Visibility

- No visibility modifiers like in Java, C++,...
  - everything is public

- Attributes starting with _ should be considered as private (better call them internal)

- **Name mangling** – partial support for private attributes
  - identifier **__xxx** (at least two leading underscores, at most one trailing underscore) is textually replaced with **_classname__xxx**

Examine and run mangling.py
Interfaces and Polymorphism

- Structural subtyping
  - aka “Duck typing”
    - “If it walks like a duck and quacks like a duck, it must be a duck.”

- There is not language construct for an interface
  - This means that interfaces form an implicit contract that is captured by comments and documentation
class List:
    def add(item): ...
def size(): ...

class Set:
    def add(item): ...
def size(): ...

def add_to_collection(collection, item):
    collection.add(item)

ls = List()
st = Set()

item = get_item()
add_to_collection(ls, item)
add_to_collection(st, item)
Note on Inheritance

- In statically-typed languages, inheritance combines two different features
  - Subclassing
    - Forming a class based on a previous class
    - The aim is to reuse code
  - Subtyping
    - A type may be used in places where supertype is expected
    - The aim is polymorphism
- In Python, however, inheritance brings only subclassing
  - Subtyping is handled at runtime based on method lookup
Properties

- Recommendation
  - directly access object variables
    - i.e., not to use getters and setters

- Sometimes getter and/or setters are necessary
  - e.g., read-only values, computed values, changes in subclasses,...

- Solution => properties
  - property ~ variable with getter, setter and deleter
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

x = property(getx, setx, delx, "'x' property.")

- Not all of the getter, setter and deleter are necessary
  - can be None
Easier specification — via `@property` decorator

```python
class C:
    def __init__(self):
        self._x = None

@property
def x(self):
    """'x' property.""
    return self._x

@x.setter
def x(self, value):
    self._x = value

@x.deleter
def x(self):
    del self._x
```

See `properties.py`
staticmethod

- Methods without self
  - similar static methods in Java and C++
  - methods logically belonging to a class but do not access any object variables

```python
class C:
    @staticmethod
    def show(msg):
        print(msg)
```

See static.py
Abstract Base Classes

- Module abc
- Support for “interfaces” and methods that must be implemented in subclasses

```python
import abc

class PluginBase(abc.ABC):
    @abc.abstractmethod
    def process(self, input):
        pass

class ToUpperPlugin(PluginBase):
    def process(self, self, input):
        return input.upper()
```

Examine and run abstract_base.py
## Abstract Base Classes

- **collections.abc**
  - special module for collection like classes

<table>
<thead>
<tr>
<th>ABC</th>
<th>Inherits from</th>
<th>Abstract Methods</th>
<th>Mixin Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td></td>
<td><strong>contains</strong></td>
<td></td>
</tr>
<tr>
<td>Hashable</td>
<td></td>
<td><strong>hash</strong></td>
<td></td>
</tr>
<tr>
<td>Iterable</td>
<td></td>
<td><strong>iter</strong></td>
<td></td>
</tr>
<tr>
<td>Iterator</td>
<td>Iterable</td>
<td><strong>next</strong></td>
<td><strong>iter</strong></td>
</tr>
<tr>
<td>Reversible</td>
<td>Iterable</td>
<td><strong>reversed</strong></td>
<td></td>
</tr>
<tr>
<td>Generator</td>
<td>Iterator</td>
<td>send, throw</td>
<td>close, <strong>iter</strong>, <strong>next</strong></td>
</tr>
<tr>
<td>Sized</td>
<td></td>
<td><strong>len</strong></td>
<td></td>
</tr>
<tr>
<td>Callable</td>
<td></td>
<td><strong>call</strong></td>
<td></td>
</tr>
<tr>
<td>Collection</td>
<td>Sized, Iterable, Container</td>
<td><strong>contains</strong>, <strong>iter</strong>, <strong>len</strong></td>
<td></td>
</tr>
<tr>
<td>Sequence</td>
<td>Reversible, Collection</td>
<td><strong>getitem</strong>, <strong>len</strong></td>
<td><strong>contains</strong>, <strong>iter</strong>, <strong>reversed</strong>, index, and count</td>
</tr>
<tr>
<td>MutableSequence</td>
<td>Sequence</td>
<td><strong>getitem</strong>, <strong>setitem</strong>, <strong>delitem</strong>, <strong>len</strong>, insert</td>
<td>Inherited Sequence methods and append, reverse, extend, pop, remove, and <strong>add</strong></td>
</tr>
<tr>
<td>ByteString</td>
<td>Sequence</td>
<td><strong>getitem</strong>, <strong>len</strong></td>
<td>Inherited Sequence methods</td>
</tr>
<tr>
<td>Set</td>
<td>Collection</td>
<td><strong>contains</strong>, <strong>iter</strong>, <strong>len</strong></td>
<td><strong>le</strong>, <strong>lt</strong>, <strong>eq</strong>, <strong>ne</strong>, <strong>gt</strong>, <strong>ge</strong>, <strong>and</strong>, <strong>or</strong>, <strong>sub</strong>, <strong>xor</strong>, and isdisjoint</td>
</tr>
<tr>
<td>MutableSet</td>
<td>Set</td>
<td><strong>contains</strong>, <strong>iter</strong>, <strong>len</strong>, add, discard</td>
<td>Inherited Set methods and clear, pop, remove, <strong>ior</strong>, <strong>iand</strong>, <strong>ixor</strong>, and <strong>isub</strong></td>
</tr>
</tbody>
</table>
Enum

- Module enum
  - enum ~ a class with several named constants

```python
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

a = Color.RED

if a is Color.RED:
    print("is red")

for color in Color:
    print(color)
```

Examine and run enums.py