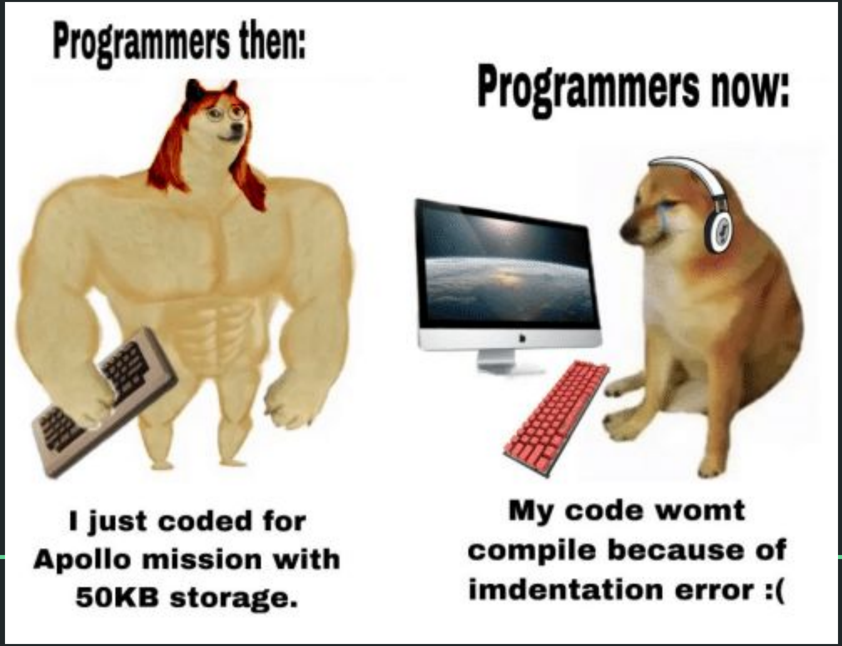
matrices in python

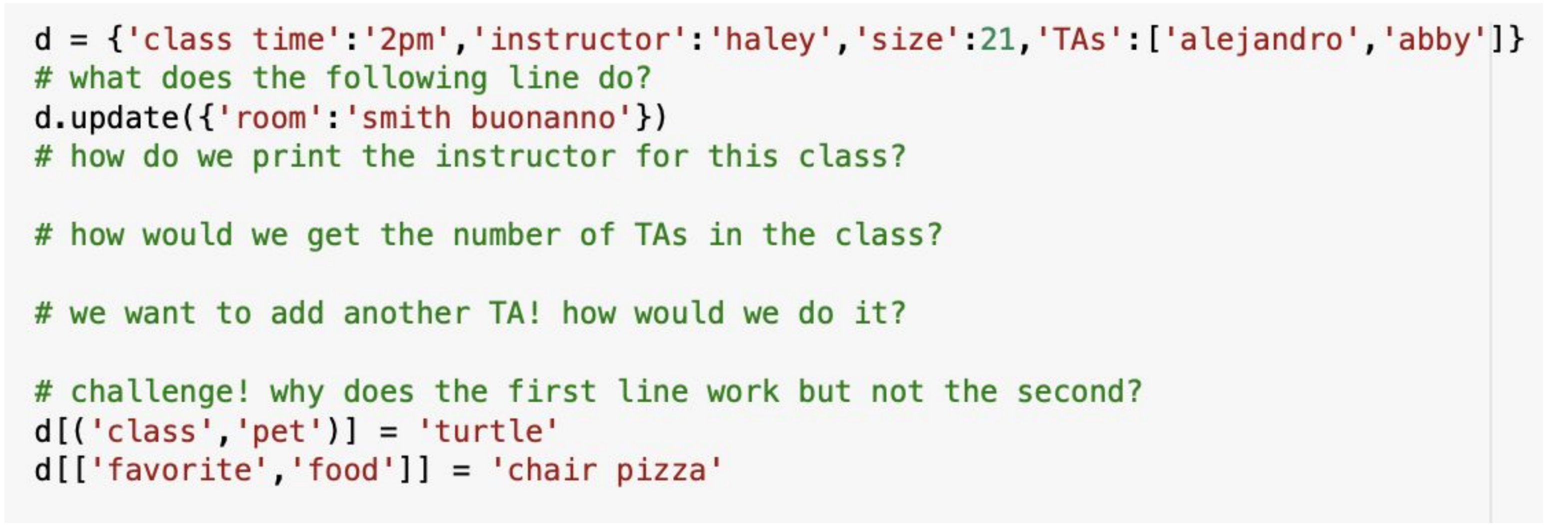

Programmers then:
Programmers now:
I just coded for Apollo mission with 50KB storage.
My code womt compile because of imdentation error :(

protip about indentation! you can use your command key with] or [to move out or in one indent, respectively (also works in matlab, and google docs for bullet points)

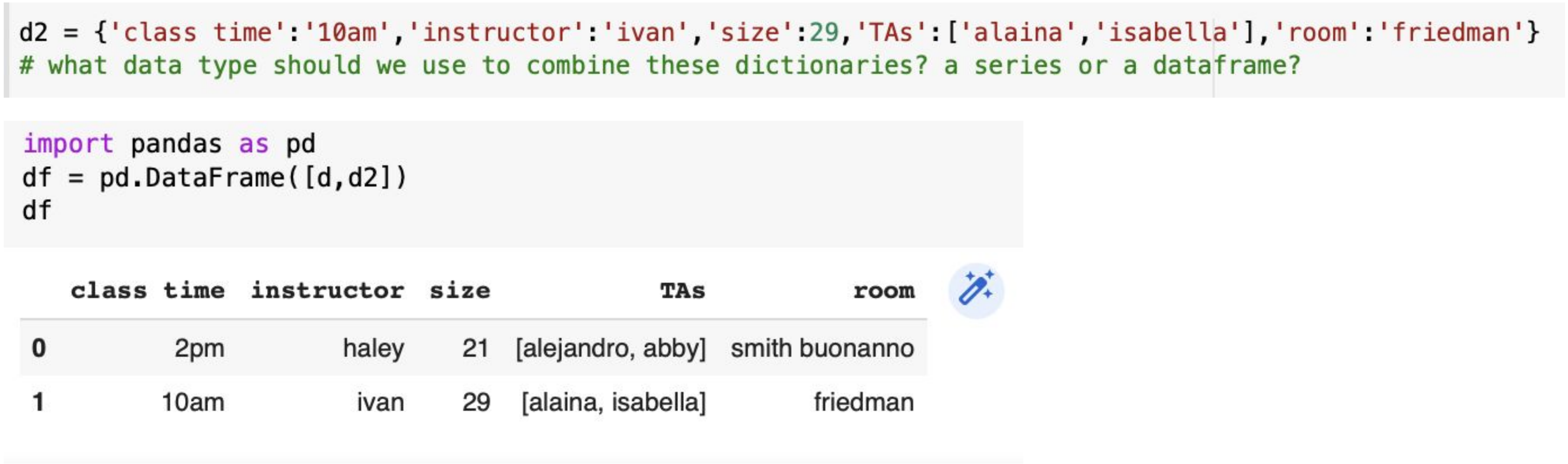
helpful documentation:
web.stanford.edu/class/physics91si/2014/handouts/matlab-python-xref.pdf

warmup / review from last time

pandas, lambdas, dictionaries, tuples



```python
d = {'class time':'2pm','instructor':'haley','size':21,'TAs':['alejandro','abby']}
# what does the following line do?
d.update({'room':'smith buonanno'})
# how do we print the instructor for this class?

# how would we get the number of TAs in the class?

# we want to add another TA! how would we do it?

# challenge! why does the first line work but not the second?
d[('class','pet')] = 'turtle'
d[['favorite','food']] = 'chair pizza'
```



```python
d2 = {'class time':'10am','instructor':'ivan','size':29,'TAs':['alaina','isabella'],'room':'friedman'}
# what data type should we use to combine these dictionaries? a series or a dataframe?
```

```python
import pandas as pd
df = pd.DataFrame([d,d2])
df
```

	class time	instructor	size	TAs	room
0	2pm	haley	21	[alejandro, abby]	smith buonanno
1	10am	ivan	29	[alaina, isabella]	friedman

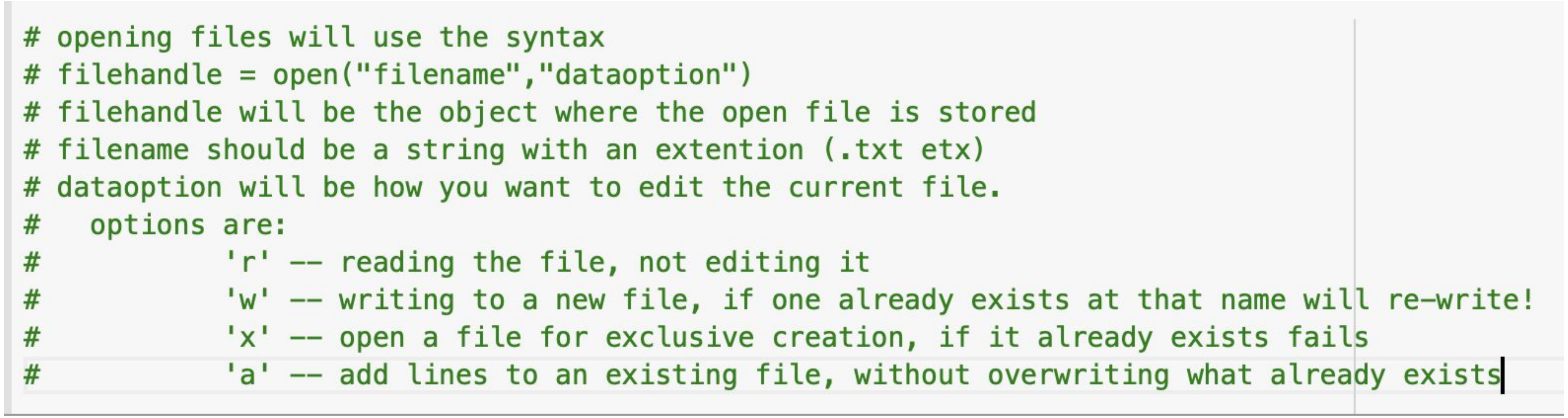
```python
# want it to be morning is 'am' or afternoon if 'pm'
df['time_of_day'] = df.apply(                  )
```

input / output

python makes it easy to load in and out data to/from files and to create files yourself!

pandas can do this with csvs and txt files, but today we'll demo plain text

```
fid = open('helloworld.txt','w')
```

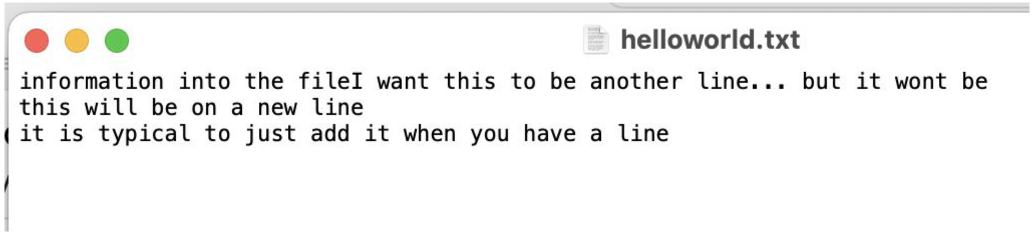


```
# opening files will use the syntax
# filehandle = open("filename","dataoption")
# filehandle will be the object where the open file is stored
# filename should be a string with an extention (.txt etx)
# dataoption will be how you want to edit the current file.
#   options are:
#           'r' -- reading the file, not editing it
#           'w' -- writing to a new file, if one already exists at that name will re-write!
#           'x' -- open a file for exclusive creation, if it already exists fails
#           'a' -- add lines to an existing file, without overwriting what already exists
```

input / output

files are open until you close them in python! you should also close them once you are done with them

```python
fid = open('helloworld.txt','w')
fid.write('information into the file')
fid.write('I want this to be another line... but it wont be')
# need to add a special character \n which means newline!
# more special characters: https://chercher.tech/python-programming/python-special-characters
fid.write('\n')
fid.write('this will be on a new line\n')
fid.write('it is typical to just add it when you have a line\n')
fid.close()
```


helloworld.txt

```
information into the fileI want this to be another line... but it wont be
this will be on a new line
it is typical to just add it when you have a line
```

input / output

now that we’ve written to a file, we might also want to read it in. python can read by line (looking for the newline we just learned – in all your documents without you even knowing!) or by character. you can also read in a whole file.

```python
fid = open('helloworld.txt','r')
# read in the first four characters from the file
my_data = fid.read(4)
print(my_data)
my_data2 = fid.read(4)
print(my_data2)
```

```
info
rmat
```

the same line of code is yielding different results… why?

```python
fid = open('helloworld.txt','r')
my_data = fid.read()
print(my_data)
```

```
information into the fileI want this to be another line... but it wont be
this will be on a new line
it is typical to just add it when you have a line
```

```python
fid = open('helloworld.txt','r')
for one_line in fid: # fid is iterable!
  print(one_line)
```

```
information into the fileI want this to be another line... but it wont be

this will be on a new line

it is typical to just add it when you have a line
```

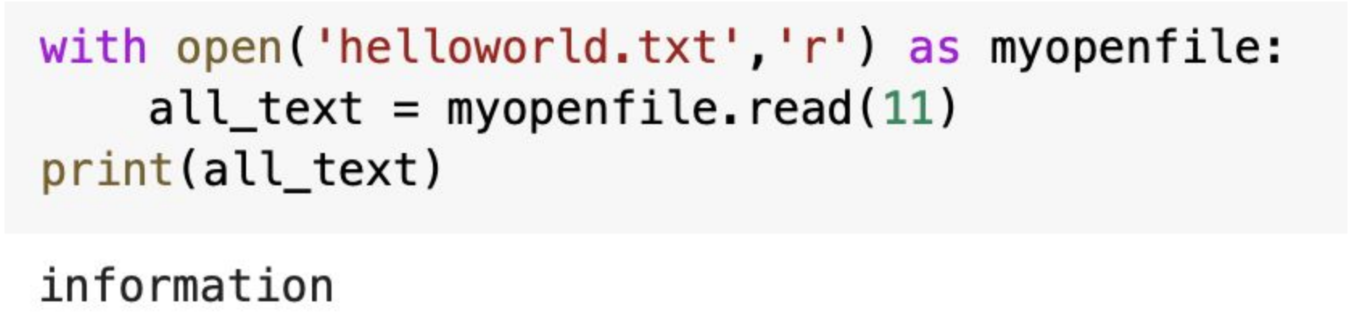
```python
fid = open('helloworld.txt','r')
all_lines = fid.readlines()
all_lines
```

```
['information into the fileI want this to be another line... but it won
 'this will be on a new line\n',
 'it is typical to just add it when you have a line\n']
```

what data type is the result of .readlines() ?

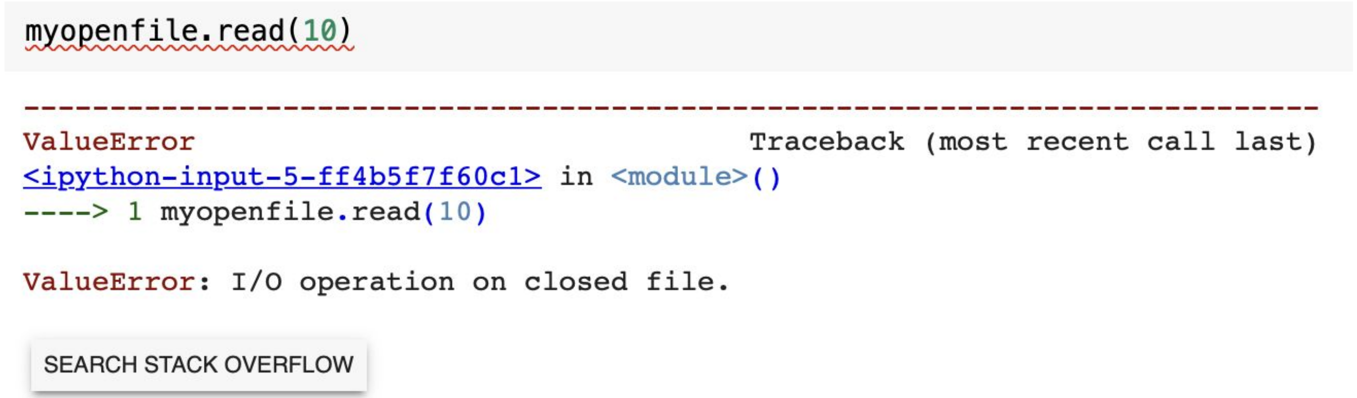
input / output

if you want to open, read in, and close your file you can use the syntax **with** to do it all at once



```python
with open('helloworld.txt','r') as myopenfile:
    all_text = myopenfile.read(11)
print(all_text)
```

```
information
```



```python
myopenfile.read(10)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-ff4b5f7f60c1> in <module>()
----> 1 myopenfile.read(10)

ValueError: I/O operation on closed file.
```

exceptions and errors

we usually try to avoid errors in our code as much as possible, but they are there to tell us something important and often give advice as to *why* the code is malfunctioning. for example if you get a **type error**, you know you have tried to use the metaphorical “chair” as a pizza ingredient, and likely passed a function the wrong data type.

there are times when it makes sense to have your code throw or catch its own errors! can you think of a reason as to when this might be helpful?

ideas:

-

there are other times when you can anticipate errors and fix or relay them, which is what we will show today. this is (part of) what it means to write *robust* code – instead of just breaking, it will either fix errors or otherwise anticipate them to give the user a helpful message

try - except - else/finally

```python
def is_factor(a,b):
    # takes in two numbers, and returns if a is evenly divisible by b
    # returns 1 if it is divisible, 0 otherwise for any numbers.
    try:
        remainder = a % b;
    except ZeroDivisionError: # divide by 0
        is_divisible = 0
    except TypeError: # modulo without numbers
        print('no result! please pass two numbers')
        is_divisible = 'no'
    else: # the code ran!
        if remainder == 0:
            is_divisible = 1
        else:
            is_divisible = 0
    finally:
        if is_divisible == 0 or is_divisible == 1:
            return is_divisible
```

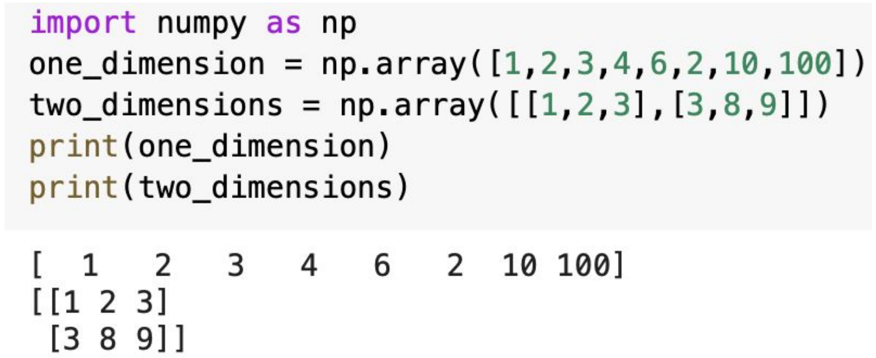
```python
print(is_factor(10,2))
print(is_factor(10,3))
print(is_factor(10,0))
print(is_factor(10,'two'))
```

```
1
0
0
no result! please pass two numbers
None
```

numpy!

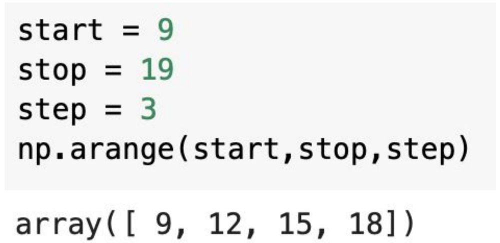
for all of you who have been missing matlab, here is the package for you :)

numpy (number python) is when we can use arrays and matrices and the same types of commands we used there (diag, eye, transpose…) on data here. not going to go over everything in lecture (see the tutorials) but basics on data creation:



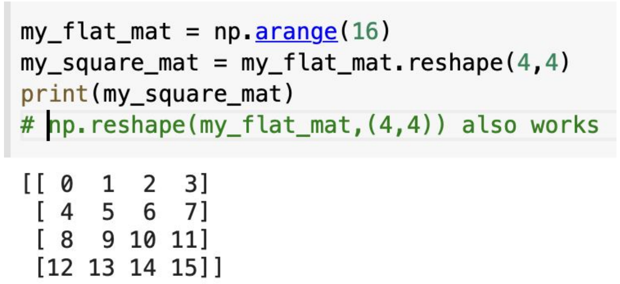
```
import numpy as np
one_dimension = np.array([1,2,3,4,6,2,10,100])
two_dimensions = np.array([[1,2,3],[3,8,9]])
print(one_dimension)
print(two_dimensions)
```

```
[   1    2    3    4    6    2   10 100]
[[1 2 3]
 [3 8 9]]
```



```
start = 9
stop = 19
step = 3
np.arange(start,stop,step)
```

```
array([ 9, 12, 15, 18])
```



```
my_flat_mat = np.arange(16)
my_square_mat = my_flat_mat.reshape(4,4)
print(my_square_mat)
# np.reshape(my_flat_mat,(4,4)) also works
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

you can do reshape with -1 if you know one of the dimensions but not the other!

numpy still uses (row,column) notation

```
my_square_mat[2,3] = 1000
my_square_mat
```

```
array([[   0,    1,    2,    3],
       [   4,    5,    6,    7],
       [   8,    9,   10, 1000],
       [  12,   13,   14,   15]])
```

```
my_square_mat[:,3] = 64
my_square_mat
```

```
array([[ 0,  1,  2, 64],
       [ 4,  5,  6, 64],
       [ 8,  9, 10, 64],
       [12, 13, 14, 64]])
```

```
my_square_mat[:,3] = np.power(my_square_mat[:,0],2)
my_square_mat
```

```
array([[  0,   1,   2,   0],
       [  4,   5,   6,  16],
       [  8,   9,  10,  64],
       [ 12,  13,  14, 144]])
```

```
my_square_mat[:,3] = np.power(my_square_mat[:,0],my_square_mat[1,:])
my_square_mat
```

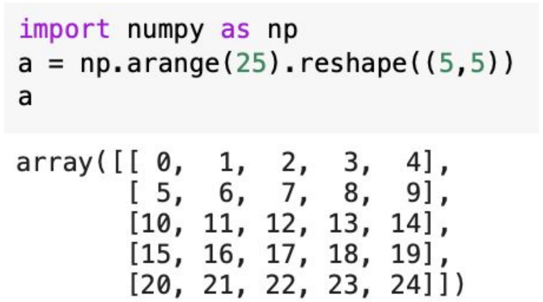
```
array([[     0,      1,      2,      0],
       [     4,      5,      6,   1024],
       [     8,      9,     10, 262144],
       [    12,     13,     14,      0]])
```

most numpy functions accept arrays or single values so they can apply math across large data structures!

numpy has great documentation!

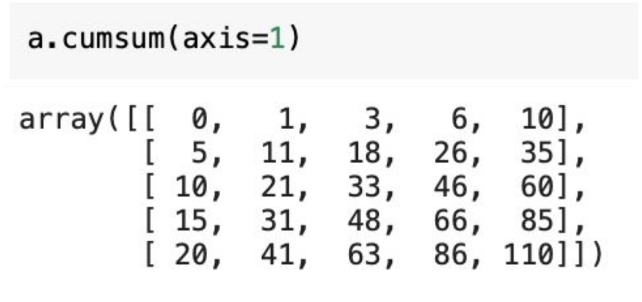
https://numpy.org/doc/stable/reference/generated/numpy.linspace.html

what do you think these numpy commands do?



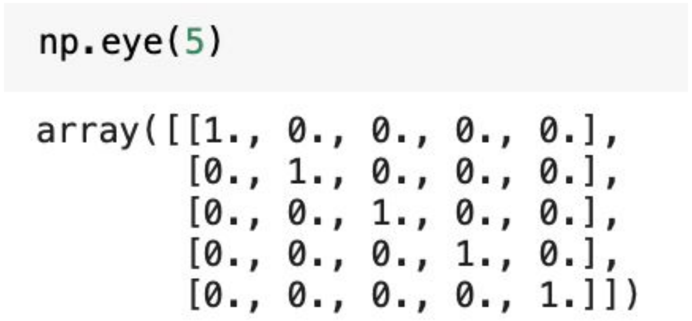
```
import numpy as np
a = np.arange(25).reshape((5,5))
a
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```



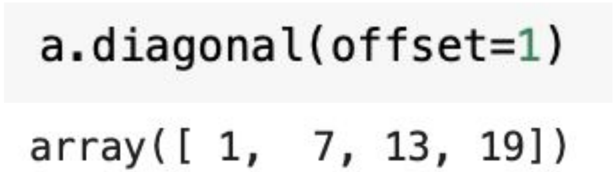
```
a.cumsum(axis=1)
```

```
array([[  0,   1,   3,   6,  10],
       [  5,  11,  18,  26,  35],
       [ 10,  21,  33,  46,  60],
       [ 15,  31,  48,  66,  85],
       [ 20,  41,  63,  86, 110]])
```



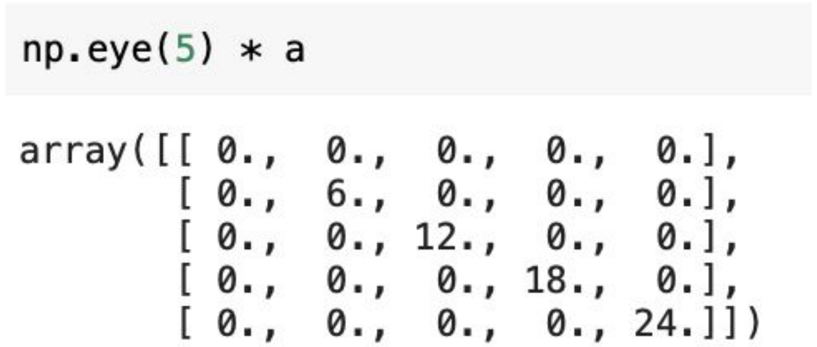
```
np.eye(5)
```

```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```



```
a.diagonal(offset=1)
```

```
array([ 1,  7, 13, 19])
```



```
np.eye(5) * a
```

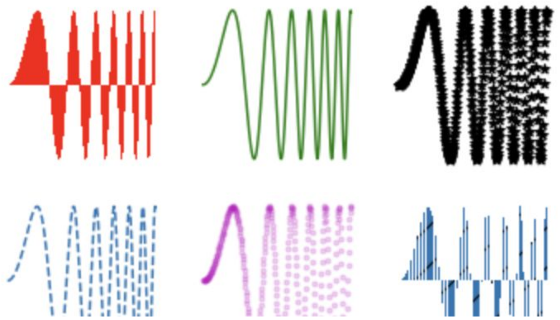
```
array([[ 0.,   0.,   0.,   0.,   0.],
       [ 0.,   6.,   0.,   0.,   0.],
       [ 0.,   0.,  12.,   0.,   0.],
       [ 0.,   0.,   0.,  18.,   0.],
       [ 0.,   0.,   0.,   0.,  24.]])
```

matplotlib

and to accompany our matlab::python for matrices? a matlab::python for plotting!

```
x = np.linspace(0, 2 * np.pi, 400)
y = np.sin(x ** 2)

plt.subplot(2,3,1)
plt.bar(x,y,0.05,color='r')
plt.axis('off')

plt.subplot(2,3,2)
plt.plot(x,y,color='g')
plt.axis('off')

plt.subplot(2,3,3)
plt.scatter(x,y,marker='*',color='k')
plt.axis('off')

plt.subplot(2,3,4)
plt.plot(x,y,linestyle='--')
plt.axis('off')

plt.subplot(2,3,5)
plt.scatter(x,y,marker='.',alpha=0.2,color='m')
plt.axis('off')

plt.subplot(2,3,6)
plt.bar(x,y,0.01,hatch='/')
plt.axis('off')
```

```
(-0.3196592653589793, 6.602844572538565, -1.099989860936727, 1.099994904020213)
```

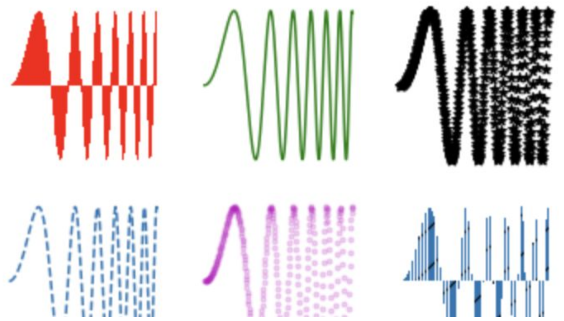


```python
fig,ax = plt.subplots(2,3)
print(ax.shape)
print(type(ax))
```

```
(2, 3)
<class 'numpy.ndarray'>
```

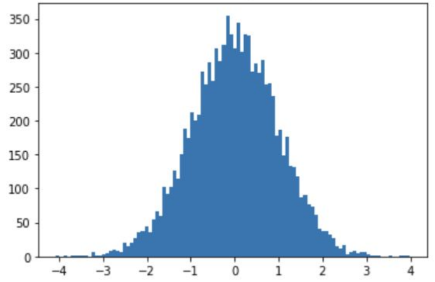
```python
x = np.linspace(0, 2 * np.pi, 400)
y = np.sin(x ** 2)

fig,ax = plt.subplots(2,3)

plt.sca(ax[0,0])
plt.bar(x,y,0.05,color='r')
plt.axis('off')

plt.sca(ax[0,1])
plt.plot(x,y,color='g')
plt.axis('off')

plt.sca(ax[0,2])
plt.scatter(x,y,marker='*',color='k')
plt.axis('off')

plt.sca(ax[1,0])
plt.plot(x,y,linestyle='--')
plt.axis('off')

plt.sca(ax[1,1])
plt.scatter(x,y,marker='.',alpha=0.2,color='m')
plt.axis('off')

plt.sca(ax[1,2])
plt.bar(x,y,0.01,hatch='/')
plt.axis('off')
```

```
(-0.3196592653589793, 6.602844572538565, -1.099989860936727, 1.099994904020213)
```



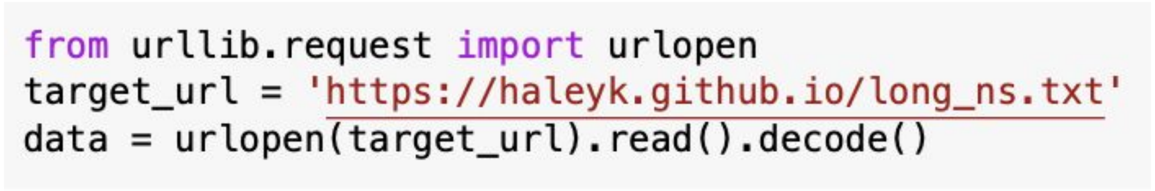
```python
ns = np.random.normal(size=10000)
plt.hist(ns,bins=100)
```

```
(array([  1.,   0.,   1.,   0.,   1.,   1.,   1.,   1.,   1.,   0.,   6.,
          2.,   2.,   3.,   4.,   8.,  10.,   9.,   6.,  21.,  15.,  21.,
         27.,  36.,  38.,  44.,  36.,  55.,  66.,  60., 103.,  93., 103.,
        126., 114., 150., 188., 175., 213., 201., 208., 272., 254., 286.,
        259., 306., 288., 311., 355., 327., 307., 345., 301., 327., 325.,
        272., 285., 271., 289., 254., 255., 236., 178., 186., 148., 176.,
        133., 132., 118.,  87.,  90.,  77.,  74.,  62.,  40.,  38.,  38.,
         32.,  27.,  15.,  12.,  16.,   3.,   6.,   9.,   4.,   6.,   6.,
          3.,   2.,   1.,   2.,   0.,   0.,   1.,   0.,   0.,   1.,   1.,
          1.]),
 array([-4.06778128, -3.98730085, -3.90682041, -3.82633998, -3.74585955,
        -3.66537912, -3.58489869, -3.50441826, -3.42393783, -3.34345739,
        -3.26297696, -3.18249653, -3.1020161 , -3.02153567, -2.94105524,
        -2.86057481, -2.78009437, -2.69961394, -2.61913351, -2.53865308,
        -2.45817265, -2.37769222, -2.29721179, -2.21673135, -2.13625092,
        -2.05577049, -1.97529006, -1.89480963, -1.8143292 , -1.73384877,
        -1.65336833, -1.5728879 , -1.49240747, -1.41192704, -1.33144661,
        -1.25096618, -1.17048575, -1.09000531, -1.00952488, -0.92904445,
        -0.84856402, -0.76808359, -0.68760316, -0.60712273, -0.52664229,
        -0.44616186, -0.36568143, -0.285201  , -0.20472057, -0.12424014,
        -0.04375971,  0.03672073,  0.11720116,  0.19768159,  0.27816202,
         0.35864245,  0.43912288,  0.51960331,  0.60008375,  0.68056418,
         0.76104461,  0.84152504,  0.92200547,  1.0024859 ,  1.08296633,
         1.16344677,  1.2439272 ,  1.32440763,  1.40488806,  1.48536849,
         1.56584892,  1.64632935,  1.72680979,  1.80729022,  1.88777065,
         1.96825108,  2.04873151,  2.12921194,  2.20969237,  2.29017281,
         2.37065324,  2.45113367,  2.5316141 ,  2.61209453,  2.69257496,
         2.77305539,  2.85353583,  2.93401626,  3.01449669,  3.09497712,
         3.17545755,  3.25593798,  3.33641841,  3.41689885,  3.49737928,
         3.57785971,  3.65834014,  3.73882057,  3.819301  ,  3.89978143,
         3.98026187]),
 <a list of 100 Patch objects>)
```



choose your own plotting adventure

bit.ly/plotting_adventure

1. using a heatmap (plt.imshow), draw a smiley face from a matrix of 1s and 0s
   a. challenge: in black and yellow
2. using the following equations, plot a pink heart
   a. theta should be sampled between 0 and 2π (getting 100 values should be enough)
   b. $x = 16sin^3(\theta)$; $y = 13cos(\theta) - 5cos(2\theta) - 2cos(3\theta) - cos(4\theta)$
3. draw two histograms with random values chosen from 1) a normal and 2) a poisson distribution
   a. challenge: learn what the alpha parameter does!
4. make a bar plot counting the number times each digit appears in our IDs
   a. 

```python
from urllib.request import urlopen
target_url = 'https://haleyk.github.io/long_ns.txt'
data = urlopen(target_url).read().decode()
```