Communicating with the Outside

Python for Genomic Data Science

Reading and Writing Files

To read or write files use the built-in function `open(filename, mode)`

Reading from a file

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

`'r'` is the default value for the mode parameter, so we can just omit it:

```
>>> f=open('myfile')
```

Writing into a file

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

If the file `'myfile'` already exists using mode `'w'` truncates its content first. To append to the end of the file, if it exists, use mode `'a'`:

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

The result of the `open()` function is a file object.

Errors When Opening a File

If you attempt to open a file that does not exist, Python will produce an error message:

```
>>> f=open('myfile')
Traceback (most recent call last):
  File "<pyshell#55>", line 1, in <module>
    open('myfile')
FileNotFoundError: [Errno 2] No such file or
directory: 'myfile'
```

A common way to let your program handle this type of error properly is to specify what to do in case of errors:

```
>>> try:
...     f = open("myfile")
... except IOError:
...     print("the file  myfile does not exist!!")

the file  myfile does not exist!!
```

IOError is a built-in Python exception which is raised if the file we try to open does not exist.

Reading From a File

myfile

```
This is the first line of the file.
Second line of the file.
```

An efficient and fast way to read the content of a file is by looping over the file object:

```
>>> for line in f:
...     print(line)
...
This is the first line of the file.

Second line of the file.
```

The content of a file can be also read using the `read()` method of the file object `f`:

```
>>> f.read()

>>>
```

The `read()` method returns the content of the file object `f` from the current position in the file to the end of the file.

Nothing gets printed here. Why?

Changing Positions Within a File Object

To change the file object's position, use `f.seek(offset, from_what)`. The position is computed from by adding `offset` to a reference point; the reference point is selected by the `from_what` argument, which in text files is only allowed to be 0 signifying the beginning of the file:

```
>>> f.seek(0)
0
>>> f.read()
'This is the first line in the file.\nThis is
the second line in the file.\n'
```

Go to the beginning of the file.

You can also read a single line from the file:

```
>>> f.seek(0)
>>> f.readline()
'This is the first line in the file.\n'
```

Writing Into a File

myfile

```
This is the first line of the file.
Second line of the file.
This is a third line.
```

`f.write(string)` writes the contents of string to the file, returning the number of characters written in Python 3.x or `None` in Python 2.x.

```
>>> f=open('/Users/mpertea/Work/courses/
python/myfile','a')
>>> f.write('This is a third line')
20
```

Closing a File Object

When you're done with a file, call `f.close()` to close it and free up any system resources taken up by the open file:

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<pyshell#100>", line 1, in <module>
    f.read()
ValueError: I/O operation on closed file.
```

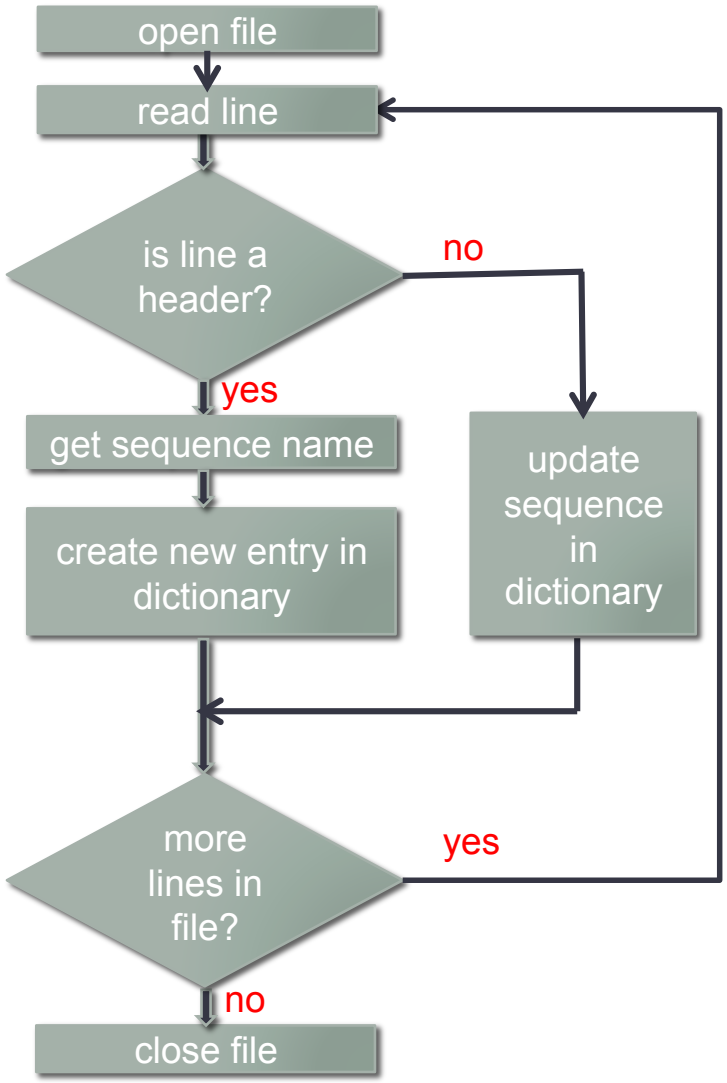
After calling `f.close()`, attempts to use the file object will automatically fail.

Reading a FASTA File

Exercise: Build a dictionary containing all sequences from a FASTA file.

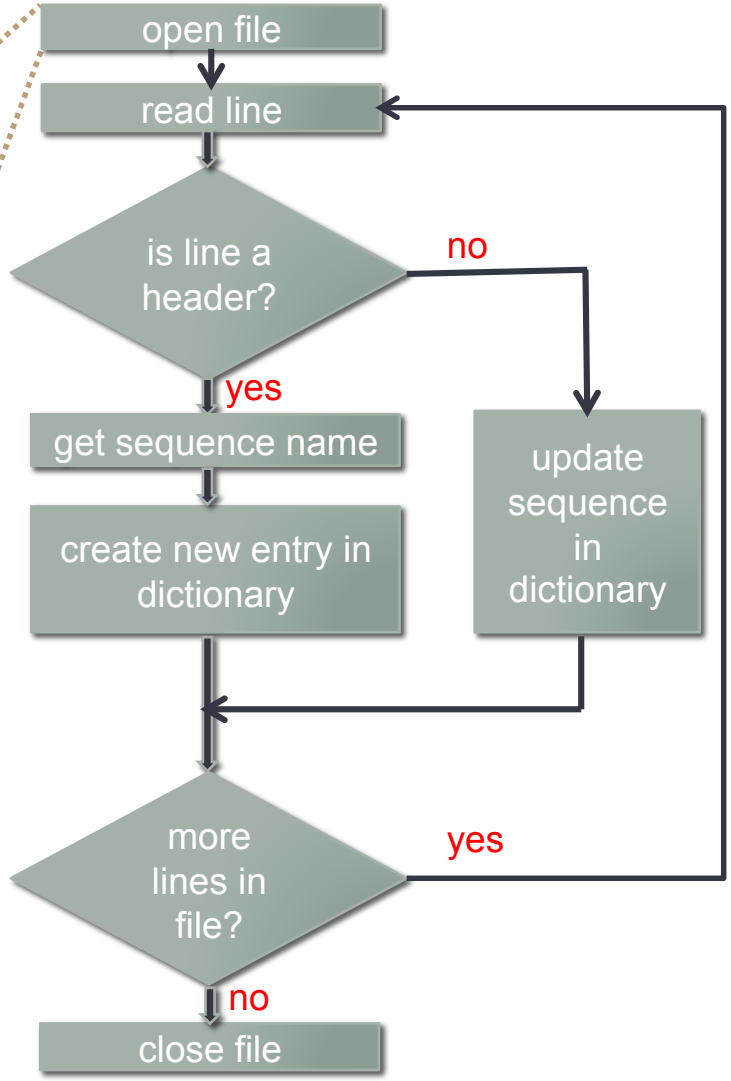
FASTA file:

```
>id1 description of id1
ATGTGTGTCCGTTGTGTAA
AGTGTGTCcccgtgttATg
gtagatttttga
>id2 description of id2
ccccagtggggagtagggc
AAAcgtatAA
```



Reading a FASTA File (cont'd)

```
try:
    f = open("myfile.fa")
except IOError:
    print("File  myfile.fa does
not exist!!")
```

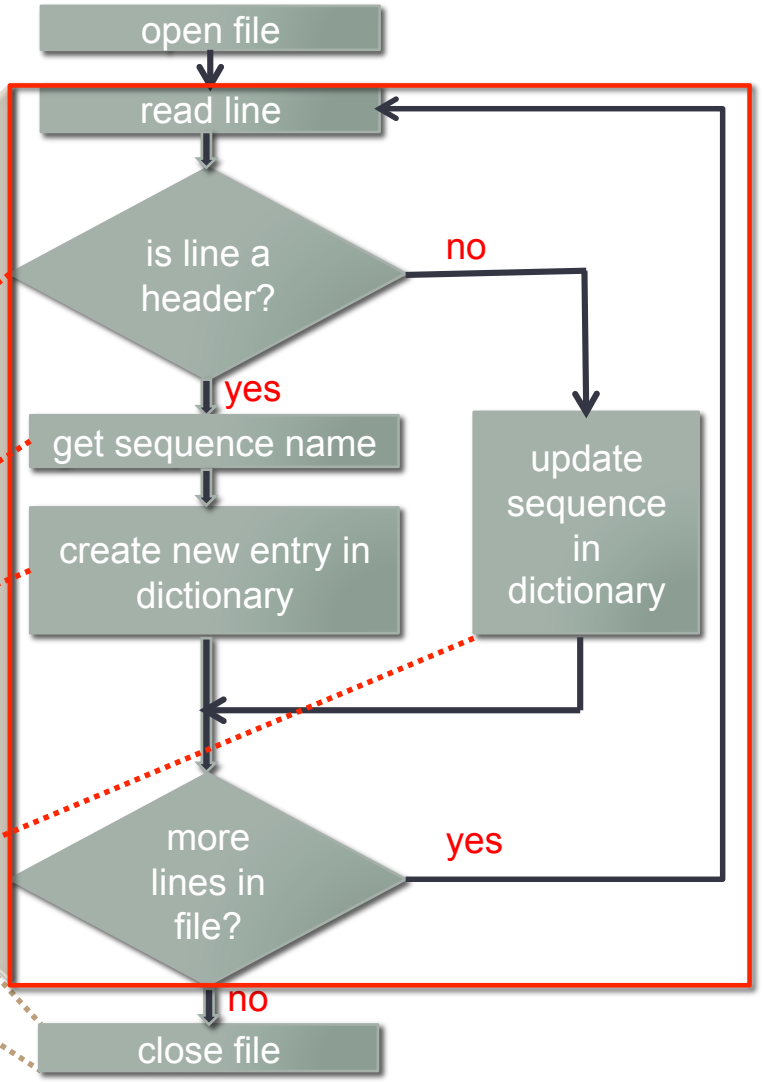


Reading a FASTA File (cont'd)

The method `rstrip([chars])` returns a copy of the string with trailing characters `chars` removed.

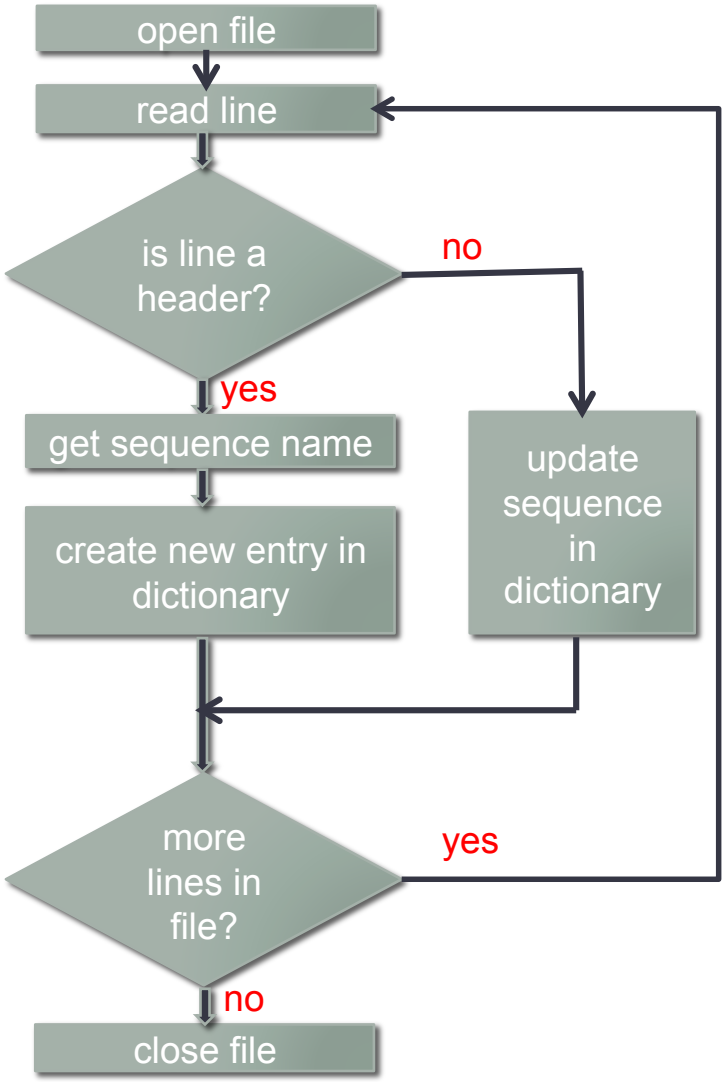
```
seqs={}
for line in f:
    # let's discard the newline at
the end (if any)
    line=line.rstrip()
    # distinguish header from sequence
    if line[0]=='>': # or
line.startswith('>')
        words=line.split()
        name=words[0][1:]
        seqs[name]=''
    else : # sequence, not header
        seqs[name] = seqs[name] + line

close(f)
```



Reading a FASTA File (cont'd)

```
try:
    f = open("myfile.fa")
except IOError:
     print("File  myfile.fa does
not exist!!")
seqs={}
for line in f:
    # let's discard the newline at
the end (if any)
    line=line.rstrip()
    # distinguish header from sequence
     if line[0]=='>': # or
line.startswith('>')
        words=line.split()
        name=words[0][1:]
        seqs[name]=''
     else : # sequence, not header
        seqs[name] = seqs[name] + line
close(f)
```



Retrieving Data From Dictionaries

We can retrieve the key and corresponding value from our dictionary using the `items()` method:

```
>>> for name,seq in seqs.items():
        print(name,seq)

id1 ATGTGTGTCCGTTGTGTAAAGTGTGTCcccgtgttATggtagatttttga
id2 ccccagtggggagtagggcAAAcgtatAA
```

Command Line Arguments

Scripts often need to process command line arguments.

Suppose a script that parses a FASTA file is called `processfasta.py`, and you want to run it on a file whose name we give as an argument in the command line:

```
>python processfasta.py myfile.fa
```

The arguments of the above command are stored in the `sys` module's `argv` attribute as a list:

```
import sys
print(sys.argv)
['processfasta.py','myfile.fa']
```

sys.argv[0] is the script's name.

Reading the Command Line Arguments in `processfasta.py`

processfasta.py

```
#!/usr/bin/python
"""
processfasta.py builds a dictionary with all sequences
from a FASTA file.
"""

import sys
filename=sys.argv[1]

try:
    f = open(filename)
except IOError:
     print("File  %s does not exist!!" % filename)
...
```

Parsing Command Line Arguments With `getopt`

Python's `getopt` module can help with processing the arguments of `sys.argv`.

Suppose the `processfasta.py` script reads a FASTA file but only stores in the dictionary the sequences bigger than a given length provided in the command line:

```
> processfasta.py –l 250 myfile.fa
```

Usage Definition For `processfasta.py`

```
def usage():
    print """
processfasta.py : reads a FASTA file and  builds a
dictionary with all sequences bigger than a given
length

processfasta.py [-h] [-l <length>] <filename>

 -h              print this message

 -l <length>     filter all sequences with a length
                 smaller than <length>
                (default <length>=0)

 <filename>      the file has to be in FASTA format

        """
```

[ ] mark optional arguments.

getopt Usage Example

```
#!/usr/bin/python
import sys
import getopt
def usage(): ...

o, a = getopt.getopt(sys.argv[1:], 'l:h')
opts = {}
seqlen=0;

for k,v in o:
    opts[k] = v
if  '-h' in opts.keys():
    usage(); sys.exit()
if len(a) < 1:
    usage(); sys.exit("input fasta file is missing")
if '-l' in opts.keys():
    if int(opts['l'])<0 :
        print("Length of sequence should be positive!"); sys.exit(0)
    seqlen=opts['-l']
…
```

o = list of optional arguments
a = list of required arguments

when ':' is added just after, this means that the option expects a value

multiple instructions can be written on the same line if they are separated by ';'

Using the System Environment

Reminder: When we run a script/program in the UNIX environment there are standard streams recognized by a computer program:

- Standard input or `stdin` is stream data (often text) going into a program. Unless redirected, standard input is expected from the keyboard which started the program.
- Standard output or `stdout` is the stream where a program writes its output data. Unless redirected, standard output is the text terminal which initiated the program.
- Standard error or `stderr` is another output stream typically used by programs to output error messages or diagnostics. It is a stream independent of standard output and can provide error messages even when `stdout` has been redirected. `stderr` can also be redirected separately:

```
my_program | my_script.sh 1>program_output.txt 2>error_messages.txt
```

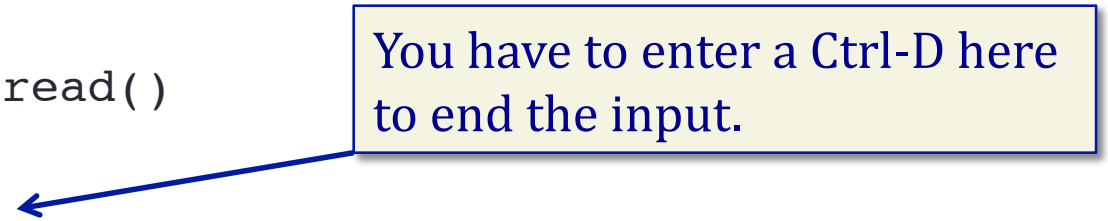
Using the System Environment (cont'd)

The sys module in Python provides file handles for the standard input, output and error:

```
>>> sys.stdin.read()
a line
another line
'a line\nanother line\n'

>>> sys.stdout.write("Some useful output.\n")
Some useful output
20

>>> sys.stderr.write("Warning: input file was not found\n")
Warning: input file was not found
34
```



Interfacing With External Programs

- You can call/execute an external program from within your script

- Helps you automate certain tasks that would be difficult for you to do within Python

Interfacing With External Programs (cont'd)

Use the `call()` function in the `subprocess` module to run an external program:

```
>>> import subprocess
>>> subprocess.call(["ls", "-l"])
0
```

return code: indicates the success or failure of the execution

A more realistic call:

```
>>> subprocess.call(["tophat","genome_mouse_idx","
PE_reads_1.fq.gz","PE_reads_2.fq.gz"])
```