Python: Beyond the Basics

Eisha Ahmed

February 28 & 29, 2020

MiCM Workshops

Mission : aims to deliver inter-disciplinary research programs and empower the use of data in health research and health care delivery

Let’s get setup!

- You should already have Python 3 running on your computer, using the IDE of your choice
- Write and run the short script below to check your Python version number
- If you have problems running Python on your computer, you can write and run most of your Python in your web browser using Google Collab: https://colab.research.google.com/ (requires google account)

```
import sys

v = sys.version.split(“ “)[0]

if sys.version_info[0] >= 3:
    print(“Running Python {}, good to go!”.format(v))
else:
    raise Exception(“Running version {}, which is too old… ”.format(v))
```

Workshop Outline

1. Object-oriented programming
   - **Demo Link:** https://colab.research.google.com/drive/176EGrPecz7nSCKph97qYX79FQMkdizeB
2. Exception handling
   - **Demo Link:** https://colab.research.google.com/drive/1M6CjWzSgIHmW14NbGLXFyxZ4jkcwrvhO
3. Recursion and file IO
   - **Demo Link:** https://colab.research.google.com/drive/14kMjIz5va-bWEz1f_-1-8MzAoiTlSBqU
4. Final challenge
   - **Demo Link:** https://colab.research.google.com/drive/1oXngcR90-3iOzuZM5c75LVHW2q2lpCJ9

Note that sample solutions to each activity, as well as the final challenge, will be posted after the workshop!

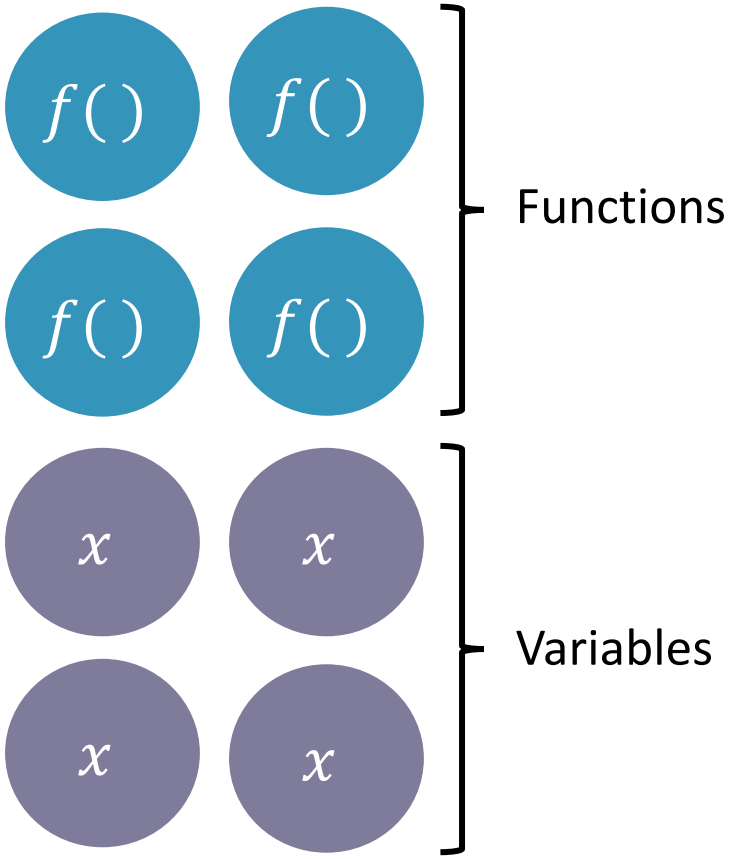
Object oriented programming (OOP)

Writing and structuring your programs

Before OOP:

Procedural Programming

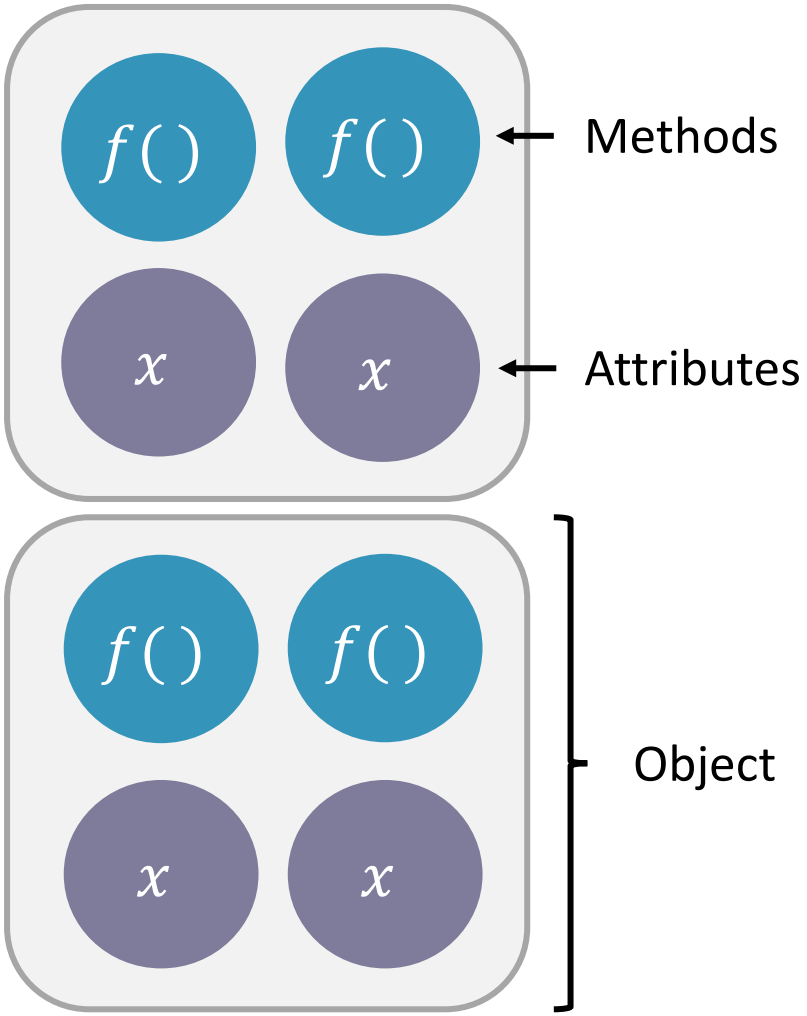
- Data stored in a number of variables
- Many functions that can act on the stored data



Straight-forward approach that can work well for small programs… larger projects can quickly become a nightmare!

Object-oriented programming

- OOP is a way to write and structure programs
- Allow us to logically group our data and functions for better code organization and re-use
- Based on the concepts of “objects” (containing data), and associated “methods”
- Applicable to many modern programming languages



OOP in Python

- Every value is an object in Python!
  - `myStr = 'hello there'` instantiates a new object of type `string`
  - Built-in string methods are exposed for that object (e.g. `lower()`, `join()`)
- Classes are templates (blueprints) for creating objects, and define a 'type'
  - You've worked with built-in types like `int` and `list`, but what if you wanted something of type `DNA`?
- Once we have a class, we can *instantiate* it to create a new object from that class

Objects

- An entity, or “thing” in your program
- E.g. an object could be a single cell

Properties (attributes)

- Position
- Size
- Age

Behaviours (methods)

- Move
- Grow
- Differentiate
- Die

OOP in Python

```
class Patient:

    def __init__(self, first, last):
        self.first = first
        self.last = last
        return

    def fullname(self):
        return '{} {}'.format(self.first, self.last)

patient1 = Patient("Eisha", "Ahmed")
print(patient1.fullname())
```

```
Eisha Ahmed
```

Member variables

Instance vs Class Variables

- Instance variables: Belong to an instance of a class, i.e. an object. Every instance has its own copy of that variable.
- Class variables: Variables that are shared across all instances of the class.
  - Can be accessed through both the class and the instance!

Member variables

Instance vs Class Variables

```
class MyClass:
    classVariable = 0

    def __init__(self, someInput):
        self.instanceVar = someInput
        return

Obj1 = MyClass("input1")
Obj2 = MyClass("input2")

print(MyClass.classVariable)  # returns 0
print(obj1.classVariable)     # returns 0
print(obj2.instanceVar)       # returns 'input2'
print(MyClass.instanceVar)    # returns an ERROR!
```

Aside: The @property decorator

Suppose we start our class as follows, and want to also have an email attribute made from the name.

```
1  class Person:
2      emailSuffix = “@mail.mcgill.ca”
3
4      def __init__(self, first, last):
5          self.first = first
6          self.last = last
7          self.email = “{}.{}{}”.format(self.first, self.last, Person.emailSuffix)
8          return
9
10 person1 = Person(“Eisha”, “Ahmed)
11 print(person1.email())
```

Q: What is the main weakness of directly assigning self.email in the **`__init__`** constructor? (line 7)

Aside: The @property decorator

One way to do this is to create a method that generates the email (lines 9-10). Note that email() is a method, and requires '()' at the end.

```
1  class Person:
2      emailSuffix = "@mail.mcgill.ca"
3
4      def __init__(self, first, last):
5          self.first = first
6          self.last = last
7          return
8
9      def email(self):
10          return "{}.{}{}".format(self.first, self.last, Person.emailSuffix)
11
12 person1 = Person("Eisha", "Ahmed)
13 print(person1.email())
```

Aside: The @property decorator

Using the @property decorator, we can treat this generated email just like an attribute! (no parenthesis required)

```
1  class Person:
2      emailSuffix = “@mail.mcgill.ca”
3
4      def __init__(self, first, last):
5          self.first = first
6          self.last = last
7          return
8
9      @property
10     def email(self):
11          return “{}.{}{}”.format(self.first, self.last, Person.emailSuffix)
12
13 person1 = Person(“Eisha”, “Ahmed)
14 print(person1.email)
```

Member methods

Instance vs Class vs Static Methods

- Instance methods: Automatically take the instance as the first argument ('self' by convention)
- Class methods: Automatically take the class as the first argument ('cls' by convention)
  - Particularly useful to act on class variables, or for making alternative constructors
- Static methods: Don't pass any arguments automatically
  - *Behave like regular functions, but we include them because of a logical connection to the class*

Member methods

Instance vs Class vs Static Methods

```
class MyClass:

    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'

obj = MyClass()
print(obj.method())
print(MyClass.classmethod())
print(MyClass.staticmethod())
```

Special (Magic) Methods

- Magic methods allow us to emulate built-in behaviour in Python
  - We can override default operators by defining our own special methods

DEFINITION

Operator Overloading

Where different operators have different implementations depending on their arguments.

Example Magic Methods

Method	Description
__init__	Implicitly called whenever a new object is created.
__repr__	Gets a string representation of object - implicitly called when we call repr() on our object. Intended to be unambiguous and used for debugging and development. *TIP: Consider implementing this method the majority of the time!*
__str__	Gets a string representation of object - implicitly called when we call str() or print() on our object. Intended for readability and to be displayed to the end user. Note that if __str__ is not defined but __repr__ is, then __repr__ will be called in its place (as a fallback).
__add__	Defines behaviour for '+' operator
__sub__	Defines behaviour for '-' operator
__eq__	Defines behaviour for '==' operator
__lt__	Defines behaviour for '<' operator
__gt__	Defines behaviour for '>' operator

Magic Methods in Python

```
class Person:

    def __init__(self, first, last):
        self.first = first
        self.last = last
        return

    def __repr__(self):
        return '{} {}'.format(self.first, self.last)

person1 = Person("James", "McGill")
print(person1)
```

OOP in Python

Activity #1

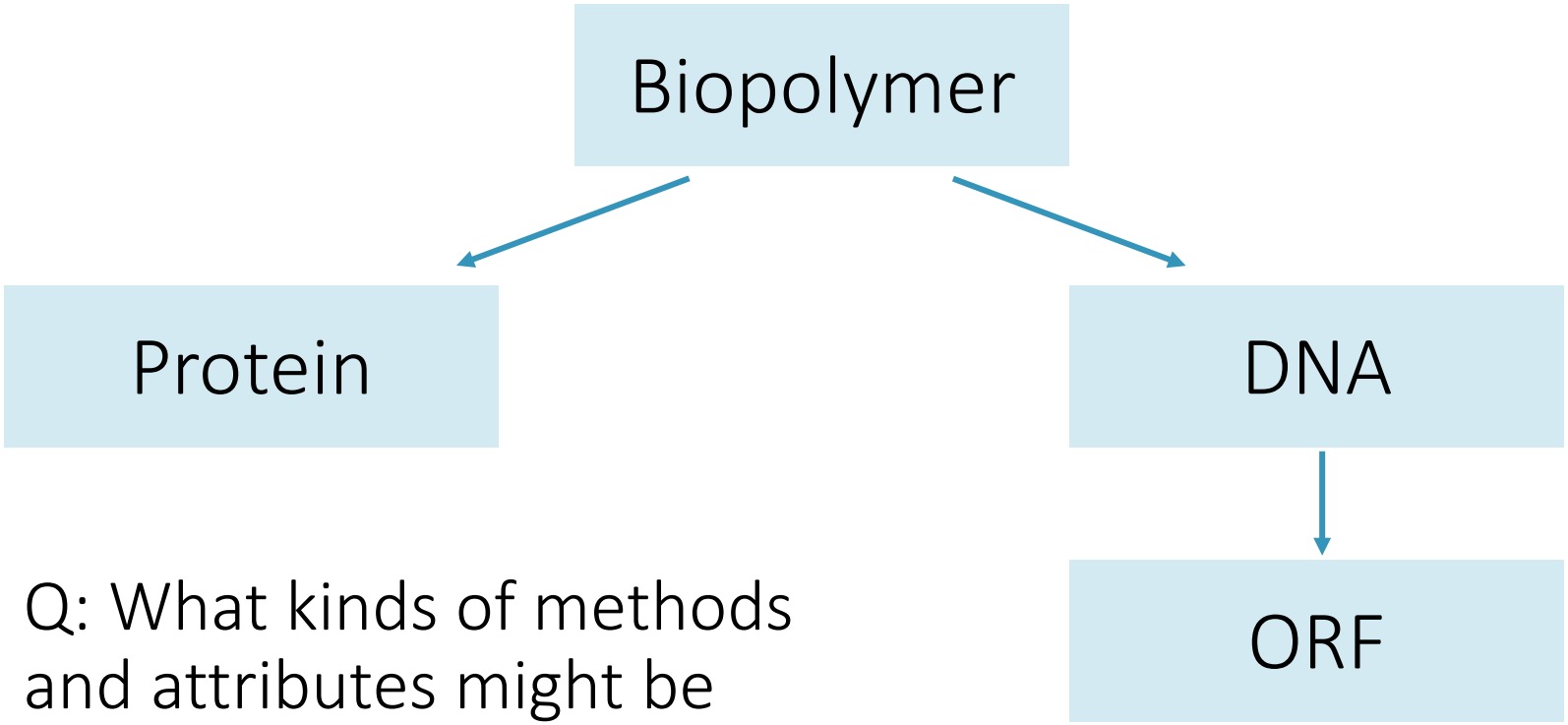
Write a class to represent DNA with the following:

- Attributes: `seq`, `name`, which should be initialized at instantiation.
- Class Variables: `validChars`, a set of the characters (nucleotides) that can be part of a valid DNA sequence.
- Instance Methods: `gcContent()`, which returns the % GC content of the sequence.
- Class Methods: `isValidDNA(myStr)`, which checks if a string is a valid DNA sequence by determining if all the characters belong to `validChars`
- Magic Methods:
  - Overload the '+' operator such that the new `Dna` object created is a concatenation of the original DNA sequence. It should also generate a concatenation of the names.
  - Overload the `__repr__` method, such that it returns a string representation of the `Dna` object in FASTA format when you call the `print()` function.

Inheritance

- We've learned how to create and instantiate classes in Python - *what if we wanted to make a new class, that extends a class we've already written?*
- **Inheritance** allows us to define a class that inherits all the methods and attributes from another class
  - **Child class (or derived class):** Class that inherits from another class
  - **Parent class (or base class):** Class being inherited from
- This captures the relationship between classes, and reduces the about of code you have to write
  - **DRY:** Don't repeat yourself

Inheritance



Q: What kinds of methods and attributes might be defined by each class and child class?

Inheritance in Python

Python Syntax

```
class ChildClass(ParentClass):
        pass
```

- Can overwrite methods in the child class by simply re-defining them
- `super()` gives you access to methods in the parent class from the child class, even if you overwrite them in the child class!
  - Great reference on Python `super()`: https://realpython.com/python-super/

OOP in Python

Activity #2

Let's build off of the DNA class you wrote earlier. Create an `Orf` class (open reading frame) that inherits from your `Dna` class.

Add the additional class variable `codonMap` (a dictionary mapping codons to amino acids), then add the following methods:

- `isValidORF()` that checks if a string is a valid open reading frame (class method).
  - HINT: To be a valid ORF, the string must (1) be a valid DNA sequence, (2) begin with a start codon, (3) end with a stop codon and have no other internal stop codons, and (4) have a length that is a multiple of 3.
- `translate()` that returns the corresponding protein sequence (instance method). Use the class variable `codonMap`.

Review: Pillars of OOP

Encapsulation

Grouping variables and related functions into objects

Inheritance

Eliminate redundant code by building a class from another class

Abstraction

Hiding complexity and simplifying interface

Polymorphism

Same function names, with different implementations

Exception Handling

‘Catching’ any errors thrown your way

Exception Handling

- Exception handling is the process of dealing with an unexpected event (error) when your program is running
  - Programs *throw* exceptions, which are *caught* by the handler
- Objective is to gracefully handle errors to prevent unexpected behaviour by telling your program what to do if an error is encountered during execution

What kind of errors would you expect from the following examples?

```
import math
x = -1
Math.log10(x)
```

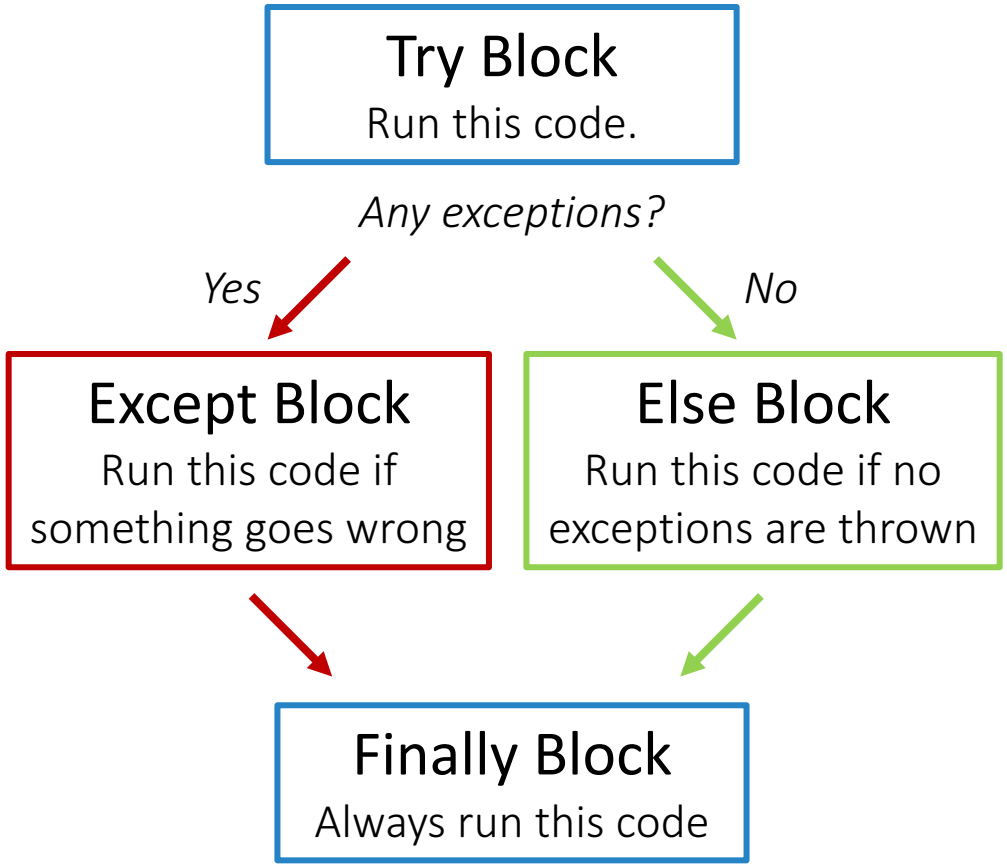
```
myVar = “hello”
myVar ** 2
```

Exception Handling

Try-except blocks

Python syntax

```
try:
    # run me
    pass
except:
    # if exceptions
    pass
else:
    # all ran well
    pass
finally:
    # run me always
    pass
```



Exception Handling

Example: Try-except blocks

```
try:
    # do something risky
    pass
except TypeError as e:
    # special handling for TypeErrors
    print("Some specific value error message")
    print(e)
except (ValueError, ZeroDivisionError) as e:
    # We can catch multiple error types at once
    print("Value and Zero division error message.")
except Exception as e:
    # catch all other exceptions not caught above
    print(e)
else:
    print("No errors.")
    pass
finally:
    # run me always
    pass
```

Common Built-In Exceptions

Example input and value exceptions

Exception Class	Description
`ZeroDivisionError`	Raised when denominator of division is zero.
`TypeError`	Raised when an operation or function is applied to an object of inappropriate type (e.g. trying to use a string as a list index).
`IndexError`	Raised when a sequence subscript (e.g. list index) is out of range.
`ValueError`	Raised when an operation or function receives an argument that has the right type, but inappropriate value.
`KeyError`	Raised when a mapping (dictionary) key is not found in the set of existing keys.

Common Built-In Exceptions

Example file handling exceptions

Exception Class	Description
`FileNotFoundError`	Raised when a file or directory is requested, but does not exist
`FileExistsError`	Raised when trying to create a file or directory that already exists
`IsADirectoryError`	Raised when a file operation is requested on a directory
`NotADirectoryError`	Raised when a directory operation is requested on a file

Full list of built-in Python exceptions:
https://docs.python.org/3/library/exceptions.html

Example: Raising exceptions

We can also raise exceptions, outside of try-except blocks:

```
raise ValueError
raise ValueError(“Descriptive message here.”)
```

Raising an exception terminates the program… but we can raise the exception later after it has been instantiated if we still have work to do:

```
error =  ValueError(“Descriptive message here.”)
print(“Do some work…”)
raise error
```

Example: Raising exceptions

We can also raise exceptions, outside of try-except blocks:

```
raise ValueError
raise ValueError(“Descriptive message here.”)
```

Raising an exception terminates the program… but we can raise the exception later after it has been instantiated if we still have work to do:

```
error =  ValueError(“Descriptive message here.”)
print(“Do some work…”)
raise error
```

Example: Custom exceptions

What if we wanted to raise custom exceptions? Exceptions in Python are objects of class `Exception`, so we can make a child class for our custom constructor!

```
class CustomException(Exception):
    pass

raise CustomException("Descriptive message here.")

# raising custom exception inside try-except block
try:
    raise CustomException
except CustomException as e:
   raise e
```

Exception Handling in Python

Activity #3

Let's revisit the DNA and ORF classes you wrote earlier, and validate the inputs when we try to set or change the sequence attribute.

You previously created `isValidDNA()` and `isValidORF()` methods – use these to check if the input sequence is valid. If not, throw a custom exception `InvalidSequence`.

Bonus:

You can simply add this validation to the `__init__` constructors for now, however it is *highly encouraged* to try re-organizing your code so such that a custom setter function is called whenever the sequence attribute is set or reset.

This way, validation will be also be performed if try to reassign the sequence value!

Recursion

Déjà vu – Writing self-referential functions

What is recursion?

- Recursion is a problem solving technique where a problem is broken down into smaller instances of the same problem
- In programming: Calling a function from within itself
- Defining a base case is crucial, otherwise your program will continue to run until you run out of memory!

Using a basic for-loop

```
def factorial(n):
    result = 1
    if n > 0:
        for i in range(n):
            result *= i
    return result
```

Using recursion:

```
def factorial(n):
    if n == 0:
        return 0
    else:
        return n*factorial(n-1)
```

Activity #4

Write a function that takes a string as an input, and returns the reverse of the string using recursion.

Hints:

- How could you break the problem of reversing a string into smaller instances of the same problem? What is the base case?
- Recall:
  - Accessing the i-th character of string: `myStr[i]`
  - Accessing a range of characters: `myStr[i:j]`
  - Accessing the i-th character to the end of the string: `myStr[i:]`

Reading and writing files

Getting data in, and information out

Reading and writing files

open() and close() functions

- `open()`: Takes multiple arguments (strings)
  - **Argument 1:** File path
  - **Argument 2:** Mode
  - Returns a file object
- `close()`: Closes file
  - Important to ensure computer resources are reallocated after you're finished! (member method)

Mode	Description
`r`	Open for reading plain text
`w`	Open for writing plain text
`a`	Open an existing file for appending data
`rb`	Open for reading binary data
`wb`	Open for writing binary data

Note: '+' appended to the end of he mode indicates that the file is for both reading and writing.

Example: Opening and closing a file in Python in 'read' mode

```
fp = open('/path/to/file.txt', 'r')
fp.close()
```

Reading and writing files

open() and close() functions

Instead of having to explicitly close a file, Python has a helpful way to automatically cleanup after yourself:

```
with open('/path/to/file.txt', 'r') as fp:
    # do some work with fp
```

This is equivalent to the following (where close() is always run)

```
try:
    fp = open('/path/to/file.txt', 'r')
    # do some work with fp
finally:
    fp.close()
```

Reading a file

The file object returned by open() has 3 methods to read data:

- `read()` – stores all the data into one text string; useful for small files where you want to do text manipulation on the entire file
- `readlines()` – reads all the lines of the file at once, and returns a list of strings (each element corresponds to one line)
- `readline()` – reads individual lines one at a time (increments); each time it is called, it reads another line. Great for handling very large files one line at a time

Reading a file line-by-line

```
with open('/path/to/file.txt', 'r') as fp:
    line = fp.readline()
    count = 0
    while line:
        count += 1
        print("[{}]: {}".format(count, line.strip()))
        line = fp.readline()
```

We can further take advantage of Python syntax to more simply easily iterate through the file object, line-by-line:

```
with open('/path/to/file.txt', 'r') as fp:
    for count, line in enumerate(fp):
        print("[{}]: {}".format(count, line.strip()))
```

Writing files

Let's create a new text file:

```
with open('/path/to/somefile.txt', 'w') as fp:
    for i in range(10):
        fp.write("This is line {:.0f}\n".format(i))
```

If somefile.txt does not already exist, then a new file with that name will be created (under 'w' mode).

WARNING: If your file already exists, its previous contents will be overwritten and lost.

Common data file formats

Datasets (tables) are often shared in one of the following file formats:

- CSV (Comma Separated Values) – Highly compatible, and prevalent format. Not easily human readable.
- TSV (Tab Separated Values) – Easy to read for humans, easy to work with, and often more efficient for software. More sound delimiters (as tabs rarely show-up in datasets, unlike commas)

Note: Both file formats have a plain text data format, thus a different extension (e.g. using .txt for TSV files) does not affect the ability of a program to read it.

Reading and writing files

Reading TSV or CSV line-by-line

Example Python function:

```
def readFile(filepath, sep):
    try:
        with open(filepath, 'r') as fp:
            for line in fp:
                rowArray = line.strip().split(sep)
                # do something with rowArray
    except FileNotFoundError as e:
        print("Error: File not found")
    except OSError as e:
        print("System-related or IO error.")
    except Exception as e:
        print("Sorry, can't read the file.")
    return
```

FASTA Files

- FASTA files (like CSV or TSV files) are plain text data, used to represent nucleotide or peptide sequences
  - Common extensions: .fasta and .fa
  - A sequence in FASTA format begins with a single-line description (preceded by '>'), followed by lines of sequence data
  - Files can contain 1 to 1000's of sequences
- Just like other plain text files, you can read FASTA files line by line (which is very helpful when working with large sequence files)

Challenge: Write your own FASTA file parser (and any associated classes) that takes a file path as an input.

Bonus

JSON file format

- JavaScript Object Notation (JSON): An open-standard, human-readable file format for data transfer
  - Language-independent format – many languages have tools and libraries to parse them
  - Text file; uses .json extension

Example of a JSON format:

```
{
  “firstname”: “Eisha”,
  ”lastname”: “Ahmed”,
  “isStudent”: true,
  ”favNumber”: 163,
  “phoneNumbers”: [
    {
      “type”: “home”,
     “number”: “514-123-4567”
    },
    {
      “type”: “home”,
     “number”: “514-123-4567”
    }
  ]
}
```

Bonus

Reading JSON files in Python

Reading a json file into a Python dictionary using `json.load()`:

```
import json

with open('/path/to/file.json', 'r') as f:
    jsonData = json.load(f)

print(jsonData)
```

Application: JSON format could be used as basic configuration files, or as a way to save and represent objects and their attributes.

Final Challenge

Modelling a protein interaction network

Putting it all together

Final Challenge - Background

You will be working with data from STRING, a biological database of known and predicted protein-protein interactions.

Preparation: Human protein interaction data was downloaded and pre-processed from the STRING database website

Download the following file (link below):
https://github.com/EishaAA/Python-Workshops/blob/master/datasets/stringdb_human_filtered_interactions_700.tsv

Pre-processing details

- **STRING database contents for human proteins**: https://string-db.org/cgi/download.pl?species_text=Homo+sapiens
- **Raw data files used:** 9606.protein.links.detailed.v11.0.txt.gz (110.1 Mb) and 9606.protein.info.v11.0.txt.gz (1.9 Mb)
- Protein interactions with filtered to include only those with a total score >= 700 (only evidence sub scores from experiments, fusion proteins, and curated databases were included)
- STRING protein IDs were mapped to their common name (alias) when possible
- Results were finally exported to a table (.tsv file), where each row corresponds to one protein-protein interaction. The file contains three (3) columns: the first two contain the protein names, and the third is the total interaction (computed as described above).

Future Challenge: After the workshop, have a go at performing this pre-processing yourself!

Putting it all together

Final Challenge

Objective: Write a program that will read from the parsed tsv file, model this data, and make queries. Once you write your program, you should be able to do the following:

- Given a protein name, list all the other proteins interact with it
  - Q: What are all the proteins reported to interact with 'FOXP3' in the dataset?
- Retrieve a list of all the proteins in the network, sorted by the number of interactions
  - Q: Which 10 proteins in the network have the greatest number of interacting partners?

Tips:

- Feel free to brainstorm, share ideas, and/or cooperate with those around you
- Keep your code organized for easy debugging (and reuse!)
- Consider using classes to organize your code (e.g. creating a `Protein` class and `GeneNetwork` class). What kind of methods/attributes might they have?
- Think about what other questions you could ask of the data, and what functionality you could add to your program that would help you answer those questions. If you have extra time, have a go at it!

For questions or conversations on Python programming

```
"{}.{}@mail.mcgill.ca".format("eisha", "ahmed")
```

Bonus: Run `import this` in Python