Introduction to Python

Thijs Smolders
Department of Chemistry - Ångström

What is Python?

- Python is a popular programming language that is widely used for a variety of tasks, such as web development, data analysis, artificial intelligence, and more. It was created by Guido van Rossum and first released in 1991. The name "Python" was inspired by the British comedy group Monty Python, emphasizing the language's focus on simplicity and readability.
- Python is known for its elegant syntax, which means that the code is easy to read and understand. This makes it a great choice for beginners and experienced programmers alike. It emphasizes code readability by using indentation and a clean structure, making it visually appealing and reducing the chances of errors.
- One of Python's strengths is its extensive standard library, which provides a wide range of pre-built modules and functions that you can use in your programs. This allows you to accomplish various tasks without having to start from scratch, saving you time and effort.

What is Python?

- Python is often compared to other programming languages such as C and Fortran. While C and Fortran are considered low-level languages, closer to the hardware, Python is a high-level language. This means that Python abstracts away many of the low-level details and provides simpler syntax and constructs, making it easier to write and understand code. In contrast, C and Fortran provide more control over the hardware and are typically used for performance-critical applications.
- Compared to C and Fortran, Python is more beginner-friendly and focuses on productivity and code readability rather than performance optimization. It is often used as a scripting language and for rapid prototyping. However, Python's versatility and extensive libraries allow it to be used for a wide range of applications, and it can also be optimized for performance using various techniques.
- Python is a powerful and versatile programming language with a focus on simplicity and readability. Its rich history, elegant syntax and extensive standard library make it a great choice for beginners and professionals alike.

The Zen of Python

The Zen of Python is a collection of 19 "guiding principles" for writing computer programs that influence the design of the Python programming language. Software engineer Tim Peters wrote this set of principles and posted it on the Python mailing list in 1999. Peters's list left open a 20th principle "for Guido to fill in", referring to Guido van Rossum, the original author of the Python language.

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

The Zen of Python

The Zen of Python is a collection of 19 "guiding principles" for writing computer programs that influence the design of the Python programming language. Software engineer Tim Peters wrote this set of principles and posted it on the Python mailing list in 1999. Peters's list left open a 20th principle "for Guido to fill in", referring to Guido van Rossum, the original author of the Python language.

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

UPPSALA UNIVERSITET

Variables and datatypes in Python

Variables in Python

In Python, a variable is like a container that can hold different types of data. It allows you to store values and access them later in your program. Think of a variable as a named storage location in the computer's memory where you can assign values and retrieve them as needed.

To use a variable, you first need to declare it by giving it a name. The name should follow certain rules, such as starting with a letter or underscore and consisting of letters, numbers, and underscores. It's important to choose meaningful and descriptive names for variables to enhance code readability.

Variables in Python

Once you have declared a variable, you can assign a value to it using the assignment operator "=" (read as "is assigned to"). For example:

```
name = "John"
age = 25
pi = 3.14159
is_student = True
```

In the example above, we declared variables named name, age, pi, and is_student. We assigned different values to each variable: a string, an integer, a float, and a boolean value, respectively. The values stored in variables can be changed or reassigned throughout your program. For instance:

```
age = 26 # Reassigning the value of age variable
is_student = False # Reassigning the value of is_student variable
```

By reassigning the values, the variables age and is_student now hold new data.

UPPSALA UNIVERSITET

Variables in Python

Variables can also be used in operations and expressions. You can perform calculations, concatenate strings, or manipulate values using variables. For example:

```
radius = 5
area = 3.14 * radius**2 # Calculating the area of a circle
greeting = "Hello, " + name # Concatenating a string with the value of the name variable
```

In the examples above, the variables radius and name are used in mathematical calculations and string concatenation, respectively.

Variables provide flexibility in programming because they can store different types of data and can be assigned new values. This flexibility allows you to perform dynamic computations and work with changing data throughout your program.

Understanding variables and their ability to store and change data is fundamental to working with Python and building more complex programs.

Data types in Python

In Python, data types are classifications that categorize different kinds of data. Each data type represents a specific kind of value that can be stored and manipulated in a program. Understanding data types is crucial because it determines how data is stored in memory and what operations can be performed on that data.

Python is a dynamically typed language, meaning you don't need to declare the data type explicitly. Python infers the data type based on the value assigned to a variable.

Data types in Python

Here are some commonly used data types in Python:

1. Integer (int): Integers are whole numbers without decimal points. For example: 5, -3, 0.

2. Float: Floats are numbers with decimal points. They represent real numbers and can also include scientific notation. For example: 3.14, -2.5, 1.0e-5.

3. String (str): Strings are sequences of characters enclosed in single quotes (') or double quotes (""). They represent text or a combination of letters, digits, and symbols. For example: "Hello", 'Python', "123".

4. Boolean (bool): Booleans represent logical values and can be either True or False. They are often used for conditions and comparisons. For example: True, False.

Data types in Python

5. List: Lists are ordered collections of items enclosed in square brackets ([]). They can contain elements of different data types and are mutable, meaning their values can be changed. For example: [1, 2, 3], ["apple", "banana", "cherry"].

6. Tuple: Tuples are similar to lists but are enclosed in parentheses (()). Unlike lists, tuples are immutable, meaning their values cannot be changed after creation. For example: (1, 2, 3), ("red", "green", "blue").

7. Dictionary: Dictionaries are unordered collections of key-value pairs enclosed in curly braces ({}). Each value is associated with a unique key, allowing efficient retrieval of values. For example: {"name": "John", "age": 25}.

8. Set: Sets are unordered collections of unique elements enclosed in curly braces ({}). They are useful for tasks that involve checking membership or removing duplicates. For example: {1, 2, 3}, {"apple", "banana", "cherry"}.

Data types in Python

You can also use type() function to determine the data type of a variable. For example:

```
x = 5
print(type(x)) # Output: <class 'int'>
```

Understanding data types is essential because different data types have different properties and support different operations. For instance, arithmetic operations are applicable to numerical types (int, float), while string concatenation and indexing are specific to strings.

By utilizing appropriate data types, you can efficiently represent and manipulate data in your programs, ensuring the accuracy and integrity of your computations.

Remember, Python provides a rich set of built-in functions and libraries that allow you to work with various data types effectively.

UPPSALA UNIVERSITET

Comments in Python

In Python, comments are used to add explanatory notes or annotations within the code. They are not executed as part of the program and are solely meant for human readers to understand the code better. Comments play a crucial role in enhancing code readability, making it easier to maintain, debug, and collaborate on projects.

Comments in Python are denoted by the hash symbol (#) and are generally placed on a separate line or at the end of a line of code. Anything after the hash symbol is considered a comment and is ignored by the interpreter.

```
# This is a comment describing the purpose of the
# program and providing some high-level information.

# Variables representing the lengths of sides of a
triangle
side_a = 3
side_b = 4
side_c = 5

# Calculating the perimeter of the triangle
perimeter = side_a + side_b + side_c

# Printing the perimeter
print("The perimeter of the triangle is:", perimeter)
# This comment clarifies the output message.
```

Manipulating datatypes

Data types in Python - list indexing

In lists, values are accessed by their index position. The index represents the position of the value within the list or array, starting from 0 for the first element. To retrieve a value, you can use square brackets [] and provide the index of the desired element.

```
my_list = ['apple', 'banana', 'orange']
print(my_list[0])  # Output: apple
print(my_list[1])  # Output: banana
print(my_list[2])  # Output: orange
print(my_list[-1]) # Output: orange
```

Data types in Python - Dictionary indexing

Dictionaries use keys to access values. Each value in a dictionary is associated with a unique key, which can be any immutable data type like strings, numbers, or tuples. To retrieve a value from a dictionary, you can use square brackets [] and provide the key.

```
my_dict = {'name': 'John', 'age': 30, 'country': 'USA'}
print(my_dict['name'])     # Output: John
print(my_dict['age'])      # Output: 30
print(my_dict['country'])  # Output: USA
```

UPPSALA UNIVERSITET

Data types in Python - list slicing

List slicing allows you to extract a portion of a list by specifying a range of indices. This technique is useful when you want to work with a subset of the original list. In Python, list slicing is done using the colon : operator inside square brackets [].

To perform list slicing, you can specify the start and end indices separated by a colon : inside the square brackets. The start index is inclusive, meaning the element at that index is included in the slice, while the end index is exclusive, meaning the element at that index is not included in the slice. If the start index is omitted, it defaults to 0, and if the end index is omitted, it defaults to the length of the list.

Data

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Slice from index 2 to index 5 (exclusive)
slice1 = my_list[2:5]
print(slice1)  # Output: [3, 4, 5]

# Slice from the beginning up to index 6
(exclusive)
slice2 = my_list[:6]
print(slice2)  # Output: [1, 2, 3, 4, 5, 6]

# Slice from index 3 to the end
slice3 = my_list[3:]
print(slice3)  # Output: [4, 5, 6, 7, 8, 9, 10]
```

In this example, we have a list my_list containing numbers from 1 to 10. The list slicing syntax is the same as for arrays. We perform slicing operations to extract specific portions of the list. slice1 captures elements from index 2 to 4, slice2 includes elements from the beginning up to index 5, and slice3 contains elements from index 3 to the end of the list.

List slicing allows you to extract subsets of a list based on indices, making it easier to work with specific portions of the list. It's a convenient feature that enables you to manipulate and process lists efficiently in Python.

Variable assignment and comparison

In Python, the symbols "=" and "==" have distinct meanings when it comes to variables. The "=" symbol is used for variable assignment, where it assigns a value to a variable. It indicates that the variable on the left side is to be assigned the value on the right side. For example, x = 5 assigns the value 5 to the variable x.

On the other hand, the "==" symbol is used for variable comparison. It is an equality operator that checks if two values or expressions are equal. For instance, x == 5 compares the value of x to 5 and returns a boolean value: True if they are equal, and False if they are not. It's important to differentiate between variable assignment and variable comparison to avoid confusion and ensure that the intended operation is performed in your code.

Control flow

Control Flow - Conditional statements

Flow control in Python refers to the ability to dictate the order in which statements and instructions are executed within a program. It allows programmers to make decisions based on certain conditions and repeat actions as needed. Python provides various constructs for flow control, including conditional statements and loops. Conditional statements, such as if, else, and elif, enable the program to execute specific blocks of code based on whether certain conditions are true or false. This allows for branching and decision-making within the program's execution.

Loops, on the other hand, such as for and while loops, allow for the repeated execution of a block of code either for a specific number of iterations or as long as a certain condition remains true. These constructs provide flexibility and control over the program's behavior, enabling it to adapt and respond to different scenarios and conditions.

Control Flow - Conditional statements

The use of flow control is crucial in programming as it allows for dynamic and responsive execution of code. By incorporating conditional statements and loops, programmers can create more intelligent and interactive programs. Conditional statements provide the ability to handle different cases and make decisions based on specific conditions, while loops facilitate repetition and iteration, allowing for efficient processing of data and implementation of algorithms. Flow control plays a vital role in controlling the logical flow of a program, ensuring that the right actions are taken at the right time, and enabling the program to respond to user input, handle exceptions, and perform various computations. Mastery of flow control is essential for effective programming and is a fundamental skill that allows developers to create sophisticated and functional applications.

Conditional statements - if / elif / else

If statements allow for conditional execution of code based on whether certain conditions are true or false. This allows for branching and decision-making within the program's execution.

```
age = 25
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
>>>>>
You are an adult.
```

Conditional statements - for

For statements allow iteration over a sequence, such as a list, and perform a block of code for each item.

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
>>>>>
apple
banana
cherry
```

Conditional statements - while

While statements enable the repeated execution of a block of code as long as a condition remains true.

```
count = 0
while count < 5:
    print("Count:", count)
    count += 1
>>>>>
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
```

UPPSALA UNIVERSITET

Function calls and classes

Function calls

Functions in Python allow you to group and reuse blocks of code, promoting modularity and reusability. They are defined using the "def" keyword, followed by the function name, optional parameters in parentheses, and a colon. The code block inside the function is indented and contains the instructions to be executed. Functions can accept input values through parameters and return output values using the "return" statement. By encapsulating code into functions, you can break down complex problems into smaller, manageable tasks, enhance code organization and readability, and easily reuse code throughout your program.

Function calls

```
def greet(name):
    """A function that greets the user."""
    print("Hello, " + name + "!")

# Function call
greet("Alice")  # Output: Hello, Alice!
greet("Bob")  # Output: Hello, Bob!
```

In the above example, the greet function is defined with a parameter name that accepts an input value. When the function is called, it prints a greeting message along with the provided name. By defining the greet function, we can easily reuse it multiple times with different names, promoting code reusability and reducing redundancy.

Function calls - do not forget the return statement!

```python
def calculate_sum(a, b):
    """A function that calculates the sum of two numbers."""
    sum_result = a + b
    return sum_result

# Function call with correct usage of return statement
result = calculate_sum(3, 5)
print("Sum:", result)  # Output: Sum: 8

# Wrong function definition without using the return statement
def calculate_sum(a, b):
    """A function that calculates the sum of two numbers."""
    sum_result = a + b

result = calculate_sum(3, 5)
print("Sum:", result)  # Output: Sum: None
```

UPPSALA UNIVERSITET

Object-Oriented Programming in Python - Classes

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects, which are instances of classes. In OOP, a class is a blueprint or template that defines the characteristics and behavior of objects. The attributes of a class represent the data associated with the objects, while the methods define the actions or operations that can be performed on those objects. By encapsulating data and related functionality within a class, OOP promotes code organization, modularity, and reusability.

In Python, classes are defined using the class keyword, followed by the class name. The attributes are defined inside the class as variables, and methods are defined as functions. To create an object (instance) of a class, you can simply call the class as if it were a function, which then creates a new object with its own set of attributes. You can access the attributes and invoke the methods of an object using dot notation. This allows you to manipulate and interact with objects based on the defined class blueprint.

Object-Oriented Programming in Python - Classes

```python
class Rectangle:
    """A class representing a rectangle."""

    def __init__(self, length, width):
        self.length = length
        self.width = width

    def calculate_area(self):
        return self.length * self.width

# Creating objects (instances) of the Rectangle class
rect1 = Rectangle(4, 6)
rect2 = Rectangle(3, 5)

# Accessing object attributes and invoking methods
print("Rectangle 1 area:", rect1.calculate_area())  # Output: Rectangle 1 area: 24
print("Rectangle 2 area:", rect2.calculate_area())  # Output: Rectangle 2 area: 15
```

UPPSALA UNIVERSITET

Object-Oriented Programming in Python - Classes

In the previous example, the Rectangle class is defined with attributes length and width, and a method calculate_area() that calculates the area of a rectangle. Two objects, rect1 and rect2, are created as instances of the Rectangle class. The attributes of each object hold specific values, and the calculate_area() method is invoked on each object to calculate and print the respective areas.

This simplified example demonstrates the basic usage of classes, objects, attributes, and methods in Python. It showcases how OOP allows you to define reusable blueprints (classes) and create multiple instances (objects) based on those blueprints, each with its own set of data (attributes) and behavior (methods).

Modules and libraries

Python Modules and Libraries

In Python, modules are files containing Python code that can be imported and used in other programs. They help organize and reuse code by grouping related functionality together. To use a module, you import it using the import statement and access its functions, classes, and variables using dot notation.

Libraries in Python are collections of modules that offer specialized functionality for specific purposes. They extend the capabilities of Python beyond its built-in features. Some commonly used libraries include math for mathematical operations, random for generating random values, and datetime for working with dates and times.

Python Modules and Libraries

```
import math
import random
import datetime

# Using the math module
square_root = math.sqrt(25)
print("Square root of 25:", square_root)  # Output: Square root of 25: 5.0

# Using the random module
random_number = random.randint(1, 10)
print("Random number between 1 and 10:", random_number)  # Output: Random number between 1 and 10: <random
value>

# Using the datetime module
current_time = datetime.datetime.now()
print("Current time:", current_time)  # Output: Current time: <current date and time>
```

UPPSALA UNIVERSITET

Python Modules and Libraries

In the example above, the math module is imported to calculate the square root of a number using the sqrt() function. The random module is used to generate a random number between 1 and 10 using the randint() function. The datetime module is employed to obtain the current date and time using the now() function.

By leveraging modules and libraries in Python, you can easily access and utilize additional functionality, saving development time and effort. They offer a wide range of ready-made solutions for common tasks and enable you to extend the capabilities of Python to suit your specific needs.

If there is anything new you want to get done, odds are someone has already built a library for that task!

Important libraries - NumPy

NumPy is a fundamental library for scientific computing in Python. It provides powerful numerical operations and multidimensional array objects, along with a collection of mathematical functions. NumPy is widely used for data manipulation, numerical computations, and working with large datasets efficiently. Its array-based approach allows for vectorized operations, which can significantly improve performance.

Important libraries - NumPy

```
import numpy as np

# Create a 1D array
arr = np.array([1, 2, 3, 4, 5])

# Perform operations on the array
arr_squared = arr ** 2
arr_sum = np.sum(arr)

# Perform matrix multiplication
matrix_a = np.array([[1, 2], [3, 4]])
matrix_b = np.array([[5, 6], [7, 8]])
matrix_result = np.dot(matrix_a, matrix_b)

# Generate random numbers
random_array = np.random.rand(5)
```

In this example, we import numpy as np and demonstrate some of its capabilities. We create a 1D array, perform mathematical operations on it, calculate the sum of its elements, and show an example of matrix multiplication using the dot() function. Finally, we generate a 1D array of random numbers using numpy.random.rand().

UPPSALA UNIVERSITET

Important libraries - pandas

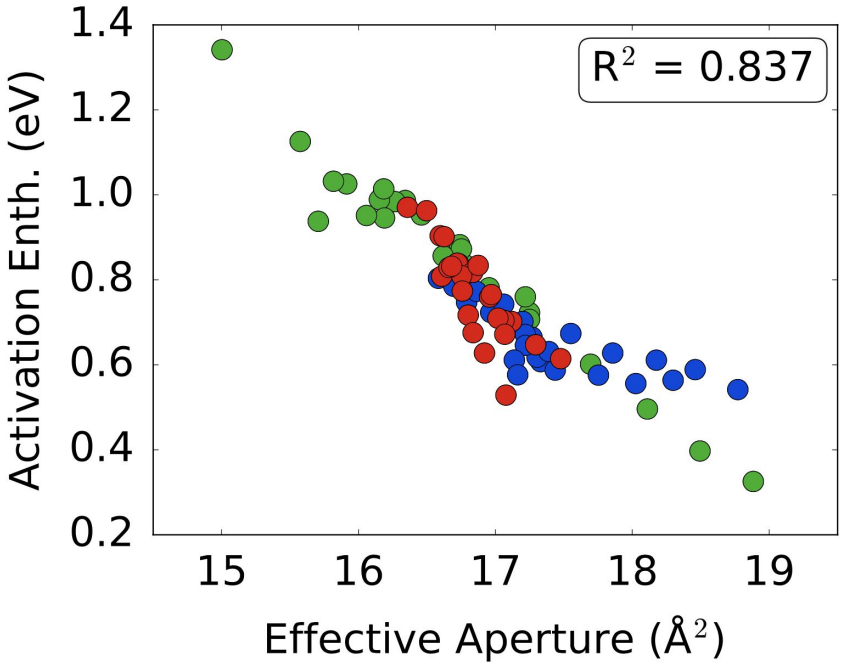
Pandas is a powerful library for data manipulation, analysis, and cleaning. It provides data structures like DataFrame and Series that allow for efficient handling and analysis of tabular data. Pandas enables tasks such as data reading from various file formats, data filtering, grouping, merging, and visualization.

```
import pandas as pd

# Read data from a CSV file
data = pd.read_csv('data.csv')

# Display the first few rows of the DataFrame
print(data.head())

# Filter data based on conditions
filtered_data = data[data['Age'] > 30]

# Group data by a specific column and calculate summary
statistics
grouped_data = data.groupby('Category')['Value'].mean()

# Merge multiple DataFrames based on a common column
merged_data = pd.merge(df1, df2, on='ID')

# Plot data using Pandas' built-in plotting capabilities
data.plot(x='Date', y='Value', kind='line')
```

In this example, we import pandas as pd and demonstrate various Pandas operations. We read data from a CSV file using read_csv(), display the first few rows of the DataFrame using head(), filter data based on a condition, calculate summary statistics using groupby(), merge multiple DataFrames using merge(), and plot data using Pandas' built-in plotting capabilities.

Important libraries - matplotlib

Matplotlib is a popular plotting library in Python that provides a wide range of visualization options. It allows you to create high-quality figures, charts, and plots for data exploration and presentation. Matplotlib provides a comprehensive set of functions and customization options for creating line plots, scatter plots, bar plots, histograms, and more.



UPPSALA UNIVERSITET

File handling

File handling in Python

File handling in Python involves working with files on your computer's file system. It allows you to read data from files or write data to files, providing a way to interact with external data sources. Python's file handling mechanisms make it straightforward to perform tasks such as reading the contents of a file, writing data to a file, or modifying existing files.

To work with files in Python, you need to follow a few steps. First, you need to open the file using the open() function, which takes the file name and the mode as arguments. The mode specifies the purpose of opening the file, such as reading, writing, or appending data. Once the file is opened, you can perform operations like reading data from the file or writing data to it. After you're done with the file, it's important to close it using the close() method to free up system resources.

File handling in Python

```
# Opening a file in write mode
file = open("sample.txt", "w")
file.write("Hello, World!")
file.close()

# Opening a file in read mode
file = open("sample.txt", "r")
data = file.read()
print(data)  # Output: Hello, World!
file.close()
```

File handling in Python

In the example above, the file "sample.txt" is opened in write mode using the open() function with the mode parameter set to "w". The write() method is used to write the text "Hello, World!" to the file. After writing, the file is closed using the close() method.

Then, the same file is opened again, but this time in read mode ("r"). The read() method is used to read the contents of the file into the data variable. Finally, the contents of the file are printed, resulting in the output "Hello, World!".

It's worth mentioning that file handling operations can potentially encounter errors, such as if the file doesn't exist, or if there are issues with file permissions. It's good practice to handle these potential errors using error handling mechanisms like try-except blocks to gracefully handle any exceptions that might occur during file handling operations.

File handling in Python - .csv files

File handling in Python allows you to easily read data from a .csv (comma separated) file and perform various operations on it. To read from a .csv file, you can use the built-in csv module in Python. First, you need to open the file using the open() function, specifying the file path and the mode ('r' for reading). Once the file is opened, you can create a csv.reader object by passing the file object to it.

This reader object provides useful methods for reading the data from the file. For example, you can use the next() method to skip the header row. Additionally, you can specify options like the delimiter character, quoting style, and handling of whitespace using the appropriate parameters when creating the reader object. For instance, you can use csv.reader(file_object, delimiter=',', quotechar='"', skipinitialspace=True) to read a comma-separated .csv file with double quotes around fields and skip initial whitespaces. This gives you flexibility in handling different file formats and data structures.

File handling in Python - .csv files

```
import csv

# Open the .csv file
with open('data.csv', 'r') as file:
    # Create a csv.reader object
    reader = csv.reader(file, delimiter=',',
      quotechar='"', skipinitialspace=True)

    # Skip the header row
    header = next(reader)

    # Read the remaining rows
    for row in reader:
        # Process the data
        print(row)
```

- In this example, we open the 'data.csv' file in read mode and create a csv.reader object named reader.
- We specify the delimiter as a comma (,), the quote character as a double quote ("), and set skipinitialspace to True to remove any leading whitespace.
- The next(reader) statement skips the header row, and then we iterate over the remaining rows using a for loop, printing each row as an example of processing the data.

File handling in Python - excel (.xslx) files

Reading data from an Excel file in Python is convenient and can be accomplished using the pandas library. First, you need to install pandas if you haven't already done so, using the command pip install pandas. Once pandas is installed, you can use the read_excel() function to read data from an Excel file. Specify the file path as the input parameter, and pandas will read the contents of the Excel file and return a DataFrame, which is a tabular data structure.

Excel files

```
import pandas as pd

# Read data from Excel file
df = pd.read_excel('data.xlsx',
    sheet_name='Sheet1')

# Print a subset of columns and
    first 5 rows of the DataFrame
subset_columns = ['Column1', 'Column3',
    'Column5']
print(df[subset_columns].head(5))
```

In this example, we use the read_excel() function from pandas to read data from the 'data.xlsx' file. The sheet_name parameter is used to specify the sheet from which to read the data. You can provide the sheet name as a string or the sheet index (starting from 0). The resulting DataFrame, stored in the variable df, contains the data from the specified sheet. You can then perform various operations on the DataFrame to filter the data as shown in this example.

Instead of displaying all columns, we create a list called subset_columns that contains specific column names or labels from the DataFrame that you want to display. By passing this list as a selection to the DataFrame (df[subset_columns]), we limit the output to only those columns.

Additionally, we've set the argument of 5 to the head() function, which displays the first 5 rows of the resulting subset DataFrame. You can modify the number of rows or adjust the column selection (subset_columns) as per your requirements.

UPPSALA UNIVERSITET

Error handling

Error handling

Error handling in Python is a crucial aspect of writing robust and reliable programs. It allows you to anticipate and handle potential errors or exceptions that may occur during the execution of your code. Python provides a try-except block structure that allows you to catch and handle exceptions gracefully, preventing your program from crashing and providing a way to respond to errors in a controlled manner.

The try-except block consists of the try statement, which encloses the code that might raise an exception, and one or more except statements that define how to handle specific types of exceptions. When an exception occurs within the try block, Python searches for a matching except block. If a matching except block is found, the code within that block is executed. If no matching except block is found, the exception is propagated to the surrounding code or the program terminates.

Error handling

```
try:
    numerator = 10
    denominator = 0
    result = numerator / denominator
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Division by zero!")
```

In this example, the code within the try block attempts to perform a division operation where the denominator is set to 0. This operation would normally raise a ZeroDivisionError. However, by surrounding the code with a try-except block and specifying the ZeroDivisionError in the except block, we can handle this error gracefully. Instead of crashing the program, the except block is executed, printing the error message "Error: Division by zero!".

Debugging

Debugging is an essential skill for developers, as it helps identify and fix issues in code. When debugging, it's important to follow some best practices to efficiently track down and resolve problems. One common technique is to use print statements strategically placed throughout your code. By inserting print statements at key points, you can check the intermediate values of variables, track the flow of execution, and identify any unexpected behavior. Print statements allow you to observe the actual output of your code, helping you understand how it's behaving and pinpointing any discrepancies between the expected and actual results. By strategically placing print statements and carefully examining their output, you can gain valuable insights into your code's execution and identify areas that need attention. However, it's crucial to remove or disable the debug print statements once you've resolved the issue, as leaving them in your code can clutter the output and impact performance. Remember, print statements are a simple yet powerful tool in your debugging arsenal, helping you gain visibility into your code's behavior and facilitating the debugging process.

Debugging

```
def calculate_average(numbers):
    total = 0
    count = 0

    for num in numbers:
        total += num
        count += 1
        print("Number:", num)
        print("Total:", total)
        print("Count:", count)

    average = total / count
    return average
```

By adding print statements within the for loop, we can see the values of num, total, and count at each iteration. This allows us to verify if the calculations are being performed correctly. It also helps in identifying any potential errors or unexpected behavior.

When executing the code, the print statements will display the intermediate values in the console, providing insights into the execution flow. After debugging and confirming that the function is working as expected, it's a good practice to remove or disable the print statements to avoid cluttering the output unnecessarily.

Using print statements strategically throughout your code allows you to gain visibility into the values of variables and track the execution flow, making it easier to identify issues and understand how your code is behaving.

Getting started with Python

Running Python in a Jupyter Notebook

Jupyter Notebooks are interactive computing environments that allow you to create and share documents containing live code, equations, visualizations, and explanatory text. The notebooks are organized into cells, where each cell can contain code, markdown text, or other multimedia content. Jupyter Notebooks support multiple programming languages, including Python, R, and Julia, making them versatile for data analysis, machine learning, and scientific research.

In a Jupyter Notebook, you can write and execute code in individual cells. To run a code cell, simply click on it and press Shift + Enter. The output of the code will be displayed directly below the cell. You can edit and rerun cells as many times as needed, allowing for an interactive and iterative coding process. Additionally, you can include markdown cells to add formatted text, images, equations, and other media to provide explanations or document your analysis. Jupyter Notebooks also support the installation of additional libraries and extensions to enhance functionality and customize the environment. Once you have completed your notebook, you can save it as a .ipynb file or export it to various formats such as HTML or PDF for easy sharing and collaboration with others.

Running Python locally

- Install Anaconda: Download and install Anaconda from the official website (https://www.anaconda.com/products/individual) based on your operating system. Follow the installation instructions.
- Launch Anaconda Navigator: Open Anaconda Navigator, the graphical user interface (GUI) provided by Anaconda for managing environments, packages, and applications.
- Create a New Environment: In Anaconda Navigator, go to the "Environments" tab and click "Create" to create a new Python environment. Specify the Python version and desired packages.
- Launch Jupyter Notebook: In the Anaconda Navigator "Home" tab, locate your created environment and click "Launch" to start Jupyter Notebook. A new browser window will open with the Jupyter interface.

Running Python in the cloud - Google Colab

1. Access Colab: Go to https://colab.research.google.com/ and sign in with your Google account.
2. Create a New Notebook: Click "New Notebook" in the Colab interface to open a Jupyter-like environment for Python code.
3. Write Code: Edit code cells and press Shift + Enter to run the code. Colab supports Python 2 and 3, and you can add and run cells as needed.
4. Utilize Features: Colab offers free GPU access, pre-installed libraries, and collaboration capabilities. Switch runtime options, add text cells, and mount Google Drive to access data files.

Managing Python environments using Anaconda

Using Anaconda as a package manager

Anaconda is a popular distribution of Python that includes a comprehensive collection of scientific computing libraries and tools. It simplifies the process of managing Python packages and creating isolated environments for different projects. Anaconda comes with its own package manager called Conda, which enables easy installation, updating, and removal of packages. With Conda, you can manage both Python packages and non-Python dependencies, making it a powerful tool for data scientists, researchers, and developers.

Using Anaconda as a package manager

Best practices for using Anaconda and Conda:

Create separate conda environments: One of the best practices when using Anaconda is to create separate conda environments for different tasks or projects. Conda environments allow you to have isolated environments with their own set of packages and dependencies. This helps in avoiding conflicts between different projects and ensures reproducibility. You can create a new environment using the command conda create --name myenv, where myenv is the name of your environment.

Using Anaconda as a package manager

Activate and deactivate environments: Once you have created a conda environment, you can activate it to start using it. Activating an environment modifies the system's PATH variable to use the packages and dependencies specific to that environment. To activate an environment, you can use the command conda activate myenv, where myenv is the name of your environment. To deactivate an environment, simply use the command conda deactivate.

Install packages: With Anaconda and Conda, you can easily install packages from the official Anaconda repositories or other channels. Use the command conda install packagename to install a specific package. You can also specify the version of the package, for example, conda install packagename=1.0.

Using Anaconda as a package manager

Manage environments with environment files: To share your conda environment with others or replicate it on another machine, it is recommended to use environment files. An environment file contains a list of all the packages and dependencies in your environment. You can create an environment file using the command conda env export > environment.yml. To recreate the environment from the file, someone else can use the command conda env create -f environment.yml.

Update and remove packages: Conda makes it easy to update and remove packages. To update a package, use the command conda update packagename. To remove a package, use conda remove packagename. You can also specify the version of the package to update or remove.

By following these best practices, you can effectively manage your Python packages, create isolated environments, and ensure reproducibility across different projects and tasks using Anaconda and Conda.

Combining conda and pip

While Conda provides a wide range of packages through the Anaconda repositories, there may be instances where you require a package that is not available in those repositories.

In such cases, you can turn to pip, the default package manager for Python. Pip has a larger package ecosystem and may have the specific package you need. However, it's generally recommended to use Conda as the primary package manager to maintain a consistent environment and take advantage of Conda's dependency resolution capabilities.

To install a package using pip within a Conda environment, follow these steps:

1. Activate your Conda environment using the command conda activate myenv, where myenv is the name of your environment.
2. Use the command pip install packagename to install the desired package via pip. Conda will handle the installation process and ensure that the package's dependencies do not conflict with the Conda-managed packages in your environment.

By combining Conda with pip, you can leverage the strengths of both package managers. Conda manages the installation and resolution of packages within the Conda environment, while pip expands your package options by tapping into the broader Python package ecosystem.

However, it's important to exercise caution when using pip within a Conda environment to avoid potential conflicts. Installing packages with pip that are already available through Conda repositories may result in redundant installations and potential compatibility issues. It's best to search for packages in the Anaconda repositories first and resort to pip only when necessary. Ideally, all packages that are available through conda should be installed first, after which the remaining packages can be installed with pip, so as to minimise compatibility issues.

Additionally, it's advisable to update Conda and the Conda-managed packages regularly to maintain compatibility and benefit from bug fixes and security patches. You can update Conda using conda update conda and update all packages in the environment using conda update --all.

Remember to refer to the official Conda and pip documentation for detailed instructions and recommendations on managing packages with these tools.

Version control using Git

What is Git?

- Git is a version control system that helps you keep track of changes made to files in a project.
    - It allows you to save different versions of your work, so you can go back and forth between them.

- Developers can review project history to find out:
    - Which changes were made?
    - Who made the changes?
    - When were the changes made?
    - Why were changes needed?

How to use Git - working with repositories

A repository, often abbreviated as "repo," is a central place where you can store and organize your files and folders related to a project. It serves as a version control system that tracks changes made to your project over time. Here's a general introduction on repositories and how to use them properly.

Properly using repositories involves maintaining good version control practices, such as committing regularly, writing descriptive commit messages, and collaborating effectively with others. It's also important to ensure that your repositories are organized, with clear directory structures and logical file naming conventions.

By leveraging repositories and version control systems, you can effectively manage and track changes to your projects, collaborate with others, and maintain a reliable history of your work.

How to use Git - working with repositories II

1. Creating a Repository: To create a repository, you typically start by initializing it with a version control system like Git. This initializes an empty repository in a specific directory on your local machine. Alternatively, you can create a repository on a hosting platform like GitHub, GitLab, or Bitbucket.
2. Adding Files: Once you have a repository, you can start adding files and folders to it. These files can be source code, documents, images, or any other files relevant to your project. Adding files to your repository allows Git to track changes made to those files over time.
3. Committing Changes: As you work on your project, you'll make modifications to the files in your repository. Git provides a way to group related changes together through commits. A commit is like a snapshot that records the state of your files at a specific point in time. It's important to create meaningful commit messages that describe the changes you made.

How to use Git - working with repositories III

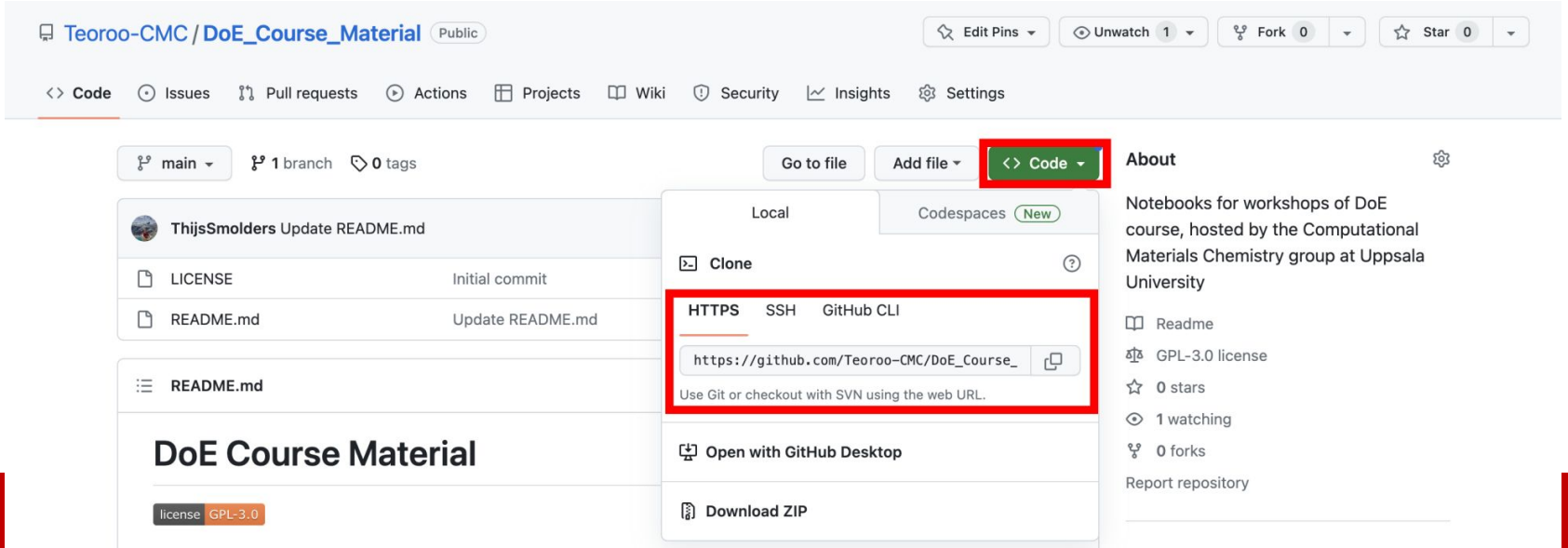
4. Branching: Repositories also support branching, which allows you to create separate lines of development. Branches are useful when you want to work on a new feature or experiment with changes without affecting the main project. You can create, switch between, and merge branches to manage different aspects of your project's development.

5. Pushing and Pulling: When you're working with a remote repository, such as on a hosting platform, you'll use commands like git push and git pull to synchronize your local repository with the remote version. Pushing sends your local commits to the remote repository, while pulling fetches changes from the remote repository and merges them with your local version.

How to use Git?

- Typically you will work on a local copy (on your own pc/laptop) of a remote repository (hosted on github.com)
- First you clone the parent repo

  git clone https://XXXXXXX



How to use Git?

- To update the remote repository with local changes, you go through a series of three steps:
  - git add
  - git commit
  - git push

```
(base) tjams20@C02XD13DJHD2 DoE_Course_Material % git add Week_1/Workshop_1/README.md
(base) tjams20@C02XD13DJHD2 DoE_Course_Material % git commit -m "docs: update README.md on Git usage"
[main 0569355] docs: update README.md on Git usage
 1 file changed, 1 insertion(+), 1 deletion(-)
(base) tjams20@C02XD13DJHD2 DoE_Course_Material % git push
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 436 bytes | 436.00 KiB/s, done.
Total 5 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/Teoroo-CMC/DoE_Course_Material.git
   07cee2e..0569355  main -> main
(base) tjams20@C02XD13DJHD2 DoE_Course_Material %
```

How to use Git?

To get changes from the repository into your local copy, you can use either git pull or git fetch in the local directory.

```
(base) tjams20@C02XD13DJHD2 DoE_Course_Material % git pull
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), 739 bytes | 82.00 KiB/s, done.
From https://github.com/Teoroo-CMC/DoE_Course_Material
   c47d80b..104dcaf  main       -> origin/main
Updating c47d80b..104dcaf
Fast-forward
 Week_1/README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
(base) tjams20@C02XD13DJHD2 DoE_Course_Material %
```