Programming Principles in Python (CSCI 503)

Scripts, Modules, and Packages

Dr. David Koop

Reading Files

- Use the `open()` method to open a file for reading
  - `f = open('huck-finn.txt')`
- Usually, add an `'r'` as the second parameter to indicate read (default)
- Can iterate through the file (think of the file as a collection of lines):
  -
    ```
    f = open('huck-finn.txt', 'r')
    for line in f:
        if 'Huckleberry' in line:
            print(line.strip())
    ```
- Using `line.strip()` because the read includes the newline, and print writes a newline so we would have double-spaced text
- Closing the file: `f.close()`

Parsing Files

- Dealing with different formats, determining more meaningful data from files
- txt: text file
- csv: comma-separated values
- json: JavaScript object notation
- Jupyter also has viewers for these formats
- Look to use libraries to help possible
  - `import json`
  - `import csv`
  - `import pandas`
- Python also has pickle, but not used much anymore

Writing Files: Use with statement

- `outf = open("mydata.txt", "w")`
- Methods for writing to a file:
  - `print(<expressions>, file= outf)`
  - `outf.write(<string>)`
  - `outf.writelines(<list of strings>)`
- Make sure to **close** the file at the end: `outf.close()`
- With statement does "enter" and "exit": don't need to call `outf.close()`

  -
    ```
    with open('output.txt', 'w') as outf:
        for k, v in counts.items():
            outf.write(k + ': ' + v + '\n')
    ```

JavaScript Object Notation (JSON)

- A format for web data
- Looks very similar to python dictionaries and lists
- Example:

```
- {"name": "Wes",
    "places_lived": ["United States", "Spain", "Germany"],
    "pet": null,
    "siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"},
                 {"name": "Katie", "age": 33, "pet": "Cisco"}] }
```

- Only contains literals (no variables) but allows null
- Values: strings, arrays, dictionaries, numbers, booleans, or null
  - Dictionary keys must be strings
  - Quotation marks help differentiate string or numeric values

Reading JSON data

- Python has a built-in `json` module

  -
    ```
    with open('example.json') as f:
        data = json.load(f)
    ```

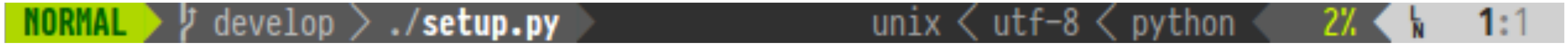
  -
    ```
    with open('example-out.json', 'w') as f:
        json.dump(data, f)
    ```
- Can also load/dump to strings:
  - `json.loads, json.dumps`

Assignment 4

- Assignment covers strings and files
- Reading & writing data to files
- Dealing with encodings and string formatting

Command-Line Interfaces

Command Line Interfaces (CLIs)

- Prompt:
  - $
  - 

```
NORMAL  develop  ./setup.py                    unix  utf-8  python  2%   1:1
```

- Commands
  - `$ cat <filename>`
  - `$ git init`
- Arguments/Flags: (options)
  - `$ python -h`
  - `$ head -n 5 <filename>`
  - `$ git branch fix-parsing-bug`

Command Line Interfaces

- Many command-line tools work with stdin and stdout
  - `cat test.txt # writes test.txt's contents to stdout`
  - `cat # reads from stdin and writes back to stdout`
  - `cat > test.txt # writes user's text to test.txt`
- Redirecting input and output:
  - `<` use input from a file descriptor for stdin
  - `>` writes output on stdout to another file descriptor
  - `|` connects stdout of one command to stdin of another command
  - `cat < test.txt | cat > test-out.txt`

Python and CLIs

- Python can be used as a CLI program
  - Interactive mode: start the REPL
    - `$ python`
  - Non-interactive mode:
    - `$ python -c <command>`: Execute a command
    - `$ python -m <module>|<package>`: Execute a module
- Python can be used to create CLI programs
  - Scripts: `python my_script.py`
  - True command-line tools: `./command-written-in-python`

Interactive Python in the Shell

- Starting Python from the shell

  -
    ```
    $ python
    ```
- `>>>` is the Python interactive prompt

  -
    ```
    >>> print("Hello, world")
    Hello, world
    ```

  -
    ```
    >>> print("2+3=", 2+3)
    2+3= 5
    ```
- This is a REPL (Read, Evaluate, Print, Loop)

Interactive Python in the Shell

- `...` is the continuation prompt
-
```
>>> for i in range(5):
...     print(i)
...
```
- Still need to indent appropriately!
- Empty line indicates the suite (block) is finished
- This isn't always the easiest environment to edit in

Ending an Interactive Session

- `Ctrl-D` ends the input stream
  - Just as in other Unix programs
- Another way to get normal termination
  - `>>> quit()`
- `Ctrl-C` interrupts operation
  - Just as in other Unix programs

Interactive Problems

- But standard interactive Python doesn't save programs!
- IPython does have some magic commands to help
  - `%history`: prints code
  - `%save`: saves a file with code
  - These are most useful outside the notebook, but you can type them in the notebook, too
- However, it is nice to be able to edit code in files and run it, too

Module Files

- A **module file** is a text file with the `.py` extension, usually name.py
- Python source on Unix is expected to be in UTF-8
- Can use any text editor to write or edit…
- …but an editor that understands Python's spacing and indentation helps!
- Contents looks basically the same as what you would write in the cell(s) of a notebook
- There are also ways to write code in multiple files organized as a package, will cover this later

Scripts, Programs, and Libraries

- Often, interpreted ~ scripts and compiled code ~ programs/libraries
  - Python does compile **bytecode** for modules that are imported
- Modifying this usual definition a bit
  - Script: a one-off block of code meant to be run by itself, users **edit the code** if they wish to make changes
  - Program: code meant to be used in different situations, with **parameters** and **flags** to allow users to customize execution without editing the code
  - Library: code meant to be called from other scripts/programs
- In Python, can't always tell from the name what's expected, code can be both a library and a program

Program Execution

- Direct Unix execution of a program
  - Add the hashbang (`#!`) line as the **first line**, two approaches
  - `#!/usr/bin/python`
  - `#!/usr/bin/env python`
  - Sometimes specify `python3` to make sure we're running Python 3
  - File must be flagged as executable (`chmod a+x`) and have line endings
  - Then you can say: `$ ./filename.py arg1 ...`
- Executing the Python compiler/interpreter
  - `$ python filename.py arg1 ...`
- Same results either way

Writing CLI Programs

- Command Line Interface Guidelines
  - Accept flags/arguments
  - Human-readable output
  - Allow non-interactive use even if program can also be interactive
  - Add help/usage statements
  - Consider subcommand use for complex tools
  - Use simple, memorable name
  - …

Accepting Command-Line Parameters

- Parameters are received as a list of strings entitled `sys.argv`
- Need to `import sys` first
- `sys.argv[0]` is the name of the program as executed
  - Executing as `./hw01.py` or `hw01.py` will be passed as different strings
- `sys.argv[n]` is the nth argument
- `sys.executable` is the python executable being run

Using Parameters

- Test `len(sys.argv)` to make sure the correct number of parameters were passed
- Everything in `sys.argv` is a **string**, often need to cast arguments:
  - `my_value = int(sys.argv[1])`
- Guard against bad inputs
  - Test before using or deal with errors
  - Use `isnumeric` or catch the exception
  - Printing help/usage statement on error can help users

The main function

- Convention: create a function named `main()`
- Customary, but not required

  -
    ```
    def main():
        print("Running the main function")
    ```
- Nothing happens in a script with this definition!

The main function

- Convention: create a function named `main()`
- Customary, but not required

  -
    ```
    def main():
        print("Running the main function")
    ```
- Nothing happens in a script with this definition!
- Need to call the function in our script!

-
  ```
  def main():
      print("Running the main function")
  main() # now, we're calling main
  ```

Using code as a module, too

- When we want to start a program once it's loaded, we include the line `main()` at the bottom of the code.
- Since Python evaluates the lines of the program during the import process, our current programs also run when they are imported into an interactive Python session or into another Python program.
- `import my_code # prints "Running the main function"`
- Generally, when we import a module, we **don't want it to execute**.

Knowing when the file is being used as a script

- Whenever a module is imported, Python creates a special variable in the module called `__name__` whose value is the name of the imported module.
- Example:

```
>>> import math
>>> math.__name__
'math'
```

- When Python code is run directly and not imported, the value of `__name__` is `'__main__'`.
- We can change the final lines of our programs to:

```
if __name__ == '__main__':
    main()
```

Modules and Packages

- Python allows you to import code from other files, even your own
- A **module** is a collection of definitions
- A **package** is an organized collection of modules
- Modules can be
  - a separate python file
  - a separate C library that is written to be used with Python
  - a built-in module contained in the interpreter
  - a module installed by the user (via conda or pip)
- All types use the same import syntax

[RealPython]

What is the purpose of having modules or packages?

What is the purpose of having modules or packages?

- Code reuse: makes life easier because others have written solutions to various problems
- Generally forces an organization of code that works together
- Standardizes interfaces; easier maintenance
- Encourages robustness, testing code

- This does take time so don't always create a module or package
  - If you're going to use a method once, it's not worth putting it in a module
  - If you're using the same methods over and over in (especially in different projects), a module or package makes sense

Module Contents

- Modules can contain
  - functions
  - variable (constant) declarations
  - import statements
  - class definitions
  - any other code
- Note that variable values can be changed in the module's namespace, but this doesn't affect other Python sessions.

Importing modules

- `import <module>`
- `import <module> as <another-identifier>`
- `from <module> import <identifer-list>`
- `from <module> import <identifer> as <another-identifier>, …`

- `import` imports from the top, `from … import` imports "inner" names
- Need to use the qualified names when using import (`foo.bar.mymethod`)
- as clause **renames** the imported name

Using an imported module

- Import module, and call functions with **fully qualified** name

  -
    ```
    import math
    math.log10(100)
    math.sqrt(196)
    ```
- Import module into current namespace and use **unqualified** name

  -
    ```
    from math import log10, sqrt
    log10(100)
    sqrt(196)
    ```

Using code as a module, too

-
  ```
  def main():
      print("Running the main function")
  main() # now, we're calling main
  ```
- Generally, when we import a module, we **don't want it to execute code.**
- `import my_code # prints "Running the main function"`
- Whenever a module is imported, Python creates a special variable in the module called `__name__` whose value is the name of the imported module.
- We can change the final lines of our programs to:

  -
    ```
    if __name__ == '__main__':
        main()
    ```
- `main()` only runs when the file is run as a script!