Chapter 6 - Object-Oriented Programming

CS 171 - Computer Programming 1
Lanzhou University

These slides use many elements provided in the main bibliographic reference for these lectures:

Programming in Python 3
A Complete Introduction to the Python Language,
2nd Edition,
Mark Summerfield

Object-Oriented Programming

- We have been using a procedural style of programming
- Python is a multiparadigm language, and allows
  - procedural programming
  - object-oriented programming
  - programming in functional style
  - any mixture of these styles
- For small programs, procedural programming is possible
  - but for more realistic programs, object orientation has many advantages
- We cover the fundamental aspects for object-oriented programming in Python

Outline

Outline

The Object-Oriented Approach

- Imagine that you are writing a program that needs to handle circles
  - potentially lots of them
- The minimum data required to represent one circle is
  - its center position `(x,y)`
  - its radius
- One simple approach is to represent a circle by a 3-tuple:

```
circle = (11, 60, 8)
```

- But his representation is far from clear; it could mean
  - `(x, y, radius)` or, as easily,
  - `(radius, x, y)`

The Object-Oriented Approach

- One simple approach is to represent a circle by a 3-tuple:

  ```
  circle = (11, 60, 8)
  ```

- But his representation is far from clear; it could mean
  - `(x, y, radius)` or, as easily,
  - `(radius, x, y)`
- Also, if we had a function `distance_from_origin(x, y)`,
  - we would have to use tuple unpacking to call it on a `circle` tuple:

    ```
    distance = distance_from_origin(*circle[:2])
    ```

  - Note that this call assumes the representation `(x, y, radius)`

The Object-Oriented Approach

- We can solve the problem of knowing the element order and of using tuple unpacking by using a named tuple:

```
import collections
Circle = collections.namedtuple("Circle", "x y radius")
circle = Circle(13, 84, 9)
distance = distance_from_origin(circle.x, circle.y)
```

- This allows us to create `Circle` 3-tuples with named attributes
- Making function calls much easier to understand
- Unfortunately, some problems still remain; for example, we can call

```
circle = Circle(33, 56, -5)
```

  - it does not make sense to have a circle with a negative radius
  - but this call executes *properly*, without an exception being raised
  - all functions, e.g., `distance_from_origin` need to perform validations

The Object-Oriented Approach

- If we want circles to be mutable, we can do so
  - using the private `collections.namedtuple._replace()` method

    ```
    circle = circle._replace(radius=12)
    ```
- Still, nothing prevents us from setting invalid data
- If we needed to frequently change circles, we could use a mutable representation;

  ```
  circle = [36, 77, 8]
  ```

  - Still, this does not give us any protection against invalid data
  - and we could actually call, e.g., `circle.sort()`

Object-Oriented Concepts and Terminology

- We need a way to pack the data that is needed to represent a circle
- And some way to restrict the methods that can be applied to the data
- Both can be achieved creating a `Circle` data type
- We start be covering necessary preliminaries and terminology

Object-Oriented Concepts and Terminology

- We use terms *class*, *type* and *data type* interchangeably
- We can create custom classes that can be used just like any built-in data type
- We have used many classes
  - `dict`
  - `str`
  - `int`
- We use the term *object* or *instance*, to refer to an instance of a class
  - 5 is an `int` object

Object-Oriented Concepts and Terminology

- Most classes encapsulate both data and the methods applicable to it
  - `str` holds a string of Unicode characters as its data
  - and supports methods such as `str.upper()`
- Many classes support additional features
  - we can concatenate two strings using the + operator
  - we can find the length of a sequence using `len()`
- These features are provided by *special methods*
  - resemble normal methods, but whose names begin and end with `__`
- If we want to create a class that supports concatenation using +
  - we can do it by implementing the method `__add__()`
- Similarly, for a class to support determining the length using `len()`
  - we can do it by implementing the method `__len__()`

Object-Oriented Concepts and Terminology

- Objects usually have attributes
  - methods are callable attributes
  - other attributes are data
- A `complex` object
  - has `imag` and `real` as attributes, and
  - special methods like `__add__()` and `__sub__()`
    - to support the binary + and - operators
  - regular methods like `conjugate()`
- Data attributes are normally implemented as *instance variables*
  - variables that are unique to a particular object
- We can also provide data attributes as *properties*
  - which is useful to perform data validation;
  - We will not explicitly cover this possibility, though;

Object-Oriented Concepts and Terminology

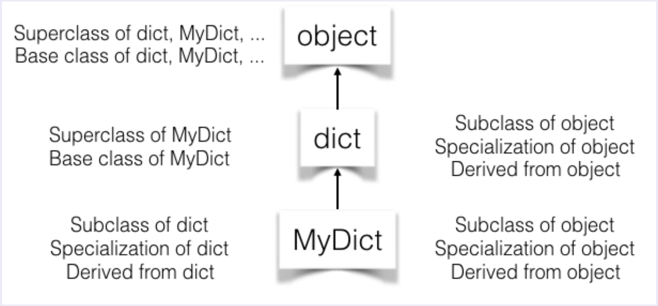
- A method is a function whose first argument is the instance on which it is called to operate
- Inside a method, several kinds of variables are accessible
  - the object's instance variables
    - accessible by qualifying their name with the instance itself
  - local variables can be created inside the method
    - accessible without qualification
  - class variables, sometimes called static variables
    - accessible qualifying their name with the class name
  - global variables
    - accessible without qualification

Object-Oriented Concepts and Terminology

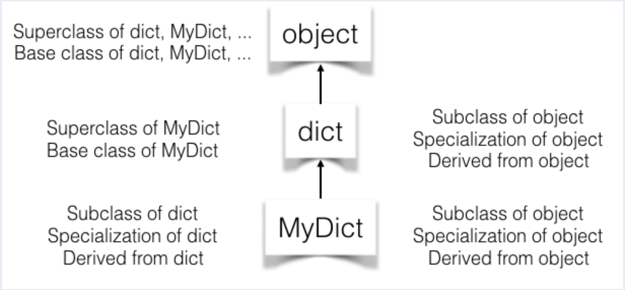
- One of the advantages of object orientation is *specialization*
- We can make a class *inherit* all the attributes of another class
- The new class is a *subclass* of the existing one
  - also called a *specialized* version of the existing class
- This allows us to exploit what was already implemented and tested
  - and add new data attributes or new functionality in a simple way
- We can also pass objects of the new class to functions and methods that were written for the original class

Object-Oriented Concepts and Terminology

- We call *base class* or *super class* to a class that is inherited
- We call *subclass* or *derived class* to a class that inherits from another
- In Python, every built-in and library class and every class we create
  - is derived directly or indirectly from the ultimate base class - `object`



Object-Oriented Concepts and Terminology



- Any method can be *overridden*, that is, re-implemented, in a subclass
- If we have a `MyDict` object, and we call a method that is defined by both `dict` and `MyDict`, Python will correctly call the `MyDict` version
  - this is called as *dynamic method binding* or *polymorphism*
- If we want to call the base class version of a method inside a re-implemented method, we should use function `super()`

Object-Oriented Concepts and Terminology

- We will use an uppercase letter as the first letter of custom classes
- We can define as many classes as we like, either directly in a program on in modules
  - class names don't have to match module names
  - modules may contain as many class definitions as needed

Outline

Custom Classes

- The syntaxes for creating new classes are:

```
class className:
    suite

class className(base_classes):
    suite
```

- Note that we have already created custom exception classes
  - Since they did not add any new attributes, we used `pass` as suite
  - As the suite was just one statement, we put it in the same line
- Methods of a class are created using `def` statements in its suite
- Instances are created by calling a class with appropriate arguments
  - For example, `x = complex(4,8)`
    - creates a complex number
    - sets `x` to be an object reference to it

Attributes and Methods

- Let us implement a very simple class, `Point`
    - to hold *(x, y)* coordinates
- We could could implement it in a file, e.g., `Shape.py`, as follows:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return math.hypot(self.x, self.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __repr__(self):
        return "Point({0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

Attributes and Methods

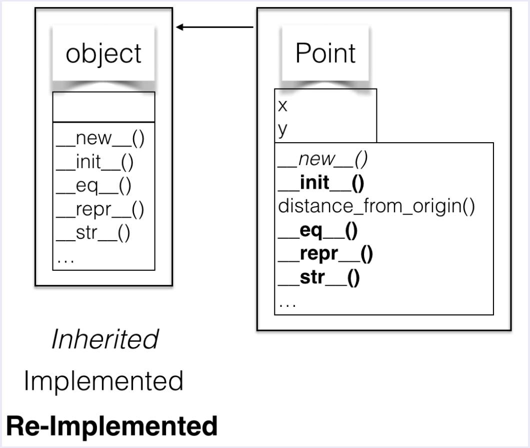
- Since no base classes are specified, `Point` is a direct subclass of `object`
  - we could have also written `class Point(object):`
- Let's start by observing the methods of `Point` in action:

```
>>> import Shape
>>> a = Shape.Point()
>>> repr(a)
'Point(0, 0)'
>>> b = Shape.Point(3, 4)
>>> str(b)
'(3, 4)'
>>> b.distance_from_origin()
5.0
>>> b.x = -19
>>> str(b)
'(-19, 4)'
>>> a == b, a != b
(False, True)
```

Attributes and Methods

- The `Point` class has two data attributes
  - `self.x`
  - `self.y`
- And five methods
  - Four of which, starting and ending with `__`, are special methods
    - These methods re-implement methods that were inherited from `object`
- Still, the class inherits other methods from the `object` class
  - Such as `__new__()`
- By importing module `Shape`, `Point` can be used like any other class
- The data attributes can be accessed directly
  - e.g., `x = a.y`
- The class integrates nicely with all Python's other classes
  - By providing support for the equality operator (`==`)
  - and for producing strings in representational and string formats
- Python can infer inequality operator (`!=`) from the equality operator
  - But we can also specify inequality

Attributes and Methods



Attributes and Methods

- The first argument of methods should be `self`
  - which is an object reference to the instance object itself
- All object attributes must be qualified by `self`
  - e.g., `self.x`
  - These include data and method attributes
  - This strategy is expected to improve clarity
- When creating an object,
  - the special method `__new__()` is called to create the object
  - then the special method `__init__()` is called to initialize it
- We will essentially only need to re-implement `__init__()`
  - as `object.__new__()` is almost always sufficient

Attributes and Methods

- It is important to understand what happens when an object is created
- For example, when calling `p = Shape.Point()`
    - Python will start by looking for the method `Point.__new__()`
    - Since it is not defined, Python will look for it in `Point`'s base classes
    - In this case, there is only one, `object`
    - `object` does implement `__new__()`, so this method will be executed
    - Then, Python looks for the initializer, `__init__()`
    - Since we re-implemented it, `Point.__init__()` is executed
    - Finally, Python sets `p` to be a reference to the new `Point` object

Attributes and Methods

- Going back to studying `Point`'s methods, one by one

```
def __init__(self, x=0, y=0):
    self.x = x
    self.y = y
```

- The two instance variables `self.x`, and `self.y`
  - are created in the initializer and
  - assigned the values of the `x` and `y` parameters

Attributes and Methods

- Going back to studying `Point`'s methods, one by one

```
    def distance_from_origin(self):
        return math.hypot(self.x, self.y)
```

- This is a conventional method
  - that performs a computation based on the object's instance variables
- It is common for methods to be short
- And often methods only need the information on the object itself
  - In this case methods only have one argument, which is `self`

Attributes and Methods

- Going back to studying `Point`'s methods, one by one

```
def __eq__(self, other):
    return self.x == other.x and self.y == other.y
```

- All instances of custom classes support == by default
  - The (default) comparison returns `False`
  - unless we compare a custom object with itself
- We can override this behavior
  - by re-implementing the `__eq__()` special method
  - as we have done above
- Python automatically supplies the `__ne__()` inequality operator (`!=`)
  - if we don't implement it ourselves

Attributes and Methods

- Going back to studying `Point`'s methods, one by one

```
    def __repr__(self):
        return "Point({0.x!r}, {0.y!r})".format(self)
```

- The built-in `repr()` function calls the `__repr__()` special method
- One of the uses of `repr()` is to produce a string
  - that can be evaluated by `eval()` to produce the original object

```
>>> p = Shape.Point(3, 9)
>>> repr(p)
'Point(3, 9)'
>>> q = eval(p.__module__ + "." + repr(p))
>>> q
Point(3, 9)
```

- Python provides every object with a few private attributes
  - one of which is `__module__`, a string holding the object's module name

Attributes and Methods

- Going back to studying `Point`'s methods, one by one

```
    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

- The built-in `str()` function works like the `__repr__()` function
  - except that it calls the object's `__str__()` special method
- The result is expected to be understandable by human readers
  - and is not expected to be suitable for passing its result to `eval()`

```
>>> p = Shape.Point(3, 9)
>>> str(p)
'(3, 9)'
```

Inheritance and Polymorphism

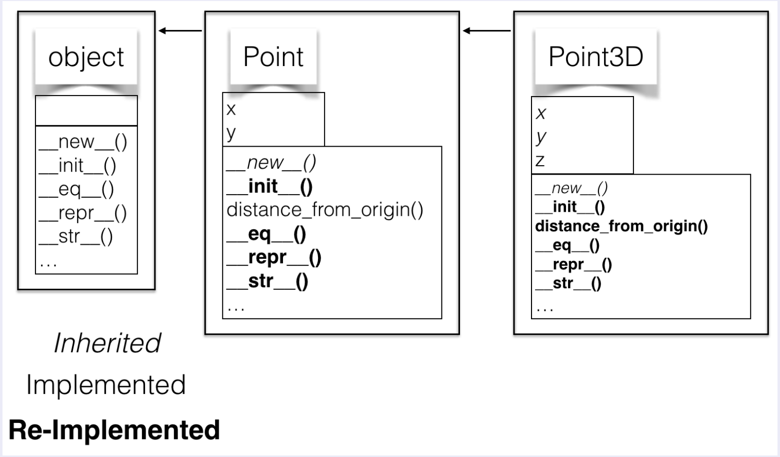
- We can rely on the class we have created to build specialized `Points`
  - Exploiting inheritance to, e.g., to use three dimensional points

```
class Point3D(Point):
    def __init__(self, x=0, y=0, z=0):
        super().__init__(x, y)
        self.z = z

    def distance_from_origin(self):
        return math.sqrt((self.x)ˆ2 + (self.y)ˆ2 + (self.z)ˆ2)

    def __eq__(self, other):
        return super().__eq__(other) and self.z == other.z

    def __repr__(self):
        return "Point3D({0.x!r}, {0.y!r}, {0.z!r})".format(self)

    def __str__(self):
        return "({0.x!r}, {0.y!r}, {0.z!r})".format(self)
```

- Inside the `__init__()` and `__eq__()` methods, we use `super()`
  - to refer to the corresponding methods on the base class

Inheritance and Polymorphism

- We had to re-implement many methods to account for dimension z
- It was also necessary to re-implement `distance_from_origin()`



Inheritance and Polymorphism

- Using `Points` and `Point3Ds`:

```
>>> import Shape
>>> x = Shape.Point(1, 2)
>>> x.distance_from_origin()
2.23606797749979
>>> y = Shape.Point(2, 1)
>>> x == y
False
>>> z = Shape.Point3D(1, 2, 3)
>>> z.distance_from_origin()
1.4142135623730951
>>> z
Point3D(1, 2, 3)
```

- Polymorphism allows any object of a given class to be used as an object of any of its base classes
  - so, we only need to re-implement if necessary;
  - otherwise, inheritance does the job;
- In `Point3D`, we could have defined *new* methods
  - i.e., methods that were not implement for `Point`;

Outline

Creating Classes That Aggregate Collections

- We start by introducing `Image`, a class to hold image data;
- We can represent a 2D color image as a 2-dimensional array
  - where each array element stores a color;
  - to store a 100*100 image we must store 10 000 colors;
- For class `Image`, we will use a more efficient approach
  - an `Image` stores a single background color, and
  - the colors of the image's points that differ from the background color;
- This can be done using a dictionary,
  - with each key being an `(x,y)` coordinate, and
  - the corresponding value being the color of that point;
- On a 100*100 image, if half its points are the background color
  - we only need to store $5000 + 1$ colors
  - which consists of a considerable memory saving;

Creating Classes That Aggregate Collections

- The full implementation of `Image` is provided in attachment
  - We will go through specific parts of it
  - to highlight the most relevant concepts in this context;
  - It is recommended that you study the full implementation.
- `Image` creates some custom exception classes
  - from which we show the first two:

    ```
    class ImageError(Exception): pass
    class CoordinateError(ImageError): pass
    ```
  - The remaining classes are created the same way
    - and like `CoordinateError`, they all inherit from `ImageError`

Creating Classes That Aggregate Collections

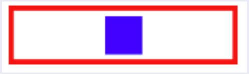
- We start by looking at an example of how Image can be used:

```
border_color = "#FF0000"    # red
square_color = "#0000FF"    # blue
width, height = 240, 60
midx, midy = width // 2, height // 2
image = Image.Image(width, height, "square_eye.img")
for x in range(width):
    for y in range(height):
        if x < 5 or x >= width - 5 or y < 5 or y >= height - 5:
            image[x, y] = border_color
        elif midx - 20 < x < midx + 20 and midy - 20 < y < midy + 20:
            image[x, y] = square_color
image.save()
image.export("square_eye.xpm")
```

- Image uses HTML-style hexadecimal strings to represent colors

Creating Classes That Aggregate Collections

- Image can export to `.xpm` format;
- The output of the example code would be the image:
  - in file `square_eye.xpm`



- We used the item access operator `[]` for setting colors in the image
  - as in `image[x, y] = border_color`;
  - This would be the same as `image[(x, y)] = border_color`
  - Using this syntax integration is easy
    - we *only* need to implement the necessary special methods;
    - in this case, we would need to implement `__setitem__()` in `Image`;

Creating Classes That Aggregate Collections

- We will now review the methods in the `Image` class
- We start with the `class` line and the initializer:

```
class Image:

    def __init__(self, width, height, filename="", background="#FFFFFF"):
        self.filename   = filename
        self.background = background
        self.data    = {}
        self.width  = width
        self.height = height
        self.colors = {self.background}
```

- A `width` and a `height` must be provided;
- The `filename` and `background` color are optional
  - since default values are provided
- `self.data` is a dictionary from `(x,y)` coordinates to color strings
- `self.colors` set is initialized with the background color
  - it is used to keep track of the unique colors used by the image

Creating Classes That Aggregate Collections

- The definition of `__getitem__` is as follows:

```
def __getitem__(self, coordinate):
    assert len(coordinate) == 2, "coordinate should be a 2-tuple"
    if (not (0 <= coordinate[0] < self.width) or
        not (0 <= coordinate[1] < self.height)):
        raise CoordinateError(str(coordinate))
    return self.data.get(tuple(coordinate), self.background)
```

- This method returns the color for a given coordinate
  - using the item access operator `[]`
    - as in `image[x, y]`
- The coordinates are validated in two ways:
  - an `assert` ensures coordinates are sequences of length 2
    - typically a 2-tuple
  - a condition ensures coordinates are within the image itself
    - otherwise an exception is raised

Creating Classes That Aggregate Collections

- The definition of `__getitem__` is as follows:

```
def __getitem__(self, coordinate):
    assert len(coordinate) == 2, "coordinate should be a 2-tuple"
    if (not (0 <= coordinate[0] < self.width) or
        not (0 <= coordinate[1] < self.height)):
        raise CoordinateError(str(coordinate))
    return self.data.get(tuple(coordinate), self.background)
```

- Regarding this instruction:
  `self.data.get(tuple(coordinate), self.background)`
  - it is using the method `dict.get()` to retrieve the color for a given coordinate
  - it is called with a default value, the value of the background color
- If the color for the coordinate has never been set
  - the background color is returned
  - instead of a `KeyError` exception being raised

Creating Classes That Aggregate Collections

- The definition of `__setitem__` is as follows:

```
def __setitem__(self, coordinate, color):
    assert len(coordinate) == 2, "coordinate should be a 2-tuple"
    if (not (0 <= coordinate[0] < self.width) or
        not (0 <= coordinate[1] < self.height)):
        raise CoordinateError(str(coordinate))
    if color == self.background:
        self.data.pop(tuple(coordinate), None)
    else:
        self.data[tuple(coordinate)] = color
        self.colors.add(color)
```

- If the user sets a coordinate's value to the background color
  - we can simply delete the corresponding dictionary item
    - any coordinate not in the dictionary will have the background color
  - we are using `dict.pop()` for this, with a dummy second argument
    - to avoid raising a `KeyError` if the coordinate is not in the dictionary
- If the color for the coordinate has never been set
  - the background color is returned
  - instead of a `KeyError` exception being raised

Creating Classes That Aggregate Collections

- The definition of `__delitem__` is as follows:

```
def __delitem__(self, coordinate):
    assert len(coordinate) == 2, "coordinate should be a 2-tuple"
    if (not (0 <= coordinate[0] < self.width) or
        not (0 <= coordinate[1] < self.height)):
        raise CoordinateError(str(coordinate))
    self.data.pop(tuple(coordinate), None)
```

- This method deletes the color for a given coordinate
  - using the item access operator `[]`
    - as in `del image[x, y]`
- If a coordinate's color is deleted, it will become the background color
- Again, we use `dict.pop()` to remove the item
  - it will work correctly whether or not an item with the coordinates exists

Creating Classes That Aggregate Collections

Question:

Given the previous implementations of `__setitem__()` and `delitem__()__`, will the colors in the `colors` set always exactly match the colors in use in an image?

Creating Classes That Aggregate Collections

Question:

As we can not provide a representational form for an image, we have not implemented `__repr__()` nor `__str__()` in class `Image`.

Can we call `repr()` or `str()` on an `Image` object?
If so, what method are we really calling?

Creating Classes That Aggregate Collections

- We want users of the `Image` class to be able to save and load images
- Two methods are provided for this
  - `save()`
  - `load()`
- We have chosen to save the data by *pickling* it
  - this is a way of converting an object into a sequence of bytes;
  - The pickled object can be a collection data type
    - such as a list or a dictionary
- A pickle can be read back directly into a Python variable
  - we don't have to do any parsing or other interpretation ourselves
- For programs that are not purely for personal use,
  - it is best to create a custom file format specific to the program;
  - We will come back to this later;

Creating Classes That Aggregate Collections

- The definition of save is as follows:

```
def save(self, filename=None):
    if filename is not None:
        self.filename = filename
    if not self.filename:
        raise NoFilenameError()

    fh = None
    try:
        data = [self.width, self.height, self.background, self.data]
        fh = open(self.filename, "wb")
        pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)
    except (EnvironmentError, pickle.PicklingError) as err:
        raise SaveError(str(err))
    finally:
        if fh is not None:
            fh.close()
```

- The first part of the function is concerned with the filename
    - if the filename of the object has not yet been set,
        - the method expects it as argument
        - if it is not given, an exception is raised

Creating Classes That Aggregate Collections

Regarding the definition of save,

- We create a list, called `data`, to hold the objects we want to save
  - This includes the dictionary `self.data` of coordinate-color items
  - But excludes the set of unique colors
    - since that data can later be reconstructed
- Then, we open the file to write in binary mode
- And we call function `pickle.dump()` to write the data object to the file
  - The pickle module can serialize data using various formats/protocols
  - The chosen protocol should be given as third argument in the call
    - Protocol 0 is ASCII and useful for debugging
    - We have used protocol 3, a compact binary format, which is why we had to open the file in binary mode

Creating Classes That Aggregate Collections

- The definition of `load` is as follows:

```
def load(self, filename=None):
    if filename is not None:
        self.filename = filename
    if not self.filename:
        raise NoFilenameError()

    fh = None
    try:
        fh = open(self.filename, "rb")
        data = pickle.load(fh)
        (self.width, self.height, self.background,
         self.data) = data
        self.colors = (set(self.data.values()) | {self.background})
    except (EnvironmentError, pickle.UnpicklingError) as err:
        raise LoadError(str(err))
    finally:
        if fh is not None:
            fh.close()
```

- The function starts off the same as `save()`
  - to handle the name of the file to load

Creating Classes That Aggregate Collections

Regarding the definition of load,

- The file to load from must be opened in read binary mode
- The data is read using the single statement
    - `data = pickle.load(fh)`
- The `data` object is an exact reconstruction of the one we saved
    - we use tuple unpacking
        - to assign each of the `data` list's items to the appropriate variable
- We observe that `self.colors` is reconstructed from the saved data

Creating Classes That Aggregate Collections

- Finally, the definition of `export` is as follows:

```
def export(self, filename):
    if filename.lower().endswith(".xpm"):
        self.__export_xpm(filename)
    else:
        raise ExportError("unsupported export format: " +
                          os.path.splitext(filename)[1])
```

- This is a generic export method
  - that uses the file extension to determine which method to call
- For now, it only supports saving to `.xpm` files
  - and it should be extended to support other formats;
  - Saving in `.xpm` format is provided by function `self.__export_xpm`
    - that we do not show, since it is not really relevant for now;
    - Its definition is provided in the attached file.

Creating Classes That Aggregate Collections

Final notes on the Image class

- It shows a typical class to hold program-specific data
- Providing access to the data items it contains
- And the ability to save and load its data to and from disk
- Providing only the essential methods it needs