Py ch16-17: Making an ADT the OO Way

2 Nov 2007
CMPT14x
Dr. Sean Ho
Trinity Western University

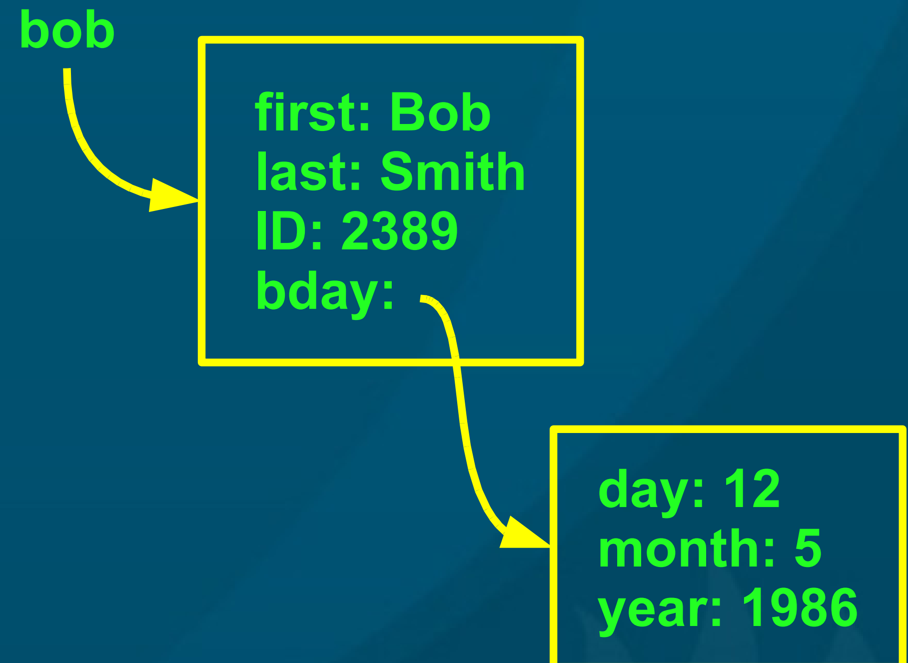
TRINITY WESTERN UNIVERSITY

Classes and instances

- We define (declare) object classes (types)
  - Attributes
  - Methods (interface)
    - Constructor and destructor
- Then we instantiate the class (declare variables)
- e.g., frac1 is a variable of type Fraction
  - frac1 is the instance,
  - Fraction is the class

More on instantiating classes

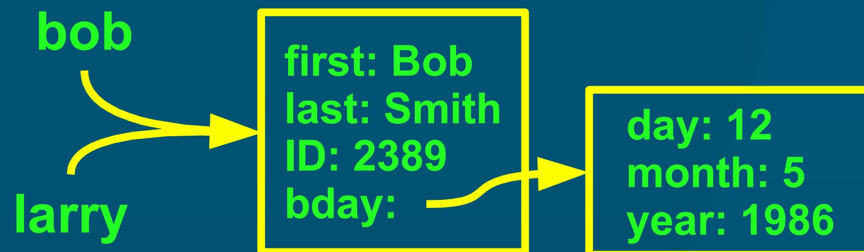
- class Date:
  - day = 0
  - month = 0
  - year = 0
- class StudentRecord:
  - firstName = ""
  - lastName = ""
  - ID = 0
  - birthdate = Date()



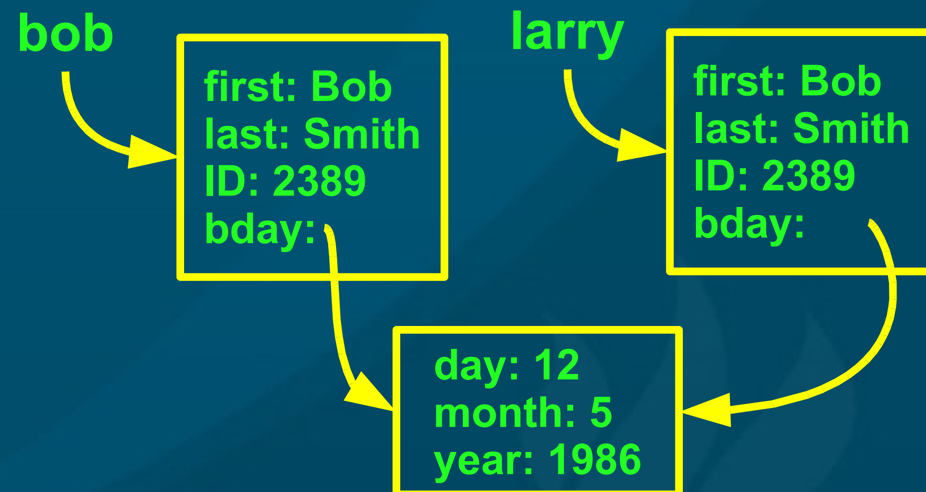
- Creating a new StudentRecord makes a new Date:
  - bob = StudentRecord()
  - bob.birthdate.year = 1986

More on copy vs. alias

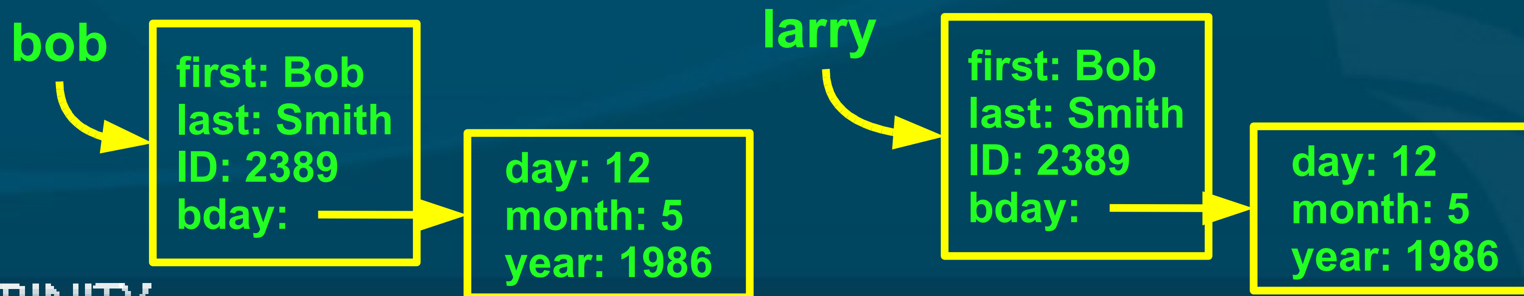
- Assignment: alias
  - larry = bob



- copy.copy(): shallow copy
  - larry = copy.copy(bob)



- copy.deepcopy(): deep copy
  - larry = copy.deepcopy(bob)



Using 'id' to look at aliases

- We can check whether two names are aliases or separate copies by using the Python built-in 'id':
  - id(student1)　　# 11563216
  - student2 = student1　　# alias
  - id(student2)　　# 11563216
  - student2 = copy.deepcopy(student1)　# copy
  - id(student2)　　# 18493888

Creating a list of objects

- Our student db is a list of StudentRecords
- Because of aliasing, we can't use this shortcut:
    - student = StudentRecord()
    - studentDB = [student] * 35
  - A list of 35 aliases to the same object!
- Use a for loop to create separate objects:
    - **studentDB = [0] * 35**
    - **for idx in range(len(studentDB)):**
        - **studentDB[idx] = StudentRecord()**

Creating a Fractions ADT

- In ch6 we sketched a Fractions library
  - Fractions were really tuples
  - Hard to hide that from the user
- OO lets us do fractions the 'right' way:
  - Fractions class:
  - Two attributes: num, denom
  - Methods: add, sub, mul, div
  - Constructor method: calls __init__()

OO: Methods

- In OO, procedures are methods of an object:
  - Messages that can be passed to the object
  - Defined within the class declaration
- First parameter to the method is always a reference to the current object: 'self'

```
class Fraction:
    def __str__(self):
        """A pretty-printed form of the fraction."""
        return "%d / %d" % (num, denom)
```

- __str__ is an example of a customization:
  - Gets called by print

Listing all entities in a class

- Special Python attribute '__dict__'
- Dictionary of all entities in the class
    - import math
    - math.__dict__
  - Lists all functions, constants, etc.
  - Can be very long for some modules!

Creating a new class

- Most class definitions will have __init__ and __str__:

```
class Fraction:
    def __init__(self):
        self.numer = 0
        self.denom = 1
    def __str__(self):
        return '%d / %d' % (self.numer, self.denom)
```

- Refer to instance variables via self.*variable*
- Docstrings for __init__ and __str__ are not usually needed unless something special is happening

Instantiating our new class

- We can now make an instance of our class:
  - f1 = Fraction()
  - f1.numer = 2
  - f2.denom = 3
  - print f1 # 2 / 3
-

Adding a method: multiply()

- Multiply takes two parameters: self, and the other fraction to add.
  - This definition goes inside the class definition:

```
def multiply(self, f2):
    """Multiply two fractions."""
    product = Fraction()
    product.numer = self.numer * f2.numer
    product.denom = self.denom * f2.denom
    return product
```

  - Need to create a new Fraction to return as the result

Using the multiply() method

- We can now multiply two fractions:

```
print f1                # 2 / 3
f2 = Fraction()
f2.numer = 1
f2.denom = 2
print f1.multiply(f2)   # 2 / 6
```

Python customizations

- Certain method names are special in Python:
  - __init__: Called by the constructor when we setup a new instance
  - __str__: Called by print
  - __mul__: Overloads the (*) operator
  - __add__: Overloads the (+) operator
  - __le__: Overloads the (<) operator
  - etc. (pretty much any operator can be overloaded!)
- http://docs.python.org/ref/specialnames.html

Using customizations

- So if we name our multiply() method __mul__() instead, we can do:
    - print f1 # 2 / 3
    - print f2 # 1 / 2
    - print f1 * f2 # 2 / 6

Parameters to the constructor

- We can pass parameters to the constructor:
  - f1 = Fraction(2,3)
- We just need to extend the __init__ function to accept more parameters:
  - def __init__(self, n, d):
    - self.numer = n
    - self.denom = d

Default parameters

- Python functions can specify defaults for the tail-end parameters:
    - def __init__(self, n=0, d=1):
        - self.numer = n
        - self.denom = d
- If __init__ is called with no parameters, n=0 d=1
- If __init__ is called with one parameter:
    - n is given and d=1
- If __init__ is called with two parameters:
    - both n and d are given.

TODO items

- HW07 due Mon: ch9 #6, 8, 16 (in Python).
- Lab07 due next Wed: ch9 (choose one):
  - #37+38: people db, matching
  - #40+41: online chequebook
  - #46: church directory
- Paper topic due next week Fri 9Nov
- Lab10 due last week of classes:
  - Choose one from Lab04-07, do in M2