ECE 20875

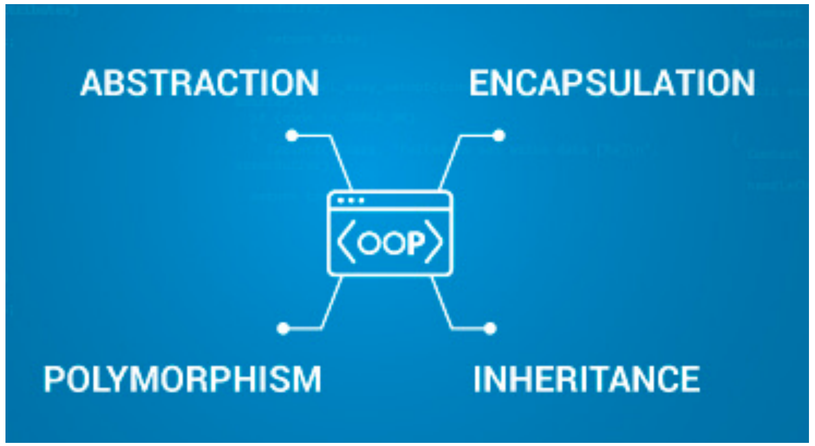
Python for Data Science

Chris Brinton and 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

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**

```
# Integers, lists, functions and objects
# (and even classes) are objects in Python
my_integer = 5
my_list = [1.0, 2, 3]
def my_function(): return 0
class MyClass: pass
my_object = EmptyClass()

# Show id and type of each object
for o in [my_integer, my_list,
          my_function, my_object, MyClass]:
    print(f'id={id(o)}, type={type(o)}’)


Output:
id=4308932128, type=<class 'int'>
id=4364494984, type=<class 'list'>
id=4363413160, type=<class 'function'>
id=4368615744, type=<class '__main__.EmptyClass'>
id=140649053790680, type=<class 'type'>
```

defining an object

- Intuition: an object is defined by

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

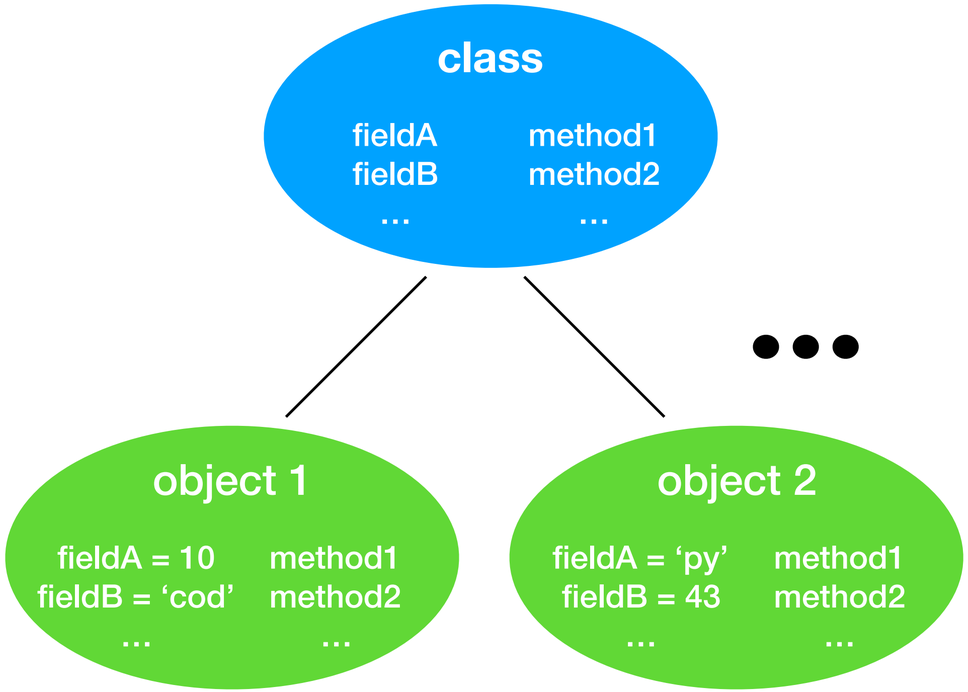
    2. What it *can do* (what operations you can perform on it)

    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



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

```
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 objects

- 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 `__add__()` method on object `a`
    - `len(s)` is invoking the `__len__()` method on object `s`

```
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 access can be done either internally (via object methods, preferred) or externally (via "hard coding", need to be careful when doing this)

```
class SimpleClass():
    def __init__(self, x):
        # internal created
        self.myx = x
    def add(self, y):
        # internal access and update
        self.myx = self.myx + y
my_object = SimpleClass(10)

# external access
print(my_object.myx) # 10

# internal update
my_object.add(15)
print(my_object.myx) # 25

# external update
my_object.myx = 200
print(my_object.myx) # 200

# external variable creation
my_object.myz = 18
print(my_object.myz) # 18

# external variable deletion
del my_object.myz
print(my_object.myz) # Error
```

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*

within the class, we have to use **self** as the first argument

```
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" %
    Employee.empCount)

  def displayEmployee(self):
    print("Name: ", self.name, ", Salary: ",
    self.salary)

 emp1 = Employee("Alice", 100000)
 emp2 = Employee("Bob", 50000)

 emp1.displayEmployee()
 emp1.displayCount()  #Total Employees: 2
 emp2.displayCount()  #Total Employees: 2
```

outside of the class, **self** is implicitly the first argument