§9.0-9.9: Sets and Records

29 Oct 2008
CMPT14x
Dr. Sean Ho
Trinity Western University

Set operations

- A set is an unordered collection of items
- Set membership: test if an item is in the set
- Set union: A ∪ B:
  - Anything that's in either A or B
- Set intersection: A ∩ B:
  - Those items which are in both A and B
- Set difference: A – B (or A \ B):
  - Those in A but not in B
- Set symmetric difference: A ^ B:
  - Those in exactly one of A or B

Sets in Python

- Python has a built-in type for sets (as does M2):
  - Instantiate with any iterable (e.g., a list):

    ```
    bagOfApples = set( [ 'Fuji', 'Gala', 'Red Delicious' ] )
    ```

  - Add an apple to the bag:

    ```
    bagOfApples.add( 'Rome' )
    ```

  - Remove an existing apple from the bag:

    ```
    bagOfApples.remove( 'Rome' )
    ```

  - Check if an apple is in the bag:

    ```
    if 'Fuji' in bagofApples:
    ```

- See Python documentation: http://docs.python.org/lib/types-set.html

Python set operators

- Operators for Python sets:
  - Union of two sets: .union() or |

    bagOfApples.union(yourApples)

    bagOfApples | yourApples
  - Intersection of two sets: .intersection() or &
  - Difference of two sets: .difference() or –
  - Symmetric difference: .symmetric_difference() or ^
  - Subset: .issubset() or <=
    - A <= B: everything in A is also in B
  - Superset: .issuperset() or >=

Bitsets

- Another way to use sets in Python is to use the binary form of an integer to represent flags:
  - e.g., file permissions

```
readFlag = 1 << 2
writeFlag = 1 << 1
execFlag = 1 << 0
myPerms = readFlag | writeFlag   # both read/write

if myPerms & readFlag:           # have read perm
```

- myPerms is called a bitset: it is a compact way of representing a set

Records

- Say we want to create a student info database:
  - First name
  - Last name
  - Student ID #
  - Year
- How do we store this?
  - Four separate lists:
    - firstNames = ['Tom', 'Alan', 'Yuri', 'Megan', ...]
    - studentID = [38, 28, 10, 49, ...]
  - Or one list of student records

User-defined types

- A record is a user-defined aggregate type:
  - Define a StudentRecord type as:
    - First name (string)
    - Last name (string)
    - Student ID (integer)
    - Year (integer between 1 and 4)
- Then we can store the whole database in one list, where each entry of the list has type StudentRecord.

Records in M2

- We define a record type in M2 like this:

```
TYPE
    StudentRecord =
        RECORD
            firstname : ARRAY [0 .. 255] OF CHAR;
            lastname : ARRAY [0 .. 255] OF CHAR;
            ID : CARDINAL;
            year : CARDINAL;
        END;
```

- Declare and initialize a new student:

```
VAR
    student1 : StudentRecord;
student1.firstname := “Joe”;
```

Records in Python: Classes

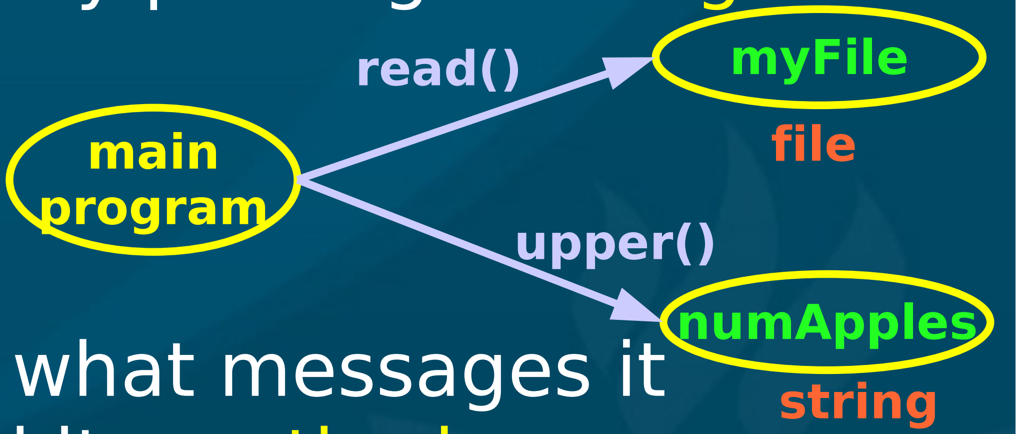
- In Python, classes are user-defined types:
    - class StudentRecord:
        - def __init__(self):
            - self.firstName = ""
            - self.lastName = ""
            - self.ID = 0
            - self.year = 0
- Instantiate a new object of type StudentRecord:
    - student1 = StudentRecord()
    - student1.firstName = 'Tom'
- student1 is an instance of the class StudentRecord
    - “x is a variable of type int”

Object-oriented programming

- Procedural paradigm: programs as lists of actions
  - Focus is on the procedures (verbs)
  - Variables, data structures get passed into procedures
    - **e.g.: string.upper('hello')**
- Object-oriented paradigm: collections of objects
  - Focus is on the data (nouns)
  - Messages get passed between objects
  - Procedures are methods belonging to objects
    - **e.g.: 'hello'.upper()**

Everything is an object

- In object-oriented programming, all data are objects:
  - Variables, procedures, even libraries
- We make things happen by passing messages between objects
  - myFile.read(16)
  - appleName.upper()



- The object itself defines what messages it accepts: these are called its methods
  - e.g., files have read(), write(), etc. strings have upper(), len(), etc.

Methods and attributes

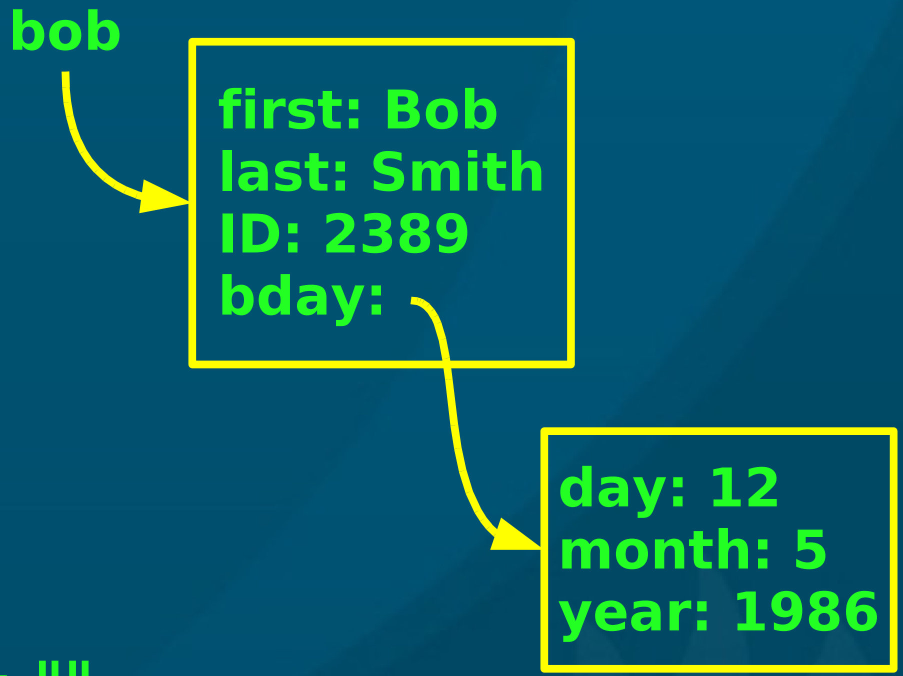
- Everything you can do with an object is encapsulated in its object definition
  - Methods make up the interface to the object
- Objects can also have attributes (variables)
- Our fractions.py ADT example:
  - Methods: get_n(), get_d(), add(), mult(), etc.
    - Everything you need to interact with a Fraction
  - Attributes: tuple (n,d)
    - Could also have two separate attributes: num, denom

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:
  - def __init__(self):
    - self.day = 0
    - self.month = 0
    - self.year = 0
- class StudentRecord:
  - def __init__(self):
    - self.firstName = ""
    - self.lastName = ""
    - self.birthdate = Date()



bob

first: Bob
last: Smith
ID: 2389
bday:

day: 12
month: 5
year: 1986

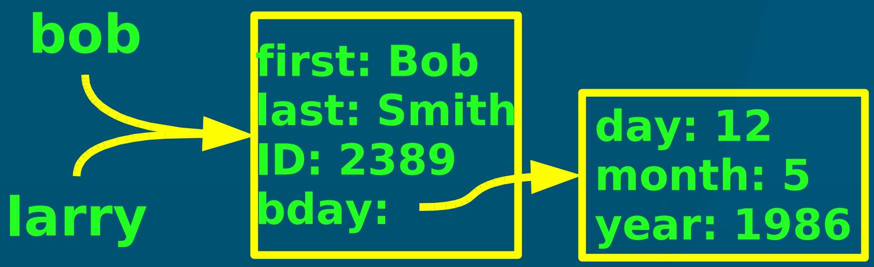
- Creating a new StudentRecord makes a new Date:
  - bob = StudentRecord()
  - bob.birthdate.year = 1986

Copy vs. alias for objects

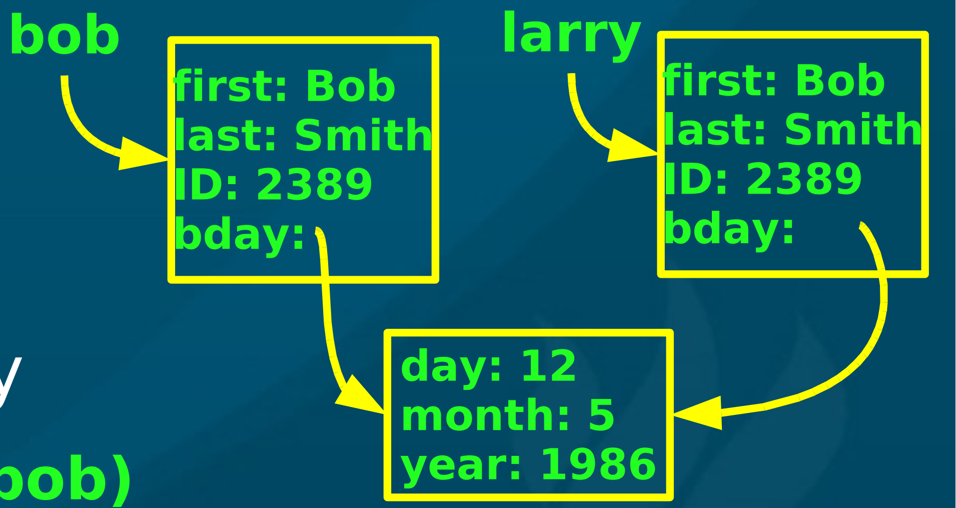
- Objects are mutable:
  - student1.ID = 25
  - student1.ID = 38
- This means assignment is just aliasing:
  - student2 = student1
  - student2.ID = 50 # affects student1.ID
- To make a separate copy, use copy.deepcopy():
  - import copy
  - student2 = copy.deepcopy(student1)
- Or create a new instance, and copy values:
  - student2 = StudentRecord()
  - student2.ID = student1.ID

More on copy vs. alias

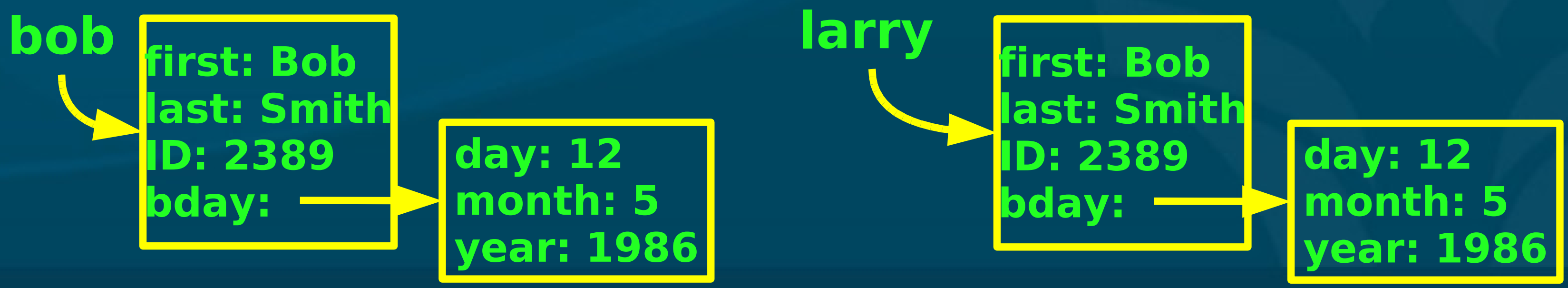
- Assignment: alias
  - **larry = bob**



- 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()**