Module 1 : Introduction to OR, Data Science and Programming

Session 1E : Python Programming Part 3

Dr Daniel Chalk

"OOPs, Dan did it again… (again)"

HSMA 4

NIHR | Applied Research Collaboration South West Peninsula

@peninsula_ARC #HSMA4

HSMA 4 Everyone

Exercise 1 – Warm Up

Let’s begin today’s session with another “warm up” exercise, to put into practice what you’ve learned so far.

Spend the next 40 minutes working through the task in the Jupyter Notebook “python_prog_workbook_6.ipynb”.

You should work as a group on this exercise, and you should discuss how you might approach this before you start coding.

Object Oriented Programming

Today, we're going to cover Object Oriented Programming (OOP) in Python, which is the preferred way to structure non-trivial programs.

Let's briefly remind ourselves of what we mean by Object Oriented Programming...

OOP Example 1

CLASS : Vehicle

Attributes
common_name : string
number_of_wheels : integer
capacity : integer

Methods
drive(speed)

The Class contains the generalised description / blueprint

Objects are *instances* of a Class that *inherit* attributes and methods from the parent Class

Inheritance

OBJECT
common_name = “Ambulance”
number_of_wheels = 4
capacity = 3

drive(speed)
fuel_refill(level)
activate_siren()
deactivate_siren()
load_patient()
unload_patient()

OBJECT
common_name = “Bicycle”
number_of_wheels = 2
capacity = 1

drive(speed)
repair_puncture()

OBJECT
common_name = “Car”
number_of_wheels = 4
capacity = 5

drive(speed)
fuel_refill(level)

OOP Example 2

CLASS : Patient

<u>Attributes</u>
name : string
patient_id : integer
age : integer

<u>Methods</u>
attend_ed()
receive_treatment()

The Class contains the generalised description / blueprint

Objects are *instances* of a Class that *inherit* attributes and methods from the parent Class

Inheritance

OBJECT
name = “Bob Jones”
patient_id = 23195392
age = 64

attend_ed()
receive_treatment()

OBJECT
name = “Mary Smith”
patient_id = 64582990
age = 51

attend_ed()
receive_treatment()

OBJECT
name = “Janet Small”
patient_id = 45189927
Age = 37

attend_ed()
receive_treatment()

Constructors

A *constructor* defines what happens when an object is *instantiated* from a class (an instance object is created from a class).

The constructor essentially "constructs" the object, and specifies the (initial) values for the attributes and methods of the object.

The constructor is a method within the Class.

Defining a Class

Let's now look at how we'd implement OOP in Python. Let's first consider how we create a class :

```
# Declare a class using the class keyword followed by a class name, which
# should start with a capital letter
class My_Class:
    # Class attributes go here, and are attributes that have the same
    # value for instances created from this class
    class_attribute_1 = "Class attribute value"

    # The constructor method.  Note the two underscores both before and after
    # init in the name.  The constructor sets up the Instance Attributes
    # (those with values specific to this instance of a class).  Any values
    # that need to be passed in to set up the attributes need to be declared
    # here, as with any function (method = a function in a class).  However,
    # all class methods must have 'self' as the first parameter (even if there
    # are no other attributes).  The 'self' parameter essentially stores
    # the instance of the class (the copy from the blueprint).  Don't worry
    # too much about this - just know that you need to include it, and that
    # when you say self.something you're referring to the instance of the
    # Class, rather than the Class itself
    def __init__(self, attribute_1, attribute_2):
        self.attribute_1 = atrribute_1
        self.attribute_2 = attribute_2

    # All classes must have at least a constructor, but may also have other
    # methods.  These methods must have self as their first parameter, but
    # may also have other parameters, as we've seen before in functions.
    def method_1(self):
        # What do we want method_1 to do

    def method_2(self, method_parameter_1):
        # What do we want method_2 to do with method_parameter_1
```

FUN FACT :
__init__ is an example of a "Dunder" method ("Dunder" short for "Double Underscore"); also known as "Magic Methods"

Don't worry too much about what these methods are – the key thing to know is that magic / dunder methods do not require you to call them manually; they are called automatically when the time is right. __init__ is called when an object is created.

Defining a Class

Let’s look at how we might set up our Vehicle class example.

CLASS : Vehicle

<u>Attributes</u>
common_name : string
number_of_wheels : integer
capacity : integer

<u>Methods</u>
drive(speed)

```
class Vehicle:
    def __init__(self, common_name, number_of_wheels, capacity):
        self.common_name = common_name
        self.number_of_wheels = number_of_wheels
        self.capacity = capacity

    def drive(self, speed):
        print (f"I'm now driving at {speed} mph.")
```

Instantiation

We’ve defined our Vehicle class now but, as with functions, nothing will happen until we use it. So let’s create an *instance* of the *class* (a process known as *Instantiation*). Let’s create an ambulance, and tell it to drive at 60mph :

```
# Create an instance of the Vehicle class called my_ambulance, that's an
# ambulance with 4 wheels and a passenger capacity of 3
my_ambulance = Vehicle("Ambulance", 4, 3)

# Tell the ambulance to drive at 60mph by running the drive() method of
# the instance.  Note - you don't need to pass 'self', this is all done
# automatically.
my_ambulance.drive(60)
```

```
I'm now driving at 60 mph.
```

Multiple Instances

The beauty of OOP is that we only need to specify the Class once, but we can generate as many instances of it as we like with minimal code :

```
mikes_ambulance = Vehicle("Ambulance", 4, 3)
seans_car = Vehicle("Car", 3, 2)
alisons_bicycle = Vehicle("Bike", 2, 1)
toms_monster_truck = Vehicle("Monster Truck", 4, 1)
```

Referencing my own attributes

Let’s say we want to add an attribute called “Owner” to the Vehicle Class, and have the owner displayed in the message when the drive method is called :

```
class Vehicle:
    def __init__(self, common_name, number_of_wheels, capacity, owner):
        self.common_name = common_name
        self.number_of_wheels = number_of_wheels
        self.capacity = capacity
        self.owner = owner

    def drive(self, speed):
        print (f"{self.owner} is now driving at {speed}mph.")

mikes_ambulance = Vehicle("Ambulance", 4, 3, "Mike")
seans_car = Vehicle("Car", 3, 2, "Sean")
alisons_bicycle = Vehicle("Bike", 2, 1, "Alison")
toms_monster_truck = Vehicle("Monster Truck", 4, 1, "Tom")

mikes_ambulance.drive(50)
seans_car.drive(20)
alisons_bicycle.drive(7)
toms_monster_truck.drive(80)

print (f"Tom's vehicle has {toms_monster_truck.number_of_wheels} wheels")
```

```
Mike is now driving at 50 mph.
Sean is now driving at 20 mph.
Alison is now driving at 7 mph.
Tom is now driving at 80 mph.
Tom's vehicle has 4 wheels
```

More on the constructor

Not all attributes have to be passed in to the constructor from externally. For example, we may have attributes that will always have a default value :

```
# Here, we only need to provide a name when creating a new instance of a
# Student - the other two attributes will be set to default values
class Student:
    def __init__(self, name):
        self.name = name
        self.completed_course = False
        self.level = 1

my_student = Student("Dan Chalk")

# The parameter names passed in don't have to match the names of the attributes
# in the class either (although often they will).  But make sure you refer to
# the correct attribute name when referencing the object.
class Student:
    def __init__(self, full_name, dob, year):
        self.name = full_name
        self.date_of_birth = dob
        self.class_year = year
```

Exercise 2

Let's try out what we've just shown you. Write a Class definition in Python for the Patient class outlined on this slide.

The attend_ed() method should randomly sample a time for the patient to remain in ED from an exponential distribution based on the mean time in minutes passed into the method, and print a message stating their name, patient ID and how long they were there. To sample from an exponential distribution, we can use the *expovariate* function of the random library, and specify lambda as the input parameter (lambda = 1 / mean).

The receive_treatment() method will randomly determine whether the patient is cured, with the probability of cure being determined by the number (between 0 and 1) passed into the method. A uniform distribution is sampled from to determine if they are cured or not based on this probability (think how you'd do this). If the patient is cured, their cured attribute is updated to True (this attribute defaults to false on instantiation, and therefore doesn't need to be passed into the constructor, but does still need setting up in it). A message is displayed to inform the user either way, with the patient's details, including name, ID and prob of cure.

Once you've defined the class, create some instances of the Patient class and call the attend_ed() and receive_treatment() methods to see what happens. You can either do this in the program, or interactively in the iPython console after you've run the code containing the Class definition.

You have 45 minutes (+10 min break). Work in your groups.

CLASS : Patient

Attributes
name : string
patient_id : integer
age : integer
cured : boolean (default = False)

Methods
attend_ed(mean_time)
receive_treatment(prob_cure)

```
random.expovariate(lambd)
```

```
random.uniform(a, b)
```

Inheritance

Sometimes we may want to have a class that is very similar to another class, but with a few added extras.

For example, rather than set up an ambulance as an instance of the Vehicle class, we could set up a separate class called Ambulance, that is same as a Vehicle class, but with the addition of having an attribute indicating whether the siren is going off (which wouldn’t apply to most other vehicles).

In OOP, we can do this using something known as *Inheritance.*

Inheritance

CLASS : Vehicle

<u>Attributes</u>
number_of_wheels : integer
capacity : integer

<u>Methods</u>
drive(speed)

PARENT

CHILD

CLASS : Ambulance

<u>Attributes</u>
number_of_wheels : integer
capacity : integer
siren_on : boolean

<u>Methods</u>
drive(speed)
turn_on_siren()
turn_off_siren()

The child inherits all of the attributes and methods of the parent, but may also have additional ones

Inheritance in Python

Inheritance is easy in Python :

```python
class Vehicle:
    def __init__(self, number_of_wheels, capacity, owner):
        self.number_of_wheels = number_of_wheels
        self.capacity = capacity
        self.owner = owner

    def drive(self, speed):
        print (f"{self.owner} is now driving at {speed}mph.")

# To create a new Child class, we simply define a class in the normal way,
# but feed the Parent class in as an input
class Ambulance(Vehicle):
    # We can use the super() function to refer to the Parent class.  Here,
    # we declare our Ambulance constructor with a new attribute - siren_on -
    # and then use super().__init__ to call the Parent constructor so we don't
    # have to repeat all of that code.  We can then just add the new bit.
    def __init__(self, number_of_wheels, capacity, owner, siren_on):
        super().__init__(number_of_wheels, capacity, owner)
        self.siren_on = siren_on

    # We can set up new methods in the normal way
    def turn_on_siren(self):
        self.siren_on = True
        print ("Siren turned on")

    def turn_off_siren(self):
        self.siren_on = False
        print ("Siren turned off")

    # And if we want to overwrite an existing method we can just give it the
    # same name as the one in our Parent class
    def drive(self, speed):
        print (f"This ambulance is driving at {speed}mph.")

my_ambulance = Ambulance(4, 3, "Dan", False)
my_ambulance.turn_on_siren()
my_ambulance.drive(70)
```

```
Siren turned on
This ambulance is driving at 70 mph.
```

Looping through Objects in Python

Python allows us to easily loop through objects, allowing us to extract attributes or call methods :

```python
import random

class Vehicle:
    def __init__(self, common_name, number_of_wheels, capacity, owner):
        self.common_name = common_name
        self.number_of_wheels = number_of_wheels
        self.capacity = capacity
        self.owner = owner

    def drive(self, speed):
        print (f"{self.owner} is now driving at {speed:.1f}mph.")

mikes_ambulance = Vehicle("Ambulance", 4, 3, "Mike")
seans_car = Vehicle("Car", 3, 2, "Sean")
alisons_bicycle = Vehicle("Bike", 2, 1, "Alison")
toms_monster_truck = Vehicle("Monster Truck", 4, 1, "Tom")

list_of_vehicles = [mikes_ambulance, seans_car, alisons_bicycle,
                    toms_monster_truck]

for vehicle in list_of_vehicles:
    if vehicle.common_name == "Bike":
        vehicle.drive(random.uniform(2,12))
    else:
        vehicle.drive(random.uniform(20,70))

    print(f"It is a {vehicle.capacity} passenger vehicle.")
```

```
Mike is now driving at 53.0mph.
It is a 3 passenger vehicle.
Sean is now driving at 37.2mph.
It is a 2 passenger vehicle.
Alison is now driving at 7.7mph.
It is a 1 passenger vehicle.
Tom is now driving at 65.7mph.
It is a 1 passenger vehicle.
```

Reusability - Modules

Another key advantage of OOP is it allows for *reusability*. In other words we can write some code and then reuse it in lots of other places, and even in different programs!

To do this, we can store a .py file that represents a ***module*** – multiple Class definitions and functions – and then import it elsewhere – either the entire module, or just certain methods.

Let’s look at an example.

Reusability - Modules

Penchord_Wizardry.py

```python
import random

# Class defining a Penchordian
class Penchordian :
    def __init__(self, name):
        self.name = name
        self.is_a_wizard = False

    def write_model(self, type_of_model):
        print (f"{self.name} is now writing a {type_of_model} model.")

    def tell_joke(self, prob_success):
        if random.uniform(0,1) < prob_success:
            print (f"{self.name} attempted a joke.  People loved it!")
        else:
            print (f"{self.name} attempted a joke.  It fell flat.")

# Function to turn someone into a wizard
# Input subject must be an object with a "name" string attribute and a
# "is_a_wizard" boolean attribute
def turn_into_a_wizard(subject):
    subject.is_a_wizard = True
    print (f"{subject.name} is now a wizard.")
```

Reusability - Modules

import_code_1.py

```
import random
# import the entire Penchord_Wizardry module (the Penchordian class and the
# turn_into_a_wizard function)
import Penchord_Wizardry

list_of_penchordian_names = ["Alison", "Dan", "Kerry", "Martin", "Mike",
                             "Sean", "Tom"]

# Randomly select three PenCHORDian names
# random.sample selects three elements from a list without replacement
# (if you want values to be able to be repicked (replacement), use
# random.choices)
chosen_penchordian_names = random.sample(list_of_penchordian_names, 3)

list_of_penchordians = []

for name in chosen_penchordian_names:
    list_of_penchordians.append(Penchord_Wizardry.Penchordian(name))

# Call a couple of the class methods on the three created Penchordian objects
for penchordian in list_of_penchordians:
    penchordian.write_model("Discrete Event Simulation")
    penchordian.tell_joke(0.1)
```

```
Martin is now writing a Discrete Event Simulation model.
Martin attempted a joke.  It fell flat.
Dan is now writing a Discrete Event Simulation model.
Dan attempted a joke.  People loved it!
Sean is now writing a Discrete Event Simulation model.
Sean attempted a joke.  It fell flat.
```

Reusability - Modules

import_code_2.py

```
# just import the turn_into_a_wizard function
from Penchord_Wizardry import turn_into_a_wizard

# Define a new class called HSMA, which has two attributes - a name, and an
# is_a_wizard boolean
class HSMA:
    def __init__(self, name):
        self.name = name
        self.is_a_wizard = False

# Create a new HSMA object, whose name is Gandalf
my_promising_HSMA = HSMA("Gandalf")

# Turn Gandalf into a wizard using the function we imported from the
# Penchord_Wizardry module
turn_into_a_wizard(my_promising_HSMA)
```

```
Gandalf is now a wizard.
```

Lunch Break

We’ll now break for lunch until 1.30pm. Have something to eat, stretch your legs, and don’t think about coding for a while.

When we return at 1.30, we’ll be talking about the final two topics in the Python Programming course – plotting graphs and debugging!

MatPlotLib

If we're going to be building models or analysing data in Python, we're likely going to want to plot graphs.

We could export the data and plot it elsewhere, but Python has very nice built in libraries for plotting.

One of the most widely used is *MatPlotLib*.

OO MatPlotLib

Traditionally, MatPlotLib has had (many) different ways of doing the same thing, including both Object Oriented and non-Object Oriented methods.

However, MatPlotLib’s developers are now recommending exclusive use of the Object Oriented way of doing things.

So that’s what we’re going to teach you here.

Importing the Libraries

As always, let's start by making the necessary imports. Here we're going to make the import that you'll be using most of the time when you're plotting with MatPlotLib :

```
import matplotlib.pyplot as plt # provides matlab-style plotting interface
```

MatPlotLib Objects

There are two objects in a MatPlotLib graph :

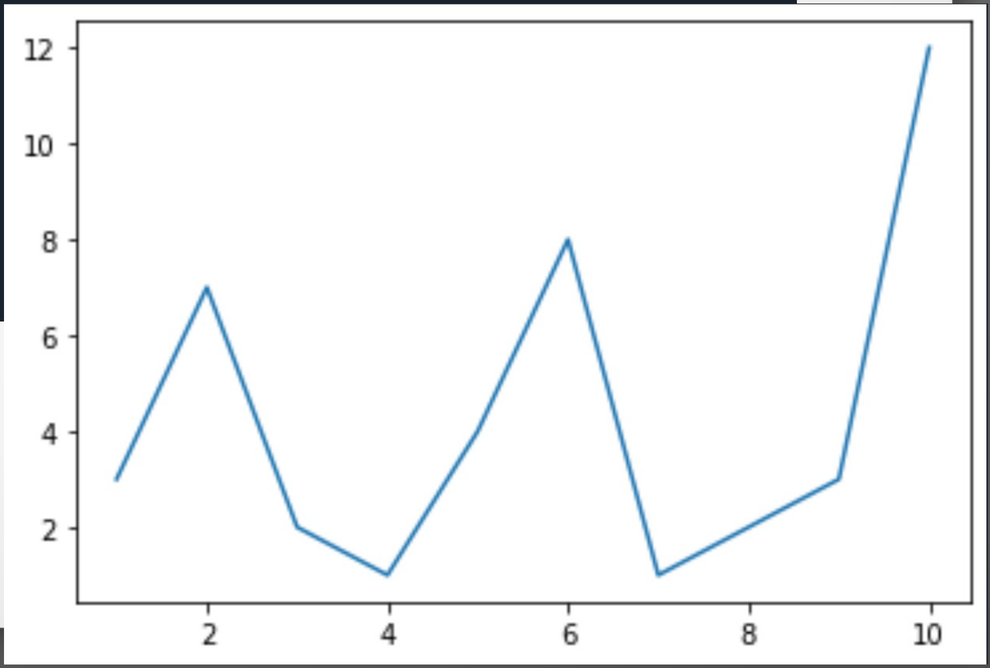
Figure – holds the graph (plotting area)

Axes – An axes object represents a subplot within a figure. Most of the time we’ll just have one axes object (that may have multiple lines / points plotted). For figures with multiple subplots (e.g. side by side, 4 in one figure etc), you’d need multiple axes objects.

Our First Plot

Let's start by plotting a basic line plot :

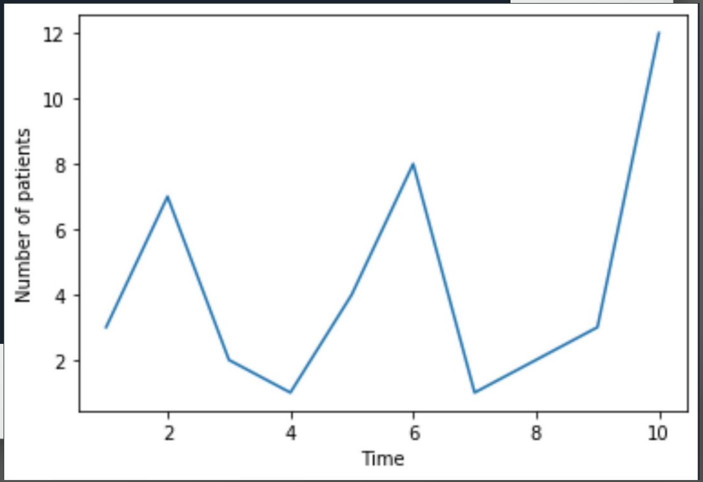
```
import matplotlib.pyplot as plt # provides matlab-style plotting interface

# Data to plot
x = [1,2,3,4,5,6,7,8,9,10]
y = [3,7,2,1,4,8,1,2,3,12]

# Create a figure object and an axes object, and add the axes object as a
# subplot of the figure object
figure_1, ax = plt.subplots()

# Plot our data (x and y here)
ax.plot(x, y)

# Show the figure
figure_1.show()
```



Labelling the axes

Usually we'd want to label our x and y axes :

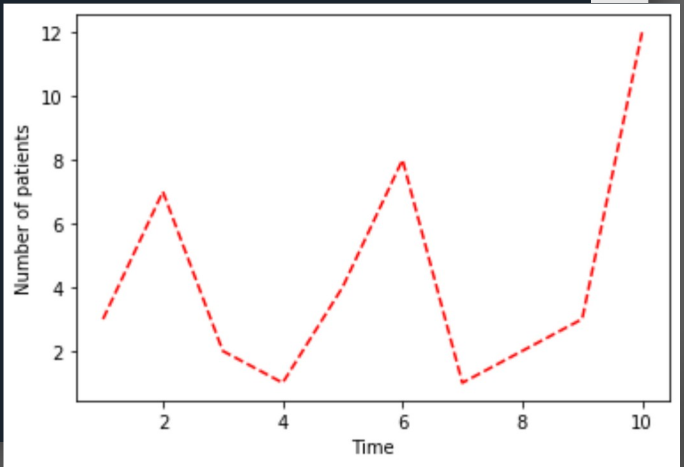
```
import matplotlib.pyplot as plt # provides matlab-style plotting interface

# Data to plot
x = [1,2,3,4,5,6,7,8,9,10]
y = [3,7,2,1,4,8,1,2,3,12]

# Create a figure object and an axes object, and add the axes object as a
# subplot of the figure object
figure_1, ax = plt.subplots()

# Set x axis and y axis labels
ax.set_xlabel('Time')
ax.set_ylabel('Number of patients')

# Plot our data (x and y here)
ax.plot(x, y)

# Show the figure
figure_1.show()
```



Changing line colour or style

Maybe change the colour and / or style of the line :

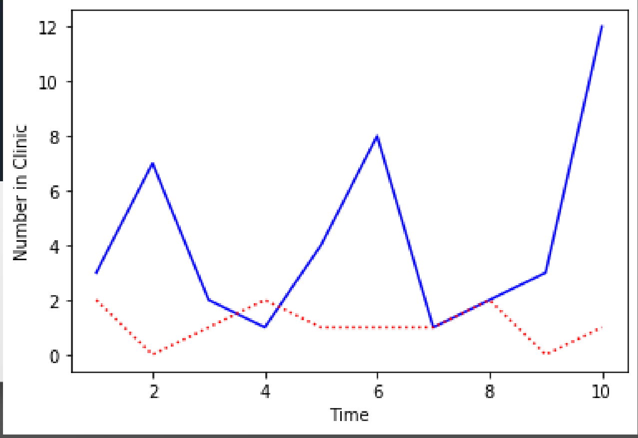
```
import matplotlib.pyplot as plt # provides matlab-style plotting interface

# Data to plot
x = [1,2,3,4,5,6,7,8,9,10]
y = [3,7,2,1,4,8,1,2,3,12]

# Create a figure object and an axes object, and add the axes object as a
# subplot of the figure object
figure_1, ax = plt.subplots()

# Set x axis and y axis labels
ax.set_xlabel('Time')
ax.set_ylabel('Number of patients')

# Plot our data (x and y here)
# Set plot to red and line style to --
ax.plot(x, y, color="red", linestyle="--")

# Show the figure
figure_1.show()
```



Multiple Data Plots within same Subplot

It's easy to add multiple data plots within the same subplot :

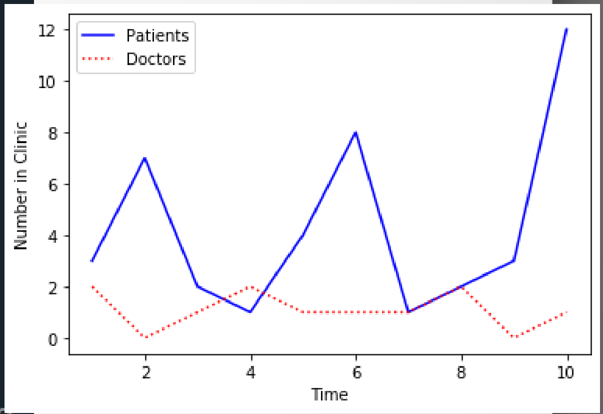
```
import matplotlib.pyplot as plt # provides matlab-style plotting interface

# Data to plot
time = [1,2,3,4,5,6,7,8,9,10]

patients = [3,7,2,1,4,8,1,2,3,12]
doctors = [2,0,1,2,1,1,1,2,0,1]

# Create a figure object and an axes object, and add the axes object as a
# subplot of the figure object
figure_1, ax = plt.subplots()

# Set x axis and y axis labels
ax.set_xlabel('Time')
ax.set_ylabel('Number in Clinic')

# Plot our data, and set each dataset we plot to a different colour / style
ax.plot(time, patients, color="blue", linestyle="-") # Plot patients over time
ax.plot(time, doctors, color="red", linestyle=":") # Plot doctors over time

# Show the figure
figure_1.show()
```



Adding a Legend

If we've got multiple data plots, we probably want a legend :

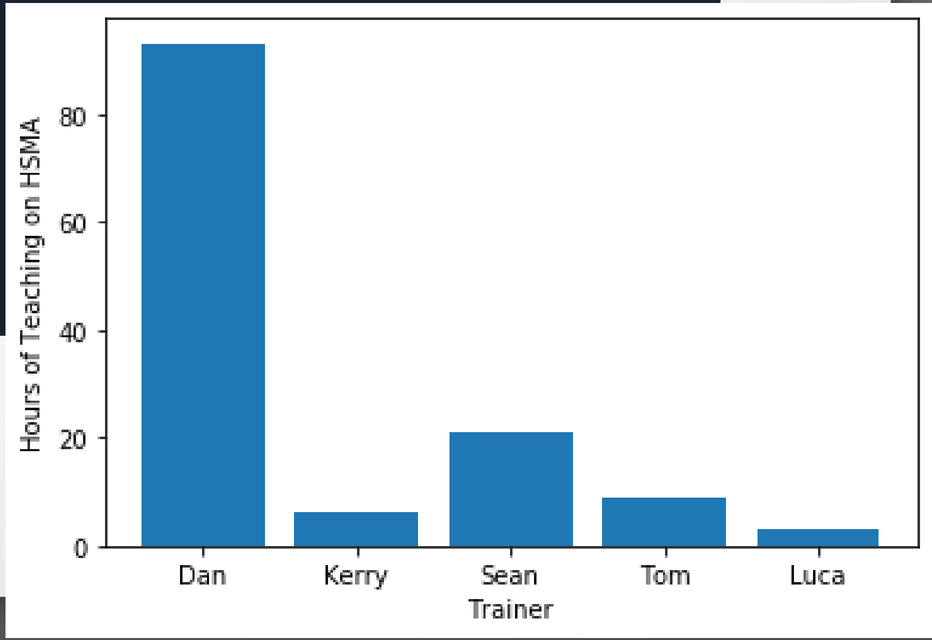
```
import matplotlib.pyplot as plt # provides matlab-style plotting interface

# Data to plot
time = [1,2,3,4,5,6,7,8,9,10]

patients = [3,7,2,1,4,8,1,2,3,12]
doctors = [2,0,1,2,1,1,1,2,0,1]

# Create a figure object and an axes object, and add the axes object as a
# subplot of the figure object
figure_1, ax = plt.subplots()

# Set x axis and y axis labels
ax.set_xlabel('Time')
ax.set_ylabel('Number in Clinic')

# Plot our data, and set each dataset we plot to a different colour / style
ax.plot(time, patients, color="blue", linestyle="-", label="Patients")
ax.plot(time, doctors, color="red", linestyle=":", label="Doctors")

# Create and set up a legend
ax.legend(loc="upper left")

# Show the figure
figure_1.show()

figure_1.savefig("figure_1.pdf")
```



Bar Charts

There are loads of plot types we can do in MatPlotLib. Let's look at another couple, starting with Bar Charts :

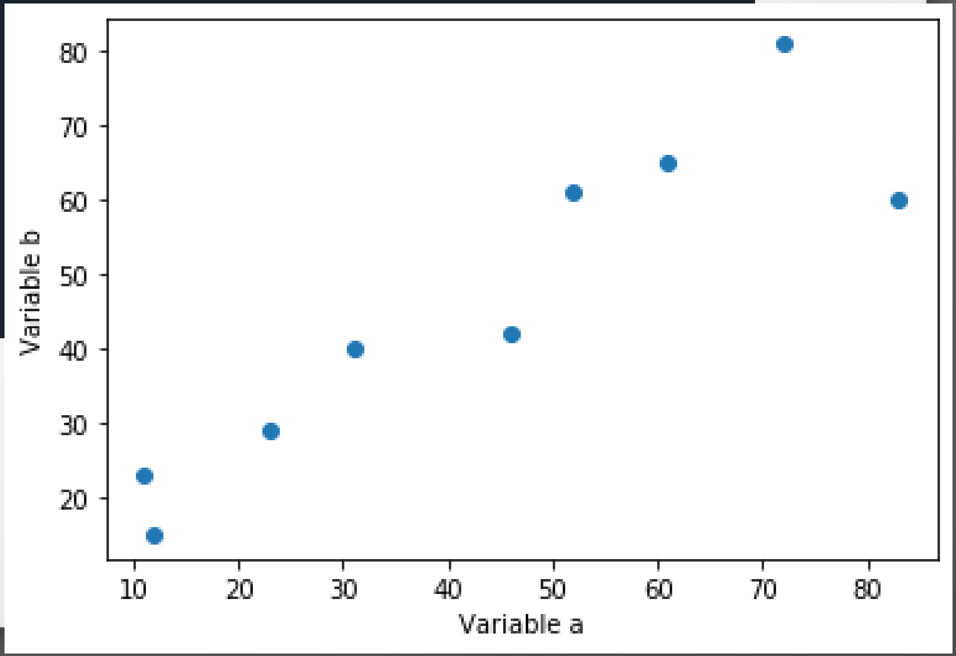
```
import matplotlib.pyplot as plt # provides matlab-style plotting interface

# Data to plot
hsma_trainers = ["Dan", "Kerry", "Sean", "Tom", "Luca"]
hours_of_teaching = [93, 6, 21, 9, 3]

figure_1, ax = plt.subplots()

ax.set_xlabel("Trainer")
ax.set_ylabel("Hours of Teaching on HSMA")

ax.bar(hsma_trainers, hours_of_teaching)

figure_1.show()
```



Scatter Plots

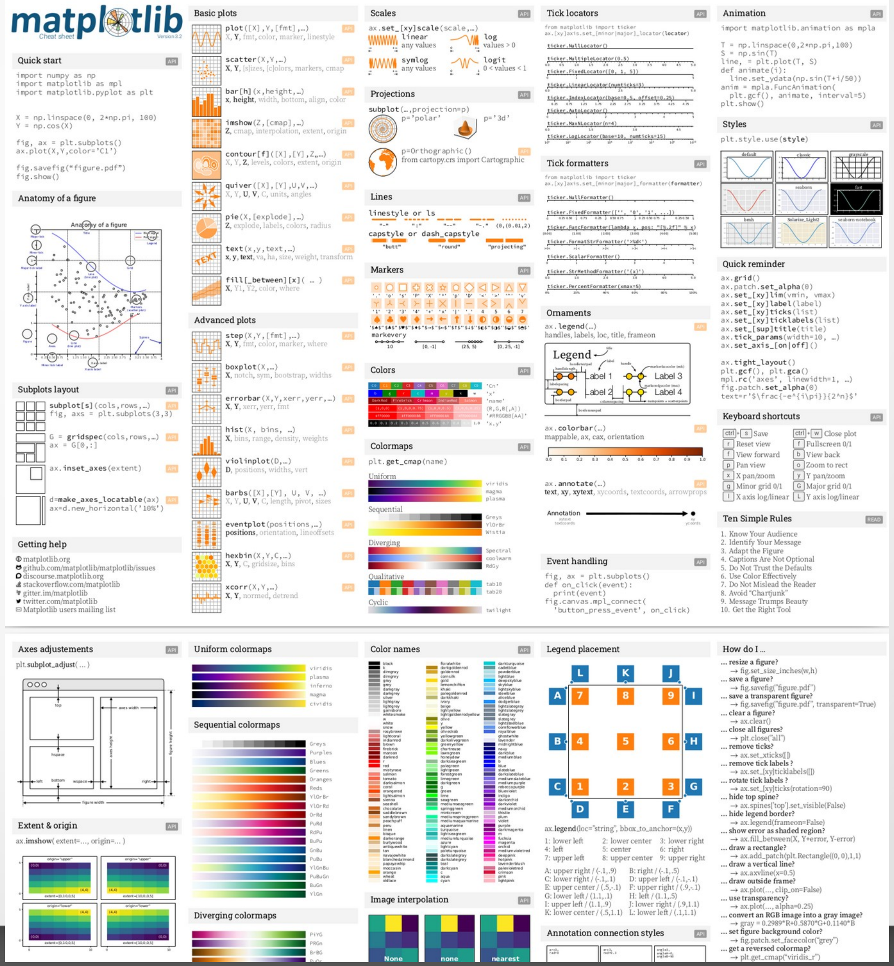
Scatter Plots :

```
import matplotlib.pyplot as plt # provides matlab-style plotting interface

# Data to plot
a = [11,52,61,72,83,12,23,31,46]
b = [23,61,65,81,60,15,29,40,42]

figure_1, ax = plt.subplots()

ax.set_xlabel("Variable a")
ax.set_ylabel("Variable b")

ax.scatter(a, b)

figure_1.show()
```



MatPlotLib Cheat Sheet

MatPlotLib has a cheat sheet containing lots of information about how to plot certain types of graphs, set up styles etc. We've provided you with this cheat sheet – consult it when you're using MatPlotLib.



You should also check out Tom's excellent MatPlotLib tutorials here :

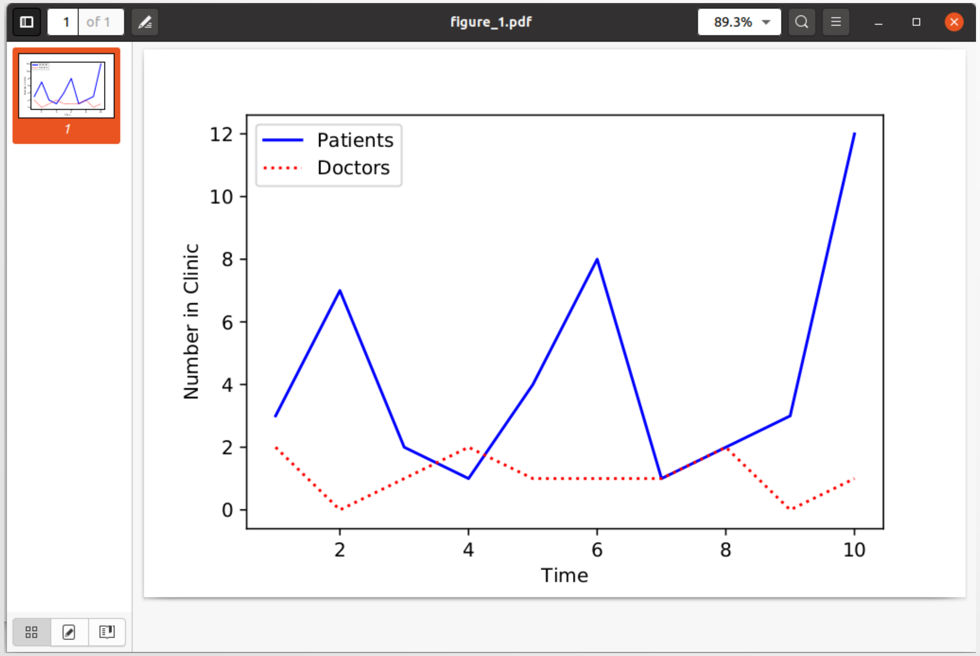
https://bit.ly/3z1AT4X

https://bit.ly/36HAuIt

Saving figures

It's easy to save figures for later. We can either copy and paste them from our IDE (just right click and select Copy), or we can save the figure directly from the code :

```
figure_1.savefig("figure_1.pdf")
```



Exercise 3

1. Write code that plots a line plot comparison of Hospital A and B mean ED attendances per day of week (the numbers in the table below). Use different colours / styles and a legend to clearly differentiate the two lines on the plot, and label each axis appropriately. You should also force start the y-axis from 0 using the *set_ylim()* method of your axes object and passing *ymin=0* as a parameter value to this method. **IMPORTANT : You MUST do this AFTER you plot the data, not before, but before you show the figure, otherwise the y-axis won't resize to the data.**

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Hospital A	242	180	156	191	231	378	345
Hospital B	310	290	317	351	341	261	295

2. Set up a new figure that plots on a bar chart the total mean ED attendances across both hospitals by day of week (ie total for Monday (mean A + mean B), total for Tuesday (mean A + mean B) etc).

Debugging

As you write more and more code, you'll find you'll spend an increasing amount of your time trying to fix problems in your code. This process is known as *debugging.*

The Python interpreter will try to help you out by flagging up what went wrong if your code fails. The usefulness of its "help" varies wildly….

Let's look at some examples.

Debugging

What's happened here?

```
number = input ("Please input a number : ")

result = number * number
```

```
Traceback (most recent call last):

  File "/home/dan/Dropbox/HSMA 3/phase_1_training/3C_Python_Prog_Part_3/debugging_1.py", line 12, in <module>
    result = number * number

TypeError: can't multiply sequence by non-int of type 'str'
```

Debugging

What’s happened here?

```
list_1 = ["Apples", "Oranges", "Strawberries"]

print (list_1[3])
```

```
Traceback (most recent call last):

  File "/home/dan/Dropbox/HSMA 3/phase_1_training/3C_Python_Prog_Part_3/debugging_1.py", line 11, in <module>
    print (list_1[3])

IndexError: list index out of range
```

Debugging

What's happened here? (from https://runestone.academy/runestone/books/published/thinkcspy/Debugging/KnowyourerrorMessages.html)

```
current_time_str = input("What is the current time (in hours 0-23)?")
wait_time_str = input("How many hours do you want to wait")

current_time_int = int(current_time_str)
wait_time_int = int(wait_time_int)

final_time_int = current_time_int + wait_time_int
print(final_time_int)
```

```
Traceback (most recent call last):

  File "/home/dan/Dropbox/HSMA 3/phase_1_training/3C_Python_Prog_Part_3/debugging_1.py", line 13, in <module>
    wait_time_int = int(wait_time_int)

NameError: name 'wait_time_int' is not defined
```

Debugging

What's happened here?

```
a = int(input("Enter first number : "))
b = int(input("Enter second number : "))

if a = b:
    print ("Numbers match")
else:
    print ("No match found")
```

```
SyntaxError: invalid syntax
```

Debugging

What's happened here?

```
a = 3
b = 7
c = 9
d = 10
e = 35
f = a * c

if ((a + b < c and d + (e - c + (f * b) > (f * d / 3)):
    print ("Yes")
```

```
SyntaxError: invalid syntax
```

Debugging

What’s happened here?

```
a = 3
b = 7
c = 9
d = 10
e = 35
f = a * c

if ((a + b < c and d + (e - c + (f * b) > (f * d / 3)))):
    print ("Yes")
else:
    # Do nothing
```

```
SyntaxError: unexpected EOF while parsing
```

Debugging

What's happened here? (this one's a bit trickier – the clue's in the error, but it's not obvious)

```
class Penchordian:
    def __init__(self, name, specialty):
        self.name = name
        self.specialty = specialty

    def start_teaching(self, subject):
        print (f"{self.name} is now teaching {subject}")

Penchordian.start_teaching("Programming")
```

```
TypeError: start_teaching() missing 1 required positional argument: 'subject'
```

Dan’s Devious Debugging Challenge

In the file debugging_challenge.py you’ll find a Python program that has a lot of errors. Your challenge is to fix them and get the code working!

Be warned, there are 53 errors (depending on how you count them)! Try to work out how the code's supposed to work first (I’ve deliberately omitted comments). That will help you a lot. Also, you may get it running, but it may not work as expected. Test everything very carefully and thoroughly.

Work in groups on this challenge for the next hour. Can you fix it?

Q & A

Any questions about what you’ve learned? Any areas you’d like to be covered again?

Further Work

Further Work :
Keep going back over what you've learned over the last three sessions, and practice writing code as much as you can. I can't understate how important this is.

In these sessions, we've covered a number of the basics of Python programming, but by no means everything!

In the upcoming sessions, you'll learn even more Python constructs and methods as we introduce you to various modelling and data science methods.