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
EXERCISE

Consider an alarm clock:

• State (properties):
  – What properties does an alarm clock have?

• Behavior (operations):
  – What does a clock do?
  – What operations could we define for a clock?
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

---

**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

*used to create instances of an iPod*
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

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
<th>Major</th>
<th>Year</th>
<th>Grades</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>012</td>
<td>CS</td>
<td>Freshman</td>
<td>CSC 110: B; CSC 120: A</td>
</tr>
<tr>
<td>Bob</td>
<td>025</td>
<td>Physics</td>
<td>Junior</td>
<td>GEO 215: B; Phys 120: C; GEO 325: A</td>
</tr>
<tr>
<td>Charlie</td>
<td>101</td>
<td>Music</td>
<td>Senior</td>
<td>MUS 210: B; MUS 423: A; CSC 110: B</td>
</tr>
</tbody>
</table>
Example: a set of students at UA

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
<th>Major</th>
<th>Year</th>
<th>Grades</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>012</td>
<td>CS</td>
<td>Freshman</td>
<td>CSC 110: B; CSC 120: A</td>
</tr>
<tr>
<td>Bob</td>
<td>025</td>
<td>Physics</td>
<td>Junior</td>
<td>GEO 215: B; Phys 120: C; GEO 325: A</td>
</tr>
<tr>
<td>Charlie</td>
<td>101</td>
<td>Music</td>
<td>Senior</td>
<td>MUS 210: B; MUS 423: A; CSC 110: B</td>
</tr>
</tbody>
</table>

Object-oriented representation:

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

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

- Charlie
  - Name: Charlie
  - ID: 101
  - Major: Music
  - Year: Senior
  - Grades: ...
Example: a set of students at UA

Objects

<table>
<thead>
<tr>
<th>Name</th>
<th>Alice</th>
<th>Bob</th>
<th>Charlie</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>012</td>
<td>025</td>
<td>101</td>
</tr>
<tr>
<td>Major</td>
<td>CS</td>
<td>Geosciences</td>
<td>Music</td>
</tr>
<tr>
<td>Year</td>
<td>Freshman</td>
<td>Junior</td>
<td>Senior</td>
</tr>
<tr>
<td>Grades</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example: a set of students at UA

<table>
<thead>
<tr>
<th>Name</th>
<th>Alice</th>
<th>ID</th>
<th>012</th>
<th>Major</th>
<th>CS</th>
<th>Year</th>
<th>Freshman</th>
<th>Grades</th>
<th>...</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Bob</th>
<th>ID</th>
<th>025</th>
<th>Major</th>
<th>Geosciences</th>
<th>Year</th>
<th>Junior</th>
<th>Grades</th>
<th>...</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Charlie</th>
<th>ID</th>
<th>101</th>
<th>Major</th>
<th>Music</th>
<th>Year</th>
<th>Senior</th>
<th>Grades</th>
<th>...</th>
</tr>
</thead>
</table>
Example: a set of students at UA

<table>
<thead>
<tr>
<th>Name</th>
<th>Alice</th>
<th>Bob</th>
<th>Charlie</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>012</td>
<td>025</td>
<td>101</td>
</tr>
<tr>
<td>Major</td>
<td>CS</td>
<td>Geosciences</td>
<td>Music</td>
</tr>
<tr>
<td>Year</td>
<td>Freshman</td>
<td>Junior</td>
<td>Senior</td>
</tr>
<tr>
<td>Grades</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example: a set of students at UA

Class

Instances of the class

<table>
<thead>
<tr>
<th>Name</th>
<th>Alice</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>012</td>
</tr>
<tr>
<td>Major</td>
<td>CS</td>
</tr>
<tr>
<td>Year</td>
<td>Freshman</td>
</tr>
<tr>
<td>Grades</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>025</td>
</tr>
<tr>
<td>Major</td>
<td>Geosciences</td>
</tr>
<tr>
<td>Year</td>
<td>Junior</td>
</tr>
<tr>
<td>Grades</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Charlie</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>101</td>
</tr>
<tr>
<td>Major</td>
<td>Music</td>
</tr>
<tr>
<td>Year</td>
<td>Senior</td>
</tr>
<tr>
<td>Grades</td>
<td>...</td>
</tr>
</tbody>
</table>
Objects

• An *object* consists of:
  – a state
    o given by the values of its attributes or *instance variables*
  – a set of behaviors
    o 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

Instance variables

- name
- id
- major
- year
- grades

Methods:

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Alice</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>012</td>
</tr>
<tr>
<td>Major</td>
<td>CS</td>
</tr>
<tr>
<td>Year</td>
<td>Freshman</td>
</tr>
<tr>
<td>Grades</td>
<td>...</td>
</tr>
</tbody>
</table>
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
    ...
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

...
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

_instance variables

These refer to attributes of the object being referred to, and so are written

_instance variables
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

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
    ...

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
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
    ...
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)

    ...
EXERCISE

Write a method `get_id` that returns a Student object's id.

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

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

a = Student(“Sally”, 202)
a.get_name() ← same as → Student.get_name(a)

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

... 
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():

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.

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:
    def __init__(self):
        self._name = ''
        self._id = ''

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

```python
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

```python
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)
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 = ??

    def count(self):
        return ???
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

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)
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__()`

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

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

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

- `__str__()` : a special method for constructing a string from an object
- called by `str()` and `print()` to convert objects to strings
EXERCISE

Write a \_\_str\_\_ method for Counter.

class Counter:
    def \_\_init\_\_(self, name):
        self._name = name
        self._count = 0

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

    ....
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`
EXERCISE

Write a `__repr__` method for Counter.

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

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

    ....
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...>
**Special methods: \_\_repr\_\_**

Example:

<table>
<thead>
<tr>
<th>class:</th>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes:</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>id</td>
</tr>
<tr>
<td></td>
<td>major</td>
</tr>
<tr>
<td></td>
<td>grades</td>
</tr>
</tbody>
</table>

```python
def \_\_str\_\_(self):
    return "Student_" + self._name + ": " + self._id
def \_\_repr\_\_(self):
    return "< name :" + self._name +
      + ", id :" + self._id \
      + ", major: " + self._major \
      + ", grades: " + str(self._grades) + ">
```

\_\_str\_(self)\_\_ called by **str(obj)**

\_\_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
    o E.g.: str("3") == str(3)

• __repr__ : aims to be unambiguous
  – "official" string representation of an object
    o 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__()
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:

\[ \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \]
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)
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 $\theta$), but with the same methods.

• Allows you to constrain objects' state.
  – 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

† 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/](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

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>One leading underscore</td>
<td><code>self._var1</code> Indicates that the attribute is &quot;not public&quot; and should only be accessed by the class's internals (convention; not enforced by Python)</td>
</tr>
<tr>
<td>One trailing underscore</td>
<td><code>self.var1_</code> Used to avoid conflicts with Python keywords, e.g., list_, class_, dict_</td>
</tr>
<tr>
<td>Two leading underscores</td>
<td><code>self.__var1</code> Invokes <em>name mangling</em>: from outside the class to enforce private e.g., <code>self.__var1</code> appears to be at <code>YourClassName.__YourClassName__var1</code></td>
</tr>
<tr>
<td>Two leading + trailing underscores</td>
<td><code>self.__var1__</code> Intended only for names that have special significance for Python, e.g., <strong>init</strong></td>
</tr>
</tbody>
</table>
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
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

.....
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
## Methods vs. functions

<table>
<thead>
<tr>
<th>Functions</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Not associated with any class or object</td>
<td>• Associated with a class or object</td>
</tr>
<tr>
<td>– invoked by name alone</td>
<td>– invoked by object.name</td>
</tr>
<tr>
<td>• Arguments passed explicitly</td>
<td>• The object for which it was called is passed implicitly</td>
</tr>
<tr>
<td>• Operates on data passed to it</td>
<td>• Can operate on data contained within the class</td>
</tr>
</tbody>
</table>
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:

```python
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__`

```python
>>> class Student:
    def __init__(self, name, id):
        self._name = name
        self._id = id
    def __eq__(self, other):
        return self._name == other._name \n        and self._id == other._id

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

>>> s1 == s2
False
>>> s1 == s3
True
```

`==` on the objects calls the `__eq__()` method of the class.
Write an \_\_eq\_\_ method for Point.

class Point:
    def \_\_init\_(\_\_\_\_, x, y):
        self._x = x
        self._y = y

....
Special methods: rich comparison

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

<table>
<thead>
<tr>
<th>Comparison operator</th>
<th>Method called</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td><code>__eq__()</code></td>
</tr>
<tr>
<td>!=</td>
<td><code>__ne__()</code></td>
</tr>
<tr>
<td>&lt;</td>
<td><code>__lt__()</code></td>
</tr>
<tr>
<td>&lt;=</td>
<td><code>__le__()</code></td>
</tr>
<tr>
<td>&gt;</td>
<td><code>__gt__()</code></td>
</tr>
<tr>
<td>&gt;=</td>
<td><code>__ge__()</code></td>
</tr>
</tbody>
</table>
Special methods: `__len__`, `__contains__`

For a class that acts like a collection of items:

<table>
<thead>
<tr>
<th>You want...</th>
<th>You write...</th>
<th>And Python calls...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the no. of items in the object <code>s</code></td>
<td><code>len(s)</code></td>
<td><code>s.__len__()</code></td>
</tr>
<tr>
<td>whether the object <code>s</code> contains an item <code>x</code></td>
<td><code>x in s</code></td>
<td><code>s.__contains__(x)</code></td>
</tr>
</tbody>
</table>
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
    o so updating one object's instance variables does not affect other objects

• 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__(), ...