Programming Principles in Python (CSCI 503)

Object-Oriented Programming

Dr. David Koop
Figure 4-2. Two-dimensional array slicing

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

In [87]: names == 'Bob'
Out[87]: array([ True, False, False, True, False, False, False], dtype=bool)

This boolean array can be passed when indexing the array:

In [88]: data[names == 'Bob']
Out[88]:
array([[-0.048 ,  0.5433, -0.2349,  1.2792],
       [ 2.1452,  0.8799, -0.0523,  0.0672]])

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

In [89]: data[names == 'Bob', 2:]
Out[89]:
array([[-0.2349,  1.2792]])

How to obtain the blue slice from array arr?
2D Array Slicing

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

```
In [87]: names == 'Bob'
Out[87]: array([ True, False, False, True, False, False, False], dtype=bool)
```

This boolean array can be passed when indexing the array:

```
In [88]: data[names == 'Bob']
Out[88]:
array([[ -0.048 ,  0.5433, -0.2349,  1.2792],
       [ 2.1452,  0.8799, -0.0523,  0.0672]])
```

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

```
In [89]: data[names == 'Bob', 2:]
Out[89]:
array([[-0.2349,  1.2792]])
```

How to obtain the blue slice from array `arr`?

[W. McKinney, Python for Data Analysis]
Figure 4-2. Two-dimensional array slicing

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

\[
\text{In } [87] \text{: } \text{names } == \text{ 'Bob'} \\
\text{Out}[87] \text{: } \text{array([ True, False, False, True, False, False, False]), dtype=bool)}
\]

This boolean array can be passed when indexing the array:

\[
\text{In } [88] \text{: } \text{data}[\text{names } == \text{ 'Bob'}] \\
\text{Out}[88] \text{: } \text{array([[-0.048 , 0.5433, -0.2349, 1.2792], [2.1452, 0.8799, -0.0523, 0.0672]])}
\]

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

\[
\text{In } [89] \text{: } \text{data}[\text{names } == \text{ 'Bob'}, 2:] \\
\text{Out}[89] \text{: } \text{array([[-0.2349, 1.2792]])}
\]

How to obtain the blue slice from array arr?

[W. McKinney, Python for Data Analysis]
Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

```python
In [87]: names == 'Bob'
Out[87]: array([ True, False, False, True, False, False, False], dtype=bool)
```

This boolean array can be passed when indexing the array:

```python
In [88]: data[names == 'Bob']
Out[88]:
array([[-0.048 ,  0.5433, -0.2349,  1.2792],
       [ 2.1452,  0.8799, -0.0523,  0.0672]])
```

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

```python
In [89]: data[names == 'Bob', 2:]
Out[89]:
array([[-0.2349,  1.2792]])
```

How to obtain the blue slice from array `arr`?

---

[W. McKinney, Python for Data Analysis]
2D Array Slicing

How to obtain the blue slice from array \(\text{arr}\)?

<table>
<thead>
<tr>
<th>Expression</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{arr}[2, 1:])</td>
<td>(2, 2)</td>
</tr>
<tr>
<td>(\text{arr}[2])</td>
<td>(3,)</td>
</tr>
<tr>
<td>(\text{arr}[2, :])</td>
<td>(3,)</td>
</tr>
<tr>
<td>(\text{arr}[2:, :])</td>
<td>(1, 3)</td>
</tr>
<tr>
<td>(\text{arr}[:, 2])</td>
<td>(3, 2)</td>
</tr>
<tr>
<td>(\text{arr}[1, 2])</td>
<td>(2,)</td>
</tr>
<tr>
<td>(\text{arr}[1:2, 2])</td>
<td>(1, 2)</td>
</tr>
</tbody>
</table>

[W. McKinney, Python for Data Analysis]
Boolean Indexing

- names == 'Bob' gives back bools 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 a 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:
    
    ```python
    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()`
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
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!)
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
  - 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
  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
  - If no method defined, goes up the inheritance hierarchy until found
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 (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
- Python has ABC (Abstract Base Class)
  - 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()`
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
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
Operator Overloading

• Dunder methods
• Examples:
  - `__add__`(self, right)
  - `__iadd__`(self, right)