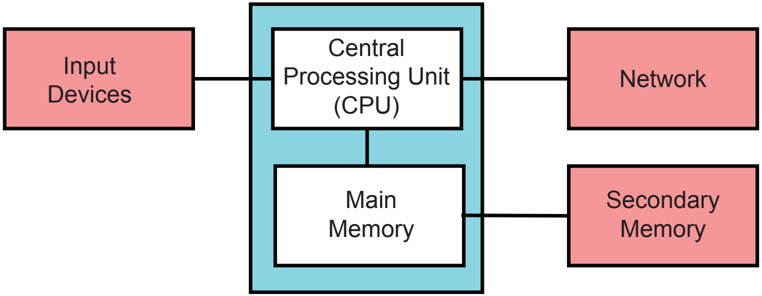
Data Foundations: FileIO, Functions, Asserts, and Sets

Instructor: Anthony Rios

Outline

File IO

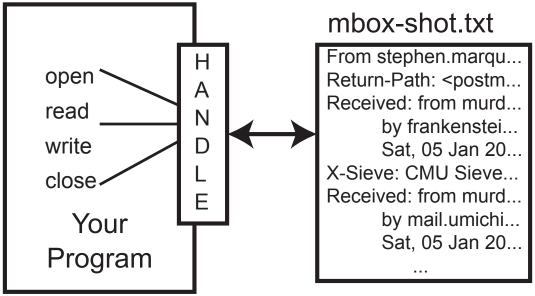


- Main memory stores **all variables**/loaded data (lists, ints, floats, strings, ...)
  - Erased when computer restarts. Applications lose access after they are closed.
- Secondary memory (hard drives, usb drives, ...)
  - Slower than main memory, but the information is not deleted when the CPU is powered off.

`https://www.youtube.com/watch?v=DKGZlaPlVLY`

Opening a File

```
>>> file_handle = open('mbox-short.txt')
>>> file_handle
<_io.TextIOWrapper name='mbox-short.txt' mode-'r' encoding='UTF-8'>
```



If successful, open returns a **file handle**. The file handle is **not** the actual data contained in the file.

File input

```
>>> file_handle = open('mbox-short.txt')

>>> file_data = file_handle.read() # Will read the entire file as a
string

>>> file_data[:31] # prints the first 31 characters of the string
'From stephen.marquard@uct.ac.za'

>>> file_handle.close() # Close the file

>>> c = file_handle.read()
File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Strings Revisited

```
>>> ‘\n’
‘\n’
```

```
>>> print(‘\n’) # 🟦 represents a blank line
🟦
🟦
```

```
>>> print(repr(‘\n’))
‘\n’
```

- “\n” is the new line symbol.
- print() always adds a new line

```
>>> ‘Ant’ in ‘Anthony’ # “in” can be used to check for sub-strings
True
```

```
>>> ‘Test’ in ‘Anthony’
False
```

Looping Over a File Line-by-Line

```
anthony@MacBook:~$ cat myfile.txt
line 1.
line 2.
line 3.
```

```
example.py
to_open = open('myfile.txt')
for line in to_open:
        print(line)
```

```
anthony@MacBook:~$ python example.py
line 1.
🟦
line 2.
🟦
line 3.
🟦
```

Looping Over a File Line-by-Line

```
anthony@MacBook:~$ cat myfile.txt
line 1.
line 2.
line 3.
```

```
example.py
to_open = open('myfile.txt')
for line in to_open:
        print(repr(line))
```

```
anthony@MacBook:~$ python example.py
line 1.\n
line 2.\n
line 3.\n
```

Looping Over a File Line-by-Line

```
anthony@MacBook:~$ cat myfile.txt
line 1.
line 2.
line 3.
```

example.py

```
to_open = open('myfile.txt')
for line in to_open:
        # .strip() removes white
        # space at the end and
        # the start of a string
        print(line.strip())
```

```
anthony@MacBook:~$ python example.py
line 1.
line 2.
line 3.
```

Exercise 1

Read the file “mbox.txt” line-by-line and calculate and print the following:

- The total number of lines in the file.
- The number of lines that contain the substring “From:”.

10 minutes

Writing to a File

myFile = open("**filename**"[, "**mode**"])

Important mode types:

- 'r' (**default**) – Read mode which is used when the file is only being read .
- 'w' – Write mode which is used to edit and write new information to the file (any existing files with the same name will be **erased** when this mode is activated) .
- 'a' – Appending mode, which is used to add new data to the **end of the file**.

Writing to a File

```
anthony@MacBook:~$ cat myfile.txt
line 1.
line 2.
line 3.
```

example.py

```
to_write = open('myfile.txt', 'w')
line = "This is a new line"
to_write.write(line)
```

```
anthony@MacBook:~$ cat myfile.txt
This is new line.
```

If "myfile.txt" does **not exist**, a new file will be **created**.

If "myfile.txt" **exists**, then the file is **overwritten**.

Appending to a File

```
anthony@MacBook:~$ cat myfile.txt
line 1.
line 2.
line 3.
```

```
example.py
to_write = open('myfile.txt', 'a')
line = "This is a new line"
to_write.write(line)
```

```
anthony@MacBook:~$ cat myfile.txt
line 1.
line 2.
line 3.This is a new line
```

Appending to a File

```
anthony@MacBook:~$ cat myfile.txt
line 1.
line 2.
line 3.
```

example.py

```
to_write = open('myfile.txt', 'a')
line = "\nThis is a new line"
to_write.write(line)
```

```
anthony@MacBook:~$ cat myfile.txt
line 1.
line 2.
line 3.
This is a new line
```

Exercise 2

Write code that reads the file “numbers.txt” line-by-line, then does the following:

- Sum all the numbers in numbers.txt, then prints the numbers to the screen.

Next, append the string "SUM: **k**" – where **k** is the calculated sum – to the end of numbers.txt as a new line.

⏱10 minutes

What is a function?

A function is a **named sequence of statements** that performs a computation

You can **call** a function by invoking it:

```
>>> type(32)
<class 'int'>
```

What is a function?

A function **takes** and **argument** and **returns** a result.

The **result** is call the **return value**.

Functions we have seen before

```
>>> a = [0, 1, 2, 3]

>>> len(a) # Function call
4 # Return value

>>> newVar = len(a) # Return value stored in newVar

>>> myString = “This is a test”

>>> myString.split() # A object-specific function
[‘This’, ‘is’, ‘a’, ‘test’]
```

Random Functions

```
>>> import random

# Returns a random number between 0.0 and 1.0 (including 0, but
not 1)
>>> random.random()
0.1113286792115236

# Returns a random int between 5 and 10 (including both)
>>> random.randint(5, 10)
3
```

Adding New Functions

```
def <funcName>(par1, par2):

        function Body
        return varName (optional)
```

Adding New Functions

example.py

```
def print_lyrics():
        print("I'm a lumberjack, and I'm okay.")
        print("I sleep all night and I work all day.")
print_lyrics()
```

```
anthony@MacBook:~$ python example.py
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Void Functions

Functions that do **not return** a value are called **void functions**

example.py

```
def print_lyrics():
        print("I'm a lumberjack, and I'm okay.")
        print("I sleep all night and I work all day.")
revVal = print_lyrics()
print(retVal)
```

anthony@MacBook:∼$ python example.py

```
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
None
```

Fruitful Functions

Functions that **return** a value are called **fruitful functions**

example.py

```
def addTwoNumbers(a, b):
        return a + b
revVal = addTwoNumbers(2,3)
print(retVal)
```

```
anthony@MacBook:~$ python example.py
5
```

Why functions?

- Creating a new function gives you an opportunity to **name a group of statements**, which makes your program easier to read, understand, and debug.
- Functions can make a program smaller by **eliminating repetitive code**. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to **debug the parts one at a time and then assemble them** into a working whole.
- Well-designed functions are often useful for many programs. Once you **write and debug one, you can reuse it**.

Exercise 3

Write a function to calculate your pay given two arguments: hoursWorks and dollarsPerHour. The function should return how much you should be paid. When calculating the final amount, give the employee 1.5 times the hourly rate for hours worked above 40 hours.

10 minutes

Asserts

Python's **asserts** are a debugging aid that tests a **condition**

Asserts

example.py

```
def addTwoNumbers(a, b):
        return a + b
revVal = addTwoNumbers(2,3)
assert(retVal == 5)
assert(retVal != 3)
print("Asserts Completed Successfully!")
```

```
anthony@MacBook:~$ python example.py
Asserts Completed Successfully!
```

Exercise 4

Create Asserts to test your code from Exercise 3. Specifically, test the following test cases:

- Check whether the function outputs 20 with inputs hoursWorks = 2 and dollarsPerHour = 10.
- Check whether the function does **not** output 410 if hoursWorks = 41 and dollarsPerHour = 10.

After all cases are passed successfully, print "Asserts Completed Successfully".

⏱ 5 minutes

What is a Set?

A set is a **collection** which is **unordered** and **unindexed unique** objects.

Python Sets

```
>>> mySet = {‘This’, ‘is’, ‘a’, ‘set’}
```

```
>>> mySet[0]
Traceback (most recent call last):
File ”<stdin>”, line 1, in <module>
TypeError: ’set’ object does not support indexing
```

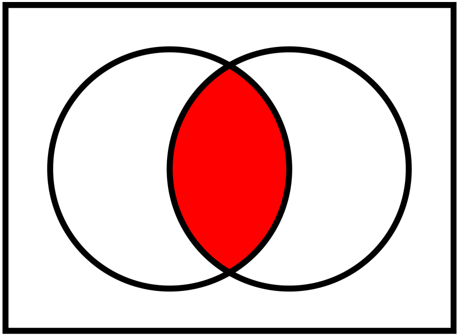
Python Sets: Unordered

example.py

```
mySet = {'This', 'is', 'a', 'set'}
for obj in mySet:
        print(obj)
```

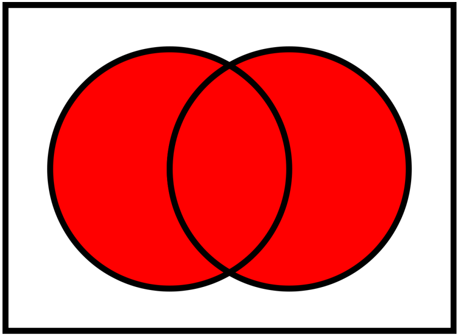
```
anthony@MacBook:~$ python example.py
set
a
This
is
```

Python Sets: Intersection



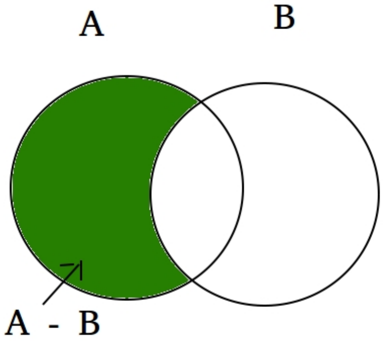
```
>>> set1 = {‘This’, ‘is’, ‘a’, ‘set’}
>>> set2 = {‘That’, ‘is’, ‘a’, ‘football’}
>>> set1.intersection(set2)
{‘is’, ‘a’}
```

Python Sets: Union



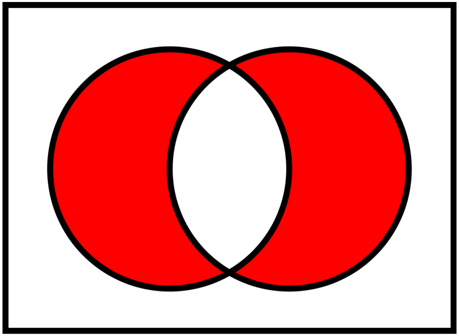
```
>>> set1 = {‘This’, ‘is’, ‘a’, ‘set’}
>>> set2 = {‘That’, ‘is’, ‘a’, ‘football’}
>>> set1.union(set2)
{‘That’, ‘This’, ‘set’, ‘is’, ‘a’, ‘football’}
```

Python Sets: Difference



```
>>> set1 = {'This', 'is', 'a', 'set'}
>>> set2 = {'That', 'is', 'a', 'football'}
>>> set1.difference(set2)
{'This', 'set'}
```

Python Sets: Symmetric Difference



```
>>> set1 = {'This', 'is', 'a', 'set'}
>>> set2 = {'That', 'is', 'a', 'football'}
>>> set1.symmetric_difference(set2)
{'That', 'This', 'set', 'football'}
```

Sets vs Lists

Why should we use a set over a list?

1. Use a set if you need to hold **unique objects**
2. Searching a set for an object using a set is **faster that lists**

Exercise 5

Write code to count the number of times a “risk” word appears from the “risk_lexicon” variable in the string variable named ”text.” Please ignore case (i.e., you should lowercase everything).

The output of your code should look like the following:

```
Risk Count: 2
```

10 minutes