CSc 120
Introduction to Computer Programming II

Adapted from slides by
Dr. Saumya Debray

04: Basics of Object-Oriented Programming

Programming paradigms

- Procedural programming:
  - programs are decomposed into procedures (functions) that manipulate a collection of data structures
- Object-oriented programming
  - programs are composed of interacting entities (objects) that encapsulate data and code

Object-oriented programming

Informally:

"Instead of a bit-grinding processor plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires."

-Dan Ingalls

What is an object?

To human beings, an object is:

"A tangible and/or visible thing; or

(a computer, a chair, a noise)

Something that may be apprehended intellectually; or

(the intersection of two sets, a disagreement)

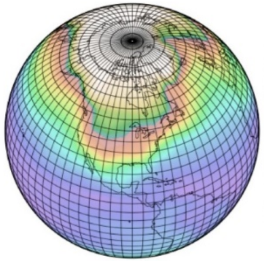
Something towards which thought or action is directed"

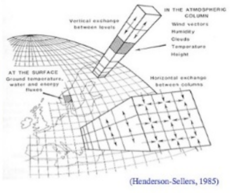
(the procedure of planting a tree)

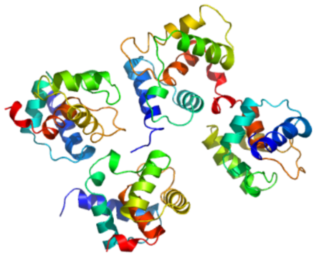
-Grady Booch

Objects

- Object-oriented programming models properties of, and interactions between, entities in the world







Objects

- Objects have state and behavior
  - the state of an object can influence its behavior
  - the behavior of an object can change its state

- State:
  - all the properties of an object and the values of those properties

- Behavior:

  how an object acts and reacts, in terms of changes in state and interaction with other objects

Object: An entity that combines state and behavior

The Class concept

- Class:

  A set of objects having the same behavior and underlying structure

- A class is a template for defining a new type of object

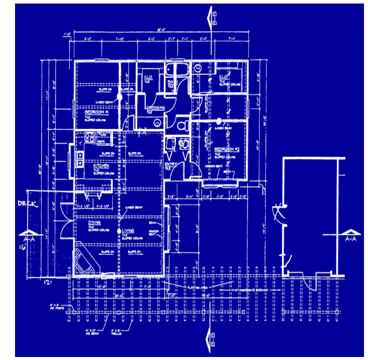
An object is an instance of a class.

Blueprint analogy

iPod blueprint

state:
current song
volume
battery life

behavior:
power on/off
change station/song
change volume
choose random song



used to create instances of an iPod

iPod #1

state:
song = "1,000,000 Miles"
volume = 17
battery life = 2.5 hrs

behavior:
power on/off
change station/song
change volume
choose random song

iPod #2

state:
song = "Letting You"
volume = 9
battery life = 3.41 hrs

behavior:
power on/off
change station/song
change volume
choose random song

iPod #3

state:
song = "Discipline"
volume = 24
battery life = 1.8 hrs

behavior:
power on/off
change station/song
change volume
choose random song

Classes

- In Python, that blueprint is expressed by a class definition

- A *class* describes the state and behavior of similar objects

- The *attributes* of a class represent the state of an instance

- The *methods* of a class describe the behavior

Example: a set of students at UA

Name	ID	Major	Year	Grades
Alice	012	CS	Freshman	CSC 110: B; CSC 120: A
Bob	025	Physics	Junior	GEO 215: B; Phys 120: C; GEO 325: A
Charlie	101	Music	Senior	MUS 210: B; MUS 423: A; CSC 110: B

Example: a set of students at UA

Name	ID	Major	Year	Grades
Alice	012	CS	Freshman	CSC 110: B; CSC 120: A
Bob	025	Physics	Junior	GEO 215: B; Phys 120: C; GEO 325: A
Charlie	101	Music	Senior	MUS 210: B; MUS 423: A; CSC 110: B

Object-oriented representation

Name	Alice
ID	012
Major	CS
Year	Freshman
Grades	…

Name	Bob
ID	025
Major	Physics
Year	Junior
Grades	…

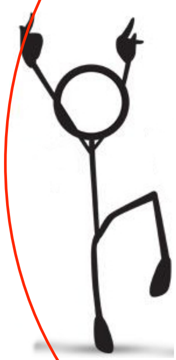
Name	Charlie
ID	101
Major	Music
Year	Senior
Grades	…

Example: a set of students at UA

Objects

Name	Alice
ID	012
Major	CS
Year	Freshman
Grades	…

Name	Bob
ID	025
Major	Geosciences
Year	Junior
Grades	…

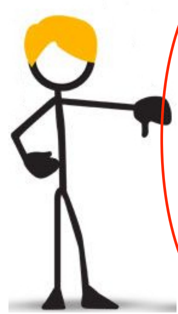


Name	Charlie
ID	101
Major	Music
Year	Senior
Grades	…

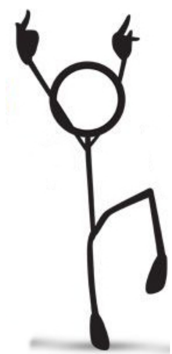
Example: a set of students at UA

Attributes
or
Instance variables

Name	Alice
ID	012
Major	CS
Year	Freshman
Grades	…



Name	Bob
ID	025
Major	Geosciences
Year	Junior
Grades	…



Name	Charlie
ID	101
Major	Music
Year	Senior
Grades	…

Example: a set of students at UA

Class

Name	
ID	
Major	
Year	
Grades	

Name	Alice
ID	012
Major	CS
Year	Freshman
Grades	…

Name	Bob
ID	025
Major	Geosciences
Year	Junior
Grades	…

Name	Charlie
ID	101
Major	Music
Year	Senior
Grades	…

Example: a set of students at UA

Class

Name	
ID	
Major	
Year	
Grades	

Instances of the class

Name	Alice
ID	012
Major	CS
Year	Freshman
Grades	…

Name	Bob
ID	025
Major	Geosciences
Year	Junior
Grades	…

Name	Charlie
ID	101
Major	Music
Year	Senior
Grades	…

Objects

- An *object* consists of:
  - a state
    - given by the values of its *instance variables*
  - a set of behaviors
    - given by its *methods* (e.g., accessing/modifying its instance variables)

- An object models an entity in a real or virtual world or system

Example: Student object

methods:
- like functions
- they look at and/or modify the instance variables of the object

instance variables

- name
- id
- major
- year
- grades

methods

- get_name(), set_name()
- get_id(), set_id()
- get_major(), set_major()
- get_year(), set_year()
- get_grades(), add_grade()
- update_grade()
- compute_GPA()

Name	Alice
ID	012
Major	CS
Year	Freshman
Grades	…

Classes

- A *class* describes the state and behaviors of a set of similar objects
  - state: given by instance variables
  - behaviors: given by the methods of the class

- These objects are *instances* of the class

Example: Student class

```
class Student:
    def __init__(self, name, id):
        self._name = name
        self._id = id

    def get_name(self):
        return self._name

    …
```

Example: Student class

```
class Student:
    def __init__(self, name, id):
        self._name = name
        self._id = id

    def get_name(self):
        return self._name
    …
```

The keyword **class** defines a class

Example: Student class

```
class Student:
    def __init__(self, name, id):
        self._name = name
        self._id = id

    def get_name(self):
        return self._name

    ...
```

indented **def**s define the methods of the class

the first non-indented line ends the class definition

Example: Student class

```
class Student:
    def __init__(self, name, id):
        self._name = name
        self._id = id

    def get_name(self):
        return self._name

    ...
```

the first argument of each method (**self**) denotes the object being referred to

by convention this argument is written 'self' — this is recommended but not mandatory

Example: Student class

```
class Student:
    def __init__(self, name, id):
        self._name = name
        self._id = id

    def get_name(self):
        return self._name
    …
```

the __init__(…) method is special:

- called when an object is created (right after its creation)
- used to initialize the object's instance variables
- the initial values are supplied as arguments to __init__(…)

Example: Student class

```
class Student:
    def __init__(self, name, id):
        self._name = name
        self._id = id

    def get_name(self):
        return self._name
    ...
```

instance variables

_name

_id

These refer to attributes of the <u>object</u> being referred to, and so are written

self._name

self._id

Example: using the Student class

```
class Student:
    def __init__(self, name, id):
        self._name = name
        self._id = id

    def get_name(self):
        return self._name

    …
```

- creating a new Student object:

  ```
  s = Student('Dennis', '543')
  ```

- invoking a method:

  ```
  name = s.get_name()
  ```

Note: **self** (the object reference) is not explicitly specified when using the object

Example: using the Student class

```
class Student:
    def __init__(self, name, id):
        self._name = name
        self._id = id

    def get_name(self):
      return self._name

    ...
```

```
def main():
    infile = get_input_file()
    student_list = [ ]
    for line in infile:
        (name, id, major, year) = parse_student_info(line)
        student = Student(name, id)
        student_list.append(student)
        student.set_major(major)
        student.set_year(year)
        ...
```

Example: using the Student class

```
class Student:
    def __init__(self, name, id):
        self._name = name
        self._id = id

    def get_name(self):
        return self._name
    ...
```

```
def main():
    infile = get_input_file()
    student_list = [ ]
    for line in infile:
        (name, id, major, year) = parse_student_info(line)
        student = Student(name, id)
        student_list.append(student)
        student.set_major(major)
        student.set_year(year)
        ...
```

create a new Student object

Example: using the Student class

```
def main():
    infile = get_input_file()
    student_list = [ ]
    for line in infile:
        (name, id, major, year) = parse_student_info(line)
        student = Student(name, id)
        student_list.append(student)
        student.set_major(major)
        student.set_year(year)
        …
```

```
class Student:
    def __init__(self, name, id):
        self._name = name
        self._id = id

    def get_name(self):
        return self._name
    …
```

create a new Student object

add this student to the list of students

Example: using the Student class

```
def main():
    infile = get_input_file()
    student_list = [ ]
    for line in infile:
        (name, id, major, year) = parse_student_info(line)
        student = Student(name, id)
        student_list.append(student)
        student.set_major(major)
        student.set_year(year)
        …
```

```
class Student:
    def __init__(self, name, id):
        self._name = name
        self._id = id

    def get_name(self):
      return self._name
    …
```

create a new Student object

add this student to the list of students

set other attributes

Example: A tally counter

Has a name.

Starts a counter at zero.

Increments the counter on a click.

Suppose we want to define a class for a *Counter*:

- Data: ???
  - *what data might we want to associate with a Counter?*
- Methods: ???
  - *what methods are required for Counter objects?*

Example: A tally counter


```
class Counter:
    def __init__(self, name):
        self._name = name
        self._count = 0

    def click(self):
        self._count += 1

    def count(self):
        return self._count
```

EXERCISE

Add a reset() method that will set the count to zero.

```
class Counter:
    def __init__(self, name):
        self._name = name
        self._count = 0

    def click(self):
        self._count += 1
    ....
```

EXERCISE

Add a get_reset_count() method that returns the number of times the counter has been reset.


```
class Counter:
    def __init__(self, name):
        self._name = name
        self._count = 0

    def click(self):
        self._count += 1
    ....
```

class Student : Other definitions

More initialization

```
class Student:
    def __init__(self, name, id, major, year):
        self._name = name
        self._id = id
        self._major = major
        self._year = year
...
def main():
    ...
    student = Student(name, id, major, year)
```

Less initialization

```
class Student:
    def __init__(self):
        self._name = ''
        self._id = ''


def main():
    ...
    student = Student()
    student.set_name(name)
    student.set_id(id)
    ...
```

class Student : Other definitions

More initialization

```
class Student:
    def __init__(self, name, id, major, year):
        self._name = name
        self._id = id
        self._major = major
        self._year = year
...
def main():
```

Less initialization

```
class Student:
    def __init__(self):
        self._name = ''
        self._id = ''


def main():
    ...
    student = Student()
    student.set_name(name)
    student.set_id(id)
    ...
```

Typically, it's better to let each class **handle its own internal details.**

Avoid letting the outside world know about the internals of the class.

This is **encapsulation**.

class Student : Other definitions

More initialization

```
class Student:
   def __init__(self, name, id, major, year):
      self._name = name
      self._id = id
      self._major = major
      self._year = year
...
def main():
```

If details have to be handled by the outside world, it **increases the complexity** of the program.

It makes it harder to **change the implementation later.**

Less initialization

```
class Student:
   def __init__(self):
      self._name = ''
      self._id = ''


def main():
   ...
   student = Student()
   student.set_name(name)
   student.set_id(id)
   ...
```

class Student : Other definitions

More initialization

```
class Student:
    def __init__(self, name, id, major, year):
        self._name = name
        self._id = id
        self._major = major
        self._year = year
…
def main():
…
    student = Student(name, id, major, year)
```

A good class (like a good function) facilitates thinking abstractly.

Note to C programmers: Don't think of this as a struct with 4 fields.

This expression causes an instance of the class Student to be created.

EXERCISE

The "+" key on the keyboard is broken. Implement Counter using another means to keep track of the count.


```
class Counter:
    def __init__(self, name):
        self._name = name
        self._count = ?

    def click(self):
        self._count = ??
    ....
```

EXERCISE

Suppose we want to define a class for a *Point*:

- Data: ???
  - *what data might we want to associate with point objects?*

- Methods: ???
  - *what methods might we want to associate with point objects?*

EXERCISE

Write a method `translate` *that changes a Point's location by a given dx, dy amount.*

Write a method `distance _from _origin` *that returns the distance between a Point and the origin, (0,0).*

Use the formula:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Encapsulation

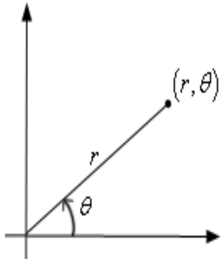
- **encapsulation**: Hiding implementation details of an object from the users of that object.
  - Encapsulation provides *abstraction*.
    - separates external view (behavior) from internal view (state)
  - Encapsulation protects the integrity of an object's data.

Benefits of encapsulation

- Provides abstraction between an object and users of the object.

- Protects an object from unwanted access by code outside the class.
  - A bank app forbids a client to change an `Account`'s balance.

- Allows you to change the class implementation.
  - `Point` could be rewritten to use polar coordinates (radius *r*, angle ϑ), but with the same methods.



- Allows you to constrain objects' state (**invariants**).
  - Example: Only allow `Points` with non-negative coordinates.

Public and private attributes

- Some languages allow the *visibility* of attributes to be
  - **public** : visible to all code

  or
  - **private** : visible only within the class†

- Our practice is to only use private attributes to enforce encapsulation

† The Python*ic* convention is that "_" at the beginning of an attribute name denotes that it is "private"

Class attribute naming conventions

one leading underscore self._var1	Indicates that the attribute is "not public" and should only be accessed by the class's internals (convention; not enforced)
one trailing underscore self.var1_	Used to avoid conflicts with Python keywords, e.g., list_, class_, dict_
two leading underscores self.__var1	Invokes *name mangling*: from outside the class to enforce private e.g., self.__var1 appears to be at YourClassName._YourClassName__var1
two leading + trailing underscores self.__var1__	Intended only for names that have special significance for Python, e.g., __init__

Classic methods styles

- getter and setter methods
  - used to access (getter methods) and modify (setter methods) a class's private variables

- helper methods
  - methods that help other methods perform their tasks

Methods vs. functions

Functions	Methods
• Not associated with any class or object – invoked by name alone • Arguments passed explicitly • Operates on data passed to it	• Associated with a class or object – invoked by object.name • The object for which it was called is passed implicitly • Can operate on data contained within the class

Printing out objects

```
>>> class Student:
        def __init__(self, name, id):
            self._name = name
            self._id = id

>>> s1 = Student('Pat', '623')
>>>
>>> print(s1)
<__main__.Student object at 0x10238b9e8>
>>>
```

- In general, the Python system doesn't know how to print user-defined objects
  - inconvenient
- Ideally, each object (or class) should be able to determine how it is printed

Printing out objects: __str__()

```
>>> class Student:
        def __init__(self, name, id):
            self._name = name
            self._id = id

        def __str__(self):
            return "Student_" + self._name + ":" + self._id
```

- __str__() : a special method for constructing a string from an object

```
>>> s1 = Student('Pat', '623')
>>> print(s1)
Student_Pat:623
>>>
```

- called by str() and print() to convert objects to strings

Special methods: __repr__

- Returns a string
  - the "official" string representation of the object
  - must look like a valid Python expression

- __repr__(obj):
  - If possible, this string should look like an expression that, when evaluated (using **eval()**), would create obj
  - otherwise, should provide a useful description for obj:

    <*...some useful information...*>

angle brackets

Special methods: __repr__

Example:

class:	Student
attributes:	name id major grades

```
def __str__(self):
    return "Student_" + self._name + ": " + self._id
```

__str(self)__
called by str(*obj*)

```
def __repr__(self):
    return "< name :" + self._name  +
                + ", id :" + self._id \
                + ", major: " + self._major \
                + ", grades: '" + str(self._grades) + ">"
```

__repr(self)__
called by repr(*obj*)

__repr__ vs. __str__

- __str__ : aims to be *readable*
  - "unofficial" string representation of an object
  - used by the end user, e.g., for printing out the object
  - not intended to be unambiguous
    - E.g.: str("3") == str(3)

- __repr__ : aims to be *unambiguous*
  - "official" string representation of an object
    - can contain detailed internal information about the object
  - used by the developer, e.g., for debugging or logging
  - used for "unofficial" representation if the class defines __repr__() but not __str()__

Special methods: __eq__

- When are two objects equal?
  - students (people): the name alone may not be enough
  - dictionaries, sets: order of elements unimportant
  - In general: depends on what the object denotes (i.e., its class)

- Python provides special methods __eq__() and __ne__() for this
  - a class can define its own __eq__() and __ne__() methods to define equality

Special methods: __eq__

Example:

```
class Student:
    def __init__(self, name, id):
        self._name = name
        self._id = id

    def __eq__(self, other):
        return self._name == other._name \
                and self._id == other._id
    …
```

Special methods: __eq__

File Edit Shell Debug Options Window Help

```
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "copyright", "credits" or "license()" for more
information.
>>> class Student:
        def __init__(self, name, id):
                self._name = name
                self._id = id
        def __eq__(self, other):
                return self._name == other._name \
                       and self._id == other._id

>>> s1 = Student('John', '123')
>>> s2 = Student('John', '456')
>>> s3 = Student('John', '123')
>>> s1 == s2
False
>>> s1 == s3
True
>>> 
```

Ln: 19 Col: 4

== on the objects calls the __eq__() method of the class

Special methods: rich comparison

__eq__() is an example of a *rich comparison* method:

Comparison operator	Method called
==	__eq__()
!=	__ne__()
<	__lt__()
<=	__le__()
>	__gt__()
>=	__ge__()

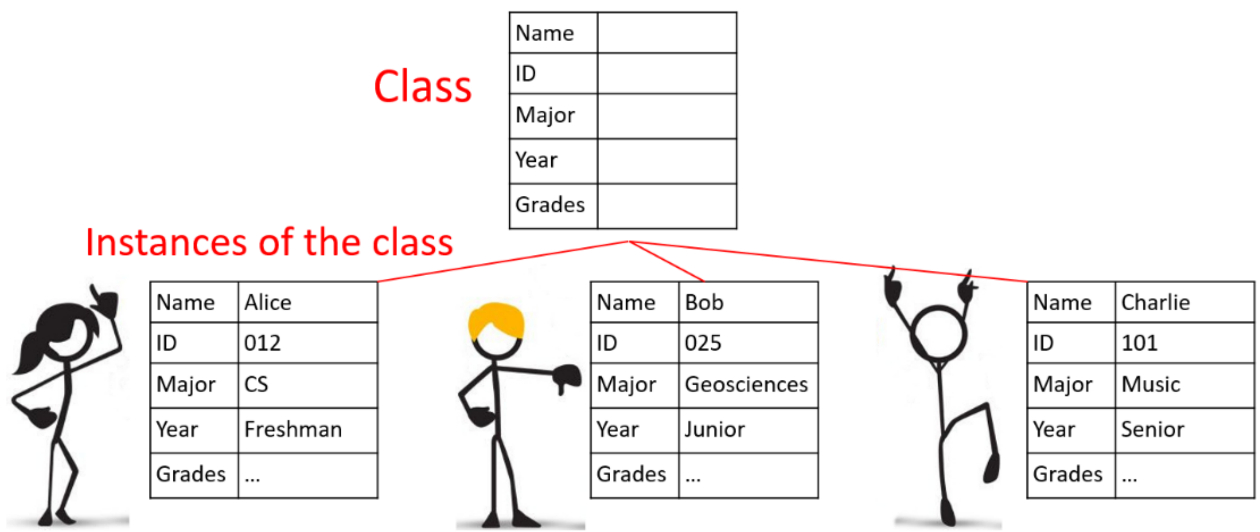
Special methods: __len__ __contains__

For a class that acts like a collection of items:

You want…	**You write…**	**And Python calls…**
the no. of items in the object s	len(s)	s.__len__()
whether the object s contains an item x	x in s	s.__contains__(x)

Summary: Class

- A class is a blueprint, or template, for the code and data associated with a collection of objects
  - the objects are *instances* of the class



Summary: Instance variables

- A variable associated with an object
  - specifies some property of that object
  - each object has its own copy of the instance variables
    - so updating one object's instance variables does not affect other objects

Name	Alice
ID	012
Major	CS
Year	Freshman
Grades	…

- Examples: Name, ID, Major, etc. of a student object

Summary: Methods

- Methods are pieces of code associated with a class (and instances of that class, i.e., objects)
  - they define the behaviors for these objects

- Examples:
  - getters: get_name(), get_id(), …
  - setters: set_name(), set_id(), …
  - special methods: __init__(), __str__(), …

Class Point

```
import math
class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y
    def translate(self, dx, dy):
        self._x = self._x + dx
        self._y = self._y + dy
    def  distance_from_origin(self):
        return math.sqrt(self._x**2+ self._y **2)
```