Files and Exceptions

1. Reading from a file

2. Storing data with json file

3. Exception

Reading from a File

Variables are a fine way to store data while your program is running, but if you want your data to persist even after your program has finished, you need to save it to a file.

Variables are a fine way to store data while your program is running, but if you want your data to persist even after your program has finished, you need to save it to a file.

You can think of a file's contents as a single string value, potentially gigabytes in size. In this chapter, we will learn how to use Python to create, read, and save files on the hard drive.

Reading the Contents of a 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:

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:

In [2]:
```
%%writefile pi_digits.txt
3.1415926535
  8979323846
  2643383279
```

```
Writing pi_digits.txt
```

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:

In [2]:
```
%%writefile pi_digits.txt
3.1415926535
  8979323846
  2643383279
```

```
Writing pi_digits.txt
```

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

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:

In [2]:
```
%%writefile pi_digits.txt
3.1415926535
  8979323846
  2643383279
```

```
Writing pi_digits.txt
```

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

In [3]:
```
file_object = open('pi_digits.txt')
print(file_object.read())
file_object.close()
```

```
3.1415926535
  8979323846
  2643383279
```

In [4]:
```
# A recommended way:
with open('pi_digits.txt') as file_object: #  file_object = open('pi_digits.t
    contents = file_object.read()          #  We do not have to call file_obj
print(contents.strip())
contents
```

```
3.1415926535
  8979323846
  2643383279
```

Out[4]:
```
'3.1415926535\n  8979323846\n  2643383279\n'
```

In [4]:
```
# A recommended way:
with open('pi_digits.txt') as file_object: #  file_object = open('pi_digits.t
    contents = file_object.read()          #  We do not have to call file_obj
print(contents.strip())
contents
```

```
3.1415926535
  8979323846
  2643383279
```

Out[4]:
```
'3.1415926535\n  8979323846\n  2643383279\n'
```

The keyword `with` closes the file once access to it is no longer needed. Notice how we call `open()` in this program but not `close()`.

In [4]:
```
# A recommended way:
with open('pi_digits.txt') as file_object: #  file_object = open('pi_digits.t
    contents = file_object.read()          #  We do not have to call file_obj
print(contents.strip())
contents
```

```
3.1415926535
  8979323846
  2643383279
```

Out[4]:
```
'3.1415926535\n  8979323846\n  2643383279\n'
```

The keyword `with` closes the file once access to it is no longer needed. Notice how we call `open()` in this program but not `close()`.

We could open and close the file by calling `open()` and `close()`, but if a bug in your program prevents the `close()` method from being executed, the file may never close! This may cause data to be lost or corrupted.

In [4]:
```
# A recommended way:
with open('pi_digits.txt') as file_object: #  file_object = open('pi_digits.t
    contents = file_object.read()          #  We do not have to call file_obj
print(contents.strip())
contents
```

```
3.1415926535
  8979323846
  2643383279
```

Out[4]:
```
'3.1415926535\n  8979323846\n  2643383279\n'
```

The keyword `with` closes the file once access to it is no longer needed. Notice how we call `open()` in this program but not `close()`.

We could open and close the file by calling `open()` and `close()`, but if a bug in your program prevents the `close()` method from being executed, the file may never close! This may cause data to be lost or corrupted.

Since `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, we can use `strip()` in the call to `print()`.

In [5]:
```
display_quiz(path+"read.json", max_width=800)
```

What is printed by the following statements?

```
# Assume that the file 'data.txt' contains: 'Python is great.'
with open('data.txt', 'r') as f:
    print(f.read())
    print(f.read())
```

Nothing is printed.

Python is great.

```
Python is great.
Python is great.
```

```
An error is raised during the second read().
```

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.

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.

A ***relative file path*** will tell Python to look for a given location relative to the directory where the currently running program file is stored. For example, we'd write:

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

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.

A ***relative file path*** will tell Python to look for a given location relative to the directory where the currently running program file is stored. For example, we'd write:

```
with open(r'text_files\abc.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 in the current directory.

In [6]:
```
%mkdir text_files
```

In [6]:
```
%mkdir text_files
```

In [7]:
```
%%writefile text_files\pi_digits2.txt
3.1415926535
  8979323846
  2643383279
```

```
Writing text_files\pi_digits2.txt
```

In [6]:
```
%mkdir text_files
```

In [7]:
```
%%writefile text_files\pi_digits2.txt
3.1415926535
  8979323846
  2643383279
```

```
Writing text_files\pi_digits2.txt
```

In [8]:
```
with open(r'text_files\pi_digits2.txt') as file_object:
    contents = file_object.read()     # read whole contents
print(contents.strip())
```

```
3.1415926535
  8979323846
  2643383279
```

We can also tell Python exactly where the file is on our 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 assign them to a variable and then pass that variable to `open()`:

We can also tell Python exactly where the file is on our 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 assign them to a variable and then pass that variable to `open()`:

```
file_path = r'C:\Users\adm\Desktop\text_files\pi_digits2.txt'
with open(file_path) as file_object:
```

```
## Linux use slash instead of backslash
file_path = '/home/Users/Desktop/text_files/pi_digits2.txt'
with open(file_path) as file_object:
```

In [9]:
```
display_quiz(path+"path.json", max_width=800)
```

Which of the following paths are relative file paths?

- /private/tmp/swtag.txt
- Stacy/Applications/README.txt
- /Users/Raquel/Documents/graduation_plans.doc
- ScienceData/ProjectFive/experiment_data.csv

Reading Line by Line

When you're reading a file, you'll often want to examine each line of the file. You might be looking for certain information in the file, or you might want to modify the text in the file in some way. You can use a `for` loop on the file object to examine each line from a file one at a time:

When you're reading a file, you'll often want to examine each line of the file. You might be looking for certain information in the file, or you might want to modify the text in the file in some way. You can use a `for` loop on the file object to examine each line from a file one at a time:

In [10]:
```
filename = 'pi_digits.txt'
with open(filename) as file_object: # file_object is also iterable!
    for line in file_object:        # It is an iterable object
        print(line.strip())
```

```
3.1415926535
8979323846
2643383279
```

When you're reading a file, you'll often want to examine each line of the file. You might be looking for certain information in the file, or you might want to modify the text in the file in some way. You can use a `for` loop on the file object to examine each line from a file one at a time:

In [10]:
```
filename = 'pi_digits.txt'
with open(filename) as file_object: # file_object is also iterable!
    for line in file_object:        # It is an iterable object
        print(line.strip())
```

```
3.1415926535
8979323846
2643383279
```

Since there is a newline in each line of file and the print function adds its own newline each time we call it, so we will end up with two newline characters at the end of each line: one from the file and one from `print()`. Using `strip()` on each line in the `print()` call eliminates these extra blank lines.

Making a List of Lines from a File

When we use `with`, **the file object returned by `open()` is only available inside the with block that contains it**. If we want to retain access to a file's contents outside the `with` block, we can store the file's lines in a `list` inside the block and then work with that `list`!

When we use `with`, **the file object returned by `open()` is only available inside the with block that contains it**. If we want to retain access to a file's contents outside the `with` block, we can store the file's lines in a `list` inside the block and then work with that `list`!

In [11]:
```
filename = 'pi_digits.txt'
with open(filename) as file_object:
    lines = file_object.readlines()

print(lines) # List of strings
pi_string = ''
for line in lines:
    pi_string += line.strip()

print(pi_string)
print(len(pi_string)) # The string is 32 characters long because it also incl
```

```
['3.1415926535\n', '  8979323846\n', '  2643383279\n']
3.141592653589793238462643383279
32
```

When we use `with`, **the file object returned by `open()` is only available inside the with block that contains it**. If we want to retain access to a file's contents outside the `with` block, we can store the file's lines in a `list` inside the block and then work with that `list`!

In [11]:
```
filename = 'pi_digits.txt'
with open(filename) as file_object:
    lines = file_object.readlines()

print(lines) # List of strings
pi_string = ''
for line in lines:
    pi_string += line.strip()

print(pi_string)
print(len(pi_string)) # The string is 32 characters long because it also incl
```

```
['3.1415926535\n', '  8979323846\n', '  2643383279\n']
3.141592653589793238462643383279
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.

Writing to a File

Writing to an Empty File

To write text to a file, we need to call `open()` with a second argument telling Python that we want to write to the file:

To write text to a file, we need to call `open()` with a second argument telling Python that we want to write to the file:

In [12]:
```
filename = 'programming.txt'

with open(filename, 'w') as file_object: # Note it will overwrite the content
    file_object.write("We love programming!")
```

To write text to a file, we need to call `open()` with a second argument telling Python that we want to write to the file:

In [12]:
```
filename = 'programming.txt'

with open(filename, 'w') as file_object: # Note it will overwrite the content
    file_object.write("We love programming!")
```

The call to `open()` in this example has two arguments. The first argument is still the name of the file we want to open. The second argument, 'w', tells Python that we want to open the file in **write mode.**

To write text to a file, we need to call `open()` with a second argument telling Python that we want to write to the file:

In [12]:
```
filename = 'programming.txt'

with open(filename, 'w') as file_object: # Note it will overwrite the content
    file_object.write("We love programming!")
```

The call to `open()` in this example has two arguments. The first argument is still the name of the file we want to open. The second argument, 'w', tells Python that we want to open the file in **write mode.**

You can open a file in read mode ('r'), write mode ('w'), append mode ('a'), or a mode that allows you to read and write to the file ('r+'). **If you omit the mode argument, Python opens the file in read-only mode by default**. Here, we use the `write()` method on the file object to write a string to the file.

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!

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!

In [13]:
```
with open('text.txt', 'w') as file_object:
    file_object.write(12)
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_53596\1310681829.py in <module>
      1 with open('text.txt', 'w') as file_object:
----> 2     file_object.write(12)

TypeError: write() argument must be str, not int
```

While reading through a file, the system maintains a ***file-position pointer*** (index) representing the location of the next character to read. Therefore, the following code snippet will allow you to append it to the end of the file.

While reading through a file, the system maintains a ***file-position pointer*** (index) representing the location of the next character to read. Therefore, the following code snippet will allow you to append it to the end of the file.

In [14]:
```
year = 2025

with open(filename, 'r+') as file_object: # The file should already be create
    spam = file_object.readlines()
    print(file_object.tell())
    print(len("We love programming!"))
    file_object.write(str(year))
```

```
20
20
```

While reading through a file, the system maintains a ***file-position pointer*** (index) representing the location of the next character to read. Therefore, the following code snippet will allow you to append it to the end of the file.

In [14]:
```
year = 2025

with open(filename, 'r+') as file_object: # The file should already be create
    spam = file_object.readlines()
    print(file_object.tell())
    print(len("We love programming!"))
    file_object.write(str(year))
```

```
20
20
```

The `tell()` function will return the current position of the file-position pointer. We can also use `seek()` to change the position. Checkout more details about file-position pointer here.

Appending to a File

If you want to add content to a file instead of writing over existing content, you can also open the file in **append mode**. When you open a file in append mode, Python doesn't erase the contents of the file before returning the file object.

If you want to add content to a file instead of writing over existing content, you can also open the file in **append mode**. When you open a file in append mode, Python doesn't erase the contents of the file before returning the file object.

Any lines you write to the file will be added at the end of the file. If the file doesn't exist yet, Python will create an empty file for you.

If you want to add content to a file instead of writing over existing content, you can also open the file in **append mode**. When you open a file in append mode, Python doesn't erase the contents of the file before returning the file object.

Any lines you write to the file will be added at the end of the file. If the file doesn't exist yet, Python will create an empty file for you.

In [15]:
```
filename = 'programming.txt'

with open(filename, 'a') as file_object:
    file_object.write("\nWe also love finding meaning in large datasets.\n")
    file_object.write("We love creating apps that can run in a browser.\n")
```

If you want to add content to a file instead of writing over existing content, you can also open the file in **append mode**. When you open a file in append mode, Python doesn't erase the contents of the file before returning the file object.

Any lines you write to the file will be added at the end of the file. If the file doesn't exist yet, Python will create an empty file for you.

In [15]:
```
filename = 'programming.txt'

with open(filename, 'a') as file_object:
    file_object.write("\nWe also love finding meaning in large datasets.\n")
    file_object.write("We love creating apps that can run in a browser.\n")
```

The `write()` function doesn't add any newlines to the text you write. So we need to add newline characters if we would like to. There is also a `writelines()` function that can write list of strings into files.

In [16]:
```
display_quiz(path+"write.json", max_width=800)
```

What is the final content of the file 'data.txt' after executing the following code?

```
with open('data.txt', 'w') as f:
    f.write('first line\n')

with open('data.txt', 'w') as f:
    f.write('second line')
```

First line

Second line

```
First line

Second lin
e
```

```
First line
Second lin
e
```

Storing Data

Many of your programs will ask users to input certain kinds of information. You might allow users to store preferences in a game or provide data for visualization.

Many of your programs will ask users to input certain kinds of information. You might allow users to store preferences in a game or provide data for visualization.

Whatever the focus of your program is, you'll store the information users provide in data structures such as `lists` and `dictionaries`. When users close a program, you'll almost always want to save the information they entered. A simple way to do this involves storing your data using the `json` module.

Many of your programs will ask users to input certain kinds of information. You might allow users to store preferences in a game or provide data for visualization.

Whatever the focus of your program is, you'll store the information users provide in data structures such as `lists` and `dictionaries`. When users close a program, you'll almost always want to save the information they entered. A simple way to do this involves storing your data using the `json` module.

The ***`json`*** module allows you to dump simple Python data structures into a file and load the data from that file the next time the program runs. **You can also use `json` to share data between different programming languages. It's a useful and portable format.**

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

The `json.dump()` function takes two arguments: a piece of data to store and a file object it can use to store the data.

The `json.dump()` function takes two arguments: a piece of data to store and a file object it can use to store the data.

In [17]:
```
import json

numbers = [2, 3, 5, 7, 11, 13]
filename = 'prime_numbers.json'

with open(filename, 'w') as f:
    json.dump(numbers, f)
```

Now we'll write a program that uses `json.load()` to read the list back into memory

Now we'll write a program that uses `json.load()` to read the list back into memory

In [18]:
```
filename = 'numbers.json'
with open(filename) as f:
    numbers = json.load(f)

print(numbers)
```

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

Exceptions

Python uses special objects called ***exceptions*** to manage errors that arise during a program's execution. Whenever an error makes Python unsure of what to do next, it creates an `exception` object. If we write code that handles the exception, the program will continue running. If you don't handle the exception, the program will halt and show a ***traceback***, which includes a report of the exception that was raised.

Python uses special objects called ***exceptions*** to manage errors that arise during a program's execution. Whenever an error makes Python unsure of what to do next, it creates an `exception` object. If we write code that handles the exception, the program will continue running. If you don't handle the exception, the program will halt and show a ***traceback***, which includes a report of the exception that was raised.

Exceptions are handled with `try-except` blocks. A `try-except` block tells Python what to do if an exception is raised.

Python uses special objects called ***exceptions*** to manage errors that arise during a program's execution. Whenever an error makes Python unsure of what to do next, it creates an `exception` object. If we write code that handles the exception, the program will continue running. If you don't handle the exception, the program will halt and show a ***traceback***, which includes a report of the exception that was raised.

Exceptions are handled with `try-except` blocks. A `try-except` block tells Python what to do if an exception is raised.

When we use `try-except` blocks, our programs will continue running even if things go wrong. Instead of tracebacks, which can be confusing for users to read, users will see friendly error messages that we write!

Handling the `ZeroDivisionError` Exception

In [19]:
```
print(5/0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_53596\1152173066.py in <module>
----> 1 print(5/0)

ZeroDivisionError: division by zero
```

In [19]:
```
print(5/0)
```

```
---------------------------------------------------------------------
-----
ZeroDivisionError                         Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_53596\1152173066.py in <module>
----> 1 print(5/0)

ZeroDivisionError: division by zero
```

The error reported at the first line in the traceback, `ZeroDivisionError`, is an exception object. Python creates this kind of object in response to a situation where it can't do what we ask. When this happens, Python stops the program and tells us the kind of exception that was raised. We can use this information to modify our program.

When we think an error may occur, you can write a `try-except` block to handle the exception that might be raised. We tell Python to try running some code and tell it what to do if the code results in a particular kind of exception.

When we think an error may occur, you can write a `try-except` block to handle the exception that might be raised. We tell Python to try running some code and tell it what to do if the code results in a particular kind of exception.

In [20]:
```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

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

We put `print(5/0)`, the line that caused the error, inside a `try` block. If the code in a `try` block works, Python skips over the `except` block. If the code in the `try` block causes an error, Python looks for an `except` block whose error matches the one that was raised and ran the code in that block. In this example, the user sees a friendly error message instead of a traceback.

We put `print(5/0)`, the line that caused the error, inside a `try` block. If the code in a `try` block works, Python skips over the `except` block. If the code in the `try` block causes an error, Python looks for an `except` block whose error matches the one that was raised and ran the code in that block. In this example, the user sees a friendly error message instead of a traceback.

In [21]:
```
try:
    print(5/0)
except:
    print("Exceptions occur!")
```

```
Exceptions occur!
```

We put `print(5/0)`, the line that caused the error, inside a `try` block. If the code in a `try` block works, Python skips over the `except` block. If the code in the `try` block causes an error, Python looks for an `except` block whose error matches the one that was raised and ran the code in that block. In this example, the user sees a friendly error message instead of a traceback.

In [21]:
```
try:
    print(5/0)
except:
    print("Exceptions occur!")
```

```
Exceptions occur!
```

If you do not add any exception type, it will capture all exceptions!

Using Exceptions to Prevent Crashes

Handling errors correctly is especially important when the program has more work to do after the error occurs. Let's create a simple calculator that does only division:

Handling errors correctly is especially important when the program has more work to do after the error occurs. Let's create a simple calculator that does only division:

In [ ]:
```
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: ")
    if second_number == 'q':
        break
    try:
        answer = int(first_number) / int(second_number)
    except ZeroDivisionError:
        print("You can't divide by 0!")
    else: # Only executed if try block is succeed
        print(answer)
    finally: # Always executed
        print("\nGive me two numbers, and I'll divide them.")
        print("Enter 'q' to quit.")
```

Here, the error may occur on the line that performs the division, so that's where we'll put the `try-except` block. This example also includes an `else` block.

Here, the error may occur on the line that performs the division, so that's where we'll put the `try-except` block. This example also includes an `else` block.

Any code that depends on the `try` block executing successfully goes into the `else` block. In addition, the `finally` clause is guaranteed to execute, regardless of whether its `try` suite executes successfully or an exception occurs.

Here, the error may occur on the line that performs the division, so that's where we'll put the `try-except` block. This example also includes an `else` block.

Any code that depends on the `try` block executing successfully goes into the `else` block. In addition, the `finally` clause is guaranteed to execute, regardless of whether its `try` suite executes successfully or an exception occurs.

We ask Python to try to complete the division operation in a `try` block, which includes only the code that might cause an error. The program continues to run, and the user never sees a traceback.

Handling the `FileNotFoundError` Exception

One common issue when working with files is handling missing files. The file you're looking for might be in a different location, the filename may be misspelled, or the file may not exist at all!

One common issue when working with files is handling missing files. The file you're looking for might be in a different location, the filename may be misspelled, or the file may not exist at all!

In [22]:
```
filename = 'alice.txt'
with open(filename) as f: # Note it is in read mode
    contents = f.read()
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_53596\1385916738.py in <module>
      1 filename = 'alice.txt'
----> 2 with open(filename) as f: # Note it is in read mode
      3     contents = f.read()

FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'
```

In [23]:
```
filename = 'alice.txt'
try:
    with open(filename) as f:
        contents = f.read()
except FileNotFoundError:
    print(f"Sorry, the file {filename} does not exist.")
```

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

In [23]:
```
filename = 'alice.txt'
try:
    with open(filename) as f:
        contents = f.read()
except FileNotFoundError:
    print(f"Sorry, the file {filename} does not exist.")
```

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

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.

In [24]:
```
display_quiz(path+"exception.json", max_width=800)
```

What is printed by the following statements?

```
try:
    x = int("abc")
    print("Conversion succeeded")
except ValueError:
    print("Conversion failed")
else:
    print("No exception occurred")
finally:
    print("Execution finished")
```

```
Conversion failed
Execution finished
```

```
Conversion succeeded
Execution finished
```

```
Conversion failed
No exception occurred
Execution finished
```

```
Conversion succeeded
No exception occurred
Execution finished
```

Exercise 1: Assuming we are designing a word game called "The Mysterious Island" and we need to load the statistics of the player and enemies each time the game begins. Try to complete the following functions `load_data()` and `save_data()` so that you can load the JSON file if it does not exist using the exception handling techniques you just learned.

In [ ]:

```
def battle(player, enemy):
    slow_print(f"You encounter a {enemy['name']} with {enemy['hp']} HP!")
    while player['hp'] > 0 and enemy['hp'] > 0:
        choice = input("Do you want to attack or escape? (a/e): ").lower()
        if choice == 'a':
            player_damage = max(random.randint(player['attack'] // 2, player[
            enemy['hp'] -= player_damage
            slow_print(f"You deal {player_damage} damage to the {enemy['name'
            if enemy['hp'] <= 0:
                break
            enemy_damage = max(random.randint(enemy['attack'] // 2, enemy['at
            player['hp'] -= enemy_damage
            slow_print(f"The {enemy['name']} deals {enemy_damage} damage to y
        elif choice == 'e':
            if random.randint(1, 10) <= 2:
                slow_print("You successfully escape from the battle!")
                return
            else:
                slow_print("You failed to escape!")
                enemy_damage = max(random.randint(enemy['attack'] // 2, enemy
                player['hp'] -= enemy_damage
                slow_print(f"The {enemy['name']} deals {enemy_damage} damage
        else:
            slow_print("Invalid choice! Try again.")
    if player['hp'] <= 0:
        slow_print("You were defeated!\nGame over!")
        return False
    else:
        slow_print(f"You defeated the {enemy['name']}!")
```

In [ ]:

```python
import json

def load_data():
    _____:  # Try to load existing game data
        with open('game_data.json', 'r') as f:
            data = _______ # Read the data here using json.load()
    _________________:  # If no saved data exists, initialize default data
        data = {
            "player": {
                "name": "Player",
                "hp": 50,
                "attack": 10
            },
            "enemies": [
                {"name": "Arbok", "hp": 10, "attack": 5},
                {"name": "Gengar", "hp": 25, "attack": 8},
                {"name": "Dragonite", "hp": 80, "attack": 15}
            ]
        }
        save_data(data)
    return data
```

In [ ]:

```
import random, time
def slow_print(text, delay=0.05):
    for char in text:
        print(char, end='', flush=True)
        time.sleep(delay)
    print()

def save_data(data):
    with open('game_data.json', 'w') as f:
        _______(____,____)# Save the data so that you can play again using j
```

In [ ]:

```
data = load_data()
player = data['player']
enemies = data['enemies']
flag = True

for enemy in enemies:
    flag = battle(player, enemy)
    if flag == False:
        break

if flag != False:
    print("Congratulations!")
```

In [ ]:
```
from jupytercards import display_flashcards
fpath= "https://raw.githubusercontent.com/phonchi/nsysu-math106A/refs/heads/m
display_flashcards(fpath + 'ch7.json')
```