Statistics, Data Analysis, and Machine Learning for Physicists
Timothy Brandt
Spring 2020

Lecture 0

This initial lecture will be more of a tutorial of python and jupyter notebooks, with a bit of an introduction to computer programming in general and python in particular. If you have taken computer science courses before, you probably know most or all of this already. The title of these notes is, in part, a pun on indexing in python. **If this material is largely new to you, you should spend some time doing programming tutorials.**

This course will require you to use a computer to do numerical analyses. Computers have improved enormously in the past several decades, but their basic properties remain the same. There is memory to store information and a processing unit to perform calculations (most fundamentally just gates like AND, OR, and XOR). Processors have gotten faster over the years, but recently they have stopped getting faster (in the sense of the clock speed) and instead gotten cheaper, smaller, and less power-hungry (there are also a number of more subtle improvements). Modern computers have processors about as fast as computers from 5 years ago, but they have more of those processors. While this is a bit of an oversimplification, it has made the field of parallel computing very topical. We will not cover that at all in this course. Some parallelization is built into the routines you will use. Parallelizing your own code is tricky, and parallelizing it efficiently for use on supercomputers is very hard.

There are still a few things about computers that you should know if you would like to perform data analysis and computations. Computers have both processors and memory, and they have several different kinds of memory. The cheapest and most abundant form is hard disk space. Unsurprisingly, this form of memory is also the slowest to access for reading and writing. Much faster than hard disk space is RAM, or random access memory. RAM is also much more expensive. As of spring 2020, 128 GB of RAM costs something like $1000, while a 128 GB flash drive (flash memory) costs around $20, and a 5 TB spinning hard drive costs around $100, or about $2.50 per 128 GB. When you load a file off of the hard disk, you typically transfer its data to RAM (which allows for much more efficient access and modification later). When you allocate memory in a computer program, you typically allocate it from RAM. There are also levels beyond RAM: the cache, even more expensive, less abundant, faster, and closer to the CPU, and finally the register on the CPU itself. Part of code optimization consists of maximizing the efficiency of transferring data to the fastest form of memory.

In addition to the computer itself, we need a set of instructions, or a computer program. This is written in a computer language. The most basic language is machine code, in which you write instructions that the hardware can interpret directly. Such code isn't very readable to humans, so it was a revolution of sorts when the first compiled languages came out. The compiler was invented by Grace Murray Hopper, who insisted that it would be easier for a few people to write a program to translate English commands to machine code than to teach many people to write machine code. So she did exactly that.

In astronomy at least, C, C++, and Fortran are relatively common compiled languages. With these languages, you still explicitly do all of the memory allocation and passing, but you use English words for your commands and have a compiler translate all of it into machine code. A relatively new development is the rise of interpreted languages like python, matlab, and IDL, which do not get *compiled before* the code is run, but rather *interpreted while* the code is run. This prevents the compiler from rearranging things and performing all manner of optimization magic, so interpreted languages tend to be slow. They make up for this with readability and flexibility, and by calling compiled routines and modules for the computationally expensive bits. In this way, a language like python can give you the best of both worlds, and python is my very strong suggestion for your language in this class.

The fact that things like memory allocation are abstracted away from you in python has both good and bad aspects. It frees you from having to write extra software, but at the expense of your knowledge and understanding of what the code is doing. On balance this is necessary and a good thing–you can worry about higher-level problems! However, a basic knowledge of what is going on behind the interpreted language will remain useful. Sometimes it really is necessary to write something in a compiled language. Other times, it will help you understand why some approaches are faster than others and what that strange error message `segmentation fault` actually means.

The rest of these notes will assume that you are using python, and that you have installed the anaconda python distribution, python version 3.7. Unlike for, e.g., Mathematica and matlab, all of these resources are free and open-source.

Python

Fundamental to any computer programming language are a number of aspects, including

1. Data types and structures
2. A library of functions
3. Control structures
4. Good coding practices

We'll briefly review them here. This will be a very abbreviated review. **If any of this is new to you, please consider going through one or more python tutorials.**

Data types and structures
Computer languages all have a few basic data types, including integers, floating point values, strings, etc. I will not review those here, except to make a single point because it has come up many times. A floating point number may be written in either decimal or scientific notation. However,
`x = 3e20`
and
`x = 3*10**20`
are not the same. The first one assigns the value 3×10^{20} to the variable `x`—which is actually be stored in binary memory as roughly

$$3 \times 10^{20} \approx \left(1 + \frac{1}{64} + \frac{1}{2048} + \frac{1}{4096}\right) \times 2^{68}.$$

The second statement takes the integer 10, raises it to the 20^{th} power (which requires a type conversion since a standard integer can only store values up to either $2^{32} - 1$ or $2^{64} - 1$), and then multiplies this by 3 and assigns the value to `x`. You should always, always, always write 3×10^{20} (or any large or small floating point number) the first way, as `3e20`. Similarly, Planck's constant in cgs is `6.626e-27`, not `6.626*10**-27`.

In our course, we will commonly use a data type called a `numpy ndarray`. These arrays can be used to represent vectors, lists of numbers, matrices, or images. If you import `numpy` at the beginning of your program with the statement

```
import numpy as np
```

you can create a 1-D array of 10 zeros with

```
x = np.zeros(10)
```

or a 2-D array of 10 zeros with

```
x = np.zeros((10, 10))
```

You can then do operations on these arrays, for example

```
y = x**2 + 5
```

An operation on an array is almost always faster than writing an explicit loop and operating on each element of the array.

There are all kinds of things you can do with indexing on arrays. For example, you can take the 50^{th} element of an array:
`y = x[49]`
(the first element is indexed by zero). You can take every third element starting with the second one:
`y = x[1::3]`
You can take the array in reverse order:
`y = x[::-1]`
and many more. Try a tutorial if you haven't seen these before; it's pretty important for effective use of `numpy`.

Libraries of Functions
Nobody wants to write every function from scratch–you want to write `sin(x)` rather than implementing a function for sine that is accurate to double precision floating point. You probably also don't want to write your own interfaces to the standard input and output. In any modern programming language, all of these things are done in standard libraries.

In python, our libraries of functions are typically found within things called *modules*. A few of these have become very standard and will be used throughout this course: `numpy`, `scipy`, and `matplotlib`. You will almost always import these three in the first few lines of your program. These three modules include all of the plotting routines you will need, all of the optimization, and all of the linear algebra. You can read through tutorials on these, but you will get a little bit of that tutorial just from completing this course.

Control structures, and a note about interpreted languages
Any modern programming language will use statements like `for`, `if`, `else`, `while`, etc. You can read up on these in a python tutorial, but they function the same as they do in any language. One important feature of an interpreted language like python is that you should avoid writing explicit loops as much as possible. The reason is that they can be slow. Take the following example. Suppose I want to generate a million random numbers and then compute the sine of each one. I can write

```
import numpy as np
n = 1000000
x = np.random.rand(n)
y = np.sin(x)
```

or I can write

```
import numpy as np
n = 1000000
x = np.random.rand(n)
y = np.zeros(n)
for i in range(n):
    y[i] = np.sin(x[i])
```

The first way will be much, much faster than the second way. The reason is that in the first case, the actual computation of sine is being done with a single function call to a routine that is written in C (that fact is hidden from you), and the loop itself is in the compiled language. In the second case, the program makes a

million separate calls to the same C routine and assigns the result to the array within python before making the next call. On my laptop the first method (without the explicit `for` loop) is *around 400 times faster* than the second method. This gives you a sense for the speed penalty you can pay with an interpreted language like python. In most cases, however, there is no need to pay much of a speed penalty for the extra readability and ease of coding in a language like python. If the speed of execution matters, then you want the main computations to be done with reasonably large `numpy` arrays. This is usually easy to do (as it is in this case). Sometimes, however, it may not be possible. In that case you have a few choices. You can switch entirely to a compiled language, you can accept the speed penalty, or you can use something like `cython` or `numba`. `Cython` lets you write code that looks mostly like python but that gets compiled into C code that you can call from python. Basically, it's a way of letting you write your own routine (kind of like `numpy`'s sine) with a more limited amount of pain. I use `cython` with some frequency in my own work. `Numba` is even easier. The following simple example will both show the simplest use of `numba` and introduce how a function is written in python:

```
import numpy as np
from numba import jit

def fastsine(x):
    return np.sin(x)

@jit
def slowsine(x, n):
    y = np.zeros(n)
    for i in range(n):
        y[i] = x[i]
    return y
```

Without the `@jit` decorator, this is equivalent to what I wrote earlier, and `fastsine` is about 400 times faster than `slowsine` for a million points. The `@jit` decorator enables some compiling of the `slowsine` function. It doesn't bring its performance all the way up to `fastsine`, but it does improve it by about a factor of 60. What had been 400 times slower is now less than 10 times slower. The use of `cython` and `numba` are beyond the scope of this course, but it is good to know that they exist and to have an idea of how and when they can help you.

Good Coding Practice
Physicists are horrible programmers. It is a stereotype that isn't quite as accurate as it used to be. To be fair, there are a lot of bad programmers, to the point where the problem is an xkcd strip:
`https://xkcd.com/2030/`
Being a bad programmer generally means writing code that is hard to follow and, somewhat relatedly, likely to have errors. All programmers make errors (bugs), but clear and clean code typically has fewer of them and is much easier to debug.

Whenever you write a program, you should make it readable for someone who initially has no idea what you are doing. There is a very good reason for this. Chances are good that **you** will come back to this code months later and need to fix something or want to reuse something. If your code is unintelligible, you will have to rewrite it from scratch. And debugging unintelligible code is a nightmare. So go find a reference on coding standards, flip through it, and try to follow it as best you can. For python, keep the following general guidelines in mind:

- **Use comments, but don't overuse them.** Make the lines of code themselves as clear and intuitive as possible. In a Jupyter notebook, as we'll see in a bit, comments are special.
- **Think before you type.** Ask yourself (and google) whether there is a nice, clear function in `numpy` or `scipy` that does what you need. Often, the answer is yes. An example: we want the 75^{th} percentile of an array `x`:

```
xsorted = np.sort(x)
y = xsorted[int(0.75*len(x))]
```

vs

```
y = scipy.stats.scoreatpercentile(x, 75)
```

The first way takes a little bit of sleuthing to figure out what is going on; maybe it uses a comment to make it clearer. The second line needs no comment, and should not have a comment (unless you need to explain *why* you are taking the 75% percentile).

- **Choose sensible and consistent names for variables.** There is a balance here. Maybe `x25` is a poor choice, for example–it means nothing. Something like `effective_temp_on_stellar_surface` is nice and descriptive, but try reading the Planck function with a lot of similarly named variables in it. Maybe `teff_star` strikes a balance in your case.
- **Fewer lines are better.** There is no prize for the longest program. The principles of coding are the same as those for scientific and practical writing: you should strive to express yourself as clearly and concisely as possible.
- **If you have to choose between being clear and concise, be clear.**
- **Only optimize your code for performance where necessary.** Optimizing sometimes means sacrificing clarity or readability. Only do this if you care how long your code takes to run. More philosophically, optimizing code takes time, and your time is the most valuable resource of all.

There are lot more guidelines in any set of coding standards, but please keep these in mind. It will make it much easier when I grade your work, and it will make it much easier for you to re-use bits and pieces of those assignments in your own work.

Jupyter Notebooks

Jupyter is a relatively new platform for data science that provides a convenient way to share code. You will e-mail me your assignments as jupyter notebooks. We will use jupyter exclusively in this class for a few reasons:

- **I don't have to compile or run your code.** Every machine's environment is different; my version of python or of a particular package may not match yours. I do not want to run your code on my computer.
- **You can easily share your work, including with future-you.** Make it a pdf, an html, and you won't have to re-run your code and worry about packages breaking either! Just take the pieces you want to re-use.
- **It looks nice.** I don't want to read through plain text code and separately flip through figures. I want to see them naturally, together.

You should be installing the anaconda python distribution, version 3.7. You can then type

```
jupyter notebook
```

into a command prompt to launch the notebook server. Start a new notebook, save them, etc. all from within your browser. For a demonstration, we'll start a new notebook that will be posted to the gauchospace page after the class. This will include markdown cells with LaTeX formatting. A well-structured juptyer notebook can be almost indistinguishable from the outline of a paper, or a section of a paper.