File input and output and conditionals

Genome 559: Introduction to Statistical and Computational Genomics

Prof. James H. Thomas

Opening files

- The built-in `open()` function returns a file object:
  `<file_object> = open(<filename>, <access type>)`

- Python will read, write or append to a file according to the access type requested:
  - `'r'` = read
  - `'w'` = write (will replace the file if it exists)
  - `'a'` = append (appends to an existing file)

- For example, open for reading a file called "hello.txt":

```
>>> myFile = open('hello.txt', 'r')
```

Reading the whole file

- You can read the entire content of the file into a single string. If the file content is the text "Hello, world!\n":

```
>>> myString = myFile.read()
>>> print myString
Hello, world!

>>>
```

why is there a blank line here?

Reading the whole file

- Now add a second line to the file ("How ya doin'?\n") and try again.

```
>>> myFile = open('hello.txt', 'r')
>>> myString = myFile.read()
>>> print myString
Hello, world!
How ya doin'?

>>>
```

Reading the whole file

- Alternatively, you can read the file into a list of strings, one string for each line:

```
>>> myFile = open('hello.txt', 'r')
>>> myStringList = myFile.readlines()
>>> print myStringList
['Hello, world!\n', 'How ya doin'?\n']
>>> print myStringList[1]
How ya doin'?
```

notice that each line has the newline character at the end

this file method returns a list of strings, one for each line in the file

Reading one line at a time

- The **`readlines()`** method puts all the lines into a list of strings.
- The **`readline()`** method returns only the next line:

```
>>> myFile = open('hello.txt', 'r')
>>> myString = myFile.readline()
>>> print myString
Hello, world!

>>> myString = myFile.readline()
>>> print myString
How ya doin'?

>>> print myString.strip() # strip the newline off
How ya doin'?
>>>
```

notice that **`readline()`** automatically keeps track of where you are in the file - it reads the next line after the one previously read

Writing to a file

- Open a file for writing (or appending):

```
>>> myFile = open('new.txt', 'w') # (or 'a')
```

- Use the `<file>.write()` method:

```
>>> myFile.write('This is a new file\n')
>>> myFile.close()
>>> Ctl-D (exit the python interpreter)
> cat new.txt
This is a new file
```

always close a file after you are finished reading from or writing to it.

`open('new.txt', 'w')` will overwrite an existing file (or create a new one)
`open('new.txt', 'a')` will append to an existing file

`<file>.write()` is a little different from `print`

- `<file>.write()` does not automatically append a new-line character.
- `<file>.write()` requires a string as input.

```
>>> newFile.write('foo')
>>> newFile.write(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: argument 1 must be string or read-only
  character buffer, not int
>>> newFile.write(str(1))  # str converts to string
```

(also of course `print` goes to the screen and `<file>.write()` goes to a file)

Conditional code execution and code blocks

`if-elif-else`

The `if` statement

```
>>> if (seq.startswith("C")):
...    print "Starts with C"
...
Starts with C
>>>
```

- A **block** is a group of lines of code that belong together.

```
if (<test evaluates to true>):
    <execute this block of code>
```

- In the Python interpreter, the ellipsis … indicates that you are inside a block (on my Win machine it is just a blank indentation).
- Python uses indentation to keep track of code blocks.
- You can use any number of spaces to indicate a block, but you must be consistent. Using one **<tab>** is simplest.
- An unindented or blank line indicates the end of a block.

The `if` statement

- Try doing an `if` statement without indentation:

```
>>> if (seq.startswith("C")):
... print "Starts with C"
  File "<stdin>", line 2
    print "Starts with C"
        ^
IndentationError: expected an indented block
```

the interpreter expects you to be inside a code block (…)

but it is not indented properly

Multiline blocks

- Try doing an `if` statement with multiple lines in the block.

```
>>> if (seq.startswith("C")):
...     print "Starts with C"
...     print "All right by me!"
...
Starts with C
All right by me!
```

When the `if` statement is true, all of the lines in the block are executed (in this case two lines in the block).

Multiline blocks

- What happens if you don't use the same number of spaces to indent the block?

```
>>> if (seq.startswith("C")):
...    print "Starts with C"
...     print "All right by me!"
  File "<stdin>", line 4
    print "All right by me!"
    ^
SyntaxError: invalid syntax
```

This is why I prefer to use a single **<tab>** character – it is always exactly correct.

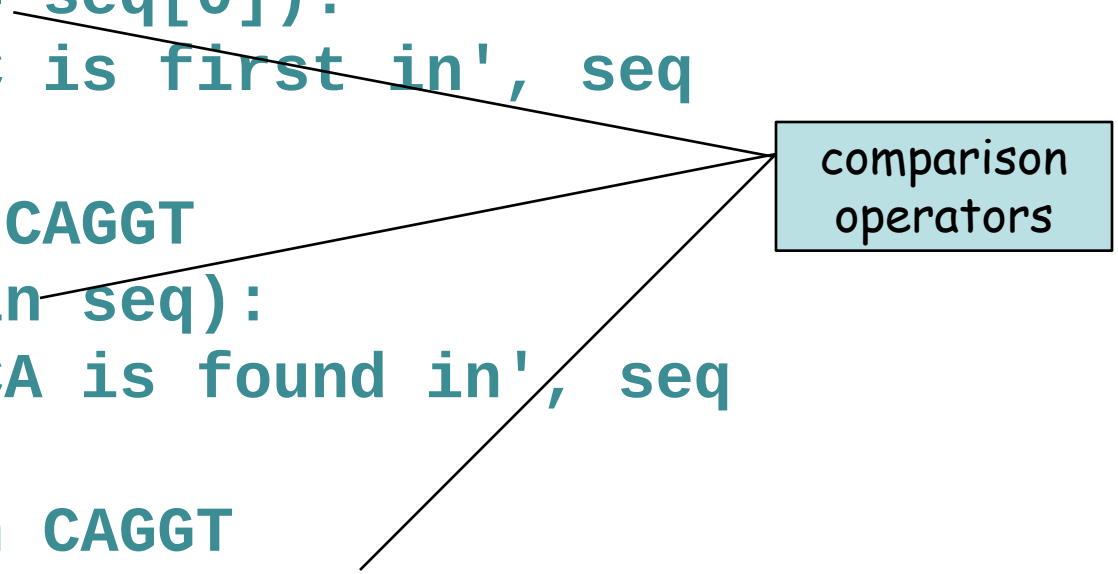
Comparison and logic operators

- Boolean: `and, or, not`
- Numeric: `< , > , ==, !=, >=, <=`
- String: `in, not in`

```
<     is less than
>     is greater than
==    is equal to
!=    is NOT equal to
<=    is less than or equal to
>=    is greater than or equal to
```

Examples

```
seq = 'CAGGT'
>>> if ('C' == seq[0]):
...     print 'C is first in', seq
...
C is first in CAGGT
>>> if ('CA' in seq):
...     print 'CA is found in', seq
...
CA is found in CAGGT
>>> if (('CA' in seq) and ('CG' in seq)):
...     print "Both there!"
...
>>>
```



Beware!

= versus ==

- Single equal assigns a value.
- Double equal tests for equality.

Combining tests

```
x = 1
y = 2
z = 3
if ((x < y) and (y != z)):
      do something
if ((x > y) or (y == z)):
      do something else
```

Evaluation starts with the innermost parentheses and works out. When there are multiple parentheses at the same level, evaluation starts at the left and moves right. The statements can be arbitrarily complex.

```
if (((x <= y) and (x < z)) or ((x == y) and not (x == z)))
```

`if-else` statements

```
if <test1>:
   <statement>
else:
   <statement>
```

- The `else` block executes only if `<test1>` is false.

```
>>> if (seq.startswith('T')):
...    print 'T start'
... else:
...    print 'starts with', seq[0]
...
starts with C
>>>
```

if-elif-else

```
if <test1>:
   <block1>
elif <test2>:
   <block2>
else:
   <block3>
```

Can be read this way:

if test1 is true then run block1, else if test2 is true run block2, else run block3

- `elif` block executes if `<test1>` is false and then performs a second `<test2>`
- Only one of the blocks is executed.

Example

```
>>> base = 'C'
>>> if (base == 'A'):
...   print "adenine"
... elif (base == 'C'):
...   print "cytosine"
... elif (base == 'G'):
...   print "guanine"
... elif (base == 'T'):
...   print "thymine"
... else:
...   print "Invalid base!"
...
cytosine
```

```
<file> = open(<filename>, 'r'|'w'|'a')
<string> = <file>.read()
<string> = <file>.readline()
<string list> = <file>.readlines()
<file>.write(<string>)
<file>.close()
```

- Boolean: `and, or, not`
- Numeric: `< , > , ==, !=, >=, <=`
- String: `in, not in`

```
if <test1>:
  <statement(s)>
elif <test2>:
  <statement(s)>
else:
  <statement(s)>
```

Sample problem #1

- Write a program **`read-first-line.py`** that takes a file name from the command line, opens the file, reads the first line, and prints the line to the screen.

```
> python read-first-line.py hello.txt
Hello, world!

>
```

Solution #1

```
import sys
filename = sys.argv[1]
myFile = open(filename, "r")
firstLine = myFile.readline()
myFile.close()
print firstLine
```

Sample problem #2

- Modify your program to print the first line without an extra new line.

```
> python read-first-line.py hello.txt
Hello, world!
>
```

Solution #2

```
import sys
filename = sys.argv[1]
myFile = open(filename, "r")
firstLine = myFile.readline()
firstLine = firstLine[:-1]
myFile.close()
print firstLine
```

remove last character

(or use `firstLine.strip()`, which removes all the whitespace from both ends)

Sample problem #3

- Write a program **`math-two-numbers.py`** that reads one integer from the first line of one file and a second integer from the first line of a second file. If the first number is smaller, then print their sum, otherwise print their multiplication. Indicate the entire operation in your output.

```
> add-two-numbers.py four.txt nine.txt
4 + 9 = 13
>
```

Solution #3

```
import sys
fileOne = open(sys.argv[1], "r")
valOne = int(fileOne.readline()[:-1])
fileOne.close()
fileTwo = open(sys.argv[2], "r")
valTwo = int(fileTwo.readline()[:-1])
fileTwo.close()
if valOne < valTwo:
  print valOne, "+", valTwo, "=", valOne + valTwo
else:
  print valOne, "*", valTwo, "=", valOne * valTwo
```

Here's a version that is more robust because it doesn't matter whether the file lines have white space or a newline:

```
import sys
fileOne = open(sys.argv[1], "r")
valOne = int(fileOne.readline().strip())
fileOne.close()
fileTwo = open(sys.argv[2], "r")
valTwo = int(fileTwo.readline().strip())
fileTwo.close()
if valOne < valTwo:
  print valOne, "+", valTwo, "=", valOne + valTwo
else:
  print valOne, "*", valTwo, "=", valOne * valTwo
```

Sample problem #4 (review)

- Write a program **`find-base.py`** that takes as input a DNA sequence and a nucleotide. The program should print the number of times the nucleotide occurs in the sequence, or a message saying it's not there.

```
> python find-base.py A GTAGCTA
A occurs twice
> python find-base.py A GTGCT
A does not occur at all
```

Hint: `S.find('G')` returns -1 if it can't find the requested string.

Solution #4

```
import sys
base = sys.argv[1]
sequence = sys.argv[2]
position = sequence.find(base)
if (position == -1):
  print base, "does not occur at all"
else:
  n = sequence.count(base)
  print base, "occurs " + n + "times"
```

Challenge problems

Write a program that reads a sequence file (seq1) and a sequence (seq2) based on command line arguments and makes output to the screen that either:

1) says seq2 is entirely missing from seq1, or
2) counts the number of times seq2 appears in seq1, or
3) warns you that seq2 is longer than seq1

```
> python challenge.py seqfile.txt GATC
> GATC is absent
(or
> GATC is present 7 times)
(or
> GATC is longer than the sequence in seqfile.txt)
```

Make sure you can handle multiline sequence files.

Do the same thing but output a list of all the positions where seq2 appears in seq1 (tricky with your current knowledge).

TIP – **`file.read()`** includes the newline characters from a multiline file

Challenge problems

Write a program that is approximately equivalent to the find and replace function of word processors. Take as arguments: 1) a string to find, 2) a string to replace with, 3) input file name, 4) output file name. You don't really need this, but try to incorporate a conditional test.

```
> f_and_r.py Watson Crick infile.txt outfile.txt
```

(should replace all appearances of "Watson" in the input file with "Crick".)

Reading

- First parts of chapters 5 and 14 from *Think Python* by Downey