https://i.ebayimg.com/images/g/vGAAAOSweW5VbLXE/s-l1600.jpg

Physics 398DLP

Design Like a Physicist

Spring 2019

George Gollin

University of Illinois at Urbana-Champaign

C++ and Python Primer/Refresher

C++ and Python Primer/Refresher

Introduction: C++ and Python

I assume you already know how to program. If you've learned to code in python or C/C++, or Java, or some other language, you'll be fine. A B- or better in CS 101, CS 125, or Physics 298owl is a suitable prerequisite. It's also fine if you've learned on your own. But if you've never programmed before, or did poorly in an intro CS course, you should delay enrollment in Physics 398DLP until after you've done some coding.

We'll be working with two different languages: a slightly stripped-down version of C++ for programs to be executed by your Arduino and Anaconda's Python for developing the software to analyze your data.

There are minor differences in the two languages that you'll need to keep in mind. If you've worked with Java, you are already familiar with a language that requires you to declare the type of each variable you plan to use. C++ also requires this, but Python does not. Another difference is the use of a semicolon to end a line in C++ but not in Python. In addition, C++ uses curly brackets to define structure—which blocks of code are inside which other blocks of code—while Python uses indentation for this.

Install the Arduino programming IDE

Go to the Arduino website https://www.arduino.cc/ and navigate to https://www.arduino.cc/en/Guide/ArduinoMega2560. Download and install the Arduino Desktop IDE (Integrated Development Environment) on your laptop.

Connect an Arduino to a USB port in your laptop. The Arduino probably comes with a blink-an-LED program preloaded, so a yellow LED near the USB connector might start blinking as soon as the board is powered.

You'll need to go to the Tools → Port menu to select which communication channel your laptop will use to talk to the Arduino. While you're at it, open a serial monitor window by following Tools → Serial Monitor. The Arduino will write information to this window as instructed by the program it is running.

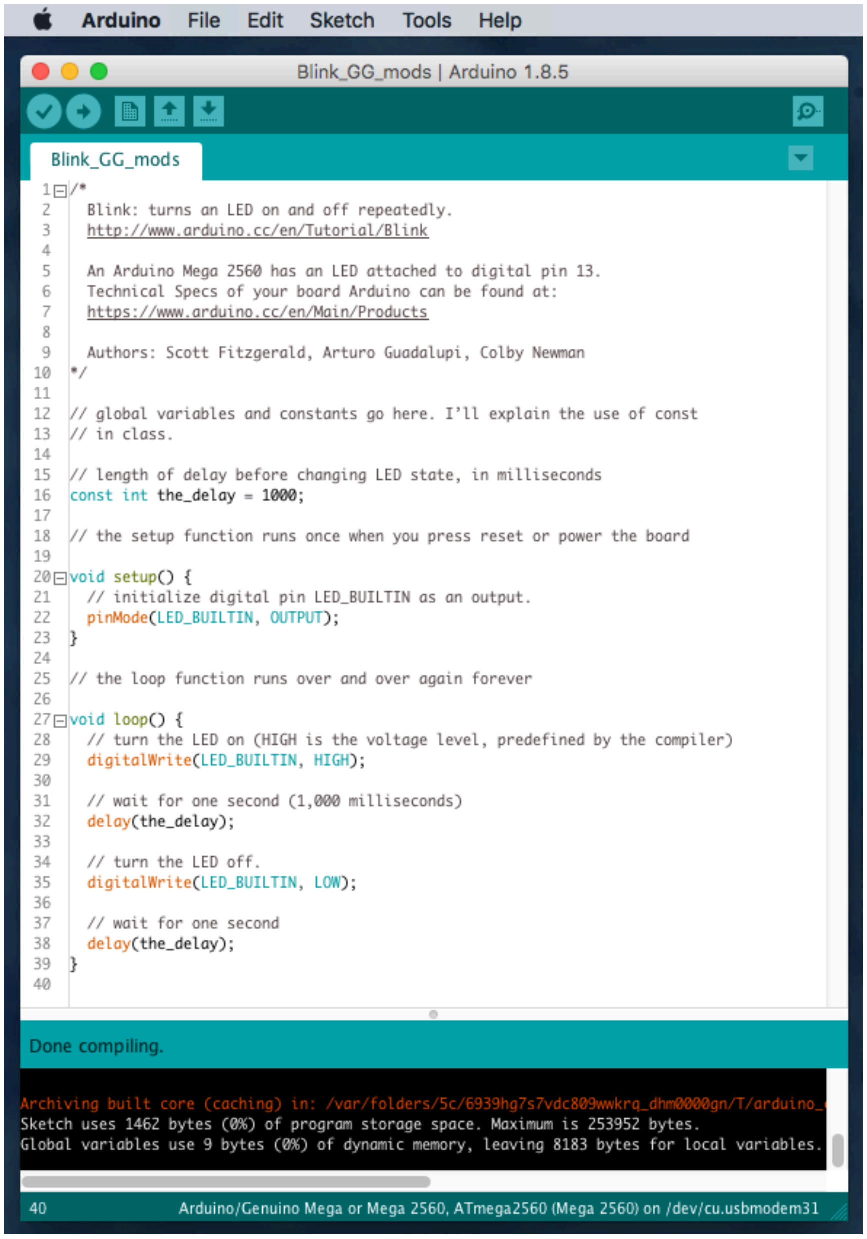
Please create a folder in which you will store your various Arduino programs. (For some reason people call an Arduino program a sketch. I think that sounds silly.)

You can find Arduino tutorials and sample programs at
https://www.arduino.cc/en/Tutorial/BuiltInExamples.

See also File → Examples → 01.Basics → Blink for a ready-to-run program, which (after a few modifications) looks like this:

```
/*
  Blink: turns an LED on and off repeatedly.
  http://www.arduino.cc/en/Tutorial/Blink

  An Arduino Mega 2560 has an LED attached to digital pin 13.
  Technical Specs of your board Arduino can be found at:
  https://www.arduino.cc/en/Main/Products

  Authors: Scott Fitzgerald, Arturo Guadalupi, Colby Newman
*/

// global variables and constants go here. I'll explain the use of const
// in class.

// length of delay before changing LED state, in milliseconds
const int the_delay = 1000;

// the setup function runs once when you press reset or power the board

void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever

void loop() {
  // turn the LED on (HIGH is the voltage level, predefined by the compiler)
  digitalWrite(LED_BUILTIN, HIGH);

  // wait for one second (1,000 milliseconds)
  delay(the_delay);

  // turn the LED off.
  digitalWrite(LED_BUILTIN, LOW);

  // wait for one second
  delay(the_delay);
}
```

Here's how it looks in the IDE's editor:


Arduino File Edit Sketch Tools Help
Blink_GG_mods | Arduino 1.8.5
Blink_GG_mods

```
/*
  Blink: turns an LED on and off repeatedly.
  http://www.arduino.cc/en/Tutorial/Blink

  An Arduino Mega 2560 has an LED attached to digital pin 13.
  Technical Specs of your board Arduino can be found at:
  https://www.arduino.cc/en/Main/Products

  Authors: Scott Fitzgerald, Arturo Guadalupi, Colby Newman
*/

// global variables and constants go here. I'll explain the use of const
// in class.

// length of delay before changing LED state, in milliseconds
const int the_delay = 1000;

// the setup function runs once when you press reset or power the board

void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever

void loop() {
  // turn the LED on (HIGH is the voltage level, predefined by the compiler)
  digitalWrite(LED_BUILTIN, HIGH);

  // wait for one second (1,000 milliseconds)
  delay(the_delay);

  // turn the LED off.
  digitalWrite(LED_BUILTIN, LOW);

  // wait for one second
  delay(the_delay);
}

```

Done compiling.

```
Archiving built core (caching) in: /var/folders/5c/6939hg7s7vdc809wwkrq_dhm0000gn/T/arduino_
Sketch uses 1462 bytes (0%) of program storage space. Maximum is 253952 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 8183 bytes for local variables.
```

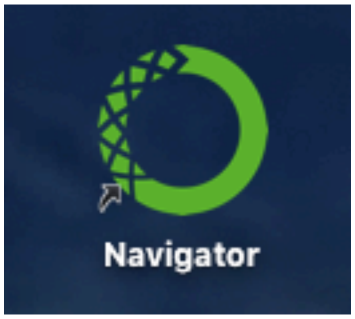
40 Arduino/Genuino Mega or Mega 2560, ATmega2560 (Mega 2560) on /dev/cu.usbmodem31

Click on the check mark to compile the program; click on the right arrow button to compile it and download the executable to the Arduino.

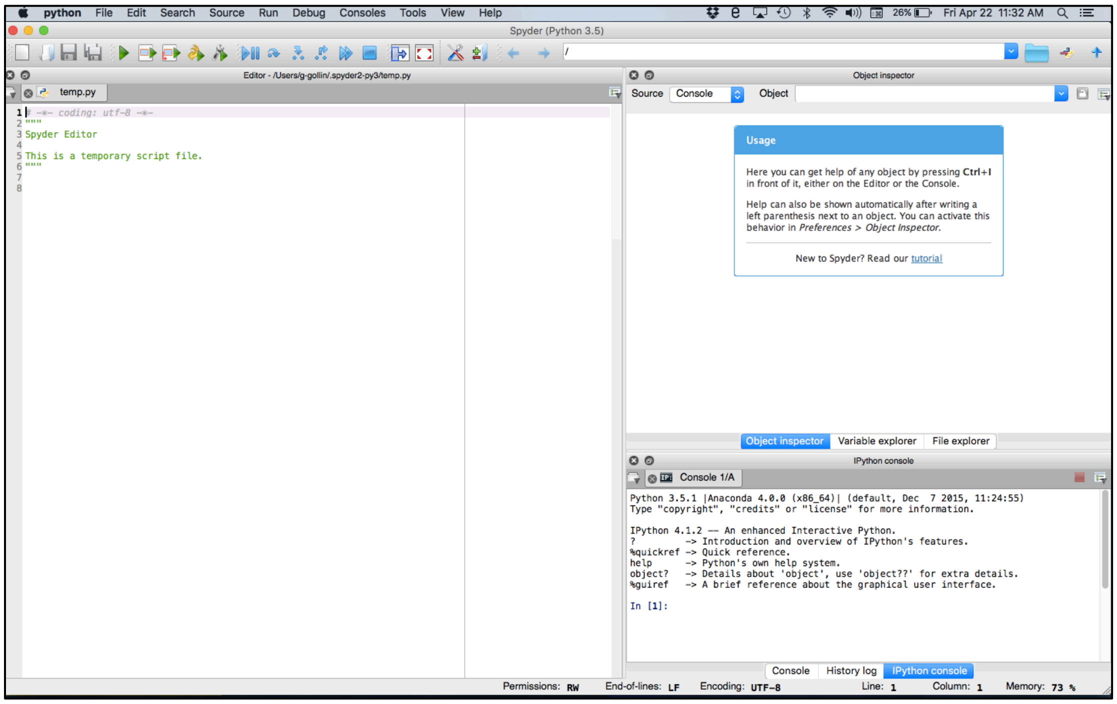
Please open the Blink example, then save it to the folder you've created to hold your programs. Compile and download it to confirm that it works. We'll come back to this later.

Install and configure Anaconda Python

Please download the installation file for the most recent version of Anaconda Python, available here: https://www.anaconda.com/download/. On a Mac the installer will create an icon/shortcut to "Anaconda Navigator" that will allow you to launch applications. On a Windows machine you might need to access Navigator here: Start → All Programs → Anaconda3 (64-bit) → Anaconda Navigator.



The Anaconda software contains a number of different programs. We will be working with spyder, the "Scientific Python Development Environment." This is an integrated development environment (IDE), which incudes an editor, a control console, a debugger, a table of program variables, and other tools.
Click on the spyder launch button in the navigator window. The development environment workspace will open.


python File Edit Search Source Run Debug Consoles Tools View Help
26% Fri Apr 22 11:32 AM
Spyder (Python 3.5)
Editor - /Users/g-gollin/.spyder2-py3/temp.py
temp.py
Object inspector
Source Console Object
Object inspector Variable explorer File explorer
IPython console
Console 1/A
Console History log IPython console
Permissions: RW End-of-lines: LF Encoding: UTF-8 Line: 1 Column: 1 Memory: 73 %

```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""
```

Usage

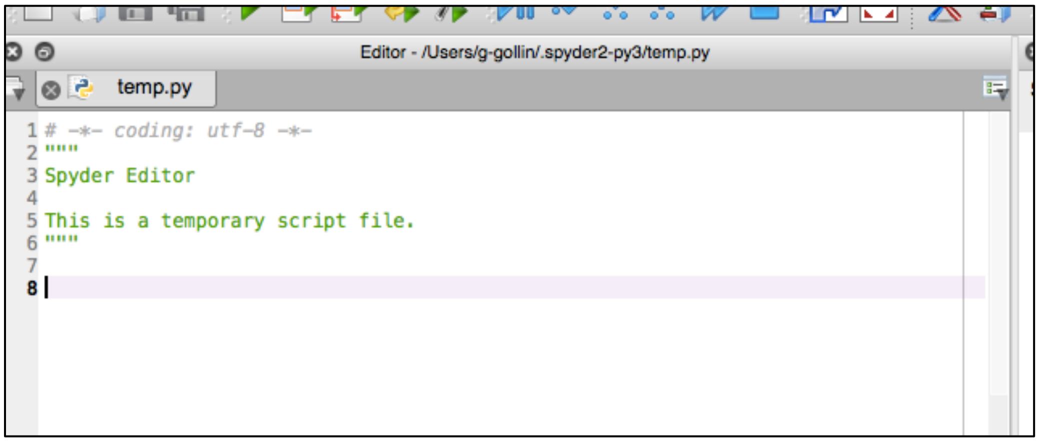
Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Object Inspector*.

New to Spyder? Read our tutorial

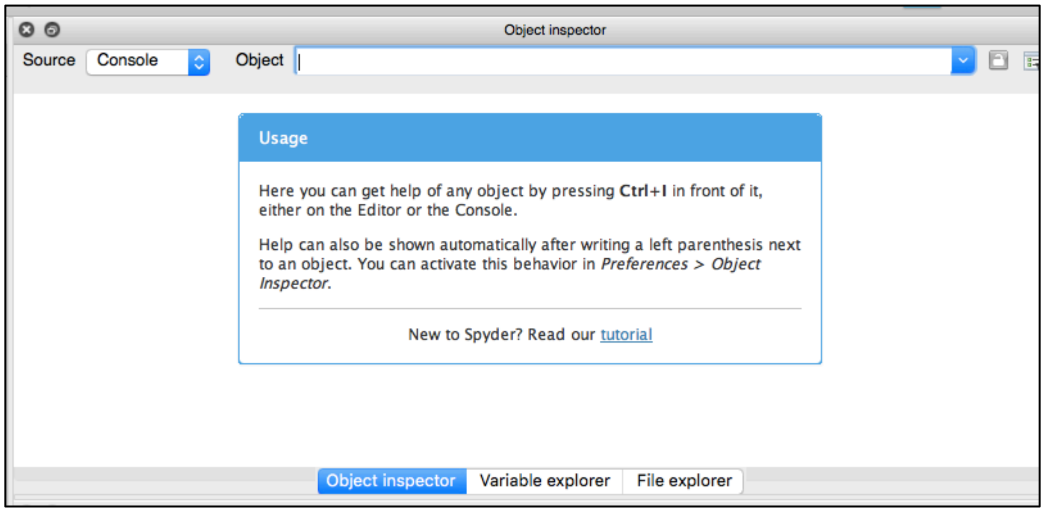
```
Python 3.5.1 |Anaconda 4.0.0 (x86_64)| (default, Dec  7 2015, 11:24:55)
Type "copyright", "credits" or "license" for more information.

IPython 4.1.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%guiref   -> A brief reference about the graphical user interface.

In [1]:
```

The window on the left is an editor, which you will use to create script files (program files containing executable instructions. The paired triple quotes enclose comments). Here's a screen shot of part of it.



```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""

```

The upper right window allows you to look at the contents of "objects," variables, and file directories. Note the tabs at the bottom of the window for selecting what is shown. You will probably find the file and variable explorer tabs most useful. Sometimes the file modification dates shown by file explorer do not update when I save changes to a file! That is surely a bug.



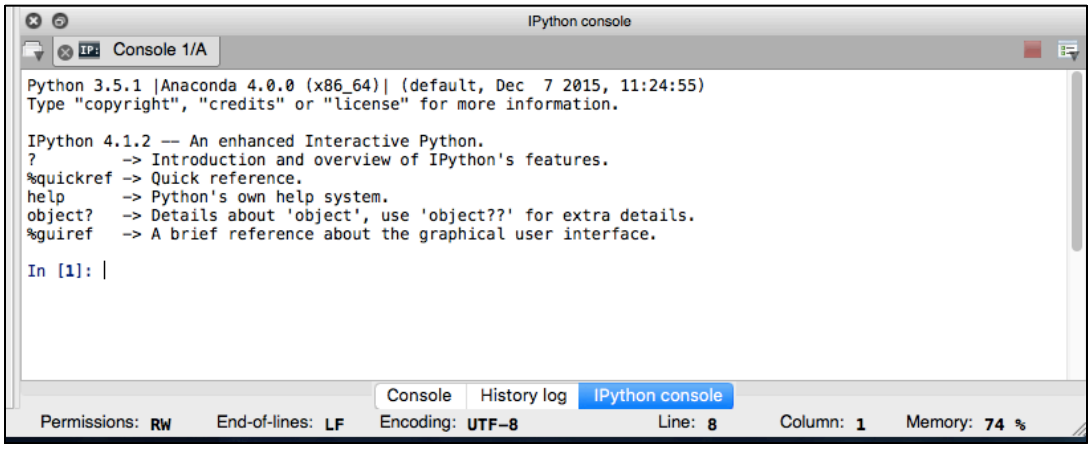
Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Object Inspector.*

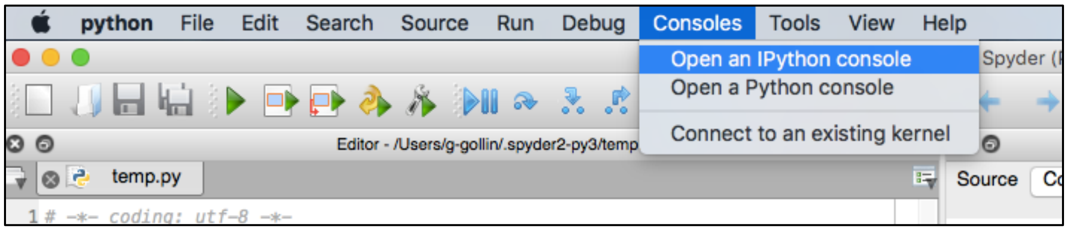
New to Spyder? Read our tutorial

The lower right window shows an iPython console, a sort of operator's station from which you can issue commands to Python. It also displays program output.



```
Python 3.5.1 |Anaconda 4.0.0 (x86_64)| (default, Dec  7 2015, 11:24:55)
Type "copyright", "credits" or "license" for more information.

IPython 4.1.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%guiref   -> A brief reference about the graphical user interface.

In [1]: |
```

If you trash the iPython console by mistake you can open a new one through the "Consoles" menu at the top of the workspace window.



There are a few parameters that you should set. Go to the Python preferences menu and do this:
preferences : run : default working directory
set to a sensibly-named folder that will hold all your scripts

preferences : current working directory : console directory
set to the same folder as that which will hold your scripts

preferences : iPython console : graphics : Backend
set to "Automatic"

preferences : History log : Settings
set "History depth" to 2000 entries

Quit spyder, then restart.

Representing structure in C++ and Python

The two languages use different conventions for defining when a block of code lives inside another block of code. C++ uses curly braces and semicolons ("{", "}", and ";") while Python uses indentation. The following three code snippets (the first two are C++ for the Arduino, the third is Python) are functionally equivalent.

```
// C++
bool debug_it = true;
Serial.begin(9600);
if (debug_it) {
Serial.print("Debug flag is set ");
Serial.println("for some reason.");
}
Serial.print("This line always prints. ");

// C++
bool debug_it = true;
Serial.begin(9600);
if (debug_it) {
       Serial.print("Debug flag is set ");
       Serial.println("for some reason.");
       }
Serial.print("This line always prints. ");

# Python
debug_it = true
if (debug_it):
       print("Debug flag is set ", "for some reason.")
print("This line always prints.")
```

This snippet isn't, however:

```
# Python
debug_it = true
if (debug_it):
       print("Debug flag is set ", "for some reason.")
       print("This line only prints when debug_it is true."
```

Comments in C++ and Python

C++ single-line comments begin with a pair of forward slashes, as shown above. You can also define a C++ multiline comment this way:

```
/*
This is a multi-
line comment
in C++.
*/
```

In Python single line comments begin with an octothorpe ("#") as shown above. Multiline comments are delimited with starting and ending triples of double quotes:

```
"""
This is a multi-
line comment
in Python.
"""
```

Continuing a statement onto the next line in C++ and Python

For improved readability, you'll want to break overly long lines of code, continuing them onto the next line. This is especially easy in C++ since the compiler doesn't consider a statement to end until the compiler encounters a semicolon. In Python you'll need to use a backslash to break the line. For example:

```
// C++ continuation example
Serial.print(
"They call me Bond. James Bond. "
);

# Python continuation example
print( \
"They call me Bond. James Bond. " \
)
```

Scope of variables

This is the stuff of headaches. Some variables are global, and knowable by all functions in a code file. Others are local, and known only inside the function in which they are defined.

Here's a Python example, in which the variable "text" is used as a local variable in functions f1 and f2, and a global variable in f3, as well as the main program.

```
# define "text" here, which makes it a global variable.
text = "text is a global variable, defined as this string."

#define functions f1, f2, and f3... "\n" is the newline character.
def f1():
     print("now printing 'text' from inside f1.")
     text = "This is 'text' as defined locally inside function f1."
     print(text, "\n")
def f2():
     print("now printing 'text' from inside f2.")
     text = "This is 'text' as defined locally inside function f2."
     print(text, "\n")
def f3():
     print("now printing 'text' (which is a global variable) from inside f3.")
     print(text, "\n")

print("about to call f1.")
f1()
print("about to call f2.")
f2()
print("about to call f3.")
f3()
print("now print 'text' from top level program.")
print(text)
```

The program's output follows:

```
about to call f1.
now printing 'text' from inside f1.
This is 'text' as defined locally inside function f1.

about to call f2.
now printing 'text' from inside f2.
This is 'text' as defined locally inside function f2.

about to call f3.
now printing 'text' (which is a global variable) from inside f3.
text is a global variable, defined as this string.

now print 'text' from top level program.
text is a global variable, defined as this string.
```

Scoping works similarly in C++. When you encounter mysterious behavior in your code, consider looking at it with an eye towards a problem with the scope of a variable.

Some Pythonic surprises

Sometimes Python will surprise you. Here are a few of the things that I have tripped over.

Exponentiation

Python's exponentiation operator is **. You might be tempted to use ^ for exponentiation, but that's not correct: the symbol "^" performs a bit-by-bit exclusive-OR between the two variables.

An example:

```
In [13]: 3**3
Out[13]: 27
In [14]: 3^3
Out[14]: 0
```

Indentation

Watch out for this! The following code

```
sum = 0
for i in range(1,100000):
     sum += i
print(sum)
```

will print a single line of output, while

```
sum = 0
for i in range(1,100000):
     sum += i
     print(sum)
```

will print 100,000 lines.

Array assignment

In most languages with which I am familiar, assigning a new variable to equal an existing variable creates a copy of the original variable in a different storage location. For example, in C++ the code snippet

```
char a[ ] = {'u', 'v', 'w'}; char b[3];
// copy a into b using the memcpy function (b = a isn't allowed for C++ arrays)
memcpy(b, a, 3);
Serial.println(b[0]);
a[0] = 'x';
Serial.println(a[0]);
Serial.println(b[0]);
```

produces the following output.

```
u
x
u
```

Note that modifying a[0] does not affect b[0].

It is different in Python, in which assigning a "new" array just provides an alternate name for the same locations in memory. The following Python code…

```
import numpy as np
a = np.array([10, 20, 30])
print("a = ", a)
b = a
b[0] = 5
print("a = ", a)
```

…yields the following output.

```
a =  [10 20 30]
a =  [ 5 20 30]
```

In Python, changing b also changes a.

Libraries

In Python you load libraries of useful stuff using import commands, generally placed near the top of your program file. For example:

```
import numpy as np
…
a =  np.array([10, 20, 30])
```

In C++ you load libraries with the include compiler directive:

```
#include <Adafruit_MCP4725.h>
…
// instantiate a DAC object named "dac":
Adafruit_MCP4725 dac;
```

You can manage your Arduino C++ libraries through the IDE (Integrated Development Environment) menu path Sketch → Include Library.

The order in which things appear in your C++ and Python programs can differ

It would be handy if Python were able to read through your code in a first pass, picking up the identities of the various functions you define, before beginning to execute your code. That way, you'd be able to put your main program near the top of the file, and append new functions as you write them at the end of your file. But Python is an interpreted, not a compiled language, and the Python interpreter doesn't jump around in your file to figure out what is where. In the scoping example we just discussed, putting the functions f1, f2, and f3 at the end of the file will throw all sorts of annoying error messages.

C++ is a compiled language: the compiler produces a machine language executable program, rather than reading (and parsing) a “script” of instructions. The compiler is able to identify the components of your program, so it would be perfectly fine to place the functions f1, f2, and f3 at the end of your file.

Structure of an Arduino program

The Arduino IDE's compiler expects to find functions named "setup" and "loop" somewhere in your program file. Setup is executed only once, immediately after the program starts running. After exiting setup, the compiler will execute loop; each time the program leaves loop it will immediately reenter the routine.

I suggest you organize the content of a program file as follows:

1. a block of explanatory comments
2. include directives for libraries
3. definitions of global variables and instantiations of objects representing hardware devices
4. setup function
5. loop function
6. other functions

Here's an example that will blink the yellow LED on the Arduino circuit board that is connected to the Arduino's pin 13. (You can find the code on the course's "Code & design repository" web page.) You'll want to open a 9600 baud serial monitor window after connecting to the Arduino: follow the menu path Tools → Port, and then Tools → Serial Monitor.

```
/*
  Blink the yellow LED that is driven by an opamp attached to the
  Arduino Mega 2560's pin 13.

  George Gollin, University of Illinois, January 7, 2019.
*/

//////////////////////// includes /////////////////////////////

// include a library for one device you'll eventually be using, namely
// an INA219 current/voltage monitor breakout board. I don't actually
// do anything with it in this program.

#include <Adafruit_INA219.h>

//////////////////////// instantiations /////////////////////////////

// instantiate a current sensor object named "ina219":
Adafruit_INA219 ina219;
```

```
///////////////////////// globals //////////////////////////////

// approximate on and off time, neglecting time to enter/exit loop
// I do not expect to change these, so declare them as constants.
const int on_milliseconds = 250;
const int off_milliseconds = 750;

// pin number for the LED
const int LED_pin = 13;

// flashes so far... note that this is a two-byte signed integer and
// will get weird after 32,767.
int flashes_so_far;

///////////////////////// setup //////////////////////////////

// The setup function runs once when you press reset, or power the board.
// Since it doesn't return a value, declare it as type "void"

void setup() {

  // fire up the serial monitor, set to 9600 baud.
  Serial.begin(9600);

  // initialize digital pin LED_pin as an output.
  pinMode(LED_pin, OUTPUT);

  // Initialize the INA219 current/voltage monitor (You probably
  // don't have one of these yet.)
  ina219.begin();

  // initialize a (global) variable...
  flashes_so_far = 0;

  // print a message. println puts a return at the end of the line.
  Serial.println("All done with setup.");
}

///////////////////////// loop //////////////////////////////

// the loop function runs over and over again, forever

void loop() {

  // turn the LED off. "HIGH" and "LOW" are system-defined.
  digitalWrite(LED_pin, LOW);

  // now wait.
  delay(off_milliseconds);

  // turn the LED on.
  digitalWrite(LED_pin, HIGH);

  // now wait.
  delay(on_milliseconds);
  // increment a counter, just for fun.
  flashes_so_far++;

  // every fifth flash call a function. % is the modulus function.
  if (flashes_so_far % 5 == 0) {
    print_something(flashes_so_far);
  }
}

///////////////////////// print_something //////////////////////////////

// the print_something function just prints a line when called.

void print_something(int the_number) {

  Serial.print("Number of LED flashes so far: ");
  Serial.println(the_number);
}
```

Output to the serial monitor looks like this:

```
All done with setup.
Number of LED flashes so far: 5
Number of LED flashes so far: 10
Number of LED flashes so far: 15
Number of LED flashes so far: 20
Number of LED flashes so far: 25
```

etc.

A Python Refresher, from Physics 298owl

Our goals are to :

- Install Anaconda's spyder Python developer's environment on your laptop;
- Experiment with Python by typing commands directly into the iPython console;
- Learn about some of the basic tools in programing, including loops, conditional statements, and mathematical operations;
- Write and execute a program that sums (part of) an infinite series for π;
- Use the numpy and matplotlib libraries to generate graphical representations of arrays of points, and continuous functions of one and two variables;
- Graph a couple of peculiar functions, one of which (as you will learn during the next unit) relates to the spacetime curvature induced by general relativistic effects in the vicinity of a massive object.

Useful Python stuff

The following table is taken from my Physics 298owl material.

A table of useful Python stuff	
Python language format	**User-defined functions**
Begin comments with a sharp sign; Put individual statements on separate	def QuadraticFormula(a, b, c):
lines or separate them with semicolons. Use \ to continue to next line.	root1 = (-b + (b**2 - 4 * a * c) ** 0.5) / (2 * a)
Assignment statements and variable types	root2 = (-b - (b**2 - 4 * a * c) ** 0.5) / (2 * a)
a = 1; b = 1.2; pi1 = 0.031416e2; pi2 = 31415.9265e-4	return [root1, root2]
SqrtMinusOne = 1j # this one is complex	
MyName = "George"; a_list = ["cat", "wombat", 2.71828]	# now call the function.
Accessing characters in a string; accessing list elements	roots = QuadraticFormula(1, 2, -8)
LetterG = MyName[0]; marsupial = a_list[1]	print("roots are ", roots[0], roots[1])
print("LetterG = ", LetterG, " marsupial = ", marsupial)	**numpy numerical library**
Logical statements	import numpy as np # put this at the top of your script
ThisIsTrue = 5 > 3; ThisIsFalse = not ThisIsTrue	print(np.sqrt(2))
AlsoTrue = 5 >= 3; AlsoFalse = 5 <= 3	MyArray = np.array([2.0] * 5) # make a numerical array
SixEqualsSix = 6 == 6; SixNEFive = 6 != 5	SqrtAllElements = np.sqrt(MyArray) # act on all elements
AnotherTrue = ThisIsTrue or ThisIsFalse	dir(np) # see what functions are in the numpy library
AnotherFalse = ThisIsTrue and ThisIsFalse;	ThetaArray = np.linspace(0, 2 * np.pi, 360)
Arithmetic functions	ThetaArray2 = np.arange(0, 2 * np.pi, 1 / 360)
ThreeSquared = 3 ** 2 # exponentiation	SineArray = np.sin(ThetaArray) # take sines of all angles
RootTwo = 2 ** 0.5	UnfilledArray = np.empty(25)
NotRootTwo = 2 ** 1/2 # watch out!	ArrayOfZeroes = np.zeros(25) # make a zero-filled array
print("watch out: ", NotRootTwo, " is not 1.414...")	# generate x and y for EACH cell in a 10 x 10 grid
SeventeenModThree = 17 % 3 # modulus	x = np.linspace(0, 10, 10); y = np.linspace(0, 10, 10)
print("17 mod 3 is ", SeventeenModThree)	xgrid, ygrid = np.meshgrid(x, y)
if blocks (note the use of whitespace and colons)	print("size of x and xgrid: ", np.size(x), np.size(xgrid))
if 5 > 3:	**Graphics**
print("5 is greater than 3")	import numpy as np
	import matplotlib.pyplot as plt
if 5 < 3:	import matplotlib.pyplot as plt
print("we will never execute this statement")	xarray = np.cos(np.linspace(0, 2 * np.pi, 100))
else:	yarray = np.sin(np.linspace(0, 2 * np.pi, 100))
print("5 is not less than 3")	plt.plot(xarray, yarray)
if 6 > 6:	# here is a fancier plot. most commands are self-explanatory
print("we will never execute this statement")	fig = plt.figure() # create a new, blank figure
elif 6 == 6:	ax = fig.gca() # "gca" is get current axes
print("This confirms that 6 is equal to 6")	ax.set_aspect("equal")
elif 7 == 7:	ax.set_xlabel("x values")
print("though true, this won't execute either")	ax.set_ylabel("y vaues")
else:	ax.set_title("A unit circle with labeled axes")
print("none of the conditions were satisfied")	ax.plot(xarray, yarray)
Loops (note the use of whitespace and colons)	
for index in range(3, 6):	# do a 3D plot of a one-turn helix.
print("index = ", index)	from mpl_toolkits.mplot3d import Axes3D
	zarray = np.linspace(0, 3, 100) # zarray is same size as xarray.
ijk = 0	fig = plt.figure() # create a blank figure and get its axes
while not ijk > 2:	ax = fig.gca(projection='3d')
ijk += 1	ax.set_xlim(-1, 1)
print("ijk = ", ijk)	ax.set_ylim(-1, 1)
	ax.set_zlim(0, 3)
for m in range (-4, 1000000000000):	ax.set_xlabel("X")
print("m = ", m)	ax.set_ylabel("Y")
if m > 1:	ax.set_zlabel("Z")
print("now break out of loop")	ax.set_title("One-turn helix")
break	ax.plot(xarray, yarray, zarray)

You should add to this table as you come upon other useful bits of Python wisdom.

Basic concepts, mostly for Python

Take a look at the table of useful Python stuff on the inside front cover of the course packet. I am going to go through most of the information presented there, but quickly so we can being writing code.

Variables and assignment statements

A variable is a name assigned to one location in memory. You manipulate the contents of that memory location by referring to it by the name of the variable. For example, to ***associate*** the name "A" with a location in memory, then ***assign it*** the value 12, you would type the following into the iPython console window.

```
A=12
```

The computer does something analogous to the "copy a1, a2" machine instruction we discussed earlier, with a1 holding the address of a word in memory that contains the integer 12, and a2 holding the memory address that has been assigned to the variable A.

To define a new variable as the sum of **`A`** and the number 4 you would type:

```
B=A+4
```

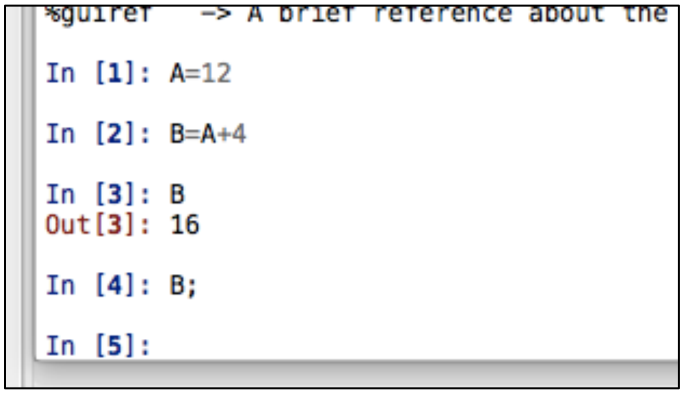
To inspect the value of **`B`** you would just type its name into the console:

```
B
```

Note that a semicolon at the end of a line suppresses the normal output produced in response to that line:

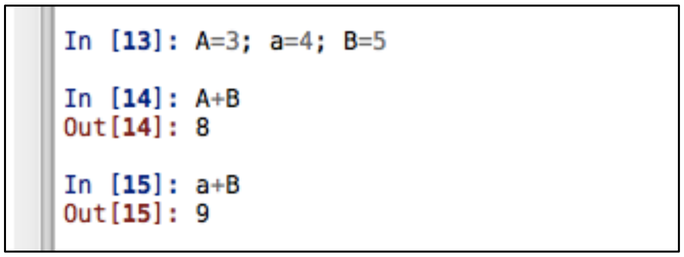
```
B;
```

yields no output. Here is a screen shot of the console with the above commands.



```
%guiref   -> A brief reference about the

In [1]: A=12

In [2]: B=A+4

In [3]: B
Out[3]: 16

In [4]: B;

In [5]:
```

You can place multiple assignments on a single line by separating them with semicolons. Note that variable names are case sensitive. Take a look:



```
In [13]: A=3; a=4; B=5

In [14]: A+B
Out[14]: 8

In [15]: a+B
Out[15]: 9
```

Keep in mind that an equal sign in Python is actually an assignment of value, and not the same thing as an equation expressing the equivalence of the left and right sides. For example, to increment the value of **`A`** by **`1`** we'd do this:

```
A=A+1
```

Kinds of variables

There are many different kinds of variables that are defined in Python. For example, the statement

```
A=12         # inline comments begin with an octothorpe
```

defines an *integer* variable. The statement

```
C=2.71828    # C is a floating point variable
```

defines a *floating point* variable, a numerical variable which is allowed to take on non-integer values. The statement

```
D=(1+2j)     # D is complex
```

defines a *complex* variable with the value 1 + 2i. ($i = \sqrt{-1}.$) Note the use of **`j`** instead of **`i`**. It is fine to mix together integer, floating point, and complex numbers in arithmetic statements:

```
In [1]: A = 12;
In [2]: B = 2.5;
In [3]: C = (5 + 7j);
In [4]: A + B + C
Out[4]: (19.5+7j)
```

The statement

```
MyName="George"     # a string!
```

defines a *string*. You may use single quotes if that is your preference. It is fine to enclose whitespace and single quotes inside double-quoted strings:

```
In [1]: AnotherString = "George's car"
In [2]: print(AnotherString)
George's car
```

A string is really a list of individual characters; you can access the *n*th character in a string this way (note that position 0 yields the first character):

```
In[11]: AnotherString[2]
Out[11]: 'o'
In[12]: AnotherString[0]
Out[12]: 'G'
```

Boolean (logical) variables can only take the values True and False.

```
In [1]: ObviouslyTrue = 3 > 2; print(ObviouslyTrue)
True
```

Python is able to convert most variables from one type to another as necessary.

Mathematical operations

Here are examples of some of the mathematical operations that Python supports. Many are self explanatory.

```
In [1]: a=8+9; print(a)          # addition, with two statements on one line!
17

In [2]: a=8/9; print(a)
0.8888888888888888

In [3]: A=3**2; print(A)         # ** means exponentiation. NB: ^ is NOT!!
9

In [4]: print(25**0.5)           # one way to take a square root
5.0

In [4]: print(pow(25,0.5))       # another way: "pow" is power
5.0
```

The % sign is used to determine the modulus of one number with respect to another. What I mean is this: the value of *a* % *b* is the remainder when *a* is divided by *b*. Some examples:

```
In[1]: 7 % 4
Out[1]: 3
In[2]: 14 % 7
Out[2]: 0
```

```
In[3]: 13 % 7
Out[3]: 6
```

You may need to import a *module* of routines that aren't already known to Python. Your Python installation includes lots of these, and Python knows how to find them if you use the import command. You will eventually find it convenient to define some of your own modules. (That's for later!) Here's how this works.

```
In [1]: print(sqrt(25))          # this won't work yet
NameError: name 'sqrt' is not defined
In [2]: import numpy as np
In [3]: print(np.sqrt(25))       # now it will work.
5.0
```

Keep in mind that your computer's internal workings use binary, not decimal, so sometimes there can be surprises. For example, the internal representation of 0.1 is inexact, as you can see in the following:

```
In[1]: 0.1 + 0.2
Out[1]: 0.30000000000000004
```

There are ways to improve the precision used by Python in its calculations, but the language isn't nearly as versatile as some others in its options for greater accuracy. For now, keep in mind that sometimes zero isn't quite zero:

```
In[1]: 0.3 - 0.1 - 0.2
Out[1]: -2.7755575615628914e-17
In[2]: abs(0.3 - 0.1 - 0.2) == 0
Out[2]: False
In[3]: abs(0.3 - 0.1 - 0.2) < 1.e-16
Out[3]: True
```

Here is something you can do to learn the level of precision offered by your computer's Python. (A "floating point" number is a real number with a decimal point. A "long double precision" number is a floating point number with a few extra digits of precision on Macs and some (but not all) windows machines. First import "numpy," a built-in numerical python module.

```
In[1]: import numpy as np  # import the numpy module, refer to it as "np"

In[2]: np.finfo(np.float)        # ask for information about floats
Out[2]: finfo(resolution=1e-15, min=-1.7976931348623157e+308,
max=1.7976931348623157e+308, dtype=float64)

In[3]: np.finfo(np.longdouble)    # ask for information about long doubles
Out[3]: finfo(resolution=1e-18, min=-1.18973149536e+4932,
max=1.18973149536e+4932, dtype=float128)
```

Logical operations

It is easy to perform logical test of the values of variables and constants. Note the use of the double equal sign.

```
In [1]: 1==2              # values are equal
Out[1]: False
In [2]: 2==2
Out[2]: True              # note that True and False begin with upper case
In [3]: 1<2               # first less than second
Out[3]: True
In [4]: 2<=2              # first less than or equal to second
Out[4]: True
In [5]: 1>=2              # first greater than or equal to second
Out[5]: False
In [6]: 6!=9              # first is not equal to second
Out[6]: True
In [7]: 6!=9 and 6==9     # logical AND
Out[7]: False
In [8]: 6!=9 or 6==9      # logical OR
Out[8]: True
```

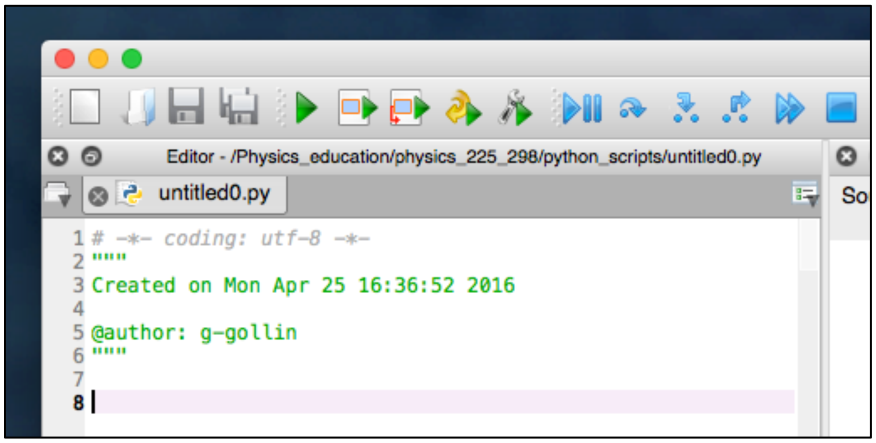
To execute a block of instructions only when a particular condition is true, indent the block of instructions following an "if" statement. Note that the if statement must end with a colon.

```
In[1]: LogicalValue = 4
In[2]: if LogicalValue < 5:
  ...:     print("LogicalValue is less than 5")
  ...:
LogicalValue is less than 5
```

It is very clumsy to execute if-blocks this way! A better way is to put a string of executable instructions into a script file, then execute the script.

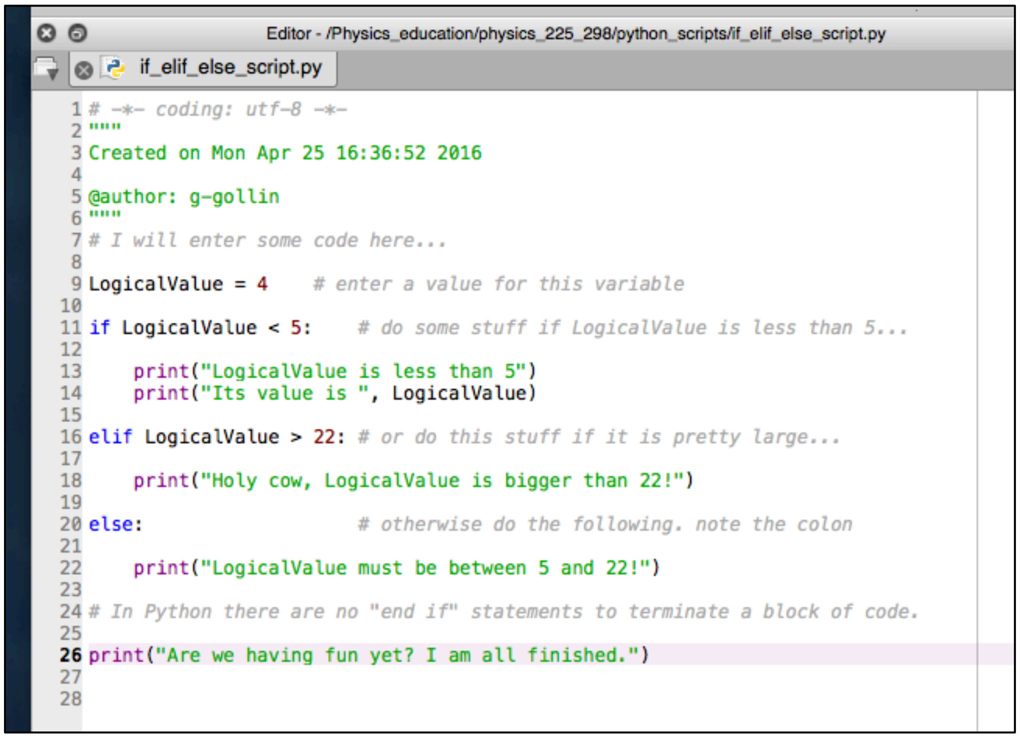
Scripts

To work with scripts, you will first need to tell spyder where to find them. Begin by creating a folder in which you will store your scripts. (I've named mine "python_scripts.") Go to the "Global working directory" window in spyder's preferences to set the startup directory.
The editor opens with an untitled default script that begins with a (three-quotation mark delimited) comment.
.


Editor - /Physics_education/physics_225_298/python_scripts/untitled0.py
untitled0.py
So

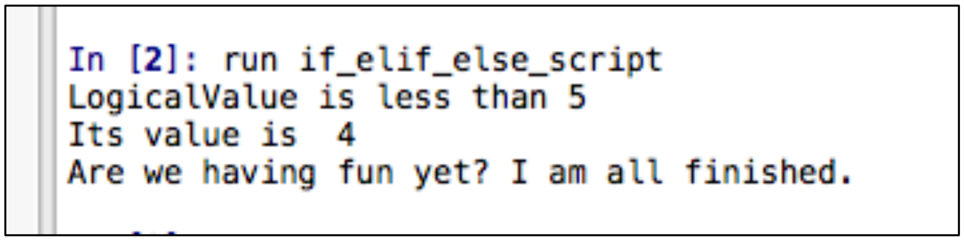
```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 25 16:36:52 2016

@author: g-gollin
"""

```

Enter some well-commented code into the editor window, then save the file. In the following screen shot I have an if-then-else-if block, followed by an example of running it from the console. The pattern of indentations is important: take careful note of it. This is how Python defines what code is inside an if block (or a loop) and what is outside. Also note the presence of the colon after the logical expression to be evaluated.


Editor - /Physics_education/physics_225_298/python_scripts/if_elif_else_script.py
if_elif_else_script.py

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 25 16:36:52 2016

@author: g-gollin
"""
# I will enter some code here...

LogicalValue = 4      # enter a value for this variable

if LogicalValue < 5:    # do some stuff if LogicalValue is less than 5...

    print("LogicalValue is less than 5")
    print("Its value is ", LogicalValue)

elif LogicalValue > 22: # or do this stuff if it is pretty large...

    print("Holy cow, LogicalValue is bigger than 22!")

else:                   # otherwise do the following. note the colon

    print("LogicalValue must be between 5 and 22!")

# In Python there are no "end if" statements to terminate a block of code.

print("Are we having fun yet? I am all finished.")


```

Run the program from the IPython console by typing “run” followed by the file name (leave off the “.py” filename extension.). It is possible that you will first need to tell the console to load the file: do this by typing “import” then the filename, omitting the .py extension. It is unclear to me when you actually need to do this!



```
In [2]: run if_elif_else_script
LogicalValue is less than 5
Its value is  4
Are we having fun yet? I am all finished.
```

Lists and arrays

Lists and arrays are rather like subscripted variables: a_0, a_1, a_2, … But there is a fundamental difference between the two: Python, before the import of a library like numpy, only knows about lists. A list can comprise elements of different types; if you try to "add" two lists you'll produce a concatenation of the two lists, rather than an element-by-element sum. For example,

```
In[1]: a = [1, 2, "cat"]
In[2]: b = [3, 4, "dog"]
In[3]: print(a + b)
[1, 2, 'cat', 3, 4, 'dog']
In[4]: type(a)
Out[4]: list
```

Note the use of the "type" function to ask Python what type of object is the variable **a**.
Here's another way to define a list with 8 elements, all of which are set to 3.

```
In [1]: a=[3]*8; a
Out[1]: [3, 3, 3, 3, 3, 3, 3, 3]
```

Recall that the first list element has index value 0, not 1. For example,

```
In [1]: a=[1, 2, 3, 8]
In [2]: print(a[0],a[3]) # print the first and last
1 8
```

You will certainly do more with arrays than with lists. Numpy can create them and do various operations on them. Copy/paste this script into a file and run it:

```
####################################################################

# This file is unit01_ArrayOperations.py. It contains a few examples
# of operations on lists and arrays

# George Gollin, University of Illinois, May 20, 2016

####################################################################

# use numpy to create arrays, which can be used for arithmetic operations.
```

```
aa = np.array([2, 3, 5])
bb = np.array([7, 9, 11])

# do an element-by-element sum:
print("aa = ", aa)
print("bb = ", bb)
print("aa + bb = ", aa + bb)

# calculate an element-by-element product:
print("aa * bb = ", aa * bb)

# add a scalar to every element of an array. Note the "newline" \n.
print("\naa + 100 = ", aa + 100)

# multiply every element of an array by a scalar
print("\naa * 6 = ", aa * 6)

# take the sqrt of every element of an array
print("\nnp.sqrt(aa) = ", np.sqrt(aa))

# take the square of every element of an array
print("\naa**2 = ", aa**2)

# take the sine of every element of an array
cc = np.array([0., np.pi/6, np.pi/4, np.pi/2])
print("\ncc (radians) = ", cc)
print("np.sin(cc) = ", np.sin(cc))

# convert radians to degrees
print("\nnp.degrees(cc) = ", np.degrees(cc))

# sum the elements in an array
print("\nnp.sum(aa) = ", np.sum(aa))

###################################################################
```

A very common mistake—I trip over this all the time—is to create a list instead of an array, then try to use it in a mathematical expression. I would suggest that you ALWAYS use numpy to make arrays: do this

```
cc = np.array([0., 3.2, 9., np.pi/6])
```

instead of this:

```
cc = [0., 3.2, 9., np.pi/6].
```

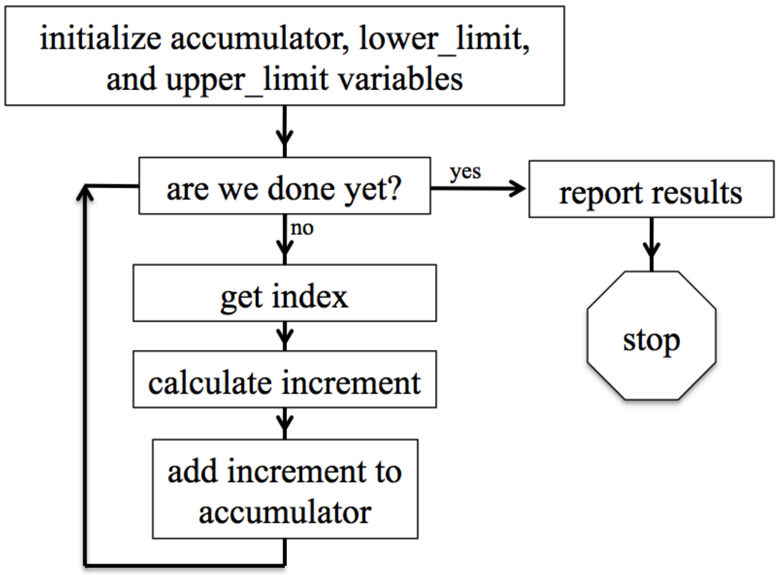
Note the placement of brackets and parentheses. What's happening here is that the np.array takes a Python list as input and produces a numpy array as output.

Loops

Most of your programs will include one or more loops. A loop is just what you'd expect it to be: a procedure that you execute many times, updating some of the variables each time you execute the loop.

When you write code you will want to be very clear about exactly what each line of your program is meant to accomplish. Unless you are already an experienced coder, you should consider drawing a diagram that illustrates what you think your software is going to do before you type a single line of code. Once you are clear about this you can begin writing code. I'll include flowcharts for some of the in-class exercises during the first several units to help you get the hang of this.

Here is a flow diagram for a typical loop. Note the names I've given to some variables: "accumulator," "lower_limit," "upper_limit," "increment," and "index."


initialize accumulator, lower_limit, and upper_limit variables
are we done yet?
yes
report results
no
get index
stop
calculate increment
add increment to accumulator

A loop to calculate the sum of a few squares

Here's the text of it; pay careful attention to the variable names in the loop. A common mistake new programmers make is to confuse the increment and accumulator variables.

```
"""
# This file is unit01_loop_structure.py. It contains a sample loop that
# calculates the sum of the squares of the numbers 1 through 10.

# George Gollin, University of Illinois, January 15, 2017
"""

# initialize variables here. take note of the names.

# the "accumulator variable" is where we sum the effects of whatever we
# calculated during successive passes through the loop. we initialize it to
# zero.  I am using the decimal point to make it a floating point variable,
# which isn't really necessary.

accumulator = 0.0

# the "increment variable" is something we'll generally need to calculate each
# pass through the loop. after calculating it we will add it to the accumulator
# variable. Since it will vary each time we go through the loop we don't need
# to initialize it here.

# specify the lower and upper limits for the loop now. Use the range function,
# which takes two integers as arguments, and creates a sequence of unity-spaced
# numbers. Note that the upper limit is not included in the sequence:
# range(1,5) gives the numbers 1, 2, 3, 4. Note that I will add 1 to the upper
# limit in my range function since range will stop short of this by 1.

lower_limit = 1
upper_limit = 10

# here's the loop. note the "whitespace" that is required, as well as the end-
# of line colon.

for index in range(lower_limit, upper_limit + 1):

    # in python we square things using a double asterisk followed by the
    # desired power. Note that a carat will not work: 3^2 is NOT 9.
    increment = index ** 2

    # now add into the accumulator.
    accumulator = accumulator + increment

    # I could have written all of this much more compactly using the +=
    # operator, but that'd be confusing, and you might find that it makes for
    # buggy, unclear code.

# we end the loop by having a line of unindented code.

print("all done! sum of squares is ", accumulator)

##################################################################

"""
Note that I could have written the code more compactly in a single line, but it
would have been harder to decipher:

   >> print(sum(np.array(range(1,11))**2))
      385
"""
```

Other loop matters

There is at least one other way to execute loops in Python, using "while" statements. For example, in the above code replace

```
for index in range(lower_limit, upper_limit + 1):
        increment = index ** 2
        accumulator = accumulator + increment
```

with

```
    index = lower_limit
    while index <= upper_limit:
        increment = index ** 2
        accumulator = accumulator + increment
        index = index + 1
```

It is possible to exit early from a loop by using the "break" command. Inserting the (properly indented) line

```
          if index > 5: break
```

into the loop will prematurely terminate it.

Functions and modules

As your programs get longer and more complicated, it might become convenient to break them up into multiple files, each containing one or more functions which are referenced by the main program, and/or by each other.
Here is an example, in which I have placed the functions SampleFunction1 and SampleFunction2 inside the file SampleFunctions.py.

```
    ###################################################################

    # This file is SampleFunctions.py. It contains a few sample functions
    # written in Python, included for pedagogical purposes.

    # George Gollin, University of Illinois, April 29, 2016

    ###################################################################

    def SampleFunction1(x, y, z):

        """
        This function returns (x * y) + z.

        Created on Thu Apr 28 16:34:11 2016

        Note that the multi-line string literal (all the stuff between the triple
    quotes)
        serves as a "docstring": it is printed in response to a help query about
    this
        function.

        Use SampleFunction1 this way:
```

```
    import SampleFunctions                                # load the module
    help(SampleFunctions.SampleFunction1)                 # ask for help
    TheAnswer = SampleFunctions.SampleFunction1(3,4,5)    # call the function

    author: g-gollin
    """

    WorkingVariable = x * y
    WorkingVariable = WorkingVariable + z
    return WorkingVariable

    # end of SampleFunction1

################################################################

# Now define a second function

################################################################

def SampleFunction2(x, y):

    """

    This function returns sqrt(x^2 + y^2). Use this way after importing the
    module:
    print(SampleFunctions.SampleFunction2(3,4))

    """
    # sum the squares of the two arguments
    WorkingVariable = x**2 + y**2

    # now take the square root.
    WorkingVariable = WorkingVariable ** 0.5

    # all done.
    return WorkingVariable

    # end of SampleFunction2

################################################################
```

For the sake of clarity I have made no attempt to write efficient code! For example, I could have shortened the executable parts of SampleFunction1 into a single line:

```
return x * y + z
```

Things to note:

- Each function begins with a few lines of text set off by triple quotes. Python treats these as a "docstring" and will spit them out in response to a help query about the function.

- There are a lot of explanatory comments. You should not be parsimonious in your inclusion of comments in your own programs!
- You refer to the functions inside a "module" using notation that is very common in object oriented languages: <module name>.<function name>. The module name is just the name of the file, with the ".py" filename extension omitted. For example,

```
Hypotenuse = SampleFunctions.SampleFunction2(5,12)
```

An exercise: an infinite series for π

Recall that we can generally find infinite series representations of transcendental functions like sin(x). In particular,

$$\tan^{-1}(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1} \qquad -1 < x \le 1.$$

Since $\tan^{-1}(1) = \pi/4$, we can write the following (slowly converging) infinite series:

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots.$$

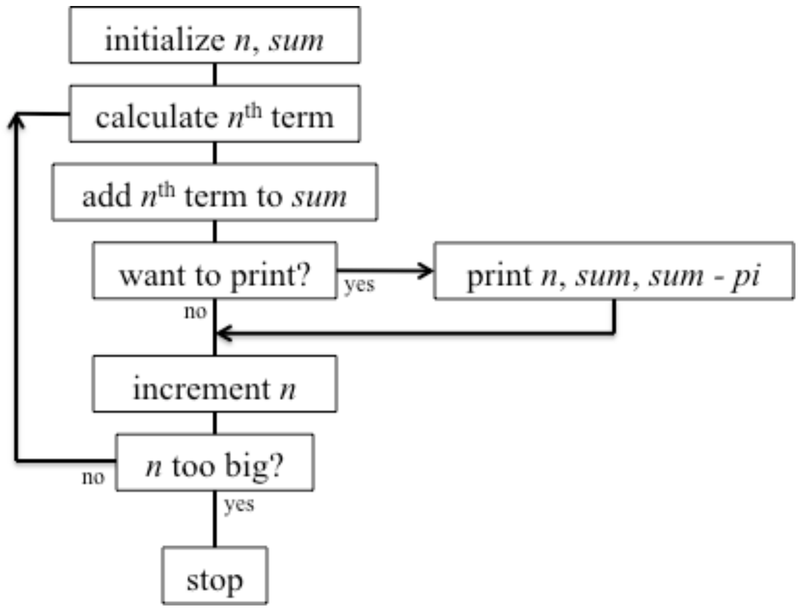
If we group adjacent terms in the series we can rewrite this as

$$\begin{aligned} \frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} &= \left(1-\frac{1}{3}\right) + \left(\frac{1}{5}-\frac{1}{7}\right) + \left(\frac{1}{9}-\frac{1}{11}\right) + \cdots \\ &= \frac{3-1}{3\cdot 1} + \frac{7-5}{7\cdot 5} + \frac{11-9}{11\cdot 9} + \cdots \\ &= \frac{2}{3} + \frac{2}{35} + \frac{2}{99} + \cdots \\ &= 2\cdot\sum_{n=0}^{\infty}\left[\frac{1}{(4n+3)(4n+1)}\right]. \end{aligned}$$

The value of π is 3.14159265358979323846264338327950288419716939937510582…, though the precision with which your computer can calculate it is probably limited to fewer digits than this.

Please write a Python script that calculates an approximation to π using the arctan series, and compare its accuracy after the n = 10 term, 100 term, 10,000 term, and 1,000,000 term. (Use a conditional statement to print something after the appropriate terms.)

You should approach this by initializing a few things, then executing a loop that calculates the n^{th} term, with n running from 0 to 999,999, summing the terms as you go. Here's a flowchart for one way to structure your program…



…and here's a listing of a template you could start with.

```
"""
Goal/purpose: This file is a template. You will build
your arctan(1) series (as well as subsequent in-class and homework assignments)
from it.
The code here actually calculates the sum of the square roots of the integers
0, 1, 2, 3, 4.

Assignment: unit 1 in-class machine exercise 2

Author(s): Monica and George

Collaborators: Monica, George, and Neal (CS professor)

Date: January 18, 2017


Reference(s):
Stack overflow web site (see
http://stackoverflow.com/documentation/python/193/getting-started-with-python-
language#t=201701181706539874984)
Physics 246 course notes

"""

###################################
# Import libraries
###################################
```

```
import numpy as np

###################################
# Define and initialize variables
###################################

# Accumulator variable
accumulator = 0

# Index and upper limit variables for the loop
lower_limit = 0
upper_limit = 4

################################################################################
# Loop to sum the square roots of a bunch of integers
################################################################################

for index in range(lower_limit, upper_limit + 1):

    # calculate increment, then add it to accumulator.
    increment = np.sqrt(index)
    accumulator = accumulator + increment

    # I could have just added np.sqrt(index) to accumulator, without defining
    # increment.

################################################################################
# End of loop. Print the results.
################################################################################

print("all done. Sum of square roots is ", accumulator)

################################################################################
#
```

Libraries

Numpy

Numpy is one of the libraries that is included with the Anaconda Python release. Wikipedia describes it this way:[1] "NumPy… is an extension to the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays."
You'll need to make Python aware of its existence by importing it:

```
In [1]: import numpy
```

You can see what lives inside numpy by issuing this command:

```
In [2]: dir(numpy)
```

[1] https://en.wikipedia.org/wiki/NumPy

```
Out[2]:
['ALLOW_THREADS',
'BUFSIZE',
'CLIP',
'ComplexWarning',
…
```

There's quite a lot there, including a sqrt routine. After importing numpy you can call its routines like this:

```
In [3]: numpy.sqrt(2)
Out[3]: 1.4142135623730951
```

If you would prefer to use a shorter name for numpy (perhaps to save some typing), you could have imported it this way:

```
In [4]: import numpy as np
In [5]: np.sqrt(2)
Out[5]: 1.4142135623730951
```

Numpy also knows the value for π:

```
In [6]: np.pi
Out[6]: 3.141592653589793
```

Matplotlib

Matplotlib "is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits… There is also a procedural "pylab" interface… designed to closely resemble that of MATLAB."[2]
You will want to import both matplotlib and pyplot. Do the following:

```
In [6]: import matplotlib
In [7]: import matplotlib.pyplot as plt
```

You will probably want to include the import statements into scripts/programs you write so that you aren't required to import them from the iPython console each time. (But there's nothing wrong with importing something multiple times.)
There is good documentation (including examples) here: http://matplotlib.org/.

[2] https://en.wikipedia.org/wiki/Matplotlib

Drawing curves in two dimensions

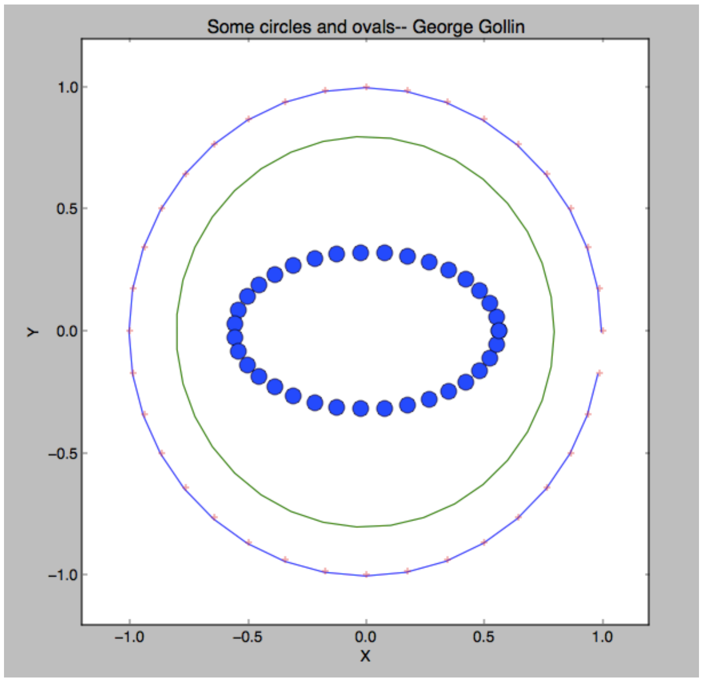
How to draw a circle

Let's load an array with a reasonably large number of *x*, *y* points that lie on a circle, then plot them and save the plot to a file. Here's a script that does this, making a number of figures in the same window.
Let's talk through what's in the file. When you make other kinds of plots, consider copying what's in this script into your own, then changing a few things to make it do what you want, rather than writing something from scratch. That'll save you time, and also the effort of understanding the minutiae of the graphics code.

```
# load arrays with coordinates of points on a unit circle, then plot them
# this file is unit02_draw_circles.py

# import the numpy and matplotlib.pyplot libraries
import numpy as np
import matplotlib.pyplot as plt

# create the arrays. first is angle, running from [0,2pi). Note the use of
# endpoint = False in linspace to make this interval open on the right. The
# first two arguments in linspace are the beginning and end of the interval
# of uniformly spaced points. The third is the total number of points.
ThetaArray = np.linspace(0, 2*np.pi, 36, endpoint=False)

# cos and sin in numpy can act on all elements in an array. Note that the
# output is also an array.
x = np.cos(ThetaArray)
y = np.sin(ThetaArray)

# set the size for figures so that they are square. (Units: inches???)
plt.figure(figsize=(8.0, 8.0))

# also set the x and y axis limits
plt.xlim(-1.2, 1.2)
plt.ylim(-1.2, 1.2)

# plot the x,y points, connecting successive points with lines
plt.plot(x,y)

# now plot the points again, on the same axes, but using red + signs:
plt.plot(x,y, 'r+')

# now redo the array of angles (and x,y points) to include the 2pi endpoint.
# make a smaller circle this time...
ThetaArray = np.linspace(0, 2*np.pi, 36, endpoint=True)
x = 0.8*np.cos(ThetaArray)
y = 0.8*np.sin(ThetaArray)

#plot it. note how the line color changes...
plt.plot(x,y)
```

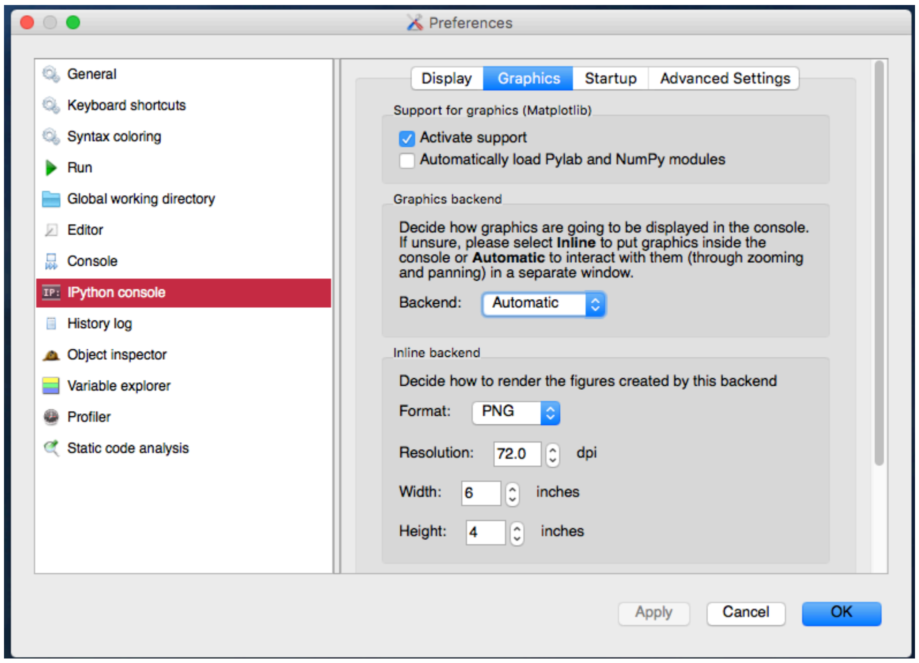
```
# now make an oval by down-scaling the x, y arrays, then plot
# it using rather large, filled blue circles.
x = 0.7 * x
y = 0.4 * y
plt.plot(x,y, 'bo', markersize=12)

# now put a title onto the plot, then label the axes
plt.title("Some circles and ovals-- George Gollin")
plt.xlabel("X")
plt.ylabel("Y")

# now save plot to a png (portable network graphics) file
plt.savefig("CirclePlotOne.png")
```

Note that ***some*** functions (such as np.sin) will act on all the elements in the array to which it is applied. This is very convenient! Here's what we get:



Drawing figures in new windows

If you forgot to attend to setting Python's parameters when you installed it last week, your version of Spyder probably puts your graphs into the same iPython console that you use to enter commands. To make plots open in new windows, go to the python preferences menu (on a Mac it lives in the "Python…" menu at the top of the screen), then select "iPython console" and "Graphics." Set the "Backend" field to Automatic.



Support for graphics (Matplotlib)

- [✓] Activate support
- [ ] Automatically load Pylab and NumPy modules

Graphics backend

Decide how graphics are going to be displayed in the console. If unsure, please select **Inline** to put graphics inside the console or **Automatic** to interact with them (through zooming and panning) in a separate window.

Backend: Automatic

Inline backend

Decide how to render the figures created by this backend

Format: PNG

Resolution: 72.0 dpi

Width: 6 inches

Height: 4 inches

Once you've done this, each figure should open in a new window.

Another exercise: graphing the magnification of a weird optical system

Imagine that you stumble upon a strange optical device that produces magnified images of objects placed downfield of a critical point x_c = 10 meters. The magnification M—the ratio of image height to object height—is guaranteed by the manufacturer to satisfy the equation

$$M(x) = \frac{1}{\sqrt{1-\frac{x_c}{x}}} \; .$$

(The manufacturer's literature warns that the device will self-destruct if it is exposed to objects closer than x_c.)

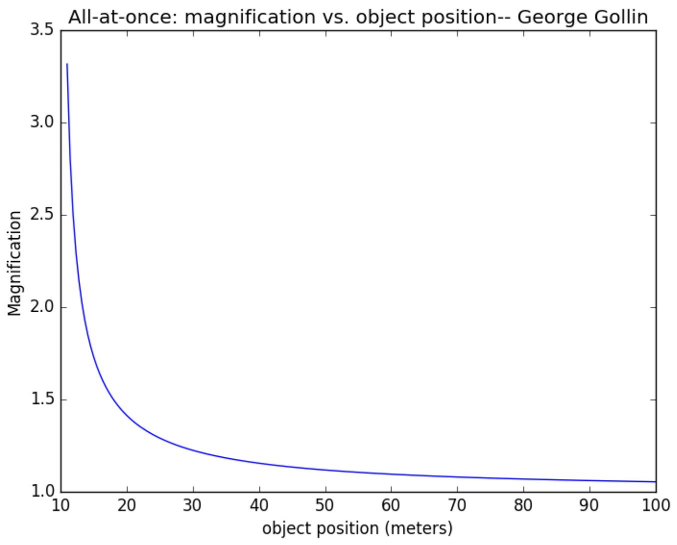
Please generate a graph of $M(x)$ vs. x for the range $1.1\, x_c < x < 10\, x_c$. Use a step size that is small enough to allow your graph to look smooth and continuous. Do it one of two ways: by coding up a loop that loads appropriate arrays or by using Python's all-at-once capabilities built into numpy for performing array calculations.

If you're not sure how to get started, make a copy of unit02_draw_circles.py and throw away what you don't need, then have it generate an array of x values, and then the corresponding magnifications.

You can put the following line of code into your program before you make a plot to close all already-open graphics windows, if you want: `plt.close("all")`.

Note that this strange function is going to appear in discussions of space-time curvature in General Relativity. In that context, x_c is replaced with the Schwarzschild radius of a compact

massive object. And the "magnification" is instead a measure of the discrepancy between 2π and the ratio of the circumference and radius of a circle with the massive object at its center.
Your result should look something like this:



Graphical representations of three dimensions

If we want to draw a curve in three dimensions, we'll need to import more libraries. You'll want something like this in your programs:

```
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt
```

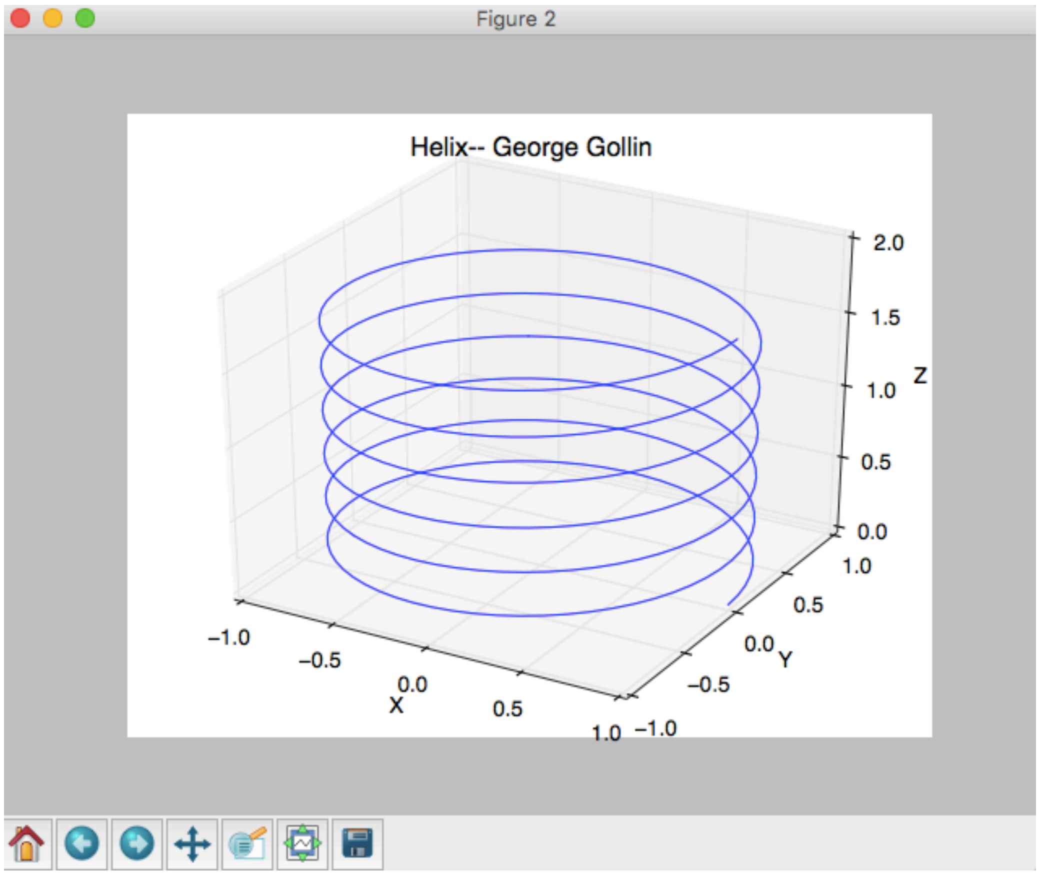
Drawing a helix

Let's say we want to draw three turns of a helix whose projection on the *x-y* plane is a circle of unit radius, and that advances along the positive *z* axis by 0.3 meters per turn. I will assume the helix begins at $(x, y, z) = (1, 0, 0)$, and that it winds in a counterclockwise direction when seen from above.
Here is a script that generates the drawing.

```
# load arrays with coordinates of points on a helix, then plot them
# this file is unit02_draw_helix.py

# import libraries. Python may complain about the first one, but you really do
# need it.
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt
```

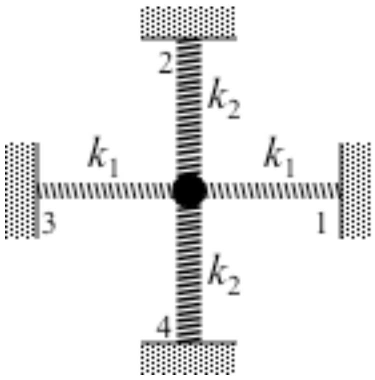
```
# total number of turns the helix will make
NumberTurns = 6

# pitch (meters per turn)
pitch = 0.3

# radius
Radius = 1.0

# points to plot per turn
PointsPerTurn = 60

# create the array of angles for successive points. the arguments to linspace
# are (1) first value; (2) last value; (3) number of equally-spaced values
# to put into the array.
ThetaArray = np.linspace(0, NumberTurns*2*np.pi, NumberTurns*PointsPerTurn)

# Now get x,y,z for each point to plot.
x = np.cos(ThetaArray)
y = np.sin(ThetaArray)
z = np.linspace(0, NumberTurns*pitch, NumberTurns*PointsPerTurn)

# now create a (blank) figure so we can set some of its attributes.
fig = plt.figure()

# "gca" is "get current axes." set the projection attribute to 3D.
ax = fig.gca(projection='3d')

# set the x, y, and z axis limits of the plot axes
ax.set_xlim(-1, 1)
ax.set_ylim(-1, 1)
ax.set_zlim(0, 2)

# label the axes and give the plot a title
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
ax.set_title("Helix-- George Gollin")

# now plot the helix.
ax.plot(x, y, z)

# now save the plot to a png (portable network graphics) file
plt.savefig("HelixPlot.png")
```

The result follows. Take note of the pan/scroll/rotate button (the cross made of double-headed arrows): it allows you to rotate a 3D figure to view it from different angles. Try this out.



How to draw a surface whose height above the x-y plane depends on a function

Imagine that a mass is constrained to move in the x-y plane, and is held in place by four springs of length L as shown in the following illustration. Springs 1 and 3 have identical spring constants k_1, while springs 2 and 4 have spring constants k_2, with $k_2 > k_1$.



If the mass is displaced from equilibrium to the point (x, y) (with displacement that is very small compared to the springs' lengths L), the system's potential energy will increase by

$$\Delta U = k_1 x^2 + k_2 y^2 .$$

How might we draw the surface that represents $U(x, y)$?

I'll assume that $U(0, 0) = 0$, $k_1 = 2$, $k_2 = 5$, and that our graph is to span the range $-3 < x < 3$, $-3 < y < 3$, with a grid employing cell size 0.1×0.1. This means that our graph will show the height above the x, y plane at $61 \times 61 = 3{,}721$ points.

We could do something like this to begin defining the grid.

```
x = np.linspace(-3, 3, 61, endpoint=True)
y = np.linspace(-3, 3, 61, endpoint=True)
```

This will give us a pair of arrays, each of 61 elements, with successive entries spaced by 0.1. But that's not really what we want: we need a list of the x, y coordinates of all 3,721 points on our grid. We can do this using the **`numpy`** function **`meshgrid`** after defining the **`x`** and **`y`** arrays (as I did a few lines ago):

```
xgrid, ygrid = np.meshgrid(x,y)
```

I appreciate that this is confusing at first. Let's consider smaller arrays so I can print them out for you. Here's what I get, after importing **`numpy as np`**:

```
# make 3-element arrays which give the x values of "columns"
# and y values of "rows."
In [1]: x = np.linspace(-1, 1, 3, endpoint=True)
In [2]: y = np.linspace(-1, 1, 3, endpoint=True)

# now list the values for the x and y arrays.
In [3]: x
Out[3]: array([-1.,  0.,  1.])
In [4]: y
Out[4]: array([-1.,  0.,  1.])

# our 3 x 3 array will have nine cells, of course, so we will need to
# load one 9-element array with the x values for each cell and another
# with the y values for each cell.
In [5]: xgrid, ygrid = np.meshgrid(x, y)

# now list the x values for each of our nine cells
In [6]: xgrid
Out[6]:
array([[-1.,  0.,  1.],
       [-1.,  0.,  1.],
       [-1.,  0.,  1.]])
# now list the y values for each of our nine cells
In [7]: ygrid
Out[7]:
array([[-1., -1., -1.],
       [ 0.,  0.,  0.],
       [ 1.,  1.,  1.]])
```

Take note of the order in which the cells appear in our arrays: the first cell is at $x = -1$ and $y = -1$. The second is at $x = 0, y = -1$; the third at $x = +1, y = -1$, and so forth.

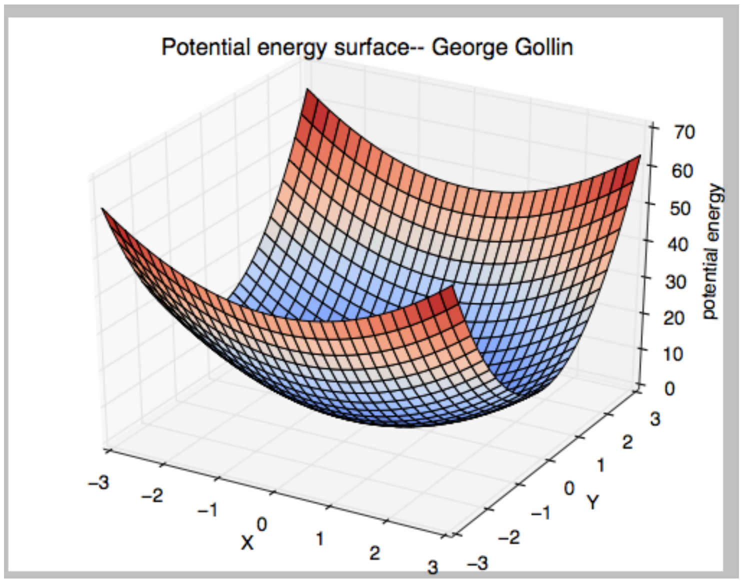
	$x = -1$	$x = 0$	$x = +1$
$y = +1$	7	8	9
$y = 0$	4	5	6
$y = -1$	1	2	3

Now we're ready to create another array that holds the potential at the *x,y* position of each cell. A listing of a program that actually generates a graph of *U* (along with the program's output) follows.

```
# load arrays with coordinates of points on a surface, then plot them
# this file is unit02_draw_surface.py

# import libraries
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt
# let's also import a color map so we can make the picture prettier
from matplotlib import cm

# define parameters for our plot
xmin = -3
xmax = -xmin
ymin = xmin
ymax = -ymin

# number of rows and columns in our grid
nrows = 61
ncolumns = 61

# number of rows and columns per grid line to be drawn
rowsPerGrid = 2
columnsPerGrid = 2

# define the coefficients in the potential
xcoeff = 2
ycoeff = 5

# create the arrays.

# first get the x values of the "columns" in the grid.
x = np.linspace(xmin, xmax, ncolumns)

# now get the y values of the "rows" in the grid.
y = np.linspace(ymin, ymax, nrows)

# now generate the x and y coordinates of all nrows * ncolumns points
```

```
xgrid, ygrid = np.meshgrid(x,y)

# now generate the potential for all points on our grid.
zsurface = xcoeff * xgrid**2 + ycoeff * ygrid**2

# now create a (blank) figure so we can set some of its attributes.
fig = plt.figure()

# "gca" is "get current axes." set the projection attribute to 3D.
ax = fig.gca(projection='3d')

# set labels and title
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("potential energy")
ax.set_title("Potential energy surface-- George Gollin")

# now put the graph into the blank figure. Note the line-continuation character.
# colormap (cmap) argument is optional.
surf = ax.plot_surface(xgrid, ygrid, zsurface, rstride=rowsPerGrid, \
cstride=columnsPerGrid, cmap=cm.coolwarm )

# now save the plot to a png (portable network graphics) file
plt.savefig("SurfacePlotCartesian.png")
```

The "rstride" and "cstride" arguments in the ax.plot_surface function tell the graphics routines how many rows and columns in the *x-y* mesh to use as the spacing between grid lines drawn on the surface. You can replace the color map specification **`cmap=cm.coolwarm`** with an RGB hexadecimal code for the color to be used on the surface, for example **`color="#FF0000"`** to use red.



Drawing a surface using polar coordinates

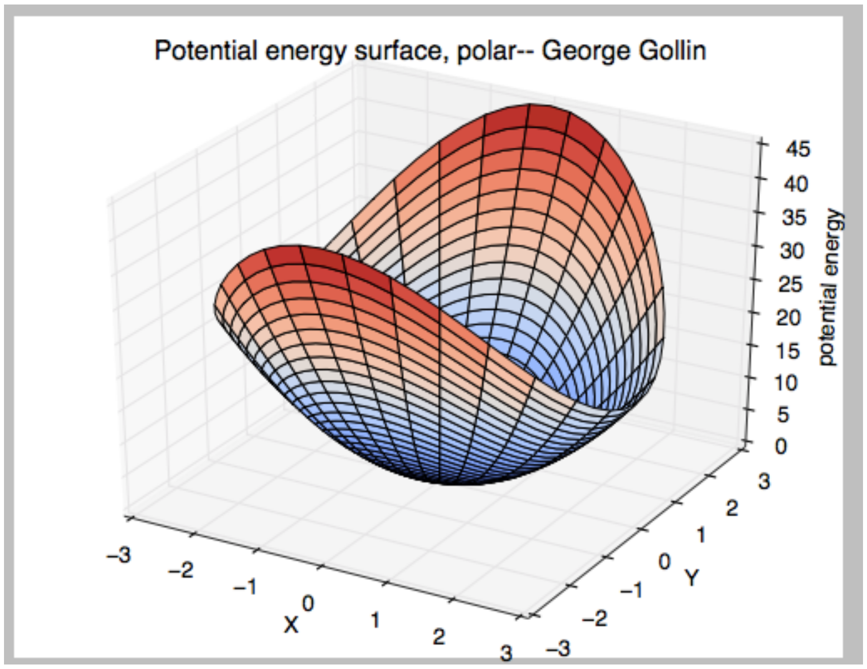
It is easy to draw a surface using cylindrical, instead of Cartesian coordinates. To do this in the previous example, replace the lines

```
x = np.linspace(ymin, ymax, nrows)
y = np.linspace(ymin, ymax, nrows)
xgrid, ygrid = np.meshgrid(x,y)
```

with something like this (after setting rmax and thetamax):

```
r = np.linspace(0, rmax, 61)
theta = np.linspace(0, thetamax, 61)
rgrid, thetagrid = np.meshgrid(r, theta)
xgrid, ygrid = rgrid*np.cos(thetagrid), rgrid*np.sin(thetagrid)
```

The result is shown below.



A final exercise: graphing a surface you'll see in General Relativity

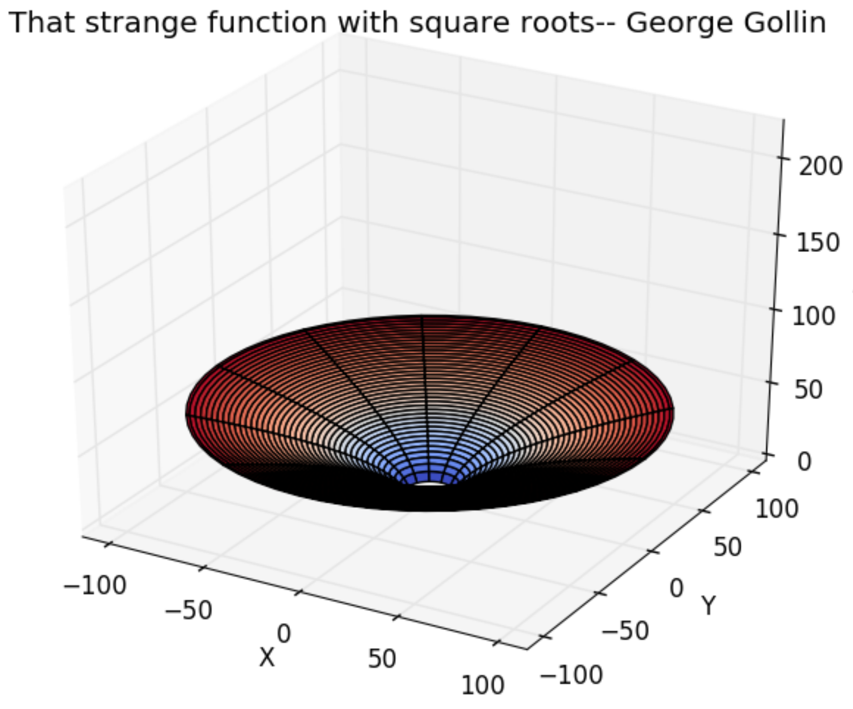
Here's another function you'll see when discussing curved spacetime, when you might use an approximation to it to generate what is called an embedding diagram. Consider the following function $z(r,\theta)$, where r, θ are the familiar polar coordinates. (Note that z is actually independent of θ.)

$$z(r,\theta) = 2r_S\left\{\sqrt{\frac{r}{r_S}-1}-\sqrt{\frac{r_0}{r_S}-1}\right\}.$$

Assume that the scale parameter r_S has the value $r_S = 10$ meters and that the constant r_0 has the value 11 meters.

Please plot the surface defined by z for $r_0 < r < 10r_0$ and $0 < \theta < 2\pi$. Your plot ought to look something like mine, below. Note that I've forced the aspect ratio to be 1:1:1 by doing this:

```
ax.set_xlim(-rmax, rmax)
ax.set_ylim(-rmax, rmax)
ax.set_zlim(0, 2*rmax)
```


That strange function with square roots-- George Gollin
X
Y
Z