Programming Principles in Python (CSCI 503/490)

Object-Oriented Programming

Dr. David Koop

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 2022 - self.year
    ```
- Instances:
  - `car1 = Vehicle('Toyota', 'Camry', 2000, 'red')`
  - `car2 = Vehicle('Dodge', 'Caravan', 2015, 'gray')`

Properties

- Properties allow transformations and checks but are accessed like attributes
- getter and setter have same name, but different decorators
- Decorators (`@<decorator-name>`) do some magic

-
  ```
  @property
  def age(self):
      return 2021 - self.year
  ```

-
  ```
  @age.setter
  def age(self, age):
      self.year = 2021 - age
  ```
- Using property:
  - `car1.age = 20`

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

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 Visibility Conventions in Python

- Remember, the naming is the convention (PEP8)
  - `public`: used anywhere
  - `_protected`: used in class and subclasses
  - `__private`: used only in the specific class
- 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`
  - This is why `__private` makes sense (tied to defining class)

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:
      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):
      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

-
  ```
  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 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`

Assignment 5

- Due next Monday
- Same Senate Stock Tracker data as A3
- Scripts, modules, packages
- Command-line program

Quiz Wednesday

- Quiz on Object-Oriented Programming

Operator Overloading

- Dunder methods (`__add__`, `__contains__`, `__len__`)
- Example:

```
- 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.

[Deitel & Deitel]

Left and Right Operands?

-
  ```
  class Square(Rectangle):
      …
      def __add__(self, right):
          return Square(self.side + right)

  Square(8) + 4 # Square(12)
  4 + Square(8) # error
  ```
- Solution: Use `__radd__` and related operators

-
  ```
  class Square(Rectangle):
      …
      def __radd__(self, left):
          return Square(left + self.side)

  4 + Square(8) # Square (12)
  ```

Ternary Operator

- In other languages: `a = b < 5 ? b + 5 : b - 5`
- Means: `if (b < 5) a = b + 5; else a = b - 5;`
- Kind of a weird construct, but can be a nice shortcut
- Python does this differently:
- `<value> if <condition> else <value>`
- Python Example: `a = b + 5 if b < 5 else b - 5`
- Reads so that the usual is listed first and the abnormal case is listed last
- "Usually this, else default to this other" (cases are pushed apart)

Checking type

- We can check the type of a Python object using the `type` method:
  - `type(6) # int`
  - `type("abc") # str`
  - `s = Square(4)`
  - `type(s) # Square`
- Allows comparisons:

  -
    ```
    if type(s) == Square:
        # ...
    ```
- But this is **False**:

  -
    ```
    if type(s) == Rectangle:
        # ...
    ```

Checking InstanceOf/Inheritance

- How can we see if an object is an **instance** of a particular class or whether a particular class is a **subclass** of another?
- Both check is-a relationship (but differently)
- `issubclass(cls1, cls2)`: checks if `cls1` is-a (subclass of) `cls2`
- `isinstance(obj, cls)`: checks if `obj` is-a(n instance of) `cls`
- Note that `isinstance` is `True` if `obj` is an instance of a class that is a subclass of `cls`

  -
    ```
    car = Car('Toyota','Camry', 2000, 'red', 4)
    isinstance(car, Vehicle) # True
    ```

Interfaces

- In some languages, can define an abstract base class
  - The structure is defined but **without implementation**
  - Alternatively, some methods are defined abstract, others are implemented
- Interfaces are important for types
  - Method can specify a particular type that can be abstract
  - This doesn't matter as much in Python
- However, Python does have ABCs (Abstract Base Classes)
  - Solution to be able to check for mappings, sequences via `isinstance`, etc.
  - `abc.Mapping, abc.Sequence, abc.MutableSequence`

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()`

[Python Glossary]

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 **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]
    ```

Mixins

- Sometimes, we just want to add a particular method to a bunch of different classes
- For example: `print_as_dict()`
- A mixin class allows us to specify one or more methods and add it as the second
- Caution: Python searches from left to right so a base class should be at the right with mixing