TASK

File IO Operations

Visit our website

Introduction

WELCOME TO THE IO OPERATIONS – INPUT TASK!

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.

A note from the
Hyperion Team

The Python community is alive and growing at a staggering rate. An active community has many benefits. New people bring new ideas, fresh perspectives, and different levels of experience.

One such benefit is the proliferation of packages. If you need functionality, there is a good chance that someone else needed it before you. The beauty of the Python community is that most of these functions become packages.

“There should be one – and preferably only one – obvious way to do it.”
– The Zen of Python

However, this is not always the case. Sometimes you have many options! Feel free to explore them. **Here** are five more packages you should know about.

OPENING FILES

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

There are two methods for opening a file in Python. Both methods use a built-in function called `open()` which accepts two arguments:

1. The first argument is the filename (including the path if necessary) that specifies the file to be accessed, for example, `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 delve 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:

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

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

   This method opens the file using a `with/as` block. This is the more 'pythonic' way of opening a file as **it automatically closes the file again** once the program reaches the end of that block of code:

   ```python
   with open('example.txt', 'r+') as file:
       # Perform operations on the open file
   ```

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 methods of reading files in Python:

1. Using the `read()` method:
   The `read()` method is used to read the entire contents of a text file and returns it as a string:

   ```
   with open('example.txt', 'r+') as file:
         lines = file.read() # Reads all the data in the file
   ```

   The code in the example above reads the entire file. You could also pass an integer argument to the `read()` method to read a specific number of characters from a file:

   ```
   with open('example.txt', 'r+') as file:
         lines = file.read(10) # Reads the first 10 characters in 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 string of data:

   ```
   with open('example.txt', 'r+') as file:
   ```

```
        lines = file.readline() # Reads a line of data in 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 = 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:
   Similar to the **`readline()`** method, this method 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
```

   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:
        for line in file:
                print("The entire line is: " + 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 = ""                               # Store the contents

with open('example.txt', 'r+') as file:   # Open the file
        for line in file:                   # Iterate through the lines
                contents = contents + line  # Add the contents of each line
        print(contents)                     # Print the contents
```

We now have the contents of an external resource (a text file) stored inside our program in a variable called `contents`. That's pretty powerful! We can then print the contents to a screen with `print(contents)`.

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")
```

We create a new file called **output.txt** (it doesn't exist yet) in write mode. Python will automatically create this file in the directory/folder that our program is in. We ask the user for their name. When they enter it, it is stored as a string in the variable called `name`. You then use the `write()` method in order to write to a file. The final line of code above will write the string value stored in the variable called `name` and a newline (`\n`) to the file that has been opened.

You must run this Python file for the file **output.txt** to be created with the output generated by this program in it.

We can write to the file again, and the current contents of the file will not be overwritten. Instead, it will be written on the 2nd line of the text file:

```
f.write("My name is on the line above in this text file.")
```

However, if you open the file again, the existing contents will be overwritten. This is important to remember when editing files.

Don't forget to close the file if you're not using ***with/as***! (For this reason, it is generally better to stick to using ***with/as***.)

```
open_file.close()
```

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”`
If the file is in the same folder.	
Correct	`“example.txt”` OR `“./example.txt”`
If the file is one folder back.	
Correct	`“../example.txt”`

FILE ENCODING

An extra optional argument can be passed to the `open()` function. This argument specifies the encoding of the file. The **official documentation for Python** explains that “serialising a string into a sequence of bytes is known as encoding, and recreating the string from the sequence of bytes is known as decoding”.

There are several different methods used to decode or encode files. The optional argument that is passed to the `open()` function specifies which of these methods to use. This should only be used in text mode. The default encoding is platform-dependent (whatever `locale.getpreferredencoding()` returns). Using the default encoding could result in some strange characters being displayed when Python reads a file.

For example:

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

We can correct this by specifying an encoding method, as shown below:

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

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

Instructions

First, read and run the **example files** provided. Feel free to write and run your own example code before doing the Practical Task to become more comfortable with the concepts covered in this task.

Practical Task 1

- Create a new Python file in the Dropbox folder for this task, and call it **dob_task.py**.
- 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 in the format displayed below:

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

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

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.

Practical 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.

Rate us

Share your thoughts

Hyperion strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.