COMP 204: Computer Tools for Life Sciences

Python programming: File Input/output (IO)

Yue Li
based on material from Mathieu Blanchette, Christopher J.F. Cameron and Carlos G. Oliver

Storing data in programs

Until now: Data analyzed in our programs are stored in variables.
Data is either:

- hard-coded in the program, e.g.,
  ```
  people = {"Mathieu":33,"Maria":23,"Jaspal":28}
  ```
  Not good because too inflexible.
  If a user wants to change the data, they need to change the program (but they might not know how)
- OR
- input by the user via the keyboard. e.g.,
  ```
  age = int(input("Enter patient age"))
  ```

Problem #2: When the program's execution ends, the result of the computation is gone!

File types

Files are ways to store data that will survive beyond the life of the execution of a program.

- Text files: sequence of characters
  - Python programs
  - Text data (e.g. html (web) files)
  - Tabular data (e.g. tab-separated file)
- Binary files: sequence of bytes that can be interpreted as numbers
  - Images
  - Sound
  - Any kind of compressed data (e.g. zipped file)
  - compiled program
  - etc. etc.

In order for a program to use files, we need to:

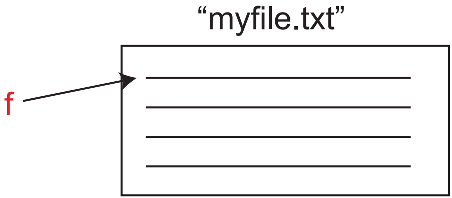
- Read files: Get data from file loaded into a program's variables
- Write files: Write the values of variables into a file to save the the information beyond the execution of the program

Reading files in Python

To read the content of a file, you need to:

- Open file: This creates a *file-stream* object. When we open a file, the file-stream points to the beginning of the file. Opening a file does not actually read the file.
- Read data (usually line by line). At any given point during the execution of the program, the file stream is at one location in the file. As you read more data, the position of the file stream moves forward in the file.
- Close file: Tells the operating system that you no longer need to access the file.

```
# file-stream object
f = open(“myfile.txt” “r”)
file_content = f.read()
f.close() # close the file
```



Opening a file

Python’s built-in **open()** function returns a file-stream object

- most commonly used with two arguments
  1. *filename* - filepath to the file to be read/written to
  2. *mode* - mode to open a file

```
# Create a string containing the full path to the
# file we want to open
filename = ”/Users/yueli/Lectures/20/patients.txt”

# open the file in reading mode
f = open(filename,”r”)

# read the content of the file, save it
# in string variable file_content
file_content = f.read()

#print(”a\tb\tc\nd\tf”)

# print the content of the file:
print(file_content)

#close the file
f.close()
```

Reading a file

`.read(size)` - Python built-in file-stream function

- reads some quantity of data and returns *single long string*
  - or bytes object in binary mode
- *size* is an optional numeric argument
  - in number of characters
- if *size* is omitted or negative
  - the entire contents of the file will be read and returned

```
f = open("/Users/yueli/Lectures/20/patients.txt","r")
file_content = f.read()
# Mike\t20\t65\t1.83\nMathieu\t33\t75\t1.81\nMaria\t23\t58\t
    1.64\nJaspal\t34\t56\t1.76\nAhmed\t65\t83\t1.78
```

Python common file opening modes

r: `f = open(myfile,'r')`

- opens a file for reading only
- file stream position is at the beginning of the file
- default mode

w: `f = open(myfile,'w')`

- opens a file for writing only
- overwrites the file if the file exists
- if the file does not exist, creates a new file for writing

a: `f = open(myfile,'a')`

- opens a file for appending
- if the file exists, file stream position is at the end of the file
- if the file does not exist, it creates a new file for writing

Python additional file opening modes

Adding **b** to a mode

- `f = open(myfile,'b')`
- opens a file in binary format

Adding + to a mode

- `f = open(myfile,'wr+')`
- opens a file for both writing and reading

For example, `f = open(myfile,'ab+')` would open a file for appending in binary format

What would the mode `f = open(myfile,'wb+')` open a file as?
Answer: open a file in binary format and writing it

Reading a file #2

.readlines(size) - Python built-in file-stream function

- reads all the remaining lines returns them as a *list of strings*
- Note: the end-of-line character '\n' is included at the end of each string (except the last one).
  - First line is ”Mathieu\t43\t75\t1.8\n”
  - An empty line is just ”\n”
  - We can remove '\n' in a String using the `rstrip()` function.
- Conveniently reads all content of the file and breaks it down into individual lines

```python
f = open("/Users/yueli/Lectures/20/patients.txt","r")

all_lines=f.readlines() # lines is a list of strings

for line in all_lines:
    print("The line is",line.rstrip())
    #print("The line is",line) # remove comment see what happens

f.close()
```

Reading a file #3: read line by line

We often don't want to read all the lines of a file at once.

- Issue: sometimes the file may be too large to fit in memory
- Instead, we use a `for` loop.
- At each iteration, read **only one line of the file** into memory
- By default, `split()` function breaks down a string into a list of strings by white space. It can use other delimiters as optional argument such as `values=line.split(sep=",")`.

```
# open file called patients.txt,
data_file = open("/Users/yueli/Lectures/20/patients.txt",
↪  "r")

line = data_file.readline()
print(line)

line = data_file.readline()
print(line)


# read the file one line at a time
```

Take action according to the content of each line of the file

We sometimes need more control over when we read lines.

Example: The first line of the file may be a header line that needs to be processed differently from the rest.

`.readline()` reads a *single line* from the file

- Returns an empty string "" if the end of file has been reached
- End-of-line character '\n' is included at the end of each string.

```python
f = open("/Users/yueli/Lectures/20/patients2.txt", "r")

line=f.readline() # patients2.txt has a header line
column_headers = line.split()

while True:
    line = f.readline()
    if line=="":  # we've reached the end of the file
        break
    values = line.split('\t')
    print(column_headers[0],":",values[0])

f.close()
```

Writing files in Python

To *write* data to a file, you also need to create a *file stream*.

- Open file: This creates a file-stream object, ready for write data into.
- Write data (usually line by line, or byte by byte). Data needs to be written in the order in which you want it to be stored in a file.
- Close file: Tells the operating system that you are done writing to it.

`.write(` *string*) writes a string to the file

Put it together: Example of reading and writing files

Example: Read patient data, calculate BMI for each, and print name and BMI to file BMI.txt.

```python
inpdir = outdir = "/Users/yueli/Lectures/20/"

input_file = open(inpdir+"patients.txt", "r")

# open BMI.txt as an output file.
output_file = open(outdir+"BMI.txt", "w")

for line in input_file:
    name,age,w,h = line.split()
    output_string = name + " has BMI " + \
        str(float(w)/float(h)**2) + "\n"
    print(output_string)
    #output_file.write(output_string)

input_file.close() # close input file
output_file.close() # close output file
```

An application in life science: Reading FASTA format

FASTA format is a file format for DNA and protein sequences
Example:

```
>Human
ACGACTACGACTACGACATCATCAGCAGCATCAGCAGCATCGAGCGACATCAGCAGACT
GACATCATCAGCGACATCTACGACTCATAATATTACATCAGCATCATATCAGCATCATA
AGCAGATCATCATGAC
>Chimp
TAAGAGAGCAGCAGACTCACTCTCTCTCAGCAGCAGCATCTACGACTACATCTACGATA
CGACATCAGCCGACTACATCTTACATCATCATCGGCGACGACAGCTCTCATCAGCATAT
AGCAGGGGGGGGCAGCATACGACATCATCAGCGATACGACATCATCGACTCATCAGACG
GACGACTACTACTACGACATATTA
>Mouse
AGACTACATAGACAGCATCATAGATCCATCAGCATACTCAGCATGAT
```

Goal: Write a function that reads a FASTA file and returns a list of tuples of the form (name,sequence).

Parsing a FASTA file: an algorithm

Challenge: The sequences are broken up in chunks of up to 60 characters. Different sequences may have different lengths.

Idea:

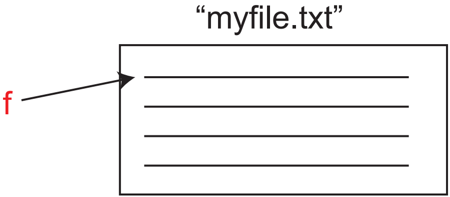
- Read file one line at a time, keeping track of (i) the last sequence name encountered, and (ii) the concatenation of the sequences encountered.
- If a line does not start with ">", it is a sequence line, so add it to the growing sequence being read
- If a line starts with ">", it is either the first line in the file, or it is not.
  - if it is the first line, them just read the name from the line, and set sequence to empty
  - if it is not the first line, then we already have stored a name and sequence by the time we got here, so we need to add them to our list of tuples before reseting them
- If a line is empty, we've reach the end of the file. Add the last name and sequence to our list

```python
def read_fasta(filename):
    """
    args:
       filename: name of FASTA file to read
    Returns:
       A list of tuples, each tuple containing
       the name of the sequence and the sequence iself
    """
    f = open(filename,"r")
    name = ""  # initialize name and seq to empty strings
    seq = ""
    list_of_seq = []   # accumulates the tuples of sequences seen so far
    while (True):
        line = f.readline().rstrip() # read a line
        if line == "":  # we've reached the end of the file
            list_of_seq.append( (name,seq) )  # add the last sequence read
            break
        elif line.startswith(">"):  # start of new sequence
            # if this is not the first sequence read in the file,
            # there is already a name and seq stored, so we add it to the list

            # reset name to the new name contained in line. reset seq to empty
            if name!="":
                list_of_seq.append( (name,seq) )

            name = line[1:] # remove the ">" character
            seq = ""  # start a new, empty sequence

        else:  # we're reading a line of sequences
            seq = seq + line
        # end of while loop
    return list_of_seq

sequences = read_fasta("/Users/yueli/Lectures/20/seq.fa")
print(sequences)
```

File IO review (added on 02/22/2019)

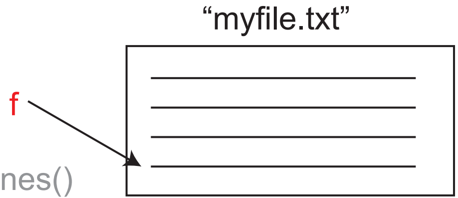
File IO review: `.read()`

```
# file-stream object
f = open(“myfile.txt” “r”)
```



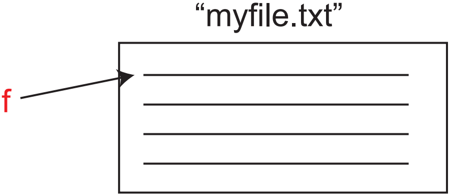
File IO review: `.read()`

```
# file-stream object
f = open(“myfile.txt” “r”)
file_content = f.read()
 # same applies to f.readlines()
```



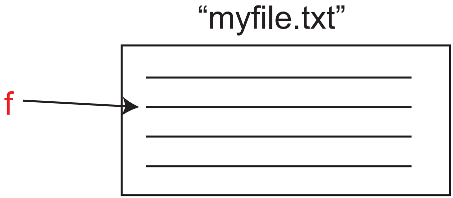
File IO review: `.readline()`

```
# file-stream object
f = open("myfile.txt" "r")
```



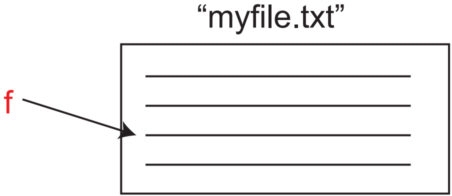
File IO review: `.readline()`

```
# file-stream object
f = open(“myfile.txt” “r”)
first_line = f.readline()
```



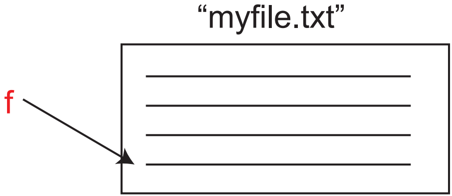
File IO review: `.readline()`

```
# file-stream object
f = open(“myfile.txt” “r”)
first_line = f.readline()
second_line = f.readline()
```



File IO review: `.readline()`

```
# file-stream object
f = open(“myfile.txt” “r”)
first_line = f.readline()
second_line = f.readline()
third_line = f.readline()
```



File IO review: `.read()` vs `.readlines()`

```python
filein = "/Users/yueli/Lectures/20/Lecture20_code/inpfile.txt"

# 1
print("way #1: f.read()")
f = open(filein, 'r')
x = f.read()
print(x)
f.close()

# 2
print("way #2: f.readlines()")
f = open(filein, 'r')
x = f.readlines()
print(x)
f.close()
```

File IO review: `.readlines()` vs `.readline()`

```python
# 2
print("way #2: f.readlines()")
f = open(filein, 'r')
x = f.readlines()
print(x)
f.close()

# 3
print("way #3: f.readline()")
f = open(filein, 'r')
x = f.readline()
print(x)
x = f.readline()
print(x.rstrip())
x = f.readline()
print(x)
f.close()
```

File IO review: `while`-loop vs `for`-loop

```python
# while-loop example
print("way #4: read line by line in while-loop")
f = open(filein, 'r')
x = f.readline()
while x != "":
    print(x)
    x = f.readline()
f.close()

# for-loop example
print("way #5: read line by line for-loop")
f = open(filein, 'r')
for x in f:
    print(x)
f.close()
```

File IO review: `.write`

```python
# write
inpfile = "/Users/yueli/Lectures/20/Lecture20_code/inpfile.txt"
outfile = "/Users/yueli/Lectures/20/Lecture20_code/outfile.txt"

filein = open(inpfile, 'r')
fileout = open(outfile, 'w')

count = 0

for line in filein:
    fileout.write(str(count) + ": " + line)
    count += 1

filein.close()
fileout.close()
```

Some other read/write functions and libraries useful for Assignment 3 (go over on Monday lecture)

JSON module

Strings can easily be written to and read from a file

Numbers take a bit more effort to read/write

- the `read()` method only returns strings, so we need to convert them to integers using `int()`
- the `write()` method accepts strings as arguments, so we need to covert numbers to strings before writing them.

Also: What if you want to save more complex data types like nested lists or dictionaries?

- parsing and serializing by hand becomes complicated
- serializing: converting an object to a string that allows the object and state to be more easily recreated

Serializing objects with JSON

Rather than having users constantly write code to read/write complex data, Python allows you to use the popular data interchange format called **JSON (JavaScript Object Notation)**

`json.dump()`	serializes an object to a text file
`json.load()`	loads serialized object from text file

```python
import json

outfile = "/Users/yueli/Lectures/20/my_file.json"
some_data = [1, 'simple', {'Yue':2.0,'Maria':3.0}]
f = open(outfile,"w")
json.dump(some_data,f) # write object into json file
f.close()

f = open(outfile,"r") # load object from json file
my_data = json.load(f)  # some_data is a list
print(my_data) # [1, 'simple', {'Yue': 2.0, 'Maria': 3.0}]
f.close()
```

Reading/writing gzip compressed files

gzip.open() provides an interface to read/write compressed files

- gzip files save *a lot of* disk space (e.g., DIAGNOSES_ICD.csv (18M) vs DIAGNOSES_ICD.csv.gz (4.5M)) (651,048 rows)
- files typically end with the '.gz' extension
- available modes: r, a, and w along with binary options

```python
import gzip

# a comma separated value (csv) file
gzfile = "/Users/yueli/Lectures/20/DIAGNOSES_ICD.csv.gz"
f = gzip.open(gzfile, "r")

#  .decode() converts bytes to string
line = f.readline().decode("utf-8")
print(line.rstrip()) #
↪  "ROW_ID","SUBJECT_ID","HADM_ID","SEQ_NUM","ICD9_CODE"
line = f.readline().decode("utf-8")
print(line.rstrip()) # 243,34,115799,8,"E8790"

f.close()
```

Reading a csv file using `pandas.read_csv()` function

`pandas.read_csv` provides an easy way to read comma-separated value (csv) file as a `DataFrame` object (more in later lectures)

```python
import pandas as pd

filename = "/Users/yueli/Lectures/20/DIAGNOSES_ICD.csv.gz"

patient_data = pd.read_csv(filename, compression="gzip")

patient_records = {} # save patient ICD-9 code into dictionary
for index,row in patient_data.iterrows(): # iterate row by row

    patId = row['SUBJECT_ID'] # access column "SUBJECT_ID"
    icd9_code = row['ICD9_CODE'] # access column "ICD9_CODE"

    patient_records.setdefault(patId, []).append(icd9_code)

    if index > 100: # iterate only the first 100 rows
        break

for k,x in patient_records.items():
    print(k, x ,sep='\t')
```