Physics 40 Lab Manual

Michael Mulhearn

September 9, 2021

Contents

Chapter 1

Installation of Scientific Python

1.1 Introduction

In this lab, you will install the software which we will be using in phy40. This is an assignment, and will be graded. You should submit a text file containing a log of all the steps you took to install the software on your computer. Make this log as specific as possible, an entry might be:

```
Downloaded windows installer from:
 https://repo.anaconda.com/miniconda/Miniconda3-latest-Windows-x86_64.exe
```

Keeping this log will also make it easier for you to get help if you have problems.

If you run into problems, do some research on a web search tool (Google, for example) to become better informed and to see if you can overcome the problem on your own before asking for help. This is an important technique in getting help with technical problems that will serve you well even outside of this class. You will find it more easy to get useful technical help, from the sort of people most capable of offering it, when it is clear from your question that you are informed and have already tried all of the obvious things. If you are still stuck after trying to solve the problem for yourself, then contact your TA or instructor with specific technical details about what is failing, and include your installation log.

If you do find a problem with these instructions or manage to overcome a technical problem yourself, make sure to note it in your log and inform your TA, in case it is helpful for other students.

1.2 Installing Miniconda3

We will be using Miniconda3 based on Python 3.7 for data analysis using Jupyter notebooks. Miniconda is a lightweight package which we can use to install all of the remaining analysis software we will need in a consistent manner across all different operating systems.

Determine which OS type and version you have on the desktop or laptop computer that you will be using for your coursework. The software here will work under Windows, Linux, or macOS. It should also work on all Chromebooks released since 2019, and some earlier Chromebooks. You should also check whether you have a 32-bit or 64-bit OS (you can find instructions for how to determine this for your particular OS version with a Google search.) Most desktop or laptop computers built in the last ten years are 64-bit.

If you are using Linux or macOS, then from within a terminal type:

```
echo $SHELL
```

to determine the shell you are using (typically "bash" these days). Record all of this information in your installation log file.

Once you have determined your OS type and version, follow the instructions below approprate to your operating system.

1.2.1 Installing under Windows

If you have already installed a version of conda (e.g. Anaconda or Miniconda) then you do not need to re-install it. Instead, find the the Anaconda Prompt in the Application menu and run it.

If you need to install Miniconda3, then download and run the appropriate installer from:

`https://docs.conda.io/en/latest/miniconda.html#`

If prompted, you should choose to:

- Accept the license / terms of use.
- Install for just the current user, not all users.

Once installed, check that you can run the "Anaconda Prompt". From the prompt, check that you can run:

```
conda --version
```

and note the output in your installation log. Then proceed to Section 1.3.

1.2.2 Installing on a Chromebook

You will need to activate Linux on your Chromebook, according to the instructions here:

`https://www.codecademy.com/articles/programming-locally-on-chromebook`

Then follow the insturctions for installing under Linux. If your Chromebook predates 2019 and does not support Linux, contact your instructor for alternative arrangements.

1.2.3 Installing Miniconda3 under Linux or macOS

If you believe you already already have a version of conda installed (such as miniconda or ananconda) , check by running

```
conda --version
```

If you see something like:

```
conda 4.9.2
```

as output (even if the version is different) then you do indeed already have conda installed, with the base environment activated, and you can skip ahead to Section 1.3. If instead you get a message like:

```
conda: command not found
```

then the easiest solution is to simply proceed with these instructions.

To install Miniconda, download the appropriate installer for your OS here:

`https://docs.conda.io/en/latest/miniconda.html\#`

For macOS, you can choose between a "package" or "bash" version. I find it easier to follow the bash version, but the package version will work too. I recommend you make the following choices if prompted:

- Accept the license / terms of use.
- Do not install for all users, but just one the current user.
- Do allow the installer to issue "conda init".

During the installation, take note of the install location in your log.

After installation with these settings, conda will automatically activate the "base" conda environment. If this annoys you, as it does me, or interferes with other software you are using, you can turn off this agressive behavior with:

```
conda config --set auto_activate_base false
```

Confirm that you have successfully installed conda by typing

```
conda --version
```

Record the output in your installation log, and proceed to Section 1.3.

1.3 Installing the Physics 40 Conda Environment

Make sure your conda is fully up to date with:

```
conda update conda
```

Then follow the prompts, e.g. selecting "y" as needed to update any out-of-date packages.

We'll be using a conda environment specifically for phy40 to avoid conflicts with any other projects on your computer, and to ensure that we all have the same software installed. To create our environment:

```
conda create -n phy40 python=3.9 numpy scipy matplotlib ipython jupyter
```

language=csh

1.4 Starting a Jupyter notebook

This course will make extensive use of the Jupyter Notebook interface to Scientific Python, which is well suited to academic work (including independent research) because it combines code with output in digestable chunks. Even when the end product is a polished peice of software, much of the initial code development can be done in the interactive session that Jupyter Notebooks provide.

To activate the phy40 environment type:

```
conda activate phy40
```

When you are done with Phy 40 for the day you can deactivate this environment (later) with:

```
conda deactivate
```

Launch jupyter notebook with:

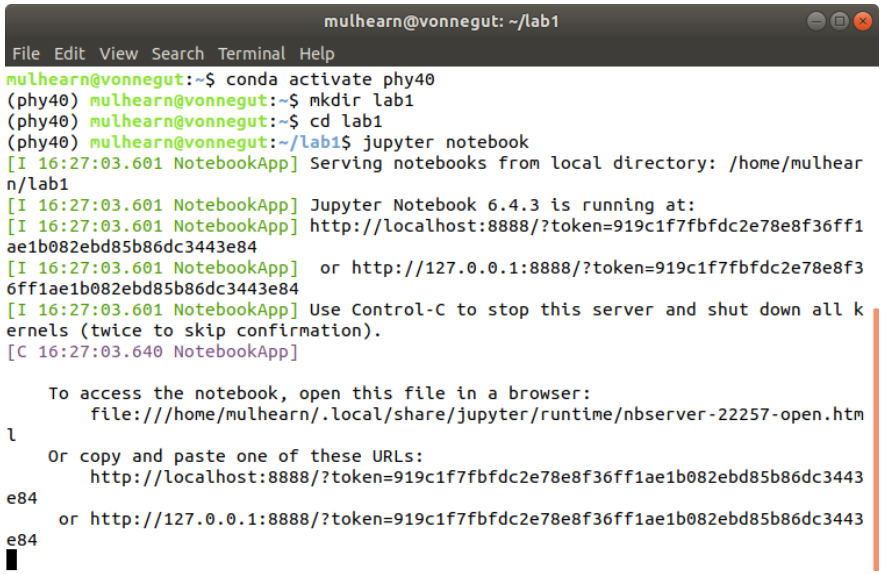

Figure 1.1: Example starting Jupyter Notebook from the Linux command line. In Windows, you will need to open the Anaconda Prompt instead of a terminal.

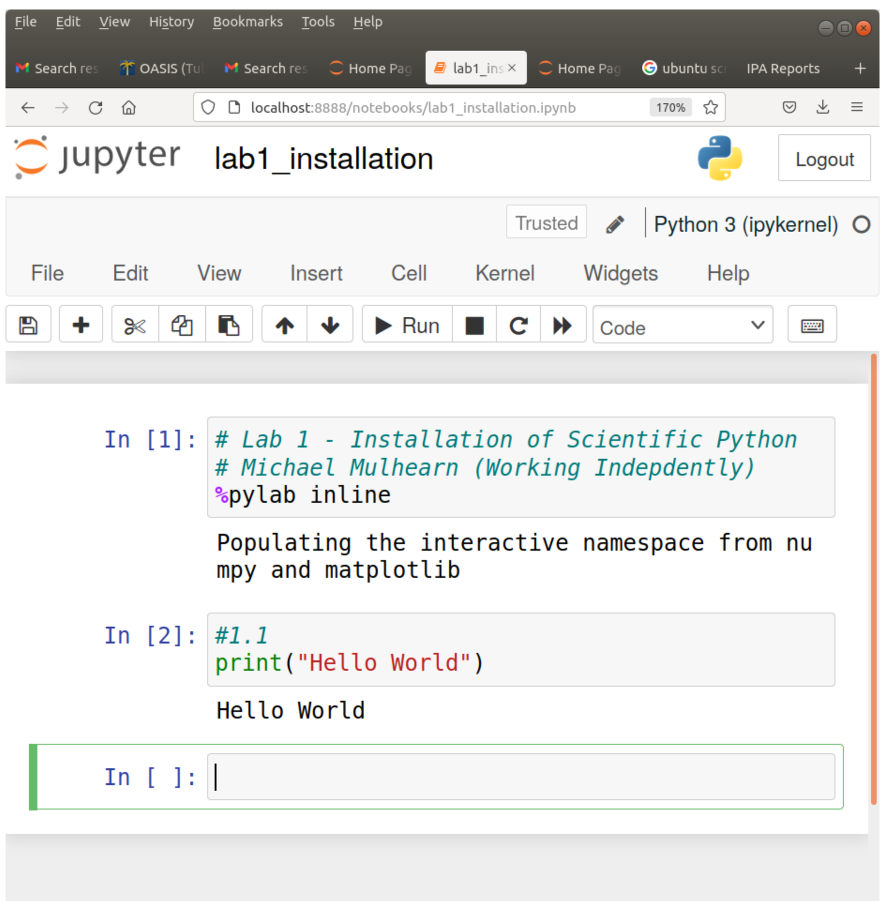

Figure 1.2: The Hello World example Jupyter Notebook.

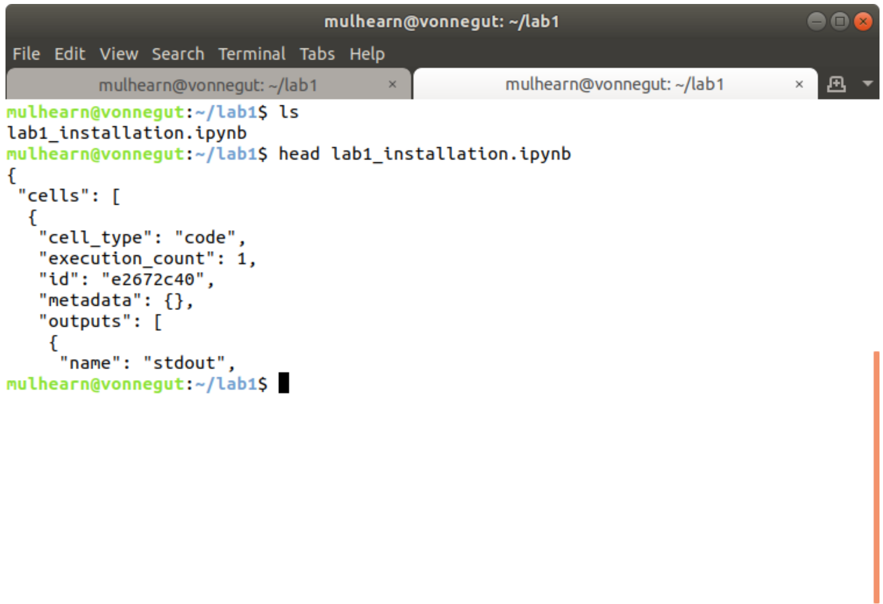

Figure 1.3: Example showing the saved Jupyter notebook. Notice that notebook file (ipynb) is not human readable on its own: it requires the Jupyter software to render it in a human readable form.

```
jupyter notebook
```

This should start the Jupyter Notebook server and open a client in your web browser. An example starting a Jupyter Notebook from Linux is shown in Fig. 1.1.

You should create one Jupyter Notebook per lab assignment, by choosing the New (Python 3) option in your client. Change the name of your notebook to something that clearly identifies the lab. Start each lab with comments (starting with "#" symbol) indicating the title of the lab, then your name followed by your lab partners. See the first cell of Fig. 1.2 for an example. This first cell is also a good place to issue the ipython "magic function":

```
%pylab inline
```

which will setup the notebook for inline plots and load the numpy and matplotlib libraries for you.

Each assignment will consist of a number of steps, clearly numbered like this one, your first step:

△ **Jupyter Notebook Exercise 1.1:** Print "hello world" using the python print command.

To keep your notebook clear, label cells (such as this one) with a comment for the assignment step number, as in the second cell of Fig. 1.2. You only need to label one cell if the assignment is fullfilled across several cells.

Jupyter Notebook checkpoints your work automatically. You should be able to see your notebook saved in the working directory where you started, as in Fig. 1.3. Notice that while the notebook file is ASCII text, it is not a human readable format. The Jupyter software is needed to render the notebook in a human readable way. To make your grader's life easier, you will be submitting PDF versions of your notebook, once all of the tasks are completed and the output is visible. There are several ways to make a PDF file from your notebook, but the most reliable is to use the "Print Preview" option to view the notebook as a PDF file within your browser, then use the print feature of your browser to print the page as a PDF file. Try this now, and make sure you can create a legible PDF file, but do not submit it to the course site, as you still have more to do. Always keep your python notebook file (ipynb) even after you submit the assignment. If you have problems, you can reproduce a PDF file from the notebook file, but it is tedious to reproduce your notebook from PDF. If you have problems producing the PDF file, you can submit the “ipynb” file as a temporary work-around, but work with your TA to sort out the problem as quickly as possible.

1.5 Submitting your assignment

Before submitting, take some time to clean up your assignments to remove anything superfluous and place the exercises in the correct order. You can also add comments as needed to make your work clear. You can use the Cell → All Output → Clear and Cell → Run All commands to make sure that all your output is up to date with the cell source.

When you are satisfied with your work, print the PDF file as described earlier and submit it to the course website.

Chapter 2

Binary Numbers

2.1 Introduction

At the heart of numerical analysis, naturally, you will find numbers. In this lab, we will explore the basic data types in Python, with particular emphasis on the computer representation of integers and real numbers. All modern programming languages do an admirable job of hiding the limitations of the computer representations of these mathematical concepts. In this chapter, we will deliberately explore their limitations.

2.2 Preparation

This lab will rely on the material from Section 1.2.2 of the Scientific Python Lecture notes.

△ **Jupyter Notebook Exercise 2.1:** Enter the following code into a cell and check the output:

```
a = 121
print(type(a))
print(a)
```

Next, in the same cell, add a line at the end setting a to a real value, `a = 1.34`, and print the type and value again. Check the output. Next, set a to Boolean value, `a = False`, and print the type and value yet again.

In many languages, such as C and C++, variables are strictly typed: you would have to decide at the start whether you want a to be of integer, float, or boolean type, and then you would not be able to change to a different type later. Python variables are references to objects, which means they only point to memory locations that contain objects with all of the data and functionality associated with that object. When you write $a = 121$ it is interpreted as "set variable a to point to a location in memory that contains a class of type integer with the value 121".

△ **Jupyter Notebook Exercise 2.2:** Enter the following code into a cell and check the output:

```
a = 12
b = a
b = 5
print(a)
print(b)
```

Why is the output "12, 5" instead of "5, 5"? When you write `b = a`, the variable b points to the same Integer class that a points to. So when you write "b = 5" why doesn't the value of a change as well? This code snippet shows that it does not. The reason is that Integers are *immutable* objects in python... their values cannot be changed. So when you write `b = 5` it is interpreted as "variable b points to a (new) Integer with value 5." The variable a continues to point to the Integer with value 12. The "is" operator used like this:

```
print(a is b)
```

tells you if a references the same object as b. Add two calls to this function to clarify the situation. One call should return True and the other False.

△ **Jupyter Notebook Exercise 2.3:** If I set a variable a to an integer value and I set b to the same integer value, do a and b refer to the same object, or to two different objects with the same value? Write a snippet of code (three lines) to find out.

2.3 Binary Representation of Integers

Computer hardware is based on digital logic: the electrical voltage of a signal is either high or low, which correspond to a mathematical zero or one. A digital clock is used to ensure that signals are only sampled at particular times, when they are guaranteed not to be in transition from a zero to one or vice versa. The Arithmetic-Logic Unit (ALU) uses digital logic gates (such as AND or OR) to perform calculations. For example, it is possible to build an adder that uses only NAND gates.

Because digital signals have only two states (zero and one), the most natural way to represet numbers in a computer is using the base two, which we call binary. In the familiar base ten, we have ten digits (from 0 to 9) and the place value increases by a factor of 10 with each digit moving toward the left. In binary, we have only two digits (0 and 1) and the place value increase by factors of two. A single digit in binary is referred to as a "bit". You add columns quickly when counting in binary: zero(0), one(1), two(10), three(11), four(100), five(101), and so on. For efficient operations, computers often group eight bits together to form a byte. Digital values are therefore commonly represented in hexidecimal (base 16) where two digits of hexidecimal describes one byte. See Table 2.1, which you can produce for yourself in Python like this:

```
print("dec: hex: bin:")
for d in range(16):
    print("{0:<2d}   0x{0:01x}   0b{0:04b}".format(d))
```

It is conventional to preprend binary numbers with "0b" and hexadecimal with "0x" otherwise we wouldn't know whether a "10" represents ten, sixteen, or two! Notice that the largest number that can be written with n bits is $2^n - 1$.

The mathematical notion of an integer can be naturally implemented by computer hardware. Although integers are represented in binary in the hardware, modern compilers and languages generally print them to screen as decimal by default. One caveat is that computers do not have an unlimited number of bits. Many computer languages use 64-bit integers, which means that only the integer values from 0 to 18446744073709551615 can be represented:

```
x = 2**64-1
print(x)
```

Table 2.1: The numbers 0 to 15 in decimal, hexadecimal, and binary.

dec:	hex:	bin:	dec:	hex:	bin:
0	0x0	0b0000	8	0x8	0b1000
1	0x1	0b0001	9	0x9	0b1001
2	0x2	0b0010	10	0xa	0b1010
3	0x3	0b0011	11	0xb	0b1011
4	0x4	0b0100	12	0xc	0b1100
5	0x5	0b0101	13	0xd	0b1101
6	0x6	0b0110	14	0xe	0b1110
7	0x7	0b0111	15	0xf	0b1111

For signed integers, one bit is used to indicate the sign (positive or negative) and so a 64-bit signed integer can represent integer values from -9223372036854775808 to 9223372036854775807. As long as an integer value is within the range covered by the integer type, the integer value can be perfectly represented.

Python uses arbitrary sized integers: it simply adds more bits as needed to represent any number. For an extremely large number, you will eventually reach practical limitations on the amount of memory and processing time available in the computer, which will limit how large of an integer can be calculated.

△ **Jupyter Notebook Exercise 2.4:** See for yourself just how huge integers can be in python by entering:

```
x = 2**8000
print(x)
```

and checking the output.

△ **Jupyter Notebook Exercise 2.5:** Print the integer 64206 in decimal, hexadecimal, and binary. Hint: just reuse the carefully formatted print statement from the example above.

△ **Jupyter Notebook Exercise 2.6:** Suppose you are tasked with rewriting the firmware for a distant satelite which has just lost one line from an eight-bit digital communications bus due to radiation damage. You now have only seven working bits! What is the maximum sized unsigned integer which you could write on this degraded seven-bit bus? What range of signed integers could you write?

△ **Jupyter Notebook Exercise 2.7:** Let's consider a four-bit signed integer, so zero is 0000 and one is 0001. Suppose the upper bit is reserved for sign, so 1XXX is a negative number. An obvious choice for representing -1 would be 1001, but there is a better choice. Consider that:

$$(-1) + 1 = 0$$

Well, if we simply ignore the last carry bit (5th bit):

$$1111 + 1 = 10000 = 0000$$

So if we define 1111 as -1, we can treat addition with negative numbers exactly the same as adding ordinary numbers. Find the representation for -2 such that

$$-2 + 2 = 10000 = 0000$$

then show that:

$$-1 + -1 = -2.$$

Python does not use this trick, but many other languages do.

2.4 Binary Representation of Real Numbers

Representing real numbers presents much more of challenge. There are an uncountably infinite number of real numbers between any two distinct rational numbers, but a computer has only finite memory and therefore a finite number of states. It is impossible for computers to exactly represent every real number. Instead, computers represent real numbers with an approximate floating point representation much like we use for scientific notation,

$$x = m \times B^n$$

where the significand m is a real number with a finite number of significant figures and the exponent n is an integer. The base B is ten for scientific notation but typically two in a floating point representation. The exponent n is typically chosen so that there is only one digit before the decimal point in the base B, e.g. 3.173×10^{-8} for scientific notation.

The limited precision of the discriminant can lead to challenges when using floating point numbers. The floating point precision is specified by the parameter ϵ (epsilon) which is the difference between one and the next highest number larger than one that can be represented. For scientific notation with four significant digits, $\epsilon = 0.001$, because we cannot represent anything between 1.000×10^0 and 1.001×10^0 with only four significant figures.

△ **Jupyter Notebook Exercise 2.8:** Determine the Python floating point ϵ by running

```
import sys
print(sys.float_info.epsilon)
```

△ **Jupyter Notebook Exercise 2.9:** Determine the Python floating point ϵ for yourself by running:

```
eps = 1.0
while eps + 1 > 1:
    eps = eps / 2
eps = eps * 2
```

Here the while loop continues running the indented code until the condition $\epsilon + 1 = \epsilon$ is met.

△ **Jupyter Notebook Exercise 2.10:** Python uses the IEEE 754 double-precision floating-point format. This format uses 64-bits overall, with 52 bits reserved for the significant. The standard uses a clever trick to save one bit, by requiring that the leading bit of the significant m is one, and defining 53 signficant figures using 52 bits:

$$m = 1.m_1 m_2 m_3 ... m_{52}$$

Here each m_i represents an individual bit (0 or 1). Predict the parameter ϵ and compare with the above.

△ **Jupyter Notebook Exercise 2.11:** It seems like ϵ should be small enough to simply ignore it in most cases, but in fact it shows up quite clearly if you apply strict equality to floating point quanties. To see the problem, run this code, checking if $\sin(\pi)$ is zero:

```
x = np.sin(np.pi)
print(x)
print(x==0)
```

Devise an alternative condition to `x==0` that properly accounts for floating point precision.

△ **Jupyter Notebook Exercise 2.12:** Here is another case where floating point precision joins the chat uninvited:

```
x = 0.1
y = x+x+x
print(y == 0.3)
```

Devise an alternative to `y==0.3` that properly accounts for floating point precision.

△ **Jupyter Notebook Exercise 2.13:** Personally, I prefer my zero's to look like zero. When floating point limitations are making them look non-zero, I like to clean then up with rounding, like this:

```
x = np.sin(2*np.pi)
print(x)
x = np.around(x,15)
print(x)
```

Yeck! What is -0??!! IEEE 754 defines two zeros -0 and 0. -0 is used to indicate that 0 was reached by rounding a negative number. This is so that $1/-0$ can be interpreted as $-\infty$ and $1/0$ as $+\infty$. If you just want to make this go away, add "+0":

```
x = np.around(x,15)+0
```

Show that this works.

△ **Jupyter Notebook Exercise 2.14:** Consider the following code:

```
a = 5;
b = 1.2343E-17;
sum = 0
sum += 5;
for i in range(1000000):
    sum = sum + b
print(sum)
```

Is there any problem here? If there is, fix it by changing only the *order* of the lines of code.

2.5 Other Data Types

Integers and floating point numbers are the real work horses of computational physics. We'll add numpy arrays in a future lab. This section will briefly introduce the remaining types.

Python includes strings as a basic type:

```
s = "hello world"
s = s + " (it's been a strange few years)"
print(s)
print(type(s))
print(s[6],s[4],s[6])
print(type(s[0]))
```

Strings are *immutable* objects that contain textual data. If you have used other languages, you might expect `s[0]` to be a "character" but in Python it is a string of length one. There is no built-in character type.

Python includes complex numbers as a basic type:

```
z = 1 + 2j
print(type(z))
print(z.real)
print(z.imag)
print(z.conjugate())
```

This is our first example of an object oriented programming (OOP) class interface. To compute the complex conjugate of z, an ordinary function would need to be passed z as an argument, or else it would not know which complex value to use for the computation. But `z.conjugate()` is a *method* of the class complex. The method is tied to the instance of complex number z by the "." and has access to all of the data it needs from z. Similarly, the `real` and `imag` are member data of class complex: they are the integers that contain the real and imaginary parts of z. Objects play a central role in Python, but in a refeshingly understated and reserved manner. It is enough for now to understand that `z.conjugate()` is much like a function that already has z as a parameter, and `z.imag` and `r.real` are just ways to access the data contained in z.

△ **Jupyter Notebook Exercise 2.15:** Define a complex number $1+i$ and multiply it by it's complex number. Show that the resulting complex number has zero for it's imaginary part.

Python provides Lists as a native container of python objects. We'll make much more use of numpy arrays, which are better suited to numerical analysis, but Python Lists occassionally play a role for various bookkeeping tasks:

```
L = ["hello", 1, 2, 3+2j, 3.45, "green"]
print(L[0])
print(L[1])
print(L[5])
L[0] = "goodbye"
print(L)
```

Here the List L contains a variety of (admittedly rather useless) objects. These objects can be referred to individually by their index. One place where lists really shine is in looping over a custom list of values:

```
for i in [1,5,10,50,100]:
    print(i)
```

△ **Jupyter Notebook Exercise 2.16:** Run the following code:

```
a = [1,2,3,4,5]
b = a
b[0] = 1
print(a[0])
print(a is b)
```

(Spits out coffee) "What the???!!" Lists are *mutable* which makes them fundamentally different from *immutable* integers. Here we assign b to point to the same object as a (a list) and then change an entry in that list. Even after the change, a and b point to the same object. This is only possible because the list object is mutable.

△ **Jupyter Notebook Exercise 2.17:** It's good to read the documentation, but it's a useful skill to figure things out for yourself too! Without looking up the documentation, write a snippet of code to determine for yourself if complex numbers are mutable (like lists) or immutable (like integers). (It's OK if your code throws an error here, but you can also comment it out if you are the sort of person that can't possibly leave it alone)

Chapter 3

Sequences and Series

3.1 Introduction

In this lab, we will apply for loops to study sequences and series. If you already have programming experience, you can complete the challenge problem in lieu of the other problems.

3.2 Preparation

This lab will rely on the material from Sections 1.2.1 to 1.2.4 of the Scientific Python Lecture notes. Most of the problems can be completed using a simple functions containg a single for loop, such as in this function:

```
  def loop(n):
    for i in range(n):
        print(i)
```

To run the code in the function, you call function, usually in a different cell:

```
loop(5)
```

△ **Jupyter Notebook Exercise 3.1:** Create a new function:

```
def powers(a,n):
   # your code here ...
```

that prints the first n powers of a. For example `mult(3,4)` should output:

```
1
3
6
9
```

In future problems, we'll describe this output simply as 1, 3, 6, 9. We won't be picky about whitespace unless we discuss it explicitly. One way to complete this is to use the three options of `range(start,stop,step)`.

3.3 Fibonacci Sequence

The Fibonacci numbers are a sequence of numbers satisfying the recursion relationship:

$$F_{n+2} = F_n + F_{n+1}$$

with $F_0 = 0$ and $F_1 = 1$. The sequence is:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

This sequence can be generated numerically by an algorithm such as this one:

```
fa := 0 # set fa to 0
fb := 1 # set fb to 1
repeat n times:
    fc := fa + fb
    print fc to screen
    fa := fb
    fb := fc
```

Note that this is not python syntax. What is the importance of the last two lines? Would the algorithm work if we exchanged their order?

△ **Jupyter Notebook Exercise 3.2:** Use the algorithm described above to implement a new function `fib(n)` which prints the next n Fibbonacci numbers after the initial 0 and 1.

3.4 Arithmetic Series

The finite arithmetic series

$$S_n = \sum_{k=0}^{n} (a + kd) = a + (a + d) + (a + 2d) + \ldots + (a + nd)$$

sums to the average of the first and last terms times the number of terms:

$$S_n = (n + 1)\frac{a + (a + nd)}{2} \tag{3.1}$$

We will assume $a = d = 1$ and calculate this finite series numerically using the following function:

```python
def arith(n):
    sum = 0
    for j in range(1,n+1):
        sum = sum + j
        #print("j: ", j, "\t sum: ", sum)
    return sum
```

Type in this function and see how it works by uncommenting the print statement (delete the # symbol that starts a comment) and calling it as `arith(5)`. The use of print statements in a loop like this or at each stage of a calculation is a simple, effective and classic debugging technique. You test your code with the print statements included, keeping n small so you don't fill your whole screen with output. Once your code is working, you comment out the unneeded print statements so that the interpreter ignores them and you no longer see the unneeded output. Why not just delete them? You can, but experience shows that if you do, you will need the line again shortly!

△ **Jupyter Notebook Exercise 3.3:** Obtain the sum of the first n terms of arithmetic series with `sum = arith(n)` for three different values of n. Each time, show that sum returned by the function matches the expected sum.

3.5 Geometric Series

The geometric series

$$\sum_{k=0}^{\infty} ar^k = a + ar + ar^2 + ar^3 + \dots$$

converges for $|r| < 1$ to:

$$\frac{a}{1-r}. \tag{3.2}$$

We will demonstrate this numerically.

△ **Jupyter Notebook Exercise 3.4:** Implement a function `geom(a,r,n)` which calculates sum of the first n terms of the geometric series with kth term ar^k. Show that it agrees with Eqn. 3.2 for $a = 2$, $r = 0.5$ $n = 100$.

△ **Jupyter Notebook Exercise 3.5:** Call you geometric series function again for $a = 3$, $r = 0.8$ and $n = 100$. Compare with the expected output calculate within python and with pencil and paper. Do they agree exactly? If not, do they agree within the floating point precision?

△ **Jupyter Notebook Exercise 3.6:** Now compare your calculated sum with Eqn. 3.2 for $a = 1$, $r = -0.9$ $n = 100$. How is the agreement? Increase n and see what happens. Why do you suppose this series is slower to converge?

3.6 Refinements

There are a few refinements you can make to your code. Don't change your working code from previous examples! Instead, copy the previous version to a new cell and make your refinements there. You don't even need to change the name of the function, Python will happily overwrite the old function implementation when it reaches the cell with the new version. Make these code improvements:

△ **Jupyter Notebook Exercise 3.7:** (Optional) Improve your Fibbonacci function so that prints the first n numbers including the initital two numbers "0" and "1". Make sure it works properly for $n = 0$, $n = 1$, $n = 2$, and so on.

△ **Jupyter Notebook Exercise 3.8:** (Optional) Extend the Arithmetic series function to include parameters a and d. Show that it works.

3.7 Fibonacci Integer Right Triangles

Starting with the number 5, every second Fibonacci number is the length of the hypotenuse of a right triangle with integer sides. The first two are:

$$5^2 = 3^2 + 4^2$$

and

$$13^2 = 5^2 + 12^2.$$

Furthermore, from the second triangle onward, the middle side is the sum of the lengths of the sides of the previous triangle, for example:

$$12 = 3 + 4 + 5.$$

△ **Jupyter Notebook Exercise 3.9:** (Optional Challenge) Use numerical methods to explicitly verify these properties for the first n Fibonacci integer right triangles.

If you would prefer, you may submit the Optional Challenge problem plus the problems from Section 3.5 to complete the assignment.