Objects and Classes

The Big Picture

Some paradigms of programming:

- Imperative Programming
- Functional Programming
- **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 built-in types: strings, lists, tuples, dictionaries, numbers
- 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.
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.

Big Idea #1: Abstraction... for Objects

Client
Use object methods to inspect or manipulate abstract state.

shape
shape.area() \(\rightarrow\) 12.56

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

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

Object Memory Diagrams

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

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

```
r = Rect(2, 1, "red")
disk = Circ(50, "yellow")
s = Rect(10, 17, "blue")
```

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

Rect Methods

Constructor function invocation

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

Method invocation

```
s.setWidth() \rightarrow 10
s.getHeight() \rightarrow 17
s.setColor() \rightarrow "blue"
s.perimeter() \rightarrow 54
s.area() \rightarrow 170
```

Some methods change the state of an object rather than returning something

```
s.setWidth(30)
s.setHeight(25)
```
**Defining a Rect Class**

```python
class Rect:
    '''A rectangle, specified by width, height, and color'''

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

def getWidth(self):
    """"""Returns the rectangle's width"""
    return self.width
```

First argument for any instance method is the receiver object that the method is invoked upon.

Keyword `__init__` is special method automatically invoked each time a new object is created.

**Invoking a Constructor Function**

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

How to invoke the constructor:

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

**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

```
Rect(width=10, height=17, color="blue")
```

**Invoking an Instance Method: getWidth**

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

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

s.getWidth()

1. Invoke `getWidth` method with `self` bound to the receiver object argument.
2. Statements in the body are executed
   ```python
   def getWidth(self):
       return self.width
   ```
3. Value is returned as result of method invocation.

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: `setWidth`

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

1. Invoke `setWidth` method with `self` bound to receiver object and `newWidth` bound to the argument

```python
s.setWidth(30)
```

2. Statements in the body are executed

```python
self.width = 30
```

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

```python
Define the entire `Rect` class:
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
```

```python
defs = Rect(10, 17, "blue")
defs.setWidth(30)
```
**__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
  ```python
def __repr__(self):
    return 'Rect <width: {}, height: {}, color: {}>'.format(
        self.width, self.height, self.color)
  ```

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 in the body of the class.
- To instantiate 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 uses objects can use any kind of object that provides the right methods, regardless of that object's internals.

- **Why not dictionaries?**
- Create your own `MutableString` class.
- **Inheritance**: Classes can inherit methods from other classes to share and reuse behavior and code.

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(factor):
        """multiply the circle radius by factor""
    def __repr__(self):
        """string representation of circle""
```

Big Idea #1+2: Abstraction and Modularity… for Polymorphism (1)

Code that interacts with an object by invoking its methods can use any object that provides the right methods, regardless of that object's internals.
A well defined abstraction allows multiple implementations. A well defined abstraction makes it easy to use alternative implementation later.

**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:]  # 'send'
In [14]: s2
Out[14]: 'send'
```

But we cannot modify an existing string.

```
In [15]: s[2] = 'n'
-----------------------------------------------------------------------------
TypeError: 'str' object does not support item assignment
```

**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):
    if fig['figType'] == 'Rect':
        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):
    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.
```

```
20-28
```

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

```python
class MutableString:
    def __init__(self, data):  # make a MutableString
        # make a MutableString
    def length(self):  # return number of chars
        # return number of chars
    def getchar(self, index):  # return char at index
        # return char at index
    def setchar(self, index, c):  # change char at index to c
        # change char at index to c
    def reverse(self):  # reverse chars of object
        # reverse chars of object
    def __repr__(self):  # string rep of object
        # string rep of object
```

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)
```

Implementation

![Implementation Diagram]

Works for both implementations of MutableString.

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?

Special Syntax & Method Names for Collection Methods

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

- `len(coll) ⇒ coll.__len__()`
- `coll[index] ⇒ coll.__getitem__(index)`
- `coll[index] = newval ⇒ coll.__setitem__(index, newval)`
- `coll1 + coll2 ⇒ coll1.__add__(coll2)`
- `elt in coll ⇒ 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]
```
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
Squ (square) are a specific kind of Rect (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

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

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

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

Parent Classes
Rect and Circ have shared method names and definitions

```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 setSize(self, w):
        self.width = w
    def setColor(self, c):
        self.color = c
    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

class Circ:
    def __init__(self, rad, c):
        self.radius = rad
        self.color = c
    def getRadius(self):
        return self.radius
    def getColor(self):
        return self.color
    def setSize(self, rad):
        self.radius = rad
    def setColor(self, c):
        self.color = c
    def area(self):
        return pi*self.radius**2
    def perimeter(self):
        return 2*pi*self.radius
    def scale(self, factor):
        self.radius *= factor
```
Parent Classes:
Define a class Fig to capture common state and behavior
Rect and Circ will inherit from Fig (see notebook)

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

class Circ(Fig):
    def __init__(self, radius, color):
        Fig.__init__(self, color)
        self.radius = radius
        # ... other definitions the same, but no need
        #  to define getColor and setColor ...

class Rect(Fig):
    def __init__(self, width, height, color):
        Fig.__init__(self, color)
        self.width = width
        self.height = height
        # ... other definitions the same, but no need
        #  to define getColor and setColor ...

Inheritance:

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, step for Sprite)

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)