Programming Principles in Python (CSCI 503/490)

Object-Oriented Programming

Dr. David Koop
Import Conventions

• Avoid wildcard imports like: `from math import *`

• Imports should be on separate lines
  - `import sys
    import os`

• Sometimes, a conditional import is required
  - `if sys.version_info >= [3,7]:
      OrderedDict = dict
    else:
      from collections import OrderedDict`

• Absolute imports best but relative imports allowed (`import .submodule`)

• Import abbreviations: `import pandas as pd; import numpy as np`
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
Finding & Installing Packages

- Python Package Index (PyPI) is the standard repository (https://pypi.org) and pip (pip installs packages) is the official python package installer.
- Anaconda is a package index, conda is a package manager.
- To install packages:
  - pip install <package-name>
  - conda install <package-name>
  - Jupyter: Add % (%pip, %conda)
- Both pip and conda support environments
  - venv
  - conda env
Assignment 4

- Assignment covers strings and files
- Reading & writing data to files
- Dealing with encodings and string formatting
- Due today
Assignment 5

- Scripts, modules, packages
- Command-line program
- Out soon
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)
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 2024 - self.year

• Instances:
  - car1 = Vehicle('Honda', 'Accord', 2009, 'red')
  - car2 = Vehicle('Dodge', 'Caravan', 2015, 'gray')
Constructor

• How an object is created and initialized
  - def __init__(self, make, model, year, color):
    self.make = make
    self.model = model
    self.year = year
    self.color = color

• __init__ denotes the constructor
  - Not required, but usually should have one
  - All initialization should be done by the constructor
  - There is only one constructor allowed
  - Can add defaults to the constructor (year=2015, color='gray')
## Instance Attributes

- Where information about an object is stored
  ```python
  def __init__(self, make, model, year, color):
      self.make = make
      self.model = model
      self.year = year
      self.color = color
  ```

- `self` is the current object
- `self.make`, `self.model`, `self.year`, `self.color` are **instance attributes**

- **No declaration** required for instance attributes like in Java or C++
  - Can be created in any instance method...
  - ...but good OOP design means they should be initialized in the constructor
Instance Methods

• Define actions for instances
  - def age(self):
    return 2024 - self.year

• Like constructors, have `self` as first argument
• `self` will be the object calling the method
• Have access to instance attributes and methods via `self`
• Otherwise works like a normal function
• Can also **modify** instances in instance methods:
  - def set_age(self, age):
    self.year = 2024 - age
Creating and Using Instances

• Creating instances:
  - Constructor expressions specify the name of the class to instantiate and specify any arguments to the constructor (not including `self`)
  - Returns new object
  - `car1 = Vehicle('Honda', 'Accord', 2009, 'red')`
  - `car2 = Vehicle('Dodge', 'Caravan', 2015, 'gray')`

• Calling an instance method
  - `car1.age()`
  - `car1.set_age(20)`
  - Note `self` is not passed explicitly, it's `car1` (instance before the dot)
Used Objects Many Times Before

• Everything in Python is an object!
  - my_list = list()
  - my_list.append(3)
  - num = int('64')
  - name = "Gerald"
  - name.upper()
Visibility

- In some languages, encapsulation allows certain attributes and methods to be hidden from those using an instance.
- public (visible/available) vs. private (internal only).
- Python does not have visibility descriptors, but rather conventions (PEP8):
  - Attributes & methods with a leading underscore (_) are intended as private.
  - Others are public.
  - You can still access private names if you want but generally shouldn't:
    - print(car1._color_hex)
  - Double underscores leads to name mangling:
    - self.__internal_vin is stored at self._Vehicle__internal_vin.
Representation methods

- Printing objects:
  
  ```python
  print(car1) # <__main__.Vehicle object at 0x7efc087c6b20>
  ```

- "Dunder-methods": ```__init__```

- Two for representing objects:
  
  - ```__str__```: human-readable
  
  - ```__repr__```: official, machine-readable

```python
>>> now = datetime.datetime.now()
">>> now.__str__()  
  '2020-12-27 22:28:00.324317'
>>> now.__repr__()
  'datetime.datetime(2020, 12, 27, 22, 28, 0, 324317)' 
```
Representation methods

• Car example:
  - class Vehicle:
    ...
    
    def __str__(self):
      return f'{self.year} {self.make} {self.model}'

• Don't call `print` in this method! Return a string
• When using, don't call directly, use `str` or `repr`
  - `str(car1)`
• `print` internally calls `__str__`
  - `print(car1)`
Other Dunder Methods

• `__eq__(<other>)`: return True if two objects are equal
• `__lt__(<other>)`: return True if object < other
• Collections:
  - `__len__()`: return number of items
  - `__contains__(item)`: return True if collection contains item
  - `__iter__()`: returns iterator
• Sequence + dict
  - `__getitem__(index)`: return item at index (which could be a key)
• + More
Properties

• Common pattern is getters and setters:
  - def age(self):
    return 2024 - self.year
  - def set_age(self, age):
    self.year = 2024 - age

• In some sense, this is no different than year except that we don't want to store age separate from year (they should be linked)

• Properties allow transformations and checks but are accessed like attributes

• @property
def age(self):
    return 2024 - self.year

carl.age # 15
Properties

• Can also define setters
• Syntax is a bit strange, want to link the two: @<property-name>.setter
• Method has the same name as the property: How?
• Decorators (@<decorator-name>) do some magic

• @property
def age(self):
    return 2024 - self.year
• @age.setter
def age(self, age):
    self.year = 2024 - age
• car1.age = 15
Properties

• Add validity checks!
• First car was 1885 so let's not allow ages greater than that (or negative ages)
• @age.setter
def age(self, age):
    if age < 0 or age > 2024 - 1885:
        print("Invalid age, will not set")
    else:
        self.year = 2024 - age
• Better: raise exception (later)
Class Attributes

• We can add class attributes inside the class indentation:

• Access by prefixing with **class name** or **self**

  ```python
  - class Vehicle:
    CURRENT_YEAR = 2024
    ...
    @age.setter
    def age(self, age):
      if age < 0 or age > Vehicle.CURRENT_YEAR - 1885:
        print("Invalid age, will not set")
      else:
        self.year = self.CURRENT_YEAR - age
  ```

• Constants should be **CAPITALIZED**

• This is not a great constant! (**EARLIEST_YEAR = 1885** would be!)
Class and Static Methods

• Use `@classmethod` and `@staticmethod` decorators
• Difference: class methods receive class as argument, static methods do not
• ```python
class Square(Rectangle):
    DEFAULT_SIDE = 10
...

@classmethod
def set_default_side(cls, s):
    cls.DEFAULT_SIDE = s

@staticmethod
def set_default_side_static(s):
    Square.DEFAULT_SIDE = s
```
Class and Static Methods

- `class Square(Rectangle):
  DEFAULT_SIDE = 10

  def __init__(self, side=None):
    if side is None:
      side = self.DEFAULT_SIDE
    super().__init__(side, side)
...

- `Square.set_default_side(20)
  s2 = Square()
  s2.side # 20

- `Square.set_default_side_static(30)
  s3 = Square()
  s3.side # 30`
Inheritance

• Is-a relationship: Car is a Vehicle, Truck is a Vehicle
• Make sure it isn't composition (has-a) relationship: Vehicle has wheels, Vehicle has a steering wheel
• Subclass is specialization of base class (superclass)
  - Car is a subclass of Vehicle, Truck is a subclass of Vehicle
• Can have an entire hierarchy of classes (e.g. Chevy Bolt is subclass of Car which is a subclass of Vehicle)
• Single inheritance: only one base class
• Multiple inheritance: allows more than base class
  - Many languages don't support, Python does
Subclass

• Just put superclass(-es) in parentheses after the class declaration
• class Car(Vehicle):
  
  def __init__(self, make, model, year, color, num_doors):
    super().__init__(make, model, year, color)
    self.num_doors = num_doors
  
  def open_door(self):
    ...

• super() is a special method that locates the base class
  - Constructor should call superclass constructor, then initialize its own extra attributes
  - Instance methods can use super, too
Instance Attribute Conventions in Python

• Remember, the naming is the convention
• public: used anywhere
• _protected: used in class and subclasses
• __private: used only in the specific class
• Note that double underscores induce name mangling to strongly discourage access in other entities
Overriding Methods

- class Rectangle:
  
  ```python
  def __init__(self, height, width):
    self.h = height
    self.w = weight
  
  def set_height(self, height):
    self.h = height
  
  def area(self):
    return self.h * self.w
  ```

- class Square(Rectangle):
  
  ```python
  def __init__(self, side):
    super().__init__(side, side)
  
  def set_height(self, height):
    self.h = height
    self.w = height
  ```

- s = Square(4)
- s.set_height(8)
- Which method is called?
- Polymorphism
- Resolves according to inheritance hierarchy
- s.area() # 64
- If no method defined, goes up the inheritance hierarchy until found
Class and Static Methods

• Use `@classmethod` and `@staticmethod` decorators
• Difference: class methods receive class as argument, static methods do not

```python
class Square(Rectangle):
    DEFAULT_SIDE = 10
...

@classmethod
def set_default_side(cls, s):
    cls.DEFAULT_SIDE = s

@staticmethod
def set_default_side_static(s):
    Square.DEFAULT_SIDE = s
```
Class and Static Methods

• class Square(Rectangle):
  DEFAULT_SIDE = 10

  def __init__(self, side=None):
    if side is None:
      side = self.DEFAULT_SIDE
    super().__init__(side, side)

  ...

• Square.set_default_side(20)
  s2 = Square()
  s2.side # 20

• Square.set_default_side_static(30)
  s3 = Square()
  s3.side # 30
Class and Static Methods

• class NewSquare(Square):
  DEFAULT_SIDE = 100

• NewSquare.set_default_side(200)
  s5 = NewSquare()
  s5.side # 200

• NewSquare.set_default_side_static(300)
  s6 = NewSquare()
  s6.side # !!! 200 !!!

• Why?
  - The static method sets Square.DEFAULT_SIDE not the
    NewSquare.DEFAULT_SIDE
  - self.DEFAULT_SIDE resolves to NewSquare.DEFAULT_SIDE