Welcome! Take out your laptop!

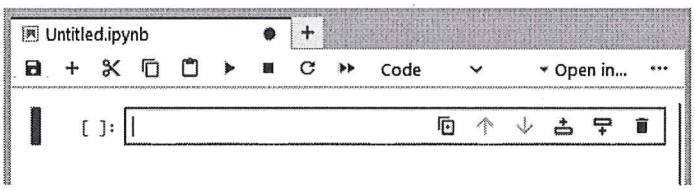
Lecture 2

Python as a basic calculator

To start, you will launch a Jupyter Notebook on your computer. A notebook is an environment where you can run code alongside text, images, etc., and where you can keep notes about what your code does. The Python environment runs on your computer, but is accessed through a browser window.

When you launch Jupyter, you can navigate through all your files. You should create (or navigate to) a folder where you're going to keep all of your ps12a work. Then, start a new iPython notebook in that folder.

Now you should be looking at a Jupyter Notebook, as shown below. The little box is called a cell. There are two types of cells - the default is a code cell. We'll stick with that for now.



Type 2+3 in the cell and then press **Shift + Enter** - this "runs the cell" - in other words, the computer starts at the top of your code cell, looks at each line that you typed, and follows the instructions. When it reaches the end of a cell, it prints out the value of the last line. You should see it print out 5.

Now you can use the next cell, or go back, edit, and rerun an old cell. You can put multiple lines in the same cell, like this:

```
5+4
2+3
```

If you want it to display both outputs, try:

```
print(5+4)
print(2+3)
```

You can also print out text, like this:

```
print('The answer is 5.')
print(f'There are {2+3} cookies.')
```

Figure 3: A Cookie Counter from the 1980s.

So far, it seems Python can add and subtract and print text, so it seems to have all the functionality[11] of a Cookie Counter, as shown in Figure 3. What if we want to do some fancier math, like take a logarithm?

[11] It's a touch more sophisticated - it can also multiple and divide.

Activity 1: Python as a fancy calculator - importing libraries and using functions

Python already seems to know what to do if you want to add or subtract. What if you want to take the log of some number?

On your calculator, you would hit a special button, the log button. In Python (and really in your calculator too) these fancy buttons are called **functions**, and a lot of them reside in a special "library." You need to import this library to use those functions (fancy buttons). The function runs an algorithm to determine the output[12].

One of the most important libraries for us is the `numpy` library. Libraries are generally imported at the beginning of a session, at the very top of your code. You only need to run the cell once to import the libraries.

```
%reset -f
import numpy as np
import matplotlib.pyplot as plt
```

Now all of the functions[13] in the numpy library are accessible via `np.function_name()`.

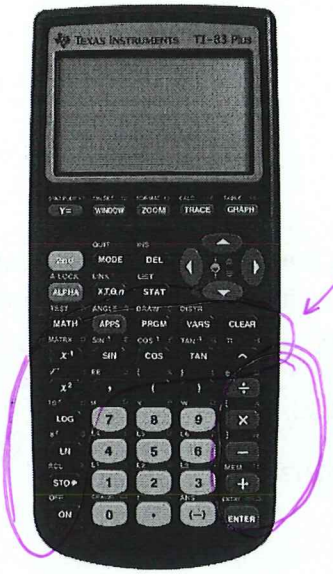


Figure 4: A graphing calculator, likely from the 2000s.

Anatomy of a Function

```
np.cos(3.14159)
```

[12] Before calculators, you would look up the value in a table. These tables were also computed using an algorithm, but humans executed the algorithm.

[13] To see what's available, you can read the docs!

Operation	Regular math syntax	Python syntax
Add	$a+b$	`a+b`
Subtract	$a-b$	`a-b`
Multiply	ab, $a \times b$	`a*b`
Divide	$\frac{a}{b}$	`a/b`
Exponentiate	a^b	`a**b`
Square root	$\sqrt{a}$	`np.sqrt(a)`
Sine	$\sin(a)$	`np.sin(a)`
Cosine	$\cos(a)$	`np.cos(a)`
Natural Log	$\ln(a)$	`np.log(a)`
Log base 10	$\log(a)$	`np.log10(a)`

Table 1: Commonly used mathematical operations in python and the numpy library. You can find a full list of mathematical functions in numpy at this link.

1. Take a couple minutes to introduce yourselves or catch up with your friends.

2. *Do you know the order of operations python uses?* Which of the following three lines are equivalent?

```
12/3*4
12/3/4
12/(3*4)
```

(12/3) · 4

equivalent

P arenthesis
E xponents
M ultiplication
D ivision
A ddition
S ubtraction

equal

equal

3. Use Python to find the values of x that satisfy $0 = 3x^2 + 4x - 10$.[14]

[14] I know the quadratic formula is universally loved by all. It's $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$.

$$x = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

$$x = \left[-4 \pm \sqrt{4^2 - 4 \cdot (3) \cdot (-10)}\right] / (2 \cdot 3)$$

← put this in python

4. What's the standard angle formed by the vector shown in Figure 5? Hint: The numpy library has a function called `arctan` and another one called `arctan2`. Look at the documentation; what's a good choice in this case?

$$\tan\theta = \frac{y}{x}$$

$$\theta = \arctan\left(\frac{y}{x}\right)$$

$$\theta = \arctan\left(\frac{-4}{-3}\right)$$

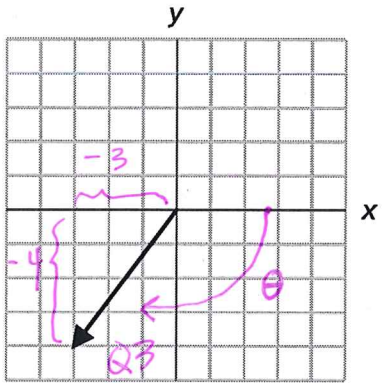


Figure 5: A vector in the third quadrant.

np.arctan doesn't know the quadrant
takes arg (Y/X)

np.arctan2 knows the quadrant, takes arg (y, x) ← keeps track of signs

Activity 2: Assigning values to variables - (kinda) beyond the calculator

The `=` sign in python is not the equals sign you are familiar with from math! Instead, the `=` sign in python is used to **assign variables**. Instead of pronouncing it "equals," we pronounce it "gets," as in "x gets 2" for `x=2`. Here's how it works:

X = 2

~~"x equals 2"~~

"x gets 2"

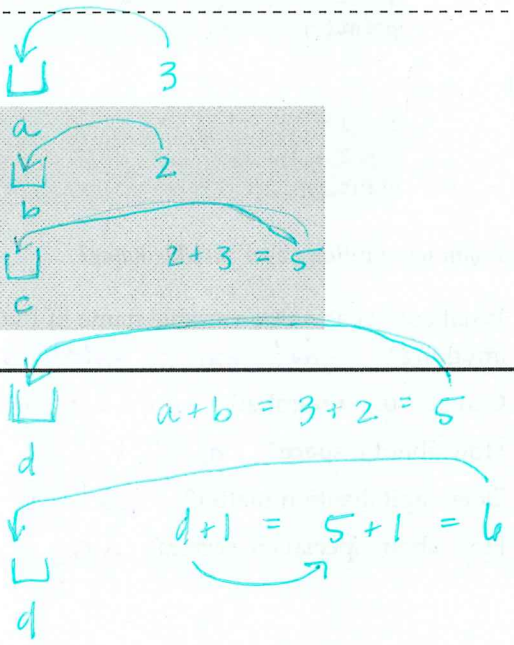
Assigning variables

Python looks to the **right** of the assignment sign, computes whatever is there, then **assigns** that value to the newly created variable on the left.

Example:

```
a = 3
b = 2
c = 2 + 3
d = a + b
d = d + 1

print(c,d)
```



What happens if we treat the equal sign more like a mathematical equation? Try running the following statement:

`a - 8 = 3 + 4`.

value

expression (not valid variable name)

What happens?

"throwing an error"

tells you what's wrong

very helpful! but sometimes confusing

---

1. *Are you understanding assignment in python?*

   For each of the following code bits, figure out what the code will print without coding it up - then you can code it up to see what happens. Assume that each snippet of code is the only code (ie, no lines came before these).

   (a)

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

   x not defined yet (not in scope of x)

   (b)

   ```
   y = 3
   y = 2 + 2*y
   print(y)
   ```

   good!

   (c)

   ```
   z - 3 = 0
   z = 2 + 2*z
   print(z)
   ```

   can't assign value to expression (z-3 is not a valid variable)

2. *Can you use anything as a variable name?*

   - What counts as a fine variable name in Python? Can you use anything? no! has to follow some rules
   - Can you use a number? yes, but not as the first character
   - How about a space? no!!!!!
   - Does capitalization matter? yes! b vs. B are different
   - How about special characters? only underscore

Activity 3: Getting fancy - defining arrays

If variables are buckets to hold a single value, an array is more like an egg carton that can keep multiple values organized in the correct order.

Here's one way to define[15] a numpy array:

```
xx = np.array([3,4,-8])
```

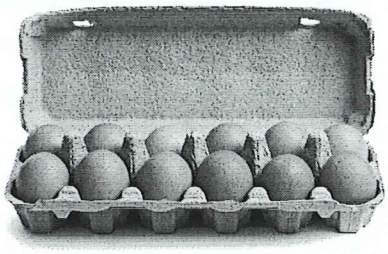
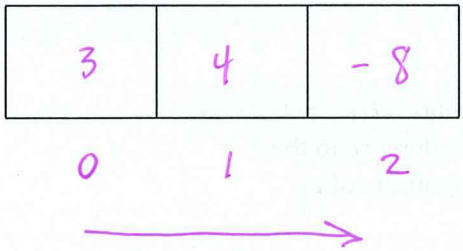


Figure 6: A carton of eggs.

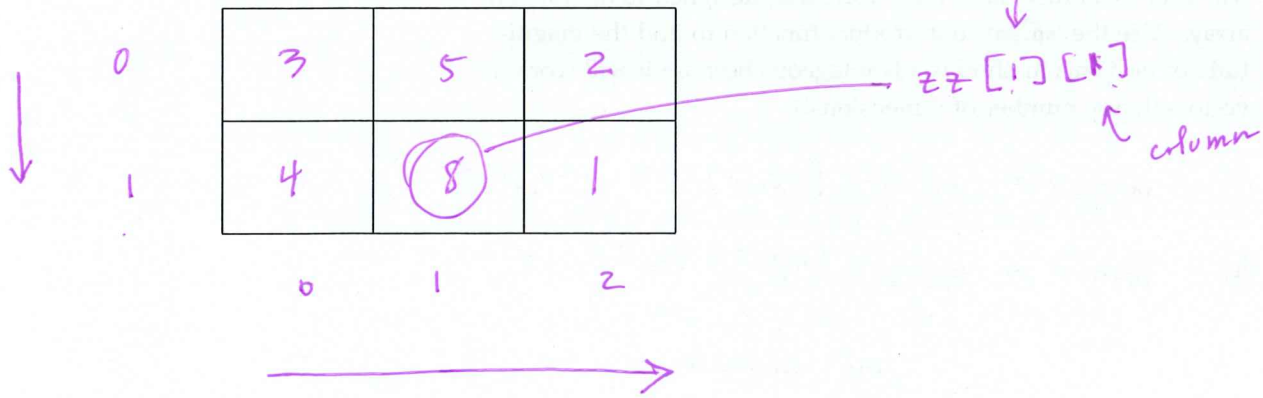
[15] Early in my coding career I developed a habit of using doubled letters for arrays (xx) and single letters for single values (x) to keep myself organized. It's stuck, but you can use any variable name you like.

How do we keep track of array elements? I visualize an array as a set of mailboxes or egg carton wells where I can store my numbers. Each "mailbox"—or *element*—has an address, called an *index*. Python starts counting elements at zero, so the index of the element "3" in the example above has an associated index of zero, the element "4" has an index of 1, and so on.



You're not restricted to 1D. You can make multidimensional arrays as long as they're rectangular - here's a two dimensional one:

```
zz = np.array([[3,5,2],[4,8,1]]) .
```



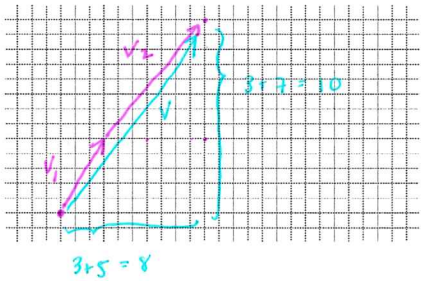
One really useful way to use arrays in physics is to use them as vectors, so let's start in that context.

---

1. Define the two vectors $\vec{v}_1 = (3,5)$ and $\vec{v}_2 = (7,8)$ as numpy arrays.

```
v1 = np.array([3,5])
```

2. Add the two vectors, graphically, below. Then add the two vectors together in numpy to form a new vector, $\vec{v}$. Do your graphical and numpy answers match? (Are they the same vector?) Try changing the variable v to be the difference instead; does that make sense too?



① `(v1[0] + v2[0], v1[1] + v2[1])`

② `v1 + v2`

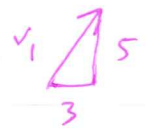
both ways are equivalent with numpy arrays! numpy knows how to do element-wise operations.

example: `v1**2` → `(v1[0]**2, v1[1]**2)`

3. You can access an element in an array via the syntax `array_name[index]`, which is pronounced "array name sub index" in reference to the mathematical notation a_i. Use this to find the magnitude of v_1.

`v1[0]` 0th column element of `v1`

4. Find the standard angle formed by v_1.



$\theta = \arctan 2(3,5)$

5. The numpy library has a lot of functions designed to operate on arrays. Use the `np.dot` dot product function to find the **magnitude** of v_1. (This method is advantageous because it will work on vectors in any number of dimensions!)

① `mag_v = np.sqrt(v1[0]**2 + v1[1]**2)`

② `mag_v = np.sqrt(np.dot(v1,v1))`

are equivalent

Activity 4: Quick Plotting

We don't usually look directly at arrays of data to form conclusions. Imagine you were looking for earthquake tremors, but your seismometer only spit out columns and columns of numbers. It would be hard to tell what was going on! In most cases, it is far preferable to visualize data in a plot.

We've already imported a Python library that will allow us to make plots: in Activity 1 we ran a line of code reading `import matplotlib.pyplot as plt`. We now have access to the functions in the pyplot library.

Plotting Best Practices

```
xx = # set up your x coordinates here
yy = # set up your y coordinates here

# begin plotting
fig,ax = plt.subplots(figsize=[6,4])
ax.plot(xx,yy)
ax.set_xlabel('my x label')
ax.set_ylabel('my y label');
```

1. Using the "Plotting Best Practices" code from above, make a plot of Harvard College class size by graduation year, using the data below:

```
year = np.array([2023,2024,2025,2026,2027])
students_in_class = np.array([1666,1420,1965,1620,1631])
```

Be sure to label your axes!

2. Replace `ax.plot` with `ax.scatter`. What changes?

3. You can always look up what different functions do by typing function_name? in a Jupyter cell and running the cell. Try it with `ax.plot` and `ax.scatter`. Can you make `ax.plot` produce a scatterplot?

xx	yy
1.85840708	-0.491954948
1.946902655	0.326629123
1.946902655	1.05390185
1.946902655	-0.900643604
2.123893805	-1.263475463
2.212389381	-0.172164119
2.389380531	-0.989541432
2.389380531	1.374094932
2.566371681	0.693081255
2.743362832	0.330249397
2.920353982	1.331053902
2.920353982	-0.441673371
3.008849558	-0.850362027

Figure 7: Numerical seismometer data: hard to say what's going on in this data just by looking.

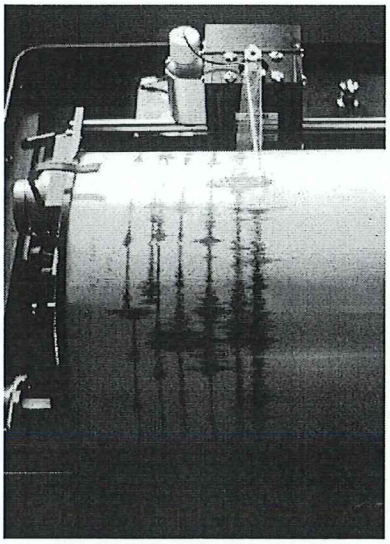


Figure 8: USGS seismograph of the eruption of Mount Pinatubo in 1991. It's clear at a glance that the seismometer recorded some tremors!

Storing and Visualizing Data

LAST WEEK WE TALKED ABOUT DEFINING VARIABLES as a way to store numbers. Today we going to focus on a different kind of variable, the array, which can store multiple numbers: kind of like a spreadsheet, but with as many dimensions as we want. We'll see how to visualize sets of data using fancier plotting commands. We'll also get our first taste of more advanced control structures: loops!

Stars Covered

- Variables, arrays, and plotting
- Conditionals[1] and loops

[1] Conditionals will be covered next week.

Detailed Learning Objectives

After this lecture, you should be able to...[2]

1. Define an array using `np.arange()` or `np.linspace()`
2. Use mathematical operations on variables and arrays
3. Perform various operations on arrays (`max`, `len`, `min`, `sum`)
4. Create basic plots and customize them
5. Use a `for` loop to repeat a code block a specified number of times

[2] Functions and keywords covered:
`np.linspace()`
`np.arange()`
`np.zeros()`
`np.ones()`
`np.zeros_like()`
`plt.plot()`
`plt.scatter()`
`for, in`

Lecture 5 Pre-reading

In this lecture, we will delve deeper into plotting and try to make the procedure less "black-box." Then, we will discuss an important coding construct called a loop, which allows you to tell the computer to repeat a set of code multiple times–no copy-pasting required!

Python plots arrays

You're probably going to want to use more than three points to make a plot. In fact, if you want to make plots of smooth curves you will want way more than that (like, maybe 100 or more). There are a few useful numpy functions for automatically generating arrays of any size.

`np.linspace(start, end, num=)` is one of the most useful functions for plotting and scientific calculation. It creates a 1-d array of arbitrary length (set by the argument `num=`), with any starting and ending values that you want (set by arguments `start` and `end`). It will determine the spacing of the points based on these parameters. A similar function is `np.arange([start, ]stop, [step, ])`, which allows you to make a 1-d array with a certain range of points spaced by a set interval - it will determine the length of the array based on these parameters.

You might also want to make some "blank" arrays full of either zeros or ones for certain mathematical operations, or to just create a placeholder array with a set shape. A few useful fuctions for this are `np.zeros(shape)`, `np.zeros_like(xx)`, `np.ones(shape)`, and `np.ones_like(xx)`. Check out the documentation for these!

```
# here is an example of np.linspace() to get an array of (-10,10) with 1000 points:
xx = np.linspace(-10, 10, 1000)
# check the shape!
print('the shape of xx is ', xx.shape())

# here is an example of np.arange() to get an array starting at 0, 1000 points long, with an interval of 0.1
yy = np.arange(0, 1000, 0.1)
# check the shape!
print('the shape of yy is ', yy.shape())

# make a array full of zeros of length 100
zz = np.zeros(100)

# make an array full of zeros of the same shape as xx
zz_xx = np.zeros_like(xx)
```

How does Pyplot plot my data?

We've been using the following skeleton to plot data:[3]

```
xx = np.array([0,1,2])
yy = np.array([2,3,5])

# begin plotting
fig, ax = plt.subplots(figsize=[6,4])
ax.plot(xx,yy)
ax.set_xlabel('my x label')
ax.set_ylabel('my y label');
```

[3] If you have used Pyplot in other contexts, you might have been using syntax like `plt.plot(xx,yy)`. This is perfectly fine as well, but if you are ever juggling multiple figures, the "object-oriented" approach taught here will be a lot easier to keep track of.

But what is actually going on here? What's the deal with `plt.subplots(figsize=[6,4])`? This, like `np.sin(3.14)`, is a *function call*, and it gets evaluated just like any other function. Fundamentally, the whole line of code is just a variable assignment, and we can break down this line of code the same way we did with the simple example of `a = 5`. See the next page.

When the interpreter reaches the line `fig, ax = plt.subplots(figsize=[6,4])`, it treats it like any other variable assignment, and goes through the three steps:

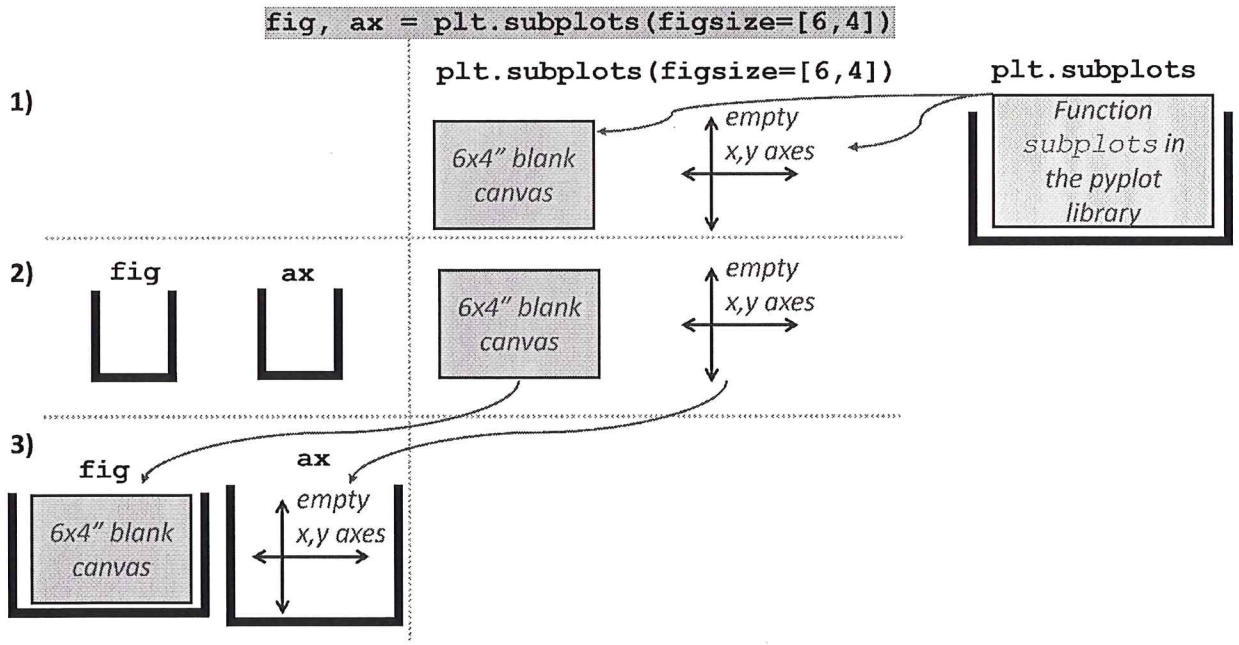
1. The interpreter evaluates the right-hand side. In this case it needs to evaluate the function `plt.subplots` with the argument `figsize=[6,4]`. The `plt.` in front of the function tells the interpreter to look for a function called `subplots` inside the `pyplot` library. The interpreter finds that function and passes it the argument.[4] This function evaluates to a set of two values. These values are not numbers, but *objects*, but they act in this case just like a number: these objects are things that can be stuck inside a variable bucket so we can find them later. The two objects are a Figure object, which is like a blank canvas, and an Axes object, which can have data plotted on it. So the right-hand side expression evaluates to a Figure object and an Axes object.

[4] We will talk in more detail about function calls next week.

2. The interpreter looks for the variable(s) specified on the left-hand side. There are two variable names, and neither one exists yet, so the interpreter creates two empty variables called `fig` and `ax`.

3. The values from the right-hand side are placed inside the variables on the left, overwriting anything that is currently inside. This is done in order of appearance, so the Figure object is placed inside the `fig` variable and the Axes object is placed inside the `ax` variable.

It's important to note that the assignment in step 3 had **nothing to do** with what we named the variables, and **depended only on their order**. We could have named the two variables `spam` and `eggs`, though those aren't very descriptive variable names. We could have been diabolical and named them `ax` and `fig`, in that order, so that `ax` would contain a Figure and `fig` would contain Axes. This would be terrible and is not recommended.



Now that we have an Axes object (which will be displayed on the Figure canvas), we can tell that Axes object to do plotting-related tasks like `ax.plot` or `ax.set_xlabel`.[5] If we have more than one set of Axes in the same figure, it is useful to give them descriptive names (such as `ax_accel`, see example below).[6] Read through the code below and think about how it produced the figure on the right. Some of the details will be unfamiliar, but can you work them out from context clues?

```
'''Make a plot of the first two seconds of acceleration, velocity,
and position for a ball dropped from rest.'''

# define a time axis and a, v, x
t = np.linspace(0,2,100)
dt = np.diff(t)[0] # this is the spacing btwn time points
a_x = np.zeros_like(t)+-9.8 #m/s^2 (constant acceleration)
v_x = np.cumsum(a_x)*dt #m/s (v is the integral of a)
x = np.cumsum(v_x)*dt #m (x is the integral of v)

# create a figure with three sets of axes
fig,ax = plt.subplots(3,figsize=[3,5])
# ax is now an array of axes. Let's name each set of axes:
ax_accel = ax[0]
ax_vel = ax[1]
ax_pos = ax[2]

# now we can separately plot a, v, x
# acceleration:
ax_accel.plot(t,a_x)
ax_accel.set_ylabel('$a$ (m/s$^2$)')

# velocity:
ax_vel.plot(t,v_x,label='$v_x$')
ax_vel.set_ylabel('$v$ (m/s)')

#position:
ax_pos.plot(t,x,label='$x$')
ax_pos.set_xlabel('time (s)')
ax_pos.set_ylabel('$x$ (m)')

plt.tight_layout() # arrange the labels nicely
```

[5] Remember that the `ax` variable, like all variables, maintains its value when you run another cell. So if you make a second cell and run `ax.plot(x2,y2)`, that trace will appear on the original axes, not a new set of axes, until you reassign a newly created set of axes to the `ax` variable.

[6] We'll talk more about the `np.linspace(0,2,100)` line in lecture. Here the function creates an array of 100 points, equally spaced between 0 and 2, which is stored in the variable `t`.

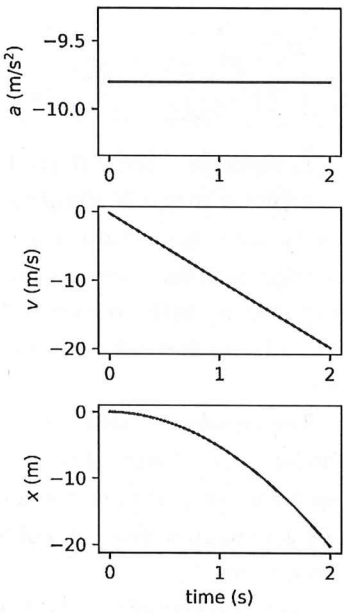


Figure 1: The plot produced by the code to the left.

Beyond sequential control: for loops

Up until now, the interpreter has read each line of code, executed it, and moved on to the next line. But sometimes we want to do the same thing a few times, with just a few tweaks. We could copy-paste the code over and over, but this gets cumbersome for $n > 2$, and totally untenable for $n \approx 5$ or more. What we need is a way to tell the

interpreter to go back and run the same code again, perhaps with the value of one or two variables changed. What we need is a `for` loop.

The idea of a for loop is that you have a block of code you want to do multiple times. This block of code is separated from the rest of your code by *whitespace*, in this case a tab. All the lines of code that are indented will be repeated in the loop, and when the loop ends, control returns to the next unindented line. A very basic `for` loop might look something like this:

```
1 iterable = np.array([0,1,2])
2 for a in iterable:
3         print(a) # this is inside the loop
4 print(a) # this is outside the loop
```

The output of the above code would look like this:[7]

```
0
1
2
2
```

[7] See if you can work out where these numbers came from. Hint: The interpreter runs the lines in this order: 1,2,3,3,3,4.

Let's break down what's happening. The keyword `for` tells the interpreter that it is about to enter a loop. The variable named directly after `for`, in this case `a`, is a variable whose value is going to change each time we run through the loop. That variable is going to sequentially take on each of the values in the object, called an *iterable* (i.e. a thing that can be iterated through), that follows the keyword `in`.

The number of times we go through the loop is determined by the length of the iterable. In this case, there are three values in `np.array([0,1,2])`, so the loop will run three times. There is only one line of code (line 3) inside the loop, so line 3 will be executed three times.

Each time control runs through the loop, it first assigns a value from the iterable to the variable `a`. It's as though there is a secret line of code at the top of the loop that says `a = 0` on the first run through the loop, then `a = 1` on the second, and finally `a = 2` on the third. That's why, when we print out `a`, we see that it first has the value 0, then 1, then 2.

When the interpreter has run through the loop three times, it sees that there are no more unused values in the iterable, so it exits the loop and executes line 4. `a` maintains its current value, 2, which is why we see another 2 in the output, printed out by line 4. `a` is now just another variable, and can be used or reused in any way.

Can you think of a situation where a loop would be useful?

Here is an equivalent (but much uglier!) piece of code without a loop:

```
iterable = np.array([0,1,2])
a = iterable[0]
print(a)
a = iterable[1]
print(a)
a = iterable[2]
print(a)
print(a)
```

Lecture 5

Activity 1: Plotting Mathematical Functions!

Say you want to plot $y = \sin(x)$ from $-\pi$ to π. By hand, you start by picking some points on the x axis. You figure out what values they give for y, and then plot the corresponding (x, y) coordinate - then connect the dots. This is done in Python the same way, except it's much faster and we can do way more dots.

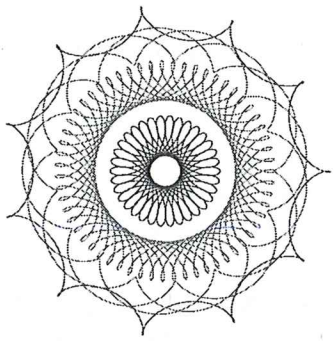
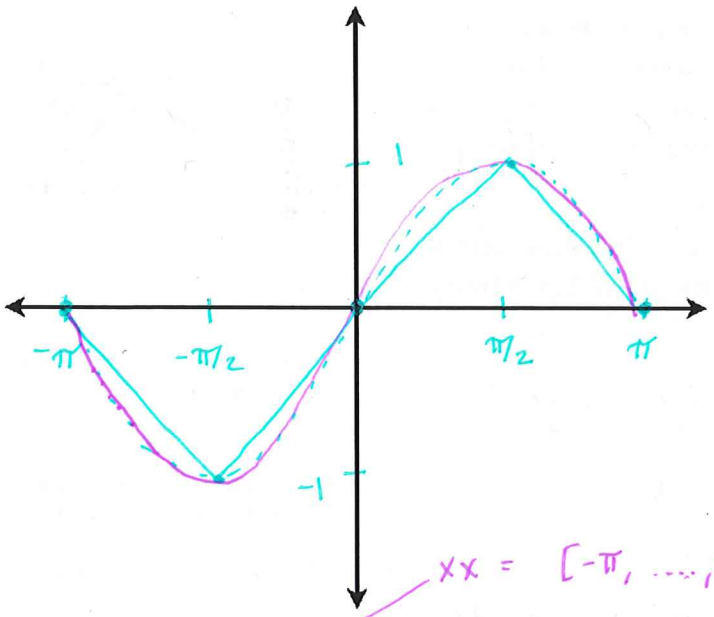


Figure 2: A spirograph generated in Python.



choose x, calc $y = \sin(x)$

x	y
0	0
$-\pi$	0
π	0
$-\pi/2$	-1
$\pi/2$	1

xx = [$-\pi$, ..., $-\pi/2$, ..., 0, ..., $\pi/2$, ..., π] 100 points

yy = np.sin(xx)

There are MANY other plotting functions, like `plt.scatter()`, `plt.hist()`. See a gallery of plots here.

Plotting Best Practices

```
xx = # set up your x coordinates here
yy = # calculate the y coords by operating on xx

# begin plotting
fig,ax = plt.subplots(figsize=[6,4])
ax.plot(xx,yy)
ax.set_xlabel('my x label')
ax.set_ylabel('my y label');
```

1. Plotting $\sin(x)$.

   (a) Follow the structure above by first creating an array xx with 11 points from $-\pi$ to π. You should use `np.linspace()`, which was described in the prereading.

   (b) Use that array to calculate the corresponding coordinates in the array yy. (Hint: Array math!)

   (c) Plot the function and customize your plot! You can change the colors[8], the line widths, the marker styles[9], the marker colors, the font size, the font, the size and shape, and on and on.[10]

   (d) Plot a second curve on your plot (try $\sin(x - np.pi/8)$). Does it shift the way you expect, relative to your plot of $\sin(x)$?

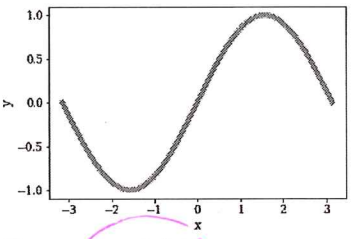


Figure 3: $y = \sin(x)$.

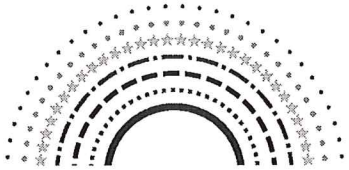


Figure 4: A plot rainbow - we'll make one of these later!

2. A ball is launched **vertically** from the surface of the earth with an initial speed of 10 m/s. Plot the y **position** as a function of time and the y **velocity** as a function of time.

v_0 ↑

$y(t) = y_0^{\,0} + v_0 t - \frac{1}{2} g t^2$

$v_y(t) = v_0 - gt$

t = np array 0 → 10 w/ 100 points

calc y from t

calc v_y from t

5 dims

[8] Color options.

[9] Marker options.

[10] Good plot checklist:

- Axes are labeled
- Axes have units
- Text on plot is legible
- Data points should be represented as dots with error bars, not connected lines
- Plot is free of clutter and easy to read
- Plot gets the message across

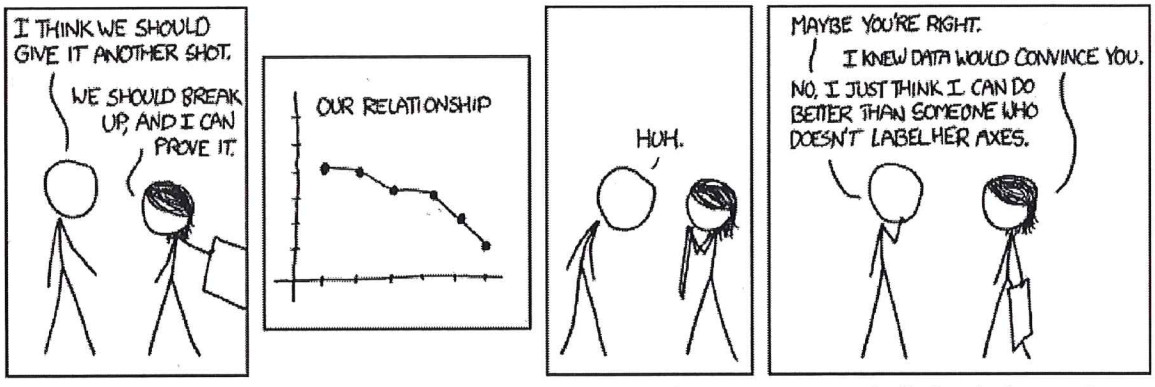


Figure 5: xkcd's *Convincing*, number 833.

Activity 2: Introduction to loops

Up until now, we've seen that Python executes each line in your code in order from top to bottom. `for` **loops** allow blocks of code to be executed multiple times while iterating through some variable. They're really useful when you have a boring repetitive task.

Anatomy of a `for` loop

```
for a in b:
    # do stuff

#this is outside loop
```

Example:

```
xx = np.array([4,16,32])

for a in xx:
        print(a)
        print(a**2)
        print(xx)

print(a)
```

```
xx = np.array([4,16,32])

for i in range(len(xx)):
        print(i)
        a = xx[i]
        print(a)
        print(a**2)
        print(xx)

print(i)
print(a)
```

There are `while` loops too, and we can talk about those later. If you know how many iterations you need, you're probably better off with a for loop, but you can achieve the same kinds of things with either one.

In Python you can also loop through multiple arrays at once using `zip`, or you can iterate through an index and a value at the same time with `enumerate`.

For Loop Practice

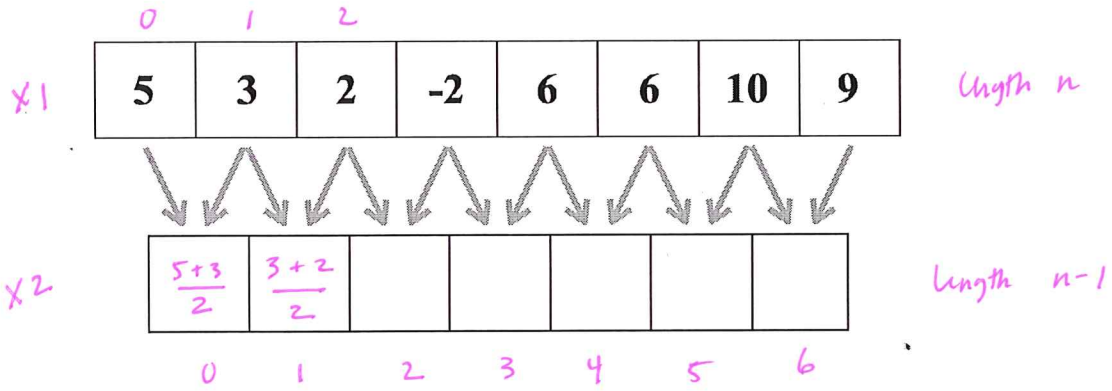
1. Sum all the integers from 1 to 1000 using a for loop, and then confirm this is the correct answer using an array and np.sum().

```
x = 0
for i in range(1000+1):        # [0,...999]
    x = x + i  or  += 1
print(x)
```

vs. `np.sum(range(1000+1))`

2. Use a for loop to compute an array `x2` where each element is the average of two adjacent elements in the original array `x1`. [11]

```
let x1 be any array
x2 = np.zeros(len(x1) - 1)
for i in range(len(x2)):
    x2[i] = (x1[i] + x1[i+1]) / 2
```



[11] Note that this array will be one element shorter than `x1`. Can you also figure out how to do this using only arrays, with no for loop?

Activity 3: Loops make code pretty

You will probably find yourself at some point copy-pasting a lot of code over and over. And then, maybe, you want to go back and change something–but you have to change something in *all* the lines of code you just copied! Save yourself this pain by using loops whenever you want to do something repeatedly.

The code below makes a rainbow plot like the one on the right. But, it relies on many copy-pasted lines of code. What if I wanted to invert the rainbow, using `-np.sin(theta)` instead of `np.sin(theta)`? I'd be in for a lot of hassle.

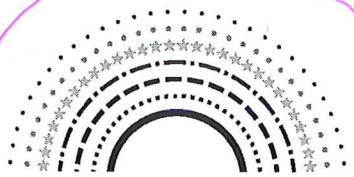


Figure 6: A plot rainbow - can we make the code as pretty as the plot?

repetitive

Rainbow Plot (ugly version)

```
theta = np.linspace(0,np.pi,32)
dr = 0.1
r = 1
fig,ax = plt.subplots(figsize=[8,8])
ax.plot(r*np.cos(theta),r*np.sin(theta),'.',color = 'red', markersize = 6, linewidth = 1)
r = r-dr
ax.plot(r*np.cos(theta),r*np.sin(theta),'.',color = 'orange', markersize = 10, linewidth = 1)
r = r-dr
ax.plot(r*np.cos(theta),r*np.sin(theta),'*', color = 'gold', markersize = 14, linewidth = 1)
r = r-dr
ax.plot(r*np.cos(theta),r*np.sin(theta),'-.', color='green', markersize = 6, linewidth=4)
r = r-dr
ax.plot(r*np.cos(theta),r*np.sin(theta),'--', color='blue', markersize = 6, linewidth=5)
r = r-dr
ax.plot(r*np.cos(theta),r*np.sin(theta),':', color='blueviolet', markersize = 6, linewidth=4)
r = r-dr
ax.plot(r*np.cos(theta),r*np.sin(theta),'-', color='darkviolet', markersize = 6, linewidth=8)
r = r-dr
ax.set_aspect(1)
ax.axis('off')
```

change r in loop

same plot

same calc

in each loop iteration:

calc r = r - dr

choose plot args from list

1. Discuss with your partner how you might convert this code into something tidier using loops.[12]

[12] Hint: you can store anything you like inside an array, including `'strings'`.

2. If time allows, implement it (you can access the rainbow plot code on today's lecture page on Canvas).

3. With a single keystroke, invert the rainbow.

CLASS NOTES

Functions and Control

LAST WEEK WE TALKED ABOUT WAYS TO STORE and visualize data: using arrays to store many numbers at once, plotting data, and storing data in external files. So far, we've used a lot of built-in functions, and functions in the `numpy` and `pyplot` libraries. This week we'll delve into defining our own functions to perform specific tasks. We'll also talk about a new form of control flow: conditional statements.

Stars Covered

- Conditionals and loops
- Functions and variable scope

Detailed Learning Objectives

After this lecture, you should be able to...[1]

1. Create (and properly document!) a user-defined function
2. Recognize global and local variables
3. Use an if/elif/else statement
4. Step through a block of code in the order that the computer would execute it

[1] Keywords covered:
`def`
`return`
`break`
`continue`
`if`
`elif`
`else`
`<, >, ==`

Lecture 8 Pre-reading

So far, we have seen one construction that we can use to non-sequentially execute code: *loops* allowed us to repeat the same block of code over and over. But, as soon as we leave the loop, the interpreter "forgets" that code. What if we have a block of code that we want to be able to (re-)run on command, without having to copy-paste it? What we need is a *function*.

Just what is a function?

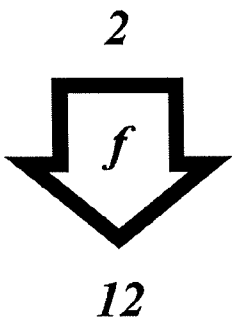


Figure 1: You may have learned about mathematical functions as "machines" or "black boxes" that operate on a number and spit out another number. This conceptualization will also serve you well when thinking about Python functions!

You are already familiar with mathematical functions, for example, the function $f(x) = 3x^2$. The name of the function is f, and it acts on one argument, x, which can be any number. The output of the function is three times the square of the argument, $3x^2$. Python functions are extremely similar, except the action of the function can be anything–any block of code we can imagine.

Here is how we might define the same function in Python syntax:

```
prefactor = 3
def f(x):
        xsquare = x**2
        return prefactor * xsquare
```

This Python function operates exactly like our mathematical function: its name is `f`, it takes one argument, `x`, and it spits out the number after the keyword `return`; that is, $3x^2$. Right now, we haven't actually operated on any numbers; we have just defined what the function `f` is going to do.[2] To actually get something out of the function, we need to *call* the function: we need to give it an argument.[3] For example, after defining the above function, we could run the code below:

```
a = 2
b = f(a)
print(b)
```

This code will print out the number 12, as expected. What might be less expected is what happens if we then try to run this line:

```
print(xsquare)
```

This line of code does **not** print out 4. Instead, it gives us an error: "`NameError: name 'xsquare' is not defined`". To understand why this happens, we need to understand a concept called *variable scope*: different variables are visible to different parts of the program.

The main control of the program has a collection of defined variables that it can use, which we have been conceptualizing as "buckets" that the interpreter has access to. These variables are called **global variables**, because they are globally available. Functions, on the other hand, have their own private space of variables, called **local**

[2] An extremely common mistake is to define a function and then expect output without ever calling the function.

[3] In math terms, we might call this *evaluating the function* at a certain value of x, and we also use that terminology in Python. The syntax in Python, as in math, is parentheses; we have already used this syntax many times!

variables, that are not accessible to the main program.[4] In the example above, `prefactor`, `a`, and `b` are global variables, while `x` and `xsquare` are local to the function `f` because they are *defined within the function*. So when the main program tried to print out `xsquare`, it went to look for a bucket labeled "`xsquare`" and was unable to find one, because that variable is local to the function and not shared.

Note that the function **is** able to see global variables.[5] In the above example, the function is able to use the global variable `prefactor`; it evaluated that variable as 3 when the function was called.[6,7]

Variable scope is one of the most confusing Python concepts we will cover in 12a, but it is important to have a fundamental understanding of functions and variable scope, or you will spend a *lot* of time debugging your code. Trust us!

[4] You could visualize this as a secret closet full of buckets that the function uses to do its operation, but does not share with the main program.

[5] The function and main program do not have a reciprocal relationship!

[6] Importantly, the function will reevaluate the `prefactor` variable each and every time it is called! So, if you changed the value of `prefactor` to 5, then the next call of `f(2)` would return 20!

[7] One more detail: there can be distinct local and global variables with the same name (though this is confusing for the human programmer, so this is **not recommended**!). The main program can't see the local variable, so it will always use the value stored in the global version. The function searches its own space first, finds the local version, and ignores the fact that there may be a global version.

continued on the next page...

If you are reading this, you are about to learn about conditionals; else you might be confused in lecture

We now know two ways to repeat blocks of code (loops and functions). What if we want to skip over some code? In this case, we use *conditionals*, which conveniently use the English-friendly syntax `if..else`.[8]

[8] There is also the less conversational `elif`, which is short for "else if".

A set of conditionals might look like this:

```
if statement:
        # if statement is True, execute the code here
elif statement_2:
        # if statement was False but statement_2
        # is True, execute the code here
else:
        # if neither of the above statements
        # are True, execute the code here
```

Here, `statement` and `statement_2` are variables or expressions that will be evaluated by the interpreter. We could use a special type of data called a Boolean[9], which is either `True` or `False`:

[9] Named for the logician George Boole.

```
statement = False
statement_2 = True
if statement:
        print('we will never run this line; statement is False')
elif statement_2:
        print('this will get printed!')
else:
        print('we will never run this line; statement_2 was True')
```

Often, instead of creating variables with the values `True` or `False`, we will place an *expression* in our conditionals that will evaluate either true or false.[10] Expressions that could be either true or false include "x equals 5"[11] or "y is less than 0"; these use the syntax outlined below:

[10] Less commonly, you could just use a number: in Python, `0` is equivalent to `False`. `1` is equivalent to `True`, and so is *every other nonzero number*, array, etc. Even `-np.pi` evaluates to `True`!

[11] Written `x == 5`. Finally, a symbol you are allowed to pronounce "equals"!

	Math	Python
Equal to	$=$	`==`
Not equal to	$\neq$	`!=`
Less than	$<$	`<`
Less than or equal to	$\leq$	`<=`
Greater than	$>$	`>`
Greater than or equal to	$\geq$	`>=`

In class, we will see how to combine conditionals with loops to create even more customizable code.

Lecture 8

Activity 1: Intro to user defined functions

LAST TIME WE SAW A FEW FUNCTIONS like the print function or np.cos(). We talked about when we call (use) those functions, they run an algorithm (more code) that we don't see. Now we'll see how we can write our own functions.

In the first week we also wrote a bit of code to solve the quadratic equation. We could imagine writing a "script" in which we'd need to use the quadratic formula more than once, or perhaps we'll need it in other programs we write.

We could copy and paste what we wrote every time we need to solve the quadratic equation, but it would end up adding a lot of clutter to our code.[12] To reduce this clutter and make our code much more readable, we can turn our code for the quadratic formula into a **user defined function**.

[12] If you are copying and pasting a block of code more than once, this is a sign that that block of code ought to be a function!

USER DEFINED FUNCTIONS ARE GREAT BECAUSE THEY:

- Can reduce the overall number of lines of code that need to be written
- Reduce the potential for mistakes
- Provide flexibility - if you need to change the function, you only have to change it in a single place
- Can eliminate global variables

You already really understand the concept of a function - it's just like a mathematical function, which performs an operation on some argument(s). $f(x) = x^2$ is a function that takes one argument x, which is a variable that could have any value,[13] and squares that number. In programming terms, we would say that the function `return`s a value x^2. Thus, typing `f(x)` evaluates to x^2, just like typing `x*x` would evaluate to the value x^2. We can also make functions that do some operation but don't return anything!

[13] We could call this argument anything we wanted–x or y or `bread`–and the function would be the same.

input → FUNC → output

Anatomy of a user defined function

"keyword arguments"

inputs

```
def your_function_name(arg1, arg2 ... kwarg = default_value ...):
    ''' This is called a 'docstring' and will pop up if you
        type your_function_name? in a Jupyter cell

        Start with a general description of the function

        Parameters
        ----------
        write descriptions of the function inputs here

        Returns
        ----------
        write descriptions of what the function returns here
    '''
    # operate on the arguments

    return ...
```

define

docs

doc string ↑ ignored

your-fun-name?

$x \rightarrow x^2$ return x^2

input argument

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

return value

for quadratic:

a, b, c → x +/-

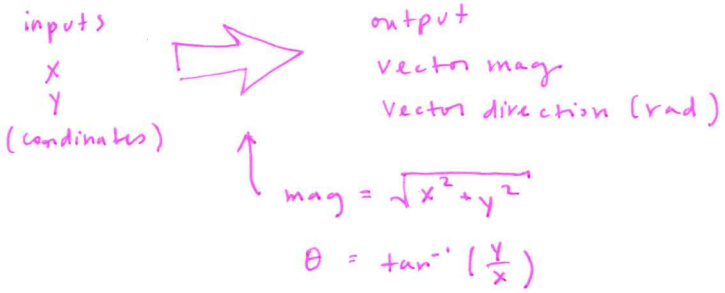
```
def quad(a,b,c):
    x_pos = (-b + np.sqrt(b**2 - 4*a*c))/2/a
    x_neg = (-b - sqrt(b^2 - 4ac))/2a

    return x_pos, x_neg
```

pseudo code

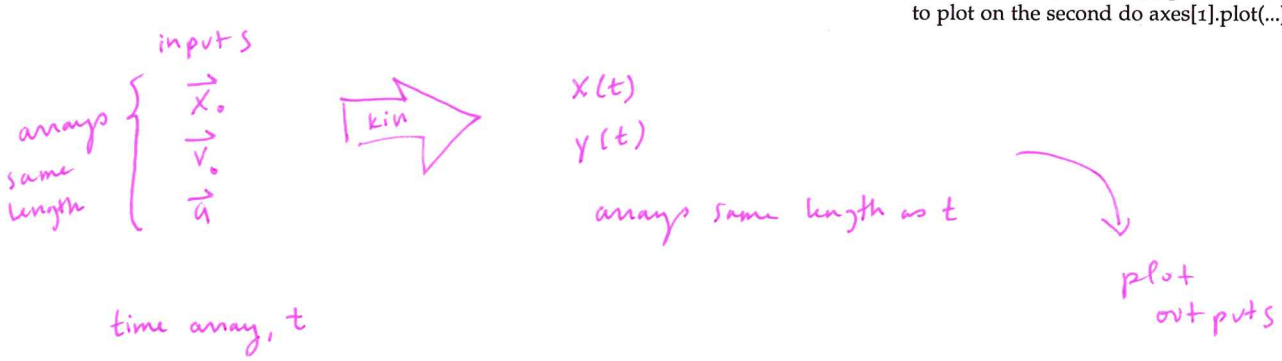
1. Write and test a user defined function called `pythagorean` that computes the hypotenuse of a right triangle given the side lengths a and b.

2. Write and test a user defined function called `xy2magdir` that takes in the components of a 2D vector and returns the vector's magnitude and standard angle (in radians). [14]



[14] Hint: You could call `pythagorean`! Also, you might want to look up the documentation for `np.arctan2()`. Note that the y component goes first!

3. Challenge: Write and test a function called `kinematics` that takes in the initial position and velocity of an object, the acceleration the object will experience, and an array of times. The function should return arrays of the position and velocity of the object evaluated at each of the times in the time array. Call your function with some input values and plot the resulting position and velocity vs time on a pair of subplots: `fig,axes = plt.subplots(2,figsize = [6,6])`. [15]



[15] This will create two subplots. To plot on the first one, do axes[0].plot(...) and to plot on the second do axes[1].plot(...)

Activity 2: Introduction to Conditionals

So far, we have two ways to make the control of the program move in a non-sequential way: loops let us execute the same code over and over, and functions let us execute a specific block of code whenever we specify. Now we'll see how we can *skip* some lines of code. `if` / `elif` / `else` statements allow your code to branch off in different directions based on the validity of some statements.

Figure 2: Choose a path!

Anatomy of an `if/else`

```
if statement:
        # if statement is true, do some stuff
elif statement_2:
        # if statement was false but statement_2
        # is true, do this stuff
else:
        # if neither of the above statements
        # are true, do this
```

Example:

```
x = 4

if x < 0:
        print('Your number is negative.')
elif x==0:
        print('Your number is zero.')
else:
        print('Your number is positive.')

print('I am done now!')
```

You don't have to go through all the iterations of a `for` loop. Often, loops are combined with conditionals, and the loop ends when a certain condition is met. The `break` keyword causes the control to immediately exit a loop, regardless of whether it is done iterating over all of the values specified by the `in` keyword.

Conversely, the `continue` keyword immediately starts the next iteration, ignoring the rest of the code in the loop.

`if/else` in a `for` loop

```
for n in np.linspace(2,3,101):
        natural_log=np.log(n)
        if natural_log >=1:
                break
print(n)
```

Example:

```
for n in [-4,36,4,-1,25]:
        if n < 0:
                continue
        print(np.sqrt(n))
```

Let's practice using loops, conditionals, and functions in combination.

1. You want to up your tennis game by determining at what angle you should serve the ball to land perfectly in the court. You measure the speed of your serve and find it to be 25 m/s. You write some code (using a user-defined function!) to calculate the trajectory of your serve as a function of angle. Find this code (`tennis.py`) on Canvas and copy-paste it into your notebook.

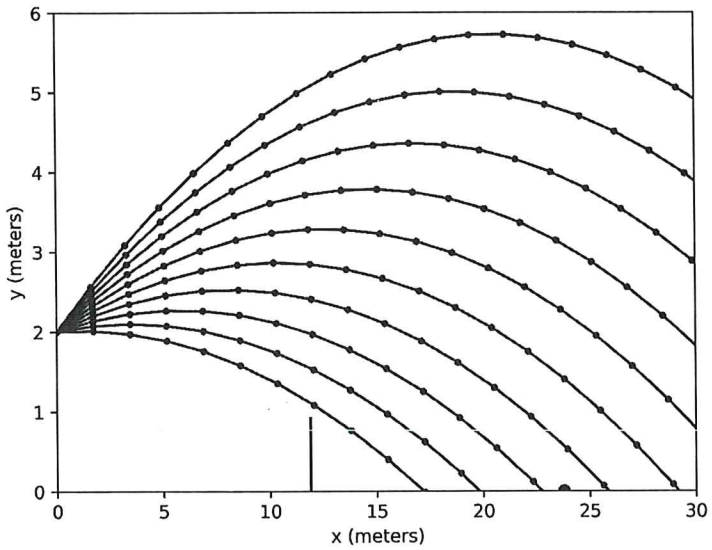
2. Talk to your neighbor to make a plan for how you will produce the following plot. (A skeleton of the plotting code is provided, including plotting the net and the back line of the court). Your code should iterate through[16] 20 different angles between -20 and 20 degrees relative to the horizontal.[17]

   - If the ball lands outside the court,[18] plot the trajectory in red.
   - If the ball lands inside the court, plot the trajectory in green.
   - If the ball would fail to pass over the net, do not plot the trajectory at all.

[16] This implies you should use a loop.

[17] This is a perfect job for `np.linspace`!

[18] This implies you will need a conditional.



3. Once you have discussed with your neighbor, work together to implement your solution and reproduce the plot.

4. The axes on the plot above are a bit misleading, since the meters on the x axis appear much smaller than the meters on the y axis! Use the command `ax.set_aspect('equal')` to set the aspect ratio correctly.

No attendance today!

OPEN JUPYTER

Lecture 2: Simulating Dipole Forces

OUR GOAL IN PYTHON FOR THE SEMESTER will be to simulate electromagnetic fields. This alternative computational view of the material will allow us to deepen our understanding of electromagnetism and extend it to realistic systems. Today we will build a simple simulation of dipole forces using skills we learned last semester in PS12a.

Why simulate?

- help visualize concepts
  microscopic
  integrals → sums
- help us with hand math
  complicated systems
- confirm results

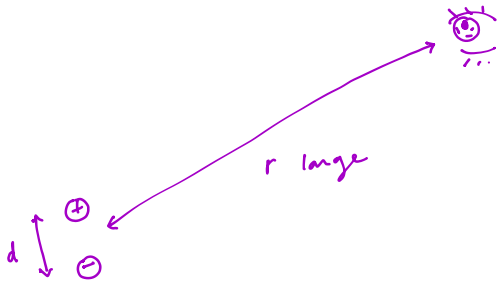
Detailed Learning Objectives

After this lecture, you should be able to...

Python skills review

1. Use functions to perform repeat calculations with varying parameters
2. Represent and work with vector quantities in Python
3. Plot the results of a calculation with independent and dependent variables
4. Visualize dipole forces

+ vector review



Announcements

- For lectures, there will be pre-activities for you to do by class. These are graded on completion and go towards participation points
- HW1 due Friday, September 12 at 5pm
- Course calendar ~~has~~ will have all the office hours/helproom and will be kept current. Labs start next week.
- Questions regarding course structure, assessments?

Calculating forces using functions

Coulomb's Law

$$\vec{F}_{12} = k \frac{q_1 q_2}{|r_{12}|^2} \hat{r}_{12}$$

magnitude; direction unit vector

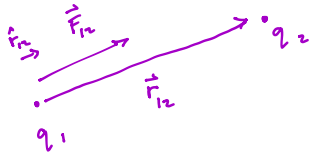
1. You probably have an idea of how to calculate the force between two charged particles in a Python notebook. Try it out for the following values:

   Coulomb

   - $q_1 = 1 \times 10^{-6}$ C is the charge of the first particle
   - $q_2 = -1 \times 10^{-6}$ C is the charge of the second particle
   - $r_1 = -10 \times 10^{-3}\hat{x}$ m is the position of the first particle
   - $r_2 = 10 \times 10^{-3}\hat{x}$ m is the position of the second particle
   - $k = 8.99 \times 10^9$ Nm^2/C is the Coulomb constant

   The magnitude of the electrostatic force between two charged particles is:

   $$F_{1,2} = \frac{kq_1q_2}{d^2}$$ ← magnitude $d = |r_{12}|$



both q's positive, repulsive force
both q's negative, repulsive force
one positive and one negative, attractive force

2. Now, what if you want to change some of the values in the problem? For example, what if we changed q_2 to a positive charge?

   Don't recopy your code. Instead, write a function.

Designing a Function

(a) Start by writing down the body of the function (the code you want it to execute)

$d = r_2 - r_1$
$F_{12} = k q_1 q_2 / d^2$

(b) What are the things we want to vary in the system every time? Make those parameters of the function.

← q_1 q_2 r_1 r_2

(c) What will stay the same every time? Leave those alone and make sure they're well defined inside the function body.

← k

(d) What do we want the function to return? Add a return line that includes all of it.

return F_{12}

(e) Try calling your function with some different parameters. Do you get what you expected?

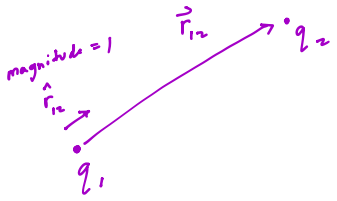
```
def fun-name(params):
    body
    return
```

Vector Review

We just calculated the magnitude of the electrostatic force, but we know that force is a vector quantity.

$$\vec{F}_{1,2} = \frac{kq_1q_2}{|\vec{r}_{1,2}|^2}\hat{r}$$

direction



We use vectors to succinctly represent quantities with both magnitude and direction. Most vectors in this class will be in two dimensions. They can be written many ways:

unit vectors

1. $\vec{v} = v_x\hat{x} + v_y\hat{y}$
2. $\vec{v} = (v_x, v_y)$ coordinate (useful for code)
3. $|\vec{v}| = \sqrt{(v_x^2 + v_y^2)}$, standard angle $= \arctan\frac{v_y}{v_x}$

magnitude direction

When using magnitude-direction form, you must specify what the angle is relative to, or use terms such as "north of east". Standard angle is counterclockwise from the $+x$ axis.

All three methods are correct, but I prefer option 1, because that is the method that makes it most clear that

$\hat{x}$ and $\hat{y}$ are independent of each other.

This means if I have an equation like this:

$$a\hat{y} + b\hat{x} = v_x\hat{x} + v_y\hat{y}$$

I can write it as two separate equations:

$b\hat{x} = v_x\hat{x}$ $\qquad$ $a\hat{y} = v_y\hat{y}$

$b = v_x$ $\qquad$ $a = v_y$

Activity: Vectors in Python

We have many ways to represent vectors on paper. How should we represent them in code? Just like in any problem we want to solve, we need to first define a coordinate system. We will usually see the typical 3-dimension xyz coordinate system in this class, although you could imaging defining a cylindrical or spherical system. Once we have decided what our coordinates are, then we can store the values for each coordinate in an array.

1. For example, can you write an array to represent the vector $\vec{v} = 3\hat{x} + 5\hat{y}$?

   (3, 5) → [3, 5]

2. Now add your vector $\vec{v}$ to a new vector $\vec{u} = 6\hat{x} + 2\hat{y}$. What is the result?

   [3, 5] + [6, 2] = [9, 7] element-wise!

3. What happens if you multiply your vectors in python? What would happen if you tried to multiply them on paper?

   [3, 5] * [6, 2] = [18, 10] element-wise!

   very convenient, but not a real vector product

   vectors don't actually multiply element-wise.
   instead need dot product or cross product.

When we work with vector quantities, we need to keep track of separate x and y components. Can you modify your function to have it return a vector quantity?

Function outputs

Functions can return multiple outputs, of any type.

Example:

```
def my_function(input_1, input_2):

    output_sum = input_1 + input_2
    output_product = input_1 * input_2
    output_string = "Hello, world!"

    return output_sum, output_product, output_string

a, b, c = my_function(5, 3)
```

The example function has three outputs, two numbers and a string. They will appear in the order specified.

1. In the example above, what are the values assigned to `a`, `b`, and `c`?

   a = 8 b = 15 c = "Hello, world!"

2. Now modify your Coulomb force function to return a vector value.

   inputs
   q_1 q_2 — numbers / scalars
   r_1 r_2 — vector: $[r_1, 0]$, $[r_2, 0]$
   calc $\vec{F}_{12}$
   calc F_x, F_y
   output: vector $[F_x, F_y]$

   see notebook!

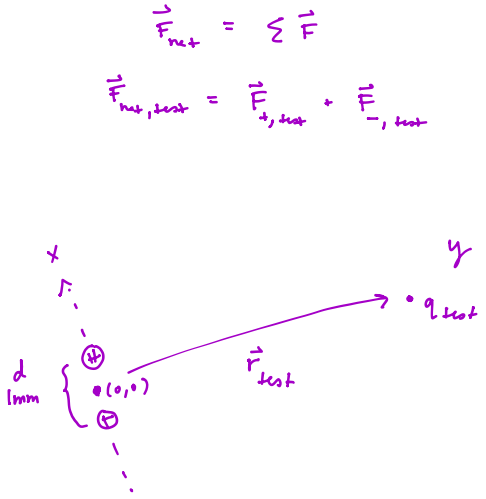
Building the dipole distribution

a distribution is two or more charges in some spatial relation

Now that we have a function that gives the vector valued force between two charges, how can we use it to represent a *distribution of charges*? We will spend much more time on charge distributions next week, with conceptual tools that will simplify our calculations and considerations (electric potential). But, the concept holds here for this simple two-charge system. We know from last semester that forces obey the law of superposition – that is, the total net force on an object is the sum of individual forces on an object. We just need to consider our test charge q_{test} and add up all the forces acting on it.

Consider a test charge q_{test} with charge 1×10^{-6} C at the position $(0, 1)$ cm from the origin. There is a dipole made up of one positive charge 1×10^{-8} C and one negative charge -1×10^{-8} C, located on the x-axis and centered on the origin with an x spacing of 1 mm.

Calculate the force on q_{test} as a result of the dipole, and explore the effect of the dipole separation on the force.



work plan:

write down system parameters

q_+ r_+ q_- r_- q_{test} r_{test} k

write function $\vec{F}_{12}$

apply F_{12} → $F_{+,test}$ $F_{-,test}$ → sum

we could repeat the process for many charges in a distribution, perhaps with a loop!