DS5010

Intro to programming for Data Science

LECTURE 5

TODAY

- review session 4
- what are objects
- classes
- object oriented programming

REVIEW

- induction
- recursion
- problem solving with recursion and divide and conquer technique
- higher order functions and lambda expressions

OBJECTS?

- we are surrounded by many objects

DIFFERENT OBJECTS?

- how do we differentiate between objects?

?

==

DIFFERENT OBJECTS?

- how about these?

?

==

DIFFERENT OBJECTS?

- or these?

?

==

DIFFERENT OBJECTS?

- and these?

?

==

DIFFERENT OBJECTS?

- objects can have different **type**s

!=

cat type

dog type

TYPE

- we categorize **similar** objects under a same hood which is called **type**
- but how can we measure this similarity?
- if your measure is tight, each object can be a separate type for its own!
- if your measure is loose, all objects can be considered to be of a same type!
- so types are defined by **convention**!

DIFFERENT OBJECTS?

- objects of same type can have different **attributes**!

color: gray

!=

color: cream

DATA ATTRIBUTES

- each type of objects has a set of **data attributes** (ex: color, age, name, …)
- by looking at attributes, you can distinguish between different types
- the data attributes themselves can be of any type
- data attributes define the **state** of an object

DIFFERENT OBJECTS?

- or these?

age: 2 months

==

can be a same cat
which became older
after a while

age: 2 years

BEHAVIORAL ATTRIBUTES

- each type of objects may have some **behavioral attributes** (ex: walk, bark, jump, grow, …)
- these attributes, also can be used to distinguish between different types of objects
- the **state** of an object may **change/evolve** in the course of time, but that object remains a same existence! This change is usually done by means of behavioral attributes.

DIFFERENT OBJECTS?

- even same looking objects may be different entities!

?

may be equal
may be not

EXAMPLE: TRAFFIC LIGHT

- type: Traffic Light
- data attributes:
  - list_of_lights
  - wait_times
  - current_light
  - is_in_use
  - location
- behavioral attributes:
  - change_light
  - increase_wait_time
  - turn_off
- each **instance** of a Traffic light type, has its own values for the above attributes.

OBJECTS IN PYTHON

- Python supports many different kinds of data

```
1234    3.14159    "Hello"          [1, 5, 7, 11, 13]

{"CA": "California", "MA": "Massachusetts"}
```

- each is an **object**, and every object has:
  - a type
  - an internal **data representation**(primitive or composite)
  - a set of procedures for **interaction** with the object
- an object is an **instance** of a type
  - 1234 is an instance of an int
  - "hello" is an instance of a string

OBJECTS IN PYTHON

- objects are **a data abstraction** of an entity that captures…

(1) an **internal** representation of the entity's **state**

- through **data attributes**

(2) an **interface** for interacting with object, and exposing/manipulating its **state**

- through methods (aka procedures/functions/ **behavioral attributes**)
- defines behaviors but hides implementation

EXAMPLE: [1,2,3,4] has type list

- how are lists **represented internally**? linked list of cells

In reality its more complicated than a simple LinkedList!

- how to **access** the internals or **manipulate** lists?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(), L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- internal representation should be private
- correct behavior may be compromised if you manipulate internal representation directly

DEFINE YOUR OWN TYPES

- use the `class` keyword to define a new type

```
class <type_name>:
    # define attributes here
```

this is a **blueprint** for creating objects which are of type `type_name`

- similar to `def`, indent code to indicate which statements are part of the **class definition**
- example: We want to define a class of objects which are of type Coordinate

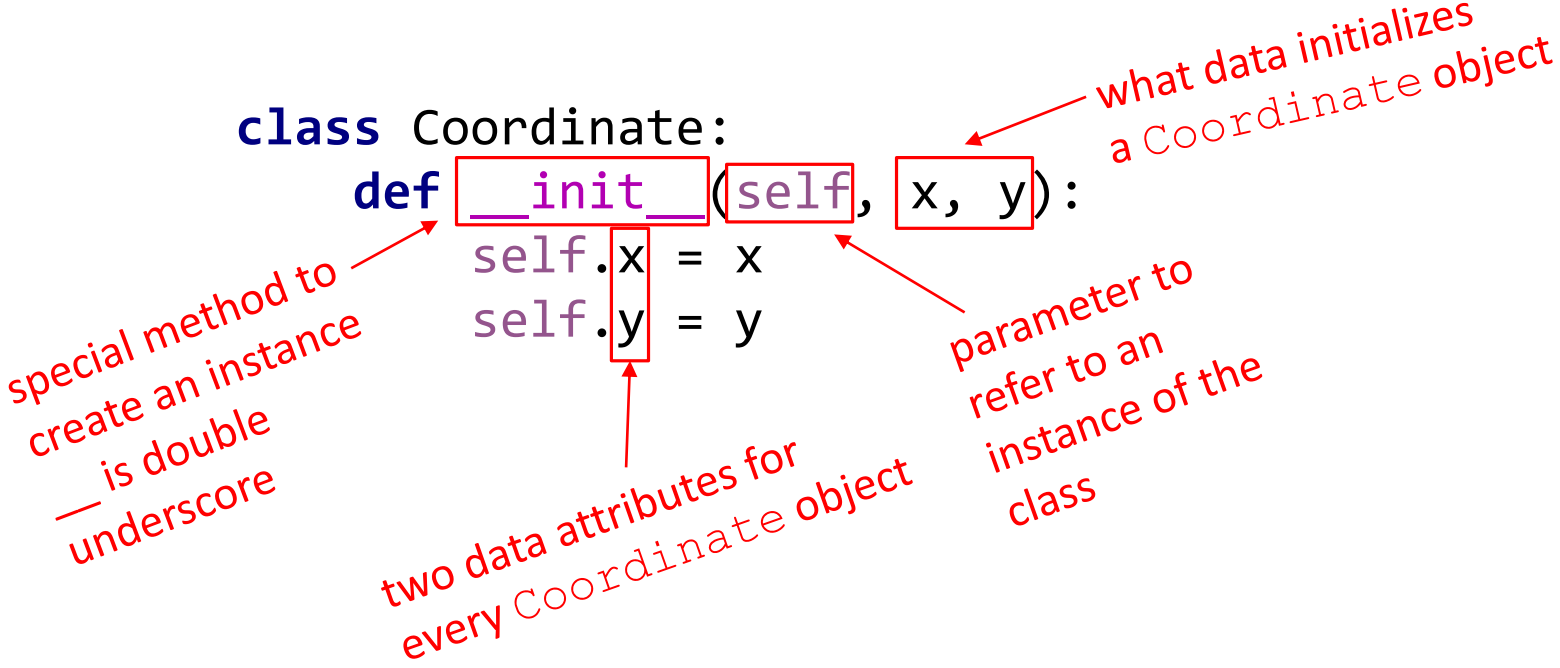
```
class Coordinate:
    # define attributes here
```

WHAT ARE ATTRIBUTES?

- data and procedures that "**belong**" to the class
- **data attributes**
  - think of data as other objects that make up the class
  - for example, a coordinate is made up of two numbers
- **methods**(procedural/behavioral attributes)
  - think of methods as functions that only work with this class
  - how to interact with the object
  - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*

DEFINING HOW TO CREATE AN INSTANCE OF A CLASS

- first have to define **how to create an instance** of object
- use a **special method called __init__** to initialize some data attributes



ACTUALLY CREATING AN INSTANCE OF A CLASS

```
c = Coordinate(3,4)
origin =
Coordinate(0,0)
print(c.x)
print(origin.x)
```

create a new object of type `Coordinate` and pass in 3 and 4 to the `__init__`

used the dot to access an attribute of instance c

- data attributes of an instance are called **instance variables**
- don't provide argument for `self`, Python does this automatically

WHAT IS A METHOD?

- procedural attribute, like a **function that works only with this class**
- Python always passes the object as the first argument
  - convention is to use **self** as the name of the first argument of all methods
- the "**.**"**operator** is used to access any attribute
  - a data attribute of an object
  - a method of an object

DEFINE A METHOD FOR THE `Coordinate` CLASS

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        x_diff_sq= (self.x - other.x)**2
        y_diff_sq= (self.y - other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

use it to refer to any instance

another parameter to method

dot notation to access data

- other than `self` and `dot` notation, methods behave just like functions (take params, do operations, return)

HOW TO USE A METHOD

```
def distance(self, other):
    # code here
```

method def

Using the class:

- conventional way

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(c.distance(zero))
```

object to call method on

name of method

parameters not including `self` (`self` is implied to be `c`)

- equivalent to

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(Coordinate.distance(c, zero))
```

name of class

name of method

parameters, including an object to call the method on, representing `self`

PRINT REPRESENTATION OF AN OBJECT

```
>>> c = Coordinate(3,4)
>>> print(c)
<__main__.Coordinateobject at 0x7fa918510488>
```

- **uninformative** print representation by default
- define a **__str__ method** for a class
- Python calls the __str__ method when used with print on your class object
- you choose what it does! Say that when we print a `Coordinate` object, want to show

```
>>> print(c)
<3,4>
```

DEFINING YOUR OWN PRINT METHOD

```
class Coordinate:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        x_diff_sq= (self.x - other.x)**2
        y_diff_sq= (self.y - other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5

    def __str__(self):
        return "<" + str(self.x) + "," + str(self.y) + ">"
```

name of special method

must return a string

WRAPPING YOUR HEAD AROUND TYPES AND CLASSES

- can ask for the type of an object instance

```
>>> c = Coordinate(3,4)
>>> print(c)
<3,4>
>>> print(type(c))
<class __main__.Coordinate>
```

return of the __str__ method

the type of object c is a class `Coordinate`

- this makes sense since

```
>>> print(Coordinate)
<class __main__.Coordinate>
>>> print(type(Coordinate))
<type 'type'>
```

a `Coordinate` is a class

a `Coordinate` class is a type of object

- use `isinstance()` to check if an object is a `Coordinate`

```
>>> print(isinstance(c, Coordinate))
True
```

SPECIAL OPERATORS

- +, -, ==, <, >, len(), print, and many others

https://docs.python.org/3/reference/datamodel.html#basic-customization

- like `print`, can override these to work with your class
- define them with double underscores before/after

```
__add__(self, other) → self + other
__sub__(self, other) → self -other
__eq__(self, other)  → self == other
__lt__(self, other)  → self < other
__len__(self)        → len(self)
__str__(self)        → print(self)
```

… and others

EXAMPLE: FRACTIONS

- create a **new type** to represent a number as a fraction
- **internal representation** is two integers
  - numerator
  - denominator
- **interface** a.k.a. **methods** a.k.a **how to interact** with Fraction objects
  - add, subtract
  - print representation, convert to a float
  - invert the fraction
- let's write the code for this. The file containing this code will be pushed to course repository for your reference.

STATIC ATTRIBUTES

- **shared** between all the instances of a class, or in other words they are class level attributes rather than instance attributes
- static data attributes are defined in class scope rather than the `__init__` scope

```
class Circle:
    PI = 3.14

    def __init__(self, c, r):
        self.center = c
        self.radius = r
```

a static attribute

instance attributes defined through `self`

- if an instance changes the static attribute, it will changed for all other instances!
- static attributes usually are accessed by using the dot notation on class names:
  - c = Circle((0,0), 2)
  - area = Circle.PI * c.radius**2

STATIC ATTRIBUTES

- like static data attributes, we can have **static methods**
- static methods are also shared between all the instances, and their implementation is **independent** of a specific instance!
- you declare a method as an static method by using a special decorator @staticmethod

```
class FlightSeat:
    counter = 0

    @staticmethod
    def increase_counter():
        FlightSeat.counter += 1
```

no need for `self` as first param

OBJECT ORIENTED PROGRAMMING (OOP)

- **EVERYTHING IN PYTHON IS AN OBJECT**(and has a type)
- can **create new objects** of some type
- can **manipulate objects**
- can **destroy objects**
  - explicitly using **del** or just “forget” about them
  - python system will reclaim destroyed or inaccessible objects –called “garbage collection”

THE POWER OF OOP

- **bundle together objects** that share
  - common attributes and
  - procedures that operate on those attributes
- use **abstraction** to make a distinction between how to implement an object vs how to use the object
- build **layers** of object abstractions that inherit behaviors from other classes of objects (see this next session)
- create our **own classes of objects** on top of Python's basic classes