Programming Principles in Python (CSCI 503)

Object-Oriented Programming

Dr. David Koop
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
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
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
  - Extra arguments should be initialized and extra instance methods
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
Duck Typing

• "If it looks like a duck and quacks like a duck, it must be a duck."

• Python "does not look at an object’s type to determine if it has the right interface; instead, the method or attribute is simply called or used"

• class Rectangle:
  
  def area(self):
      ...

• class Circle:
  
  def area(self):
      ...

• It doesn't matter that they don't have a common base class as long as they respond to the methods/attributes we expect: shape.area()
Multiple Inheritance

- Can have a class inherit from two different superclasses
- HybridCar inherits from Car and Hybrid
- Python allows this!
  - class HybridCar(Car, Hybrid): ...
- Problem: how is super() is defined?
  - Diamond Problem
  - Python use the \textbf{method resolution order} (MRO) to determine order of calls
Method Resolution Order

• The order in which Python checks classes for a method

• `mro()` is a `class` method

• `Square.mro()` # `[__main__.Square, __main__.Rectangle, object]`

• Order of base classes matters:
  - `class HybridCar(Car, Hybrid):
      pass`
    `HybridCar.mro()` # `[__main__.HybridCar, __main__.Car, __main__.Hybrid, __main__.Vehicle, object]`
  - `class HybridCar(Hybrid, Car):
      pass`
    `HybridCar.mro()` # `[__main__.HybridCar, __main__.Hybrid, __main__.Car, __main__.Vehicle, object]`
Operator Overloading

• Dunder methods (__add__, __contains__, __len__)  
• Example:

```python
- class Square(Rectangle):
  ...
  @property
  def side(self):
    return self.h
  def __add__(self, right):
    return Square(self.side + right.side)
  def __repr__(self):
    return f'{self.__class__.__name__}({self.side})'

new_square = Square(8) + Square(4)
new_square # Square(12)
```
Operator Overloading Restrictions

• Precedence cannot be changed by overloading. However, parentheses can be used to force evaluation order in an expression.
• The left-to-right or right-to-left grouping of an operator cannot be changed.
• The “arity” of an operator—that is, whether it’s a unary or binary operator—cannot be changed.
• You cannot create new operators—only overload existing operators.
• The meaning of how an operator works on objects of built-in types cannot be changed. You cannot change + so that it subtracts two integers.
• Works only with objects of custom classes or with a mixture of an object of a custom class and an object of a built-in type.
Assignment 6

- Object-oriented Programming
- Track University Enrollment
  - Academic (student, graduate student, instructor)
  - Course (name, department, number of credits, instructor, enrolled students)
  - Schedule (a person's list of courses)
  - Registrar (keeps track of all people and courses)
- Methods for checking conflicts (e.g. disallow student to have overlapping courses, take too many credits)
- Methods for changing course time (check the new time works for everyone)
- Due Friday, March 19
Ternary Operator

- \( a = b < 5 \ ? \ b + 5 : b - 5 \)
- Kind of a weird construct, but can be a nice shortcut
- `<value>` if `<condition>` else `<value>`
- \( \text{absx} = x \text{ if } x \geq 0 \text{ else } -x \)
- Reads so that the usual is listed first and the abnormal case is listed last
- "Usually this, else default to this other"
Exercise

• Create Stack and Queue classes
  - Stack: last-in-first-out
  - Queue: first-in-first-out
• Define constructor and push and pop methods for each
Object-Based Programming

• With Python's libraries, you often don't need to write your own classes. Just
  - Know what libraries are available
  - Know what classes are available
  - Make objects of existing classes
  - Call their methods
• With inheritance and overriding and polymorphism, we have true object-oriented programming (OOP)
Named Tuples

- Tuples are immutable, but cannot refer to with attribute names, only indexing
- Named tuples add the ability to use dot-notation

```python
from collections import namedtuple
Car = namedtuple('Car', ['make', 'model', 'year', 'color'])
car1 = Car(make='Toyota', model='Camry', year=2000, color='red')
car2 = Car('Ford', 'F150', 2018, 'gray')
```

- Can use kwargs or positional or mix
- Access via dot-notation:
  - `car1.make # "Toyota"
  - `car2.year # 2018"
SimpleNamespace

• Named tuples do not allow mutation
• SimpleNamespace does allow mutation:

• from types import SimpleNamespace
car3 = SimpleNamespace(make='Toyota', model='Camry', year=2000, color="red")

• car3.num_doors = 4 # would fail for namedtuple
• Doesn't enforce any structure, though
Typing

- Dynamic Typing: variable's type can change (what Python does)
- Static Typing: compiler enforces types, variable types generally don't change
- Duck Typing: check method/attribute existence, not type
- Python is a dynamically-typed language (and plans to remain so)
- …but it has recently added more support for type hinting/annotations that allow static type checking
- Type annotations change nothing at runtime!
Type Annotations

- \texttt{def area(width : float, height : float) -> float:}
  \hspace{1cm} return width * height

- colon (:) after parameter names, followed by type

- arrow (\texttt{->}) after function signature, followed by type (then final colon)

- area("abc", 3) # runs, returns "abcabcabc"

- These won't prevent you from running this function with the wrong arguments or returning a value that doesn't satisfy the type annotation

- Extensions for collections allows inner types to be specified:
  - from typing import List
    \hspace{1cm} names : List[str] = ['Alice', 'Bob']

- Any and Optional, too
mypy

• A static type checker for Python that uses the type annotations to check whether types work out
• $ mypy <script.py>
  - Writes type errors tagged by the line of code that introduced them
  - Can also reveal the types of variables at various parts of the program
• There is an extension for Jupyter (mypy_ipython), but it basically works by converting all cells to a script and then running mypy
  - Cells not tagged in error messages
  - Re-running cells introduces multiple copies of error
  - Deleting cells doesn't remove errors
Type Checking in Development Environments

- PyCharm can also use the type hints to do static type checking to alert programmers to potential issues
- Microsoft VS Code Integration using Pyright
Type Checking Pros & Cons

- **Pros:**
  - Good for documentation
  - Improve IDEs and linters
  - Build and maintain cleaner architecture

- **Cons:**
  - Takes time and effort!
  - Requires modern Python
  - Some penalty for typing imports (can be alleviated)
When to use typing

- No when learning Python
- No for short scripts, snippets in notebooks
- Yes for libraries, especially those used by others
- Yes for larger projects to better understand flow of code
Data Classes

• from dataclasses import dataclass
  @dataclass
  class Rectangle:
    width: float
    height: float
  Rectangle(34, 21) # just works!

• Does a lot of boilerplate tasks
  - Creates basic constructor (__init__)
  - Creates __repr__ method
  - Creates comparison dunder methods (==, !=, <, >, <=, >=)
Data Classes

• Requires type annotations, but just like other type annotations, they are not checked at runtime!
• Rectangle("abc", "def") # no error!
• Use mypy to check typing
• If typing is not important, use typing.Any for types
• from typing import Any
  from dataclasses import dataclass
  @dataclass
class Rectangle:
    width: Any
    height: Any
Data Classes

• Can add methods as normal
• from dataclasses import dataclass
  @dataclass
class Rectangle:
    width: float
    height: float

def area(self):
    return self.width * self.height

• Supports factory methods for more complicated inits
• __post_init__ method for extra processing after __init__