#3-3: Text-Based Interaction

CS SCHOLARS – PROGRAMMING

Submit Bonus Lecture Ideas!

Next Wednesday we have time to run a bonus lecture on a Computer Science topic that you all are interested in.

If there's a topic you'd specifically like to learn about, please **send Prof. Kelly a Slack or email message** by Thursday EOD with your topic.

We'll do a poll on Friday in class to choose the topic.

Learning Goals

Create interactive programs using **repeated input and output** via text

Read and write data from **files**

Use string operations and methods to extract data from **plaintext**

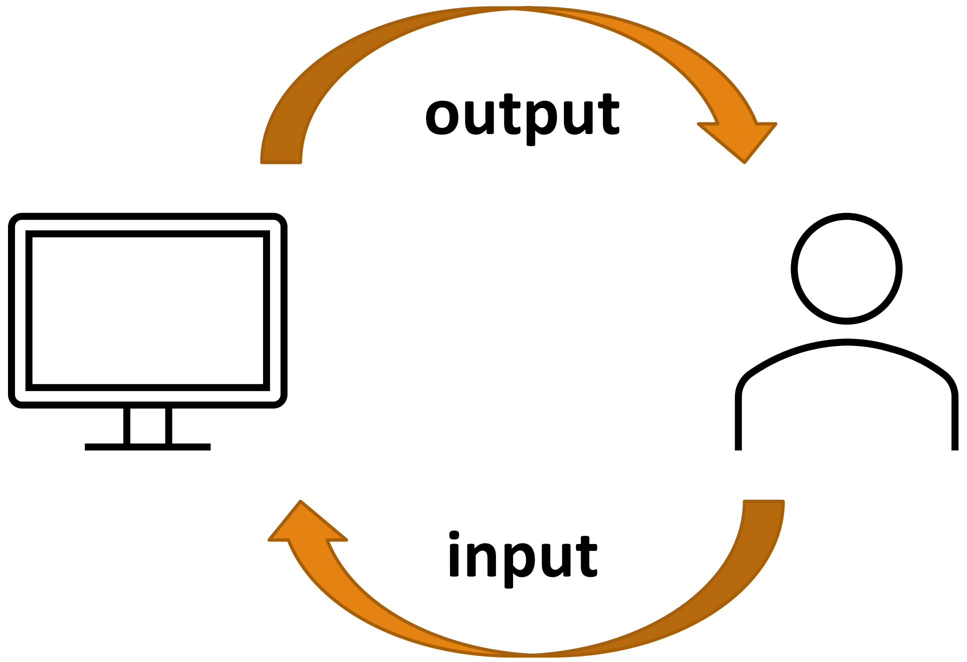
Input and Output

Interacting with the User

Now we have everything that we need to build programs that can authentically interact with the person using them (the 'user').

When we build an interactive program, we have to take **input** from the user, process it in the system, and use it to produce **output** that will be shown to the user in some way.

This process repeats indefinitely until some goal is met.



Text-based Interaction

One way to process input and create output is via **text-based interaction**.

Get all user input through the `input` function we learned previously. Produce all output as text displayed via the `print` function to the interpreter.

To repeat the procedure, use a loop. Since we usually do not know exactly how many repetitions will be needed, a `while` loop is often used (with the loop control variable based on the user's input).

You'll usually want to use a **variable** to collect input from the user. The type of that variable depends on your goal!

Example: Data Entry

Let's write a simple program that gets multiple inputs from the user and process them like a data stream.

We'll need to give the user a way to signal that they're done entering numbers. This can by done with a special input, like the string `'q'`.

Note that a new input is collected and a new output is shown in every iteration of the loop.

```
numbers = []
value = input("Enter a number, or q to quit:")
while value != "q":
    num = int(value)
    numbers.append(num)
    print("Current numbers:", numbers)
    value = input("Enter a number, or q to quit:")
print("Total sum:", sum(numbers))
```

Activity: Longest String

You do: Write a bit of code that repeatedly asks the user to enter strings. When the user enters an empty string (presses the enter key without typing anything), the function will output the longest string the user typed. You can break ties however you want.

Validating Input

We may sometimes need to **validate** user input, to make sure it matches the requirements.

For example, in the data entry program we might want to ensure that the input actually is a number, and a positive number, before accepting it. We can use methods and operators to ensure this.

It often helps to move processing user input into a **helper function**, to avoid overcomplicating the original function. We'll talk about these more next week!

```python
def getUserInput():
    while True: # returning will stop loop
        value = input("Enter a number, or q to quit:")
        if value == "q":
            return value
        elif value.isnumeric():
            value = int(value)
            if value > 0:
                return value

def processData():
    numbers = []
    value = getUserInput()
    while value != "q":
        numbers.append(value)
        print("Current numbers:", numbers)
        value = getUserInput()
    print("Total sum:", sum(numbers))
```

Activity: reorderList

You do: write a function that takes a list of strings and asks the user to enter a new order for the strings, one value at a time. For example, if we call `reorderList(["a", "b", "c", "d"])`, the user could enter `c`, then `a`, then `d`, then `b`, and the function would return the list `[ "c", "a", "d", "b" ]`.

Make sure your program has clear requests for input, understandable output, a well-managed interaction loop, and validates the user input!

Reading Data from Files

Reading Data From Files

As we interact with the user, we may also want to refer to data stored elsewhere on the computer. That means we need to **read data from a file**.

All the files on your computer are organized in **directories**, or **folders**. When you're working with files, always make sure you know which sequence of folders your file is located in. A sequence of folders from the top-level of the computer to a specific file is called a **filepath**.

For example, **Users > krivers > Documents > sample.txt** refers to the file **sample.txt** in the **Documents** folder, which is in the **krivers** folder, which is in the **Users** folder, which is at the top level of the computer.

Opening Files in Python

To interact with a file in Python we'll need to access its contents. We can do this by using the built-in function `open(filepath)`. This will create a **File object** which we can read from or write to.

```
f = open("/Users/krivers/Documents/sample.txt")
```

`open` can either take a full filepath or a **relative path** (relative from the location of the python file). It's usually easiest to put the file you want to read/write in the same directory as the python file so you can simply refer to the filename directly.

```
f = open("sample.txt")
# if .py file is in Documents, will search for this file there
```

Reading and Writing from Files

When we open a file we need to specify whether we plan to **read from** or **write to** the file. This will change the **mode** we use to open the file.

```
filename = "sample.txt"
f = open(filename, "r") # read mode
text = f.read() # reads the whole file as a single string
# or
lines = f.readlines() # reads the lines of a file as a list of strings

f = open("sample2.txt", "w") # write mode
f.write(text) # writes a string to the file
```

Only one instance of a file can be kept open at a time, so you should always **close** a file once you're done with it.

```
f.close()
```

Be Careful When Programming With Files!

WARNING: when you write to files in Python, backups are not preserved. If you overwrite a file, the previous contents are gone forever.

Be careful when writing to files! Make sure you're using the correct filename before you run the program. Avoid overwriting original data whenever possible; you can always wait and delete it after you're done.

Activity: Read a File

You do: Download the file `sample.txt` from the schedule page and move it to the same folder as a python script.

Try using `open` and `read` to open the file and read the contents, then `print` the contents.

Common file reading issues:

- make sure the file is actually in the same directory as your python script (check directory in the `%cd` line when you run Thonny)
- make sure the filename you've entered is actually the right filename (including the filetype at the end!)
- Using repl.it? Upload the file to the folder you're working in.

Extracting Data from Text

Parsing File Data

Once we've read text data from a file, what do we do with it?

You need to identify what kinds of **patterns** exist in the data and use that information to structure it. The patterns you identify may depend on which question you're trying to answer.

Questions to Ask

When parsing data from a file, start by identifying the pattern; then ask yourself a few questions about that pattern.

- Does the pattern occur across lines, or some other delimiter?
- Where is the information in a single line/section?
- What comes before or after the information you want?

Tools to Use

Once you've identified where the information is located, use **string slicing** and **string methods** to separate out the information you need.

Slicing (`s[start:end:step]`) can be used to remove parts of the data that are unnecessary.

The **split** method (`s.split(".")`) can be used to break up data that is separated by a known delimiter.

The **index** method (`s.index(":")`) can be used to find the location of the beginning or end of a section. That can be combined with slicing or splitting to isolate the needed data.

A new method called **strip** (`s.strip()`) can be used to remove whitespace (spaces, tabs, and newlines) from the front and back of a string. This is useful for isolating the core text of a string.

Example: Parsing a Chat Log

`chat.txt` is a dataset based on a chat log from a previous class. (All student names have been modified to preserve student privacy).

How could we get the names of everyone who participated in the chat? What's the **pattern**?

```
14:54:28      From  Malika : Could I use
recursion for AuthorMap?

14:56:03      From  Ed : yep

15:00:22      From  Arman : what is
str.digits?

15:01:21      From  Margaret Reid-Miller   to
Kelly Rivers(Privately) : We only hear the
music when you speak

15:08:31      From  Ed : how would you know if
it were O(n**.5)?
```

Example: Parsing a Chat Log

Each message occurs on an individual line; split the text based on newlines (`"\n"`).

`"From"` occurs before each name and `" : "` occurs afterwards. `index` to find those locations and slice based on them.

Use `strip` to clear extra whitespace.

```
f = open("chat.txt", "r")
text = f.read()
f.close()

people = [ ]
for line in text.split("\n"):
    start = line.index("From ") + \
            len("From ")
    line = line[start:]
    end = line.index(" : ")
    line = line[:end]
    line = line.strip()
    people.append(line)
print(people)
```

Example: Parsing a Chat Log

A few lines don't match the pattern; account for those too.

If statements are useful when something breaks a pattern.

```
...
    line = line[:end]
    if "(Privately)" in line:
        end = line.index("to")
        line = line[:end]
    line = line.strip()
...
```

Activity: Get Speaking Roles

You Do: you can download the full text of books in the public domain from Project Gutenberg. Go to the site, download the Plain Text version of Romeo and Juliet, then write a Python program to scrape the names of all the speaking roles in the play from the document.

Note: you can technically find a list of all the speaking roles at the top of the document. Don't rely on this! Challenge yourself to find a pattern within **the play itself** that will let you extract the names.

Learning Goals

Create interactive programs using **repeated input and output** via text

Read and write data from **files**

Use string operations and methods to extract data from **plaintext**