

CSc 120

Introduction to Computer Programming II

02: 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

What is an object?

To human beings, an object is:

"A tangible and/or visible thing; or
(a computer, a chair, a noise)

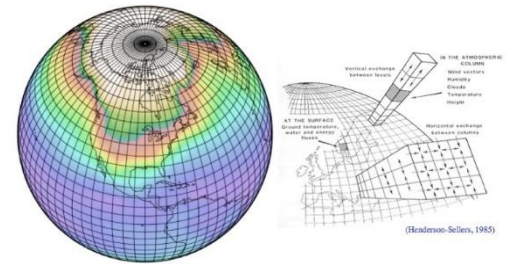
Something that may be apprehended intellectually; or
(the intersection of two sets, a disagreement)

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
- What are some properties of Angry Birds?
- How do they interact?
- What about physical locations on the planet?



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

EXERCISE (Whiteboard)

Consider an ipod:

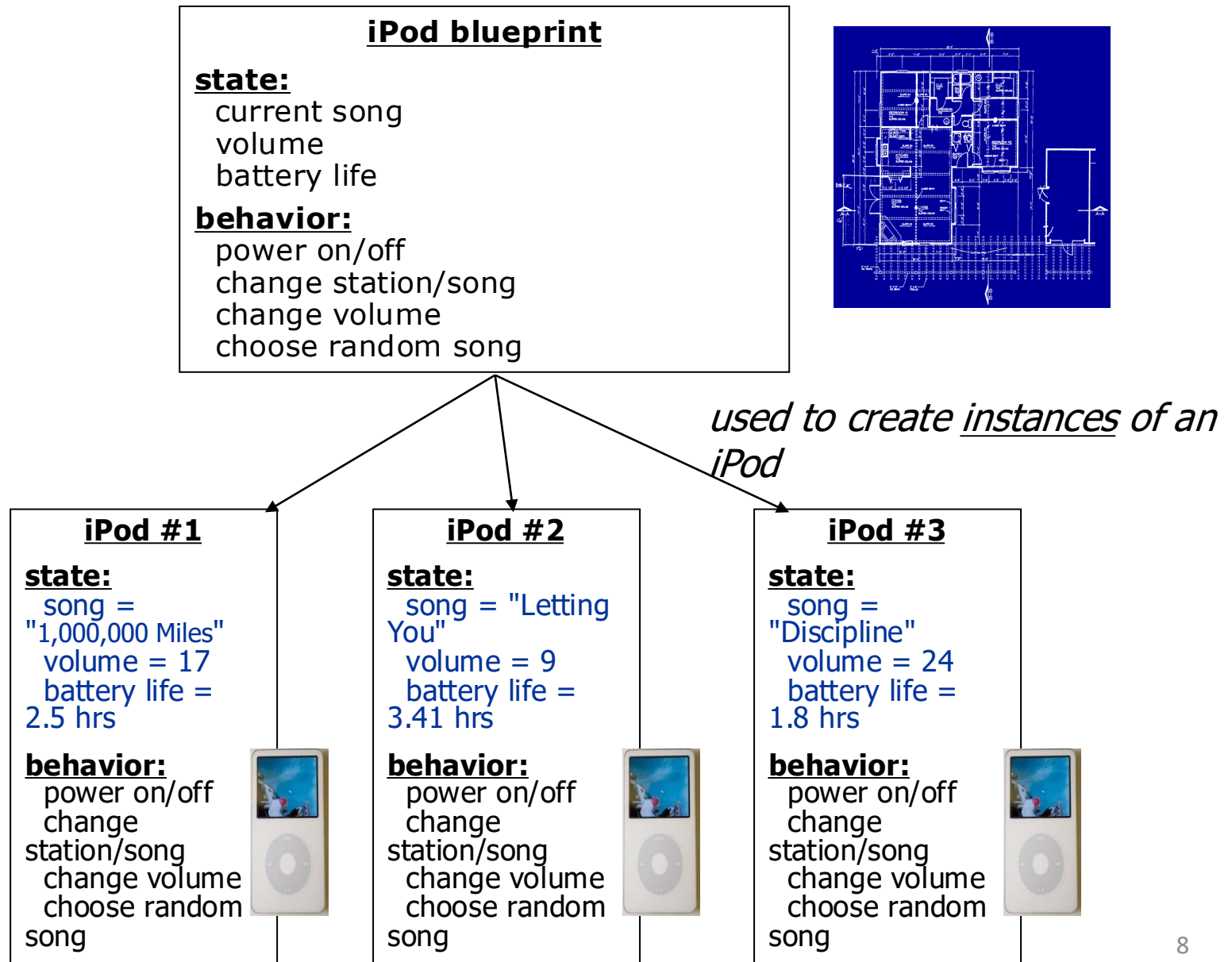
- State (properties):
 - *What properties does an ipod have?*
- Behavior (operations):
 - *What does an ipod do?*
 - *What operations could we define for an ipod?*

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



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

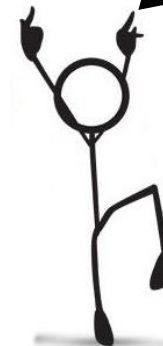
Object-oriented representation



Name	Alice
ID	012
Major	CS
Year	Freshman
Grades	...



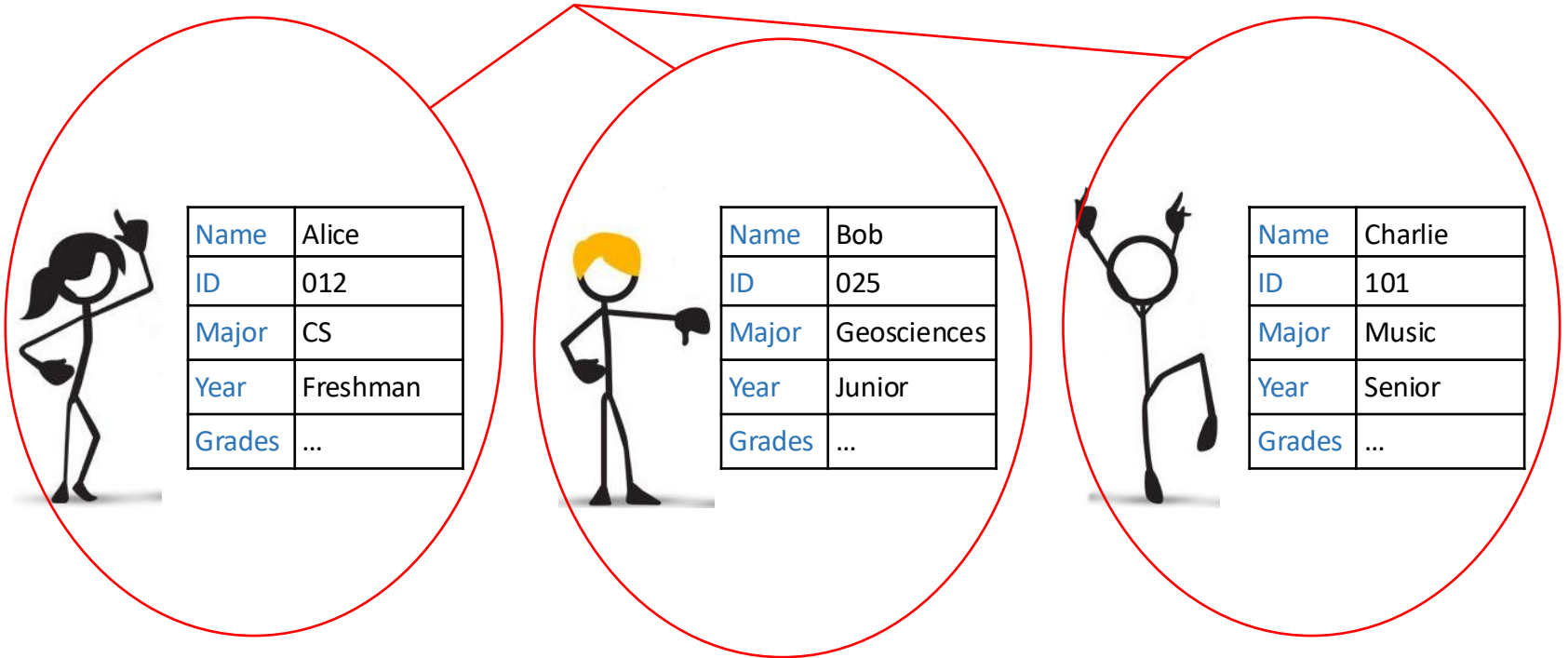
Name	Bob
ID	025
Major	Physics
Year	Junior
Grades	...



Name	Charlie
ID	101
Major	Music
Year	Senior
Grades	...

Example: a set of students at UA

Objects

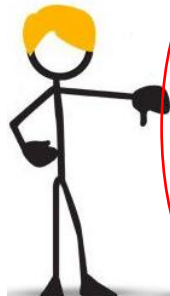


Example: a set of students at UA

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 attributes or *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
- The class is the template for making objects

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 **defs** 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 object 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()

Method invocation

```
class Student:
```

```
    def __init__(self, name, id):
```

```
        self._name = name
```

```
        self._id = id
```

```
    def get_name(self):
```

```
        return self._name
```

```
...
```

```
a = Student("Sally", 202)    # create a Student object
```

```
a.get_name()                # invoke a method
```

Think of "self" as an *alias* to the current object when the method is called.

EXERCISE –ICA-7 prob 1

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

- 1. Write a method `get_id` that returns a `Student` object's id.*
- 2. Create a `Student` object with name 'Harry' and id 342.*

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*:

- Attributes: ???
 - *what data might we want to associate with a Counter?*
- Methods: ???
 - *what methods are required for Counter objects?*
- Discuss with your neighbors...

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 – ICA-7 prob 2a

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 – ICA-7 prob 2b

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
    ....
```



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
```

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

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

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

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

EXERCISE - Whiteboard

Write a `__str__` method for Counter.

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

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



Solution

Write a `__str__` method for Counter.

```
class Counter:
```

```
    def __init__(self, name):
```

```
        self._name = name
```

```
        self._count = 0
```

```
    ...
```

```
    def __str__(self):
```

```
        return "Counter: " + self._name + "-> " + \
               str(self._count)
```



TERMINOLOGY

•

```
class Student:
```

```
    def __init__(self, name, id):
```

```
        self._name = name
```

```
        self._id = id
```

```
    def get_name(self):
```

```
        return self._name
```

Terminology

Provide the names of the items pointed to by the arrows.

class Student:

def __init__(self, name, id):

self._name = name

self._id = id

def get_name(self):
return self._name

...

? _____

? _____

? _____

? _____

Terminology

Provide the names of the items pointed to by the arrows.

class Student:

def __init__(self, name, id):

self._name = name

self._id = id

def get_name(self):
return self._name

...

-- class definition

-- constructor

-- instance variables or
attributes

-- method definition

Terminology

What happens at the arrow?

```
class Student:
```

```
    def __init__(self, name, id):
```

```
        self._name = name
```

```
        self._id = id
```

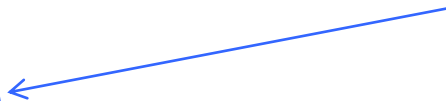
```
    def get_name(self):
```

```
        return self._name
```

```
...
```

```
a = Student("Sally", 202)
```

?



Terminology

What happens at the arrow?

```
class Student:
```

```
    def __init__(self, name, id):
```

```
        self._name = name
```

```
        self._id = id
```

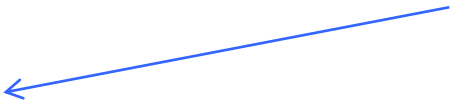
```
    def get_name(self):
```

```
        return self._name
```

```
...
```

```
a = Student("Sally", 202)
```

-- the `__init__()` constructor method is called and a Student object is created



EXERCISE-ICA-8 prob 1

Download the counter-with-str.py file (next to ICA-8)

Do prob 1, a) thru e)

```
class Counter:
```

```
    def __init__(self, name):
```

```
        self._name = name
```

```
        self._count = 0
```

```
    def click(self):
```

```
        self._count += 1
```

```
    ....
```



Recall: `__str__()`

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

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

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

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

- called by `str()` and `print()` to produce a string from an object's data

Special methods: `__repr__`

- Returns a string
 - the "official" string representation of the object
 - should look like a valid Python expression
- `__repr__(obj)`:
 - if `__str__` is not defined, Python will look for `repr`

Special methods: `__repr__`

Example:

class:	Student
attributes:	name id major

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

} `__str__(self)`
called by `str(obj)`

```
def __repr__(self):  
    return "Student(" + self._name + \  
        ", " + str(self._id) + \  
        ", " + self._major + ")"
```

} `__repr__(self)`
called by `repr(obj)`

Example: Point class

class Point:

```
def __init__(self, x, y):
```

```
    self._x = x
```

```
    self._y = y
```

Methods:

- *what methods might we want to associate with point objects?*
 - *change a point object's position by a given amount*
 - *compute its distance from the origin (0,0)*

EXERCISE (Whiteboard)

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). (Need to import math library to call `math.sqrt()`)

Use the formula:

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

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)
```

class Student : Initializing attributes

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

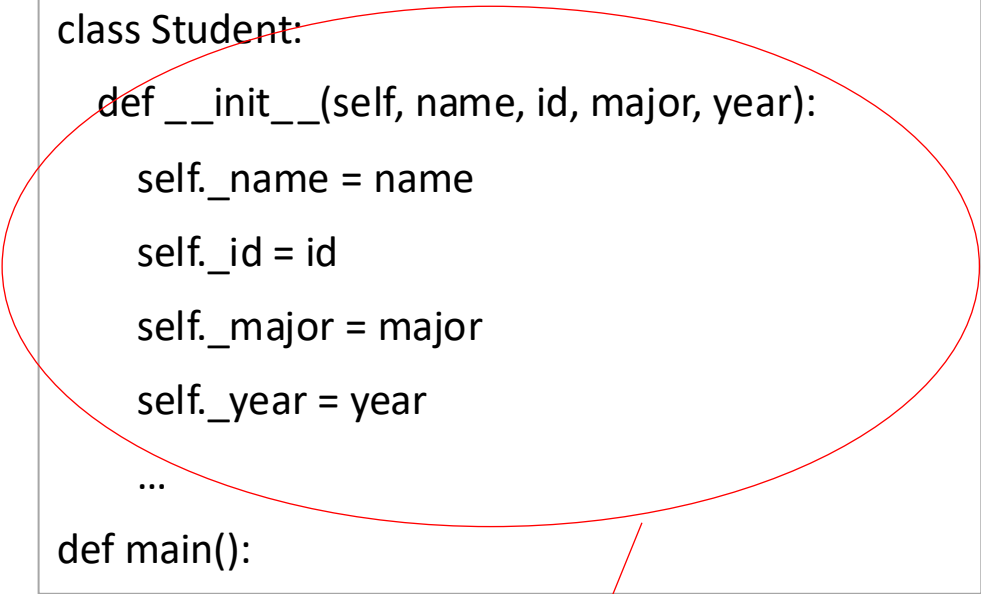
```
class Student:
    def __init__(self):
        self._name = ''
        self._id = -1
        ...

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

class Student : Initializing attributes

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 = -1
        ...

    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 : Initializing attributes

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)
        ...
```

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.**

class Student : Initializing attributes

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.

The methods are part of the object!

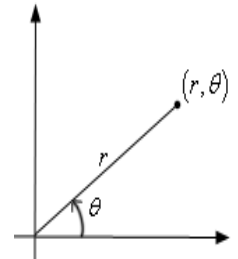
This expression causes an instance of the class Student to be created.

Encapsulation

- **encapsulation**: Hiding implementation details of a class
 - Goal: Minimize how much of the internal state is visible to the outside world
 - Allows you to change the implementation
 - Allows you to think at a higher level of abstraction
 - separates external view (behavior) from internal view (state)
 - Protects the 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.
 - Example: Only allow `Points` with non-negative coordinates.



EXERCISE – ICA-8 Prob 2

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 = ??

    def count(self):
        return ???
```



EXERCISE – ICA-8 Prob (sol)

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

    def click(self):
        self._count.append(1)

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

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__`

```
class Student:
```

```
...
```

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

```
...
```

- Is the special method used like this?
 `s1.__eq__(s2)`
- No. We are able to use the “==” operator
 `s1 == s2`

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

EXERCISE – ICA-9 prob 1

Write an `__eq__` method for Point.

Special methods: rich comparison

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

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

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 <code>s</code>	<code>len(s)</code>	<code>s.__len__()</code>
whether the object <code>s</code> contains an item <code>x</code>	<code>x in s</code>	<code>s.__contains__(x)</code>

EXERCISE – ICA-9 probs 2-3

Do problems 2 thru 3:

Implement two more methods for the Point class.

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

† Our *Pythonic* convention is that "_" at the beginning of an attribute name denotes that it is "private"

† <https://www.python.org/dev/peps/pep-0008/>

† It is a signal to the user that they should not modify the instance variable.

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 by Python)
one trailing underscore self._var1_	Used to avoid conflicts with Python keywords or functions, e.g., _list_, _class_, _dict_
two leading underscores self.__var1	Invokes <i>name mangling</i> : 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 method styles

- more terminology
- 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
 - not used outside of the class

Example: setter

```
class Point:
```

```
    def __init__(self, x, y):
```

```
        self._x = x
```

```
        self._y = y
```

```
    def move_to(self, x, y):
```

```
        self._x = x
```

```
        self._y = y
```

```
    def get_x(self):
```

```
        return self._x
```

```
    def get_y(self):
```

```
        return self._y
```



setter

Example: setter

```
class Point:
```

```
    def __init__(self, x, y):
```

```
        self._x = x
```

```
        self._y = y
```

```
    def move_to(self, x, y):
```

```
        self._x = x
```

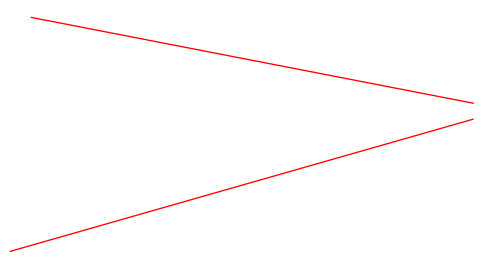
```
        self._y = y
```

```
    def get_x(self):
```

```
        return self._x
```

```
    def get_y(self):
```

```
        return self._y
```



getters

EXERCISE – ICA-9

Do problem 4.

Don't leave before the end of lecture!

(Do the challenge problem 5. if you finish prob. 4.)

We will continue with the lecture.

Example: getter

```
class BookData:
```

```
    def __init__(self, author, title, rating):
```

```
        self._author = author
```

```
        self._title = title
```

```
        self._rating = rating
```

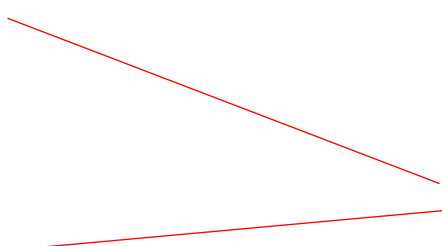
```
    def get_author(self):
```

```
        return self._author
```

```
    def get_rating(self):
```

```
        return self._rating
```

```
    ....
```



getters

Methods vs. functions

Functions

- Not associated with any class or object
 - invoked by name alone
- Arguments passed explicitly
- Operates on data passed to it

Methods

- 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

Methods

- Methods sometimes need temporary variables
 - use variables as in functions
 - don't use an instance variable for something temporary
 - e.g.,

```
for i in range(len(self._alist)):
```
- Classes often need helper methods
 - a method that helps other methods in the class perform a task
 - not used outside of the class
 - define them like any other method
 - call them within the class using self, e.g.:
 - self.helper(...)

Problem (Whiteboard)

- a) Write a method called `clean_word()`. Have it remove the punctuation of a string in text and return the cleaned version*
- b) Call it in from `__init__()`*

```
class Word:
```

```
    def __init__(self, text):
```

```
        # store a clean version of the word
```

```
        # strip off punctuation and convert to lowercase
```

```
        self._word = text.strip("!.:;,?").lower()
```

```
    def __str__(self):
```

```
        return "Word(" + self._word + ")"
```

Solution

Write a helper method `clean_word()` for method for `Word`.

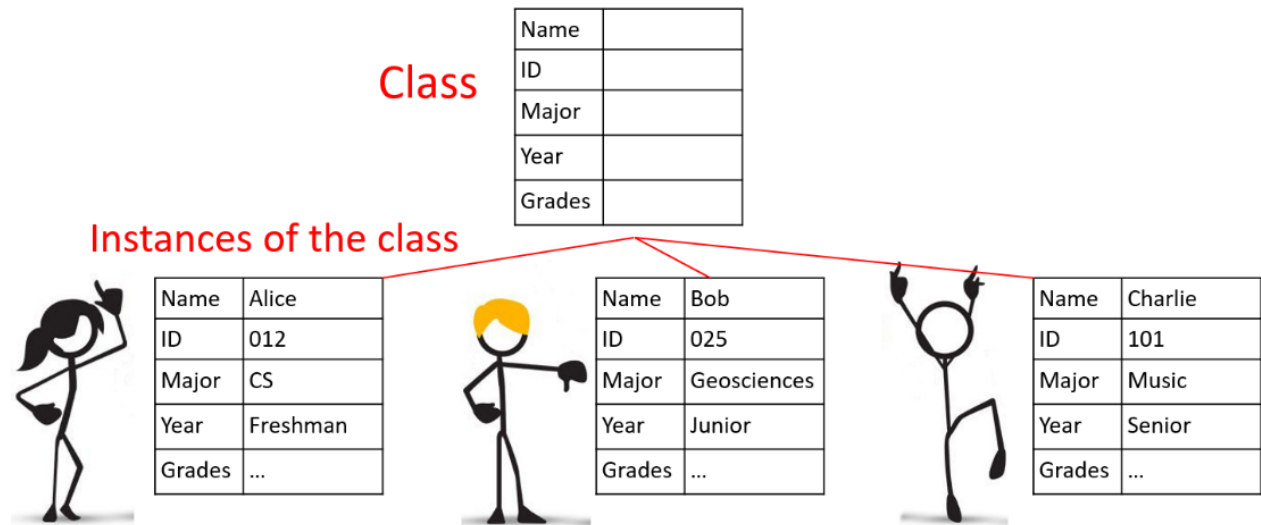
```
class Word:
    def __init__(self, text):
        self._word = self.clean_word(text)

    def clean_word(self, text):
        # strip off punctuation and convert to lowercase
        return text.strip(".!:,?-").lower()

    def __str__(self):
        return "Word(" + self._word + ")"
```

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
 - updating one object's instance variables does not affect other objects



Name	Alice
ID	012
Major	CS
Year	Freshman
Grades	...

- Examples:
- `self._name`, `self._id`, etc. of a Student object
- `self._x` and `self._y` of a Point 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__()`, `__eq__()`, ...

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