MEA 462 - Observational Methods and Data Analysis in Marine Physics

Introduction to Python Programming

Joseph B. Zambon

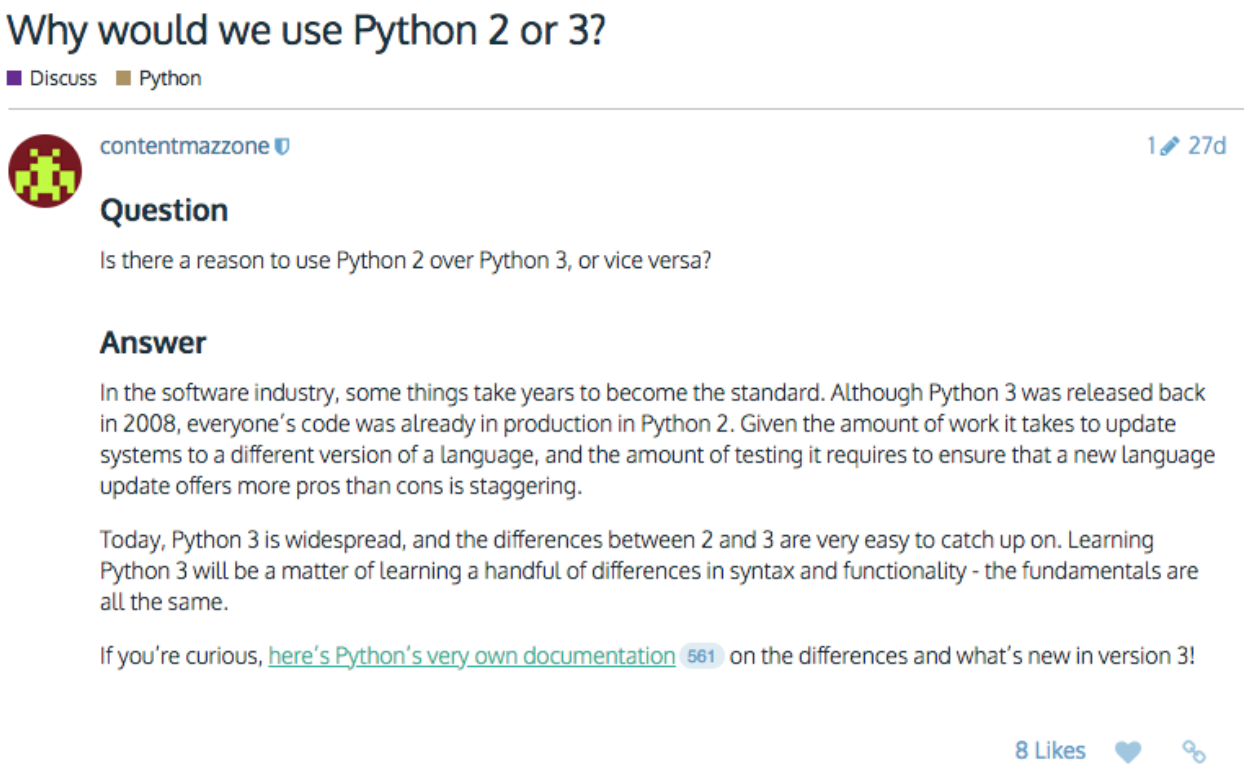
Session 2: 31 March 2021
Introduction to Modules, Variables, Calculations, Assignments and Arrays

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).



Why would we use Python 2 or 3?

Discuss Python

contentmazzone 1 27d

Question

Is there a reason to use Python 2 over Python 3, or vice versa?

Answer

In the software industry, some things take years to become the standard. Although Python 3 was released back in 2008, everyone's code was already in production in Python 2. Given the amount of work it takes to update systems to a different version of a language, and the amount of testing it requires to ensure that a new language update offers more pros than cons is staggering.

Today, Python 3 is widespread, and the differences between 2 and 3 are very easy to catch up on. Learning Python 3 will be a matter of learning a handful of differences in syntax and functionality - the fundamentals are all the same.

If you're curious, here's Python's very own documentation 561 on the differences and what's new in version 3!

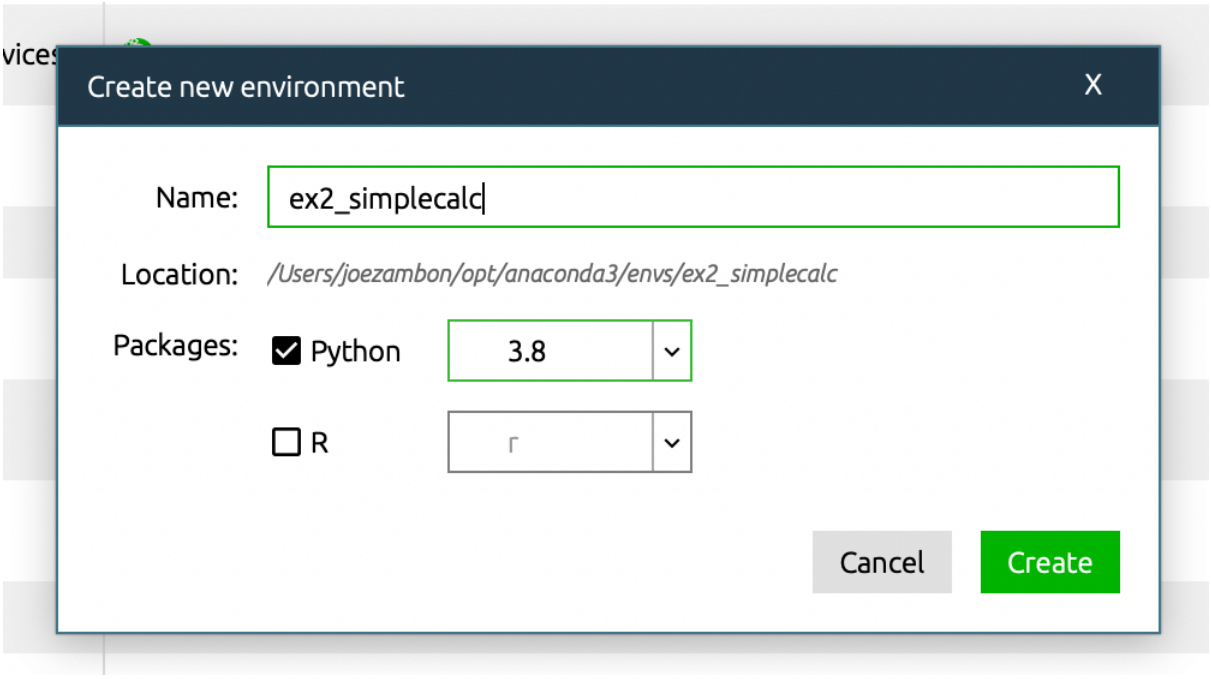
8 Likes

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.

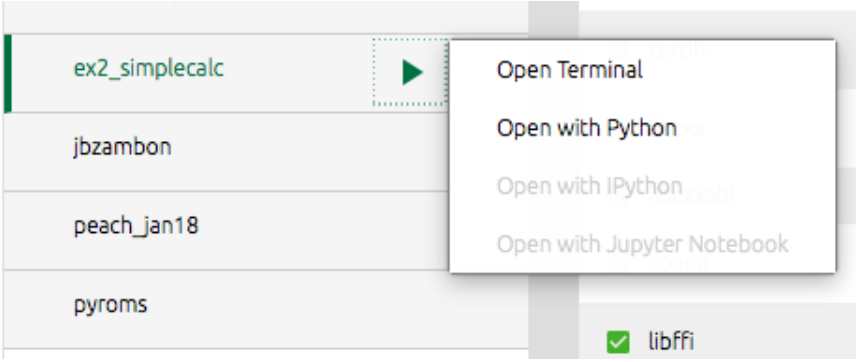
> The **End Of Life** date (EOL, sunset date) for **Python** 2.7 has been moved five years into the future, to 2020. This decision was made to clarify the status of **Python** 2.7 and relieve worries for those users who cannot yet migrate to **Python** 3. See also PEP 466. Nov 3, 2008

(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.

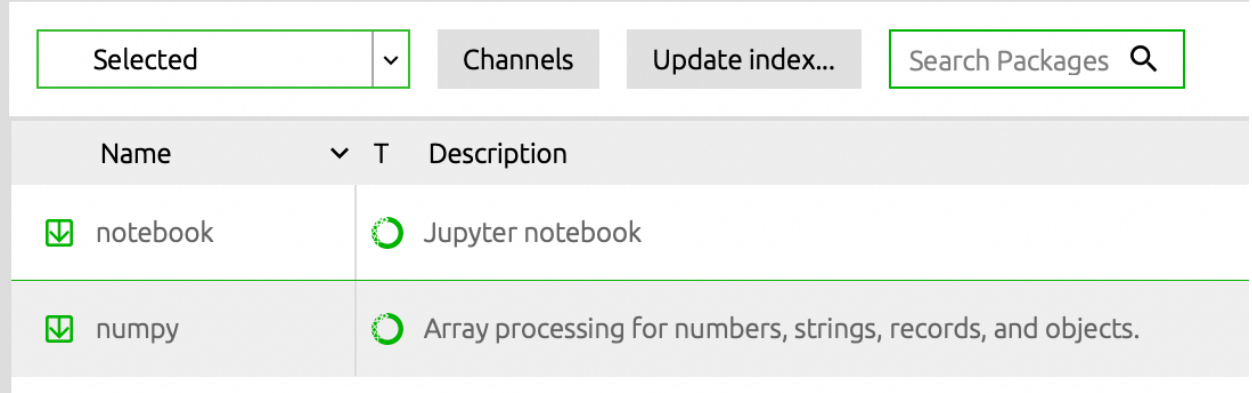


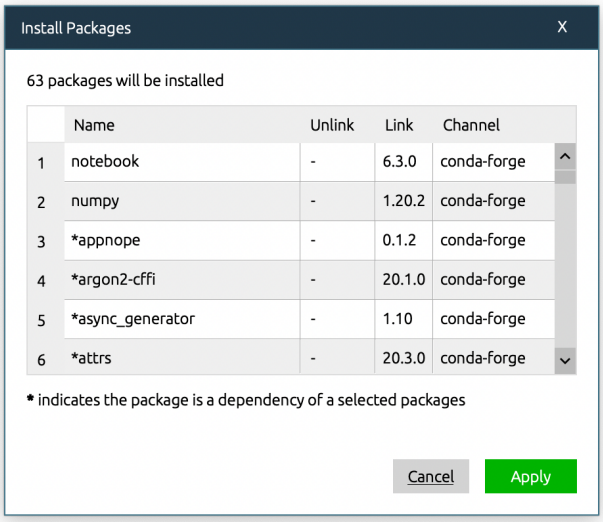
Since this is a new environment, it does not include a number of the required packages to run Notebook, and we will need to reinstall these.



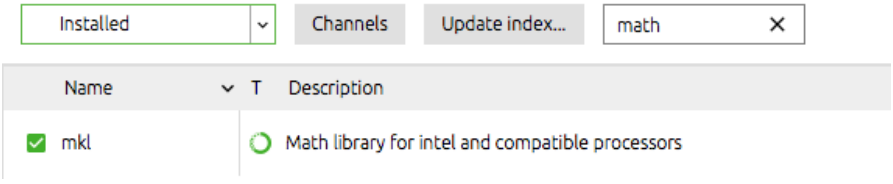
I had trouble finding the second package, “numpy” that we will be using. It did not get referenced by default so click “Update index…” if you’re unable to locate it.

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.

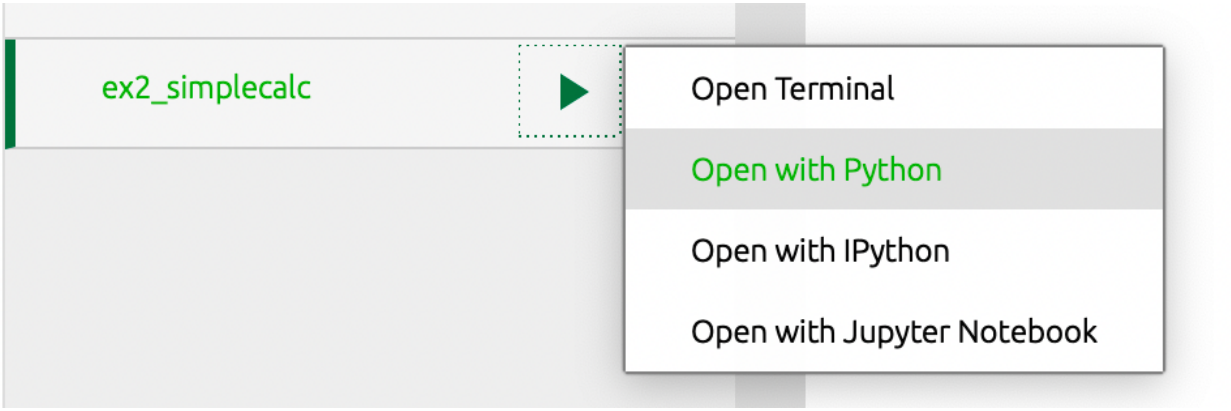


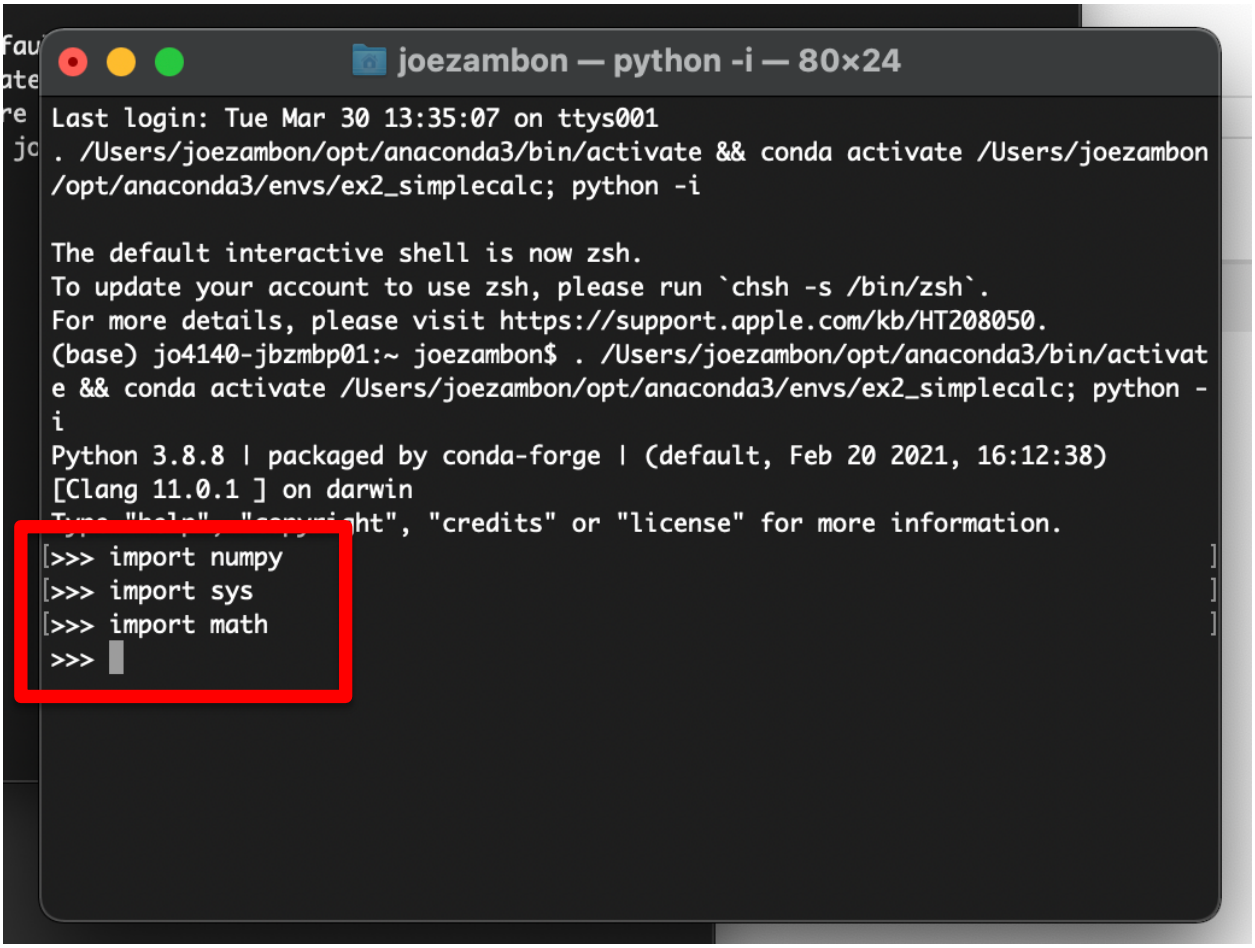


We will also be using the “math” and “sys” modules, but they are typically installed by default.

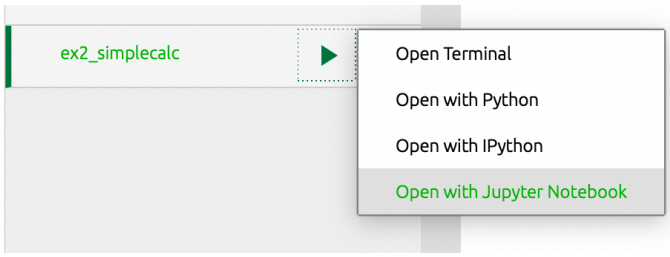


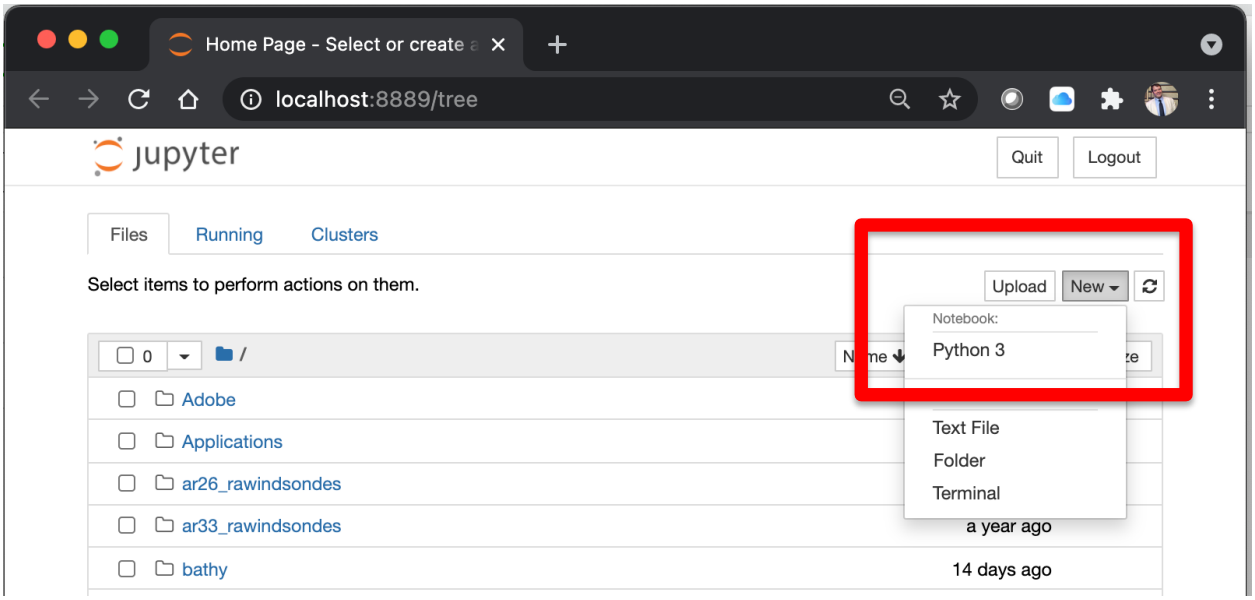
Anytime you're not sure if you have the proper modules installed before you begin coding, a quick check is to open up a Python terminal and "import _____". If no errors appear, you should be good to use. In the remaining part of this tutorial we will be using numpy, math, and sys.





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


ex2_simplecalc
Open Terminal
Open with Python
Open with IPython
Open with Jupyter Notebook


Home Page - Select or create
localhost:8889/tree
jupyter
Quit
Logout
Files
Running
Clusters
Select items to perform actions on them.
Upload
New
Notebook:
Python 3
Text File
Folder
Terminal
0
/
Adobe
Applications
ar26_rawindsondes
ar33_rawindsondes
a year ago
bathy
14 days ago

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.

```
In [ ]: # Exercise 2: ex2_simplecalc.py
        #
        # Program to demonstrate usage of Modules, Variables, Calculations,
        # Assignments, and Arrays
        #
        # Joseph B. Zambon
        #  jbzambon@ncsu.edu
        #  31 March 2021
```

```
In [1]: print('This is a "Hello World" message.')

        This is a "Hello World" message.
```

```
In [2]: helloworld = 'This is also a "Hello World" message, but first we assigned it as a \
        string variable called "helloworld"'
        print(helloworld)

        This is also a "Hello World" message, but first we assigned it as a string variable called "h
        elloworld"
```

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!

```
In [24]: helloworld = 'This is also a "Hello World" message, but first we assigned it as a
         string variable called "helloworld"'
         print(helloworld)

           File "<ipython-input-24-9580cf240e88>", line 1
             helloworld = 'This is also a "Hello World" message, but first we assigned it as a
                                                                                              ^
         SyntaxError: EOL while scanning string literal
```

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

```
In [4]: longline = 'This is a really, really long string variable but we\'re going to break it up\
        \nover 3 lines with a continuation character in the code and a newline character\
        \nin the variable.'
        print(longline)

        This is a really, really long string variable but we're going to break it up
        over 3 lines with a continuation character in the code and a newline character
        in the variable.
```

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).

```
In [5]: longerline = 'But hey, what if I want to actually pring any of these special characters?\
        \nEasy!  Just escape it with a backslash (\\) like this... \\n'
        print(longerline)
```

```
But hey, what if I want to actually pring any of these special characters?
Easy!  Just escape it with a backslash (\) like this... \n
```

Escape Sequence	Meaning
newline	Ignored
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
ooo	ASCII character with octal value *ooo*
\x*hh*...	ASCII character with hex value *hh*...

(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.

```
In [7]: helloworld = 'This is also a "Hello World" message, but first we assigned it as a \
                     string variable called "helloworld"'
        print(helloworld)
```

```
This is also a "Hello World" message, but first we assigned it as a              string vari
able called "helloworld"
```

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.

```
In [9]: concatenated_string = 'This is another really long string but I\'m going to format it\n' + \
                              'such that my variables are easily readable by someone scanning\n' + \
                              'through my code.'
        print(concatenated_string)
```

```
This is another really long string but I'm going to format it
such that my variables are easily readable by someone scanning
through my code.
```

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

```
In [30]: type(concatenated_string)
Out[30]: str
```

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!

Array types and conversions between types

NumPy supports a much greater variety of numerical types than Python does. This section shows which are available, and how to modify an array's data-type.

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
intc	Identical to C `int` (normally `int32` or `int64`)
intp	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`.
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

Additionally to `intc` the platform dependent C integer types `short`, `long`, `longlong` and their unsigned versions are defined.

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.

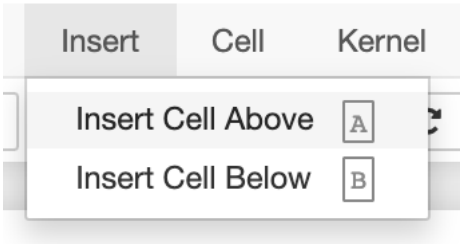
To add a cell below your header (first) cell, just click to the left of the cell window (below "In [ ]:"). The entire cell is now highlighted in blue, meaning you're editing the cell in relation to the rest of the program (rather than what is inside of the cell itself). This allows you to move the entire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

```
In [ ]: # Exercise 2: ex2_simplecalc.py
        #
        # Program to demonstrate usage of Modules, Variables, Calculations,
        # Assignments, and Arrays
        #
        # Joseph B. Zambon
        #  jbzambon@ncsu.edu
        #  31 March 2021
```

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

```
In [ ]: # Exercise 2: ex2_simplecalc.py
        #
        # Program to demonstrate usage of Modules, Variables, Calculations,
        # Assignments, and Arrays
        #
        # Joseph B. Zambon
        #  jbzambon@ncsu.edu
        #  31 March 2021
```

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



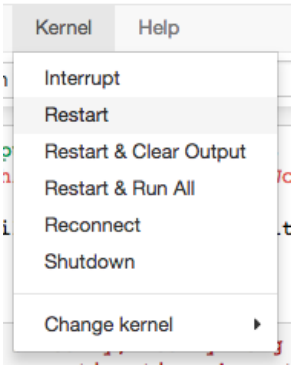
There is also a keyboard shortcut for Above (a) and Below (b). Just make sure you're editing outside of the cell (blue highlight) otherwise you will just be adding a's and b's to your code within the cell.

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).

In [ ]:
```
# Exercise 2: ex2_simplecalc.py
#
# Program to demonstrate usage of Modules, Variables, Calculations,
# Assignments, and Arrays
#
# Joseph B. Zambon
#  jbzambon@ncsu.edu
#  31 March 2021
```

In [ ]:
```
# Included modules
import numpy  # conda install numpy
              # Imported to express various data types
```

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, 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.

In [ ]:
```
# Included modules
import numpy  # conda install numpy
              # Imported to express various data types
import math   # installed typically by default
              # Imported to play with pi to double precision
```

```
In [35]: print(math.pi)

         3.141592653589793
```

```
In [49]: precise_pi = math.pi
         print(precise_pi)
         int8_pi = numpy.int8(precise_pi)
         print(int8_pi)

         3.141592653589793
         3
```

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.

```
In [12]: # Included modules
         import numpy  # conda install numpy
                       # Imported to express various data types
         import math   # Installed typically by default
                       # Imported to play with pi to double precision
         import sys    # Installed typically by default
                       # Imported to demonstrate memory usage
```

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.

In [120]:
```
# Various precisions of pi and memory usage
#Signed integers (positive or negative)
print('Signed integers')
int8_pi = numpy.int8(math.pi * 10**1) #(-128 to 127)
print('int8_pi   consumes ' + str(sys.getsizeof(int8_pi)) + ' bytes ' + \
      'and represents pi as: ' + str(int8_pi / 10**1))
int16_pi = numpy.int16(math.pi * 10**4) #(-32768 to 32767)
print('int16_pi  consumes ' + str(sys.getsizeof(int16_pi)) + ' bytes ' + \
      'and represents pi as: ' + str(int16_pi / 10**4))
int32_pi = numpy.int32(math.pi * 10**8) #(-2147483648 to 2147483647)
print('int32_pi  consumes ' + str(sys.getsizeof(int32_pi)) + ' bytes ' + \
      'and represents pi as: ' + str(int32_pi/10**8))
int64_pi = numpy.int64(math.pi * 10**16) #(-9223372036854775808 to 9223372036854775807)
print('int64_pi  consumes ' + str(sys.getsizeof(int64_pi)) + ' bytes ' + \
      'and represents pi as: ' + str(int64_pi / 10**16))

#Unsigned integers (positive-only)
print('\nUnsigned integers')
uint8_pi = numpy.uint8(math.pi * 10**1) #(0 to 255)
print('uint8_pi  consumes ' + str(sys.getsizeof(uint8_pi)) + ' bytes ' + \
      'and represents pi as: ' + str(uint8_pi / 10**1))
uint16_pi = numpy.uint16(math.pi * 10**4) #(0 to 65535)
print('uint16_pi consumes ' + str(sys.getsizeof(uint16_pi)) + ' bytes ' + \
      'and represents pi as: ' + str(uint16_pi / 10**4))
uint32_pi = numpy.uint32(math.pi * 10**9) #(0 to 4294967295)
print('uint32_pi consumes ' + str(sys.getsizeof(uint32_pi)) + ' bytes ' + \
      'and represents pi as: ' + str(uint32_pi / 10**9))
uint64_pi = numpy.uint64(math.pi * 10**16) #(0 to 18446744073709551615)
print('uint64_pi consumes ' + str(sys.getsizeof(uint64_pi)) + ' bytes ' + \
      'and represents pi as: ' + str(uint64_pi / 10**16))
```

```
Signed integers
int8_pi   consumes 25 bytes and represents pi as: 3.1
int16_pi  consumes 26 bytes and represents pi as: 3.1415
int32_pi  consumes 28 bytes and represents pi as: 3.14159265
int64_pi  consumes 32 bytes and represents pi as: 3.141592653589793

Unsigned integers
uint8_pi  consumes 25 bytes and represents pi as: 3.1
uint16_pi consumes 26 bytes and represents pi as: 3.1415
uint32_pi consumes 28 bytes and represents pi as: 3.141592653
uint64_pi consumes 32 bytes and represents pi as: 3.141592653589793
```

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?

In [130]:
```
# Extend beyond allowable range of int8

int8_extended = numpy.int8(math.pi * 10**2) #(-128 to 127)
print(int8_extended)
print('int8_pi   consumes ' + str(sys.getsizeof(int8_extended)) + ' bytes ' + \
      'and represents pi as: ' + str(int8_extended / 10**2))
```

```
58
int8_pi   consumes 25 bytes and represents pi as: 0.58
```

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!

```
In [131]: # Extend beyond allowable range of int8

          int8_extended = numpy.int8(math.pi * 10**2) #(-128 to 127)
          print(int8_extended + 256)
          print('int8_pi   consumes ' + str(sys.getsizeof(int8_extended)) + ' bytes ' + \
                'and represents pi as: ' + str((int8_extended + 256) / 10**2))

          314
          int8_pi   consumes 25 bytes and represents pi as: 3.14
```

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.

```
In [133]: #Floats (decimal precision)
          print('\nUnsigned integers')
          float16_pi = numpy.float16(math.pi) #(0 to 255)
          print('float16_pi consumes ' + str(sys.getsizeof(float16_pi)) + ' bytes ' + \
                'and represents pi as: ' + str(float16_pi))
          float32_pi = numpy.float32(math.pi) #(0 to 65535)
          print('float32_pi consumes ' + str(sys.getsizeof(float32_pi)) + ' bytes ' + \
                'and represents pi as: ' + str(float32_pi))
          float64_pi = numpy.float64(math.pi) #(0 to 4294967295)
          print('float64_pi consumes ' + str(sys.getsizeof(float64_pi)) + ' bytes ' + \
                'and represents pi as: ' + str(float64_pi))

          Unsigned integers
          float16_pi consumes 26 bytes and represents pi as: 3.14
          float32_pi consumes 28 bytes and represents pi as: 3.1415927
          float64_pi consumes 32 bytes and represents pi as: 3.141592653589793
```

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.

In [144]:
```
# Efficiently calculate the circumference of the Earth using
#  different precisions of pi

earth_radii = numpy.array([6353,6384])  #The earth is an oblate-spheroid, its
                                        #radius actually increases from pole to
                                        #equator.  Define an array with the
                                        #minimum (pole) and maximum (equator) radii
print('The range of radii for the oblate-spheroid Earth is between:\n' + \
      '                                                             ' + \
      str(earth_radii[0]) + 'km and ' + str(earth_radii[1]) + 'km')
earth_radius = numpy.mean(earth_radii)  #Simply use the mean of the radii
print('The calculated mean radius of the earth is:\n' + \
      '                                                             ' + \
      str(earth_radius) + 'km')
#Floats (decimal precision)
float16_pi = numpy.float16(math.pi) #(0 to 255)
float16_circ = 2 * float16_pi * earth_radius
print('Using ' + str(float16_pi) + ', the calculated mean circumference of the earth is:\n' + \
      '                                                             ' + \
      str(float16_circ) + 'km')
float32_pi = numpy.float32(math.pi) #(0 to 65535)
float32_circ = 2 * float32_pi * earth_radius
print('Using ' + str(float32_pi) + ', the calculated mean circumference of the earth is:\n' + \
      '                                                             ' + \
      str(float32_circ) + 'km')
float64_pi = numpy.float64(math.pi) #(0 to 4294967295)
float64_circ = 2 * float64_pi * earth_radius
print('Using ' + str(float64_pi) + ', the calculated mean circumference of the earth is:\n' + \
      '                                                             ' + \
      str(float64_circ) + 'km')
```

```
The range of radii for the oblate-spheroid Earth is between:
                                                             6353km and 6384km
The calculated mean radius of the earth is:
                                                             6368.5km
Using 3.14, the calculated mean circumference of the earth is:
                                                             40002.140625km
Using 3.1415927, the calculated mean circumference of the earth is:
                                                             40014.466742277145km
Using 3.141592653589793, the calculated mean circumference of the earth is:
                                                             40014.46562877319km
```

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.)

In [148]:
```
print('The difference between float16 and float32 is: ' + str(abs(float16_circ-float32_circ)))
print('The difference between float16 and float64 is: ' + str(abs(float16_circ-float64_circ)))
print('The difference between float32 and float64 is: ' + str(abs(float32_circ-float64_circ)))
```

```
The difference between float16 and float32 is: 12.326117277145386
The difference between float16 and float64 is: 12.325003773192293
The difference between float32 and float64 is: 0.0011135039530927315
```

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.

In [ ]:
```
# Included modules (shortened version)
import numpy as np          # conda install numpy
                            # Imported to express various data types
import math                 # Installed typically by default
                            # Imported to play with pi to double precision
from sys import getsizeof   # Installed typically by default
                            # Imported to demonstrate memory usage
```

Here's how some of the revised code looks.

In [27]:
```
#Floats (decimal precision)
print('Floats')
float16_pi = np.float16(math.pi) #(0 to 255)
print('float16_pi consumes ' + str(getsizeof(float16_pi)) + ' bytes ' + \
      'and represents pi as: ' + str(float16_pi))
float32_pi = np.float32(math.pi) #(0 to 65535)
print('float32_pi consumes ' + str(getsizeof(float32_pi)) + ' bytes ' + \
      'and represents pi as: ' + str(float32_pi))
float64_pi = np.float64(math.pi) #(0 to 4294967295)
print('float64_pi consumes ' + str(getsizeof(float64_pi)) + ' bytes ' + \
      'and represents pi as: ' + str(float64_pi))
```

```
Floats
float16_pi consumes 26 bytes and represents pi as: 3.14
float32_pi consumes 28 bytes and represents pi as: 3.1415927
float64_pi consumes 32 bytes and represents pi as: 3.141592653589793
```

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!