Programming Principles in Python (CSCI 503/490)

Object-Oriented Programming

Dr. David Koop
Arrays: Boolean Indexing

- names == 'Bob' gives back booleans that represent the element-wise comparison with the array names
- Boolean arrays can be used to index into another array:
  - data[names == 'Bob']
- Can even mix and match with integer slicing
- Can do boolean operations (&, |) between arrays (just like addition, subtraction)
  - data[(names == 'Bob') | (names == 'Will')]
- Note: or and and do not work with arrays
- We can set values too! data[data < 0] = 0
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): 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)
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')
Components

• Constructor: `__init__`
• Instance Attributes: `self.make`, `self.model`, `self.year`
• Instance Methods: `def age`, `def set_age`
• Using classes and instances:
  - `car1 = Vehicle('Toyota', 'Camry', 2000, 'red')`
  - `car1.set_age(20)`
• Visibility: no declaration, convention with underscore: `_color_hex`
• String Representation: define `__str__`, call `str()`
Assignment 5

- Scripts and Modules
- Write a three modules in a Python package with methods to process the Senate stock tracking data
- Write a script with command-line arguments to analyze this data using the new package
- Turn in a zip file with package and script
- No notebook required, but useful to test your code as you work
  - `%autoreload` or `importlib.reload`
Properties

• Common pattern is getters and setters:
  - def age(self):
    return 2021 - self.year
  - def set_age(self, age):
    self.year = 2021 - 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 2021 - self.year

• car1.age # 21
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 2021 - self.year`
• `@age.setter
def age(self, age):
    self.year = 2021 - age`
• `car1.age = 20`
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 > 2021 - 1885:
        print("Invalid age, will not set")
    else:
        self.year = 2021 - 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 = 2021
    ...
    @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`
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`
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?
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
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

s = Square(4)
s.set_height(8)
```

- Which method is called?
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()

- Which method is called?
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

  def area(self):
    return self.h * self.w

- s = Square(4)
- s.set_height(8)

  - Which method is called?
  - Polymorphism
  - Resolves according to inheritance hierarchy

- s.area() # 64
  - Which method is called?
  - If no method defined, goes up the inheritance hierarchy until found
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"

```python
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 uses the **method resolution order** (MRO) to determine order of calls