Week 1: Start as you mean to go on!

Software engineering for scientific computing

Dr K Clough, Topics in Scientific computing, Autumn term 2023

Important announcements

1. There is one lecture and one lab per week. **I strongly encourage you to attend both in person, and attempt all the exercises.** The lecture will cover the material for the tutorial that happens *the following week*. This will help to consolidate the material since you will see it two weeks running, rather than all in one day.

2. There are 2 courseworks:

   • Item 1: 20% Coursework 1 - quite hard
   (released start of Week 4 and due end of Week 9)

   • Item 2: 80% Coursework 2 - quite straightforward
   (released start of Week 10 and due January 2026).

Teaching approach

1. If the work does not challenge you, the work cannot change you
2. This is a safe space, it is ok to be wrong or confused about simple things
3. I focus on the basics, the goal is to understand these not to show off advanced programming skills
4. You are responsible for your own learning, and you know what format works best for you
5. However, the students that did very well last year came to the lectures and the labs, asked questions and engaged with the course, I knew who they were and they didn’t (blindly) use ChatGPT.

What do you think is the most important thing when writing a scientific / mathematical code?

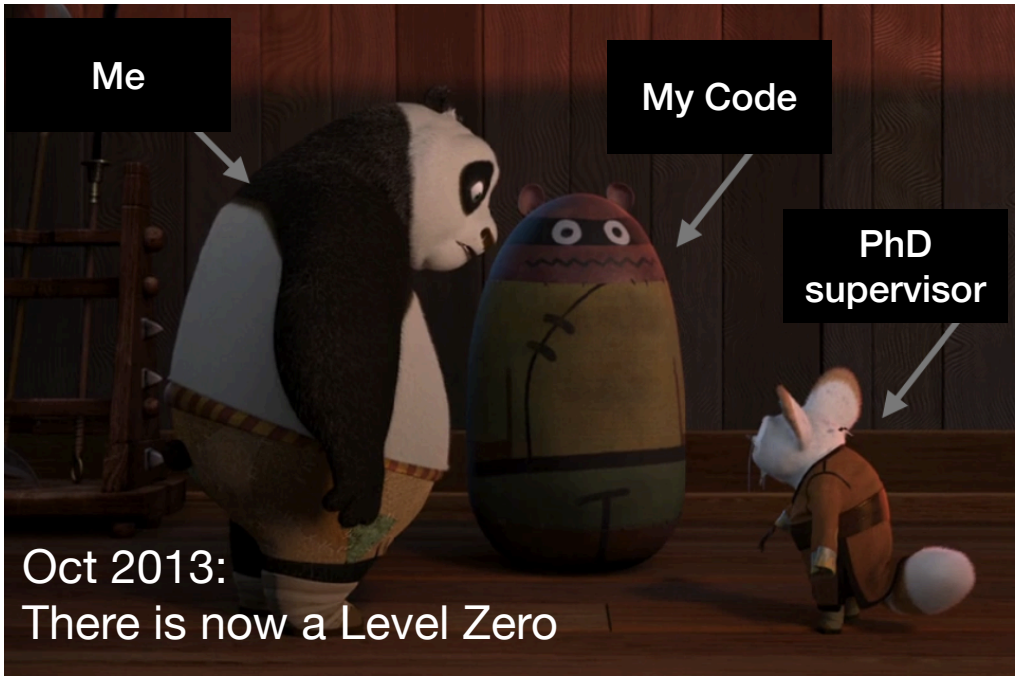
1. Having it run very fast
2. Well tested, getting the right answer
3. Readability / understandability of the code
4. Reproducibility of the results
5. Using the most advanced code tools available

Prioritise these in order of importance

What do you think is the most important thing when writing a scientific / mathematical code?

1. Having it run very fast — Not worth optimising for 2x speed up, for 10x maybe
2. Well tested, getting the right answer — Well yes… as a minimum
3. Readability / understandability of the code — After 10 years of code work I think this is number 1!
4. Reproducibility of the results — UK research councils require this
5. Using the most advanced code tools available — Not really bothered about this!

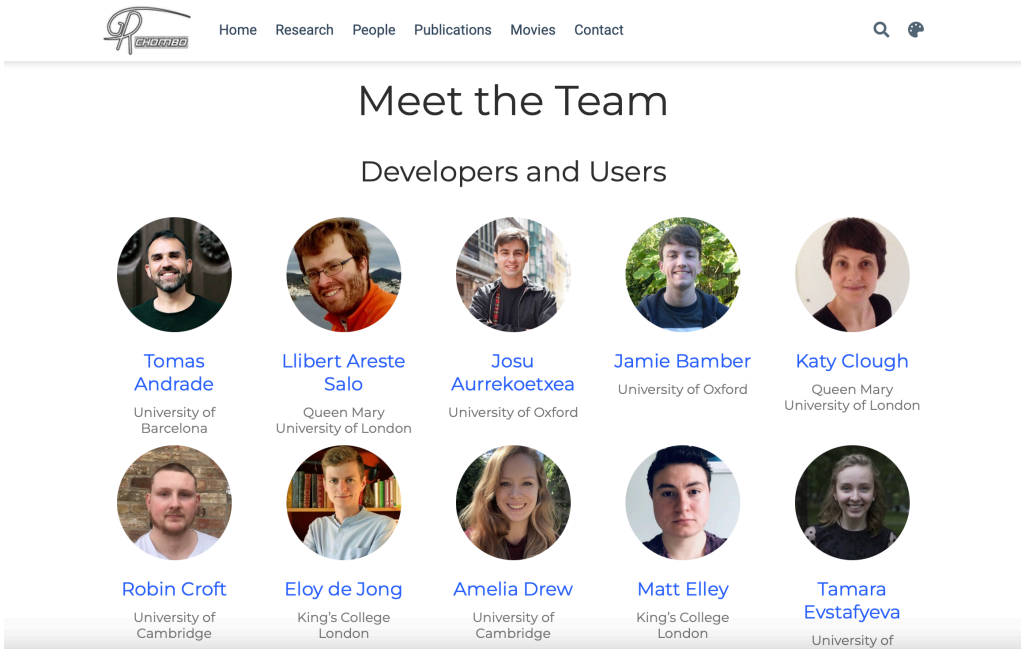
My own experience

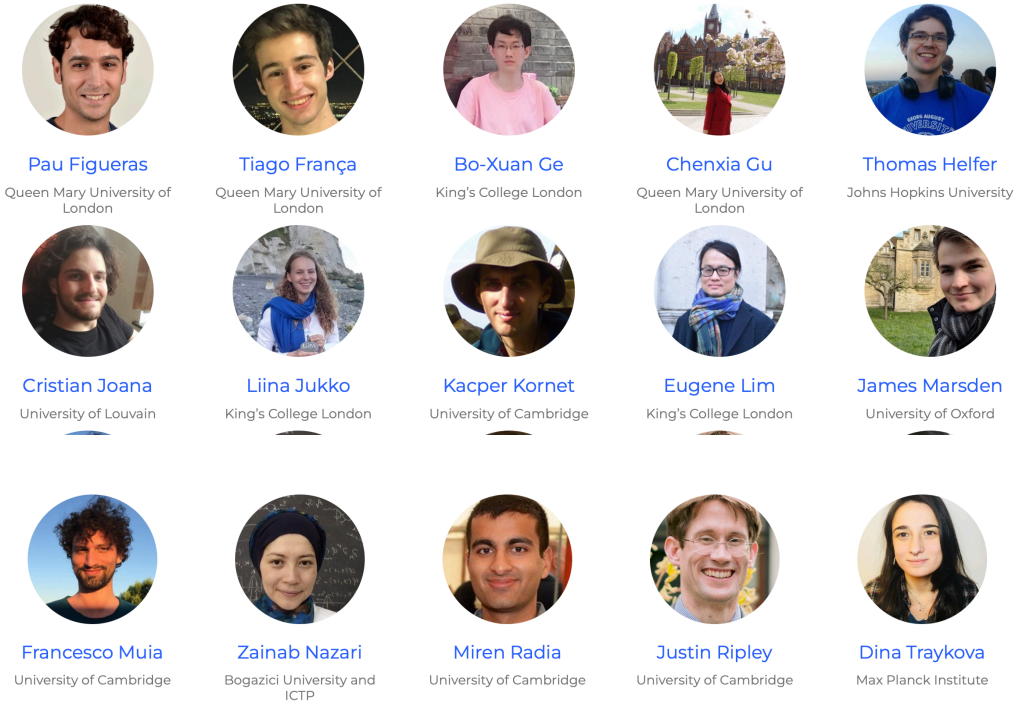


Started my PhD with an unreadable, non working code

```
  // Now we need to fix the algebraic constraints
  const DisjointBoxLayout& level_domain = m_state_new.disjointBoxLayout();
  DataIterator dit0 = level_domain.dataIterator();
  int nbox = dit0.size();
#pragma omp parallel for default(shared) schedule(dynamic)
  for(int ibox = 0; ibox < nbox; ++ibox) {
    DataIndex di = dit0[ibox];
    const Box& b = level_domain[di];
    FArrayBox& state_fab = m_state_new[di];

    FORT_FIXBSSNCONSTRF(CHF_FRAn(state_fab,c_h,s_num_comps_h),
                        CHF_FRAn(state_fab,c_A,s_num_comps_A),
                                 CHF_CONST_REAL(m_dx),
                        CHF_CONST_REAL(m_time),
                        CHF_CONST_REAL(m_p.center[0]),
                        CHF_CONST_REAL(m_p.center[1]),
                        CHF_CONST_REAL(m_p.center[2]),
                        CHF_BOX(b));

  // And enforce non zero chi and non negative alpha condition
  // And check for nan and Inf if onoffparam2 is on

    BoxIterator bit (b);
    for (bit.begin (); bit.ok (); ++bit)
    {

      IntVect iv = bit ();

        if (m_p.onoffparam2 == 1) {

        bool nanerror = 0;

          for (int comp = 0; comp < m_state_new.nComp (); ++comp)
          {
             Real val = state_fab (iv,comp);

             if (isnan(val) || isinf(val) || Abs(val)>1.e40)
                         {
                           pout() << " r = " << sqrt(pow((m_dx*iv[0]-m_p.center[0]),2)
                                                   + pow((m_dx*iv[1]-m_p.center[1]),2)
                                                   + pow((m_dx*iv[2]-m_p.center[2]),2))
                                  << " time = " << m_time
                                  << " comp = " << s_state_names[comp]
                                  << " val = " << val << std::endl;

                           nanerror = 1;

                         }
          }
```

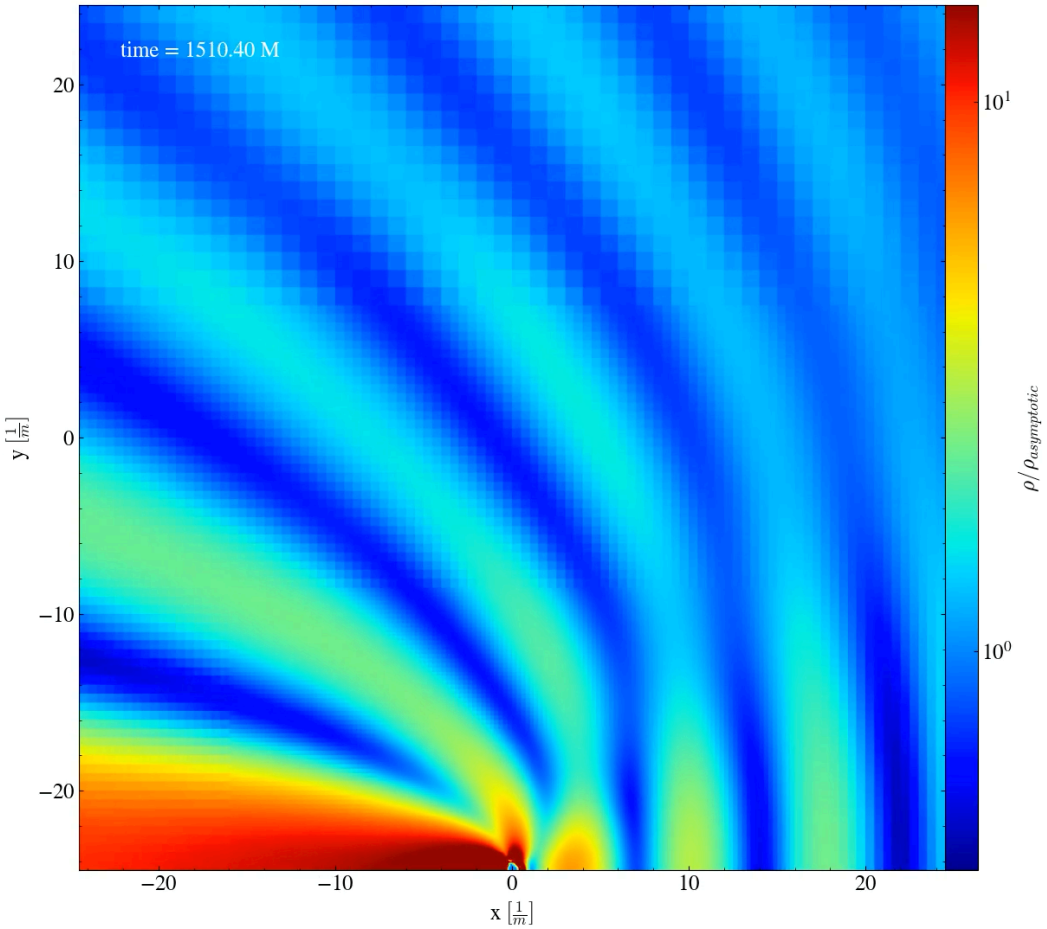
My own experience





Now a large team of developers:
Sharing code makes us all more productive, but relies on it being readable and well tested

My own experience



Simulation of energy density of dark matter around a moving black hole

Plan for today

1. Good grammar for good code - types, variables, assignment, functions, loops, conditionals.
2. Python libraries - NumPy, SciPy and Matplotlib as examples
3. Good coding practise - version control, defensive programming, comments
4. First tutorial - space: the final frontier…

Types

- Main simple types are int (signed integer), float (signed decimal number) bool (boolean - true or false), or str (sequences of characters)
- Python assigns the type automatically, which is *usually* helpful
- A type contains information for the computer about how operations work on that type, for example, what should the + operator does will differ with integers versus strings.
- *(We can also have user defined types called classes that we will discuss later)*

This works

```
In [5]: a = 1
        b = 2
        c = a/b
        print(c)
```

```
0.5
```

This is better

```
In [6]: a = 1.0
        b = 2.0
        c = a/b
        print(c)
```

```
0.5
```

Unless you really meant

```
In [11]: a = 1
         b = 2
         c = int(a/b)
         print(c)
```

```
0
```

Variables

- use lower case for variable names, with underscores between words

- usually they should be NOUNS:

```
number_of_cats = 3
weight_of_cat_kg = 4.5
name_of_cat = 'Fluffy'
```

What are the types here?

- exception for bools which should be named like they are asking a question

```
is_a_cat = True
eats_fish = False
is_fluffy = True
```

Assignment

Assignment of one variable to another in python
is a *label not a copy*
(unlike many other code languages e.g. C++)

```
a = 2
b = a
```

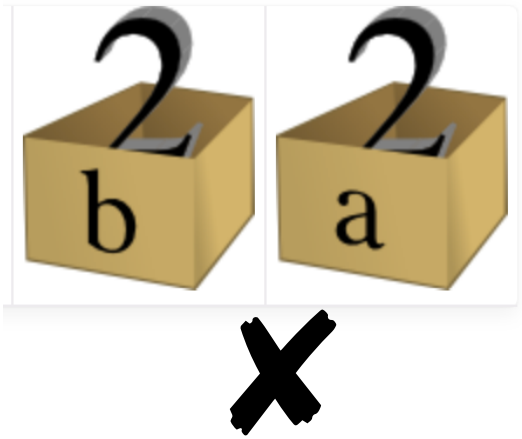

```
a = [1,2,3,4]
b = a
b[0] = 100
print(a)
print(b is a)
print(b == a)
```

What is printed here?

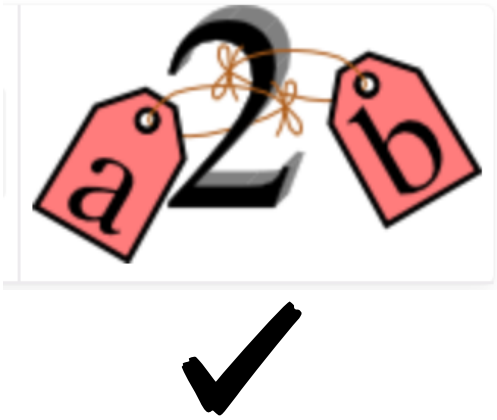
Assignment

Assignment of one variable to another in python is a *label not a copy*
(unlike many other code languages e.g. C++)

```
a = 2
b = a
```



✗



✓

```
a = [1,2,3,4]
b = a
b[0] = 100
print(a)
print(b is a)
print(b == a)
```

```
[100, 2, 3, 4]
True
True
```

Assignment

But sometimes python tries to be “clever”, and decides that you really meant to reassign the variable, not amend the original one.

The place this will come into play most is with arrays, in particular numpy arrays that we will see later. Just be aware of it as a source of bugs and always experiment with a trivial example if unsure, e.g.

```
a = 2
b = a
print(b is a)
b = 3
print(a)
print(b is a)
```

```
True
2
False
```

```
a = [1,2,3,4]
b = [5,6,7,8]
b[:] = a
b[0] = 100
print(a)
```

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

```
a = [1,2,3,4]
b = [5,6,7,8]
b = a
b[0] = 100
print(a)
```

```
[100, 2, 3, 4]
```

Spot the difference here!

Functions

```
def calculate_mass(density, volume) :
    mass = density * volume
    return mass

def is_greater_than_five(x) :
    return (x > 5)
```

- Must indent the **body** of the function
- Needs a colon after the definition
- Takes in **inputs** - may also leave them empty but always need the brackets ()
- Returns an **output** - if none is given the return is void (the value ‘None’ is returned) - my suggestion is to always return a value to confirm success
- Variables declared in the body (e.g. ‘sum’ above) are **local** - they cannot be accessed outside the function, e.g. ‘mass’ above
- Functions do things so they are named as VERBS unless they return a bool in which case they are named in the same way as the boolean variables. Use lower case letters and underscores to separate words as with variables

Functions

```
In [11]: def add_two_numbers(first_number,second_number) :
             sum = first_number + second_number
             return sum

         def f1(a,b) :
             return a+b
```

Q: Why is the first function better than the second one?

Functions

```
In [11]: def add_two_numbers(first_number,second_number) :
             sum = first_number + second_number
             return sum

         def f1(a,b) :
             return a+b
```

Q: Why is the first function better than the second one?

- It has a name that is a verb, in lower case with underscores

- The name is descriptive and the purpose of the function is immediately clear

- The purpose of the inputs and the return value is also clear

- NEVER NEVER use things like a, b, tmp or foo for variable names, you are just hurting someone in the future (probably yourself, or me)

Functions

```
In [11]: def add_two_numbers(first_number,second_number) :
             sum = first_number + second_number
             return sum

         def f1(a,b) :
             return a+b
```

Q: Why might I use a function?

Functions

```
In [11]: def add_two_numbers(first_number,second_number) :
             sum = first_number + second_number
             return sum

         def f1(a,b) :
             return a+b
```

Q: Why might I use a function?

- Makes code more modular
- Therefore more readable
- Therefore easier to debug
- Avoids repetition of code, which again reduces error and makes updating easier (imagine we find a way to speed up a function, we only need to adjust it in one place)

Loops - for

- naming should make iteration clear

- ‘i’ is often used as an index, but I prefer to add ‘i’ to the front of the iterator object

- Very often we will be iterating through arrays in which case we can directly iterate, but often having the index is useful

- Can add a “break” to exit for a given condition - useful for error handling.

```
cheeses = ['edam', 'brie', 'cheddar']
i = 1
for cheese in cheeses :
    print("Cheese number ", i, "is ", cheese)
    i = i + 1
```

```
Cheese number  1 is  edam
Cheese number  2 is  brie
Cheese number  3 is  cheddar
```

Loops - for

- naming should make iteration clear

- ‘i’ is often used as an index, but I prefer to add ‘i’ to the front of the iterator object

- Very often we will be iterating through arrays in which case we can directly iterate, but often having the index is useful

- Can add a “break” to exit for a given condition - useful for error handling.

```
cheeses = ['edam', 'brie', 'cheddar']
i = 1
for cheese in cheeses :
    print("Cheese number ", i, "is ", cheese)
    i = i + 1
```

```
Cheese number  1 is  edam
Cheese number  2 is  brie
Cheese number  3 is  cheddar
```

```
cheeses = ['edam', 'brie', 'cheddar']
for icheese, cheese in enumerate(cheeses) :
    print("Cheese number ", icheese+1, "is ", cheese)
```

```
Cheese number  1 is  edam
Cheese number  2 is  brie
Cheese number  3 is  cheddar
```

Loops - for

- naming should make iteration clear

- ‘i’ is often used as an index, but I prefer to add ‘i’ to the front of the iterator object

- Very often we will be iterating through arrays in which case we can directly iterate, but often having the index is useful

- Can add a “break” to exit for a given condition - useful for error handling.

```
cheeses = ['edam', 'brie', 'cheddar']
i = 1
for cheese in cheeses :
    print("Cheese number ", i, "is ", cheese)
    i = i + 1
```

```
Cheese number  1 is  edam
Cheese number  2 is  brie
Cheese number  3 is  cheddar
```

```
cheeses = ['edam', 'brie', 'cheddar']
for icheese, cheese in enumerate(cheeses) :
    print("Cheese number ", icheese+1, "is ", cheese)
```

```
Cheese number  1 is  edam
Cheese number  2 is  brie
Cheese number  3 is  cheddar
```

```
cheeses = ['edam', 'brie', 'cheddar']
for i, cheese in enumerate(cheeses) :
    print("Cheese number ", i+1, "is ", cheese)
    if (cheese == 'brie') :
        print("I don't like brie!")
        break
```

```
Cheese number  1 is  edam
Cheese number  2 is  brie
I don't like brie!
```

Loops - while

- I rarely use while, since you can usually reframe it as a for loop with a break.

- However, where it better matches the purpose of the loop, it can make the code more readable

```
number_of_cats = 0
while (number_of_cats < 3) :
    number_of_cats += 1
    print("Adding another cat, now have ", number_of_cats)
```

```
Adding another cat, now have  1
Adding another cat, now have  2
Adding another cat, now have  3
```

Loops

Q: What is my_number at the end of these loops?

```
my_number = 0
my_numbers = [1,2,3]
for inum, number in enumerate(my_numbers) :
    my_number += inum
    my_number *= number

print(my_number)

while my_number < 20 :
    my_number += 1

print(my_number)
```

Loops

Q: What is my_number at the end of these loops?

```
my_number = 0
my_numbers = [1,2,3]
for inum, number in enumerate(my_numbers) :
    my_number += inum
    my_number *= number

print(my_number)

while my_number < 20 :
    my_number += 1

print(my_number)
```

```
12
20
```

Conditionals - if, elif, else

- Always cover all the options
- Any nonzero number is interpreted as True, but avoid this and try to always define properly as a bool
- Boolean operators are ‘and’ ‘or’ or ‘not’
- Often used for error checking

```
is_a_cat = True
is_fluffy = False

if (is_a_cat == False) :
    print("It is not a cat")
elif(is_a_cat and is_fluffy) :
    print("It is a fluffy cat")
else :
    print("Something is wrong! Cats are always fluffy!")
```

```
Something is wrong! Cats are always fluffy!
```

Conditionals

Q: What is going wrong here?

```
my_float = 1.0e16
your_float = my_float - 1e-6

if(my_float == your_float) :
    print("We both have the same number")
else :
    print("The numbers are different")
```

```
We both have the same number
```

Conditionals

Usually with floats you want to do something like this:

```
my_float = 1.0
your_float = my_float + 1.0e-6

tolerance = 1.0e-3
if (abs(my_float - your_float) < tolerance) :
    print("We both have (roughly) the same number")
else :
    print("Our numbers are different")
```

```
We both have (roughly) the same number
```

Naming quiz: Which of the following would (usually) be an acceptable name?

1. A bool called fluffy_cat
2. A float called size_of_cat
3. A bool called is_not_hungry
4. A function called my_cat()
5. An integer called my_float_value
6. A function called integrate_area()

Naming quiz: Which of the following would (usually) be an acceptable name?

1. A bool called fluffy_cat — Better called is_fluffy_cat
2. A float called size_of_cat — Good! Better if unit added!
3. A bool called is_not_hungry — In principle ok, but double negatives confusing
4. A function called my_cat() — Should be a verb! What is the function doing?
5. An integer called my_float_value — Why is an integer called float?
6. A function called integrate_area() — Looks good!

Plan for today

1. ~~Good grammar for good code - Types, variables, functions, loops~~
2. Python libraries - NumPy, SciPy and Matplotlib as examples
3. Good coding practise - version control, defensive programming, comments
4. First tutorial - space: the final frontier…

NumPy = numerical python

- Provides an object called an ndarray and routines for acting on them: *mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.*
- Similar to the built in type ‘list’ but allows more rapid operations due to the use of precompiled C functions behind the scenes.

```
import numpy as np
a = np.array([1,2,3,4])
b = np.array([2,4,6,8])
print(a*b)
```

```
[ 2  8 18 32]
```

NumPy = numerical python

```
import numpy as np
print(np.sin(np.pi/2.0))
```

```
1.0
```

```
from numpy import *
print(sin(pi/2.0))
```

```
1.0
```

Q: Why is this dangerous?

NumPy = numerical python

```
import numpy as np
print(np.sin(np.pi/2.0))
```

```
1.0
```

✔

Note the dot allows us to access functions and objects within NumPy (more detail when we learn classes)

```
from numpy import *
print(sin(pi/2.0))
```

```
1.0
```

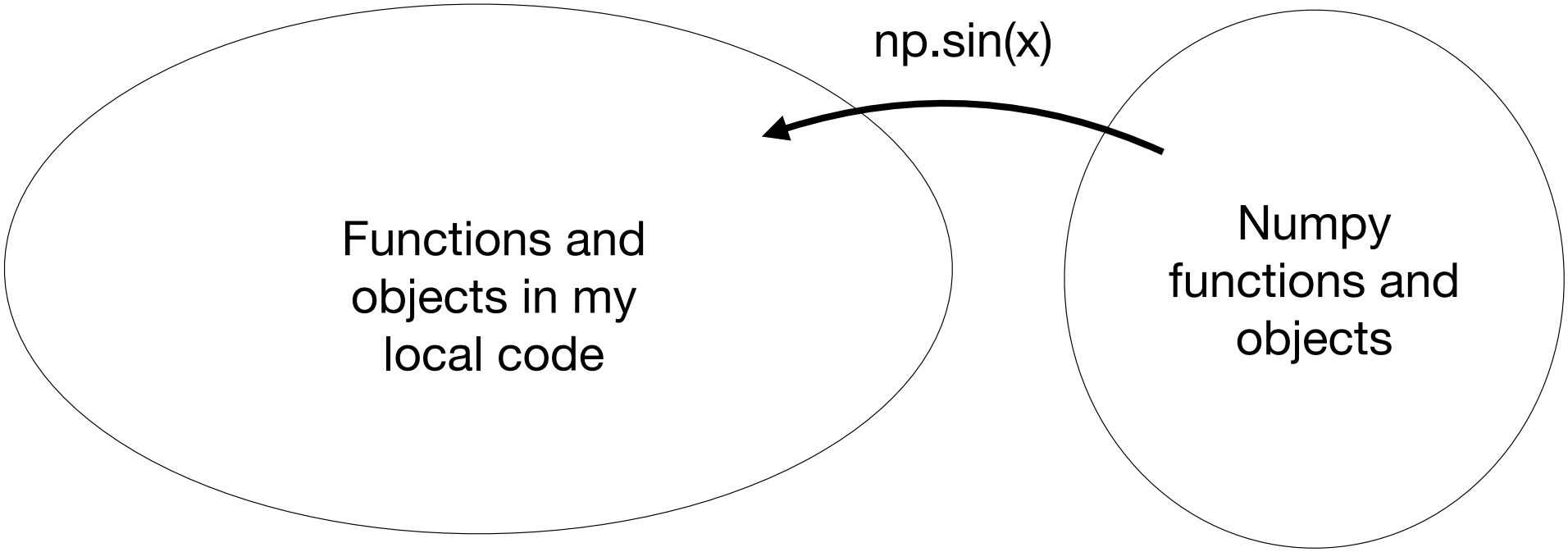
✘

Potential for function overlap - numpy contains a HUGE number of functions and objects
if you define a function called sin()
how will in know which to use?

NumPy = numerical python

```
import numpy as np
print(np.sin(np.pi/2.0))
```

```
1.0
```

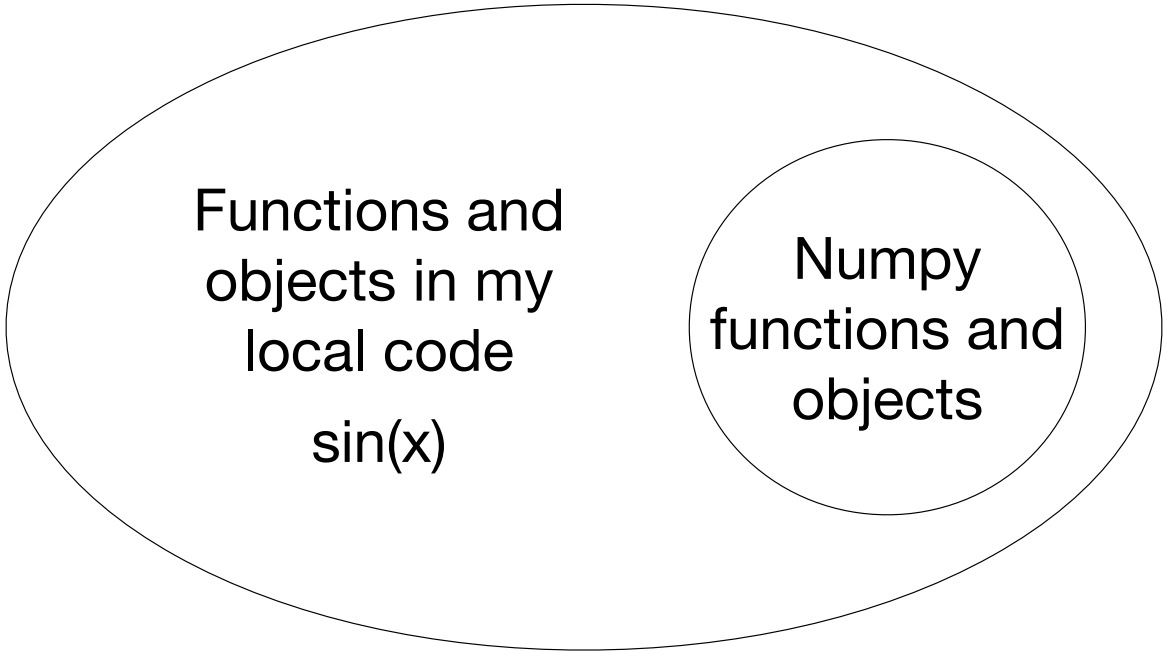


Go and get the function from NumPy and apply it to x - crossing a barrier

NumPy = numerical python

```
from numpy import *
print(sin(pi/2.0))
```

```
1.0
```



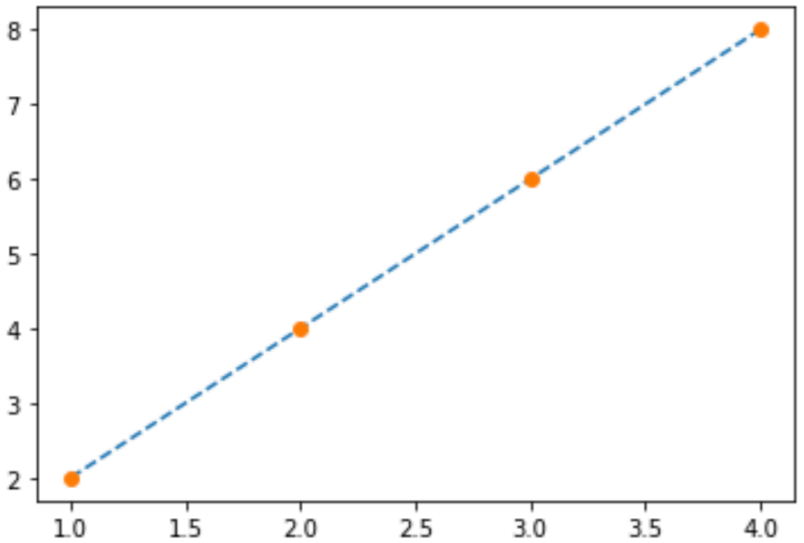
NumPy functions exist in the current scope, no barrier to access!

MatPlotLib - library for plotting and visualisation

- Again conventional to import using an alias, now ‘plt’
- Naturally makes plots nice
- So many examples available via their website https://matplotlib.org or google…
- Best to just learn as you go from existing examples

```
import numpy as np
a = np.array([1,2,3,4])
b = np.array([2,4,6,8])
```

```
import matplotlib.pyplot as plt
plt.plot(a,b, '--')
plt.plot(a,b, 'o')
```



SciPy = Scientific Python

SciPy.org

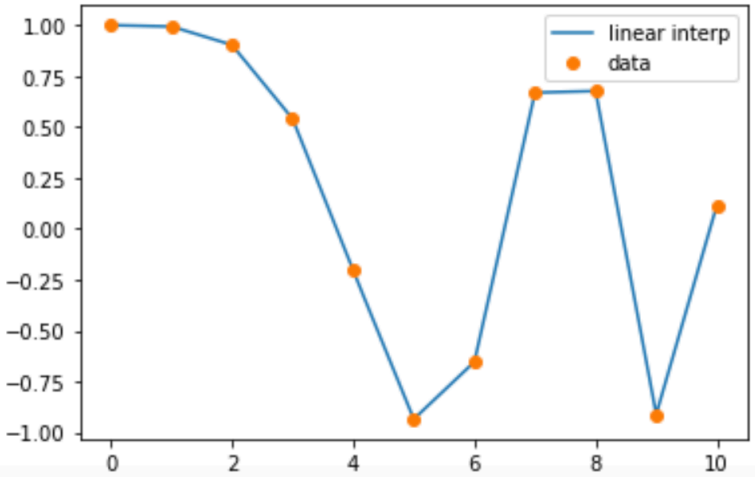
- Advanced extensions to NumPy
- Library of numerical algorithms and domain-specific toolboxes, including signal processing, optimization, statistics, and much more
- Typical to import whole submodules from scipy and then index into them using the dot

```
from scipy import misc
import matplotlib.pyplot as plt

face = misc.face()
plt.imshow(face)
plt.show()
```

> Just remember to have fun, make mistakes, and persevere.

SciPy = Scientific Python

```
import numpy as np
x = np.linspace(0, 10, num=11)
y = np.cos(-x**2 / 9.0)
xnew = np.linspace(0, 10, num=1001)
ynew = np.interp(xnew, x, y)
import matplotlib.pyplot as plt
plt.plot(xnew, ynew, '-', label='linear interp')
plt.plot(x, y, 'o', label='data')
plt.legend(loc='best');
```



Q: Why might you need this?

Plan for today

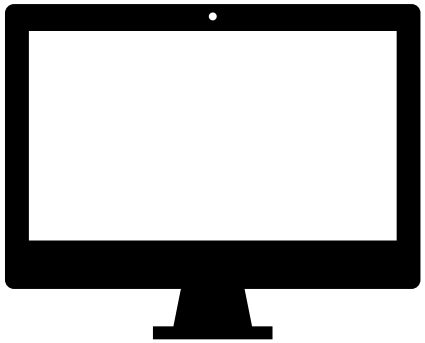
1. ~~Good grammar for good code - Types, variables, functions, loops~~
2. ~~Python libraries - numpy, scipy and matplotlib as examples~~
3. Good coding practise - version control, defensive programming, comments
4. First tutorial - space: the final frontier…

3 points for good code practise

1. Use version control, and use it frequently
2. Defensive programming
3. Commenting (but not too much)

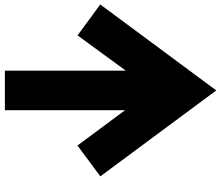
Git - a form of version control

Git - think of it like a manual version of dropbox



Your computer

Your files are stored here
UNTIL YOU ACCIDENTALLY
DELETE THEM



git push

The cloud

Your files are stored here FOREVER

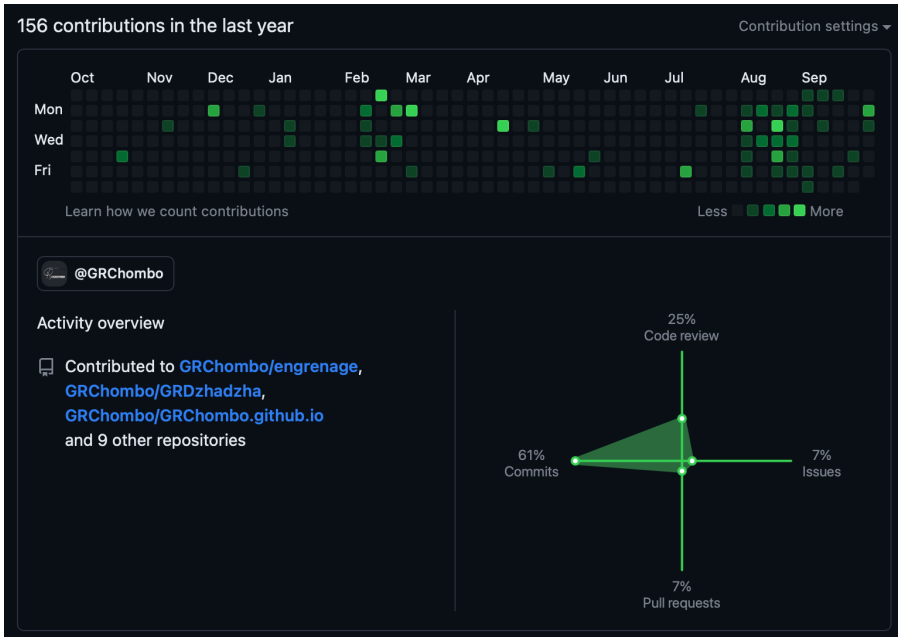
Git - a form of version control

Q: Why use version control?

Buried deep in the ice is the GitHub code vault — humanity's safeguard against devastation

ABC News Breakfast / By Nate Byrne

Posted Wed 12 Aug 2020 at 10:13pm, updated Fri 14 Aug 2020 at 5:36am



Git - a form of version control

Q: Why use version control?

So many reasons!

- Stores code so it is not lost
- Able to revert to previous versions when broken
- Able to spot bugs by checking all commits
- Collaboration made easier if several people developing code
- Easy to add documentation and code update management tools
- Can also store data and output files
- Knowledge of coding and experience proven for future employers
- Immortality of your code

Git - a form of version control

- Process for each week tutorial (see the wiki page https://github.com/KAClough/TopicsInSciComp/wiki/Updating-your-git-repository)

  1. Make a **Pull Request** from my repository into your fork to get the updates
  2. **Create a branch** called e.g. tutorial/week2, and **check it out**
  3. Make your changes, add and commit them
  4. **Push** your changes to the branch - at least daily
  5. Repeat until exercises complete, check them against solutions
  6. **Merge** them with your main branch using a **pull request**

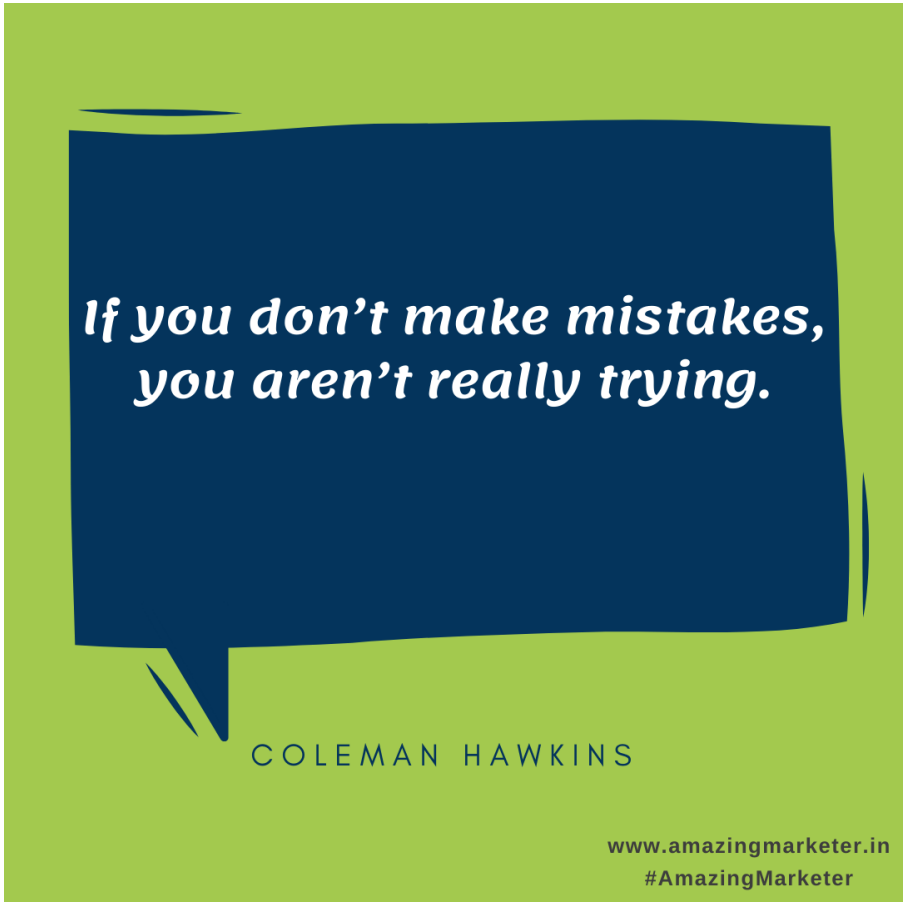
Defensive programming

You will make mistakes.

Often really silly ones!

And you won't see them.

This is ok! Just be prepared for it!



Defensive programming - assertions

Goal is to write code that checks itself. Therefore we use **assertions** (better than the conditional statements seen previously).

```
numbers = [1, -1, 2, 3]
total = 0.0
for num in numbers:
    assert num > 0.0, 'Data should only contain positive values'
    total += num
print('total is:', total)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/var/folders/p9/hydj_8nx5w3c8rkwjmgvty5r0000gp/T/ipykernel_96023/65554243.py in <module>
      2 total = 0.0
      3 for num in numbers:
----> 4     assert num > 0.0, 'Data should only contain positive values'
      5     total += num
      6 print('total is:', total)

AssertionError: Data should only contain positive values
```

- Fail early, fail often - check regularly, and in the place closest to where the object is initialised

- If you find a bug, always add an assertion or test to avoid it happening again

- Assertions also help readers of the code to check their understanding of it

Defensive programming - test driven development

1. Write a function
2. Call it interactively on two or three different inputs
3. If it produces the wrong answer, fix the function and re-run that input

1. Write a set of test functions
2. Write a function that should pass those tests
3. If it produces the wrong answer, fix the function and re-run the test functions

Which process is better?

Defensive programming - test driven development

If people write tests after writing the thing to be tested, they are subject to confirmation bias, i.e., they subconsciously write tests to show that their code is correct, rather than to find errors

Writing tests helps you to figure out what the function is actually supposed to do

Defensive programming - timing

Timing functions is a really good way to find bottlenecks.

In simple codes long execution times often mean you are doing something wrong.

```
import time

start = time.time()
print("hello")
end = time.time()
time_in_seconds = end - start
print(time_in_seconds)
```

```
hello
0.0007958412170410156
```

Commenting

```python
#Function that calculates the mass given the density and volume
def calculate_mass(density, volume) :
    mass = density * volume
    return mass

# Assigns the value of 3 to a
a = 3

# Uses Equation (3.2) in Clough et. al. 2022 Phys.Rev.Lett. 129 (2022)
g_tt = E + 0.5 * V_of_phi
```

Q: Which of these comments are useful?

Commenting

```python
#Function that calculates the mass given the density and volume
def calculate_mass(density, volume) :
    mass = density * volume
    return mass

# Assigns the value of 3 to a
a = 3

# Uses Equation (3.2) in Clough et. al. 2022 Phys.Rev.Lett. 129 (2022)
g_tt = E + 0.5 * V_of_phi
```

Only the last one - the others are redundant and
can be seen by reading the code (assuming you know Python)

In tech industry the modern paradigm is to reduce comments to an absolute minimum by writing longer and more descriptive variable/function names

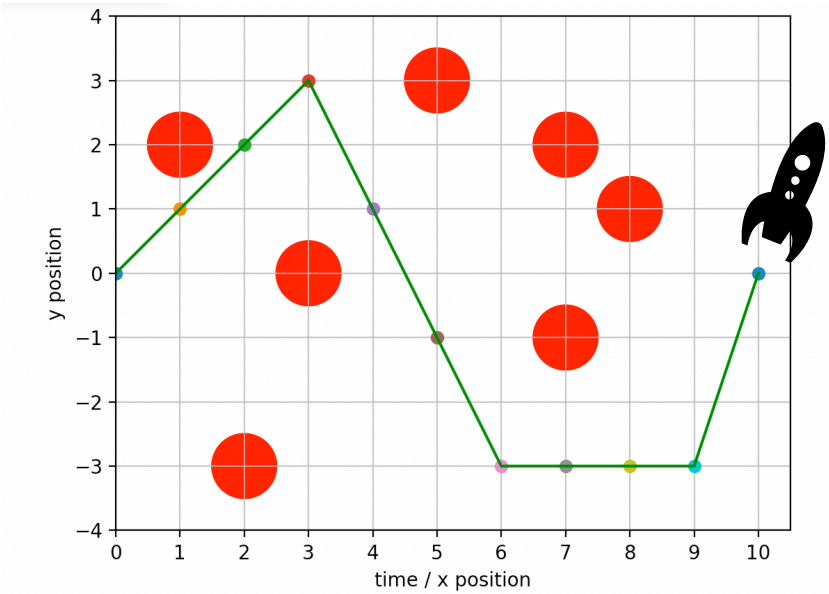
In scientific coding a few overly detailed comments doesn't hurt, especially explaining a non trivial function or Python method to a fellow student and non code expert

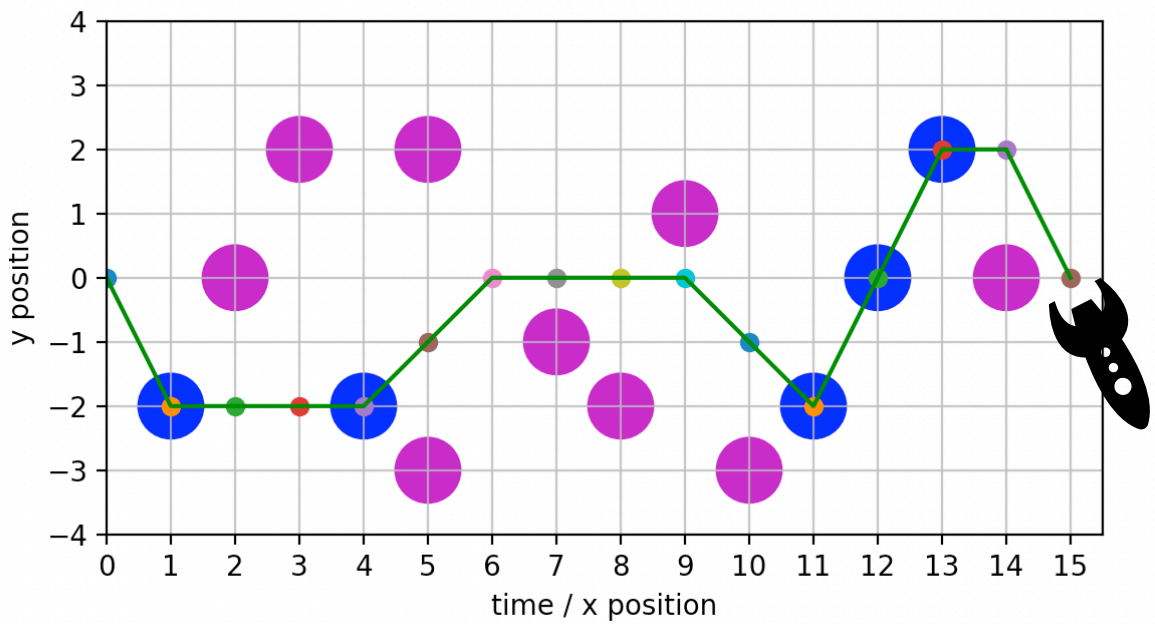
Summary

1. Good grammar creates good code!
2. Python libraries contain useful functions and objects that we will use
3. Good coding practise includes using version control, doing defensive programming, and commenting (but not too much or redundantly)

Tutorial problem next week:

Space the final frontier! See the Week 2 notebook at https://github.com/KAClough/TopicsInSciComp/tree/main/Notebooks





These are not necessarily the optimum solutions!