Programming Principles in Python (CSCI 503/490)

Modules and Packages

Dr. David Koop

Command Line Interfaces (CLIs)

- Prompt:
  - $
  - 

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

Consoles, Terminals, and Shells in Jupyter

- Terminal mirrors the terminal in Linux terminals, Terminal.app (macOS), and PowerShell (Windows)
  - Runs more than just python
- Console provides IPython interface
  - Easier multi-line editing
  - Reference past outputs directly, other bells and whistles
- Shell will run in the Terminal app
- Can also use shell commands in the notebook using `!`
  - `!cat <filename>`
  - `!head -n 10 <filename>`

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)

Module Files

- A **module file** is a text file with the `.py` extension, usually name.py
- Python source on Unix is 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

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

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 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!

Assignment 4

- Due Thursday
- UDSA Food Price Data
- Reading & Writing Files
- Iterators
- Numeric Aggregation
- String Formatting
- CSCI 503 students compute and output two additional fields

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

How does import work?

- When a module/package is imported, Python
  - Searches for the module/package
    - Sometimes this is internal
    - Otherwise, there are directory paths (environment variable `PYTHONPATH`) that python searches (accessible via `sys.path`)
  - Loads it
    - This will run the code in specified module (or `__init__.py` for a package)
  - Binds the loaded names to a namespace

[RealPython]

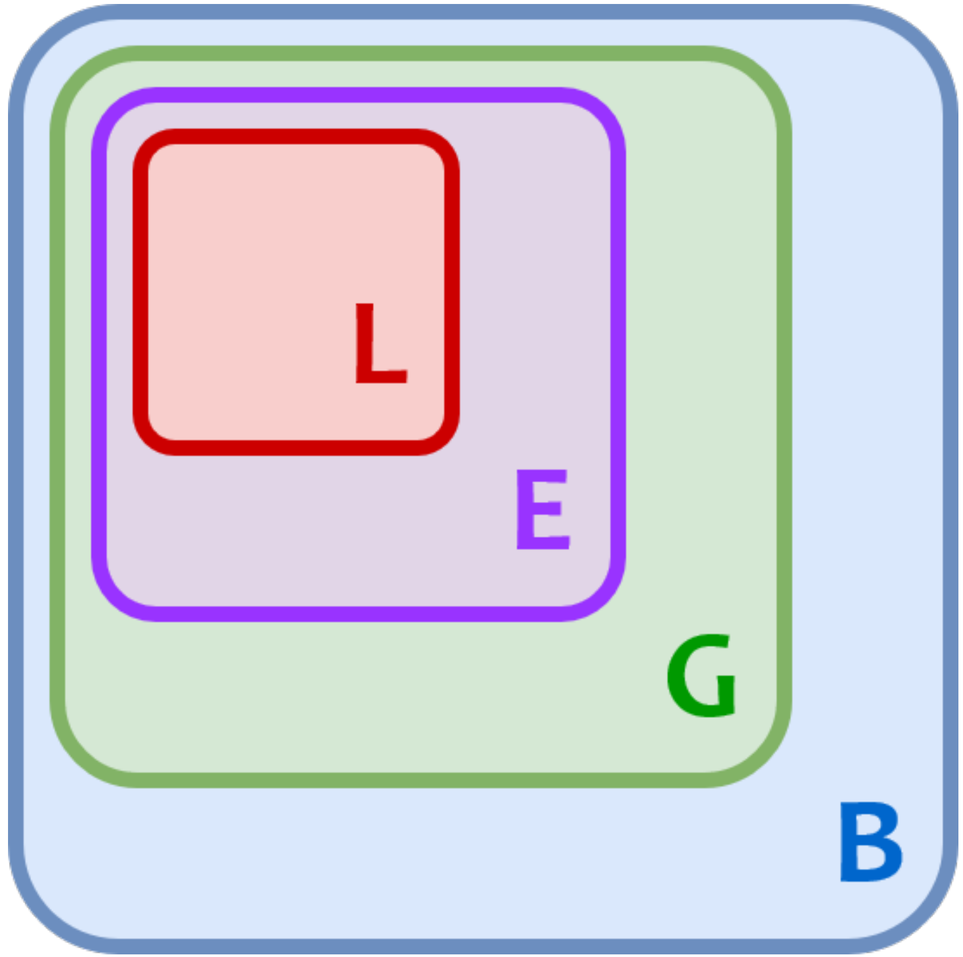
Namespaces

- An import defines a separate **namespace** while from…import adds names to the current namespace
- Four levels of namespace
  - builtins: names exposed internally in python
  - global: names defined at the outermost level (wrt functions)
  - local: names defined in the current function
  - enclosing: names defined in the outer function (when nesting functions)
-
```
def foo():
    a = 12
    def bar():
        print("This is a:", a)
```

`a` is in the **enclosing** namespace of `bar`

Namespaces

- Namespace is basically a dictionary with names and their values
- Accessing namespaces
  - `__builtins__`, `globals()`, `locals()`
- Examine contents of a namespace: `dir(<namespace>)`
- Python checks for a name in the sequence: local, enclosing, global, builtins
- To access names in outer scopes, use `global` (global) and `nonlocal` (enclosing) declarations



[RealPython]

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)
    ```

Wildcard imports

- Wildcard imports import all names (non-private) in the module
- What about
  - `from math import *`
- Avoid this!
  - Unclear which names are available!
  - Confuses someone reading your code
  - Think about packages that define the same names!
- Allowed if republishing internal interface (e.g. in a package, you're exposing functions defined in different modules

Import Guidelines (from PEP 8)

- Imports should be on separate lines
  - ~~`import sys, os`~~
  -
```
import sys
import os
```
- When importing multiple names from the same package, do use same line
  - `from subprocess import Popen, PIPE`
- Imports should be at the **top** of the file (order: standard, third-party, local)
- Avoid wildcard imports in most cases

Conditional or Dynamic Imports

- Best practice is to put all imports at the beginning of the py file
- Sometimes, a conditional import is required
  -
```
if sys.version_info >= [3,7]:
    OrderedDict = dict
else:
    from collections import OrderedDict
```
- Can also dynamically load a module
  - `import importlib`
  - `importlib.import_module("collections")`
  - The `__import__` method can also be used

Absolute & Relative Imports

- Fully qualified names
  - `import foo.bar.submodule`
- Relative names
  - `import .submodule`
- Absolute imports recommended but relative imports acceptable

Import Abbreviation Conventions

- Some libraries and users have developed particular conventions
- `import numpy as np`
- `import pandas as pd`
- `import matplotlib.pyplot as plt`
- This can lead to problems:
  - `sympy` and `scipy` were both abbreviated `sp` for a while…

Reloading a Module?

- If you re-import a module, what happens?
  - `import my_module`
    `my_module.SECRET_NUMBER # 42`
  - Change the definition of `SECRET_NUMBER` to `14`
  - `import my_module`
    `my_module.SECRET_NUMBER # Still 42!`
- Modules are **cached** so they are not reloaded on each import call
- Can reload a module via `importlib.reload(<module>)`
- Be careful because **dependencies** will persist! (Order matters)

Python Packages

- A package is basically a collection of modules in a directory subtree
- Structures a module namespace by allowing dotted names
- Example:

```
- test_pkg/
        __init__.py
        foo.py
        bar.py
        baz/
            fun.py
```

- For packages that are to be executed as scripts, `__main__.py` can also be added

What's __init__.py used for?

- Used to be required to identify a Python package (< 3.3)
- Now, only required if a package (or sub-package) needs to run some initialization when it is loaded
- Can be used to specify metadata
- Can be used to import submodule to make available without further import
  - `from . import <submodule>`
- Can be used to specify which names exposed on import
  - underscore names (`_internal_function`) not exposed by default
  - `__all__` list can further restrict, sets up an "interface" (applies to wildcard)

What is __main__.py used for?

- Remember for a module, when it is run as the main script, its `__name__` is `__main__`
- Similar idea for packages
- Used as the entry point of a package when the package is being run (e.g. via `python -m`)
  - `python -m test_pkg` runs the code in `__main__.py` of the package