58, why?

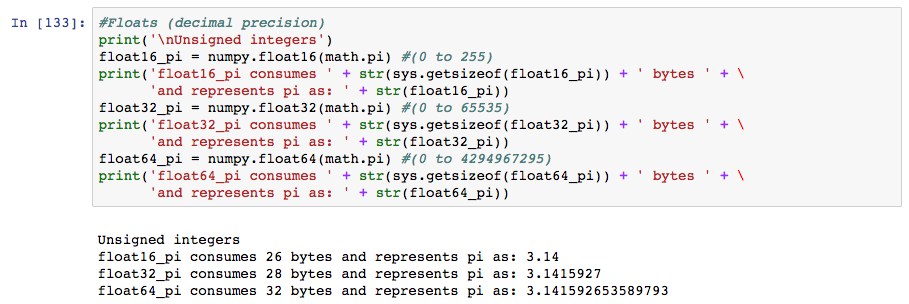


Recall that the number of available elements in an 8-bit number are 256 (-128 to 127). Add 256 (number of elements in 8-bits) to 58 and what do you get? Your answer, 314!



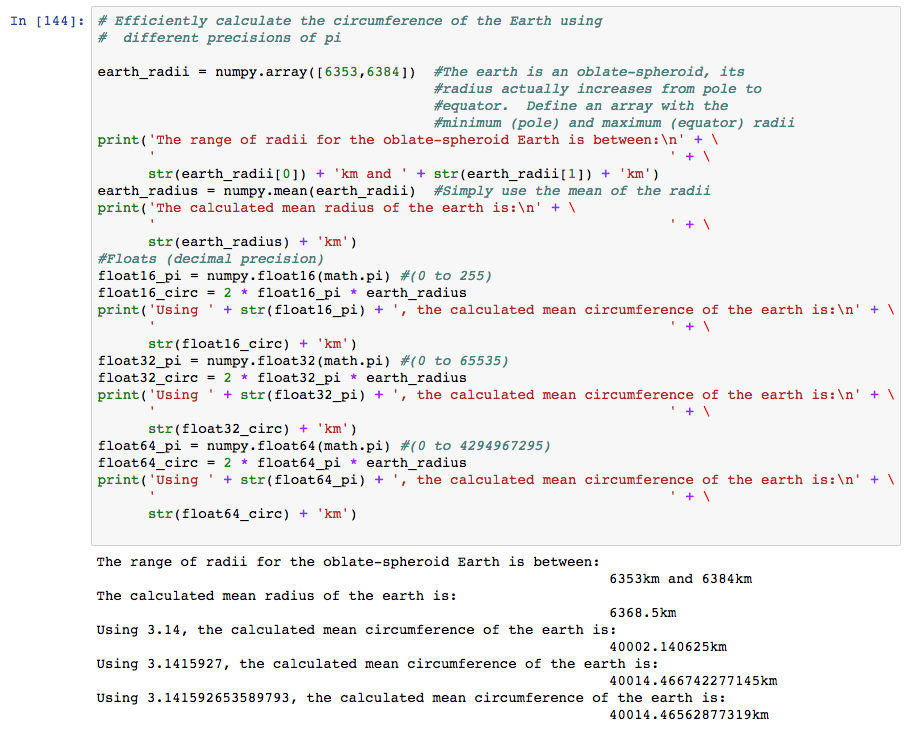
The number 58 came as the result of a coding bug known as Integer Overflow (<http://en.wikipedia.org/wiki/Integer_overflow>). For a funny (i.e. nerdy) tidbit about Integer Overflow, Google “Nuclear Gandhi” (https://en.wikipedia.org/wiki/Nuclear\_Gandhi).

So, this method of multiplying by a factor of 10, then dividing by the same factor of 10 seems to be a great way to represent long strings of decimal numbers, right? Well, we’re not the first to think of this. You just worked through how Floating Point Arithmetic works! (<http://en.wikipedia.org/wiki/Floating-point_arithmetic>) Just like integer arithmetic, or any computer logic, it uses a number of bits to define numbers.

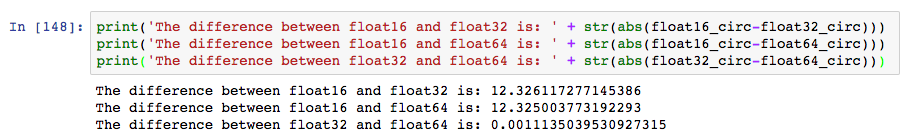


Now that we know floating-point numbers use the same amount of memory space, let’s start using them to finally get to the answer of the Earth’s circumference. Go back to the cell we were previously working on and define Pi as various floats. We know that Pi goes on and on and on, but how precise do you need to be? Weigh precision against memory usage for your particular application while considering Significant Figures (<http://en.wikipedia.org/wiki/Significant_figures>).

The rule of multiplication of significant figures states “For quantities created from measured quantities by multiplication and division, the calculated result should have as many significant figures as the measured number with the least number of significant figures.” We’re working with 4 significant figures for our Earth’s radius, so Pi should not have less than this (i.e. rounding 3.141592653589793 to 3.142). As a result, float16 does not have sufficient resolution and float64 has too much resolution with float32 in the “Goldilocks” zone.

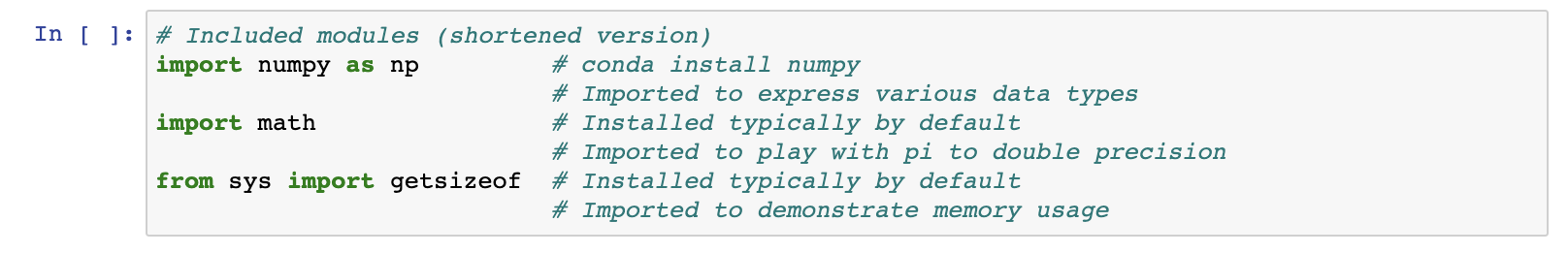


What’s the difference between the different values of float16, float32, and float64 representations of Pi? (Note: the built-in “abs” function returns the absolute value.)

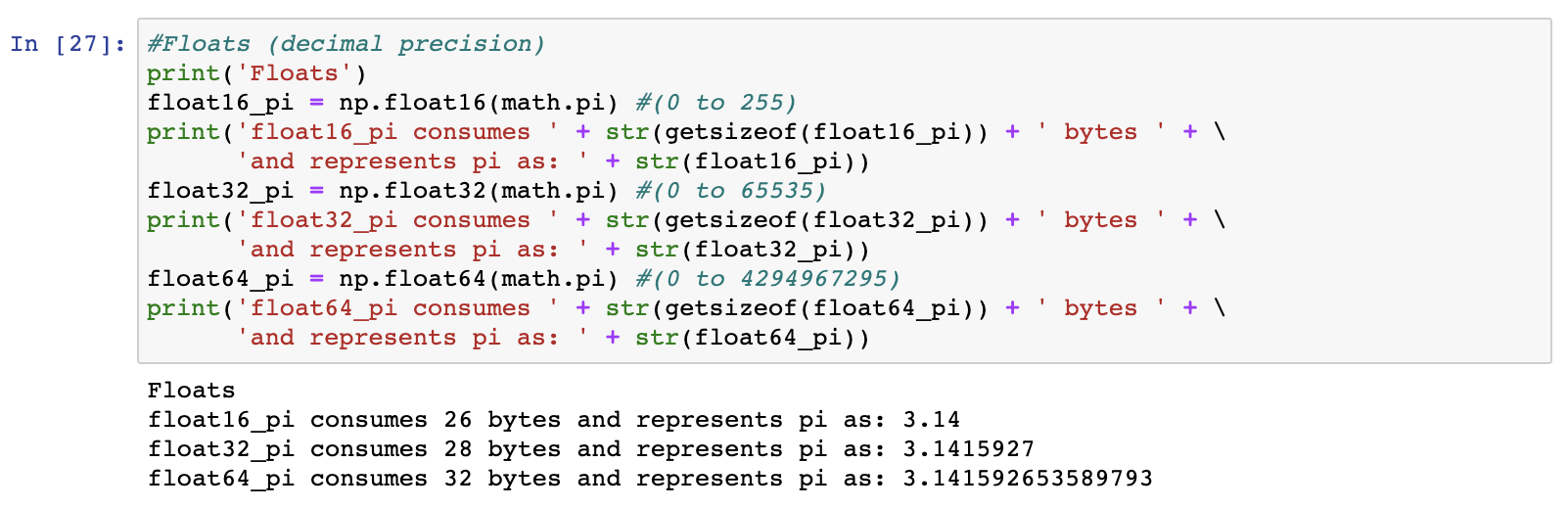


This again demonstrates that in this case, using Significant Figures, float32 is sufficient to resolve Pi in calculating the circumference of the Earth when using radii with 4 significant figures.

Some final notes… having to type the module name every time can get pretty annoying. Thankfully, Python has a couple of shortcuts to prevent this in your module declarations. You can shorten the code by “import X as Y” and import specific routines within modules as “from X import Y”. For example, we used the numpy routines frequently and we only used the getsizeof() routine from sys. We can clean this up. Here’s how the module declarations change.



Here’s how some of the revised code looks.



Congratulations! You have now used modules, variables, calculations, assignments, and arrays in Python. Along the way you learned how computer logic calculates results differently using differently sized memory allocations – a common pitfall to programming in any language!

With this foundation, we will now get to the good stuff – applying Python code for scientific computing, and sharing those codes with the world!