Introduction to Computer Programming for the Physical Sciences
Timothy Brandt

# Chapter 1

This course will introduce you to the use of a computer to do numerical analyses. You'll almost certainly need to do some computing in the future, whether in physics class, in lab, in some other class, or in a job. My goal for this class is that you are comfortable solving equations, doing basic computations, making plots, and fitting data on a computer.

Computers have improved enormously in recent years. This course isn't about computers themselves, though, so I won't discuss anything about they actually work. The only thing I would like you to know about computers for now is that they take a specific set of instructions and execute them exactly. Computers don't think, so an error is almost always the fault of the programmer. This is both frustrating and liberating. It means that all of the responsibility is on you, but also that you can fix anything that goes wrong. Part of this class will also try to instill good habits that will make it easier to avoid mistakes (also called bugs), and easier to find and correct them.

A computer language is a set of instructions that a computer can understand. In the bad old days, programmers would have to write in machine code, which the computer 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, and we are forever in her debt.

There are many computer languages, and I am guessing that you will have heard of some of them (C, C++, Fortran, Matlab, Python, Java, BASIC, etc.). This course will use the Python language. Python is a relatively recent language under continual development. It is an interpreted language, which means that the computer reads and executes one line at a time. This is different from a compiled language (like C, C++, Fortran, etc.) where a compiler first translates your entire program into machine code, and then you run it. Python is fairly easy to start using, certainly easier (in my opinion) than something like C or Fortran. We will be using Python in conjunction with Jupyter notebooks. This is another recent development that allows you to combine comments, notes in both regular text and Latex (a way of formatting nice math expressions that physicists often use), code, and figures all in one nice package.

For this class, I will urge you to use either the Anaconda Python distribution or Google Colab. Both are free. Anaconda Python will make many things easier, but it does require you to install (free) software. If you cannot install it on your machine, you will be able to do everything in this class with Google Colab. For that, you need only a web browser and an internet connection.

At this stage, you may or may not have written a line of code before. If you haven't, you will do so shortly at your first section. Here, I will introduce a bit of vocabulary to have around as a quick reference. This is far from complete but should introduce a lot of the terminology that I will use. If I say a word and you don't know what it means, please ask. Many of these words have the same meaning that they do in math.

## A Line of Code

A line of code, usually synonymous with a **statement**, tells the computer to do something. You can also include lines or notes that you want a human to read, but you do not want a computer to execute. These are called **comments**. Comments in Python are anything that comes after the `#` symbol. The computer will ignore this symbol and anything that comes after it (until the next line). For example,

```
x = 5 # Set the variable x equal to the number five
```

means exactly the same thing to a computer as

`x = 5`
The line
`# The code below does the integral`
is completely ignored.

## Selected Terminology

**variable**: some text (like a word) that stores a value. The text must start with a letter or underscore.
**comment**: text to help a human understand code but which the computer ignores
**function**: a set of instructions to take some inputs (called **argument(s)**), do something, and produce an output
**loop**: a set of instructions that is executed many times
**conditional**: a set of instructions that may or may not be executed (think `if`/`else`)
**bug**: a mistake, usually one that results in an incorrect answer, unexpected behavior, a crash, etc.
**module**: a Python package that includes functions, quantities, all sorts of things. We'll use the modules `numpy`, `scipy`, and `matplotlib` in this course.
**floating point**: a real number (not necessarily an integer) that the computer stores in scientific notation
**lambda**: this is a special Python thing for defining a function in one line—you cannot name a wavelength variable `lambda`!

## Basic arithmetic

The simplest thing you might do a computer is arithmetic, like you would do on a handheld calculator. Just like for a calculator, you have basic operations like `+`, `-`, `*`, `/`, and `**` (the last one means to raise to a power). For example, to compute

$$y = x^2 + 5x + 3(z - 1) \tag{1}$$

you could write
`y = x**2 + 5*x + 3*(z - 1)`
This will store the result of the right-hand-side of the equation to the variable `y`. Future lines of code in your program can do things with the variable `y`. For example, later on, you could write `q = y**2`. Remember that Python will follow the standard order of operations! A great many bugs result from following an order of operations that you did not expect.

### A quick note about scientific notation

We write numbers in scientific notation as, e.g.,

$$c = 3 \times 10^8 \tag{2}$$

(in SI units). You may well need the value of $c$ for your calculation. You could write
`c = 3*10**8`
However, you should not do this. Instead, please write
`c = 3e8`
These are interpreted very differently by the computer. The first one takes the integer 10, raises it to the $8^{\text{th}}$ power, and then multiplies this by 3 and assigns the value to `c`. The second one assigns the value of $3 \times 10^8$ directly to the variable `c`. The second way is faster, easier to read, and less vulnerable to bugs, as you will see or have seen in your first section. For a quick look at one reason why you should write scientific notation as, e.g., `3e8`, try the following:

```
print(1/3e8**2)
print(1/3*10**8**2)
```

Which of the lines above gives the correct answer to $1/c^2$? Can you see why given the order of operations?

Also, as a quick note, there is a largest number and a smallest number (i.e. closest to zero) that a computer can represent because of details of how it stores numbers in memory. Any number larger than about $10^{308}$ will be stored as infinity (and will generate a warning called overflow), while a number closer to zero than $10^{-308}$ will be stored as zero (and generate an underflow warning). If you get one of these warnings, it's because some part of your program gave a result that was beyond one of these numbers. The reason why is that a computer has a specific way of representing a floating point number, as

$$\pm a \cdot 2^b \tag{3}$$

where there are only 64 bits (zeros or ones) to store the sign, the value of $a$ (between one and two), the sign of $b$, and the value of $b$. The largest $b$ that can be represented by 10 bits is $2^{10} - 1$, and $2^{2^{10}-1} \approx 10^{308}$.

## 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 and math routines you will need, along with much, much more. We will take a closer look at these later.

## 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. 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.
- **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 for us to 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 submit your assignments electronically as Jupyter notebooks (or as `pdf`s). We will use Jupyter exclusively in this class for a few reasons:

- **We 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.** Rather than having to read through plain text code and separately flip through figures, we can see them naturally, together.

You should be installing the anaconda Python distribution, version 3.10. You can then type

```
jupyter notebook
```

into a command prompt to launch the notebook server. You can also launch the notebook server from an app. Or, if you cannot get Jupyter installed on your system, you can use the same thing online through Google Colab. A well-structured Juptyer notebook can be almost indistinguishable from the outline of a paper, or a section of a paper.

## A Final Note

Programming can seem intimidating and scary. But you will all get the hang of it. Just remember the most important thing: **make your ideas, and your intentions, clear in everything that you write.** Computers almost never make mistakes, but they will blindly follow incorrect instructions. When you make your code and your thought process clear, you are helping yourself more than anyone else. The person who will debug your code, and who might use it later and have to understand it all over again, is you.

Introduction to Computer Programming for the Physical Sciences
Timothy Brandt

# Chapter 2

## Functions

Last section, we introduced programming, Python, and Jupyter notebooks. You should have all opened a notebook and written a few lines of code. Now, we will introduce a few concepts that you will need to do things that would be difficult, or impossible, on a calculator. The first thing that we will introduce is a **function**. A function in math is a set of instructions for taking input(s) and mapping them to output(s). In order to be a function, the input must uniquely determine the output. In a program, the word function means the same thing: it takes one or more inputs (called **arguments**), does something, and produces one or more outputs. As an example,

```
y = numpy.sin(0.5)
```

will assign $\sin(0.5)$ to the variable $y$. Sine is a function included in the `numpy` module, which we will spend more time talking about later. For now, I'll be using functions from `numpy`.

You will both use built-in functions and write your own. Writing a function means giving the full set of instructions for how to get an output from an input. It will also use two Python commands: `def`, which tells Python that you are defining something, and `return`, which tells Python what the function should return as its output. As an example, we can define a function to take the square of a number:

$f(x) = x^2$.

This function in Python looks like the following:

```
def f(x):
    return x**2
```

You can see the use of `def` to define the function, and the use of `return` to show that the function should return the square of the argument. The line defining the function also ends with a colon. Finally, notice how the line below the function definition is indented (you indent with four spaces, or with tabs). Indenting in Python is very important: it is how Python knows that the following lines are part of this function. Once you want to write a line that is *not* part of the function, it should not be indented inside the function. For example, the following:

```
def f(x):
    return x**2
print(f(5))
```

will print `25` to the screen ($f(5)$). The `print` statement is not indented inside the function, so it is not part of the function. What if I write

```
def f(x):
    return x**2
    print(f(5))
```

What do you think will happen?

The `print` statement is now inside the function, but it comes after the `return` statement. Python never actually does anything for this `print` statement.

Now, I could probably have done a better job with my function definition: seeing `f(x)` later in my code is probably not going to be helpful. Your functions can have any name you want, as long as it follows the same rules as variable names. How about a better name for my function:

```
def sqr(x):
    return x**2
```

That's better. Now, I can call `sqr(x)` later and understand what I am doing.

Let's do one more example. Write a function, call it `increment()`, that adds one to its argument and returns the result.

We might write

```
def increment(x):
    return x + 1
```

One more conceptual question that puzzled me for a while when I started to learn coding. I named the function `sqr`; that name is now reserved unless I redefine it later. What about `x`, which I used in the definition of the function (`def sqr(x)`)? The answer is that this only refers to `x` inside the function, and nowhere else in my program. I could have used any valid variable and it wouldn't matter. In fact, I could even use a variable that I had defined elsewhere. To see this, try

```
x = 5
def sqr(x):
    return x**2
print(x)
```

What do you think we will get? Try it out!

For last section, you debugged some code that attempted to calculate the Planck function. We could also define the Planck function, which would make things a lot easier if you wanted to re-use it. The Planck function depends on two variables, not one: temperature $T$ and frequency $\nu$ (or wavelength $\lambda$, but remember that `lambda` can't be a variable name!). So, I can define the Planck function as a function of two variables:

```
def Bnu(T, nu):
    c = 2.998e8
    kB = 1.381e-23
    h = 6.646e-34
    return 2*h*nu**3/c**2/(np.exp(h*nu/(kB*T)) - 1)
```

In this case, `c`, `kB`, and `h` are only available inside the function; you can't get them outside the function. You can then call

```
print(Bnu(500, 5e13))
```

or use it in another piece of your calculation.

If you look closely, you can see that there is a function used within this function: `exp()` is within `Bnu()`. This is completely fine. You can use any function within a module that you `import`, or any function that you have previously defined within this Jupyter notebook. You can even pass a function to another function.

Take the following example:

```
def func_sqr(f, x):
    return f(x)**2
```

I pass the function `f`, and within `func_sqr`, I call `f(x)`. This returns the square of the function `f`. For example, I could use this to compute $\sin^2 x$ as

```
sinsq_x = func_sqr(np.sin, x)
```

This is exactly equivalent to

```
sinsq_x = np.sin(x)**2
```

It is a direct replacement of `f` with `np.sin`. Both lines above are fine in Python, so this is a valid way to pass a function.

Now, **I cannot write** `(np.sin**2)(x)`. That requires Python to figure out what `np.sin**2` is, and it tries to square `np.sin`, to multiply the function by itself. But **Python can't multiply functions** (or indeed do any binary operation between functions). If you try to get, for example,

```
np.sin*np.cos
```

you'll get another error. Two functions could have completely different sets of arguments, so how can Python multiply them together? And they might return something other than a number, something that cannot be squared. Imagine, for example, a function of two variables (like the two-argument arctangent) and a function of one variable. What would it mean to multiply them together? What argument(s) would such a function take?

**In sum,** if you're not sure whether your passing of a function is legal, you can try the following test. Copy and paste the argument directly into the function call. For example:

```
val = func_sqr(np.sin**2, 5)
```

would end up calling something like

```
(np.sin**2)(5)
```

That's not the correct way of writing the line of code, so this will break. But if you write

```
val = func_sqr(np.sin, 5)
```

would end up calling something like

```
np.sin(5)
```

which is fine. If your code passes this search-and-replace test, it should work for passing the function.

## Conditionals

Often, when you write a program, you want to do something differently depending on some condition. As an example, suppose we want to define the absolute value function, `abs(x)`. This should return `x` if `x` $\geq 0$, and `-x` otherwise. To do this, we can use Python's `if` statement. An example will show how it works:

```
if 1 > 0:
    print("One is greater than zero.")
else:
    print("Zero is greater than one.")
```

The message inside quotation marks is called a string, and we can use it to print a message. We won't use strings much in this class. Other than that, there are a couple of things to notice. Right after the `if` statement comes something that is either True or False. That statement ends with a colon. Then, the next line is indented: it is only executed if the `if` statement is True. Below this is an `else` statement, indented at the same level as `if`. It also gets a colon. Then, below the else statement, is some code that is only executed if the `if` statement turns out to be False.

Now let's use `if/else` to write an absolute value function, called `abs(x)`.

```
def abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

When you use `if/else`, you will pass the `if` statement something that is either True or False (it is **boolean**). We have seen a few examples of boolean statements, for example

```
1 > 0
```

is True (you can actually set something to True in Python using `True`). Here are a two more examples of True and False statements:

```
0 >= 1     # This is False

0 == 0     # This is True
```

The first one asks if $0 \geq 1$. The second looks a little unusual. It's not a typo. Remember that a single equal sign *assigns a value to a variable*. So we can't write `0 = 0`: this would attempt to assign the value zero to the variable `0`. The double equal sign *tests* for equality.

Now let's apply the ideas of functions and conditionals to the sinc function, which comes up a lot in Fourier transforms and signal processing. The sinc function is defined as

$$\text{sinc}(x) = \begin{cases} 1 & x = 0 \\ \frac{\sin x}{x} & \text{otherwise} \end{cases} \tag{1}$$

It is available in `numpy`, but let's see how we could write it using sine. You'll need to handle the case of $x = 0$ with an `if` statement and a test for equality:

```
def sinc(x):
    if x == 0:
        return 1
    else:
        return np.sin(x)/x
```

Finally, let's apply both of the ideas from this class to one more physics scenario. The electric potential a

distance $r$ from the center of a spherical shell of charge $Q$ and radius $R$ is

$$V(r) = \begin{cases} \frac{kQ}{r} & r > R \\ \frac{kQ}{R} & r \leq R \end{cases} \tag{2}$$

where $k$ is Coulomb's constant, $8.988 \times 10^9$ in SI units. The potential is constant inside the shell. How would you write the electric potential of the shell as a function of $Q$, $r$, and $R$?

```
def potential_sphere(Q, R, r):
    k = 8.988e9
    if R < r:
        return k*Q/R
    else:
        return k*Q/r
```

Note: you could write this more compactly using the function `max(a, b)`:

```
def potential_sphere(Q, R, r):
    k = 8.988e9
    return k*Q/max(R, r)
```

This is certainly shorter (i.e. fewer lines of code). But which do you think is clearer?

Introduction to Computer Programming for the Physical Sciences
Timothy Brandt

# Chapter 3

## Loops

In this class we will introduce loops. Loops are essential to a large fraction of programs. In physics, we have seen examples so far of single calculations. But sometimes we need to do something more than once. For example, if we want to solve an equation numerically, we usually start with a good guess and then make progressively better guesses. This requires a loop.

There are two basic ways of writing a loop in python: `for`, and `while`. We will start with a `for` loop to show you how that works. The `for` statement will start a line. We will then use two special constructions that you will get used to: `in`, and `range`. `range` makes a counter. For example, `range(5)` counts from zero up to (but not including) five: 0, 1, 2, 3, 4.

```
for i in range(5):
    print(i)
```

This loop takes `i` from zero to four and prints it. The variable `i` is a counter here, and you can use it in the loop. Just like for a function definition and the `if/else` statement, everything that belongs to the loop is indented.

We can use this loop to add up all of the numbers from zero to four, inclusive. To do this, we start with a value and then add to it at each step in the loop. I'll use `total` for the variable representing the total, and I'll start by setting it equal to zero:

```
total = 0
```

Now we will use a loop to add to the total:

```
for i in range(5):
    total = total + i
print(total)
```

And you get the answer, 10. You can empirically verify the formula

$$\sum_{n=1}^{N} n = \frac{N(N+1)}{2} \tag{1}$$

Within the loop, you add the counter `i` to the total at each step.

Now let's apply this to a famous series in math:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}. \tag{2}$$

First, let's print the answer:

```
print(np.pi**2/6)
```

Now let's use a loop in the same way as before. We will start at one rather than zero, so we will use, e.g., `range(1, 10)` to run from 1 to 9, inclusive:

```
total = 0
for i in range(1, 10):
    total = total + 1/i**2
    print(i, total)
```

So, the sum from 1 through 9 is about 1.54, compared to 1.645 for $\pi^2/6$.

Now let's see how `while` loops work, and use one to see how long it takes to get within 1% of the right answer. A `while` loop looks a little different. It starts with `while`, then a condition that could be either `True` or `False`, then a colon. The loop will stop as soon as the condition is `False`. For example:

```
total = 0
i = 0
while total < np.pi**2/6:
    i = i + 1
    total = total + 1/i**2
```

What do you think this will do?

This is an **infinite loop**. The sum will never actually be $\pi^2/6$; it will always be a little bit less. If you run this in a code cell it will keep going forever. Luckily, Jupyter notebooks have an Interrupt Kernel option (Kernel → Interrupt). Clicking that should end the infinite loop and let you fix it.

We don't want to run things until we get the exact answer, because when we iterate toward the right answer we will never actually get there. We want to keep going until we are *close*. There are two ways that we can define close:

1. Our answer is almost the same as the correct answer; or
2. The change we are making at a given step is very small.

In the example we have now, we know the correct answer ($\pi^2/6$), so we could use either approach. Other times we might not know the correct answer, so we would have to use Option 2. Let's see how we could apply both to fix our infinite loop. First, let's fix it using the first option, with our knowledge of the correct answer. The problem is the comparison statement: we want the comparison to be small, less than some **tolerance**. I'll set

```
tolerance = 1e-5
```

to require that I be within $10^{-5}$ of the right answer. My comparison statement should then compare my total with the right answer. I will want to use absolute values for the comparison: I could be within `tolerance` of the right answer either on the high side or the low side. The absolute error is then

```
error = np.abs(total - correct_value)
```

I'll give all of you a minute to rewrite the infinite loop so that it stops when the answer we have is within `tolerance` of the correct answer ...

```
tolerance = 1e-5
total = 0
i = 0
while np.abs(total - np.pi**2/6) > tolerance:
    i = i + 1
```

```
    total = total + 1/i**2
```

Now it will break. Notice how the loop keeps going while the error is larger than the tolerance. The sign of the comparison is important. I can see how many steps that took by printing out the value of `i` at the end, adding the one statement

```
print(i)
```

and I get 100001 (about $10^5$ steps). That's a lot!

In this case I compared the total with the right answer. I could instead decide to stop when the increment is small, i.e., close to zero. This is a comparison that is easiest to make inside the loop. I can do that by combining an `if` statement with our last topic for today, `break`. This statement looks like

```
if increment < tolerance:
    break
```

within a loop. This will break out of the loop (not the `if statement`). Let's see how we could add this to the loop above:

```
tolerance = 1e-5
total = 0
i = 0
while np.abs(total - np.pi**2/6) > tolerance:
    i = i + 1
    increment = 1/i**2
    total = total + increment
    if np.abs(increment) < tolerance:
        break
```

Let's unpack this, since it is the longest block of code we have yet seen in this class. The while statement works exactly the same as earlier, but there is another check within the loop. The loop will end when the total is within `tolerance` of the answer (as before), *or* when `increment` is smaller than `tolerance`. Let's see which one actually triggered the break, and what we got, using

```
print(i, increment, np.abs(total - np.pi**2/6)
```

We see this time that we broke at `i = 317`, much sooner than before. Our error on the total was about `3e-3`.

In practice, here's a suggestion for how to write your loops using `break` statements. We'll also use the `for` loop rather than the `while` loop. The following structure should be versatile enough to serve you in most applications. I'll first set the *maximum* number of steps or iterations that I will allow. I'll use 1000 for now:

```
max_iter = 1000
```

Now I can write this as a `for` loop, and use a `break` statement inside:

```
total = 0
tolerance = 1e-5
max_iter = 1000
for i in range(1, max_iter):
    increment = 1/i**2
    total = total + increment
    if np.abs(increment) < tolerance:
```

```
        break
```

We will use this basic construction later in this class to implement a Newton-Raphson solver to solve equations, and to perform numerical integrals. It should be sufficient to do most of the tasks you will need in undergraduate physics that require loops.

Let's have a bit more practice on this, again with a series (we'll apply it to other applications later in the class). Let's approximate the natural logarithm of 1.5 using the series

$$\ln(1+x) = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{x^n}{n} = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots \tag{3}$$

How would you approximate this series using a `for` loop, breaking when the increment is less than $10^{-5}$? Go ahead and write it yourself, using the template above for the other series that evaluates to $\pi^2/6$.

The first part will be the same. In fact, the *only* difference will be the increment:

```
x = 0.5   # we will approximate ln(1 + x) = ln(1.5)
total = 0
tolerance = 1e-5
max_iter = 1000
for i in range(1, max_iter):
    increment = (-1)**(i + 1)*x**i/i
    total = total + increment
    if np.abs(increment) < tolerance:
        break
```

If we print `i` at the end of the loop, we find that we broke after 13 iterations, and that our answer is within $\ln 1.5$ within about $3 \times 10^{-6}$.

I'll introduce one more small tweak. If the answer is very large, say $10^8$, it probably doesn't make sense to use a tolerance of $10^{-7}$. But a **fractional tolerance** of $10^{-7}$ (or a tolerance of 10) may make sense. We might say that

```
ftol = 1e-7
```

which we could use as

```
if np.abs(increment) <= ftol*np.abs(total):
    break
```

This approach is very common, and `ftol` is the most common variable name for this tolerance. I moved `total` to the right-hand-side in case it is zero, so that I am not dividing by zero. The approach can still fail. If the correct answer is very close zero, then as my guess gets better, my requirement to break the loop continues to get stricter and stricter, and I might never break. Using a fractional tolerance isn't foolproof, and works poorly if the answer is too close to zero. But then again almost nothing in numerics is foolproof. You can only do your best, write your code to maximize clarity and minimize mistakes, and learn what to look for.

To recap what we discussed this class, we introduced loops to compute something by successively better approximations until we are doing well enough. This basic approach describes numerical integration, solving equations, and lots of other applications that you can't just solve analytically. We introduced `for` and `while` loops, and then the `break` statement. If there is one template to burn into your memory, that will apply to as many physics situations as possible, I suggest the following:

```
for i in range(max_iter):
    do_something
    if condition:
        break
```

The `do_something` step could be to approximate an integral, to add a term to a series, to take a successive approximation to an equation, etc.

## An Aside: Recursion

I'll spend the last few minutes talking about recursive functions. You may have seen recursion relations in math. For example:

$$f(n) = \begin{cases} 1 & n = 1 \\ 3f(n-1) + 1 & n > 1 \end{cases} \tag{4}$$

We can achieve something similar in a function by calling itself. Let's implement the example above. We'll need to use an `if` statement to see if we are at $n = 0$:

```
def recursive_3np1(n):
    if n == 1:
        return 1
```

Now if we are not at $n = 1$, we implement the recursion relation by calling the function at `n - 1`:

```
def recursive_3np1(n):
    if n == 1:
        return 1
    else:
        return 3*recursive_3np1(n - 1) + 1
```

This will insert the function into itself many times. You do need to be a little careful though. What if you call `recursive_3np1` with an argument of 0.5? It will become an infinite loop. You can implement checks at the beginning of your code to guard against that, but that sort of defensive programming is beyond the scope of this class.

There's one particular example of a recursion relation that will be useful in the homework, and that is for the derivative. The $n$-th derivative may be defined recursively as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{5}$$

$$f^{(n)}(x) = \lim_{h \to 0} \frac{f^{(n-1)}(x+h) - f^{(n-1)}(x)}{h} \tag{6}$$

So, you could approximate the $n$-th derivative by writing a function for

$$f^{(n)}(x) = \begin{cases} \frac{f(x+h)-f(x)}{h} & n = 1 \\ \frac{f^{(n-1)}(x+h)-f^{(n-1)}(x)}{h} & n > 1 \end{cases} \tag{7}$$

for a function $f$, a value $x$, and some small $h$.

Introduction to Computer Programming for the Physical Sciences
Timothy Brandt

# Chapter 4

## Numpy, Arrays, and Indexing

Thus far, you have used `numpy` only for a couple of functions, like sine and the exponential. Here, we will introduce the module as a more powerful tool for a lot of what you will do in physics computing.

The core of `numpy`'s usefulness is because it can do things with arrays, not just numbers. An array is a new thing that we haven't used yet. It's like a matrix, with many elements, and indeed can function within `numpy` exactly as a matrix. A lot of numerical computing is actually linear algebra, so there is a huge amount of effort put into making linear algebra fast and elegant.

We'll start with an introduction to what an array is. Let's start with the `numpy` function `linspace`. You can use it with, for example,

```
x = np.linspace(x1, x2, n)
```

where `x1` and `x2` are the smallest and largest elements in the resulting array, and `n` is the number of elements. The command

```
x = np.linspace(0, 1, 10)
```

will produce an array of ten elements. The first will be zero, the last will be one, and there will be eight elements in between ($\frac{1}{9}$, $\frac{2}{9}$, etc.). You can show this with the statement

```
print(x)
```

The array will be printed enclosed in square brackets `[]`. I want to give you one other way of making an array here:

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

This will make an array of integers from 0 up to 9 (it's similar to `range`, but it makes an array rather than an iterator–don't worry too much about the difference). Once you have a `numpy` array, you can operate on every element in that array. For example,

```
y = x**2
```

This will square every element of the array `x` and save the result as an array called `y`.

All of the binary operations you have used so far, `+`, `-`, `*`, `/`, and `**`, will operate element-wise. Earlier, you wrote a function to square a number,

```
def sqr(x):
    return x**2
```

**This will work just as well on an array.** If you have this function definition, you can try the following:

```
print(sqr(5))
```

```
x = np.linspace(0, 3, 4)
print(sqr(x))
```

You can write a polynomial or an exponential function on an array just as well. For example, the line

```
y = x**3 + 5*x**2 - 7
```

could have `x` as an array or a number, and operate in the same way.

The above examples multiplied an array by a number. We can also multiply two arrays together, for example

```
z = x*y
```

This will multiply the two arrays `x` and `y` element by element. If `x` and `y` have different sizes (called **shapes** in `numpy`), we will get an error message:

```
x = np.arange(5)
y = np.arange(6)
z = x*y
```

will produce an error that the arrays have incompatible shapes.

The next topic is how you can get an individual part of an array. This general topic is called **indexing**. You can get the individual element `i` in an array using

```
x[i]
```

So, for example, you can set

```
x = np.arange(5)
```

An array is indexed from zero. So, you can get the first element of `x` by `x[0]`. Try it:

```
print(x[0])
```

There are a few fancy things you can do with indexing of `numpy` arrays. We'll explore a few of them in the homework and section, but we will introduce them here:

- **The first element** in an array `x` is `x[0]`
- **The second element** in `x` is `x[1]`
- **The last element** in `x` can be written as `x[n - 1]` (if `x` has `n` elements)
- **The last element** in `x` can *also* be written as `x[-1]`
- **The second-to-last element** in `x` can be written as `x[-2]`
- **A subarray** of elements, e.g., 1 through 5 (the second through sixth elements) is `x[1:6]`.
- **Every other element** in `x` (starting with the first one) can be written as `x[0::2]` or `x[::2]`.
- Similarly, **every third element** would be `x[::3]`.
- **The array in reverse order** would be `x[::-1]`.

Let's practice this a little bit, starting with

```
x = np.arange(5) + 2
```

Print the first element of this array, the last element, every other element, and the elements in reverse order.

One other thing to know about arrays: you can select elements from an array using a condition. For example, using

```
x = np.arange(5) + 2
```

there are five elements, the first element is 2, and the last element is 6. You can select all elements that are greater than 5 with

```
x[x > 5]
```

What actually happens is that `x > 5` it itself an array, but an array of `True` and `False`:

```
x = np.arange(5) + 2
print(x)
print(x > 5)
```

will return

```
[2 3 4 5 6]
[False False False False True]
```

The conditional will operate element-by-element, returning a boolean array (i.e. an array of `True` and `False`). Using this array as an index will extract only those elements where the boolean array is `True`. In this case, it will extract only the elements that are greater than 5, i.e., only the last element. We will end up with the single-element array `[6]`.

This has been a lot of indexing, so let's practice a little bit more. We'll start with an array of `x` values and `y = np.exp(x)`:

```
x = np.linspace(-5, 5, 100)
y = np.exp(x)
```

1. What is the first element of `y`?
2. What is the last element of `y`?
3. Set `z` equal to the elements of `y` that are greater than 2.
4. What is the first element of the array `z` that you defined above? Is it the smallest element of `z`?

One more set of functions that are handy if you want the largest or smallest element of an array, you can use `min` or `max`.

Now let's look at an example of this with the Planck function you have seen before in this course:

$$B_\nu(\nu, T) = \frac{2h\nu^3}{c^2} \frac{1}{\exp\left(\frac{h\nu}{k_\mathrm{B}T}\right) - 1} \tag{1}$$

The Planck function defines the photon energy radiated as a function of the photon's frequency $\nu$ for an absorbing ("black") body at temperature $T$. To derive it, you need to know quantum mechanics (indeed, it played an important role in the development of quantum mechanics). You should have written this function before, but for practice, we'll do it again:

```
def Bnu(T, nu):
    c = 2.998e8
    kB = 1.381e-23
    h = 6.646e-34
    return 2*h*nu**3/c**2/(np.exp(h*nu/(kB*T)) - 1)
```

In the past, we have used this as a function of two numbers $\nu$ and $T$. Now, we can consider it to be a function of arrays and/or numbers. For example, you can feed `Bnu` with a number for `T` and an array for `nu`, or a number for `nu` and an array for `T`. You could also give it arrays for both `nu` and `T`, but in that case, you would need them to have compatible shapes. Much more commonly, you would use this function with a number for one input and an array for the other.

Let's make a concrete example using the function `Bnu` defined above. Let's say we want the Planck function for a 500 K emitter at many frequencies. We can call `Bnu` with the single number 500 for `T`, but an array for `nu`. Let's first try `nu` going from 0 to 1000 Hz (we'll use 100 steps), and see what we get for `Bnu`:

```
nu = np.linspace(0, 1000, 100)
T = 500
Intensity = Bnu(T, nu)
```

Now we can print the first element of both `nu` and the first element of `Intensity`:

```
print(nu[0], Intensity[0])
```

and we get zero for both (you can check the equation for why). We can also print the last element:

```
print(nu[-1], Intensity[-1])
```

Now, what if we want to check whether the maximum intensity was somewhere in the range between 0 and 1000 Hz? How might I do that? Well, if the maximum intensity is at the endpoint, then it probably isn't in the middle, so I can check that:

```
print(max(Intensity))
```

and it turns out that I indeed get the intensity at the endpoint. The frequency of the peak intensity is from another formula, which you could derive by differentiating Equation (1) and setting the derivative equal to zero. But for now, let's try a different frequency range, from 0 to $10^{14}$ Hz, and try again (I'll use 1000 steps this time):

```
nu = np.linspace(0, 1e14, 1000)
T = 500
Intensity = Bnu(T, nu)
```

Now if I try

```
print(nu[-1], Intensity[-1])
print(max(Intensity))
```

I find that the maximum intensity is *not* at the endpoint; it must be somewhere in the middle.

For the next topic, let's look at how we could figure out which frequency in our list had the highest intensity. There are several ways of doing this. Take a few minutes to think of one possible way, and sketch out the code (you don't need to actually get it working; we're after the ideas for now).

Ok, a few ways. First, we can step through each intensity and see if that one is the max:

```
maxI = max(Intensity)
for i in range(len(nu)):
    if Intensity[i] == maxI:
        maxnu = nu[i]
        break
```

I used `len(nu)` to get the length of `nu`, the number of elements it has.

We can also step through intensity and see if that intensity is more than our current max (basically the same as above, but instead of using `break`, we :

```
maxI = Intensity[0]
for i in range(len(nu)):
    if Intensity[i] > maxI
        maxnu = nu[i]
        maxI = Intensity[i]
```

Finally, we can do this in one line:

```
maxnu = nu[Intensity == max(Intensity)]
```

In the line above,

```
Intensity == max(Intensity)
```

is an array that is `True` where `Intensity` is at its maximum, and `False` otherwise. Calling `Intensity` with this array of indices pulls out only those element(s) of `nu` where `Intensity` is maximized. In the section and the homework, we'll have a bit more practice on arrays and indexing.

Finally, I wanted to say a little bit about how a function sometimes doesn't work as straightforwardly with an array. Let's revisit the absolute value function, which we introduced in the last class with an `if` statement. One way of writing the absolute value would be

```
def my_abs(x):
    if x < 0:
        return -x:
    else:
        return x:
```

Earlier, I said that we could reuse a function like the Planck function, calling it with an array. We cannot use `my_abs` in that way. If we try to call `my_abs` with an array, we get an error. `x < 0` will be an array of `True` and `False`—which element of that array should control the `if`? Since `if` statements do not play nicely with arrays, we need an alternative. You have a few options:

We can assume that the input is an array, and write the absolute value function like

```
def my_abs_array(x):
    val = x*1
    val[x < 0] *= -1
    return val
```

This, however, will only work if `x` is an array. If you look closely, you'll notice that I had an extra line in the beginning, `val = x*1`, not `val = x`. The difference is subtle. When I write `val = x*1`, `val` is a new array that has the same values as `x`. However, if I write `val = x`, `val` points to the same array as `x`, and modifying `val` will also modify `x`. They are two different variables pointing to the same thing, rather than two different variables pointing to two different arrays (that happen to have the same values, initially).

If I call `my_abs_array` with just a value, e.g. `my_abs_array(-5)`, I will get an error. Making a function like this work with both floating point numbers and arrays is hard.

An alternative would be to use the earlier function `my_abs` within a loop:

```
y = x*0
for i in range(len(x)):
    y[i] = my_abs(x[i])
```

Introduction to Computer Programming for the Physical Sciences
Timothy Brandt

# Chapter 5

## Plotting, File Input and Output

In this chapter, we will get some practice using Python to make plots, read in data, and write out data. The reading, writing and saving will be a bit easier if you are using Anaconda on your own computer rather than Google Colab. It's possible either way, and I encourage you to work through the quirks of your configuration and operating system.

Let's start with plotting. We will use `matplotlib` throughout this, with the line

```
import matplotlib.pyplot as plt
```

This will give us enough flexibility to do a whole lot of plotting with just a few lines. For an initial example, let's plot

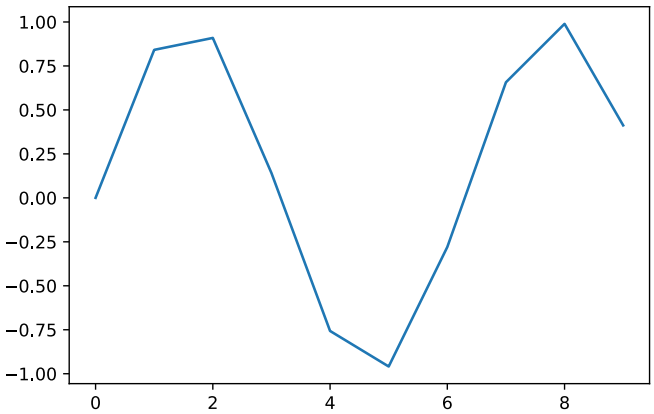
$y = \sin x$

Plotting on the computer means making dots at the locations of points, connecting points, etc. A graphing calculator does the same thing, but it usually chooses the points for you automatically. We'll need to be a little bit more explicit for Python (and for most other languages). For example, let's see what happens if we set

```
x = np.arange(10)
y = np.sin(x)
```

This sets $x = 0, 1, 2, \ldots, 9$ and $y = \sin 0, \sin 1, \sin 2, \ldots, \sin 9$. Let's see what this looks like with

```
plt.plot(x, y)
```




The plot doesn't look great. There are a few problems:

1. The curve is sampled coarsely, so it looks jagged.
2. The numbers on the axes are a bit small and hard to read.
3. The axes are unlabeled.

The second point, the one about the font sizes, can be fixed with a line at the beginning of the program where you can set your default parameters for plotting. For my tastes, most of the choices `matplotlib` makes are good, but there are some that I don't like. The biggest ones are the line width and font size. I usually put the following lines at the start of my program:

```
from matplotlib import rc
rc('font',**{'size':  16})
rc('lines', **{'linewidth':3.0})
rc('axes', **{'labelsize':16})
```

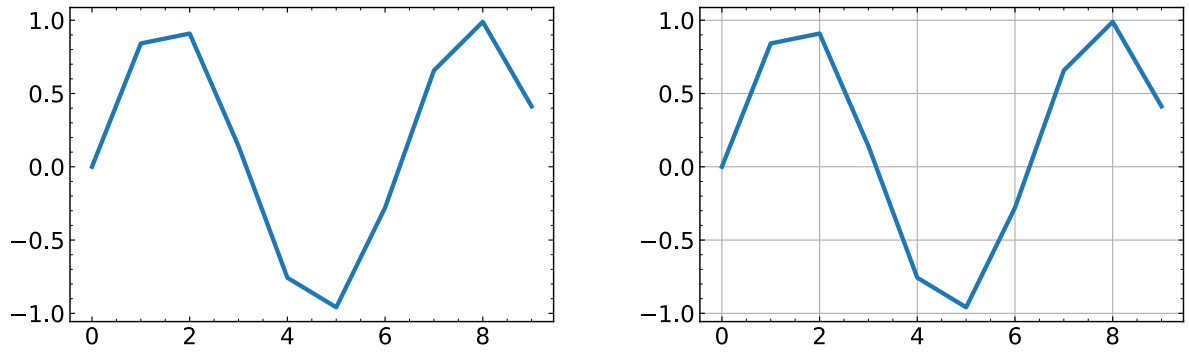
I also prefer my ticks on the inside of the plot, along with minor ticks. You can change this behavior with

```
rc('xtick', **{'direction':'in', 'top':True, 'minor.visible':True})
rc('ytick', **{'direction':'in', 'right':True, 'minor.visible':True})
```

If you want, you can also turn on grid lines with

```
plt.grid()
```

We can see what difference this makes (grid lines on the right):




For the plots above, the `rc` parameter tweaks came at the beginning of the notebook, while I drew the plots with nothing but

```
plt.plot(x, y)
```

for the plot on the left, or

```
plt.plot(x, y)
plt.grid()
```

for the plot on the right.

You can also put these lines into a `matplotlib` style file. You can add to this file as you like, in a plain text editor. To make the plot use the settings in this file without all of the `rc` lines, simply use something like

```
plt.style.use('mystylefile.mplstyle')
plt.plot(x, y)
```

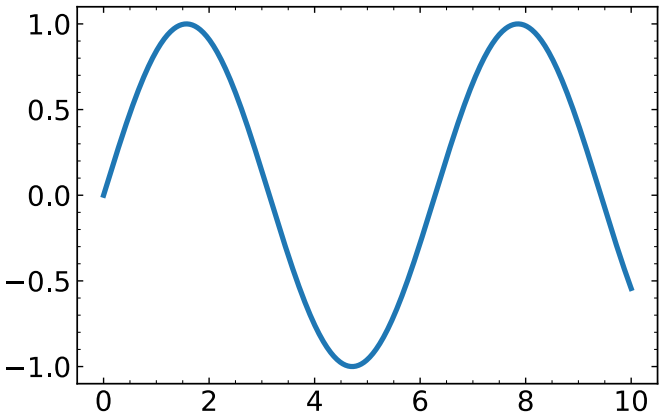
where the file `mystylefile.mplstyle` is a plain text file (you can give it any name you like, but I suggest keeping the file extension `mplstyle`). I often use one that simply looks like

```
font.size : 16
lines.linewidth : 3.0
savefig.facecolor : white
axes.labelsize : 16
xtick.direction : in
xtick.top : True
xtick.minor.visible : True
ytick.direction : in
ytick.right : True
ytick.minor.visible : True
```

Now we'll look at the other problems with the plots. First, there aren't enough points (we only plotted ten points). We can use more points using `np.linspace`; that will make the curve look a lot smoother. Also, if we want to plot just a few points (or if we indeed have only a few points, for example if they are measurements), we can plot points without connecting them.

Let's start with making the lines smooth. For this, we will want to define `x` differently, for example, with 1000 points:

```
x = np.linspace(0, 10, 1000)
y = np.sin(x)
plt.plot(x, y)
```



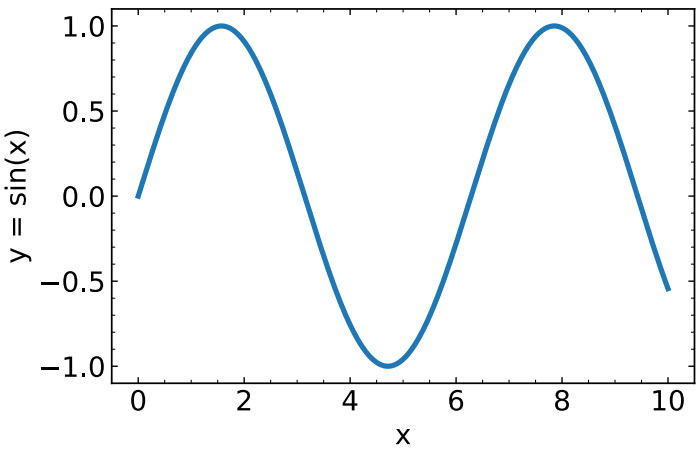

This looks nice. We still have to label our axes. We can do this with

```
plt.xlabel("x")
plt.ylabel("y = sin x")
```

Unfortunately, this ends up cutting off the axis label when I save the figure for these notes, so I do need one more line to make it size the figure appropriately:

```
plt.tight_layout()
```

We finally get the following plot:




Again, apart from the settings I use at the beginning, the commands to make this plot were

```
x = np.linspace(0, 10, 1000)
y = np.sin(x)
plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('y = sin(x)')
plt.tight_layout()
```

So it isn't too bad. And if you have a style file like the one included on the course page, all you need is one more line at the beginning:

```
plt.style.use('mystylefile.mplstyle')
```

If you want to save your figure, just add a single line at the end:

```
plt.savefig('name.pdf')
```

or

```
plt.savefig('name.png')
```

depending on whether you want png or pdf format (you would replace `name` with the file name you want to use). I do not suggest jpg for plots with lines and/or text: jpg is designed for photos and does not keep clean edges on figures. If you are using Google Colab, you'll probably need to mount your Google drive and save your figures there: it is a little bit more complicated than if you are using Anaconda Python installed on your computer.

Now, let's see how we can plot with points rather than lines. For this, we'll return to our sine curve with just ten points:

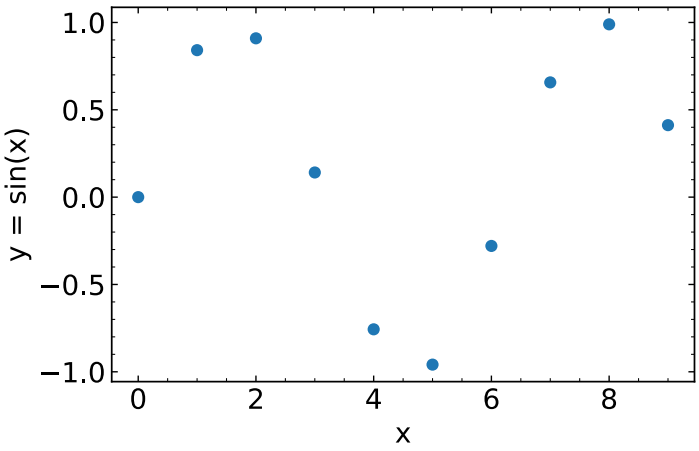
```
x = np.arange(10)
y = np.sin(x)
```

We certainly do not want to connect these with lines. To do better, we'll need to introduce two more arguments that you can give to `plt.plot()`: `marker` and `linestyle`. If we want to draw points instead of lines, we need two commands:

1. Draw points with `marker='o'` (or some other symbol; you can look at your choices, and you can choose the point size with `markersize`),
2. Do not draw lines with `linestyle=' '`

We'll see what that plot looks like:

```
plt.plot(x, y, marker='o', linestyle=' ')
plt.xlabel('x')
plt.ylabel('y = sin(x)')
```

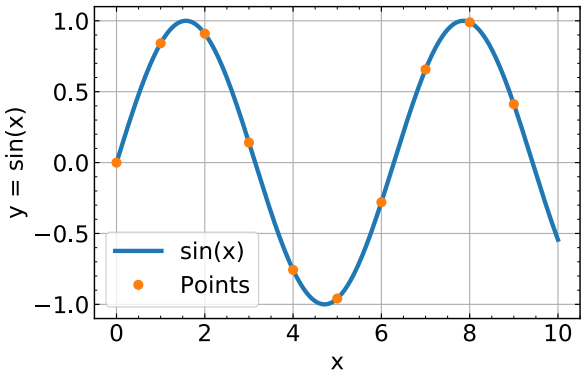



For the final part of today's exploration of plotting, we can also mix points and lines, and we will introduce labels and legends. The `label` will be one more keyword to `plt.plot()`, to be used if you call `plt.legend()` later. Just remember that `matplotlib` will draw lines in the order in which you give it the commands. So, each new curve will be on top of the old curves. I'll first plot $\sin x$ using a lot of points, and then plot ten points on top. When I call `plot`, `matplotlib` will add new things to the same figure (the same `axes` in `matplotlib` lingo) unless you explicitly clear the figure. I'll also add in a grid so you can see how that looks. I'll show the plot and code on a new page, along with a sort of plotting cheat sheet.

# Handy Plotting One Page "Cheat Sheet"

## Sample Code:

```
x = np.linspace(0, 10, 1000)
y = np.sin(x)
plt.plot(x, y, label='sin(x)')
x_pts = np.arange(10)
y_pts = np.sin(x_pts)
plt.plot(x_pts, y_pts, label='Points', linestyle=' ', marker='o')
plt.xlabel('x')
plt.ylabel('y = sin(x)')
plt.grid()
plt.legend()
```




## Itemized Guidelines

- You'll want to include something like the following at the start of your program (these reflect my own preferences), or import a style file.

  ```
  import matplotlib.pyplot as plt
  from matplotlib import rc
  rc('font',**{'size':16})
  rc('lines', **{'linewidth':3.0})
  rc('savefig', **{'facecolor':'white'})
  rc('axes', **{'labelsize':16})
  rc('xtick', **{'direction':'in', 'top':True, 'minor.visible':True})
  rc('ytick', **{'direction':'in', 'right':True, 'minor.visible':True})
  ```

  or, in place of the `rc` lines,

  ```
  plt.style.use('mystylefile.mplstyle')
  ```

- Your workhorse will then be `plt.plot()` (but see also `plt.semilogx()`, `plt.semilogy()`, and `plt.loglog()`).
- You must give points $x$ and $y$ to `plt.plot()`. You may also give it lots of other, optional arguments, and you can explore those on your own. A couple of the most important are `label`, `linestyle`, and `marker`. Other very common ones are `color`, `markersize`, and `linewidth`. To plot only points without lines, use `linewidth=' '` and give some other kind of argument to `marker` (you can check the options using Google).
- You can turn on faint gridlines with `plt.grid()`.
- You can label your $x$ and $y$ axes with `plt.xlabel('xname')` and `plt.ylabel('yname')`. If you don't like the default axis limits, you can also manually set them with `plt.xlim(xmin, xmax)` and `plt.ylim(ymin, ymax)` (putting in numbers for all arguments).
- You can save your figure with `plt.savefig('filename')`; I strongly recommend either png or pdf format. You might also find that you need to include `plt.tight_layout()` to make sure that you don't cut off axis labels.

Now that we can plot, we might want to plot data rather than just mathematical functions (even those that we use all of the time in physics). This discussion will assume that you have Anaconda python installed on your system. If you are using Google Colab, you will have a bit of a challenge in dealing with file input and output. I'll first show you how to load in data, and then how to save data.

The main function I suggest for reading in data is `np.genfromtxt()`. You need to give the function a filename, and this file needs to be in a place where Python knows where to find it. This might be one of the trickier parts, since giving paths to files can vary quite a bit depending on your operating system. **As a suggestion, the directory that Python saves your notebooks into will be the easiest directory from which to load in data files**.

Once you can give Python the path to your file, you can give that filename to `np.genfromtxt()`. It will look something like

```
np.genfromtxt('myfile.txt')
```

The most important parameters for you to know about are `delimiter` and `skip_header`. The `delimiter` value tells Python how values are separated. For example, if you are reading in a `.csv` file, you will want to use `delimiter=','` (note the quotes!). A plain text file might have a space as the delimiter, which is what `numpy` will assume if you don't tell it otherwise. The `skip_header` value tells Python how many lines to skip at the beginning of your file.

When you load in a text file using `np.genfromtxt`, it will show up as a `numpy` array. For example,

```
mydata = np.genfromtxt('myfile.txt')
```

will load up my data from the file `'myfile.txt'` into the array `mydata`. This array will have two dimensions: if you want the first column, for example, it would be

```
firstcol = mydata[:, 0]
```

The colon in the first index position means to take data from all of the rows; the zero in the second index position means to take only the first element of each row. If I wanted to take the first row of data, I would use

```
firstrow = mydata[0]
```

Saving your data to a file looks basically the same as loading it in; just use something like

```
np.savetxt('myfile.txt', mydata)
```

where `mydata` is the array of your data. We'll explore more about how to write data later. For now, hopefully this will help you to load in data that you are given, that you write yourself in a text editor, or that you write in a spreadsheet editor like Excel or Google Sheets.

Introduction to Computer Programming for the Physical Sciences
Timothy Brandt

## Lists and Arrays: What's the Difference?

In this class we haven't really used Python lists. We have used Python arrays. They look similar, but they are quite different under the hood. Take the following example:

```
x = [0, 1, 2, 3, 4]
print(x)
```

prints

```
[0, 1, 2, 3, 4]
```

And

```
x = np.arange(5)
print(x)
```

prints

```
[0 1 2 3 4]
```

It almost looks the same–there just aren't commas. But these two things are very different. A list is a collection of any set of things. For example,

```
x = [0, 'hello', np.sin]
```

is a list containing the number 0, the string `hello`, and the function `np.sin`. Lists can mix and match anything. You can add to them, remove from them, etc. (we haven't covered any of that). This flexibility comes at a price. Suppose that I want to square each element of a list. There is no guarantee that this is even allowed. One the elements of the list is `np.sin`, which I am not allowed to square. Also, if I add to the list, the computer will have to look for space in its memory to place the extra element. Lists are very complicated to manage under the hood.

Arrays look similar but have important differences from lists. They must follow much stricter rules. For example, when you make an array, you need to specify how big that array is, and every element of the array should have the same type (a floating point number, for example). This guarantees that, for example, squaring each element or taking the sine of each element is allowed. Arrays also live in blocks of memory, where each element is stored next to the previous element. This makes it much, much faster to operate on an array, because the computer can be very efficient under the hood thanks to the restrictions that arrays must follow.

To give an example, I'll make `x` as both a list and an array:

```
x = np.linspace(-5, 5, 1000000)
xlist = list(x)
```

Now, I will compute the sine of each element of `x`, first using `np.sin` on the array, and then by doing it separately on each element of the list (don't worry about understanding the syntax of the second way). I'll use `%timeit` to measure how long it takes each way:

```
%timeit y = np.sin(x)
```

```
6.89 ms ± 383 µs per loop
```

```
%timeit y = [np.sin(val) for val in xlist]
```

```
924 ms ± 11.6 ms per loop
```

The method with arrays was *almost 150 times faster*. In numerical work, you usually want to use arrays rather than lists, both because they behave nicely with functions and mathematical operations, and because they are usually much faster.

Introduction to Computer Programming for the Physical Sciences
Timothy Brandt

## Common Bugs and Error Messages

I'll try to summarize a few common bugs and errors here, with error messages and typical causes. I won't be able to cover everything, but if you are stuck, please check!

### Common Bugs That Do Not Produce an Error Message

Grouping and order of operations (e.g. `b/2*a` rather than `b/(2*a)`)

Defining a variable more than once. If you write `x = 5` and later write `x = np.arange(5)`, the earlier definition of `x` will be forgotten. If you use the same name for more than one variable you can run into trouble, for example, if you use `h` for Planck's constant and for a small interval.

Good practice to avoid bugs: variables that you use in a function should almost always be either passed as arguments or defined within the function. For example,

```
def f(t, omega, phi):
    return np.sin(omega*t + phi)
```

is much better than

```
def f(t):
    return np.sin(omega*t + phi)
```

In the lower case, it is easy to forget that `omega` and `phi` will be used by the function, and that their values could change. If you change `omega` then `f(t)` changes, but the call to `f` hides this fact. You want, for example, `f(1)` to give the same result in one part of your code as in another.

### Common Warnings

Divide by zero (like it sounds). A positive number divided by zero will give `inf`, a negative number divided by zero will give `-inf`, and zero divided by zero will give `nan` (not a number). Example:

```
x = np.linspace(-5, 5, 11)
y = np.sin(x)/x
```

Overflow or underflow. This happens when a function or operation gives a number too big or too close to zero to represent in floating point. For example, `np.exp(1000)` will give `inf` with an overflow warning, and `np.exp(-1000)` will give zero (perhaps with, perhaps without an underflow warning). Example:

```
x = np.arange(1000)
y = np.exp(x)
```

**Common Errors**

`NameError` This usually means that you forgot to define or import something. If your kernel crashed or restarted, you might have to re-run the cell(s) where you defined functions, variables, and/or imported modules.

`SyntaxError` This means that your code cannot be interpreted and run at all. Look for things like missing parentheses making it so that a line of code doesn't actually end. Example:

```
def f(x):
    return np.sin(5*(x + 2)
y = f(3)
```

`IndexError` This means that you tried to do something with an element of an array that doesn't exist or isn't allowed. For example, you tried to access the tenth element of an array with only five elements:

```
x = np.arange(5)
y = x[10]
```

`TypeError: unsupported operand type(s)` You probably tried to combine two incompatible kinds of things. For example, `5*np.sin` will produce this error, because aren't allowed to multiply a number and a function. Another example:

```
y = np.exp + 1
```

`ValueError: The truth value of an array with more than one element is ambiguous.` This tends to happen when you combine `if` with an array. For example:

```
x = np.arange(5)
if x < 3:
    print("x is less than 3")
```

`x < 3` is actually an array, with five values (some `True` and some `False`). The `if` statement doesn't know which of these value(s) to use. For this reason, you cannot call an `if` statement with an array.

Introduction to Computer Programming for the Physical Sciences
Timothy Brandt

## A Few Notes on Latex

Latex (or LaTeX) is a program to format books and articles. It is especially good at rendering math formulae. Most physics research papers are written in Latex, as are these notes. These notes will not teach you Latex. However, because you can use Latex in markdown cells within Jupyter notebooks, this is a good place to start to learn how to write pretty math formulae. This sheet will give you only the briefest introduction to how to write Latex in a markdown cell. It will also introduce you to a tool to translate Python code into Latex formulae; this may make it easier for you to spot bugs. **For this entire sheet, I will assume that you are working in a Jupyter notebook.** You can use Latex to write articles, and even make Power Point-style presentations, but those are topics for another day, and another setting.

Within a markdown cell, you can write a Latex formula by putting text within two `$` signs. For example,

```
$x + 1$
```

will show up as

$x + 1$.

Latex lets you use parentheses, exponentiation, sums, integrals, limits, etc., etc. (basically any math formula you see in any textbook). You'll probably find yourself getting familiar with these over time as you use them. One useful site, if you want to know how to write a symbol in Latex and don't know how, is `https://detexify.kirelabs.org/classify.html`

One very useful thing is to be able to translate a line of Python code into Latex. This can make it easier to spot mistakes, for example with order of operations, and to check whether your line of code does what you intend. For example, suppose we want to write

$z = e^{-x} + x(y^2 + 3x)$

In Python, we would write

```
import numpy as np
z = np.exp(-x) + x*(y**2 + 3*x)
```

We can translate this line into Latex using a module called `sympy` (included in both Anaconda Python and Google Colab):

```
from sympy import latex, sympify
print(latex(sympify('exp(-x) + x*(y**2 + 3*x)')))
```

**There are a couple of things that I needed to change.** I first changed `np.exp` to `exp`, and

you would need to do similar things with, for example, `np.sin` and `np.cos`. Other than that, this approach will work with expressions, but it doesn't like the `z =` part. So, I just put the line of code after `=` into quotes, change `np.exp` to `exp`, and ask `sympy` to translate. I get

```
x \left(3 x + y^{2}\right) + e^{- x}
```

If I write, in a markdown cell,

```
$ x \left(3 x + y^{2}\right) + e^{- x} $
```

I get

$x(3x + y^2) + e^{-x}$

You can try this yourself: just copy and paste your math expressions from Python as

```
from sympy import latex, sympify
print(latex(sympify('math_expression')))
```

(replacing `math_expression` with your actual expression) and see what you get. It will hopefully help you learn Latex and catch bugs, all at the same time.

I'll include one more example here. The quadratic formula is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

If I want to write that in Python, for the positive root, I would write

```
x_plus = (-b + np.sqrt(b**2 - 4*a*c))/(2*a)
```

I'll translate this via `sympy`. Please remember:

1. Put the expression in quotes (single quotes `'` and double quotes `"` will both work)
2. Replace `np.sqrt` with `sqrt`
3. Include only the right-hand side

```
from sympy import latex, sympify
print(latex(sympify('(-b + sqrt(b**2 - 4*a*c))/(2*a)')))
```

This gives

```
\frac{- b + \sqrt{- 4 a c + b^{2}}}{2 a}
```

Putting this between `$` in a markdown cell:

```
$ \frac{- b + \sqrt{- 4 a c + b^{2}}}{2 a} $
```

gives

$\frac{-b+\sqrt{-4ac+b^2}}{2a}$

which is what we want. What if we forgot some parentheses? For example, what if we wrote

```
x_plus = -b + np.sqrt(b**2 - 4*a*c)/2*a
```

I can translate this into Latex using

```
from sympy import latex, sympify
print(latex(sympify('-b + sqrt(b**2 - 4*a*c)/2*a')))
```

and get

```
\frac{a \sqrt{- 4 a c + b^{2}}}{2} - b
```

Putting this between `$` signs in a markdown cell, I would have

```
$ \frac{a \sqrt{- 4 a c + b^{2}}}{2} - b $
```

which renders as

$\frac{a\sqrt{-4ac+b^2}}{2} - b$

Whoops! This makes it a lot easier to see my mistakes.