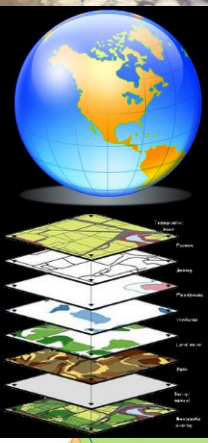
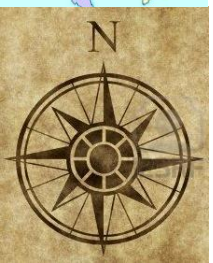
# GIS Programming

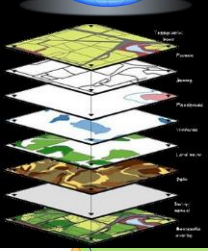
Moffat Magondu

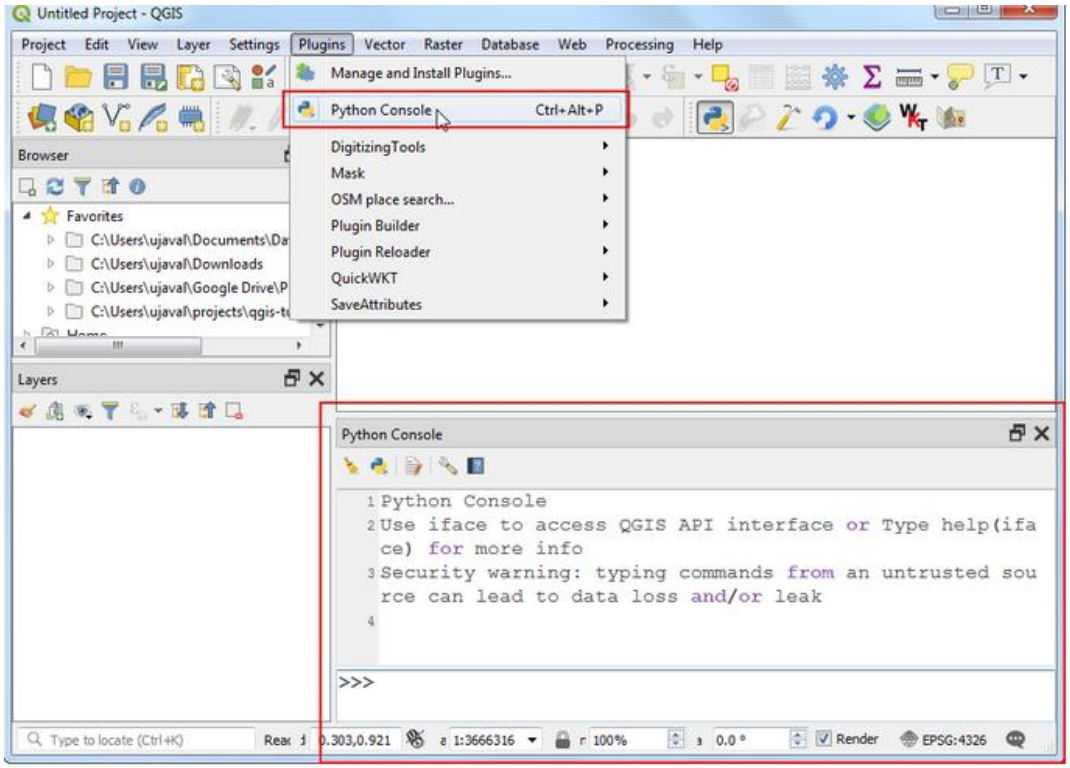

# PYTHON IN QGIS

- Python is a great versatile programming language that is widely used in the GIS world and beyond. Knowing how to program will enable you to be more efficient by automating workflows and making your work reproducible.
- In this lesson we introduces the concepts of Python programming within the QGIS environment.
- Where can you use Python in QGIS?
  - Issue commands from Python Console
  - Automatically run python code when QGIS starts
  - Write custom expressions
  - Write custom actions
  - Create new processing algorithms
  - Create plugins
  - Create custom standalone applications
- Python bindings are also available for QGIS Server, including Python plugins and Python bindings that can be used to embed QGIS Server into a Python application.

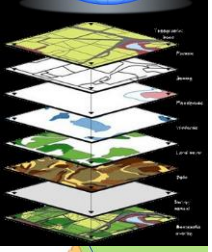



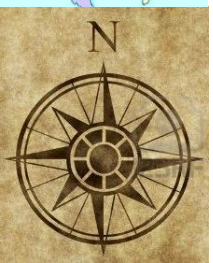

# PYTHON IN QGIS

- QGIS Comes with a built-in Python Console and a code editor where you can write and run Python code.
- Go to Plugins → Python Console to open the console. Issue commands from Python Console









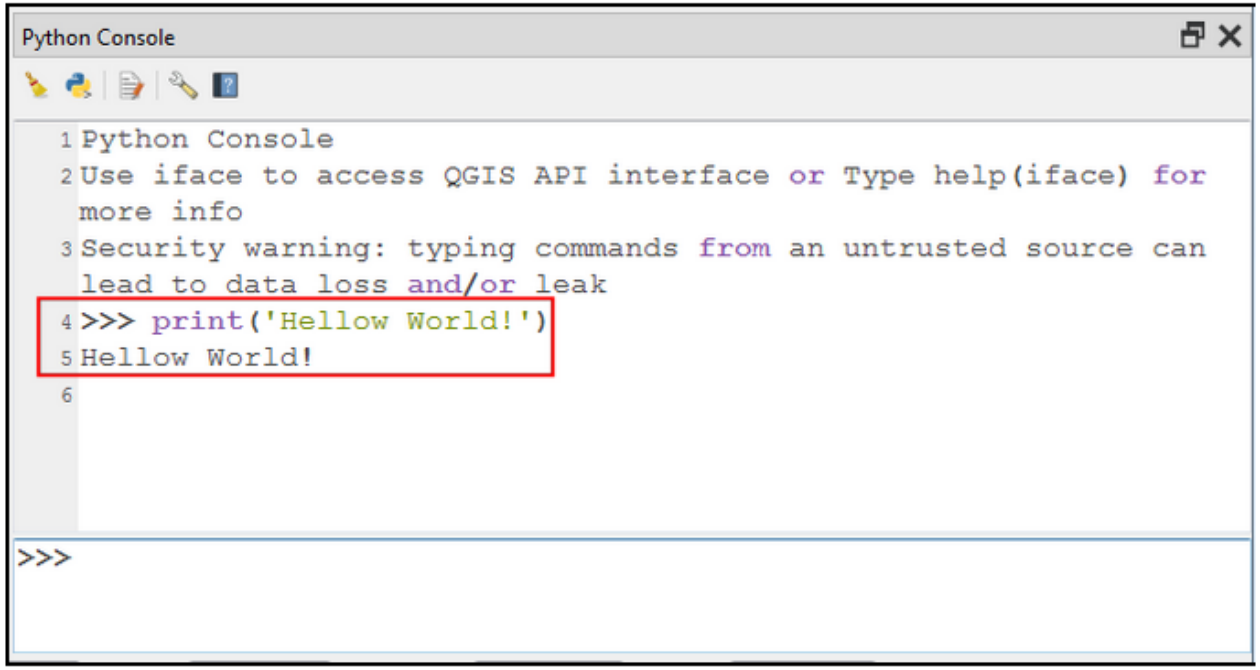
# PYTHON IN QGIS

Scripting in python console

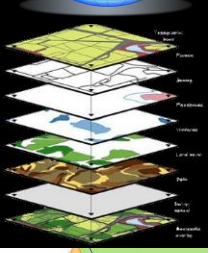
- At the >>> prompt, type in the following command and press Enter.

```
print('Hello World!')
```

- Here you are running Python's print() function with the text 'Hello World'. The output of the statement will be printed below.



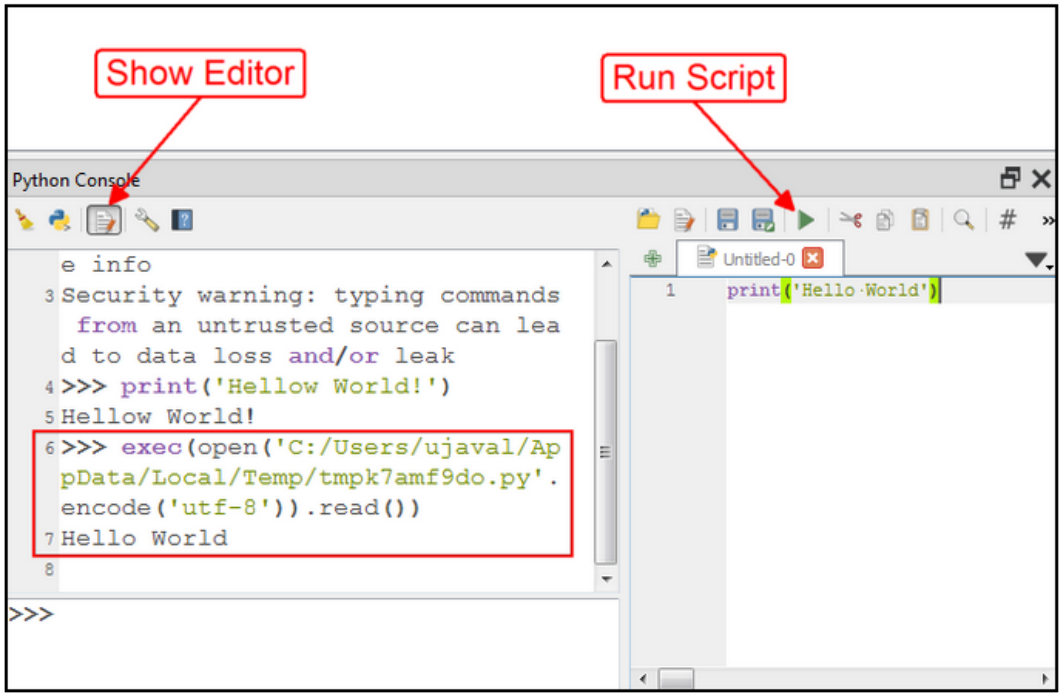


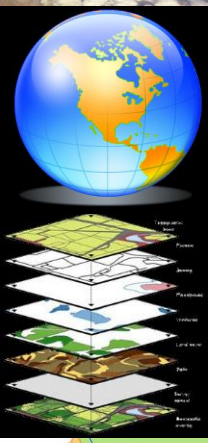


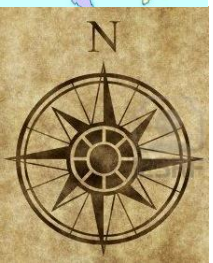


# PYTHON IN QGIS

- While console is useful for typing 1-2 lines of code or printing information contained in a variable, you should use the built-in editor for typing longer scripts or code snippets.
- Click the Show Editor button to open the editor panel. Enter the code and click the Run Script button to execute it. The results will appear in the console as before. If you are working on a longer script, you can also click the Save button in the editor to save the script for future use..







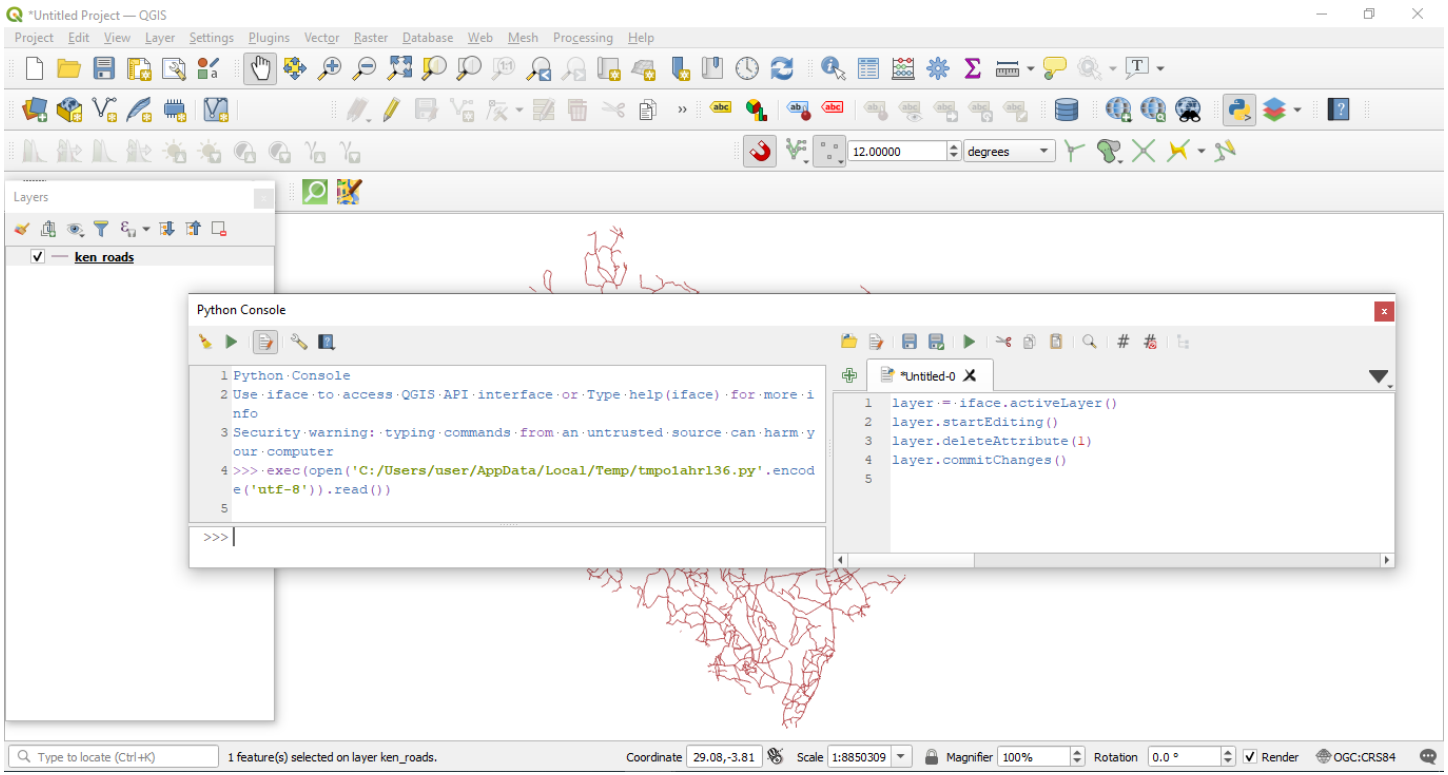

# PYTHON IN QGIS

- QGIS provides a Python API (Application Programming Interface), commonly known as **PyQGIS.**
- The API is vast and very capable. Almost every operation that you can do using QGIS - can be done using the API. This allows developers to write code to build new tools, customize the interface and automate workflows.
- Let’s try out the API to perform some GIS data management tasks. We delete one of the columns of layer loaded on canvas. In this case road.shp
- Write this code in the python consule.

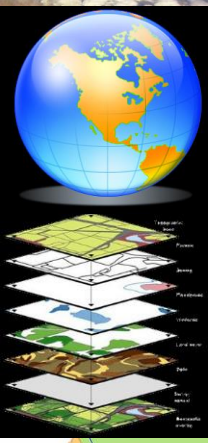
```
layer = iface.activeLayer()
layer.startEditing()
layer.deleteAttribute(1)
layer.commitChanges()
```

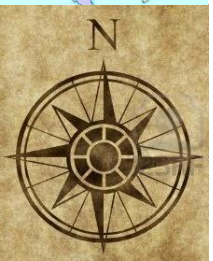
# PYTHON IN QGIS

- You will see that the 2nd column is now deleted from the attribute table.




# PYTHON IN QGIS

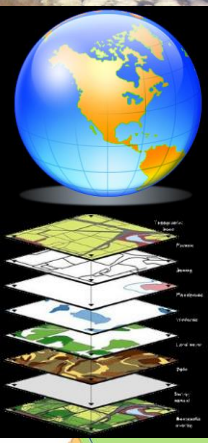


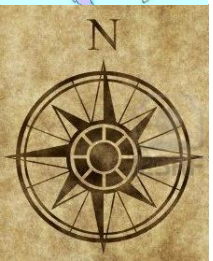




- layer = iface.activeLayer(): This line uses the iface object and runs the activeLayer() method which returns the currently selected layer in QGIS. The method returns the reference to the layer which is saved in the layer variable.
- layer.startEditing(): This is equivalent to putting the layer in the editing mode.
- layer.deleteAttribute(1): The deleteAttribute() is a method from QgsVectorLayer class. It takes the index of the attribute to be deleted. Here we pass on index 1 for the second attribute. (index 0 is the first attribute)
- layer.commitChanges(): This method saves the edit buffer and also disables the editing mode.

- This gives you a preview of the power of the API. To harness the full power of the PyQGIS API.

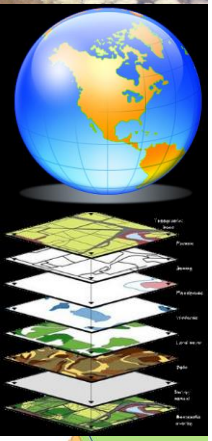

# PYTHON IN QGIS

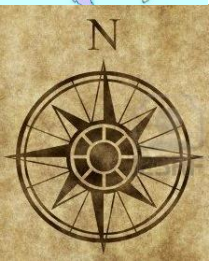



- layer = iface.activeLayer(): This line uses the iface object and runs the activeLayer() method which returns the currently selected layer in QGIS. The method returns the reference to the layer which is saved in the layer variable.
- layer.startEditing(): This is equivalent to putting the layer in the editing mode.
- layer.deleteAttribute(1): The deleteAttribute() is a method from QgsVectorLayer class. It takes the index of the attribute to be deleted. Here we pass on index 1 for the second attribute. (index 0 is the first attribute)
- layer.commitChanges(): This method saves the edit buffer and also disables the editing mode.

- This gives you a preview of the power of the API. To harness the full power of the PyQGIS API.

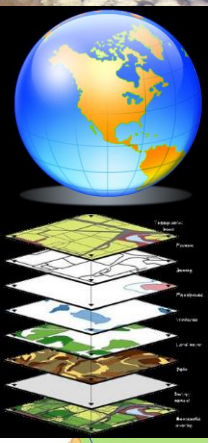


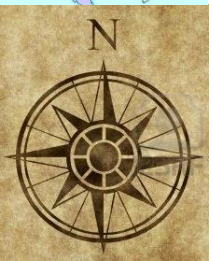




# PYTHON IN QGIS

Running Python code when QGIS starts

- There are two distinct methods to run Python code every time QGIS starts.
    1. Creating a startup.py script: Every time QGIS starts, the user's Python home directory is searched for a file named startup.py. If that file exists, it is executed by the embedded Python interpreter.
    2. Setting the PYQGIS_STARTUP environment variable to an existing Python file: You can run Python code just before QGIS initialization completes by setting the PYQGIS_STARTUP environment variable to the path of an existing Python file. This code will run before QGIS initialization is complete. This method is very useful for cleaning sys.path, which may have undesireable paths, or for isolating/loading the initial environment without requiring a virtual environment

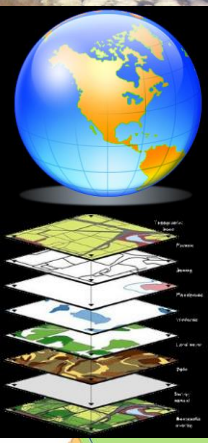


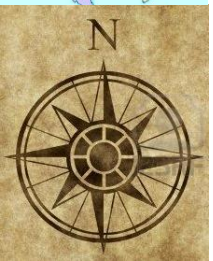




# PYTHON IN QGIS

Python applications

- It is often handy to create scripts for automating processes. With PyQGIS, this is perfectly possible — import the *qgis.core* module, initialize it and you are ready for the processing.
- Or you may want to create an interactive application that uses GIS functionality — perform measurements, export a map as PDF, … The *qgis.gui* module provides various GUI components, most notably the map canvas widget that can be incorporated into the application with support for zooming, panning and/or any further custom map tools.
- PyQGIS custom applications or standalone scripts must be configured to locate the QGIS resources, such as projection information and providers for reading vector and raster layers.
- QGIS Resources are initialized by adding a few lines to the beginning of your application or script. The code to initialize QGIS for custom applications and standalone scripts is similar



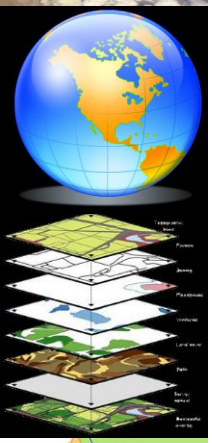



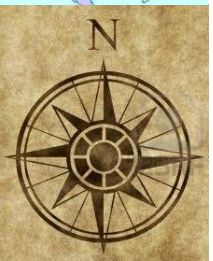

# PYTHON IN QGIS

Python applications

Using PyQGIS in standalone scripts:

- To start a standalone script, initialize the QGIS resources at the beginning of the script:
- First we import the *qgis.core* module and configure the prefix path. The prefix path is the location where QGIS is installed on your system. It is configured in the script by calling the *setPrefixPath()* method. The second argument of *setPrefixPath()* is set to True, specifying that default paths are to be used.
- The QGIS install path varies by platform; the easiest way to find it for your system is to use the Scripting in the Python Console from within QGIS and look at the output from running: *QgsApplication.prefixPath()*
- After the prefix path is configured, we save a reference to *QgsApplication* in the variable *qgs*. The second argument is set to *False*, specifying that we do not plan to use the GUI since we are writing a standalone script.
- With *QgsApplication* configured, we load the QGIS data providers and layer registry by calling the *initQgis()* method.
- With QGIS initialized, we are ready to write the rest of the script. Finally, we wrap up by calling *exitQgis()* to remove the data providers and layer registry from memory



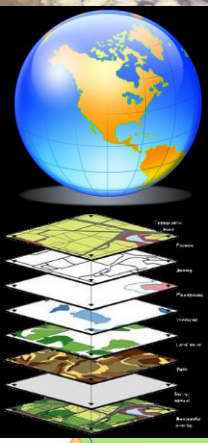



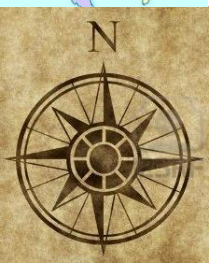

# PYTHON IN QGIS

Python applications

Using PyQGIS in standalone scripts:

- To start a standalone script, initialize the QGIS resources at the beginning of the script:
- First we import the *qgis.core* module and configure the prefix path. The prefix path is the location where QGIS is installed on your system. It is configured in the script by calling the *setPrefixPath()* method. The second argument of *setPrefixPath()* is set to True, specifying that default paths are to be used.
- The QGIS install path varies by platform; the easiest way to find it for your system is to use the Scripting in the Python Console from within QGIS and look at the output from running: *QgsApplication.prefixPath()*
- After the prefix path is configured, we save a reference to *QgsApplication* in the variable *qgs*. The second argument is set to *False*, specifying that we do not plan to use the GUI since we are writing a standalone script.
- With *QgsApplication* configured, we load the QGIS data providers and layer registry by calling the *initQgis()* method.
- With QGIS initialized, we are ready to write the rest of the script. Finally, we wrap up by calling *exitQgis()* to remove the data providers and layer registry from memory



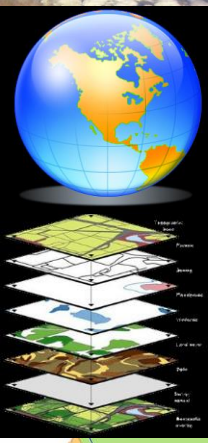


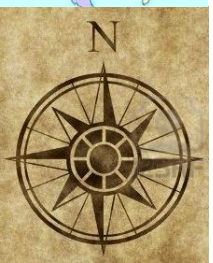

# PYTHON IN QGIS

Python applications

Using PyQGIS in standalone scripts:

```
from qgis.core import *

# Supply path to qgis install location
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# Create a reference to the QgsApplication.  Setting the
# second argument to False disables the GUI.
qgs = QgsApplication([], False)

# Load providers
qgs.initQgis()

# Write your code here to load some layers, use processing
# algorithms, etc.

# Finally, exitQgis() is called to remove the
# provider and layer registries from memory
qgs.exitQgis()
```



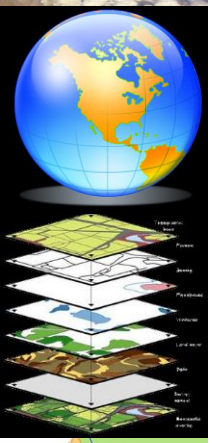

# PYTHON IN QGIS

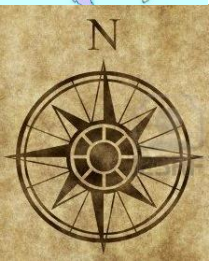
Python applications

Using PyQGIS in custom applications:

- The only difference between Using PyQGIS in standalone scripts and a custom PyQGIS application is the second argument when instantiating the *QgsApplication*. Pass *True* instead of *False* to indicate that we plan to use a GUI

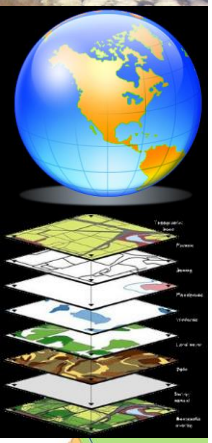
```
from qgis.core import *

# Supply the path to the qgis install location
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# Create a reference to the QgsApplication.
# Setting the second argument to True enables the GUI.  We need
# this since this is a custom application.

qgs = QgsApplication([], True)
```

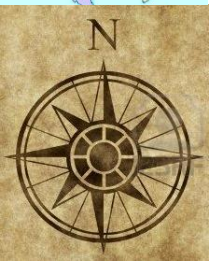



# Classes and objects

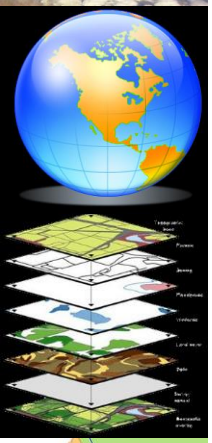
- A class can be thought of as a template.
- It contains certain functions and properties.
- Why Classes?
  - Makes the code ‘modular’. Classes can be re-used and enhanced by other classes.
  - Classes allow us to avoid code duplication
  - Hides the implementation detail from the user of the class
- A new class is defined using the word ***class***. All classes have a function called***__init__(),*** which is always executed when a new object is being created.
- There is also the keyword ***self*** which refers to the current instance of the class.
- The code uses the ***super*** keyword to refer to the parent class.
- Example
  - You have a class called *Car()*
  - A blueprint to build a car with given specifications
  - A class Car() which takes 2 parameters color and type
  - The class has Car() functions that know operate the car: start(), drive(), stop()

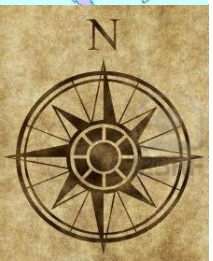



# Classes and objects

- To use a class in a program, you must construct an object from a class. This is called creating an ***instance*** of the class
- To create an instance of a class, you initialize it with some parameters
- A new object is constructed from the template using the supplied parameter.
- A car object with parameters: color=blue and type=automatic
  - my_car = Car('blue', 'automatic')
- You can call class functions using that object
  - my_car.start() to start the car



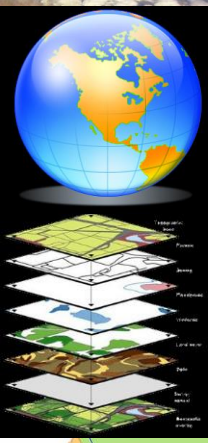

# Classes and objects

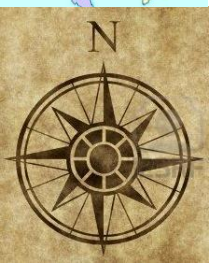
- Defining a python class

```
class Car:

    def __init__(self, color, type):
        self.color = color
        self.type = type
        self.started = False
        self.stopped = False

    def start(self):
        print('Car Started')
        self.started = True
        self.stopped = False
```

- Instance is an object of a class
- You can create an instance of a class by calling the constructor

```
my_car1 = Car('blue', 'automatic')

my_car2 = Car('red', 'manual')
```





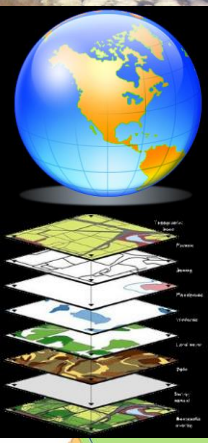

# Classes and objects

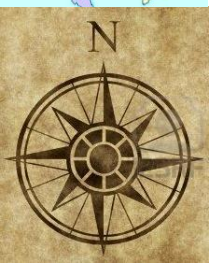
Method

- Classes have functions
- When you call a function from an object it is called a ***method***
- Methods are passed on the reference to the current object using the *self* keyword
  - if you see a function with the first parameter as *self*, it needs to be called on an object

Using a method:

```
class Car:

    def __init__(self, color, type):
        self.color = color
        self.type = type
        self.started = False
        self.stopped = False

    def start(self):
        print('Car Started')
        self.started = True
        self.stopped = False



my_car = Car('blue', 'automatic')

my_car.start()
```





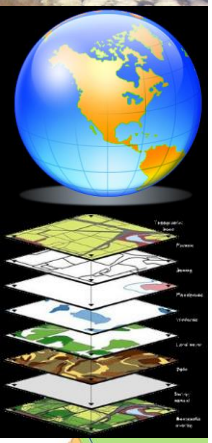

# Classes and objects

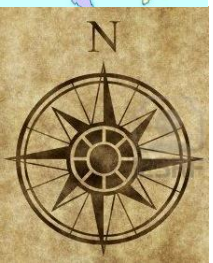
Method

- Classes have functions
- When you call a function from an object it is called a ***method***
- Methods are passed on the reference to the current object using the *self* keyword
  - if you see a function with the first parameter as *self*, it needs to be called on an object

Using a method:

```
class Car:

    def __init__(self, color, type):
        self.color = color
        self.type = type
        self.started = False
        self.stopped = False

    def start(self):
        print('Car Started')
        self.started = True
        self.stopped = False



my_car = Car('blue', 'automatic')

my_car.start()
```



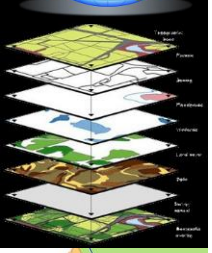

# Classes and objects

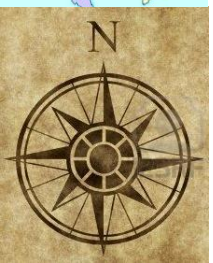
Attributes

- Class can have variables
- Class variables are called attributes
- Instance Attributes
  - They are associated with a particular object
  - Every object can have a different value
  - Defined inside the __init__() constructor

```
class Car:

    def __init__(self, color, type):
        self.color = color
        self.type = type
        self.started = False
        self.stopped = False


my_new_car = Car('blue', 'automatic')
my_old_car = Car('red', 'manual')

print(my_new_car.color)
print(my_old_car.color)
```





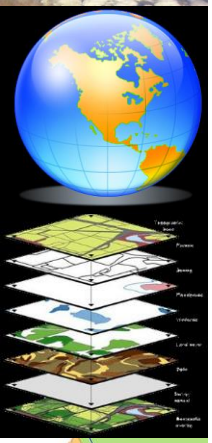



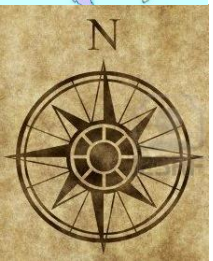

# Classes and objects

Attributes

- Class Attributes
  - They are associated with a class
  - Every objects will have the same value
  - Defined outside of the __init__() constructor
  - Can be accessed from a class

```
class Car:

    model = 'Civic'

    def __init__(self, color, type):
        self.color = color
        self.type = type
        self.started = False
        self.stopped = False


my_new_car = Car('blue', 'automatic')
my_old_car = Car('red', 'manual')

print(my_new_car.model)
print(my_old_car.model)

print(Car.model)
```






# Classes and objects

Inheritance

- Classes can be derived from another class.
- The derived class ‘inherits’ all features of the base class.
- This allows you to build a hierarchy of classes where classes get more and more specialized without having to implement all the features.

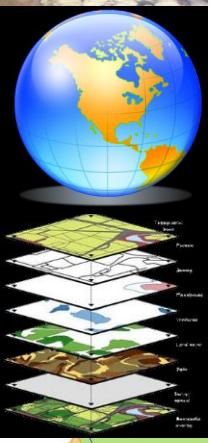
```
class Car:

    def __init__(self, color, type):
        self.color = color
        self.type = type
        self.started = False
        self.stopped = False

    def start(self):
        print('Car Started')
        self.started = True
        self.stopped = False
```

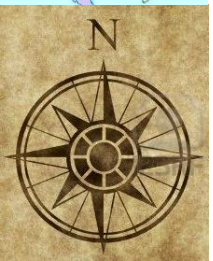
```
class Sedan(Car):

    def __init__(self, color, type, seats):
        super().__init__(color, type)
        self.seats = seats
```

```
class ElectricSedan(Sedan):

    def __init__(self, color, type, seats,
                 range_km):
        super().__init__(color, type, seats)
        self.range_km = range_km
```

```
my_car = Sedan('blue', 'automatic', 5)
print(my_car.color)
print(my_car.seats)
my_car.start()
```

```
my_future_car = ElectricSedan(
        'red', 'automatic', 5, 500)
print(my_future_car.color)
print(my_future_car.seats)
print(my_future_car.range_km)
my_future_car.start()
```

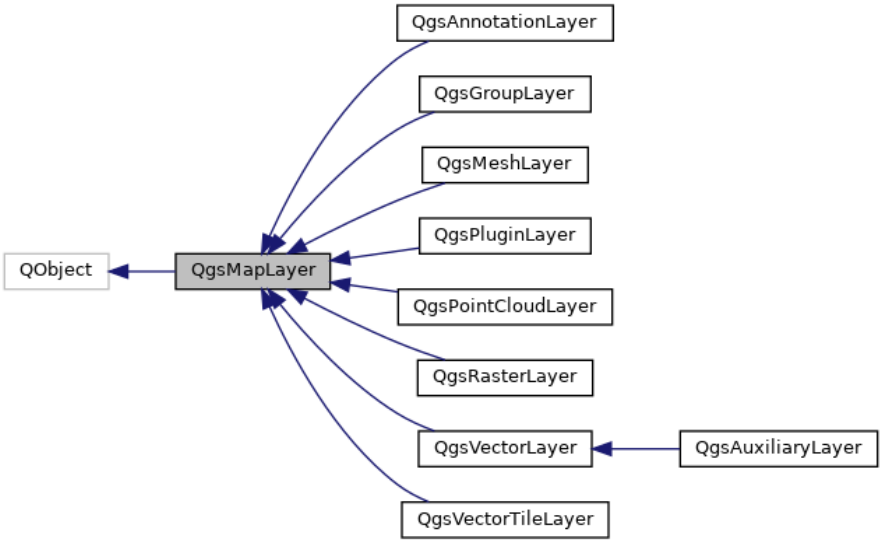


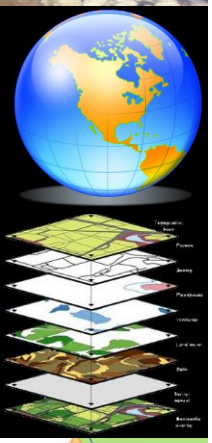


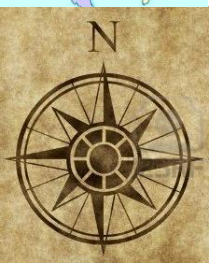
# PyQGIS

Inheritance in PyQGIS

- All PyQGIS classes are derived from the Base Class called QObject()
- Example:
  - QgsMapLayer() is the Base class for all map layer types.
    - QgsRasterLayer() is derived from QgsMapLayer()
    - QgsPointCloudLayer() is derived from QgsMapLayer()
    - QgsVectorLayer() is derived from QgsMapLayer()
      - QgsAuxiliaryLayer() is derived from QgsVectorLayer()






# PyQGIS

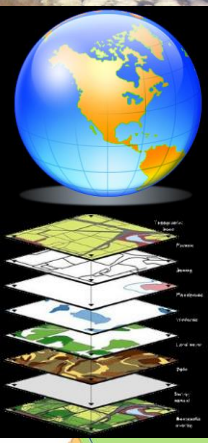
Calculating distance using PyQGIS

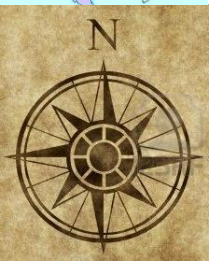
- Example: A basic but important operation in a GIS is the calculation of distance and areas. We will see how you can use PyQGIS APIs to compute distances.
- We will compute distance between the following 2 coordinates for Nairobi and Mombasa cities.
- QGIS provides the class *QgsDistanceArea* that has methods to compute distances and areas. To use this class, we must create an object by instantiating it.
- Class documentation: https://qgis.org/pyqgis/master/core/QgsDistanceArea.html
- the class constructor doesn't take any arguments. We can use the default constructor to create an object and assign it to the *d variable*.

```
d = QgsDistanceArea()
```

- all calculations will be performed using ellipsoidal algorithms. As we want to compute the distance between a pair of latitude/longitudes, we need to use the ellipsoidal algorithms. Let's set the ellipsoid WGS84 for our calculation. We can use the method *setEllipsoid().* Remember, methods should be applied on objects, and not classes directly.

```
d.setEllipsoid('WGS84')
```



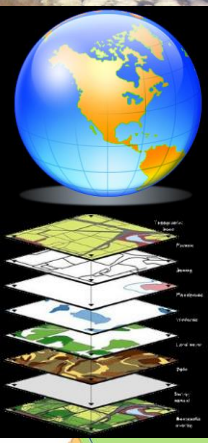


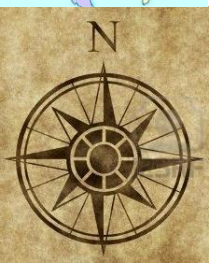

# PyQGIS

Calculating distance using PyQGIS

- Now our object d is capable of performing ellipsoidal distance computations. Browsing through the available methods in the *QgsDistanceArea* class, we can see a *measureLine()* method. This method takes a list *QgsPointXY* objects. We can create these objects from our coordinate pairs and pass them on to the *measureLine()* method to get the distance. The output will be in meters. We divide it by 1000 to convert it to kilometers.

```
lat1, lon1 = ken_nairobi
lat2, lon2 = ken_mombasa
# Remember the order is X,Y
point1 = QgsPointXY(lon1, lat1)
point2 = QgsPointXY(lon2, lat2)

distance = d.measureLine([point1, point2])
print(distance/1000)
```



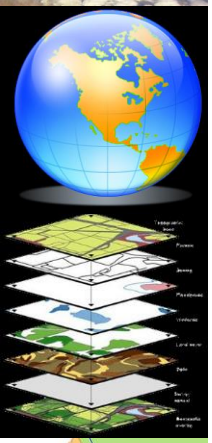


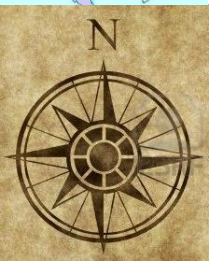

# PyQGIS

Calculating distance using PyQGIS

- Putting it all together, below is the complete code to calculate the distance between a pair of coordinates using PyQGIS. When you run the code in the Python Console of QGIS, all the PyQGIS classes are already imported. If you are running this code from a script or a plugin, you must explicitly import the *QgsDistanceArea* class.

```
from qgis.core import QgsDistanceArea
ken_nairobi = (36.826, -1.303)
ken_mombasa = (39.667, -4.05052)
d = QgsDistanceArea()
d.setEllipsoid('WGS84')
lat1, lon1 = ken_nairobi
lat2, lon2 = ken_mombasa
 # Remember the order is X,Y
point1 = QgsPointXY(lon1, lat1)
point2 = QgsPointXY(lon2, lat2)
distance = d.measureLine([point1, point2])
print(distance/1000)
```

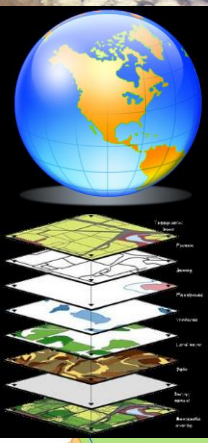


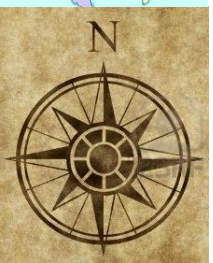




# PyQGIS

Calculating distance using PyQGIS

- Assignment: Let’s say you want to make a stop at Machakos on the way from Nairobi to Mombasa .Calculate the total ellipsoidal distance considering a stop at Machakos.






# PyQGIS

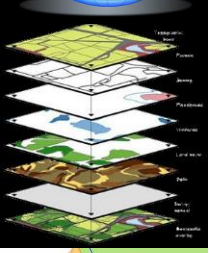
QGIS Interface API (QgisInterface)

- *QgisInterface* class - which provides methods for interaction with the QGIS environment.
- When QGIS is running, a variable called *iface* is set up to provide an object of the class *QgisInterface* to interact with the running QGIS environment.
- This interface allows access to the map canvas, menus, toolbars and other parts of the QGIS application. Python Console and Plugins can use *iface* to access various parts of the QGIS interface.
- Change Title of QGIS Main Window:

```
title = iface.mainWindow().windowTitle()
new_title = title.replace('QGIS', 'My QGIS')
iface.mainWindow().setWindowTitle(new_title)
```

- Remove Raster and Vector Menus

```
vector_menu = iface.vectorMenu()
raster_menu = iface.rasterMenu()
menubar = vector_menu.parentWidget()
menubar.removeAction(vector_menu.menuAction())
menubar.removeAction(raster_menu.menuAction())
```

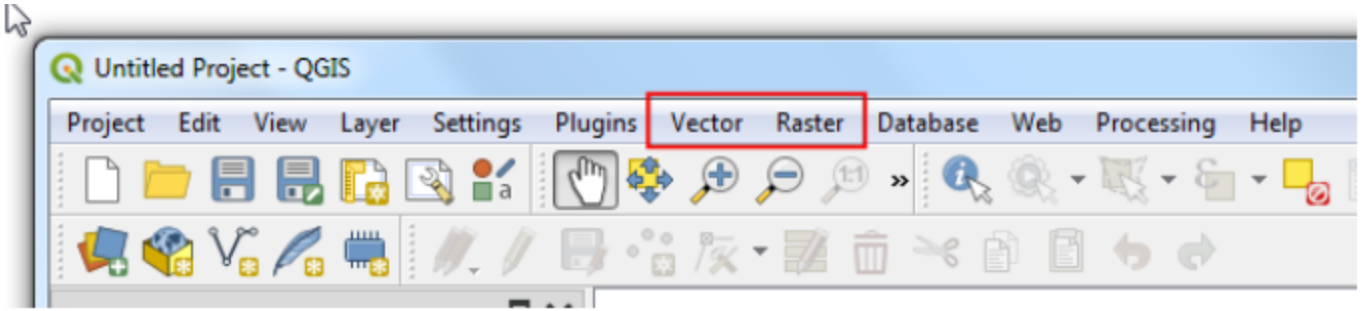



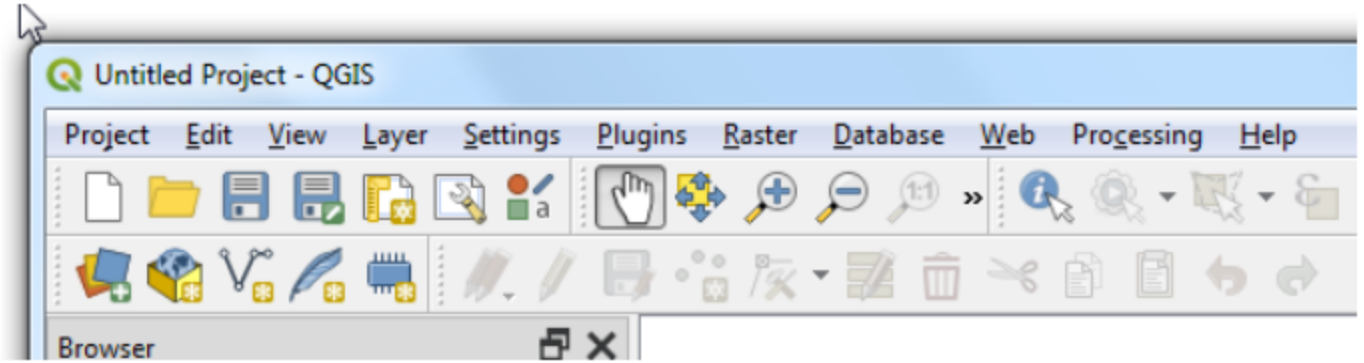

# PyQGIS

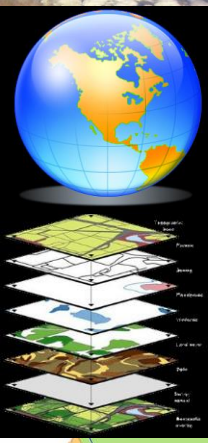
QGIS Interface API (QgisInterface)

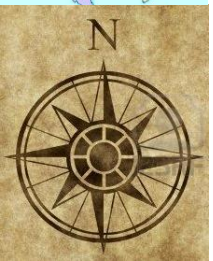
- Remove Raster and Vector Menus

```
vector_menu = iface.vectorMenu()
raster_menu = iface.rasterMenu()
menubar = vector_menu.parentWidget()
menubar.removeAction(vector_menu.menuAction())
menubar.removeAction(raster_menu.menuAction())
```





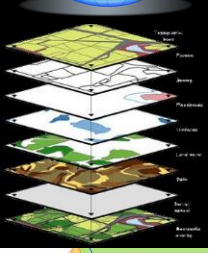




# PyQGIS

QGIS Interface API (QgisInterface)

- Add A New Menu Item
- A new button or menu item is created using *QAction().* Here we create an action and then connect the click signal to a method that opens a website.

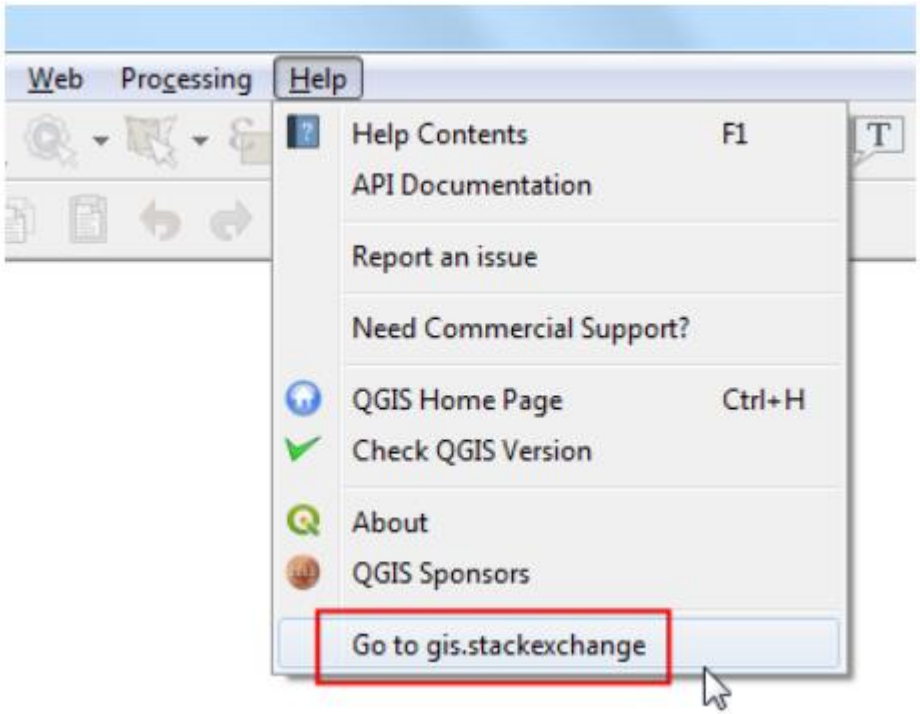
```
import webbrowser


def open_website():
    webbrowser.open('https://gis.stackexchange.com')


website_action = QAction('Go to gis.stackexchange')
website_action.triggered.connect(open_website)
iface.helpMenu().addSeparator()
iface.helpMenu().addAction(website_action)
```

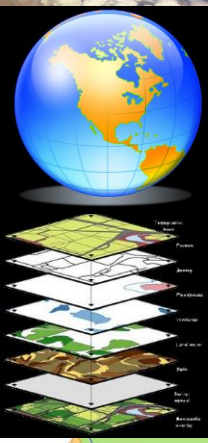



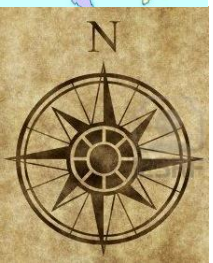

# PyQGIS

QGIS Interface API (QgisInterface)

- Add A New Menu Item



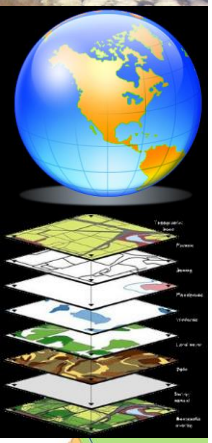



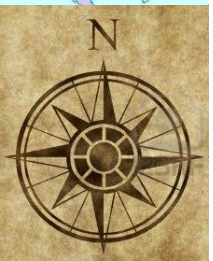

# PyQGIS

QGIS Interface API (QgisInterface)

- Add a button to a toolbar

```
import os
from datetime import datetime

icon = 'question.svg'
data_dir = os.path.join(os.path.expanduser('~'), 'Downloads', 'pyqgis_masterclass')
icon_path = os.path.join(data_dir, icon)

def show_time():
    now = datetime.now()
    current_time = now.strftime("%H:%M:%S")
    iface.messageBar().pushInfo('Current Time', current_time)

action = QAction('Show Time')
action.triggered.connect(show_time)
action.setIcon(QIcon(icon_path))
iface.addToolBarIcon(action)
```

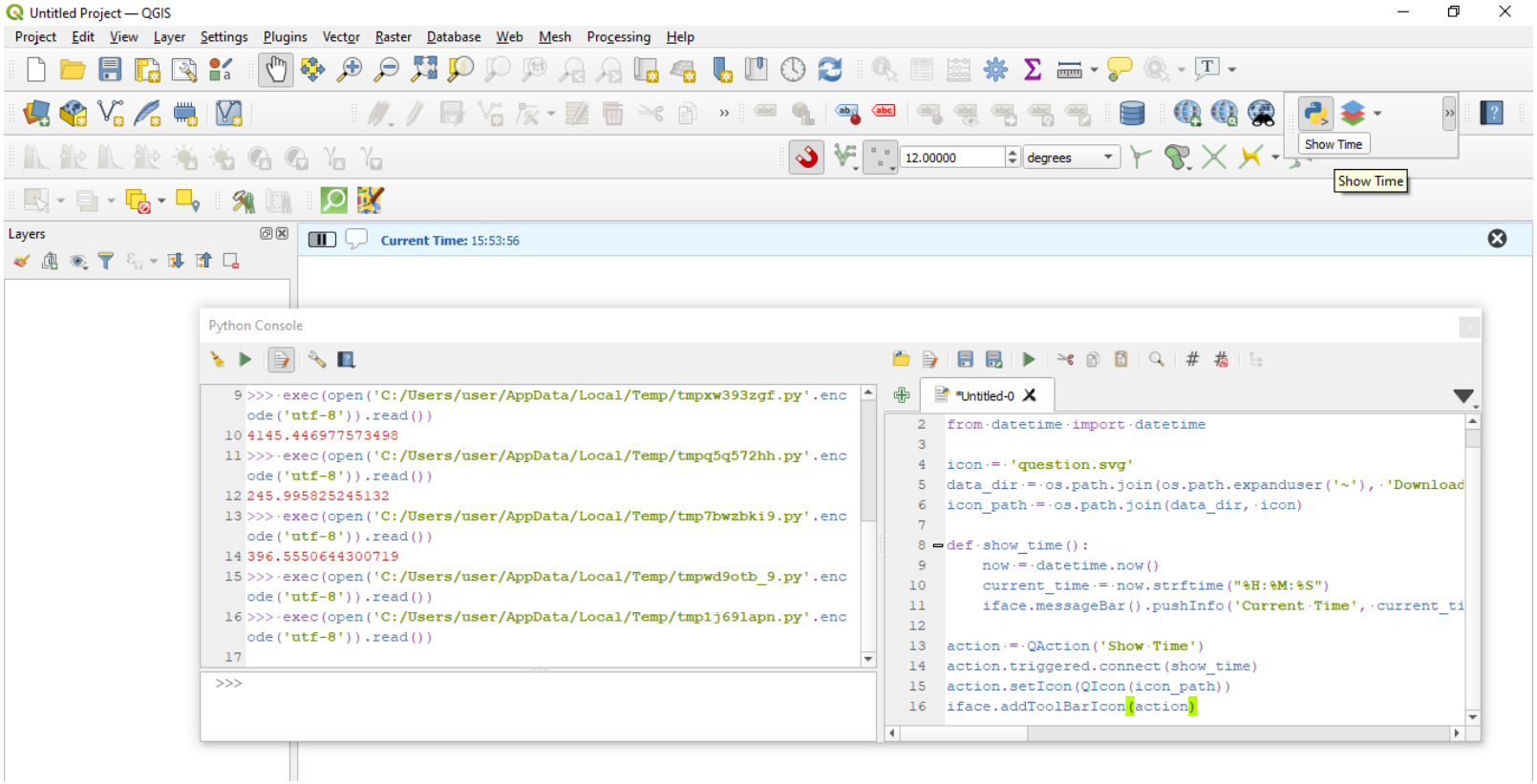


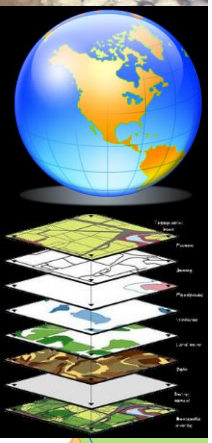



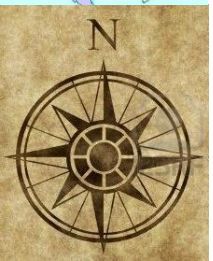

# PyQGIS

## QGIS Interface API (QgisInterface)

- Add a button to a toolbar






# PyQGIS

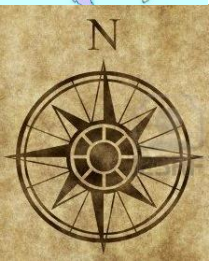
QGIS Interface API (QgisInterface)

- Add New Layers:

```
roads="C:/JKUAT/Lectures_notes/GIS programming/python demo/data/ken_roads/ken_roads.shp"

layer = iface.addVectorLayer(roads,"roads layer","ogr")

if not layer:

    print("layer failed to load")
```

- This creates a new layer and adds it to the current QGIS project (making it appear in the layer list) in one step. The function returns the layer instance or None if the layer couldn't be loaded.
- Data sources are identified by an URI (Uniform Resource Identifier) - For files on computer the URI is the file path - For databases, the URI is constructed using the *QgsDataSourceUri* class and encodes, the database path, table, username, password etc. - For web layers, such as WMF/WFS etc, the URI is the web URL




# PyQGIS

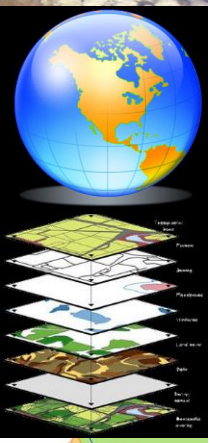
QGIS Interface API (QgisInterface)

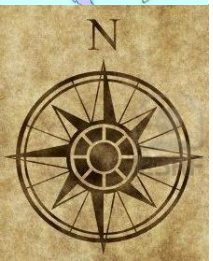
Add New Layers from other sources:

- PostGIS database - data source is a string with all information needed to create a connection to PostgreSQL database.
- *QgsDataSourceUri* class can generate this string for you. Note that QGIS has to be compiled with Postgres support, otherwise this provider isn't available:

```
uri = QgsDataSourceUri()
# set host name, port, database name, username and password
uri.setConnection("localhost", "5432", "dbname", "johny", "xxx")
# set database schema, table name, geometry column and optionally
# subset (WHERE clause)
uri.setDataSource("public", "roads", "the_geom", "cityid = 2643", "primary_key_
↪field")

vlayer = QgsVectorLayer(uri.uri(False), "layer name you like", "postgres")
```

- Note: The False argument passed to uri.uri(False) prevents the expansion of the authentication configuration parameters, if you are not using any authentication configuration this argument does not make any difference.



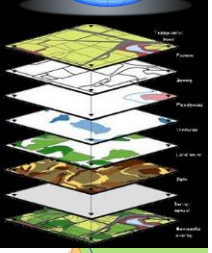

# PyQGIS

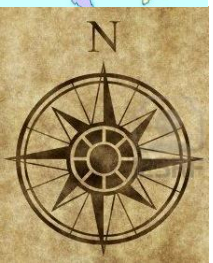
QGIS Interface API (QgisInterface)

Add New Layers from other sources:

- CSV or other delimited text files — to open a file with a semicolon as a delimiter, with field “x” for X coordinate and field “y” for Y coordinate you would use something like this:

```
uri = "file://{}/testdata/delimited_xy.csv?delimiter={}&xField={}&yField={}".
↪format(os.getcwd(), ";", "x", "y")
vlayer = QgsVectorLayer(uri, "layer name you like", "delimitedtext")
QgsProject.instance().addMapLayer(vlayer)
```





# PyQGIS

QGIS Interface API (QgisInterface)

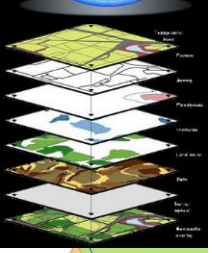
Add New Layers from other sources:

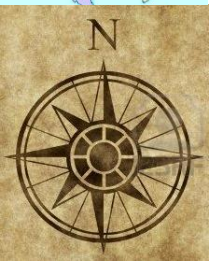
- Raster Layers: For accessing raster files, GDAL library is used. It supports a wide range of file formats. To load a raster from a file, specify its filename and display name. Similarly to vector layers, raster layers can be loaded using the *addRasterLayer* function of the *QgisInterface* object.

```
iface.addRasterLayer(path_to_tif, "layer name you like")
```

```
# get the path to a tif file  e.g. /home/project/data/srtm.tif
path_to_tif = "qgis-projects/python_cookbook/data/srtm.tif"
rlayer = QgsRasterLayer(path_to_tif, "SRTM layer name")
if not rlayer.isValid():
    print("Layer failed to load!")
```

- To load a PostGIS raster: PostGIS rasters, similar to PostGIS vectors, can be added to a project using a URI string.





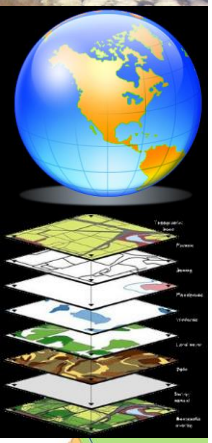

# Classes and objects

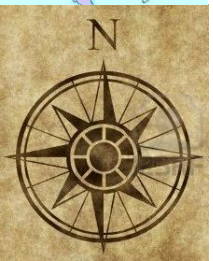
QGIS Interface API (QgisInterface)

Add New Layers from other sources:

- Raster layers can also be created from a WCS service:

```
layer_name = 'modis'
url = "https://demo.mapserver.org/cgi-bin/wcs?identifier={}".format(layer_name)
rlayer = QgsRasterLayer(uri, 'my wcs layer', 'wcs')
```



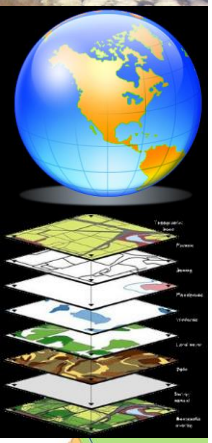

# Classes and objects

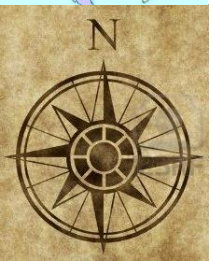
Retrieving information about attributes:

- You can retrieve information about the fields associated with a vector layer by calling fields() on a *QgsVectorLayer* object:

```
roads="C:/JKUAT/Lectures_notes/GIS programming/python demo/data/ken_roads/ken_roads.shp"

layer = iface.addVectorLayer(roads,"roads layer","ogr")

for field in layer.fields():

   print(field.name(),field.typeName())
```

```
vlayer = QgsVectorLayer("testdata/airports.shp", "airports", "ogr")
for field in vlayer.fields():
    print(field.name(), field.typeName())
```


# Classes and objects

Assignment:

- Write a script that loads a vector and raster layer from your datasets and retrieves attribute information about the layers.
- Enhance the script so as to control the symbology of the layers and annotation of labels.