Objects and Classes

CS111 Computer Programming
Department of Computer Science
Wellesley College

The Big Picture

Some paradigms of programming:

- **Imperative Programming**
  - uses explicit loops, conditionals, variables

- **Functional Programming**
  - Uses functions as building blocks (e.g., map, filter)

- **Object Oriented Programming (OOP)**
  - uses objects and their interactions to design programs.

**Today**: Study objects with simple examples

**Next time**: Use objects in animations

---

**Objects**

An **object** is a **data value** that has **state** and **behaviors**.

Examples:

- Python builtin types: numbers, strings, lists, tuples, dictionaries,
  - cs1graphics: circles, canvases, points, ...

Object **behaviors** are defined by **methods** that can be invoked on an object. A method is a function that has direct access to the state of an object.

For example, methods of a string object include:

- "Computer Science".lower()
- "Computer Science".split()
- "Computer Science".index('c')

Different kinds of objects can handle the same method in different ways.

For example, .scale(s)

- scales the radius of a cs1graphics Circle object by s
- scales the width and height of a cs1graphics Rectangle object by s

---

**Classes**

A **class** defines the characteristics of a set of objects or **instances**, including:

- **Representation of State**:
  - how each object's state is stored as **instance variables** in that object

- **Implementation of Behavior**:
  - how object behavior is implemented as code for each object's **methods**

A class is used as a template for making objects of one kind.

- An object made from a class is called an **instance** of the class.

Example of classes include:

- `str`, `list`, `int`, `dict`
- `Canvas`, `Polygon`, `Circle`, `Layer`

Type `help(str)` or `help(Canvas)` to see all the methods of these classes.
Class Examples: Geometric Figures

Define geometric figure classes for circles, rectangles, etc. (Different from those in cs1graphics.)

State:
- The state of a circle is its radius and color
- The state of a rectangle is its width, height, and color
- The state of a square is its side length and color

Behavior:
- Methods that return the state elements
- Methods that change the state elements
- Methods to compute the perimeter & area of each figure

Unlike cs1graphics, we will not draw these figures.

Object Memory Diagrams

We call our classes Circ and Rect to distinguish them from the cs1graphics Circle and Rectangle classes.

Objects drawn as boxes named by class name containing instance variables (also known as attributes, members, or fields)

Distinct instances of one class have the same methods and instance variable names, but they each hold distinct instance variable contents.

Big idea #1: Abstraction... for Objects

Client
Use object methods to inspect or manipulate abstract state.

Implementation
Concrete state (instance variables) and behavior (method code)

Contract / API
Methods
- .area()
- .getColor()
- .setWidth(x)
- .setHeight(y)

class Circ:
- .radius
- .color

To Notice
- The constructor name starts in uppercase; it has the name of the class.
- Some methods only return a value, they are known as getter methods.
- Some methods change values, are known as setter methods.

Rect Methods

Constructor function invocation

s = Rect(10, 17, "blue")

Method invocation

s.setWidth()  10
s.setHeight()  17
s.setColor()   "blue"

To Notice
- Some methods change the state of an object rather than returning something
Defining a Rect Class

```python
def __init__(self, w, h, c):
    self.width = w
    self.height = h
    self.color = c
```

Converting a Constructor Function

```
Rect(10, 17, "blue")
```

1. Invoke `__init__` method with `self` bound to new empty object for the class and other parameters bound to constructor function arguments.

Starting execution of `__init__` method.

```python
self.width = w
self.height = h
self.color = c
```

To Notice
- The special method `__init__` has two underscores at the start and end.
- The constructor function calls implicitly `__init__`.
- Although the `__init__` definition has four parameters, during the invocation we provide only three arguments. The special variable “self” doesn’t receive a value explicitly.

Invoking a Constructor Function

```python
invoking Rect w:
```

```python
Rect(10, 17, "blue")
```

2. Execute statements in body of `__init__` method.
   These may add new instance variables to `self` object.

Invoking a Constructor Function

```
Rect(10, 17, "blue")
```
Invoking a Constructor Function

Rect(10, 17, "blue")

2. Execute statements in body of __init__ method.
   These may add new instance variables to self object.

```
self.width = w
self.height = h
self.color = c
```

Invoking a Constructor Function

Rect(10, 17, "blue")

2. Execute statements in body of __init__ method.
   These may add new instance variables to self object.

```
self.width = w
self.height = h
self.color = c
```

Important: in this example, parameter names are different from instance variable names in the object (w vs self.width).

You could name them the same, (width instead of w), but the functionality would remain the same.

Invoking a Constructor Function

Rect(10, 17, "blue")

3. Object is returned as the result of constructor invocation

Invoking an Instance Method: getWidth

s = Rect(10, 17, "blue")

```
def getWidth(self):
    return self.width
```

s.getWidth()

Why self? Behavior is associated with an object.

• Invoking a method sends a message (getWidth) to the receiver object.
• The receiver object responds by running its getWidth method on itself.
• A method is just a function, but needs an argument indicating the receiver object.
• The receiver is a special argument that also determines what getWidth method is used.
• The receiver argument is written differently to distinguish it from other arguments.
Invoking an Instance Method: `getWidth`

1. `s = Rect(10, 17, "blue")`

2. Statements in the body are executed

3. Value is returned as result of method invocation.

```python
def getWidth(self):
    return self.width

s = Rect(10, 17, "blue")
s.getWidth()  # 10
```

Invoking an Instance Method: `setWidth`

1. `s = Rect(10, 17, "blue")`

2. Statements in the body are executed

3. Value is returned as result of method invocation.

```python
def setWidth(self, newWidth):
    self.width = newWidth

s = Rect(10, 17, "blue")
s.setWidth(30)  # s.width = 30
```
Invoking an Instance Method: `setWidth`

```python
def setWidth(self, newWidth):
    self.width = newWidth
```

```python
s = Rect(10, 17, "blue")
s.setWidth(30)
```

3. Here, nothing is returned as result of method invocation since there is no `return` statement.

Define the entire `Rect` class

```python
class Rect:
    def __init__(self, w, h, c):
        self.width = w
        self.height = h
        self.color = c
    def getWidth(self):
        return self.width
    def getHeight(self):
        return self.height
    def getColor(self):
        return self.color
    def setWidth(self, newWidth):
        self.width = newWidth
    def setHeight(self, newHeight):
        self.height = newHeight
    def setColor(self, newColor):
        self.color = newColor
    def area(self):
        return self.width * self.height
    def perimeter(self):
        return 2 * (self.width + self.height)
    def scale(self, factor):
        self.width *= factor
        self.height *= factor
```

To Notice

- All methods of the class have the special variable `self` as their first parameter.
- Instance variables are accessed through the special variable `self` in all methods.
- We will normally have one line of space after each function definition, but here it is omitted due to lack of space.
- Additionally, docstrings are missing for the same reason.

___`repr`___ method

- What happens if you print a `Rect` object? (try this in the notebook)
  ```python
  s = Rect(10, 17, "blue")
  print s
  ```

- We’d like to control how the object is represented when displayed. This is done with the `__repr__` method
  - Method names with `__` around them are handled by Python in a special way
  - `__repr__` always takes only `self` as an argument and returns a string
  ```python
  def __repr__(self):
      return 'Rect <width: {}, height: {}, color: {}>'.format(
          self.width, self.height, self.color)
  ```

Now define the `Circ` class in the notebook

```python
class Circ:
    def __init__(self, radius, color):
        """make a circle with the given radius and color""
    def getRadius(self):
        """return radius of circle""
    def getColor(self):
        """return color of circle""
    def setRadius(self, radius):
        """change radius""
    def setColor(self, color):
        """change color""
    def area(self):
        """return area of circle""
    def perimeter(self):
        """return perimeter (circumference)""
    def scale(self, factor):
        """multiply the circle radius by factor""
    def __repr__(self):
        """string representation of circle""
```

To Notice

- All methods of the class have the special variable `self` as their first parameter.
- Instance variables are accessed through the special variable `self` in all methods.
- We will normally have one line of space after each function definition, but here it is omitted due to lack of space.
- Additionally, docstrings are missing for the same reason.
So far in this lecture:

- **Objects** are data values with **state** (stored in instance variables) and **behavior** (implemented as methods).
- **Classes** are templates for creating objects of one kind.
- Class definitions start with the keyword **class** and all the methods included in the body of the class.
- To **initialize** an object we invoke the **constructor**, a function with the class's name, which creates a new object instance, invokes **_init_** on it, and returns the new instance.
- The first parameter of a method definition is the special variable **self**.

**Coming up**

- **Polymorphism**: Code can use any kind of object with the same methods.
- **Why not dictionaries?**
  - Create your own **MutableString** class.
  - **Inheritance**: Classes can inherit methods from other classes to share code when needed.

---

**Big Idea #1+2: Abstraction and Modularity...**

A well defined abstraction allows multiple implementations.

A well defined abstraction makes it easy to use alternative implementation later.

**Contract / API:**

**Methods**

- **Diameter-based Circ**
  - `__init__` (diam, color)
  - `__str__` (diam, color)
  - `area()` (diam)

- **Radius-based Circ**
  - `__init__` (radius, color)
  - `__str__` (radius, color)
  - `area()` (radius)

**Implementation**

```python
# diameter-based Circ
class Circ:
    def __init__(self, diam, color):
        self.diam = diam
        self.color = color

def area(self):
    return math.pi * (self.diam/2)**2

# radius-based Circ
class Circ:
    def __init__(self, radius, color):
        self.radius = radius
        self.color = color

def area(self):
    return math.pi * self.radius**2
```

**Client**

```python
s = Circ(...) # ...
print s.area()
```

Works for any representation of Circ. No change needed in client code when Circ implementation changes.

---

**Why not Dictionaries?**

**One attempt**: what if we wish to swap Circ implementations?

```python
def makeRectDict(w, h, c):
    return {'figType': 'Rect', 'width': w, 'height': h, 'color': c}
def makeCircDict(r, c):
    return {'figType': 'Circ', 'radius': r, 'color': c}
def setWidth(fig, width): # call with setWidth(myFig, 3), vs. myFig.setWidth(3)
    if fig['figType'] == 'Rect':
        return fig['width'] = width
    elif fig['figType'] == 'Circ':
        raise Exception('Circ does not support setWidth')
    # ... Add one branch for every type of shape.

def area(fig): # call with area(myFig), vs. myFig.area()
    if fig['figType'] == 'Rect':
        return fig['width'] * fig['height']
    elif fig['figType'] == 'Circ':
        return math.pi * fig['radius']**2
    # ... Add one branch for every type of shape.
```
Python Strings are Immutable

In [11]: s = 'seed'

In [12]: s[2] = 'n'
--------------------------------------------------
TypeError: 'str' object does not support item assignment

We can create new strings that are based on existing strings:

In [13]: s2 = s[:2] + 'n' + s[3:]

In [14]: s2
Out[14]: 'send'

But we cannot modify an existing string.

Let’s Define a Mutable String Class

Create a `MutableString` class that supports the following behaviors:

- getting a character at a given index
- changing a character at a given index to a new value
- getting the length of the mutable string
- reversing the mutable string
- returning a string representation of the mutable string

```python
c = MutableString('deer')
print c
deer
print c.length() 4
print c.getchar(3) r
print c.reverse()
print c  reed
print c.setchar(0, 's')
print c  seed
print c.setchar(2, 'n')
print c  send
```

Mutable String Class

```python
class MutableString:
    def __init__(self, data):
        pass
    def length(self):
        pass
    def getchar(self, index):
        pass
    def setchar(self, index, c):
        pass
    def reverse(self):
        pass
    def __repr__(self):
        pass
```

Big Idea #1+2: Abstraction and Modularity… support alternative implementations

A well defined abstraction allows multiple implementations.

A well defined abstraction makes it easy to use alternative implementation later.

Client

```python
s = MutableString("hello world")
s.setchar(0, 'c')
print s
"cello world"
```

Implementation

String-based MutableString

```python
<table>
<thead>
<tr>
<th>Contract / API: Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>area()</td>
</tr>
</tbody>
</table>

Works for both implementations of MutableString.

List-based MutableString

```python
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```
**Special Syntax & Method Names for Collection Methods**

Python translates special syntax for familiar collection operators into method names that involve __. For example:

- `len(coll)` results in `coll.__len__()`
- `coll[index]` results in `coll.__getitem__(index)`
- `coll[index] = newval` results in `coll.__setitem__(index, newval)`
- `coll1 + coll2` results in `coll1.__add__(coll2)`
- `elt in coll` results in `coll.__contains__(elt)`

So in `MutableString`, if we rename `length` to `__len__`, `getchar` to `__getitem__`, and `setchar` to `__setitem__`, we can use the special syntax:

```python
s = MutableString('deer')
print(len(s))
s[3] = 'p'
print(s)
print(s[3])
```

---

**A hierarchy of figures**

- Notice that `Circ` and `Rect` have:
  - Methods with the same name but different implementations: `area`, `perimeter`, `scale`
  - Methods with the same name AND the same implementation: `getColor`, `setColor`
  - Methods that are unique to each class in name as well as implementation: `getRadius`, `getRadius`, `getWidth`, `setWidth`, `getHeight`, `setHeight`

- This will be the case for other figure classes we might define too (Square, Triangle, etc.)

- How can we follow the DRY (Don't Repeat Yourself) principle?

---

**Inheritance**

*Inheritance* is a mechanism for abstracting over data and behaviors common to different classes.

When class B *inherits* from class A, class B acquires all of the attributes (instance variables and methods) from class A.

New attributes (instances variables and methods) can be added to B. B is called the *child* class or the *subclass*.

A is called the *parent* class or the *superclass*. 

---
Child Classes
Squares are a specific kind of rectangle

```python
class Squ(Rect):
    # inherits all Rect methods
    # plus can define extra ones
    def getSideLen(self):
        return self.width
    def setSideLen(self, sideLen):
        self.width = sideLen
        self.height = sideLen
```

Abstraction:
User of Squ does not need to know that it uses width and height

A subclass method with the same name as an inherited superclass method replaces or “overrides” the inherited method.

```python
def __init__(self, sideLen, color):
    # use the Rect constructor
    Rect.__init__(self, sideLen, sideLen, color)
```

Parent Classes:
Define a class Fig to capture common state and behavior
Rect and Circ will inherit from Fig (see notebook)

```python
class Fig:
    def __init__(self, color):
        self.color = color
    def getColor(self):
        return self.color
    def setColor(self, color):
        self.color = color
    def area(self):
        pass
    def perimeter(self):
        pass
    def scale(self, factor):
        pass
```

Circ and Rect can now inherit from Fig

```python
class Circ(Fig):
    def __init__(self, radius, color):
        Fig.__init__(self, color)
        self.radius = radius
        self.color = color
        def area(self):
            return pi*self.radius**2
        def perimeter(self):
            return 2*pi*self.radius
        def scale(self, factor):  
            self.radius *= factor
```

```python
class Rect(Fig):
    def __init__(self, width, height, color):
        Fig.__init__(self, color)
        self.width = width
        self.height = height
```

Concepts in this slide:
Code in blue shows repetition.

Parent Classes:
Rect and Circ have shared method names and definitions
Inheritance: 
cs1graphics

Classes are often defined in inheritance hierarchies to maximize sharing and reuse of behavior (method code).

Benefits of Object-Oriented Programming

Modularity:
Common states and behaviors packaged up

Polymorphism:
Same method can do different things for different types of objects (e.g., `perimeter` for `Fig`)

Encapsulation:
Hide details of how we store the object's information (e.g., different implementations of `MutableString`)
- Not actually enforced in Python.
- Java supports private variables and methods that can be used only within methods of the class.

Inheritance:
Subclasses share states and behaviors of superclasses (avoiding repetition of code)