OS Library

OS is part of Python’s standard utility modules. It provides a portable way of using operating system dependent functionality.

It was found that when accessing data that is in a different folder but in the same directory, to resolve path issues, functions that reside within the OS library were used!

Some useful functions:

os.getcwd() returns the current working directory of the file.

os.path.join(<path1>,<path2>) joins together both paths, which is useful if we have our code in path1 and some data that we are going to use or some files in path2.

Finally, we can just change our directory to match the joined path that we got from os.path.join(). Assume that the path is saved to a variable called newPath. We then use os.chdir(path) to change our directory to that new path, so once again we can access any files within that updated path.

An example can be found below:

We are in a directory called PythonCarp/PythonCarp which visually looks like this:

A close-up of a logo

Description automatically generated

We then enter PythonCarp/PythonCarp/swc-python/

A blue folder with black text

Description automatically generated

Now our data is in the data folder, which shares the path as our code up to data so swc-python/data.

So to join the paths we do:

#This line joins path of current work directory, obtained by os.getcwd() and the

#the path where the data is stored

path = os.path.join(os.getcwd(),'swc-python/data/')

#To simplify everything we just change our current directory to that of data

os.chdir(path)

This joins the path of the directory of our code (accessed from os.getcwd() and of the data which shares the same path as our code, with the exception of swc-python/data at the end.

We then just change the directory of our code to this new path using os.chdir(path) to make sure we can access everything in either directory!

NumPy

The expression numpy.loadtxt(...) is a **function call** that asks python to run the function loadtxt which belongs to the NumPy library.

numpy.loadtxt() has two parameters:

* The name of the file we want to read.
* The delimiter that separates values on a line.

These both need to be character strings, so we put them in quotes.

Once again, numpy.loadtxt() reads the data in, it does nothing else. If we just were to say numpy.loadtxt(), the program would output the data once on the console and that would be it. So, we have to assign this data to a variable.

The code so far is:

#Importing NumPy

import numpy as np

#Import is to resolve path issues

import os

#This line joins path of current work directory, obtained by os.getcwd() and the

#the path where the data is stored

path = os.path.join(os.getcwd(),'swc-python/data/')

#To simplify everything we just change our current directory to that of data

os.chdir(path)

#Using numpy to load the file in, and store it in data

data = np.loadtxt(fname='inflammation-01.csv', delimiter=',')

Now, we can find what type data is by simply running:

print(type(data))

Which outputs <class ‘numpy.ndarray’>

This output specifically tells us that data is an N - Dimensional array, the functionality for which is provided by the NumPy library.

Now in inflammation-01.csv, the rows are the individual patients, and the columns are their daily inflammation measurements.

Note however, the type() function will return the type of the variable itself. So in this case, we know that data is an N - dimensional array. But what does this N - dimensional array actually hold? i.e what is the type of data held inside the variable?

For this, we use .dtype. In this case it would be:

print(data.dtype)

Here, we output float64.

We can also see the array’s shape by calling:

print(data.shape)

This outputs (60,40), so we know that the inflammation-01.csv file saved in the data variable has 60 rows and 40 columns.

Now, indexing rules!

If we want to get a single number from the array, we must provide an index in square brackets after the variable name, just as we do in math when referring to an element of a matrix.

Our variable, data, has two dimensions, so we will need to use two indices to refer to one specific value.

data is a 60 x 40 matrix. So if we want to access the element in the very top left, it means the element in the first row and the first column. Now, in Python, we start counting from the number 0. So, that element is in [0,0] index.

And to access that index in the variable data, we would simply do:

data[0,0]

By the same logic, the value in the middle of the 40x60 matrix would be in the 29 x 19th element (remember subtract by 1 as Python starts counting from 0). We can index out both of these elements from the data matrix by:

#Printing the first and middle value of the data matrix

print(("The first value in data is {}.").format(data[0,0]))

print(("The middle value in data is {}.").format(data[29,19]))

We can also then slice the data!

An index like [30,20] selects a single element of an array in that index. However, we can also select whole sections.

For example, we can select the first ten days (columns) of data, and the first four patients (rows) like this:

#Printing the first 4 rows (patients) and 10 columns (days)

print(data[0:4,0:10])

Which outputs:

[[0. 0. 1. 3. 1. 2. 4. 7. 8. 3.]

[0. 1. 2. 1. 2. 1. 3. 2. 2. 6.]

[0. 1. 1. 3. 3. 2. 6. 2. 5. 9.]

[0. 0. 2. 0. 4. 2. 2. 1. 6. 7.]]

The slice 0:4 means: Start at an index and 0 and go up to, but not including, index 4.

Analyzing Data

NumPy has several useful functions that take an array as an input, and performs operations on its values.

For example, if we want to find the average inflammation for all patients on all days, we can ask NumPy to compute data’s mean value:

#Calculating the average inflammation for all patients on all days, i.e the mean of data

print(np.mean(data))

Which outputs 6.14875

Now, mean is a function that takes an array as an argument.

FUN NOTE: Not all functions have an input

For example, checking the current time doesn’t require any input:

import time

print(time.ctime())

Outputs: Fri Sep 1 00:17:24 2023

HOWEVER, even for functions that don’t take in any arguments, we still need parentheses, (), to tell Python to go and do something for us.

Let’s use three other NumPy functions to get some descriptive values about the dataset. We’ll also use multiple assignment, a convenient Python feature that will enable us to do this all in one line:

#Using multiple assignment and calculating the maximum and minimum inflammation, and the std of inflammation

maxval,minval,stdval = np.amax(data),np.amin(data),np.std(data)

#Formatting the output

print(("For data, the maximum value is {}, minimum is {}, and std is {}.").format(maxval,minval,stdval))

The output is:

For data, the maximum value is 20.0, minimum is 0.0, and std is 4.613833197118566.

When analysing data though, we often want to look at variations in statistical values, such as the maximum inflammation per patient or the average inflammation per day. One way to do this is to create a new temporary array of the data we want, then ask it do the calculation.

For example, if we want to calculate the maximum inflammation for patient 0, we need to index out all the data for patient 0, which is essentially everything in row 0. This tells us the inflammation per day of patient 0.

patient\_0 = data[0,:] #Indexing everything for the first patient

#Outputting the maximum inflammation of patient 0

print(("The maximum inflammation of patient 0 is {}.").format(np.amax(patient\_0)))

Which outputs:

The maximum inflammation of patient 0 is 18.0.

We actually don’t even need to store the row in a variable of its own! We could just do the following for patient 2 for example:

#Outputting the maximum inflammation of patient 2

print(("The maximum inflammation of patient 2 is {}.").format(np.amax(data[2,:])))

The output of this code is:

The maximum inflammation of patient 2 is 19.0.

Finally, what if we need the maximum inflammation for each patient over all days, or the average for each day?

The diagrams below represent the context of this:

A graph of a patient and a patient

Description automatically generated

This diagram shows that we would want to perform the operations across an axis! In this case, axis = 1 means to perform the operations across rows, and axis = 0 means to perform the operations along the columns.

To support this functionality, most array functions allows us to specify the axis we want to work on. If we ask for the average across axis 0 (rows in our 2D example), we get:

#Calculating the average across the rows

print(np.mean(data,axis=0))

This outputs:

A number and numbers on a white background

Description automatically generated

This matrix represents the average inflammation per day for all patients.

Now, if we wanted the average inflammation per patient across all days, we would do:

print(np.mean(data,axis=1))

This gives:

A number of numbers on a white background

Description automatically generated

Stacking Arrays

Arrays can be concatenated and stacked on top of one another, using NumPy’s **vstack** and **hstack** functions for vertical and horizontal stacking respectively.

An example can be seen here:

#Stacking Arrays now

#Defining 3 x 3 array

A = np.array([[1,2,3],[4,5,6],[7,8,9]])

print(("\nInitial matrix A is \n{}.\n").format(A))

#Defining an array B that horizontally stacks the array A, so the new array would be

#[A A]

B = np.hstack([A,A])

print(("B, horizontally stacked A is \n{}.\n").format(B))

#Defining an array C that vertically stacks the array A so C' = [A A]

C = np.vstack([A,A])

print(("C, vertically stacked A is \n{}.\n").format(C))

The output is:

A screenshot of a computer program

Description automatically generated

Summary:

A screenshot of a computer program

Description automatically generated

VISUALIZING DATA TUTORIAL

We are now operating with the exact same data that we were working with in the previous tutorial. So far, the code is:

#Importing NumPy

import numpy as np

#Import is to resolve path issues

import os

#This line joins path of current work directory, obtained by os.getcwd() and the

#the path where the data is stored

path = os.path.join(os.getcwd(),'swc-python/data/')

#To simplify everything we just change our current directory to that of data

os.chdir(path)

#Using numpy to load the file in, and store it in data

data = np.loadtxt(fname='inflammation-01.csv', delimiter=',')

Now, let’s start using matplotlib!!! For this, we just import the library as;

#Import matplotlib pyplot for plotting

import matplotlib.pyplot

We can now plot a heat map by doing;

#Using matplotlb to plot a heat map

image = matplotlib.pyplot.imshow(data)

matplotlib.pyplot.show()

Result:

A chart of a blue and green gradient

Description automatically generated with medium confidence

In this figure;

* Each row in the heat map corresponds to a patient in the clinical trial dataset. Remember, in the data set there are 60 rows, meaning 60 patients.
* Each column in the heat map corresponds to a day in the dataset (40 days) so 40 columns.
* Blue pixels in the heat map represent low values.
* Yellow pixels in the heat map represent high values.

We can see from the figure that the general number of inflammation flare - ups for the patients rises and falls over a 40 day period.

Other conclusions we can make from this heat map are:

* The patients take their medication once their inflammation flare ups begin.
* It takes around 3 weeks for the medication to take effect and begin reducing flare ups (pixels start shifting from yellow to blue at around day 21).
* Flare ups appear to drop to zero by the end of the clinical trial.

We can then simply plot the average inflammation per day over time, as:

#Now plotting the average inflammation per day over time

#This would be taking the average along columns thus axis = 0

averageInflammation = np.mean(data,axis=0)

averagePlot = matplotlib.pyplot.plot(averageInflammation)

matplotlib.pyplot.show()

The result of this code would be:

A graph with a line

Description automatically generated

Remember there are 40 days, so on the x - axis it is days. The y - axis is then the average inflammation per day. This makes sense! Remember this is just the average across columns so its just mean(data,axis=0). 0 is along column, 1 is along rows.

So what did the code to?

* We put the average inflammation per day across all patients in the variable ‘averageInflammation’.
* We then ask matplotlib,pyplot to create and display a line graph of those values.

The result is a reasonably linear rise and fall, and in line with the observations made previously. HOWEVER, a good data scientist doesn’t just consider the average of a dataset, so let’s have a look at two other statistics:

Firstly, the maximum value of inflammations per day (along columns so axis=00, which we use amax for, can be found as:

#Plotting the maximum inflammation per day against the number of days

#Again, remember that we are taking average per day therefore along columns thus axis = 0

maxInflammation = np.amax(data,axis=0)

maxPlot = matplotlib.pyplot.plot(maxInflammation)

matplotlib.pyplot.show()

A graph with a triangle

Description automatically generated

And then the minimum:

A graph with a line

Description automatically generated

Grouping Plots

You can group similar plots in a single figure using **subplots**.

The script below uses a number of new commands:

1. The function *matplotlib.pyplot.figure()* creates a space into which we will place all of our plots.
2. Parameter *figsize* tells Python how big to make this space.
3. Each subplot is placed into the figure using its *add\_suplot* method. This method takes 3 parameters:
   1. First denotes has many total ROWS of subplots there are.
   2. Second refers to how many COLUMNS of subplots there are.
   3. Final parameter denotes which subplot your variable is referencing (left-right, top-bottom).

Each subplot is stored in a different variables (axes1,axes2,axes3). Once a subplot is created;

* The axes can be titled using the set-xlabel() command & the set-ylabel() too.

The code below creates these 3 plots side by side:

A graph with numbers and a triangle

Description automatically generated

#Creating a space where we are going to put all 3 plots side by side

#The space will be 10 x 3.0

fig = matplotlib.pyplot.figure(figsize=(10.0,3.0))

#We are now going to create 3 subplot into the figure

#The figures will be horizontally stacked, so 1 row, 3 columns:

sub1 = fig.add\_subplot(1,3,1) #First subplot so '1' in third place

sub2 = fig.add\_subplot(1,3,2) #Second subplot so '2' in third place

sub3 = fig.add\_subplot(1,3,3)

#Filling in and labelling each subplot

#Subplot 1

sub1.set\_ylabel('Average Daily Inflammation')

sub1.set\_xlabel('Day')

sub1.plot(np.mean(data,axis=0))

#Subplot 2

sub2.set\_ylabel('Maximum Daily Inflammation ')

sub2.set\_xlabel('Day')

sub2.plot(np.amax(data,axis=0))

#Subplot 3

sub3.set\_ylabel('Minimum Daily Inflammation ')

sub3.set\_xlabel('Day')

sub3.plot(np.amin(data,axis=0))

#Configuring the total figure and subplot

fig.tight\_layout()

#Saving the figure as a picture

matplotlib.pyplot.savefig('Inflammation.png')

matplotlib.pyplot.show()

Now similarly, matplotlib normally sets x and y axes limits to the minimum and maximum of our data, depending on the data range.

If we want to change the x and y axes limits, we just use set\_ylim(min,max) or set\_xlim(min,max).

Just as an extension to this program, we will redo everything, but now stack 2 plots on top of each other, and include the plot of the standard deviation of the data:

A graph of different levels of numbers

Description automatically generated with medium confidence

With the code for this being:

#Creating figure and adding subplots

fig2 = matplotlib.pyplot.figure(figsize=(10.0,6.0))

sub1 = fig2.add\_subplot(2,2,1)

sub2 = fig2.add\_subplot(2,2,2)

sub3 = fig2.add\_subplot(2,2,3)

sub4 = fig2.add\_subplot(2,2,4)

#Adding details and plots on the subplot. Now sub1

sub1.set\_ylabel('Average Daily Inflammation')

sub1.plot(np.mean(data,axis=0))

sub1.set\_xlabel('Day')

sub1.set\_ylim(0,15)

#Adding details to sub2

sub2.set\_ylabel('Maximum Daily Inflammation ')

sub2.set\_xlabel('Day')

sub2.plot(np.amax(data,axis=0))

sub2.set\_ylim(0,25)

#Adding details to sub3

sub3.set\_ylabel('Minimum Daily Inflammation')

sub3.plot(np.amin(data,axis=0))

sub3.set\_xlabel('Day')

sub3.set\_ylim(0,5)

#Adding details to sub4

sub4.set\_ylabel('Standard Deviation of Inflammation')

sub4.plot(np.std(data,axis=0))

sub4.set\_xlabel('Day')

#Tightening Up

fig2.tight\_layout()

#Saving the figure as a picture

matplotlib.pyplot.savefig('Inflammation2.png')

matplotlib.pyplot.show()

Python Lists

Unlike NumPy arrays, lists are built into the language so we do not have to load a library to use them. We create a list by putting values inside square brackets and separating the values with commas. An example is:

odds = [1,3,5,7]

print(‘odds are:’,odds)

Indexing is the same for lists.

Data which can be modified in place is called: mutable. Date which CANNOT be modified in place is immutable.

Strings and numbers are immutable. Note, this does not mean that immutable variables are constant can never change. All it actually means is that when you want to change the value of an immutable variable, you can only replace the old value with a completely new value. You cannot change some part in place, it doesn’t work.

You can also nest lists. For example:

veg = [ [‘lettuce’,’lettuce’,’peppers’,’zucchini’],[‘lettuce’,’lettuce’,’peppers’,’zucchini’]]

So for example, veg[0] = [‘lettuce’,’lettuce’,’peppers’,’zucchini’]

Then finally, veg[0][1] = ‘lettuce’, and veg[0][2] = ‘peppers’

These are nested lists.

Heterogeneous lists are lists in python that can contain elements of different types. Simply, not homo, so not the same data type.

You can also just append elements to a list. For example:

oldList = [1,2,3]

newList = oldList.append(3)

newList = [1,2,3,3]

Just like we append, we can also remove elements from a list quite easily. This involves using the pop() function:

oldList = [1,2,3];

newList = oldList.pop(1)

newList = [1,3]

We essentially just popped the second element in the list. Very easily removed it!

We can also reverse a list using .reverse()

Non-continuous Slices

So far, we’ve seen that we can slice continuous blocks of consecutive elements in a list. However, we can take a subset of entries that aren’t next to each other in the sequence either!

We achieve this by providing a third argument to the range within the brackets, called the *step size*.

The example below shows how you can take for example every third entry in a list:

#A quick example of indexing with a step increment for example

primes = [2,3,5,7,11,13,17,19,23,29,31,37]

subset = primes[0:12:3] #Indexing elements with a step size of 3

print(("Indexed elements are {}").format(subset))

Output being:

Indexed elements are [2, 7, 17, 29]

See! So we take every third entry, so because our start index was 0, we start at the beginning with 2, skip, skip, TAKE, skip, skip, TAKE

Some other notes:

* You can just index from the beginning of a sequence by just omitting the first index in the range. So just for example data[:2], which will index from the beginning of the sequence to the 2nd element.

The end of the sequence can also simply be characterised by len(sequence\_name).

Python Loops Syntax

Going quickly.

For loop:

for \_\_ in \_\_ :

body

Very simply. More general:

**for** variable **in** collection:

#do things here

That’s it.

We can go over an example here when computing the value of a polynomial. With a quick note to another function.

The built in function **enumerate** takes a sequence (a list for example), and generates a new sequence of the same length, Each element of the new sequence is a pair composed of the index (0,1,2,...) and the value from the original sequence:

**for** idex,val **in** enumerate(a\_list):

#Do something

Suppose you have encoded a polynomial as a list of coefficients in the following way:

* The first element is the constant term.
* The second element is the coefficient of the linear term.
* The third is coefficient of quadratic term, etc

Code would be:

#Code without an enumeration test

x = 5

coefs = [2,4,3]

y = coefs[0]\*(x\*\*0)+coefs[1]\*(x\*\*1)+coefs[2]\*(x\*\*2)

print(y)

So basically what is happening here is that for a known x, and for known coefficients, we can calculate the value of a polynomial (the y), if we just multiply the respective coefficient by their corresponding power of x.

We know that the first coefficient in coefs, coefs[0] correspond to the coefficient of the constant term (x to the power of 0). Similarly, the second coefficient, coefs[1] corresponds to the coefficient of x to the power of one.

We can such use an enumeration to calculate y. This would be a for loop and would look like:

#Doing the same by using an enumeration

y = 0

for i,coef in enumerate(coefs):

y = y+((coef)\*(x\*\*i))

Using an enumeration basically just lets us increment through two different variables at the same time, which is quite convenient in this case, as we can simultaneously iterate through the index and the coefficient.

Analyzing Data from Multiple Files

Say we wanted to get a list of all the files in the ‘data’ directory whose names start with inflammation- and end with .csv.

The following library will help us to achieve this:

import glob

The **glob** library contains a function, **glob**, that finds files and directories whose names match a pattern. We provide those patterns as string:

1. The character \* matches zero or more characters.
2. The character ? matches any one character.

We use this to get the names of all the CSV files in the current directory:

print(glob.glob('inflammation\*.csv'))

['inflammation-05.csv', 'inflammation-11.csv', 'inflammation-10.csv', 'inflammation-04.csv', 'inflammation-12.csv', 'inflammation-06.csv', 'inflammation-07.csv', 'inflammation-03.csv', 'inflammation-02.csv', 'inflammation-01.csv', 'inflammation-09.csv', 'inflammation-08.csv']

This example shows that glob,glob’s result is a list of file and directory paths in arbitrary order. This means we can just loop over it to do something in each filename in turn.

Let’s say, we want to start by analayzing just the first 3 files in alphabetical order! We use the **sorted** built - in function to generate a new sorted list from the glob.glob output. The code is:

#Code follows the 'Analyzing Data from Multiple Files' class

import glob

import os

import numpy as np

import matplotlib.pyplot as mpl

path = os.path.join(os.getcwd(),'swc-python/data/')

os.chdir(path)

FileNames=sorted(glob.glob('inflammation\*.csv')) #Obtaining and sorting all file names

FileNames = FileNames[0:3] #Taking only the first 3 for now

for i in FileNames:

print(i)

data = np.loadtxt(fname=i,delimiter=',') #Loading data

fig = mpl.figure(figsize=(10.0,6.0))

#Adding and configuring all subplots

#We have 1 row and 3 column subplots

sub1 = fig.add\_subplot(1,3,1)

sub1.plot(np.mean(data,axis=0)) #Average

sub1.set\_xlabel("Day")

sub1.set\_ylabel("Average Inflammation per day")

sub2 = fig.add\_subplot(1,3,2)

sub2.plot(np.amax(data,axis=0)) #Maximum

sub2.set\_xlabel("Day")

sub2.set\_ylabel("Maximum Inflammation per day")

sub3 = fig.add\_subplot(1,3,3)

sub3.plot(np.amin(data,axis=0)) #Minimum

sub3.set\_xlabel("Day")

sub3.set\_ylabel("Minimum Inflammation per day")

#Configuring the figure

fig.suptitle(("For data sheet {}").format(i))

fig.tight\_layout()

#Show

mpl.show()

The result being:

A line graph of different types of information

Description automatically generated with medium confidence

A graph of different types of information

Description automatically generated with medium confidence

A graph of different types of information

Description automatically generated with medium confidence

KeyPoints:

1. Use glob.glob(pattern) to create a list of files whose names match a pattern.
2. Use \* in a pattern to match zero or more characters, and ? to match any single character.

Making Choices in Python

These are done by using Boolean logic, with this case being conditional statements like the if else with the syntax:

if (condition):

#Do something

else:

#Do something

Pretty easy. You can add another layer to this, which is just the elif statement:

if (condition):

#Do something

elif (condition):

#Do something else

else:

#D something else

Comparing in Python:

A white background with black text

Description automatically generated

You can also use **and**/**or** to combine and make more complicated or effective Boolean conditions/gates:

Some cool notes:

* You can check if a string starts with a certain start by using **startswith.** For example:

‘String’startswith(‘Str’) = True

It is case sensitive!!

* In place operators do things in place and are quite compact and easy to use. For example:

A screenshot of a computer

Description automatically generated

Using Functions

The function definition opens with the keyword **def** followed by the name of the function and a parenthesized list of parameter names.

The body of the function - the statements that are executed when it runs- is indented below the definition line.

The body concludes with a return keyword followed by the return value.

When we call a function, the values we pass to it are assigned to those variables, so that we can use them inside the function. Inside the function, we use a **return statement** to send a result back to whoever assigned for it.

Let’s code and run a function:

#Coding our function:

def fahrToCelsius(temp):

#Input: A temperature value in Fahrenheit

#Output: The equivalent temperature in Celsius

return ((temp-32)\*(5/9))

#Calling it

print(fahrToCelsius(32))

Simply outputs 0.0!!

Be careful with scope. A variable that is defined within a function is local to that function. It’s scope is restricted to the function. Anything (user/other function/operation) that is not done INSIDE that function CANNOT use that variable.

Be very careful with scope. If you define something OUTSIDE of the function, unless it is in another function,you will be able to use it in your function as that variable would have global scope. The reverse is NOT TRUE.

Be very careful!!

Default Parameters

Some functions that have multiple arguments, will still be able to run and work fine without throwing an error, if one has not been entered.

Let’s look at the np.loadtxt() function.

Previously, we have used it as:

data = np.loadtxt(fname=i,delimiter=',') #Loading data

It seems like to make this function work, we needed to insert the name of the file from which we are importing the data (fname=i) here for example, as well as the delimiter of (delimeter=‘,’).

In fact however, we can pass the filename to loadtxt without the fname. However surprisingly, we need to completely write out ‘delimiter’..

This means, that this would work:

data = np.loadtxt(i,delimiter=',') #Loading data

But this would not at all:

data = np.loadtxt(i, ',') #Loading data

You have to say delimiter=

We can see what arguments we can ignore and so on, by calling the function ‘help()’ which, as the name suggests, provides a lot of help & information on the function that is supplied as an argument. Here:

#Requesting help on the load.txt function:

help(np.loadtxt)

Output:

Help on function loadtxt in module numpy:

loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0, encoding='bytes', max\_rows=None, \*, like=None)

Load data from a text file.

There’s a lot of information here, but the most important part is the first couple of lines:

loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0, encoding='bytes', max\_rows=None, \*, like=None)

This tells us that loadtxt has one parameter called fname that DOESN’T have a default value, while the eight others do.

Now, to **define a default argument**, do the following in the function declaration/definition:

def rescale(array,lowValue = 0,highValue = 1):

#Body that does something

In this function, array does NOT have a default value, but lowValue and highValue do. All we need to do is literally assign a default value for the arguments we are interested in setting a default to!

Errors and Exceptions

Often, whenever we encounter an error in our code, we use the **traceback** to understand what happened, and how best to approach the issue!

#This function intentionally has something wrong with it:

def favIceCream():

iceCream = ['chocolate','vanilla','strawberry']

print(iceCream[3])

favIceCream()

This throws an error. if we look directly at the error, we can understand exactly what happened:

File "/Users/emileelazar/Desktop/PythonCarp/PythonCarp/swc-python/AnalyzingMultipleFiles.py", line 58, in favIceCream

print(iceCream[3])

IndexError: list index out of range

This traceback has 3 levels:

1. The first shows the file in which the error occurred, then the line then the FUNCTION within which it occurred.
2. The second shows what actually triggered the error, in this case print(iceCream[3])
3. The third shows the actual error, which is an IndexError, and a quick line which describes ‘list index out of range’.

Syntax Error: Check the line that triggered the syntax error, you will be able to identify exactly what happened quite easily.

Indentation Error: Thrown when an indent occurred that wasn’t excepted.

Variable Name Error: This is a very common type of error, and quite simply occurs when you try to use a variable that does not exist.

Index Errors: This is what we saw above, it just happens when you try to exit an index that just doesn’t exist.

File Errors: Associated with reading and writing files. Many different types of file errors:

1. File Not Found Error: If you try to read a file that does not exist. This also happens due to path/directory issues and is one of the most annoying errors ever.
2. UnsupportedOperationError: If you try to write to a file that was read - only.

Defensive Programming

How can I make my programs more reliable?

1. Assertions

The first step toward getting the right answers from our programs is to assume that mistakes WILL happen and to guard against them.

This is called **defensive programming**, and the most common way to do it is to add assertions to our code so that it checks itself as it runs.

An assertion is simply a statement that something must be true at a certain point in a program.

When Python sees one, it evaluates the assertion’s condition. If it’s true, Python does nothing. If it’s false, Python halts the program immediately and prints the error message if one is provided.

For example, this piece of code halts as soon as the loop encounters a value that isn’t positive:

#Creating code that

numbers = [1.5,2.3,0.7,-0.001,4.4]

total = 0.0

for i in numbers:

assert i>0.0 #Data should only contain positive values

total += i

print('Total is:',total)

When we run it, look how the assertion produces its value:

Traceback (most recent call last):

File "/Users/emileelazar/Desktop/PythonCarp/PythonCarp/swc-python/AnalyzingMultipleFiles.py", line 56, in <module>

assert i>0.0 #Data should only contain positive values

AssertionError

It shows us how the assertion was violated, and you can see from this why it would be helpful to include comments for assertions.

Broadly speaking, assertions fall into 3 categories:

1. Precondition: Something that must be true at the start of a function in order for it to work correctly.
2. Postcondition: Something that the function guarantees is true when it finishes.
3. Invariant: Something that is always true at a particular point inside a piece of code.