CS2110 Lecture 22 Mar. 15, 2021

- HW6 will be available tomorrow, due next Thursday, March 25
- DS Assignment will be 6 available this afternoon or tomorrow morning, due this Friday 5pm. Attendance at tomorrow's discussion section is not required.
- Today and Wednesday
  - Classes and objected oriented programming: Chapters 17, 18, and 19

Ch 17, 18, 19. Classes and Object-oriented (OO) programming

- This is a very important topic for modern programming.
  - Many many real-world systems are heavily object-oriented. E.g. to program iOS/iPhone/iPad, you'll have to deal with large complex OO libraries/frameworks
- It's a very big topic.
  - terms like: class, attribute, object, method, instance, inheritance, abstraction, encapsulation, information hiding, polymorphism, …
  - we'll cover the basics

Introduction to Classes

- defs lets us add new functions. Extremely useful for breaking down large program into components, building modules or libraries of computational tools
- classes let us define whole new types. Think of a class as a set of objects (the *instances* of the class)having 1) attributes and,possibly also 2) operations (called methods) defined on them.
  - You are alraedy familiar with types: int, float, Boolean, string, list, tuple, dictionary
  - **with class definitions you can create your own types**. Programs can be much clearer, easier to understand and maintain when written in terms of appropriate types and instances of those types

Instead of using, say, a list or dictionary to represent a person:

```
p = ['jim', 58, 'blue', 'professor']
```

and using basic list operations to extract age

```
p[1]                    # access age
```

define and use a Person class and related attributes and operations

```
p = Person(...)
p.birthdate
p.eyecolor
p.occupation
p.getAge()
p.computeCreditRating()
```

Classes can provide abstraction. We can use objects without knowing details of how data is stored

- documentation tells you how to use objects but doesn't need to tell you implementation details. In fact, the implementation details can be changed without you having to worry about it

```
# assumes personLists1 and 2 have birthdate stored at index 1
def olderThan (personList1, personList2):
    return (personList1[1] < personList2[1])


# assumes person1 and 2 are objects with birthdate attributes
def olderThan(person1, person2):
    return (person1.birthdate < person2.birthdate)


# assumes person1 and 2 are objects with getAge() methods
def olderThan(person1, person2):
    return (person1.getAge() > person2.getAge())
```

In the first example, the olderThan function needs to understand how a person is represented – as a list in which the second element contains the age.

In the second, we need to know that a Person object has a birthdate attribute.

In the third, we only need to know the Person class has a getAge() operation/method defined on it. We don't need to know exactly what attributes are used to represent a Person. *We don't know and don't need to know.*

Basic Python Types and Classes

Basic python types are actually themselves classes.

- list objects are instances of the list class
- the operations defined for a class are called methods.

You've been using methods via the dot notation:

```
[1,2,3].append(4)
```

- Earlier I suggested you think about such methods as strange function call syntax
  [1,2,3].append(4) → append([1,2,3],4)
- That is useful but if try it exactly like that, you'll get an exception

```
>>> append([1,2,3],4)
```

Methods *are indeed* functions – just special ones specific to a class. The list append method is defined *as part of* the definition of the list class.

- Execute help(list) in Python shell to see things defined for the list class
- Turns out you *can* directly call append in "plain" function style, if we use append's "full" name – list.append (the append function owned by the list class)

```
>>> list.append([1,2,3],4)
```

- Similarly, see help(int). + actually shorthand for __add__ method for integers.

```
>>> a = 3
>>>a.__add__(4)
7
```

(more on these __foo__ functions later)

Defining classes

- In Python (and other languages) to define a **class**, you define object **attributes** (also often called **properties**) and the **methods** (operations) that can be invoked on objects (instances) of that class. General form:

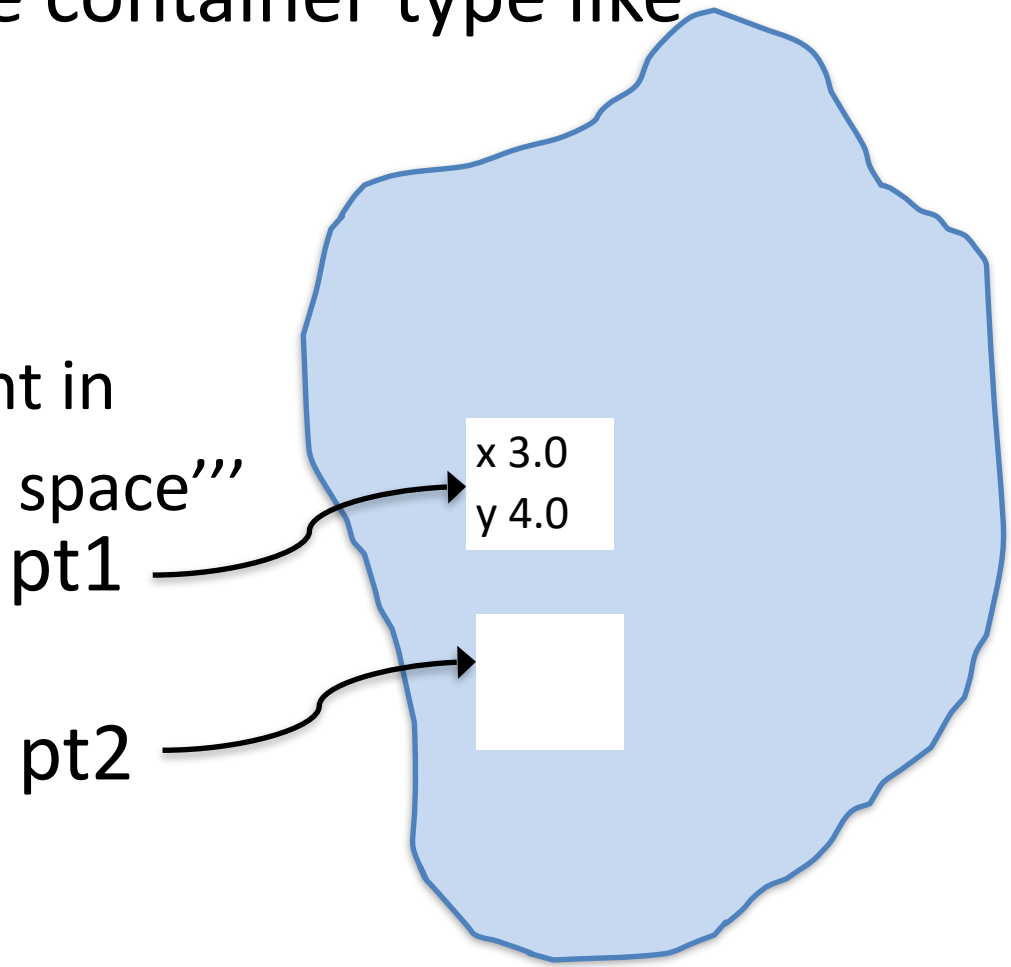
```
class Myclass ():
    classAttribute1 = …
    …

    def method1(self, …):
        self.objectAttribute1 = …
        self.objectAttribute2 = …
        … computation in terms of properties and arguments passed to method…
        return …

    def method2(self, …)
        … computation in terms of properties and arguments passed to method ...
```

- Note: variable name self is a convention (standard practice/usage). The first argument to a method is always the object that invoked the method. It is *legal* to name it anything but please stick to standard practice – use 'self'

One way to use classes: as simple containers of attributes, but without methods. It is useful though most people would not call this "object oriented programming". It's simply using classes as another simple container type like lists and dictionaries.

E.g. >>> class Point:
'''represents a point in 2D space'''

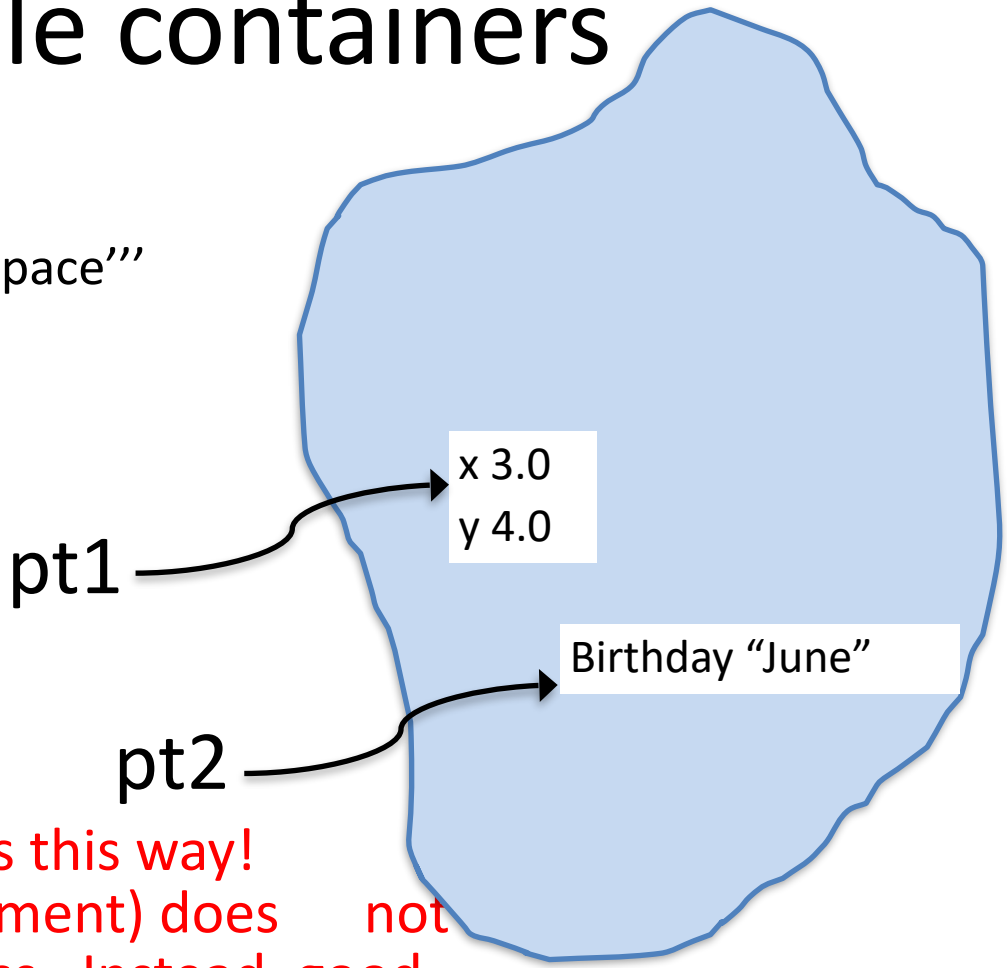
>>> pt1 = Point()
>>> pt1.x = 3.0
>>> pt1.y = 4.0
>>> pt2 = Point()



Classes as simple containers

E.g. >>> class Point:
'''represents a point in
2D space'''

```
>>> pt1 = Point()
>>> pt1.x = 3.0
>>> pt1.y = 4.0
>>> pt2 = Point()
>>> pt2.birthday = "June" ???
```

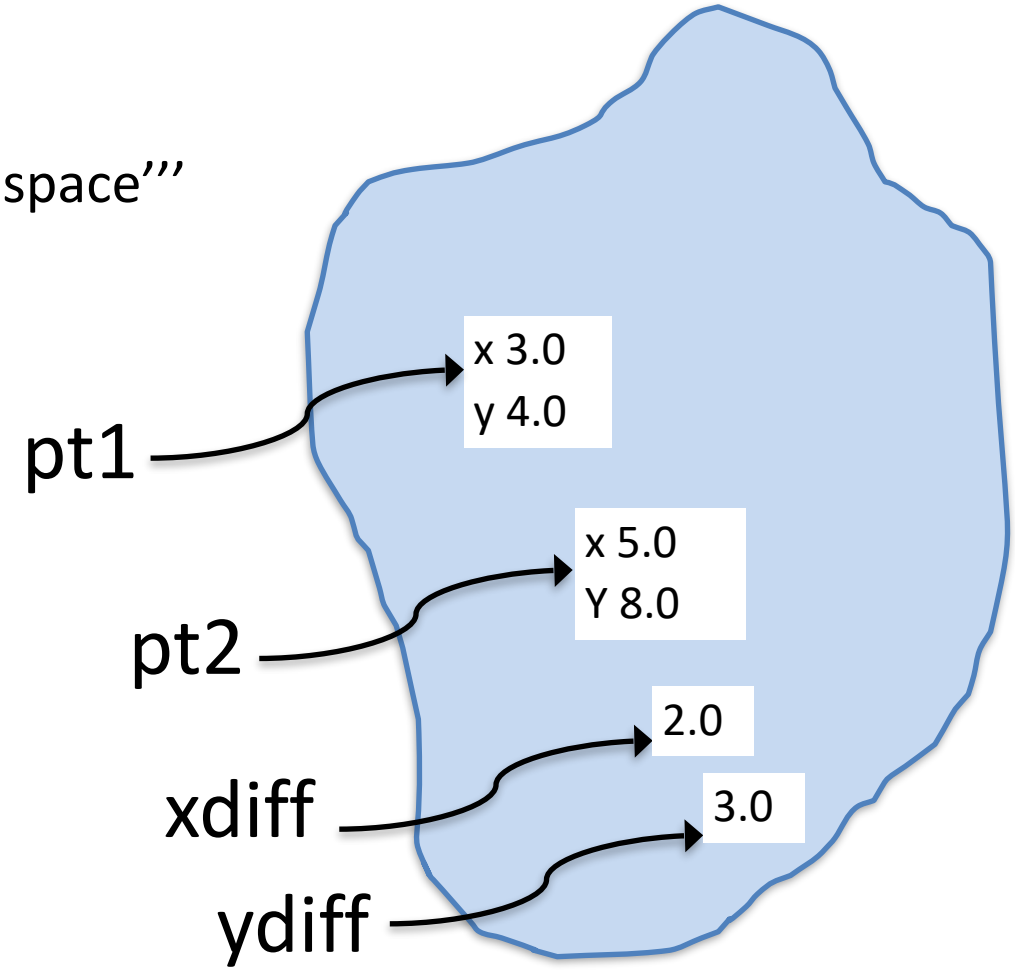


We *usually* don't want to do things this way! empty class definition (plus a comment) does not define the interface/API to the class. Instead, good practice is to define a class via methods that are used to work with instances of the class. And, though legal, it is often considered good practice **not** to access attributes directly (pt1.x) but only via methods ("getters" and "setters") pt1.getX()

Classes as simple containers

E.g. >>> class Point:
'''represents a point in
2D space'''

```
>>> pt1 = Point()
>>> pt1.x = 3.0
>>> pt1.y = 4.0
>>> pt2 = Point()
>>> pt2.x = 5.0
>>> pt2.y = 8.0
>>> xdiff = pt2.x – pt1.x
>>> ydiff = pt2.y – pt1.y
```



Even though not super common to program this way, it is important to know how to access attributes since when you define methods you'll write will access attributes directly.

Classes as simple attribute containers vs. "object oriented programming"

- in what people usually call **object-oriented programming** we don't usually want to do things this way – writing top-level functions that access object attributes directly (point.x, etc.). It's more like using structs in a C
- HOWEVER, it can be useful
- will demonstrate a Time class (time1.py) using this approach and then, after introducing methods, see how we can convert that to the more standard object-oriented style

So … first:

time1.py demonstrates

- a Time class with three attributes
- Several regular functions (not Time class methods) operating on Time objects

Then:

– classes with methods – a more “object oriented way
time2.py time2Alt.py

Classes with methods (the more "object-oriented" way)

- General rule for defining classes:
  - **always** define an __init__ method initializing values for all properties/attributes (e.g. hour, minutes, seconds for Time)
  - define methods that represent the "public interface" to the class. Users should work with instances of the class only via these methods rather than by accessing object attributes directly. First argument to a method is always the object that invokes it. Standard practice is to use variable name 'self'

__init__ methods and “constructors”

```
class Time:
    def __init__ (self, hour = 0, minutes = 0, second = 0):
        self.hour = hour
        self.minutes = minutes
        self.seconds = seconds

>>> t1 = Time(3, 24, 59)
>>> t.hour
3
>>> t.seconds
59
```

HOW DOES THIS WORK??

When you create an object using a “constructor”: e.g. Time(…)

1. Python first creates empty object
2. Passes that empty object to __init__ with any additional arguments provided to constructor
3. returns the new object (even though there is no “return” line in init)

Make things look nice using __repr__ and/or __str__ methods

```
class Time
        def __init__(. . .):
                . . .
        def __repr__(self):
                return "Time({}, {}, {})".format(
                                self.hour, self.minutes, self.seconds)
        def __str__(self):
                ampm = "AM" if self.hour <12 else "PM"
                return "{:02d}:{:02d}:{:02d} {}".format(
                                self.hour%12,self.minutes,self.seconds, ampm)
```

```
>>> t = Time(10,23,59)
>>> t
Time(10,23, 59)
>>> print(t)
10:23:59 AM
>>>  str(t)
“10:23:59 AM”
```

__repr__ and __str__ methods: used to define how object displays or gets converted to string. Many Python programmers don’t know the distinction between the two. You don’t need to know. If you’re only going to define one, define __repr__. However, many people argue that best practice is: __repr__ should produce string that is what you would type in to create object similar object, while __str__ should simply yield a nice “readable” form.

Notes on development of classes

- Look at implementation of
  - incrementTime(self)
  - laterTime(self)

  methods in time2.py. Same basic code as in time1.py but now in OO style. First argument to a method is always object that invokes the method, and standard practice is to use var name 'self'

- Nice feature of classes: you can **overload** operators. That is, you can define how +, -, <, etc. apply to objects of classes that you define
  - __add__ for + (and __radd__)
  - __lt__ for <
  - __eq__ for ==, etc.

  See how these are used in time2.py

Notes on development of classes

- AGAIN, best practice as a user of class is avoid directly accessing object attributes. I.e. when you have a time object t, don't use t.hour. Use only methods. WHY?
- If we only use methods, the class developer can change in the internal representation (maybe to make things more efficient). E.g instead of using three attributes – hour, minutes, seconds - to represent time in the Time class, could just use seconds! Can still make all the methods work the same, print in human friendly form, etc. implementation. See time2Alt.py

Next time

Continue our quick look at object-oriented programming:

- class attributes – not in interactive text (but in 18.3 of pdf of non-interactive text)
- Ch 19 – inheritance