10 FILES AND EXCEPTIONS

10.1 Reading from a file

☝ Reading from a file is particularly useful in data analysis applications, but it's also applicable to any situation in which you want to analyze or modify information stored in a file. ☝ When you want to work with the information in a text file, the first step is to read the file into memory. You can read the entire contents of a file, or you can work through the file one line at a time.

10.1.1 Reading an Entire File

☝ To begin, we need a file with a few lines of text in it. Let's start with a file that contains pi to 30 decimal places with 10 decimal places per line:

pi_digits.txt

```
3.1415926535
  8979323846
  2643383279
```

☝ Here's a program that opens this file, reads it, and prints the contents of the file to the screen:
file_reader.py

```
with open('pi_digits.txt') as file_object:
contents = file_object.read()
print(contents)
```

❗ Here, `open('pi_digits.txt')` returns an object representing pi_digits.txt. Python stores this object in file_object, which we'll work with later in the program.

☝ nce we have a file object representing pi_digits.txt, we use the `read()` method in the second line of our program to read the entire contents of the file and store it as one long string in contents. When we print the value of contents, we get the entire text file back:

```
3.1415926535
  8979323846
  2643383279
```

☝ The only difference between this output and the original file is the extra blank line at the end of the output.

☝ The blank line appears because read() returns an empty string when it reaches the end of the file; this empty string shows up as a blank line. If you want to remove the extra blank line,you can use rstrip() in the print statement:

```
with open('pi_digits.txt') as file_object:
   contents = file_object.read()
   print(contents.rstrip())
```

Now the output matches the contents of the original file exactly.

```
3.1415926535
  8979323846
  2643383279
```

10.1.2 File Paths

☝ Sometimes, depending on how you organize your work, the file you want to open won't be in the same directory as your program file ☝ To get Python to open files from a directory other than the one where your program file is stored, you need to provide a file path, which tells Python to look in a specific location on your system ☝ Because text_files is inside python_work, you could use a relative file path to open a file from text_files. A relative file path tells Python to look for a given location relative to the directory where the currently running program file is stored

---

☝ On Linux and OS X, you'd write:

```
with open('text_files/filename.txt') as file_object:
```

☝ This line tells Python to look for the desired .txt file in the folder text_files and assumes that text_files is located inside python_work (which it is).On Windows systems, you use a backslash `(\)` instead of a forward slash `(/)` in the file path:

```
with open('text_files\filename.txt') as file_object:
```

☝ You can also tell Python exactly where the file is on your computer regardless of where the program that's being executed is stored. This is called an absolute file path. ☝ Absolute paths are usually longer than relative paths, so it's helpful to store them in a variable and then pass that variable to open(). ☝ On Linux and OS X, absolute paths look like this:

```
file_path = '/home/ehmatthes/other_files/text_files/filename.txt'
with open(file_path) as file_object:
```

☝ and on Windows they look like this:

```
file_path = 'C:\Users\ehmatthes\other_files\text_files\filename.txt'
with open(file_path) as file_object:
```

exclamation:Windows systems will sometimes interpret forward slashes in file paths correctly. If you're using Windows and you're not getting the results you expect, make sure you try using backslashes.

10.1.3 Reading Line by Line

☝ You can use a for loop on the file object to examine each line from a file one at a time: **file_reader.py**

```
❶ filename = 'pi_digits.txt'

❷ with open(filename) as file_object:
❸     for line in file_object:
      print(line)
```

☝ When we print each line, we find even more blank lines:

```
3.1415926535

  8979323846
```

```
2643383279
```

☝ These blank lines appear because an invisible newline character is at the end of each line in the text file. The print statement adds its own newline each time we call it, so we end up with two newline characters at the end of each line: one from the file and one from the print statement.

10.1.5 Working with a File's Contents

☝ After you've read a file into memory, you can do whatever you want with that data. **pi_string.py**

```
  filename = 'pi_digits.txt'
  with open(filename) as file_object:
    lines = file_object.readlines()
❶ pi_string = ''
❷ for line in lines:
    pi_string += line.rstrip()
❸ print(pi_string)
  print(len(pi_string))
```

☝ At ❸, we print this string and also show how long the string is:

```
3.1415926535 8979323846 2643383279
36
```

☝ The variable pi_string contains the whitespace that was on the left side of the digits in each line, but we can get rid of that by using strip() instead of rstrip():

```
filename = 'pi_30_digits.txt'
with open(filename) as file_object:
   lines = file_object.readlines()
pi_string = ''
for line in lines:
   pi_string += line.strip()
print(pi_string)
print(len(pi_string))
```

☝ Now we have a string containing pi to 30 decimal places.

```
3.14159265358979323846264338327
32
```

❗ When Python reads from a text file, it interprets all text in the file as a string. If you read in a number and want to work with that value in a numerical context, you'll have to convert it to an integer using the int() function or convert it to a float using the float() function.

10.1.6 Large Files: One Million Digits

☝ If we start with a text file that contains pi to 1,000,000 decimal places instead of just 30, we can create a single string containing all these digits.We don't need to change our program at all except to pass it a different file. **pi_string.py**

```
filename = 'pi_million_digits.txt'
with open(filename) as file_object:
    lines = file_object.readlines()
pi_string = ''
for line in lines:
   pi_string += line.strip()
print(pi_string[:52] + "...")
print(len(pi_string))
```

☝ The output shows that we do indeed have a string containing pi to 1,000,000 decimal places:

```
3.14159265358979323846264338327950288419716939937510...
1000002
```

10.1.7 Is Your Birthday Contained in Pi?

☝ Let's use the program we just wrote to find out if someone's birthday appears anywhere in the first million digits of pi.

```
filename = 'pi_million_digits.txt'
with open(filename) as file_object:
   lines = file_object.readlines()
pi_string = ''
for line in lines:
   pi_string += line.rstrip()
❶birthday = input("Enter your birthday, in the form mmddyy: ")
❷if birthday in pi_string:
   print("Your birthday appears in the first million digits of pi!")
```

```
else:
   print("Your birthday does not appear in the first million digits of pi.")
```

☝ Let's try it:

```
Enter your birthdate, in the form mmddyy: 120372
Your birthday appears in the first million digits of pi!
```

☝ My birthday does appear in the digits of pi! Once you've read from a file, you can analyze its contents in just about any way you can imagine.

10.2 Writing to a file

10.2.1 Writing to an Empty File

☝ To write text to a file, you need to call open() with a second argument telling Python that you want to write to the file. **write_message.py**

```
filename = 'programming.txt'
❶ with open(filename, 'w') as file_object:
❷ file_object.write("I love programming.")
```

programming.txt

```
I love programming.
```

❗ Python can only write strings to a text file. If you want to store numerical data in a text file, you'll have to convert the data to string format first using the str() function.

10.2.2 Writing Multiple Lines

☝ The write() function doesn't add any newlines to the text you write. So if you write more than one line without including newline characters, your file may not look the way you want it to:

```
filename = 'programming.txt'
with open(filename, 'w') as file_object:
   file_object.write("I love programming.")
   file_object.write("I love creating new games.")
```

☝ If you open programming.txt, you'll see the two lines squished together:

```
I love programming.I love creating new games.
```

☝ Including newlines in your write() statements makes each string appear on its own line:

```
filename = 'programming.txt'
with open(filename, 'w') as file_object:
   file_object.write("I love programming.\n")
   file_object.write("I love creating new games.\n")
```

☝ The output now appears on separate lines:

```
I love programming.
I love creating new games.
```

10.2.3 Appending to a File

☝ If you want to add content to a file instead of writing over existing content,you can open the file in append mode. ☝ Let's modify write_message.py by adding some new reasons we love programming to the existing file programming.txt: **write_message.py**

```
filename = 'programming.txt'
❶ with open(filename, 'a') as file_object:
❷    file_object.write("I also love finding meaning in large datasets.\n")
     file_object.write("I love creating apps that can run in a browser.\n")
```

---

programming.txt

```
I love programming.
I love creating new games.
I also love finding meaning in large datasets.
I love creating apps that can run in a browser.
```

10.3 Exceptions

☝ Exceptions are handled with 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 you use try-except blocks, your programs will continue running even if things start to go wrong. Instead of tracebacks, which can be confusing for users to read, users will see friendly error messages that you write.

10.3.1 Handling the ZeroDivisionError Exception

☝ Let's look at a simple error that causes Python to raise an exception. **division.py**

```
print(5/0)
```

☝ Of course Python can't do this, so we get a traceback:

```
Traceback (most recent call last):
   File "division.py", line 1, in <module>
     print(5/0)
❶ ZeroDivisionError: division by zero
```

10.3.2 Using try-except Blocks

☝ Here's what a try-except block for handling the ZeroDivisionError exception looks like:

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

☝ We put print(5/0), the line that caused the error, inside a try block. ☝ In this example, the code in the try block produces a ZeroDivisionError,so Python looks for an except block telling it how to respond. Python then runs the code in that block.

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

10.3.3 Using Exceptions to Prevent Crashes

☝ Let's create a simple calculator that does only division: **division.py**

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")
```

```
while True:
❶ first_number = input("\nFirst number: ")
  if first_number == 'q':
    break
❷ second_number = input("Second number: ")
  if second_number == 'q':
     break
❸ answer = int(first_number) / int(second_number)
  print(answer)
```

☝ This program does nothing to handle errors,so asking it to divide by zero causes it to crash:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.
First number: 5
Second number: 0
Traceback (most recent call last):
  File "division.py", line 9, in <module>
    answer = int(first_number) / int(second_number)
ZeroDivisionError: division by zero
```

10.3.4 The else Block

☝ We can make this program more error resistant by wrapping the line that might produce errors in a try-except block. ☝ This example also includes an else block. Any code that depends on the try block executing successfully goes in the else block:

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")
while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
       break
    second_number = input("Second number: ")
❶   try:
        answer = int(first_number) / int(second_number)
❷   except ZeroDivisionError:
        print("You can't divide by 0!")
❸   else:
        print(answer)
```

☝ The except block tells Python how to respond when a ZeroDivisionError arises ❷.

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.
First number: 5
Second number: 0
You can't divide by 0!
First number: 5
Second number: 2
2.5
First number: q
```

☝ By anticipating likely sources of errors, you can write robust programs that continue to run even when they encounter invalid data and missing resources. Your code will be resistant to innocent user mistakes and malicious attacks.

10.3.5 Handling the FileNotFoundError Exception

☝ Let's try to read a file that doesn't exist. **alice.py**

```
filename = 'alice.txt'
with open(filename) as f_obj:
  contents = f_obj.read()
```

☝ Python can't read from a missing file, so it raises an exception:

```
Traceback (most recent call last):
  File "alice.py", line 3, in <module>
    with open(filename) as f_obj:
FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'
```

☝ In this example, the open() function produces the error, so to handle it, the try block will begin just before the line that contains open():

```
filename = 'alice.txt'
try:
   with open(filename) as f_obj:
       contents = f_obj.read()
except FileNotFoundError:
   msg = "Sorry, the file " + filename + " does not exist."
   print(msg)
```

☝ In this example, the code in the try block produces a FileNotFoundError,so Python looks for an except block that matches that error. Python then runs the code in that block, and the result is a friendly error

message instead of a traceback:

```
Sorry, the file alice.txt does not exist.
```

10.3.6 Analyzing Text

☝ Let's pull in the text of Alice in Wonderland and try to count the number of words in the text.

```
>>> title = "Alice in Wonderland"
>>> title.split()
['Alice', 'in', 'Wonderland']
```

☝ To count the number of words in Alice in Wonderland,we'll use split() on the entire text. Then we'll count the items in the list to get a rough idea of the number of words in the text:

```
filename = 'alice.txt'
try:
    with open(filename) as f_obj:
        contents = f_obj.read()
except FileNotFoundError:
    msg = "Sorry, the file " + filename + " does not exist."
    print(msg)
else:
    # Count the approximate number of words in the file.
❶   words = contents.split()
❷   num_words = len(words)
❸   print("The file " + filename + " has about " + str(num_words) + " words.")
```

☝ The output tells us how many words are in alice.txt:

```
The file alice.txt has about 29461 words.
```

10.3.7 Working with Multiple Files

☝ Let's add more books to analyze. **word_count.py**

```
 def count_words(filename):
❶ """ Count the approximate number of words in a file."""
   try:
```

```
        with open(filename) as f_obj:
            contents = f_obj.read()
    except FileNotFoundError:
        msg = "Sorry, the file " + filename + " does not exist."
        print(msg)
    else:
        # Count approximate number of words in the file.
        words = contents.split()
        num_words = len(words)
        print("The file " + filename + " has about " + str(num_words) +" words.")
filename = 'alice.txt'
count_words(filename)
```

☝ I've intentionally left siddhartha.txt out of the directory containing word_count.py, so we can see how well our program handles a missing file:

```
def count_words(filename):
    --snip--
filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'little_women.txt']
for filename in filenames:
    count_words(filename)
```

☝ The missing siddhartha.txt file has no effect on the rest of the program's execution:

```
The file alice.txt has about 29461 words.
Sorry, the file siddhartha.txt does not exist.
The file moby_dick.txt has about 215136 words.
The file little_women.txt has about 189079 words.
```

10.3.8 Failing Silently

☝ Python has a pass statement that tells it to do nothing in a block:

```
def count_words(filename):
    """ Count the approximate number of words in a file. """
    try:
        --snip--
    except FileNotFoundError:
❶       pass
    else:
        --snip--
filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'little_women.txt']
for filename in filenames:
    count_words(filename)
```

☝ Users see the word counts for each file that exists, but they don't see any indication that a file was not found:

```
The file alice.txt has about 29461 words.
The file moby_dick.txt has about 215136 words.
The file little_women.txt has about 189079 words.
```

10.3.9 Deciding Which Errors to Report

- How do you know when to report an error to your users and when to fail silently?

☝ If users know which texts are supposed to be analyzed, they might appreciate a message informing them why some texts were not analyzed.

☝ If users expect to see some results but don't know which books are supposed to be analyzed, they might not need to know that some texts were unavailable. Giving users information they aren't looking for can decrease the usability of your program.

☝ Python's error-handling structures give you finegrained control over how much to share with users when things go wrong; it's up to you to decide how much information to share.

10.4 Storing data

10.4.1 Using json.dump() and json.load()

☝ Let's write a short program that stores a set of numbers and another program that reads these numbers back into memory. The first program will use json.dump() to store the set of numbers, and the second program will use json.load(). **number_writer.py**

```
  import json
  numbers = [2, 3, 5, 7, 11, 13]
❶ filename = 'numbers.json'
❷ with open(filename, 'w') as f_obj:
❸     json.dump(numbers, f_obj)
```

☝ This program has no output, but let's open the file numbers.json and look at it. The data is stored in a format that looks just like Python:

```
[2, 3, 5, 7, 11, 13]
```

☝ Now we'll write a program that uses json.load() to read the list back into memory: **number_reader.py**

```
 import json
❶ filename = 'numbers.json'
❷ with open(filename) as f_obj:
❸     numbers = json.load(f_obj)
  print(numbers)
```

☝ Finally we print the recovered list of numbers and see that it's the same list created in number_writer.py:

```
[2, 3, 5, 7, 11, 13]
```

10.4.2 Saving and Reading User-Generated Data

☝ Let's start by storing the user's name: **remember_me.py**

```
  import json
❶ username = input("What is your name? ")
   filename = 'username.json'
   with open(filename, 'w') as f_obj:
❷    json.dump(username, f_obj)
❸    print("We'll remember you when you come back, " + username + "!")
```

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

☝ Now let's write a new program that greets a user whose name has already been stored: **greet_user.py**

```
  import json
  filename = 'username.json'
  with open(filename) as f_obj:
❶   username = json.load(f_obj)
❷   print("Welcome back, " + username + "!")
```

```
Welcome back, Eric!
```

☝ We need to combine these two programs into one file. **remember_me.py**

```
import json
# Load the username, if it has been stored previously.
# Otherwise, prompt for the username and store it.
filename = 'username.json'
try:
❶  with open(filename) as f_obj:
❷      username = json.load(f_obj)
❸  except FileNotFoundError:
❹      username = input("What is your name? ")
❺      with open(filename, 'w') as f_obj:
           json.dump(username, f_obj)
           print("We'll remember you when you come back, " + username + "!")
    else:
        print("Welcome back, " + username + "!")
```

☝ If this is the first time the program runs, this is the output:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

☝ Otherwise:

```
Welcome back, Eric!
```

10.4.3 Refactoring

☝ Often, you'll come to a point where your code will work, but you'll recognize that you could improve the code by breaking it up into a series of functions that have specific jobs. This process is called refactoring. ☝ We can refactor remember_me.py by moving the bulk of its logic into one or more functions.

---

remember_me.py

```
import json
def greet_user():
❶   """ Greet the user by name. """
   filename = 'username.json'
   try:
      with open(filename) as f_obj:
         username = json.load(f_obj)
   except FileNotFoundError:
```

```
        username = input("What is your name? ")
        with open(filename, 'w') as f_obj:
            json.dump(username, f_obj)
             print("We'll remember you when you come back, " + username + "!")
    else:
        print("Welcome back, " + username + "!")
 greet_user()
```

☝ Let's refactor greet_user() so it's not doing so many different tasks.

```
 import json
 def get_stored_username():
❶    """ Get stored username if available."""
       filename = 'username.json'
       try:
          with open(filename) as f_obj:
             username = json.load(f_obj)
       except FileNotFoundError:
❷        return None
       else:
           return username
 def greet_user():
    """ Greet the user by name. """
    username = get_stored_username()
❸   if username:
         print("Welcome back, " + username + "!")
     else:
         username = input("What is your name? ")
         filename = 'username.json'
         with open(filename, 'w') as f_obj:
             json.dump(username, f_obj)
             print("We'll remember you when you come back, " + username + "!")
 greet_user()
```

☝ We should factor one more block of code out of greet_user(). If the username doesn't exist, we should move the code that prompts for a new username to a function dedicated to that purpose:

```
import json
def get_stored_username():
   """ Get stored username if available."""
   --snip--
def get_new_username():
   """ Prompt for a new username."""
   username = input("What is your name? ")
   filename = 'username.json'
   with open(filename, 'w') as f_obj:
      json.dump(username, f_obj)
```

```
    return username
def greet_user():
    """ Greet the user by name. """
    username = get_stored_username()
    if username:
        print("Welcome back, " + username + "!")
    else:
        username = get_new_username()
        print("We'll remember you when you come back, " + username + "!")
greet_user()
```

10.5 Summary

☝🏽 In this chapter, you learned:

- how to work with files;how to read an entire file at once and read through a file's contents one line at a time.
- how to write to a file and append text onto the end of a file;
- what are exceptions and how to handle the exceptions you're likely to see in your programs;
- how to store Python data structures so you can save information your users provide, preventing them from having to start over each time they run a program.