Python Programming Part 2

Python Programming Part 2

Dr Daniel Chalk

Exercise 1 – Warm Up

The first thing we're going to do today is a warm up exercise to give you practice on what you've been taught so far.

You're going to work in small groups and have a go at "python_prog_workbook_4" in Jupyter Lab.

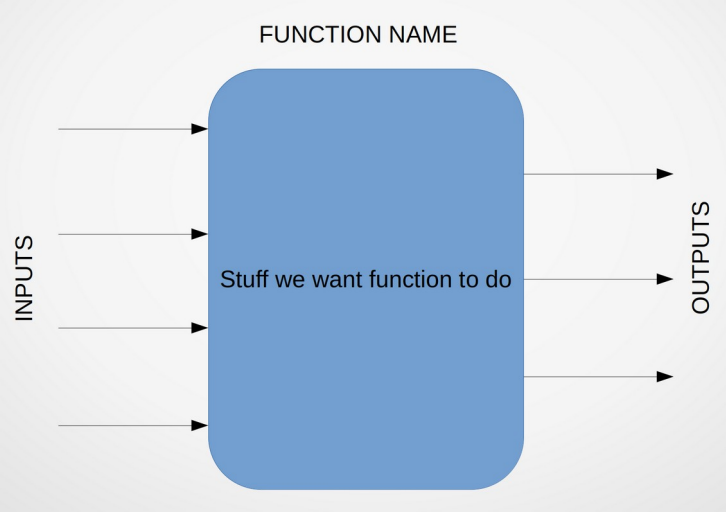
You'll have 30 minutes to complete the exercise.

Writing Functions

As well as the many in-built functions and those in external libraries, we can also write our own functions.

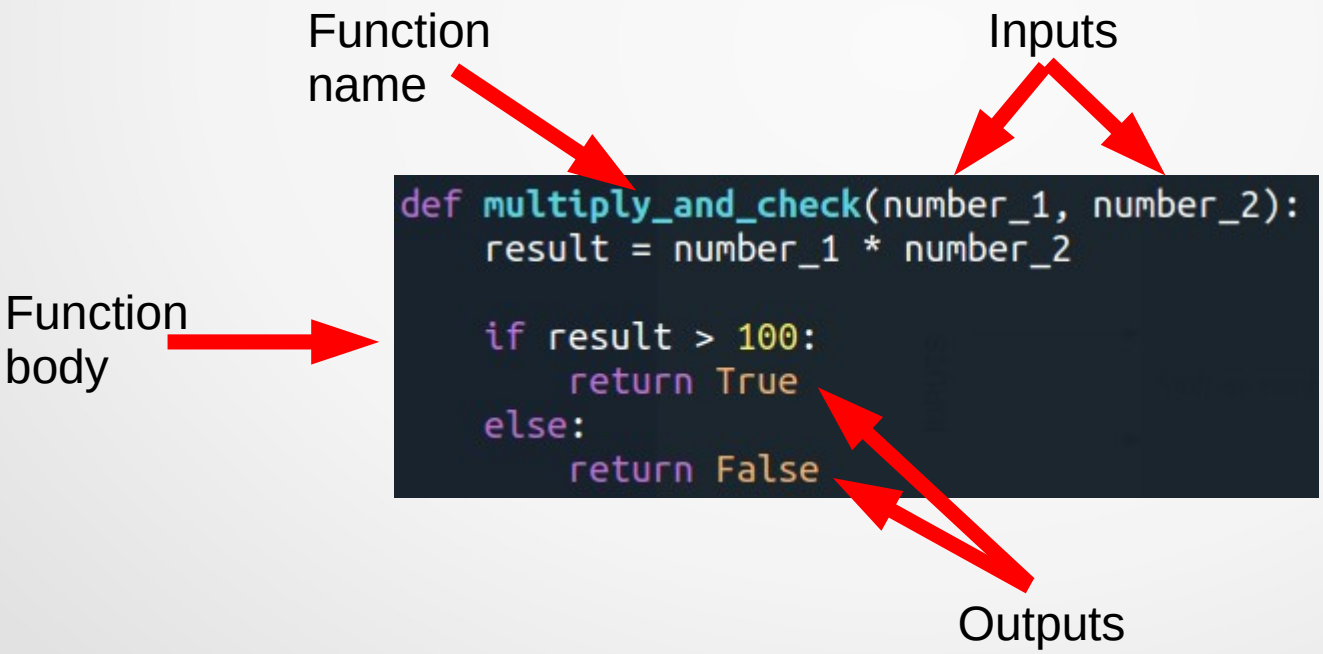
Indeed, most programs will (and should) have their own functions, so we don't have to repeat the same code multiple times.

To define a function we use the *def* command. We specify the function name, the input(s) to the function (if any), and the block of code representing the function's inner workings. We also usually need to return an output.



Function Example

Let's consider an example. Let's say we want to write a function that takes two numbers, multiplies them together, works out whether the result is higher than 100 and returns a Boolean to indicate this.



Calling a Function

Now we've written our function, we can call it at any time in our program. We need to ensure we give it the inputs it is expecting (in this case two inputs) and store the output of the function appropriately. **You must define the function before calling it!**

```
def multiply_and_check(number_1, number_2):
    result = number_1 * number_2

    if result > 100:
        return True
    else:
        return False

user_input_1 = int(input("Enter number 1 : "))
user_input_2 = int(input("Enter number 2 : "))

is_result_over_100 = multiply_and_check(user_input_1, user_input_2)

if is_result_over_100 == True:
    print ("I knew you'd think of big numbers!")
```

Note :
The names of the inputs when function is called don't have to match those used in the function definition, but the type of input MUST match

```
Enter number 1 : 480

Enter number 2 : 54
I knew you'd think of big numbers!
```

Try and Except

Sometimes, things in our program won't work as intended. For example, in our previous example, if the user inputs something other than an integer (e.g. some text or a number like 2.1), Python will return an error and terminate the program when it runs this :

```
user_input_1 = int(input("Enter number 1 : "))
user_input_2 = int(input("Enter number 2 : "))
```

But that's messy. It would be far better to catch the error and deal with it. One way to do this is to use *try* and *except.*

Try and Except

A try and except statement is the most general way to handle errors in our Python code. It instructs Python to *try* a block of code, and if it runs into any problems, to run the block of code specified in *except*.

```
import random

def multiply_and_check(number_1, number_2):
    result = number_1 * number_2

    if result > 100:
        return True
    else:
        return False

try:
    user_input_1 = int(input("Enter number 1 : "))
except:
    print("Sorry, you didn't input an integer")
    print("We'll use a random integer instead")
    user_input_1 = random.randint(1,1000)
    print("Let's use ", user_input_1)

try:
    user_input_2 = int(input("Enter number 2 : "))
except:
    print("Sorry, you didn't input an integer")
    print("We'll use a random integer instead")
    user_input_2 = random.randint(1,1000)
    print("Let's use ", user_input_2)

is_result_over_100 = multiply_and_check(user_input_1, user_input_2)

if is_result_over_100 == True:
    print ("I knew you'd think of big numbers!")
```

```
Enter number 1 : Dan
Sorry, you didn't input an integer
We'll use a random integer instead
Let's use  397

Enter number 2 : Mike
Sorry, you didn't input an integer
We'll use a random integer instead
Let's use  60
I knew you'd think of big numbers!
```

Else and Finally

We can add to a try except statement by using *else* and *finally*.

```
try:
    # try this code
except:
    # run this if an error (exception) occurs
else:
    # run this only if you didn't get any exceptions
finally:
    # run this regardless of whether or not there were exceptions
```

Exercise 2

Let’s practice defining and using functions, and using exception handling.

Spend the next 40 minutes working through the Jupyter Notebook “python_prog_workbook_5.ipynb”

Reading and Writing Files

In practice, we often want to read in data from files (as our input data) and write data to files (as our output data).

For example, we may have a file that contains the input parameters for our model, and we want to write the results of the model to file once it's run.

Python has a number of methods for doing this.

Opening and Closing Files

Whenever we read and write files, we first need to open a file (even if it doesn't yet exist), and then close it once we're finished reading or writing (this is important).

To open a file :

```
f = open("filename.txt", "r")
```

We can now refer to the file by using f.

The "r" argument in the open function means "open file in read-only mode". Here are all the different access modes for the open function...

Open File Access Modes

“r” read only, start at beginning of file
“rb” read only, binary format (usually for images, video), start at beginning of file
“r+” read and write, start at beginning of file
“rb+” read and write, binary mode, start at beginning of file

“w” write only, start at beginning of file
“wb”, write only, binary format, start at beginning of file
“w+” write and read, start at beginning of file
“wb+” write and read, binary format, start at beginning

“a” append mode (start at end of file)
“ab” append mode, binary mode
“a+” append and read
“ab+” append and read, binary mode

The “w...” and “a...” modes will create the file if it doesn’t exist.

Closing Files

Once we've finished reading from and writing to a file, we need to close it.

This is very important – if we don't do this, very odd things will happen…

```
# Open file
f = open("filename.txt", "r")

# Read what we need from our file

# Close file
f.close()

# Or we can use the with statement to only hold the file
# open whilst we do the actions in the indented block,
# after which the file will automatically close
with open("filename.txt", "r") as f:
    # Read what we need from our file
```

Reading Files Line by Line

Reading an entire file in one go isn't always useful.

Usually, we would want to read in each line of the file one by one, and store them separately, in a list for example.

We can use the *readlines()* function to do this. This function returns a list of the lines in a file.

```
f = open("filename.txt", "r")

list_of_lines = f.readlines()

f.close()
```

CSV Files

In practice, we often want our input (and output) files to be in *comma separated values* (.csv) format. Data in spreadsheets can be easily exported to .csv.

Example of .csv format :

John, 28, Plymouth,
Mary, 56, Exeter,
Bob, 37, Truro,

.csv format is useful, because our data is often separated into different fields (columns) and records (rows).

Reading CSV Files

We can import a library called *csv* that allows us to easily read and write csv files :

```
import csv # import csv library

list_of_names = []
list_of_ages = []
list_of_locations = []

with open("filename.csv", "r") as f:
    # create a csv reader object that reads the file with comma delimiters
    reader = csv.reader(f, delimiter=",")

    # for each row (line) in the csv file
    for row in reader:
        # add the first value before a comma to the list of names
        list_of_names.append(row[0])

        # add the second value to the list of ages
        list_of_ages.append(row[1])

        # add the third value to the list of locations
        list_of_locations.append(row[2])
```

```
John,28,Plymouth,
Mary,56,Exeter,
Bob,37,Truro,
```

list_of_ages	list	3	['28', '56', '37']
list_of_locations	list	3	['Plymouth', 'Exeter', 'Truro']
list_of_names	list	3	['John', 'Mary', 'Bob']

Writing CSV Files

```
# Writing csv files is easy with the csv library too!

# Let's say we have some lists we want to write to file
list_of_penchordians = ["Alison", "Andy", "Dan", "Kerry", "Martin", "Mike",
                        "Sean", "Tom"]
list_of_numbers = [1,2,3,4,5,6,7,8]
list_of_lists = [list_of_penchordians, list_of_numbers]

with open("output.csv", "w") as f:
    # Create a csv writer objects that writes a csv file with comma delimiters
    writer = csv.writer(f, delimiter=",")

    # Write a row for each of our lists within lists
    for sublist in list_of_lists:
        writer.writerow(sublist)
```

Alison	Andy	Dan	Kerry	Martin	Mike	Sean	Tom
1	2	3	4	5	6	7	8

Exercise 3

You have been given a file (input_data.csv) that contains a dataset of fabricated patient data. The data contains a number of patient records, and each record consists of the patient's first name, their gender, their age and the county in which they live.

<u>You have been asked to write a program that does the following (recommend you use Spyder for this exercise) :</u>

1) Reads in the patient data file, and stores the four different fields in four different lists.
2) Calculates the number of patient records that have been read in, and prints a message informing the user how many records have been read.
3) Calculates the average age of patients in the data and prints a message to the user informing them of the average age (to 2 d.p.)
4) Calculates how many patients live in each of the three counties in the data, and displays the results.
5) Calculates the % split between male and female patients in the data, and displays this result to the user (to 2 d.p.).
6) Writes the results of 2-5 above to a file called "results.csv", with the first row indicating the name (column header) of the result (Number of records, Average age, Number in Cornwall etc), and the second row the results themselves.

***** A couple of extra pieces of info on the next slide… *****

File Reading and Writing Exercise

EXTRA INFO :
The function *sum(x)* takes the sum of x, where x is an iterable object, such as a list.

The function *round(x, y)* rounds x to y decimal places.

You should be able to complete the exercise using the information above and what you have been taught so far.

You have 45 minutes.

Lunch

We’ll now take a break until 1.30pm. Have some lunch, stretch your legs and take a well-deserved break!

NumPy

NumPy is an important Python library for scientific computing.

It provides a number of features, and key amongst them is the ability to easily work with large multi-dimensional arrays **MUCH more efficiently** (computationally).

Because of this, NumPy is particulary important when working in Machine Learning (as you'll see later in the course).

Getting started with NumPy

NumPy is included in many scientific distributions of Python, such as the Anaconda distribution you have installed.

Therefore in order to work with it in our code, we just need to import the library. Convention dictates that we import the library with the alias “np”, so that we refer to NumPy as “np” in the rest of our code :

```
import numpy as np
```

The NumPy Multidimensional Array

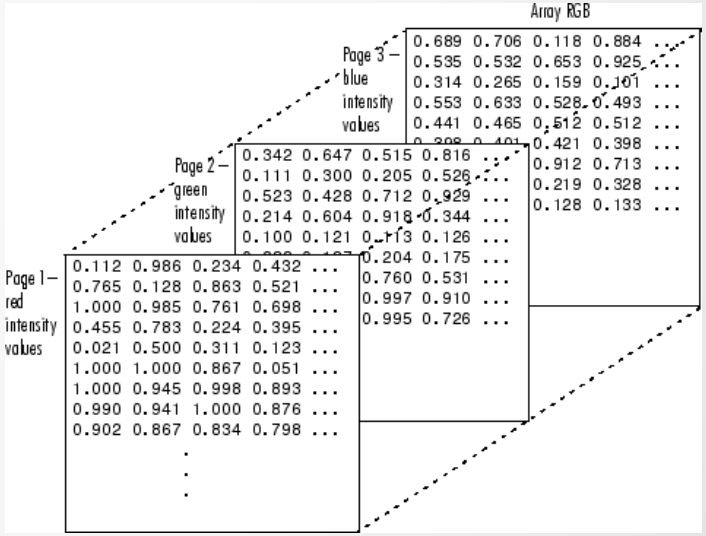
The main object in NumPy is the multidimensional array. As with most things in life, things are always better in multiple dimensions!

Before we go any further, let’s explain what we mean by a multidimensional array.

The NumPy Multidimensional Array

Here's a one dimensional (1D) array :
[1, 2, 3]

Here's a two dimensional (2D) array (an array within an array) :
[[1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]]



Here's a three dimensional (3D) array
(an array within an array within an array) :
[[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
  [[10, 20, 30], [40, 50, 60], [70, 80, 90]],
  [[100, 200, 300], [400, 500, 600], [700, 800, 900]]]

Dimensions in NumPy array are referred to as axes.

The NumPy Multidimensional Array

1D Array Practical Example :

List of ages : 32, 71, 56, 54, 28

[32, 71, 56, 54, 28]

The NumPy Multidimensional Array

2D Array Practical Example :

List of ages for each group :

Group 1 : 32, 71, 56, 54, 28
Group 2 : 17, 28, 22, 18, 56
Group 3 : 98, 88, 10, 12, 5

```
[ [32, 71, 56, 54, 28],
  [17, 28, 22, 18, 56],
  [98, 88, 10, 12, 5] ]
```

The NumPy Multidimensional Array

3D Array Practical Example :

List of ages for each group within each department :

Department 1
Group 1 : 32, 71, 56, 54, 28
Group 2 : 17, 28, 22, 18, 56
Group 3 : 98, 88, 10, 12, 5

Department 2
Group 1 : 8, 9, 15, 16, 23
Group 2 : 21, 37, 45, 42, 46
Group 3 : 51, 67, 16, 16, 21

[[[32, 71, 56, 54, 28], [17, 28, 22, 18, 56], [98, 88, 10, 12, 5]],
  [[8, 9, 15, 16, 23], [21, 37, 45, 42, 46], [51, 67, 16, 16, 21]]]

Declaring a NumPy Array

Declaring a NumPy array is easy :

```
import numpy as np

a = np.array([1,2,3,4,5])
```

a is now stored as a NumPy array. In practice, we often want to read in a series of lists to form a NumPy array. This is easy too :

```
list_1 = [1,2,3,4,5]
list_2 = [2,4,6,8,10]
list_3 = [3,6,9,12,15]

a = np.array([list_1, list_2, list_3])
print(a)
```

```
[[ 1  2  3  4  5]
 [ 2  4  6  8 10]
 [ 3  6  9 12 15]]
```

shape

We can use the *shape* function to find the shape of a NumPy array, in terms of the number of dimensions (if 2 or more), and the number of elements per dimension.

In a 1D array this give us the number of elements in the array :

```
a = np.array([1,2,3])
print (a.shape)
```

```
(3,)
```

e.g. list of three age values

In a 2D+ array this give us the number of rows and columns in multi-dimensional space

```
a = np.array([[1,2,3],[4,5,6]])
print (a.shape)
```

```
(2, 3)
```

e.g. two groups of three age values

```
a = np.array([[[1,2,3],[4,5,6]],
              [[10,20,30],[40,50,60]]])

print (a.shape)
```

```
(2, 2, 3)
```

e.g. two departments with two groups, each with three age values

ndim

We can use the *ndim* function to find out the number of dimensions / axes in our array.

```
c = np.array([[[1,2,3],[4,5,6]],
              [[10,20,30],[40,50,60]]])

print (c.ndim)
```

```
3
```

Homogeneity

NumPy requires arrays to be homegenous. This means that you have to have the same number of columns in each row (in our example before, that would be the same number of age values per group).

This isn't really valid :

```
d = np.array([[1,2,3], [1,2]])
```

but if you wrote the above, you wouldn't get an error. Instead, NumPy will store references to two separate lists within the overall array, and give a shape of (2,).

This is not recommended as you will be unable to use a number of key NumPy features if you do this.

Mathematical Operations

NumPy allows us to apply mathematical operations to arrays easily. Let's imagine we have an array a, and we want to double the value of every element in the array :

```
a = np.array([1,2,3,4,5])

a = a * 2
print (a)
```

```
[ 2  4  6  8 10]
```

Slicing

Slicing allows us to easily carve up bits of our NumPy array to deal with only certain sections.

```
b = np.array([ [1,2,3,4,5], [6,7,8,9,10] ])

# Double the value of every value in the second row only and store in a new
# array
c = b[1] * 2
print (c)

# Print out the value in the third column of the first row
print (b[0][2])
```

```
[12 14 16 18 20]
3
```

Statistics

We can also perform statistical operations on our NumPy array easily. Let's say we want to find the mean of our array :

```
a = np.array([1,2,3,4,5])

# Print the mean of array a
print (a.mean())

b = np.array([ [1,2,3,4,5], [6,7,8,9,10] ])

# Print the mean across rows (e.g. mean of element 0, mean of element 1 etc)
print (np.mean(b, axis = 0))

# Print the mean across columns (e.g. mean of each row)
print (np.mean(b, axis = 1))
```

```
3.0
[3.5 4.5 5.5 6.5 7.5]
[3. 8.]
```

Dot Product

The *dot product* of two arrays of identical length multiplies the *n*th element of array a with the *n*th element of array b, and adds all of these multiplications together to give a single answer.

Example :

a = [1, 2, 3]
b = [2, 4, 6]

$a \cdot b = (1 \times 2) + (2 \times 4) + (3 \times 6) = 2 + 8 + 18 = 28$

In NumPy we simply use the dot() function :

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

dp = np.dot(a, b)
print (dp)
```

```
28
```

vstack

The *vstack* function allows us to easily add more rows to an existing NumPy array :

```
b = np.array([ [1,2,3,4,5], [6,7,8,9,10] ])

list_to_add = [3,6,9,12,15]

b = np.vstack([b, list_to_add])

print (b)
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [ 3  6  9 12 15]]
```

hstack

The *hstack* function allows us to do the same, but for adding columns :

```
b = np.array([ [1,2,3,4,5], [6,7,8,9,10] ])

column_to_add = [ [6], [12] ]

b = np.hstack([b, column_to_add])

print(b)
```

```
[[ 1  2  3  4  5  6]
 [ 6  7  8  9 10 12]]
```

Removing Duplicate Data

The *unique* function allows us to easily remove duplicate values from an array :

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

# Overwrite b with a version of b with only unique values
b = np.unique(b)

print(b)

b = np.array([ [1,2,3], [4,5,6], [1,2,3] ])

# Remove duplicate rows (axis=0 for rows, 1 for cols, 2+ for 3D+ cols)
b = np.unique(b, axis=0)
print (b)
```

NumPy Speed Demo

Why should we bother using NumPy? Is it really that much faster?

Let’s have a look at a demo.

Exercise 4

You have been given data that contains three fields relating to patients. One represents patient ages, one the number of previous admissions they have had, and one whether they were admitted following their latest attendance to ED (1 if they were, 0 if they weren't).

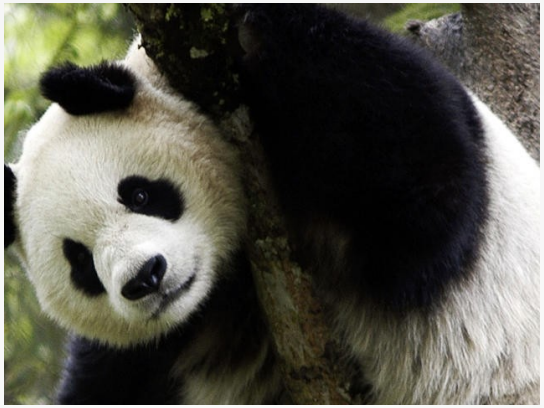
Ages = 18, 19, 22, 23, 24, 27, 29, 31, 34, 46
Previous admissions = 0, 3, 2, 0, 0, 1, 2, 0, 0, 1
Admitted this time = 1, 0, 0, 0, 0, 1, 1, 1, 0, 0

Write code that :
1. Stores the three data fields above in three separate lists
2. Combines the lists into a single 2D NumPy Array called *data*
3. Calculates and prints the mean age of patients in the NumPy Array by using slicing on the appropriate row, rounded to 1 d.p.
4. Calculates and prints the total age of those admitted this time using slicing and a dot product calculation
5. Calculates the mean number of previous admissions for those aged under 30 in the data using slicing (assume that the lists are static, and the list of ages is in ascending order; remember [a : b] notation from your lists training…), rounded to 1 d.p.

You have 30 minutes.

Pandas

Pandas are beautiful creatures native to China



Pandas is also a beautiful open source Python library that provides powerful data structures and analysis tools

The level of geekiness of the person you're talking to can be easily judged by which one of these they assume you're talking about when you say "pandas"

Pandas Overview

Pandas is very powerful for manipulating data in large arrays, and allows for indexing of data.

NumPy and Pandas are often used together, with NumPy used for mathematical functions applied to the data, and Pandas used to manipulate the data.

As with NumPy, there is a conventional alias under which Pandas should be imported :

```
import pandas as pd
```

Pandas DataFrame

One of the most useful structures in Pandas is the Pandas DataFrame. A DataFrame is like a table with different columns (which can have names) for different data fields, and different rows for each entry in the data.

Imagine a table in Excel. Only much more powerful, and with the potential to be multi-dimensional beyond two dimensions.

Patient ID	Name	Age	Location
1	Bob Rogers	46	Plymouth
2	Fred Miles	31	Truro
3	Janet Smith	27	Exeter

Creating a DataFrame

Let’s create a new empty dataframe, and set up some lists of data we want to add to the dataframe.

```
import pandas as pd

df = pd.DataFrame()

list_of_patient_IDs = [1,2,3]
list_of_names = ["Bob Rogers", "Fred Miles", "Janet Smith"]
list_of_ages = [46,31,27]
list_of_locations = ["Plymouth", "Truro", "Exeter"]
```

Creating a DataFrame

Now we can setup some columns in our DataFrame by giving the columns names and specifying the data we want in the column (in this case, the data from the lists we created) :

```
df['patient_ID'] = list_of_patient_IDs
df['name'] = list_of_names
df['age'] = list_of_ages
df['location'] = list_of_locations
```

Creating a DataFrame

Now let's print the DataFrame to see what it looks like :

```
print (df)
```

```
   patient_ID          name  age  location
0           1   Bob Rogers   46  Plymouth
1           2   Fred Miles   31     Truro
2           3  Janet Smith   27    Exeter
```

You'll notice that a fifth column has been added automatically – this is the index column. Like a database, a Pandas DataFrame needs a unique identifier for each entry. However, in many cases, a more useful unique identifier will be in our data already (the patient_ID in this case). Let's see how we can make this the index...

Specifying a DataFrame Index

To set the index for the DataFrame, we use the set_index() function. We simply state the name of the column we want to be the index, and set the inplace flag to True (to indicate we want to make a change to an existing DataFrame without creating a new one) :

```
df.set_index('patient_ID', inplace=True)

print (df)
```

```
patient_ID
1             Bob Rogers   46  Plymouth
2             Fred Miles   31     Truro
3            Janet Smith   27    Exeter
```

Retrieving specific records using loc

We can retrieve specific records from our DataFrame using the *loc* method, and specifying the index / indices that we want to retrieve.

```
# Retrieve single entry
# e.g. Print the record with index value of 2
print (df.loc[2])

# Retrieve multiple entries specified in a list
# e.g. Print the records with indices 2 and 3
print (df.loc[[2,3]])

# Retrive multiple entries specified by a slice
# e.g. Print the records with indices 1 to 3 (inclusive)
print (df.loc[1:3])
```

```
name         Fred Miles
age                  31
location          Truro
Name: 2, dtype: object
                    name  age location
patient_ID
2            Fred Miles   31    Truro
3           Janet Smith   27   Exeter
                    name  age  location
patient_ID
1            Bob Rogers   46  Plymouth
2            Fred Miles   31     Truro
3           Janet Smith   27    Exeter
```

Retrieving specific columns

Pandas allows us to easily retrieve specific columns of data (notee the index column is automatically included in the output too) :

```
print (df['name'])
```

```
patient_ID
1      Bob Rogers
2      Fred Miles
3     Janet Smith
Name: name, dtype: object
```

Adding more rows (records)

We can add additional rows (records) to a Pandas DataFrame. The most efficient way to do this is to declare a new DataFrame containing the new row(s) to be added, then use the *append* method to join the new DataFrame to the old one. We use a dictionary to establish the new value(s) associated with each column :

```
df2 = pd.DataFrame({"patient_ID":[4,5],
                    "name":["Lucy Rivers", "Dave Jones"],
                    "age":[28,61],
                    "location":["Plymouth", "Plymouth"]})

df2.set_index("patient_ID", inplace=True)

df = df.append(df2)

print (df)
```

patient_ID			
1	Bob Rogers	46	Plymouth
2	Fred Miles	31	Truro
3	Janet Smith	27	Exeter
4	Lucy Rivers	28	Plymouth
5	Dave Jones	61	Plymouth

Adding more columns (fields)

Adding columns to a DataFrame is much easier. We simply set up a list of the values we want to add in the new column, then add the list as a new column to the DataFrame :

```
list_of_previous_admission = [True, True, False, False, False]
df['previously_admitted'] = list_of_previous_admission
print (df)
```

```
patient_ID
1            Bob Rogers   46  Plymouth              True
2            Fred Miles   31     Truro              True
3           Janet Smith   27    Exeter             False
4           Lucy Rivers   28  Plymouth             False
5            Dave Jones   61  Plymouth             False
```

Exercise 5

You have been given a table containing data on Minor Injury Units :

MIU Number	Location	X-Ray Facilities?	Opening Time
463	Lostwithiel	Yes	Morning
571	St Austell	No	Afternoon
614	Truro	Yes	Afternoon
732	Looe	No	Morning
817	Camborne	Yes	Evening

Write code that :
1) Stores the four columns as separate lists (in real applications, you can use functions to read in real data from a .csv file, in case you're worried!)
2) Creates a new Pandas DataFrame that contains this data in named columns, with MIU Number as the index
3) Prints out the data for MIU Numbers 571 and 732
4) Prints out the location column of the data
5) Adds a new column that stores the mean number of patients seen per day, which are 4, 3, 3, 1, 2 respectively, then prints the dataframe.
6) Adds a new row representing the data for MIU number 901, which is a morning MIU with X-Ray facilities in Callington, that sees 5 patients per day, then prints the dataframe.

You have 30 minutes to complete the exercise.

Further Reading

For the remainder of this session, and continuing as additional work outside of today, read the following sections on pythonhealthcare.org :

23. Pandas : Basic Statistics
https://pythonhealthcare.org/2018/04/04/23-pandas-basic-statistics/

25. Reading and Writing CSV files using NumPy and Pandas
https://pythonhealthcare.org/2018/04/04/25-reading-and-writing-csv-files-using-numpy-and-pandas/

29. Sorting and ranking with Pandas
https://pythonhealthcare.org/2018/04/05/29-sorting-and-ranking-with-pandas/

30. Using masks to filter data, and perform search and replace, in NumPy and Pandas
https://pythonhealthcare.org/2018/04/07/30-using-masks-to-filter-data-and-perform-search-and-replace-in-numpy-and-pandas/

31. Summarising data by groups in Pandas using pivot_tables and groupby
https://pythonhealthcare.org/2018/04/08/31-summarising-data-by-groups-in-pandas-using-pivot_tables-and-groupby/

33. Subgrouping data in Pandas with groupby
https://pythonhealthcare.org/2018/04/10/33-subgrouping-data-in-pandas-with-groupby/

109. Saving intact Pandas DataFrames using ‘pickle’
https://pythonhealthcare.org/2018/12/21/109-saving-intact-pandas-dataframes-using-pickle/