Hyperiondev

Task

IO Operations

Visit our website

Introduction

Until now, the Python code you've been writing has only received input in one manner and has only displayed output in one way: You type input using the keyboard and its results are displayed on the console. But what if you want to read information from a file on your computer and write that information to another file? This process is called file **I/O** (the "I/O" stands for "input/output"), and Python has some built-in functions that handle this for you.

OPENING FILES

Files are an important source of information in Python, so let's explore the most common methods for opening text files (`.txt`) using Python.

There are two methods for opening a file in Python, and both methods use a built-in function called `open()` that accepts two arguments:

1. The first argument is the filename (including the path, if necessary) that specifies the file to be accessed, e.g., `example.txt`.
2. The second argument is the mode or access modifier, which determines how the file will be accessed, such as '`r+`' for reading and writing mode.

Now that we understand the arguments required to open a file, let's dive into the two methods of opening a file:

1. Using the `open()` function:

   This method involves simply calling the `open()` function and performing operations on the file. It's important to note that after you finish working with the file, you should explicitly close it using the `close()` method to release the system resources:

   ```
   file = open("example.txt", "r+")
   # Perform operations on the open file
   file.close()
   ```

   When you open a file, the operating system allocates resources to manage this file, such as memory for the file's contents and file descriptors to keep track of the file. If you don't close the file using the `close()` method these resources remain allocated, which can lead to resource leaks and eventually exhaust the system's capacity to manage files. This is why it's crucial to close files when you're done working with them.

2. Using the `open()` function with the `with` and `as` statements:

   This method of opening and managing a file in Python is referred to as using a **context manager**. It leverages the `with` statement, which ensures that resources are properly managed, and **automatically closes the file** once the block of code within the `with` statement is exited.

   ```
   with open('example.txt', 'r+') as file:
       # Perform file operations on the open file.
       # The code must be indented to be part of the 'with' block.
   ```

   The `with` block ensures that the file is properly closed after its code finishes, even if an error occurs. This is a safer and cleaner way to handle files because it manages resource allocation and deallocation automatically. Using a context manager for file handling in Python is a recommended and 'pythonic' approach, enhancing error handling, reducing resource leaks, and improving code readability by eliminating the need for explicit file closure.

Now, let's review the different modes available when opening a file in Python. The mode is specified as the second argument in the `open()` function and determines how the file will be accessed.

Here are the commonly used modes:

Mode	Description
`r`	Opens the file for **reading only**. An I/O error is thrown if the file doesn't exist.
`r+`	Opens the file for both **reading and writing**. An I/O error is thrown if the file doesn't exist.
`w`	Opens the file for **writing only**. It creates a new file if it doesn't exist, and if the file already exists, the previous content is overwritten.
`w+`	Opens the file for **reading and writing**. It creates a new file if it doesn't exist, and if the file already exists, the previous content is overwritten.
`a`	Opens the file for **writing only**, **creating it** if it doesn't exist. Any data written will be **appended** (thus `a`) at the end of the file, preserving the existing content.
`a+`	Opens the file for **reading and writing**, **creating it** if it doesn't exist. Any data written will be **appended** (thus `a`) at the end of the file, preserving the existing content.

These modes provide flexibility for various file operations, such as read-only access, read-write access with or without overwriting existing content, and appending data to the end of the file. Choose the appropriate mode based on your specific needs when opening a file.

READING FILES

There are four common approaches when reading files in Python. In the examples below, we will use the **`with`** statement context manager:

1. Using the **`read()`** method:

   The **`read()`** method is used to read the entire contents of a text file and then return the file contents as a string:

   ```
   with open("example.txt", "r+") as file:
       # 'lines' will contain a string with all the data from the
   file.
       lines = file.read()  # Reads all data from the file.
   ```

   The code in the example above reads the entire file. Alternatively, you can specify the number of characters to read by passing an integer argument to the **`read()`** method:

   ```
   with open("example.txt", "r+") as file:
       lines = file.read(10)  # Reads 10 characters from the file.
   ```

2. Using the **`readline()`** method:

   The **`readline()`** method is used to read a single line from a text file. It stops when it encounters a newline character (**`\n`**) and returns the line as a string:

   ```
   with open("example.txt", "r+") as file:
       line = file.readline()  # Reads a single line from the file.
   ```

3. Using the `readlines()` method:

   The `readlines()` method is used to read the contents of a text file **line by line** and returns them as a list of strings. Each line in the file becomes an element in the list:

```
with open("example.txt", "r+") as file:
    # 'lines' contains each line of the text file as a list element.
    lines = file.readlines() # Reads each line of data in the file
```

   The key distinction between the two methods is that `readline()` reads one line at a time and returns it as a string, while `readlines()` reads all lines and returns them as a list of strings.

4. Looping through the file:

   This approach involves looping through the file line by line and performing operations on each line:

```
with open("example.txt", "r+") as file:
    for line in file:
        # Reads each line of data in the file
        print(line)  # Displays each line within 'example.txt'.
```

   Let's take a look at how we can apply the looping method to perform operations on a line in the file:

```
with open("example.txt", "r+") as file:
    # Iterate through each line in the file
    for line in file:
        # Print the entire line
        print("The entire line is: " + line)

        # Print the first character of the line
        print("The first character of this line is: " + line[0])
```

We could build up all the lines of the text file into one large string called **`contents`** as follows:

```
contents = ""  # Initialise an empty string to store the contents

with open("example.txt", "r+") as file:
    # Open the file and iterate through each line
    for line in file:
        contents = contents + line  # Append each line to 'contents'

# Print the contents outside the 'with' statement
print(contents)
```

We've successfully retrieved the contents from the text file and stored it within our program using a variable named **`contents`**. That's pretty powerful because it allows us to directly manipulate the file's data. For example, we can then print the contents to a screen with **`print(contents)`**. Additionally, an alternative approach is to store each **`line`** in a list by appending it to an initially empty list, providing flexibility in how we handle and process the file's data.

```
lines = []  # Initialise an empty list to store each line

with open("example.txt", "r+") as file:
    # Open the file and iterate through each line
    for line in file:
        lines.append(line)  # Append each line to the 'lines' list

# Print the lines stored in the list
print(lines)
```

WRITING DATA TO A TEXT FILE

Now, let's see how to create a new text file and write data to it using the `w` access mode:

```
name = input("Enter name: ")

with open("output.txt", "w") as f:
    f.write(name + "\n")
```

In the example above, we first prompt the user to enter their name. Once the user inputs their name, it is stored as a string in the variable `name`. Following this, we create a new file named **output.txt** in write mode. If this file does not already exist, Python automatically creates it in the directory where our program is located. Using the `write()` method, we then write the value stored in the name variable along with a newline character (`\n`) to the newly created file. This newline character is necessary to ensure that each subsequent write operation starts on a new line within the file.

To create the file **output.txt** with the output generated by this program, you need to run this Python file. After running the file, you will be prompted to enter a name. Once you enter the name, the file will be opened, and the entered name will be written to **output.txt**.

We can continue writing to the file within the same `with` statement block. Any additional `write()` statements will append content to the file without overwriting the existing contents. For example, after writing to the file initially, subsequent `write()` statements within the `with` statement will append new content to the text file.

```
name = input("Enter name: ")

with open("output.txt", "w") as f:
    f.write(name + "\n")
    f.write("My name is on the line above in this text file.")
```

However, it's important to note that if you open the file again with modes like `w` or `w+`, the existing contents will be overwritten. This behaviour also occurs if the program is run multiple times, where each execution with these modes replaces the previous content in the file. It's crucial to be mindful of this behaviour when using these write modes in Python file handling.

Don't forget to close the file if you're not using **`with`** and **`as`**. For this reason, it is generally better to stick to using **`with`** and **`as`** in Python file handling.

```
open_file = open("text_file_name.txt", "w")

open_file.close()  # Manually closing the opened file.
```

FILE DIRECTORY

A file directory, also known as a folder, is a container that can hold multiple files and other directories. When working with files in Python, you may need to specify the file location or path to access or manipulate files in different directories.

It's important to note that you should never hard-code file locations. The correct way is to specify the location of a file relative to the current working directory, i.e., the 'relative file path'.

Relative file paths can use special symbols like '**`.`**' (to represent the current directory) and '**`..`**' (to represent the parent directory).

In addition, slashes are used to specify the directory hierarchy and as directory separators within relative paths. The forward slash '**/**' is commonly used as a directory separator in Unix-like systems (e.g., Linux, macOS) while the backslash '****' is commonly used as a directory separator in Windows systems.

Wrong	`C:/Users/MyName/Document/Python/Hyperion-Dev/example.txt`
Use the following if the file is in the same folder.	
Correct	`example.txt` OR `./example.txt`
Use the following if the file is one folder back.	
Correct	`../example.txt`

FILE ATTRIBUTES IN PYTHON

When working with files in Python, you can get useful details like whether a file exists, its size, and the last time it was modified. These attributes help you manage files more effectively and avoid errors. Here's how you can retrieve these attributes:

1. To check if a file exists:

```
import os

print(os.path.exists("example.txt"))  # Returns True if the file exists
```

2. To get the size of a file:

```
import os

print(os.path.getsize("example.txt"))  # Returns the size of the file in
bytes
```

3. To check the time the file was last modified:

```
import os
import time

# Get the last modification time of 'example.txt'
last_modified = os.path.getmtime('example.txt')

# Convert the timestamp to a human-readable format
last_modified_str = time.ctime(last_modified)

# Print the last modification time
print(last_modified_str)
```

4. Handling a `FileNotFoundError` exception:

   Handling **exceptions** like `FileNotFoundError` during file operations in Python is crucial for enhancing code robustness. By anticipating and managing cases where files may not exist, your program can provide informative feedback to users and prevent unexpected crashes.

```
try:
    # Attempt to open 'example.txt' in read mode
    with open("example.txt", "r") as file:
        # Read the entire content of the file into 'content'
        content = file.read()

    # Print the content of the file
    print(content)

except FileNotFoundError:
    # Handle the case where 'example.txt' does not exist
    print("The file does not exist. Please check the file path and try
again.")
```

   If you'd like to read further on this subject, dive in and experiment with some of **these functions and modules**.

   By leveraging file attributes, you will become a file-handling ninja in Python! These attributes equip you to tackle various file-related challenges, from ensuring data integrity to optimising program performance.

FILE ENCODING

When you open a file in Python, you can also specify how the text in the file is encoded. Encoding is like a translator between human-readable text and the computer's bytes. When you turn text into bytes, it's called **encoding**, and when you turn bytes back into text, it's called **decoding**. You may read more on this in the **official documentation for Python**.

You can add an extra optional argument to the `open()` function to tell Python which encoding method to use. This is important because different systems and files might use different encoding methods, and using the wrong one can cause weird characters to appear, for example:

```
["ï»¿This is an example file!\n", "You've read from it! Congrats!"]
```

To avoid this, you can specify the encoding method when you open a file. One common encoding method is `utf-8`, which works well for most text files. Here's how you can use it:

```
file = open("example.txt", "r+", encoding="utf-8")
```

This tells Python to use the `utf-8` encoding, which should display your text correctly.

See the **codecs module** for the list of supported encodings.

Take note

The task(s) below is/are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select “Request Review”, the task is automatically complete, you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you’ve done that, feel free to progress to the next task.

Instructions

First, read and run the **example files** provided. Feel free to write and run your own example code before doing the auto-graded tasks to become more comfortable with the concepts covered here.

Auto-graded task 1

1. Create a new Python file in the folder for this task, and call it **dob_task.py**.
2. In your Python file, write a program that reads the data from the text file provided (**DOB.txt**) and prints it out in two different sections: one for names and another for birthdates, as shown in the format displayed below:

Name
Orville Wright
Rogelio Holloway
Marjorie Figueroa
... etc.

Birthdate
21 July 1988
13 September 1988
9 October 1988
... etc.

Before submitting your task, test your program to ensure that it correctly reads **DOB.txt** and outputs the data in the required format. This step verifies that the file references are correct and that Python can access **DOB.txt** from the same folder. You may need to move the **DOB.txt** file to the same directory as your **dob_task.py** file to ensure proper access.

Note: Remember to ensure that the text folder is in the appropriate file directory or Python won't be able to find it when running your program. Get this right first by running the example files, and then do the task.

Auto-graded task 2

Follow these steps:

- Create a file called **student_register.py**.
- Write a program that allows a user to register students for an exam venue.
- First, ask the user how many students are registering.
- Create a `for` loop that runs for that number of students.
- Each time the loop runs the program should ask the user to enter the next student ID number.
- Write each of the ID numbers to a text file called **reg_form.txt**.
- Include a dotted line after each student ID because this document will be used as an attendance register, which the students will sign when they arrive at the exam venue.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.