6 Exceptions and File Input/Ouput

Lab Objective: *In Python, an* exception *is an error detected during execution. Exceptions are important for regulating program usage and for correctly reporting problems to the programmer and end user. An understanding of exceptions is essential to safely read data from and write data to external files, and being able to interact with external files is important for analyzing data and communicating results. In this lab we learn exception syntax and file interaction protocols.*

Exceptions

An *exception* formally indicates an error and terminates the program early. Some of the more common exception types are listed below, along with the kinds of problems that they typically indicate.

Exception	Indication
`AttributeError`	An attribute reference or assignment failed.
`ImportError`	An `import` statement failed.
`IndexError`	A sequence subscript was out of range.
`NameError`	A local or global name was not found.
`TypeError`	An operation or function was applied to an object of inappropriate type.
`ValueError`	An operation or function received an argument that had the right type but an inappropriate value.
`ZeroDivisionError`	The second argument of a division or modulo operation was zero.

```
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined

>>> [1, 2, 3].fly()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'fly'
```

Raising Exceptions

Most exceptions are due to coding mistakes and typos. However, exceptions can also be used intentionally to indicate a problem to the user or programmer. To create an exception, use the keyword `raise`, followed by the name of the exception class. As soon as an exception is raised, the program stops running unless the exception is handled properly.

```
>>> if 7 is not 7.0:                # Raise an exception with an error message.
...     raise Exception("ints and floats are different!")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: ints and floats are different!

>>> for x in range(10):
...     if x > 5:                   # Raise a specific kind of exception.
...         raise ValueError("'x' should not exceed 5.")
...     print(x, end=' ')
...
0 1 2 3 4 5
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ValueError: 'x' should not exceed 5.
```

Problem 1. Consider the following arithmetic "magic" trick.

1. Choose a 3-digit number where the first and last digits differ by 2 or more (say, 123).
2. Reverse this number by reading it backwards (321).
3. Calculate the positive difference of these numbers ($321 - 123 = 198$).
4. Add the reverse of the result to itself ($198 + 891 = 1089$).

The result of the last step will always be 1089, regardless of the original number chosen in step 1 (can you explain why?).

The following function prompts the user for input at each step of the magic trick, but does not check that the user's inputs are correct.

```
def arithmagic():
    step_1 = input("Enter a 3-digit number where the first and last "
                                          "digits differ by 2 or more: ")
    step_2 = input("Enter the reverse of the first number, obtained "
                                            "by reading it backwards: ")
    step_3 = input("Enter the positive difference of these numbers: ")
    step_4 = input("Enter the reverse of the previous result: ")
    print(str(step_3), "+", str(step_4), "= 1089 (ta-da!)")
```

Modify `arithmagic()` so that it verifies the user's input at each step. Raise a `ValueError` with an informative error message if any of the following occur:

- The first number (`step_1`) is not a 3-digit number.
- The first number's first and last digits differ by less than 2.
- The second number (`step_2`) is not the reverse of the first number.
- The third number (`step_3`) is not the positive difference of the first two numbers.
- The fourth number (`step_4`) is not the reverse of the third number.

(Hint: `input()` always returns a string, so each variable is a string initially. Use `int()` to cast the variables as integers when necessary. The built-in function `abs()` may also be useful.)

Handling Exceptions

To prevent an exception from halting the program, it must be handled by placing the problematic lines of code in a `try` block. An `except` block then follows with instructions for what to do in the event of an exception.

```
# The 'try' block should hold any lines of code that might raise an exception.
>>> try:
...     print("Entering try block...")
...     raise Exception("for no reason")
...     print("No problem!")        # This line gets skipped.
... # The 'except' block is executed just after the exception is raised.
... except Exception as e:
...     print("There was a problem:", e)
...
Entering try block...
There was a problem: for no reason
>>> # The program then continues on.
```

In this example, the name `e` represents the exception within the `except` block. Printing `e` displays its error message. If desired, `e` can be raised again with `raise e` or just `raise`.

The try-except control flow can be expanded with two other blocks, forming a code structure similar to a sequence of `if-elif-else` blocks.

1. The `try` block is executed until an exception is raised (if at all).
2. An `except` statement specifying the same kind of exception that was raised in the try block "catches" the exception, and the block is then executed. There may be multiple except blocks following a single try block (similiar to having several `elif` statements following a single `if` statement), and a single except statement may specify multiple kinds of exceptions to catch.
3. The `else` block is executed if an exception was **not** raised in the try block.
4. The `finally` block is always executed if it is included.

```
>>> try:
...     print("Entering try block...", end='')
...     house_on_fire = False
...     raise ValueError("The house is on fire!")
... # Check for multiple kinds of exceptions using parentheses.
... except (ValueError, TypeError) as e:
...     print("caught an exception.")
...     house_on_fire = True
... else:                           # Skipped due to the exception.
...     print("no exceptions raised.")
... finally:
...     print("The house is on fire:", house_on_fire)
...
Entering try block...caught an exception.
The house is on fire: True

>>> try:
...     print("Entering try block...", end='')
...     house_on_fire = False
... except ValueError as e:         # Skipped because there was no exception.
...     print("caught a ValueError.")
...     house_on_fire = True
... except TypeError as e:          # Also skipped.
...     print("caught a TypeError.")
...     house_on_fire = True
... else:
...     print("no exceptions raised.")
... finally:
...     print("The house is on fire:", house_on_fire)
...
Entering try block...no exceptions raised.
The house is on fire: False
```

The code in the `finally` block is always executed, even if a `return` statement or an uncaught exception occurs in any block following the try statement.

```
>>> def implode():
...     try:                       # Try to return immediately...
...         return
...     finally:                   # ...but 'finally' goes before 'return'.
...         print("Goodbye, world!")
...
>>> implode()
Goodbye, world!
```

See `https://docs.python.org/3/tutorial/errors.html` for more examples.

Achtung!

An `except` statement with no specified exception type catches **any** exception raised in the corresponding `try` block. This approach can mistakenly mask unexpected errors. Always be specific about the kinds of exceptions you expect to encounter.

```
>>> def divider(x, y):
...     try:
...         return x / yy           # The mispelled yy raises a NameError.
...     except:                     # Catch ANY exception.
...         print("y must not equal zero!")
...
>>> divider(2, 3)
y must not equal zero!

>>> def divider(x, y):
...     try:
...         return x / yy
...     except ZeroDivisionError:   # Specify an exception type.
...         print("y must not equal zero!")
...
>>> divider(2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divider
NameError: name 'yy' is not defined     # Now the mistake is obvious.
```

Problem 2. A *random walk* is a path created by a sequence of random steps. The following function simulates a random walk by repeatedly adding or subtracting 1 to a running total.

```
from random import choice

def random_walk(max_iters=1e12):
    walk = 0
    directions = [1, -1]
    for i in range(int(max_iters)):
        walk += choice(directions)
    return walk
```

A `KeyboardInterrupt` is a special exception that can be triggered at any time by entering `ctrl+c` (on most systems) in the keyboard. Modify `random_walk()` so that if the user raises a `KeyboardInterrupt` by pressing `ctrl+c` while the program is running, the function catches the exception and prints "Process interrupted at iteration i". If no `KeyboardInterrupt` is raised, print "Process completed". In both cases, return `walk` as before.

Note

The built-in exceptions are organized into a class hierarchy. For example, the `ValueError` class inherits from the generic `Exception` class. Thus a `ValueError` **is** a `Exception`, but a `Exception` is **not** a `ValueError`.

```
>>> try:
...     raise ValueError("caught!")
... except Exception as e:          # A ValueError is a Exception.
...     print(e)
...
caught!                             # The exception was caught.

>>> try:
...     raise Exception("not caught!")
... except ValueError as e:         # A Exception is not a ValueError.
...     print(e)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: not caught!            # The exception wasn't caught!
```

See `https://docs.python.org/3/library/exceptions.html` for the complete list of built-in exceptions and the exception class hierarchy.

File Input and Output

A file object acts as an interface to a file stream, meaning it allows a program to read from or write to external files. The built-in function `open()` creates a file object. It accepts the name of the file to open and an editing mode. The mode determines the kind of access that the user has to the file. There are four common modes:

- `'r'`: **read**. Open an existing file for reading. The file must already exist, or `open()` raises a `FileNotFoundError`. This is the default mode.
- `'w'`: **write**. Create a new file or **overwrite an existing file** (careful!) and open it for writing.
- `'x'`: **write new**. Create a new file and open it for writing. If the file already exists, `open()` raises a `FileExistsError`. This is a safer form of `'w'` because it never overwrites existing files.
- `'a'`: **append**. Open a file for writing, appending new data to the end of the file if it already exists.

```
>>> myfile = open("hello_world.txt", 'r')   # Open a file for reading.
>>> print(myfile.read())                    # Print the contents of the file.
Hello,                                      # (it's a really small file.)
World!

>>> myfile.close()                          # Close the file connection.
```

The With Statement

An `IOError` indicates that some input or output operation has failed. A simple `try-finally` control flow can ensure that a file stream is closed safely.

The `with` statement provides an alternative method for safely opening and closing files. Use `with open(<filename>, <mode>) as <alias>:` to create an indented block in which the file is open and available under the specified alias. At the end of the block, the file is automatically and safely closed, even in the event of an exception. This is the preferred file-reading method when a file only needs to be accessed briefly.

```
>>> myfile = open("hello_world.txt", 'r')   # Open a file for reading.
>>> try:
...     contents = myfile.readlines()       # Read in the content by line.
... finally:
...     myfile.close()                      # Explicitly close the file.

# Equivalently, use a 'with' statement to take care of errors.
>>> with open("hello_world.txt", 'r') as myfile:
...    contents = myfile.readlines()
...                                        # The file is closed automatically.
```

In both cases, if the file `hello_world.txt` does not exist in the current directory, `open()` raises a `FileNotFoundError`. However, errors in the `try` or `with` blocks do not prevent the file from being safely closed.

Reading and Writing

Open file objects have an implicit *cursor* that determines the location in the file to read from or write to. After the entire file has been read once, either the file must be closed and reopened, or the cursor must be reset to the beginning of the file with `seek(0)` before it can be read again.

Some of more important file object attributes and methods are listed below.

Attribute	Description
`closed`	`True` if the object is closed.
`mode`	The access mode used to open the file object.
`name`	The name of the file.

Method	Description
`close()`	Close the connection to the file.
`read()`	Read a given number of bytes; with no input, read the entire file.
`readline()`	Read a line of the file, including the newline character at the end.
`readlines()`	Call `readline()` repeatedly and return a list of the resulting lines.
`seek()`	Move the cursor to a new position.
`tell()`	Report the current position of the cursor.
`write()`	Write a single string to the file (spaces are **not** added).
`writelines()`	Write a list of strings to the file (newline characters are **not** added).

Only strings can be written to files; to write a non-string type, first cast it as a string with `str()`. Be mindful of spaces and newlines to separate the data.

```
>>> with open("out.txt", 'w') as outfile:    # Open 'out.txt' for writing.
...     for i in range(10):
...         outfile.write(str(i**2)+' ')     # Write some strings (and spaces).
...
>>> outfile.closed                           # The file is closed automatically.
True
```

Problem 3. Define a class called `ContentFilter`. Implement the constructor so that it accepts the name of a file to be read.

1. If the file name is invalid in any way, prompt the user for another filename using `input()`. Continue prompting the user until they provide a valid filename.

   ```
   >>> cf1 = ContentFilter("hello_world.txt")  # File exists.
   >>> cf2 = ContentFilter("not-a-file.txt")   # File doesn't exist.
   Please enter a valid file name: still-not-a-file.txt
   Please enter a valid file name: hello_world.txt
   >>> cf3 = ContentFilter([1, 2, 3])          # Not even a string.
   Please enter a valid file name: hello_world.txt
   ```

   (Hint: `open()` might raise a `FileNotFoundError`, a `TypeError`, or an `OSError`.)

2. Read the file and store its name and contents as attributes (store the contents as a single string). Make sure the file is securely closed.

String Formatting

The `str` class has several useful methods for parsing and formatting strings. They are particularly useful for processing data from a source file and for preparing data to be written to an external file.

Method	Returns
`count()`	The number of times a given substring occurs within the string.
`find()`	The lowest index where a given substring is found.
`isalpha()`	`True` if all characters in the string are alphabetic (a, b, c, ...).
`isdigit()`	`True` if all characters in the string are digits (0, 1, 2, ...).
`isspace()`	`True` if all characters in the string are whitespace (`" "`, `'\t'`, `'\n'`).
`join()`	The concatenation of the strings in a given iterable with a specified separator between entries.
`lower()`	A copy of the string converted to lowercase.
`upper()`	A copy of the string converted to uppercase.
`replace()`	A copy of the string with occurrences of a given substring replaced by a different specified substring.
`split()`	A list of segments of the string, using a given character or string as a delimiter.
`strip()`	A copy of the string with leading and trailing whitespace removed.

The `join()` method translates a list of strings into a single string by concatenating the entries of the list and placing the principal string between the entries. Conversely, `split()` translates the principal string into a list of substrings, with the separation determined by the a single input.

```
# str.join() puts the string between the entries of a list.
>>> words = ["state", "of", "the", "art"]
>>> "-".join(words)
'state-of-the-art'

# str.split() creates a list out of a string, given a delimiter.
>>> "One fish\nTwo fish\nRed fish\nBlue fish\n".split('\n')
['One fish', 'Two fish', 'Red fish', 'Blue fish', '']

# If no delimiter is provided, the string is split by whitespace characters.
>>> "One fish\nTwo fish\nRed fish\nBlue fish\n".split()
['One', 'fish', 'Two', 'fish', 'Red', 'fish', 'Blue', 'fish']
```

Can you tell the difference between the following routines?

```
>>> with open("hello_world.txt", 'r') as myfile:
...     contents = myfile.readlines()
...
>>> with open("hello_world.txt", 'r') as myfile:
...     contents = myfile.read().split('\n')
```

Problem 4. Add the following methods to the `ContentFilter` class for writing the contents of the original file to new files. Each method should accept a the name of a file to write to and a keyword argument `mode` that specifies the file access mode, defaulting to `'w'`. If `mode` is not `'w'`, `'x'`, or `'a'`, raise a `ValueError` with an informative message.

1. `uniform()`: write the data to the outfile with uniform case. Include an additional keyword argument `case` that defaults to `"upper"`.

   If `case="upper"`, write the data in upper case. If `case="lower"`, write the data in lower case. If `case` is not one of these two values, raise a `ValueError`.

2. `reverse()`: write the data to the outfile in reverse order. Include an additional keyword argument `unit` that defaults to `"line"`.

   If `unit="word"`, reverse the ordering of the words in each line, but write the lines in the same order as the original file. If `unit="line"`, reverse the ordering of the lines, but do not change the ordering of the words on each individual line. If `unit` is not one of these two values, raise a `ValueError`.

3. `transpose()`: write a "transposed" version of the data to the outfile. That is, write the first word of each line of the data to the first line of the new file, the second word of each line of the data to the second line of the new file, and so on. Viewed as a matrix of words, the rows of the input file then become the columns of the output file, and vice versa. You may assume that there are an equal number of words on each line of the input file.

Also implement the `__str__()` magic method so that printing a `ContentFilter` object yields the following output. You may want to calculate these statistics in the constructor.

```
Source file:              <filename>
Total characters:         <The total number of characters in the file>
Alphabetic characters:    <The number of letters>
Numerical characters:     <The number of digits>
Whitespace characters:    <The number of spaces, tabs, and newlines>
Number of lines:          <The number of lines>
```

(Hint: list comprehensions are **very** useful for some of these functions. For example, what does `[line[::-1] for line in lines]` do? What about `sum([s.isspace() for s in data])`?)

Compare your class to the following example.

```
# cf_example1.txt
A b C
d E f
```

```
>>> cf = ContentFilter("cf_example1.txt")
>>> cf.uniform("uniform.txt", mode='w', case="upper")
>>> cf.uniform("uniform.txt", mode='a', case="lower")
>>> cf.reverse("reverse.txt", mode='w', unit="word")
>>> cf.reverse("reverse.txt", mode='a', unit="line")
>>> cf.transpose("transpose.txt", mode='w')
```

```
# uniform.txt
A B C
D E F
a b c
d e f
```

```
# reverse.txt
C b A
f E d
d E f
A b C
```

```
# transpose.txt
A d
b E
C f
```

Additional Material

Custom Exception Classes

Custom exceptions can be defined by writing a class that inherits from some existing exception class. The generic `Exception` class is typically the parent class of choice.

```
>>> class TooHardError(Exception):
...     pass
...
>>> raise TooHardError("This lab is impossible!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.TooHardError: This lab is impossible!
```

This may seem like a trivial extension of the `Exception` class, but it is useful to do because the interpreter never automatically raises a `TooHardError`. Any `TooHardError` must have originated from a hand-written `raise` command, making it easier to identify the exact source of the problem.

Chaining Exceptions

Sometimes, especially in large programs, it is useful raise one kind of exception just after catching another. The two exceptions can be linked together using the `from` statement. This syntax makes it possible to see where the error originated from and to "pass it up" to another part of the program (**warning**: this feature was added in Python 3).

```
>>> try:
...     raise TooHardError("This lab is impossible!")
... except TooHardError as e:
...     raise NotImplementedError("Lab is incomplete") from e
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
__main__.TooHardError: This lab is impossible!

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NotImplementedError: Lab is incomplete
```

More String Formatting Tools

Concatenating string values with non-string values can cumbersome and tedious. The `str` class's `format()` method makes it easier to insert non-string values into the middle of a string. Write the desired output in its entirety, replacing non-string values with curly braces `{}`. Then use the `format()` method, entering each replaced value in order.

```
# Join the data using string concatenation.
>>> day, month, year = 10, "June", 2017
>>> print("Is today", day, str(month) + ',', str(year) + "?")
Is today 10 June, 2017?

# Join the data using str.format().
>>> print("Is today {} {}, {}?".format(day, month, year))
Is today 10 June, 2017?
```

This method is extremely flexible and provides many convenient ways to format string output nicely. Consider the following code for printing out a simple progress bar from within a loop.

```
>>> iters = int(1e7)
>>> chunk = iters // 20
>>> for i in range(iters):
...     print("\r[{:<20}] i = {}".format('='*((i//chunk)+1), i),
...                                                     end='', flush=True)
...
```

Here the string `"\r[{:<20}]"` used in conjunction with the `format()` method tells the cursor to go back to the beginning of the line, print an opening bracket, then print the first argument of `format()` left-aligned with at least 20 total spaces before printing the closing bracket. The comma after the print command suppresses the automatic newline character, keeping the output of each individual print statement on the same line.

Printing at each iteration dramatically slows down the progression through the loop. How does the following code solve that problem?

```
>>> for i in range(iters):
...     if not i % chunk:
...         print("\r[{:<20}] i = {}".format('='*((i//chunk)+1),i),
...                                                     end='', flush=True)
...
```

See `https://docs.python.org/3/library/string.html#format-string-syntax` for more examples and specific syntax for using `str.format()`. For a more robust progress bar printer, research the `tqdm` module.

Standard Library Modules for I/O

The standard library has other tools for input and output operations. For details on each module, see `https://docs.python.org/3/library`.

Module	Description
`csv`	CSV (comma separated value) file writing and parsing.
`io`	Support for file objects and `open()`.
`os`	Communication with the operating system.
`os.path`	Common path operations such as checking for file existence.
`pickle`	Create portable serialized representations of Python objects.