COMP 364: Computer Tools for Life Sciences

Python programming: File IO

Christopher J.F. Cameron and Carlos G. Oliver

Reading/writing files in Python

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

```
1 with open(filepath,"r") as f:
2     read_date = f.read()
3 f.closed()  # returns True
4
5 # or a less pythonic way
6 f = open(filepath,"r")
7 read_date = f.read()
8 f.close()
9 f.closed()  # returns True
```

Python common file modes

r

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

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

- 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 modes

Adding **b** to a mode

- opens a file in binary format

Adding **+** to a mode

- opens a file for both writing and reading

'*newline* = *None*' universal read line mode

For example, **ab** would open a file for appending in binary format

What would the mode **wb+** open a file as?

What's a file stream?

A **file stream** is the way Python reads in a file

- the stream consists of characters

For example, the following text file, 'secrets.txt':

```
1 # COMP 364 MIDTERM SOLUTIONS
2 **DO NOT SHARE WITH STUDENTS
3 Q1) The solution is clearly
```

What the file stream looks like:

'# COMP 364 MIDTERM SOLUTIONS\n**DO NOT SHARE WITH STUDENTS\nQ1) The solution is clearly\n'

Reading a file

.read(size) - Python built-in file-stream method

- reads some quantity of data and returns it as a 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

```
1 with open("secret.txt","r") as f:
2     print(f.read(10))
3 # prints: # COMP 364
```

Reading a file #2

.readline() reads a single line from the file

- a newline character ('\n') is left at the end of the string
- '\n' is omitted on the last line of the file
  - if the file doesn't end with '\n'

A blank line will be represented by '\n'

If **.readline()** returns an empty string

- the end of the file has been reached

```
1 with open("secret.txt","r") as f:
2     print(f.readline())
3 # prints:
4 #'# COMP 364 MIDTERM SOLUTIONS
5 #'
```

A more Pythonic way

For reading lines from a file

- you can loop over the file object
- this is memory efficient, fast, and leads to simple code

```
with open("secret.txt","r") as f:
    for line in f:
        print(line)
# prints:
#'# COMP 364 MIDTERM SOLUTIONS
#
#**DO NOT SHARE WITH STUDENTS
#
#Q1) The solution is clearly
#'
```

Reading a file #3

If you want to read all the lines of a file in a list

- you can use **list(***file-stream object***)**

To read the remaining lines in a file

- **.readlines()**

```
1  with open("secret.txt","r") as f:
2      lines = f.readlines()
3
4  with open("secret.txt","r") as f:
5      lines_2 = list(f)
6
7  print(lines==lines_2)    #   prints True
```

Writing to a file

.write(*string***)** writes the contents of string to the file

- returning the number of characters written

```
1  with open("tmp.txt","w") as f:
2      print(f.write("# COMP 364 MIDTERM SOLUTIONS"))
3  # prints: 28
4
5  lines = ["# COMP 364 MIDTERM SOLUTIONS",
6           "**DO NOT SHARE WITH STUDENTS",
7           "Q1) The solution is clearly"]
8  with open("tmp.txt","w") as f:
9      print(f.write("\n".join(lines)))
10 # prints: 85
```

Methods to track the stream

.tell() returns the file-stream's current position

- position is an integer
- relative to the beginning of the file
- number is in characters in text mode
  - bytes in binary mode

```
1  with open("secret.txt","r") as f:
2      print("pos:",f.tell())
3      # .rstrip() removes the newline
4      print(f.readline().rstrip())
5      print("pos:",f.tell())
6  # prints:
7  # pos: 0
8  # # COMP 364 MIDTERM SOLUTIONS
9  # pos: 29
```

Methods to track the stream

.seek(*offset***,** *from_what***)** changes the file-stream's position

- position is computed from adding offset to a reference point
- reference point is selected by the *from_what* argument
  - 0 measures from the beginning of the file
  - 1 uses the current file position
  - 2 uses the end of the file as the reference point
  - defaults to 0
- in text files, only seeks relative to the beginning of the file are allowed
  - binary files allow other *from_what* options

```
1 f = open("secret.txt","r")
2 f.seek(5,0)
3 print(f.read(5)) # prints: 'P 364'
```

gzip compressed files

gzip.open()

Provides a simple interface to compress/decompress binary files

- files typically end with the '.gz' extension
- available modes: r, a, and w
  - along with binary options (i.e., ab)

```
1  import gzip
2
3  with gzip.open("secret.txt.gz", "r") as f:
4      #   .decode() converts bytes to string
5      print(f.readline().decode("utf-8"))
6      # prints: '# COMP 364 MIDTERM SOLUTIONS
7      # '
```

JSON module

Strings can easily be written to and read from a file

Numbers take a bit more effort

- since the **read()** method only returns strings
- will have to be passed to a function like **int()**
  - which takes a string like '123'
  - returns its numeric value 123

When you want to save more complex data types like nested lists and 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 writing and debugging code

- Python allows you to use the popular data interchange format
- called **JSON (JavaScript Object Notation)**
- to save complicated data types to files

.dumps() returns JSON formatted str using a conversion table

```
1 import json
2
3 json_object = json.dumps([1, 'simple', (2.0,3.0)])
4 print(json_object)
5 # prints: [1, "simple", [2.0, 3.0]]
```

JSON conversion table

Python	JSON
dict	object
list, tuple	array
str	string
int, float, int- & float-derived Enums	number
True	true
False	false
None	null

Reading/writing JSON files

.dump() serializes an object to a text file

```
1 import json
2
3 with open("./tmp.json","w") as f:
4     json.dump([1, 'simple', (2.0,3.0)],f)
```

.load() loads serialized object from text file

```
1 import json
2
3 with open("./tmp.json","r") as f:
4     json_var = json.load(f)
5 print(json_var) # [1, 'simple', [2.0, 3.0]]
```

FASTA format

FASTA format is a text-based format

- can represent either nucleotide or peptide seuqences
- nucleotides or amino acids are represented as single-letter codes
- FASTA refers to "FAST-All" because it works with any alphabet

The first line of a FASTA file always starts with either '>' or ';'

- ';' indicates a comment line
  - comments not typically used
- '>' identifies a line that provides a unique description of the sequence

FASTA format #2

After a description line

- the sequence itself is described in standard one-letter code
- repetitive sequences are typically shown in lower case

```
1  ;example FASTA file
2  >sequence_1
3  ADQLTEEQIAEFKEAFSL
4  >sequence_2
5  LCLYTHIGRNIYYGSYLY
6  >sequence_3
7  LLILILLLLLLALLSPDM
```

Example FASTA file

```
>hg19 - chr22 - random_sample
AGATGATGATGTAAAATGTCTTACAAGGTAAAAAAAATGACTTTCAAATA
TTAGTGGGTTTTACTGTGAGAATTATAACTACTTCATTACAGCTTTATAC
TTGTATTTTATGTGTATTTAAACTTTTTAGATGTAAAACTTTTGTGTTCA
AAATATGTAAAGACACTAATCTTTATTACTACTTTTTCTTGACCGATAGA
CTTTCAGGAAAAATAAATGTGCGAGAGCGGTATGTTTGGGAAGTTATTGT
TGTCAGTTTATGAAGAATAGTCTACAGTTATTGGGAAATAAGATACATAA
AGCCTCAGATTGCATTTATGTTATGATGAGATAGATAAAGGTATTATTTG
AGAAACTCATTGTGTTGAGTCTAAGAAACAATTGATTTCCTGATTCAAAC
ACCAGAGATAGACCAAAAAAGGAAGTAATTAAGTCTACTTTAATGATAAA
TACTTATTGACACATATCAGAAAGTGATTAAACACTATGGACTGTATAAT
AAGCATTTACATATGTTTCTTTGACAAAGCCTAGCTTTATAATAcggtcg
tctctcagtatctgtcagggattggttccaggaaccacccccccaaactcc
tgcccacatctcactcccatgaacactaaaatccacagactcaagtccct
gatacaaaatgtcatagtatttgcatataaactatgcacatcctcccata
tattttaaatatttttagattacttataatatctaatacaatataaatgt
```

Exercise

Now that we know basic file IO methods

- let's read in an example FASTA file: 'hg19.chr22.ref_genome.sample.txt.gz'

Step 1: open the file for reading

Step 2: read two lines at a time

Step 3: track position of *file-stream*

Step 4: end file parsing if at end of file

Step 5: print description and sequence to user

Step 6: convert bytes objects to 'utf-8' strings

Step 7: check that proper FASTA format is followed

Possible Python implementation

```
1  import gzip
2
3  filepath="./hg19.chr22.ref_genome.sample.txt.gz"
4  with gzip.open(filepath, "r") as f:
5      while True:
6          line_1 = f.readline().decode("utf-8").rstrip()
7          line_2 = f.readline().decode("utf-8").rstrip()
8          if not len(line_2) == 0:
9              if line_1.startswith(">"):
10                 print("\n".join([line_1, line_2]))
11                 continue
12         break
```