Numerical Computing in Python

Lukas Hager

2024-04-01

Overview

Numerical Computing

- We want to use python as a tool to compute economic models
- As python is open-source, many libraries exist for this
  - `numpy`: Workhorse library for vectorized computing
  - `pandas`: Data cleaning and manipulation
  - `scipy`: Statistical computing methods

Learning Objectives

- Understand how to use basic commands in `numpy`
- Grasp why vectorization is valuable and how to use it
- Leverage these tools to solve problems in Economics

numpy

Why do I care?

- You know how to use lists in python
  - What are vectors if not lists?
  - What are matrices if not lists of lists?
- Why bother using a package when we can do whatever is in the package ourselves?

You should care!

- `numpy` provides many advantages
  - **Fast**: interfaces with C/C++, which makes it much faster than doing things yourself
  - **Standard**: you can trust `numpy` to do an operation properly, and more importantly, *in a way that's numerically stable*
  - **Broadcasting**: `numpy` gives us the `ndarray`, which makes matrix and array operations much easier
- Documentation

Vectorization

Basic Problem[1]

- Suppose I want to generate a list of the numbers from 0 to 999,999 and then multiply each value of the list by 2
- We can do this with our existing tools

[1] Taken from Wes McKinney

Basic Problem (Lists)

```
raw_range = range(1000000)
raw_list = list(raw_range)
%timeit raw_list_x_2 = [x*2 for x in raw_list]
```

```
26.7 ms ± 531 µs per loop (mean ± std. dev. of 7 runs, 10 loops
```

Basic Problem (Lists)

```python
raw_range = range(1000000)
raw_list = list(raw_range)
%timeit raw_list_x_2 = [x*2 for x in raw_list]
```

```
26.7 ms ± 531 µs per loop (mean ± std. dev. of 7 runs, 10 loops
```

```python
new_list = []
%timeit for x in raw_list: new_list.append(x*2)
```

```
48.9 ms ± 3.41 ms per loop (mean ± std. dev. of 7 runs, 10 loops
```

Basic Problem (Lists)

```
raw_range = range(1000000)
raw_list = list(raw_range)
%timeit raw_list_x_2 = [x*2 for x in raw_list]
```

```
26.7 ms ± 531 µs per loop (mean ± std. dev. of 7 runs, 10 loops
```

```
new_list = []
%timeit for x in raw_list: new_list.append(x*2)
```

```
48.9 ms ± 3.41 ms per loop (mean ± std. dev. of 7 runs, 10 loops
```

- ms is a millisecond

Basic Problem (numpy)

```python
import numpy as np

raw_array = np.arange(1000000)
%timeit raw_array_x_2 = 2 * raw_array
```

```
856 µs ± 42.6 µs per loop (mean ± std. dev. of 7 runs, 1,000 loo
```

Basic Problem (numpy)

```python
import numpy as np

raw_array = np.arange(1000000)
%timeit raw_array_x_2 = 2 * raw_array
```

```
856 µs ± 42.6 µs per loop (mean ± std. dev. of 7 runs, 1,000 loo
```

- μs is a millionth of a second ($\mu s =$ ms / 1000)
- `numpy` solution is roughly 100 times faster than list solution
- "Do I have to use `np` as a prefix?"
  - Technically no, but do it

Inner Product

Recall that for two $n \times 1$ vectors x_1 and x_2, we can compute the inner product as

$$x_1^\top x_2 = \sum_{i=1}^{n} x_{1i} x_{2i}$$

Inner Product Function (Lists)

```python
def inner_product_slow(x_1: list, x_2: list) -> float:
    """
    Compute the inner product of two lists
    """

    inner_prod = 0
    for i in range(len(x_1)):
        inner_prod += x_1[i] * x_2[i]
    return inner_prod

x_1 = list(range(100))
x_2 = list(range(100, 200))
%timeit inner_product_slow(x_1,x_2)
```

```
5.27 µs ± 86.8 ns per loop (mean ± std. dev. of 7 runs, 100,000
```

np.inner

```python
x_1_arr = np.arange(100)
x_2_arr = np.arange(100,200)
%timeit np.inner(x_1_arr, x_2_arr)
```

```
608 ns ± 5.88 ns per loop (mean ± std. dev. of 7 runs, 1,000,000
```

np.inner

```
x_1_arr = np.arange(100)
x_2_arr = np.arange(100,200)
%timeit np.inner(x_1_arr, x_2_arr)
```

```
608 ns ± 5.88 ns per loop (mean ± std. dev. of 7 runs, 1,000,000
```

```
np.inner(x_1_arr,x_2_arr) == inner_product_slow(x_1,x_2)
```

```
True
```

Exercise: Factorial

Can you rewrite our factorial function using `numpy`? Is it faster? Hint: look up `np.prod`.

Solutions: Factorial

```
%timeit np.prod(np.arange(1,11))
```

```
1.69 µs ± 21.9 ns per loop (mean ± std. dev. of 7 runs, 1,000,00
```

Solutions: Factorial

```
%timeit np.prod(np.arange(1,11))
```

```
1.69 µs ± 21.9 ns per loop (mean ± std. dev. of 7 runs, 1,000,00
```

Our old method:

```
%%time

out = 1
for i in range(1,11):
    out *= i
```

```
CPU times: user 3 µs, sys: 1 µs, total: 4 µs
Wall time: 2.86 µs
```

Matrix Multiplication

If a matrix A has dimension $n \times k$ and matrix B has dimension $k \times \ell$, then they can be multiplied and the resulting matrix AB has dimension $n \times \ell$ and elements given by

$$\begin{bmatrix} a_{11} & \dots & a_{1k} \\ a_{21} & \dots & a_{2k} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nk} \end{bmatrix} \begin{bmatrix} b_{11} & \dots & b_{1\ell} \\ b_{21} & \dots & b_{2\ell} \\ \vdots & \ddots & \vdots \\ b_{k1} & \dots & b_{k\ell} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{k} a_{1i}b_{i1} & \dots & \sum_{i=1}^{k} a_{1i}b_{i\ell} \\ \sum_{i=1}^{k} a_{2i}b_{i1} & \dots & \sum_{i=1}^{k} a_{2i}b_{i\ell} \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{k} a_{ni}b_{i1} & \dots & \sum_{i=1}^{k} a_{ni}b_{i\ell} \end{bmatrix}$$

- Note that this also defines the result of matrix A and $k \times 1$ vector c
- Further, matrix multiplication is not commutative
  - Here, we can compute $A \times B$, but not $B \times A$
  - Even if A, B square (so both have dimension $m \times m$), it is not guaranteed that $AB = BA$

Matrix Multiplication in numpy

```python
A = np.arange(10).reshape((5,2))
B = np.arange(10, 20).reshape((2,5))

print(f'Matrix A:\n {A}')
print(f'Matrix B:\n {B}')
print(f'Product:\n {A @ B}')
```

```
Matrix A:
 [[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
Matrix B:
 [[10 11 12 13 14]
 [15 16 17 18 19]]
Product:
 [[ 15  16  17  18  19]
 [ 65  70  75  80  85]
 [115 124 133 142 151]
```

General Note: Reshaping Arrays

- `numpy` works with arrays – think of these as generalized matrices
  - This encompasses scalars (1×1), vectors $(n \times 1)$, matrices $(n \times k)$, and higher-dimensional objects $(a \times b \times c \times \ldots)$
- Sometimes we want to change the dimensions of an array
  - For example, `numpy` often initializes vectors with dimension 1 (just a length)

```
len_one_arr = np.ones(10)
len_one_arr.shape
```

```
(10,)
```

General Note: Reshaping Arrays

- `numpy` works with arrays – think of these as generalized matrices
  - This encompasses scalars (1×1), vectors $(n \times 1)$, matrices $(n \times k)$, and higher-dimensional objects $(a \times b \times c \times \ldots)$
- Sometimes we want to change the dimensions of an array
  - For example, `numpy` often initializes vectors with dimension 1 (just a length)

```
len_one_arr = np.ones(10)
len_one_arr.shape
```

```
(10,)
```

In some contexts, we want this array to have two dimensions – say, $n \times 1$. We then have to reshape:

```
len_two_arr = len_one_arr.reshape((-1,1))
len_two_arr.shape
```

```
(10, 1)
```

Reshaping using −1 Argument

We can always leave one dimension as -1 to tell `numpy` to just make that dimension whatever is necessary to complete the array. For example, if we wanted an array of dimension $5 \times 2 \times 1$, we could do

```
len_three_arr = len_one_arr.reshape((-1,2,1))
len_three_arr.shape
```

```
(5, 2, 1)
```

Reshaping using −1 Argument

We can always leave one dimension as -1 to tell `numpy` to just make that dimension whatever is necessary to complete the array. For example, if we wanted an array of dimension $5 \times 2 \times 1$, we could do

```
len_three_arr = len_one_arr.reshape((-1,2,1))
len_three_arr.shape
```

```
(5, 2, 1)
```

But we'll fail if we don't give `numpy` a divisible number:

```
len_one_arr.reshape((3,-1))
```

```
ValueError: cannot reshape array of size 10 into shape (3,newaxi
```

Indexing in numpy

Define a matrix of random numbers:

```
my_arr = np.arange(20).reshape((4,5))
my_arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

Indexing in numpy

Define a matrix of random numbers:

```
my_arr = np.arange(20).reshape((4,5))
my_arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

To access the first row:

Indexing in numpy

Define a matrix of random numbers:

```
my_arr = np.arange(20).reshape((4,5))
my_arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

To access the first row:

```
my_arr[0,:]
```

```
array([0, 1, 2, 3, 4])
```

Indexing in numpy

Define a matrix of random numbers:

```
my_arr = np.arange(20).reshape((4,5))
my_arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

To access the first row:

```
my_arr[0,:]
```

```
array([0, 1, 2, 3, 4])
```

To access the first column:

Indexing in numpy

Define a matrix of random numbers:

```
my_arr = np.arange(20).reshape((4,5))
my_arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

To access the first row:

```
my_arr[0,:]
```

```
array([0, 1, 2, 3, 4])
```

To access the first column:

```
my_arr[:,0]
```

Indexing in numpy

Define a matrix of random numbers:

```
my_arr = np.arange(20).reshape((4,5))
my_arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

To access the first row:

```
my_arr[0,:]
```

```
array([0, 1, 2, 3, 4])
```

To access the first column:

```
my_arr[:,0]
```

Indexing in numpy

Define a matrix of random numbers:

```
my_arr = np.arange(20).reshape((4,5))
my_arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

To access the first row:

```
my_arr[0,:]
```

```
array([0, 1, 2, 3, 4])
```

To access the first column:

```
my_arr[:,0]
```

Arithmetic in numpy

Multiplication of a scalar to an array works as expected:

```
5. * my_arr
```

```
array([[ 0.,  5., 10., 15., 20.],
       [25., 30., 35., 40., 45.],
       [50., 55., 60., 65., 70.],
       [75., 80., 85., 90., 95.]])
```

Arithmetic in `numpy`

Multiplication of a scalar to an array works as expected:

```
5. * my_arr
```

```
array([[ 0.,  5., 10., 15., 20.],
       [25., 30., 35., 40., 45.],
       [50., 55., 60., 65., 70.],
       [75., 80., 85., 90., 95.]])
```

What about dividing a scalar by an array?

```
1. / my_arr
```

```
array([[       inf, 1.        , 0.5       , 0.33333333, 0.25
       [0.2       , 0.16666667, 0.14285714, 0.125     , 0.111111
       [0.1       , 0.09090909, 0.08333333, 0.07692308, 0.071428
       [0.06666667, 0.0625    , 0.05882353, 0.05555556, 0.052631
```

Array Operations

What about multiplying an array by an array?

```
my_arr * my_arr
```

```
array([[  0,   1,   4,   9,  16],
       [ 25,  36,  49,  64,  81],
       [100, 121, 144, 169, 196],
       [225, 256, 289, 324, 361]])
```

Array Operations

What about multiplying an array by an array?

```
my_arr * my_arr
```

```
array([[  0,   1,   4,   9,  16],
       [ 25,  36,  49,  64,  81],
       [100, 121, 144, 169, 196],
       [225, 256, 289, 324, 361]])
```

We can also get boolean arrays with element-wise comparisons

```
my_arr > 10.
```

```
array([[False, False, False, False, False],
       [False, False, False, False, False],
       [False,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True]])
```

Boolean Indexing

Let's create a new array

```
new_arr = np.array([4,1,0,6,6,2,6,9,6,7]).reshape((5,2))
new_arr
```

```
array([[4, 1],
       [0, 6],
       [6, 2],
       [6, 9],
       [6, 7]])
```

Boolean Indexing

Let's create a new array

```
new_arr = np.array([4,1,0,6,6,2,6,9,6,7]).reshape((5,2))
new_arr
```

```
array([[4, 1],
       [0, 6],
       [6, 2],
       [6, 9],
       [6, 7]])
```

If we want to subset to only columns where the sum of the numbers is more than 10, we can do this directly (more on `np.sum` soon). First, we get a boolean array

```
cond = np.sum(new_arr, axis = 1) > 10
cond
```

```
array([False, False, False,  True,  True])
```

Boolean Indexing

Let's create a new array

```
new_arr = np.array([4,1,0,6,6,2,6,9,6,7]).reshape((5,2))
new_arr
```

```
array([[4, 1],
       [0, 6],
       [6, 2],
       [6, 9],
       [6, 7]])
```

If we want to subset to only columns where the sum of the numbers is more than 10, we can do this directly (more on `np.sum` soon). First, we get a boolean array

```
cond = np.sum(new_arr, axis = 1) > 10
cond
```

```
array([False, False, False,  True,  True])
```

Exercise: Absolute Value

Compute the absolute value of an arbitrary array (do not use any built-in absolute value functions).

Solutions: Absolute Value

```
def my_abs(arr: np.array) -> np.array:
    """Compute absolute value of an array"""

    neg_elements = arr < 0
    arr[neg_elements] = arr[neg_elements] * -1
    return arr

my_abs(np.arange(-3,3))
```

```
array([3, 2, 1, 0, 1, 2])
```

Solutions: Absolute Value

```
def my_abs(arr: np.array) -> np.array:
    """Compute absolute value of an array"""

    neg_elements = arr < 0
    arr[neg_elements] = arr[neg_elements] * -1
    return arr

my_abs(np.arange(-3,3))
```

```
array([3, 2, 1, 0, 1, 2])
```

Alternative using handy `np.sign` function:

```
np.arange(-3,3) * np.sign(np.arange(-3,3))
```

```
array([3, 2, 1, 0, 1, 2])
```

Universal Functions

Vectorized Math

`numpy` has built-in vectorized mathematical functions (called universal functions, or `ufuncs`):

```
np.exp(my_arr)
```

```
array([[1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.008553
        5.45981500e+01],
       [1.48413159e+02, 4.03428793e+02, 1.09663316e+03, 2.980957
        8.10308393e+03],
       [2.20264658e+04, 5.98741417e+04, 1.62754791e+05, 4.424133
        1.20260428e+06],
       [3.26901737e+06, 8.88611052e+06, 2.41549528e+07, 6.565996
        1.78482301e+08]])
```

Vectorized Math

numpy has built-in vectorized mathematical functions (called universal functions, or ufuncs):

```
np.exp(my_arr)
```

```
array([[1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.008553
        5.45981500e+01],
       [1.48413159e+02, 4.03428793e+02, 1.09663316e+03, 2.980957
        8.10308393e+03],
       [2.20264658e+04, 5.98741417e+04, 1.62754791e+05, 4.424133
        1.20260428e+06],
       [3.26901737e+06, 8.88611052e+06, 2.41549528e+07, 6.565996
        1.78482301e+08]])
```

```
np.sqrt(my_arr)
```

```
array([[0.        , 1.        , 1.41421356, 1.73205081, 2.
       [2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.
       [3.16227766, 3.31662479, 3.46410162, 3.60555128, 3.741657
       [3.87298335, 4.        , 4.12310563, 4.24264069, 4.358898
```

Aggregating Functions

```
my_arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

Aggregating Functions

```
my_arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

```
np.sum(my_arr)
```

```
190
```

Aggregating Functions

```python
my_arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

```python
np.sum(my_arr)
```

```
190
```

```python
np.sum(my_arr, axis = 0)
```

```
array([30, 34, 38, 42, 46])
```

Aggregating Functions

```python
my_arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

```python
np.sum(my_arr)
```

```
190
```

```python
np.sum(my_arr, axis = 0)
```

```
array([30, 34, 38, 42, 46])
```

```python
np.sum(my_arr, axis = 1)
```

```
array([10, 35, 60, 85])
```

Why Is This Useful?

- These are element-wise operations (we do the function once for each element of an array)
- In python, we'd have to use a for loop (slow!)
- `numpy` has a faster language (C/C++/Fortran) run these loops
- **Implication**: if you can run a calculation as a vectorized `numpy` operation using these functions, it will be *much* faster than a loop.

Binary Universal Functions

Sometimes we have `ufuncs` that accept two arguments:

```python
a = new_arr[:,0]
b = new_arr[:,1]
print(a,b)
```

```
[4 0 6 6 6] [1 6 2 9 7]
```

Binary Universal Functions

Sometimes we have `ufuncs` that accept two arguments:

```
a = new_arr[:,0]
b = new_arr[:,1]
print(a,b)
```

```
[4 0 6 6 6] [1 6 2 9 7]
```

```
np.maximum(a,b)
```

```
array([4, 6, 6, 9, 7])
```

Binary Universal Functions

Sometimes we have `ufuncs` that accept two arguments:

```
a = new_arr[:,0]
b = new_arr[:,1]
print(a,b)
```

```
[4 0 6 6 6] [1 6 2 9 7]
```

```
np.maximum(a,b)
```

```
array([4, 6, 6, 9, 7])
```

Note that this is a contrived example (that is, splitting the array into two separate arrays and using `np.maximum`)...because we have the `np.max` aggregator!

```
np.max(new_arr, axis = 1)
```

```
array([4, 6, 6, 9, 7])
```

Broadcasting

What is Broadcasting?

- Suppose we have an array and a vector with a dimension in common:

```
vec = np.arange(5).reshape((-1,5))
mat = np.ones(15).reshape((-1,5))
print(f'vec shape: {vec.shape}, mat shape: {mat.shape}')
```

```
vec shape: (1, 5), mat shape: (3, 5)
```

What is Broadcasting?

- Suppose we have an array and a vector with a dimension in common:

```
vec = np.arange(5).reshape((-1,5))
mat = np.ones(15).reshape((-1,5))
print(f'vec shape: {vec.shape}, mat shape: {mat.shape}')
```

```
vec shape: (1, 5), mat shape: (3, 5)
```

We know that we can do matrix multiplication, but what happens if we try to naively divide?

```
(mat / vec).shape
```

```
(3, 5)
```

What is Broadcasting?

- Suppose we have an array and a vector with a dimension in common:

```
vec = np.arange(5).reshape((-1,5))
mat = np.ones(15).reshape((-1,5))
print(f'vec shape: {vec.shape}, mat shape: {mat.shape}')
```

```
vec shape: (1, 5), mat shape: (3, 5)
```

We know that we can do matrix multiplication, but what happens if we try to naively divide?

```
(mat / vec).shape
```

```
(3, 5)
```

What if we divide in the other direction?

```
(vec / mat).shape
```

```
(3, 5)
```

Broadcasted Results

Let's look at these arrays to try to diagnose what's happening:

```python
print(vec)
print(mat)
print(mat / vec)
```

```
[[0 1 2 3 4]]
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
[[       inf 1.         0.5        0.33333333 0.25      ]
 [       inf 1.         0.5        0.33333333 0.25      ]
 [       inf 1.         0.5        0.33333333 0.25      ]]
```

Broadcasted Results

Let's look at these arrays to try to diagnose what's happening:

```
print(vec)
print(mat)
print(mat / vec)
```

```
[[0 1 2 3 4]]
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
[[       inf 1.         0.5        0.33333333 0.25      ]
 [       inf 1.         0.5        0.33333333 0.25      ]
 [       inf 1.         0.5        0.33333333 0.25      ]]
```

```
print(vec / mat)
```

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

Broadcasted Results

Let's look at these arrays to try to diagnose what's happening:

```
print(vec)
print(mat)
print(mat / vec)
```

```
[[0 1 2 3 4]]
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
[[       inf 1.         0.5        0.33333333 0.25      ]
 [       inf 1.         0.5        0.33333333 0.25      ]
 [       inf 1.         0.5        0.33333333 0.25      ]]
```

```
print(vec / mat)
```

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

Broadcasting Errors

`numpy` will only broadcast if

1. Arrays have the same number of dimensions, and
2. All but one of the dimensions match.

So this operation fails because the number of dimensions do not match:

```
mat / np.ones(3)
```

```
ValueError: operands could not be broadcast together with shapes
```

Broadcasting Errors

And this operation fails because the arrays differ across multiple dimensions:

```
np.ones((5,2,1)) / np.ones((4,1,1))
```

```
ValueError: operands could not be broadcast together with shapes
```

Broadcasting Errors

And this operation fails because the arrays differ across multiple dimensions:

```
np.ones((5,2,1)) / np.ones((4,1,1))
```

```
ValueError: operands could not be broadcast together with shapes
```

Lesson: if you're broadcasting, make it clear to `numpy` what behavior you want!

Example: Covariance Matrix

Broadcasting makes calculating the variance/covariance matrix of vector of random variables easy

Example: Covariance Matrix

Broadcasting makes calculating the variance/covariance matrix of vector of random variables easy

```
draws = np.random.multivariate_normal(
    mean = np.arange(6,9),
    cov = np.diag(np.arange(1,4)),
    size = 1000
)
demeaned_draws = draws - np.mean(draws, axis = 0)
demeaned_draws.T @ demeaned_draws / 1000
```

```
array([[ 0.92337133,  0.00848744, -0.08387021],
       [ 0.00848744,  2.0970935 ,  0.16712617],
       [-0.08387021,  0.16712617,  3.22424038]])
```

Exercise: Standardizing Features

We now know enough `numpy` to efficiently “standardize” elements of a matrix (or more accurately, to rewrite `sklearn.preprocessing.StandardScaler`):

$$z_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j}$$

This is useful if your features have different scales for certain models (i.e. LASSO). Please write a function `standardize` to do this (that is, calculate the mean and standard deviation of each column, and return the array with standardized values).

Solutions: Standardizing Features

```python
def standardize(arr: np.array) -> np.array:
    """
    Standardize the features in `arr`
    """

    mu_j = np.mean(arr, axis = 0)
    sigma_j = np.std(arr, axis = 0)

    return (arr - mu_j) / sigma_j

standardize(new_arr)
```

```
array([[-0.17149859, -1.31876095],
       [-1.88648444,  0.32969024],
       [ 0.68599434, -0.98907071],
       [ 0.68599434,  1.31876095],
       [ 0.68599434,  0.65938047]])
```

Solutions: Standardizing Features

```
def standardize(arr: np.array) -> np.array:
    """
    Standardize the features in `arr`
    """

    mu_j = np.mean(arr, axis = 0)
    sigma_j = np.std(arr, axis = 0)

    return (arr - mu_j) / sigma_j

standardize(new_arr)
```

```
array([[-0.17149859, -1.31876095],
       [-1.88648444,  0.32969024],
       [ 0.68599434, -0.98907071],
       [ 0.68599434,  1.31876095],
       [ 0.68599434,  0.65938047]])
```

Note here that we are broadcasting.

Random Number Generation

Random Draws

`numpy` allows for easy simulation draws. For example, we can draw 100,000 times from a standard normal distribution:

```
draws = np.random.standard_normal(100000)
print(np.mean(draws), np.std(draws))
```

```
-0.00010437185458277056 0.9943332478622053
```

Random Draws

`numpy` allows for easy simulation draws. For example, we can draw 100,000 times from a standard normal distribution:

```
draws = np.random.standard_normal(100000)
print(np.mean(draws), np.std(draws))
```

```
-0.00010437185458277056 0.9943332478622053
```

As noted by McKinney, this procedure is significantly (at least one order of magnitude) faster than python's default `random` module – use `numpy.random`!

Seeds

- For reproducibility, it's good practice to set seeds so that you can regenerate results for random processes
- In `numpy`:

```
rng = np.random.default_rng(seed=481) # rng = random number ge
rng_draws = rng.standard_normal(100000)
print(np.mean(rng_draws), np.std(rng_draws))
```

```
-0.0016440274114192328 1.00077298022399
```

Exercise: Random?

We're skeptical that `numpy` is actually giving us random numbers, so we want to test it by doing the following procedure 1000 times:

1. Simulate 1000 standard normal random variables.
2. Take the mean of those 1000 simulations.

Armed with our 1000 sample means, we're going to calculate the mean and standard deviation of the sample means. What should they be? Do we get close? Please set a seed for reproducibility.

Solutions: Random?

Via the Central Limit Theorem, we know that the sample mean $\overline{x}$ of n i.i.d. random variables with mean μ is distributed

$$\overline{x} \sim \mathcal{N}\left(\mu, \frac{\sigma^2}{n}\right)$$

```
%%time
rng = np.random.default_rng(seed=481)

sample_means = []
for i in range(1000):
    sample_means.append(np.mean(rng.standard_normal(size=1000

print(np.mean(sample_means), np.std(sample_means))
```

```
-2.977436149096266e-05 0.031892612678989775
CPU times: user 8.89 ms, sys: 153 µs, total: 9.05 ms
Wall time: 9.05 ms
```

Solutions: Random? (Faster)

```
%%time
rng = np.random.default_rng(seed=481)

random_draws = rng.standard_normal(size=1000*1000).reshape((1
sample_means = np.mean(random_draws, axis = 0)

print(np.mean(sample_means), np.std(sample_means))
```

```
-2.977436149096318 7e-05 0.030131789839109725
CPU times: user 5.87 ms, sys: 722 µs, total: 6.59 ms
Wall time: 6.37 ms
```

Solutions: Random? (Faster)

```
%%time
rng = np.random.default_rng(seed=481)

random_draws = rng.standard_normal(size=1000*1000).reshape((1
sample_means = np.mean(random_draws, axis = 0)

print(np.mean(sample_means), np.std(sample_means))
```

```
-2.9774361490963187e-05 0.030131789839109725
CPU times: user 5.87 ms, sys: 722 µs, total: 6.59 ms
Wall time: 6.37 ms
```

Are these close to the truth?

```
print(0, np.sqrt(1./1000))
```

```
0 0.03162277660168379
```

Application: Using `numpy` For Regressions

Salary Data

Suppose we have data that looks like this:

Salary	Experience	SAT Score	College GPA	Master's Degree
60,000	2	1200	3.2	0
90,000	1	1400	3.8	1
30,000	19	900	2.5	0
200,000	10	1400	3.8	1
150,000	5	1000	3.0	0
30,000	4	1100	3.9	0

We'd like to understand what drives salary. What should we do?

Linear Regression

Given feature (explanatory variables) matrix $\mathbf{X}$ and outcome variable y, , define residuals from the model for a choice of coefficient vector β as

$$\underbrace{e}_{n\times 1} = \underbrace{y}_{n\times 1} - \underbrace{\mathbf{X}}_{n\times k}\underbrace{\beta}_{k\times 1}$$

Then the coefficients from an OLS (ordinary least squares) regression are the solution to the problem

$$\min_{\beta}\ e^{\top}e$$

OLS Solution

If $\mathbf{X}^\top \mathbf{X}$ is invertible, then the solution to the problem has a closed-form solution given by

$$\hat{\beta} = (\underbrace{\mathbf{X}^\top \mathbf{X}}_{k \times k})^{-1} \underbrace{\mathbf{X}^\top}_{k \times n} \underbrace{y}_{n \times 1}$$

Loading Data

First, we need to load the data[2]

```
X = np.array(
    [
        [2,1200,3.2,0],
        [1,1400,3.8,1],
        [19,900,2.5,0],
        [10,1400,3.8,1],
        [5,1000,3.0,0],
        [4,1100,3.9,0]
    ]
)
y = np.array([60000, 90000, 30000, 200000, 150000, 30000]).res

print(X.shape, y.shape)
```

```
(6, 4) (6, 1)
```

[2]Don't do this in real life! Using pandas to load from a CSV or another file format.

Calculating Estimator

```
beta_hat = np.linalg.inv(X.T @ X) @ X.T @ y
beta_hat
```

```
array([[   720.48960622],
       [   114.35428578],
       [-18598.07654699],
       [ 51613.9979523 ]])
```

Calculating Estimator

```
beta_hat = np.linalg.inv(X.T @ X) @ X.T @ y
beta_hat
```

```
array([[   720.48960622],
       [   114.35428578],
       [-18598.07654699],
       [ 51613.9979523 ]])
```

What's missing?

Adding Intercept

```
X_int = np.concatenate(
    [
        np.ones(X.shape[0]).reshape(-1,1),
        X,
    ],
    axis = 1
)
X_int
```

```
array([[1.0e+00, 2.0e+00, 1.2e+03, 3.2e+00, 0.0e+00],
       [1.0e+00, 1.0e+00, 1.4e+03, 3.8e+00, 1.0e+00],
       [1.0e+00, 1.9e+01, 9.0e+02, 2.5e+00, 0.0e+00],
       [1.0e+00, 1.0e+01, 1.4e+03, 3.8e+00, 1.0e+00],
       [1.0e+00, 5.0e+00, 1.0e+03, 3.0e+00, 0.0e+00],
       [1.0e+00, 4.0e+00, 1.1e+03, 3.9e+00, 0.0e+00]])
```

axis

We do need the axis argument to put the intercept vector in the right place:

```
np.concatenate(
    [
        np.ones(X.shape[0]).reshape(-1,1),
        X,
    ]
)
```

```
ValueError: all the input array dimensions except for the concat
```

np.r_ and np.c_

If you know that you're adding rows or columns to a matrix, numpy has two handy functions to do just that:

```
np.c_[np.ones(X.shape[0]).reshape(-1,1), X]
```

```
array([[1.0e+00, 2.0e+00, 1.2e+03, 3.2e+00, 0.0e+00],
       [1.0e+00, 1.0e+00, 1.4e+03, 3.8e+00, 1.0e+00],
       [1.0e+00, 1.9e+01, 9.0e+02, 2.5e+00, 0.0e+00],
       [1.0e+00, 1.0e+01, 1.4e+03, 3.8e+00, 1.0e+00],
       [1.0e+00, 5.0e+00, 1.0e+03, 3.0e+00, 0.0e+00],
       [1.0e+00, 4.0e+00, 1.1e+03, 3.9e+00, 0.0e+00]])
```

np.r_ and np.c_

If you know that you're adding rows or columns to a matrix, `numpy` has two handy functions to do just that:

```
np.c_[np.ones(X.shape[0]).reshape(-1,1), X]
```

```
array([[1.0e+00, 2.0e+00, 1.2e+03, 3.2e+00, 0.0e+00],
       [1.0e+00, 1.0e+00, 1.4e+03, 3.8e+00, 1.0e+00],
       [1.0e+00, 1.9e+01, 9.0e+02, 2.5e+00, 0.0e+00],
       [1.0e+00, 1.0e+01, 1.4e+03, 3.8e+00, 1.0e+00],
       [1.0e+00, 5.0e+00, 1.0e+03, 3.0e+00, 0.0e+00],
       [1.0e+00, 4.0e+00, 1.1e+03, 3.9e+00, 0.0e+00]])
```

Note the brackets (this is technically an indexing routine)

`np.hstack` and `np.vstack`

Alternatively, we can perform the operation with `np.hstack`:

```
np.hstack((np.ones(X.shape[0]).reshape(-1,1), X))
```

```
array([[1.0e+00, 2.0e+00, 1.2e+03, 3.2e+00, 0.0e+00],
       [1.0e+00, 1.0e+00, 1.4e+03, 3.8e+00, 1.0e+00],
       [1.0e+00, 1.9e+01, 9.0e+02, 2.5e+00, 0.0e+00],
       [1.0e+00, 1.0e+01, 1.4e+03, 3.8e+00, 1.0e+00],
       [1.0e+00, 5.0e+00, 1.0e+03, 3.0e+00, 0.0e+00],
       [1.0e+00, 4.0e+00, 1.1e+03, 3.9e+00, 0.0e+00]])
```

np.hstack and np.vstack

Alternatively, we can perform the operation with `np.hstack`:

```
np.hstack((np.ones(X.shape[0]).reshape(-1,1), X))
```

```
array([[1.0e+00, 2.0e+00, 1.2e+03, 3.2e+00, 0.0e+00],
       [1.0e+00, 1.0e+00, 1.4e+03, 3.8e+00, 1.0e+00],
       [1.0e+00, 1.9e+01, 9.0e+02, 2.5e+00, 0.0e+00],
       [1.0e+00, 1.0e+01, 1.4e+03, 3.8e+00, 1.0e+00],
       [1.0e+00, 5.0e+00, 1.0e+03, 3.0e+00, 0.0e+00],
       [1.0e+00, 4.0e+00, 1.1e+03, 3.9e+00, 0.0e+00]])
```

Note that we're passing a `tuple` here

Regress With Intercept

```
beta_hat_int = np.linalg.inv(X_int.T @ X_int) @ X_int.T @ y
beta_hat_int
```

```
array([[ 1.75082781e+05],
       [-1.22516556e+03],
       [-1.49834437e+01],
       [-2.62417219e+04],
       [ 9.73509934e+04]])
```

Regress With Intercept

```
beta_hat_int = np.linalg.inv(X_int.T @ X_int) @ X_int.T @ y
beta_hat_int
```

```
array([[ 1.75082781e+05],
       [-1.22516556e+03],
       [-1.49834437e+01],
       [-2.62417219e+04],
       [ 9.73509934e+04]])
```

We can check with `sklearn` (future lectures)

```
from sklearn.linear_model import LinearRegression
reg_model = LinearRegression(fit_intercept=False).fit(X_int, 
reg_model.coef_
```

```
array([[ 1.75082781e+05, -1.22516556e+03, -1.49834437e+01,
        -2.62417219e+04,  9.73509934e+04]])
```

Regress With Intercept

```
beta_hat_int = np.linalg.inv(X_int.T @ X_int) @ X_int.T @ y
beta_hat_int
```

```
array([[ 1.75082781e+05],
       [-1.22516556e+03],
       [-1.49834437e+01],
       [-2.62417219e+04],
       [ 9.73509934e+04]])
```

We can check with `sklearn` (future lectures)

```
from sklearn.linear_model import LinearRegression
reg_model = LinearRegression(fit_intercept=False).fit(X_int, 
reg_model.coef_
```

```
array([[ 1.75082781e+05, -1.22516556e+03, -1.49834437e+01,
        -2.62417219e+04,  9.73509934e+04]])
```

We can check formally using `numpy`:

Application: Likelihood-Based Estimation Using `numpy`

Likelihood Example

- Assume that we observe n observations $(x_1, ..., x_n)$
- Each observation is the number of heads we flipped out of k coin flips
  - More generally called number of successes from k trials

Likelihood Example

- Assume that we observe n observations $(x_1, ..., x_n)$
- Each observation is the number of heads we flipped out of k coin flips
  - More generally called number of successes from k trials

- If the flips are i.i.d. (so the observations are i.i.d.), then each of these counts is drawn from a binomial distribution with parameter p
- How can we calculate this parameter?

Likelihood Function (Observation)

- In our example, the probability of observing ℓ heads from k flips is

$$\mathbb{P}(\ell|p) = \binom{k}{\ell} p^{\ell}(1-p)^{k-\ell}$$

- To get ℓ successes from k (independent) trials, we need
  - An event with probability p (a success) to occur ℓ times
  - An event with probability $1-p$ (a failure) to occur $k-\ell$ times
  - $\implies$ the probability of **one sequence** of successes and failures in k trials is $p^{\ell}(1-p)^{k-\ell}$
- How many sequences are there?
  - For example, getting 2 heads in 3 flips can occur **3** ways: HHT, HTH, THH
  - Total number of sequences: $\binom{k}{\ell}$

Likelihood Function (Data)

This means that for a given parameter p we can write the probability of observing our data given p as

$$\mathbb{P}(x_1, ...x_n|p) = \prod_{i=1}^{n} \mathbb{P}(x_i|p)$$

It's often more convenient (and numerically stable) to work with the logarithm of the likelihood function:

$$\mathcal{L}_n(x_1, ...x_n|p) = \log\left(\mathbb{P}(x_1, ...x_n|p)\right) = \sum_{i=1}^{n} \log\left(\mathbb{P}(x_i|p)\right)$$

This is why it's important for our observations to be independent!

Maximum Likelihood Estimation

Given a likelihood function, we can define MLE estimator for our example as the value of p that maximizes the probability of observing our data. Formally, it is the solution to the problem

$$\max_{\hat{p}} \ \mathcal{L}_n(x_1, ...x_n | \hat{p})$$

Maximum Likelihood Estimation

Given a likelihood function, we can define MLE estimator for our example as the value of p that maximizes the probability of observing our data. Formally, it is the solution to the problem

$$\max_{\hat{p}} \; \mathcal{L}_n(x_1, ...x_n|\hat{p})$$

How can we solve this problem?

Solving For $\hat{p}_{MLE}$ (Analytical)

$$\begin{aligned}
\frac{\partial \mathcal{L}_n(x_1, ...x_n|p)}{\partial p} &= \sum_{i=1}^{n} \frac{\partial \log\left(\mathbb{P}(x_i|p)\right)}{\partial p} \\
&= \sum_{i=1}^{n} \frac{\partial}{\partial p} \left\{ \log \binom{k}{x_i} + x_i \log(p) + (k - x_i) \log(1-p) \right\} \\
&= \sum_{i=1}^{n} \frac{x_i}{p} - \frac{k - x_i}{1-p} \\
&= \frac{\sum_{i=1}^{n} x_i}{p} - \frac{nk - \sum_{i=1}^{n} x_i}{1-p}
\end{aligned}$$

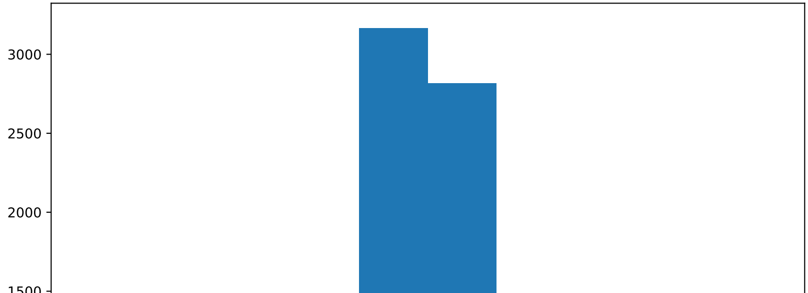
Set to zero to find the maximum:

$$\frac{\sum_{i=1}^{n} x_i}{\hat{p}_{MLE}} = \frac{nk - \sum_{i=1}^{n} x_i}{1 - \hat{p}_{MLE}} \implies \hat{p}_{MLE} = \frac{1}{n} \sum_{i=1}^{n} \frac{x_i}{k}$$

Using numpy and scipy For MLE

Let's solve this problem using the closed-form solution and a computational solution.

```
import matplotlib.pyplot as plt
k = 100
p = .35
n = 10000
x_i = np.random.binomial(n=k, p=p, size=n)
plt.hist(x_i)
plt.show()
```



Analytical Solution

```
p_mle = (1./n) * np.sum(x_i/k)
p_mle
```

```
0.350005
```

Log-Likelihood Function

```
def neg_ll(theta: float, data: np.array, k: int) -> float:
    """
    Compute the negative (why?) log-likelihood for a
    binomial distribution given data and a specific parameter
    The factorial term is omitted (why?)
    """

    p_successes = data * np.log(theta)
    p_failures = (k-data) * np.log(1-theta)

    return -np.sum(p_successes + p_failures)

print(f'Log-Likelihood when we\'re far away: {-neg_ll(.8, x_i
print(f'Log-Likelihood when we\'re close: {-neg_ll(.3, x_i, k
```

```
Log-Likelihood when we're far away: -1124227.9545703332
Log-Likelihood when we're close: -653233.4315635557
```

Maximization

```python
import scipy as sp

sp.optimize.minimize(
    fun=neg_ll, # the objective function
    x0=.25, # starting guess
    args=(x_i, k), # additional parameters passed to neg_ll
    bounds = ((0,1),), # bounds for the optimization
    method = 'Nelder-Mead' # optionally pick an algorithm
)
```

```
       message: Optimization terminated successfully.
       success: True
        status: 0
           fun: 647449.7342306746
             x: [ 3.500e-01]
           nit: 15
          nfev: 30
 final_simplex: (array([[ 3.500e-01],
                       [ 3.500e-01]]), array([ 6.474e+05,  6.474
```

Why Nelder-Mead?

What happens if we let `scipy` use its default solver (BFGS, I believe)?

```
sp.optimize.minimize(
    fun=neg_ll, # the objective function
    x0=.25, # starting guess
    args=(x_i, k), # additional parameters passed to neg_ll
    bounds = ((0,1),) # bounds for the optimization
)
```

```
 message: CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
 success: True
  status: 0
     fun: 672201.8665470625
       x: [ 2.500e-01]
     nit: 1
     jac: [-5.334e+05]
    nfev: 6
    njev: 3
hess_inv: <1x1 LbfgsInvHessProduct with dtype=float64>
```

Why Nelder-Mead?

What happens if we let scipy use its default solver (BFGS, I believe)?

```
sp.optimize.minimize(
    fun=neg_ll, # the objective function
    x0=.25, # starting guess
    args=(x_i, k), # additional parameters passed to neg_ll
    bounds = ((0,1),) # bounds for the optimization
)
```

```
 message: CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH
 success: True
  status: 0
     fun: 672201.8665470625
       x: [ 2.500e-01]
     nit: 1
     jac: [-5.334e+05]
    nfev: 6
    njev: 3
hess_inv: <1x1 LbfgsInvHessProduct with dtype=float64>
```

Be careful!

Bayes Rule

Recall Bayes Rule:

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)} = \frac{\mathbb{P}(B|A)\mathbb{P}(A)}{\mathbb{P}(B)}$$

Bayes Rule

Recall Bayes Rule:

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)} = \frac{\mathbb{P}(B|A)\mathbb{P}(A)}{\mathbb{P}(B)}$$

Rewriting in a more useful format for our application:

$$\begin{aligned}\underbrace{\mathbb{P}(\theta|x_1, ..., x_n)}_{\text{Posterior}} &= \frac{\mathbb{P}(x_1, ..., x_n|\theta)\mathbb{P}(\theta)}{\mathbb{P}(x_1, ..., x_n)} \\ &\propto \underbrace{\mathbb{P}(x_1, ..., x_n|\theta)}_{\text{Likelihood}} \underbrace{\mathbb{P}(\theta)}_{\text{Prior}}\end{aligned}$$

Sampling From Posterior

- Suppose we want to sample from the posterior distribution
  - Hard to do without a functional form for the posterior
- Easier question: for two parameters θ_1 and θ_2, what should their relative frequency be in my sample?
  - Equivalently: if θ_2 has twice the posterior probability mass/density as θ_1, how many draws of θ_2 should I have in my sample relative to θ_1?

Monte Carlo Simulation

Suppose I have two “candidate” draws for my sample from the posterior, θ_1 and θ_2. By Bayes Rule, I know the ratio of their posterior probabilities can be calculated as

$$\frac{\mathbb{P}(\theta_1|x_1,...,x_n)}{\mathbb{P}(\theta_2|x_1,...,x_n)} = \frac{\mathbb{P}(x_1,...,x_n|\theta_1)\mathbb{P}(\theta_1)}{\mathbb{P}(x_1,...,x_n|\theta_2)\mathbb{P}(\theta_2)} \equiv \alpha$$

where α is the “acceptance rate”. Then to pick which parameter to include in our sample, just need to simulate a random variable that will pick θ_1 with probability $\min\{\alpha, 1\}$ and θ_2 otherwise.

Intuition

$$\frac{\mathbb{P}(\theta_1|x_1,...,x_n)}{\mathbb{P}(\theta_2|x_1,...,x_n)} \equiv \alpha$$

- What will we choose when $\mathbb{P}(\theta_1|x_1,...,x_n) > \mathbb{P}(\theta_2|x_1,...,x_n)$?
- What will we choose when $\mathbb{P}(\theta_2|x_1,...,x_n) > \mathbb{P}(\theta_1|x_1,...,x_n)$?

Markov Chain

How should we generate θ_1 and θ_2 if we don't know what the distribution looks like?

- We want to "wander", but generally go in the "right" direction (samples with high posterior probability)
- Use a Markov Chain with our acceptance rate
  - Formally, a Markov Chain is a sequence with "memorylessness" property

    $$\mathbb{P}(X_n = x_n \mid X_{n-1} = x_{n-1}, \dots, X_0 = x_0) = \mathbb{P}(X_n = x_n \mid X_{n-1} = x_{n-1})$$

  - That is, only the most recent value in the sequence impacts the next value in the sequence
  - For our purposes, $\theta_i = \theta_{i-1} + \varepsilon_i$ where $\mathbb{E}[\varepsilon_i] = 0$
- Note that

$$x_n = \sum_{i=0}^{n-1} x_i + \varepsilon_n = x_0 + \sum_{i=1}^{n} \varepsilon_i$$

Exercise: Markov Chain

Generate 1000 draws from a Markov Chain that starts at zero with your choice of noise term. Set a seed for reproducibility.

Solutions: Markov Chain (Slower)

```
%%time
rng = np.random.default_rng(seed = 481)

chain = []
x = 0

for i in range(1000):
    x += rng.standard_normal()
    chain.append(x)
```

```
CPU times: user 380 µs, sys: 31 µs, total: 411 µs
Wall time: 405 µs
```

Exercise: Markov Chain (Faster)

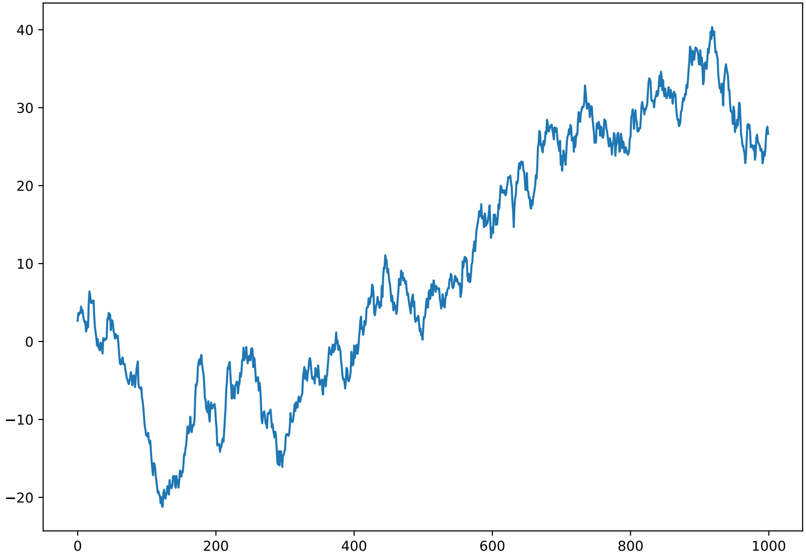
```python
%%time
rng = np.random.default_rng(seed = 481)

chain = np.cumsum(rng.standard_normal(size = 1000))
```

```
CPU times: user 165 µs, sys: 76 µs, total: 241 µs
Wall time: 207 µs
```

Markov Chain Visualized

```
plt.plot(chain)
```



Metropolis-Hastings Algorithm (Simplified)

Put it all together into this (simplified version) of the Metropolis-Hastings algorithm:

1. Choose starting point (θ_1) and noise distribution that generates ε_i
2. For each of M iterations:
   i. Generate new candidate point $\theta'_i = \theta_{i-1} + \varepsilon_i$
   ii. Calculate the likelihood of the data at both parameters, $\mathcal{L}(\theta'_i|x_i), \mathcal{L}(\theta_{i-1}|x_i)$
   iii. Calculate acceptance rate $\alpha_i = \frac{\mathcal{L}(\theta'_i|x_i)}{\mathcal{L}(\theta_{i-1}|x_i)}$
   iv. Simulate a uniform random variable u_i on the interval [0,1] – if $u_i < \alpha_i$, set $\theta_i = \theta'_i$, and if not, set $\theta_i = \theta_{i-1}$.
3. Return the sequence $\theta_1, ..., \theta_M$.

Metropolis-Hastings Implementation

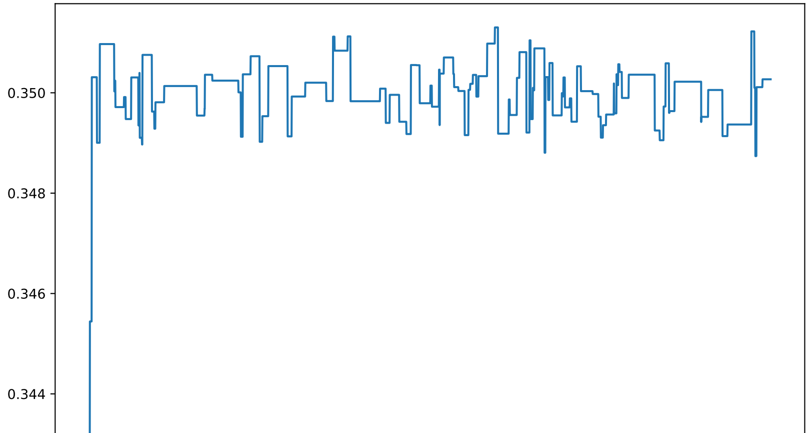
```
def met_hast(n_iterations: int) -> list:
    """
    Run Metropolis-Hastings algorithm for our binomial exampl
    """
    post_draws = []
    theta_old = .5
    for i in range(n_iterations):
        theta_new = theta_old + np.random.normal(loc = 0, sca
        ll_old = -neg_ll(theta_old, x_i, k)
        ll_new = -neg_ll(theta_new, x_i, k)
        acceptance_rate = np.exp(ll_new-ll_old)
        if np.random.uniform() < acceptance_rate:
            post_draws.append(theta_new)
            theta_old = theta_new
        else:
            post_draws.append(theta_old)

    return post_draws
```

Metropolis-Hastings Trace Plot

```
simulation_draws = met_hast(10000)
print(f'Mean of posterior draws: {round(np.mean(simulation_dra
plt.plot(range(10000), simulation_draws)
```

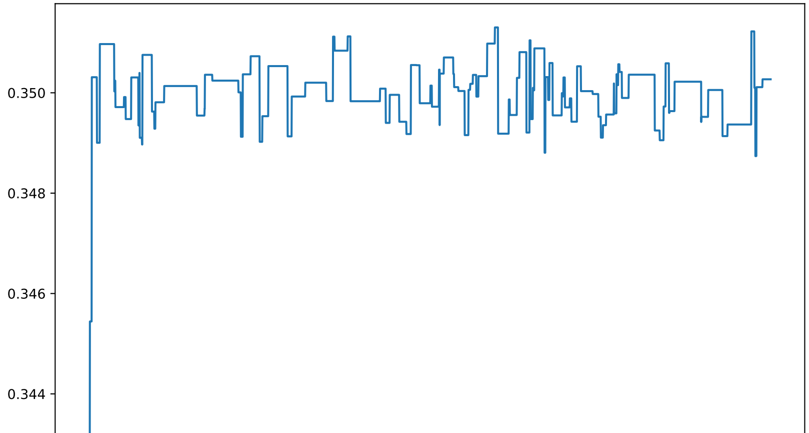
```
Mean of posterior draws: 0.35
```



Metropolis-Hastings Trace Plot

```
simulation_draws = met_hast(10000)
print(f'Mean of posterior draws: {round(np.mean(simulation_dra
plt.plot(range(10000), simulation_draws)
```

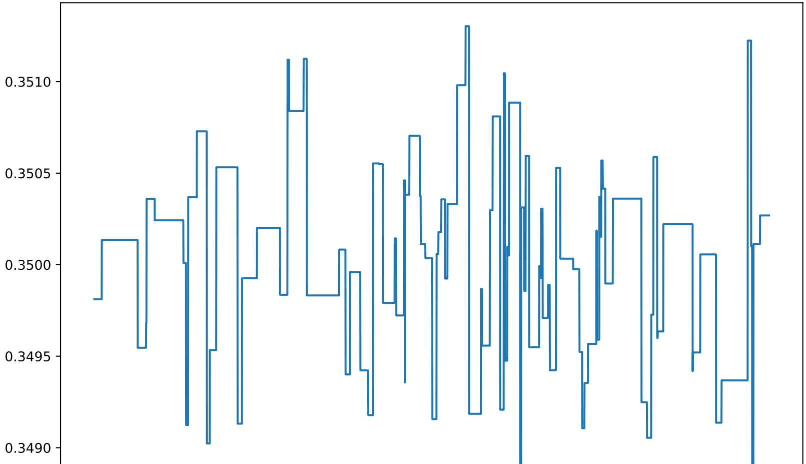
```
Mean of posterior draws: 0.35
```



Metropolis-Hastings Trace Plot (Remove Burn-In)

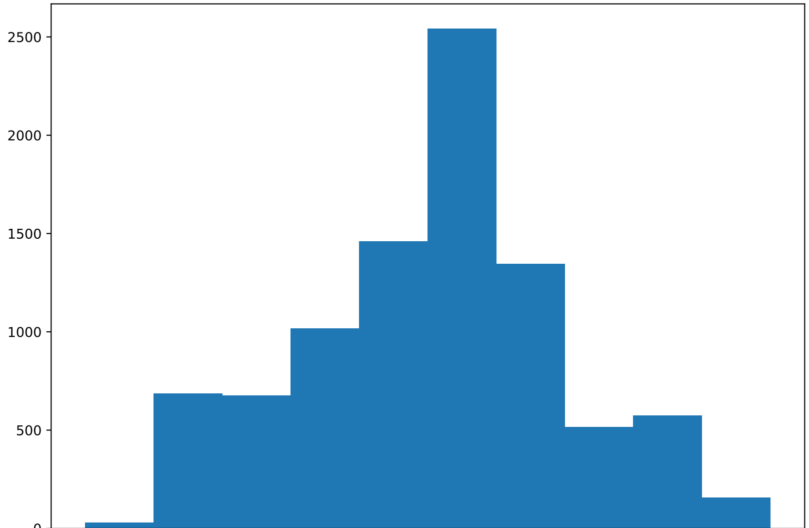
```
print(f'Mean of posterior draws: {round(np.mean(simulation_dra
plt.plot(range(1000,10000), simulation_draws[1000:])
```

```
Mean of posterior draws: 0.35
```



Metropolis-Hastings Draws

```
plt.hist(simulation_draws[1000:])
plt.show()
```



2500
2000
1500
1000
500
0