Lecture 4

- Libraries
- Random
- Statistics
- Command-Line Arguments
- `slice`
- Packages
- APIs
- Making Your Own Libraries
- Summing Up

Libraries

- Generally, libraries are bits of code written by you or others that you can use in your program.
- Python allows you to share functions or features with others as "modules".
- If you copy and paste code from an old project, chances are you can create such a module or library that you could bring into your new project.

Random

- `random` is a library that comes with Python that you could import into your own project.
- In your terminal window type `code generate.py`. In your text editor, code as follows:

```
import random

coin = random.choice(["heads", "tails"])
print(coin)
```

  Notice that the list within `choice` has square braces, quotes, and a comma. Since you have passed in two items, Python does the math and gives a 50% chance for heads and tails. Running your code, you will notice that this code, indeed, does function well!

- We can improve our code. `from` allows us to be very specific about what we'd like to import. Prior, our `import` line of code is bringing the entire contents of the functions of `random`. However, what if we want to only load a small part of a module? Modify your code as follows:

```
from random import choice

coin = choice(["heads", "tails"])
print(coin)
```

  Notice that we now can import just the `choice` function of `random`. From that point forward, we no longer need to code `random.choice`. We can now only code `choice` alone. `choice` is loaded explicitly into our program. This saves system resources and potentially can make our code run faster!

- Moving on, consider the function `random.randint(a, b)`. This function will generate a random number between `a` and `b`. Modify your code as follows:
  Notice that our code will randomly generate a number between `1` and `10`.
- We can introduce into our card `random.shuffle(x)` where it will shuffle a list into a random order.

```
import random

cards = ["jack", "queen", "king"]
random.shuffle(cards)
for card in cards:
    print(card)
```

  Notice that `random.shuffle` will shuffle the cards in place. Unlike other functions, it will not return a value. Instead, it will take the `cards` list and shuffle them inside that list. Run your code a few times to see the code functioning.

- We now have these three ways above to generate random information.
- You can learn more in Python's documentation of `random`.

Statistics

- Python comes with a built-in `statistics` library. How might we use this module?

  Notice that we imported a different library called `statistics`. The `mean` function takes a list of values. This will print the average of these values. In your terminal window, type `python average.py`.

- Consider the possibilities of using the `statistics` module in your own programs.
- You can learn more in Python's documentation of `statistics`.

Command-Line Arguments

- So far, we have been providing all values within the program that we have created. What if we wanted to be able to take input from the command-line? For example, rather than typing `python average.py` in the terminal, what if we wanted to be able to type `python average.py 100 90` and be able to get the average between `100` and `90`?
- `sys` is a module that allows us to take arguments at the command line.
- `argv` is a function within the `sys` module that allows us to learn about what the user typed in at the command line. Notice how you will see `sys.argv` utilized in the code below. In the terminal window, type `code name.py`. In the text editor, code as follows:

```
import sys
```

- There is a small problem with our program as it stands. What if the user does not type in the name at the command line? Try it yourself. Type `python name.py` into the terminal window. An error `list index out of range` will be presented by the compiler. The reason for this is that there is nothing at `sys.argv[1]` because nothing was typed! Here's how we can protect our program from this type of error:

```
import sys

try:
    print("hello, my name is", sys.argv[1])
except IndexError:
    print("Too few arguments")
```

Notice that the user will now be prompted with a useful hint about how to make the program work if they forget to type in a name. However, could we be more defensive to ensure the user inputs the right values?

- Our program can be improved as follows:

```
import sys

if len(sys.argv) < 2:
    print("Too few arguments")
elif len(sys.argv) > 2:
    print("Too many arguments")
```

- Right now, our code is logically correct. However, there is something very nice about keeping our error checking separate from the remainder of our code. How could we separate out our error handling? Modify your code as follows:

```
import sys

if len(sys.argv) < 2:
    sys.exit("Too few arguments")
elif len(sys.argv) > 2:
    sys.exit("Too many arguments")

print("hello, my name is", sys.argv[1])
```

Notice how we are using a built-in function of `sys` called `exit` that allows us to exit the program if an error was introduced by the user. We can rest assured now that the program will never execute the final line of code and trigger an error. Therefore, `sys.argv` provides a way by which users can introduce information from the command line. `sys.exit` provides a means by which the program can exit if an error arises.

- You can learn more in Python's documentation of `sys`.

`slice`

- `slice` is a built-in function that allows us to take a `list` and tell the compiler where we want the compiler to consider the start of the `list` and the end of the `list`.

```
    sys.exit("Too few arguments")

for arg in sys.argv:
    print("hello, my name is", arg)
```

Notice that if you type `python name.py David Carter Rongxin` into the terminal window, the compiler will output not just the intended output of the names, but also `hello, my name is name.py`. How then could we ensure that the compiler ignores the first element of the list where `name.py` is currently being stored?

- `slice` can be employed in our code to start the list somewhere different! Modify your code as follows:

```
import sys

if len(sys.argv) < 2:
    sys.exit("Too few arguments")

for arg in sys.argv[1:]:
    print("hello, my name is", arg)
```

Notice that rather than starting the list at `0`, we use square brackets to tell the compiler to start at `1` and go to the end using the `1:` argument. Running this code, you'll notice that we can improve our code using relatively simple syntax.

Packages

- `cowsay` is a well-known package that allows a cow to talk to the user.
- Python has a package manager called `pip` that allows you to install packages quickly onto your system.
- In the terminal window, you can install the `cowsay` package by typing `pip install cowsay`. After a bit of output, you can now go about using this package in your code.

- In your terminal window type `code say.py`. In the text editor, code as follows:

```
import cowsay
import sys

if len(sys.argv) == 2:
    cowsay.cow("hello, " + sys.argv[1])
```

  Notice that the program first checks that the user inputted at least two arguments at the command line. Then, the cow should speak to the user. Type `python say.py David` and you'll see a cow saying "hello" to David.

- Further modify your code:

```
import cowsay
import sys
```

- You can learn more on PyPI's entry for `cowsay`
- You can find other third-party packages at PyPI

APIs

- APIs or "application program interfaces" allow you to connect to the code of others.
- `requests` is a package that allows your program to behave as a web browser would.
- In your terminal, type `pip install requests`. Then, type `code itunes.py`.
- It turns out that Apple iTunes has its own API that you can access in your programs. In your internet browser, you can visit https://itunes.apple.com/search?entity=song&limit=1&term=weezer and a text file will be downloaded. David constructed this URL by reading Apple's API documentation. Notice how this query is looking for a `song`, with a `limit` of one result, that relates to the `term` called `weezer`. Looking at this text file that is downloaded, you might find the format to be similar to that we've programmed previously in Python.
- The format in the downloaded text file is called JSON, a text-based format that is used to exchange text-based data between applications. Literally, Apple is providing a JSON file that we could interpret in our own Python program.
- In the terminal window, type `code itunes.py`. Code as follows:

```
response = requests.get("https://itunes.apple.com/search?entity=song&limit=1&term=" + sys.argv[1])
print(response.json())
```

  Notice how the returned value of `requests.get` will be stored in `response`. David, having read the Apple documentation about this API, knows that what is returned is a JSON file. Running `python itunes.py weezer`, you will see the JSON file returned by Apple. However, the JSON response is converted by Python into a dictionary. Looking at the output, it can be quite dizzying!

- It turns out that Python has a built-in JSON library that can help us interpret the data received. Modify your code as follows:

```
import json
import requests
import sys

if len(sys.argv) != 2:
    sys.exit()

response = requests.get("https://itunes.apple.com/search?entity=song&limit=1&term=" + sys.argv[1])
print(json.dumps(response.json(), indent=2))
```

  Notice that `json.dumps` is implemented such that it utilizes `indent` to make the output more readable. Running `python itunes.py weezer`, you will see the same JSON file. However, this time, it is much more readable. Notice now that you will see a dictionary called `results` inside the output. Inside that dictionary called `results` there are numerous keys present. Look at the `trackName` value in the output. What track name do you see in your results?

```
if len(sys.argv) != 2:
    sys.exit()

response = requests.get("https://itunes.apple.com/search?entity=song&limit=50&term=" + sys.argv[1])

o = response.json()
for result in o["results"]:
    print(result["trackName"])
```

  Notice how we are taking the result of `response.json()` and storing it in `o` (as in the lowercase letter). Then, we are iterating through the `results` in `o` and printing each `trackName`. Also notice how we have increased the limit number of results to `50`. Run your program. See the results.

- You can learn more about `requests` through the library's documentation.
- You can learn more about JSON in Python's documentation of JSON.

Making Your Own Libraries

- You have the ability as a Python programmer to create your own library!
- Imagine situations where you may want to re-use bits of code time and time again or even share them with others!

```
def goodbye(name):
```

```
    print(f"goodbye, {name}")
```

Notice that this code in and of itself does not do anything for the user. However, if a programmer were to import this package into their own program, the abilities created by the functions above could be implemented in their code.

- Let's see how we could implement code utilizing this package that we created. In the terminal window, type `code say.py`. In this new file in your text editor, type the following:

```
import sys

from sayings import goodbye

if len(sys.argv) == 2:
    goodbye(sys.argv[1])
```

Notice that this code imports the abilities of `goodbye` from the `sayings` package. If the user inputed at least two arguments at the command line, it will say "goodbye" along with the string inputed at the command line.

Summing Up

- Libraries
- Random
- Statistics
- Command-Line Arguments
- Slice
- Packages
- Making Your Own Libraries