Lab 6

Exceptions and File I/O

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. Understanding exceptions allows us to safely read data from and export data to external files, and being able to read from and write to files is important to analyzing data and communicating results. In this lab we present exception syntax and file interaction protocols.*

Exceptions

Every programming language has a formal way of indicating and handling errors. In Python, we raise and handle *exceptions*. Some of the more common exceptions are listed below, along with the kinds of problems that they typically indicate.

Exception	Indication
`AttributeError`	An attribute reference or assignment failed.
`IndexError`	A sequence subscript was out of range.
`NameError`	A local or global name was not found.
`SyntaxError`	The parser encountered a syntax error.
`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.

```
>>> 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'
```

See `https://docs.python.org/2/library/exceptions.html` for the complete list of Python's built-in exception classes.

Raising Exceptions

Many exceptions, like the ones demonstrated above, are due to coding mistakes and typos. 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.

Exception objects can be initialized with any number of arguments. These arguments are stored as a tuple attribute called `args`, which serves as the string representation of the object. We typically provide a single string detailing the reasons for the error.

```
# Raise a generic exception, without an error message.
>>> if 7 is not 7.0:
...     raise Exception
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception

# Now raise a more specific exception, with an error message included.
>>> for x in range(10):
...     if x > 5:
...         raise ValueError("'x' should not exceed 5.")
...     print(x),
...
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 (for example, 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 = raw_input("Enter a 3-digit number where the first and last "
                                        "digits differ by 2 or more: ")
    step_2 = raw_input("Enter the reverse of the first number, obtained "
                                          "by reading it backwards: ")
    step_3 = raw_input("Enter the positive difference of these numbers: ")
    step_4 = raw_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:

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

(Hint: `raw_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.

```
# The 'try' block should hold any lines of code that might raise an exception.
>>> try:
...     raise Exception("for no reason")
...     print "No exception raised"
... # The 'except' block is executed when an exception is raised.
... except Exception as e:
...     print "Exception raised", e
...
Exception raised 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.

The try-except control flow can be expanded with two other blocks. The flow proceeds as follows:

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 optional `else` block is executed if an exception was *not* raised in the try block. Thus either an except block or the else block is executed, but not both.
4. Lastly, the optional `finally` block is always executed if it is included.

```
>>> try:
...     raise ValueError("The house is on fire!")
... # Check for multiple kinds of exceptions using parentheses.
... except (ValueError, TypeError) as e:
...     house_on_fire = True
... else:                           # Skipped due to the exception.
...     house_on_fire = False
... finally:
...     print "The house is on fire:", house_on_fire
...
The house is on fire: True

>>> try:
...     house_on_fire = False
... except Exception as e:
...     house_on_fire = True
... else:                           # Executed because there was no exception.
...     print "The house is probably okay."
... finally:
...     print "The house is on fire:", house_on_fire
...
The house is probably okay.
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.

```
>>> try:
...     raise ValueError("The house is on fire!")
... finally:                        # Executes before the error is reported.
...     print "The house may be on fire."
...
The house may be on fire.
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: The house is on fire!
```

See `https://docs.python.org/2/tutorial/errors.html` for more examples.

Problem 2. A *random walk* is a path created by a series of random steps. The following function simulates a random walk where at every step, we either step forward (adding 1 to the total) or backward (adding -1).

```
from random import choice

def random_walk(max_iters=1e12):
    walk = 0
    direction = [1, -1]
    for i in xrange(int(max_iters)):
        walk += choice(direction)
    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`, the function handles 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 Python exceptions are organized into a class hierarchy. This can lead to some confusing behavior.

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

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

It turns out that the `ValueError` class inherits from the `StandardError` class. Thus a `ValueError` *is* a `StandardError`, so the first except statement was able to catch the exception, but a `StandardError` is *not* a `ValueError`, so the second except statement could not catch the exception.

The complete exception class hierarchy is documented at `https://docs.python.org/2/library/exceptions.html#exception-hierarchy`.

File Input and Output

Python has a useful `file` object that acts as an interface to all kinds of different file streams. 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 to use when opening the file. There are three common options:

`'r'`: **read**. This is the default mode. The file must already exist.

`'w'`: **write**. This mode creates the file if it doesn't already exist and **overwrites everything** in the file if it does already exist.

`'a'`: **append**. New data is written to the end of the file. This mode also creates a new file if it doesn't already exist.

```
>>> myfile = open("in.txt", 'r')        # Open 'in.txt' with read-only access.
>>> print(myfile.read())                # Print the contents of the file.
Hello World!                            # (it's a really small file.)
>>> myfile.close()                      # Close the file connection.
```

The With Statement

An `IOError` indicates that some input or output operation has failed. If a file cannot be opened for any reason, an `IOError` is raised and the file object is not initialized. 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 closed. This is the preferred file-reading method when a file only needs to be accessed briefly. The more flexible `try`-`finally` approach is typically better for working with several file streams at once.

```
>>> myfile = open("in.txt", 'r')        # Open 'in.txt' with read-only access.
>>> 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("in.txt", 'r') as myfile: # Open 'in.txt' with read-only access.
...    contents = myfile.readlines()    # Read in the content by line.
...                                     # The file is closed automatically.
```

In both cases, if the file `in.txt` does not exist in the current directory, an `IOError` will be raised. However, errors in the `try` or `with` blocks will 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 xrange(10):
...         outfile.write(str(i**2)+' ')    # Write some strings (and spaces).
...
>>> outfile.closed                          # The file is closed automatically.
True
```

Executing this code replaces whatever used to be in `out.txt` (whether or not it existed previously) with the following:

```
0 1 4 9 16 25 36 49 64 81
```

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 filename argument is not a string, raise a `TypeError`.

   (Hint: The built-in functions `type()` and `isinstance()` may be useful.)

2. Read the file and store its name and contents as attributes (store the contents as a single string). Securely close the file stream.

String Formatting

Python's `str` type 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'

>>> " o_O ".join(words)
'state o_O of o_O the o_O 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 its 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("in.txt", 'r') as myfile:
...     contents = myfile.readlines()
...
>>> with open("in.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 either `'w'` 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:

```
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>
```

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.

Assertions

An `AssertionError` is a special exception that is used primarily for software testing. The `assert` statement is a shortcut for testing the truthfulness of an expression and raising an `AssertionError`.

```
>>> assert <expression>, <message>

# Equivalently...
>>> if not <expression>:
...     raise AssertionError(<message>)
...
```

The CSV Module

The CSV format (comma separated value) is a common file format for spreadsheets and grid-like data. The `csv` module in the standard library contains functions that work in conjunction with the built-in `file` object to read from or write to CSV files. See `https://docs.python.org/2/library/csv.html` for details.

More String Formatting

Concatenating string values with non-string values is often cumbersome and tedious. Consider the problem of printing a simple date:

```
>>> day, month, year = 14, "March", 2015
>>> print("Is today " + str(day) + str(month) + ", " + str(year) + "?")
Is today 14 March, 2015?
```

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.

```
>>> print("Is today {} {}, {}?".format(day, month, year))
Is today 14 March, 2015?
```

This method is extremely flexible and provides many convenient ways to format string output nicely. Suppose, for example, that we would like to visualize the progress that a program is making through a loop. The following code prints out a simple status bar.

```
>>> from sys import stdout

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

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. Finally, `sys.stdout.flush()` flushes the internal buffer so that the status bar is printed in real time.

Of course, printing at each iteration dramatically slows down the progression through the loop. How does the following code solve that problem?

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

See `https://docs.python.org/2/library/string.html#format-string-syntax` for more examples and specific syntax for using `str.format()`.