Iterators

- An object that has the capabilities to iterate over all its items
- Stores it's "state" by remembering the next item to give
- Only requested amount of data is given at a single instance
- Prerequisite to Generators (Very important for big data)
- `my_iter = iter([1, 4, 2])`
- `my_iter.next() # gives 1`
- `my_iter.next() # gives 4`
- `for x in my_iter:`

Generators

- Similar to iterator but can only iterate over object once
- Values are not all stored in memory but are "generated" when requested
- Do not need to wait for all values to be loaded
- Functions are made into generators when the keyword `yield` is used instead of `return`

```
# a generator that yields items instead of returning a list
def firstn(n):
    num = 0
    while num < n:
        yield num
        num += 1

sum_of_first_n = sum(firstn(1000000))
```

Scope

- Local – Within a function
  - Can be nested functions
  - Can only be accessed by other objects in the same local scope

    ```
    def myfunc():
        x = 300
        def myinnerfunc():
            print(x)
        myinnerfunc()
    ```

Iterators

- An object that has the capabilities to iterate over all its items
- Stores it's "state" by remembering the next item to give
- Only requested amount of data is given at a single instance
- Prerequisite to Generators (Very important for big data)
- `my_iter = iter([1, 4, 2])`
- `my_iter.next() # gives 1`
- `my_iter.next() # gives 4`
- `for x in my_iter:`

Generators

- Similar to iterator but can only iterate over object once
- Values are not all stored in memory but are "generated" when requested
- Do not need to wait for all values to be loaded
- Functions are made into generators when the keyword `yield` is used instead of `return`

```
# a generator that yields items instead of returning a list
def firstn(n):
    num = 0
    while num < n:
        yield num
        num += 1

sum_of_first_n = sum(firstn(1000000))
```

Scope

- Local – Within a function
  - Can be nested functions
  - Can only be accessed by other objects in the same local scope

    ```
    def myfunc():
        x = 300
        def myinnerfunc():
            print(x)
        myinnerfunc()
    ```

- Global – Within the main body of a python script
  - Available for everyone

    ```
    x = 300
    def myfunc():
        print(x)
    myfunc()
    ```

- Variables of the same name can exist in different scopes
  - Global `x` and local `x`

    ```
    x = 300
    def myfunc():
        global x
        x = 200
    myfunc()
    ```

Virtual Environments

- A standalone environment for installing and running python scripts / packages
- Stores everything in a single directory
- Packages are no longer installed globally
- Recommended when:
  - Working on machines that are not yours
  - Running off-the-shelf code (version control)
  - In doubt (Does not hurt your program and you can still access the same files on your machine)
- Command Line:
  - `python3 -m venv myVenv # creates the virtual environment "myVenv"`
  - `source myVenv/bin/activate # activates the virtual enviroment`
  - `deactivate # deactivates the virtual environment`

Modules

- A way to reuse code without copy / paste
- Save reusable code a `.py` file

  ```
  def greeting(name):
      print("Hello, " + name)
  # saved to myModule.py
  ```

- Import module using `import`
  - `import myModule`
- Use functions by treating the entire script as an object
  - `myModule.greeting("King")`
- Works with objects stored in `.py` files as well
- If module file is not in the same directory, then you need to set the path to the module file
  - `import sys`
  - `sys.path.append("location/of/module")`
  - `import myModule`

Passing Arguments in the Command Line

- We often don't want to hard code paths to files
  - We might want to work with different files
  - We might want someone else to run the code on their own machine
- We can pass arguments from command line using the `sys` library
  - `import sys`
  - `list_of_arguments = sys.argv`
- We give the arguments when we call the python script
  - `python my_script.py arg_1 arg_2 arg_3`
  - `sys.argv = ["my_script.py", "arg_1", "arg_2", "arg_3"]`

sys Library Activity

- Create a python script called `myScript.py` with the following code

```
import sys
print(sys.argv)
my_var = sys.argv[1]
print(my_var)
```

- Run the python script as:
  - `python3 myScript.py "hello" 5 10 "leaf"`

Reading a File

- Types of files
  - `.txt, .csv, .json`
  - Suffix gives some idea for how the data was stored
- Require a location
  - Relative path `".\my\path\myFile.txt"`
    * Relative to the current directory that you are in
  - Absolute path `"C:\Users\user\my\path\myFile.txt"`
    * The full path name

```
my_file_path = "path\to\file.txt
my_file␣=␣open(my_file_path,␣"r")
my_list␣=␣my_file.readlines()
␣␣␣␣
```

- The second argument of `open()` says what mode the file is to be open. `"r"` is for read only. `"w"` will create a new file to write to and delete an old file of the same name if it exists.
- `readlines() gives a list of lines based on new line characters`
- Sometimes there are invisible characters at the end of the line that can cause issues. Built-in functions `strip()` is useful to remove leading and trailing whitespace characters.
- Sometimes data is nicely stored in a structured or semi-structured format (`JSON`, `CSV`)
- We can use the structure to make our lives easier

JSON – Javascript Object Notation

- Common structured format used to store data
- Looks like a dictionary
- `import json`
- Looks identical to dictionaries
- `x = "{"name": "King, "profession": "professor"}" # written as a JSON string`
- `y = json.load(x) # parses string representation into structured JSON`
- `y` is a python dictionary
- `x = json.dump(y) # converts dictionary to a JSON string`
  - Use this to store `JSON` formatted data in files

CSV – Comma-Separated Values

- Does not need commas to separate value
- Can use any character to separate values
  - The character used is called the delimiter
- We can read in without additional libraries and split the text using the delimiter
- We can also use a library called `csv`

```
import csv
csvfile = open(data_file_path, newline = "\n")
my_reader = csv.reader(csvfile, delimiter = ";")
for curr_line_arr in my_reader:
    print(curr_line_arr)
```

Testing Functionality

- A great way to see how a function works or what you can do with it is to open a python interpreter by just typing `python` and run simple code.