File input and output

Genome 373

Genomic Informatics

Elhanan Borenstein

Reading the whole file

```
>>> 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)

File input and output - Summary

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

Conditionals

`if-elif-else`

Genome 373

Genomic Informatics

Elhanan Borenstein

The `if` statement

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

Starts with C
>>>
```

The `if` statement

- But … what if you want to perform more than one command if the condition is true?
- A **block** is a group of lines of code that belong together.

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

- 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
```

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).

Comparison 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)))
```

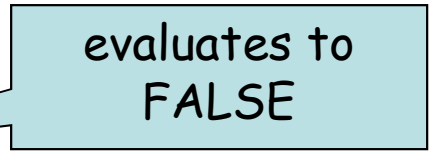
`if-else` statements

```
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
```

Hints on variable names

- Pick names that are descriptive
- Change a name if you decide there's a better choice
- Give names to intermediate values for clarity
- Use the name to describe the type of object
- Very locally used names can be short and arbitrary

```
listOfFileLines = myFile.readlines()
seqString = "GATCTCTATCT"
myDPMatrix = [[0,0,0],[0,0,0],[0,0,0]]

intSum = 0
for i in range(5000):
    intSum = intSum + listOfInts[i]
(more code)
```

Comment your code!

- Any place a # sign appears, the rest of the line is a comment (ignored by program).

- Blank lines are also ignored – use them to visually group code.

```
import sys
query = sys.argv[1]
myFile = open(sys.argv[2], "r")
lineList = myFile.readlines()    # put all the lines from a file into a list

# now process each file line to remove the \n character, then
# search the line for query and record each result in a list of ints
intList = []
for line in lineList:
    position = line.find(query)
    intList.append(position)
etc.
```

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.