cs109a_hw0

August 14, 2023

```
[ ]: # Initialize Otter
import otter
grader = otter.Notebook("cs109a_hw0.ipynb")
```

1 CS109A Introduction to Data Science

1.1 Homework 0

Harvard University Fall 2023 Instructors: Pavlos Protopapas & Kevin Rader

---

1.2 Welcome to CS109a!

CS109A attracts students with diverse backgrounds, interests, and goals. As a result, students' experiences with the course prerequisites can be quite varied. Some tudents may, for example, have extensive coding experience, but perhaps can't remember how to represent a projection with linear algebra. Others may have a strong statistics background, but are less confident when it comes to Python.

Homework 0 is a way to assess your comfort with the skills and concepts foundational to the course. It is not expected that you should be able to recall everything from memory while working through the assignment. What maters is the ability to use the context and resources available to fill in any gaps and work toward your goal. This is what we do 'in the real world' after all! However, if you do find HW0 to be too challenging, it is perhaps best to spend more time with the prerequiste material and register for CS109A the next time around.

1.2.1 Table of contents

- **3** Math and Statistics
  - Linear Algebra
  - Partial Derivatives
  - LaTeX
  - Joint & Conditional Probability
  - Random Variables & PDFs

1.3 HW0 User Guide

HW0 is a large document, but there isn't much coding required to complete it.

- If you already know how to create a virtual environment from the provided .yml file then you can jump right to part 2 (though you should read through part 1 eventually).
- The Python content covered in part 2 will start being put to use as early as the first lab and HW assignment
- The math topics in part 3 will begin to make an appearance by HW2

2 Part 1: Setting the Stage

A data scientist's toolkit is a modular collection of various technologies. Taking the time to familiarize yourself with these different components and how they work together will help you be more effective and open new possibilities to you. Students will also be provided with computing environments they can connect to through their web browser that are sufficient for completing all course work. *But everyone is expected to be able to create similar environments on computer systems of their own.*

Prerequisites for this section include: - Using a terminal application to run shell commands - Facility with the filesystem: reading/writing files, navigating the file tree, etc.

2.1 Why Python

Our primary tool in this course is the Python programming language. It has a reputaion for having a clean, readable syntax, sometimes described as being "like pseudocode you can run." Python is also distributed with a large standard library. This "batteries included" approach has made Python one of the most popular general purpose programming languages around today.

As a result, the community of Python users is enormous with many contributing to open source projects. The Python Package Index (PyPi) hosts over 465,000 different projects! And that's not all. There are also specialized repositories for scientific computing packages, such as conda-forge. And then there are all the other public projects hosted on Github, Gitlab, and elsewhere. Python gives us access to this software ecosystem.

2.2 The Shell & Terminal

Most modern operating systems make use of Python, so you likely already have some version installed on your machine!

You can check by querying your **shell**. The shell is a programming language that gives users access to the operating system's services. An interactive shell runs in a read, evaluate, print, and loop

sequence (REPL). These days, we interact with the shell through a terminal emulator, which we'll simply (though not quite correctly) refer to as the **terminal**.

The terminal is the program that grants you a text interface through which you can enter commands to the interactive shell.

> **Note:** Windows user have PowerShell, but it lacks many useful commands. This will work for the course, but please consider trying out Git Bash, or better still, Windows Subsystem for Linux (WSL) for a true UNIX shell experience!

2.2.1 Checking for Python on Your System

Let's put a (UNIX) shell command to use. PowerShell users should try the similar Get-Command.

The `which` command returns the pathnames of the files (or links) which would be executed in the current environment, had its arguments been given as commands in the shell. It does this by searching the PATH environmental variable for executable files matching the names of the arguments.

```
which python
```

You'll be shown a path like `/usr/bin/python`.

If it was found, you can envoke `python` and ask for its version.

```
python --version
```

> You can also run shell commands from within a Jupyter notebook by prepending a `!`

```
[ ]: # !command sends `command` to the shell that launched Jupyter
     !python --version
```

You should see some version >= 3.8

> **Note:** If your call to `which` threw an error saying it couldn't find `python`, or if your version is < 3.8, then you're going to need to install Python. Don't worry. The installation steps in the next section will take care of this.

Now you can use Python's *own* interactive shell, or "REPL," in the terminal by envoking `python`. You'll have access to modules from the standard Python library like `datetime`, `math`, `collections`, `itertools`, `re`, etc. But we'd also like to install 3rd party packages.

However, we need to take care in *how* we install packages because: 1. Some of your coding projects will require conflicting versions of the same package that can't coexist. 2. If your operating system makes use of Python, altering the system installation could have unintended consequences. 3. We can't hope to make our results reproducible without documenting the specific software versions that produced them.

What we need is some way to easily create and manage multiple isolated Python installations, each with their own set of package dependencies.

2.3 Virtual Environments & Package Managers

A virtual environment is a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages. Just what we're looking for!

Most Python distributions will include: * venv - to create and manage virtual environments * pip - to download and manage 3rd party packages (from PyPi by default)

For more information on these tools, you can read the Python tutorial entry on virtual environments & packages

But we can generalize this idea of a Python virtual environment to *all languages*. This is useful because our data science projects may have dependencies outside of Python itself.

2.3.1 Conda - Language Agnostic Environments

Conda provides cross-platform package, dependency, and environment management for any language—Python, R, Ruby, Lua, Scala, Java, JavaScript, C/ C++, etc.

You *could* get conda by installing Anaconda, a Python distribution that includes conda and many other packages, but it is very large any many of the packages you likely won't use. A better option might be miniconda, a more minimal distribution that includes Python, pip, conda, and a few other useful packages.

But conda can be slow...

Conda is written in Python and its dependency resolution method can be rather slow. The Mamba project is a reimplementation of the conda package manager in C++. It resolves dependencies much faster, makes use of multi-threading for parallel package downloading, and uses an almost identical command syntax to conda, so switching over if you're used to conda should be easy.

For these reasons *we recommend using some version of* ***mamba*** *over conda*. And the easiest way to get mamba on your system is through `micromamba`.

2.3.2 Micromamba - Package & Environment Manager Installation

Micromamba is a single executable we can use to bootstrap conda environments on Linux, macOS, or Windows. It will even install the desired version of Python for your environment. Consult the documentation for up-to-date installation instructions on your OS.

In the final step of the installation, micromamba will alter your shell's configuration file so that that both the `micromamba` executable and the directory used to store your virtual environments are added to your PATH.

We're now ready to create an environment for CS109A!

Creating an Environment We've provided a `cs109a.yml` file that lists the packages we'll be using in the course. It should be in the same directory as this notebook.
Go ahead and take a peek! it's just a plain text file. YMAL (sometimes abrevieated 'YML') is a "human-friendly" data serialization language commonly used in configuration files with a minimal syntax.

We can instruct micromamba to create a new environment from our yml specification:

```
micromamba create -f cs109a.yml
```

Activating an Environment Once all the packages have been installed, we can activate our new environment:

```
micromamba activate cs109a
```

You'll see your shell prompt now shows your active environment. You can use `deactivate` to turn it off.

2.4 Jupyter

Much of the course material, lecture supplementals, labs, exercises, assignments, etc., will be in the form of Jupyter Notebooks. They allow for the embedding of runnable code in a document of formatted text. For more about this general approach, take a look at the literate programming paradigm.

If you created your environment with the CS109A yml file above, you will already have the Jupyter Lab package installed. To explore the Jupyter Lab interface, you can look at their tutorial & documentation.

You can launch Jupyter from your terminal with:

```
jupyter lab
```

The output in the terminal will display an address you can paste into your browser to access the now running Jupyter Lab server, but it should open in your default broswer automatically.

> You can edit your shell's config file to make interacting with these tools more comfy.
> A few ideas: - alias mm="micromamba" # shorten micromamba command - alias

109="micromamba activate cs109a" # fast activate - 109 # activate cs109a environment for every new terminal (requires previous alias) - alias jl="jupyter lab" # fast lab

Q1 - Check your installation

Assume this Jupyer notebook was launch from inside your new cs109a environment, you should now have access to all the packages listed in the cs109a.yml file.

[ ]:
```
# See the "import ... as ..." contructs below:
# they're aliases/shortcuts for the package names. As a result,
# we can call methods such as plt.plot() instead of matplotlib.pyplot.plot()
import numpy as np
from scipy import stats
import pandas as pd
import matplotlib.pyplot as plt
```

[ ]:
```
grader.check("q1")
```

3 Part 2: Python Poetry Challenge

CS109A assumes students are already familiar with the basics of Python programming.

To assess your level of Python preparedness, you'll be creating a **random poem generator** from a collection of Emily Dickinson poems made available through Project Gutemberg.

Core Python skills utilized in completing this challenge include:

- File I/O
- Data structures
  - strings
  - lists & slicing
  - dictionaries
- Logical operators
- Iteration
- Functions
- Classes
- Debugging
- Reading documentation
- Interpretting code

Q2.1 - Read in the file

The document we'll be working with is a digital transcription assembling three early Emily Dickinson poetry collections for which the U.S. copyright has expired.

We've included this document with HW0 as a plain text file. The filename is`pg12242.txt` and it's located in the `data` subdirectory.

Read the contents of `pg12242.txt` into a variabled named `text`. The arcane file name is a stipulation of The Project Gutenberg License included in the file.

Take the contents of the file and save the substring containing the first 10,000 characters as the variable `head` and the substring containing the final 25,000 characters as `tail`.

[ ]:
```
...
    text = ...

head = ...
tail = ...
```

[ ]:
```
# We can inspect the beginning of the content
print(head)
```

[ ]:
```
# final 25,000 characters
print(tail)
```

[ ]:
```
# entire contents
print(text)
```

[ ]:
```
grader.check("q2.1")
```

Q2.2 - Remove non-poem content

Keeping in mind that this is a compilation of three publications (or 'series') and inspecting the `head` and `tail` sections of the text we printed above, we can identify some useful structure in the document:

- Project Gutenberg: Header - Metadata
- Series 1: Preface
- **Series 1: Poems**
- Series 2: Preface
- **Series 2: Poems**
- Series 3: Preface
- **Series 3: Poems**
- Project Gutenberg: Footer - License

Recognizing and making use of structure is a large part of data science.

Create three variables, `series1`, `series2`, and `series3`, each consiting of one of the 3 sections of `text` dedicated to the poem content, *exluding* prefaces and the Project Gutenberg metadata and license. Be sure to strip any trailing or leading whitespace from the final strings.

> **Hints:** - Each preface section begins with `POEMS\n` - Each poem section after a preface starts with a numbered subsection beginning with `I.` - The Project Gutenberg License section begins with `End of Project Gutenberg` - You may import standard library modules such as re to help here, but this task can also be done using only Python built-ins with no additional imports. - Review indexing and slicing if needed - The string `.index()` method returns the index of the first occurance of a target string in a source string.

```
series1 = ...
series2 = ...
series3 = ...
```

```
grader.check("q2.2")
```

We will then join the three series into a single string using `join`. For the delimiter we use six newline characters. The reason for this choice will soon become clear.

```
# join series into single string connected by "\n\n\n\n\n\n"
poems_text = ('\n'*6).join([series1.strip(),
                            series2.strip(),
                            series3.strip()])
```

And we can check to see that we only have the poem content.

```
print(poems_text[:445])
print('\n---\n')
print(poems_text[-275:])
```

Q2.3 - Remove annotation

The first poem contains some editor annotation in square brackets.

```
print(poems_text[:200], '...')
```

This is the only annotation like this in the text. You can see that these are the only two square brackets that appear in the string.

```
# Confirm that brackets occur only once
assert (poems_text.count("[") == 1) and (poems_text.count("]") == 1)
```

Create `poems_text_clean` which has this annotation (and the additional blank line below it) removed.

Hints: The string method `split` could be put to use here

```
poems_text_clean = ...
```

```
print(poems_text_clean[:100], '...')
```

```
grader.check("q2.3")
```

Q2.4 - Identify sections and numbering

Now let's remove some of the other content that was *not* the work of the original author: the section titles and poem numbers. It is helpful to notice that all such lines begin with roman numerals.

Implement the `startswith_rn` function which we can use to identify lines that serve as section titles or poem numbering.

> **Note:** The function does not need to be sophisticated enough to reject 'invalid' roman numerals such as 'IIIV.' We are assuming (correctly) that these sorts of strings do not

occur and so the function only needs to be powerful enough to recognize those numerals that do occur at the start of headers in the text while excluding the titles and the content of any of the poems.

[ ]:
```
def startswith_rn(s: str) -> bool:
    '''
    Returns: True if `s` starts with a roman numeral and False otherwise
    Ex:
        startswith_rn("III. NATURE.") -> True
        startswith_rn("I'm nobody! Who are you?") -> False
    '''
    ...
```

[ ]:
```
grader.check("q2.4")
```

Q2.5 - Remove sections and numbering

Use your new `startswith_rn` function to help you remove the lines starting with roman numerals from `poems_text_clean`. Strip any whitespace from the ends of the final result and store the string in the variable `poems_text_nonum`

[ ]:
```
poems_text_nonum = ...
```

[ ]:
```
print(poems_text_nonum[:500])
```

[ ]:
```
grader.check("q2.5")
```

Q2.6 - Create a list of poems

Next, we'd like to split the text up into a list of poems.

To investigate how we might do this it is important to view all the charcters in the text. The `pprint` function from the standard library allows us to display the text with the newline characters `\n` while also making the output easier to read by including the line breaks as well.

[ ]:
```
from pprint import pprint
```

[ ]:
```
pprint(poems_text_nonum[:2000])
```

Notice how the poems are separated by multiple newline characters. Further exploration shows the minimum number of newlines between poems to be *6*.

Use this information to create `poem_list` where the elements of the list are the poems in the order they occur in `poems_text_nonum`. For now, we will include the title if the editors provided one as part of the poem. Each poem should be stripped of leading and trailing whitespace.

Hint: Be sure to exclude any empty strings elements from your final `poem_list`.

[ ]:
```
poem_list = ...
```

[ ]:
```
print(poem_list[109])
```

[ ]:
```
print(f"There are a total of {len(poem_list)} poems.")
```

[ ]:
```
grader.check("q2.6")
```

Q2.7 - Create a poem dictionary

We now have all the poems separated, but we might want some means of accessing specific poems without having to remember their arbitrary position in the list. This is where the key-value pair structure of dictionaries will help.

We will create a dictionary, `d`, where the keys are the poem titles and the values are the poems themselves.

If the editors did not provide a title, we'll use the first line of the poem as the key.

Recall that some poems were given identical titles. And we know dictionary keys must be unique. Our approach will be to use incrementing numerical labels to denote poems with duplicate names beyond the first encountered. The titles should then look like: "ETERNITY.", "ETERNITY. (2)", "ETERNITY. (3)", etc.

We have provided one possible implementation that is nearly complete. Only one line needs to be filled in.

You are also welcome to create your own solution from scratch.

[ ]:
```
# helper functions
# Answers: what number should I increment to having now seen a duplicate?
next_title_num = lambda x: x[-2] + 1 if x[-2].isnumeric() else 2
# Answers: what was the number of the previous poem with this title?
prev_title = lambda d, k: sorted([k for k in d.keys() if k.startswith(k)])[-1]

def update(d: dict, k: str, v: str) -> None:
    '''
    Adds key-value pair 'k' & 'v' to dictionary 'd'
    Uses helper functions to increment key string if key already exists
    Dictionary is changed inplace; Returns None.
    '''
    if d.get(k):
        k = f'{k} ({next_title_num(prev_title(d,k))})'
    d[k] = v
```

[ ]:
```
is_editor_title = lambda x: x.endswith('.') and x.isupper()
has_editor_title = lambda x: is_editor_title(x.split('\n')[0]) # check 1st line␣
 ↪for editor title

d = {}
for p in poem_list:
    # first line will always be the key
    k = p[:p.index('\n')]
    if has_editor_title(p):
        # find string that should be the value (poem minus title)
```

```
        # YOUR CODE HERE
        p = ...
    # add new new pair to dictionary
    # update function handles altering the key if neccessary
    # (i.e., incrementing the numerical suffix of the title)
    update(d, k, p)
```

[ ]:
```
print(d['HOPE.'])
```

[ ]:
```
grader.check("q2.7")
```

Q2.8 - Most frequent long words

We can also ask statistical questions of the data such as, "what are the most common words?"

Find the 10 most frequently used words longer than 10 characters. Ignore case and any punctuation at the end of the word when counting. Consider only the poem text and not the editor provided titles. Store the top 10 words in a list called `top_words` in order of decreasing frequency.

You may find the `punc` variable below useful. It is the set of the punctuation characters occurring at the end of word in the text. You can assume that no words end with multiple punctuation characters.

[ ]:
```
# punctuation characters to remove when counting words
punc = {w[-1] for p in d.values() for w in p.lower().split() if not w[-1].
 ↪isalpha()}
punc
```

[ ]:
```

top_words = ...
```

[ ]:
```
top_words
```

[ ]:
```
grader.check("q2.8")
```

Q2.9 - Random Poem Generator

We're now ready to create our random poem generator.

Complete the `PoetryCollection` class below by defining a `__init__` method. It should take the author's full name and your poem dictionary as arguments and define the class attributes `author`, `collection`, and `size` as described in the docstring.

Then instantiate a `PoetryCollection` using our Emily Dickinson data and call it `poems`.

[ ]:
```
class PoetryCollection():
    """
    Attributes
    ----------
    author : str
        full name of author
```

```
    collection : dict
        dictionary of (title, poem) key-value pairs
    size : int
        number of poems in collection

    Methods
    -------
    random_poem(seed: int = None) -> str
        returns random poem; use seed for reproducibility (default seed=None)
    """
    ...
        self.author = ...
        self.collection = ...
        self.size = ...

    def random_poem(self, seed : int = None) -> str:
        rng = np.random.default_rng(seed)
        return str(rng.choice(list(self.collection.values())))
```

[ ]:
```
poems = ...
```

[ ]:
```
print('Author:', poems.author)
```

[ ]:
```
print('Number of poems in collection:', poems.size)
```

[ ]:
```
print(poems.random_poem(seed=109))
```

[ ]:
```
grader.check("q2.9")
```

4 Part 3 - Math and Statistics

4.0.1 Linear Algebra

In data science we frequently encounter problems that can be represented (or approximated) by a system of linear equations.

In this toy example, you've started to collect data about some books in your collection:

Title	New/Cover Price ($)	Years Since Publication	Has Underlining (Yes/No)	Used Price ($)
Perceptrons (Expanded)	35	35	1	15
Social Empiricism	20	21	0	11
Philosophical Investigations	42	13	1	27

`Has Underlining` is 1 if the book is marked up with notes and 0 otherwise.

We will (very naïvely) assume that a book's used price is a linear combination of the three recorded features which capture information about the book's original price, age, and condition.

That is, there exist coefficients β_1, β_2, and β_3 such that:

$$\beta_1 \cdot \text{new price} + \beta_2 \cdot \text{years since publication} + \beta_3 \cdot \text{has underlining} = \text{used price}$$

If you knew the coefficient values then these three features could be used to *predict* the used price of a book for which that data is unavailable, or perhaps to check if a newly observed used price is fair.

If we represent the set of unknown coeffients as a column vector

$$\beta = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

and represent the three features of our books as a matrix, X, and the used prices as a column vector, y

$$X = \begin{bmatrix} 35 & 35 & 1 \\ 20 & 21 & 0 \\ 42 & 13 & 1 \end{bmatrix} \quad y = \begin{bmatrix} 15 \\ 11 \\ 27 \end{bmatrix}$$

then we can calculate the linear combinations for all rows in X with a single matrix multiplication:

$$X\beta = y$$

Or, written out with the full matrices and vectors:

$$\begin{bmatrix} 35 & 35 & 1 \\ 20 & 21 & 0 \\ 42 & 13 & 1 \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = \begin{bmatrix} 15 \\ 11 \\ 27 \end{bmatrix}$$

Q3.1 - Arrays & Matrices with Numpy

Use Numpy's ndarray constructor, `np.array()` to create the X matrix and y column vector as shown above. It is common to pass the constructor a list or list-of-lists as an argument. Don't be afraid to check the documentation. >**Hint:** Remember, y is a *column* vector. You can check the dimensions of an ndarray with the `.shape` attribute. y's shape should be `(3,1)`. The array's `.reshape()` method is one way to address this. When in doubt, a quick internet search will often turn up Stack Overflow answers to similar questions.

[ ]:
```
X = ...
y = ...
```

[ ]:
```
grader.check("q3.1")
```

Q3.2 - Matrix inverse

We'd like to solve for β algebraically, and we are in luck! Because our X matrix is square and full rank we can take its inverse, denoted X^{-1}.

A matrix multiplied by its inverse is the identity matrix of the same shape: $A^{-1}_{n,n}A_{n,n} = I_{n,n}$

And of course, multipling a vector or matrix by the identity matrix leaves it unchanged (provided the shapes allow for a valid matrix multiplication): $IA = A$

This is all we need to solve for β. Just multiply both sides of the equation by X^{-1} on the left and simplify:

$$\begin{aligned} X\beta &= y \\ X^{-1}X\beta &= X^{-1}y \\ I\beta &= X^{-1}y \\ \beta &= X^{-1}y \end{aligned}$$

With the information presented above, use Numpy to find β and store it in `beta`. >**Hints:** Numpy has a `linalg` sub-module containing many useful functions related to linear algebra. You'll need (1) a method to take the inverse of a matrix and (2) a means of calculating a matrix product. The documention is very thorough.

[ ]:
```
beta = ...
```

[ ]:
```
# display resulting betas
beta
```

[ ]:
```
grader.check("q3.2")
```

Q3.3 - Making a prediction

Now let's use your β vector to predict the used price of a book not in our original data.

Title	New/Cover Price	Years Since Publication	Has Underlining	Used Price
Linear Algebra Done Right	44	7	0	?

Create the ndarray `new_box` to represent the three feature values of the book.
Then multiple it by `beta` and store the results in `pred_price`.

[ ]:
```
new_book = ...
pred_price = ...
```

[ ]:
```
print(f'Predicted Price: ${pred_price[0]:.2f}')
```

[ ]:
```
grader.check("q3.3")
```

We will almost never be so lucky as to be working with a square data matrix in a problem like this. And our assumption that used prices *are indeed* linear combinations of the feature values, with no variation, is far too strong. In the course we will see ways of relaxing both these assumptions.

4.0.2 Multivariate Calculus

Q3.4 - Partial Derivatives for Optimization

Calculus is a powerful tool for optimization. For example, imagine we have a "loss" or "cost" function, $f(x, y)$, where x and y are parameters we can control. The goal is to find the values of x and y that minimize f.

$$\arg\min_{x,y} f(x, y)$$

We know that a function's derivative is zero at extrema (provided it is differentiable there). Taking the partial derivatives of f with respect to each of the parameters, setting them equal to zero, and solving for the respective parameter values can then provide us with potential minima. We can confirm that a set of parameter values correspond to a minima if we can also show that the 2nd derivatives evaluated at those values are all > 0 (again, provided the original function is twice differentiable there).

Your goal is to use this strategy to find the x and y that minimize the loss function:

$$f(x, y) = xy + (y + 1)^2 + 2x^2 + 4$$

You can also do this using the gradient operator, $\nabla \cdot$

One of the main benifits of working with jupyter notebooks is the ability to display nicely formatted mathematical notation. Use LaTeX in a markdown cell(s) below to show your work. Then store the x and y values you find that minimize f in the variables `x_min` and `y_min` respectively.

You can find an intro tutorial here if your LaTeX is a bit rusty.

Type your answer here, replacing this text.

[ ]:
```
x_min = ...
y_min = ...
```

[ ]:
```
grader.check("q3.4")
```

4.0.3 Probability & Statistics

As we move through the course we will look at a probabalistic method for modeling data. There, we assume that the value of interest, y, for each observation, x, is a random variables described by a specific family of probability distributions (e.g., Gaussian, binomal, etc.), and that the parameter(s) of these distributions, such as the mean, is related to the values in x. We then try and learn what this relationship is.

But before we get there we should remind ourselves about how to work with probabilities.

Q3.5 - Joint probabilities

First, let's think about the joint probability of several i.i.d. random variables.

What is the probability of rolling three 3's in a row with a fair 6-sided die?
Save the result in `dice_prob`.

[ ]:
```
dice_prob = ...
dice_prob
```

[ ]:
```
grader.check("q3.5")
```

Q3.6 - A 'branching' example

Here's a slightly more involved scenario:
In your pocket you have two coins: a **fair coin** and a **biased coin** that is twice as likely to land heads than tails.

You take a coin from your pocket at random and flip it 4 times. You see:

$$H, T, H, T$$

Given that you were equally likely to choose either coin, what was the probability of this series of outcomes?
Save your result in `coin_prob`.

[ ]:
```
coin_prob = ...
coin_prob
```

[ ]:
```
grader.check("q3.6")
```

Q3.7 - Conditional Probability

What is the probability of observing H, H, H *conditional* on having pull the biased coin from your pocket? Recall that this coin is twice as likely to land heads than tails.

That is, calculate $P(H, H, H|B)$ where B is the event of drawing the biased coin from your pocket.

This time, rather than calculating the value programatically, show your work with LaTeX in a markdown cell below.

Type your answer here, replacing this text.

Q3.8 - Sampling from a PDF with Numpy

Numpy also gives us the ability to sample from probability distributions through its Random Generator class, which can be created with the `np.random.default_rng()` constructor.

[ ]:
```
# instantiate random generator /w seed of 109
rng = np.random.default_rng(seed=109)
```

Let X be a normally distributed random variable with a mean of 4 and a standard deviation of 2. That is:

$$X \sim \mathcal{N}(4, 2)$$

Sample `N` realizations of X using `rng.normal()` and store them in the variable `sample`.
Next, calculate the mean and standard deviation of the sample and store the results in the variables `mean` and `std`.

Try several values for `N`. Start with just 10 and increase the sample size until the sample mean and standard deviation are within 0.1 of those of the data generating distribution.

Hint: check the documention on the generators normal() method.

[ ]:
```
# number of samples
N = ...

# N samples from normal distribution /w mean=4 & std=2
sample = ...
```

[ ]:
```
mean = ...
std = ...
print(f"sample mean: {mean:.2f}")
print(f"sample standard deviation: {std:.2f}")
```

The histogram below visualizes the empirical distribution of your sample as well as the sample mean and standard deviation.

This uses the Matplotlib library which will be one of our main vizualization tools in this course.

[ ]:
```
# provided plotting code
plt.hist(sample)
plt.axvline(mean, c='r', label=f'mean = {mean:.2f}')
plt.axvline(mean+std, c='k', ls='--', label=f'stdev = {std:.2f}')
plt.axvline(mean-std, c='k', ls='--')
plt.title(f'{N} Samples of $X \sim \mathcal{{N}}(4, 2)$')
plt.ylabel('count')
plt.xlabel("$X$ value")
plt.legend();
```

[ ]:
```
grader.check("q3.8")
```

This concludes HW0. Thank you!