import numpy as np
import matplotlib.pyplot as plt

Always spell colour as color, the american way

Week 1: Python as a calculator

Addition	x + y	add(x,y)
Subtraction	x - y	subtract(x,y)
Multiplication	x * y	multiply(x,y)
Division	x / y	divide(x,y)
Remainder	x % y	remainder(x,y)
Test of equality	x == y	
Tests of inequality	x > y; x < y	x >= y; x <= y
Modulus	abs()	
Power	x**y	pow(x,y)
Scientific notation	2.5*10**7	2.5e7
Square root	sqrt(x)	
Log_e	log(x)	
Log_{10}	log10(x)	
Exponential	exp(x)	e**x
Sine	sin(x)	
Cosine	cos(x)	
Tangent	tan(x)	
Inverse Sine	arcsin(x)	
Inverse Cosine	arccos(x)	
Inverse Tangent	arctan(x)	
Hyperbolic Sine	sinh(x)	
Hyperbolic Cosine	cosh(x)	
Hyperbolic Tangent	tanh(x)	
Inverse Hyperbolic Sine	arcsinh(x)	
Inverse Hyperbolic Cosine	arccosh(x)	
Inverse Hyperbolic Tangent	arctanh(x)	
Convert angle from degrees to radians	deg2rad(x)	radians(x)
Convert angle form radians to degrees	rad2deg(x)	degrees(x)

Dealing with Variables

Definition of variables is inbuilt into Python.

Function	Syntax	Alternative Syntax
Define variable as integer	x = 5	x = int(5)
Define variable as float	x = 5.0	x = float(5)
Display value of variable	print(x)	print x
Increment variable	x = x + 1	

Week 2: Arrays, Vector Algebra and Graph Plotting

Using arange() or linspace(), generate an array, a, containing the numbers 0 to 10 in increasing order	a_start = 0 a_stop = 10 a_step = 1 a= arange(a_start, a_stop, a_step) print (a)
Using arange() or linspace(), generate an array, a, containing the numbers 10 to 0 in decreasing order	b_start = 10 b_stop = 0 b_points = 11 b = linspace(b_start, b_stop, b_points) print (b)

Function	Syntax
Scalar / Dot Product	dot(x,y)
Vector / Cross Product	cross(x,y)
Length of a Vector (Norm)	norm(x)

Function	Syntax
Generating an array of 1s of length n	ones(n)
Generating an array of 0s of length n	zeros(n)
Generating an array of n points, evenly spaced between a_start and a_stop	linspace(a_start, a_stop, n)
Generating an array of n points, uniform on a log scale between 10a_start and 10a_stop	logspace(a_start, a_stop, n)
Generating an array of point between a_start and a_stop with step size step arange(a_start, a_stop, a_step)	arange(a_start, a_stop, a_step)

PLOTTING

Function	Syntax
Open a new plotting window	figure()

Graph of sin(x) against x†	plot(x, sin(x))
Clear the plotting window	clf()
Graph of array b against array a where both axes are log scales	loglog(a, b)
Graph of array b against array a with log scale on the x axis	semilogx(a, b)
Graph of array b against array a with log scale on the y axis	semilogy(a, b)

Colour	Red	Blue	Green
Code	r	b	g
Line style	Solid	Dashed	Dotted
Code	-	--	:
Marker Style	Plus	Cross	Circle
Code	+	x	o

Function	Syntax
Label x-axes	xlabel("x-axis label here", fontsize = 12)
Label y-axes	ylabel("y-axis label here", fontsize = 12)
Adding a title	title("graph title here", fontsize = 16)
Adding a grid	grid( )
Adding a legend if lines were labelled when plotted	legend(loc="best")
Adding a legend if lines were not labelled when plotted	legend(["label 1", "label 2", ….])
Change the limits of the x axis to between x_min and x_max	xlim(x_min, x_max)
Change the limits of the y axis to between y_min and y_max	ylim(y_min, y_max)
Saving a plot	savefig("name_of_file.png")

Plot and **save** a graph of sin(x), cos(x) and sin(2x) for x between 0 and 2π. Make sure that:

- The range of the x-axis is from 0 to 2π.
- Each of the lines of the plot are a different colour and style
- The plot has labels on the x and y axes

```
x=linspace(0, 2*pi, 100)
figure()
plot(x, sin(x), "b-", linewidth = 2, label = 'sin(x)' )
plot(x, cos(x), "g--", linewidth = 2, label = 'cos(x)' )
plot(x, sin(2*x), "r:", linewidth = 2, label = 'sin(2x)' )
xlabel("x", fontsize = 12)
ylabel("f(x)", fontsize = 12)
grid()
legend()
xlim([0,2*pi])
ylim([-1,1])
savefig('2trigonometry_x.png')
```

• The plot has a legend	

- Change the marker size by specifying the value

plot(x, sin(x), 'yo', markersize=2)

- Change the line width by specifying the value

plot(x, sin(x), linewidth=2)

Week 3: Arrays, Statistics and Polynomials

Function		Syntax
Length of an array		len (a)
Extract an entry from an array Remember that the entries of an array of length n are numbered from 0 to n-1.	Return entry 0	a[0]
	Return entry 4	a[4]
	Return the last entry	a[-1]
	Return the second to last entry	a[-2]
Return part of a 1D array a	return entries r to (s - 1)	a[r:s]
	return entries r to end	a[r:]
	return entries start to (s - 1)	a[:s]
	return entries s to q from end	a[s:-q]
	return entries r to (s-1) with a step of t	a[r:s:t]
Maximum and minimum	return the maximum entry	max(a)
	return the position (argument) of maximum entry	argmax(a)
	return the minimum entry	min(a)
	return the position (argument) of minimum entry	argmin(a)
Sort array in ascending order		sort(a)
Sum the entries in an array		sum(a)
Basic statistics	calculate the mean of the entries	mean(a)
	calculate the median of the entries	median(a)
	calculate the variance of the entries	var(a)
	calculate the standard deviation of the entries	std(a)

Truth testing	identify entries of a that are smaller than x (returns array of True/False)	a<x
	return all entries of a that are less than x	a[a<x]
	return all entries of b that correspond to entries of a < x (only works is a, b are same size)	b[a<x]

Histograms

Histograms are contained in the Matplotlib package.

Function	Syntax
Make a histogram of data r	hist(r)
Histogram with specific number of bins	hist(r, bins=20)
Histogram data over a given range	hist(r, range=(0.1)
Histogram with y-axis on a log scale	hist(r, log=True)
Histogram with given transparency	hist(r, alpha=0.4)

Polynomials

Polynomials are part of the NumPy package.

Function	Syntax
Generate a polynomial $p(x) = ax^2 + bx + c$	p = poly1d([a, b, c])
Evaluate a polynomial, p, at an array of points x	p(x)
Calculate the roots of a polynomial p	roots(p)
Calculate the coefficients of a polynomial from its roots (x-q)(x-r)(x-s)	poly([q, r, s])
Generate a polynomial with roots q, r, s	p = poly1d(poly([q,r,s]))
Differentiate a polynomial)	polyder(p)
Integrate a polynomial	polyint(p)
Find coefficients of a best fit polynomial of degree n to data arrays x and y	polyfit(x, y, n)

Week 4: A First Program: Straight Line Fitting

```
# Data from experiment enered as arrays
x_data = np.array([0, 0.1, 0.2, 0.4, 0.5, 0.6,0.8])
y_data = np.array([0.055, 0.074, 0.089, 0.124, 0.135, 0.181, 0.193])
```

```
#Find and plot first order line of best fit
#determine the best fit 1st degree polynomial coefficients
p_coeff = np.polyfit(x_data, y_data, 1)
#determine best fit polynomial
p = np.poly1d(p_coeff)
#set an approporiate array of x values so the line of best fir can be plotted
x = np.linspace(min(x_data), max(x_data), 100)
plt.plot(x, p(x))
plt.plot(x_data, y_data, linestyle= 'none', marker = 'x' )

# Add titles/labels and a legend to the graph
plt.title('Hookes law')
plt.xlabel('Mass [kg]')
plt.ylabel('Meters [m]')

# Format the graph to be easier to read
plt.grid()

# Uncertainties on the slope and intercept
n= len(x_data) #number of elements in array
D= np.sum(x_data**2) - 1/n * np.sum(x_data)**2
x_bar = np.sum(x_data) / n #or =mean(x_data) #average x value
p_coeff= np.polyfit (x_data, y_data, 1)
p_coeff, residuals, _, _, _ = np.polyfit(x_data, y_data, 1, full=True) # residulas is the
differente between observed and predicted values of data

#find errors in m and c
dm_squared= 1/(n-2)*residuals/D
dc_squared = 1/(n-2)*(D/n + x_bar**2)*residuals/D

# Define terms
dm = np.sqrt(dm_squared)
dc = np.sqrt(dc_squared)
slope = p_coeff[0] # slope is the gradient
intercept = p_coeff[1]

# Wrtie out results
print ("The slope of the best fit is", slope)
print ("The error of the slope is", dm)
print ("The intercept of the best fit is", intercept)
print ("The error of the intercept is", dc)
```

Week 5: Strings and Data Input/Output

Strings

Function	Syntax
Make a string variable **s** be all upper case	`s.upper()`
Make a string variable **s** be all lower case	`s.lower()`
Swap the case of each letter in a string s	`s.swapcase()`
Combine two strings, s and t	`s + t`
Split a string into words (separate at spaces)	`s.split()`
Join a set of words, introducing a separator sep between them.	`sep.join(words)`
Test if 'word' is in string S (returns Boolean)	`"word" in s`
Count instances of letter or word l in s	`s.count("l")`
Replace all instances of x in string s with y	`s.replace("x", "y")`
Convert variable x of any data type to a string	`str(x)`
Line break in string	`"\n"`
Tab in string	`"\t"`

String Formatting

Function	Syntax
print any variable inside a string	`print( f"The value is {x}" )`
print a float (with 2 decimal places)	`print( f"The value is {x:.2f}" )`
print a float using exponential format	`print( f"The value is {x:.2e}" )`
print a number, using exponential format if necessary	`print( f"The value is {x:.2g}")`

Data Input

Function	Syntax
Print "prompt" to console and wait for string input	`input("prompt")`

File Handling Methods

Function	Syntax
Open a file in writing mode and call it "f"	`f = open("FileName.txt","w")`

Open a file in reading mode and call it "f"	`f = open("FileName.txt","r")`
Write a line to a currently open file "f"	`f.write("Text to write\n")`
Read the next unread line from a currently open file "f"	`f.readline()`
Read an entire currently open file "f"	`f.read()`
Close a currently open file "f"	`f.close()`

Reading and writing numpy arrays

Function	Syntax
Read data from a file (with name "filename.txt") into an array **a**, and skip the first row	`a = loadtxt("filename.txt", skiprows=1)`
Read data from a file object f into an array **a**	`a = loadtxt(f)`
Save an array **a** into a file named "filename.txt"	`savetxt("filename.txt", a)`
Save an array **a** into a file object f	`savetxt(f, a)`
Stack two 1D arrays a and b into a 2D array c	`c = column_stack(a,b)`

Week 6: Two Dimensional Arrays

Syntax Summary

Defining 2-d Arrays

Function	Syntax
Define a 2-d array	`a = array([[1,2],[3,4]])`
Define a 2-d array of zeros	`z = zeros( (10, 10) )`
Define a 2-d array of ones	`o = ones( (10, 10) )`
Define a 2-d grid of numbers	`y, x = mgrid[0:5,0:5]`
Stack 1-d arrays to make a 2-d array	`a = column_stack([x,y])`

Defining 2-d Arrays

Function	Syntax

Return an entry of a 2-d array	`a[2,2]`
Return part of a 2-d array	`a[0:2, 0:2]`
return entries with a step greater than one	`a[0:6:2, 0:4:2]`
1d slice of a 2-d array	`a[:,0]` `a[0,:]`
return all entries as a 1-d array	`a.flatten()`

Matrices

Function	Syntax
generate a matrix	`a = matrix([[1,2],[2,3]])`
invert a matrix	`a.I`
tranpose a matrix	`a.T`
eigenvalues and eigenvectors of a matrix	`values, vectors = eig(a)`

Plotting 2-d arrays

Function	Syntax
make a contour plot of 2-d data	`contourf(x,y,z,100)`
display a 2-d array	`imshow(a)`
Show the colour bar	`colorbar()`

```
r0 = d2[0,:] #Extract the first Row and call it r0
c0 = d2[:,0] #Extract the first Column and call it c0
c3 = d2[:,3] #Extract the 4th Column and call it c4
```

1. As with 1d arrays, you can access any entry you like, The order should be familiar from matrices where M_{21} identifies the entry in the second row, first column. The difference with python is that the numbering starts at zero. For example

```
print d2[0,0] # Value in first row, first column
print(d2[2,0]) # Value in third row, first column
print(d2[0,2]) # Value in first row, third column
```

2.
3. You can also access entries starting from the end, e.g.

```
print(d2[-1,-1]) # Value in last row, last column
print(d2[-1,0]) # Value in last row, first column
```

4.
5. You can access a range of entries by specifying the range that you are interested in, for example

```
b2 = d2[0:2,0:2] #Values from the first 2 rows and first 2 columns (2x2)
print(b2)
```

6.
7. More generally, you can also specify every other entry, every 5th entry, etc. In the 1D case, you used the syntax [start:stop:step]. In the 2D case, the syntax is **[start_row:stop_row:step_row, start_column:stop_column:step_column]**. As an example,

```
c = d2[ : :2 , : :2]
```

8. print(c)

marker = 'x'

Use string formatting techniques to write out an appropriate number of decimal places in your answers
f.write(f"The intercept of the line of best fit: {round(intercept,3)}\n")

Week 7: Consolidation

Reading in data
f = open("earthquake_two.txt", "r")
text = f.read()
f.close()

extract year of each earthquake
year = np.loadtxt("earthquake_two.txt", skiprows = 15, usecols = [0])
#first column is column 0
#the data does not start until row 16, so need to kip the first 15 rows

extract magnitude of each earthquake
magnitude = np.loadtxt("earthquake_two.txt", skiprows = 15, usecols = [7])
# use column 7 as data is in the 8th column and the first column is counted as column 0

plot a histogram (in log scale) - number of earthquakes vs magnitude
plt.hist(magnitude, log=True, bins=50)
#magnitude is thing where finding the number of
#log=True as its in log scale
bins is how many intervals data is divided into, the more bins the skinnier the lines

Text.shape[0] gives the number of rows
Text.shape[1] gives the number of columns
[:,0] is first colum
[:,7] is 8th column
Columns go down, rows go across

row	row	row

column
column
column

e^x is exp(x)

deleimter=”,” means that the computer should treat every bit of data between a , as a separate piece of data. It tells it that the , is not part of the data and is actually a separator

np.meshgrid(x, y) - turns 1D arrays into matrices

cmap=’magma’ - is the colour map library for python, changes the colour of the image

// - means divide and discard the remainder

Week 8: Numerical Integration

Integration

The `trapezoid()` and `polyint()` functions are available in **NumPy**. More complex integration methods are coded in the `scipy.integrate` package. Note that this also has a version of `trapezoid()`, which appears to be the same as what's in **NumPy**.

Function	Syntax
Integration by trapezium rule	`trapezoid()`
Cumulative integration by trapezium rule	`cumulative_trapezoid()`
Integration by Simpson's rule	`simpson()`
Analytically integrate a polynomial	`polyint()`

Importing Libraries

Function	Syntax
Import a module from scipy (in this case, integrate)	`from scipy import integrate`

---

p = np.poly1d([1, 1, 1]) - to formal quadratic, this is x^2 +x + 1
q = np.polint(p) - integrates p
Integral = q(max lim) - q(min lim) - to solve integral

```
start = 1
stop = -1
step = 2
w[start:stop:step] = 4
```

write a quadratic

```
p = np.poly1d([3, -4, -7, 1, 2, 10])
print(p)
```

integrate p between -1.5 and 2.5 using left rectangular integration with 100 intervals (101 points)

```
#create 101 x values between the limits, as there is 100 intervals
x = np.linspace(-1.5,2.5,101)
#calculate the corresponding y values
y = p(x)

#calculate the length of each rectangle
dx  = (2.5 - (-1.5))/100

#approximate area as sum of 2 rectangles: dx*y[1] and dx*y[2]
rect_left = np.sum(y*dx)
print("Left rectangular integration:",rect_left)
```

integrate p between -1.5 and 2.5 using trapezium integration with 100 intervals using your code

```
#create 101 x values between the limits, as there is 100 intervals
x = np.linspace(-1.5,2.5,101)
#calculate the corresponding y values
y = p(x)

int_trapz = np.trapz(y,x)
print("Trapezium integration result:", int_trapz)
```

integrate p between -1.5 and 2.5 using Simpson's rule integration with 100 intervals using your code

```
w = np.ones(101)
w[1:-1:2] = 4
w[2:-1:2] = 2
w_simp = w/3
simp_int = (sum(w_simp*y))*dx
print("Simpson integration using intervals result:", simp_int)
```

integrate p between -1.5 and 2.5 using Simpson's rule integration with 100 intervals using SciPy simpson() function

```
from scipy import integrate

#create 101 x values between the limits, as there is 100 intervals
x = np.linspace(-1.5,2.5,101)
#calculate the corresponding y values
y = p(x)

int_simps = integrate.simps(y,x)
print("Simpson integration result using SciPy:", int_simps)
```

subplots

In the top plot, plot the polynomial p(x) given above from x= -1.5 to 2.5.

```
plt.subplot(2,1,1) # how many plots, which column, which row
x = np.linspace(-1.5,2.5,101)
plt.plot(x, p(x))
plt.xlim(-1.5,2.5)
plt.ylim(-15,40)
plt.xlabel("x")
plt.ylabel("integral of p(x)")
```

In the bottom plot, plot the integral of p(x), starting from -1.5, as a function of x. Evaluate this using both analytical integration polyint() and numerical integration using cumulative_trapezoid(). The two answers should agree.

```
plt.subplot(2,1,2)
cumtrapz = integrate.cumulative_trapezoid(p(x), x, axis = -1, initial = -1.5)
plt.plot(x,q(x), label = 'analytical integration')
plt.plot(x, cumtrapz, label = 'numerical integration')
plt.xlim(-1.5,2.5)
plt.ylim(-15,40)
plt.xlabel("x")
plt.ylabel("integral of p(x)")
plt.legend()
```

Week 9: For Loops, If Statements and Euler's Method

Function	Syntax

For loop	`for x in values:`
	`  print(x)`
	`  # indented loop code`

If statement	`if x > 3:`
	`  print("x is greater than 3")`
	`else:`
	`  print("x is less than (or equal to) 3")`

In Python, a **for loop** iterates over all of the elements of a list, string or array. The standard structure of a **for loop** is:

```
values = array([7, 6, 42])
total = 0
for val in values:
    print(f'Current value is {val}')
    total += val

print(f'The total sum is {total}')
```

[Line 1] is creating an array of items that the loop will iterate over. The loop will be run once for each item in this list. You can use lists here, or lists of words, letters or numbers.

[Line 2] sets up a variable **`total`** with value zero. This is not a necessary part of for loops, it is included here to help illustrate the process.

[Line 3] starts the loop. It takes the first item in the list **`values`** and calls it **`val`**, then runs the loop, then takes the second item and calls it **`val`**, until there are no more items.

[Line 4] is inside the loop. **Any indented** lines are in the loop. This will print out whatever item is currently called "val".

[Line 5] is also inside the loop. The code

```
total += val
```

does the same as

```
total = total + val
```

but saves you from repeating total. This syntax is quite commonly used, as it makes code shorter and easier to read, but it can take some getting used to.

[Line 6] is blank

[Line 7] is outside the loop because it is not indented. This line will only be run once the loop has gone over each item in the list **values**, it prints out the total.

```
days = ["Monday", "Tuesday", "Wednesday"]
# loop over values
for day in days:
    print(f"today is {day}" )
Result:
today is Monday
today is Tuesday
today is Wednesday
```

Example: Fibonacci Numbers

The Fibonacci numbers are a series of numbers that start off with:

0, 1, 1, 2, 3, 5, 8, ...

The rule for generating the next number is that it must be equal to the sum of the previous two numbers. Algebraically, that is

Fn = Fn-1 + Fn-2

where the subscript n means the n^{th} Fibonacci number. We must also specify ***F0 = 0, F1 = 1***.

It is straightforward to write a code to generate the first 50 Fibonacci numbers, using a for loop to calculate each subsequent number based on the previous values.

```
# set up empty array of 50 zeros
n_fib = 50
fib = zeros(n_fib, dtype=int)
# set values of first 2 numbers in the "fib" array
fib[0] = 0
fib[1] = 1
# loop to calculate the remaining values
for i in range(2,n_fib):
   fib[i] = fib[i-1] + fib[i-2]

print(fib)
```

`x % 2` gives the remainder after the integer division (when dealing with only integers such as in this case, otherwise a common type) of `x/2`. The `%` is called the modulo operator. Of course when the remainder is 0, the number is even.

To determine if a number is even or odd using an if statement:

```
n = int( input("Please enter a number\n"))

if (n %2 == 0):
    print("You entered an even number")
else:
    print("You entered an odd number")
```

breaking out of loops

```
numbers = np.arange(30)

for num in numbers:
    if num > 10:
        print(f"{num} is bigger than 10")
        break

print(num)
#it will tell you that 11 is bigger than 10 and exit the loop, so that it ends the loop
```

Euler's Method

To briefly describe Euler's Method, if we know that dy/dx = f(x,y) and have an initial values x0 and y0, we can find the value of y at **x1 = x0 + h** as

$$\mathbf{y(x_1) = y(x_0) + h\, f(x_0, y_0)}$$

Once the value is known at x_1 we can use the same procedure to calculate it at the next point. This method becomes much more accurate if smaller steps are used, and this can be easily accomplished using for loops in Python.

Step by Step Method

Before we automate the procedure using for loops, let us work out a short example showing what each step of a for loop will do. To simplify the example, we'll use freefall (no air resistance). **F = mg**, so using Euler's method with a time step of dt, we have $\mathbf{v_1 = v_0 + g\, dt}$:

```
# Manually calculate 2 time steps
t = zeros(3)
```

```
v = zeros(3)
dt = 1 # time step
g = - 9.81

t[0] = 0 # Assume initial time is zero
v[0] = 0 # Assume initial velocity is zero
t[1] = t[0] + dt
v[1] = v[0] + dt * g
t[2] = t[1] + dt
v[2] = v[1] + dt * g
```

This method is OK if you only want to calculate a couple of points, but after that becomes very tedious. The simple solution to this is to create an iterative program which can run the same line as many times as you want, using **for loops**.

Iterative Method

First, make an array of time for the loop to run over, and an array of zeros to store your velocities. Don't forget to define all variables used in you loop, in this case **g**:

```
t_max = 10 # maximum time
dt = 0.1 # time step
g = -9.81
n_steps = int(t_max/ dt)
t = zeros(n_steps)
v = zeros(n_steps)

# set initial conditions
t[0] = 0
v[0] = 0

for i in range(1,n_steps):
    t[i] = t[i-1] + dt
    v[i] = v[i-1] + dt * g
```

If you plot v against t, you will get a straight line with a slope of $9.81 ms^{-2}$. Of course, we didn't really need a for loop to solve this problem.

Integrating to find position

We have calculated an approximation of the velocity at discrete points. It is easy to add one extra line of code to work out the position as a function of time. Using Euler's method, we can approximate

```
x[i] = x[i-1] +  dt*v[i-1]
```

So, the loop at the end of the code becomes:

```
for i in range(1,n_steps):
    t[i] = t[i-1] + dt
```

```
    v[i] = v[i-1] + dt * g
    x[i] = x[i-1] + dt * v[i-1]
```

Free Fall with Air Resistance

Let's move onto a problem that is more difficult to solve with pen and paper. We will include the effect of air resistance into freefall. Air resistance is well modelled by $\mathbf{F_{air} = - kv^2}$. It acts against the direction of motion, and the force is proportional to $\mathbf{v^2}$. The value of the constant k will depend upon the area of the object and its drag coefficient.

You should now be in a position to write a code to calculate the velocity as a function of time for an object in freefall with air resistance. You can simply alter your code for the system without air resistance to add in the extra terms. Remember that Euler's method gives:

```
v[i] = v[i-1] +    dt * F(v[i-1])/ m  # F/m is acceleration
```

Week 10: Second Order Differential Equations