Problem Solving: Patterns for Object Data

- Common patterns when designing instance variables
  - Keeping a Total
  - Counting Events
  - Collecting Values
  - Managing Object Properties
  - Modeling Objects with Distinct States
  - Describing the Position of an Object

Patterns: Keeping a Total

- Examples
  - Bank account balance
  - Cash Register total
  - Car gas tank fuel level
- Variables needed
  - totalPrice
- Methods Required
  - add (addItem)
  - clear
  - getTotal

```
class CashRegister :
    def addItem(self, price):
        self._itemCount =
           self._itemCount + 1
        self._totalPrice =
           self._totalPrice + price

    def clear(self):
        self._itemCount = 0
        self._totalPrice = 0.0

    def getTotal(self):
        return self._totalPrice
```

Patterns: Counting Events

- Examples
  - Cash Register items
  - Bank transaction fee
- Variables needed
  - itemCount
- Methods Required
  - Add
  - Clear
  - Optional: getCount

```
class CashRegister:
    def addItem(self, price):
        self._itemCount =
           self._itemCount + 1
        self._totalPrice =
           self._totalPrice + price

    def clear(self):
        self._itemCount = 0
        self._totalPrice = 0.0

    def getCount(self):
        return self._itemCount
```

Patterns: Collecting Values

- Examples
  - Multiple choice question
  - Shopping cart
- Storing values
  - List
- Constructor
  - Initialize to empty collection
- Methods Required
  - Add

```
class Cart:
    def _ _init_ _(self) :
        self._choices = []

    def addItem(self, name) :
       self._choices.append
          (choice)
```

Patterns: Managing Properties

A property of an object can be set and retrieved

- Examples
  - Student: name, ID
- Constructor
  - Set a unique value
- Methods Required
  - set
  - get

```
class Student :
    def _ _init_ _
       (self, aName, anId) :
        self._name = aName
        self._id = anId

    def getName(self) :
        return self._name

    def setName(self, newName) :
        self._name = newName

    def getId(self) :
        return self._id

    # No setId method
```

Patterns: Modeling Object States

Some objects can be in one of a set of distinct states

- Example: A fish
  - Hunger states:
    - Not Hungry
    - Somewhat Hungry
    - Very Hungry
- Methods will change the state
  - eat
  - move

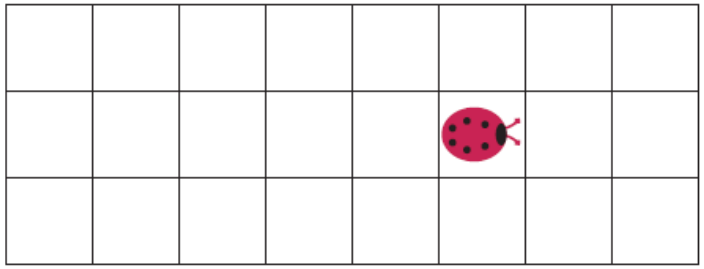
```
class Fish:
    NOT_HUNGRY = 0
    SOMEWHAT_HUNGRY = 1
    VERY_HUNGRY = 2

    def eat(self) :
        self._hungry =
        Fish.NOT_HUNGRY

    def move(self) :
        if self._hungry <
          Fish.VERY_HUNGRY :
           self._hungry =
           self._hungry + 1
```

Patterns: Object Position

- Examples
  - Game object
  - Bug (on a grid)
  - Cannonball
- Storing values
  - Row, column, direction, speed. . .
- Methods Required
  - move
  - turn



```
class Bug:
    def _ _init_ _
        (self, aRow, aColumn,
         aDirection, speed) :
         self._row = aRow
         self._column = aColumn
         self._direction =
            direction
# 0 = N, 1 = E, 2 = S, 3 = W
         . . .

  def moveOneUnit(self):
      if (self._direction == 0):
          self._row =
             self._row - 1
      . . .
```

Self-check

A shuttle bus drives along Main Street, starting at the intersection of Main and First and going up to Main and Twentieth, then turning around. Each call to the drive method moves the bus by one block. Rearrange the following lines. First list the constructor (which yields a bus at First Street heading towards Twentieth Street). Then list the methods in alphabetical order.

```
return self._street
```

```
self._direction = 1
```

```
self._street = 1
```

```
def getLocation(self) :
```

```
self._direction = -self._direction
```

```
self._street = self._street + self._direction
```

```
def __init__(self) :
```

```
class Bus :
```

```
if (self._street == 20 and self._direction ==
1) or (self._street == 1 and self._direction
== -1) :
```

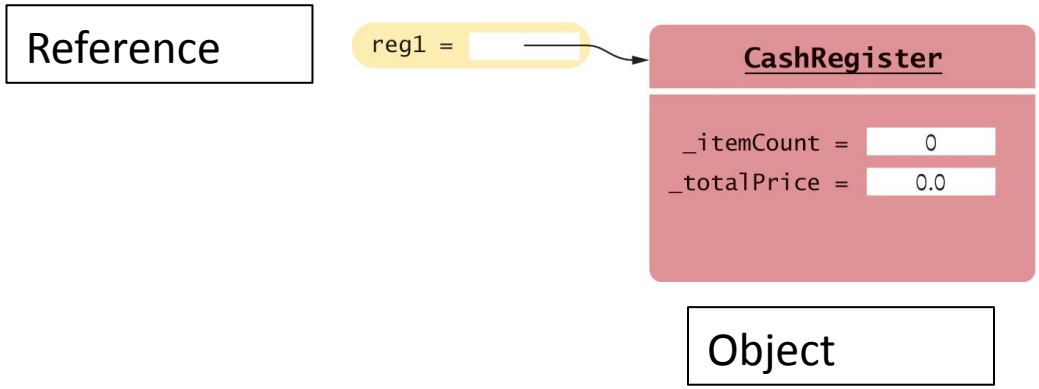
```
def drive(self) :
```

Object References

- In Python, a variable does not actually hold an object
- It merely holds the *memory location* of an object
- The object itself is stored in another location:

```
reg1 = CashRegister()
```

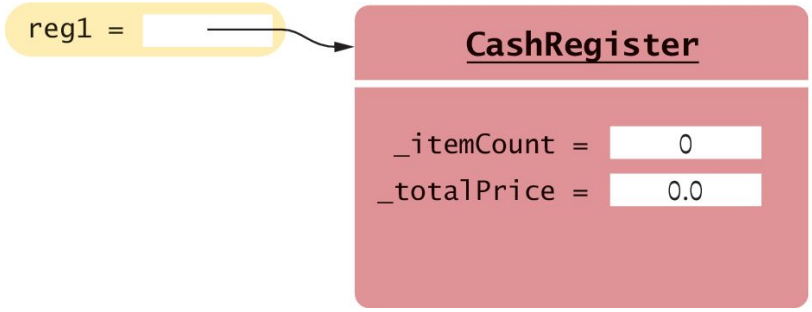
The constructor returns a reference to the new object, and that reference is stored in the reg1 variable.



Shared References

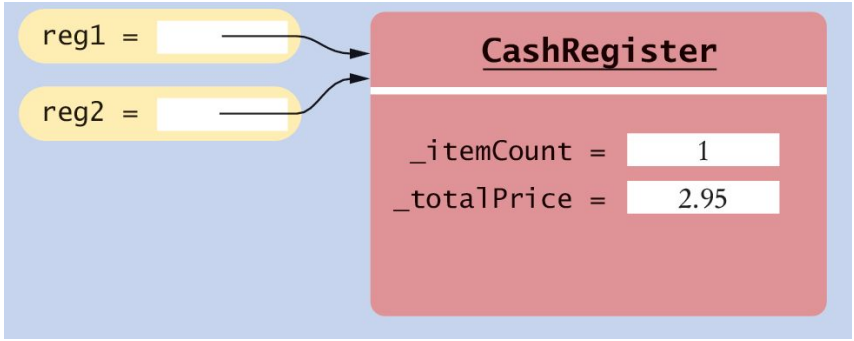
- Multiple object variables may contain references to the same object ('aliases')
  - Single Reference

```
reg1 = CashRegister
```



  - Shared References

```
reg2 = reg1
```



The internal values can be changed through either reference

Testing if References are Aliases

- Checking if references are aliases, use the `is` or the `not is` operator:

```
if reg1 is reg2 :
    print("The variables are aliases.")
if reg1 is not reg2 :
    print("The variables refer to different objects.")
```

- Checking if the data contained within objects are equal use the `==` operator:

```
if reg1 == reg2 :
    print("The objects contain the same data.")
```

The None reference

- A reference may point to 'no' object
  - You cannot invoke methods of an object via a None reference – causes a run-time error

```
reg = None
print(reg.getTotal())    # Runtime Error!
```

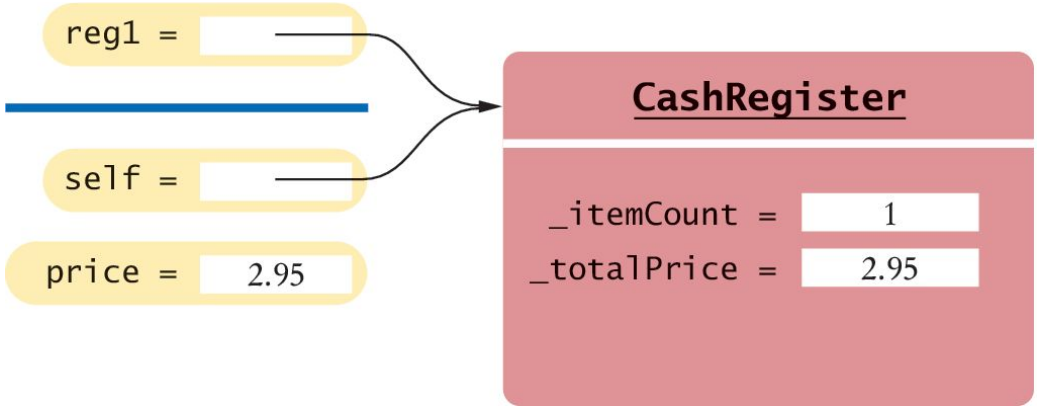
  - To test if a reference is None before using it:

```
middleInitial = None    # No middle initial

if middleInitial is None :
  print(firstName, lastName)
else :
  print(firstName, middleInitial + ".", + lastName)
```

The `self` reference

- Every method has a reference to the object on which the method was invoked, stored in the `self` parameter variable
  - It is a reference to the object the method was invoked on:



  - It can clarify when instance variables are used:

```
def addItem(self, price):
  self.itemCount = self.itemCount + 1
  self.totalPrice = self.totalPrice + price
```

Using `self` to Invoke Other Methods

- You can also invoke a method on `self`:

```
def _ _init_ _(self) :
    self.clear()
```

  - In a constructor, `self` is a reference to the object that is being constructed
  - The `clear()` method is invoked on that object

Passing `self` as a Parameter

- Suppose, for example, you have a `Person` class with a method `likes(self, other)` that checks, perhaps from a social network, whether a person likes another

```
def isFriend(self, other) :
    return self.likes(other) and other.likes(self)
```

Object Lifetimes: Creation

- When you construct an object with a constructor, the object is created, and the `self` variable of the constructor is set to the memory location of the object
  - Initially, the object contains no instance variables.
  - As the constructor executes statements such as instance variables are added to the object

```
self._itemCount = 0
```

- Finally, when the constructor exits, it returns a reference to the object, which is usually captured in a variable:

```
reg1 = CashRegister()
```

Object Lifetimes: Cleaning Up

- The object, and all of its instance variables, stays alive as long as there is at least one reference to it.
- When an object is no longer referenced at all, it is eventually removed by a part of the virtual machine called the “garbage collector”

```
reg1 = CashRegister()   # New object referenced by reg1
reg1 = CashRegister()
   # Another object referenced by reg1
   # First object will be garbage collected
```

Writing a Fraction Class

- So far we have worked with floating-point numbers but computers store binary values, so not all real numbers can be represented precisely
- In applications where the precision of real numbers is important, we can use *rational numbers* to store exact values
  - This helps to reduce or eliminate round-off errors that can occur when performing arithmetic operations
  - A rational number is a number that can be expressed as a ratio of two integers: 7/8
  - The top value is called the *numerator* and the bottom value, which cannot be zero, is called the *denominator*

Designing the Fraction Class

- We want to use our rational numbers as we would use integers and floating point values
- Thus, our `Fraction` class must perform the following operations:
  1. Create a rational number
  2. Access the numerator and denominator values, individually
  3. Determine if the rational number is negative or zero
  4. Perform normal mathematical operations on two rational numbers (addition, subtraction, multiplication, division, exponentiation)
  5. Logically compare two rational numbers
  6. Produce a string representation of the rational number
- The objects of the Fraction class will be **immutable** because none of the operations modify the objects' instance variables

Required Data Attributes

- Because a rational number consists of two integers, we need two instance variables to store those values:

```
self._numerator = 0
self._denominator = 1
```

- At no time should the rational number be converted to a floating-point value or we will lose the precision gained from working with rational numbers

Representing Values Equivalently

- Signed values
  - Negative and positive rational numbers each have two forms that can be used to specify the corresponding value
  - Positive values can be indicated as $1/2$ or $-1/-2$, and negative values as $-2/5$ or $2/-5$
  - When performing an arithmetic operation or logically comparing two rational numbers, it will be much easier if we have a single way to represent a negative value
  - For simplicity, we choose to set only the numerator to a negative value when the rational number is negative, and both the numerator and denominator will be positive integers when the rational number is positive

Representing Values Equivalently

- Equivalent fractions
  - For example, 1/4 can be written as 1/4, 2/8, 16/64, or 123/492
  - It will be much easier to perform the operation if the number is stored in reduced form

The Constructor (1)

- Because `Fraction` objects are immutable, their values must be set when they are created. This requires parameter variables for both the numerator and denominator

```
def _ _init_ _(self, numerator, denominator) :
```

- The method must check for special cases:
  - Zero denominators
  - The number represents zero or a negative number

The Constructor

```
def _ _init_ _(self, numerator = 0, denominator = 1) :
    if denominator == 0 :
        raise ZeroDivisionError("Denominator cannot be zero.")
    if numerator == 0 :
        self._numerator = 0
        self._denominator = 1
    else :
        if (numerator < 0 and denominator >= 0 or
            numerator >= 0 and denominator < 0) :
           sign = -1
        else :
           sign = 1
```

The Constructor

```
a = abs(numerator)
b = abs(denominator)
while a % b != 0 :
    tempA = a
    tempB = b
    a = tempB
    b = tempA % tempB
self._numerator = abs(numerator)        # b * sign
self._denominator = abs(denominator)    #b
```

Testing the Constructor

```
frac1 = Fraction(1, 8) # Stored as 1/8
frac2 = Fraction(-2, -4) # Stored as 1/2
frac3 = Fraction(-2, 4) # Stored as -1/2
frac4 = Fraction(3, -7) # Stored as -3/7
frac5 = Fraction(0, 15) # Stored as 0/1
frac6 = Fraction(8, 0) # Error! exception is raised.
```

Comparing Fractions (1)

- In Python, we can define and implement methods that will be called automatically when a standard Python operator (+, *, ==, <) is applied to an instance of the class
- For example, to test whether two fractions are equal, we could implement a method:
  - `isequal()` and use it as follows:

```
if frac1.isequal(frac2) :
    print("The fractions are equal.")
```

Comparing Fractions (2)

- Of course, we would prefer to use the operator ==
- This is achieved by defining the special method:

  _ _eq_ _():

```
def _ _eq_ _(self, rhsValue) :
    return (self._numerator == rhsValue.numerator and
            self._denominator == rhsValue.denominator)
```

- Automatically calls this method when we compare two `Fraction` objects using the == operator:

```
if frac1 == frac2 : # Calls frac1._ _eq_ _(frac2)
    print("The fractions are equal.")
```