Programming Principles in Python (CSCI 503/490)

Object-Oriented Programming

Dr. David Koop

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

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?

- 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

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!

Assignment 4

- Books in German
- Reading & Writing Files
- Iterators
- Converting certain values
- String Formatting
- CSCI 503 students compute and output statistics to compare authors

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]

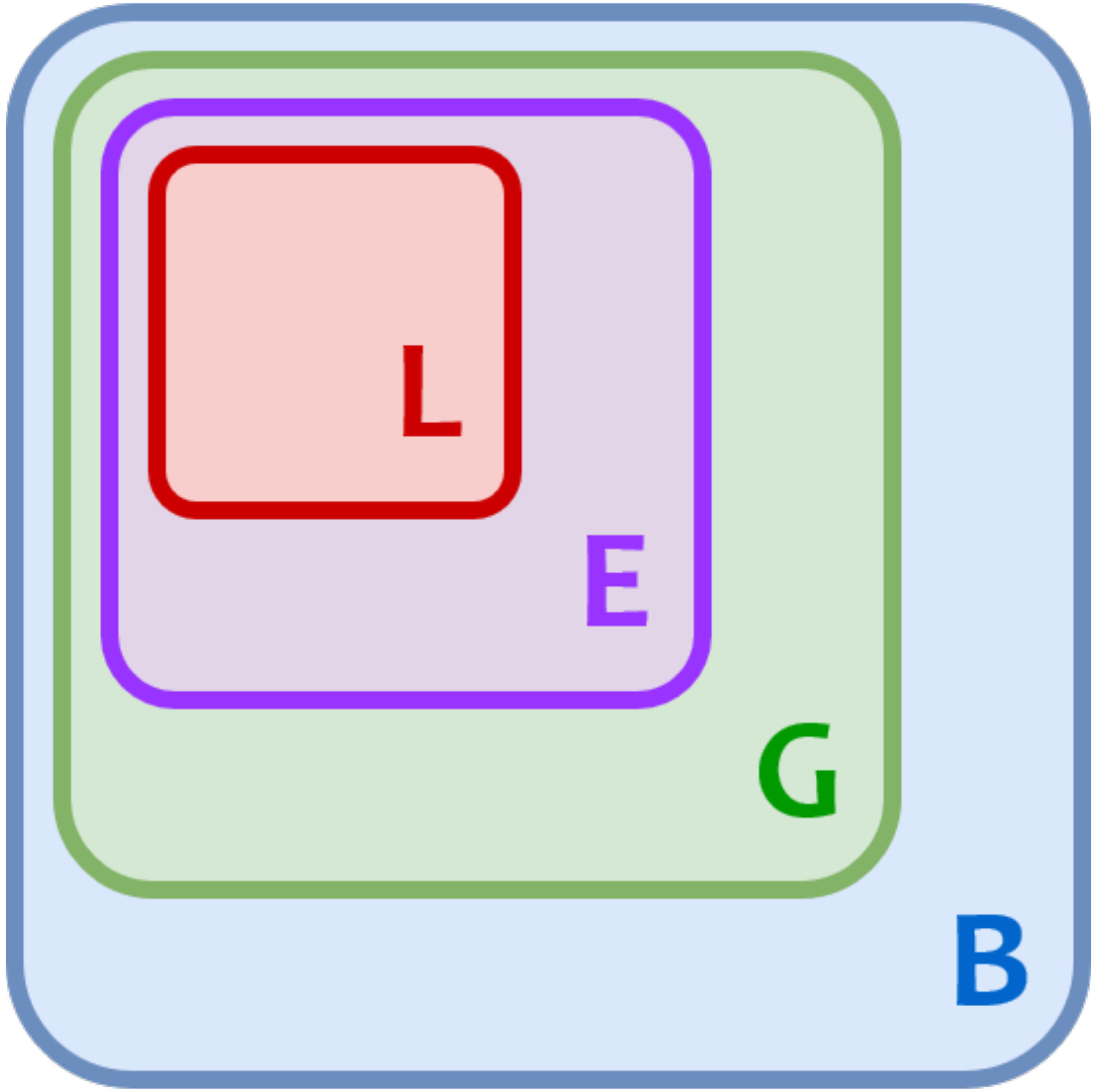
Namespaces

- 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]

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

Example

Finding Packages

- Python Package Index (PyPI) is the standard repository (https://pypi.org) and pip (pip installs packages) is the official python package installer
  - Types of distribution: source (sdist) and wheels (binaries)
  - Each package can specify dependencies
  - Creating a PyPI package requires adding some metadata
- Anaconda is a package index, conda is a package manager
  - conda is language-agnostic (not only Python)
  - solves dependencies
  - conda deals with non-Python dependencies
  - has different channels: default, conda-forge (community-led)

Installing Packages

- `pip install <package-name>`
- `conda install <package-name>`
- In Jupyter use:
  - `%pip install <package-name>`
  - `%conda install <package-name>`
- Arguments can be multiple packages
- Be careful! Security exploits using package installation and dependencies (e.g. Alex Birsan)

Environments

- Both pip and conda support environments
  - venv
  - conda env
- Idea is that you can create different environments for different work
  - environment for cs503
  - environment for research
  - environment for each project

Object-Oriented Programming

Object-Oriented Programming Concepts

- ?

Object-Oriented Programming Concepts

- Abstraction: simplify, hide implementation details, don't repeat yourself
- Encapsulation: represent an entity fully, keep attributes and methods together
- Inheritance: reuse (don't reinvent the wheel), specialization
- Polymorphism: methods are handled by a single interface with different implementations (overriding)

Object-Oriented Programming Concepts

- **Abstraction**: simplify, hide implementation details, don't repeat yourself
- **Encapsulation**: represent an entity fully, keep attributes and methods together
- Inheritance: reuse (don't reinvent the wheel), specialization
- Polymorphism: methods are handled by a single interface with different implementations (overriding)

Vehicle Example

- Suppose we are implementing a city simulation, and want to model vehicles driving on the road
- How do we represent a vehicle?
  - Information (attributes)
  - Methods (actions)

Vehicle Example

- Suppose we are implementing a city simulation, and want to model vehicles driving on the road
- How do we represent a vehicle?
  - Information (attributes): make, model, year, color, num_doors, engine_type, mileage, acceleration, top_speed, braking_speed
  - Methods (actions): compute_estimated_value(), drive(num_seconds, acceleration), turn_left(), turn_right(), change_lane(dir), brake(), check_collision(other_vehicle)

Other Entities

- Road, Person, Building, ParkingLot
- Some of these interact with a Vehicle, some don't
- We want to store information associated with entities in a structured way
  - Building probably won't store anything about cars
  - Road should not store each car's make/model
  - …but we may have an association where a Road object keeps track of the cars currently driving on it

Object-Oriented Design

- There is a lot more than can be said about how to best define classes and the relationship between different classes
- It's not easy to do this well!
- Software Engineering
- Entity Relationship (ER) Diagrams
- Difference between Object-Oriented Model and ER Model

Class vs. Instance

- A **class** is a blueprint for creating instances
  - e.g. Vehicle
- An **instance** is an single object created from a class
  - e.g. 2000 Red Toyota Camry
  - Each object has its own attributes
  - Instance methods produce results unique to each particular instance

Classes and Instances in Python

- Class Definition:
  - ```
    class Vehicle:
        def __init__(self, make, model, year, color):
            self.make = make
            self.model = model
            self.year = year
            self.color = color

        def age(self):
            return 2021 - self.year
    ```
- Instances:
  - `car1 = Vehicle('Toyota', 'Camry', 2000, 'red')`
  - `car2 = Vehicle('Dodge', 'Caravan', 2015, 'gray')`
    ```

Test 1