Physics 524

Survey of Instrumentation and Laboratory Techniques

2024

University of Illinois at Urbana-Champaign

Unit 1b: Anaconda Scientific Python

Goal for this week

- Learn what's inside the box: a simple model of a computer;
- 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;

A table of useful python stuff

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)

What's under the hood

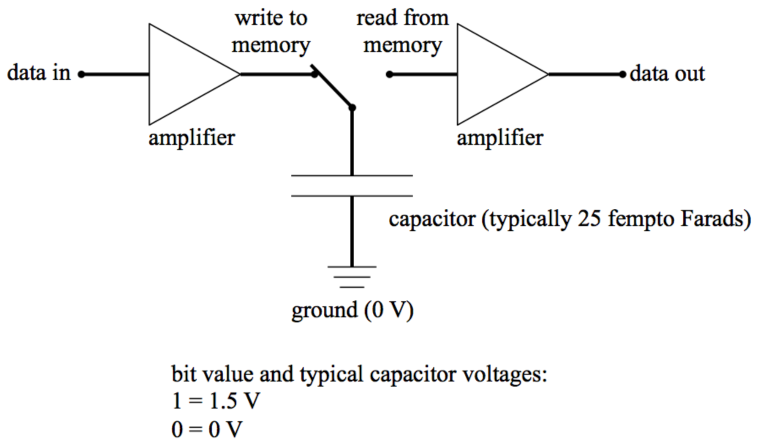
I suspect that most of you aren't all that familiar with what is going on inside your laptops at the most primitive level, where it is appropriate to think of your computer as a complex web of voltage and current sources, and capacitances, and interconnects, and field effect transistors. So let's sand all the paint off, and discuss an atomistic model for what's at the heart of our extraordinarily sophisticated personal computers. We'll do this by constructing a simple, basic, unsophisticated model of a computer that we could build from parts that were available (except for the DRAM chip) decades ago.

Bits, bytes, and words

hardware representation of a bit

First things first: how do we construct a circuit that can represent a zero or a one? We'll assign zero to 0 volts and one to 1.5 volts.

Here's one way to do it, with one amplifier charging (or discharging) a small capacitor when storing a bit (a 1 or a 0) in memory, and another amplifier reporting the capacitor voltage when reading the value stored in memory. No current flows into the right side amplifier's input, so it doesn't drain charge off the capacitor.



The word "bit" is a contraction of "binary digit." In real DRAMs (Dynamic Random Access Memories) the switch is a field effect transistor.

bytes, words

We group eight bits together to form a byte. If we assign the n^{th} bit in a byte to represent 2^n, one byte can hold a value anywhere from 0 (00000000_2) to 255 (11111111_2), where I am using a "2" subscript to indicate base 2.

We group bytes together to form words. The number of bytes per word depends on the architecture of a particular memory or processor chip; common values are two, four, and eight bytes per word. Most of your laptops are 64-bit machines, corresponding to eight-byte words.

It is convenient to represent the content of a word using hexadecimal notation, with two hexadecimal digits per byte. A couple of examples, in which I've separated the two halves of a byte by a space for clarity:

- $0011\ 1111_2$ is $3F_{16}$ or 63_{10}
- $0000\ 0000_2$ is 0
- $1111\ 1111_2$ is FF_{16} or 255_{10}.

In Python you'd tell the computer that you're writing the hexadecimal representation of a number by preceding the hex digits with 0x. Note that it's a zero, not the letter "o" before the "x." You'd use 0b for binary and 0d for decimal, which is the default. In other words, 0x3F = 0b00111111 = 0d63 while 0d63 = 63.

What does a word actually represent?

For readability I'll leave a blank space between bytes when writing hexadecimal representations of 64-bit words. Let's say you find that a word in memory contains 0x 20 20 20 20 20 63 61 74. What does it actually mean? That depends on the context.

ASCII

If you are told that the word holds the ASCII ("American Standard Code for Information Interchange") representation of something, you'd find a table of ASCII codes and translate it as "   cat" since 0x20 is a blank space, while 0x63 is a "c" and so forth.

integers

If another word contains 0x 00 00 00 00 00 00 00 2B and is known to represent an integer, you'd unpack it as (2 × 16) + (B × 1) = 32 + 11 = 43 since B is the hexadecimal representation of 11. If the high order bit were set so that our word contained 0x 80 00 00 00 00 00 00 2B, we'd interpret that as a negative integer instead. (There are some subtleties in how computers represent negative numbers, which I will skip. See material on the web about "one's complement" and "two's complement.")

floating point and precision

The representation of "floating point" numbers is quite different. By floating point, I mean something with a decimal point, e.g. 3.1415926535… and the like. The IEEE (Institute of Electrical and Electronics Engineers) standard for 64-bit floating point is this:[1]

what	**which bit(s)**	**range of values**
sign	63 (1 bit)	[0, 1]
exponent	52 – 62 (11 bits)	[-1022, 1023]
significand	0 – 51 (52 bits)	[0, 4.5035996×10^{15}]

For example, $2.71828 = 271{,}828 \times 10^{-5}$; here 271,828 is the significand while -5 is the exponent.

The guaranteed precision corresponds to slightly less than 16 decimal digits. There are some additional subtleties involving "cohorts," "hidden bits," and so forth. The Wikipedia article I cite as a reference has a good discussion of the details.

Here's what I mean by "precision." Imagine that you add the following numbers:

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

```
 1,234,567,890.0
+              0.123456789
 -------------------------
 1,234,567,890.123456789
```

The available machine precision could yield the following surprising result:

```
 1,234,567,890.0
+              0.123456789
 -------------------------
 1,234,567,890.123456
```

It's just something to keep in the back of your mind when you write code.

GG's simple-minded computer model

Memory

In 1945 John von Neumann proposed[2] that a "very high speed automatic digital computing system" could use memory to store both its program and the data on which the program would act. It was a brilliant realization, and that's how modern computers have been designed ever since.

We program our computer by loading the appropriate instructions and data into memory, then allowing the computer's CPU (Central Processing Unit) to read (and execute) instructions from memory.

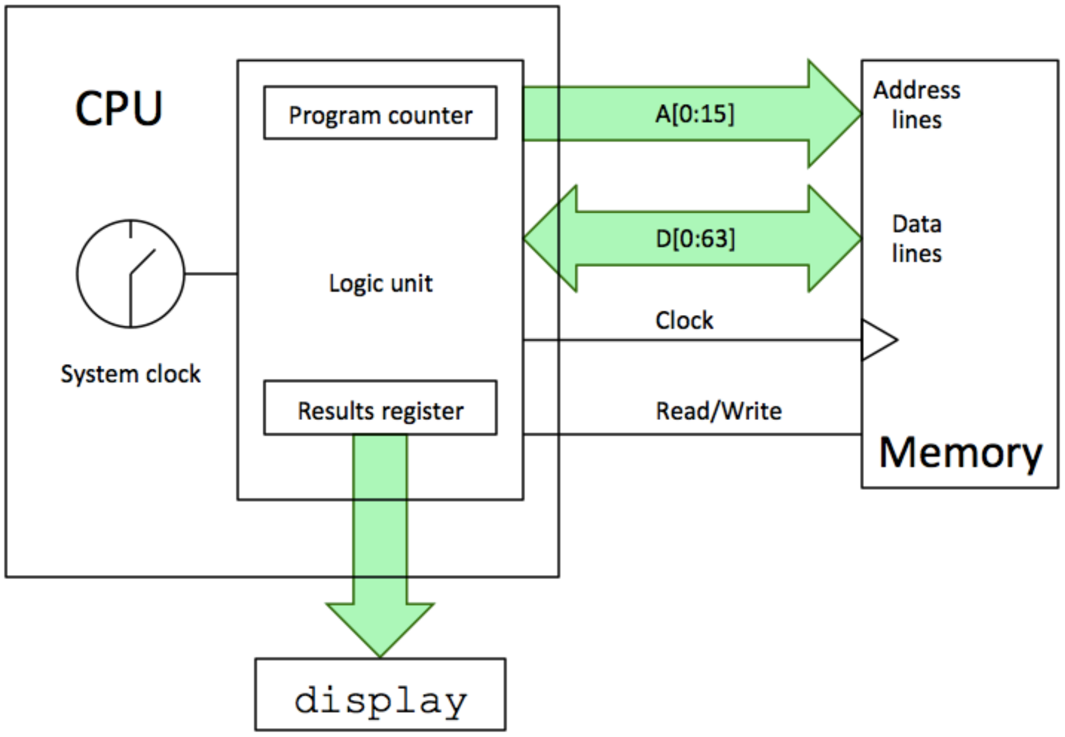
CPU

The heart of a computer is its central processing unit. In the following diagram I show a model for a simple CPU, including its communication lines and one register, a part of the CPU that communicates with the outside world.

Each time the system clock "ticks," the Logic unit fetches, then executes an instruction from the memory address specified in the Program counter.

Instructions for our toy computer might contain three fields: an operation code (op code) and a pair of addresses a1 and a2. To add the contents of a1 to the contents of a2, storing the result in a2, we'd have the operating system load an instruction into memory with the appropriate op code and address values.

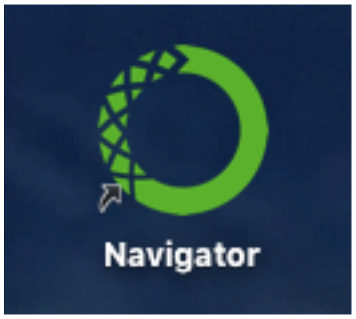
[2] https://en.wikipedia.org/wiki/First_Draft_of_a_Report_on_the_EDVAC



That's enough about hardware for the time being.

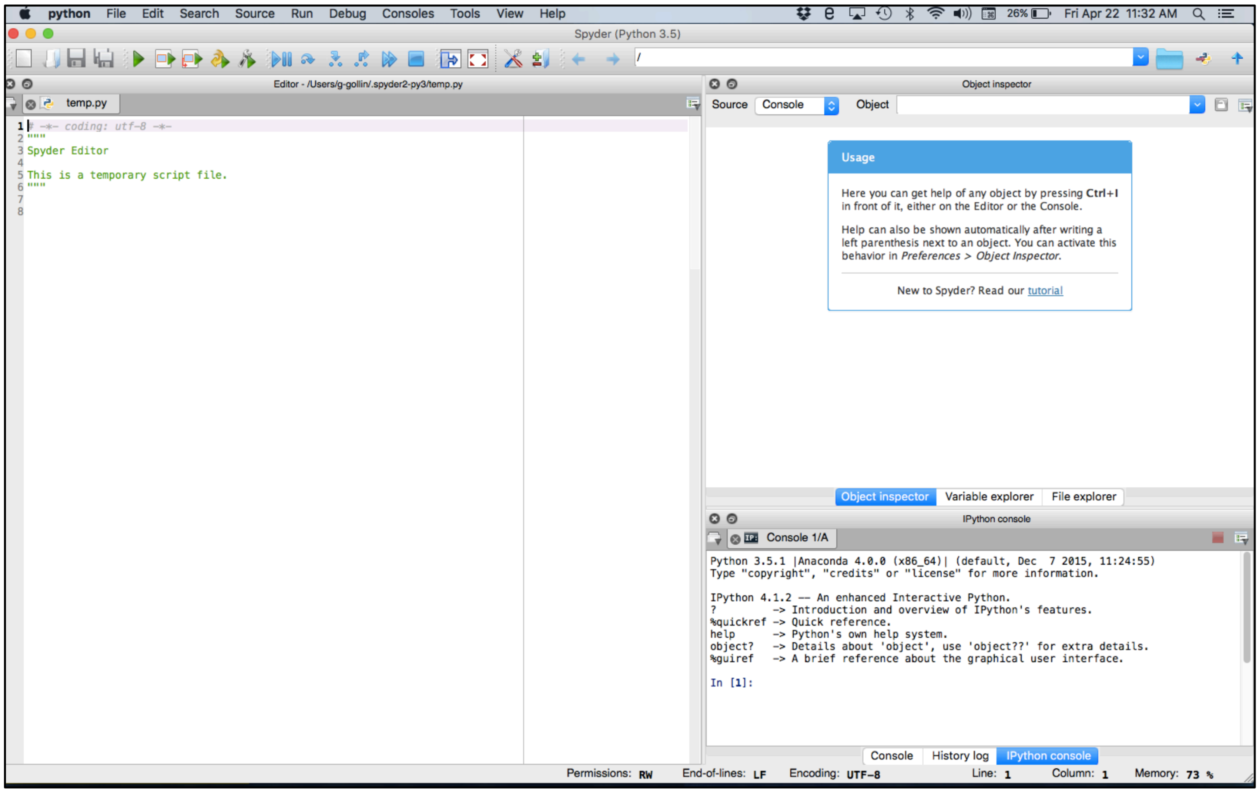
Installing Anaconda Python

Please copy the installation file from the USB thumb drive to a sensibly-named folder on your system disk. After you have copied (or downloaded) the installation file, execute it to install Python. 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.



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

This is a temporary script file.
"""

```

Usage

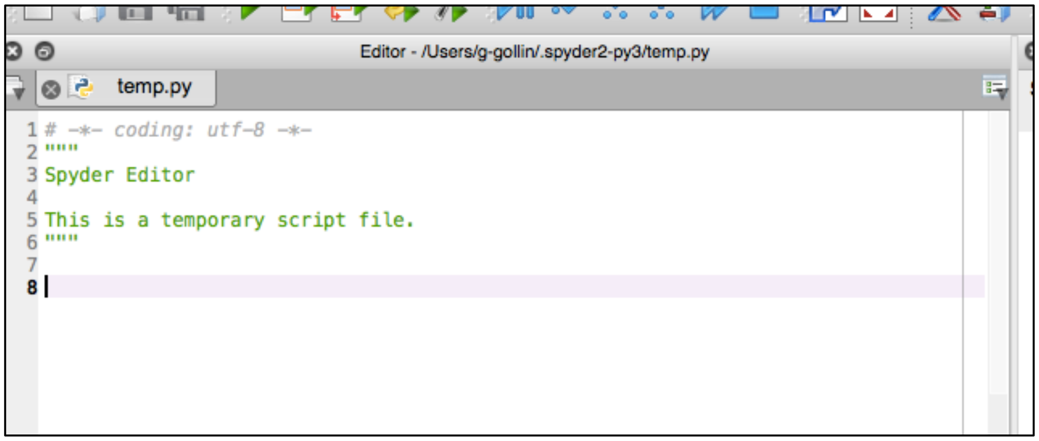
Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

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

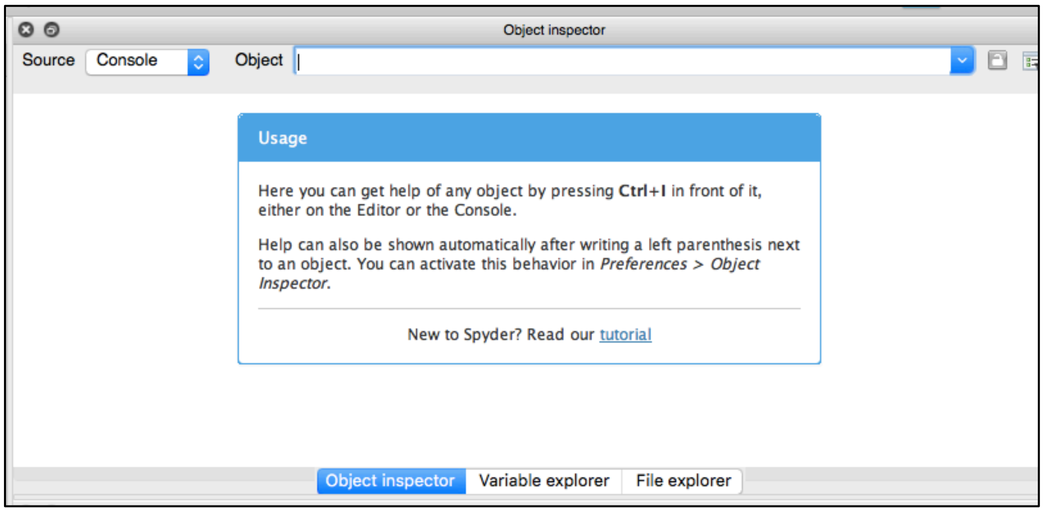
```
Python 3.5.1 |Anaconda 4.0.0 (x86_64)| (default, Dec  7 2015, 11:24:55)
Type "copyright", "credits" or "license" for more information.

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" (more on this towards the end of the course), 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.



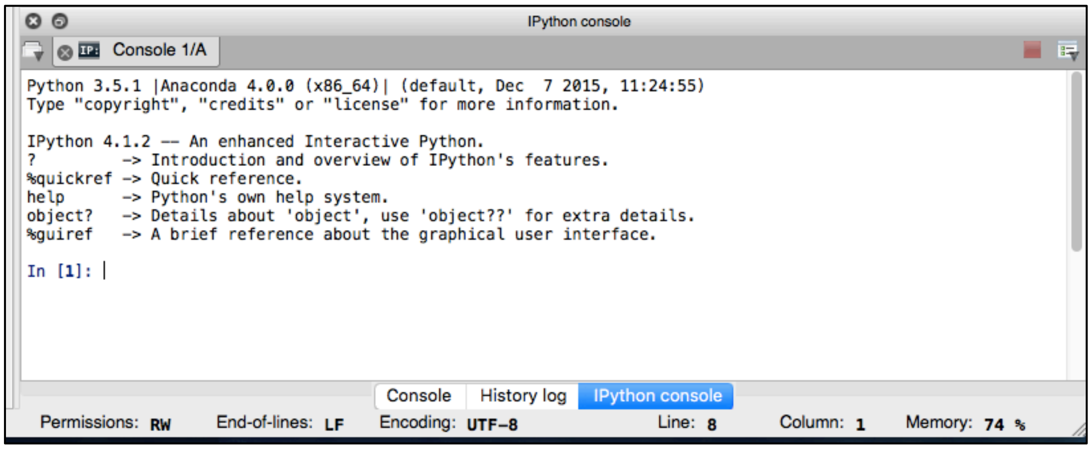
Usage

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

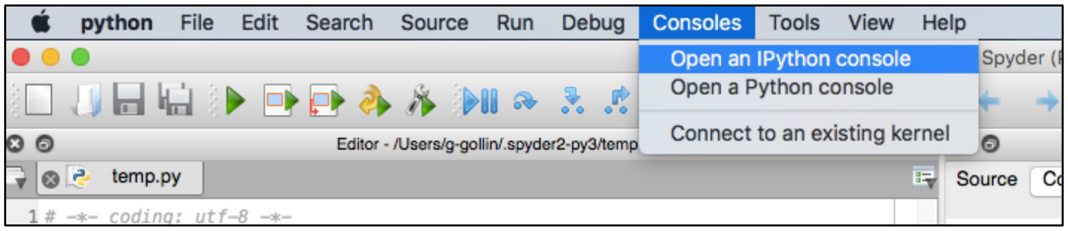
The lower right window shows an iPython console, a sort of operator's station from which you can issue commands to Python. It also displays program output.



```
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]: 
```

Permissions: RW End-of-lines: LF Encoding: UTF-8 Line: 8 Column: 1 Memory: 74 %

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.



```
# -*- coding: utf-8 -*-
```

Configuring Python

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.

Basic concepts

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
```

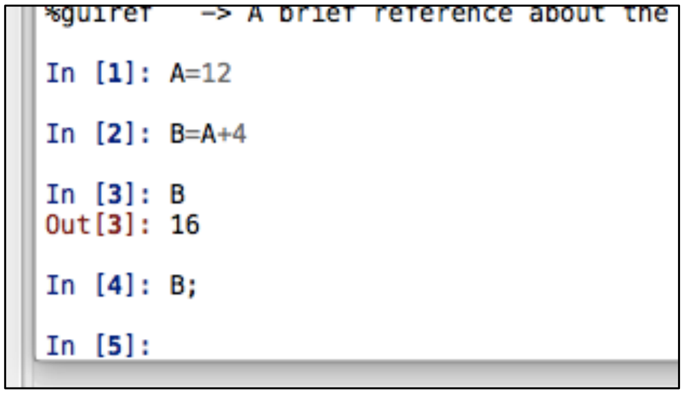
To inspect the value of **B** you would just type its name into the console:

```
B
```

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

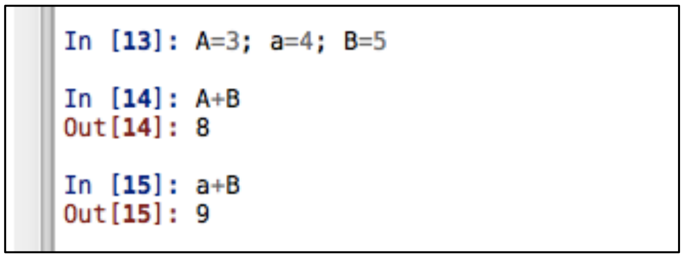
```
B;
```

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
```

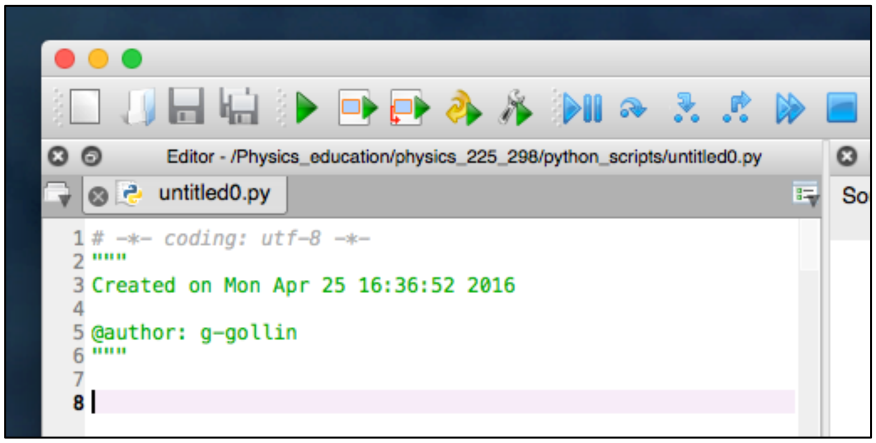
It is very clumsy to execute if-blocks this way! A better way is to put a string of executable instructions into a script file, then execute the script.

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.

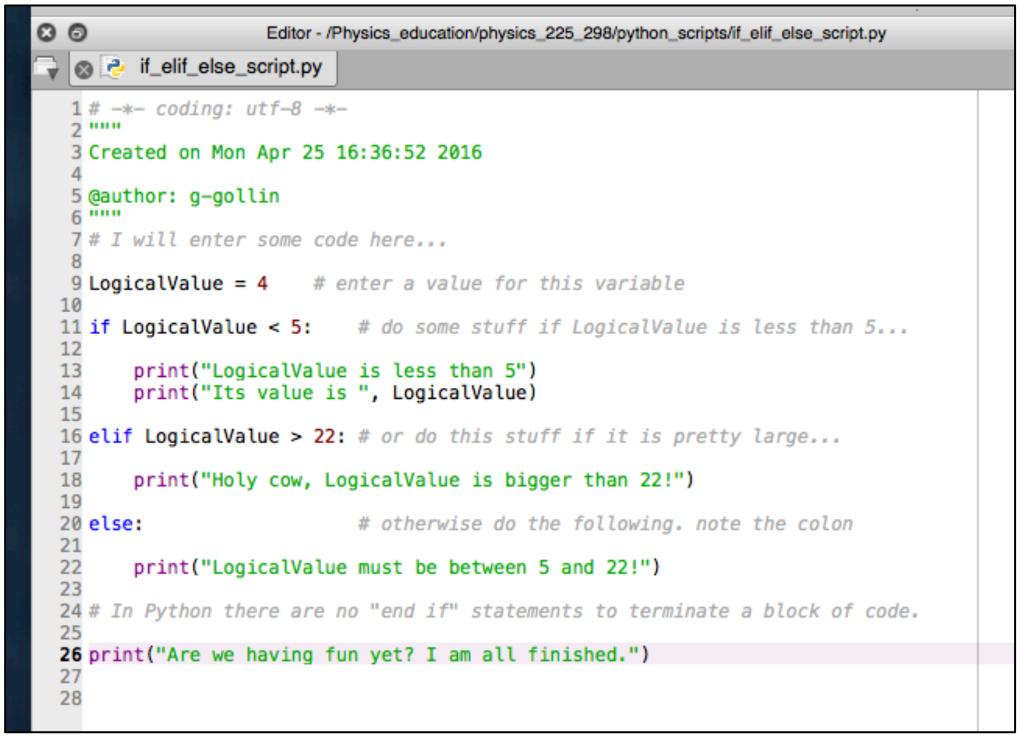
.


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
"""

```

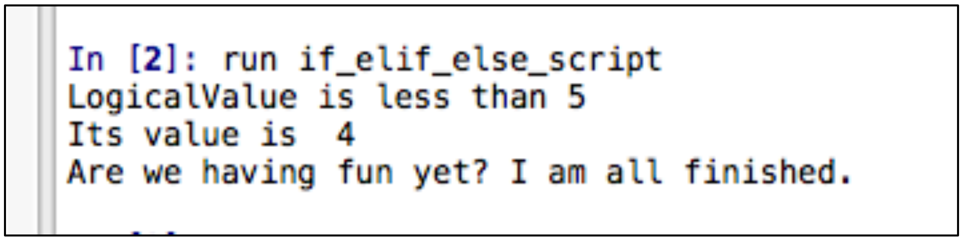
Enter some well-commented code into the editor window, then save the file. In the following screen shot I have an if-then-else-if block, followed by an example of running it from the console.

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. Download this script from the course web site 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].
```

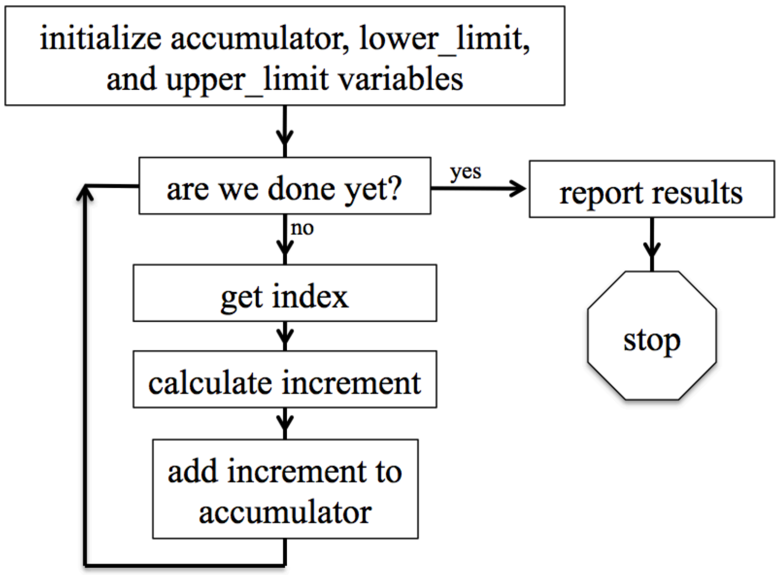
Note the placement of brackets and parentheses. What's happening here is that the np.array takes a Python list as input and produces a numpy array as output.

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
calculate increment
add increment to accumulator
stop

A loop to calculate the sum of a few squares

Download the file unit01_loop_structure.py.txt from the course web site's code repository, then strip off the .txt filename extension.

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)
```

In-class machine exercise 1: 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

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

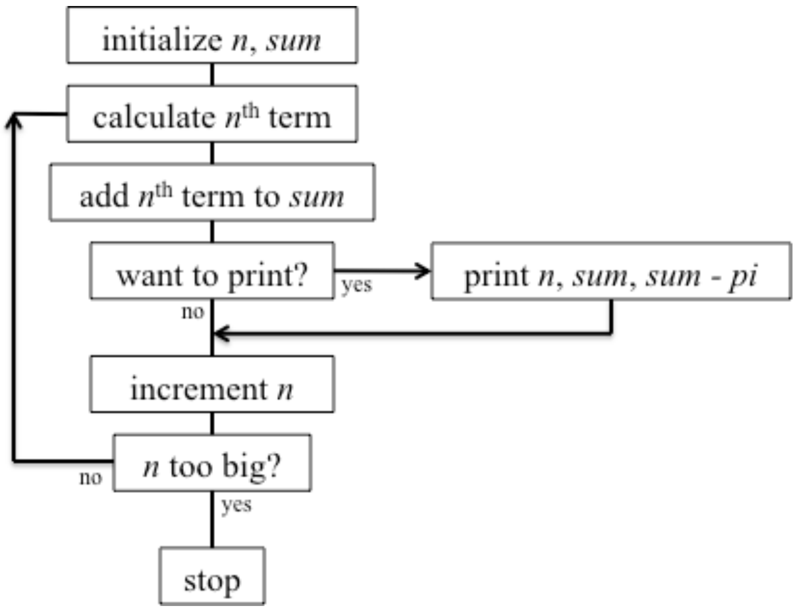
If we group adjacent terms in the series we can rewrite this as

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

The value of π□□□ 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 nth 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 template into which you can drop your code: you might find it useful.

```
"""
Goal/purpose: You will code an arctan(1) series.
The code here actually calculates the sum of the square roots of the integers
0, 1, 2, 3, 4.

Author(s): George G

Collaborators: Yuk Tung and George

File: who_knows.py

Date: August 2, 2024

Reference(s):
Stack overflow web site (see
http://stackoverflow.com/documentation/python/193/getting-started-with-python-
language#t=201701181706539874984)
Physics 524 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)

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

This week's homework assignment (due at the first class meeting next week)

1. A much better infinite series for π

Please reread all the in-class material, taking note of things that are unclear so that you can ask about them during office hours.

In class we worked on an arctan series to evaluate π. A much more rapidly converging series was discovered by the brilliant Indian mathematician Srinivasa Ramanujan[3]. It is

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{9801}\sum_{k=0}^{\infty}\frac{(4k)!(1103+26390k)}{(k!)^4\,396^{4k}},$$

where $k!$ ("k factorial") is $1 \times 2 \times 3 \times \cdots \times k$ and $(4k)!$ is $1 \times 2 \times 3 \times \cdots \times 4k$.

Please write a Python script that calculates an approximation to π using the Ramanujan series, and comment on its accuracy after 1, 2, and 3 terms. (Recall that the value of π is 3.14159265358979323846264338327950288419716939937510582…)

Note that there are even faster-converging formulas than this! One, mentioned in Wolfram MathWorld,[4] adds 50 digits of precision for each additional term.

For full credit, print out your three approximations and how each compares to π.

2. Relativistic spaceflight

As undergraduates you probably learned about some of the surprising consequences of special relativity: a moving clock ticks slowly, a moving object becomes shorter along its direction of motion, the velocity addition formula never yields a superluminal[5] speed.

Imagine that we have two frames of reference, which I'll call O and O'. ("O" stands for "origin," I suppose.) Frame O is at rest with respect to the earth, while O' is fixed to a starship coasting at constant speed $v_{starship}$ along the positive x axis, according to observers on earth.

Though identical clocks on earth and on the starship were manufactured to tick once per second, earth observers will see the starship's clocks ticking slowly. When an earth clock measures a time interval Δt, a starship clock will measure a shorter interval $\Delta t'$ with

$$\Delta t' = \Delta t\sqrt{1-\frac{v_{starship}^2}{c^2}}\ .$$

[3] https://en.wikipedia.org/wiki/Srinivasa_Ramanujan
[4] http://mathworld.wolfram.com/PiFormulas.html
[5] cool word, isn't it?

Imagine that the starship launches a shuttle craft in the positive *x* direction, moving with speed *u* with respect to the starship. Naturally, earth observers will see the shuttle moving faster than *u* since the starship is already moving in the *x* direction. We use the relativistic velocity addition formula to calculate the shuttle's speed as seen by observers on earth:

$$v_{shuttle} = \frac{u + v_{starship}}{1 + \frac{u v_{starship}}{c^2}}.$$

It is easy to show that in the limit that $v_{starship}$ and *u* approach *c*, $v_{shuttle}$ also approaches (but does not exceed) *c*. Note that the ***exact*** value of the speed of light is $c = 299{,}792{,}458$ m/s.

Let's use the time dilation and velocity addition formulas to analyze the motion of a starship undergoing the uniform *proper* acceleration $g = 9.81$ m/s^2. By "proper acceleration" I mean the acceleration sensed by someone on the starship. Naturally, an observer on earth will see that the starship never exceeds the speed of light, so its acceleration (according to earth observes) will approach zero.

We *could* work up an analytic description of $x_{starship}(t)$ and $v_{starship}(t)$ by doing some integrals. But why not just break the flight time up into small time intervals and sum the changes in position and velocity in a loop? That's the kind of thing computers are good at.

We'd like to answer the following questions, assuming the ship starts at rest near the earth. After a certain amount of time in space, how far has the ship gone, according to earth observers? How fast is it moving? How much time has passed on the earth and starship clocks?

The key is to keep in mind that during ten second of ship's time (corresponding to a longer time interval, according to an earth observer), the ship's speed has increased by 98.1 m/s according to a cosmonaut on the starship. As a result, earth observers will see that the ship's velocity has increased from $v_{starship}$ to

$$\frac{98.1 + v_{starship}}{1 + \frac{98.1 v_{starship}}{c^2}}.$$

During ten seconds of ship's time, earth observers will see that the ship has moved a distance that is approximately equal to the product of the ship's speed at the ***start*** of the interval and the ***duration*** of the interval, *according to earth clocks*:

$$\Delta x \approx v_{starship} \frac{10 \text{ seconds}}{\sqrt{1 - \frac{v_{starship}^2}{c^2}}}.$$

(I am using the approximation that the time dilation factor is constant during the short time interval.)

Please do the following. Consider a one-way voyage of four year's duration as measured by a clock on the starship. (Take the length of a year to be exactly 365.25 days, where one day is

24 hours long.) By breaking the outbound voyage into ten second intervals (***as measured by the starship clock***), write a program that calculates how far the starship has traveled (according to earth observers), how fast it is moving (according to earth observers), and how much a clock on the earth has advanced, at the end of the voyage. You will do this by looping over elapsed time intervals, each of ten second's ship time duration.

Your code should update the position of the ship, then the reading on the earth clock, then the velocity as seen by earth during each pass through the loop. To keep track of what your program is doing, have it print out regular updates of the ship's position and velocity (as measured in the earth frame), as well as the elapsed time in both frames.

Here's what I mean. If at some particular time the ship is traveling at 0.6 *c* according to earth observers, then 10 seconds of ship time will correspond to 12.5 seconds of earth time. In 12.5 seconds of earth time the ship will move approximately 12.5 × 0.6 × 299,792,458 meters. At the end of the interval the ship's new velocity will be

$$\frac{98.1+0.6c}{1+\frac{(98.1)(0.6c)}{c^2}}.$$

Keep in mind that the ship's calendar includes a leap year, so that there are 1461 days, or 126,230,400 seconds of ship's time in the four-year voyage.

For your information: my version of the program yields these answers:

```
final ship time (weeks) 208.71428571428572
final earth time (weeks) 1571.41469651
final ship time (seconds) 126230400.0
final earth time (seconds) 950391608.447
final ship speed (% c) 99.94834273779534
final ship distance (lightyears)  29.1632735122
```

If your code is correct it will agree with mine to splendidly impressive precision. If your code isn't correct, then dig into it with the iPython debugger to look for problems.

For full credit, determine (and print) the final earth time along with final ship speed and time for a four year journey as shown above.