Lab 5: File I/O

CSE/IT 107

NMT Computer Science

> "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 don't need to waste my time with a computer just because I am a computer scientist."
>
> — Edsger W. Dijkstra

---

1 Introduction

2 File I/O

Knowing how to work with files is important, since it lets us store and retrieve data beyond the scope of a 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.

```
1  utput_file = open("hello.txt", "w")
2
3  print("Hello World", file=output_file)
4  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.

```
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.

2.1 Error Handling

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")9
Traceback (most recent call last):
File "<pyshell#0>", line 1, in <module>
```

```
open("not_a_file.txt", "r")
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 following program uses this to display an error message rather than crashing.

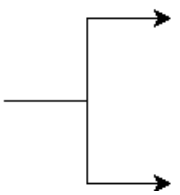
```
filename = input("What file should be read? ")

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

  input_file.close()
except FileNotFoundError:
  print("Could not find file {}.".format(filename))
```

If you wish to catch an error, check the error message for the name of the error that you need to catch with your `except` statement. If the specified error does not occur inside the `try` block, then the `except` block will be skipped. A single `try` block can have several `except` statements following it for catching multiple types of errors.

3 Instantiating Turtles

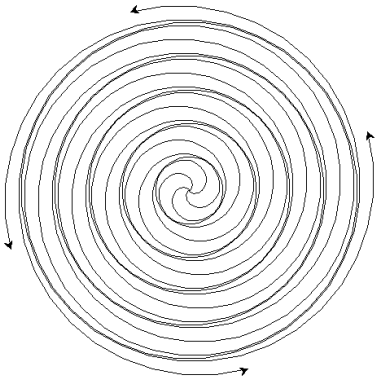
Similarly to being able to open multiple files, we can also create multiple turtles to draw more complex designs. This is done using the `turtle.Turtle()` function. This function returns a turtle object, which we can call every other normal turtle function on.

```
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 Program Boilerplate

Most Python programs will have a standard format to them that looks something like this:

```
#import statements

#other function declarations

#Program starting point - first function run
```

```
6  def main():
7      #All function calls and program logic should start here
8
9  #Boilerplate that must be at the very end of your program
10 if __name__ == '__main__':
11     main()
```

All program logic should exist inside functions. That is: the only code that exists at the lowest level of indentation should be imports, function declarations, and the `if __main__ == '__main__':` statement.

There are several reasons for this boilerplate code. Primarily, it provides a common structure to your programs – when trying to determine what a program does, you can know to look for the `main()` function first.

Additionally, this format will be useful once you start writing programs that consist of more than one file. Without this syntax, a file's code will execute when it is imported, while with this syntax importing the file will only allow its functions to be used.

You are required to use this format on all future labs in this class.

5 Exercises

Boilerplate

Remember that all future labs (including these) *must* use the boilerplate syntax introduced in Section 4.

save.py Write a program that takes in a filename as input, 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.

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.

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.

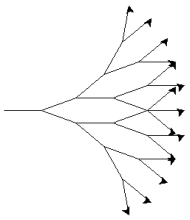
```
>>> print("this is a phrase".split())
['this', 'is', 'a', 'phrase']
```

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:



6 Submitting

Files to submit:

- save.py
- word_count.py
- navigate3.py

You may submit your code as either a tarball (instructions below) or as a .zip file. Either one should contain all files used in the exercises for this lab. The submitted file should be named either `cse107_firstname_lastname_lab5.zip` or `cse107_firstname_lastname_lab5.tar.gz` depending on which method you used.

For Windows, use a tool you like to create a `.zip` file. The TCC computers should have `7z` installed. For Linux, look at lab 1 for instructions on how to create a tarball or use the "Archive Manager" graphical tool.

Upload your tarball or .zip file to Canvas.