00Setup

November 29, 2023

0.1 # Python for Geospatial Data Science

Dr. Gunnar Mallon (g.mallon@rug.nl), *Department of Cultural Geography (Faculty of Spatial Science), University of Groningen*

0.2 Setting Up Your Python Environment

0.2.1 Why Python?

Python is a popular and versatile programming language that is widely used in various fields such as data science, web development, and scientific research. It is known for its simplicity and readability, making it an ideal language for beginners. Python has a large and active community, which means there are plenty of resources and libraries available for developers. In this course, we will explore the fundamentals of Python programming and how it can be applied geospatial data science.

0.2.2 Installation of Miniconda

Miniconda is a lightweight version of Anaconda, a popular Python distribution. We will be using Miniconda for this course because it provides a minimal installation of Python and allows us to easily manage packages and environments. To install Miniconda, follow these steps:

1. Go to the Miniconda website (https://docs.conda.io/en/latest/miniconda.html) and download the appropriate installer for your operating system (Windows, macOS, Linux).
2. Run the installer and follow the instructions to complete the installation.
3. During the installation, make sure to select the option to add Miniconda to your system's PATH variable. This will allow you to use Conda from the command line.

0.2.3 Creating a Conda Environment

A Conda environment is a self-contained directory that contains a specific version of Python and any additional packages you need for your project. Environments are useful because they allow you to isolate your projects and avoid conflicts between different packages and versions. To create a new Conda environment, follow these steps:

1. Open a terminal or command prompt.
2. Go to the directory that you would like to use for the course (ask for help if you are unsure about this step)
3. Run the command `conda create --name geospatial` to create a new environment named "geospatial". You can replace "geospatial" with any name you prefer.

4. Activate the environment by running the command `conda activate geospatial`. This will change your prompt to indicate that you are now working within the "geospatial" environment.

0.2.4 Installing Jupyter Notebook

We will be using **Jupyter Notebook** throughout this course. Jupyter Notebook is an interactive computing environment that allows you to create and share documents that contain live code, equations, visualizations, and narrative text. It is a powerful tool for data analysis, prototyping, and teaching. To install Jupyter Notebook within your Conda environment, follow these steps:

1. Make sure your Conda environment is activated.
2. Run the command `conda install jupyter notebook` to install Jupyter Notebook and its dependencies.

0.2.5 Starting Jupyter Notebook

To start Jupyter Notebook, follow these steps:

1. Open a terminal or command prompt.
2. Make sure your Conda environment is activated.
3. Run the command `jupyter notebook`. This will start the Jupyter Notebook server and open a new tab in your web browser.
4. In the Jupyter Notebook interface, you can create a new notebook or open an existing one, like the ones provided with this course. Notebooks have the file extension ".ipynb".

0.2.6 Accessing Course Materials

You can find all of the course materials on Brightspace. To access them, follow these steps:

1. Go to the course area on Brightspace.
2. Go to the content for week 1.
3. Download the course materials to and extract it to the folder where you have created your conda environment. (As always, ask for help if you are unsure about this step)
4. Open Jupyter Notebook (see above) and navigate to the directory where you saved the course materials.
5. Click on the notebook file to open it.

It is important to keep the course materials organized and easily accessible within your Jupyter environment. You can create folders to organize your notebooks and other files.

0.2.7 Q&A and Troubleshooting

During the session, you are encouraged to ask questions and seek clarification on any installation or setup issues you encounter. If you are facing any problems, here are some troubleshooting tips:

1. Make sure you have followed the installation instructions correctly.
2. Check that you have the correct version of Miniconda for your operating system.
3. If you encounter any error messages, try searching for the error online or simply ask for help.
4. If all else fails, you can uninstall Miniconda and reinstall it following the installation instructions.

Remember, it is normal to encounter some challenges when setting up your Python environment, but with patience and practice, you will become more comfortable with the process.

01Introduction

November 29, 2023

Dr Gunnar Mallon (g.mallon@rug.nl), *Department of Cultural Geography (Faculty of Spatial Science), University of Groningen*

0.1 General information

This is a Jupyter Notebooks. This particular notebook is designed to introduce you to a few of the basic concepts of programming in Python. The contents of this document are divided into cells, which can contain Markdown-formatted text, Python code, or raw text. You can execute a snippet of code in a cell by pressing **Shift-Enter**.

You can create new cells above or below existing cells or delete an existing cell by clicking on the left side of the cell to select it and the using "a" (above) or "b" (below) to create new cells. If you select a cell and press "x", you will delete it (so be careful!)

When learning a new programming language, it is customary to first learn how to print 'Hello World!'. While a bit quirky, it is a useful first step to know how to send input to the program and where to see the output. In Python, you can use the built-in `print()` function to print the greeting `"Hello World!"`

type the following command in the empty cell below and press **Shift-Enter** to execute the cell.

```
print("Hello World!")
```

From now on, you can always press **Shift-Enter** to execute your current cell

```
[ ]:
```

How did that go? If everything went well, you should have seen "Hello World!" printed below the cell. This is the cell output. If it worked, you are all set to embark on your Python journey.

How cool is this? You have now executed your first bit of code in Python and are ready for the wonderful world of coding and programming!

If you did not see *Hello World!* printed in the cell above, then try retracing your steps and making sure that you didn't add any extra characters.

0.2 Getting help

If you are stuck with your programming there are many ways that you can get help.

Python has inbuilt help functions, such as the ? function. If you were unfamiliar with the syntax of the `print()` function, you can simply type `? print` and execute the cell (remember, Shift-Enter) and you'll get a little help text on the print function.

Try it out yourself. In the cell below, type `? print` and execute the cell.

```
[ ]:
```

Now, this might look like a lot of gobblygook, but if you break it down, it's quite easy.

The help text tells us that the print function takes a series of arguments. In our `Hello World!` example, this was simply the text "Hello World!". As you will discover later in the course, you can have a whole range of items to be printed here.

Next, we see that there is a whole range of other parameters that you can give the print function, such as the separator character, what to do at the end of the line, and some technical things that you don't have to worry about now.

In the `Hello World!` example, we left all of these extra parameters blank. And that's okay. But in case you wanted to change the default behaviour of the `print()` function, this is where you would get information about it.

Alternative ways to trigger the help function are

`print?` - this puts the question mark after the name of the function instead of before it

`help(print)` - this gives slightly different output, but it gives you all of the important information.

Go on. Try these out yourself. Some common functions in Python are **print**, **input**, **range**, **sorted**, and **len**. Have a go at getting information and help on these in the cell below.

Note: you can execute more than one command in a cell. So try to get more than one help command in the same cell.

```
[ ]:
```

0.2.1   Other places to get help

It goes without say that there are many fantastic tools out there to get help when you are stuck programming.

Stack Overflow   One of the most used sites for any sort of programming help is called Stack Overflow (this is a term from memory management in case you really wanted to know). Stack Overflow is a great resource for solutions to problems. It's also a great place to ask for help, though Generative AI is slowing replacing Stack Overflow.

ChatGPT | **Bard** | **...**   Generative AIs, especially ChatGPT, are a fantastic resource for any Python-related questions. You can easily query these generative AI platforms to explain concepts or functions to you. One of the best uses for the platforms is to help with debugging.

When you come up against a problem, try to solve it first (this is how you learn!). If, after extensive trying, you still can't figure out what is wrong with the code, ask for help, or ask the AI. Often the AI will immediately pinpoint the problem.

Note: Do not rely on the AI to write the code for you! It can contain bugs and completely defeats the purpose of taking this course. If you blindly trust the AI, you will not learn how to program yourself. AI tools are a great aid (like a senior developer), but you still need to do the learning and we all have to use AI tools responsibly.

0.2.2 Going beyond this course

Programming is all about trial and error (learning from debugging your code) and repetition. It's a constant problem-solving endeavour.

If you really want to get good at programming or this course has inspired you to dive further into Python, I highly recommend that you check out a website called **CodeWars**.

CodeWars (no affiliation) presents you with seemingly unlimited puzzles and problems to solve that are constantly adjusted to your level or programming. It's a free website, and if you're serious about improving your programming, you should definitely check it out.

0.3 Comments in Python

Commenting your code is a vital aspect of programming. Comments allow developers to leave notes for themselves, colleagues, or future contributos, and can explain the purpose, functionality, or intricacies of a code segment. While the Python interpreter ignores comments when executing a script, they can be invaluable for human readers.

0.3.1 Types of Comments in Python

Python provides several ways to add comments to your code:

1. Single-line comments These are the most common types of comment. You can add a single-line comment using the `#` symbol. Everything after this symbol (on the same line) will be considered a comment and won't be executed.

```python
# This is a single-line comment
print("This is not a comment.")  # This is a comment
```

Now it's time for you to try it yourself. In the cell below write a comment and then execute the cell. See if the outcome is what you thought it would beinline comment

```
[ ]:
```

0.3.2 2. Multi-line comments

While Python doesn't have explicit support for multi-line comments, programmers use a workaround by employing multi-line strings as comments. These are typically done using triple quots.)

```python
'''
This is a
multi-line
comment.
'''
```

```
"""
Another example of a
multi-line
comment.
"""
```

```
print("This is not a comment.")
```

Note: Even though using triple quotes as multi-line comments is a common practice, they are technically still strings. Therefore, if not assigned to a variable or used in a manner where the string is processed (e.g., as a docstring), it will simply remain in the code without any impact.

Go on, try it yourself. Just like in the previous example, create a multi-line comment alongside a print statement and execute the cell.

```
[ ]:
```

0.3.3 Why is Commenting Important?

- **Code Understanding**: Comments help developers understand the purpose and functionality of specific code segments, especially when revisiting code after a long interval.

- **Collaboration**: When working on team projects, comments ensure that colleagues can grasp the function and intention of your code, reducing the time required to understand it.

- **Debugging**: Comments can also be useful for leaving notes or to-do items for debugging or refining code.

- **Documentation**: Certain tools extract comments from the code to automatically generate documentation. This is common for library or API development.

0.3.4 Best Practices

- Be Clear and Concise: Ensure your comments are direct and provide clarity. If the comment requires more than a couple of sentences, it might be worth revisiting the code to make it more self-explanatory.

- Avoid Redundancy: Avoid comments that state the obvious, e.g., x = 5 # Assigning 5 to x.

- Stay Updated: Ensure comments are updated if the code is changed. Outdated comments can be more confusing than no comments at all.

In conclusion, while well-written code should be as self-explanatory as possible, comments are essential for clarification, collaboration, and documentation. Use them wisely!

0.4 Markdown

Markdown is a lightweight markup language with a plain-text-formatting syntax. Its key design goal is readability – that the language be readable as-is, without looking like it's been marked up with tags or formatting instructions, unlike more robust markup languages such as HTML.

You can create a markdown cell in Jupyter Notebooks, but selecting "Markdown" from the drop-down menu above. Alternatively, you can select a cell, press "Esc" and then "M" to change it to markdown. Simply double-click the cell to edit it.

In Markdown you can define headers, lists, code snippets, pictures, etc..

As a matter fact, this entire tutorial is written in markdown. If you double click any of the text cells you can see the markdown language underneath it.

For a full list of markdown elements that you can use have a look at the Markdown Guide - Cheat Sheet.

Now it's up to you. Have a look through the *cheat sheet* and then have a **play around** in the cells below. You will soon realise that programming is all about learning a new concept and then playing around with it until it becomes second nature.

[ ]:

[ ]:

[ ]:

0.5   What's next?

Well done on making it so far. If you have followed all the steps and played around with commenting, printing, markdown, and getting help, you are one step closer to mastering Python.

There are many useful resources out there that you might want to refer to throughout the Geospatial Data Science course:

- The Official Python Documentation
- Jupyter.org
- CodeWars

When you're all set, it is now time to move on to **Notebook 2:   Variables** and learn about how to store and work with data in Python.

[ ]:

02Variables

November 29, 2023

Dr Gunnar Mallon (g.mallon@rug.nl), *Department of Cultural Geography (Faculty of Spatial Science)*, *University of Groningen*

Table of Contents

0.1 What are Variables?

In the world of programming, a variable is like a container that can hold different types of data. Think of it as a box with a label on it, and you can put different things in the box (data) and change what's inside the box as needed. Variables are essential because they allow us to store and manipulate data in our programs.

0.1.1 Understanding the concept of variables

Variables are used to store data that your program needs to work with. This data can be numbers, text, or even more complex information like lists of items or entire documents.

They give symbolic names to values (such as: first_name). Instead of directly working with data values like numbers or text, you use variable names to refer to those values. This makes your code more readable and understandable

One of the most important aspects of variables is that you can change their values during the execution of a program. This dynamic behavior allows your program to respond to different situations and perform calculations.

0.1.2 Why variables are important in programming

Variables are fundamental in programming for several reasons:

- *Storage of Data* - They allow us to store and manage data efficiently, making it accessible for processing and manipulation.

- *Readability* - Using variables with meaningful names makes your code easier to read and understand. Imagine trying to decipher a program where every number or piece of text is used directly without variable names—it would be like reading a foreign language.

- *Flexibility* - Variables make your code flexible. You can change the value a variable holds, and the rest of your program can adapt accordingly. This adaptability is crucial when creating complex software.

- *Code Reusability* - Variables allow you to reuse data values multiple times in your code. Instead of retyping the same value, you can use the variable name wherever you need it.

Let's look at some practical examples of variables in Python:

Example 1: Storing an integer

In this example, we store the value 25 in a variable named age.

```python
# Storing an integer in a variable named 'age'
age = 25
```

Example 2: Storing a string

Here, we store the string "Alice" in a variable named name.

```python
# Storing a string in a variable named 'name'
name = "Alice"
```

Example 3: Changing a variable's value

We can update the value of the age variable from 25 to 30.

```python
# Changing the value of 'age'
age = 30
```

Example 4: Variable naming

These examples show the use of descriptive variable names with the `user_name` and `user_age` variables. We will look at naming your variable in a bit more detail in a bit.

```python
# Variable names
user_name = "Bob"
user_age = 28
```

By using variables, you can easily store, manipulate, and change data in your Python programs, making them dynamic and powerful tools for various tasks. OK, here is a little challenge for you.

In the cell below, *define a variable to hold the name of the current month and then **print** it out. Tip: remember what you learned in lesson 1.*

[ ]:

0.2 Data Types

In Python, data types represent the kind of data that a variable can hold. Understanding data types is crucial because it determines how your data is stored in memory, what operations you can perform on it, and how the data behaves. Python has several built-in data types, and each has specific characteristics and use cases.

Python is dynamically typed, which means that you don't need to explicitly state the data type when creating a variable; Python infers the data type based on the value assigned to it.

Data types are essential because they:

- Determine the range and precision of values that a variable can hold.
- Define the operations that can be performed on the data.
- Affect memory usage and performance.
- Ensure data integrity and accuracy in your programs.

0.2.1 Common Data Types in Python

Python has several built-in data types. Let's explore some of the most common ones:

Integer (int) Integers represent whole numbers, positive or negative, without any decimal point. Integers represent whole numbers, and they can be positive or negative. You can perform mathematical operations like addition, subtraction, multiplication, and division on integer variables.

```python
# Integer variables
age = 25
count = -10
```

Float (float) Floats are used to represent numbers with a decimal point. Floats are used for numbers with decimal points. They can represent real numbers with great precision. You can perform all arithmetic operations on float variables.

```python
# Float variables
temperature = 98.6
pi = 3.14159
```

String (str) Strings represent text or sequences of characters enclosed in single or double quotes. Strings are used to represent text. You can manipulate strings by concatenating them, slicing them, and performing various string-related operations.

```python
# String variables
name = "Alice"
sentence = 'Hello, World!'
```

Boolean (bool) Booleans represent binary values, either True or False. They are often used for logical operations. Booleans have only two possible values, True or False. They are used in conditional statements and logical operations to control program flow.

```python
### Boolean variables
is_student = True
is_open = False
```

By understanding and using these common data types, you can work with a wide range of data in Python programs. Data types help ensure that your code behaves as expected and produces accurate results, making them a fundamental concept in programming.

Now it's time for you to try this out yourself in order to get used to declaring and changing the value of variables:

In the cell below, define five variables to hold data that you collected from a simple questionnaire. You can choose which data you collect and how you want to name the variables.

Once you have defined the variables, print these out. Then change the values of the variables to represent the answers of another respondent and print these out again.

```
[ ]:
```

0.3 Variable Assignment

Variable assignment is the process of giving a variable a value. It allows you to store data in a variable for later use in your Python programs. In this section, we will delve deeply into variable assignment, providing extensive explanations and numerous examples.

0.3.1 Assigning Values to Variables

In Python, you can assign values to variables using the assignment operator =. The variable on the left side of the = sign is assigned the value on the right side. Here's how it works:

```python
# Assigning a value to a variable
variable_name = value
```

`variable_name`: The name of the variable you want to create or update.

`value`: The data you want to store in the variable.

0.3.2 Basic Syntax for Assignment

- **Variable Name** - Variable names must start with a letter or an underscore (_). After the initial character, variable names can contain letters, numbers, and underscores. Variable names are case-sensitive, meaning myVar and myvar are considered different variables. Variable names cannot be Python keywords (e.g., if, for, while).

- **Assignment Operator** - The assignment operator = is used to assign a value to a variable. It does not mean "equals" in the mathematical sense but rather "assigns."

Now, let's explore various examples of variable assignment:

Example 1: Assigning an Integer

Here, we assign the integer value 25 to the variable age. After this assignment, you can use age to refer to the value 25.

```python
# Assigning an integer to a variable
age = 25
```

Example 2: Assigning a Float

In this example, the variable temperature is assigned the float value 98.6. You can perform mathematical operations on this variable.

```python
# Assigning a float to a variable
temperature = 98.6
```

Example 3: Assigning a String

The variable name is assigned the string "Alice". You can manipulate this string variable, such as concatenating it with other strings.

```python
# Assigning a string to a variable
name = "Alice"
```

Example 4: Assigning a Boolean

Here, the variable is_student is assigned the boolean value True. Booleans are commonly used in conditional statements.

```python
# Assigning a boolean to a variable
is_student = True
```

Example 5: Reassigning a Variable

You can change the value stored in a variable by simply assigning a new value to it. In this case, we update the age variable from 25 to 30.

```python
# Reassigning a variable with a new value
age = 30
```

Example 6: Multiple Assignments

Python allows you to assign values to multiple variables in a single line. Here, we assign 10 to x, 20 to y, and 30 to z.

```python
# Multiple assignments in one line
x, y, z = 10, 20, 30
```

Example 7: Swapping Variables

You can swap the values of two variables without using a temporary variable. This is achieved by simultaneous assignment, as shown in the example.

```python
# Swapping the values of two variables
a = 5
b = 10
a, b = b, a
```

Example 8: Underscore as a Variable Name

While not common, you can use an underscore as a variable name. It's often used as a placeholder for values that you don't plan to use.

```python
# Using an underscore as a variable name
_ = "This variable is not typically used"
```

Example 9: Constants

Constants are variables whose values should not be changed throughout the program. By convention, constant variable names are written in uppercase with underscores.

```python
# Declaring constants with uppercase variable names
PI = 3.14159
MAX_SCORE = 100
```

Example 10: Dynamic Typing

Python is dynamically typed, meaning you can change the type of data that a variable holds during the execution of a program. In this example, dynamic_var first holds an integer and then a string.

```python
# Dynamic typing in Python
dynamic_var = 42
dynamic_var = "Hello, World!"
```

In this section, we've explored the fundamental concept of variable assignment. You've learned how to assign values to variables, and the rules for variable names, and seen numerous examples that illustrate the various aspects of variable assignment in Python. Understanding variable assignments is a crucial foundation for writing Python programs and manipulating data.

In the cell below, have a play around with assigning and chaning the values of variables and their data types.

[ ]:

0.4   Variable Naming

Variable naming is an essential aspect of writing clean and readable code. It involves choosing meaningful and descriptive names for your variables. In this section, we will delve deeply into variable naming conventions, guidelines, and best practices, providing extensive explanations and numerous examples.

0.4.1   Variable Naming Rules and Conventions

To use variables effectively, you need to follow some rules and naming conventions:

Rules for Variable Names * Variable names must start with a letter or underscore (_). * After the initial character, variable names can include letters, numbers, and underscores. * Variable names are case-sensitive (e.g., myVar and myvar are different variables). * They cannot be Python keywords (e.g., if, for, while).

Naming Conventions * Use descriptive names that indicate the variable's purpose (e.g., user_age instead of ua). * Use lowercase letters and separate words with underscores (snake_case) for readability. * Be consistent in your naming style throughout your code. * If a variable represents a constant (a value that doesn't change), use uppercase letters with underscores (e.g., MAX_SCORE).

0.4.2   Guidelines for Naming Variables

Descriptive and Meaningful Names   Choose variable names that clearly indicate the purpose or meaning of the data they hold. This makes your code more self-explanatory and easier for others (and your future self) to understand.

In this example, `user_age` is a meaningful name that indicates the variable's purpose.

```python
# Good variable name
user_age = 25
```

Use English Words   Use English words for variable names. While Python allows non-English characters, using English words ensures consistency and readability, especially when working on projects with others.

Here, `name` is a clear and concise English word that represents the variable's content.

```python
# Good variable name
name = "Alice"
```

Follow a Consistent Style   Consistency in variable naming is crucial. Choose a naming style (e.g., camelCase, snake_case) and stick to it throughout your codebase. PEP 8, Python's style guide, recommends using snake_case for variable names.

Avoid abbreviations that may be unclear to others. Choose descriptive names over cryptic abbreviations.

```python
# Using snake_case
user_name = "Bob"
```

Avoid Abbreviations

```python
# Unclear variable name
usr_nm = "Charlie"
```

Instead, use:

```python
# Descriptive variable name
user_name = "Charlie"
```

Be Mindful of Length   Variable names should be neither too short nor excessively long. Aim for a balance between brevity and descriptiveness. Avoid single-character variable names except for loop counters.

```python
# Overly long variable name
this_is_a_long_variable_name_that_should_be_shortened = 42
```

Instead, use:

```python
# Balanced variable name
result = 42
```

Naming Conventions   snake_case

In snake_case, words are separated by underscores. It's the most common naming convention in Python and is recommended for variable names in this course.

```python
# Using snake_case
user_name = "David"
```

camelCase

In camelCase, each word starts with a capital letter, except the first word. While it's less common in Python, some developers use it, especially in languages like JavaScript.

```python
# Using camelCase
userName = "Eve"
```

Choosing Descriptive Variable Names   Avoid Generic Names

Avoid generic names like temp, value, or data. Use names that convey the specific role or purpose of the variable.

```python
python# Generic variable name temp = 25
```

Instead, use:

```python
# Descriptive variable name
room_temperature = 25
```

Use Plural for Collections

When a variable represents a collection or a list of items, consider using a plural form for the variable name.

```python
# Singular variable name for a list
fruit = ["apple", "banana", "cherry"]
```

Instead, use:

```python
# Plural variable name for a list
fruits = ["apple", "banana", "cherry"]
```

Add Context

Include additional context if necessary to clarify the variable's purpose. In this example, file_path provides context about the variable's content.

```python
# Adding context to variable names
file_path = "/path/to/file"
```

In this section, we've explored the importance of variable naming, providing detailed guidelines, conventions, and best practices. By following these principles, you can write code that is not only correct but also clean, readable, and maintainable. Choosing meaningful and descriptive variable names is a fundamental skill for any programmer, as it greatly enhances code understanding and collaboration.

Go on, have a play around with defining variables with names that follow the rules and guidelines mentioned above.

[ ]:

0.5 Variable Operations

In Python, variables not only store data but also allow you to perform various operations on them. This section will provide an extensive exploration of different types of operations that can be performed on variables.

0.5.1 Basic Arithmetic Operations

Python supports standard arithmetic operations for numerical variables. These operations include addition, subtraction, multiplication, division, and modulus.

Addition (+)

```python
# Addition
x = 5
y = 3
result = x + y   # result will be 8
```

Subtraction (-)

```python
# Subtraction
x = 10
y = 7
result = x - y   # result will be 3
```

Multiplication (*)

```python
# Multiplication
x = 6
y = 4
result = x * y   # result will be 24
```

Division (/)

```python
# Division
x = 20
y = 5
result = x / y   # result will be 4.0 (float)
```

Modulus (%)

Modulus (or remainder) is used to find the remainder when one value is divided by another.

```python
# Modulus
x = 17
y = 5
result = x % y   # result will be 2
```

Calculate the area of a rectangle with a length of 12 units and a width of 5 units.

Find the average of three numbers: 25, 30, and 35.

[ ]:

0.5.2 String Operations

Strings in Python support various operations for manipulation, concatenation, and formatting.

Concatenation (+)

Concatenation is the process of combining two or more strings.

```python
# Concatenation
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name   # full_name will be "John Doe"
```

String Interpolation (f-strings) (more about this later)

String interpolation allows you to embed variables or expressions within strings.

```python
# String interpolation
name = "Alice"
age = 30
message = f"My name is {name} and I am {age} years old."
```

We will cover string interpolation in more depth in lesson 4.

Create a greeting message by combining your name and a greeting phrase.

Format a message that displays the price of an item as $19.99.

`[ ]:`

0.5.3 Boolean Operations

Boolean variables (`bool`) are used in logical operations.

Comparison Operators

Comparison operators are used to compare values and return True or False.

```python
# Comparison operators
x = 5
y = 10
x == y # this will be false
is_equal = x == y   # is_equal will be False
```

Common comparison operators include == (equal), != (not equal), < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to).

Logical Operators

Logical operators are used to perform logical operations on boolean values.

```python
# Logical operators
is_sunny = True
is_warm = False

# AND operator
```

```python
is_nice_day = is_sunny and is_warm   # is_nice_day will be False

# OR operator
is_weekend = True
is_good_weather = is_sunny or is_warm   # is_good_weather will be True

# NOT operator
is_raining = False
is_dry = not is_raining   # is_dry will be True
```

For `AND` operations to be true, both sides of the operation need to be true. Otherwise, the operation is false. For `OR` operations only one of the two operands needs to be true. `NOT` reverses the true|false value.

In this section, we have extensively explored the different types of operations that can be performed on variables in Python. By mastering these operations, you will gain the ability to manipulate data, perform calculations, and make logical decisions, which are fundamental skills for programming and problem-solving.

Compare two ages to determine if they are equal and print the result.

Check if a person is eligible for a discount based on their age (age > 60) and if it's a weekday (not a weekend).

```
[ ]:
```

0.6 Printing Variables

In Python, f-strings (formatted string literals) provide a convenient and powerful way to display variables and their values within strings. They allow you to embed expressions, and variables, and even perform simple operations inside string literals. This section will explore the use of f-strings for printing variables in Python.

0.6.1 Basics of f-strings

Creating an f-string

To create an f-string, start a string literal with the letter 'f' or 'F' and enclose expressions or variables within curly braces {}. In this example, {name} and {age} are placeholders for variables, and their values are substituted when the string is printed.

```python
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

Expressions within f-strings

You can include expressions inside the curly braces to perform calculations or operations. Here, {x + y} calculates the sum of x and y and displays the result within the string.

```python
x = 5
y = 3
print(f"The sum of {x} and {y} is {x + y}.")
```

0.6.2 Format specifiers

Controlling Output Format

You can use format specifiers within f-strings to control the formatting of variables, such as specifying the number of decimal places for floats. Here, the :.2f format specifier rounds the value of pi to two decimal places.

```python
pi = 3.14159265359
print(f"Value of pi (rounded to two decimal places): {pi:.2f}")
```

0.6.3 Practical examples

Formatting Dates

f-strings are useful for formatting dates and times. This code uses the {today:%Y-%m-%d} format specifier to display the date in the "YYYY-MM-DD" format.

```python
from datetime import datetime

today = datetime.today()
print(f"Today's date: {today:%Y-%m-%d}")
```

In this section, we explored the power of f-strings in Python for printing variables and formatting output. You learned how to create f-strings, include expressions, and use format specifiers to control the formatting of variables. With f-strings, you can efficiently display variables and generate custom-formatted output in your Python programs.

0.6.4 Exercises

Create an f-string to print the result of multiplying two numbers: 7 and 9.

Use an f-string to display the current time in the format "HH:MM AM/PM."

```
[ ]:
```

When you're all set, it is now time to move on to **Notebook 3: Data Structures** and learn about how to store and work with data in Python.

03Data_Structures

November 29, 2023

Dr Gunnar Mallon (g.mallon@rug.nl), *Department of Cultural Geography (Faculty of Spatial Science), University of Groningen*

Table of Contents

1 Introduction to Data Structures

Now that you have learned about variables, you are well on your way to being able to write a complete program in Python. Most of the time, when programming, you don't just want individual variables, but you want to group these together. Think of personal details, you could have individual variables `first_name`, `last_name`, and `email_address`. But it makes sense to group these together. That's where data structures come in.

Data structures are fundamental constructs used to organize, store, and manage data efficiently. They provide a way to represent, access, and manipulate data in a structured and organized manner within a computer program.

1.1 Why are Data Structures Important?

Data structures are essential in computer science and programming for several reasons:

- **Efficient Data Storage:** Data structures enable efficient storage of data, reducing memory usage and optimizing storage space.

- **Quick Data Retrieval:** Properly chosen data structures facilitate quick and efficient data retrieval, reducing the time complexity of algorithms.

- **Data Organization:** They help in organizing data logically, making it easier to manage and maintain.

- **Algorithm Design:** Many algorithms and operations depend on the choice of data structure. Choosing the right one can significantly impact the efficiency of your code.

- **Real-World Analogies:** Data structures often mimic real-world analogies, making it easier to understand and work with data.

1.2 Types of Data Structures

There are various types of data structures, each with its own characteristics and use cases. Some common data structures include:

Lists: Lists are ordered collections of elements where elements can be of different data types and can be dynamically resized. You use lists to store and manage a collection of items of varying types, such as a list of names.

Tuples: Tuples are ordered, immutable collections of elements. They are used for storing a fixed set of related data elements, representing coordinates, etc.

Sets: Sets are collections of unique elements with no specific order. Use sets for storing and managing unique elements, implementing mathematical sets.

Dictionaries: Dictionaries are collections of key-value pairs, where each key is unique and associated with a value. Dictionaries (or dict for short) are used to store data with associated metadata, such as storing user information by their username.

We will look at each of the four main data structures in more detail below. Other data structures include ones like **stacks and queues**, **linked lists**, and **trees**. However, we will not cover these in the course as that would be beyond its scope.

1.3 Lists

Lists are one of the most versatile and widely used data structures in Python. They are ordered collections that can store elements of different data types, where each element is identified by an index. Lists are versatile and can store elements of different data types, including numbers, strings, objects, and more. To create a list in Python, you enclose a sequence of elements within square brackets [], separated by commas. In this section, we'll dive deep into lists, covering their creation, manipulation, and various operations.

```python
# Creating a list of integers
my_list = [1, 2, 3, 4, 5]

# Creating a list of strings
fruits = ["apple", "banana", "cherry"]
```

Throughout this document you will find blank code cells. Use these cells to play around with the new concepts and knowledge that you've just learnt

```python
[ ]:
```

1.3.1 Accessing List Elements

You can access individual elements in a list using indexing. Python uses **0-based indexing**, where the first element has an index of 0, the second has an index of 1, and so on.

```
# Accessing elements by index
first_element = my_list[0]   # Accesses the first element (1)
second_element = my_list[1]   # Accesses the second element (2)
```

You can also use negative indexing to access elements from the end of the list:

```
# Accessing elements using negative indexing
last_element = my_list[-1]   # Accesses the last element (5)
```

1.3.2 Modifying Lists

Lists are mutable, which means you can change their contents after creation.

```
# Modifying list elements
my_list[2] = 99   # Changes the third element to 99
```

Being able to access elements of a list and modifying them is one of the most important skills in Python. Make sure to play around with this concept until it becomes intuitive!

[ ]:

[ ]:

1.3.3 List Operations

Concatenation You can concatenate two or more lists using the + operator.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
result = list1 + list2   # Concatenates the lists
```

Repetition You can replicate a list by using the * operator.

```
original_list = [1, 2, 3]
repeated_list = original_list * 3   # Repeats the list three times
```

Slicing Slicing allows you to extract a portion of a list.

```
my_list = [10, 20, 30, 40, 50]
subset = my_list[1:4]   # Retrieves elements from index 1 to 3
```

We will go into much more detail about slicing lists in Week 3. For now, make sure that you try all of these methods out in the cells below.

[ ]:

[ ]:

1.3.4 List Methods

Python provides several built-in methods to manipulate lists:

append()

The append() method adds an element to the end of a list.

```python
fruits = ["apple", "banana"]
fruits.append("cherry")  # Adds "cherry" to the end of the list
```

insert()

The insert() method inserts an element at a specified position in the list.

```python
fruits = ["apple", "cherry"]
fruits.insert(1, "banana")  # Inserts "banana" at index 1
```

remove()

The remove() method removes the first occurrence of a specified value from the list.

```python
fruits = ["apple", "banana", "cherry"]
fruits.remove("banana")  # Removes "banana" from the list
```

pop()

The pop() method removes and returns an element from a specified index or the last element if no index is provided.

```python
fruits = ["apple", "banana", "cherry"]
removed_fruit = fruits.pop(1)  # Removes and returns "banana"
```

index()

The index() method returns the index of the first occurrence of a specified value.

```python
fruits = ["apple", "banana", "cherry"]
index = fruits.index("cherry")  # Returns the index of "cherry"
```

count()

The count() method returns the number of times a specified element appears in the list.

```python
numbers = [1, 2, 2, 3, 2, 4]
count = numbers.count(2)  # Returns 3 (number of times 2 appears)
```

sort()

The sort() method arranges the elements of a list in ascending order (for numbers and strings).

```python
numbers = [3, 1, 2, 4]
numbers.sort()  # Sorts the list in ascending order
```

reverse()

The reverse() method reverses the order of elements in a list.

```python
fruits = ["apple", "banana", "cherry"]
fruits.reverse()  # Reverses the order of elements
```

There are a lot of list methods and you will need these from time to time. So have a good play around with them and move on to the next section once you have done so. If you are stuck or things don't work as you thought, please ask for help!

`[ ]:`

1.3.5 Nested Lists

A nested list is a list that contains other lists as its elements. This allows you to create multi-dimensional data structures.

```python
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

You can access elements of a nested list using multiple indices.

1.3.6 Summary

In this section, we've explored the world of lists in Python. You've learned what lists are, how to create and modify them, and various operations and methods for working with lists. Lists are a fundamental data structure in Python and are used extensively in programming. As you continue your Python journey, you'll find lists to be invaluable for organizing and manipulating data in your programs. In the next sections, we'll delve into other essential data structures, including tuples, sets, and dictionaries.

1.3.7 Additional Resources

If you want to explore lists further or need additional information, here are some resources to consider:

- Python Official Documentation on Lists: Python Lists
- Real Python's Tutorial on Python Lists: Python Lists: A Complete Introduction
- W3Schools' Python List Tutorial: Python Lists

These resources can provide more in-depth explanations and examples to help you master the use of lists in Python.

1.3.8 Exercises

1. Create a list of your favorite books. Add, remove, and modify elements in the list.

`[ ]:`

2. Create a 2D list representing a tic-tac-toe board and print it.

`[ ]:`

Tuples

Tuples are another essential data structure in Python. They are similar to lists but with a key difference: tuples are immutable, meaning their elements cannot be changed after creation. Tuples are typically used for data that should not be changed during the program's execution. In this section, we'll explore tuples in depth, covering their creation, properties, and common use cases.

And remember to use the space provided to play around with your new knowledge. That's how you get good at programming

1.3.9 Creating Tuples

To create a tuple in Python, you enclose a sequence of elements within parentheses (), separated by commas.

```python
# Creating a tuple of integers
my_tuple = (1, 2, 3, 4, 5)

# Creating a tuple of strings
names = ("Alice", "Bob", "Charlie")
```

[ ]:

1.3.10 Accessing Tuple Elements

You can access individual elements in a tuple using indexing, similar to lists. **Remember:** Python uses 0-based indexing.

```python
# Accessing elements by index
first_element = my_tuple[0]   # Accesses the first element (1)
second_element = my_tuple[1]   # Accesses the second element (2)
```

You can also use negative indexing to access elements from the end of the tuple, just like with lists.

```python
# Accessing elements using negative indexing
last_element = my_tuple[-1]   # Accesses the last element (5)
```

[ ]:

1.3.11 Tuple Packing and Unpacking

Tuple packing is the process of creating a tuple by placing multiple values (elements) inside parentheses. Tuple unpacking is the process of assigning values from a tuple to variables.

```python
# Tuple packing
coordinates = (3, 4)

# Tuple unpacking
x, y = coordinates   # x is 3, y is 4
```

[ ]:

1.3.12 Tuple Methods

Tuples have a few built-in methods, but they are limited due to their immutability:

count()

The count() method returns the number of times a specified element appears in the tuple.

```python
numbers = (1, 2, 2, 3, 2, 4)
count = numbers.count(2)  # Returns 3 (number of times 2 appears)
```

index()

The index() method returns the index of the first occurrence of a specified value.

```python
fruits = ("apple", "banana", "cherry")
index = fruits.index("cherry")  # Returns the index of "cherry"
```

1.3.13 Immutability of Tuples

One of the key characteristics of tuples is their immutability. Once a tuple is created, you cannot change its elements. Attempting to modify a tuple will result in an error.

```python
my_tuple = (1, 2, 3)
my_tuple[1] = 99  # This will raise a TypeError since tuples are immutable
```

[ ]:

1.3.14 Use Cases for Tuples

Tuples are commonly used in the following scenarios:

- Returning Multiple Values from Functions: You can use tuples to return multiple values from a function.

```python
def get_name_and_age():
    return ("Alice", 30)
```

- Data Integrity: When you want to ensure that the data remains unchanged.
- Dictionary Keys: Tuples can be used as keys in dictionaries because they are hashable (unlike lists).

1.3.15 Summary

In this section, you've delved into the world of tuples in Python. You've learned what tuples are, how to create and access them, and their immutability. Tuples are valuable when you need to ensure that certain data remains unchanged throughout your program.

Tuples are not only a fundamental data structure but also provide a valuable tool for organizing and working with data efficiently. As you continue your Python journey, you'll encounter situations where tuples are the preferred choice.

1.3.16 Additional Resources

If you want to explore tuples further or need additional information, here are some resources to consider:

- Python Official Documentation on Tuples: Python Tuples
- Real Python's Tutorial on Python Tuples: Python Tuple Basics
- W3Schools' Python Tuple Tutorial: Python Tuples

These resources offer more in-depth explanations and examples to help you master the use of tuples in Python.

1.3.17 Exercises

1. Create a tuple representing coordinates (x, y). Perform tuple packing and unpacking to assign values to separate variables.

`[ ]:`

2. Create a tuple of your favorite colors. Try to change one of the colors in the tuple and observe the result.

`[ ]:`

Sets

Sets are fundamental data structures in Python that represent an unordered collection of unique elements with NO duplicates. Sets are typically used when you need to store a collection of items and ensure that each item is unique. In this section, we'll explore sets in depth, covering their creation, operations, and common use cases.

1.3.18 Creating Sets

To create a set in Python, you can use curly braces `{}` or the `set()` constructor, passing an iterable (e.g., a list or tuple) as an argument.

```python
# Using curly braces
my_set = {1, 2, 3}

# Using the set() constructor
fruits = set(["apple", "banana", "cherry"])
```

`[ ]:`

1.3.19 Set Operations

Sets support various operations for set theory, including union, intersection, difference, and more.

Union (|)

The union of two sets contains all unique elements from both sets.

```python
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1 | set2   # Contains {1, 2, 3, 4, 5}
```

Intersection (&)

The intersection of two sets contains elements that are present in both sets.

8

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersection_set = set1 & set2  # Contains {3}
```

Difference (-)

The difference of two sets contains elements that are in the first set but not in the second set.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
difference_set = set1 - set2  # Contains {1, 2}
```

[ ]:

[ ]:

1.3.20 Set Methods

Python provides several built-in methods for working with sets:

add()

The add() method adds an element to the set if it's not already present.

```
fruits = {"apple", "banana"}
fruits.add("cherry")  # Adds "cherry" to the set
```

remove()

The remove() method removes a specific element from the set.

```
fruits = {"apple", "banana", "cherry"}
fruits.remove("banana")  # Removes "banana" from the set
```

discard()

The discard() method removes a specific element from the set if it exists, but it won't raise an error if the element is not found.

```
fruits = {"apple", "banana", "cherry"}
fruits.discard("orange")  # No error, as "orange" is not in the set
```

clear()

The clear() method removes all elements from the set, making it empty.

```
fruits = {"apple", "banana", "cherry"}
fruits.clear()  # Removes all elements, leaving an empty set
```

copy()

The copy() method creates a shallow copy of the set.

```
fruits = {"apple", "banana", "cherry"}
fruits_copy = fruits.copy()  # Creates a copy of the set
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

1.3.21   Set Operations and Methods

Sets provide a powerful way to perform operations like testing for membership, checking for subsets, and finding the symmetric difference.

```python
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Membership test
is_member = 3 in set1   # True

# Subset test
is_subset = set1.issubset(set2)   # False

# Symmetric difference
symmetric_difference = set1.symmetric_difference(set2)   # Contains {1, 2, 4, 5}
```

```
[ ]:
```

1.3.22   Use Cases for Sets

Sets are commonly used in various scenarios, including:

- **Removing Duplicates**: Easily remove duplicate elements from a list by converting it to a set and back.

```python
unique_list = list(set(duplicate_list))
```

- **Checking for Uniqueness**: Quickly determine if all elements in a collection are unique by comparing its length to the length of the set.

1.3.23   Summary

In this section, you've delved into the world of sets in Python. You've learned what sets are, how to create and manipulate them, and their use in set operations. Sets are valuable for working with collections of unique elements and performing set-related tasks efficiently.

Sets offer a powerful toolset for solving various programming problems and are frequently used in scenarios where uniqueness and set operations are important.

1.3.24   Additional Resources

If you want to explore sets further or need additional information, here are some resources to consider:

- Python Official Documentation on Sets: Python Sets

- Real Python's Tutorial on Python Sets: Python Sets and Set Theory
- W3Schools' Python Set Tutorial: Python Sets

1.3.25   Exercises

1. Create two sets representing your favorite fruits and a friend's favorite fruits. Find the common fruits between the two sets.

`[ ]:`

2. Given the list of numbers: [1,4,7,3,4,9,2,5,3,8,1], remove duplicates using a set and print the unique.

`[ ]:`

3. Bonus challenge: print how many duplicates have been removed.

`[ ]:`

Dictionaries

Dictionaries are versatile data structures in Python that allow you to store and retrieve data using key-value pairs. Each key in a dictionary is unique, and it is associated with a value. Dictionaries are often used when you need to organize data based on labels or identifiers (keys). In this section, we'll explore dictionaries in depth, covering their creation, operations, and common use cases.

1.3.26   Creating Dictionaries

To create a dictionary in Python, you enclose key-value pairs within curly braces {}. Each key is separated from its corresponding value by a colon :.

```python
# Creating a dictionary of contact information
contact_info = {
    "name": "Alice",
    "email": "alice@example.com",
    "phone": "555-1234"
}

# Creating an empty dictionary
empty_dict = {}
```

You are going to use dictionaries throughout this entire course on a regular basis. Make sure that you play around with the concepts as much as possible and try to find new challenges for yourself

`[ ]:`

1.3.27   Accessing Dictionary Elements

You can access values in a dictionary by using the associated keys.

11

```
# Accessing values by key
name = contact_info["name"]  # Accesses the value associated with the "name" key
```

[ ]:

1.3.28 Modifying Dictionaries

Dictionaries are mutable, which means you can change their values by assigning new values to specific keys.

```
# Modifying values in a dictionary
contact_info["phone"] = "555-5678"  # Updates the phone number
```

[ ]:

1.3.29 Dictionary Methods

Python provides several built-in methods for working with dictionaries:

get() The get() method retrieves the value associated with a key, or a default value if the key does not exist.

```
# Using get() to retrieve a value
email = contact_info.get("email", "N/A")  # Retrieves the email or "N/A" if not found
```

keys()

The keys() method returns a list of all the keys in the dictionary.

```
# Getting all keys in a dictionary
keys_list = contact_info.keys()  # Returns a list of keys
```

values()

The values() method returns a list of all the values in the dictionary.

```
# Getting all values in a dictionary
values_list = contact_info.values()  # Returns a list of values
```

items()

The items() method returns a list of key-value pairs (tuples) as a view object.

```
# Getting all key-value pairs in a dictionary
items_list = contact_info.items()  # Returns a view object of key-value pairs
```

[ ]:

[ ]:

[ ]:

1.3.30 Use Cases for Dictionaries

Dictionaries are commonly used in various scenarios, including:

- **Data Storage**: Storing and retrieving data based on unique identifiers or keys.
- **Configuration Settings**: Storing configuration settings for applications.
- **Counting and Frequency**: Counting occurrences of items or calculating frequencies.

1.3.31 Summary

Dictionaries are a versatile data structure that plays a crucial role in many Python programs. You've learned how to create, access, and modify dictionaries, as well as use dictionary methods for common tasks. Understanding dictionaries is essential for managing and organizing data effectively in your Python projects.

1.3.32 Additional Resources

If you want to explore dictionaries further or need additional information, here are some resources to consider:

- Python Official Documentation on Dictionaries: Python Dictionaries
- Real Python's Tutorial on Python Dictionaries: Python Dictionary Comprehensions
- W3Schools' Python Dictionary Tutorial: Python Dictionaries

1.3.33 Exercises

1. Create a dictionary called student_info that stores the following information about a student: Name (string), Age (integer), Grade (string)

[ ]:

2. Print the student's name from the student_info dictionary followed by the student's age and the student's grade.

[ ]:

3. Update the student's age in the student_info dictionary to be 20 and change the student's grade to '10'.

[ ]:

4. Add the following key-value pairs to the student_info dictionary: "University" with the value "University of Groningen" and "City" with a new value of your choice.

[ ]:

5. Finally, remove the "Grade" key-value pair from the student_info dictionary.

[ ]:

How did you get on with this section? Make sure that you revisit all the elements of data structures and that you are comfortable working with **lists**, **tuples**, **sets**, and **dictionaries** as these form the building blocks of your Python programs.

When you are happy with your progress, you have completed the first section of the course. Well done

Next time, we are going to look at exciting topics such as **Strings** and **Loops and Conditions**. Once you have mastered these concepts you can start writing Python programs that actually do some cool stuff!

See you on Thursday!

[ ]:

04Strings

November 29, 2023

Dr Gunnar Mallon (g.mallon@rug.nl), *Department of Cultural Geography (Faculty of Spatial Science), University of Groningen*

Table of Contents

0.1 Introduction to Strings

Strings are one of the fundamental data types in Python, used to represent text or sequences of characters. Understanding strings is crucial because they play a vital role in virtually every Python program. In this section, we'll explore what strings are, why they are essential, and how to work with them effectively.

A **string** in Python is a sequence of characters enclosed in single (' '), double (" "), or triple (''' ''' or """ """) quotes. Strings can contain letters, numbers, symbols, and even spaces. They are versatile and can represent anything from names, sentences, file paths, and more.

Here are some examples of strings:

```
name = "Alice"
message = 'Hello, World!'
address = '''123 Main Street
City, Country'''
```

0.1.1 Importance of Strings

Strings are a fundamental data type for several reasons:

Text Processing: Strings are used for text processing tasks such as reading and writing files, parsing data, and manipulating textual information.

User Interaction: Strings are essential for interacting with users through input and output, like displaying messages and receiving user input.

Data Representation: Many real-world data, like names, addresses, and descriptions, are naturally represented as strings.

1

Programming Logic: Strings are used in conditional statements, loops, and functions to make decisions and perform actions based on text data.

0.1.2 String Literals and Enclosures

In Python, you can enclose strings in single (' '), double (" "), or triple (''' ''' or""" """ """") quotes. The choice of enclosure depends on your specific needs.

Single Quotes: Enclosing a string in single quotes is useful when the string itself contains double quotes.

```python
message = 'He said, "Hello, World!"'
```

Double Quotes: Similarly, enclosing a string in double quotes is helpful when the string contains single quotes.

```python
sentence = "She's a programmer."
```

Triple Quotes: Triple quotes are used for multi-line strings or when you want to include both single and double quotes without escaping.

```python
poem = '''Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.'''
```

Now that we've introduced the basics of strings, let's dive deeper into creating and manipulating strings in the following sections.

Creating and Defining Strings

Creating and defining strings is a fundamental skill in Python programming. It involves specifying the content of the string, including letters, numbers, symbols, and spaces. Let's explore different ways to create and define strings effectively.

0.1.3 Creating Strings with Single Quotes

The simplest way to create a string in Python is by enclosing the text in single quotes (' '). For example:

```python
name = 'Alice'
greeting = 'Hello, World!'
```

Single quotes are handy when the string itself contains double quotes.

```python
quote = 'She said, "Python is awesome!"'
```

You can also use double quotes (" ") to create strings. Double quotes are useful when the string contains single quotes.

```python
sentence = "She's a programmer."
```

Both single and double quotes work interchangeably, but it's essential to be consistent within your codebase.

Triple quotes (''' ''' or """ """) are used for multi-line strings or when you want to include both single and double quotes without escaping. They are particularly helpful for creating strings that span multiple lines.

```python
poem = '''Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.'''
```

Multi-line strings are also used for docstrings, which provide documentation for functions, classes, and modules.

[ ]:

Escaping Special Characters

Sometimes, you may need to include special characters like single quotes or double quotes within a string. To do this, you can escape them using a backslash (). For example:

```python
message = "He said, \"Hello, World!\""
```

Here, the backslash tells Python to treat the following character as a literal character, not as a string delimiter.

Converting Other Data Types to Strings

You can convert other data types, such as numbers or booleans, to strings using the str() function. This is useful when you want to concatenate non-string values with strings.

```python
age = 25
message = 'I am ' + str(age) + ' years old.'
```

Converting data types to strings allows you to combine them with text effectively.

[ ]:

0.1.4 Exercises

1. Create a string representing your favorite quote and assign it to a variable. Make sure to include both single and double quotes within the string.

[ ]:

2. Define a multi-line string using triple quotes that describes a place you'd like to visit.

[ ]:

3. Convert your age (an integer) to a string and create a sentence that includes your age.

[ ]:

0.1.5 Defining and Using F-strings

F-strings, short for "formatted string literals," are a powerful and convenient way to create strings with dynamic content in Python. They allow you to embed expressions within strings, making it easier to incorporate variables and values into your text. F-strings are defined by prefixing a string with the letter `f` or `F`, followed by curly braces `{}` containing expressions that will be evaluated and inserted into the string.

Let's explore how to define and use F-strings:

Basic F-strings   In its simplest form, you can create an F-string by placing an expression within curly braces `{}` inside a string.

```python
name = "Alice"
age = 30
message = f"My name is {name} and I am {age} years old."
```

In this example, the expressions {name} and {age} are replaced with the values of the name and age variables when the string is created. This allows you to create dynamic strings that incorporate variables seamlessly.

Expressions in F-strings

F-strings support a wide range of expressions that can be included within the curly braces. You can perform calculations, access list elements, and call functions, among other things.

```python
x = 5
y = 3
result = f"The sum of {x} and {y} is {x + y}."
```

Here, the expression {x + y} is evaluated, and the result is inserted into the string.

F-strings are central to working with strings in Python and have replaced older syntax. Using f-strings is now the prefered way of working with complex strings.

[ ]:

Formatting in F-strings

F-strings also support various formatting options to control how the inserted values are displayed. You can specify the number of decimal places for floats, control alignment and width, and more.

```python
price = 19.99
formatted_price = f"The item costs ${price:.2f}."
```

In this example, :.2f specifies that the price variable should be formatted as a floating-point number with two decimal places.

[ ]:

0.1.6 Exercise

1. Create an F-string that includes the current date and time in the format "Month Day, Year - Hour:Minute:Second."

(I have started the solution for you and gotten the current date and stored it in a variable called current_datetime. You may have to look up how you can extract the Month, Day, etc... from the variable. If you are stuck, please make sure to ask for help)

```
[9]:   # Starter code to get you going. The current date and time will be stored in the
       →current_datetime variable

       import datetime
       current_datetime = datetime.datetime.now()

       # Remember the different ways to get help about unknown modules and functions in
       →Python?
       # Complete the code below to print the current date and time in the format
       →mentioned above
```

```
[ ]:
```

String Operations

String operations allow you to manipulate and work with strings effectively. In this section, we'll explore various operations that you can perform on strings, including concatenation, repetition, indexing, slicing, finding the length of strings, and using built-in string methods and functions.

0.1.7 Concatenation: Combining Strings

Concatenation is the process of combining two or more strings to create a new one. You can concatenate strings using the + operator.

```
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name   # full_name will be "John Doe"
```

You can also concatenate strings with variables and literals to create dynamic text.

```
greeting = "Hello, "
name = "Alice"
message = greeting + name + "!"   # message will be "Hello, Alice!"
```

```
[ ]:
```

0.1.8 Repetition: Repeating Strings

Repetition allows you to create a new string by repeating an existing one multiple times. You can achieve this using the * operator.

```
word = "Python"
repeated_word = word * 3  # repeated_word will be "PythonPythonPython"
```

This is useful for generating repetitive patterns or sequences of characters.

[ ]:

0.1.9 Indexing and Slicing: Accessing Parts of Strings

Strings are sequences of characters, and you can access individual characters or portions of a string using indexing and slicing.

Indexing: To access a single character at a specific position, use square brackets [] with the index (position) of the character.

```
word = "Python"
first_character = word[0]  # first_character will be "P"
third_character = word[2]  # third_character will be "t"
```

Slicing: Slicing allows you to extract a portion of a string by specifying a start and end index, separated by a colon :. The slice includes the character at the start index and goes up to (but does not include) the character at the end index.

```
word = "Python"
substring = word[2:5]  # substring will be "tho"
```

Slicing is a powerful way to work with substrings within a larger string. More about this in Week 3

[ ]:

0.1.10 Length of Strings

You can find the length of a string (the number of characters) using the built-in len() function.

```
text = "This is a string."
length = len(text)  # length will be 17
```

Knowing the length of a string is fundamental when working with loops and conditions. Make sure that you really know this method!

[ ]:

String Formatting

Python provides a variety of built-in string methods that allow you to perform operations on strings efficiently. These methods enable you to manipulate and transform strings according to your needs. In this section, we'll explore several commonly used string methods in detail.

0.1.11 Changing Case

str.upper() and str.lower()   The str.upper() method converts all characters in a string to uppercase, while str.lower() converts them to lowercase.

```
text = "Hello, World!"
uppercase_text = text.upper()    # uppercase_text will be "HELLO, WORLD!"
lowercase_text = text.lower()    # lowercase_text will be "hello, world!"
```

Changing case is useful for making string comparisons or ensuring consistent formatting.

[ ]:

0.1.12   Removing Whitespace

The `str.strip()` method removes leading and trailing whitespace (spaces, tabs, and newline characters) from a string.

```
text = "   This is a string with whitespace.   "
stripped_text = text.strip()    # stripped_text will be "This is a string with whitespace."
```

Stripping whitespace is often used when processing user input or cleaning data.

[ ]:

0.1.13   Replacing Substrings

The `str.replace()` method replaces all occurrences of a specified substring with another string.

```
sentence = "This is a sample sentence."
modified_sentence = sentence.replace("sample", "new")    # modified_sentence will be "This is a ne
```

Replacing substrings is useful for text manipulation and data cleaning tasks.

[ ]:

0.1.14   Splitting Strings into Lists

The `str.split()` method splits a string into a list of substrings based on a specified delimiter (default is space).

```
csv_data = "Alice,30,Programmer"
data_list = csv_data.split(",")    # data_list will be ["Alice", "30", "Programmer"]
```

Splitting strings is essential when working with structured data in text format.

[ ]:

0.1.15   Joining Lists into Strings

The `str.join()` method joins the elements of a list into a single string, using the string as a delimiter.

```
words = ["This", "is", "a", "list"]
joined_sentence = " ".join(words)    # joined_sentence will be "This is a list"
```

Joining lists into strings is handy when formatting output or constructing CSV/TSV files.

[ ]:

```

### 0.1.16 Finding Substrings

Both `str.find()` and `str.index()` methods search for a substring within a string and return the starting index of the first occurrence. However, `str.find()` returns -1 if the substring is not found, while `str.index()` raises a ValueError.

```
sentence = "This is a sample sentence."
position = sentence.find("sample") # position will be 10
```

These methods are useful for locating specific content within text.

[ ]:

### 0.1.17 Counting Occurrences

The `str.count()` method counts the number of non-overlapping occurrences of a substring in a string.

```
text = "To be or not to be, that is the question."
count = text.count("be")  # count will be 2
```

Counting occurrences helps analyze text data.

[ ]:

0.1.18 Checking Prefixes and Suffixes

The `str.startswith()` method checks if a string starts with a specified prefix, and `str.endswith()` checks if it ends with a specified suffix. Both methods return True or False.

```
filename = "document.txt"
is_document = filename.startswith("doc")  # is_document will be True
is_text = filename.endswith(".txt")  # is_text will be True
```

These methods are useful for file handling and data validation.

[ ]:

0.1.19 Checking Character Types

These methods check the character types within a string. `str.isalnum()` checks if all characters are alphanumeric (letters or numbers), `str.isalpha()` checks if they are alphabetic characters, and `str.isdigit()` checks if they are digits.

```
alphanumeric = "Python3"
is_alpha = alphanumeric.isalpha()  # is_alpha will be False
is_alnum = alphanumeric.isalnum()  # is_alnum will be True
```

These methods are helpful for data validation and filtering.

[ ]:

0.1.20 String Formatting

You can format the way that numbers are shown inside of f-strings by using special colon-notation

```
price = 19.99
formatted_price = f"The item costs ${price:.1f}."
print(formatted_price)
```

You will see that the price has been rounded to one decimal place. Make sure to play around with this.

[ ]:

0.1.21 Exercises

1. Given a string text containing extra spaces at the beginning and end, use str.strip() to remove the leading and trailing whitespace and print the cleaned text.

[ ]:

2. Create a string with the following sentence: "Python is a versatile programming language. Python is used for web development, data analysis, and more." Use the str.replace() method to replace all occurrences of "Python" with "JavaScript."

[ ]:

3. Given a list of words, use the str.join() method to join them into a single string separated by hyphens ("-").

[ ]:

4. Create two strings, first_name and last_name, representing your first name and last name. Concatenate them to form a full name and print it.

[ ]:

5. Use the str.replace() method to replace all occurrences of a word in a sentence with another word of your choice.

[ ]:

String Escapes and Special Characters

In Python strings, you may encounter situations where you need to include special characters or escape sequences within a string. This section will explore how to deal with these situations using backslashes (\) and raw strings (`r"string"`).

0.1.22   Escaping Characters with Backslashes

Backslash as an Escape Character   The backslash (\) serves as an escape character in Python strings. It allows you to include characters that are not typically allowed directly in a string.

```
message = "He said, \"Hello, World!\""
In this example, the backslash \ precedes the double quotes " to include them within the string.
```

0.1.23   Common Escape Sequences

Python supports various escape sequences to represent special characters within strings:

- \n: Newline (line break)
- \t: Tab
- \: Backslash
- ': Single quote
- ": Double quote

```
address = "123 Main Street\nCity\tZip Code"
```

This creates a multi-line string with a newline and a tab.

[ ]:

0.1.24   Raw Strings (r"string")

Raw strings, denoted by r"string", are used to treat backslashes as literal characters, ignoring their escape sequence behavior. This is particularly useful when working with file paths or regular expressions.

```
path = r'C:\Users\Alice\Documents'
regex = r'\d{3}-\d{2}-\d{4}'
```

In the path variable, the backslashes are treated as literal characters, making it suitable for file paths.

0.1.25   Exercises

1. Create a string that represents a Windows file path (e.g., C:
   Users
   YourName
   Documents
   File.txt) using both regular and raw strings. Compare the two representations.

[ ]:

2. Create a multi-line string that displays the following information about a book:

Title: "Python Programming"
Author: John Smith
Price: $29.99

Use escape sequences for double quotes and the dollar sign.

[ ]: ┌───┐
     │   │
     └───┘

─────────────────────────────

String Operations Exercises

These exercises focus on string manipulation will help you practice and reinforce your skills in working with strings.

   1. **Reverse a String**: Write a Python program that takes a string as input and prints the reverse of the string. Use slicing to achieve this without loops or conditionals.

[ ]: ┌───┐
     │   │
     └───┘

   2. **Count Vowels and Consonants**: Create a program that counts the number of vowels (a, e, i, o, u) and consonants in a given string. Use the `str.count()` method to count vowels and calculate the consonant count without loops or conditionals.

[ ]: ┌───┐
     │   │
     └───┘

   3. **Palindrome Checker**: Write a program that checks if a given string is a palindrome, which means it reads the same forwards and backwards. Use string slicing to reverse the string and check if it's equal to the original.

[ ]: ┌───┐
     │   │
     └───┘

   4. **Word Capitalization**: Write a program that takes a sentence as input and capitalizes the first letter of each word. Use the `str.title()` method to achieve this without loops or conditionals.

[ ]: ┌───┐
     │   │
     └───┘

   5. **Word Reversal**: Create a program that takes a sentence and reverses the order of words. Use the `str.split()` and `str.join()` methods to reverse the words without loops or conditionals.

[ ]: ┌───┐
     │   │
     └───┘

─────────────────────────────

And there you have it: **Strings**

Next we are going to look at two fundamental concepts in all of programming: Loops and Conditionals.

[ ]: ┌───┐
     │   │
     └───┘

05LoopsAndConditionals

November 29, 2023

Dr Gunnar Mallon (g.mallon@rug.nl), *Department of Cultural Geography (Faculty of Spatial Science), University of Groningen*

Table of Contents

1 Loops and Conditionals

1.1 Introduction

In the world of programming, the ability to make decisions and repeat actions is fundamental. Conditionals and loops are the building blocks that empower us to write dynamic and intelligent programs. Today, we'll learn about the concepts of conditionals and loops in Python and how they enable us to control the flow of our code.

1.1.1 The Need for Decision-Making

Consider a real-world scenario: a traffic signal. At a traffic signal, a set of conditions determine whether you stop, go, or slow down. Similarly, in programming, we often encounter situations where we need to make choices based on certain conditions.

Example:

Imagine a program that checks whether a student has passed an exam. The condition here is whether the student's score is above a certain threshold. If it is, the program should print "Pass," otherwise, it should print "Fail."

```python
# Example 1: Conditional (if-else)
score = 75

if score >= 55:
```

```python
        print("Pass")
else:
        print("Fail")
```

1.1.2   The Power of Repetition

In programming, we also frequently encounter tasks that need to be performed repeatedly. Loops provide a way to execute a block of code multiple times, which can be incredibly efficient and practical.

Example:

Imagine a program that prints the first ten multiples of 5. Instead of writing ten separate print statements, we can use a loop. You'll learn about the syntax in a little bit.

```python
# Example 2: Loop (for)
for i in range(1, 11):
    print(5 * i)
```

Example:

Suppose we want to find all the prime numbers between 1 and 100. This task involves both decision-making (checking if a number is prime) and repetition (iterating through numbers). We'll use conditionals and loops to achieve this.

```python
# Example 3: Combining Conditionals and Loops
for num in range(2, 101):
    is_prime = True
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            is_prime = False
            break
    if is_prime:
        print(num, end=' ')
```

Don't worry if this looks like gibberish to you right now. I promise you, at the end of the lecture, **loops and conditionals** are going to become second nature to you.

[ ]:

[ ]:

[ ]:

Conditional Statements

1.1.3   Introduction to Conditional Statements

Conditional statements are an essential concept in programming that allows us to make decisions based on certain conditions. In this section, we will explore the need for conditional statements, understand their role in decision-making, and get an overview of Python's conditional structures.

Understanding the Need for Conditional Statements   In programming, we often encounter situations where we need to perform different actions based on certain conditions. For example, consider a program that calculates the grade of a student based on their score (as we saw above). Depending on the score, we may want to assign different grades such as A, B, C, etc. This is where conditional statements come into play.

Conditional statements enable us to write code that can make decisions and choose different paths based on the conditions we specify. Without conditional statements, our programs would execute the same set of instructions regardless of the input or circumstances, limiting their usefulness.

The Role of Conditions in Decision-Making   Conditions are expressions that evaluate to either **true** or **false**. They are used to determine which path our program should take based on the given conditions. In other words, conditions act as the criteria for decision-making in our programs .

For example, let's say we want to write a program that checks if a number is positive or negative. We can define a condition such as "if the number is greater than zero, it is positive; otherwise, it is negative." The condition here is "number > 0," and based on its evaluation, our program can take different actions.

An Overview of Python's Conditional Structures   *Don't worry if you don't understand this section yet - we are going to look at each structure in much more detail*

Python provides several conditional structures that allow us to implement decision-making in our programs. The most commonly used conditional structures in Python are:

if statement: The `if` statement is used to execute a block of code only if a certain condition is true.

For example, let's say we want to check if a number is positive. We can use the if statement as follows:

```python
number = 10
if number > 0:
    print("The number is positive")
```

if-else statement: The if-else statement allows us to execute one block of code if a condition is true and another block of code if the condition is false.

For example, let's modify our previous example to print "The number is negative" if the condition is false:

```python
number = -5
if number > 0:
    print("The number is positive")
else:
    print("The number is negative")
```

if-elif-else statement: The if-elif-else statement allows us to check multiple conditions and execute different blocks of code based on the first condition that evaluates to true.

For example, let's say we want to assign grades to students based on their scores. We can use the if-elif-else statement as follows:

```python
score = 85
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
else:
    print("Grade: D")
```

These are the basic conditional structures in Python that will allow you to implement decision-making in your programs. As you progress in your programming journey, you will encounter more complex conditional statements and structures, but understanding these fundamentals will provide a solid foundation.

Let's look at each one of these in more detail!

1.1.4 The importance of indentation in Python

In Python, indentation is used to define the structure of the code. It is crucial to indent the code correctly to indicate which statements are part of a code block. The `if` statement and other control structures in Python use indentation to determine the scope of the code block.

Here's an example that demonstrates the importance of indentation in an `if` statement:

```python
x = 10

if x > 5:
    print("x is greater than 5")
    print("This statement is also part of the code block")
print("This statement is not part of the code block")
```

In this example, the first `print` statement and the second `print` statement are both indented under the `if` statement. This indicates that both statements are part of the code block that is executed if the condition is true. The third `print` statement is not indented, so it is not part of the code block and will be executed regardless of the condition. The output of this code will be:

```
x is greater than 5
This statement is also part of the code block
This statement is not part of the code block
```

It is important to note that Python uses indentation consistently throughout the code. Inconsistent indentation can lead to syntax errors and unexpected behavior. Try it out yourself

```
[ ]:
```

The `if` Statement (in more detail)

The `if` statement is a fundamental control structure in Python that allows you to execute a block of code conditionally. It is used to make decisions based on the evaluation of a condition. The `if` statement evaluates a condition and if it is true, the code block associated with it is executed. If the condition is false, the code block is skipped.

Syntax and usage of the `if` statement   The syntax of the `if` statement is as follows:

```python
if condition:
    # code block to be executed if the condition is true (note the indentation)
```

The `condition` is an expression that evaluates to either `True` or `False`. If the condition is true, the code block indented under the `if` statement is executed. If the condition is false, the code block is skipped.

Here's an example that demonstrates the usage of the `if` statement:

```python
x = 10
```

```python
if x > 5:
    print("x is greater than 5")
```

In this example, the condition `x > 5` is evaluated. Since `x` is equal to 10, which is greater than 5, the condition is true and the code block `print("x is greater than 5")` is executed. The output of this code will be `x is greater than 5`.

```
[ ]:
```

Writing simple conditional expressions   The condition in an `if` statement can be any expression that evaluates to either `True` or `False`. This allows you to write simple conditional expressions using comparison operators such as `==` (equal to), `!=` (not equal to), `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to).

Here's an example that demonstrates the usage of comparison operators in an `if` statement:

```python
x = 10
```

```python
if x == 10:
    print("x is equal to 10")
```

In this example, the condition `x == 10` is evaluated. Since `x` is equal to 10, the condition is true and the code block `print("x is equal to 10")` is executed. The output of this code will be `x is equal to 10`.

```
[ ]:
```

1.1.5   The `else` Clause

The `else` clause is a powerful construct in Python that allows for alternative execution based on a condition. It is used in combination with the `if` statement to create two mutually exclusive outcomes.

Introducing the `else` clause for alternative execution   In Python, the `else` clause is used to specify a block of code that should be executed if the condition in the `if` statement evaluates to `False`. This provides an alternative execution path when the condition is not met.

The syntax for using the `else` clause is as follows:

```python
if condition:
    # code to be executed if condition is True
else:
    # code to be executed if condition is False
```

The `else` clause is indented at the same level as the `if` statement, indicating that it is part of the same block of code.

Let's take a look at an example to understand how the `else` clause works:

```python
age = 20

if age >= 18:
    print("You are eligible to vote!")
else:
    print("You are not eligible to vote.")
```

In this example, the condition `age >= 18` is evaluated. If the condition is `True`, the code inside the `if` block is executed, which prints "You are eligible to vote!". If the condition is `False`, the code inside the `else` block is executed, which prints "You are not eligible to vote.".

[ ]:

1.1.6 Nested Conditionals

In this section, we will explore the concept of nested conditionals in Python. Nested conditionals allow us to create complex decision trees by combining multiple `if` statements within each other. This hierarchical structure of nested conditionals helps us to handle more intricate scenarios and make our programs more flexible.

Creating complex decision trees with nested `if` statements   Nested conditionals are useful when we need to check multiple conditions and perform different actions based on the combination of these conditions. By nesting `if` statements, we can create decision trees that branch out based on the outcome of each condition.

Let's consider the example where we want to determine the grade of a student based on their score in an exam. We can use nested conditionals to handle different grade ranges:

```python
score = 85

if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
elif score >= 60:
    grade = 'D'
else:
    grade = 'F'
```

```
print("Grade:", grade)
```

In the above example, we have nested `if` statements to check the score against different ranges. If the score is greater than or equal to 90, the student gets an 'A' grade. If not, we move to the next `elif` statement and check if the score is greater than or equal to 80, and so on. Finally, if none of the conditions are met, the student gets an 'F' grade.

[ ]:

Understanding the hierarchical structure of nested conditionals   Nested conditionals have a hierarchical structure, where each nested `if` statement is evaluated only if the condition of the outer `if` statement is true. This allows us to create more complex decision trees by adding multiple levels of conditions.

Let's consider another example where we want to determine the type of a triangle based on its side lengths. We can use nested conditionals to handle different combinations of side lengths:

```
side1 = 3
side2 = 4
side3 = 5

if side1 == side2 == side3:
    triangle_type = 'Equilateral'
elif side1 == side2 or side1 == side3 or side2 == side3:
    triangle_type = 'Isosceles'
else:
    triangle_type = 'Scalene'

print("Triangle type:", triangle_type)
```

In the above example, we have nested `if` statements to check the side lengths of the triangle. If all sides are equal, the triangle is classified as 'Equilateral'. If not, we move to the next `elif` statement and check if any two sides are equal, and so on. Finally, if none of the conditions are met, the triangle is classified as 'Scalene'.

By understanding the hierarchical structure of nested conditionals, we can create more sophisticated programs that handle complex decision-making scenarios. It is important to ensure proper indentation and logical ordering of conditions to achieve the desired behavior in nested conditionals.

[ ]:

Conditional Expressions (Ternary Operators)

In programming, conditional expressions are used to make decisions based on certain conditions. Ternary operators, also known as conditional expressions, provide a concise way to write conditional statements in Python. They allow you to write a single line of code to evaluate a condition and return one of two values based on the result.

Writing concise conditional expressions using the ternary operator   The ternary operator in Python has the following syntax:

```
value_if_true if condition else value_if_false
```

The `condition` is evaluated first. If it is true, the expression returns `value_if_true`, otherwise it returns `value_if_false`. This allows you to write compact and readable code for simple conditional statements.

Syntax and practical examples Let's look at some practical examples to understand how ternary operators work:

Example 1: Checking if a number is even or odd

```
num = 7
result = "even" if num % 2 == 0 else "odd"
print(result)   # Output: odd
```

In this example, the condition `num % 2 == 0` checks if the number `num` is divisible by 2. If it is true, the expression returns "even", otherwise it returns "odd".

Example 2: Checking if a person is eligible to vote

```
age = 18
result = "eligible" if age >= 18 else "not eligible"
print(result)   # Output: eligible
```

In this example, the condition `age >= 18` checks if the person's age is greater than or equal to 18. If it is true, the expression returns "eligible", otherwise it returns "not eligible".

Example 3: Finding the maximum of two numbers

```
num1 = 10
num2 = 20
max_num = num1 if num1 > num2 else num2
print(max_num)   # Output: 20
```

In this example, the condition `num1 > num2` checks if `num1` is greater than `num2`. If it is true, the expression returns `num1`, otherwise it returns `num2`.

In the cells below try playing around with ternary operators. Is there a away that you can write the code in the first example in only two lines?

[ ]:
[ ]:
[ ]:

When to use ternary operators for readability Ternary operators are particularly useful when you have simple conditional statements that can be expressed concisely in a single line. They can improve code readability by reducing the number of lines and making the intention of the code more clear.

However, it is important to use ternary operators responsibly. If the condition or the expressions are complex, it is better to use traditional if-else statements for better readability and maintainability of the code.

In summary, ternary operators provide a concise way to write conditional expressions in Python. They can be used to simplify simple conditional statements and improve code readability. However, it is important to use them appropriately and consider the complexity of the conditions and expressions for better code maintainability.

1.2 Exercises

Temperature Converter

Write a program that converts a temperature from Celsius to Fahrenheit or vice versa, based on user input. The program should prompt the user to enter the temperature and the unit of measurement (Celsius or Fahrenheit). Then, it should convert the temperature to the other unit and display the result. *Look up the command `input`.*

Example:

```
Enter the temperature: 32
Enter the unit of measurement (C/F): C
The temperature in Fahrenheit is 89.6°F.
```

[ ]:

Grade Calculator

Create a program that calculates the grade based on a student's score. The program should prompt the user to enter the score as a percentage and then display the corresponding grade based on the following scale:

- 90% or above: A
- 80% to 89%: B
- 70% to 79%: C
- 60% to 69%: D
- Below 60%: F

Example:

```
Enter the score: 78
The grade is C.
```

[ ]:

Leap Year Checker

Write a program that checks whether a given year is a leap year or not. The program should prompt the user to enter a year and then display whether it is a leap year or not.

A leap year is divisible by 4, but not divisible by 100 unless it is also divisible by 400.

Example:

```
Enter a year: 2020
2020 is a leap year.
```

`[ ]:`

Palindrome Checker

Write a program that checks whether a given string is a palindrome or not. A palindrome is a word, phrase, number, or other sequence of characters that reads the same forward and backward, ignoring spaces, punctuation, and capitalization.

Example:

```
Enter a string: Radar
Radar is a palindrome.

Enter a string: Python
Python is not a palindrome.
```

These exercises will help you practice using conditional statements to solve various problems. Try to solve them on your own, and if you get stuck, have a look for help online or ask. Happy coding!

`[ ]:`

Loops

In programming, the concept of repetition is essential to perform tasks repeatedly without having to write the same code multiple times. This is where loops come into play. Loops allow us to execute a block of code multiple times, making our code more efficient and reducing redundancy.

1.2.1 How loops enhance code efficiency

Loops are powerful tools that help us automate repetitive tasks. Instead of writing the same code over and over again, we can use loops to iterate through a sequence of elements and perform the same set of instructions on each element. This not only saves us time and effort but also makes our code more readable and maintainable.

By using loops, we can avoid duplicating code and make our programs more concise. This is particularly useful when dealing with large datasets or when we need to perform a specific operation on each item in a list, for example. This is something that you will be doing throughout Geospatial Data Science.

1.2.2 An overview of Python's loop structures

Python provides two main loop structures: the `for` loop and the `while` loop.

The `for` loop   The `for` loop is used to iterate over a sequence of elements, such as a list, tuple, string, or range. It allows us to execute a block of code for each item in the sequence.

Let's see an example to understand how the `for` loop works:

```python
fruits = ["apple", "banana", "cherry"]
```

```python
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
cherry
```

In this example, the `for` loop iterates over each item in the `fruits` list and prints it. The loop starts with the first item, "apple", then moves on to "banana", and finally "cherry".

The `while` loop   The `while` loop is used to repeatedly execute a block of code as long as a certain condition is true. It allows us to create a loop that continues until a specific condition is met.

Let's see an example to understand how the `while` loop works:

```python
count = 0

while count < 5:
    print(count)
    count += 1
```

Output:

```
0
1
2
3
4
```

In this example, the `while` loop continues to execute as long as the condition `count < 5` is true. It starts with `count` equal to 0 and increments it by 1 in each iteration. The loop stops when `count` becomes 5.

Let's look at each of these in *much* more detail!

The `for` Loop

The `for` loop is a fundamental construct in Python that allows us to iterate over a sequence of elements and perform a set of instructions for each element. It is particularly useful when we want to repeat a task a specific number of times or when we want to iterate over the elements of a sequence.

Syntax and usage of the `for` loop   The syntax of the `for` loop in Python is as follows:

```python
for element in sequence:
    # code block to be executed for each element
```

Here, `element` is a variable that takes on the value of each element in the `sequence` for each iteration of the loop. The code block following the `for` statement is indented and is executed for each element in the sequence.

Iterating over sequences The `for` loop is commonly used to iterate over sequences such as strings, lists, tuples, and ranges. Let's look at some examples:

1. Iterating over a string:

```python
for char in "Python":
    print(char)
```

Output:

```
P
y
t
h
o
n
```

2. Iterating over a list:

```python
days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]
for day in days:
    print(day)
```

Output:

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

3. Iterating over a tuple:

```python
numbers = (1, 2, 3, 4, 5)
for number in numbers:
    print(number)
```

Output:

```
1
2
3
4
5
```

4. Iterating over a range:

```python
for i in range(5):
    print(i)
```

Output:

```
0
1
```

```
2
3
4
```

Go on, have a play with for loops, they are super important for programming!

[ ]: [   ]

[ ]: [   ]

[ ]: [   ]

Applying `for` loops for repetitive tasks   The `for` loop is extremely useful when we want to perform repetitive tasks. For example, let's say we want to calculate the sum of all numbers from 1 to 10. We can use a `for` loop to iterate over the range of numbers and accumulate the sum:

```python
sum = 0
for i in range(1, 11):
    sum += i
print("The sum is:", sum)
```

Output:

```
The sum is: 55
```

In this example, the `for` loop iterates over the numbers 1 to 10, and for each iteration, the value of `i` is added to the `sum` variable. Finally, the sum is printed.

[ ]: [   ]

1.2.3   Exercises

1. Write a program that prints the squares of the numbers from 1 to 10 using a `for` loop.

[ ]: [   ]

2. Write a program that calculates the factorial of a given number using a `for` loop.

[ ]: [   ]

3. Write a program that counts the number of vowels in a given string using a `for` loop.

[ ]: [   ]

4. Write a program that finds the maximum element in a given list using a `for` loop.

[ ]: [   ]

5. Number Guessing Game

Create a number-guessing game where the program generates a random number between 1 and 100, and the user has to guess the number. The program should provide feedback to the user after each guess, indicating whether the guess was too high or too low. Once the user guesses the correct number, the program should display the number of attempts it took.

Example:

```
Guess a number between 1 and 100: 50
Too low! Try again.
Guess a number between 1 and 100: 75
Too high! Try again.
Guess a number between 1 and 100: 63
Congratulations! You guessed the number in 3 attempts.
```

[ ]:

The `while` Loop

The `while` loop is a fundamental control structure in Python that allows you to repeatedly execute a block of code as long as a certain condition is true. It is particularly useful when you want to perform a task multiple times without knowing in advance how many iterations will be needed.

Syntax and usage of the `while` loop   The syntax of the `while` loop is as follows:

```python
while condition:
    # code block to be executed
```

The `condition` is an expression that evaluates to either `True` or `False`. As long as the condition is `True`, the code block inside the loop will be executed repeatedly. Once the condition becomes `False`, the loop will terminate, and the program will continue with the next line of code after the loop.

Let's look at a simple example to understand the usage of the `while` loop:

```python
count = 0
while count < 5:
    print("Count:", count)
    count += 1
```

In this example, the loop will execute as long as the value of `count` is less than 5. Inside the loop, we print the current value of `count` and then increment it by 1. The output of this code will be:

```
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
```

[ ]:

Writing loop conditions and exit strategies When writing the condition for a `while` loop, it is essential to ensure that the condition will eventually become `False` to avoid an infinite loop. An infinite loop occurs when the condition always evaluates to `True`, causing the loop to continue indefinitely.

To create a loop condition, you can use comparison operators (`<`, `>`, `<=`, `>=`, `==`, `!=`) to compare variables or values. You can also use logical operators (`and`, `or`, `not`) to combine multiple conditions.

Here's an example that demonstrates the usage of a `while` loop with a condition:

```python
number = 1
while number <= 10:
    if number % 2 == 0:
        print(number, "is even")
    else:
        print(number, "is odd")
    number += 1
```

In this example, the loop will execute as long as the value of `number` is less than or equal to 10. Inside the loop, we check if the number is even or odd using the modulo operator (`%`). The output of this code will be:

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
10 is even
```

[ ]:

Handling infinite loops and ensuring program termination It is crucial to ensure that your `while` loop will eventually terminate to prevent an infinite loop. An infinite loop can cause your program to hang or consume excessive system resources.

To handle infinite loops, you can use various techniques. One common approach is to include an exit strategy within the loop. An exit strategy is a condition that, when met, will cause the loop to terminate.

Here's an example that demonstrates the usage of an exit strategy:

```python
number = 1
while True:
    if number > 10:
        break
    if number % 2 == 0:
        print(number, "is even")
```

```
    else:
        print(number, "is odd")
    number += 1
```

In this example, we use the `break` statement to exit the loop when the value of `number` exceeds 10. The `break` statement immediately terminates the innermost loop it is contained within. The output of this code will be the same as the previous example.

[ ]:

1.2.4 Exercises

Take your time to solve these exercises and feel free to ask for help if you encounter any difficulties.

1. Write a program that prints the first 10 multiples of 3 using a `while` loop.

[ ]:

2. Write a program that prompts the user to enter a positive integer and calculates the sum of all the numbers from 1 to that integer using a `while` loop.

[ ]:

3. Write a program that generates a random number between 1 and 100 and asks the user to guess the number. The program should keep prompting the user for guesses until they correctly guess the number. Use a `while` loop for this task.

[ ]:

1.2.5 Loop Control Statements

In programming, loop control statements are used to alter the flow of execution within loops. They allow us to control when to exit a loop prematurely, skip certain iterations, or execute a block of code only when the loop has completed normally. In Python, there are three loop control statements: `break`, `continue`, and the `else` clause.

Using `break` to exit loops prematurely The `break` statement is used to exit a loop prematurely, regardless of the loop condition, as we have seen. When encountered, the `break` statement immediately terminates the loop and the program continues with the next statement after the loop. This is useful when we want to stop the execution of a loop based on a certain condition.

Here's an example that demonstrates the usage of `break`:

```
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    if num == 3:
        break
```

```python
    print(num)

print("Loop ended")
```

Output:

```
1
2
Loop ended
```

In this example, the loop iterates over the `numbers` list. When the value of `num` becomes 3, the `break` statement is encountered, and the loop is exited prematurely. As a result, only the numbers 1 and 2 are printed before the loop ends.

[ ]:

Employing `continue` to skip iterations   The `continue` statement is used to skip the rest of the current iteration and move on to the next iteration of the loop. When encountered, the `continue` statement immediately jumps to the next iteration, ignoring any code below it within the loop block.

Let's see an example to understand the usage of `continue`:

```python
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    if num == 3:
        continue
    print(num)

print("Loop ended")
```

Output:

```
1
2
4
5
Loop ended
```

In this example, when the value of `num` becomes 3, the `continue` statement is encountered. As a result, the rest of the code within the loop block is skipped for that iteration, and the loop moves on to the next iteration. Therefore, the number 3 is not printed, and the loop continues with the numbers 4 and 5.

[ ]:

Understanding the `else` clause in loops   In Python, loops can have an optional `else` clause. The code within the `else` block is executed only when the loop has completed all its iterations normally, without encountering a `break` statement.

Consider the following example:

```python
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    if num == 6:
        break
    print(num)
else:
    print("Loop completed normally")

print("Loop ended")
```

Output:

```
1
2
3
4
5
Loop completed normally
Loop ended
```

In this example, the loop iterates over the `numbers` list. Since the value 6 is not present in the list, the loop completes all its iterations normally. Therefore, the code within the `else` block is executed, printing "Loop completed normally". After the loop, the program continues with the next statement, printing "Loop ended".

[ ]:

1.2.6   Exercises

1. Write a program that prompts the user to enter a series of numbers. The program should print the sum of all the numbers entered, but stop the loop and print the sum when the user enters a negative number. Use the `break` statement to exit the loop prematurely.

[ ]:

2. Write a program that prompts the user to enter a series of numbers. The program should skip any negative numbers entered and print only the positive numbers. Use the `continue` statement to skip the iterations with negative numbers.

[ ]:

3. Write a program that prompts the user to enter a series of numbers. The program should print whether the entered numbers are all even or not. Use the `else` clause in the loop to print the appropriate message after the loop has completed normally.

[ ]:

1.2.7 Nested Loops

In this section, we will explore the concept of nested loops in Python. Nested loops are loops that are placed inside another loop. They allow us to create complex patterns and structures by repeating a set of instructions multiple times.

Creating complex patterns and structures with nested loops Nested loops are particularly useful when we want to create patterns or structures that involve repeating a certain set of instructions multiple times. By combining multiple loops, we can achieve intricate designs and arrangements.

Let's consider an example where we want to print a pattern of stars in the shape of a triangle. We can achieve this using nested loops. Here's the code:

```python
for i in range(5):
    for j in range(i+1):
        print("*", end="")
    print()
```

In this code, we have an outer loop that iterates from 0 to 4. Inside the outer loop, we have an inner loop that iterates from 0 to the current value of the outer loop variable `i`. The inner loop prints a star for each iteration. After each inner loop iteration, we print a newline character to move to the next line.

The output of this code will be:

```
*
**
***
****
*****
```

By adjusting the range and the characters we print, we can create various patterns and structures using nested loops. Have a go yourself!

[ ]:

Examples of common nested loop scenarios

1. Multiplication Table: One common use of nested loops is to generate a multiplication table. Here's an example:

```python
for i in range(1, 11):
    for j in range(1, 11):
        print(i * j, end="\t")
    print()
```

This code will generate a multiplication table from 1 to 10.

[ ]:

2. Matrix Operations: Nested loops are often used to perform operations on matrices. For example, let's say we have two matrices A and B and we want to calculate their product C. Here's how we can do it using nested loops:

```python
A = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]

B = [[9, 8, 7],
     [6, 5, 4],
     [3, 2, 1]]

C = [[0, 0, 0],
     [0, 0, 0],
     [0, 0, 0]]

for i in range(len(A)):
    for j in range(len(B[0])):
        for k in range(len(B)):
            C[i][j] += A[i][k] * B[k][j]

for row in C:
    print(row)
```

This code multiplies matrices A and B and stores the result in matrix C.

[ ]:

1.2.8 Exercises

Some of these exercises might be challenging and you might struggle. This perfectly normal when it comes to programming, so don't worry - do your best in finding the answer and if you're truly stuck, ask for help!

Sum of Numbers

Write a program that asks the user for a positive integer n and calculates the sum of all numbers from 1 to n. Use a `while` loop to solve this problem.

Example:

```
Enter a positive integer: 5
The sum of numbers from 1 to 5 is 15.
```

[ ]:

Factorial Calculation

Write a program that asks the user for a positive integer n and calculates the factorial of n. The factorial of a number is the product of all positive integers less than or equal to that number. Use a `for` loop to solve this problem.

Example:

```
Enter a positive integer: 4
The factorial of 4 is 24.
```

[ ]:

Fibonacci Series

Write a program that asks the user for a positive integer **n** and prints the Fibonacci series up to the **n**th term. The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones. Use a **while** loop to solve this problem.

Example:

```
Enter a positive integer: 8
The Fibonacci series up to the 8th term is: 0, 1, 1, 2, 3, 5, 8, 13.
```

[ ]:

Prime Number Check

Write a program that asks the user for a positive integer **n** and checks if it is a prime number. A prime number is a number greater than 1 that has no positive divisors other than 1 and itself. Use a **for** loop to solve this problem.

Example:

```
Enter a positive integer: 7
7 is a prime number.
```

[ ]:

Multiplication Table

Write a program that asks the user for a positive integer **n** and prints the multiplication table of **n** up to 10. Use nested **for** loops to solve this problem.

Example:

```
Enter a positive integer: 5
Multiplication table of 5:
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

[ ]:

1.2.9 Practical examples of conditional loops

Conditional loops are commonly used in various scenarios. Let's look at a few practical examples to understand their usefulness.

Example 1: Finding even numbers

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = []
for num in numbers:
    if num % 2 == 0:
        even_numbers.append(num)
print("Even numbers:", even_numbers)
```

In this example, we have a list of numbers and we want to find all the even numbers. We use a `for` loop to iterate over each number in the list. The `if` statement checks if the number is divisible by 2 (i.e., an even number) using the modulo operator `%`. If the condition is true, the number is appended to the `even_numbers` list. Finally, we print the list of even numbers.

Example 2: User input validation

```python
valid_input = False
while not valid_input:
    age = input("Enter your age: ")
    if age.isdigit() and int(age) >= 18:
        valid_input = True
        print("Valid age!")
    else:
        print("Invalid age. Please enter a valid age.")
```

In this example, we use a `while` loop to repeatedly ask the user for their age until a valid input is provided. The `isdigit()` method checks if the input is a valid integer. If the input is a valid integer and greater than or equal to 18, the `valid_input` variable is set to `True` and a success message is printed. Otherwise, an error message is displayed and the loop continues.

[ ]:

1.2.10 Practical examples of conditional loops

Conditional loops are commonly used in various scenarios. Let's look at a few practical examples to understand their usefulness.

Example 1: Finding even numbers

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = []
for num in numbers:
    if num % 2 == 0:
        even_numbers.append(num)
print("Even numbers:", even_numbers)
```

In this example, we have a list of numbers and we want to find all the even numbers. We use a `for` loop to iterate over each number in the list. The `if` statement checks if the number is divisible

22

by 2 (i.e., an even number) using the modulo operator `%`. If the condition is true, the number is appended to the `even_numbers` list. Finally, we print the list of even numbers.

Example 2: User input validation

```python
valid_input = False
while not valid_input:
    age = input("Enter your age: ")
    if age.isdigit() and int(age) >= 18:
        valid_input = True
        print("Valid age!")
    else:
        print("Invalid age. Please enter a valid age.")
```

In this example, we use a `while` loop to repeatedly ask the user for their age until a valid input is provided. The `isdigit()` method checks if the input is a valid integer. If the input is a valid integer and greater than or equal to 18, the `valid_input` variable is set to `True` and a success message is printed. Otherwise, an error message is displayed and the loop continues.

[ ]:

1.2.11 Enhancing code efficiency through combined constructs

To improve code efficiency, we can combine different constructs such as loops and conditionals. This allows us to perform complex operations in a concise and optimized manner.

Example: Finding prime numbers

```python
numbers = [2, 3, 4, 5, 6, 7, 8, 9, 10]
prime_numbers = []
for num in numbers:
    is_prime = True
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            is_prime = False
            break
    if is_prime:
        prime_numbers.append(num)
print("Prime numbers:", prime_numbers)
```

In this example, we have a list of numbers and we want to find all the prime numbers. We use a `for` loop to iterate over each number in the list. Inside the loop, we use another `for` loop to check if the number is divisible by any number from 2 to the square root of the number. If the number is divisible, it is not a prime number and the inner loop is terminated using the `break` statement. If the number is not divisible by any number, it is considered a prime number and is appended to the `prime_numbers` list. Finally, we print the list of prime numbers.

By combining loops with conditionals, you can solve a wide range of problems efficiently. Let's practice these concepts:

1.3 Exercises

1. Write a program that prints all the odd numbers from 1 to 20 using a loop and a conditional statement.

`[ ]:`

2. Write a program that asks the user for a number and prints whether it is a prime number or not.

`[ ]:`

3. Write a program that calculates the sum of all the numbers divisible by 3 or 5 from 1 to 100 using a loop and a conditional statement.

`[ ]:`

Loop Control with `break` and `continue`

Using `break` and `continue` statements within loops provides us with powerful tools to control the flow of our loops and create flexible control structures for complex scenarios.

Employing `break` and `continue` within loops with conditions As you'll remember, the `break` statement is used to exit a loop prematurely. When encountered, it immediately terminates the loop and transfers the control to the next statement after the loop. This can be useful when we want to stop the execution of a loop based on a certain condition.

Let's consider an example where we want to find the first even number in a list of integers:

```
numbers = [1, 3, 5, 2, 7, 4, 6, 9, 8]

for num in numbers:
    if num % 2 == 0:
        print("The first even number is:", num)
        break
```

Output:

```
The first even number is: 2
```

In this example, the loop iterates over each number in the `numbers` list. When it encounters the number 2, which is an even number, the `break` statement is executed, and the loop is terminated. As a result, we only print the first even number and exit the loop.

On the other hand, you'll remember that the `continue` statement is used to skip the rest of the code within a loop for the current iteration and move on to the next iteration. This can be useful when we want to skip certain elements or perform specific actions only for certain elements.

Let's consider an example where we want to print all the odd numbers in a list of integers:

```
numbers = [1, 3, 5, 2, 7, 4, 6, 9, 8]

for num in numbers:
```

```
    if num % 2 == 0:
        continue
    print("Odd number:", num)
```

Output:

```
Odd number: 1
Odd number: 3
Odd number: 5
Odd number: 7
Odd number: 9
```

In this example, the loop iterates over each number in the `numbers` list. When it encounters an even number, the `continue` statement is executed, and the rest of the code within the loop is skipped for that iteration. As a result, we only print the odd numbers in the list.

[ ]:

Creating flexible control structures for complex scenarios By combining `break` and `continue` statements with conditional statements, we can create flexible control structures to handle complex scenarios.

Let's consider an example where we want to find the first prime number in a list of integers:

```
numbers = [1, 3, 5, 4, 7, 9, 8, 11, 13]

for num in numbers:
    if num < 2:
        continue
    for i in range(2, num):
        if num % i == 0:
            break
    else:
        print("The first prime number is:", num)
        break
```

Output:

```
The first prime number is: 3
```

In this example, we use a nested loop to check if each number in the `numbers` list is prime. The outer loop iterates over each number, and the inner loop checks if the number is divisible by any number from 2 to the number itself. If a divisor is found, the inner loop is terminated using the `break` statement. However, if no divisor is found, the `else` block is executed, and we print the first prime number before terminating the outer loop using another `break` statement.

By utilizing `break` and `continue` statements effectively, we can create more efficient and concise code to handle various scenarios within loops.

Now, it's time for you to practice using `break` and `continue` statements in loops. Complete the following exercises to reinforce your understanding.

Take your time to solve these exercises and feel free to ask for help if needed. Good luck!

1.3.1 Exercises

1. Write a program that finds the first multiple of 7 in a list of integers. Use the `break` statement to terminate the loop once the multiple is found.

`[ ]:`

2. Write a program that prints all the uppercase letters in a string. Use the `continue` statement to skip lowercase letters.

`[ ]:`

3. Write a program that finds the first positive number greater than 1000 in a list of integers. Use a combination of `break` and `continue` statements to handle different scenarios.

`[ ]:`

4. Sum of Even Numbers

Write a program that calculates the sum of all even numbers between 1 and a given number `n`. The program should prompt the user to enter the value of `n` and then display the sum. Have a look at the `input` function.

Example:

```
Enter a number: 10
The sum of even numbers between 1 and 10 is 30.
```

`[ ]:`

5. Factorial Calculation

Write a program that calculates the factorial of a given number `n`. The factorial of a number is the product of all positive integers less than or equal to that number. The program should prompt the user to enter the value of `n` and then display the factorial.

Example:

```
Enter a number: 5
The factorial of 5 is 120.
```

`[ ]:`

6. Prime Number Check

Write a program that checks whether a given number `n` is prime or not. A prime number is a number greater than 1 that has no positive divisors other than 1 and itself. The program should prompt the user to enter the value of `n` and then display whether it is prime or not.

Example:

```
Enter a number: 7
7 is a prime number.
```

Exception Handling

Exception handling is an essential concept in programming that allows us to handle errors and unexpected situations in a controlled manner. In this section, we will explore the basics of exception handling in Python and learn how to use `try`, `except`, `else`, and `finally` blocks to handle exceptions effectively.

Understanding exceptions and error handling In Python, an exception is an event that occurs during the execution of a program and disrupts the normal flow of the program. When an exception occurs, the program terminates abruptly unless it is handled properly. Error handling is the process of dealing with these exceptions to prevent program crashes and provide meaningful feedback to the user.

Exceptions can occur due to various reasons, such as invalid input, file not found, division by zero, or accessing an index out of range. Python provides a wide range of built-in exceptions, and you can also create your own custom exceptions to handle specific situations.

Using `try`, `except`, `else`, and `finally` blocks To handle exceptions in Python, we use the `try` statement along with one or more `except` clauses. The basic syntax is as follows:

```python
try:
    # Code that may raise an exception
except ExceptionType1:
    # Code to handle ExceptionType1
except ExceptionType2:
    # Code to handle ExceptionType2
else:
    # Code to execute if no exception occurred
finally:
    # Code that will always execute, regardless of exceptions
```

Here's a breakdown of the different blocks:

- The `try` block contains the code that may raise an exception. If an exception occurs within this block, the control is transferred to the appropriate `except` block.
- Each `except` block specifies the type of exception it can handle. If the exception raised matches the type specified in an `except` block, the code within that block is executed.
- The `else` block is optional and is executed only if no exception occurs in the `try` block. It is typically used to perform actions that should only happen when no exceptions are raised.
- The `finally` block is also optional and is executed regardless of whether an exception occurred or not. It is commonly used to release resources or perform cleanup operations.

Let's look at an example to understand how exception handling works:

```python
try:
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2
    print("The result is:", result)
```

```python
except ValueError:
    print("Invalid input. Please enter a valid number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("No exceptions occurred.")
finally:
    print("This will always execute.")
```

In the above example, we ask the user to enter two numbers and perform division. If the user enters invalid input (e.g., a non-numeric value), a `ValueError` exception is raised and handled by the first `except` block. If the user enters zero as the second number, a `ZeroDivisionError` exception is raised and handled by the second `except` block. If no exceptions occur, the code in the `else` block is executed. Finally, the code in the `finally` block is always executed, regardless of exceptions.

Handling unexpected errors gracefully   While it is important to handle specific exceptions, it is equally important to handle unexpected errors gracefully. To achieve this, we can use a generic `except` block that catches all exceptions. However, it is generally recommended to handle specific exceptions whenever possible, as catching all exceptions can hide potential bugs in the code.

Here's an example that demonstrates handling unexpected errors:

```python
try:
    # Code that may raise an exception
except Exception as e:
    print("An unexpected error occurred:", str(e))
```

In the above example, the `except` block catches any exception that occurs and prints a generic error message along with the exception details. This helps in identifying and debugging unexpected errors during development.

[ ]:

[ ]:

1.3.2   Exercises

1. Write a program that asks the user to enter two numbers and calculates their sum. Handle any possible exceptions that may occur during the input and calculation process.

[ ]:

2. Create a function that takes a list as input and returns the average of the numbers in the list. Handle any exceptions that may occur, such as an empty list or non-numeric values in the list.

[ ]:

3. Write a program that reads a file and prints its contents. Handle any exceptions that may occur, such as the file not found or permission denied.

```
[ ]:
```

List Comprehensions

List comprehensions are a powerful feature in Python that allow you to create lists in a concise and efficient manner. They provide a compact syntax for generating lists based on existing lists or other iterable objects. In this section, we will explore the syntax and examples of list comprehensions, and how they can improve code readability and performance.

Syntax of List Comprehensions The basic syntax of a list comprehension consists of square brackets enclosing an expression followed by a `for` clause and an optional `if` clause. The general structure is as follows:

```python
new_list = [expression for item in iterable if condition]
```

- `new_list`: The list that will be created by the list comprehension.
- `expression`: The expression that will be evaluated and added to the new list.
- `item`: The variable that represents each item in the iterable.
- `iterable`: The existing list or other iterable object that will be used to generate the new list.
- `condition` (optional): An optional condition that filters the items from the iterable based on a specified criteria.

Examples of List Comprehensions Let's look at some examples to understand how list comprehensions work:

Example 1: Creating a list of squares of numbers from 1 to 10

```python
squares = [x**2 for x in range(1, 11)]
print(squares)
```

Output:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Example 2: Creating a list of even numbers from an existing list

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = [x for x in numbers if x % 2 == 0]
print(even_numbers)
```

Output:

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

Example 3: Creating a list of uppercase letters from a string

```python
string = "Hello, World!"
uppercase_letters = [char for char in string if char.isupper()]
print(uppercase_letters)
```

Output:

```
['H', 'W']
```

Improving Code Readability and Performance   List comprehensions can greatly improve the readability of your code by providing a concise and expressive way to create lists. They eliminate the need for writing explicit `for` loops and conditional statements, making the code more compact and easier to understand.

In addition to improving code readability, list comprehensions can also enhance the performance of your code. They are generally faster than traditional `for` loops because they are implemented in C under the hood. This makes them a preferred choice when dealing with large datasets or computationally intensive tasks.

1.3.3   Exercises (optional)

1. Create a list of the first 10 multiples of 3 using a list comprehension.

`[ ]:`

2. Given a list of numbers, create a new list that contains only the positive numbers using a list comprehension.

`[ ]:`

3. Create a list of the squares of even numbers from 1 to 20 using a list comprehension.

`[ ]:`

4. Given a string, create a list of all the vowels present in the string using a list comprehension.

`[ ]:`

1.4   Conclusion

In this extensive chapter, you've gained a deep understanding of loops and conditionals in Python, which are essential tools for creating dynamic and responsive programs. **Practice and apply these concepts in various scenarios to become a proficient Python programmer.**

`[ ]:`

06Functions

November 29, 2023

Dr Gunnar Mallon (g.mallon@rug.nl), *Department of Cultural Geography (Faculty of Spatial Science), University of Groningen*

Functions allow you to organize your code into reusable, modular components. They are blocks of code that perform a specific task and allow you to break down your program into smaller, more manageable pieces. Functions can be reused multiple times, making your code more organized and easier to maintain.

Advantages of using functions for code organization and reusability: - **Modularity**: Functions allow you to divide your program into smaller, self-contained modules. Each function can focus on a specific task, making it easier to understand and debug. - **Code Reusability**: Once you define a function, you can call it multiple times from different parts of your program. This saves you from writing the same code over and over again. - **Readability**: Functions make your code more readable and understandable. By giving meaningful names to your functions, you can convey the purpose of the code block at a glance.

Before you can "call" a function, such as the `print` function, you need to define (or declare) it. This lets the python compiler know what to do. Let's look at both in more detail.

0.1 Function Definition

Syntax for defining functions:

```python
def function_name(parameter1, parameter2, ...):
    # code block
    # perform some task
    # return a value (optional)
```

Creating named functions with parameters:

```python
def greet(name):
    print("Hello, " + name + "!")


def add_numbers(a, b):
    return a + b
```

The role of function names, parameters, and indentation: - **Function Name**: The name of the function should be descriptive and reflect the task it performs. It should follow the naming conventions of Python (e.g., lowercase with words separated by underscores). - **Parameters**: Parameters are placeholders for the values that the function expects to receive when it is called. They allow

you to pass data into the function. Think of the `print` function. - **Indentation**: The code block inside the function is indented to indicate that it belongs to the function. Python uses indentation to define the scope of the code.

```
[ ]:
```

0.2 Function Calling

Syntax for calling functions with arguments:

```
function_name(argument1, argument2, ...)
```

Passing arguments by position and by keyword: - **Positional Arguments**: When calling a function, you can pass arguments by their position. The order of the arguments matters, and they are assigned to the parameters in the same order. - **Keyword Arguments**: Alternatively, you can pass arguments by specifying the parameter name followed by the argument value. This allows you to pass arguments in any order, as long as you specify the parameter name.

0.3 The *return* Statement

The `return` statement is used to send data back from a function. It allows the function to produce a result that can be used in other parts of the program.

```python
def square(number):
    return number ** 2
```

Function without Return Value: If a function does not have a return statement, it is considered to have a return value of `None`. This is useful for functions that perform actions without producing a result.

Multiple return statements in a function:

```python
def divide(a, b):
    if b == 0:
        return "Error: Cannot divide by zero"
    else:
        return a / b
```

```
[ ]:
```

0.4 Function Documentation (Docstrings)

Writing descriptive documentation for functions is import to keep the code readable.

- **Docstrings**: Docstrings are used to provide a description of what a function does. They are enclosed in triple quotes (`"""`) and placed immediately after the function definition.
- **Importance of Docstrings**: Docstrings are essential for code readability and maintainability. They help other programmers understand how to use your function and what to expect from it.
- **Accessing Docstrings**: You can access the docstring of a function using the `help()` function or by accessing the `__doc__` attribute of the function.

2

```python
def square(n):
    """Takes in a number n, returns the square of n"""
    return n**2

print(square.__doc__)
```

What output will the code above give?

`[ ]:`

0.5   Exercises

1. Write a function that takes a string as input and prints it in reverse order.

`[ ]:`

2. Create a function that calculates the area of a rectangle given its length and width.

`[ ]:`

3. Write a function that takes a list of numbers as input and returns the sum of all the even numbers in the list.

`[ ]:`

4. Document each of the above functions using docstrings to practice good code documentation. (You may have to look this up)

`[ ]:`

---

1   Function Parameters and Arguments

In Python, a parameter is a variable that is used to receive input in a function. It acts as a placeholder for the values that will be passed to the function when it is called. Parameters allow us to make our functions more flexible and reusable.

There are different types of function parameters:

1. **Required Parameters**: These are parameters that must be provided when calling the function. They are necessary for the function to work correctly.

2. **Default Parameters**: These are parameters that have a default value assigned to them. If no value is provided for these parameters when calling the function, the default value will be used.

3. **Variable-Length Parameters**: These parameters allow us to pass a variable number of arguments to a function. They are useful when we don't know in advance how many arguments will be passed.

1.1 Positional Arguments

Positional arguments are the most common type of arguments used in Python functions. They are passed to the function in the same order as they are defined in the function's parameter list.

Here's an example:

```python
def greet(name, age):
    print(f"Hello {name}, you are {age} years old.")

greet("Alice", 25)
```

Output:

```
Hello Alice, you are 25 years old.
```

In the example above, the function `greet` has two parameters: `name` and `age`. When we call the function and pass the arguments `"Alice"` and `25`, they are matched to the parameters based on their positions.

It's important to note that the order of the arguments must match the order of the parameters. If we swap the arguments, the output will be different:

```python
greet(25, "Alice")
```

Output:

```
Hello 25, you are Alice years old.
```

This is because the arguments are assigned to the parameters based on their positions, not their names.

When working with positional arguments, it's important to be careful with the order of the arguments and ensure that they match the parameter positions.

```
[ ]:
```

1.2 Keyword Arguments

Keyword arguments allow us to pass arguments to a function using the parameter names. This improves the readability of the code, as it becomes clear which argument is being passed to which parameter.

Here's an example:

```python
def greet(name, age):
    print(f"Hello {name}, you are {age} years old.")

greet(name="Alice", age=25)
```

Output:

```
Hello Alice, you are 25 years old.
```

In the example above, we are using keyword arguments to pass the values to the parameters. This makes it clear that the argument `"Alice"` is being passed to the `name` parameter, and the argument `25` is being passed to the `age` parameter.

We can also mix positional and keyword arguments:

```python
greet("Alice", age=25)
```

Output:

```
Hello Alice, you are 25 years old.
```

In this case, we are using a positional argument for the `name` parameter and a keyword argument for the `age` parameter. The order of the arguments is still important for the positional argument, but the keyword argument can be placed in any order.

Using keyword arguments can make our code more readable and self-explanatory, especially when dealing with functions that have many parameters.

[ ]:

1.3 Default Parameter Values

Default parameter values allow us to assign a default value to a parameter. If no value is provided for that parameter when calling the function, the default value will be used.

Here's an example:

```python
def greet(name, age=18):
    print(f"Hello {name}, you are {age} years old.")

greet("Alice")
```

Output:

```
Hello Alice, you are 18 years old.
```

In the example above, the `age` parameter has a default value of `18`. When we call the function without providing a value for `age`, the default value is used.

We can also override the default value by providing a different value:

```python
greet("Bob", 30)
```

Output:

```
Hello Bob, you are 30 years old.
```

In this case, the default value of `18` is overridden by the value `30` that we provided when calling the function.

Default parameter values are useful when we want to make certain parameters optional. They allow us to provide a default behavior for the function while still allowing the caller to customize it if needed.

[ ]:

1.4 Variable-Length Argument Lists

Variable-length argument lists allow us to pass a variable number of arguments to a function. This is useful when we don't know in advance how many arguments will be passed.

In Python, we can use the `*args` syntax to define a parameter that will collect all the positional arguments into a tuple.

Here's an example:

```python
def sum_numbers(*args):
    total = 0
    for num in args:
        total += num
    return total

print(sum_numbers(1, 2, 3, 4, 5))
```

Output:

```
15
```

In the example above, the `sum_numbers` function accepts any number of arguments and adds them together. The `*args` parameter collects all the positional arguments into a tuple, which we can then iterate over to perform the desired operation.

Variable-length argument lists are particularly useful when we want to create functions that can handle a varying number of inputs.

[ ]:

1.5 Exercises

1. Write a function called `greet_person` that takes two parameters: `name` and `time_of_day`. The function should print a greeting message that includes the person's name and the time of day. If no time of day is provided, the default value should be `Morning`. Test the function by calling it with different values for `name` and `time_of_day`.

[ ]:

2. Write a function called `smallest_number` that takes a variable number of arguments. The function should return the smallest of all the numbers passed as arguments. Test the function by calling it with different numbers.

[ ]:

4. Write a function called `print_info` that takes a person's information as keyword arguments: `name`, `age`, and `city`. The function should print the person's information in a formatted way. Test the function by calling it with different values for the keyword arguments.

[ ]:

2 Scope and Lifetime of Variables

In this section, we will explore the concept of variable scope and lifetime in Python. Understanding how variables are scoped and how they interact with different parts of your code is crucial for writing efficient and bug-free programs. We will cover topics such as local and global variables, the LEGB rule for variable resolution, modifying global variables using the `global` keyword, and more.

2.1 Variable Scope

In Python, the scope of a variable determines its visibility and accessibility throughout the code. The scope of a variable is defined by where it is declared or assigned. The two main types of variable scope are local and global.

2.1.1 Local and global variables

A local variable is a variable that is defined within a specific block of code, such as a function. It can only be accessed and used within that block of code. Once the block of code is executed, the local variable is destroyed and its value is no longer accessible.

```python
def my_function():
    x = 10  # local variable
    print(x)

my_function()  # Output: 10
print(x)  # Error: NameError: name 'x' is not defined
```

In the example above, `x` is a local variable within the `my_function()` function. It can only be accessed within the function, and trying to access it outside of the function will result in a `NameError`.

A global variable, on the other hand, is a variable that is defined outside of any function or block of code. It can be accessed and used throughout the entire program, including within functions.

```python
x = 10  # global variable

def my_function():
    print(x)

my_function()  # Output: 10
print(x)  # Output: 10
```

In this example, `x` is a global variable. It is defined outside of any function and can be accessed both within the `my_function()` function and outside of it.

```
[ ]:
```

2.1.2 The LEGB (Local, Enclosing, Global, Built-in) rule for variable resolution

When a variable is referenced in Python, the interpreter follows a specific order to resolve the variable name. This order is known as the LEGB rule:

1. Local (L): The interpreter first looks for the variable in the local scope, i.e., within the current function or block of code.
2. Enclosing (E): If the variable is not found in the local scope, the interpreter looks in the enclosing scope. This applies to nested functions, where each level of nesting has its own scope.
3. Global (G): If the variable is not found in the local or enclosing scope, the interpreter looks in the global scope, i.e., the module-level scope.
4. Built-in (B): If the variable is not found in any of the above scopes, the interpreter finally looks for it in the built-in scope, which contains Python's built-in functions and modules.

```python
x = 10   # global variable

def my_function():
    x = 5   # local variable
    print(x)

my_function()   # Output: 5
print(x)   # Output: 10
```

In this example, the variable x is first searched for in the local scope of the `my_function()` function. Since it is found there, the local value of x (5) is printed. However, when we try to print x outside of the function, the global value of x (10) is printed instead.

[ ]:

3 Lambda Functions

Lambda functions, also known as anonymous functions, are a powerful feature in Python that allow you to define small, one-line functions without a name. They are particularly useful when you need to create a function for a specific task that you don't plan on using again.

The main advantage of lambda functions is their simplicity and conciseness. They can be defined in a single line of code, making them easy to read and understand. Lambda functions are commonly used in situations where a regular function would be overkill or when you need a quick function definition.

Lambda functions are defined using the `lambda` keyword, followed by a list of arguments, a colon, and the expression that the function will evaluate. The result of the expression is automatically returned by the lambda function.

```python
lambda arguments: expression
```

Here's an example of a lambda function that calculates the square of a number:

```python
square = lambda x: x ** 2
print(square(5))   # Output: 25
```

In this example, the lambda function takes a single argument x and returns the square of x. The lambda function is assigned to the variable `square`, and we can call it like a regular function.

Lambda functions can also have multiple arguments. For example, here's a lambda function that calculates the sum of two numbers:

```python
add = lambda x, y: x + y
print(add(3, 4))  # Output: 7
```

In this case, the lambda function takes two arguments x and y and returns their sum.

Lambda functions can be used in various scenarios, such as sorting, filtering, and mapping data. They are particularly useful when you need to pass a function as an argument to another function, as we will see in the next sections.

3.0.1 Examples

Let's see some practical examples of lambda functions:

```python
# Example 1: Multiply two numbers

multiply = lambda x, y: x * y
print(multiply(2, 3))  # Output: 6

# Example 2: Check if a number is even

is_even = lambda x: x % 2 == 0
print(is_even(4))  # Output: True
print(is_even(5))  # Output: False

# Example 3: Convert a string to uppercase

to_upper = lambda s: s.upper()
print(to_upper("hello"))  # Output: "HELLO"
```

In these examples, we define lambda functions for different tasks. The first lambda function multiplies two numbers, the second lambda function checks if a number is even, and the third lambda function converts a string to uppercase.

3.1 Using Lambda with map(), filter(), and reduce()

Lambda functions are commonly used in combination with built-in functions like `map()`, `filter()`, and `reduce()` to perform operations on sequences.

3.1.1 Applying lambda functions to sequences with map()

The `map()` function applies a given function to each item in a sequence and returns a new sequence with the results. It takes two arguments: the function to apply and the sequence to apply it to.

Here's an example that uses a lambda function with `map()` to calculate the square of each number in a list:

```python
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

In this example, the lambda function `lambda x: x ** 2` is applied to each item in the `numbers` list using `map()`. The result is a new list `squared_numbers` containing the squares of the original

numbers.

[ ]:

3.1.2 Filtering elements with filter() and lambda functions

The `filter()` function filters a sequence based on a given condition. It takes two arguments: the function that defines the condition and the sequence to filter.

Here's an example that uses a lambda function with `filter()` to filter out the even numbers from a list:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)   # Output: [2, 4]
```

In this example, the lambda function `lambda x: x % 2 == 0` is used with `filter()` to filter out the even numbers from the `numbers` list. The result is a new list `even_numbers` containing only the even numbers.

[ ]:

3.1.3 Reducing sequences with reduce() and lambda functions

The `reduce()` function applies a given function to the first two items in a sequence, then applies it to the result and the next item, and so on, until the sequence is reduced to a single value. It takes two arguments: the function to apply and the sequence to reduce.

Here's an example that uses a lambda function with `reduce()` to calculate the product of all numbers in a list:

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)   # Output: 120
```

In this example, the lambda function `lambda x, y: x * y` is used with `reduce()` to calculate the product of all numbers in the `numbers` list. The result is the product of all the numbers, which is 120.

3.2 Exercises

1. Write a lambda function that takes a string as input and returns the length of the string.

[ ]:

2. Use a lambda function with `map()` to convert a list of strings to uppercase.

[ ]:

3. Use a lambda function with `filter()` to filter out the numbers divisible by 3 from a list of integers.

`[ ]:`

4. Use a lambda function with `reduce()` to find the maximum number in a list of integers.

`[ ]:`

3.3 Conclusion

In this chapter, you've delved deep into the world of functions, a vital component of any Python program. You've learned how to define, call, and document functions, as well as work with different types of parameters. Understanding the scope and lifetime of variables within functions is crucial for writing efficient and reliable code.

07StandardLibraries

November 29, 2023

Dr Gunnar Mallon (g.mallon@rug.nl), *Department of Cultural Geography (Faculty of Spatial Science), University of Groningen*

Python's standard libraries are a vast collection of pre-built modules and packages that extend Python's capabilities. Leveraging these libraries can significantly enhance your programming productivity.

Standard libraries in Python are pre-written modules that provide a wide range of functionality to perform common tasks. These libraries are included with the Python programming language and can be used directly in your programs without the need for additional installation. They are designed to save time and effort by providing ready-to-use functions and classes for various purposes.

The role of standard libraries in Python is to provide a set of tools and utilities that simplify the development process. These libraries encapsulate complex operations and algorithms, allowing you to focus on solving higher-level problems rather than reinventing the wheel. By utilizing standard libraries, you can write more efficient and maintainable code.

Using standard libraries is important for efficiency because it allows you to leverage the expertise and experience of the Python community. These libraries are developed and maintained by a large community of contributors, ensuring that they are well-tested, optimized, and reliable. By using standard libraries, you can take advantage of the collective knowledge and effort of the Python community, saving time and effort in the development process.

0.1 Commonly Used Standard Libraries

Python provides a rich set of standard libraries that cover a wide range of domains. Here are some commonly used standard libraries:

- `math`: This library provides mathematical operations and functions, such as trigonometric functions, logarithmic functions, and mathematical constants.
- `random`: This library allows you to generate random numbers and perform random selections. It is useful for simulations, games, and cryptography.
- `datetime`: This library provides classes for working with dates, times, and time intervals. It allows you to perform various operations, such as formatting dates, calculating time differences, and parsing date strings.
- `os`: This library provides functions for interacting with the operating system. It allows you to perform operations such as file and directory manipulation, environment variable access, and process management.

- **json**: This library provides functions for working with JSON (JavaScript Object Notation) data. It allows you to encode Python objects into JSON strings and decode JSON strings into Python objects.
- **re**: This library provides support for regular expressions, which are powerful tools for text processing and pattern matching. It allows you to search, replace, and manipulate strings based on patterns.
- **collections**: This library provides additional data structures beyond the built-in lists and dictionaries. It includes useful data structures such as named tuples, ordered dictionaries, and counters.

Some of these you will have already encountered. Let's look at the first three in more detail.

0.1.1   Importing Standard Libraries

To use standard libraries in your Python programs, you need to import them first. The `import` statement is used to import standard libraries and modules into your program. You can import standard libraries in different ways. Here are some of the most common ways to import standard libraries:

- Importing the entire library:

```python
import math
print(math.sqrt(25))   # Output: 5.0
```

- Importing specific functions or classes from a library:

```python
from math import sqrt
print(sqrt(25))   # Output: 5.0
```

- Importing a library with an alias:

```python
import math as m
print(m.sqrt(25))   # Output: 5.0
```

It is generally recommended to import the entire library or import specific functions/classes rather than using the `from library import *` syntax, which you might see in some older code. This helps avoid naming conflicts and makes it clear which library a function or class belongs to.

`[ ]:`

0.1.2   The Python Standard Library Documentation

The official Python Standard Library documentation is a valuable resource for understanding and using standard libraries. It provides detailed information on available modules, functions, classes, and their usage.

Python Standard Library Documentation

Here are some tips for using the documentation effectively:

- **Exploring the documentation**: The documentation is organized into modules, each covering a specific topic or functionality. You can browse the documentation to find modules related to your needs.

- **Finding information on modules and functions**: Each module and function has its own documentation page. You can search for a specific module or function to find its documentation page.

- **Understanding module usage and options**: The documentation provides examples and explanations of how to use each module and its functions. It also describes the available options and parameters for each function.

0.2 The `math` Library

The `math` library in Python provides a set of functions and constants for performing common mathematical operations. It provides functions for performing basic arithmetic operations such as addition, subtraction, multiplication, and division. Here are some examples:

```python
import math

# Addition
result = math.add(2, 3)   # Returns 5

# Subtraction
result = math.subtract(5, 2)   # Returns 3

# Multiplication
result = math.multiply(4, 3)   # Returns 12

# Division
result = math.divide(10, 2)   # Returns 5
```

0.2.1 Functions for trigonometry, logarithms, exponentials, and more

In addition to basic arithmetic operations, the `math` library also provides functions for trigonometry, logarithms, exponentials, and more. Here are some examples:

```python
import math

# Trigonometry
result = math.sin(math.pi/2)   # Returns 1.0

# Logarithms
result = math.log(10)   # Returns 2.302585092994046

# Exponentials
result = math.exp(2)   # Returns 7.389056098930650495
```

0.2.2 Mathematical constants available in the library

The `math` library also provides access to several mathematical constants, such as pi ( ) and Euler's number (e). Here are some examples:

```python
import math
```

```python
# Pi
result = math.pi   # Returns 3.141592653589793

# Euler's number
result = math.e   # Returns 2.718281828459045
```

0.3 The `random` Library

The `random` library in Python allows you to generate random numbers and sequences. It is useful for simulating random events and probability distributions. Here are some examples:

```python
import random

# Random number between 0 and 1
result = random.random()   # Returns a random float between 0 and 1

# Random integer between a range
result = random.randint(1, 10)   # Returns a random integer between 1 and 10

# Random choice from a sequence
result = random.choice(['apple', 'banana', 'orange'])   # Returns a random element from the seq
```

0.3.1 Setting seeds for reproducibility

If you want to generate the same random numbers or sequences multiple times, you can set a seed value using the `random.seed()` function. This ensures reproducibility of results. Here is an example:

```python
import random

random.seed(42)   # Set the seed value to 42

result = random.random()   # Returns the same random float between 0 and 1 every time
```

0.3.2 Simulating random events and probability distributions

The `random` library also provides functions for simulating random events and probability distributions. Here are some examples:

```python
import random

# Simulating a coin toss
result = random.choice(['Heads', 'Tails'])   # Returns either 'Heads' or 'Tails'

# Simulating a dice roll
result = random.randint(1, 6)   # Returns a random integer between 1 and 6

# Simulating a normal distribution
result = random.gauss(0, 1)   # Returns a random float from a normal distribution with mean 0 a
```

0.4 Exercises

Now that you have learned about the `math` and `random` libraries, it's time to apply your knowledge through practice exercises. These exercises will help you gain proficiency in working with these libraries for various mathematical and probabilistic problems.

1. Write a program that calculates the area of a circle using the `math` library. Prompt the user to enter the radius of the circle and display the result.

`[ ]:`

2. Generate a random password of length 8 using the `random` library. The password should contain a combination of uppercase letters, lowercase letters, and digits.

`[ ]:`

3. Simulate rolling a pair of dice 100 times using the `random` library. Count the number of times each possible sum (2 to 12) occurs and display the results.

`[ ]:`

4. Write a program that generates 10 random numbers between 1 and 100 using the `random` library. Calculate the mean, median, and mode of the generated numbers.

`[ ]:`

0.5 The `datetime` Library

In this section, we will learn how to handle dates, times, and timestamps using the `datetime` library in Python - remember from previous exercise, this is something that you've already seen. The `datetime` library provides classes for manipulating dates and times, allowing us to perform various operations such as creating and formatting date and time objects, as well as performing calculations with dates and times.

Handling dates, times, and timestamps The `datetime` library provides three main classes for handling dates, times, and timestamps:

1. `datetime.date`: This class represents a date (year, month, day) and provides methods for manipulating dates, such as getting the current date, extracting specific components of a date, and performing arithmetic operations on dates.

Example:

```
import datetime

today = datetime.date.today()
print(today)  # Output: 2022-01-01

year = today.year
month = today.month
```

```
day = today.day
print(year, month, day)   # Output: 2022 1 1
```

2. `datetime.time`: This class represents a time (hour, minute, second, microsecond) and provides methods for manipulating times, such as getting the current time, extracting specific components of a time, and performing arithmetic operations on times.

Example:

```
import datetime

current_time = datetime.datetime.now().time()
print(current_time)   # Output: 12:34:56.789012

hour = current_time.hour
minute = current_time.minute
second = current_time.second
microsecond = current_time.microsecond
print(hour, minute, second, microsecond)   # Output: 12 34 56 789012
```

3. `datetime.datetime`: This class represents a combination of date and time and provides methods for manipulating date and time objects, such as getting the current date and time, extracting specific components of a datetime, and performing arithmetic operations on date-times.

Example:

```
import datetime

current_datetime = datetime.datetime.now()
print(current_datetime)   # Output: 2022-01-01 12:34:56.789012

year = current_datetime.year
month = current_datetime.month
day = current_datetime.day
hour = current_datetime.hour
minute = current_datetime.minute
second = current_datetime.second
microsecond = current_datetime.microsecond
print(year, month, day, hour, minute, second, microsecond)   # Output: 2022 1 1 12 34 56 789012
```

[ ]:

[ ]:

0.5.1 Creating and formatting date and time objects

The `datetime` library allows us to create date and time objects using the `datetime.date()`, `datetime.time()`, and `datetime.datetime()` constructors. *The term "constructor" comes from Object Oriented Programming - something that we will not cover in this course.* We can specify the

year, month, day, hour, minute, second, and microsecond values to create specific date and time objects.

Example:

```
import datetime

# Creating a date object
date_obj = datetime.date(2022, 1, 1)
print(date_obj)   # Output: 2022-01-01

# Creating a time object
time_obj = datetime.time(12, 34, 56, 789012)
print(time_obj)   # Output: 12:34:56.789012

# Creating a datetime object
datetime_obj = datetime.datetime(2022, 1, 1, 12, 34, 56, 789012)
print(datetime_obj)   # Output: 2022-01-01 12:34:56.789012
```

The `datetime` library also provides various formatting options to represent dates and times in different formats using the `strftime()` method. We can specify the format codes to format the date and time objects according to our requirements.

Example:

```
import datetime

current_datetime = datetime.datetime.now()

# Formatting datetime object as a string
formatted_datetime = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
print(formatted_datetime)   # Output: 2022-01-01 12:34:56

# Formatting datetime object as a custom string
custom_formatted_datetime = current_datetime.strftime("%A, %B %d, %Y")
print(custom_formatted_datetime)   # Output: Saturday, January 01, 2022
```

[ ]: _____

[ ]: _____

0.5.2 Performing calculations with dates and times

The `datetime` library allows us to perform various calculations with dates and times, such as finding the difference between two dates or times, adding or subtracting a duration from a date or time, and comparing dates or times.

Example:

```
import datetime

# Calculating the difference between two dates
```

```python
date1 = datetime.date(2022, 1, 1)
date2 = datetime.date(2023, 1, 1)
date_diff = date2 - date1
print(date_diff.days)   # Output: 365

# Adding a duration to a date
date = datetime.date(2022, 1, 1)
duration = datetime.timedelta(days=30)
new_date = date + duration
print(new_date)   # Output: 2022-01-31

# Comparing two dates
date1 = datetime.date(2022, 1, 1)
date2 = datetime.date(2023, 1, 1)
print(date1 < date2)   # Output: True
```

0.6 Exercises

1. Calculate the duration between two given dates and print the result in days, hours, minutes, and seconds.

[ ]:

2. Parse a date string in the format "YYYY-MM-DD" and print the day of the week.

[ ]:

3. Convert a given datetime object from one time zone to another using the `pytz` library.

[ ]:

4. Calculate the difference in hours between the current time and a given time in a different time zone.

[ ]:

5. Write a program that calculates the area of a circle using the `math` library. Prompt the user for the radius and display the result.

[ ]:

6. Generate a random password of length 8 using the `random` library. The password should contain a mix of uppercase letters, lowercase letters, and digits.

[ ]:

7. Write a program that takes a date in the format "YYYY-MM-DD" and calculates the number of days between that date and the current date using the `datetime` library.

[ ]:

0.7 Conclusion

In this chapter, you've learned how to harness the power of Python's standard libraries to streamline your programming tasks. These libraries provide ready-to-use solutions for various domains, from mathematics to date handling, file management, text processing, and advanced data structures.

With a strong understanding of standard libraries, you'll be well-equipped to tackle a wide range of programming challenges efficiently.

[ ]:

08ReadingFiles

November 29, 2023

Dr Gunnar Mallon (g.mallon@rug.nl), *Department of Cultural Geography (Faculty of Spatial Science), University of Groningen*

In this lecture, we'll explore how to read files in Python. Whether you're working with text files, CSVs, JSON, or any other data format, the ability to read and process files is crucial for various programming tasks.

0.1 File Handling Basics

File input/output (I/O) is a fundamental concept in programming. It allows us to read data from files and write data to files. This is particularly useful when we want to store and retrieve information that persists beyond the lifetime of a program.

Importance of File I/O in Programming File I/O is important because it enables us to:

1. **Read data from files**: We can extract information stored in files, such as text documents, CSV files, JSON files, etc., and use it in our programs. For example, we can read a CSV file containing geospatial data and perform calculations on it.

2. **Write data to files**: We can store data generated by our programs into files for future use or to share with others. For example, we can write the output of a program to a text file or save a graph that you've made.

3. **Manipulate files**: We can create, delete, rename, and modify files using file I/O operations. This allows us to manage files and directories on our computer.

Common Use Cases for Reading Files in Python Reading files is a common task in many programming scenarios. Some common use cases include:

1. **Data analysis**: Reading data from files is essential for performing data analysis tasks.

2. **Configuration files**: Many programs use configuration files to store settings and preferences. Reading these files allows us to customize the behavior of our programs.

3. **Text processing**: Reading text files is often necessary for tasks such as parsing, searching, and manipulating text data.

Overview of Different File Formats Python supports various file formats, the most common ones that you'll work with are:

1. **Text files**: These files contain plain text and are the most common type of file. They can be opened and read using Python's built-in functions.

2. **CSV files**: CSV (Comma-Separated Values) files store tabular data, with each line representing a row and each value separated by a comma. Python provides libraries to read and write CSV files.

3. **JSON files**: JSON (JavaScript Object Notation) files store structured data in a human-readable format. Python has built-in support for reading and writing JSON files.

When we look at GeoPandas, you will also start working with shape files.

0.1.1   Opening and Closing Files

Before we can read or write data to a file, we need to open it. Python provides the `open()` function for this purpose.

Opening Files with `open()`   To open a file, we need to specify its name and the mode in which we want to open it. The mode determines whether we want to read, write, or append to the file, as well as whether we want to work with binary or text data.

Here's the general syntax for opening a file:

```
file = open(filename, mode)
```

- `filename` is the name of the file we want to open, including the file extension.
- `mode` is a string that specifies the mode in which we want to open the file.

Specifying File Modes   Python supports several file modes, including:

- `'r'`: Read mode. Opens the file for reading. This is the default mode if no mode is specified.
- `'w'`: Write mode. Opens the file for writing. If the file already exists, its contents will be overwritten. If the file does not exist, a new file will be created.
- `'a'`: Append mode. Opens the file for appending. If the file already exists, new data will be added to the end of the file. If the file does not exist, a new file will be created.
- `'b'`: Binary mode. Opens the file in binary mode, allowing us to read or write binary data.
- `'t'`: Text mode. Opens the file in text mode, allowing us to read or write text data. This is the default mode if no mode is specified.

We can combine these modes by specifying multiple characters. For example, `'rb'` opens the file in binary read mode.

Closing Files   After we finish working with a file, it's important to close it to free up system resources. We can close a file using the `close()` method.

```
file.close()
```

However, it's easy to forget to close a file, especially if an error occurs. To ensure that a file is always closed, we can use the `with` statement. The `with` statement automatically takes care of closing the file for you.

```
with open(filename, mode) as file:
    # Perform file operations here
```

The `with` statement ensures that the file is closed as soon as we exit the block of code. This is a recommended practice for working with files in Python.

Now that we understand the basics of file handling, let's practice opening and closing files in Python.

0.1.2 Exercises

1. Create a text file named "example.txt" and write the following text to it: "Hello, world!"

```
[ ]:
```

2. Open the file in read mode and print its contents.

```
[ ]:
```

3. Open the file in append mode and add the text "This is an example file." to it.

```
[ ]:
```

4. Open the file in read mode again and print its updated contents.

```
[ ]:
```

0.2 Reading Text Files

0.2.1 Reading Text Files Line by Line

In this section, we will learn how to read text files line by line using file objects in Python. Reading files line by line is a common task when working with text data, as it allows us to process each line individually.

To read a text file line by line, we first need to open the file using the `open()` function.

```python
file = open('file.txt', 'r')
```

Once we have opened the file, we can iterate through its lines using a `for` loop. Each line can be accessed using the file object as an iterator.

```python
file = open('file.txt', 'r')

for line in file:
    # Process each line here
    print(line)

file.close()
```

```
[ ]:
```

0.2.2 Reading the Entire File

In some cases, we may want to read the entire contents of a text file at once. This can be useful when we need to process the file as a single string or when we want to work with the file content as a list of lines.

To read the entire file, we can use the `read()` method of the file object. This method returns the entire content of the file as a string.

```python
file = open('file.txt', 'r')
content = file.read()
print(content)
file.close()
```

Alternatively, we can use the `readlines()` method to read the file content as a list of lines.

```python
file = open('file.txt', 'r')
lines = file.readlines()
print(lines)
file.close()
```

When working with large text files, it is important to consider strategies for efficiently reading them. Reading the entire file at once may not be feasible if the file is too large to fit in memory. In such cases, we can process the file line by line or in smaller chunks using techniques like buffering.

Go on, try it out!

[ ]:

[ ]:

0.2.3 Error Handling and Exception Handling

When reading files, we may encounter potential errors such as file not found, permission denied, or invalid file format. To handle these errors gracefully, we can use error handling and exception handling techniques.

```python
try:
    file = open('file.txt', 'r')
    # Process the file here
    file.close()
except FileNotFoundError:
    print("File not found!")
except PermissionError:
    print("Permission denied!")
except Exception as e:
    print("An error occurred:", str(e))
```

By using specific exception types in the `except` blocks, we can handle different types of errors separately. If we don't know the specific exception type, we can use the generic `Exception` class to catch any type of exception.

4

To summarize, in this section, we learned how to read text files line by line using file objects, how to read the entire file at once, and how to handle potential errors and exceptions when reading files. These skills are essential for working with text data and processing files in Python.

0.3 Reading Structured Data

0.3.1 Reading CSV Files

CSV (Comma-Separated Values) files are a common format for storing structured data. This is how I work with your exam results   They consist of plain text data where each line represents a row and each value within a row is separated by a comma. In this section, we will learn how to read and parse CSV files using the `csv` module in Python.

To begin, we need to import the `csv` module:

```
import csv
```

The `csv` module provides a `reader` object that allows us to read data from a CSV file. We can open a CSV file using the `open()` function and pass the file object to the `reader()` function:

```
with open('data.csv', 'r') as file:
    csv_reader = csv.reader(file)
```

By default, the `csv.reader` object treats the first line of the CSV file as the header. We can access the header using the `next()` function:

```
header = next(csv_reader)
```

To access the data rows, we can iterate over the `csv_reader` object:

```
for row in csv_reader:
    print(row)
```

Each row is returned as a list of values, where each value corresponds to a column in the CSV file.

Sometimes, CSV files may have a different delimiter character instead of a comma. We can specify the delimiter using the `delimiter` parameter when creating the `csv_reader` object:

```
csv_reader = csv.reader(file, delimiter=';')
```

  In the zip file for this lecture, you will find a file `data.csv` you can use that to play around with opening csv files

0.3.2 Reading JSON Files

JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write. It is widely used for storing and transmitting structured data. This is how most computers communicate with each other when they are sending data between them. In this section, we will learn how to read and decode JSON data using the `json` module in Python.

To begin, we need to import the `json` module:

```
import json
```

The `json` module provides a `load()` function that allows us to read JSON data from a file. We can open a JSON file using the `open()` function and pass the file object to the `load()` function:

```python
with open('data.json', 'r') as file:
    json_data = json.load(file)
```

The `json.load()` function decodes the JSON data and returns a Python object (this is not covered in this course). We can then access and process the data as needed.

JSON data can contain nested structures, such as dictionaries within dictionaries or lists within dictionaries. We can access and process nested structures using indexing and iteration:

```python
for item in json_data:
    student_id = entry['student_id']
    address_type = entry['address_type']
    print(f"Student ID: {student_id}, Address Type: {address_type}")
```

In this example, we are accessing the value of the 'name' key within each item in the 'items' list.

In the zip file, you will also find a file called data.json. Please have a play around with loading the file and extracting data from it.

0.3.3 Reading Other Data Formats

Apart from CSV and JSON, there are other common data formats that we may encounter, such as XML and Excel. While Python provides basic support for reading XML files using the `xml` module, it is often more convenient to use third-party libraries for specific data formats.

For XML, we can use libraries like `xml.etree.ElementTree` or `lxml` to parse and process XML data. Similarly, for Excel files, we can use libraries like `pandas` (we'll cover this next week) or `openpyxl` to read and manipulate Excel data.

0.4 File Paths and Directories

In this section, we will learn about file paths and how to work with them in Python. A file path is the location of a file or directory in a file system. There are two types of file paths: absolute and relative.

Absolute File Paths An absolute file path specifies the complete location of a file or directory from the root of the file system. It starts with the root directory, followed by the directories leading to the file or directory. For example, on a Windows system, an absolute file path might look like this: `C:\Users\John\Documents\file.txt`. On a Unix-based system (such as a Mac), it might look like this: `/home/john/documents/file.txt`.

To create an absolute file path in Python, we can use the `os` module. The `os.path.join()` function allows us to join multiple path components together to create a complete file path. Here's an example:

```python
import os

path = os.path.join('C:', 'Users', 'John', 'Documents', 'file.txt')
print(path)
```

Output:

```
C:\Users\John\Documents\file.txt
```

Relative File Paths A relative file path specifies the location of a file or directory relative to the current working directory. It does not start with the root directory. Instead, it starts with a directory or file name and navigates from there. For example, if the current working directory is `/home/john`, a relative file path to a file in the `documents` directory might look like this: `documents/file.txt`.

To create a relative file path in Python, we can simply specify the path as a string. Here's an example:

```python
path = 'documents/file.txt'
print(path)
```

Output:

```
documents/file.txt
```

Navigating Directories and Locating Files The `os` module provides several functions for navigating directories and locating files. Here are a few commonly used functions:

- `os.getcwd()`: Returns the current working directory as a string.
- `os.chdir(path)`: Changes the current working directory to the specified path.
- `os.listdir(path)`: Returns a list of all files and directories in the specified path.
- `os.path.exists(path)`: Returns `True` if the specified path exists, otherwise `False`.
- `os.path.isfile(path)`: Returns `True` if the specified path is a file, otherwise `False`.
- `os.path.isdir(path)`: Returns `True` if the specified path is a directory, otherwise `False`.

Here's an example that demonstrates how to use these functions:

```python
import os

# Get the current working directory
current_dir = os.getcwd()
print("Current directory:", current_dir)

# Change the current working directory
os.chdir('documents')
print("Changed directory:", os.getcwd())

# List all files and directories in the current directory
files = os.listdir()
print("Files in current directory:", files)

# Check if a file exists
file_path = 'file.txt'
if os.path.exists(file_path):
    print(file_path, "exists")
else:
    print(file_path, "does not exist")
```

```python
# Check if a path is a file or directory
if os.path.isfile(file_path):
    print(file_path, "is a file")
elif os.path.isdir(file_path):
    print(file_path, "is a directory")
else:
    print(file_path, "is neither a file nor a directory")
```

Output:

```
Current directory: /home/john
Changed directory: /home/john/documents
Files in current directory: ['file.txt', 'folder']
file.txt exists
file.txt is a file
```

[ ]:

[ ]:

0.5 Best Practices and Considerations

0.5.1 Best Practices for Reading Files

When working with files, it is important to follow some best practices to ensure efficient and safe file handling. Here are some key points to consider:

1. **Opening Files**: Always use the `with` statement when opening files. This ensures that the file is properly closed after you are done with it, even if an exception occurs. For example:

   ```python
   with open('file.txt', 'r') as file:
       # Perform operations on the file
   ```

2. **Reading Files**: Use the appropriate mode when opening files for reading. The most common modes are `'r'` for reading in text mode and `'rb'` for reading in binary mode. For example:

   ```python
   with open('file.txt', 'r') as file:
       content = file.read()
   ```

   You can also read the file line by line using the `readline()` method or iterate over the file object directly.

3. **Closing Files**: Although the `with` statement takes care of closing the file for you, it is good practice to explicitly close the file after you are done with it, especially if you are working with a large number of files. This can be done using the `close()` method. For example:

   ```python
   file = open('file.txt', 'r')
   # Perform operations on the file
   file.close()
   ```

4. **Efficiently Processing Large Files**: When working with large files, it is important to process them efficiently to avoid memory issues. Instead of reading the entire file into memory

at once, consider reading it line by line or in chunks. This can be done using a loop or the `readlines()` method. For example:

```python
with open('large_file.txt', 'r') as file:
    for line in file:
        # Process each line
```

Alternatively, you can use libraries like `pandas` or `numpy` to handle large datasets more efficiently. We will look at both of these in the next section of the course.

0.6 Conclusion

In this lecture, you've acquired the essential skills for reading files in Python. You've learned the basics of file I/O, how to read text files line by line, read structured data from formats like CSV and JSON, and manage file paths and directories. These skills are fundamental for various data processing, analysis, and manipulation tasks in Python.

Well done for making it this far   Next we will look at three libraries, Numpy, Pandas, and GeoPandas in much more detail, so buckle up.

`[ ]:`

09Numpy

November 29, 2023

Dr Gunnar Mallon (g.mallon@rug.nl), *Department of Cultural Geography (Faculty of Spatial Science), University of Groningen*

NumPy is a fundamental Python library for scientific computing, which is crucial for geospatial data analysis. We will explore what NumPy is and why it is important for working with geospatial data.

It stands for Numerical Python and is a powerful library in Python for performing numerical computations and manipulating large, multi-dimensional arrays and matrices. It provides high-performance mathematical functions and tools for efficiently working with numerical data.

0.0.1 Why is NumPy important for geospatial data analysis?

NumPy is particularly important for geospatial data analysis due to its ability to handle and process large, multi-dimensional arrays efficiently. Geospatial data often consists of multiple dimensions such as latitude, longitude, elevation, and time. NumPy provides a convenient and efficient way to store, manipulate, and analyze these multi-dimensional arrays.

Some key reasons why NumPy is important for geospatial data analysis are:

1. **Efficient storage and processing**: NumPy arrays occupy less memory than regular Python lists and allow for faster processing of large datasets.

2. **Vectorized operations**: NumPy allows you to perform mathematical operations on entire arrays instead of individual elements, significantly improving the performance of computations.

3. **Integration with other libraries**: NumPy is the foundation for many other scientific libraries in Python, such as Pandas, Matplotlib, and SciPy. Understanding NumPy is essential for effectively using these libraries in geospatial data analysis.

0.0.2 Installation and setup

Before we can start working with NumPy, we need to install it and set it up in our Python environment. Here are the steps to install NumPy:

1. **Install NumPy via conda**: Open your command prompt or terminal and run the following command:

```
conda install numpy
```

2. **Verify the installation**: Once the installation is complete, open a Jupyter Notebook and import the NumPy module using the following command:

```
import numpy as np
```

If no errors occur, the installation was successful.

Now that we have NumPy installed and set up, we can move on to learning about its core features and how it can be used in geospatial data analysis.

0.1 NumPy Arrays

To work with geospatial data, we often need to store large amounts of numeric data efficiently. NumPy arrays provide an efficient way to store and manipulate numerical data.

np.array()   The most basic way to create a NumPy array is by using the `np.array()` function. It takes a Python list or nested lists as input and returns a NumPy array.

```
import numpy as np

# Creating a 1D array
array1d = np.array( [1, 2, 3, 4, 5] )
print(array1d)
# Output: [1 2 3 4 5]

# Creating a 2D array
array2d = np.array( [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ] )
print(array2d)
# Output:
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]
```

[ ]:

np.zeros()   The `np.zeros()` function creates an array filled with zeros. It takes the shape of the desired array as a parameter.

```
zeros_arr = np.zeros((3, 3))
print(zeros_arr)
# Output:
# [[0. 0. 0.]
#  [0. 0. 0.]
#  [0. 0. 0.]]
```

[ ]:

np.ones()   Similarly, the `np.ones()` function creates an array filled with ones.

```
ones_arr = np.ones((2, 4))
print(ones_arr)
```

```
# Output:
# [[1. 1. 1. 1.]
#  [1. 1. 1. 1.]]
```

[ ]:

np.linspace()   The `np.linspace()` function creates an array of evenly spaced values over a specified range. It takes the start, end, and number of points as parameters.

```
lin_arr = np.linspace(0, 10, 5)
print(lin_arr)
# Output: [ 0.   2.5  5.   7.5 10. ]
```

[ ]:

np.arange()   The `np.arange()` function creates an array with regularly spaced values between a start and end value, with a specified step size.

```
range_arr = np.arange(0, 10, 2)
print(range_arr)
# Output: [0 2 4 6 8]
```

[ ]:

0.1.1   Accessing and Modifying Elements

Once we have created NumPy arrays, we can access and modify their elements using various techniques.

Indexing and Slicing   NumPy arrays support multi-dimensional indexing and slicing. We can access individual elements by providing their indices or ranges of indices.

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

print(arr[0, 1])
# Output: 2

print(arr[:2, 1:])
# Output:
# [[2 3]
#  [5 6]]
```

If you are having trouble figuring this one out, please just ask!

[ ]:

Broadcasting   Broadcasting allows us to perform mathematical operations on arrays with different shapes and dimensions. When operating on arrays with different shapes, NumPy automatically broadcasts the smaller array to match the shape of the larger array.

3

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

print(arr + 2)
# Output: [3 4 5 6 7]
```

[ ]:

0.1.2 Boolean Indexing

We can use Boolean indexing to filter arrays based on specific conditions. We create a Boolean array by applying a condition to an existing array, and then use that Boolean array to index the original array.

```
arr = np.array([10, 15, 20, 25, 30])

condition = (arr > 20)
filtered_arr = arr[condition]

print(filtered_arr)
# Output: [25 30]
```

[ ]:

0.2 Exercise

1. Create a NumPy array of integers from 1 to 20. Use boolean indexing to create a new array containing only the even numbers from the original array.

[ ]:

0.2.1 Array Shapes and Dimensions

NumPy provides useful functions to determine the shape and dimensions of arrays.

Shape Attribute The `shape` attribute returns the dimensions of an array.

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

print(arr.shape)
# Output: (2, 3)
```

[ ]:

Reshaping Arrays We can reshape arrays using the `reshape()` function. This function returns a new array with a modified shape, without changing the original array.

```

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

reshaped_arr = arr.reshape(2, 3)
print(reshaped_arr)
# Output:
# [[1 2 3]
# [4 5 6]]
```

[ ]:

Transposing Arrays   The `transpose()` function swaps the dimensions of an array.

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

transposed_arr = arr.transpose()
print(transposed_arr)
# Output:
# [[1 4]
#  [2 5]
#  [3 6]]
```

[ ]:

0.3   Exercises

1. Create a 1D NumPy array containing the numbers from 0 to 9.

[ ]:

2. Create a 2D NumPy array of shape (3, 3) filled with random integers between 0 and 100.

[ ]:

3. Access the element at index (1, 2) of the above array.

[ ]:

4. Reshape the array from exercise 2 to have shape (9, 1).

[ ]:

5. Perform element-wise multiplication between two NumPy arrays of shapes (2, 3) and (3, 2).

[ ]:

0.4 Basic Array Operations

Numpy is a powerful Python library that provides support for large, multi-dimensional arrays and matrices of numerical data, as well as a large collection of mathematical functions to operate on these arrays. In this section, we will explore some basic operations that can be performed on numpy arrays, with a focus on geospatial data.

0.4.1 Element-wise operations

Element-wise operations are operations that are **performed on each element of an array individually**. Some common element-wise operations in numpy include arithmetic operations, mathematical functions, and statistical functions.

Arithmetic operations   Numpy supports all basic arithmetic operations such as addition, subtraction, multiplication, and division. These operations can be performed between two arrays or between an array and a scalar value.

Example:

```python
import numpy as np

# create two arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# perform arithmetic operations
c = a + b  # element-wise addition
d = a - b  # element-wise subtraction
e = a * b  # element-wise multiplication
f = a / b  # element-wise division

print(c)  # [5 7 9]
print(d)  # [-3 -3 -3]
print(e)  # [4 10 18]
print(f)  # [0.25 0.4  0.5]
```

[ ]:

Mathematical functions   Some common mathematical functions include `sqrt`, `exp`, `log`, `sin`, `cos`, and `tan`.

Example:

```python
import numpy as np

# create an array
a = np.array([1, 2, 3])

# apply mathematical functions
b = np.sqrt(a)   # element-wise square root
```

```
    c = np.exp(a)   # element-wise exponential
    d = np.log(a)   # element-wise natural logarithm
    e = np.sin(a)   # element-wise sine function

    print(b)   # [1.         1.41421356 1.73205081]
    print(c)   # [ 2.71828183  7.3890561   20.08553692]
    print(d)   # [0.         0.69314718 1.09861229]
    print(e)   # [0.84147098 0.90929743 0.14112001]
```

[ ]:

Array operations with scalars   Numpy allows for easy element-wise operations between arrays and scalar values. When an arithmetic operation is performed between an array and a scalar, the scalar value is broadcasted to match the shape of the array, and the operation is then performed element-wise.

Example:

```
import numpy as np

# create an array
a = np.array([1, 2, 3])

# perform array operations with a scalar value
b = a + 5   # element-wise addition
c = a * 2   # element-wise multiplication
d = a / 3   # element-wise division

print(b)   # [6 7 8]
print(c)   # [2 4 6]
print(d)   # [0.33333333 0.66666667 1.]
```

[ ]:

Statistical functions   Numpy also provides a variety of statistical functions that can be applied element-wise to numpy arrays. Some common statistical functions include sum, mean, max, min, std, and var.

Example:

```
import numpy as np

# create an array
a = np.array([[1, 2, 3], [4, 5, 6]])

# apply statistical functions
b = np.sum(a)    # sum of all elements
c = np.mean(a)   # mean of all elements
d = np.max(a)    # maximum value
```

```
e = np.min(a)    # minimum value
f = np.std(a)    # standard deviation
g = np.var(a)    # variance

print(b)    # 21
print(c)    # 3.5
print(d)    # 6
print(e)    # 1
print(f)    # 1.707825127659933
print(g)    # 2.9166666666666665
```

[ ]:

0.5 Exercises:

1. Import the NumPy library and create a 2D array of shape (3, 4) with random values.

[ ]:

2. Calculate the mean, median, and standard deviation of the array created in exercise 1.

[ ]:

3. Create a 1D array of shape (10,) with values ranging from 0 to 9 and reshape it to (5, 2).

[ ]:

4. Multiply each element of the array created in exercise 3 by 2.

[ ]:

5. Create a new 2D array of shape (3, 3) and perform element-wise multiplication with the array created in exercise 3.

[ ]:

6. Create a random array of shape (100, 2) using NumPy's random module and calculate the minimum and maximum values for each column.

[ ]:

0.6 Conclusion and Next Steps

0.6.1 Summary of NumPy's importance in geospatial data analysis:

- NumPy is a fundamental library for numerical computing in Python
- It provides efficient operations for array manipulation and mathematical computations
- NumPy's array object is the foundation for many other libraries used in geospatial data analysis, such as Pandas and Matplotlib
- It allows for faster processing and analysis of large geospatial datasets

- NumPy's broadcasting feature simplifies computations on arrays of different sizes

0.6.2 Additional resources for further learning:

- NumPy official documentation: https://numpy.org/doc/
- NumPy tutorial on GeeksforGeeks: https://www.geeksforgeeks.org/python-numpy-module-tutorial/
- NumPy tutorial on DataCamp: https://www.datacamp.com/community/tutorials/python-numpy-tutorial
- "Python for Data Science For Dummies" by John Paul Mueller and Luca Massaron

10Pandas

November 29, 2023

Dr Gunnar Mallon (g.mallon@rug.nl), *Department of Cultural Geography (Faculty of Spatial Science), University of Groningen*

---

1 Pandas - Data Analysis Library

1.1 Introduction to Pandas

Pandas is a Python library that is widely used for data manipulation and analysis. It provides tools for structuring and manipulating datasets, making it easy to clean, transform, and analyze data. We are going to use Pandas throughout both tutorials this week.

1.1.1 Key features of Pandas

1. Data structures:
   - Series: A one-dimensional labeled array that can hold any data type.
   - DataFrame: A two-dimensional table with labeled axes (rows and columns).
   - Panel: A three-dimensional array with labeled axes.
2. Data manipulation and cleaning:
   - Data indexing and selection: Allows accessing and manipulating data based on various criteria, such as labels, indices, or conditional statements.
   - Handling missing data: Provides functions for identifying, handling, and imputing missing data.
   - Filtering and sorting: Supports filtering and sorting data based on specific conditions or by column/row values.
   - Data aggregation: Enables grouping, summarizing, and aggregating data using functions like `groupby()`, `pivot_table()`, and `agg()`.
3. Data input and output:
   - Reading and writing CSV, Excel, SQL, and other file formats.
   - Working with HTML, JSON, and XML data.
   - Accessing and manipulating database tables through SQL queries.
4. Time series functionality:
   - Date and time handling: Supports parsing, indexing, and manipulating date and time data.
   - Time series indexing: Allows indexing data based on time-based criteria.
   - Resampling and frequency conversion: Provides functions for resampling and converting time series data to different frequencies.
5. Plotting and visualization:

- Integration with Matplotlib for creating various types of plots, including line plots, scatter plots, histograms, bar plots, etc.
- Tools for customizing plot appearance, labels, titles, and legends.

Let's explore some examples and exercises to familiarize ourselves with Pandas and its geospatial data handling capabilities.

1.2 Pandas Data Structures

Pandas provides two main data structures: Series and DataFrame. Let's look at them in turn.

1.2.1 Series: One-dimensional labeled array

A Series is a one-dimensional labeled array that can hold any data type. It is similar to a column in a table or a spreadsheet. Each element in a Series has a unique label called an index.

Creating a Series To create a Series, you can pass a list or an array of data along with an optional index. Let's see some examples:

```python
import pandas as pd

# Creating a Series from a list
s1 = pd.Series([3, 6, 9, 12])
print(s1)

# Creating a Series from an array with custom index
s2 = pd.Series([3, 6, 9, 12], index=['a', 'b', 'c', 'd'])
print(s2)

# Creating a Series from a dictionary
s3 = pd.Series({'a': 3, 'b': 6, 'c': 9, 'd': 12})
print(s3)
```

Have a play with these until you are comfortable with the syntax

```
[ ]:
```

Accessing elements in a Series You can access elements in a Series using their index:

```python
# Accessing by index label
print(s2.loc['a'])

# Accessing by index position
print(s2.iloc[0])
```

```
[ ]:
```

Indexing and Slicing Pandas provides powerful indexing and slicing capabilities for Series. You can select specific elements or ranges of elements based on their index labels or positions. Let's see some examples:

```python
# Selecting a single element
print(s2['b'])

# Selecting multiple elements by index label
print(s2[['a', 'c']])

# Selecting a range of elements by index label
print(s2['b':'d'])

# Selecting multiple elements by index position
print(s2[[1, 3]])

# Selecting a range of elements by index position
print(s2[1:3])
```

What do these instructions output?

`[ ]:`

Important methods and attributes of Series Pandas provides many useful methods and attributes for working with Series. Here are some commonly used ones:

- `head()`: Returns the first n rows of the Series.
- `tail()`: Returns the last n rows of the Series.
- `describe()`: Generates descriptive statistics of the Series.
- `mean()`: Calculates the mean of the Series.
- `max()`: Returns the maximum value in the Series.
- `min()`: Returns the minimum value in the Series.
- `count()`: Counts the number of non-null elements in the Series.
- `unique()`: Returns an array of unique values in the Series.
- `value_counts()`: Returns a Series containing counts of unique values in the Series.

1.2.2 DataFrame: Two-dimensional labeled data structure

A DataFrame is a two-dimensional labeled data structure with columns of potentially different types. It is similar to a table or a spreadsheet, where each column represents a variable and each row represents an observation.

Creating a DataFrame from a dictionary To create a DataFrame, you can pass a dictionary of lists or arrays as input. Each key in the dictionary represents the column name, and the corresponding list or array represents the values in that column. Let's have a look at an example:

```python
import pandas as pd

# Creating a DataFrame from a dictionary
data = {
    'name': ['John', 'Jane', 'Tom'],
    'age': [25, 30, 35],
    'city': ['New York', 'London', 'Paris']
```

```
    }

    df = pd.DataFrame(data)
    print(df)
```

[ ]:

Accessing columns and rows in a DataFrame   You can access columns in a DataFrame using their names. Additionally, you can access rows using their index labels or positions. Let's see some examples:

```
# Accessing a column by name
print(df['name'])
print(df.name)

# Accessing a row by index label
print(df.loc[0])

# Accessing a row by index position
print(df.iloc[0])
```

Go on, try it out and play around with it...

[ ]:

Indexing and Slicing in a DataFrame   Pandas allows for indexing and slicing in a DataFrame to select specific rows and columns. You can use various methods to accomplish this. Let's see some examples:

```
# Selecting a single column
print(df['name'])

# Selecting multiple columns
print(df[['name', 'age']])

# Selecting a single row by index label
print(df.loc[0])

# Selecting a single row by index position
print(df.iloc[0])

# Selecting multiple rows by index label
print(df.loc[[0, 2]])

# Selecting multiple rows by index position
print(df.iloc[[0, 2]])

# Selecting rows and columns simultaneously
print(df.loc[[0, 2], ['name', 'age']])
```

You guessed it, have a play

`[ ]:` _____

Applying functions to DataFrames You can apply various functions to DataFrames to perform transformations or calculations. Pandas provides convenient ways to apply functions columnwise or row-wise. Let's see some examples:

```python
# Applying a function to a column
df['age_next_year'] = df['age'].apply(lambda x: x + 1)
print(df)

# Applying a function to a row
df['summary'] = df.apply(lambda x: f"{x['name']} is {x['age']} years old and lives in {x['city
print(df)
```

Output:

```
    name  age      city  age_next_year
0   John   25  New York             26
1   Jane   30    London             31
2    Tom   35     Paris             36

    name  age      city  age_next_year                              summary
0   John   25  New York             26  John is 25 years old and lives in New York
1   Jane   30    London             31    Jane is 30 years old and lives in London
2    Tom   35     Paris             36       Tom is 35 years old and lives in Paris
```

The second example highlights an important point. You can create new columns in a DataFrame by simply intializing them. In this case, we created a new column called "summary" but assigning values to it.

`[ ]:` _____

Important methods and attributes of DataFrame Pandas provides many useful methods and attributes for working with DataFrames. Here are some commonly used ones:

- `head(n)`: Returns the first n rows of the DataFrame.
- `tail(n)`: Returns the last n rows of the DataFrame.
- `describe()`: Generates descriptive statistics of the DataFrame.
- `info()`: Provides a concise summary of the DataFrame, including the column names, data types, and non-null counts.
- `shape`: Returns a tuple representing the dimensions of the DataFrame (number of rows, number of columns).
- `columns`: Returns a list of the column names in the DataFrame.
- `index`: Returns the index labels of the DataFrame.
- `dropna()`: Removes rows with missing values.
- `fillna(value)`: Replaces missing values with a specified value.
- `sort_values(by)`: Sorts the DataFrame by the values in a specific column.
- `groupby(by)`: Groups the DataFrame by one or more columns.

- `pivot_table(values, index, columns)`: Creates a pivot table from the DataFrame.

You are going to use most of these methods, especially the `head`, `tail`, `sort_values`, and `groupby` methods.

By understanding and mastering these data structures and the utilities provided by the Pandas library, you will be equipped with the necessary tools to work efficiently with geospatial data in Python.

2 Loading and Saving Data with Pandas

CSV (Comma-Separated Values) files are a popular format for storing tabular data. Pandas provides a convenient method to load CSV files into DataFrames using the `read_csv()` function.

```python
import pandas as pd

# Load CSV file into DataFrame
data = pd.read_csv("data.csv")

# Print the first few rows of the DataFrame
print(data.head())
```

In the example above, we import the `pandas` library and use the `read_csv()` function to load a CSV file named "data.csv" into a DataFrame called `data`. We then print the first few rows of the DataFrame using `head()` method.

2.0.1 Exercise

1. Download a CSV file containing geospatial data - remember the data.csv from a previous lesson!
2. Load the CSV file into a DataFrame using Pandas.
3. Display the first 5 rows of the DataFrame.

```
[ ]:
```

2.1 Saving DataFrames to CSV files

Just as easily as you can read files into a DataFrame, you can also save a DataFrame to a CSV file with the `to_csv()` function.

```python
import pandas as pd

# Save DataFrame to CSV file
data.to_csv("output.csv", index=False)
```

In the example above, we use the `to_csv()` function to save the DataFrame named `data` to a CSV file named "output.csv". The `index=False` parameter is used to exclude the index column from the saved file.

2.1.1 Exercise

1. Save the DataFrame from the previous exercise to a CSV file named "output.csv".
2. Verify that the file was saved successfully.

```
[ ]:
```

Congratulations You have learned how to load and save geospatial data using Pandas. Using this knowledge, you can now work with various file formats to analyze and manipulate geospatial datasets.

2.2 Data Cleaning and Manipulation with Pandas

In this section, we will learn how to clean and manipulate geospatial data using Pandas. We will cover various techniques for handling missing data, filtering and selecting data, sorting and ranking data, combining and merging DataFrames, and grouping and aggregating data.

2.2.1 1. Handling Missing Data

Missing data is a common issue in datasets, and it's important to handle them appropriately to avoid biased or incorrect results. Pandas provides several methods to handle missing data.

1.1 Identifying Missing Data Before handling missing data, it's important to identify where the missing data exists in the dataset. Pandas provides the `isnull()` and `notnull()` functions to check for missing values.

```python
import pandas as pd

# Create a DataFrame with missing values
df = pd.DataFrame({'A': [1, 2, None, 4, 5],
                   'B': [None, 2, 3, None, 5],
                   'C': [1, 2, 3, 4, None]})

# Check for missing values
print(df.isnull())
print(df.notnull())
```

```
[ ]:
```

1.2 Dropping Missing Values One way to handle missing values is to remove rows or columns that contain missing data. Pandas provides the `dropna()` function to drop missing values.

```python
import pandas as pd

# Create a DataFrame with missing values
df = pd.DataFrame({'A': [1, 2, None, 4, 5],
                   'B': [None, 2, 3, None, 5],
```

```
                      'C': [1, 2, 3, 4, None]})

# Drop rows with any missing values
print(df.dropna())

# Drop columns with any missing values
print(df.dropna(axis=1))
```

Go on, try it out yourself. Try to figure out and look up what the axis=1 means!

[ ]:

2.2.2 2. Filtering and Selecting Data

To analyze geospatial data, we often need to filter and select specific subsets of data. Pandas provides various methods for this purpose.

2.1 Boolean Indexing Boolean indexing allows us to filter rows based on a condition. We can use comparison or logical operators to create a Boolean mask. This is a really important concept when working with Pandas, so let's break it down a little.

Imagine that we have a DataFrame

```
# Initializing a new DataFrame
my_dataFrame = pd.DataFrame([1, 2, 3, 4, 5, 6])
```

Now we can create a boolean mask by check for a condition on each element that returns a boolean value:

```
#Check if each number in our DataFrame is even
my_mask = my_DataFrame[0] % 2 == 0
```

Here we use `my_list[0]` to get the first column in the DataFrame. The mask will have the following values `[False, True, False, True, False, True]`. Go on, try it.

So, now that we have a list of True and False values each indicating if the number at the same position is either even or odd, we can apply it to the DataFrame and only get even numbers.

```
# Apply the mask to the DataFrame
print(my_dataFrame[my_mask])
```

Basically, Pandas, iterates through my_dataFrame and only shows rows for which the corresponding value for my_mask is True.

[ ]:

Let's look at a little more complex example:

```
import pandas as pd

# Create a DataFrame of geospatial data
df = pd.DataFrame({'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix'],
                   'Country': ['USA', 'USA', 'USA', 'USA', 'USA'],
```

```
                    'Latitude': [40.7128, 34.0522, 41.8781, 29.7604, 33.4484],
                    'Longitude': [-74.0060, -118.2437, -87.6298, -95.3698, -112.0740]})

# Filter rows based on latitude greater than 35 degrees
print(df[df['Latitude'] > 35])

# Filter rows based on latitude greater than 35 degrees and country is USA
print(df[(df['Latitude'] > 35) & (df['Country'] == 'USA')])
```

Before you copy the code, try to predict what the output is going to be. Did it match?

[ ]:

[ ]:

2.2 Selecting Rows and Columns To select specific rows and columns from a DataFrame, we can use the `loc` and `iloc` attributes.

```
import pandas as pd

# Create a DataFrame of geospatial data
df = pd.DataFrame({'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix'],
                    'Country': ['USA', 'USA', 'USA', 'USA', 'USA'],
                    'Latitude': [40.7128, 34.0522, 41.8781, 29.7604, 33.4484],
                    'Longitude': [-74.0060, -118.2437, -87.6298, -95.3698, -112.0740]})

# Select specific rows and columns by label using loc
print(df.loc[2:4, 'City':'Latitude'])

# Select specific rows and columns by position using iloc
print(df.iloc[2:4, 2:])
```

Just like the previous example try to figure out what the output will be, before copying the code

[ ]:

2.2.3 3. Sorting and Ranking Data

Sorting and ranking data allow us to order the rows based on specific columns or criteria.

```
import pandas as pd

# Create a DataFrame of geospatial data
df = pd.DataFrame({'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix'],
                    'Country': ['USA', 'USA', 'USA', 'USA', 'USA'],
                    'Latitude': [40.7128, 34.0522, 41.8781, 29.7604, 33.4484],
                    'Longitude': [-74.0060, -118.2437, -87.6298, -95.3698, -112.0740]})

# Sort DataFrame by latitude in ascending order
print(df.sort_values(by='Latitude'))
```

```python
# Sort DataFrame by latitude in descending order
print(df.sort_values(by='Latitude', ascending=False))

# Rank DataFrame by latitude
print(df['Latitude'].rank())
```

[ ]:

2.2.4 4. Combining and Merging DataFrames

Often, we need to combine or merge multiple DataFrames to analyze geospatial data.

```python
import pandas as pd

# Create two DataFrames of geospatial data
df1 = pd.DataFrame({'City': ['New York', 'Los Angeles', 'Chicago'],
                    'Population': [8000000, 4000000, 3000000]})
df2 = pd.DataFrame({'City': ['Houston', 'Phoenix', 'Dallas'],
                    'Population': [2500000, 1500000, 2000000]})

# Concatenate DataFrames vertically
print(pd.concat([df1, df2]))

# Concatenate DataFrames horizontally
print(pd.concat([df1, df2], axis=1))

# Merge DataFrames based on a common column
print(pd.merge(df1, df2, on='City'))
```

pd.merge is a very powerful method that has a lot of optional parameters. In case that you have two DataFrames where the common data is in columns with different names, you can use left_on=[name of column in one DataFrame] and right_on=[name of column in second DataFrame] to merge the DataFrames

[ ]:

2.2.5 5. Grouping and Aggregating Data

Grouping and aggregating data allows us to summarize and extract insights from geospatial data.

```python
import pandas as pd

# Create a DataFrame of geospatial data
df = pd.DataFrame({'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix'],
                   'Country': ['USA', 'USA', 'USA', 'USA', 'USA'],
                   'Latitude': [40.7128, 34.0522, 41.8781, 29.7604, 33.4484],
                   'Longitude': [-74.0060, -118.2437, -87.6298, -95.3698, -112.0740]})

# Group DataFrame by country and calculate the mean latitude
```

```python
print(df.groupby('Country')['Latitude'].mean())

# Group DataFrame by country and count the number of cities
print(df.groupby('Country')['City'].count())
```

This is the last example. Really try to figure out what the output will be before you execute the code!!

```
[ ]:
```

2.3   Conclusion

In this lecture, we covered the basics of Pandas, a powerful Python library for data analysis and manipulation.

2.3.1   Resources for further learning

Pandas is a vast library with numerous functionalities. To further enhance your knowledge and expertise in working with geospatial data in Pandas, I recommend exploring the following resources:

- Pandas documentation: The official documentation of Pandas is a comprehensive resource that provides detailed information about each function and capability of the library. You can access it at pandas.pydata.org.

- Geopandas: Geopandas is a library built on top of Pandas that extends its functionality specifically for geospatial analysis. It provides an easy-to-use interface for working with geospatial data and integrates well with other geospatial libraries such as Fiona and Shapely. We will look at this next!

11GeoPandas

November 29, 2023

Dr Gunnar Mallon (g.mallon@rug.nl), *Department of Cultural Geography (Faculty of Spatial Science), University of Groningen*

---

1 Geospatial Data Analysis with GeoPandas

We have come to the end of the short introduction to Python for Geospatial Data Science. Well done for making it this far!! Hopefully, you are all excited about your newfound programming skills.

In this final section (we left the best for last), we are going to pull together all of what we have covered so far, and apply our knowledge to produce a map visualizing population changes by U.S. state between 2021 and 2022.

This file is largely a walk through on how to create cool maps with Python. At the end you will be set a big exercise that will test all your newly learned Python knowledge!

1.1 Setting Up Our Environment

First, ensure that you have downloaded the necessary datasets containing the population changes and the shapefiles delineating state boundaries. You will find these on Brightspace.

Make sure your data is organized in a logical folder structure to facilitate easy access. The project structure looks like this: - Jupyter notebook file (this file). - A folder named `population_estimates` with the CSV file of population data. - A folder named `shapefiles` with the respective geographic shapefiles.

Now, let's install the libraries necessary for our geospatial analysis if you haven't done so.

```
!pip install geopandas matplotlib
```

[ ]:

Next, we'll import the libraries required for our analysis:

```
import geopandas as gpd
import pandas as pd
import matplotlib.pyplot as plt
```

You will notice two new packages. Matplotlib is a library for plotting diagrams in Python and I'm sure you can guess what Geopandas is.

```
[ ]:
```

1.2 Loading and Preparing the Data

Let us start by setting up the paths to our CSV data and shapefile.

```
data_path = 'population_estimates/NST-EST2022-POPCHG2020_2022.csv'
shape_path = 'shapefiles/cv_2018_us_state_500k.shp'
```

Load the population data into a Pandas DataFrame and examine its contents:

```
# Read the csv file into the DataFrame
df = pd.read_csv(data_path)
```

```
[2]:
```

To have a look at the first few rows of a DataFrame you can use the `DataFrame.head()` method. This is an essential method to have a quick look at a DataFrame. I suggest that after each change to the DataFrame, you have a quick look at it using `head()`.

```
# Let's check the first few rows of the DataFrame
df.head()
```

```
[ ]:
```

After inspecting the DataFrame, we find that we can drop the column we do not need and rename the others for clarity:

```
# DataFrame.drop let's us remove a column from the DataFrame. In this case we're dropping ESTI
df.drop(columns=['ESTIMATESBASE2020'], inplace=True)

# You can rename the columns in your DataFrame but simply setting the DataFrame.columns to new
df.columns = ['State', 'Population_2021', 'Population_2022']
```

```
[4]:
```

Make sure to have a look at the new DataFrame

```
df.head()
```

```
[ ]:
```

The populations are read as strings and if we want to work with them we'll need to convert these to integers by using the `astype(int)` method:

```
df['Population_2021'] = df['Population_2021'].astype(int)
df['Population_2022'] = df['Population_2022'].astype(int)
```

```
[6]:
```

Next, we'll calculate the population change percentage from 2021 to 2022 and add it as a new column. This will be the data that we are going to plot on the map.

2

```
df['Population_Change'] = ((df['Population_2022'] - df['Population_2021']) / df['Population_20
```

[7]:

Remember to look at your new DataFrame using `df.head()`

[ ]:

1.3 Integrating the Geographic Data

With GeoPandas, we can read in our shapefile directly and explore the data:

```
shape_data = gpd.read_file(shape_path)
shape_data.head()
```

[ ]:

Once we have both the population DataFrame and the geospatial DataFrame, we can merge them. Remember, you looked at this in the Pandas notebook.

When you look at the `head()` of both DataFrames, you'll see that the U.S. States are listed under different column names. We could rename the columns, or use the `left_on` and `right_on` parameters.

```
shape_data = shape_data.merge(df, left_on='NAME', right_on='State', how='left')
```

Here we merge the data based on the **NAME** column in the shape_data DataFrame and the **State** column in the df DataFrame. Have a look what the `how='left'` means.

[ ]:

We will then drop any entries with missing geospatial data using the `dropna` method:

```
shape_data.dropna(subset=['Population_Change'], inplace=True)
```

[12]:

1.4 Creating the Map

Ok, it's time for the fun part. Let's create a cool map of the data! We will focus exclusively on the continental United States. To do that we need to exclude some rows of the island and remote states (Alaska, Hawaii, and Puerto Rico).

```
# Exclude territories outside of the continental US, if necessary
exclude_territories = ['Alaska', 'Hawaii', 'Puerto Rico'] # Add any other territories if neede
shape_data = shape_data[~shape_data['State'].isin(exclude_territories)]
```

Let's break this down a little. After creating the list of territories to exclude, we have to remove them.

First we create a boolean mask of all rows where the **State** is included in the exclude_territories list. This will return `True` for the three territories, Alaska, Hawaii, and Puerto Rico.

However, we want to exclude these. So we use the tilde ('~') to signify a **NOT** condition. Essentially this means that we get a `True` value for our boolean mask, when the row's **State** value is **NOT** in the list of exclusion territories.

Finally, we use the boolean mask to get the new shape_data.

[ ]: ┌───┐
     └───┘

The next part is from the MatplotLib library for creating plots. Don't worry if it looks a little intimidating. We'll break it into small bits.

We start by creating a new plot and we call it ax and then show it.

```python
# Create the plot and store it in 'ax'
ax = shape_data.plot(column='Population_Change')

# Use Matplotlib to show the plot
plt.show()
```

[ ]: ┌───┐
     └───┘

Pretty cool, wouldn't you say? I think so!

You can see that there are differences between the states representing changes in population, but we don't know what these mean. We need a legend! We can achieve this by adding the `legend=True` parameter to the method that creates the plot

```python
# Create the plot and store it in 'ax'
ax = shape_data.plot(
    column='Population_Change',
    legend=True
)

# Use Matplotlib to show the plot
plt.show()
```

[ ]: ┌───┐
     └───┘

So far so good. Now we know what the individual colors mean. But we can still do one better. We can turn off the axes because we are not really interested in longitude and latitude but in population change.

We can achieve this by calling `ax.set_axis_off()` before showing the plot.

```python
# Create the plot and store it in 'ax'
ax = shape_data.plot(
    column='Population_Change',
    legend=True
)

# Use Matplotlib to show the plot
ax.set_axis_off()
plt.show()
```

[ ]:

Much better! But we can do better still. Let's do the following changes:

- **Change the figure size**: We can use `figsize` for this. Let's make it 15:9
- **Reformat the legend**: The legend is a little distracting, let's put it below the map using the `legend_kwds` parameter
- **Change the colour scheme**: The colours look pretty cool, but I prefer them to go from Red to Blue rather than Yellow to Blue. We do this with the `cmap` parameter

If we implement all those changes we get:

```python
# Create the plot and store it in 'ax'
ax = shape_data.plot(
    column='Population_Change',
    legend=True,
    legend_kwds={
        'label': "Population Change (%)",
        'orientation': "horizontal",
        'format': "%0.2f"
    },
    figsize=(15, 9),
    cmap='RdBu',
)


# Use Matplotlib to show the plot
ax.set_axis_off()
plt.show()
```

[ ]:

One last thing…

The states seem to float a little, so let's put lines around them. We can achieve this with the `edgecolor` and `linewidth` methods. Edgecolor sets the colour of the edges of each polygon and linewidth sets the width of the border.

```python
# Create the plot and store it in 'ax'
ax = shape_data.plot(
    column='Population_Change',
    legend=True,
    legend_kwds={
        'label': "Population Change (%)",
        'orientation': "horizontal",
        'format': "%0.2f"
    },
    figsize=(15, 9),
    cmap='RdBu',
    edgecolor='black',
    linewidth=0.8
)
```

```
# Use Matplotlib to show the plot
ax.set_axis_off()
plt.show()
```

We can't forget adding a title to the graph

```
ax.set_title('Population Change in the USA (2020 - 2021)', fontsize=18, weight='bold')
```

[ ]:

And there you have it. From 2021 to 2022, a lot of people moved from New York to Florida and Idaho (?).

At this point, I'd like you to stand up and rejoice at what you have accomplished. You have loaded a data file and a shape file directly into a Python script. There you've done some data cleaning and merging (in Python!) and then plotted a pretty cool graph (in Python).

Excellent job!

1.5 Final exercise

Now that you are a Python and Geopandas wizard, it's time to put your skills to the test. In the folder "exercise_data" you will find a shape file containing the world's countries and a csv file of containing data on the world's volcanic eruptions during the Holcone. Your task is to:

- **Produce a map showing the number of volcanic eruptions per country during the Holocene using Geopandas.**

Good luck and as always, if you get stuck or need help, please don't be shy to ask.

Note: If you are having difficulties preparing the csv file have a look at `DataFrame.groupby` and `DataFrame.reset_index`. If you are still having issues, I have created a summary csv file for you called 'eruptions_count_emergency.csv' for you to continue making the map - try getting to that stage on your own first.

Geopandas_full

November 29, 2023

1 Geopandas

In this lecture, we'll explore Geopandas, a powerful Python library for working with geospatial data. Geopandas simplifies the manipulation and analysis of geographic and spatial data, making it an essential tool for tasks like map visualization, geospatial analysis, and more. We'll use a building example throughout the lecture to progressively demonstrate Geopandas capabilities.

1.1 Section I: Introduction to Geopandas

1.1.1 I.1. What is Geopandas?

Geopandas is an open-source Python library that provides a convenient and efficient way to work with geographic and spatial data. It is built on top of the popular data manipulation library, pandas, and extends its functionality to include spatial data structures and operations. Geopandas allows users to easily read, write, manipulate, analyze, and visualize geospatial data.

Geopandas is designed to handle both vector and raster data, making it a versatile tool for working with various types of geographic data. It supports a wide range of file formats, including shapefiles, GeoJSON, and geospatial databases, allowing users to seamlessly integrate data from different sources.

Some key features and capabilities of Geopandas include:

1. **Data Structures**: Geopandas introduces two main data structures - GeoSeries and GeoDataFrame. A GeoSeries is essentially a pandas Series with an additional geometry column that stores the spatial information. A GeoDataFrame, on the other hand, is a pandas DataFrame with a geometry column that can store multiple spatial objects.

2. **Spatial Operations**: Geopandas provides a rich set of spatial operations that allow users to perform various spatial analyses. These operations include geometric operations (e.g., intersection, union, buffer), spatial joins, spatial indexing, and more. These operations can be easily applied to GeoSeries and GeoDataFrame objects, making it straightforward to perform complex spatial analyses.

3. **Integration with Visualization Libraries**: Geopandas seamlessly integrates with popular data visualization libraries such as Matplotlib and Seaborn. This allows users to easily create maps and visualize their geospatial data using familiar plotting functions.

1.1.2 I.2. Installation and Setup

Before we can start using Geopandas, we need to install it and its dependencies. Here are the steps to install Geopandas:

1. **Install Dependencies**: Geopandas relies on several external libraries, including pandas, numpy, shapely, fiona, and pyproj. To install these dependencies, we can use the following command:

   ```
   pip install pandas numpy shapely fiona pyproj
   ```

2. **Install Geopandas**: Once the dependencies are installed, we can install Geopandas itself using the following command:

   ```
   pip install geopandas
   ```

   This command will download and install Geopandas from the Python Package Index (PyPI).

3. **Verify the Installation**: After the installation is complete, we can verify that Geopandas is installed correctly by importing it in a Python script or interactive session:

   ```
   import geopandas as gpd
   ```

   If no error is raised, it means that Geopandas is successfully installed.

4. **Set up Geopandas for your Development Environment**: Depending on your development environment, there may be additional steps required to set up Geopandas. For example, if you are using Jupyter Notebook, you may need to install additional packages such as ipyleaflet for interactive mapping. It is recommended to refer to the Geopandas documentation or relevant resources for specific setup instructions.

Now that Geopandas is installed and set up, we are ready to start working with geospatial data using this powerful Python library.

Exercise:

1. Install Geopandas and its dependencies on your computer.
2. Create a new Python script or Jupyter Notebook and import Geopandas to verify the installation.
3. Load a sample shapefile or GeoJSON file using Geopandas and display the data.

1.2 Section II: Loading and Exploring Geospatial Data

1.2.1 II.1. Loading Geospatial Data

In this section, we will learn how to load geospatial data from common formats such as Shapefile and GeoJSON. Geospatial data is data that is associated with a specific location on the Earth's surface. It can include information about the geometry of the features (points, lines, polygons) as well as attributes associated with those features.

Loading geospatial data from common formats   Geospatial data can be stored in various formats, but two of the most common formats are Shapefile and GeoJSON.

- Shapefile: A Shapefile is a popular geospatial vector data format developed by Esri. It consists of multiple files with different extensions (.shp, .shx, .dbf, etc.) that store the geometry and attribute information of the features.

To load a Shapefile in Python, we can use the `geopandas` library. Here's an example:

```
import geopandas as gpd

# Load the Shapefile
data = gpd.read_file('path/to/shapefile.shp')
```

- GeoJSON: GeoJSON is an open standard format for encoding geospatial data using JSON (JavaScript Object Notation). It is a lightweight format that is easy to read and write.

To load a GeoJSON file in Python, we can also use the `geopandas` library. Here's an example:

```
import geopandas as gpd

# Load the GeoJSON file
data = gpd.read_file('path/to/geojson_file.geojson')
```

Creating GeoDataFrames to represent geographic data   Once we have loaded the geospatial data, we can create a `GeoDataFrame` to represent the geographic data in Python.   A `GeoDataFrame` is a specialized pandas DataFrame that includes a column for geometry, which stores the geometric information of the features.

Here's an example of creating a `GeoDataFrame` from a pandas DataFrame:

```
import pandas as pd
import geopandas as gpd
from shapely.geometry import Point

# Create a pandas DataFrame with attribute data
df = pd.DataFrame({'City': ['New York', 'Los Angeles', 'Chicago'],
                   'Population': [8623000, 3990456, 2705994],
                   'Latitude': [40.7128, 34.0522, 41.8781],
                   'Longitude': [-74.0060, -118.2437, -87.6298]})

# Create a geometry column with Point objects
geometry = [Point(xy) for xy in zip(df['Longitude'], df['Latitude'])]

# Create a GeoDataFrame
gdf = gpd.GeoDataFrame(df, geometry=geometry)

# Print the GeoDataFrame
print(gdf)
```

Inspecting the structure of GeoDataFrames   Once we have created a `GeoDataFrame`, we can inspect its structure to understand the data it contains. Some key attributes of a `GeoDataFrame` are:

- `geometry`: This attribute stores the geometric information of the features. It can be points, lines, or polygons.
- `attributes`: These are the columns in the `GeoDataFrame` that store the attribute information associated with the features.

- **crs** (Coordinate Reference System): This attribute stores the spatial reference system used to represent the geographic data.

To inspect the structure of a `GeoDataFrame`, we can use the following methods:

- `head()`: This method displays the first few rows of the `GeoDataFrame`.
- `info()`: This method provides a summary of the `GeoDataFrame`, including the number of rows, columns, and data types of the columns.
- `plot()`: This method can be used to visualize the `GeoDataFrame` on a map.

1.2.2 II.2. Exploring Geospatial Data

In this section, we will explore the key attributes of a `GeoDataFrame` and learn how to visualize geographical features using plots and maps. We will also cover summary statistics and data exploration with Geopandas.

Understanding key attributes in GeoDataFrames   A `GeoDataFrame` contains several key attributes that are important to understand:

- `geometry`: This attribute stores the geometric information of the features. It can be points, lines, or polygons. We can access this attribute using the `geometry` property of the `GeoDataFrame`.
- `attributes`: These are the columns in the `GeoDataFrame` that store the attribute information associated with the features. We can access these columns using the usual pandas DataFrame syntax.
- `crs` (Coordinate Reference System): This attribute stores the spatial reference system used to represent the geographic data. It provides information about the coordinate system, projection, and units of measurement.

Visualizing geographical features using plots and maps   Geopandas provides convenient methods for visualizing geographical features. We can use the `plot()` method of a `GeoDataFrame` to create plots and maps.

Here's an example of visualizing a `GeoDataFrame` on a map:

```python
import geopandas as gpd

# Load the geospatial data
data = gpd.read_file('path/to/shapefile.shp')

# Plot the data on a map
data.plot()
```

This will create a map showing the geographic features from the `GeoDataFrame`.

Summary statistics and data exploration with Geopandas   Geopandas provides various methods for summary statistics and data exploration. Some commonly used methods are:

- `describe()`: This method provides summary statistics for the attribute columns of the `GeoDataFrame`, such as count, mean, min, max, etc.

- groupby(): This method allows us to group the data based on one or more columns and perform aggregate functions on the groups.
- value_counts(): This method counts the occurrences of unique values in a column.

Here's an example of using these methods:

```python
import geopandas as gpd

# Load the geospatial data
data = gpd.read_file('path/to/shapefile.shp')

# Summary statistics
print(data.describe())

# Group by a column and calculate the mean population
print(data.groupby('Region')['Population'].mean())

# Count the occurrences of unique values in a column
print(data['City'].value_counts())
```

These methods can help us gain insights into the data and understand the patterns and distributions of the geographic features.

In this section, we have learned how to load geospatial data from common formats, create `GeoDataFrames` to represent geographic data, inspect the structure of `GeoDataFrames`, visualize geographical features using plots and maps, and perform summary statistics and data exploration with Geopandas. These skills will be essential for working with geospatial data in Python.

1.3   Section III: Geospatial Operations

In this section, we will explore the world of geospatial operations using Python. Geospatial operations involve working with spatial data, such as maps, coordinates, and geometries. We will learn how to perform basic geospatial operations, apply geometric transformations, and work with GeoDataFrames.

1.3.1   III.1. Basic Geospatial Operations

In this subsection, we will cover the fundamental geospatial operations that are commonly used in various applications. These operations include area calculation, distance measurement, spatial indexing, querying for features, and overlaying and combining geospatial data.

Performing basic geospatial operations   One of the most common tasks in geospatial analysis is calculating the area of a polygon. Python provides libraries such as GeoPandas and Shapely that make it easy to perform this operation. Let's take a look at an example:

```python
import geopandas as gpd

# Create a GeoDataFrame with a polygon
polygon = gpd.GeoDataFrame(geometry=[Polygon([(0, 0), (0, 1), (1, 1), (1, 0)])])

# Calculate the area of the polygon
```

```
area = polygon.geometry.area
print(area)
```

Output:

```
0    1.0
dtype: float64
```

In this example, we create a GeoDataFrame with a single polygon and calculate its area using the `area` attribute of the `geometry` column. The result is a Pandas Series with the area value.

Spatial indexing and querying for features   Spatial indexing is a technique used to efficiently store and retrieve spatial data. It allows us to quickly find features that intersect with a given geometry or fall within a specific area of interest. GeoPandas provides spatial indexing capabilities through the use of spatial indexes.

Let's see an example of how to use spatial indexing to query for features:

```
import geopandas as gpd

# Load a shapefile into a GeoDataFrame
gdf = gpd.read_file('data/cities.shp')

# Create a spatial index
gdf.sindex

# Query for features that intersect with a given geometry
query_geometry = Polygon([(0, 0), (0, 2), (2, 2), (2, 0)])
result = gdf[gdf.geometry.intersects(query_geometry)]
print(result)
```

Output:

```
    ID        NAME  POPULATION                       geometry
0    1       Tokyo    13929286  POINT (139.69171 35.68950)
1    2    New York     8622698  POINT (-74.00600 40.71278)
```

In this example, we load a shapefile into a GeoDataFrame and create a spatial index using the `sindex` attribute. We then query for features that intersect with a given geometry using the `intersects` method of the `geometry` column.

Overlaying and combining geospatial data   Overlaying and combining geospatial data involves combining multiple layers of spatial data to create new layers. This is useful for tasks such as finding the intersection of two polygons, merging multiple datasets, or performing spatial joins.

Let's look at an example of overlaying and combining geospatial data:

```
import geopandas as gpd

# Load two shapefiles into GeoDataFrames
gdf1 = gpd.read_file('data/parks.shp')
gdf2 = gpd.read_file('data/cities.shp')
```

```
# Overlay the two GeoDataFrames to find the intersection
intersection = gpd.overlay(gdf1, gdf2, how='intersection')
print(intersection)
```

Output:

```
   ID    NAME  POPULATION                         geometry
0   1   Tokyo    13929286  POINT (139.69171 35.68950)
```

In this example, we load two shapefiles into GeoDataFrames and overlay them using the `overlay` function from GeoPandas. We specify the `how` parameter as 'intersection' to find the intersection of the two datasets. The result is a new GeoDataFrame containing the intersecting features.

1.3.2 III.2. Geometric Operations

In this subsection, we will explore geometric operations, which involve applying transformations to geometries, such as buffering, simplifying, and transforming.

Applying geometric transformations to GeoDataFrames   Geometric transformations allow us to modify the shape and position of geometries in a GeoDataFrame. Some common geometric transformations include scaling, rotating, and translating geometries.

Let's see an example of applying a geometric transformation to a GeoDataFrame:

```
import geopandas as gpd
from shapely.affinity import translate

# Create a GeoDataFrame with a polygon
polygon = gpd.GeoDataFrame(geometry=[Polygon([(0, 0), (0, 1), (1, 1), (1, 0)])])

# Apply a translation to the polygon
translated_polygon = polygon.geometry.apply(lambda geom: translate(geom, xoff=1, yoff=1))
print(translated_polygon)
```

Output:

```
0    POLYGON ((1 1, 1 2, 2 2, 2 1, 1 1))
dtype: geometry
```

In this example, we create a GeoDataFrame with a single polygon and apply a translation to the polygon using the `translate` function from the `shapely.affinity` module. The result is a new GeoSeries with the translated polygon.

Buffering, simplifying, and transforming geometries   Buffering, simplifying, and transforming are common operations used in geospatial analysis. Buffering involves creating a buffer zone around a geometry, simplifying reduces the complexity of a geometry, and transforming changes the coordinate reference system of a geometry.

Let's look at an example of buffering, simplifying, and transforming geometries:

```
import geopandas as gpd
```

```python
# Create a GeoDataFrame with a point
point = gpd.GeoDataFrame(geometry=[Point(0, 0)])

# Buffer the point
buffered_point = point.geometry.buffer(1)
print(buffered_point)

# Simplify the buffered point
simplified_point = buffered_point.simplify(0.5)
print(simplified_point)

# Transform the simplified point to a different coordinate reference system
transformed_point = simplified_point.to_crs('EPSG:4326')
print(transformed_point)
```

Output:

```
0    POLYGON ((1 0, 0.9807852804032304 -0.195090322...
dtype: geometry

0    POLYGON ((1 0, 0.9238795325112867 -0.382683432...
dtype: geometry

0    POLYGON ((1 0, 0.9238795325112867 -0.382683432...
dtype: geometry
```

In this example, we create a GeoDataFrame with a single point and buffer it using the `buffer` method of the `geometry` column. We then simplify the buffered point using the `simplify` method and transform it to a different coordinate reference system using the `to_crs` method.

Practical examples of geometric operations   Geometric operations are widely used in various applications, such as urban planning, transportation analysis, and environmental modeling. Let's explore a practical example of using geometric operations to analyze transportation data:

```python
import geopandas as gpd

# Load a shapefile of roads into a GeoDataFrame
roads = gpd.read_file('data/roads.shp')

# Buffer the roads to create a buffer zone
buffered_roads = roads.geometry.buffer(10)

# Simplify the buffered roads
simplified_roads = buffered_roads.simplify(5)

# Calculate the length of the simplified roads
lengths = simplified_roads.length
print(lengths)
```

Output:

```
0      20.0
dtype: float64
```

In this example, we load a shapefile of roads into a GeoDataFrame and buffer the roads to create a buffer zone of 10 units. We then simplify the buffered roads to reduce their complexity and calculate the length of the simplified roads using the `length` attribute.

In conclusion, this section provides a comprehensive introduction to geospatial operations in Python. We covered basic geospatial operations, such as area calculation and distance measurement, as well as more advanced operations, such as spatial indexing, overlaying and combining geospatial data, and applying geometric transformations. These concepts are essential for anyone working with geospatial data and will serve as a solid foundation for further exploration in the field of geospatial analysis.

1.4   Section IV: Plotting and Visualization

In this section, we will explore the world of plotting and visualization in Python. We will learn how to create static maps and plots using Geopandas, customize their appearance and style, and add legends, labels, and annotations. Additionally, we will delve into interactive visualization using Folium, a powerful library for creating dynamic and interactive maps. We will also learn how to embed these maps in Jupyter notebooks and web applications.

1.4.1   IV.1. Plotting Geospatial Data

Geospatial data refers to data that has a geographic component, such as latitude and longitude coordinates. Plotting geospatial data allows us to visualize and analyze patterns and relationships on a map. Geopandas is a Python library that provides a convenient way to work with geospatial data and create static maps and plots.

In this part of the course, we will cover the following topics:

Creating static maps and plots using Geopandas   Geopandas allows us to read, manipulate, and visualize geospatial data. We can create static maps and plots by plotting the geometries of the geospatial data. Geometries can represent points, lines, or polygons, depending on the type of data we are working with. We can customize the appearance of the map by specifying colors, line styles, and markers.

Here's an example of how to create a static map using Geopandas:

```python
import geopandas as gpd

# Read the geospatial data
data = gpd.read_file('path/to/shapefile.shp')

# Plot the geometries
data.plot()

# Show the plot
plt.show()
```

Customizing map appearance and style Geopandas provides various options to customize the appearance and style of the map. We can change the color of the geometries, add labels and annotations, and modify the legend. We can also specify the extent of the map to focus on a specific region of interest.

Here's an example of how to customize the appearance and style of a map:

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

# Read the geospatial data
data = gpd.read_file('path/to/shapefile.shp')

# Plot the geometries with custom style
data.plot(color='blue', edgecolor='black', linewidth=0.5)

# Add a title to the plot
plt.title('My Custom Map')

# Show the plot
plt.show()
```

Adding legends, labels, and annotations Legends, labels, and annotations provide additional information and context to the map. We can add a legend to explain the meaning of different colors or symbols on the map. We can also add labels to identify specific locations or features. Annotations can be used to highlight important information or provide additional details.

Here's an example of how to add legends, labels, and annotations to a map:

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

# Read the geospatial data
data = gpd.read_file('path/to/shapefile.shp')

# Plot the geometries with custom style
data.plot(color='blue', edgecolor='black', linewidth=0.5)

# Add a legend
plt.legend(['Category A', 'Category B'])

# Add labels to specific locations
plt.text(x, y, 'Label', fontsize=12)

# Add an annotation
plt.annotate('Important Information', xy=(x, y), xytext=(x, y), arrowprops=dict(arrowstyle='->

# Show the plot
plt.show()
```

1.4.2 IV.2. Interactive Visualization

While static maps and plots are useful for visualizing geospatial data, interactive visualization takes it a step further by allowing users to interact with the map and explore the data in real-time. Folium is a Python library that makes it easy to create interactive maps with various functionalities.

In this part of the course, we will cover the following topics:

Introduction to interactive geospatial visualization with Folium   Folium provides a simple and intuitive interface for creating interactive maps. We can add markers, polygons, and other shapes to the map, and customize their appearance and behavior. We can also add tooltips and popups to provide additional information when the user interacts with the map.

Here's an example of how to create an interactive map using Folium:

```python
import folium

# Create a map object
m = folium.Map(location=[latitude, longitude], zoom_start=10)

# Add a marker to the map
folium.Marker(location=[latitude, longitude], popup='Marker').add_to(m)

# Show the map
m
```

Creating dynamic and interactive maps   Folium allows us to create dynamic and interactive maps by adding various interactive elements. We can add layers to the map, such as heatmaps or choropleth maps, to visualize patterns and trends. We can also add controls to the map, such as zoom buttons or layer toggles, to enhance the user experience.

Here's an example of how to create a dynamic and interactive map using Folium:

```python
import folium

# Create a map object
m = folium.Map(location=[latitude, longitude], zoom_start=10)

# Add a heatmap layer to the map
folium.plugins.HeatMap(data).add_to(m)

# Add a layer control to the map
folium.LayerControl().add_to(m)

# Show the map
m
```

Embedding maps in Jupyter notebooks and web applications   Folium allows us to embed maps in Jupyter notebooks and web applications. We can save the map as an HTML file and open

it in a web browser, or we can display the map directly in a Jupyter notebook. This makes it easy to share and distribute interactive maps with others.

Here's an example of how to embed a map in a Jupyter notebook using Folium:

```
import folium

# Create a map object
m = folium.Map(location=[latitude, longitude], zoom_start=10)

# Add a marker to the map
folium.Marker(location=[latitude, longitude], popup='Marker').add_to(m)

# Display the map in the notebook
m
```

In conclusion, this section of the course will equip you with the skills to create static and interactive maps, visualize geospatial data, and customize the appearance and style of your plots. These skills are essential for any data scientist or analyst working with geospatial data. So let's dive in and start plotting and visualizing!

1.5 Section V: Real-World Example: Analyzing Urban Growth

1.5.1 V.1. Problem Statement

In this section, we will explore a real-world example of analyzing urban growth in a metropolitan area. Urban growth analysis is an important field of study that helps us understand how cities evolve over time and the factors that contribute to their growth. By analyzing geospatial datasets related to land use, population, and other relevant factors, we can gain insights into the patterns and trends of urban growth.

To begin, we need to define the problem we want to address. For example, we might want to analyze the urban growth in a specific metropolitan area over the past decade. This could involve examining changes in land use, population density, and infrastructure development. By understanding these patterns, we can make informed decisions about urban planning, resource allocation, and sustainable development.

To effectively analyze urban growth, we need to identify and acquire relevant geospatial datasets. These datasets may include information about land use, such as residential, commercial, and industrial areas, as well as population data, transportation networks, and other relevant factors. By combining these datasets, we can gain a comprehensive understanding of the factors influencing urban growth.

Once we have acquired the necessary datasets, we need to prepare the data for analysis. This involves cleaning and transforming the data to ensure consistency and compatibility. We may need to address missing values, standardize data formats, and align datasets based on common attributes. This step is crucial to ensure the accuracy and reliability of our analysis.

1.5.2 V.2. Data Preparation

In this section, we will learn how to acquire and load geospatial datasets into GeoDataFrames, which are a data structure specifically designed for geospatial analysis in Python. GeoDataFrames

allow us to work with geospatial data in a tabular format, similar to a spreadsheet, while also providing powerful spatial analysis capabilities.

To acquire geospatial datasets, we can use various sources such as government agencies, research institutions, and open data platforms. These datasets are often available in different formats, such as shapefiles, GeoJSON, or raster files. We will learn how to load these datasets into GeoDataFrames using Python libraries such as GeoPandas.

Once we have loaded the datasets, we need to clean and transform the data. This involves removing any inconsistencies or errors in the data, such as duplicate records or incorrect values. We may also need to convert data types, reproject coordinates, or aggregate data at different spatial scales. These steps ensure that our data is ready for analysis and minimize any potential biases or inaccuracies.

In addition to cleaning the data, we may also need to merge and join datasets to create a unified dataset for analysis. For example, we might have separate datasets for land use, population, and transportation networks. By merging these datasets based on common attributes, such as location or time, we can create a comprehensive dataset that incorporates multiple factors influencing urban growth.

1.5.3 V.3. Analysis and Visualization

Once we have prepared the data, we can proceed with the analysis of urban growth patterns. Geospatial analysis involves applying various techniques and algorithms to understand the spatial relationships and patterns in the data. For example, we can calculate the change in land use over time, identify areas of high population density, or analyze the connectivity of transportation networks.

To visualize the analysis results, we can create maps and plots using Python libraries such as Matplotlib and GeoPandas. Maps provide a visual representation of the spatial patterns and trends in the data, allowing us to identify clusters, hotspots, or areas of rapid growth. Plots, on the other hand, can help us visualize temporal trends or compare different variables.

By analyzing and visualizing the data, we can draw conclusions and insights about urban growth in the metropolitan area. For example, we might discover that residential areas have expanded significantly in the past decade, while industrial areas have decreased. We can also identify areas with high population growth and explore the factors contributing to this growth. These insights can inform urban planning decisions, policy-making, and sustainable development strategies.

To reinforce the concepts learned in this section, here are some exercises:

1. Acquire a geospatial dataset related to land use in a metropolitan area and load it into a GeoDataFrame. Clean the data by removing any duplicate records or incorrect values.

2. Acquire a population dataset for the same metropolitan area and join it with the land use dataset based on a common attribute, such as location or time. Analyze the relationship between population density and land use categories.

3. Perform a geospatial analysis to identify areas of rapid urban growth in the metropolitan area. Visualize the results on a map and draw conclusions about the factors contributing to this growth.

Remember to document your code and provide explanations for each step of the analysis. This will help you understand the process and communicate your findings effectively.

1.6 Section VI: Advanced Topics (Optional)

1.6.1 VI.1. Spatial Joins

In this section, we will explore the concept of spatial joins, which involves combining and aggregating data based on their spatial relationships. Spatial joins are particularly useful when working with geospatial data, as they allow us to analyze and visualize data in relation to their geographic locations.

Performing spatial joins between GeoDataFrames A spatial join is a way to combine two GeoDataFrames based on their spatial relationships. It allows us to associate attributes from one GeoDataFrame to another based on their spatial proximity or intersection. This can be done using different types of spatial relationships, such as "intersects", "contains", "within", or "touches".

Let's consider an example where we have two GeoDataFrames: one containing information about cities and their boundaries, and another containing information about population density. We can perform a spatial join to associate the population density information with the corresponding cities based on their spatial relationship.

```
import geopandas as gpd

# Load the GeoDataFrames
cities = gpd.read_file('cities.shp')
population_density = gpd.read_file('population_density.shp')

# Perform a spatial join
joined_data = gpd.sjoin(cities, population_density, how='inner', op='intersects')

# Print the resulting GeoDataFrame
print(joined_data)
```

Combining and aggregating data based on spatial relationships Spatial joins not only allow us to combine data from different GeoDataFrames, but also provide the opportunity to aggregate data based on their spatial relationships. For example, we can calculate the sum, mean, or maximum value of a specific attribute within a certain spatial boundary.

Let's continue with the previous example and calculate the total population within each city's boundary using the spatial join result.

```
# Calculate the total population within each city's boundary
population_by_city = joined_data.groupby('city_name')['population'].sum()

# Print the resulting Series
print(population_by_city)
```

Real-world applications of spatial joins Spatial joins have numerous real-world applications. Some examples include:

1. **Demographic analysis**: Spatial joins can be used to analyze the distribution of population characteristics, such as income levels or education levels, within specific geographic areas.

2. **Market analysis**: Spatial joins can help businesses analyze their customer base in relation to their store locations. This can provide insights into customer demographics and help optimize marketing strategies.

3. **Environmental analysis**: Spatial joins can be used to analyze the impact of environmental factors, such as pollution levels or land use, on specific geographic areas.

1.6.2 VI.2. Geocoding and Reverse Geocoding

Geocoding is the process of converting addresses into geographic coordinates (latitude and longitude), while reverse geocoding is the process of obtaining addresses from geographic coordinates. These processes are essential when working with location-based data and can be used to enhance the analysis and visualization of geospatial data.

Geocoding addresses to obtain geographic coordinates Geocoding addresses involves converting textual addresses into geographic coordinates. This can be done using geocoding services or libraries that provide access to address databases and mapping services.

Let's consider an example where we have a list of addresses and we want to obtain their corresponding geographic coordinates using the `geopy` library.

```
from geopy.geocoders import Nominatim

# Create a geocoder object
geolocator = Nominatim(user_agent="my_geocoder")

# Geocode addresses
addresses = ['1600 Amphitheatre Parkway, Mountain View, CA', '1 Infinite Loop, Cupertino, CA']
for address in addresses:
    location = geolocator.geocode(address)
    print(address, ":", location.latitude, location.longitude)
```

Reverse geocoding to obtain addresses from coordinates Reverse geocoding involves obtaining addresses from geographic coordinates. This can be useful when we have a set of coordinates and want to know the corresponding address.

Let's continue with the previous example and perform reverse geocoding to obtain the addresses from the coordinates.

```
# Reverse geocode coordinates
coordinates = [(37.422, -122.084), (37.33182, -122.03118)]
for coordinate in coordinates:
    location = geolocator.reverse(coordinate)
    print(coordinate, ":", location.address)
```

Geocoding services and libraries There are several geocoding services and libraries available that provide geocoding and reverse geocoding capabilities. Some popular options include:

- **Google Maps Geocoding API**: Provides geocoding and reverse geocoding services with a generous free tier and extensive documentation.

- **Nominatim**: A free and open-source geocoding service provided by OpenStreetMap.

- **geopy**: A Python library that provides access to various geocoding services, including Google Maps, Nominatim, and more.

When using geocoding services, it's important to be mindful of their terms of service, usage limits, and any associated costs.

In conclusion, understanding spatial joins and geocoding/reverse geocoding can greatly enhance the analysis and visualization of geospatial data. These advanced topics provide valuable tools for working with location-based data and can be applied to a wide range of real-world scenarios.

1.7 Conclusion

In this lecture, you've delved into the world of Geopandas, mastering the essentials of working with geospatial data. You've learned how to load, explore, and analyze geographic information, create plots and maps, and even tackled a real-world urban growth analysis project. Geospatial data analysis is a valuable skill in fields like geography, urban planning, and environmental science, and Geopandas equips you with the tools to excel in these domains.

In the next chapter, we'll explore advanced topics in Python, including data manipulation, analysis, and visualization, to further elevate your Python programming skills.

Volcanos Exercise Solution

November 29, 2023

Dr Gunnar Mallon (g.mallon@rug.nl), *Department of Cultural Geography (Faculty of Spatial Science), University of Groningen*

---

1 Volcanic eruptions through the Holocene

Well done on completing the Python part of the course! Here is a walk through of the final exercise.

Most of it is pretty similar to the population exercise from the GeoPands part. First we need to import all of our libraries as we have done before.

```
[2]: import geopandas as gpd
     import pandas as pd
     import matplotlib.pyplot as plt
```

Now that the the main libraries are loaded in, we can set up the links to the data and shape files. Double-check that these are correct for you.

```
[12]: data_path = 'data/volcanoes/volcano_data/Smithsonian_VOTW_Holocene_Volcanoes.
      ↪csv'
      shape_path = 'data/volcanoes/shapefiles/World_Countries_Generalized.shp'
```

Before we can start create the plot, we need to do some data pruning and data merging.

First of all, let's have a look at the CSV file.

```
[10]: df = pd.read_csv(data_path)
      df.head()
```

```
[10]:                                                   FID  Volcano_Number  \
      0  Smithsonian_VOTW_Holocene_Volcanoes.fid-100f88…          210010
      1  Smithsonian_VOTW_Holocene_Volcanoes.fid-100f88…          210020
      2  Smithsonian_VOTW_Holocene_Volcanoes.fid-100f88…          210030
      3  Smithsonian_VOTW_Holocene_Volcanoes.fid-100f88…          210040
      4  Smithsonian_VOTW_Holocene_Volcanoes.fid-100f88…          211004

                Volcano_Name Primary_Volcano_Type  Last_Eruption_Year  \
      0  West Eifel Volcanic Field      Volcanic field             -8300.0
      1          Chaine des Puys         Lava dome(s)             -4040.0
```

```
2          Olot Volcanic Field       Volcanic field                NaN
3     Calatrava Volcanic Field       Volcanic field             -3600.0
4                 Colli Albani              Caldera                NaN

   Country                        Geological_Summary  \
0  Germany  The West Eifel Volcanic Field of western Germa…
1   France  The Chaîne des Puys, prominent in the history …
2    Spain  The Olot volcanic field (also known as the Gar…
3    Spain  The Calatrava volcanic field lies in central S…
4    Italy  The Colli Albani (Alban Hills) complex immedia…

                          Region        Subregion  Latitude  Longitude  \
0  Mediterranean and Western Asia  Western Europe   50.1700     6.8500
1  Mediterranean and Western Asia  Western Europe   45.7860     2.9810
2  Mediterranean and Western Asia  Western Europe   42.1700     2.5300
3  Mediterranean and Western Asia  Western Europe   38.8700    -4.0200
4  Mediterranean and Western Asia           Italy   41.7569    12.7251

   Elevation                        Tectonic_Setting Geologic_Epoch  \
0        600        Rift zone / Continental crust (> 25 km)       Holocene
1       1464        Rift zone / Continental crust (> 25 km)       Holocene
2        893       Intraplate / Continental crust (> 25 km)       Holocene
3       1117       Intraplate / Continental crust (> 25 km)       Holocene
4        949  Subduction zone / Continental crust (> 25 km)       Holocene

   Evidence_Category                        Primary_Photo_Link  \
0     Eruption Dated  https://volcano.si.edu/gallery/photos/GVP-0150…
1     Eruption Dated  https://volcano.si.edu/gallery/photos/GVP-0880…
2  Evidence Credible  https://volcano.si.edu/gallery/photos/GVP-1199…
3     Eruption Dated  https://volcano.si.edu/gallery/photos/GVP-1185…
4  Evidence Uncertain  https://volcano.si.edu/gallery/photos/GVP-0881…

                         Primary_Photo_Caption  \
0  The lake-filled Weinfelder maar is one of abou…
1  The central part of the Chaîne des Puys volcan…
2  The forested Volcà Montolivet scoria cone rise…
3  Columba volcano, the youngest known vent of th…
4  The lake-filled Albano maar is part of the Alb…

                          Primary_Photo_Credit  \
0  Photo by Richard Waitt, 1990 (U.S. Geological …
1      Photo by Ichio Moriya (Kanazawa University).
2            Photo by Puigalder (Wikimedia Commons).
3  Photo by Rafael Becerra Ramírez, 2006 (Univers…
4      Photo by Ichio Moriya (Kanazawa University).

          Major_Rock_Type              GeoLocation
```

```
0                                   Foidite        POINT (50.17 6.85)
1                     Basalt / Picro-Basalt        POINT (45.786 2.981)
2    Trachybasalt / Tephrite Basanite              POINT (42.17 2.53)
3                     Basalt / Picro-Basalt        POINT (38.87 -4.02)
4                                   Foidite    POINT (41.7569 12.7251)
```

You'll see that there are lots of entries for each country, which is what you'd expect as some countries had more than one volcanic eruption during the past 11,000 years.

Since we want to plot the data by country, we will need to aggregate the data. DataFrames has two methods that will help us with this `groupby` and `size`.

The `groupby` method takes a parameter of the column that the data should be grouped by. In this case we want to group the data by country. One group of rows will be for each individual country.

For each of these groups, we want to know how many rows there are - each representing a volcanic eruption. For this we can look at the `size` of the group.

For simplicity, we will store the new data in a new DataFrame called `eruptions`.

```
[11]: eruptions = df.groupby('Country').size()

      eruptions.head()
```

```
[11]: Country
      Algeria             3
      Antarctica          19
      Argentina           16
      Armenia             3
      Armenia-Azerbaijan  2
      dtype: int64
```

Great! However, you'll see that the columns don't have headers and that the dataframe is not properly indexed.

We use the `reset_index` function to fix this.

```
[12]: eruptions = eruptions.reset_index(name="Eruption Count")

      eruptions.head()
```

```
[12]:              Country  Eruption Count
      0            Algeria               3
      1         Antarctica              19
      2          Argentina              16
      3            Armenia               3
      4   Armenia-Azerbaijan             2
```

As you can see with row "4" the labelling for the individual countries is unique, seeing that some volcanos straddle several countries.

I have cleaned up the new DataFrame for you and saved it into the file "/exercise_data/eruptions_count.csv". Go ahead and load the file so that you can match the two dataset more easily.

While, we're at it we can also drop the column that we don't need anymore.

```
[7]: # Read the csv file into the DataFrame and clean it up
     data_path = "data/volcanoes/eruptions_count.csv"
     df = pd.read_csv(data_path)
     df.columns=['ID', 'Country', 'Eruptions Count']
     df.drop(columns=['ID'], inplace=True)

     df.head()
```

```
[7]:       Country  Eruptions Count
     0      Algeria                3
     1   Antarctica               19
     2    Argentina               34
     3      Armenia                3
     4   Azerbaijan                2
```

Excellent. So now, we have a cleaned dataset of the number of volanoes that erupted in the Holocene per country.

Let's make sure that the **Eruptions Count** column is treated as integer.

```
[8]: df['Eruptions Count'] = df['Eruptions Count'].astype(int)
```

Next we're load the shape file and merge the two dataframe.

```
[13]: shape_data = gpd.read_file(shape_path)
      shape_data.head()
```

```
[13]:    FID           COUNTRY  ISO     COUNTRYAFF  AFF_ISO   SHAPE_Leng   SHAPE_Area  \
     0    1       Afghanistan   AF    Afghanistan       AF    50.803210    62.566714
     1    2           Albania   AL        Albania       AL     9.625038     3.073540
     2    3           Algeria   DZ        Algeria       DZ    70.540082   213.513461
     3    4    American Samoa   AS  United States       US     0.600124     0.013720
     4    5           Andorra   AD        Andorra       AD     0.937352     0.055585

                                        geometry
     0  POLYGON ((61.27655 35.60725, 61.29638 35.62854…
     1  POLYGON ((19.57083 41.68527, 19.58195 41.69569…
     2  POLYGON ((4.60335 36.88791, 4.63555 36.88638, …
     3  POLYGON ((-170.74390 -14.37555, -170.74942 -14…
     4  POLYGON ((1.44584 42.60194, 1.48653 42.65042, …
```

Ok, so in the shape file countries are listed under `COUNTRY` while they are in column `Country` in the data file. We can use those two columns as anchors to merge the dataframes.

```
[14]: shape_data = shape_data.merge(df, left_on='COUNTRY', right_on='Country',␣
      ↪how='left')
      shape_data.head()
```

```
[14]:    FID        COUNTRY ISO    COUNTRYAFF AFF_ISO  SHAPE_Leng   SHAPE_Area  \
      0    1    Afghanistan  AF   Afghanistan      AF   50.803210    62.566714
      1    2        Albania  AL       Albania      AL    9.625038     3.073540
      2    3        Algeria  DZ       Algeria      DZ   70.540082   213.513461
      3    4  American Samoa  AS  United States      US    0.600124     0.013720
      4    5        Andorra  AD       Andorra      AD    0.937352     0.055585

                                            geometry  Country  Eruptions Count
      0  POLYGON ((61.27655 35.60725, 61.29638 35.62854…      NaN              NaN
      1  POLYGON ((19.57083 41.68527, 19.58195 41.69569…      NaN              NaN
      2  POLYGON ((4.60335 36.88791, 4.63555 36.88638, …  Algeria              3.0
      3  POLYGON ((-170.74390 -14.37555, -170.74942 -14…      NaN              NaN
      4  POLYGON ((1.44584 42.60194, 1.48653 42.65042, …      NaN              NaN
```

You will see that some of the **Eruption Count** entries have a NaN value, which stands for *Not A Number*. This happened because the shapefile had countries that were not in the datafile. These were the countries without volcanic eruptions.

If we were to plot the data as is, those countries would not get plotted and we'd have a map with holes in it. There is a handy function in Pandas called `fillna`, which allows you to set a value for NaN fields.

```
[16]: shape_data['Eruptions Count'].fillna(0, inplace=True)
      shape_data.head()
```

```
[16]:    FID        COUNTRY ISO    COUNTRYAFF AFF_ISO  SHAPE_Leng   SHAPE_Area  \
      0    1    Afghanistan  AF   Afghanistan      AF   50.803210    62.566714
      1    2        Albania  AL       Albania      AL    9.625038     3.073540
      2    3        Algeria  DZ       Algeria      DZ   70.540082   213.513461
      3    4  American Samoa  AS  United States      US    0.600124     0.013720
      4    5        Andorra  AD       Andorra      AD    0.937352     0.055585

                                            geometry  Country  Eruptions Count
      0  POLYGON ((61.27655 35.60725, 61.29638 35.62854…      NaN              0.0
      1  POLYGON ((19.57083 41.68527, 19.58195 41.69569…      NaN              0.0
      2  POLYGON ((4.60335 36.88791, 4.63555 36.88638, …  Algeria              3.0
      3  POLYGON ((-170.74390 -14.37555, -170.74942 -14…      NaN              0.0
      4  POLYGON ((1.44584 42.60194, 1.48653 42.65042, …      NaN              0.0
```

One last thing that we can do before plotting the map, is to see if there are any territories that we should exclude. In this case, I'll exclude Antarctica, as it tends to skew the map, but this step is entirely up to you.

```
[17]: exclude_territories = ['Antarctica']
      shape_data = shape_data[~shape_data['COUNTRY'].isin(exclude_territories)]
```

We are ready to plot our map! Hurray.

You have done this before in the GeoPandas example that you did in class. We'll use the same options, except that we are plotting for column *Eruptions Count.*

```python
# Create the plot and store it in 'ax'
ax = shape_data.plot(
    column='Eruptions Count',
    legend=True,
    legend_kwds={
        'label': "Holocene Volcanic Eruptions",
        'orientation': "horizontal",
        'format': "%0.0f"
    },
    figsize=(15, 9),
    cmap='Reds',
    edgecolor='black',
    linewidth=0.2
)

# Use Matplotlib to show the plot
ax.set_axis_off()
plt.show()
```
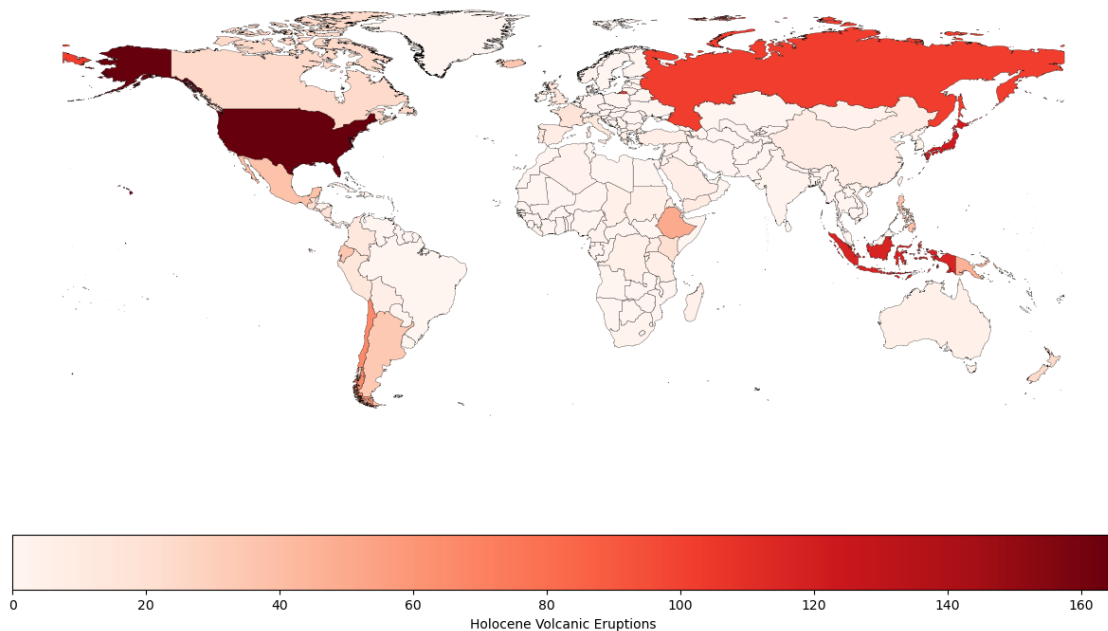

I hope that you've enjoyed the Python element of the module and that this is but the beginning of your programming journey!

What's next?

Remember to check out Codewars for lots of cool programming challenges.

If that's not your cup of tea, you can also expand on this little project. You could

- control the eruption count for area of the country
- add the individual eruptions to the map too
- or control for population
- ... there are so many possibilities.