CSc 120
Introduction to Computer Programming II

03: Basics of Object-Oriented Programming
Objects

• Very often, computer programs model properties of, and interactions between, entities in the world

• The code for such programs can be made cleaner and simpler by modeling these entities directly

⇒ "objects"
# 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**
Example: a set of students at UA

Objects

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

2. Bob
   - ID: 025
   - Major: Geosciences
   - Year: Junior
   - Grades: ...

3. 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
Attributes or Instance variables
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>…</td>
</tr>
<tr>
<td>Bob</td>
<td>025</td>
<td>Geosciences</td>
<td>Junior</td>
<td>…</td>
</tr>
<tr>
<td>Charlie</td>
<td>101</td>
<td>Music</td>
<td>Senior</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>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></td>
</tr>
<tr>
<td>Bob</td>
<td>025</td>
<td>Geosciences</td>
<td>Junior</td>
<td></td>
</tr>
<tr>
<td>Charlie</td>
<td>101</td>
<td>Music</td>
<td>Senior</td>
<td></td>
</tr>
</tbody>
</table>
Objects

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

• Intuitively, an object models an entity in a real or virtual world or system
Example: Student 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()

**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 contents and behaviors of a set of similar objects
  – contents: given by instance variables
  – behaviors: given by the methods of the class

• These objects are instances of the 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
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'
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
    _name
    _id

These refer to properties 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()

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
    ...

Example: using the Student class

```python
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

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 fields
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 = ""
        ... 
    
def main():
        ... 
        student = Student()
        student.set_name(name)
        student.set_id(id)
        ...
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 a good idea to **let each class handle its own internal details.**

Minimize how much the outside world knows about the internals of the class.

This is **encapsulation.**
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)
    student.set_name(name)
    student.set_id(id)

...
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)
Other Examples

• Can you think of other examples of objects that we've seen in the Python language?
Other Examples

• Can you think of other examples of objects that we've seen in the Python language?
  - list
  - tuple
  - str
  - dict

https://docs.python.org/3/library/

...actually, everything in Python is an object!
EXERCISE

Suppose we want to define a class for a course:

• Data:
  – *what data might we want to associate with course objects?*

• Methods:
  – *what methods might we want to associate with course objects?*
Public and private variables

- Variables in a class are generally divided into two kinds:
  - `public`: visible to all code
  - `private`: visible only within the class

- This can improve code quality by hiding internal details of a class where appropriate ("encapsulation")

† Python uses the convention that "__" at the beginning of a variable/method name denotes "private"
  - "name mangling" used on such names to avoid clashes
## Class attribute naming conventions

<table>
<thead>
<tr>
<th>One leading underscore</th>
<th>Indicate that the attribute is &quot;not public&quot; and should only be accessed by the class's internals (convention; not enforced)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>self._var1</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>One trailing underscore</th>
<th>Used to avoid conflicts with Python keywords, e.g., <code>list_</code>, <code>class_</code>, <code>dict_</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>self.var1_</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Two leading underscores</th>
<th>Invokes name mangling: from outside the class, <code>self.__var1</code> appears to be at <code>_YourClassName__var1</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>self.__var1</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Two leading + trailing underscores</th>
<th>Intended only for names that have special significance for Python, e.g., <code>__init__</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>self.__var1__</code></td>
<td></td>
</tr>
</tbody>
</table>
Classic Method 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