

# ASTRON 3705 / PHYSICS 3704

---

Statistics and Data Science

Spring 2025



# Instructor Information

---

- ❖ Lecturer: Prof. Jeffrey Newman
  - ❖ (I'll answer to Jeff or Jeffrey, Dr. Newman, Professor, Professor Newman... )
- ❖ Office: 310 Allen Hall
- ❖ Email: [jnewman@pitt.edu](mailto:jnewman@pitt.edu)
- ❖ Phone: (412) 592-3853
- ❖ Office hours: Monday 3-3:30 or by appointment



# Office Hours

---

- ❖ How does Monday 2:30-3:30 PM work for everyone for scheduled office hours?
  - ❖ Different days?
  - ❖ Different times?
- ❖ I am also available by appointment (especially after 9 PM), typically on Zoom.



# Continuing where we left off: so what do we mean by “probability”?

---

- ❖ Probability theory arose out of the analysis of gambling games.
- ❖ In such a well-controlled situation, we can define the probability of an event as the fraction of times it will occur if we infinitely repeat an experiment - i.e., its frequency.

$$P(x) = \lim_{n \rightarrow \infty} \frac{n_x}{n_t}$$

- ❖ This is the oldest definition of probability - the “frequentist” view - but not the only one possible (more on that later). Note P is at most 1.



# An example

---

- ❖ Consider flipping two coins. The possible things that could happen are:
  - ❖ 1st coin heads, second heads
  - ❖ 1st coin heads, second tails
  - ❖ 1st coin tails, second tails
  - ❖ 1st coin tails, second heads
- ❖ If heads and tails are equally likely, then each of these possibilities is equally likely, so we'd expect each of these possibilities to occur  $1/4$  of the time.
- ❖ So the probability that both coins give the same result is  $1/4 + 1/4 = 1/2$ ; if we flip coins an infinite number of times, half the time we'd get this result.



# Let's test this.

---

- ❖ We can't flip coins an infinite number of times - or even a million - in the course of an hour.
- ❖ However, we can emulate that process in a computer.
- ❖ A simulation where we randomly generate data in some way is generally referred to as a *Monte Carlo* simulation.
- ❖ They are an excellent way to test statistical methods - or to interpret what is going on in your data.



# Getting ready to run Python

---

- ❖ Who hasn't used UNIX before?
- ❖ Has anyone not installed miniforge python yet?



# Preparing to use Python

---

- ❖ Previously, I used Anaconda for this class...
  - ❖ No longer being treated as free for university use
- ❖ Instead we will use:
  - ❖ miniforge : to install python and manage packages
  - ❖ Visual Studio Code (vscode): as a programming environment



# Preparing to use Python

---

## 1) Install miniforge

- ❖ Go to <https://github.com/conda-forge/miniforge> to find instructions for your OS to download and install
- ❖ Follow those instructions!



# Preparing to use Python

---

- 2) Set up a new python environment to work in for the class
  - ❖ We will use the mamba package manager for this; see [https://mamba.readthedocs.io/en/latest/user\\_guide/mamba.html#mamba](https://mamba.readthedocs.io/en/latest/user_guide/mamba.html#mamba) for details
  - ❖ mamba is like conda (used by Anaconda), but much faster



# Preparing to use Python

---

- ❖ In a terminal window, type:

```
mamba init
```

- ❖ Then start a new shell / terminal. In that terminal, type:

```
mamba create --name NAME_OF_YOUR_ENVIRONMENT
```

```
mamba activate NAME_OF_YOUR_ENVIRONMENT
```

```
mamba install datascience jupyter matplotlib astropy scikit-learn
```

- ❖ This switches to the new environment and installs the listed packages
- ❖ `NAME_OF_YOUR_ENVIRONMENT` can be whatever string name you want to call the environment; e.g., it could be `datascience` for this class



# Preparing to use Python

---

- ❖ If you want Python to use this environment in new terminals, you need to add

`mamba activate NAME_OF_YOUR_ENVIRONMENT`

to the end of your `.bash_profile` / `.zshenv` file.

- ❖ Otherwise you will be returned to the `base` environment in new terminals



# Preparing to use Python

---

## 3) Download and install Visual Studio Code

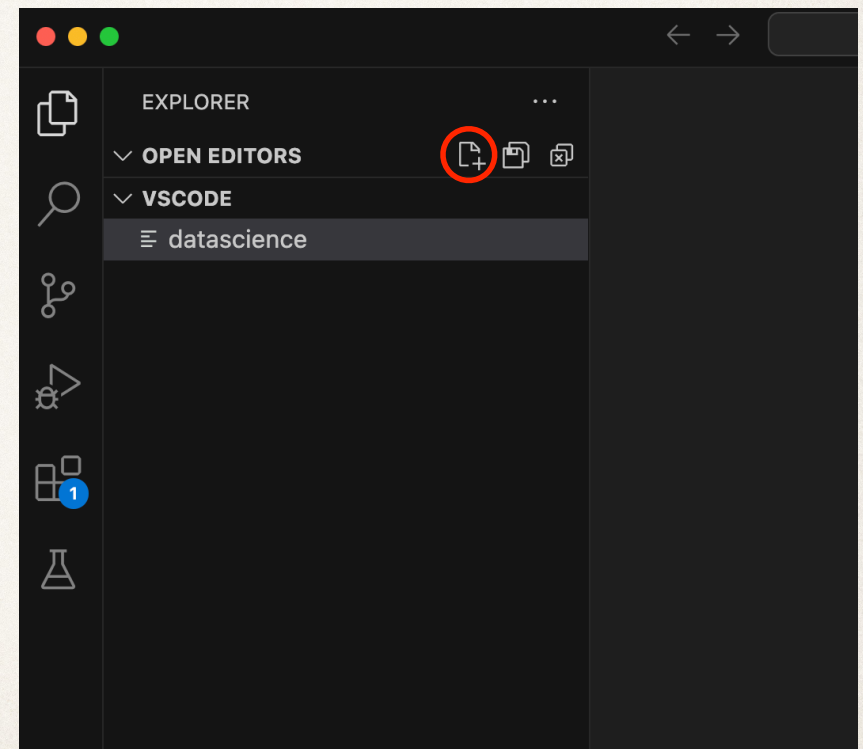
- ❖ You can get it at <https://code.visualstudio.com/>
- ❖ Then download and install the vscode Python extension from <https://marketplace.visualstudio.com/items?itemName=ms-python.python>



# Preparing to use Python

4) Start up vscode and test it

- ❖ Near the top left corner, use the indicated icon to open a new text file

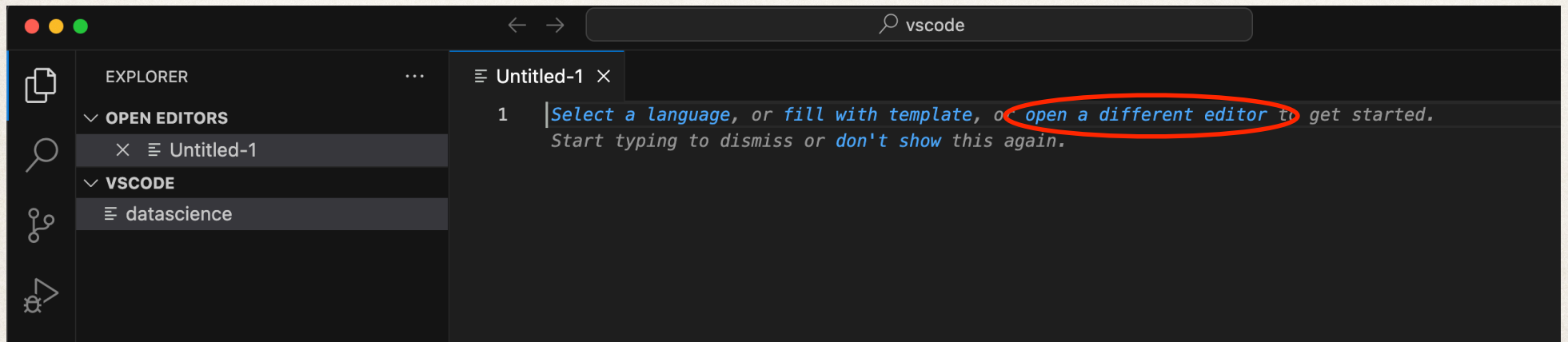




# Preparing to use Python

## 4) Start up vscode and test it

- ❖ In the new window, you can use 'select a language' or (easier) 'open a different editor' to select Python

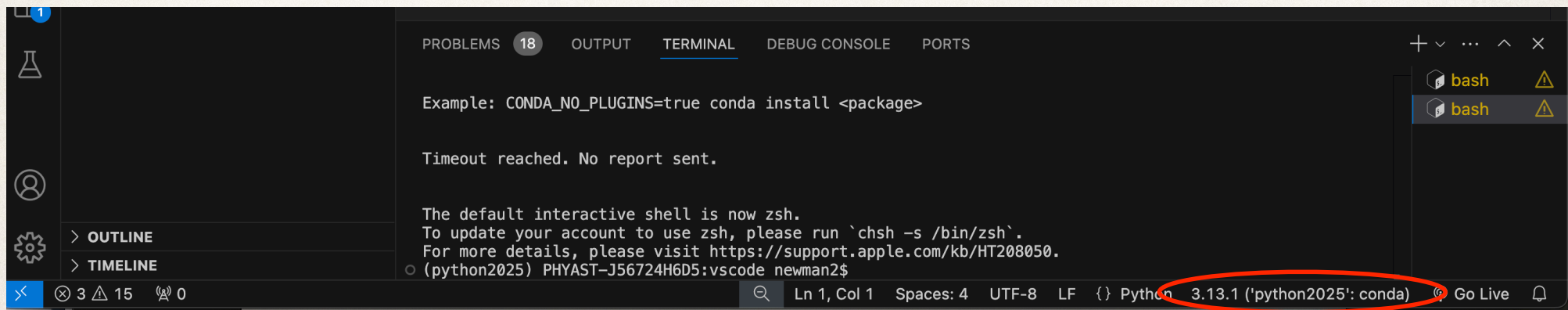




# Preparing to use Python

## 4) Start up vscode and test it

- ❖ Be sure to set vscode to use your new virtual environment!
- ❖ Your current python environment is listed near the bottom right; click on that to choose the right one (which will become default)





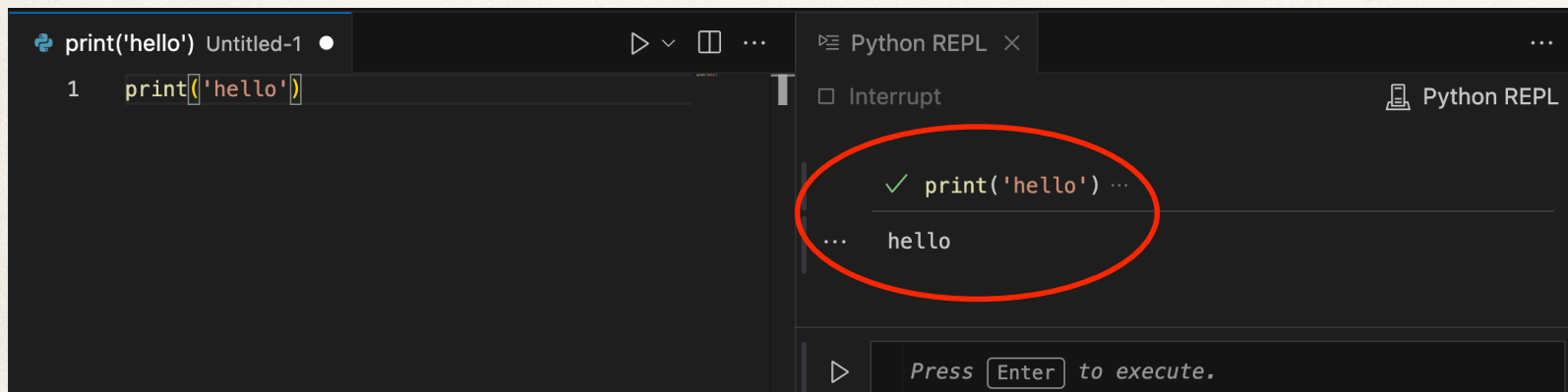
# Preparing to use Python

---

- ❖ Now, let's see if things are working
- ❖ In the editor, type

```
print('hello')
```

and press shift-enter (i.e., hold down shift and press enter). The results hopefully look something like this...



The screenshot shows a Python REPL interface. On the left, a code editor window titled 'print('hello') Untitled-1' contains the line `1 print('hello')`. On the right, the 'Python REPL' window shows the execution of the code. A red circle highlights the output, which consists of a green checkmark, the code `print('hello') ...`, and the string `hello`. Below the output, there is a prompt `...` and a message `Press [Enter] to execute.`



# Preparing to use Python

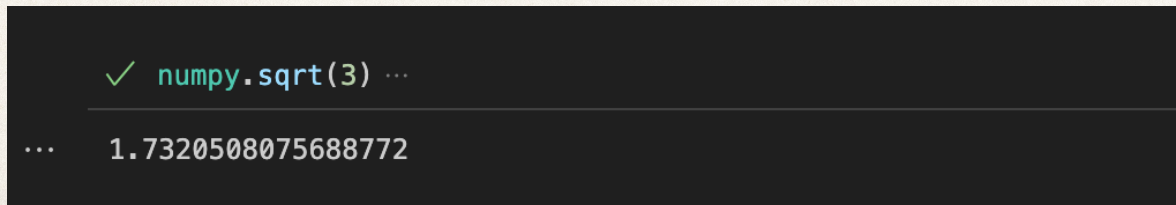
---

- ❖ Next, In the editor, type

```
import numpy  
numpy.sqrt(3)
```

and press shift-enter after each line. The results hopefully look something like this... if so you should be ready for Monday!

- ❖ The results show up in a 'Python REPL' window - that stands for Read, Evaluate, Print, Loop : basically testing code one line at a time



```
✓ numpy.sqrt(3) ...  
... 1.732050807568772
```



# Warming up with Python

---

Python allows you to issue commands at a prompt. It is an interpreted language (like IDL or Matlab); it can process commands one at a time.

Compare to a language like c : we didn't have to declare the variable first, or set its type. One thing to watch out for, though: operating on a variable in Python can change its type!

For today, I want us to work in what vscode calls an 'interactive window'.

- 1) In your file, right click and choose 'Run in interactive window - Run current file in interactive window'
- 2) That should open a new window at the right of the screen. Enter any further commands in it, not the file (or enter them in the file and then choose to run them in the interactive window). **We do not want to use the Python REPL window - some things won't work there!**



# Procedure

---

**For items marked with an \* :**

- 1) Each person in the group should predict the result
- 2) Everyone should compare predictions
- 3) Execute the command to test your prediction
- 4) Everyone should try to explain the result. If you can't explain it, ask me!



# Warming up with Python

---

Using the Python prompt:

- 1) Create a variable `a`, and set its value to the integer 2
- 2) Calculate `a` times 3
- 3) Calculate `a` divided by 3
- \* 4) Calculate `a` cubed



# Finding out about a variable (or function!)

---

- ❖ At the prompt, do:

`?a`      `# or`

`a?`

- ❖ Compare to what you get from

`print(a)`

- ❖ `?` is a 'magic' command built into ipython/jupyter but not standard python. It is most useful for finding out variable types (and, later, array sizes): in this case, `a` is an integer variable, abbreviated `int`, and has value 2.
- ❖ Note 1: `%whos` provides basic information on all variables in memory and can also be useful.
- ❖ Note 2: If you copy commands to the python prompt, you'll sometimes need to do it line by line, not as a block, to see output from each one.



# Changing variable types

---

**Try all of these (remember, if you copy do only one at a time):**

`a + 2`

`a + 2.0`

`a / 3`

`a // 3`

`a // 3.0`

If you apply a simple mathematical operation other than (non-integer) division to an integer with another integer, you get an integer

If you combine an integer with a floating point variable (something with a decimal or in E notation, e.g. 1000.=1E3), the integer is converted to a float before calculation, except when you apply integer division via `//` (in which case it is temporarily treated as an int for the division)



# In Python, there are generally many ways to do something....

---

❖ Try:

`a`

❖ `a` on its own is a Python expression, which gets evaluated to produce output. **Also try:**

`print(a)`

- ❖ `print()` is a standard Python function -- there are a very limited set of functions that make up the standard language.
- ❖ `print()` outputs the contents of whatever is passed to it as a parameter (here, `a`).



# Functions in Python

---

- ❖ As in many languages, the inputs for a Python function are put in parentheses. Going off of other languages, you might expect:

```
b=sqrt(a)
```

to work. **Try it!**

- ❖ There are VERY FEW mathematical functions built in to Python. The numpy package will provide most common functions. **Try:**

```
import numpy as np
```

```
b=np.sqrt(a)
```

```
b
```

Bring up the help on `np.sqrt` (via `?np.sqrt`).



# Functions in Python

---

- ❖ NOTE 1: In many cases, `??routine_name` will bring up source code in addition to the help information (not true here, though). **vscode** **does** provide a direct link to the source code at the top of the help information (try clicking on it!)
- ❖ NOTE 2: Python will often change the type of a variable if needed to make sense as input to a function!



# Packages in Python

---

- ❖ Packages are sets of Python functions that you can use in your programs. I had you install a variety of useful packages when you set up miniforge.
- ❖ There are a number of ways to do imports. Some examples:
- ❖ a) import numpy as its own package with routine names that require '`numpy.`' as prefix to indicate which version of e.g. `sqrt` is meant:

```
import numpy
```

- ❖ b) import numpy as its own package with routine names that require '`np.`' as prefix:

```
import numpy as np
```

- ❖ We could also set up an alias like this after importing numpy:

```
np=numpy
```

- ❖ c) Deprecated (**i.e., not a good idea**): import all of numpy as a set of individual routines or modules:

```
from numpy import *
```

- ❖ If you did all of those, each of these should do the same thing:

```
sqrt(a)
```

```
numpy.sqrt(a)
```

```
np.sqrt(a)
```



# Python functions

---

- ❖ Generally, the preferred style of Python programming is to explicitly list the package (abbreviation is OK) with the routine. I.e., in writing a program, use `np.sqrt()`, not `sqrt()`, to avoid the possibility of bugs. `sqrt()` would often do what you want, but not always!.
- ❖ Specifying `np.sqrt` indicates that you specifically want the sqrt routine from the numpy package, and not any other.
- ❖ Some key numpy functions: `exp` ( $=e^x$ ), `sin` (of angle in radians), `log` ( $=\ln(x)$ ), `log10` ( $=\log_{10}(x) = \log(x)$  to an astronomer), `abs` (= absolute value)
- ❖ Ways to find Python routines:
  - ❖ 1) search the documentation (most development environments provide links)
  - ❖ 2) Google for what you want + python : e.g. "logarithm python"



# Examples of Python data types

---

- ❖ Some common data types in Python include floating point (`float`):

```
c=1.5E1
```

```
?c
```

- ❖ boolean (`bool`)

```
c = 5 > 3
```

```
c
```

```
?c
```

- ❖ and string (`str`)

```
c='15'
```

```
*print(c + '10')
```

```
*print(c*3)
```

- ❖ For more details, explore the first part of [https://github.com/astropgh/python-boot-camp-2024/blob/main/notebooks/01-PythonIntro/2\\_Data\\_Structures.ipynb](https://github.com/astropgh/python-boot-camp-2024/blob/main/notebooks/01-PythonIntro/2_Data_Structures.ipynb) . You should be able to load this file into vscode.



# tuples, lists, and arrays

---

- ❖ Like most other languages, in Python a single variable name can indicate a set of data packaged together. Unlike some languages, there are distinct ways of doing this, and the sub-items need not all be of the same type. **Try:**

```
atup=(1, '20', 3)
?atup
```

Here atup is a tuple: indicated by (optional) parentheses & commas

```
blist=[1, '20', 3]
?blist
```

Here blist is a list: indicated by square brackets & commas

```
atup + (1,)
blist + [1]
blist + (1,)
blist + 1
```

For tuples and lists, + indicates concatenation (adding to end), just like with strings.

We can only combine like with like.



# tuples, lists, and arrays

---

Try:

```
ctmp=[1,20,3]
carr = np.array(ctmp)
?carr
```

```
* carr + 1
* carr + [1]
np.sqrt(carr)
```

Here carr is an array: this data type is not built-in to Python, but rather implemented in numpy. (Be sure you have done `import numpy as np`!)

Numpy arrays are more flexible than lists.

Operators and functions will act on all elements of the array, NOT append.

Computation on numpy arrays can be very efficient!



# Why all these types?

---

- ❖ We won't worry much about tuples. Their key feature is that they are 'immutable': we can add elements to a tuple, but we can't change an element once it's in. Python functions often use tuples as outputs or inputs. **Try:**

```
atup[0]=10
```

```
blist[0]=10
```

```
carr[0]=10
```

```
carr
```

Here we try changing the first element of each variable. Note: Python starts counting at 0!

- ❖ Lists are more useful than tuples in most cases, but we'll primarily use arrays. numpy is frequently optimized for operations that do something to a whole array at a time.



# Arrays (and variables in general) are 'objects' in Python

---

- ❖ numpy arrays have a series of functions associated with them, that act on the array itself ('methods'). To see a list, type

`carr.`

and scroll through the list that pops up on screen.

- ❖ Each of these is a function that would be applied to/evaluated based on `carr`.

\* ❖ **Predict and check the values of `carr.max()` and `carr.sum()`**



# Now, let's change how we use Python

---

To eliminate some busywork, I've created an iPython notebook file ("**Lecture 2.ipynb**"). To use it:

- o) Download the file from Canvas, ideally to a directory you will use for all your class notebooks
- 1) Load the notebook into vscode (choose Open in the file menu).
- 2) Read and follow the instructions in the file. **Work with your group** -- stay in sync with each other, ask each other for help/clarification. Things you should actively do in the notebook are generally highlighted in **boldface** text.
- 3) BE SURE TO MOVE THE FILE TO A COMMON DIRECTORY YOU WILL USE FOR THE SEMESTER WHEN YOU ARE DONE FOR THE DAY, IF YOU DIDN'T DOWNLOAD TO ONE WHEN YOU STARTED!



# Initializing arrays

---

- ❖ We need not list all the elements of an array by hand in creating it. e.g.:

```
np.linspace(0,10,100)
```

```
# 100 evenly spaced #s, including 0 and 10
```

```
# anything after a pound sign is a comment
```

```
a = np.linspace(0,10,100)
```

```
*print(a)
```

- ❖ Note: Python functions can be provided parameters separated by commas ('args' or 'arguments'), or via specified keywords with an equals sign ('kwargs' or 'keyword arguments') like `num=` here.



# Initializing arrays

---

- ❖ Some other useful functions for initializing arrays: `np.zeros()`, `np.ones()`, `np.full`, `np.arange()` (compare to `range()` )
- ❖ **1) Make a 50 element array named `threes`, each element of which is 3 .**



# Accessing arrays

---

- ❖ We often only want to look at or use one or some elements of an array in a calculation (this is called 'slicing'). e.g.:

`alin[0],alin[100]`

- Python, like c (which it was developed in) and some other languages, identifies the first element of an array as element ZERO, not ONE!

`alin[0:10]`

\* `alin[0:1]`

`alin[:]`

- So in a 101 element array `akin`, the first element is `alin[0]`, the fifth would be `alin[4]`, and the last is `alin[100]` (or `alin[-1]` !!).

\* `alin[10:20:2]`

`alin[90:-2]`

`alin[-1:90]`

- As an additional complication, when slicing, Python does NOT include the last element specified, but stops at the one before it.



# Variable names in Python are labels

---

- ❖ In Python, multiple variable names can refer to the same intrinsic object in memory.

```
a = [1,2,3,4]
```

```
b=a
```

```
b[3]=10
```

```
b
```

```
a
```

```
from copy import copy
```

```
c=copy(a)    # c = np.copy(a) would create an array
```

```
              # that is a copy of a
```

```
c[3]=5
```

```
a
```

Note: this is not how variables work in most computer languages!



# Variable names in Python are labels

In Python, a "name" or "identifier" is like a parcel tag (or nametag) attached to an object.

```
a = 1
```



Here, an integer 1 object has a tag labelled "a".

If we reassign to "a", we just move the tag to another object:

```
a = 2
```



Now the name "a" is attached to an integer 2 object.

The original integer 1 object no longer has a tag "a". It may live on, but we can't get to it through the name "a".

If we assign one name to another, we're just attaching another nametag to an existing object:

```
b = a
```



The name "b" is just a second tag bound to the same object as "a".

from:

<https://web.archive.org/web/20180411011411/http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html>



# For loops... and avoiding them

---

❖ Let's say:

❖ `x=np.linspace(0,2*np.pi,50000)`

❖ and we want to plot  $\sin(x)$  vs.  $x$ . In most languages, we'd do this with a for loop. In python, there are many ways to do this:

❖ 1) one inefficient way, by appending to a list, then converting to an array:

❖ `y_tmp = []` # create an empty list

❖ `for x1 in x: y_tmp.append(np.sin(x1))`

❖ `y=np.array(y_tmp)`

❖ 2) Another inefficient way: create an array of the right size, and use `enumerate()`, a built-in function in Python, to loop over its elements:

❖ `y=np.zeros(size(x))` # `y = x*0`. would also work

❖ `for i,x1 in enumerate(x): y[i]=np.sin(x1)`



# For loops... and avoiding them

---

❖ Let's say:

```
x=np.linspace(0,2*np.pi,50000)
```

❖ and we want to plot  $\sin(x)$  vs.  $x$ . In most languages, we'd do this with a for loop. In python, there are many ways to do this...

❖ 1) one inefficient way, by appending to a list, then converting to an array:

```
y_tmp = [] # create an empty list
for x1 in x: y_tmp.append(np.sin(x1))
y=np.array(y_tmp)
```

❖ 2) Another inefficient way: create an array of the right size, and use `enumerate()`, a built-in function in Python, to loop over its elements:

```
y=np.zeros(size(x)) # y = x*0. would also work
for i,x1 in enumerate(x): y[i]=np.sin(x1)
```



# For loops... and avoiding them

---

- ❖ 3) Another inefficient way: use a "list comprehension" to calculate all the elements of y, then convert to an array (the `map()` function was an older way to do this)

```
y = [np.sin(x1) for x1 in x]  
y=np.array(y)
```

- ❖ 4) **Much much much more efficient:** numpy functions can operate on numpy arrays all at once

```
y=np.sin(x)
```

- ❖ This is >100 times faster than the other methods! (You can test this with the `%timeit` or `%%timeit` jupyter/ipython magic command)



# Plotting

---

- ❖ Python has powerful capabilities for plotting, whether incidental plots or publication-quality ones. Often, you'll want to plot something to diagnose if your code works, see what results are, etc. Let's do something basic:
- ❖ We created an  $x$  array and  $y=\sin(x)$  array already. Now plot each one:  

```
plt.plot(y)           #plots y vs. index number  
plt.plot(x, 'ro')     #overplots x, as red circles
```

Notes: 1) `plt` is actually an alias for `matplotlib.pyplot`

2) Arbitrary Python expressions can be used as the input to many functions.  
e.g., `plt.plot(2*x)` or `plt.plot(x+y)` work...



# Plotting

---

- ❖ In the previous example we just gave `plt.plot()` an array of values, and it assumed `x=[0,1,2,3,...]`
- ❖ To plot `y` vs. `x`, do (for instance):  
`plt.clf()`      # Clear the plot before doing a new one.  
`plt.plot(x,y,'b-')`
- ❖ Note that you list `X` first, then `Y`!
- ❖ Do `?plt.plot` to see a list of the different plot styles.
- ❖ You can specify color ('r', 'g', 'b', etc. for red / green / blue) then style ('-' for lines, '--' for dashed, ':' for dotted line, '.' for points, 'o' for circles, etc.):
  - ❖ e.g.: `plt.plot(x,y,'k--')` will plot a black dashed line



# If you're looking for something more to do:

---

- ❖ Try to generate  $y=x^{3/2}$  using all the different alternative methods with/without for loops. Confirm that you are getting consistent results. Use `%%timeit` (with a large number of array elements in `x`, e.g. 100k) to compare execution speed.
- ❖ Generate plots of the curves for Gaussian distributions of mean 0 and different sigmas (I expect you to be able to find the formulae you need online). Overplot them, using different colors / line styles / symbols. Use the `label` keyword in the calls to `plt.plot()`, add a legend with `plt.legend()`, and add axis titles with `plt.xlabel()` and `plt.ylabel()`. Search for information on the functions to find the syntax you need!
- ❖ A couple more examples are in the notebook



# Back to our goal: testing the results from coin flips

---

- ❖ Consider flipping two coins. The possible things that could happen are:
  - ❖ 1st coin heads, second heads
  - ❖ 1st coin heads, second tails
  - ❖ 1st coin tails, second tails
  - ❖ 1st coin tails, second heads
- ❖ If heads and tails are equally likely, then each of these possibilities is equally likely, so we'd expect each of these possibilities to occur  $1/4$  of the time.
- ❖ So the probability that both coins give the same result is  $1/4 + 1/4 = 1/2$ ; if we flip coins an infinite number of times, half the time we'd get this result.



# Generating random numbers

---

- ❖ So, first we need to generate random numbers!
- ❖ In Python, we can use the `numpy.random.rand()` function to generate random numbers, evenly distributed between 0 & 1.
- ❖ The syntax is:

```
import numpy.random as random  
randnum=random.rand(ndim1 , ndim2 , ndim3...)
```

where `ndim1`, `ndim2`, etc. are the size of the 1st, 2nd, etc. dimensions of the array (unlike with e.g. `np.zeros`, this is not provided as a tuple; `random.random_sample()` can do the same thing but use tuples to specify dimensions ).



# Making heads & tails from random #s

---

- ❖ We want to have tails 50% of the time and heads 50% of the time.
- ❖ Since `random.rand()` generates random #s evenly distributed between 0 & 1, we can call the result 'tails' if the result is 0-0.5, and 'heads' if it is 0.5-1.
- ❖ How do you do a logical test like that in Python?
  - ❖ With an IF statement, just like in other languages. Try this, repeatedly (hit shift-enter in a code box containing this code) to rerun it:

```
if random.rand(1) < 0.5:  
    print("tails")    #note: indented by 4 spaces, no less!  
else:  
    print ("heads")
```

- ❖ Note: if you really wanted to do this in one line, you could do:

```
result = ("tails" if random.rand(1) < 0.5 else "heads") ; print(result)
```



# Let's break that down:

---

```
if random.rand(1) < 0.5:
    print("tails") #note: indented by 4 spaces, no less!
else:
    print("heads")
```

- ❖ In Python, the logical part of the if statement is followed by a colon. The set of expressions to be executed if it is true (or false) are all indented by 4 spaces to indicate that they are executed together; in other languages you might use braces ( { } ) or BEGIN...END statements for that.

- ❖ I.e.: the syntax is:

```
if (some expression):
    (what to do if that expression is true)
```

- ❖ What is evaluated as 'True' in Python? Try:

```
if 1: print("1: True")      # if statements with no else
if 0: print("0: True")      # can be done in 1 line easily
if 2: print("2: True")
if 0.4: print("0.4: True")
```



# The else part is optional...

---

```
if random.rand(1) < 0.5:
    print("tails")  #note: indented by 4 spaces, no less!
else:
    print("heads")
```

- ❖ The underlying pattern is:

```
if (some expression):
    (what to do if that expression is true)
else:
    (what to do if it's not true)
```

- ❖ The command following else is run if the expression is not true.
  - ❖ But what is going on with `random.rand(1) < 0.5` anyway?
- ❖ This is a *relational* expression in Python: it tests whether the expression on the left is less than (<) the expression on the right.



# Other "relational" tests in Python:

---

- ❖ `<=`: less than or equal to
- ❖ `<`: less than
- ❖ `==`: equal to
- ❖ `!=`: not equal to
- ❖ `>=`: greater than or equal to
- ❖ `>`: greater than
- ❖ We can combine these with standard Boolean logic (`and`, `or`, `not`). e.g.:

`a=4`

```
* if ( (a < 5) and not(a == 7) ): print(a)
```



# Back to coins

---

- ❖ We simulated one coin flip with:

```
if random.rand(1) < 0.5:
    print("tails") #note: indented by 4 spaces, no less!
else:
    print("heads")
```

- ❖ Now, let's do 1000 flips. Remember the syntax:

```
randnum=random.rand(ndim1 , ndim2 , ndim3...):
```

```
Nsims=1000
```

```
print( (random.rand(Nsims) > 0.5).sum(), 'heads, for a fraction of ',(random.rand(Nsims) > 0.5).sum()/Nsims)
```

- ❖ **What happened here?**

- ❖ (Note: we could have gotten the same results as `(...).sum()` by using `np.sum(random.rand(Nsims) > 0.5)` but it's good to get used to using object methods.



# What was the problem?

---

- ❖ The random # generator gets reinitialized whenever you run `random.rand()` !
- ❖ Write a couple of lines of code that will print out the total & fraction of heads for the same sample of 'coin flips' instead of two different samples. A reminder: our old code is:  

```
Nsims=1000  
print( (random.rand(Nsims) > 0.5).sum(), \  
'heads, for a fraction of ', \  
(random.rand(Nsims) > 0.5).sum()/Nsims )
```
- ❖ Once it's working, increase Nsims to 10000, 100\_000, 1\_000\_000, 10\_000\_000 and see what happens when you execute the same command repeatedly (with shift-enter).



# Now, we'll move the goalposts...

---

- ❖ What I'd like us to do is test how the frequency of two coins coming up the same behaves, as the number of simulations increases.
- ❖ We could do this by repeatedly executing similar lines of code, but it's very likely we'd make a mistake.
- ❖ So let's make our very own Python function!
- ❖ A function will take some input, and provide an output (possibly just an indication of successful completion); they can be almost arbitrarily complicated.



# Creating a function

---

- ❖ We want our function to have only one input - the number of tests to do - and one output - the fraction of times we get the same result on both coins.
- ❖ The first line tells Python we are defining a function, and what its inputs are:  
`def sim2coins(ntests):`



# Creating a function

---

```
def sim2coins(ntests):
```

- ❖ We might add a comment telling us what the routine does:

```
# simulate ntests tosses of 2 coins
```

- ❖ Python ignores anything following a # (unless it's within a string variable). All other lines of code in the function should be indented by 4 spaces.

- ❖ Now, let's simulate the first coin, with 0=tails, 1=heads:

```
coin1=random.rand(ntests) > 0.5
```

- ❖ and the second:

```
coin2=random.rand(ntests) > 0.5
```

- ❖ Next, we set what the function will return - it could be any sort of Python variable, array, etc. In Python, returning something is optional.

```
return np.sum( coin1 == coin2 )/ntests
```



# To recap:

---

```
def sim2coins(ntests):  
    # simulate ntests tosses of 2 coins  
    coin1=random.rand(ntests) > 0.5  
    coin2=random.rand(ntests) > 0.5  
    return np.sum( coin1 == coin2 )/ntests
```

To test this out, in a Python code box do:

```
print( sim2coins(1000) )
```

and see if it works! Also try:

```
a=sim2coins(1000)
```



# Putting the function in a file

---

- ❖ 1) Open up a new file in vscode: e.g., hit the new file button next to VSCODE at the top left of the window.
- ❖ In general, Python scripts/programs should be put in files ending in .py , so give your file a filename accordingly: **sim2coins.py**
- ❖ 2) At the top of the file, put the lines:

```
import numpy as np
# we need to do the imports necessary
# for the module within the module's file
import numpy.random as random
```
- ❖ Then copy the text of **sim2coins()** to the new file and save it.
- ❖ You can then load the function into memory with

```
%run sim2coins.py
```
- ❖ This will treat it as a python 'script': a set of commands executed just like they were at a command prompt.



# Putting the function in a file

---

- ❖ One file can actually contain many functions or procedures - e.g. subroutines that are used as part of a main program.
- ❖ The end of each function is indicated by the end of the indented region.
- ❖ The file can also contain python commands, just as you'd enter them at the command line.



# Making new python modules

---

- ❖ If you want to make new python modules (e.g. `scipy.random` is a module, containing python functions like `rand()`), just put all the functions that go together in a file named `YOUR_MODULE_NAME_HERE.py` located in your python search path.
- ❖ To add the directory we just created to this path, you'll need to set up an environment variable. If you use bash or zsh (the previous and current default shells for Macs), the way to do this is to add the following at the end of your `.bash_profile` file:

```
PYTHONPATH="${PYTHONPATH}:/Users/USERNAME/python"
```

```
export PYTHONPATH
```



# Making new python modules

---

❖ In CSH/TCSH, add the following lines at the end of your .cshrc or .tcshrc file:  
`setenv PYTHONPATH ${PYTHONPATH}:/Users/USERNAME/python`

❖ For Windows, it's messier...

❖ After you've done this, you can do things like:

```
import sim2coins as s2c  
print(s2c.sim2coins(1000))
```

❖ Try it out!