Arrays

- The core data structure in `numpy` is the array. Moving forward, "array" will always refer to a `numpy` array.

- One way to instantiate an array is to pass a list as an argument to the function `np.array`.

```
In [14]: 1 np.array(4, 9, 1, 2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[14], line 1
----> 1 np.array(4, 9, 1, 2)

TypeError: array() takes from 1 to 2 positional arguments but 4 were given
```

most important parts are at the bottom!

```
In [13]: 1 np.array([4, 9, 1, 2])
```

```
Out[13]: array([4, 9, 1, 2])
```

Out[13]: array([4, 9, 1, 2])

- Arrays, unlike lists, must be **homogenous** – all elements must be of the

In [16]:
```
1 np.array(['hello', 'hi', 'michigan'])
```

Out[16]: array(['hello', 'hi', 'michigan'], dtype='<U8')

strings up to length 8

In [17]:
```
1 np.array(['hi'])
```

Out[17]: array(['hi'], dtype='<U2')

strings up to length 2

In [15]:
```
1 # All elements are converted to strings!
2 np.array(['hello', 2025])
```

Out[15]: array(['hello', '2025'], dtype='<U21')

also often say these operations are **vectorized**.

In [18]:
```
1 temps = [68, 72, 65, 64, 62, 61, 59, 64, 64, 63, 65, 62]
2 temps
```

Out[18]: [68, 72, 65, 64, 62, 61, 59, 64, 64, 63, 65, 62]

In [ ]:
```
1 temp_array = np.array(temps)
```

no [ ] !

In [ ]:
```
1 # Increase all temperatures by 3 degrees.
2 ...
```

temps is already a list.

In [ ]:
```
1 # Halve all temperatures.
2 ...
```

In [ ]:
```
1 # Convert all temperatures to Celsius
```

operations are much quicker than if we used a vanilla Python `for`-loop.

Also, the fact that arrays must be homogenous lend themselves to more efficient representations in memory.

- We can time code in a Jupyter Notebook. Let's try and square a long sequence of integers and see how long it takes with a Python loop:

In [36]:
```
%%timeit
squares = []
for i in range(1_000_000):
    squares.append(i * i)
```

```
31.3 ms ± 639 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

- In vanilla Python, this takes about 0.03 seconds per loop.

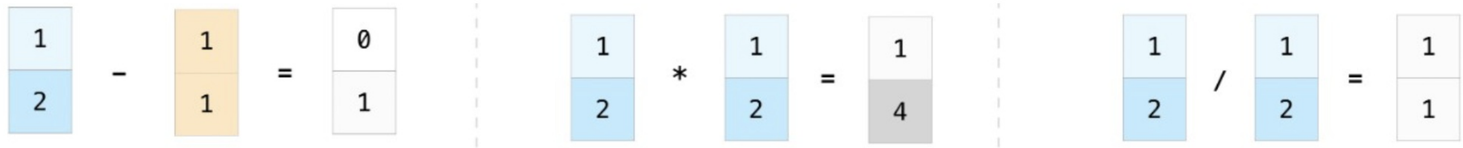
  In `numpy`: 30x speedup!

In [37]:
```
%%timeit
squares = np.arange(1_000_000) ** 2
```

```
1.2 ms ± 28.6 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

- We can apply arithmetic operations to multiple arrays, provided they have the same length.

- The result is computed **element-wise**, which means that the arithmetic operation is applied to one pair of elements from each array at a time.




```
In [38]: a = np.array([4, 5, -1])
         b = np.array([2, 3, 2])

In [39]: a + b
Out[39]: array([6, 8, 1])

In [40]: a / b
Out[40]: array([ 2.  ,  1.67, -0.5 ])

In [41]: a ** 2 + b ** 2
Out[41]: array([20, 34,  5])
```

Array methods

- Arrays come equipped with several handy methods; some examples are below, but you can read about them all here.

In [44]:
```
1 arr = np.array([3, 8, 4, -3.2])
```

In [45]:
```
1 arr.mean()
```

Out[45]: 2.95

In [46]:
```
1 arr.sum()
```

Out[46]: 11.8

In [47]:
```
1 arr.max()
```

Out[47]: 8.0

In [48]:
```
1 arr.argmax()
```

Out[48]: 1

In [ ]:
```
1 ...
```

In [ ]:
```
1 # An attribute, not a method.
2 ...
```

Array methods

- Arrays come equipped with several handy methods; some examples are below, but you can read about them all here.

In [44]:
```
arr = np.array([3, 8, 4, -3.2])
```

In [45]:
```
arr.mean()
```

Out[45]: 2.95

In [46]:
```
arr.sum()
```

Out[46]: 11.8

In [47]:
```
arr.max()
```

Out[47]: 8.0

In [48]:
```
arr.argmax()
```

Out[48]: 1

one array.

In [55]:
```
(arr ** 2 + 2 * arr + np.log1p(arr ** 2)).min()
```

Out[55]: 6.259478844465547

In [ ]:
```
# An attribute, not a method.
...
```

Activity

🎉 Congrats! 🎉 You won the lottery 💰. Here's how your payout works: on the first day of September, you are paid $0.01. Every day thereafter, your pay doubles, so on the second day you're paid $0.02, on the third day you're paid $0.04, on the fourth day you're paid $0.08, and so on.

September has 30 days.

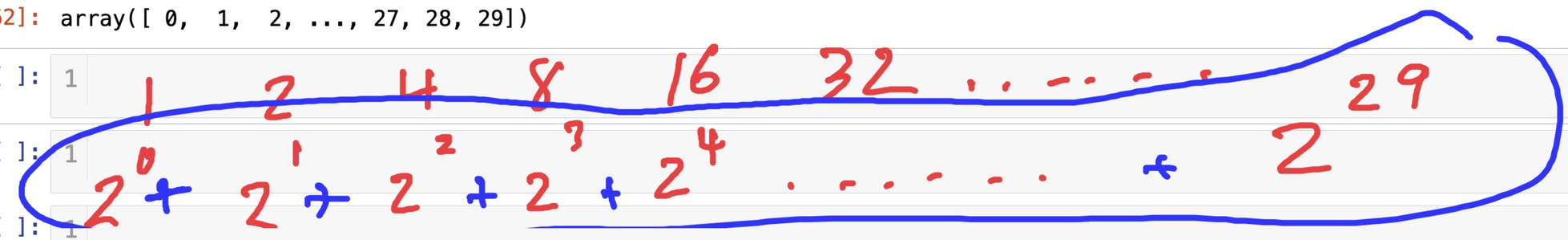
Write a **one-line expression** that uses the numbers `2` and `30`, along with the function `np.arange` and at least one array method, that computes the total amount **in dollars** you will be paid in September. No `for`-loops or list comprehensions allowed!

We have a 📹 walkthrough video of this problem (also posted on the course website under "⏯️ videos" for Lecture 3), but don't watch it until you've tried it yourself!

In [62]:
```
1 np.arange(30)
```

Out[62]:
```
array([ 0,  1,  2, ..., 27, 28, 29])
```

In [ ]:
```
1
```



In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

Boolean filtering

- Comparisons with arrays yield **Boolean** arrays! These can be used to answer questions about the values in an array.

In [67]:
```
1 temp_array
```

Out[67]:
```
array([20.  , 22.22, 18.33, 17.78, 16.67, 16.11, 15.  , 17.78, 17.78,
       17.22, 18.33, 16.67])
```

In [69]:
```
1 temp_array >= 18
```

Out[69]:
```
array([ True,  True,  True, False, False, False, False, False, False,
       False,  True, False])
```

- How many values are greater than or equal to 18?

In [70]:
```
1 (temp_array >= 18).sum()
```

Out[70]:
```
4
```

In [ ]:
```
1 ...
```

```
Out[67]: array([20.  , 22.22, 18.33, 17.78, 16.67, 16.11, 15.  , 17.78, 17.78,
                17.22, 18.33, 16.67])
```

```
In [69]: 1 temp_array >= 18
```

```
Out[69]: array([ True,  True,  True, False, False, False, False, False, False,
                False,  True, False])
```

- How many values are greater than or equal to 18?

```
In [70]: 1 (temp_array >= 18).sum()
```

```
Out[70]: 4
```

```
In [71]: 1 np.count_nonzero(temp_array >= 18)
```

```
Out[71]: 4
```

- What fraction of values are greater than or equal to 18?

```
In [72]: 1 (temp_array >= 18).mean()
```

```
Out[72]: 0.3333333333333333
```

$$\text{mean} = \frac{\text{sum}}{n} = \frac{\#\,\text{true}}{\#\,\text{total}}$$

```
In [71]: 1 np.count_nonzero(temp_array >= 18)

Out[71]: 4
```

- What fraction of values are greater than or equal to 18?

```
In [72]: 1 (temp_array >= 18).mean()

Out[72]: 0.3333333333333333
```

- Which values are greater than or equal to 18?

```
In [74]: 1 temp_array

Out[74]: array([20.  , 22.22, 18.33, 17.78, 16.67, 16.11, 15.  , 17.78, 17.78,
                17.22, 18.33, 16.67])

In [75]: 1 temp_array >= 18

Out[75]: array([ True,  True,  True, False, False, False, False, False, False,
                False,  True, False])

In [73]: 1 temp_array[temp_array >= 18]

Out[73]: array([20.  , 22.22, 18.33, 18.33])
```

treats boolean array as mask

```
In [71]: 1 np.count_nonzero(temp_array >= 18)

Out[71]: 4
```

- What fraction of values are greater than or equal to 18?

```
In [72]: 1 (temp_array >= 18).mean()

Out[72]: 0.3333333333333333
```

- Which values are greater than or equal to 18?

```
In [74]: 1 temp_array

Out[74]: array([20.  , 22.22, 18.33, 17.78, 16.67, 16.11, 15.  , 17.78, 17.78,
                17.22, 18.33, 16.67])

In [75]: 1 temp_array >= 18

Out[75]: array([ True,  True,  True, False, False, False, False, False, False,
                False,  True, False])

In [73]: 1 temp_array[temp_array >= 18]

Out[73]: array([20.  , 22.22, 18.33, 18.33])
```

treats boolean array as mask

Note: & and | vs. and and or

- In Python, the standard symbols for "and" and "or" are, literally, `and` and `or`.

In [78]:
```
1 if (5 > 3 and 'h' + 'i' == 'hi') or (-2 > 0):
2     print('success')
```

```
success
```

Note: & and | vs. and and or

- In Python, the standard symbols for "and" and "or" are, literally, `and` and `or`.

In [78]:
```
if (5 > 3 and 'h' + 'i' == 'hi') or (-2 > 0):
    print('success')
```

```
success
```

- But, when taking the **element-wise** and/or of two arrays, the standard operators don't work. Instead, use the **bitwise** operators: `&` for "and", `|` for "or".

In [81]:
```
temp_array
```

Out[81]:
```
array([20.  , 22.22, 18.33, 17.78, 16.67, 16.11, 15.  , 17.78, 17.78,
       17.22, 18.33, 16.67])
```

In [82]:
```
# Don't forget parentheses when using multiple conditions!
temp_array[(temp_array % 2 == 0) | (temp_array == temp_array.min())]
```

Out[82]:
```
array([20., 15.])
```

```
Out[83]: array([[5, 1, 9, 7],
                [9, 8, 2, 3],
                [2, 5, 0, 4]])
```

```
In [84]: 1 # nums has 3 rows and 4 columns.
         2 nums.shape
```

```
Out[84]: (3, 4)
```

- In addition to creating 2D arrays from scratch, we can also create 2D arrays by *reshaping* other arrays.

```
In [87]: 1 nums
```

```
Out[87]: array([[5, 1, 9, 7],
                [9, 8, 2, 3],
                [2, 5, 0, 4]])
```

```
In [86]: 1 nums.reshape((2, 6))
```

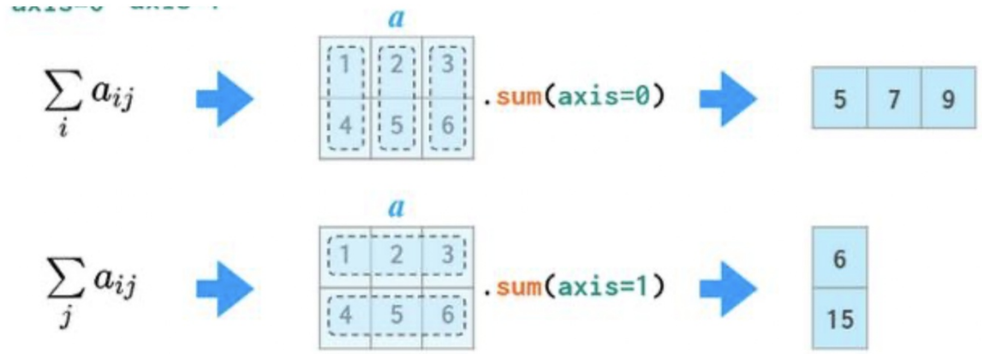
```
Out[86]: array([[5, 1, 9, 7, 9, 8],
                [2, 3, 2, 5, 0, 4]])
```

reshape to 2 rows, 6 columns.

```
In [ ]: 1
```

```
In [ ]: 1
```

```
In [ ]: 1
```



In [89]:
```
a = np.arange(1, 7).reshape((2, 3))
a
```

Out[89]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

- If we specify `axis=0`, `a.sum` will "compress" along axis 0.

In [90]:
```
a.sum(axis=0)
```

Out[90]:
```
array([5, 7, 9])
```

sum of columns

- If we specify `axis=1`, `a.sum` will "compress" along axis 1.

In [91]:
```
a.sum(axis=1)
```

Out[91]:
```
array([ 6, 15])
```

sum of rows.

- You can use `[square brackets]` to **slice** rows and columns out of an array, too.

- The general convention is:

```
array[<row positions>, <column positions>]
```

In [94]:
```
a
```

Out[94]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [95]:
```
# Accesses row 0 and all columns.
a[0, 1]
```

Out[95]: 2

In [98]:
```
a[0, :]
```

Out[98]: array([1, 2, 3])

In [99]:
```
a[0]
```

Out[99]: array([1, 2, 3])

In [100]:
```
a[:, 1]
```

Out[100]: array([2, 5])

?

Activity

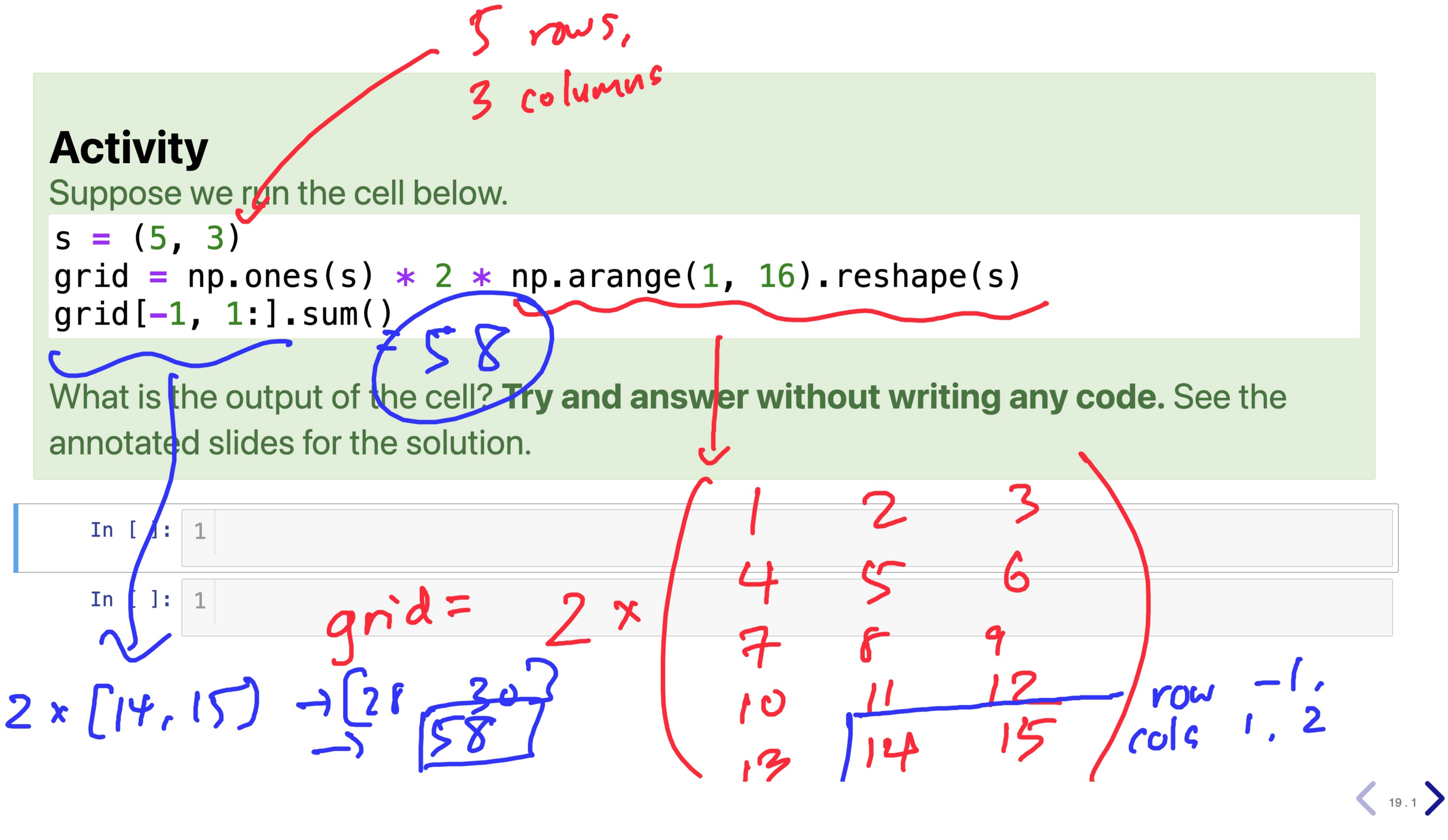
Suppose we run the cell below.

```
s = (5, 3)
grid = np.ones(s) * 2 * np.arange(1, 16).reshape(s)
grid[-1, 1:].sum()
```

What is the output of the cell? **Try and answer without writing any code.** See the annotated slides for the solution.

In [ ]: 1

In [ ]: 1



value, green value, and blue value. Each of these can vary from 0 (least intensity) to 255 (most intensity).

Experiment with RGB colors here.

In [102]:
```
1 img = np.asarray(Image.open('imgs/junior.jpeg'))
2 img
```

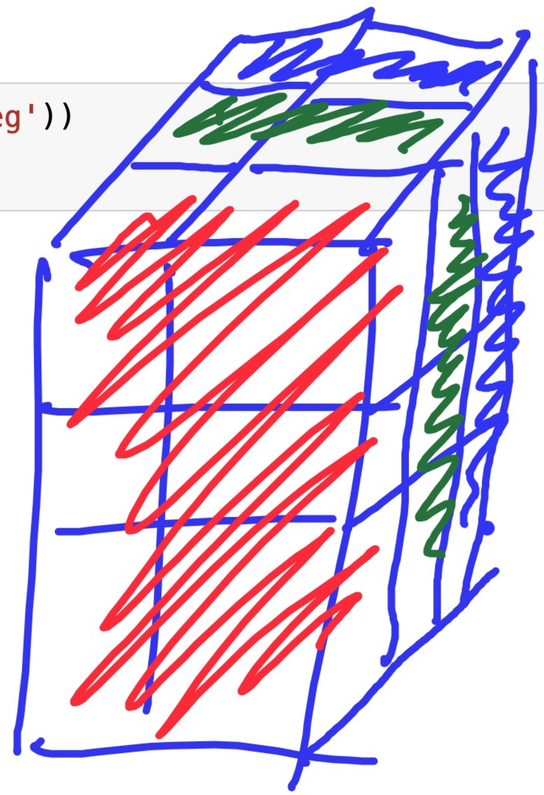
Out[102]:
```
array([[[ 98,  62,  46],
        [ 89,  56,  39],
        [ 88,  56,  41],
        ...,
        [123,  78,  59],
        [125,  78,  60],
        [128,  81,  63]],

       [[ 96,  60,  44],
        [ 89,  56,  39],
        [ 89,  57,  42],
        ...,
        [124,  79,  60],
        [125,  78,  60],
        [127,  80,  62]],

       [[ 94,  58,  42],
        [ 89,  56,  39],
        [ 89,  57,  42],
        ...,
        [125,  78,  60],
        [125,  78,  60],
        [126,  79,  61]],

       ...,

       [[ 89,  50,  11],
        [ 85,  46,   7],
        [ 81,  42,   3],
```



Matrix multiplication

- In the coming weeks, we'll start to rely more and more on tools from linear algebra.
  You'll need this in Homework 2!

- Suppose the matrix A and vectors $\vec{x}$ and $\vec{y}$ are defined as follows:

$$A = \begin{bmatrix} 2 & -5 & 1 \\ 0 & 3 & 2 \end{bmatrix}_{2\times 3} \qquad \vec{x} = \begin{bmatrix} 1 \\ -1 \\ 4 \end{bmatrix}_{3\times 1} \qquad \vec{y} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}_{2\times 1}$$

In [ ]:
```
A = np.array([[2, -5, 1],
              [0,  3, 2]])
x = np.array([[1],
              [-1],
              [4]])
y = np.array([[3],
              [-2]])
```

(3, 1)

[1, -1, 4]

- Suppose the matrix A and vectors $\vec{x}$ and $\vec{y}$ are defined as follows:

$$A = \begin{bmatrix} 2 & -5 & 1 \\ 0 & 3 & 2 \end{bmatrix} \qquad \vec{x} = \begin{bmatrix} 1 \\ -1 \\ 4 \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$$

In [115]:
```
A = np.array([[2, -5, 1],
              [0,  3, 2]])
x = np.array([[1],
              [-1],
              [4]])
y = np.array([[3],
              [-2]])
```

- We can use `numpy` to compute various quantities involving A, $\vec{x}$, and $\vec{y}$.

  For instance, what is the result of the product $A\vec{x}$?

  See the annotated slides for the math worked out.

$$(2)(1) + (-5)(-1) + 1(4)$$
$$= 2 + 5 + 4 = 11$$

In [116]:
```
A @ x
```

Out[116]:
```
array([[11],
       [ 5]])
```

- Suppose the matrix A and vectors $\vec{x}$ and $\vec{y}$ are defined as follows:

$$A = \begin{bmatrix} 2 & -5 & 1 \\ 0 & 3 & 2 \end{bmatrix} \qquad \vec{x} = \begin{bmatrix} 1 \\ -1 \\ 4 \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$$

In [115]:
```
A = np.array([[2, -5, 1],
              [0,  3, 2]])
x = np.array([[1],
              [-1],
              [4]])
y = np.array([[3],
              [-2]])
```

- We can use `numpy` to compute various quantities involving A, $\vec{x}$, and $\vec{y}$.

  For instance, what is the result of the product $A\vec{x}$?

  See the annotated slides for the math worked out.

In [118]:
```
A * y
```

not matrix multiplication!

Out[118]:
```
array([[  6, -15,   3],
       [  0,  -6,  -4]])
```

- The sequence of Fibonacci numbers,

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

can be computed using matrix multiplication!

- It can be shown (with induction!) that if $f_1 = 1, f_2 = 1$, and $f_n = f_{n-1} + f_{n-2}, n \geq 2$, then:

$$\begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

In [125]:
```
fib = np.array([[1, 1],
                [1, 0]])
```

In [128]:
```
fib ** 8
```

Out[128]:
```
array([[1, 1],
       [1, 0]])
```

In [127]:
```
np.linalg.matrix_power(fib, 8)
```

Out[127]:
```
array([[34, 21],
       [21, 13]])
```

not the same!

Random sampling

- `np.random.choice` and `np.random.multinomial` allow us to draw random **samples**.

- `np.random.choice` returns randomly selected element(s) from the provided sequence.
  By default, this is done **with** replacement, but it can be done without replacement, too.

In [227]:
```
unique_names = np.unique(np.load('data/sp25-names.npy', allow_pickle=True))
unique_names
```

Out[227]:
```
array(['Abhinav', 'Alvin', 'Andrew', ..., 'Toby', 'Trent', 'Yu'],
      dtype=object)
```

In [233]:
```
(unique_names == 'Suraj').sum()
```

Out[233]:
```
1
```

by default, sample WITH REPLACEMENT.

In [232]:
```
np.random.choice(unique_names, 5)
```

Out[232]:
```
array(['Suraj', 'Suraj', 'Andrew', 'Andrew', 'Abhinav'], dtype=object)
```

In [ ]:
```
...
```

Random sampling

- `np.random.choice` and `np.random.multinomial` allow us to draw random **samples**.

- `np.random.choice` returns randomly selected element(s) from the provided sequence.
  By default, this is done **with** replacement, but it can be done without replacement, too.

In [227]:
```
unique_names = np.unique(np.load('data/sp25-names.npy', allow_pickle=True))
unique_names
```

Out[227]:
```
array(['Abhinav', 'Alvin', 'Andrew', ..., 'Toby', 'Trent', 'Yu'],
      dtype=object)
```

In [233]:
```
(unique_names == 'Suraj').sum()
```

Out[233]:
```
1
```

by default, sample WITH REPLACEMENT

In [232]:
```
np.random.choice(unique_names, 5)
```

Out[232]:
```
array(['Suraj', 'Suraj', 'Andrew', 'Andrew', 'Abhinav'], dtype=object)
```

In [ ]:
```
...
```

Example: Coin flipping

- **Question**: What is the probability that I see between 40 and 50 heads, inclusive, when I flip a fair coin 100 times?

- To estimate this probability, we need to:
    - Flip 100 fair coins and write down the number of heads,
    - and repeat that process many, many times.

In [291]:
```
1 np.random.multinomial(100, [0.5, 0.5])
```

Out[291]:
```
array([50, 50])
```

In [292]:
```
1 np.random.multinomial(100, [0.5, 0.5], 100_000)
```

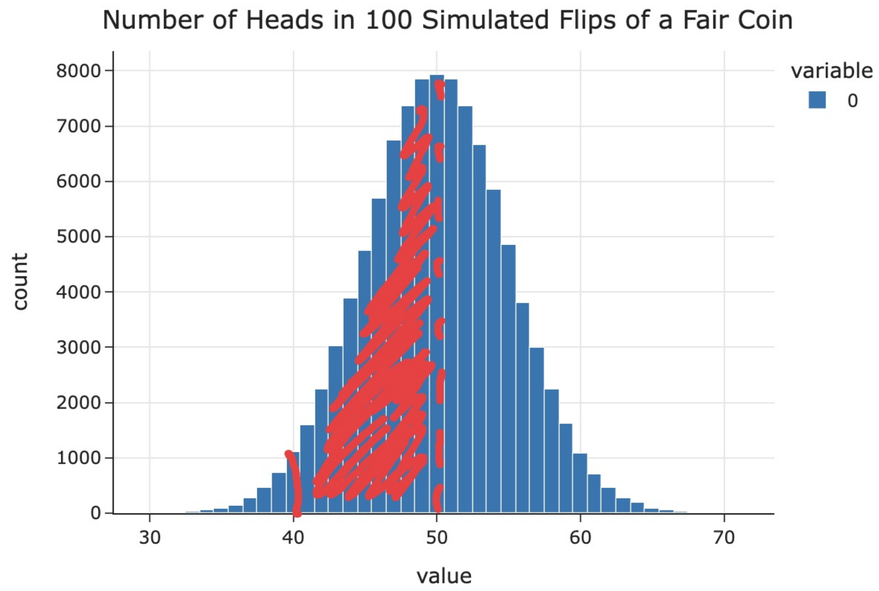
Out[292]:
```
array([[43, 57],
       [50, 50],
       [53, 47],
       ...,
       [54, 46],
       [55, 45],
       [55, 45]])
```

10 million coins flipped!!!..

In [ ]:
```
1 # outcomes is an array with 100,000 elements,
2 # each of which is the number of heads in 100 simulated flips of a fair coin.
3 outcomes = ...
4 outcomes
```

Estimating a probability from empirical results

In [301]:
```
1 px.histogram(outcomes, title='Number of Heads in 100 Simulated Flips of a Fair Coin')
```



$P(40 \leq \text{heads} \leq 50)$

$$\frac{\text{area}\ \square}{\text{area}\ \cup}$$

Example: Airplane seats ✈️

- A **permutation** of a sequence is a reshuffling of its elements.
  `np.random.permutation` returns a permutation of the specified sequence.

- Suppose a flight on Wolverine Airlines is scheduled for n passengers, all of whom have an assigned seat.

- The airline loses track of seat assignments and everyone sits in a random seat.
  What is the probability that *nobody* is in their originally assigned seat?

$$\frac{1}{e}.$$