ECE 20875

Python for Data Science

David Inouye and Qiang Qiu

(Adapted from material developed by Profs. Milind Kulkarni, Stanley Chan, Chris Brinton, David Inouye)

objects and classes
Python is OOP

• Like C++ and Java, Python is an **object-oriented programming (OOP)** language

• An **object** is Python’s abstraction for data
  • A bundle of *data* and *operations* that execute on this data

• Everything in Python is an object
  • All data is represented by objects or relations between objects
  • This includes “simple” data like integers and floats
  • Even functions are special objects in Python
we’ve been using OOP all along

- Some **classes** we’ve used so far or will use soon (you can see all of their source code on [github](https://github.com/scikit-learn/scikit-learn/blob/fd237278e/sklearn/linear_model/_base.py#L389))
  
  - [sklearn.linear_model.LinearRegression](https://github.com/scikit-learn/scikit-learn/blob/fd237278e/sklearn/linear_model/_base.py#L389)
  
  - [sklearn.svm.SVC](https://github.com/scikit-learn/scikit-learn/blob/fd237278e/sklearn/svm/_classes.py#L428)

- Some instance **variables** we’ve used so far or will use soon:
  
  

- Some **methods** we’ve used so far or will use soon:
  
  - `re.sub(...)`
  
  - `np.sort(...)`
  
  - [LogisticRegression.fit(...)](https://github.com/scikit-learn/scikit-learn/blob/fd237278e/sklearn/linear_model/logistic.py#L1011)
every object in Python has …

1. an **identity**, accessed through the `id()` function
   - Unique “name” for an object, like its address in memory, which never changes

2. a **type**, accessed through the `type()` function
   - This defines the operations that you can perform on an object (asking for its length, adding to it, etc.)
   - Also defines the possible values this object can take

3. a **value**, which defines the data associated with the object
   - Think the contents of a list, or the value of an integer
   - Objects whose values can change (e.g., a dictionary) are **mutable**, while objects whose values cannot be changed (e.g., a tuple) are **immutable**
defining an object

• Intuition: an object is defined by

  1. Where it is (what box of memory contains its information)

  2. What it can do (what operations you can perform on it)

  3. What it has (what data those operations will operate on)

• Formally, an object is defined as an instance of a class

  • A class is like a fill-in-the-blank sheet, template, or blueprint

  • An instance is like a template that has been filled in with particular values or an actual building/object

• Any data scientist can write their own ML class and submit it to scikit-learn

  • Must follow the common basic API (https://scikit-learn.org/stable/developers/develop.html): estimator, predictor, transformer, model
instantiating objects from classes

• We define what an object has (variables) and what it can do (methods) by creating a class for that object

• Think of this as a template for an object that specifies what information and actions this object has

• There are two types of class attributes:
  1. variables (either class variables or instance variables), which hold the data we want in an object
  2. methods, which are the functions we want to be able to invoke on an object

• __init__(): Special constructor method automatically invoked for each new class instance

```python
class Foo:
    x = 7  # this will be accessible to all Foos
    # it is a class variable

    # this is called when a new Foo is created
def __init__(self, i):
        self.y = i  # this is specific to each Foo
        # it is an instance variable

        # this will be available to all Foos
        # it is a class method
def bar(self):
            return self.x + self.y

# defining objects as instances of class Foo
a = Foo(1)  # a.x = 7, a.y = 1
b = Foo(2)  # b.x = 7, b.y = 2

# invoking the bar method on the objects
print(a.bar())  # prints 8
print(b.bar())  # prints 9
```
Manipulating an object involves *invoking operations* on the object.

- Intuition: Think of this as “sending a message” to an object, i.e., asking an object to handle an action.

- Including things you might not think of!
  
  - \( x = a + b \) is invoking the \texttt{__add__()} method on object \( a \).
  
  - \( \text{len}(s) \) is invoking the \texttt{__len__()} method on object \( s \).
  
  - We can also overwrite these default methods if we want different functionality! (see example on the right)

```python
class MultipleLists():
    def __init__(self):
        self.lists = []
    def __add__(self, a):
        newlists = MultipleLists()
        newlists.lists = self.lists.copy()
        newlists.lists.append(a)
        return newlists
    def __len__(self):
        return sum([len(a) for a in self.lists])
    def __str__(self):
        return ', '.join([f'L{i+1}={a}' for i, a in enumerate(self.lists)])

many_lists = MultipleLists()
print(many_lists)  # ''
print(len(many_lists))  # 0

many_lists = many_lists + [3,5,1]
print(many_lists)  # L1=[3, 5, 1]
print(len(many_lists))  # 3

many_lists += [8, 4]
print(many_lists)  # L1=[3, 5, 1], L2=[8, 4]
print(len(many_lists))  # 5
```
creating, updating and accessing variables in objects

• Accessing **variables** in objects uses the “.” notation: my_object.x (MyClass.x for class variables)

• Under the hood, this is also invoking methods!

• Object variables can generally be:
  • created/deleted (if mutable object and user-created)
  • updated (if mutable object)
  • accessed

• Variable updates can be done either **internally** (via object methods, preferred) or **externally** (via “hard coding”, need to be careful when doing this)
the special role of **self** in defining or calling methods on objects

- When you call a method on an object, the object itself is always passed as the *first argument* of the method.
  - The object is called **self**.
  - Think of this like the `this` parameter in Java or C++ (except that it shows up explicitly in the argument list).
- By accessing `self.x`, we can create or access variables that are *specific to this object*.

```python
class Employee:
    empCount = 0
def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.empCount += 1
def displayCount(self):
    print("Total employees: \d\d")
Employee.empCount

def displayEmployee(self):
    print("Name: \s\s, \s\salary: \s\s")
emp1 = Employee("Alice", 100000)
emp2 = Employee("Bob", 50000)
emp1.displayEmployee()
emp1.displayCount()  # Total Employees: 2
emp2.displayCount()  # Total Employees: 2
```