GG3209 - Spatial Analysis with GIS

Technical Practices using Python

Fernando Benitez-Paez

2025-07-03

Table of contents

About this site

This book has been developed as part of the module **GG3209 – Spatial Analysis with GIS** at the School of Geography and Sustainable Development, University of St Andrews. This module is spitted in two parts, the first 4 weeks includes QGIS and Multi-criteria evaluation, the second part related to the use of Python for the use and analysis of spatial data.

This part will establish a comprehensive introduction to **Python** (an easy-to-learn and powerful programming language) and its use for manipulating spatial data and deploying spatial analysis models. Python has been cataloged as one of the most popular programming technologies and is widely used as a scripting language in the GIScience world. If you have signed to this module you will learn how to use Python in multiple environments, more specifically using a popular tool called Jupyter Notebooks, then learn how to manipulate vector and raster data and finish by integrating spatial modelling using coding environments and clustering methods as helpful methodologies for dissertations.

This module include Lecture+Lab sessions. Lectures will be delivered first, followed by lab practices included in this site. Students will be expected to work on these during the lab sessions but may also have to continue their own for the rest of the week, since the scheduled time may not be sufficient to finish everything.

All students enrolled in GG3209 have access to the University's ArcGIS Online **Organisational Account** using their University credentials. We will use this platform in some of the Labs. If you havent experimenting and working with this platform before, to get started, please log in at: https://uostandrews.maps.arcgis.com

Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

```
1 + 1
```

```
[1] 2
```

Python Environment Installation

This guide provides step-by-step instructions for setting up the Python environment for second part of the **module GG3209** on Windows and macOS. The environment includes all necessary libraries for data handling, clustering, visualization, large datasets, and hotspot analysis.

Estimated time to install: 20 to 30 minutes.

Make sure you you have an stable internet connection. If you are using eduroam at the university, you might experience delay or issues when the internet connection isn't stable, if that is the case make sure you have a stable connection at home to complete this process properly. In case the problem persist please a case and request and appointment with our IT office at gsditsupport@st-andrews.ac.uk

Table of Contents

Prerequisites

Before starting, ensure you have:

- ☐ A stable internet connection (large downloads required)
- ☐ At least 5GB of free disk space
- ☐ Administrator privileges on your computer

☐ Basic familiarity with command line interface (we will have a show demo about this during the lectures)

If you don't know how to validate this, please make an appointment with out IT service who can support you with this gsditsupport@st-andrews.ac.uk

---

Installing Miniconda (Required for All Students)

Why Miniconda? Miniconda provides a clean, minimal Python installation with only essential packages. This ensures all students start with identical environments and reduces potential conflicts. It's lightweight, fast, and gives us complete control over installed packages.

Windows Installation

1. **Download Miniconda**:
   - Visit https://docs.conda.io/en/latest/miniconda.html
   - Download **"Miniconda3 Windows 64-bit"** (approximately 50MB)
   - **Important**: Download the Python 3.10 version specifically
2. **Install Miniconda**:
   - Double-click the downloaded `.exe` file
   - Click "Next" through the installation wizard
   - **CRITICAL**: When asked about PATH, select **"Add Miniconda3 to my PATH environment variable"**
   - Accept all other default settings
   - Complete installation (takes 2-3 minutes)
3. **Verify Installation**:
   - Open **Command Prompt** (Press `Win + R`, type `cmd`, press Enter)
   - Type `conda --version` and press Enter
   - You should see: `conda 23.x.x` (or similar version number)
   - Type `python --version` and press Enter
   - You should see: `Python 3.10.x`

macOS Installation

1. **Download Miniconda**:
   - Visit https://docs.conda.io/en/latest/miniconda.html
   - **For Apple Silicon Macs (M1/M2)**: Download "Miniconda3 macOS Apple M1 64-bit pkg"
   - **For Intel Macs**: Download "Miniconda3 macOS Intel x86 64-bit pkg"
   - **Important**: Download the Python 3.10 version specifically
2. **Install Miniconda**:
   - Double-click the downloaded `.pkg` file
   - Follow the installation wizard
   - Accept all default settings
   - Complete installation
3. **Verify Installation**:
   - Open **Terminal** (Press `Cmd + Space`, type "Terminal", press Enter)
   - Type `conda --version` and press Enter
   - You should see: `conda 23.x.x` (or similar version number)
   - Type `python --version` and press Enter
   - You should see: `Python 3.10.x`

Post-Installation Setup (All Students)

After successful installation, run these commands to ensure consistency:

```
# Update conda to latest version
conda update -n base -c defaults conda

# Configure conda for optimal performance
conda config --set auto_activate_base false
conda config --add channels conda-forge
conda config --set channel_priority strict

# Verify configuration
conda info
```

Expected Output: You should see conda-forge listed as a channel with highest priority.

Creating the Environment

Step 1: Download the Environment File

Save the environment configuration as `environment.yml` in a folder of your choice (e.g., `Documents/gg3209/`).

Step 2: Open Command Line Interface

All students must use the correct command line interface to ensure consistency.

Windows Students

- Press `Win + R`, type `cmd`, press Enter
- **Alternative**: Search for "Command Prompt" in Start Menu
- **Important**: Use Command Prompt, NOT PowerShell or other terminals

macOS Students

- Press `Cmd + Space`, type "Terminal", press Enter
- **Alternative**: Go to Applications > Utilities > Terminal
- **Important**: Use Terminal, NOT other command line apps

Step 3: Navigate to Your Project Directory

All students should create the same folder structure:

```
# Windows students
mkdir C:\gg3209 #sds stands for spatial dtata science
cd C:\gg3209

# macOS students
mkdir ~/gg3209
cd ~/gg3209
```

Step 4: Download and Verify Environment File

Before creating the environment, ensure you have the correct file:

1. Save the `environment.yml` file in your project directory. This file is located in Moodle, go there and download it and place it in the project directory you created earlier.
2. Verify the file exists:

```
# All students run this command
dir environment.yml    # Windows
ls environment.yml     # macOS
```

You should see the file listed. If not, ensure you saved it correctly.

Step 5: Create the Environment (Critical Step)

This is where consistency matters most. All students run the exact same commands:

```
# Create environment from file
conda env create -f environment.yml

# This will take 15-30 minutes
# You will see many packages being downloaded and installed
# Wait for "done" message before proceeding
```

Expected Output:

```
Collecting package metadata (repodata.json): done
Solving environment: done
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate spatial-data-science
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

Step 6: Activate the Environment

All students must activate the environment before using it – THIS IS VERY IMPORTANT:

```
# Activate the environment
conda activate gg3209
```

Success Indicator: Your command prompt should now show `(gg3209)` at the beginning:

```
# Windows example
(gg3209) C:\gg3209>

# macOS example
(gg3209) username@computer:~/gg3209$
```

Step 7: Final Verification

All students run the same verification script:

Once you have created your python environment, now it is important you test that everything is properly installed. Using the same `terminal` or `command prompt` window run the following command.

```
# Make sure your environment is activated
conda activate gg3209

# Run the test
python test_installation.py

------
# Windows example
(gg3209) C:\gg3209\> python verification_script.py

# macOS example
(gg3209) username@computer:~/gg3209/$ python verification_script.py
```

Expected Output for All Students:

```
Python version: 3.10.x
Python location: [path to conda environment]
  All required libraries imported successfully!
```

```
Environment setup is complete and consistent!
GeoPandas version: 1.1.1
Pandas version: 2.3.1
NumPy version: 1.26.4
```

If successful, you should see version numbers and check-marks.

Running Jupyter Lab

Starting Jupyter Lab

```
# Make sure environment is activated (optional if you know you have activated it)
conda activate gg3209

# Make sure you are in your project directory e.g. GG3209

cd GG3209

# Launch Jupyter Lab
jupyter lab
```

This will:

- Start the Jupyter server

- Open your default web browser

- Display the Jupyter Lab interface

Creating Your First Notebook

1. Click “Python 3 (ipykernel)” under “Notebook”
2. Test with a simple spatial analysis:

```
import geopandas as gpd
import matplotlib.pyplot as plt
import geodatasets

# Create a simple test
```

```
world = gpd.read_file(geodatasets.get_path("naturalearth.land"))
world.plot(figsize=(10, 6))
plt.title('World Map Test')
plt.show()
```

Stopping Jupyter Lab

- In your browser: File > Shut Down
- In command line: Press `Ctrl + C` (Windows) or `Cmd + C` (Mac)

GG3209-Specific Setup Instructions

First Day of Class Checklist

All students must complete before first lab:

- ☐ **Install Miniconda** (Python 3.10)
- ☐ **Create spatial-data-science environment**
- ☐ **Activate environment successfully**
- ☐ **Run verification script** (must pass)
- ☐ **Start Jupyter Lab** (must open in browser)
- ☐ **Create test notebook** with basic spatial analysis

Weekly Environment Check

Run this command weekly to ensure consistency:

```
python -c "
import geopandas as gpd
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import folium
from esda.getisord import G_Local
```

```
from sklearn.cluster import DBSCAN
print('  Environment is working correctly')
print(f'  GeoPandas version: {gpd.__version__}')
print(f'  Pandas version: {pd.__version__}')
"
```

Submission Requirements

For all assignments, include this environment information:

```
# Add this cell at the top of every notebook
import sys
import geopandas as gpd
import pandas as pd

print(f"Python version: {sys.version}")
print(f"GeoPandas version: {gpd.__version__}")
print(f"Pandas version: {pd.__version__}")
print(f"Environment: gg3209")
```

This ensures all students are working with identical software versions and helps instructors debug issues consistently.

Next Steps

Once your environment is set up:

1. **Complete the verification tests**
2. **Try the sample notebook**
3. **Explore the example datasets**
4. **Begin your spatial analysis project**

Your spatial data science environment for the course GG3209 is now ready for the first Labs, this part as complicated as it seems now will allow you to create and practice all your python skills.

Troubleshooting Guide

With Python and in particular with many new ways to use your computer, create folder and scripts things can get messy and very confusing, this is very normal and it is part of the process. The first thing is do not get frustrated or trying all sort of things without understanding what is the root of the issue. S you need to think systematically and then find out the solution. Here I have listed a few potential and common issues you might find. So first take all of this before escalating your issue to the lecturer or the IT support team.

Before Seeking Help

All students should complete this checklist:

- ☐ Miniconda is installed and `conda --version` works
- ☐ Environment was created without errors
- ☐ Environment shows `(gg3209)` when activated
- ☐ Verification script runs successfully
- ☐ Jupyter Lab starts without errors

Class Support Protocol

1. **First**: Check this troubleshooting guide
2. **Second**: Ask a classmate (compare outputs)
3. **Third**: Post your issue in the MS Teams channel of this course:
    - Your operating system (Windows/macOS)
    - Exact error message
    - Commands you ran
    - Output from verification script

Standardized Error Reporting

When reporting problems, always include:

```
# Run these commands and include output
conda info
conda list geopandas
python --version
jupyter --version
```

Issue 1: "conda: command not found"

This is the most common issue for beginners.

Windows Solution:

```
# Option 1: Use the correct command prompt
# Search for "Anaconda Prompt" in Start Menu if available
# Or reinstall Miniconda ensuring PATH is added

# Option 2: Manually add to PATH
set PATH=%PATH%;C:\Users\%USERNAME%\miniconda3\Scripts
set PATH=%PATH%;C:\Users\%USERNAME%\miniconda3
```

macOS Solution:

```
# Add to PATH temporarily
export PATH="$HOME/miniconda3/bin:$PATH"

# Add to PATH permanently
echo 'export PATH="$HOME/miniconda3/bin:$PATH"' >> ~/.bash_profile
source ~/.bash_profile

# For zsh users (macOS Catalina and later)
echo 'export PATH="$HOME/miniconda3/bin:$PATH"' >> ~/.zshrc
source ~/.zshrc
```

Issue 2: Environment Creation Fails

Common causes and solutions:

```
# Solution 1: Clean conda cache
conda clean --all

# Solution 2: Update conda first
conda update conda

# Solution 3: Try creating environment with explicit solver
conda env create -f environment.yml --solver=classic

# Solution 4: Check internet connection and try again
```

```
# Large downloads may timeout on slow connections
```

Issue 3: Different Python Versions

All students must have Python 3.10 for consistency.

```
# Check your Python version
python --version

# If incorrect, remove environment and recreate
conda env remove --name gg3209
conda env create -f environment.yml
```

Issue 4: Package Conflicts During Installation

This indicates environment file issues:

```
# Solution: Use mamba for faster, more reliable solving
conda install mamba -n base -c conda-forge
mamba env create -f environment.yml
```

Issue 5: Jupyter Lab Won't Start

Consistency check:

```
# Ensure environment is activated
conda activate gg3209

# Verify Jupyter installation
jupyter --version

# If missing, reinstall
conda install jupyter jupyterlab -c conda-forge

# Start Jupyter Lab
jupyter lab
```

Issue 6: Import Errors Despite Successful Installation

Environment activation problem:

```
# Always activate environment first
conda activate gg3209

# Check which Python you're using
which python     # macOS/Linux
where python     # Windows

# Should show path to conda environment, not system Python
```

Issue 7: PDF Generation Not Working

LaTeX installation issues:

```
# Verify LaTeX installation
pdflatex --version

# If missing, install manually:
# Windows: Download MiKTeX from https://miktex.org/
# macOS: Install MacTeX from https://tug.org/mactex/

# Test PDF conversion
jupyter nbconvert --to pdf test_notebook.ipynb
```

Emergency Reinstallation

If all else fails, complete clean installation:

```
# Remove environment
conda env remove --name gg3209

# Clean all caches
conda clean --all

# Recreate environment
conda env create -f environment.yml
```

Additional Resources

Use the following instructions as a guide for extra resources and better familiarity with working with Python. In case you want to manage your python environment, export your outcomes to PDF, and get extra learning resources.

Environment Management - Useful Commands

```
# List all environments
conda env list

# Activate environment
conda activate gg3209

# Deactivate environment
conda deactivate

# Update all packages in environment
conda update --all

# Install additional package
conda install package-name

# Remove environment
conda env remove --name gg3209
```

Updating the Environment

```
# Update environment from file
conda env update -f environment.yml --prune
```

Exporting Your Environment

```
# Export current environment
conda env export > my-environment.yml
```

PDF Generation

You are required to submit your work as report to MMS in a PDF format, and most of the outcomes created in this modules are Jupyter Notebooks, so you will need to export them as PDF. This environment includes comprehensive PDF generation capabilities for creating professional scientific documents from Jupyter notebooks. This includes:

- **LaTeX-based PDF generation** for high-quality academic formatting
- **Web-based PDF conversion** for quick exports
- **Scientific document formatting** with proper citations and references
- **Professional layout templates** for reports and dissertations

Testing PDF Generation

After setting up your environment, test the PDF generation capabilities:

1. Open a terminal or command prompt windows and lunch python.
2. locate a folder where you can create a python script.
3. Create the

```
import nbformat
from nbconvert import PDFExporter
import os

print("Testing PDF generation capabilities...")

# Test LaTeX availability
try:
    import subprocess
    result = subprocess.run(['pdflatex', '--version'],
                            capture_output=True, text=True)
    if result.returncode == 0:
        print("  LaTeX/PDFLaTeX is available")
```

```
    else:
        print("  LaTeX not found")
except:
    print("  LaTeX not available")

# Test nbconvert PDF export
try:
    exporter = PDFExporter()
    print("  nbconvert PDF exporter is available")
except Exception as e:
    print(f"  nbconvert PDF exporter error: {e}")

# Test WeasyPrint (alternative PDF generator)
try:
    import weasyprint
    print("  WeasyPrint is available")
except Exception as e:
    print(f"  WeasyPrint error: {e}")

# Test ReportLab (programmatic PDF creation)
try:
    from reportlab.pdfgen import canvas
    print("  ReportLab is available")
except Exception as e:
    print(f"  ReportLab error: {e}")

print("PDF generation test complete!")
```

Converting Notebooks to PDF

Method 1: Command Line (Recommended)

```
# Activate environment
conda activate gg3209

# Convert notebook to PDF via LaTeX
jupyter nbconvert --to pdf your_notebook.ipynb

# Convert with custom template, optional
jupyter nbconvert --to pdf --template classic your_notebook.ipynb
```

```
# Convert with bibliography support, optional.
jupyter nbconvert --to pdf --template article your_notebook.ipynb
```

Method 2: Jupyter Lab Interface

1. Open your notebook in Jupyter Lab
2. Go to **File > Export Notebook As > PDF**
3. Choose export options
4. Save the generated PDF

Method 3: Programmatic Conversion (optional)

```
import nbformat
from nbconvert import PDFExporter

# Read notebook
with open('your_notebook.ipynb', 'r') as f:
    nb = nbformat.read(f, as_version=4)

# Convert to PDF
pdf_exporter = PDFExporter()
pdf_exporter.template_name = 'classic'
(body, resources) = pdf_exporter.from_notebook_node(nb)

# Save PDF
with open('output.pdf', 'wb') as f:
    f.write(body)
```

Professional PDF Features

Creating Professional Reports

Template Structure

To create a template notebook with:

```
# Report template structure
"""
# Title: Professional Spatial Data Science Report
## Author: Your Name
## Date: Current Date
## Abstract
Brief description of the analysis...

## 1. Introduction
Research question and objectives...

## 2. Methodology
### 2.1 Data Sources
### 2.2 Analytical Methods
### 2.3 Software and Tools

## 3. Results
### 3.1 Descriptive Statistics
### 3.2 Spatial Analysis
### 3.3 Hotspot Analysis

## 4. Discussion
Interpretation of results...

## 5. Conclusions
Summary and recommendations...

## References
Academic citations...

## Appendices
Additional materials...
"""
```

Professional Visualization for PDF

```
import matplotlib.pyplot as plt
import seaborn as sns

# Configure matplotlib for high-quality PDF output
```

```
plt.rcParams['figure.dpi'] = 300
plt.rcParams['savefig.dpi'] = 300
plt.rcParams['font.size'] = 12
plt.rcParams['axes.titlesize'] = 14
plt.rcParams['axes.labelsize'] = 12
plt.rcParams['xtick.labelsize'] = 10
plt.rcParams['ytick.labelsize'] = 10
plt.rcParams['legend.fontsize'] = 11
plt.rcParams['figure.titlesize'] = 16

# Use professional color palette
sns.set_palette("husl")

# Create publication-ready figures
fig, ax = plt.subplots(figsize=(8, 6))
# Your plotting code here
plt.tight_layout()
plt.savefig('figure.png', dpi=300, bbox_inches='tight')
plt.show()
```

Troubleshooting PDF Generation

Common Issues and Solutions

Issue: LaTeX not found

```
# Solution: Install LaTeX distribution
# Windows: Download MiKTeX or TeX Live
# macOS: Install MacTeX
conda install texlive-core texlive-latex-extra
```

Issue: PDF conversion fails

```
# Solution: Use alternative method
jupyter nbconvert --to html your_notebook.ipynb
# Then use browser to print to PDF
```

Issue: Figures not appearing in PDF

```
# Solution: Ensure figures are saved inline
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams['savefig.format'] = 'png'
plt.rcParams['savefig.bbox'] = 'tight'
```

Issue: Long code cells breaking across pages

```
# Solution: Use page breaks and cell splitting
from IPython.display import display, HTML
display(HTML('<div style="page-break-before: always;"></div>'))
```

Best Practices for PDF Generation

1. **Use consistent formatting** throughout your notebook
2. **Include descriptive markdown** for each analysis step
3. **Add figure captions** and table descriptions. One of the most common issue when you create a PDF.
4. **Use appropriate figure sizes** for print media
5. **Test PDF generation** regularly during development
6. **Include proper citations** and references
7. **Use professional fonts** and color schemes
8. **Optimize images** for print quality (optional)

And more resources

Learning Materials

- **Geopandas Documentation**: https://geopandas.org/
- **PySAL Documentation**: https://pysal.org/
- **Spatial Data Science Book**: https://geographicdata.science/book/

Data Sources

- **Natural Earth**: https://www.naturalearthdata.com/
- **OpenStreetMap**: https://www.openstreetmap.org/
- **Census Data**: https://www.census.gov/

Community Support

- **Stack Overflow**: Use tags `geopandas`, `spatial-analysis`, `python`
- **GitHub Issues**: For specific library problems
- **Reddit**: r/gis and r/Python communities

Last updated: July 2025, For questions or issues, please refer to the troubleshooting guide

Lab No 1: Intro to Python

Overview

Welcome to the second part of the module of GG3209 Spatial Analysis with GIS. This part will take advantage of the initial part, which provided you with a solid understanding of spatial data formats (vector-raster) and use them to perform multiple types of analysis like the so-called Multi-Criteria Evaluation (MCE) using the widely popular Open-Source GIS tool, QGIS.

Now, in this part, you will be guided to handle and use another powerful tool in the geospatial field, Python. It is a free and open-sourced scripting language that was commonly used to automate tasks in the GIS world. Nowadays, it is one of the most popular programming languages[1], especially in GIScience. It is widely used in the private and public sectors and academia for cutting-edge research, where scripts are created and shared using this language to share new methods, knowledge, data, and analysis through multiple scientific fields.

In fact, most companies or institutes where you might want to apply will be happily interested in your development skills using Python and will validate your current contribution or work in platforms like GitHub, where you can share and disseminate your coding project. Think about its impact as we cluster R for spatial statistical analysis, and Python is mainly used for escalable and robust spatial analysis. Every day, more packages[2] and code repositories are shared and maintained for easy use and installation, allowing developers or analysts from all backgrounds and expertise to use and integrate them into their own code.

This part of the module and this lab workbook are meant to be an introduction to Python. As with any new language, you need to learn the basic rules (grammar) to write your own script, and then, with practice and more practice, you will soon become a Python developer.

[1] https://www.stackscale.com/blog/most-popular-programming-languages/

[2] https://pypi.org/

> Do not be afraid of failure or errors, even during the installation process; it has happened to all of us, regardless of the level of expertise or number of projects created. In programming, failure is part of the process; what is essential is to find the foundation of any issue and understand how code, logic, and syntax work in harmony to get the results you are expecting.

Estimated time of completion: 45 Minutes

Learning Outcomes

- Run Python scripts using **ArcGIS Pro** and **ArcGIS Online**
- Conducted xgeoprocessing tasks (e.g., buffering, proximity analysis)
- Created and saved notebooks and web maps
- Practiced using both ArcPy and ArcGIS Python API

Lab No 2: Python Basics - Part 1

Introduction

Now you have acquired some basic skills of creating new notebooks, run your python environment, the next step and certainty the long process is to learn the python basics of this programming language. This will give you an idea of the 'grammar' that python includes.

Please go to each cell and address the challenges/questions when is required.

Content:

1. How to deal with **variables**.
2. Explain the difference between **Python data types**.
3. Perform mathematical and logical operations.
4. Work with **lists**, **tuples**, **sets**, and **dictionaries**.
5. Apply appropriate **methods** to different data types.

If you need more information or examples, here it is a great resource w3school.com.

Variables

You can think of a **Variable** as anything that related to information or data. A Variable can be a file path on your local machine, could be also a path to a shapefile, could be just a number or letter. Once you create a variable you are assigning a certain space of memory on your computer to storage that information. So you could use it or call it to reference the associated data or object for use in processes and analyses. Note that there are some rules for variable names:

There are certain considerations:

- cannot start with a number or special character (or, can only start with a letter or an underscore) (for example, x1 is valid while 1x is not.).
- can only contain letters, numbers, or underscores. No other special characters are allowed (for example, x, x1, _x, and x1_ are all valid. x$ is not valid.).

- are case-sensitive (for example, x1 and X1 are two separate variables).

Since Python is an object-based language, you will primarily interact with your data using variables. The *print()* function is used to print the data referenced by the variable.

```
x = 1
y = "Python"
x1 = 2
y1 = "Spatial Data"
_x = 3
_y = "Web GIS"
print(x)
print(y)
print(x1)
print(y1)
print(_x)
print(_y)
```

You can also assign data to multiple variables as a single line of code as demonstrated below. In Python variable names are dynamic, so you can overwrite them. This can, however, be problematic if you overwrite a variable name accidentally. So, use unique names if you do not want to overwrite prior variable.

```
x, x1, _x = 1, 2, 3
print(x)
print(x1)
print(_x)
```

Assignment Operators are used to assign values or data to variables. The most commonly used operator is =. However, there are some other options that allow variables to be manipulated mathematically with the resulting value saved back to the original data object that the variable references. These additional assignment operators can be useful, but you will use the = operator most of the time.

For example, += will add a value to the current value and assign the result back to the original variable. In the first example below, x references the value 2. The += assignment operator is then used to add 3 to x and save the result back to x. Work through the provided examples and make sure you understand the use of each operator.

```
x = 2
print(x)
x += 3
print(x)
```

```
x = 2
print(x)
x -= 3
print(x)

x = 2
print(x)
x *= 3
print(x)

x = 2
print(x)
x /= 3
print(x)

x = 2
print(x)
x **= 3
print(x)
```

Important Note:

In the cell bellow, you will run small experiment that explains some important behavior of Python. You have defined a variable a that holds a **list** of three values. We will discuss **lists** later in this section. Now, you create a new variable b and assign it to be equal to a. Layer you edit the variable a by appending a new value to the list (You will see how this is done later, so don't worry if you don't understand how this works yet). When you print a and b, you can see that both variables contain the same set of numbers in the list even though you added 8 to a after setting b equal to a. Or, the change that you made to a was also applied to b.

In Python, certain types of objects, such as lists, are **mutable**. This means that it is possible to change the data stored in memory and referenced to the variable. When a mutable object is altered, all variables that point to it will reflect this change. What this means practically is that setting b equal to a results in a and b pointing to the same object or data in memory. So, changes to a or b will be reflected in both variables since the data being referenced by both have been updated.

If you have experience in **R**, and I guess you do, you will see how Python is different, as in R, setting a variable equal to another variable would make a copy that was not linked and could be altered without changing the original variable.

To test whether two variables reference the same object in memory, you can use the *is* keyword. if *True* is returned, then they reference the same object. You can also print the object ID, which represents the memory address for the object, using the *id()* function. Using both methods below, you can see that *a* and *b* reference the same object.

```
a = [5, 6, 7]
b = a
a.append(8)
print(a)
print(b)

print(a is b)
print(id(a))
print(id(b))
```

What if you wanted to make a copy of a variable referencing mutable data that does not reference the same object?

For example, you may want to be able to make changes to *a* that do not impact *b*. This can be accomplished using the *copy()* or *deepcopy()* functions from the **copy** module.

Check how now we import a new package, module or library to get new functionalities.

In the experiment below, You have defined *b* as a deep copy of *a*. Now, changes made to *a* do not impact *b*. This is because they do not reference the same object in memory since *deepcopy()* makes a copy of the object or data to a new location in memory. This is confirmed using *is* and *id()*.

```
import copy
a = [5, 6, 7]
b = copy.deepcopy(a)
a.append(8)
print(a)
print(b)

print(a is b)
print(id(a))
print(id(b))
```

Comments (Important for clear and scalable coding)

Now please pay attention as this is the key of many developers and analysts, you need to learn how to commenting your code. **Comments** are used to make your code more readable and are not interpreted by the computer. Instead, they are skipped and meant for humans. Different languages use different syntax to denote comments. Python uses the hashtag or pound sign. You can add comments as new lines or following code on the same line.

Unfortunately, Python does not have specific syntax for multi-line comments. However, this can be accomplished by adding hashtags before each separate line or using a multi-line string that is not assigned to a variable. Examples are shown below.

It is generally a good idea to comment your code for later use and for use by others. In fact you need to comment your code for all assignments and exercises you run in this part of the course.

```
#Single-line comment
x = 1
y = 2 #Another single-line comment
#A
#multi-line
#comment
z = 3
"""
Another multi-line comment
"""
w = 4
```

Data Types

A variety of **data types** are available in Python to store and work with a variety of input. Below are explanations of the data types which you will use most often. There are additional types that we will not discuss.

When creating a variable, it is not generally necessary to explicitly define the data type. However, this can be accomplished using **constructor functions** if so desired. Constructor functions can also be used to change the data type of a variable, a process known as **casting**.

Available constructor methods include *str()*, *int()*, *float()*, *complex()*, *list()*, *tuple()*, *dict()*, *set()*, and *bool()*.

To determine the data type, you can use the *type()* function. See the examples below where I convert an integer to a float and then a float to a string.

- **Numeric**
  - **Int** = whole numbers
  - **Float** = numbers with decimal values
  - **Complex** = can include imaginary numbers
- **Text**
  - **String** = characters or numbers treated as characters
- **Boolean**
  - **Boolean** = logical *True* or *False*
- **Sequence**
  - **List** = list of features that can be re-ordered, allows for duplicates, and is indexed
  - **Tuple** = list of features that cannot be re-ordered, allows for duplicates, and is indexed
- **Mapping**
  - **Dictionary** = list of features that can be re-ordered, does not allow duplicates, is indexed, and contains **key** and **value** pairs
- **Set**
  - **Set** = list of features that are unordered, not indexed, and does not allow for duplicates

```
#Create a variable and check the data type
x = 1
print(type(x))
#Change the data type
x = float(x)
print(type(x))
x= str(x)
print(type(x))
```

Numbers

Regardless of the the type (integer, float, or complex), numbers are defined without using quotes. If a number is placed in quotes it will be treated as a string as demonstrated below. This is important, since the behavior of the data is altered. In the example, x represents 1 as a number while y represents "1" as a string (note the quotes). Adding x to itself will yield 2 (1 + 1). Adding y to itself will yield "11", or the two strings are combined or **concatenated**.

```
#Create variables
x = 1
y = "1"
print(x + x)
print(y + y)
```

Numbers support mathematical operations, as demonstrated below. If you are not familiar with these concepts, **modulus** will return the remainder after division while **floor division** will round down to the nearest whole number after division.

If a whole number has no decimal values included or no period (1 vs. 1. or 1.0), this implies that the output is in the integer data type as opposed to float type.

```
x = 4
y = 3
print(x + y) #Addition
print(x - y) #Subtraction
print(x * y) #Multiplication
print(x / y) #Division
print(x % y) #Modulus
print(x ** y) #Exponentiation
print(x // y) #Floor Division
```

Strings

Strings are defined using single or double quotes. If quotes are included as part of the text or string, then you can use the other type to define the data as text. Again, numbers placed in quotes will be treated as strings.

```
x = "Python"
y = "is great" #Must use double quotes since a single quote is use in the string
z = "2" #Number treated as a string
print (x,y,z)
```

Portions of a string can be sliced out using **indexes**.

Very important note: In Python the indexing starts at 0 as opposed to 1., like in **R** So, the first character is at index 0 as opposed to index 1. Negative indexes can be used to index relative to the end of the string. In this case, the last character has an index of -1.

Indexes combined with square brackets can be used to slice strings. Note that the last index specified in the selection or range will not be included and that spaces are counted in the indexing.

```
x = "GG3209 Spatial Analysis with GIS"
print(x[0:6])
print(x[7:14])
print(x[15:23])
print(x[-3:])
```

Strings can be combined or **concatenated** using the addition sign. If you want to include a number in the string output, you can **cast** it to a string using *str()*. In the example below, note the use of blank spaces so that the strings are not ran together.

The *len()* function can be used to return the length of the string, which will count blank spaces along with characters.

```
x = "Spatial"
xx="GG"
y = 3209
z = "Analysis"
w = "With Python"
strng1 = xx + str(y) + " "+ x + " " + z + " " + w
print(strng1)
print(len(strng1))
```

Method

A **method** is a function that belongs to or is associated with an object. Or, it allows you to work with or manipulate the object and its associated properties in some way. Data types have default methods that can be applied to them.

Methods applicable to strings are demonstrated below. Specifically, methods are being used to change the case and split the string at each space to save each component to a list.

```
x = "GG3209 Spatial Analysis with GIS"
print(x.upper())
print(x.lower())
lst1 = x.split(" ")
print(lst1)
```

When generating strings, issues arise when you use characters that have special uses or meaning in Python. These issues can be alleviated by including an **escape character** or backslash as demonstrated below.

```
s1 = "Issue with \"quotes\" in the string."
s2 = "C:\\data\\project_1" #Issue with file paths.
s3 = "Add a new line \nto text string"
print(s1)
print(s2)
print(s3)
```

Booleans

Booleans can only be *True* or *False* and are often returned when an expression is logically evaluated.

A variety of **comparison operators** are available. Note the use of double equals; a single equals cannot be used since it is already used for variable assignment, or is an assignment operator, and would thus be ambiguous.

- **Comparison Operators**
  - **Equal**: ==
  - **Not Equal**: !=
  - **Greater Than**: >
  - **Greater Than or Equal To**: >=
  - **Less Than**: <
  - **Less Than or Equal To**: <=

Logical statements or multiple expressions can be combined using **Logical Operators**.

- **Logical Operators**:
  - A AND B: **and**
  - A OR B: **or**
  - A NOT B: **not**

```
x = 3
y = 7
z = 2
print(x == 7)
print(x > y)
print(x < y)

print(x < y and x > z)
print(x < y and x < z)
print(x < y or x < z)
```

You can also assign Booleans to a variable. Note that you do not use quotes, as that would cause the text to be treated as a string instead of a Boolean.

```
x = "True"
y = True
print(type(x))
print(type(y))
```

Lists

Now one of the very relevant type of objects in Python. **Lists** allow you to store multiple numbers, strings, or Booleans in a single variable. Square brackets are used to denote lists.

Items in a **list** are ordered, indexed, and allow for duplicate members. Indexing starts at 0. If counting from the end, you start at -1 and subtract as you move left. A colon can be used to denote a range of indexes, and an empty argument before the colon indicates to select all elements up to the element following the colon while an empty argument after the colon indicates to select the element at the index specified before the colon and all features up to the end of the list. The element at the last index is not included in the selection.

Python lists can contain elements of different data types.

```
lst1 = [6, 7, 8, 9, 11, 2, 0]
lst2 = ["A", "B", "C", "D", "E"]
lst3 = [True, False, True, True, True, False]
print(lst1[0])
print(lst1[0:3])
print(lst2[-4:-1])
print(lst2[:3])
print(lst2[3:])
lst4 = [1, 2, "A", "B", True]
print(type(lst4[0]))
print(type(lst4[2]))
print(type(lst4[4]))
```

When the *len()* function is applied to a list, it will return the number of items or elements in the list as opposed to the number of characters. When applied to a string item in a list, this function will return the length of the string.

```
lst1 = ["A", "B", "C", "D", "E"]
print(len(lst1))
print(len(lst1[0]))
```

The code below shows some example methods for strings.

```
lst1 = ["A", "B", "C", "D", "E"]
lst1.append("F") #Add item to list
print(lst1)
lst1.remove("F") #Remove item from a list
print(lst1)
lst1.insert(2, "ADD") #Add item to list at defined position
print(lst1)
lst1.pop(2) #Remove item at specified index or the last item if no index is provided
print(lst1)
```

As explained above, in order to copy a list and not just reference the original data object, you must use the *copy()* or *deepcopy()* method. Simply setting a new variable equal to the original list will cause it to reference the original data object, so changes made to the old list will update to the new list. This is demonstrated in the example below.

```
lst1 = ["A", "B", "C", "D", "E"]
lst2 = lst1
lst3 = lst1.copy()
print(lst2)
print(lst3)
lst1.append("F")
print(lst2)
print(lst3)
```

Lists can be concatenated together, or a list can be appended to another list, using the methods demonstrated below.

```
lst1 = ["A", "B", "C"]
lst2 = ["D", "E", "F"]
lst3 = lst1 + lst2
print(lst1)
print(lst2)
print(lst3)
lst1.extend(lst2)
print(lst1)
```

Lastly, lists can contain other lists, tuples, or dictionaries, which will be discussed below. In the example, *lst2* contains four elements, the last of which is a list with three elements.

```
lst1 = ["A", "B", "C"]
lst2 = ["D", "E", "F", lst1]
print(lst2)
```

Tuples

Tuples are similar to lists in that they are ordered and allow duplicate elements. However, they cannot be altered by adding items, removing items, or changing the order of items. To differentiate them from lists, parenthesis are used as opposed to square brackets. Think of tuples as lists that are protected from alteration, so you could use them when you want to make sure you don't accidentally make changes.

If you need to change a tuple, it can be converted to a list, manipulated, then converted back to a tuple.

```
t1 = (1, 3, 4, 7)
print(type(t1))
```

Dictionaries (not the one for spelling)

Dictionaries are unordered, changeable, indexed, and do not allow for duplicate elements. In contrast to lists, tuples, each **value** is also assigned a **key**.

And here is the key –> Values can be selected using the **associated key**.

You can also use the key to define a value to change.

Similar to lists, you must use the *copy()* or *deepcopy()* method to obtain a copy of the dictionary that will not reference the original data or memory object.

```
cls = {"code": "GG3209", "Name":"Spatial Analysis with Python" }
print(cls)
print(cls["Name"])
cls["Code"] = 461
print(cls)
```

Multiple dictionaries can be combined into a **nested dictionary**, as demonstrated below.

The keys can then be used to extract a sub-dictionary or an individual element from a sub-dictionary.

```
cls1 = {"prefix" : "GG", "Number" : 3209, "Name": "Spatial Analysis with Python"}
cls2 = {"prefix" : "GG", "Number" : 3210, "Name": "Advanced Analysis with Python"}
cls3 = {"prefix" : "GG", "Number" : 3211, "Name": "Introduction to Remote Sensing"}
cls4 = {"prefix" : "GG", "Number" : 3212, "Name": "Web GIS"}
clsT = {
    "class1" : cls1,
    "class2" : cls2,
    "class3" : cls3,
    "class4" : cls4
}
print(clsT)
print(clsT["class1"])
print(clsT["class1"]["Name"])
```

Additional Types

Arrays

Arrays are similar to lists; however, they must be declared.

They are sometimes used in place of lists as they can be very compact and easy to apply mathematical operations. However in this course, we will primarily work with **NumPy** arrays, which will be discussed in more detail in a later module.

If you wanna know how an array looks like, here is an example

```
my_array = [1, 2, 3, 4, 5]
print(my_array[0])   # Output: 1
print(my_array[2])   # Output: 3
```

You can also modify elements in the array by assigning a new value to a specific index:

```
my_array[1] = 7
print(my_array)      # Output: [1, 7, 3, 4, 5]
```

Classes

Now some a more complex type of object in Python: **Classes** are used to define specific types of objects in Python and are often described as templates.

Once a class is defined, it can be copied and manipulated to create a **subclass**, which will inherit properties and methods from the parent class but can be altered for specific uses. You get more details in the next Notebook (Part 2). You will also see example uses of classes in futher examples.

One use of classes is to define specialized data models and their associated methods and properties. For example, classes have been defined to work with geospatial data types.

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        print("Woof!")

my_dog = Dog("Buddy", "Golden Retriever")
print(my_dog.name)    # Output: Buddy
print(my_dog.breed)   # Output: Golden Retriever
my_dog.bark()         # Output: Woof!
```

In the example above, we've defined a class called **Dog** that has a constructor method "**init** that takes two **parameters**, *name* and *breed*.

Inside the constructor, we assign the passed-in values to *instance* (re-create the object) variables with the same names using the *self* keyword.

We've also defined a method called bark that simply prints out "Woof!" when called. To call this method on *an instance* of the Dog class, we first create an instance of the class by calling the constructor and passing in the required parameters, and then we call the bark method on that instance.

Final remarks

Before moving on, I wanted to note which data types are **mutable** and which are **immutable**.

Again, data or objects that are mutable can be altered after they are defined (such as adding a new element to a list).

Mutable types include **lists**, **sets**, and **dictionaries**.

Immutable types include **booleans**, **integers**, **float**, **complex**, **strings**, and **tuples**.

Next Step

Once you finish this, now clone **Python_Basics_Part2**, In that notebook, we will discuss more components of Python including functions, control flow, loops, modules, and reading data from disk.

References

1. PythonGIS
2. Python Data Spatial

Lab No 3: Python Basics - Part 2

Introduction

We will now discuss additional components of the Python language including **functions**, **flow control**, **loops**, **modules**, and **reading/writing** data from disk. It is important you learn effective ways to make your code efficient, as python can be memory-consuming, but at the same time learn how to create Loops, and flow control will help to run tasks in an automatic ways making your scripts a powerful tool.

If you have some experience with this type of conditions or components from your previous courses with R, you will see that the logic is the same. Reading and writing files using python is also a key skill to learn, as in most of the cases you will need to access files, or folders where your spatial data is located.

Please go through every cell, reading carefully all descriptions and run the code cell to see the examples, you later might want to use this notebook to recall how to create the following components:

Content

1. Define and use **functions**.
2. Use **If...Else** statements, **While Loops**, and **For Loops**.
3. Describe and interpret **classes** and **methods**.
4. Access and use **modules** and **libraries**.
5. Work with local files and directories.
6. Use **f-strings** and **list comprehension**.

If you need more information or examples Here it is a great resource w3school.com.

2. Functions

Functions are probably one of the key components in any programming language, **Functions** do something when called. You can think of those as tools. **Methods**, which we will discuss

in more detail later in this notebook, are like functions except that they are tied to a specific **object**.

When creating a new **class**, you can define methods specific to the new class. Functions generally have the following generic syntax:

```
output = function(Parameter1, Parameter2). #Think of parameters as Inputs., so you have outpu
```

In contrast, methods will have the following generic syntax:

```
output = object.method(Parameter1, Parameter2).
```

Below, You are generating a simple function that multiplies two numbers together.

The *def* keyword is used when defining a function. Within the parenthesis, a list of **parameters**, which are often specific inputs or required settings, can be provided.

```
In the below example, the function accepts two parameters: *a* and *b*. On the next line, inc
```

Once a function is created, it can be used. In **Example 1**, You will see two **arguments**, or values assigned to the parameters, and save the result to a variable *x*.

Then when using a function, it is also possible to provide the arguments as *key* and *value* pairs, as in **Example 2**.

When creating a function, default arguments can be provided for parameters when the function is defined. If arguments are not provided when the function is used, then the default values will be used, as demonstrated in **Example 3**.

2.1 Indentation

This is a good time to stop and describe indentation.

Python makes use of **whitespace**, indentations, or tabs to denote or interpret units of code. This is uncommon, as most other languages use brackets or punctuation of some kind. So, it is important to properly use indentations or your code will fail to execute correctly or at all.

```
#Example 1
def multab(a,b):
    return a*b

x = multab(3,6)
```

```
print(x)

#Example 2
x = multab(a=5, b=3)
print(x)

#Example 3
def multab(a=1,b=1):
    return a*b
x = multab()
print(x)
```

Challenge 2.1

In the next code cell, create a function that transform the distance in miles to kilometers between London and Edinburgh. **Try no to google or use ChatGPT for this challenge, as those will provide more advance suggestion, you will only need the previous cell description to run this challenge.**

```

```

2.2 Options *args and **kwargs for functions

There are a few other options when defining functions that increase flexibility. For example, what if you wanted to edit the function created above so that it can accept more than two arguments and so that the number of arguments can vary? This can be accomplished using either ***args** or ****kwargs**.

***args**:

A single asterisk (*) is used to unpack an iterable, such as a list, whereas two asterisks (**) are used to unpack a dictionary. Using *args allows you to to provide a variable number of non-keyword arguments (or, each argument does not need to be assigned a key). In contrast, **kwargs is used to provide a variable number of keyword arguments (or, each argument must have a key).

In the **first example** below, You altered the function from above to accept two or more arguments.

Within the function, later you define a variable *x1* that initially assigned a value of 1. Then, inside of a **for loop**, which will be discussed later, you iteratively multiply *x1* by the next provided value. To test the function, a feed it the values 1 through 5 as non-keyword arguments.

The result is calculated as 1x1 –> 1x2 –> 2x3 –> 6x4 –> 24x5 –> 120.

Note that the single asterisk is key here. The work "args" could be replaced with another term, as the second part of the example demonstrates. What is important is that * is used to unpack an iterable.

```
# Example 1

def multab(*args):
    x1 = 1
    for a in args:
        x1 *= a
    return x1

x = multab(1, 2, 3, 4, 5)
print(x)

#Example 2

def multab(*nums):
    x1 = 1
    for a in nums:
        x1 *= a
    return x1

x = multab(1, 2, 3, 4, 5)
print(x)
```

****kwargs**:

The next example demonstrates the use of **kwargs. Here, the arguments must have keys. Again, what is important here is the use of ** to unpack a dictionary: "kwargs" can be replaced with another term. Note the use of the *.values()* method for a dictionary. This allows access to the values as opposed to the associated keys.

```
def multab(**kwargs):
    x1 = 1
    for a in kwargs.values():
```

```
        x1 *= a
    return x1

x = multab(a=1, b=2, c=3, d=4, e=5)
print(x)

def multab(**nums):
    x1 = 1
    for a in nums.values():
        x1 *= a
    return x1

x = multab(a=1, b=2, c=3, d=4, e=5)
print(x)
```

The next cell demonstrates the use of the single asterisk to unpack an iterable, in this case a list. Each element in the list is returned separately as opposed to as a single list object. This is the same functionality implemented by *args.

```
x = [2,3,4,5]
print(*x)
```

Lastly, it is possible to use both ***args** and ****kwargs** in the same function.#

However, ***args** must be provided before ****kwargs**. In the example below, the parameter a is provided an argument of 1 while the parameter b is provided an argument of 2. 3 would be associated with *args, since it is not assigned a key, while 4 and 5 would be associated with **kwargs since they are assigned a key.

```
# Create a function
def multab(a=2, b=2, *args, **kwargs):
    x1 = 1 #Define variables
    x1 *= a
    x1 *= b
    if args: # Conditional
        for arg in args: #Loop
            x1 *= arg
    if kwargs:
        for kwarg in kwargs.values():
            x1 *= kwarg
    return x1
```

```
x = multab(1, 2, 3, c=3, d=4)
print(x)
# Read description below to understand how this function works, looks more complicated tha
```

As the examples above demonstrate, *args and **kwargs increase the flexibility of functions in Python by allowing for variable numbers of arguments. Even if you do not make use of these options, they are important to understand, as many functions that you encounter will make use of them. So, knowledge of this functionality will aid you in understanding how to implement specific functions and interpret the associated documentation.

NOTE:

The following options will you give an additional level of skill in Python. Although they are rarely included in python basics, we consider that if you can master those, you will have an extra level of expertise and will definitely help you to make more efficient programs using python.

2.3 Lambda

A **lambda function** is a special function case that is generally used for simple functions that can be anonymous or not named. They can accepted multiple arguments but can only include one expression. Lambda functions are commonly used inside of other functions.

```
lam1 = lambda a, b, c: str(a) + " " + str(b) + " " + str(c)
print(lam1("Geospatial", "Data", "Science"))

a = "Geospatial"
b = "Dasta"
c = "Science"
```

2. Scope

Variables are said to have **global scope** if they can be accessed anywhere in the code.

In contrast, **local scope** implies that variables are only accessible in portions of the code.

For example, by default new variables defined within a function cannot be called or used outside of the function. If you need to specify a variable within a function as having global scope, the *global* keyword can be used.

In the first example below, the variables xG, yG, and z have global scope, so can be called and used outside of the function. In contrast, variables xL and yL have local scope and are not available outside of the function. If you attempt to run the last line of code, which is commented out, you will get an error.

```
xG = 2
yG = 3
def Func1(a, b):
    xL = a*a
    yL = b*b
    return xL + yL

z = Func1(xG, yG)
print(xG)
print(z)
#print(xL+3) will not work due to local scope
```

If you need a variable declared inside of a function to have global scope, you can use the *global* keyword as demonstrated below.

```
xG = 2
yG = 3
def Func1(a, b):
    global xL
    xL = a*a
    global yL
    yL = b*b
    return xL + yL

z = Func1(xG, yG)
print(xG)
print(z)
print(xL+3)
```

3. Pass

It is not possible to leave a function empty when it is defined. As you develop code, you can make use of the *pass* keyword as a placeholder before adding content to a function. This will allow you to work with your code without errors until you complete the content within the function. *pass* can also be used within incomplete class definitions and loops.

```
def multab(x, y):
    pass
```

Conditionals - Control Flow

1. If...Else

All coding languages allow for **control flow** in which different code is executed depending on a condition.

If...Else statements are a key component of how this is implemented. Using logical conditions that evaluate to **True** or *False*, it is possible to program different outcomes. Think about this as the rules, if something is **True**, then **do this**, but if something is **False**, then **do that**.

The first example uses only **if**. So, if the condition evaluates to **True**, the remaining code will be executed. If it evaluates to **False** then nothing is executed or returned. In this case, the condition evaluates to **True**, so the text is printed.

Again, remember that indentation is very important in Python. The content of the *if* statement must be indented or the code will fail.

```
x = 7
if x > 6:
    print(str(x) + " is greater than 6.")
```

It is common to have a default statement that is executed if the condition evaluates to *False* as opposed to simply doing nothing. This is the use of an *else* statement. No condition is required for the *else* statement since it will be executed for any case where the *if* condition evaluates to *False*. Again, note the required indentation.

```
x = 3
if x > 6:
    print(str(x) + " is greater than 6.")
else:
    print(str(x) + " is less than or equal to 6.")
```

What if you want to evaluate against more than one condition? This can be accomplished by incorporating one or multiple *elif* statements. The code associated with the *else* statement will only be executed if the *if* and all *elif* statements evaluate to *False*.

All statements should be mutually exclusive or non-overlapping so that the logic is clear. In the second example, I have changed the first condition to $x >= 6$, so now the condition in the

if and *elif* statements overlap. When the code is executed, the result from the *if* statement is returned. Since the first condition evaluated to *True*, the associated code was executed and the *elif* and *else* statements were ignored. If I swap the *if* and *elif* conditions, a different result is obtained. So, the order matters. In short, this ambiguity can be avoided by using conditions that are mutually exclusive and non-overlapping.

```
x = 6
if x > 6:
    print(str(x) + " is greater than 6.")
elif x == 6:
     print(str(x) + " is equal to 6.")
else:
    print(str(x) + " is less than 6.")
```

```
x = 6
if x >= 6:
    print(str(x) + " is greater than 6.")
elif x == 6:
     print(str(x) + " is equal to 6.")
else:
    print(str(x) + " is less than 6.")
```

```
x = 6
if x == 6:
     print(str(x) + " is equal to 6.")
elif x >= 6:
     print(str(x) + " is greater than 6.")
else:
    print(str(x) + " is less than 6.")
```

2. While Loop

While loops are used to loop code as long as a condition evaluates to *True*. In the example below, a variable i is initially set to 14.

The loop executes as long as i remains larger than 7. At the end of each loop the -= **assignment operator** is used to subtract 1 from i. Also, i is simply a variable, so you do not need to use i specifically. For example, i could be replaced with x.

Please consider the following, **One potential issue with a while loop** is the possibility of an **infinite loop** in which the loop never stops because the condition never evaluates to

False. For example, if I change the assignment operator to +=, the condition will continue to evaluate to *True* indefinitely.

```
i = 14
while i > 7:
    print(i)
    i -= 1
```

3. For Loop

For Loops will execute code for all items in a sequence. For loops make use of data types that are **iterable**, or that can return each individual element in the data object sequentially (for example, each string element in a list). Data types that are iterable include **lists**, **tuples**, **strings**, **dictionaries**, and **sets**.

In the first example below, a for loop is being used to print every value in a list. In the next example, each character in a string is printed sequentially. Both lists and strings are iterable, so can be looped over.

```
lst1 = [3, 6, 8, 9, 11, 13]
for i in lst1:
    print("Value is: " + str(i))
```

```
str1 = "Remote Sensing"
for c in str1:
    print(c)
```

Combining a for loop and If…Else statements allows for different code to be executed for each element in a sequence or iterable, such as a list, based on conditions, as demonstrated in the code below. In later modules, you will see example use cases for working with and analyzing spatial data. Note the levels of indentation used, which, again, are very important and required when coding in Python. The content in the for loop is indented with one tab while the content within the *if*, *elif*, and *else* statements, which are include in the for loop, are indented with two tabs.

```
lst1 = [3, 6, 8, 9, 11, 13]
for i in lst1:
    if i < 8:
        print(str(i) + " is less than 8.")
    elif i == 8:
        print(str(i) + " is equal to 8.")
```

```
    else:
        print(str(i) + " is greater than 8.")
```

The *range()* function is commonly used in combination with for loops. Specifically, it is used to define an index for each element in an iterable that can then be used within the loop.

In the example below, *range()* is used to define indices for all elements in a list. The *len()* function returns the length, so the returned indices will be 0 through the length of the list, in this case 4. The for loop will iterate over indices 0 through 3 (the last index is not included). This allows for the index to be used within the loop. In the example, the index is used to extract each element from the list and save it to a new local variable (*country*), which is then provided to a print statement.

```
countries = ["Belgium", "Mexico", "Italy", "India"]
for i in range(len(countries)):
    country = countries[i]
    print("I would like to visit " + country + ".")
```

Another function that is commonly used in combination with for loops is *enumerate()*. For each element in the iterable, *enumerate()* will return an index and the element. Both can then be used within the loop.

In the first example below, *enumerate()* is used to create an index and extract each element in the list sequentially. In this case, the enumeration is not necessary, since the index is not needed. However, in the next example, the index is used to print different results depending on whether the index is even or odd. So, *enumerate()* is needed because I need access to both the index and the data element within the loop.

```
countries = ["Belgium", "Mexico", "Italy", "India"]
for index, country in enumerate(countries):
    print("I would like to visit " + country + ".")
```

```
countries = ["Belgium", "Mexico", "Italy", "India"]
for index, country in enumerate(countries):
    if index%2 == 0:
        print("I would like to visit " + country + ".")
    else:
        print("I would not like to visit " + country + ".")
```

There are some other useful techniques for flow control, code development, and error handling that we will not discuss in detail here. For example, **try**, **except**, and **finally** can be used to to handle errors and provide error messages. **break** is used to terminate a for loop based on

a condition. **continue** can be used to skip the remaining steps in an iteration and move on to the next iteration in a loop.

List Comprehension - Great Feature from Python!

List comprehension is a nice feature in Python.

This technique allows you to create a new list from elements in an existing list and offers more concise syntax than accomplishing the same task using a for loop.

In the first example, You will see that we use a list comprehension to return each element (*c*) in *lst1* to a new lst, *lst2*, if the string starts with the letter "B". Here is how you would read the syntax within the square brackets:

- "Return the element (*c*) from the list (*lst1*) if its first character is"B".

In the second example, no condition is used. Instead, you are concatenating "I would like to visit" to each country and returning each result as an element in a new list.

You will see examples of list comprehension in later modules. It is a very handy technique.

```
# Example 1

lst1 = ["Belgium", "Mexico", "Italy", "India", "Bulgaria", "Belarus"]
lst2 = [c for c in lst1 if c[0] == "B"]
print(lst2)
```

```
# Example 2

lst1 = ["Belgium", "Mexico", "Italy", "India", "Bulgaria", "Belarus"]
lst2 = ["I would like to visit " + c for c in lst1]
print(lst2)
```

Classes

We briefly covered **classes** in the previous notebook. Here, we will provide a more in-depth discussion.

Since Python is an object-based language, it is important to be able to define different types of objects. Classes serve as blueprints for creating new types of objects. Thus, the concept, use, and application of classes is very important in Python.

To create a new class, you must use the *class* keyword. In the following example, you are generating a new class called *Course*. To construct a class, you must use the *___init___()* function. Within this function, you can assign values to object properties, and *___init___()* will be executed any time a new instance of the class is created. When You create an instance of my *Course* class, three properties will be initiated: **the subject code**, **course number**, and **course name*.

self references the current instance of the class. Although you can use a term other than *self*, *self* is the standard term used. Regardless of the term used, it must be the first parameter in *___init___()*.

After the *___init___()* function, you then define a **method** that will be associated with the class. This method, **printCourse()**, will print the course name. Again, methods are functions that are associated with an object or class.

In order to use the method, You must first create an instance of the class and provide arguments for the required parameters. You can then apply the method to the instance to print the course info.

Once an instance of a class is created, the arguments assigned to properties can be changed using the following generic syntax: instance.property = new argument. Here you are changing the *number* parameter of the *x* instance of the *Course* class to 350. and then use my *printCourse()* method again.

```
class Course:
    def __init__(self, subject, number, name):
        self.subject = subject
        self.number = number
        self.name = name
    def printCourse(self):
        print("Course is " + self.subject + ": " + self.name + " with the code: " + str(se

x = Course("Spatial Analysys", 33209, "Using Python")
x.printCourse()
type(x)

x.number = 350
x.printCourse()
```

Once a class is created, **subclassess** can be derived from it. By default, subclasses will take on the properties and methods of the parent or superclass. However, you can alter or add properties and methods. This allows you to start with a more generic blueprint and refine it for a specific use case, as opposed to building a new class from scratch.

In the example below, I have redefined the *Course* class then subclassed it to create the *Undergrad* class. I have added a parameter, which requires redefining the *___init___()* function. The *super()* function returns all of the parameters and methods of the parent class. The use of *super()* provides control over what is inherited by the child class from the parent. We will not explore all possible use cases here. I then define a new method called *printCourse2()*.

Once an instance of the *Undergrad* class is created, I can use both methods, since the *printCourse()* method was inherited from the parent class.

```
class Course:
    def __init__(self, subject, number, name):
        self.subject = subject
        self.number = number
        self.name = name
    def printCourse(self):
        print("Course is " + self.subject + " " + self.name + ", Code:  "+ str(self.number

class Undergrad(Course):
    def __init__(self, subject, number, name, level):
        super().__init__(subject, number, name)
        self.level = level
    def printCourse2(self):
       print("Undergrad Course is " + self.subject + " "  + self.name + ". " + "Must be a

x = Undergrad("Spatial Analysys", 3209, "With Python", "for New students")
x.printCourse()
x.printCourse2()
```

You will not be required to created classes and subclasses in this course. However, it is important to understand this functionality of Python for making use of modules and libraries, as this is used extensively. For example, the PyTorch library, which is used for deep learning, makes extensive use of classes, and using it will require subclassing available classes.

Math Module

The functionality of Python is expanded using **modules**. Modules represent sets of code and tools and are combined into **libraries**. In this class, we will explore several modules or libraries used for data science including **NumPy**, **Pandas**, **matplotlib**, and **scikit-learn**. We will also explore libraries for geospatial data analysis including **GeoPandas** and **Rasterio**.

As an introduction to modules and libraries, we will now explore the **math module**. To use a module, it first needs to be **imported**. You can then use the methods provided by the

module. When using a method from the math module, you must include the module name as demonstrated below.

```
import math

x = 4.318
print(math.cos(x))
print(math.sin(x))
print(math.tan(x))
print(math.sqrt(x))
print(math.log10(x))
print(math.floor(x))
```

You can also provide an alias or shorthand name for the module when importing it by using the *as* keyword. This can be used to simplify code.

```
import math as m

x = 4.318
print(m.cos(x))
print(m.sin(x))
print(m.tan(x))
print(m.sqrt(x))
print(m.log10(x))
print(m.floor(x))
```

If you want to import each individual function from a module and not need to use the module name or alias name in your code, you can use the import syntax demonstrated below. This is generally best to avoid, especially if the module is large and/or has many functions.

```
from math import *

x = 4.318
print(cos(x))
print(sin(x))
print(tan(x))
print(sqrt(x))
print(log10(x))
print(floor(x))
```

You can also import individual functions or subsets of functions as opposed to the entire module.

```
from math import cos, sin
x = 4.318
print(cos(x))
print(sin(x))
```

Working with Files

1. Read Files

As a **data scientist** or **geospatial data scientist**, you need to use Python to work with and analyze files on your local machine.

First, a file or folder path can be assigned to a variable. On a Windows machine, you will need to either:

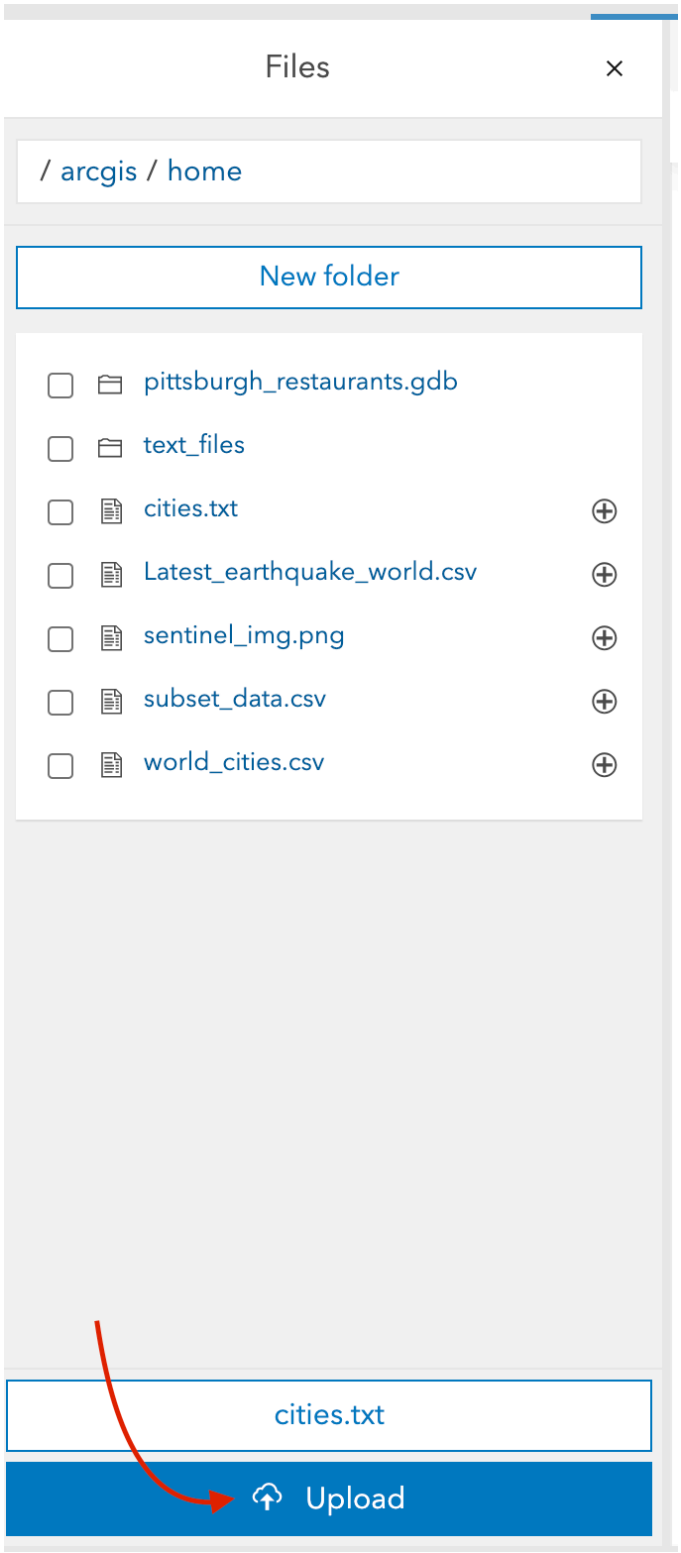
- Change the backslashes to forward slashes or
- Use the backslash escape character ahead of any backslash.

This is tricky depending of the operating system.

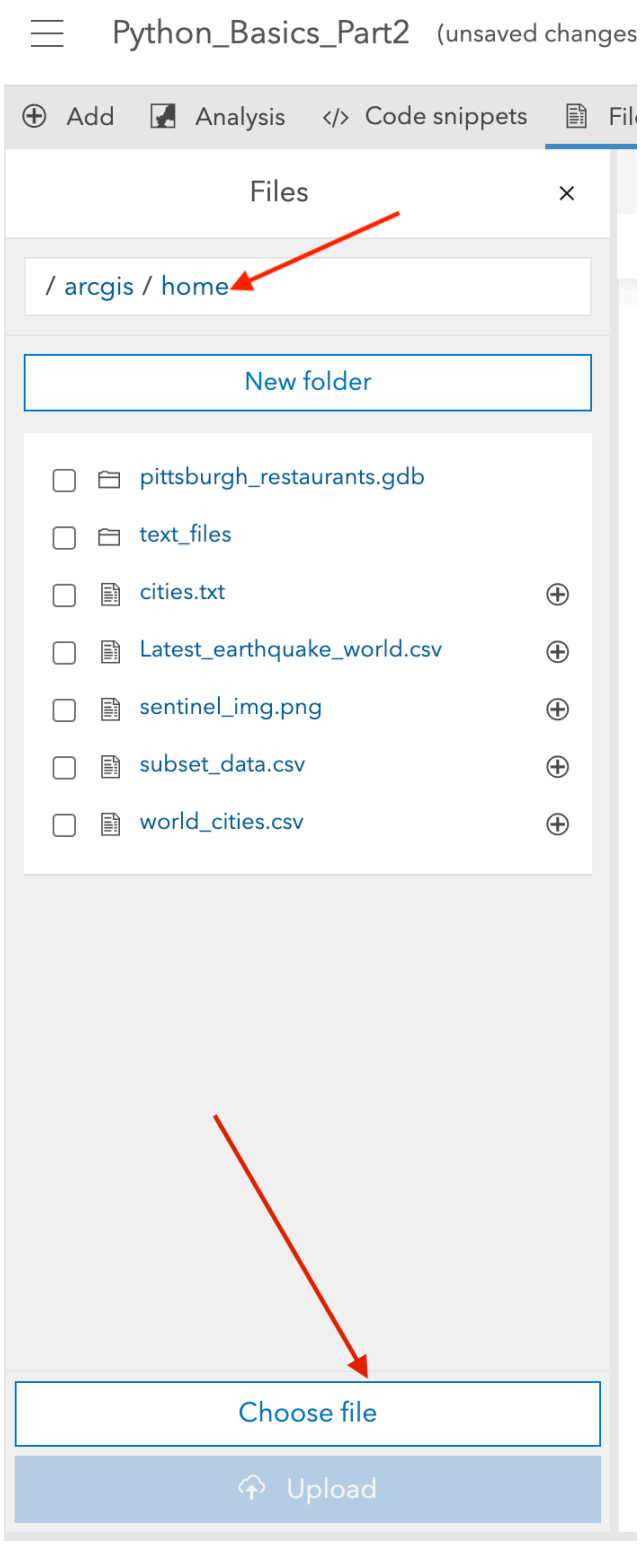
```
txt1 = "D:/data/text_files/t1.txt" #Must change backslash to forward slash
txt1 = "D:\\data\\text_files\\t1.txt" #Or, use the backslash escape character
```

For the following instructions you need to upload certain datasets to your Cloud GIS. In case you are working locally, the workflow is similar, you need to provide the appropriate path. For now we will upload the data into you Cloud GIS, so you can read it using the notebook hosted in ArcGIS Online.

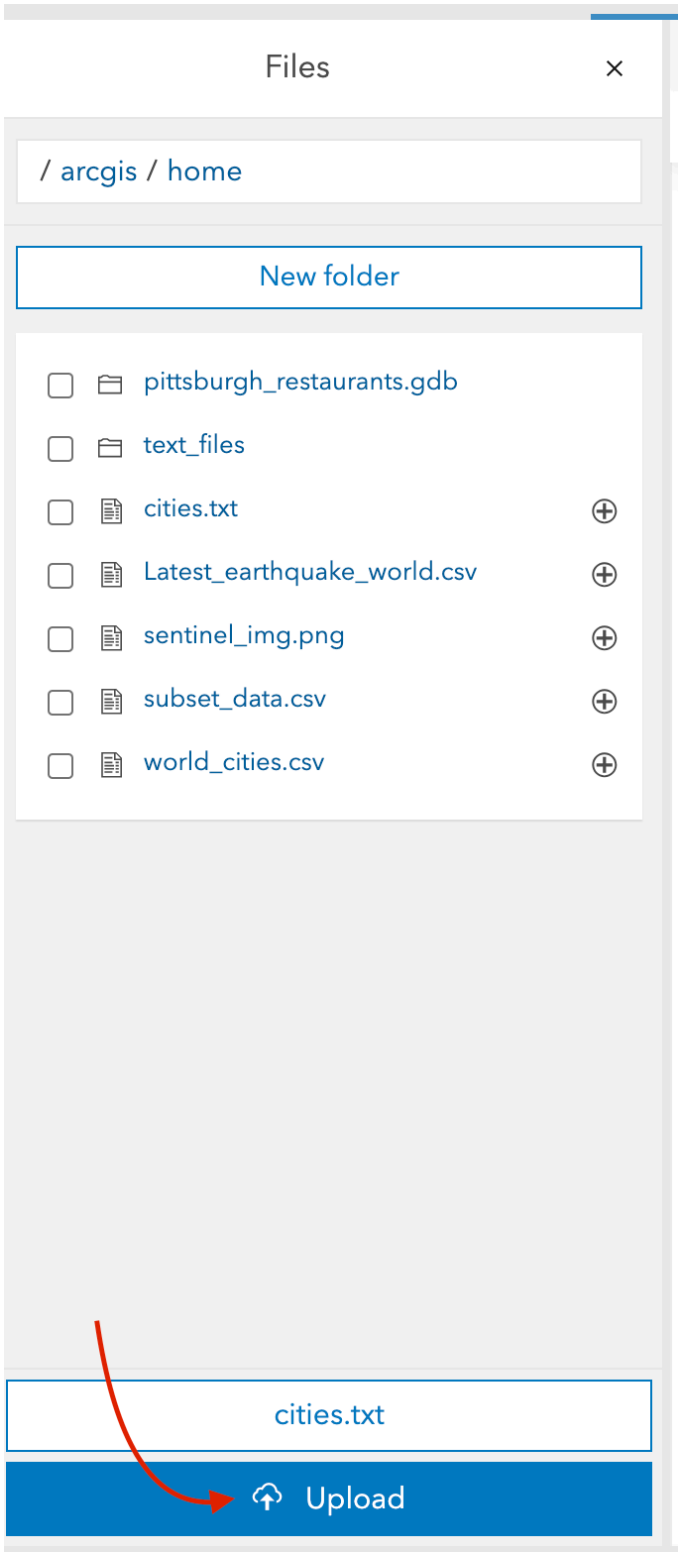
In this Notebook, click in Files.


Files
/ arcgis / home
New folder
pittsburgh_restaurants.gdb
text_files
cities.txt
Latest_earthquake_world.csv
sentinel_img.png
subset_data.csv
world_cities.csv
cities.txt
Upload

Download the **data** folder from Moodle, then click in **home**, then click in **Choose file** and select **cities.txt**


Python_Basics_Part2 (unsaved changes
Add
Analysis
Code snippets
Files
/ arcgis / home
New folder
pittsburgh_restaurants.gdb
text_files
cities.txt
Latest_earthquake_world.csv
sentinel_img.png
subset_data.csv
world_cities.csv
Choose file
Upload

Once you have chosen the file, then name will appear to confirm, now clic Upload.


Files
/ arcgis / home
New folder
pittsburgh_restaurants.gdb
text_files
cities.txt
Latest_earthquake_world.csv
sentinel_img.png
subset_data.csv
world_cities.csv
cities.txt
Upload

Now you have the **cities.txt** in your portal and now you can call the call., if you click in the **plus** symbol you will get a new cell from ArcGIS Notebook to help you to understand what is the correct way to call this dataset. For future instructions make your you have uploaded the required dataset, from Data folder included in Moodle.

To read a text file you can just use the module with, like this:

```
with open('/arcgis/home/cities.txt', "r") as f:
    contents = f.read()
    print(contents)
```

In the following example, we first import the **csv module**, which provides functionality for working with CSV files.

We then use the **with** statement to open the file "world_cities.csv" in read mode (r) and assign the file object to the variable csvfile.

We pass this file object to the csv.reader function to create a reader object that can be used to iterate over the rows of the CSV file.

Inside the with block, we loop over each row in the CSV file using a for loop, and print out each row using the print function.

Note that each row is represented as a list of strings.

Now, upload the **world_cities.csv** file, using the same process we described earlier.

```
import csv

with open('/arcgis/home/world_cities.csv', "r") as csvfile:
    reader = csv.reader(csvfile)
    for row in reader:
        print(row)
```

Specific modules or libraries will allow you to read in and work with specific types of data. For example, in the code below you are using **Pandas** to read in a comma-separated values (CSV) file as a Pandas DataFrame. We will discuss **Pandas** in a later part of this module.

In case you did not notice we use the keyword **import** to literally import the functionally of an external package that requires previous installation. For now we can import it without any issue. In case you need to import a new module that has not been included in your initial setup, you can still install it first, then make the import.

```
import numpy as np #Import numpy package, but then we said as np just for quick reference
import pandas as pd #Import pandas package, but then we said as pd just for quick referenc
```

```
# Sometime get the rigth path is tricky, so before entering to the function, you can add a
cities_df = pd.read_csv('/arcgis/home/world_cities.csv', sep=",", header=0, encoding="ISO-
cities_df.head(10)
```

You can also use other modules like **matplotlib** to read and plot external files like an image, like this:

```
# importing required libraries
import matplotlib.pyplot as plt
import matplotlib.image as img

# reading the image
testImage = img.imread('/arcgis/home/sentinel_img.png')

# displaying the image
plt.imshow(testImage)
```

Working with Directories

Instead of reading in individual files, you may want to access entire lists of files in a directory. The example below demonstrates one method for accomplishing this using the **os** module and **list comprehension**. Specifically, it will find all TXT files in a directory and write their names to a list.

Only the file name is included in the generated list, so I use additional list comprehension to add the full file path and generate a new list.

```
import os

direct = '/arcgis/home/text_files'

files = os.listdir(direct)
files_txt = [i for i in files if i.endswith('.txt')]
print(files_txt)

txtlst = [direct + s for s in files_txt]
print(txtlst)
```

The code below demonstrates three other methods for reading in a list of TXT files from a directory. The first method uses the *listdir()* method from the **os** module, the second uses the *walk()* method form the **os** module (which allows for recursive searching within subdirectories), and the last method uses the **glob** module.

You will see many other examples in this course of how to read files and lists of file names.

```
from os import listdir

def list_files1(directory, extension):
    return (f for f in listdir(directory) if f.endswith('.' + extension))

from os import walk

def list_files2(directory, extension):
    for (dirpath, dirnames, filenames) in walk(directory):
        return (f for f in filenames if f.endswith('.' + extension))

from glob import glob
from os import getcwd, chdir

def list_files3(directory, extension):
    saved = getcwd()
    chdir(directory)
    it = glob('*.' + extension)
    chdir(saved)
    return it

direct = '/arcgis/home/text_files/'
method1 = list(list_files1(direct, "txt"))
method2 = list(list_files2(direct, "txt"))
method3 = list_files3(direct, "txt")
print(method1)
print(method2)
print(method3)
```

f-Strings

In Python, **f-strings** can be used to format printed strings or include variables within strings. This technique can be useful for generating well-formatted and useful print statements or dynamically using variables in printed strings.

In the first example, I am printing x but with formatting defined to round to two decimal places. In the next example, I multiply x by itself in the printed string. Note that f-strings will begin with f followed by the statement in parenthesis. Variables or code will be placed in

curly brackets, and formatting can be defined using syntax after a colon. In the first example, ":2f" indicates to round to two decimal places.

In the third example, I am calling x in a statement while in the fourth example I am using the *.upper()* method to convert the string to all uppercase in the printed statement. Lastly, I have edited the *enumerate()* example above with an f-string to print the country name and assigned index in the for loop.

Throughout this course, you will see examples of f-strings for providing better formatted and/or more interpretable print output.

```
x = 0.123456789
print(f'The value is {x:.2f}')
x = 6
print(f'{x} times 2 is equal to {x*x}')
x = "blue"
print(f'My favorite color is {x}.')
x = "blue"
print(f'My favorite color is {x.upper()}.')

countries = ["Belgium", "Mexico", "Italy", "India"]
for index, country in enumerate(countries):
    print(f'The index for {country} is {index}.')
```

What's next

With these two notebooks(Part No1, and Part No2) you got a comprehensive list of components that you now can use to write python code. Now it is just a matter of practice and more practice.

Initially you will think this is extremely complicated or long, but with the exercises you will realised that is actually pretty simple.

Like I said before, if you have any particular question aks the available TA in the room.

Well done, now you have covered the basics for python it is time to PRACTICE a bit with less guidance.

Open the **Exercises_PythonBasics** and complete all the challenges there.

Data Sources

This is a list of web portals where you can access open and authoritative **geospatial data** for the UK and Scotland. These sources can be used to download shapefiles, connect to live web services, and enrich your **ArcGIS Online** projects with meaningful local datasets.

---

1. Scottish Spatial Data Infrastructure (SSDI)

The official portal for discovering and accessing spatial datasets from Scottish public bodies.

Data Available: - Administrative boundaries - Environmental and natural heritage data - Planning and infrastructure - Marine and coastal datasets - Data formats: Shapefiles, GeoJSON, WMS, WFS

---

2. Scotland's Environment Web

A partnership platform providing environmental datasets and interactive maps.

Data Available: - Land cover and land use - Air and water quality - Biodiversity and habitats - Climate and emissions - Tools: Map viewers, WMS services, downloadable shapefiles

---

3. Spatial Hub (Improvement Service)

Aggregates and publishes spatial data provided by all 32 Scottish local authorities.

Data Available: - Planning applications - Housing land audits - School catchments - Local development plans - Data formats: Shapefiles, WMS, GeoJSON (registration may be required)

---

4. UK Government Data Portal (data.gov.uk)

The central open data portal for the UK government.

Data Available: - Transport networks - Health and social care - Demographics and census - Crime and safety - Environment and energy - Formats: CSV, GeoJSON, Shapefiles, APIs, WMS/WFS services

---

5. Ordnance Survey OpenData

The UK's national mapping agency providing a range of open and premium geographic datasets.

Data Available: - OpenMap Local (general-purpose vector mapping) - OS Open Roads, OS Open Rivers - Boundary-Line (administrative boundaries) - Access via: Downloads, APIs, and ArcGIS-ready formats

---

6. National Records of Scotland (NRS) Geography

Provides the official statistical geographies for Scotland.

Data Available: - Data zones and Intermediate Zones - Census output areas - Health boards, local authorities - Formats: Shapefiles, GeoJSON

---

7. Office for National Statistics (ONS) Geography

Geospatial portal from the ONS offering boundary data and census geography.

Data Available: - Statistical geographies (LSOA, MSOA) - Census 2011 and 2021 boundaries - Parliamentary constituencies - Downloadable in ESRI Shapefile and GeoPackage formats

---

8. DEFRA Data Services Platform

UK Government's portal for environment-related data and services.

Data Available: - Flood risk zones - Agricultural land classification - River networks and water quality - Waste and recycling facilities - Access via: Shapefiles, APIs, WMS/WFS

---

9. OpenStreetMap (Geofabrik UK Extracts)

Extracts from OpenStreetMap for Great Britain, including Scotland.

Data Available: - Buildings, highways, land use, points of interest - Routable and editable map data - Formats: .osm.pbf, shapefiles (via tools like osmconvert or QGIS plugins)

---

10. Edinburgh GeoPortal

A geospatial data repository from Edinburgh with open datasets.

Data Available: - Local and global environmental data - Terrain and elevation - Land cover and vegetation indexes - Datasets relevant for climate change, ecology, and earth observation

---

11. Glasgow GeoPortal

A geospatial data repository from Glasgow with open datasets.

Data Available: - Local and global environmental data - Terrain and elevation - Land cover and vegetation indexes - Datasets relevant for climate change, ecology, and earth observation

---

12. ArcGIS Living Atlas (UK content)

ESRI's curated collection of geographic information, including UK-specific content.

Data Available: - Demographics, base maps, boundaries - Real-time environmental data - Accessible directly from ArcGIS Online for instant use

---

13. ArcGIS Living Atlas (UK content)

ESRI's curated collection of geographic information, including UK-specific content.

Data Available: - Demographics, base maps, boundaries - Real-time environmental data - Accessible directly from ArcGIS Online for instant use

---

14. Urban Big Data Centre

Urban Big Data Centre is a dynamic national research hub and data service, championing the use of smart data to inform policymaking and enhance the quality of urban life.

Data Available: - Transport and Mobility, Housing and property, Labor Market, Environment.

Tips for students

- Many of these platforms offer **shapefiles**, **WMS** or **ArcGIS REST endpoints**, which can be added directly to your **ArcGIS Online** web map.
- Make sure to always **cite the data source** in your apps or reports.
- For reproducibility, **record the download date** and dataset version.
- Use **filtering and geoprocessing tools** in ArcGIS Online to tailor data to your study area.

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.