Beginner's Python Cheat Sheet - Files and Exceptions

Why work with files? Why use exceptions?

Your programs can read information in from files, and they can write data to files. Reading from files allows you to work with a wide variety of information; writing to files allows users to pick up where they left off the next time they run your program. You can write text to files, and you can store Python structures such as lists in data files as well.

Exceptions are special objects that help your programs respond to errors in appropriate ways. For example if your program tries to open a file that doesn't exist, you can use exceptions to display an informative error message instead of having the program crash.

Reading from a file

To read from a file your program needs to specify the path to the file, and then read the contents of the file. The `read_text()` *method returns a string containing the entire contents of the file.*

Reading an entire file at once

```
from pathlib import Path

path = Path('siddhartha.txt')
contents = path.read_text()

print(contents)
```

Working with a file's lines

It's often useful to work with individual lines from a file. Once the contents of a file have been read, you can get the lines using the `splitlines()` *method.*

```
from pathlib import Path

path = Path('siddhartha.txt')
contents = path.read_text()

lines = contents.splitlines()

for line in lines:
    print(line)
```

Writing to a file

The `write_text()` *method can be used to write text to a file. Be careful, this will write over the current file if it already exists. To append to a file, read the contents first and then rewrite the entire file.*

Writing to a file

```
from pathlib import Path

path = Path("programming.txt")

msg = "I love programming!"
path.write_text(msg)
```

Writing multiple lines to a file

```
from pathlib import Path

path = Path("programming.txt")

msg = "I love programming!"
msg += "\nI love making games."
path.write_text(msg)
```

Appending to a file

```
from pathlib import Path

path = Path("programming.txt")
contents = path.read_text()

contents += "\nI love programming!"
contents += "\nI love making games."
path.write_text(contents)
```

Path objects

The `pathlib` *module makes it easier to work with files in Python. A* `Path` *object represents a file or directory, and lets you carry out common directory and file operations.*

With a relative path, Python usually looks for a location relative to the .py file that's running. Absolute paths are relative to your system's root folder ("/").

Windows uses backslashes when displaying file paths, but you should use forward slashes in your Python code.

Relative path

```
path = Path("text_files/alice.txt")
```

Absolute path

```
path = Path("/Users/eric/text_files/alice.txt")
```

Get just the filename from a path

```
>>> path = Path("text_files/alice.txt")
>>> path.name
'alice.txt'
```

Path objects (cont.)

Build a path

```
base_path = Path("/Users/eric/text_files")
file_path = base_path / "alice.txt"
```

Check if a file exists

```
>>> path = Path("text_files/alice.txt")
>>> path.exists()
True
```

Get filetype

```
>>> path.suffix
'.txt'
```

The try-except block

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

Handling the `ZeroDivisionError` exception

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

Handling the `FileNotFoundError` exception

```
from pathlib import Path

path = Path("siddhartha.txt")
try:
    contents = path.read_text()
except FileNotFoundError:
    msg = f"Can't find file: {path.name}."
    print(msg)
```

Knowing which exception to handle

It can be hard to know what kind of exception to handle when writing code. Try writing your code without a `try` *block, and make it generate an error. The traceback will tell you what kind of exception your program needs to handle. It's a good idea to skim through the exceptions listed at docs.python.org/3/library/exceptions.html.*

The else block

The `try` block should only contain code that may cause an error. Any code that depends on the `try` block running successfully should be placed in the else block.

Using an `else` block

```
print("Enter two numbers. I'll divide them.")

x = input("First number: ")
y = input("Second number: ")

try:
    result = int(x) / int(y)
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print(result)
```

Preventing crashes caused by user input

Without the `except` block in the following example, the program would crash if the user tries to divide by zero. As written, it will handle the error gracefully and keep running.

```
"""A simple calculator for division only."""

print("Enter two numbers. I'll divide them.")
print("Enter 'q' to quit.")

while True:
    x = input("\nFirst number: ")
    if x == 'q':
        break

    y = input("Second number: ")
    if y == 'q':
        break

    try:
        result = int(x) / int(y)
    except ZeroDivisionError:
        print("You can't divide by zero!")
    else:
        print(result)
```

Deciding which errors to report

Well-written, properly tested code is not very prone to internal errors such as syntax or logical errors. But every time your program depends on something external such as user input or the existence of a file, there's a possibility of an exception being raised.

It's up to you how to communicate errors to your users. Sometimes users need to know if a file is missing; sometimes it's better to handle the error silently. A little experience will help you know how much to report.

Failing silently

Sometimes you want your program to just continue running when it encounters an error, without reporting the error to the user. Using the `pass` statement in an `except` block allows you to do this.

Using the `pass` statement in an `except` block

```
from pathlib import Path

f_names = ['alice.txt', 'siddhartha.txt',
        'moby_dick.txt', 'little_women.txt']

for f_name in f_names:
    # Report the length of each file found.
    path = Path(f_name)
    try:
        contents = path.read_text()
    except FileNotFoundError:
        pass
    else:
        lines = contents.splitlines()
        msg = f"{f_name} has {len(lines)}"
        msg += " lines."
        print(msg)
```

Avoid bare except blocks

Exception handling code should catch specific exceptions that you expect to happen during your program's execution. A bare `except` block will catch all exceptions, including keyboard interrupts and system exits you might need when forcing a program to close.

If you want to use a `try` block and you're not sure which exception to catch, use `Exception`. It will catch most exceptions, but still allow you to interrupt programs intentionally.

Don't use bare except blocks

```
try:
    # Do something
except:
    pass
```

Use `Exception` instead

```
try:
    # Do something
except Exception:
    pass
```

Printing the exception

```
try:
    # Do something
except Exception as e:
    print(e, type(e))
```

Storing data with json

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. The JSON data format is not specific to Python, so you can share this kind of data with people who work in other languages as well.

Knowing how to manage exceptions is important when working with stored data. You'll usually want to make sure the data you're trying to load exists before working with it.

Using `json.dumps()` to store data

```
from pathlib import Path
import json

numbers = [2, 3, 5, 7, 11, 13]

path = Path("numbers.json")
contents = json.dumps(numbers)
path.write_text(contents)
```

Using `json.loads()` to read data

```
from pathlib import Path
import json

path = Path("numbers.json")
contents = path.read_text()
numbers = json.loads(contents)

print(numbers)
```

Making sure the stored data exists

```
from pathlib import Path
import json

path = Path("numbers.json")

try:
    contents = path.read_text()
except FileNotFoundError:
    msg = f"Can't find file: {path}"
    print(msg)
else:
    numbers = json.loads(contents)
    print(numbers)
```

Practice with exceptions

Take a program you've already written that prompts for user input, and add some error-handling code to the program. Run your program with appropriate and inappropriate data, and make sure it handles each situation correctly.