# Introduction[¶](#Introduction)

Python is one of the best known programming languages. It has also gained both credence and fame in GIS for enabling conducting both simple and complex GIS applications. More than that, programming in GIS is fun. And sometimes the harder it gets, the more impetus you have in coding until you get to the solution.

For today, being still fresh into the new year, we shall start simple. It shall be no more than showing how to plot points, calculate the distance between two points and overlaying them on a new map. The only twist is that we shall do so in [jupyter notebooks](https://jupyter.org/).

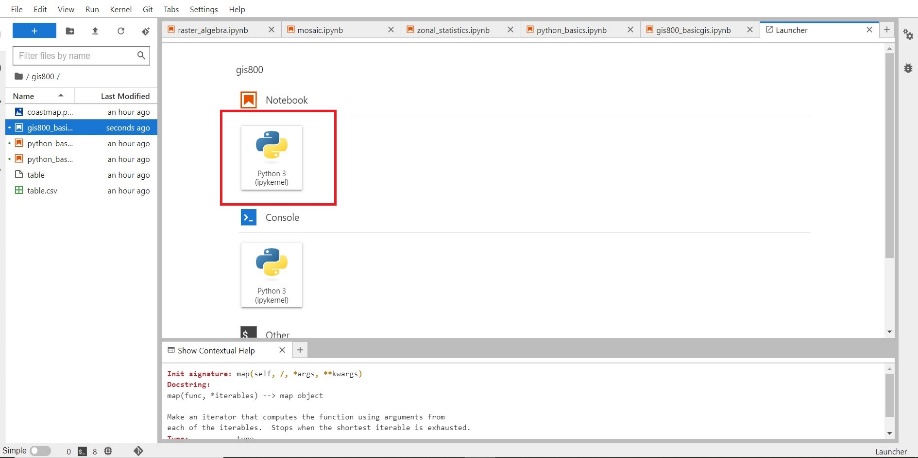
Go to jupyter and create a new python notebook. This can be done by clicking on the icon labelled as "Notebook" which is encircled with a red box as shown below. If you have not installed Jupyter, kindly follow the instructions [here](https://jupyter.org/install), but we strongly suggest you do so using [Anaconda](https://www.anaconda.com/).

In [1]:

from IPython import display

display.Image("E:/documents/gis800\_articles/jupyter/simple\_gis/jupyter.jpg", alt="Launching Jupyter Notebook", width= 400, height=200)

Out[1]:

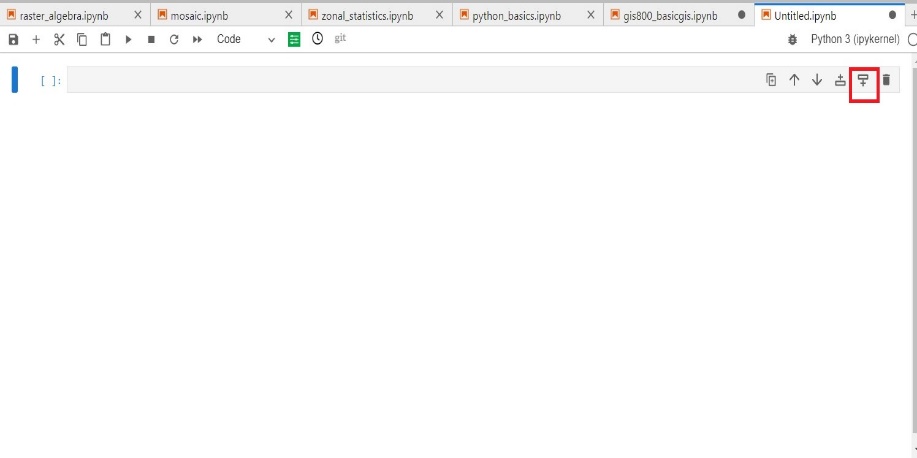
****

Once launched, you will have an empty interface similar to the one below.

In [2]:

display.Image("E:/documents/gis800\_articles/jupyter/simple\_gis/interface.jpg", alt="Notebook Interface", width=400, height=200)

Out[2]:

****

You can add more cells using one of the drop icons shown by a red rectangle in the image above. Having created a substantial number of cells to act as a playground for your coding practice, it's time to start coding!

## Create a Point Geometry in Jupyter Notebooks[¶](#Create-a-Point-Geometry-in-Jupyter-Note)

Let's begin the honors. We shall first import the necessary packages to enable spatial operations using Python much the same way we do in R. But not so fast!

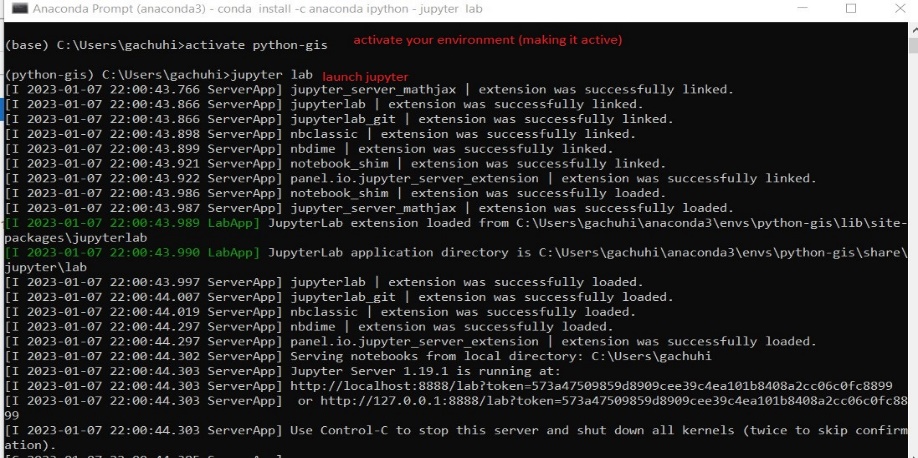
Just like in R where you download the packages for a particular task using the open sesame keyword install.packages("package-name"), the process is also similar in Jupyter. The manner is albeit different. Before the packages are called into the notebook interface, you have to load them into your working environment in Jupyter. Think of an [environment](https://www.geeksforgeeks.org/python-virtual-environment/) as a warehouse where all the tools and dependencies of your project are separated from other warehouses. That is, the tools and dependencies are not shared by other warehouses which could be for the same or different projects. The working environment keeps your work 'safe' and free from 'influence' from other projects.

So, in order to load a new package that will be called into your Jupyter notebook interface, take one step back to your Anaconda command prompt. If you are a beginner and are wondering where the Anaconda command prompt (abbreviated as cmd) comes into the picture, we normally start Jupyter notebooks from the Anaconda cmd with the secret key Jupyter lab. You will see this in action in a bit. In fact, here is the cmd with all this access keywords.

In [3]:

display.Image("E:/documents/gis800\_articles/jupyter/simple\_gis/cmd.jpg", alt="The Anaconda Command Prompt (cmd)", width=400, height=200)

Out[3]:

****

Having cleared the mist on that earlier bit, let's see how one can install a package into anaconda, which can then be loaded into Jupyter notebooks. Think of Anaconda as a large software package that comes with Jupyter installed in it, much the same way Microsoft Office comes with Word pre-installed. Enough of talk. Let's give an example of installing a package into anaconda and subsequently loading it into a Jupyter notebook.

Packages are installed into your anaconda environment in the following format:

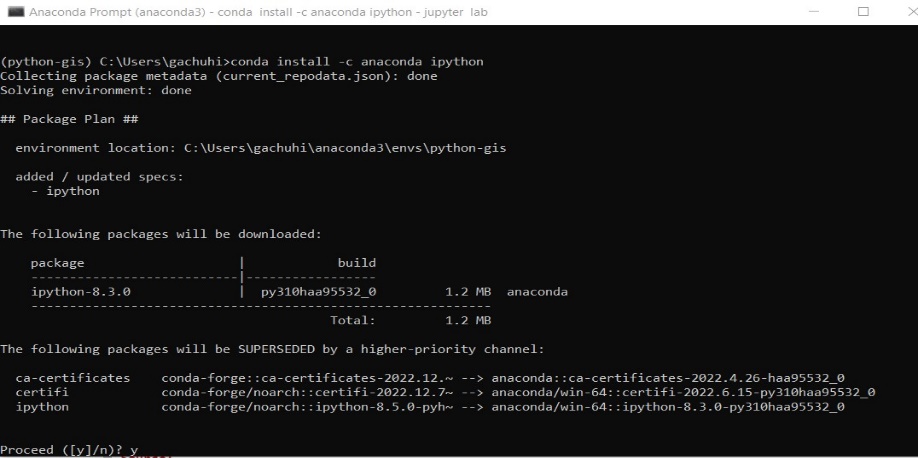
conda install -c anaconda <package name>

As an example, here we show how to install the package ipython which was crucial in seeing the images that you see in this article.

In [4]:

display.Image("E:/documents/gis800\_articles/jupyter/simple\_gis/installing.jpg", alt="Installing a package into Anaconda", width=400, height=200)

Out[4]:

****

Just follow the prompts if you want to have your package installed, including answering "y" for "Yes" in the last question. Anaconda will print the following statements in the cmd if your package was installed without fuss:

Downloading and Extracting Packages

Preparing transaction: done

Verifying transaction: done

Executing transaction: done

We've spent enough time in the dark world of the cmd. Let's get back to the bright Jupyter notebook interface. It's about time we started doing some programming. Remember when after installing a package in R (because my background in programming first started with R), we would load the package into the R script using the protocol library(package name)? Well, guess what, Python uses a not too dissimar format.

Here are the first packages we shall load into Jupyter notebook.

In [5]:

# import necessary packages

import numpy as np

from shapely.geometry import Point

We want to draw the point coordinates of a place, some interesting place, say the Vasco Da Gama pillar in Kenya. We know the coordinates of the place are Lat-Lon (40.1276701, -3.2236304). In Python, point coordinates are drawn using the function Point().

In [6]:

# draw a coordinate of a place - Vasco Da Gama Pillar

vasco = Point(40.1276701, -3.2236304)

Let's plot the above point we've created.

In [7]:

# Show the vasco da Gama point

vasco

Out[7]:

b''

There goes our first success of plotting an object in Python. But just because it's a point from coordinates does not make it a geometry object. Our point object does not have a [geometry column](https://geopandas.org/en/stable/docs/user_guide/data_structures.html#geodataframe), which is crucial for geospatial operations. The geometry column is what contains the spatial information of any object. But we shall get to it in a bit.

## A Geodataframe. Does 'geo' in the dataframe bring any difference?[¶](#A-Geodataframe.-Does-'geo'-in-the-dataf)

We would like to convert our point object into a geo-dataframe. Why? Isn't it already good the way it is?

No.

If you would like to have your point object attributes, and they can be many, sometimes in their thousands, then you need to convert it from just a simple list, to a geodataframe. The geodataframe is what will enable your object be assigned a Coordinate Reference System (CRS) from which other spatial operations can be performed on it. Some of these operations are impossible if it just remains the way it is.

We shall do the conversion in two simple steps. Convert the point object to a dataframe, and thereafter to a geodataframe.

In [8]:

# import the necessary packages

import pandas as pd # imports pandas package which has numerous tools for dataframe manipulation

import geopandas as gpd # works like pandas, but suited for dataframes with geometry column aka spatial dataframes

Now let's transform our point object of vasco into a dataframe first.

In [9]:

# Transform vasco point coordinates to a dataframe

vasco = pd.DataFrame({'Longitude': [40.1276701],

'Latitude': [-3.2236304]})

Now let's insert the missing piece of the puzzle by converting the data frame into a geo dataframe.

In [10]:

# Convert the dataframe of vasco point to geoDataFrame

vasco\_gpd = gpd.GeoDataFrame(vasco,

geometry=gpd.points\_from\_xy(vasco.Longitude, vasco.Latitude),

crs="EPSG:4326")

Are you curious to see how the geo dataframe is different from the normal dataframe? You can easily find this out by using the method head() which shows the first few rows of a dataframe. A geo dataframe is simply a dataframe with an added spatial component, the geometry column.

In [11]:

# Lets view the contents of vasco geodataframe

vasco\_gpd.head()

Out[11]:

|  | **Longitude** | **Latitude** | **geometry** |
| --- | --- | --- | --- |
| **0** | 40.12767 | -3.22363 | POINT (40.12767 -3.22363) |

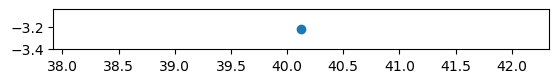
Nailed it! Let's plot it.

In [12]:

vasco\_gpd.plot()

Out[12]:

<AxesSubplot: >

****

Just a one lonely small point with no surrounding neighbours to help us understand where it is exactly. Since we had earlier mentioned we want to measure the distance between two points, we shall add another point geometry of an interesting place-- the Gedi Ruins. The point coordinates for our newest kid on the block are Lat-Lon (39.9623916, -3.334307).

Let's add it.

In [13]:

# Let's add the coordinates for Gedi ruins to those of Vasco

gedi = pd.DataFrame({'Longitude': [39.9623916],

'Latitude': [-3.334307]})

coast\_points = vasco.append(gedi) # Appending the points of Gedi to the first geodataframe of vasco

C:\Users\gachuhi\AppData\Local\Temp\ipykernel\_1536\1267294226.py:5: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

coast\_points = vasco.append(gedi) # Appending the points of Gedi to the first geodataframe of vasco

Let's see how our geodataframe is faring out after adding an additional point geometry.

In [14]:

# Show coast\_points attributes

coast\_points.head()

Out[14]:

|  | **Longitude** | **Latitude** | **geometry** |
| --- | --- | --- | --- |
| **0** | 40.127670 | -3.223630 | POINT (40.12767 -3.22363) |
| **0** | 39.962392 | -3.334307 | None |

Something's amiss in our table above, but it still makes sense. Our gedi object was still a dataframe, and not a geodataframe when appending to the spatially referenced vasco object. No wonder the geometry value of Gedi Ruin's attributes are None. No need to worry. It shall be fixed in no time.

In [15]:

# Convert the coast\_points dataframe to GeoDataframe

coast\_points\_gpd = gpd.GeoDataFrame(coast\_points,

geometry=gpd.points\_from\_xy(coast\_points.Longitude, coast\_points.Latitude),

crs="EPSG:4326")

coast\_points\_gpd.head()

Out[15]:

|  | **Longitude** | **Latitude** | **geometry** |
| --- | --- | --- | --- |
| **0** | 40.127670 | -3.223630 | POINT (40.12767 -3.22363) |
| **0** | 39.962392 | -3.334307 | POINT (39.96239 -3.33431) |

## Plotting a map in Jupyter Notebook[¶](#Plotting-a-map-in-Jupyter-Notebook)

We know you are itching to leave the coding and move to plotting and seeing the two points on a map. You are dying to see how a map looks like in Jupyter, the true temperance of a modern day cartographer! Don't rush yet. Our Points don't have names, and thus no one can easily tell which coordinates are for Vasco da Gama and which are for Gedi Ruins. Let's put in some names to reference each of our point object's coordinates.

In [16]:

# Add names of the places to the points

coast\_points\_gpd.insert(loc=0,

column='Names',

value=['Vasco da Gama Pillar', 'Gedi Ruins']

)

coast\_points\_gpd.head()

Out[16]:

|  | **Names** | **Longitude** | **Latitude** | **geometry** |
| --- | --- | --- | --- | --- |
| **0** | Vasco da Gama Pillar | 40.127670 | -3.223630 | POINT (40.12767 -3.22363) |
| **0** | Gedi Ruins | 39.962392 | -3.334307 | POINT (39.96239 -3.33431) |

Still an issue though. Our indexing is incorrect. It looks like when adding the Gedi object, which was an object variable of a single row at index[0], this property was carried to the geodataframe as well and was not auto-incremented. Thus the two zeroes. If you are beginner to programming, just take index as position of something, and all numbering in computers begins from zero, as is our case here.

Let's sort this out using the reset\_index() function. A small tip before you move on. Jupyter also shows the help documentation for each function. See this [article](https://towardsdatascience.com/15-tips-and-tricks-for-jupyter-notebook-that-will-ease-your-coding-experience-e469207ac95c) for more details on how to view the help content in your notebook.

In [17]:

# Arrange indexing

coast\_points\_gpd.reset\_index(inplace=True)

coast\_points\_gpd.head()

Out[17]:

|  | **index** | **Names** | **Longitude** | **Latitude** | **geometry** |
| --- | --- | --- | --- | --- | --- |
| **0** | 0 | Vasco da Gama Pillar | 40.127670 | -3.223630 | POINT (40.12767 -3.22363) |
| **1** | 0 | Gedi Ruins | 39.962392 | -3.334307 | POINT (39.96239 -3.33431) |

By now your patience has already run out. It's either we map it, or we just pack and go. But before we dive in to the mapping bit, we will make this confession--plotting maps in Python can sometimes be more labourious than in R. That's our biased confession. Your's could be different.

Okay, here comes the map!

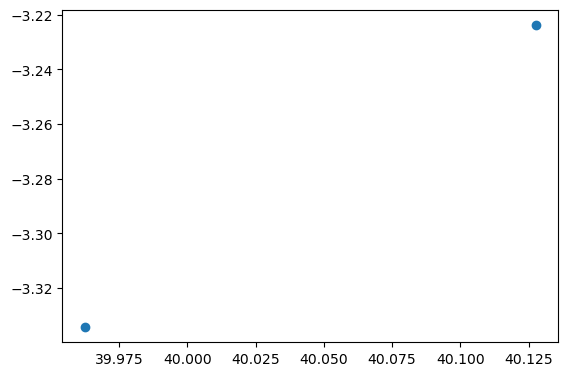
In [18]:

# Plot Distance Between Vasco da Gama and Gedi Ruins

coast\_points\_gpd.plot()

Out[18]:

<AxesSubplot: >

****

Easy, but not fair enough. Let's add some sophistication to the above plain map with say, some annotation.

In [19]:

# Call the matplotlib package used to plot scatterplots, bargraphs, and among many others, maps

import matplotlib.pyplot as plt

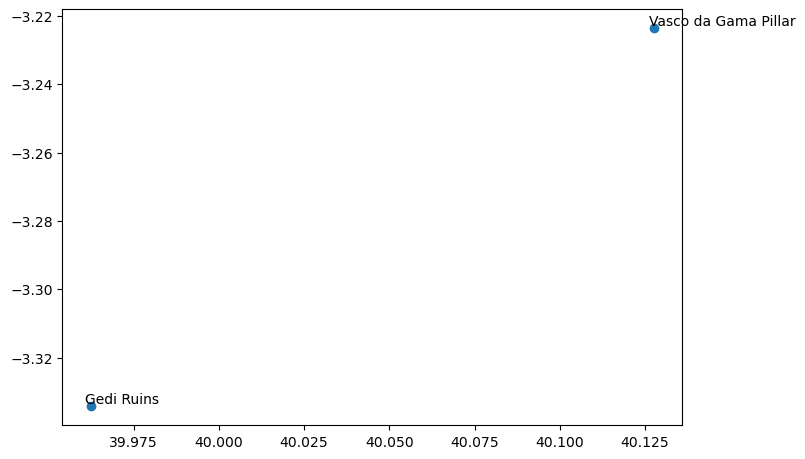
In [24]:

fig, ax = plt.subplots(figsize=(8, 6))

coast\_points\_map = coast\_points\_gpd.plot(ax=ax)

for name, lon, lat in zip(coast\_points\_gpd.Names, coast\_points\_gpd.Longitude, coast\_points\_gpd.Latitude):

coast\_points\_map.annotate(name, xy=(lon, lat), xytext=(-4, 2), textcoords='offset points')

****

Now its time to measure the distance between our two points of interest. Can we, the tourist, be able to visit the two places all in one day? It's for python to find out.

In [21]:

# Measure line distance between the two points of Vasco da Gama Pillar and Gedi ruins

coast\_points\_crs = coast\_points\_gpd.to\_crs("EPSG:32737")# Convert to local datum

coast\_points\_crs.crs

Out[21]:

<Derived Projected CRS: EPSG:32737>

Name: WGS 84 / UTM zone 37S

Axis Info [cartesian]:

- E[east]: Easting (metre)

- N[north]: Northing (metre)

Area of Use:

- name: Between 36°E and 42°E, southern hemisphere between 80°S and equator, onshore and offshore. Kenya. Mozambique. Tanzania.

- bounds: (36.0, -80.0, 42.0, 0.0)

Coordinate Operation:

- name: UTM zone 37S

- method: Transverse Mercator

Datum: World Geodetic System 1984 ensemble

- Ellipsoid: WGS 84

- Prime Meridian: Greenwich

We have converted the dataframe from the WGS84 format to the local datum EPSG:32737 which is in metres. There are many tools to calculate distance between two points in Python. The geopy package is one of them, but it does not need the geodataframe to be converted to local datum format. However, we have done that because it is best practice to use a local CRS for distance calculation, rather than a global one where measurement values are in degrees.

In [22]:

from geopy import distance

point1 = (40.127670, -3.223630) # coordinates for Vasco Da Gama Pillar

point2 = (39.962392, -3.334307) # coordinates for Gedi Ruins

print(distance.distance(point1, point2).km)

20.639578744143794

The two points, along a straight line from each other, are 20km apart. Note the word "straight line" because we did not follow a particular route, which could be either longer or shorter. To calculate the true distance via a path, say a road, we would need a line geometry of the road. This is possible, but beyond the scope of this article.

You might also be curious as to why we hardcoded the coordinates rather than extract them from the geodataframe using a method such as iloc or loc. Truth of the matter is, we tried all we could, but Python kept throwing back several errors, something to do with strings not being float values. Therefore, we resulted to hardcording the coordinates although this is redundancy at its peak.

Time for the icing on the cake. Where are these points situated on real ground? A web map will transform the dull colorless map to something livelier. We shall use contextily package which does a beautiful job of putting spatial objects in their geographic context.

In [23]:

import contextily as cx

fig, ax = plt.subplots(figsize=(8, 6))

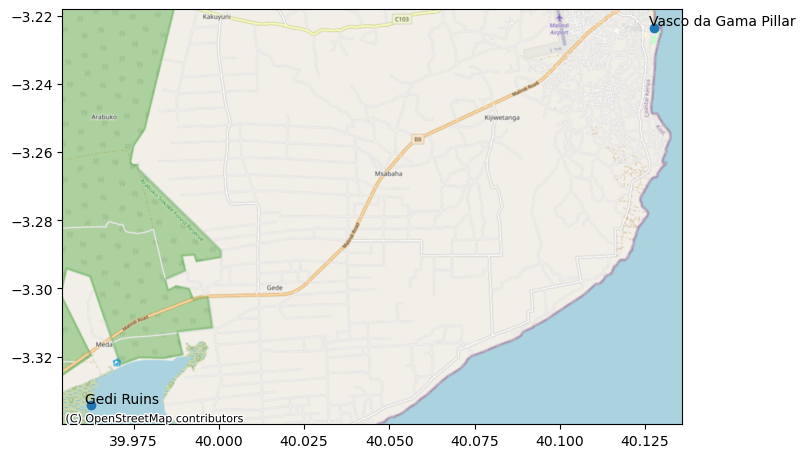
coast\_points\_map = coast\_points\_gpd.plot(ax=ax)

for name, lon, lat in zip(coast\_points\_gpd.Names, coast\_points\_gpd.Longitude, coast\_points\_gpd.Latitude):

coast\_points\_map.annotate(name, xy=(lon, lat), xytext=(-4, 2), textcoords='offset points')

cx.add\_basemap(ax, crs=coast\_points\_gpd.crs, source=cx.providers.OpenStreetMap.Mapnik)

plt.savefig("E:/documents/gis800\_articles/jupyter/simple\_gis/coastmap.png", dpi=400, orientation='landscape') # Saving it to a directory

****

# Conclusion[¶](#Conclusion)

We have used Python within the Jupyter Notebook's platform to conduct a few geospatial operations. The advantage of Jupyter Notebooks compared to other platforms such as Pycharm is the ability to code on the go. That is, results of operations appear in the cells that follow. Other than that, Jupyter notebooks is an equally good platform as Pycharm for conducting geospatial operations via a web interface.

In [ ]:

In [ ]: