Functions and I/O

August 2024

Data Science Session Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: **Questions**

Data Science Session Housekeeping cont.

- For all **non-academic questions**, please submit a query: **www.hyperiondev.com/support**
- Report a **safeguarding** incident: **www.hyperiondev.com/safeguardreporting**
- We would love your **feedback** on lectures: **Feedback on Lectures**

Learning Outcomes

- **Apply** user-defined functions in Python to automate repetitive tasks.
- **Explain** the concepts of function scope, parameters, and arguments in Python.
- **Create** and **handle** files using Python's I/O functions, including reading from and writing to files, with error management.

HyperionDev

Problem Statement

Consider the problems we have looked at so far. In the Teacher's Pet program, we made use of a lot of repeated or recycled code because we were trying to accomplish the same thing in multiple places in our program.

- Since we try to avoid repetition as much as possible, and we want to prioritise well-organised and easily modifiable code, how can we avoid this?

Problem Statement

In Data Science, we will often encounter programs that calls for intake of raw data, data analysis and output of processed results. So far, we could only get input from a user via the terminal and we produce output to the terminal.

- How can we accept input via other sources?
- Can we produce output to these sources as well?
- How can we read and alter existing text files, and create new ones?

Lecture Overview

- Functions
  - Parts of a Function
  - Built-in
  - User-defined
  - Scope
- I/O
  - Reading
  - Writing

HyperionDev

Functions

HyperionDev

Functions

A block of organised, reusable code that accomplishes a specific task.

- A function can be **called repeatedly** throughout your code.
- Functions can either be **user-defined** or **built-in**.
- This helps us **minimise repeating lines of code** unnecessarily.
- The main benefits of using functions are:
  - It improves code **modularity, management** and **maintenance**.
  - It makes our code more **readable**.
  - It **reduces potential errors**.

HyperionDev

Functions

- We've already used some functions in our code so far:
  - **print("Message"):** outputs message
  - **input("Message to user"):** Displays message to user, returns input
  - **type(value):** returns the type of our value
  - **int(value):** converts our value to an Integer
  - **float(value):** converts our value to a Float
  - **bool(value):** converts our value to a Boolean
  - **str(value):** converts our value to a String
  - **range(start, stop, step):** returns a list of numbers, starting at *start* and stopping before *stop* and using a *step* between the numbers

  - See more in the function dictionary...

Functions

- To declare a function in Python, we use the **def** keyword.
- We have to provide a **name** for our function (using variable naming conventions), a list of **parameters** (placeholders for function inputs) in brackets, a colon and **body** of the function indented.
- We also need to add a **return statement** for functions that return a value. This is not necessary for all functions e.g. functions that modify a state.

```
# Syntax of a user-defined function
def functionName(parameter1, parameter2):
    # function block containing statements
    # which accomplishes a specific task
    result = "Output"
    return result
```

HyperionDev

Functions

- After defining a function, we **call or invoke** it to use it in our code.
- We call a function with its name followed by a list of **arguments** enclosed in brackets, if required by the functions.
- **Arguments** are the input values provided to the function and take the place of the **parameters** defined in the function in the **same position**.

```
# Function which calculates the sum of two numbers
def calculateSum(a, b):
    return a + b


sum1 = calculateSum(800982390, 247332) # 801229722
sum2 = calculateSum(sum1, 3)   # 801229725
```

HyperionDev

Scope

The area of visibility and accessibility of a variable in a program.

- The **scope** of a variable determines **where in the code it can be seen**.
- Functions in Python have **local scope**, meaning variables declared **inside a function** are only **accessible within** that function.
- Variables declared outside of a function, known as **global variables**, can be accessed anywhere.
- Python has different **types of scope:**
  - Global Scope
  - Local Scope
  - Block Scope

HyperionDev

Scope

- **Global scope:** Variables defined outside of any function, accessible throughout the entire program.
- **Local scope:** Variables defined within a function, only accessible within that function.
- **Block scope:** Some languages have block scope, where variables defined within a block (e.g., within an `if` statement or a loop) are only accessible within that block.

Input/Output

HyperionDev

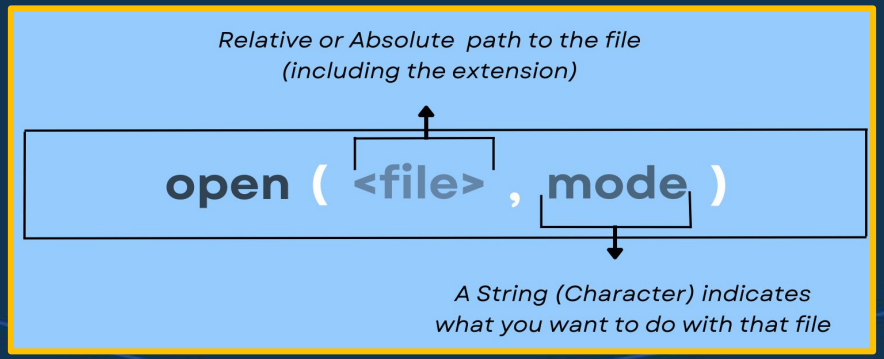
File I/O

Process of reading data from files (input) or writing data to files (output) using a computer program.

- **File I/O** stands for **Input/Output** operations involving files.
- File I/O is all about your program **interacting with files**:
  - Taking in information from them
  - Putting information into them.
- **File handling** in Python refers to the process of **working with files** on your **computer's storage**.
- Python provides built-in functions and methods for performing various file operations.

File Handling

- In Python, file I/O operations are performed using the **open() function** and various **methods** associated with **file objects**.
- The open() function is used to **open a file** and **returns a file object**.
- You specify the **file name/path** and the **mode** in which you want to open the file ('r' for reading, 'w' for writing, 'a' for appending, etc.).



HyperionDev

File Modes

Mode	Description
'r'	Opens a file for reading.
'w'	Open a file for writing. If file does not exist, it creates a new file. If file exists it truncates the file.
'a'	Open a file in append mode. If file does not exist, it creates a new file.
'+'	Open a file for reading and writing (updating)

File Modes

- **Read** text from a file with the **mode 'r'**

```
file = open('file.txt', 'r')
file.read()
```

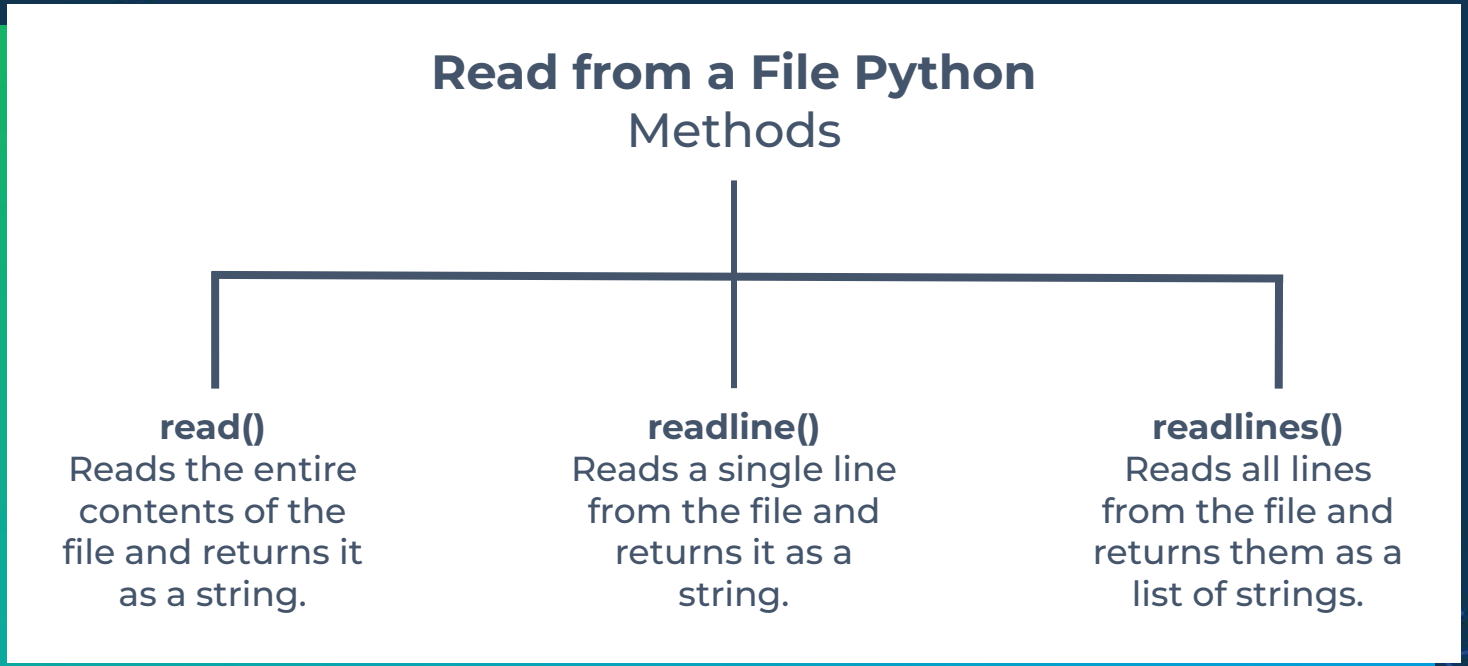
- **Write** text to a file with the **mode 'w'**

```
file = open('file.txt', 'w')
file.write("Hello World!")
```

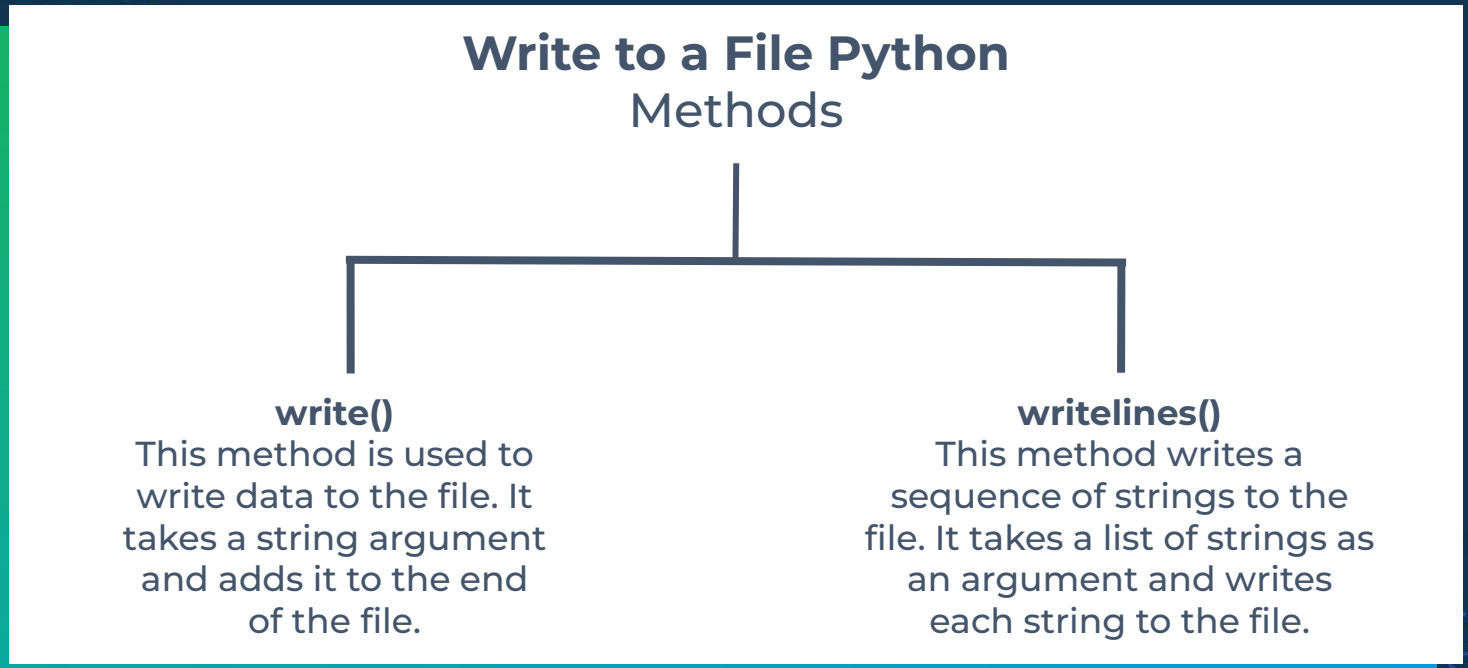
- **Append** text to an existing file with the **mode 'a'**

```
file = open('file.txt', 'a')
file.write("\nThis is a new line.")
```

File Handling (Reading)



File Handling (Writing)



Resource Management

- When we create a new file object in our programs, we have to be mindful of the resource we have created and that we properly clean them up after we are done with them.
- **Implicitly using the with statement:**
  - Ensures that resources are properly cleaned up after use, even if an error occurs.
- **Explicitly with the close() method:**
  - Involves manually opening and closing files using the open() function for opening and the close() method for closing.

Resource Management

```
# Creating and destroying a file object
# Implicitly using with statement
with open('filename.txt', 'r') as file:
    content = file.read()

# Explicitly using open and close
file = open('filename.txt', 'r')
content = file.read()
file.close()
```

HyperionDev

Try / Except

- You can wrap file I/O operations inside a try block and catch specific exceptions using except blocks.

```
try:
    with open('file.txt', 'r') as file:
        content = file.read()
    print(content)
except FileNotFoundError:
    print("File not found!")
except PermissionError:
    print("Permission denied to open the file!")
except IOError as e:
    print(f"An I/O error occurred: {e}")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

HyperionDev

Finally

- We can also use a finally block to ensure that certain cleanup actions are always performed, regardless of whether an exception occurred or not.

```
try:
    file = open('file.txt', 'r')
    content = file.read()
    print(content)
except FileNotFoundError:
    print("File not found!")
finally:
    file.close()  # Ensure file is closed even if an exception occurs
```

HyperionDev

Questions and Answers

HyperionDev

Thank you for attending

HyperionDev