Introduction to Computational Physics (PHYS105)

Lecture 3: Functions and Graphs

Carl Gwilliam
(C.Gwilliam@liverpool.ac.uk)

UNIVERSITY OF LIVERPOOL

Attendance code: 198014

Lecture 2: Recap

- Last week we learnt about various python data structures
  - Both sequences (index) and mappings (key:value)

- Built-in structures
  - Lists `[]`: mutable sequences of variables of any type
  - Tuples `()`: like lists but immutable (can't change elements)
  - Strings `""` or `''`: immutable sequences of characters
  - Dicts `{}`: mapping of keys to values

- NumPy arrays
  - More efficient structures for numerical calculations on the same type of value in 1 or more dimensions
  - Can be used to read data from file and write back out

- All have similar syntax for accessing/changing the elements
  - Can use negative indices to count from the back
  - And slices to get a range of elements [lo:hi)

```
# List
mylist = [1,"two", 3.0]
print (mylist[1])
print (mylist[0:2])

# Tuple
mytuple = (1, "two", 3.0)
print(mytuple[-1])

# Dict
mydict = {"one": 1,
          "two" : "deux",
          "three" : 3.0}
print(mydict["one"])

# Numpy arrays
import numpy as np
mynp = np.array([[1,2,3], [10,20,30]])
print(mynp[1,2])
```

```
two
[1, 'two']
3.0
1
30
```

Lecture 2: Formative problem solutions

- Exercise 1: Print last item of shopping list then add two missing items to it

```
# Create list
shopping = ["cheese", "butter", "bread", "apples", "oranges"]

# Print last item (-1 easier)
print("Last item = ", shopping[-1])

# Make list of missing items
missing = ["chocolate", "wine"]

# Can concatenate lists using '+'
# (just like we saw with nums/strs)
shopping = shopping + missing
print(shopping)
```

```
Last item =  oranges
['cheese', 'butter', 'bread', 'apples', 'oranges', 'chocolate', 'wine']
```

  - For the last item could use `4` or `-1` as index
  - For lists the `+` operator performs concatenation
    - Could also have used `append()` to add items one-by-one or `extend()` to add new list
  - A few people mistakenly created a list of lists by

```
print([shopping + missing])
```

```
[['cheese', 'butter', 'bread', 'apples', 'oranges', 'chocolate', 'wine']]
```

← Two brackets = nested list

- Exercise 2: Extend the particle mass dict
  - Adding W, Z and H bosons

```
print(mydict)

# Assign value to key (in MeV)
mydict["W"] = 80e3
mydict["Z"] = 91e3
mydict["H"] = 125e3

print(mydict)
```

```
{'electron': 0.511, 'muon': 105.7, 'tau': 1777}
{'electron': 0.511, 'muon': 105.7, 'tau': 1777,
'W': 80000.0, 'Z': 91000.0, 'H': 125000.0}
```

  - Could also use `update()` with 2nd dict
  - Most did this correctly but a few created a new dict with all the items from scratch
    - Rather than adding new items to existing dict
  - Also, many put these masses in GeV
    - Rather than consistently in MeV

Lecture 2: Formative problem solutions (2)

- Exercise 3: Create array using `np.linspace`, printing the array and **number** of steps
  - Note: you can assign multiple variables on one line using commas

```
import numpy as np

# Define parameters
start, stop, step = 10.0, 22.0, 3

# Number of streps is one less than entries
nEntries= int((stop - start)/step) + 1
print("Number of steps",nEntries - 1)

# Linspace takes number of entries
myArray = np.linspace(start, stop, nEntries)
print("myArray =",myArray)
```

```
Number of steps 4
myArray = [10. 13. 16. 19. 22.]
```

  - Some people entered nEntries = 5 by hand
  - Many also printed the step size or number of entries rather than the number of steps
    - Which is the number of entries minus 1

- Exercise 4: Print out 2x2 lower right matrix
  - Remember slices indices are inclusive on lower bound but exclusive on upper one

```
[100]: # Oritinal matrix
       print("matrix = \n",matrix)
```

```
matrix =
 [[11. 12. 13. 14.]
 [21. 22. 23. 24.]
 [31. 32. 33. 34.]]
```

```
[99]: # Slice matrix to get submatrix
      submatrix = matrix[1:3, 2:4]
      print("matrix[1:3, 2:4] =\n", submatrix)
```

```
matrix[1:3, 2:4] =
 [[23. 24.]
 [33. 34.]]
```

Must use form with single square brackets [x, y] for this, not [x][y]

  - NB: As go to last index, don't need upper bound

```
[123]: submatrix = matrix[1:, 2:]
       print("matrix[1:3, 2:4] =\n", submatrix)
```

```
matrix[1:3, 2:4] =
 [[23. 24.]
 [33. 34.]]
```

  - Some people created a new array of the elements from scratch rather than slicing

Lecture 2: Formative problem solutions (3)

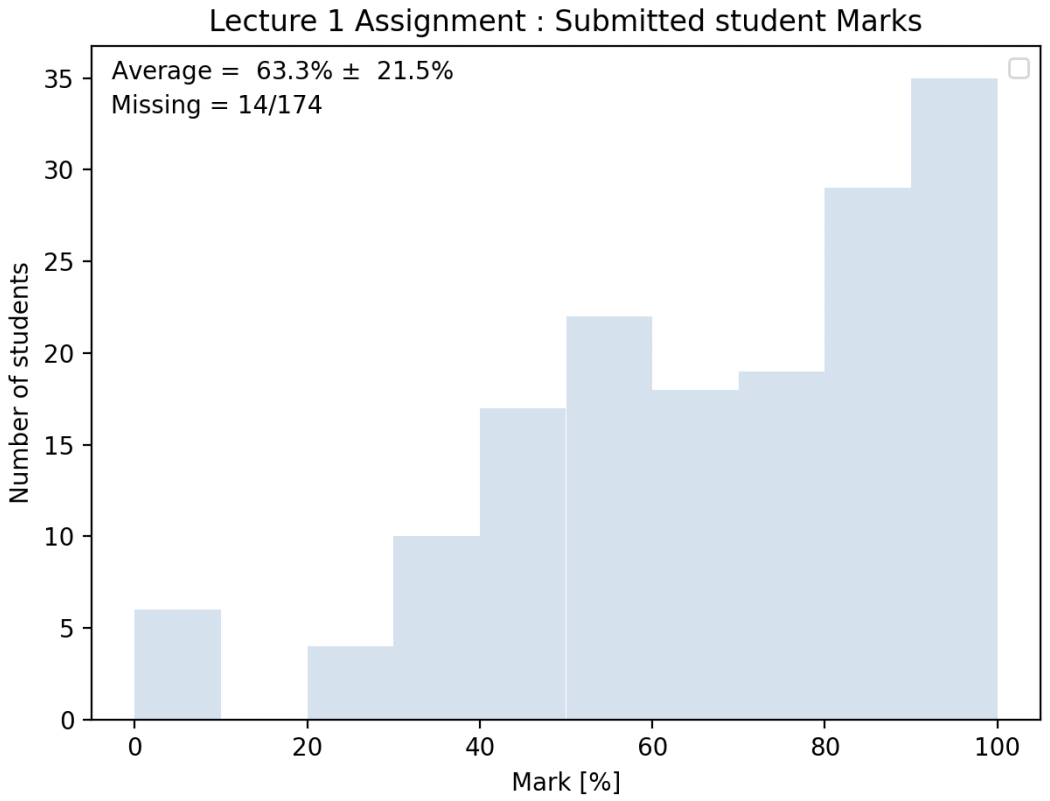
- Exercise 5: Swap the matrix rows and cols
  - Using `np.transpose()`

```
[117]: print("Submatrix =\n", submatrix)
       print("Transpose =\n",np.transpose(submatrix))
       print("Submatrix =\n", submatrix)
```

```
Submatrix =
 [[23. 24.]
 [33. 34.]]
Transpose =
 [[23. 33.]
 [24. 34.]]
Submatrix =
 [[23. 24.]
 [33. 34.]]
```

  - Note: transpose function doesn't change original but makes a transposed copy
    - Can assign result to a variable

- Lecture 2 summative: marks released in Canvas
  - Analysed using python with matplotlib ☺

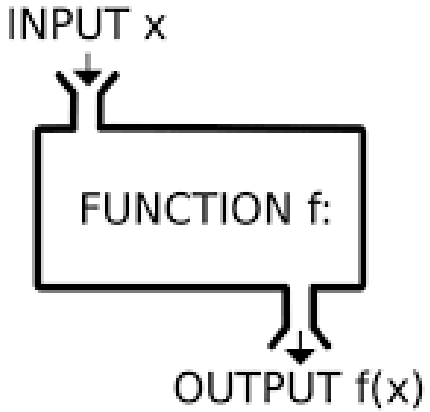


Functions

- To be able to manipulate our data we need to look at functions in more detail

- Functions are self-contained pieces of code that, when called, accomplish a specific task
  - They take in data as comma-separated arguments, process it, and return some result

- We have already used several functions
  - Directly available functions e.g `print()` and `type()`
    - https://docs.python.org/3/library/functions.html
  - Functions in built-in modules e.g. `math.sin()`
  - Functions from third-party modules e.g. `np.linspace()`



- The power of functions is that, once written, they can be used over and over again
  - Performing the same task on different inputs to get new results (without rewriting code)

- This week we will look at writing and using our own functions
  - Generally, if you write code that you will use (or think you will use) more than once, make a function!
  - You will lose marks throughout the course if you don't make appropriate use of functions

NumPy functions

- Before doing so, we will briefly revisit NumPy functions to see a particularly powerful feature

- Often, we want to apply a function to a whole array
  - E.g. perform a math operation like exponentiation

- Since the built-in `math.exp` function only takes one input variable we'd need to call it for each element

- However, the equivalent NumPy function, `np.exp`, can also take arrays as arguments:

```
[128]: np.exp(x_array)

[128]: array([3.00416602e+00, 1.27410596e+03, 5.40364937e+05])
```

  - This is significantly faster so we'll use the np version

- The same is true of pretty much all NumPy functions
  - And math operators also work on whole NumPy arrays

```
[125]: x_array = np.linspace(1.1, 13.2, 3)

[130]: import math
       print(math.exp(x_array[0]))
       print(math.exp(x_array[1]))
       print(math.exp(x_array[2]))

3.0041660239464334
1274.105955173454
540364.9372466916

[127]: math.exp(x_array)

---------------------------------------------------------------------------
TypeError
Traceback (most recent call last)
<ipython-input-127-5676efe574fc> in <module>
----> 1 math.exp(x_array)

TypeError: only size-1 arrays can be converted to Python scalars
```

```
print(x_array)
print(x_array * 2)
```

```
[ 1.1   7.15 13.2 ]
[ 2.2 14.3 26.4]
```

Writing our own functions

- We define a python function using the `def` keyword, followed by the name of the function
  - After this we have a comma-separated list of any arguments in brackets and a terminating colon (`:`)

- The body of the function does the work
  - It consists of indented lines of code
  - An optional `return` statement to give any result(s)

```
def function_name(arg1, arg2, ...):
    code
    return result1, result2, ...
```

- Let's look at an example for calculating the area of a triangle:

[133]:
```
# Function taking two arguments
def triangle_area(base, height):
    # Calculate the area
    area = 0.5*base*height
    # Return the result
    return area
```

[135]:
```
b = 4 # mm
h = 5 # mm

result = triangle_area(b, h)

print("Area (mm^2) = ", result)
```

```
Area (mm^2) =  10.0
```

- Once we have defined it, we call it as with the built-in functions
  - Passing the arguments in brackets and assigning the return value to a variable
  - Note: the names of the variables outside the function don't need to match those inside

Arguments and return values

- We can return multiple values from a function
  - The result is a tuple
  - Each element can be assigned to a separate variable

[150]:
```
def circle_properties(radius):
    area = np.pi*radius**2
    perimeter = 2*np.pi*radius
    # Return both values
    return area, perimeter
```

[153]:
```
result = circle_properties(10)
print("Area =", result[0], "Perim =", result[1])
```

```
Area = 314.1592653589793 Perim = 62.83185307179586
```

[154]:
```
area, perim = circle_properties(10)
print("Area =", area, "Perim =", perim)
```

```
Area = 314.1592653589793 Perim = 62.83185307179586
```

- Arguments can also have default values
  - By assigning to a value in definition
  - If argument left out when called, get default

[158]:
```
def circle_properties(radius, pi = np.pi):
    area = pi*radius**2
    perimeter = 2*pi*radius
    # Return both values
    return area, perimeter
```

```
area, perim = circle_properties(10)
print("Area =", area, "Perim =", perim)
```

```
Area = 314.1592653589793 Perim = 62.83185307179586
```

```
area, perim = circle_properties(10, 6.28)
print("Area =", area, "Perim =", perim)
```

```
Area = 628.0 Perim = 125.60000000000001
```

Named arguments

- So far, when calling a function we have passed in the arguments in the order they are defined
  - Python uses the order to work out which argument is which
  - In general, calling with a different order doesn't make sense

```
[141]: def happy_birthday(name, age):
           print ("Happy", age, "birthday", name, "!")
```

```
[142]: happy_birthday("Carl", 41)
Happy 41 birthday Carl !
```

```
[143]: happy_birthday(41, "Carl")
Happy Carl birthday 41 !
```

- These are called positional arguments, but you can also specify the arguments by name
  - And in this case the order doesn't matter

```
[144]: happy_birthday(age = 41, name = "Carl")
Happy 41 birthday Carl !
```

- When combined with default arguments, these allow us to call a function in different ways
  - E.g. can leave out the first argument unlike with positional args

```
[145]: def happy_birthday(name = "to you", age = 21):
           print ("Happy", age, "birthday", name, "!")
```

```
[148]: happy_birthday(age = 41)
Happy 41 birthday to you !
```

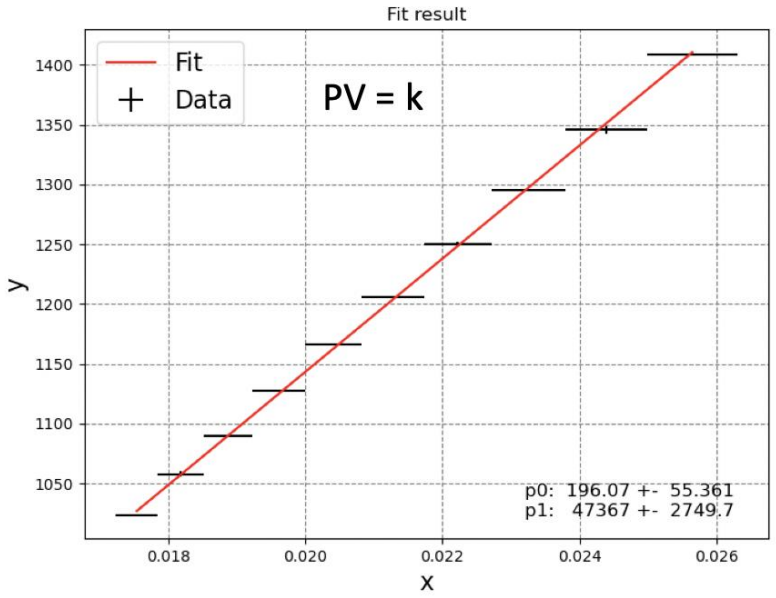
```
[149]: happy_birthday(41)
Happy 21 birthday 41 !
```

  - This is useful when have many optional arguments as we shall see

Plotting

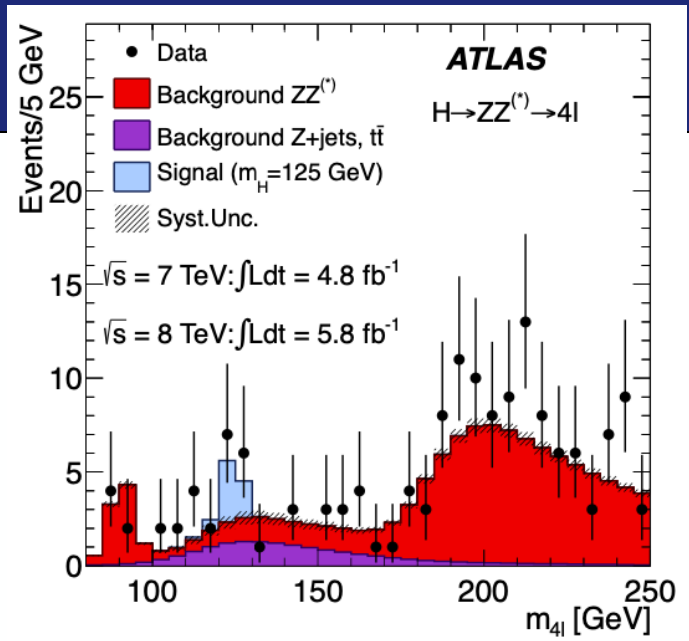
- As mentioned last time, we often want to visualise data e.g.
  - Experimental results, results of calculations
  - Comparisons of the two

- This will be the case in your PHYS106 laboratories
  - Will plot the data taken and fit them e.g.

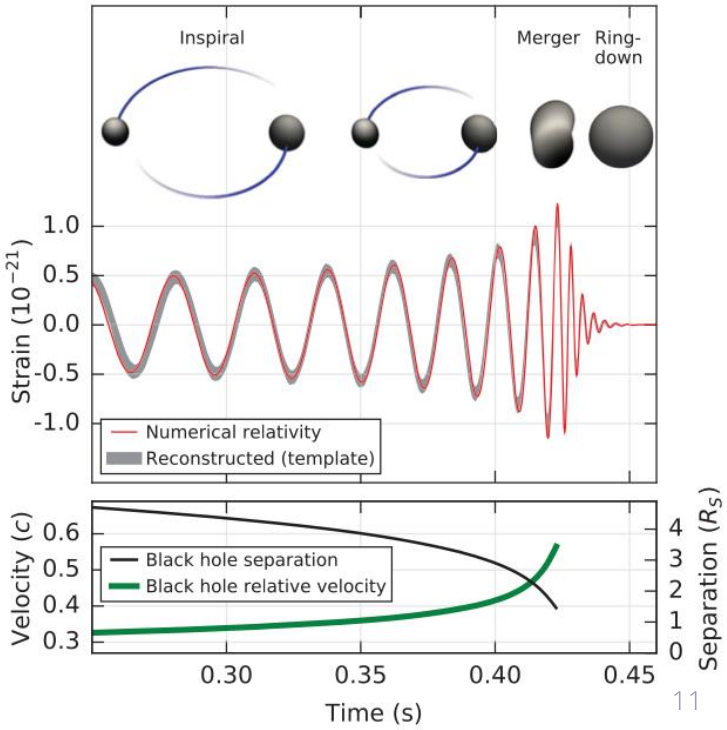


- It is also the case in almost all scientific papers
  - And in many other fields too

Higgs Discovery



Gravitational wave discovery



Matplotlib Histograms

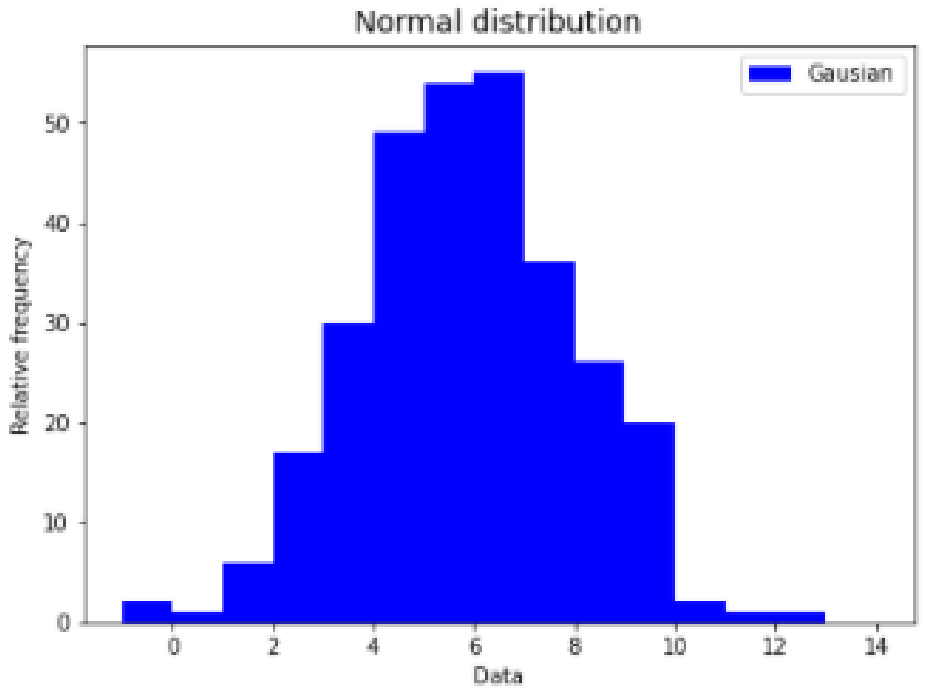
- We have seen how matplotlib can plot hists
- We create a `figure()`, giving the size tuple
  - And set the title + axis labels
- We plot the histogram with `hist()`
  - We can explicitly specify the bin edges
- It also takes many other params
  - These are named args with default values

```
hist(x, bins=None, range=None, density=False, weights=None,
     cumulative=False, bottom=None, histtype='bar', align='mid',
     orientation='vertical', rwidth=None, log=False, color=None,
     label=None, stacked=False, *, data=None, **kwargs)
```

- We can also add a `legend()`
  - Which shows the hist labels

```
import matplotlib.pyplot as plt
%matplotlib inline

# Construct a figure with a title and axis labels
plt.figure(figsize = (7, 5))
plt.title('Normal distribution', fontsize = 14)
plt.xlabel('Data')
plt.ylabel('Relative frequency')

# Set the bin edges
bin_lo, bin_hi, nbins = -1, 14, 15
bin_edges = np.linspace(bin_lo, bin_hi, nbins+1)

# Make a histogram and display it, with legend
plt.hist(gaussArr, bins = bin_edges, color = 'b', label = "Gausian")
plt.legend()
plt.show()
```

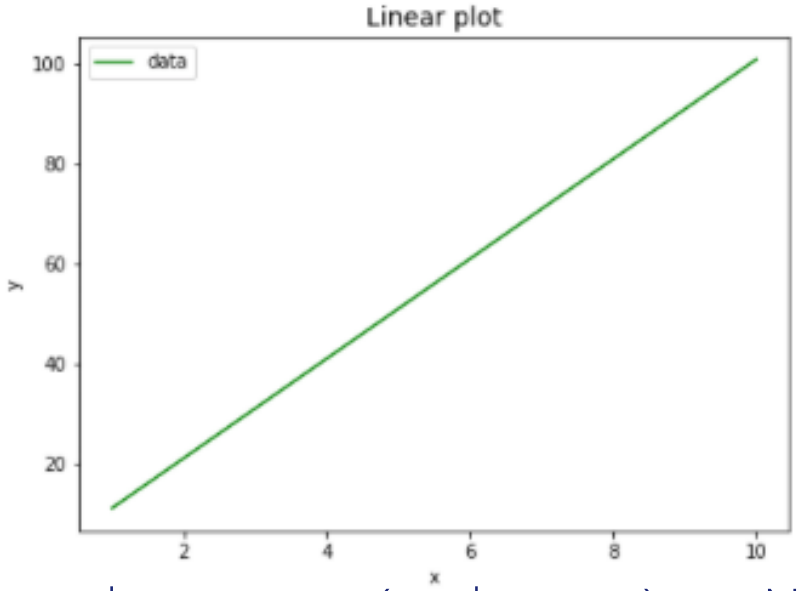
edges =
bins + 1



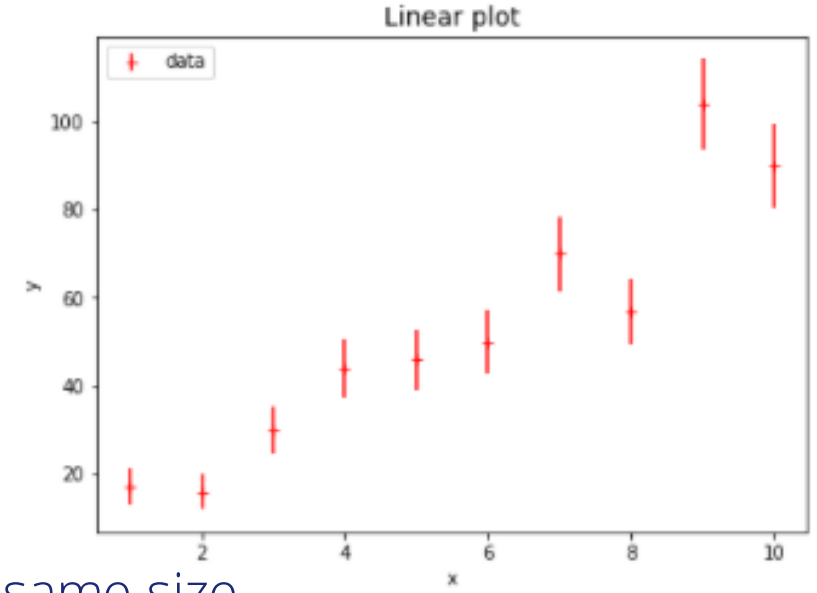
Other Matplotlib plots

- It also has many, many other types of plots, from simple to complex. Two important ones are:
  - Data as lines or markers: `plot(x, y, …)`
  - Data with errors: `errorbar(x, y, yerr =, …)`

[197]:
```
# Construct a figure with a title and axis labels
plt.figure(figsize = (7, 5))
plt.title('Linear plot', fontsize = 14)
plt.xlabel('x')
plt.ylabel('y')

# Make a line plot
plt.plot(xdata, ydata, linestyle = "-", color = 'g', label = 'data')
plt.legend(loc = 2)
plt.show()
```



[198]:
```
# Construct a figure with a title and axis labels
plt.figure(figsize = (7, 5))
plt.title('Linear plot', fontsize = 14)
plt.xlabel('x')
plt.ylabel('y')

# Make a plot with error bars
plt.errorbar(x, y, yerr = yerrors, marker = '+',
             linestyle = "", color = 'r', label = 'data')
plt.legend(loc = 2)
plt.show()
```

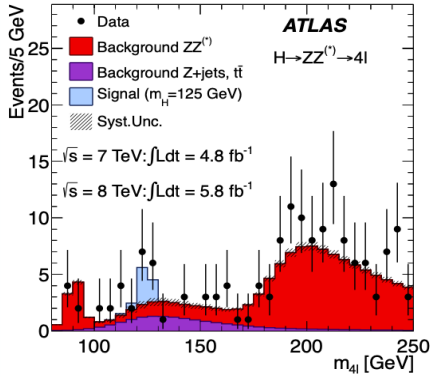


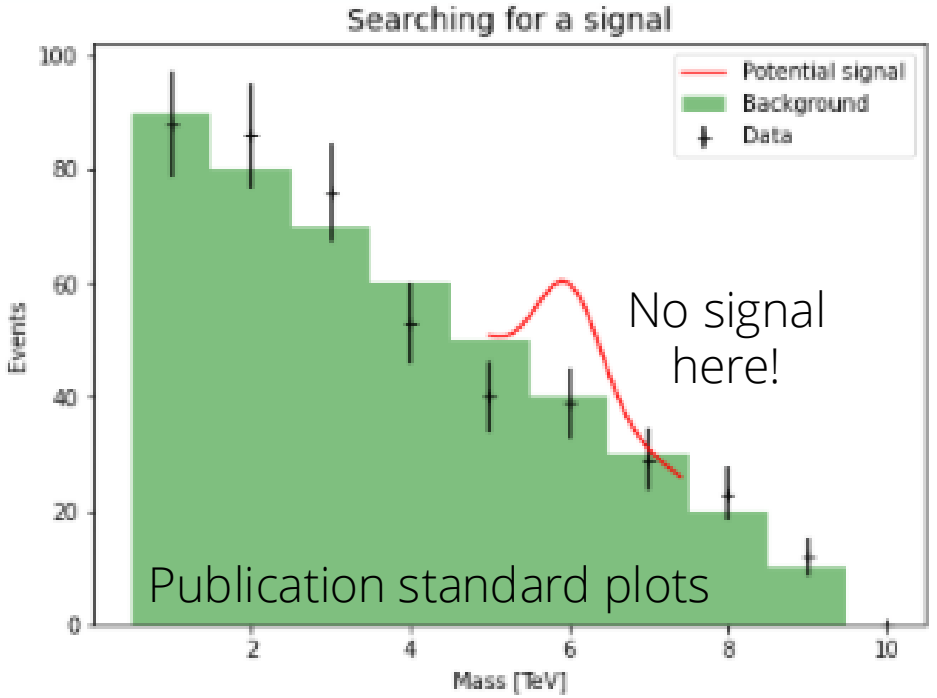
- In each case `x`, `y` (and `yerr`) are NumPy arrays of the same size
  - Many of the other options you can pass are the same for all e.g. color

Complete Plot

- You can plot multiple results on one figure
  - Simply call the appropriate function for each set of data before showing the figure
- The results can be plotted as different types
  - We can mix `hist()`, `plot()` and `errorbar()`
- E.g.: search for a new 6 TeV particle
  - Plot background simulation as hist
  - Plot potential signal as a line
  - Plot data as points with error bars
- Can also save the figure to a file
  - Using `savefig()` with the output name
  - Useful for inputting figures to lab reports etc

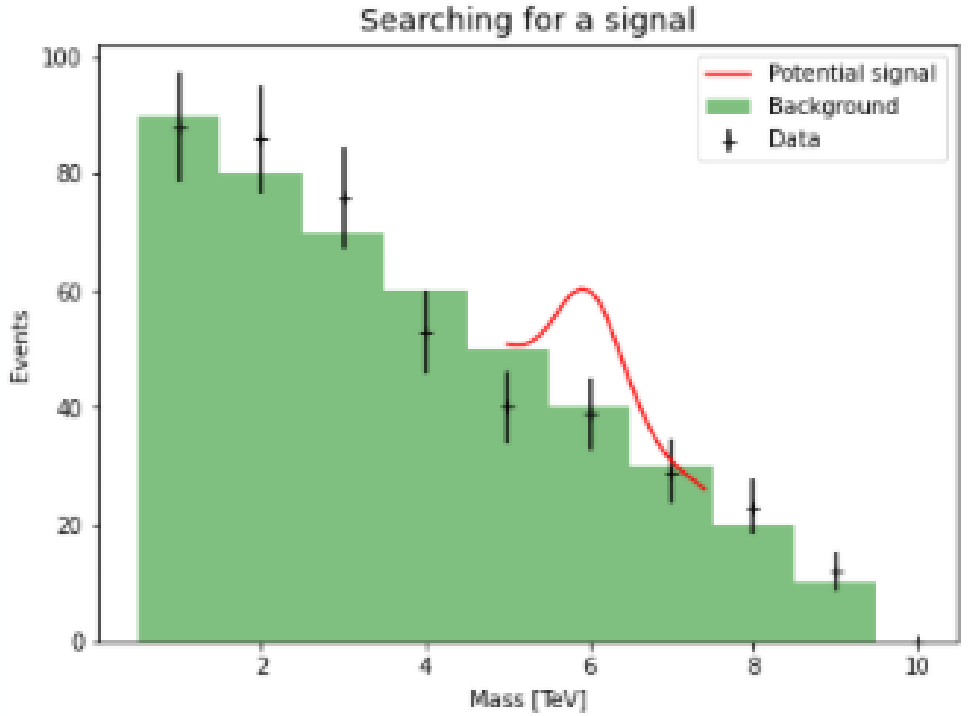
```
plt.figure(figsize = (7, 5))
plt.title('Searching for a signal', fontsize = 14)
plt.xlabel('Mass [TeV]')

plt.ylabel('Events')

plt.hist(background, bins = np.linspace(0.5, 9.5, 10), color = "g",
         alpha=0.5, label = "Background")
plt.errorbar(xdata, ydata, yerr = yerror, color = 'black',
             marker = "+", linestyle = "", label = "Data")
plt.plot(xsignal, ysignal, color = "r", label = "Potential signal")
plt.legend()
plt.savefig("MassPlot.png")
plt.show()
```





Summary

- This week we looked at how to write and use our own functions
  - Allow us to perform calculations multiple times without rewriting code
- You should get into the practice of writing functions
  - Will save you work and time in the long run!
- We also looked in more detail at plotting
  - Allowing us to visualise and compare results
- Saw different types of visualisation
  - Lines, points, histograms
  - And how to combine these
- Along the way we had more practice with NumPy



- You will exercise these and see more features in the problem class