Programming Principles in Python (CSCI 503)

Object-Oriented Programming

Dr. David Koop
Arrays

- Usually a fixed size—lists are meant to change size
- Are mutable—tuples are not
- Store only one type of data—lists and tuples can store anything
- Are faster to access and manipulate than lists or tuples
- Can be multidimensional:
  - Can have list of lists or tuple of tuples but no guarantee on shape
  - Multidimensional arrays are rectangles, cubes, etc.
NumPy Arrays

• import numpy as np

• Creating:
  - data1 = [6, 7, 8, 0, 1]
  - arr1 = np.array(data1)
  - arr1_float = np.array(data1, dtype='float64')
  - np.ones((4,2)) # 2d array of ones
  - arr1_ones = np.ones_like(arr1) # [1, 1, 1, 1, 1]

• Type and Shape Information:
  - arr1.dtype # int64 # type of values stored in array
  - arr1.ndim # 1 # number of dimensions
  - arr1.shape # (5,) # shape of the array
Array Operations

- \(a = \text{np.array([1,2,3])}\)
  \(b = \text{np.array([6,4,3])}\)

- (Array, Array) Operations (**Element-wise**)
  - Addition, Subtraction, Multiplication
  - \(a + b \# \text{array([7, 6, 6])}\)

- (Scalar, Array) Operations (**Broadcasting**):
  - Addition, Subtraction, Multiplication, Division, Exponentiation
  - \(a ** 2 \# \text{array([1, 4, 9])}\)
  - \(b + 3 \# \text{array([9, 7, 6])}\)
Indexing

- Same as with lists
  - `arr1 = np.array([6, 7, 8, 0, 1])`
  - `arr1[1]`
  - `arr1[-1]`

- What about two dimensions?
  - `arr2 = np.array([[1,2,3],
                     [4,5,6],
                     [7,8,9]])`
  - `arr[1][1]`
  - `arr[1,1]` # shorthand

In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions. So in the $2 \times 2 \times 3$ array $arr3d$:

```
In [76]: arr3d = np.array([[[1, 2, 3],
                       [4, 5, 6]],
                      [[7, 8, 9]]])
Out[76]:
array([[[1, 2, 3],
        [4, 5, 6]],[
        [7, 8, 9]])
```

`arr3d[0]` is a $2 \times 3$ array:

```
In [77]: arr3d[0]
Out[77]:
array([[1, 2, 3],
        [4, 5, 6]])
```

Both scalar values and arrays can be assigned to `arr3d[0]`:

```
In [78]: old_values = arr3d[0].copy()
In [79]: arr3d[0] = 42
Out[81]:
array([[[42, 42, 42],
        [42, 42, 42]],[
        [7, 8, 9]],
        [10, 11, 12]])
```

D. Koop, CSCI 503, Spring 2021
Slicing

• 1D: Similar to lists
  - arr1 = np.array([6, 7, 8, 0, 1])
  - arr1[2:5] # np.array([8,0,1]), sort of

• Can **mutate** original array:
  - arr1[2:5] = 3 # supports assignment

• Slicing returns **views** (copy the array if original array shouldn't change)
  - arr1.copy() or arr1[2:5].copy() will copy
Assignment 5

• Scripts and Modules
• Write a three modules in a Python package with methods to process Pokémon data
• Write a script to retrieve Pokémon information via command-line arguments
• MaxCP formula fixed by 2021-02-28
• Turn in a zip file with package
• No notebook required, but useful to test your code as you work
  - %autoreload or importlib.reload
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:

```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`?
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],
       [ 2.1452,  0.8799]])
```

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

How to obtain the blue slice from array \texttt{arr}?

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

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

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

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

[Source: 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:

\[
\text{In [87]: names == 'Bob'}
\]

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

This boolean array can be passed when indexing the array:

\[
\text{In [88]: data[names == 'Bob']}
\]

\[
\text{Out[88]: array([[-0.048 , 0.5433, -0.2349, 1.2792],
                      [ 2.1452, 0.8799, -0.0523, 0.0672]], dtype=float64)}
\]

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):

\[
\text{In [89]: data[names == 'Bob', 2:]}\]

\[
\text{Out[89]: array([[-0.2349, 1.2792],
                      [ 2.1452, 0.8799]], dtype=float64)}
\]

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[:, :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>
Slicing

• 2D+: comma separated indices as shorthand:
  - arr2 = np.array([[1.5,2,3,4],[5,6,7,8]])
  - a[1:2,1:3]
  - a[1:2,:] # works like in single-dimensional lists

• Can combine index and slice in different dimensions
  - a[1,:] # gives a row
  - a[:,1] # gives a column

• Slicing vs. indexing produces different shapes!
  - a[1,:] # 1-dimensional
  - a[1:2,:] # 2-dimensional
More Reshaping

• reshape:
  - arr2.reshape(4,2) # returns new view

• resize:
  - arr2.resize(4,2) # no return, modifies arr2 in place

• flatten:
  - arr2.flatten() # array([1.5, 2., 3., 4., 5., 6., 7., 8.])

• ravel:
  - arr2.ravel() # array([1.5, 2., 3., 4., 5., 6., 7., 8.])

• flatten and ravel look the same, but ravel is a view
Array Transformations

• Transpose
  - `arr2.T` # flip rows and columns

• Stacking: take iterable of arrays and stack them horizontally/vertically
  - `arrh1 = np.arange(3)`
  - `arrh2 = np.arange(3, 6)`
  - `np.vstack([arrh1, arrh2])`
  - `np.hstack([arr1.T, arr2.T])` # ???
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
Object-Oriented Programming Concepts

- ?
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)
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)
  - Methods (actions)
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)
Other Entities

- Road, Person, Building, ParkingLot
- Some of these interact with a Vehicle, some don't
- We want to store information associated with entities in a structured way
  - Building probably won't store anything about cars
  - Road should not store each car's make/model
  - …but we may have an association where a Road object keeps track of the cars currently driving on it
Object-Oriented Design

- There is a lot more than can be said about how to best define classes and the relationship between different classes
- It's not easy to do this well!
- Software Engineering
- Entity Relationship (ER) Diagrams
- Difference between Object-Oriented Model and ER Model
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:
  
  ```python
  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:
  
  ```python
  car1 = Vehicle('Toyota', 'Camry', 2000, '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=2021, color='gray')
Instance Attributes

• Where information about an object is stored
  - `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

• There is 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 2021 - 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 = 2021 - 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('Toyota', 'Camry', 2000, '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:**
  
  - `print(car1) # <__main__.Vehicle object at 0x7efc087c6b20>`

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

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

- ```py
>>> 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`
  - `__getitem__(index)`: return item at `index` (which could be a key)

• + More
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`
  ```python
def age(self):
    return 2021 - self.year
  ```
• `@age.setter`
  ```python
def age(self, age):
    self.year = 2021 - age
  ```
• `carl.age = 20`
Properties

- Add validity checks!
- First car was 1885 so let's not allow ages greater than that (or negative ages)
- `@age.setter`
  ```python
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**

- 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!)