Leiden University
Medical Center

Python Programming

Standard library, reading and writing files

Mihai Lefter

Outline

Working with modules

- A module allows you to share code in the form of libraries.
- You've seen one example: the sys module in the standard library.
- There are many other modules in the standard library, as we'll see soon.

What modules look like

- Any Python script can in principle be imported as a module.
- We can import whenever we can write a valid Python statement, in a script or in an interpreter session.
- If a script is called `script.py`, then we use `import script`.
- This gives us access to the objects defined in `script.py` by prefixing them with `script` and a dot.
- Keep in mind that this is not the only way to import Python modules.
- Refer to the Python documentation to find out more ways to do imports.

Using seq_toolbox.py as a module

Open an interpreter and try importing your module:

IPython

```
In [1]: import seq_toolbox
```

Does this work? Why?

Using seq_toolbox.py as a module

IPython

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-1-ccf54d4de53d> in <module>()
----> 1 import seq_toolbox

~/.../seq_toolbox.py in <module>()
     35
     36
---> 37 input_seq = sys.argv[1]
     38 print("The sequence '{}' has a %GC of {:.2f}".format(
     39           input_seq, calc_gc_percent(input_seq)))

IndexError: list index out of range
```

Improving our script for importing

- During a module import, Python executes all the statements inside the module.
- To make our script work as a module (in the intended way), we need to add a check whether the module is imported or not:

```
if __name__ == '__main__':
    input_seq = sys.argv[1]
    print "The sequence '{}' has %GC of {:.2f}".format(
              input_seq, calc_gc_percent(input_seq))
```

- Now try importing the module again.
  - What happens? Can you still use the module as a script?

Using modules

- When a module is imported, we can access the objects defined in it:

IPython

```
In [2]: import seq_toolbox
In [3]: seq_toolbox.calc_gc_percent
Out[3]: <function seq_toolbox.calc_gc_percent>
```

Using modules

- When a module is imported, we can access the objects defined in it:

IPython

```
In [6]: import seq_toolbox
In [7]: seq_toolbox.calc_gc_percent
Out[7]: <function seq_toolbox.calc_gc_percent>
```

- By the way, remember we added docstring to the calc_gc_percent function?
- After importing our module, we can read up on how to use the function in its docstring:

IPython

```
In [8]: seq_toolbox.calc_gc_percent?
In [9]: seq_toolbox.calc_gc_percent('ACTG')
Out[9]: 50.0
```

Using modules

- We can also expose an object inside the module directly into our current namespace using the `from ... import ...` statement:

IPython

```
In [10]: from seq_toolbox import calc_gc_percent
In [11]: calc_gc_percent('AAAG')
Out[11]: 25.0
```

(A simple guide on) How modules are discovered

- In our case, Python imports by checking whether the module exists in the current directory.
- This is not the only place Python looks, however.
- A complete list of paths where Python looks for modules is available via the sys module as sys.path. It is composed of (in order):
  - The current directory.
  - The PYTHONPATH environment variable.
  - Installation-dependent defaults.

os module

- provides a portable way of using various operating system-specific functionality.
- It is a large module, but the one of the most frequently used bits is the file-related functions.

IPython

```
In [12]: import os
In [13]: os.getcwd()    # Get current directory.
Out[13]: '/home/student/projects/programming-course'

In [14]: my_filename = 'input.fastq'
In [15]: os.path.splitext(my_filename)    # Split the extension and filename.
Out[15]: ('input', '.fastq')

In [16]: os.path.isdir('/home')    # Checks whether '/home' is a directory.
Out[16]: True
```

math module

- Useful math-related functions can be found here.
- Other more comprehensive modules exist (numpy, your lesson tomorrow), but nevertheless math is still useful.

IPython

```
In [17]: import math
In [18]: math.log(10)    # Natural log of 10.
Out[18]: 2.302585092994046

In [19]: math.log(100, 10)    # Log base 10 of 100.
Out[19]: 2.0

In [20]: math.sqrt(2)    # Square root of 2.
Out[20]: 1.4142135623730951
```

random module

- The random module contains useful functions for generating pseudo-random numbers.

IPython

```
In [21]: import random
In [22]: math.log(10)    # Natural log of 10.
Out[22]: 0.9562916447281146

In [23]: random.randint(2, 17)    # Random integer between 2 and 17, inclusive.
Out[23]: 13

In [24]: # Random choice of any items in the given list.
         random.choice(['apple', 'banana', 'grape', 'kiwi', 'orange'])
Out[24]: 'grape'
```

argparse module

- Using sys.argv is neat for small scripts, but as our script gets larger and more complex, we want to be able to handle complex arguments too.
- The argparse module has handy functionalities for creating command-line scripts.

argparse module

- Open your script/module in a text editor and replace `import sys` with `import argparse`.
- Remove all lines / blocks referencing `sys.argv`.
- Change the `if __name__ == '__main__'` block to be the following:

```
if __name__ == '__main__':
    # Create our argument parser object.
    parser = argparse.ArgumentParser()
    # Add the expected argument.
    parser.add_argument('input_seq', type=str,
                        help="Input sequence")
    # Do the actual parsing.
    args = parser.parse_args()
    # And show the output.
    print "The sequence '{}' has %GC of {:.2f}".format(
              args.input_seq,
              calc_gc_percent(args.input_seq))
```

Working with text files

- Opening files for reading or writing is done using the `open` function.
- It is commonly used with two arguments, `name` and `mode`:
  - `name` is the name of the file to open.
  - `mode` specifies how the file should be handled.
- These are some of the common file modes:
  - r: open file for reading (default).
  - w: open file for writing.
  - a: open file for appending content.

Reading files

- Let's go through some ways of reading from a file.

IPython

```
In [25]: fh = open('data/short_file.txt')
```

- fh is a file handle object which we can use to retrieve the file contents.
- One simple way would be to read the whole file contents:

IPython

```
In [26]: fh.read()
Out[26]: 'this short file has two lines it is used in the example code'
```

Reading files

- Executing fh.read() a second time gives an empty string. This is because we have "walked" through the file to its end.

IPython

```
In [27]: fh.read()
Out[27]: ''
```

- We can reset the handle to the beginning of the file again using the `seek()` function.
- Here, we use 0 as the argument since we want to move the handle to position 0 (beginning of the file):

IPython

```
In [28]: fh.seek(0)
```

Reading files

- In practice, reading the whole file into memory is not always a good idea.
- It is practical for small files, but not if our file is big (e.g., bigger than our memory).
- In this case, the alternative is to use the `readline()` function.

IPython

```
In [29]: fh.readline()
Out[29]: 'this short file has two lines'

In [30]: fh.readline()
Out[30]: 'it is used in the example code'

In [31]: fh.readline()
Out[31]: ''
```

Reading files

- More common in Python is to use the for loop with the file handle itself.
- Python will automatically iterate over each line.

```
for line in fh:
    print line
```

- Now that we're done with the file handle, we can call the `close()` method to free up any system resources still being used to keep the file open.
- After we closed the file, we can not use the file object anymore.

Writing files

- When writing files, we supply the w file mode explicitely:

IPython

```
In [32]: fw = open('data/my_file.txt', 'w')
```

- fw is a file handle similar to the fh that we've seen previously.
- It is used only for writing and not reading, however.

Writing files

- To write to the file, we use its `write()` method.
- Remember that Python does not add newline characters here
- (as opposed to when you use the print statement), so to move to a new
- line we have to add \n ourselves.

IPython

```
In [33]: fw.write('This is my first line ')
In [34]: fw.write('Still on my first line\n')
In [35]: fw.write('Now on my second line')
```

Writing files

- As with the r mode, we can close the handle when we're done with it. The file can then be reopened with the r mode and we can check its contents.

IPython

```
In [36]: fw.close()
```

Be cautious when using file handles

- When reading / writing files, we are interacting with external resources that may or may not behave as expected.
- For example:
  - We don't always have permission to read / write a file.
  - The file itself may not exist.
  - We have a completely wrong idea of what's in the file.

```
try:
    f = open('data/short_file.txt')
    for line in f:
        print int(line)
except ValueError:
    print 'Seems there was a line we could not handle'
finally:
    f.close()
    print 'We closed our file handle'
```

Be cautious when using file handles

- This option is highly recommended:

```
with open("welcome.txt") as f: # Use file to refer to the file object
    for line in f:
        #do something with data
```

Improving our script to allow input from a file

- The script should accept as its argument a path to a file containing sequences.
- It will then compute the GC percentage for each sequence in this file.
- There are at least two things we need to do:
  - Change the argument parser so that it deals with a new execution mode.
  - Add some statements to read from a file.

Leiden University Medical Center

Acknowledgements

Martijn Vermaat
Jeroen Laros
Jonathan Vis