Files and Exceptions

University of Wyoming COSC 1010

Adapted from: *Python Crash Course 3rd Ed* By Eric Matthes

Files and Exceptions

- Hopefully by now you have a grasp on all of the basics in Python
- You should be able to write and organize multi-faceted programs
- Now we can move on to more advanced applications, like opening and working with files
- this allows programs to quickly have access to and work with lots of data

Files and Exceptions

- In addition we will look at handling errors
  - This will allow your program to handle unexpected behavior
- In addition we will talk about *exceptions* which are special objects in Python to manage errors

Files and Exceptions

- Working with files and saving data makes programs easier for people to use
- It allows people to choose what data to enter and when to enter it
- It also introduces persistence to your programs, their state will reflect previous interactions
- Handling exceptions allows you to deal with situations when files don't exists, or when your program may crash

Reading from a File

- Text files can contain incredible amounts of data
- They could contain data about weather, traffic, literary works, etc
- Reading from a file is particularly useful in data analysis applications

- It is applicable for any situation where you amy want to read, write, or analyze the contents of a file
- You could write a program that reads in the contents of a text file, adn rewrites the file with formatting to be read by a browser

Reading the Contents of a File

- We can start with a file that contains the first 30 decimal places of pi
  ```
  3.1415926535
    8979323846
    2643383279
  ```
- We can then write a program that opens the file

In [ ]:
```
from pathlib import Path

path = Path('pi.txt')
contents = path.read_text()
print(contents)
```

```
3.1415926535
 8979323846
 2643383279
```

Reading the Contents of a File

- To work with the contents of a file, we need to tell Python the *path*
- A *path* is the exact location of a file or folder ona system
- Python includes a module *pathlib* that makes working with files easy, no matter the OS of the system
- A module that provides this kind of functionality is often called a *library*

Reading the Contents of a File

- We started by importing the `Path` class from `pathlib`
- There is a lot that can be done with a path object pointing to a file
  - You can check if the file exists before trying to work with it
  - Read the contents
  - Write new data to it

Reading the Contents of a File

- We built a `Path` object representing the file `pi.txt`
- The object is assigned to the variable `path`
- The file is saved in the same directory as the Python code, so the file name is all that is needed
- With a `Path` object representing the file `.read_text()` can be used to read the contents
- The contents are returned as a string, which are then printed

Reading the Contents of a File

- There is a difference when compared to the original file
  - It will print an extra blank line when it is output
- The blank line appears because `.read_text()` returns an empty string when it reaches the end of the file, displaying as a blank line
- This empty string can be removed with `.rstrip()`

Relative and Absolute File Paths

- When you pass a simple file name to `Path`, Python looks in the current directory, where the code is being stored
- Sometimes the file you want to open won't be in the same directory
- Perhaps you will have a sub directory called `data`
- As a result you would no longer be able to pass in just the file name
  - As Python won't look in sub directories for the file
- To get python to open the file you will need to tell it where the file is stored

Relative and Absolute File Paths

- There are two main ways to specify paths in programming
  - *Relative* paths
  - *Absolute* paths

Relative and Absolute File Paths

- A *relative* path tells Python to look for a given location relative to your current location
- If `data` were inside your current directory, a path needs to be built including `data`
  - `path = Path('data/pi.txt')`

Relative and Absolute File Paths

- You can also tell python where the file is on your computer regardless of where the program is being executed
- This is called teh *absolute* path
- An *absolute* path can be used if a relative path doesn't work
- Absolute paths start at the root of the file system
- using absolute paths you can read files from any location on your system

Accessing a File's Lines

- When working with a file, often you will want to examine each line of the file
- Maybe you;re looking for certain information, or want to modify the text in some fashion
- You may want to read through a file of weather data and work with any line that includes the word *sunny*
- To look at the lines `.splitlines()` can be used

In [ ]:
```
path = Path('pi.txt')
contents = path.read_text()

lines = contents.splitlines()
for line in lines:
    print(line)
```

```
3.1415926535
 8979323846
 2643383279
```

Accessing a File's Lines

- That again reads the files contents
- It then splits it into a list, based on the lines
  - `.splitlines()` returns a list

- We can then iterate through the list to print all the lines

Working with a File's Contents

- After you've read the contents into memory, you can do whatever you want with them
- For example, we can join all the digits of pi into a single string
  - With no white space in it!

In [ ]:
```
path = Path('pi.txt')
contents = path.read_text()

lines = contents.splitlines()
pi_string = ""
for line in lines:
    pi_string += line.strip()

print(len(pi_string))
print(pi_string)
```

```
32
3.14159265358979323846264338327 9
```

Working with a File's Contents

- Again the file is read
- Rather than printing each line in the file we instead string the whitespace
- And append them to a string

Large Files

- Our initial file was relatively small
- Instead, we can look at a file with Pi to a million decimal places
- However, despite the drastically larger file the process is the same
- We can turn it into a single string with no breaks
- Python has no inherit limits to how much data you can work with
- You can handle as much data as your system's memory can handle

In [ ]:
```
path = Path('pi_million_digits.txt')
contents = path.read_text()

lines = contents.splitlines()
pi_string = ""
for line in lines:
    pi_string += line.strip()

print(len(pi_string))
print(pi_string[:52])
```

```
1000002
3.14159265358979323846264338327950288419716939937510
```

Is Your Birthday in PI?

- With a million digits there are a lot of possible number combinations
- So, a question you could ask is "Is my birthday in Pi?"
- Well, now with the power of programming we can figure out!
- We will need pi as a string
- And a date as a string!

In [ ]:
```
path = Path('pi_million_digits.txt')
contents = path.read_text()

lines = contents.splitlines()
pi_string = ""
for line in lines:
    pi_string += line.strip()

birthday = input("Please provide your birthday in mmddyy format")

if birthday in pi_string:
    print("Your birthday is in Pi!")
else:
    print("Birthday not found in Pi")
```

```
Your birthday is in Pi!
```

Writing to a File

- One of the simplest ways to save data is to write it to a file
- When you write text to a file, the output wills till be available after the program stops running
- You can examine output after a program finishes running
- You can even share the output file with others if you need!
- Or, write a program that reads the data back in

Writing a single Line

- Once you have path defined, you can write to a file with `.write_text()`
- This works in a similar fashion to how `.read_text()` worked
- But, we will need to supply an argument
  - What we want to write to the file!
- Note, python can only write strings to files

In [ ]:
```
path = Path('example.txt')
path.write_text('Now we can save data!')
```

Out[ ]: 21

Writing Multiple Lines

- `.write_lines()` does a few things behind the scenes
- If the file that `path` points to doesn't exist, it creates the file
- After writing the string it also makes sure that the file is properly closed
  - This helps avoid files being corrupted if not closed correctly
- To write more than one line, a string containing all lines needs to be built
- Then `.write_text()` can be called with that string

In [ ]:
```
to_write = "He always sings raggy music to the cattle as he swings back and forward
to_write += "He's got a syncopated gaiter, and you ought to hear the meter to the r
to_write += "when they hear that he's 'a-comin', cause the western folks all know,
to_write += "Ragtime Cowboy, Talk about your Cowboy, Ragtime Cowboy Joe. "

path = Path("ragtime.txt")
path.write_text(to_write)
```

Out[ ]: 437

Writing Multiple Lines

- We defined the variable `to_write` that holds all the lines we wish to write out
- Then the `+=` operator is used to add the subsequent lines
- Each line includes a newline character `\n` so there will be line breaks in files
- Spaces, tabs, blanks, etc can be used to format the string as you want

Exceptions

- Python uses special objects called *exceptions* to manage errors
- Whenever an error occurs that make sPython unsure of what to do next, an exception is created
- If you write code that handles the exception the program will keep running, rather than crash
- IF the exceptions aren't handled the program will stop and a *traceback* will be shown

Exceptions

- Exceptions are handled in *try-except* blocks
- A *try-except* block asks Python to do something, but it also tells python what to do if an exception is raised
- When using `try-except` blocks your program will continue running even if things go wrong
- Instead of tracebacks users would see a message written by you the developer!
- To demonstrate let's do something impossible, like dividing by zero!

Handling the ZeroDivisionError Exception

- Dividing by zero is a simple error that would cause Python to raise an exception
- It is impossible to divide by zero, so trying to get python to will cause an error

In [ ]:
```
print(5/0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/tmp/ipykernel_25817/1152173066.py in <module>
----> 1 print(5/0)

ZeroDivisionError: division by zero
```

Handling the ZeroDivisionError Exception

- AS Python is unable to divide by zero, we get a traceback
- `ZeroDivisionError` is an exception object
- Python creates this kind of object in response to a situation where it can't do what is asked of it
- When this happens Python stops the program and states the exception
- This information can be used to modify a program

Using `try-except` Blocks

- At some points in your code you may have areas that are prone to errors
- You can use a `try-except` block to handle any exception that may be raised
- This is done by telling Python to try to run some code
- And what to do if the code results in an error
- You can also monitor for specific errors, like `ZeroDivisionError`

In [ ]:
```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by 0!")
```

```
You can't divide by 0!
```

Using `try-except` Blocks

- The `print(5/0)` line is within a try block
- If the code in the `try` block works Python skips over the `except`
- If the code in the `try` causes an error then Python will look for a matching block and run the code within
- Here Python looked for a `ZeroDivisionError` except block
  - Python found it, so the coded in that block was run

Using Exceptions to Prevent Crashes

- Handling `errors` correctly is especially important when the program has more work to do after the error
- This happens often in programs that prompt users for input
- If the input is not properly validated an error could occur, crashing the program
- Instead, we can utilize `try-except` blocks to alow programs to gracefully continue
- It typically isn't a good idea to let users see tracebacks

In [ ]:
```
print("Give me two numbers and I will divide them\n'q' to quit")

while True:
    first_num = input("\nfirst number: ")
    if first_num == 'q':
        break
    second_num = input('second number: ')
    if second_num == 'q':
        break
```

```
    answer = int(first_num)/int(second_num)
    print(answer)
```

```
Give me two numbers and I will divide them
'q' to quit
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/tmp/ipykernel_25817/3123755407.py in <module>
      8     if second_num == 'q':
      9         break
---> 10     answer = int(first_num)/int(second_num)
     11     print(answer)

ZeroDivisionError: division by zero
```

The `else` Block

- The program can be made more resilient by wrapping the division line in a `try-except`
- We can expand it to also include an `else` block
- Any code that is dependent on the `try` block executing correctly goes in the `else`
- Then it will execute upon successful completion of the `try`

In [ ]:
```
print("Give me two numbers and I will divide them\n'q' to quit")

while True:
    first_num = input("\nfirst number: ")
    if first_num == 'q':
        break
    second_num = input('second number: ')
    if second_num == 'q':
        break
    try:
        answer = int(first_num)/int(second_num)
    except ZeroDivisionError:
        print("Can't divide by zero!")
    else:
        print(answer)
```

```
Give me two numbers and I will divide them
'q' to quit
Can't divide by zero!
```

The `else` Block

- Now Python will again try to do the division
  - But, will keep running even if the second number is a zero
- Any code that depends on the try block succeeding is added to the `else` block

- So, if the division is successful then the answer will print
- The `except` block tells python how to respond when a `ZeroDivisionError` arises

The `else` Block

- The only code that should go in a `try` block is code that may cause an error
- Sometimes you may have additional code that should be run only if the `try` block succeeds, which should be in the `else`
- By anticipating likely sources of errors you can write robust programs that continue to run even if there is an error
- Your code will be resistant to innocent user mistakes and malicious attacks

Handling FileNotFoundError Exceptions

- One common issue when working with files is handling missing files
- The file you're looking for might:
  - Be in a different location
  - The file name may be misspelled
  - The file may not exist
- All of these can be handled with a single `try-except` block

In [ ]:
```
from pathlib import Path

path = Path("fakefile.nowhere")
contents = path.read_text(encoding='utf-8')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
/tmp/ipykernel_25817/2510817648.py in <module>
      2
      3 path = Path("fakefile.nowhere")
----> 4 contents = path.read_text(encoding='utf-8')

/usr/lib/python3.10/pathlib.py in read_text(self, encoding, errors)
   1132         """
   1133         encoding = io.text_encoding(encoding)
-> 1134         with self.open(mode='r', encoding=encoding, errors=errors) as f:
   1135             return f.read()
   1136

/usr/lib/python3.10/pathlib.py in open(self, mode, buffering, encoding, errors, newl
ine)
   1117         if "b" not in mode:
   1118             encoding = io.text_encoding(encoding)
-> 1119         return self._accessor.open(self, mode, buffering, encoding, errors,
   1120                                    newline)
   1121

FileNotFoundError: [Errno 2] No such file or directory: 'fakefile.nowhere'
```

Handling FileNotFoundError Exceptions

- There we tried `.read_text()` which previously worked with no issue
- But, now we tried to open a file that doesn't exist
- The `encoding` argument is needed when that system's encoding default doesn't match the file
- Regardless, Python can't read the file as it doesn't exist
  - So, an exception is raised

Handling FileNotFoundError Exceptions

- The traceback is a longer one than we have seen previously, so we can talk about making sense of longer tracebacks
- When presented with a long traceback it is often best to start at the end
  - `FileNotFoundError: [Errno 2] No such file or directory: 'fakefile.nowhere'`
  - This is the actual error
- Earlier in the file shows where the error occurred in the file

Handling FileNotFoundError Exceptions

- To handle the error we can again utilize a `try-except` block
  - We know from the traceback the error is `FileNotFoundError`
- So the code can be modified to then handle such errors
- If the file doesn't exist the program likely doesn't have anything else it needs to do

In [ ]:
```
from pathlib import Path

path = Path("fakefile.nowhere")
try:
    contents = path.read_text(encoding='utf-8')
except FileNotFoundError:
    print("uh oh, file not found")
else:
    print(contents)
```

```
uh oh, file not found
```

Analyzing Text

- You can analyze text files that represent entire books
  - You can find public domain books on *https://gutenberg.org*
  - Project Gutenberg maintains a collection of literary works that are available in the public domain
- For this example we can pull in *The Great Gatsby* and calculate the number of words

In [ ]:
```
path = Path("greatgatsby.txt")
try:
    contents = path.read_text()
except FileNotFoundError:
    print(f"Sorry {path} not found")
else:
    print(f"The file {path} has about {len(contents.split())} words")
```

```
The file greatgatsby.txt has about 51225 words
```

Analyzing Text

- We opened the file `greatgatsby.txt`
- We then read all the text into the variable contents
- Then we split contents on spaces, and get the length of the resulting array
- The count is higher than the book truly has, as Project Gutenberg includes some additional information

Working with Multiple Files

- You could automate this to include multiple books to analyze
- Moving the analysis to a function is a good way to start
- Then, you can hold a list of file names you with to work with
- Looping through the list you could then form a path from each and read as normal

Failing Silently

- Previously we printed a message when a code block failed
- However, we don't have to do so
- You don't need to report every exception that you catch
- In fact sometimes you will want the program to fail without notification, continuing on as if nothing happened
- To do so you would place a `pass` statement in the except block

In [ ]:
```
try:
    print(5/0)
except:
    pass
```

Failing Silently

- The pass can also act as a placeholder
- It's a reminder that you are explicitly doing nothing in a code block
  - But, you may want to do something with it later

Deciding Which Errors to Report

- How do you know which errors are worth reporting?
- If a user knows which texts are supposed to be analyzed, then they may appreciate a notification on why some texts weren't
- However, if a user expects results but doesn't necessarily know which texts are included they may not need to know which failed
- Giving users information they aren't looking for can reduce usability of a program