Lab 6: File I/O

CSE/IT 107

NMT Department of Computer Science and Engineering

> "The danger that computers will become like humans is not as big as the danger that humans will become like computers." ("Die Gefahr, dass der Computer so wird wie der Mensch ist nicht so groß, wie die Gefahr, dass der Mensch so wird wie der Computer.")
>
> — Konrad Zuse

> "First, solve the problem. Then, write the code."
>
> — John Johnson

> "I dont need to waste my time with a computer just because I am a computer scientist."
>
> — Edsger W. Dijkstra

Introduction

In previous labs, we taught you how to use functions, math, lists, strings, and the like. We will combine all that in this lab and teach you how to interact with files. This will lead us to do some exciting data analysis!

Contents

1 File I/O

Knowing how to work with files is important, since it lets us store and retrieve data beyond the scope of the single execution of a program. To open a file for reading or writing we will use the `open()` function. The following example opens a file and writes "Hello World" to it.

```
output_file = open("hello.txt", "w")

print("Hello World", file=output_file)
output_file.close()
```

Files can also be written to by using `.write(contents)`. This method will write only the characters given to it, so a newline `\\n` must be included for a newline.

```
output_file = open("hello.txt", "w")

output_file.write("Hello World\n")
output_file.close()
```

The arguments to the `open()` function are, in order, the name of the file to open and the mode in which to open the file. "w" means that the file is to be opened in write mode. If the file does not exist, this will create the file. If it does exist, then the contents of the file will be cleared in preparation for the new ones.

Other options include "a", which is similar to "w" but will not clear the contents of an existing file and will instead append the new data to the end, and "r" which will read the file instead. If "r" is used and the file does not exist, then an error will occur. The following code takes a filename as user input, then prints out the entire contents of that file.

Mode	What it does
a	Create file if does not exist, open file, append contents to the end
w	Create file if does not exist, open file, write contents to the beginning of file
r	Open file, permit reading only

```
filename = input("What file should be read? ")

input_file = open(filename, "r")
for line in input_file:
    print(line, end="")

input_file.close()
```

The additional concepts introduced in these examples are:

- The `print()` function can have an additional "file" parameter passed to it to allow writing to a file. This causes it to send its output to the file rather than the screen, though otherwise it performs identically.

- The `print()` function has an additional optional "end" parameter. This allows you to specify what should be printed after the main string given to it. This is important because it defaults to `"\n"`, which causes a newline after every print statement. By changing "end" to `""` we prevent a newline from being added after every line of the file is printed. This is because each line in the file already has a newline at the end of it, so we don't need `print()` to add its own.
- `.close()` is used to properly tell Python that you are done with a file and close your connection to it. This isn't *strictly* required, but without it you risk the file being corrupted or other programs being unable to access that file.
- When reading from a file, Python can use a `for` loop to go through each line in sequence. This works identically to if you think of the file as a list with every line being a different element of the list. The entirety of the file can also be read into a single string using the `.read()` function.

```
>>> input_file = open("test.py", "r")
>>> contents = input_file.read()
>>> print(contents)
filename = input("What file should be read? ")

input_file = open(filename, "r")
for line in input_file:
    print(line, end="")

input_file.close()
```

- `.readlines()` can be used to read all of a file at once, though it splits the file into a list. Each element of the list will be one line of the file being read.

1.1 Files Using `with`

Since every file that you open should be closed after use, Python has an easy way to do this for you. Using the `with` command, your file will automatically be closed when the `with` block finished executing.

```
filename = input("Enter filename: ")

with open(filename, "r") as file:
    for line in file:
        print(line, end="")
# file.close() is not necessary, because "with" closed it for us
```

Using `with`

Always use the `with` statement to deal with file I/O in Python.

2 Exceptions

An *exception* is an error message in Python. When a certain operation encounters an error, it can *raise* an exception that is then passed on to the user:

```
>>> 43 / 0
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ZeroDivisionError: division by zero
```

However, in a lot of cases you may want to handle an exception as a developer and not have it displayed to the user. This is where the `try` and `except` statements come in.

When working with input and output it is important to check for exceptions.

- For example, when we try to open a file that does not exist we'd like to exit the program safely or recover rather than observing the unexpected results.
- Exception handling in python consists of "try-except" blocks. In a "try-except" block we execute instructions in the `try` block and catch errors in one or more following `except` blocks.
- The `except` block is only executed if an exception is caught in the `try` block.
- Additionally, when an error is caught in the `try` block we stop executing commands in the `try` block and jump to the first `except` or optional `finally` block.

The following example throws a division by zero error and prints "division by zero":

```
prime = 7

try:
    result = prime / 0
    result = 7*42
except ZeroDivisionError as err:
    print(err)
```

Looking at the code above, since the error is thrown on line 5, line 6 is never executed.

If you are experimenting with code and want to know the name of an exception that is thrown, take a look at the error message:

```
>>> float('obviously not convertable')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: could not convert string to float: 'obviously not convertable'
```

The part highlighted in red here is the name of the error message, `ValueErrror`.

Hence, if you are getting user input and want to check whether it is the correct type, use a try-except block around the conversion:

```
try:
    x = int(input('Enter an integer number: '))
except ValueError:
    print('You did not enter an integer!')
else:
    print('You entered {}'.format(x))
```

Notice that you can use an `else` block to be executed if no exception was thrown.

Any `except` block that does not list built-in exceptions will catch all exceptions not listed in previous `except` blocks. For example, the following code will throw an error if the user enters anything but an integer:

```
try:
    x = int(input("Enter a number: "))
except:
    print("Unknown error.")
    raise
else:
    print("You entered: " + str(x))
```

The `raise` keyword causes a stack trace and prints out additional information if an exception is encountered. The `else` block is always optional and will always be executed if no exceptions are thrown in the `try` block. There are many more built-in exceptions such as `IOError` that can be found here:

`https://docs.python.org/3.2/library/exceptions.html`

If you try to open a file that does not exist for reading, Python will display an error message:

```
>>> open("not_a_file.txt", "r")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'not_a_file.txt'
```

In this case, the error is `FileNotFoundError`. Normally having an error occur will end your program, but we can use `try-except` in order to perform a special action in case of an error.

- The `finally` block is where clean-up actions are performed and is always executed after leaving the `try` block.

The following is a good example of using `with` and exception handling together.

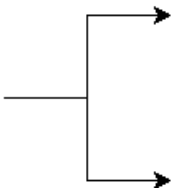
```
filename = input('Enter a filename >>> ')

try:
```

```
    with open(filename, 'r') as f:
        sum = 0
        for line in f:
            sum += float(line)
except FileNotFoundError:
    print('Filename you entered is wrong')
except ValueError:
    print('The file you entered had a line containing a non-float element.')
else:
    print('Sum of all numbers in file is {}'.format(sum))
finally:
    print('Goodbye.')
```

3 Instantiating Turtles

Similarly to being able to open multiple files, we can also create multiple turtles to draw more complex designs. This is done using the `turtle.Turtle()` function. This function returns a turtle object, which we can call every other normal turtle function on.

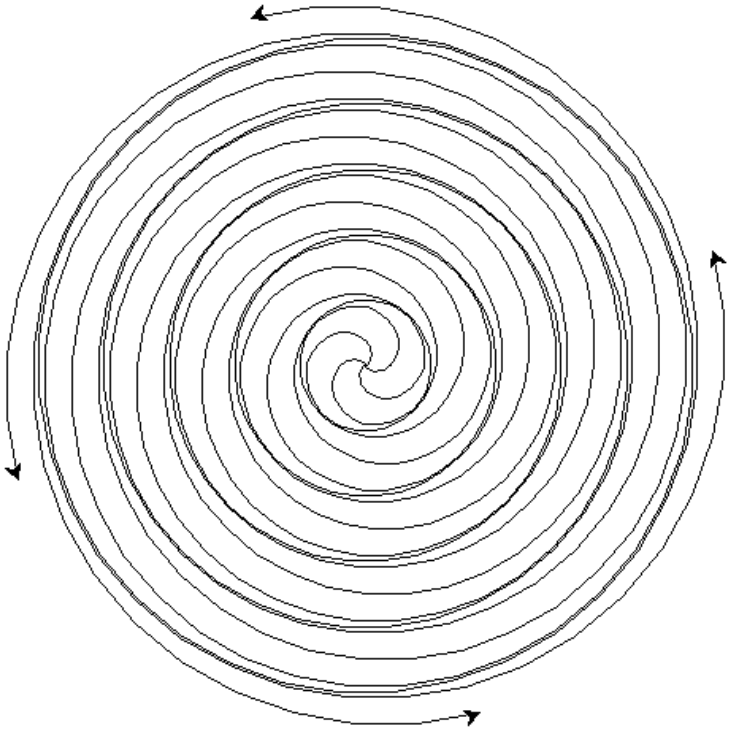
```
import turtle

first = turtle.Turtle()
second = turtle.Turtle()

first.forward(50)
second.forward(50)
first.left(90)
second.right(90)
first.forward(50)
second.forward(50)
first.right(90)
second.left(90)
first.forward(50)
second.forward(50)
```



If we add a group of turtles to a list, we can easily apply the same commands to all of them, as in this example:

```
import turtle

turtles = []
first = turtle.Turtle()
first.speed(0)
turtles.append(first)

second = turtle.Turtle()
second.speed(0)
second.right(90)
turtles.append(second)

third = turtle.Turtle()
third.speed(0)
third.right(180)
turtles.append(third)

fourth = turtle.Turtle()
```

```
fourth.speed(0)
fourth.right(270)
turtles.append(fourth)

for i in range(200):
    for turt in turtles:
        turt.forward(i/5)
        turt.left(10)
```



4 Matplotlib

Matplotlib is a Python library that provides MATLAB-like plotting functions. Simple plots such as bar graphs and line graphs are very easy to create using matplotlib.

Using the `.plot()` and `.bar()` methods of `matplotlib.pyplot`, we can quickly show line and bar graphs. `.plot()` and `.bar()` take a dataset for x and a dataset for y, with many more optional parameters such as `align`. These examples use data from

`http://nmt.edu/~olegm/382labs/2cities.csv`

- Data set is three columns: a data point number, the temperature in Tucson, and the temperature in Eugene in degrees Celsius.

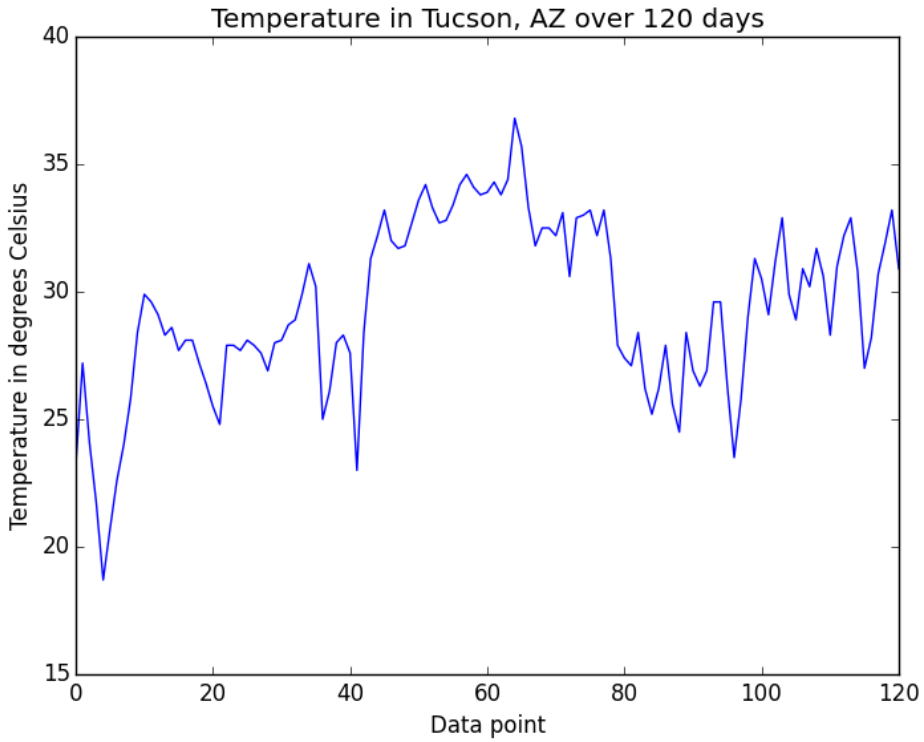


Figure 1: Line plot example: temperature data from Tucson, AZ.

The following code will show a line plot of the dataset. You can see the graph it produces in Figure 1.

```
1 import matplotlib.pyplot as plt
2
3 with open("2cities.csv", "r") as f:
4     lines = f.readlines()
5
6 temps = []
7 for line in lines:
```

```
    cols = line.split()
    try:
        if len(cols) > 1:
            temps.append(float(cols[1]))
    except ValueError:
        pass # could not convert to float, must be first row with headers

plt.plot(list(range(len(temps))), temps)
plt.xlabel("Data point")
plt.ylabel("Temperature in degrees Celsius")
plt.title("Temperature in Tucson, AZ over 120 days")
plt.show()
```

Next, we are trying to create a histogram of the data.

- A histogram is a bar graph where a bar represents a temperature *range*. We call one of these ranges a bin or a bucket.
- For example, the temperatures in Tucson in the data set range from 18° Celsius to 37° Celsius. We would create a histogram spanning temperatures from 15° to 40° Celsius with bins of size 5° Celsius. That means a bin for 15 to 20 degrees Celsius, a bin for 20 to 25 degrees Celsius, etc.

```
import matplotlib.pyplot as plt

with open("2cities.csv", "r") as f:
    lines = f.readlines()

temps = []
for line in lines:
    # columns in 2cities.csv are [data point number, tucson temp, eugene temp]
    cols = line.split()
    try:
        if len(cols) > 1:
            temps.append(float(cols[1]))
    except ValueError:
        pass # could not convert to float, must be first row with headers

numbuckets = 5
binnumbers = range(0, numbuckets)
counts = []
for bucketno in binnumbers:
    bucket = []
    for temp in temps:
        if 15 + bucketno * 5 < temp < 15 + (bucketno + 1) * 5:
            bucket.append(temp)
    # Count how many temperatures are in this bucket.
    # This count will be the height of the bar in the graph.
    counts.append(len(bucket))

xlabels = []
for bucket in binnumbers:
```

```
    xlabels.append("{} to {}C".format(15 + bucket * 5, 15 + (bucket + 1) * 5))

plt.bar(binnumbers, counts, align="center")
plt.xticks(binnumbers, xlabels)
plt.xlabel("Temperature in Celsius")
plt.ylabel("Number of times occured")
plt.title("Histogram of Temperatures in Tucson, AZ")
plt.show()
```

This code will produce this output of the input data sorted into 10 bins of size 5 (not showing empty bins) as seen Figure 2.

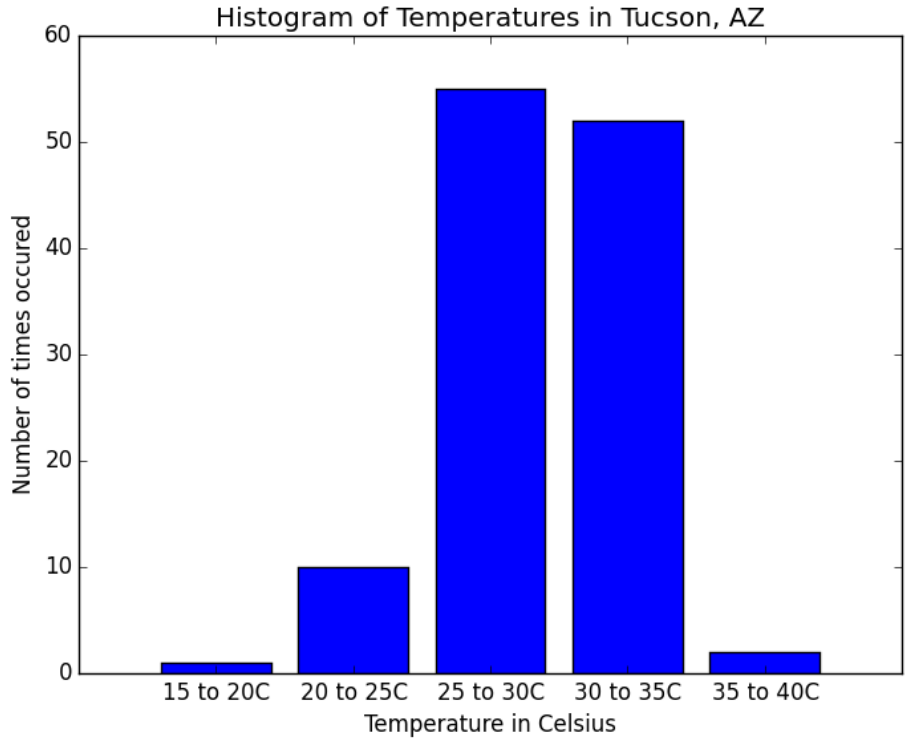


Figure 2: Input data sorted into 5 bins of size 5° Celsius.

4.1 More Documentation

- `.plot()` function documentation:

  `http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot`

- `.bar()` function documentation:

  `http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.bar`

- Documentation overview for `matplotlib.pyplot`:

  `http://matplotlib.org/api/pyplot_summary.html`

5 Exercises

Using `with`

Always use the `with` statement to deal with file I/O in Python. See Section 1.1.

Exercise 5.1 (save.py).
Write a program that takes in a filename, then takes in a series of lines of input until a blank line is entered, writing each line to the file with the given name. After the blank line is entered, properly close the file before ending the program.

Exercise 5.2 (word_count.py).
Write a program that takes in a filename and string as input. Then print how many times that string appears inside the chosen file. If the file does not exist, continue asking for a filename until one is given that exists. Use your source code file as test input.

Exercise 5.3 (diff.py).
Write a "diff" program that prints out the differences, line by line, of two files. Your program should ask the user for the names of two files, then print the differences between them. Follow the format output as shown below. Make sure to use proper error handling techniques for file I/O.

Example:

```
John goes to work.
Keith and Kyle went to the Ensiferum concert.
Alice ate an apple pie.
Joe cut down a tree.
The dog jumped over the wall.
```

Listing 1: file1.txt

```
John goes to work.
Coral went to a Kesha concert.
Alice ate an apple pie.
Joe planted a tree.
The dog jumped over the wall.
```

Listing 2: file2.txt

```
Enter file name 1 >>> file1.txt
Enter file name 2 >>> file2.txt

2c2
< Keith and Kyle went to the Ensiferum concert.
---
> Coral went to a Kesha concert.
4c4
< Joe cut down a tree.
---
> Joe planted a tree.
```

Exercise 5.4 (navigate3.py).
Modify navigate.py so that, rather than take instructions from the command line, it reads from a file (specified by user input) to determine what the turtle will do. Additionally, you will be adding the "split" command. This command will use instantiation of new turtles in order to draw multiple lines at once. Every new command will apply to every turtle that currently exists. The file will have on instruction per line. The possible instructions are:

forward X Move all turtles forward X.

left X Turn all turtles X degrees to the left.

right X Turn all turtles X degrees to the right.

split X Split all turtles into new turtles. Each new turtle will be turned X degrees to the right.

In order to properly implement split, you will probably need to look up the turtle functions `.position()`, `.setposition()`, `.setheading()`, `.heading()`, `.penup()`, and `.pendown()`. Turtle documentation is available at

`https://docs.python.org/3/library/turtle.html`

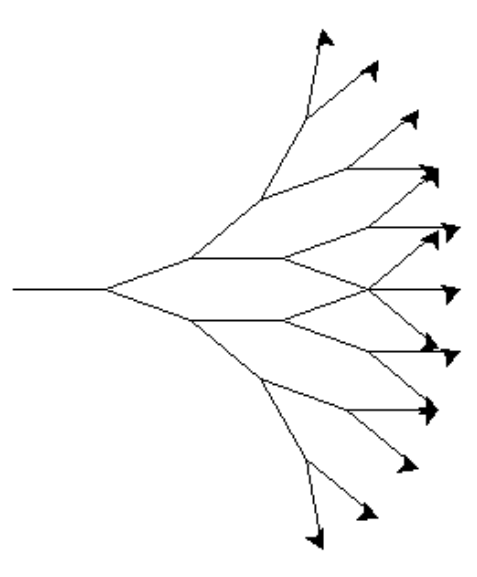
You will also likely use the `.split()` command when getting input, which splits a single string into an array around the string's spaces.

```
>>> print("this is a phrase".split())
['this', 'is', 'a', 'phrase']
>>> print("hi,hi,hi".split(",")
['hi', 'hi', 'hi']
```

Suggestion: Split each command into a different function to help keep their logic separate.
Sample Input File:

```
forward 50
left 20
split 40
forward 50
left 20
split 40
forward 50
left 20
split 40
forward 50
left 20
split 40
forward 50
left 20
```

Sample Output



Exercise 5.5 (testscores.py).
Download the file `actsat.txt` provided on Canvas. It contains the following data by column:

Column 1 2-letter state/territory code (includes DC)

Column 2 % of graduates in that state taking the ACT

Column 3 Average composite ACT score

Column 4 % of graduates in that state taking the SAT

Column 5 Average SAT Math score

Column 6 Average SAT Reading score

Column 7 Average SAT Writing score

Each SAT section is out of 800 points, while the ACT is out of 36 points.
Write a program that generates the following charts:

1. A histogram of average ACT scores with bins of size 1 between a score of 18 and 24.
2. A double bar chart that compares the composite ACT score with the total score of all 3 SAT tests for all 51 states/territories.
3. Produce the same chart as in part 2, but only for states in which less than 50% take the ACT and more than 50% take the SAT. (There should be 21 states/territories like this.)

You may have to do some research on the matplotlib website on how to create a double bar graph. Take a look at the example code on their website. Remember to label your axes and title your graph.

6 Submitting

You should submit your code as a tarball. It should contain all files used in the exercises for this lab. The submitted file should be named

`cse107_firstname_lastname_lab6.tar.gz`

Upload your tarball to Canvas.

List of Files to Submit