北京航空航天大學
BEIHANG UNIVERSITY

Lecture 5: Classes & Objects

Xiaoqian Sun / Sebastian Wandelt

Beihang University

Object-oriented programming

Today's Lecture Goals

- To understand the concepts of classes, objects and encapsulation
- To implement instance variables, methods and constructors
- To be able to design, implement, and test your own classes
- To understand the behavior of object references

You will learn how to discover, specify, and implement your own classes, and how to use them in your programs.

Object-Oriented Programming

- You have learned structured programming
  - Breaking tasks into subtasks
  - Writing re-usable methods to handle tasks
- We will now study Objects and Classes
  - To build larger and more complex programs
  - To model objects we use in the world

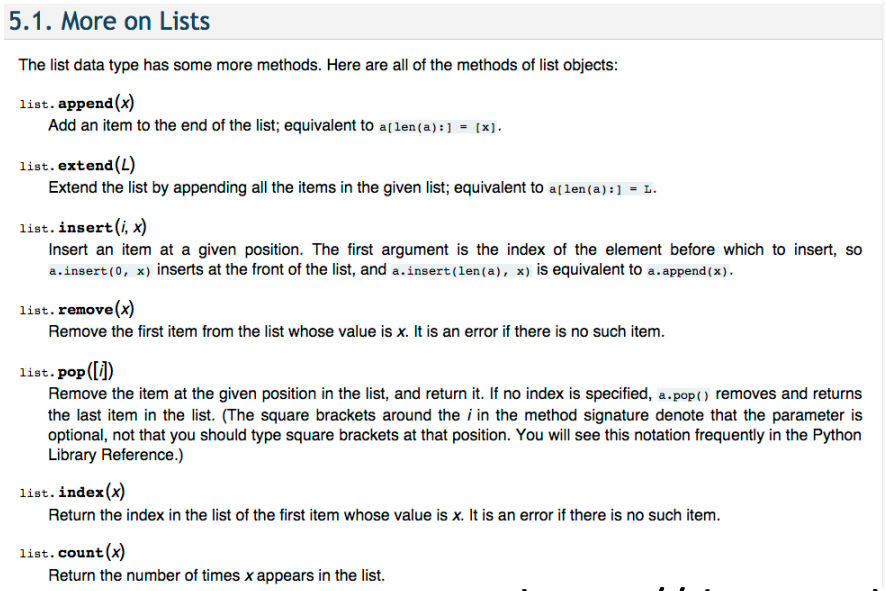
A class describes objects with the same behavior. For example, a Car class describes all passenger vehicles that have a certain capacity and shape.

Objects and Programs

- You have learned how to structure your programs by decomposing tasks into functions.
  - Experience shows that it does not go far enough. It is difficult to understand and update a program that consists of a large collection of functions.
- To overcome this problem, computer scientists invented **object-oriented programming**, a programming style in which tasks are solved by collaborating objects.
- Each object has its own set of data, together with a set of methods that act upon the data.

Objects and Programs

- You have already experienced this programming style when you used strings, lists, and file objects. Each of these objects has a set of methods.
- For example, you can use the `add()` or `len()` methods to operate on list objects.



5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**`(x)`
Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**`(L)`
Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.

`list.`**`insert`**`(i, x)`
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**`remove`**`(x)`
Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.`**`pop`**`([i])`
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

`list.`**`index`**`(x)`
Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.

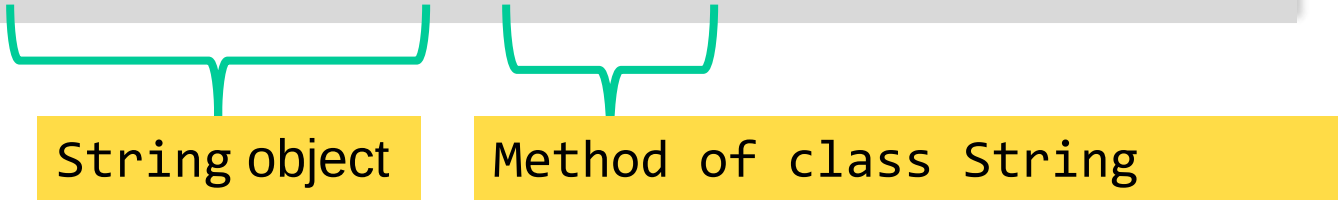
`list.`**`count`**`(x)`
Return the number of times *x* appears in the list.

https://docs.python.org/2/tutorial/datastructures.html

Python Classes

- A class describes a set of objects with the same behavior.
  - For example, the `str` class describes the behavior of all strings.
  - This class specifies how a string stores its characters, which methods can be used with strings, and how the methods are implemented.
  - For example, when you have a `str` object, you can invoke the `upper` method:

```
s="Hello, World"
s.upper()
```



Python Classes

- In contrast, the `list` class describes the behavior of objects that can be used to store a collection of values.
- This class has a different set of methods.
- For example, the following call would be illegal—the `list` class has no `upper()` method.

```
["Hello", "World"].upper()
```

- However, `list` has an `add()` method, and the following call is legal.

```
["Hello", "World"].add()
```

Public Interfaces

- The set of all methods provided by a class, together with a description of their behavior, is called the public interface of the class.
- When you work with an object of a class, you do not know how the object stores its data, or how the methods are implemented.
  - You do not need to know how a `str` object organizes a character sequence, or how a list stores its elements.
- All you need to know is the public interface—which methods you can apply, and what these methods do.

Public Interfaces

- The process of providing a public interface, while hiding the implementation details, is called **encapsulation**.
- If you work on a program that is being developed over a long period of time, it is common for implementation details to change, usually to make objects more efficient or more capable.
- When the implementation is hidden, the improvements do not affect the programmers who use the objects.

Implementing a Simple Class

- Example: Tally Counter: A class that models a mechanical device that is used to count people
  - For example, to find out how many people attend a concert or board a bus
- What should it do?
  - Increment the tally
  - Get the current total

Using the `Counter` Class

- First, we construct an object of the class (object construction will be covered shortly):
- When one first assigns a value to an instance variable, the instance variable is created.

```
tally = Counter()   # Creates an instance
```

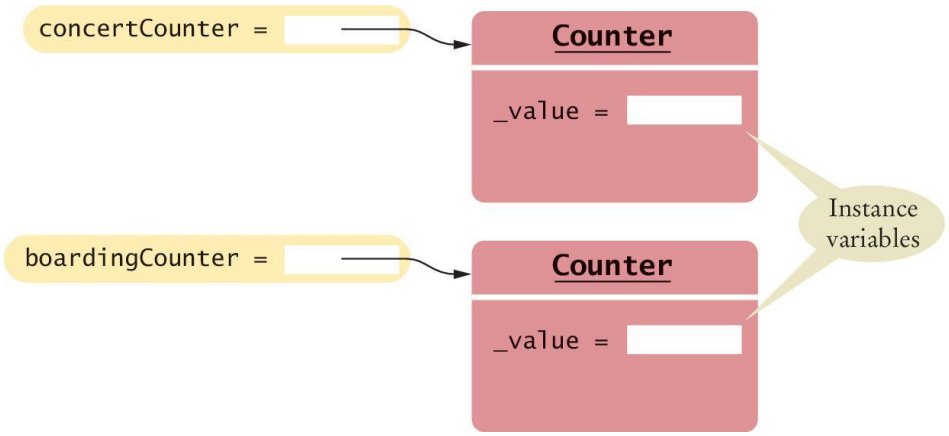
Using the `Counter` Class

- Next, we invoke methods on our object.

```
tally.reset()
tally.click()
tally.click()
result = tally.getValue()  # Result is 2
tally.click()
result = tally.getValue() # Result is 3
```

Instance Variables

- An object stores its data in **instance variables**.
- An *instance* of a class is an object of the class.
- In our example, each `Counter` object has a single instance variable named `_value`.
  - For example, if `concertCounter` and `boardingCounter` are two objects of the `Counter` class, then each object has its own `_value` variable



Instance Variables

- Instance variables are part of the implementation details that should be hidden from the user of the class.
  - With some programming languages an instance variable can only be accessed by the methods of its own class.
  - The Python language does not enforce this restriction.
  - However, the underscore indicates to class users that they should not directly access the instance variables.

Class Methods

- The methods provided by the class are defined in the class body.
- The `click()` method advances the `_value` instance variable by 1.

```
def click(self) :
    self._value = self._value + 1
```

- A method definition is very similar to a function with these exceptions:
  - A method is defined as part of a class definition.
  - The first parameter variable of a method is called `self`.

Example of Encapsulation

- The `getValue()` method returns the current `_value`:

```
def getValue(self) :
    return self._value
```

- This method is provided so that users of the `Counter` class can find out how many times a particular counter has been clicked.
- A class user should not directly access any instance variables. Restricting access to instance variables is an essential part of encapsulation.

Complete Simple Class Example

File: example.py

```
6  from counter import Counter
7
8  tally = Counter()
9  tally.reset()
10 tally.click()
11 tally.click()
12
13 result = tally.getValue()
14 print("Value:", result)
15
16 tally.click()
17 result = tally.getValue()
18 print("Value:", result)
```

File: counter.py

```
7  class Counter :
8     ## Gets the current value of this counter.
9     #  @return the current value
10    #
11    def getValue(self) :
12       return self._value
13
14    ## Advances the value of this counter by 1.
15    #
16    def click(self) :
17       self._value = self._value + 1
18
19    ## Resets the value of this counter to 0.
20    #
21    def reset(self) :
22       self._value = 0
```

Program execution

```
Value: 2
Value: 3
```

Public Interface of a Class

- When you design a class, start by specifying the public interface of the new class
  - What tasks will this class perform?
  - What methods will you need?
  - What parameters will the methods need to receive?
  - Example: A Cash Register Class

Task	Method
Add the price of an item	`addItem(price)`
Get the total amount owed	`getTotal()`
Get the count of items purchased	`getCount()`
Clear the cash register for a new sale	`clear()`

  - Since the '`self`' parameter is required for all methods it was excluded for simplicity.

Writing the Public Interface

```
## A simulated cash register that tracks the item count and the total amount due.
#
class CashRegister :
   ## Adds an item to this cash register.
   # @param price: the price of this item
   #
   def addItem(self, price) :
      # Method body

   ##  Gets the price of all items in the current sale.
   #  @return the total price
   #
   def getTotal(self):  ...
```

Class comments document the class and the behavior of each method

The method declarations make up the *public interface* of the class

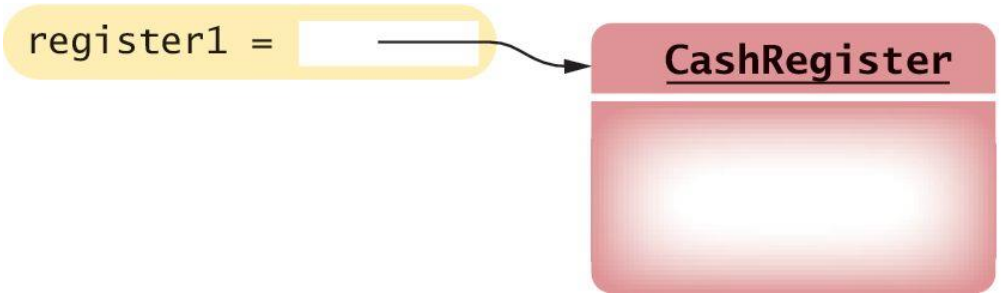
The data and method bodies make up the *private implementation* of the class

Using the Class

- After defining the class we can now construct an object:

```
register1 = CashRegister()
   # Constructs a CashRegister object
```

- This statement defines the `register1` variable and initializes it with a reference to a new `CashRegister` object



Using Methods

- Now that an object has been constructed, we are ready to invoke a method:

```
register1.addItem(1.95) # Invokes a method.
```

Using Methods

- Do you remember how to use dictionaries?

```
D={‘bill’:1, ‘hillary’:2, ‘George’:3}
count=D.len() # Invokes a method.
keys=D.keys() # Invokes a method.

if k in D: # Invokes a method (more complicated).
```

Accessor and Mutator Methods

- Many methods fall into two categories:

  1) Accessor Methods: '**get**' methods
     - Asks the object for information without changing it
     - Normally returns the current value of an attribute

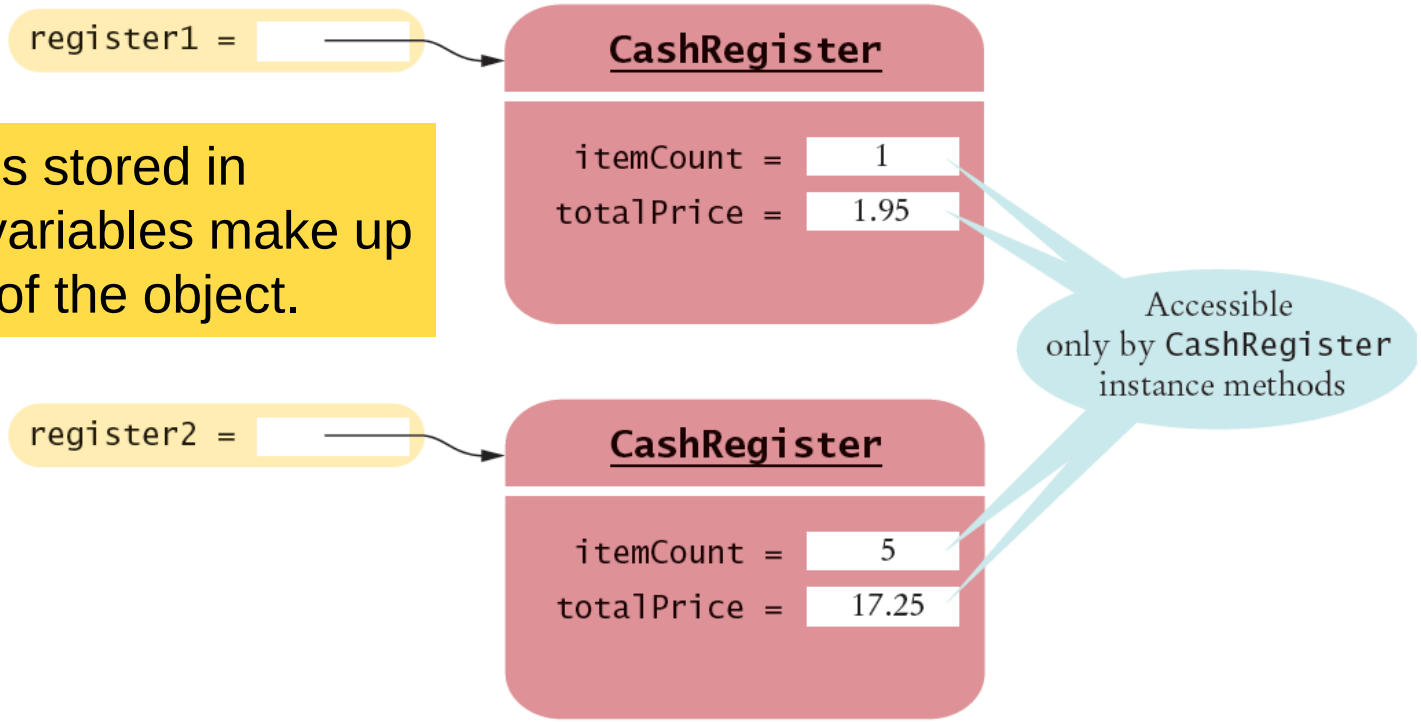
```
def getTotal(self):
def getCount(self):
```

  2) Mutator Methods: '**set**' methods
     - Changes values in the object
     - Usually take a parameter that will change an instance variable

```
def addItem(self, price):
def clear(self):
```

Instance Variables of Objects

- Each object of a class has a separate set of instance variables.



The values stored in instance variables make up the **state** of the object.

Designing the Data Representation

- An object stores data in instance variables
  - Variables declared inside the class
  - All methods inside the class have access to them
    - Can change or access them
  - What data will our `CashRegister` methods need?

Task	Method	Data Needed
Add the price of an item	`addItem(price)`	`total, count`
Get the total amount owed	`getTotal()`	`total`
Get the count of items purchased	`getCount()`	`count`
Clear cash register for a new sale	`clear()`	`total, count`

An object holds instance variables that are accessed by methods

Programming Tip

- All instance variables should be private and most methods should be public.
  - Although most object-oriented languages provide a mechanism to explicitly hide or protect private members from outside access, Python does not.
- It is common practice among Python programmers to use names that begin with a single underscore for private instance variables and methods.
  - The single underscore serves as a flag to the class user that those members are private.

Constructors

- A *constructor* is a method that initializes instance variables of an object
  - It is automatically called when an object is created

```
# Calling a method that matches the name of the class
# invokes the constructor
register = CashRegister()
```

- Python uses the special name `_ _init_ _` for the constructor because its purpose is to initialize an instance of the class:

```
def _ _init_ _(self) :
    self._itemCount = 0
    self._totalPrice = 0
```

Default and Named Arguments (1)

- Only one constructor can be defined per class.
- But you can define constructor with *default argument values* that simulate multiple definitions

```
class BankAccount :
    def __init__(self, initialBalance = 0.0) :
        self._balance = initialBalance
```

  - If no value is passed to the constructor when a `BankAccount` object is created the default value will be used.

```
joesAccount = BankAccount()   # Balance is set to 0
```

Default and Named Arguments (2)

- If a value is passed to the constructor that value will be used instead of the default one.

```
joesAccount = BankAccount(499.95)
   # Balance is set to 499.95
```

- Default arguments can be used in any method and not just constructors.

Syntax: Constructors

Syntax

```
class ClassName :
   def __init__(self, parameterName1, parameterName2, . . .) :
      constructor body
```

The special name `__init__` is used to define a constructor.

A constructor defines and initializes the instance variables.

```
class BankAccount :
   def __init__(self) :
      self._balance = 0.0
   . . .
```

```
class BankAccount :
   def __init__(self, initialBalance = 0.0) :
      self._balance = initialBalance
   . . .
```

There can be only one constructor per class. But a constructor can contain default arguments to provide alternate forms for creating objects.

Constructors: `Self`

- The first parameter variable of every constructor must be `self`.
- When the constructor is invoked to construct a new object, the `self` parameter variable is set to the object that is being initialized.

Refers to the object being initialized

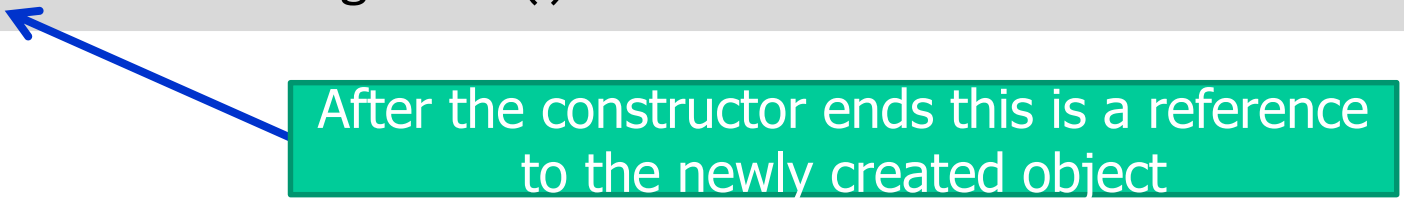
```
def _ _init_ _(self) :
    self._itemCount = 0
    self._totalPrice = 0
```

```
register = CashRegister()
```

After the constructor ends this is a reference to the newly created object

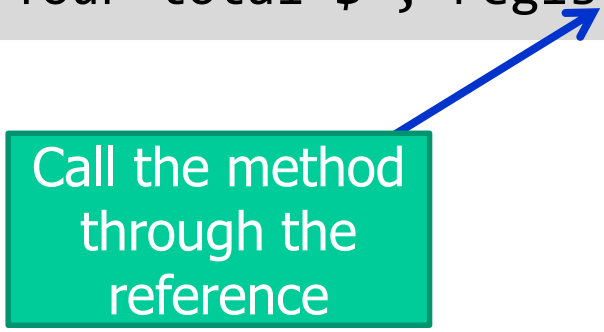
Object References

```
register = CashRegister()
```



- This reference then allows methods of the object to be invoked.

```
print("Your total $", register.getTotal())
```



Common Error

- After an object has been constructed, you should not directly call the constructor on that object again:

```
register1 = CashRegister()
register1._ _init_ _()   # Bad style
```

Common Error

- The constructor can set a new `CashRegister` object to the cleared state, but you should not call the constructor on an existing object. Instead, replace the object with a new one:

```
register1 = CashRegister()
register1 = CashRegister()   # OK
```

In general, you should never call a Python method that starts with a double underscore. The are intended for specific internal purposes (in this case, to initialize a newly created object).

Implementing Methods

- Implementing a method is very similar to implementing a function except that you access the instance variables of the object in the method body.

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

Task	Method
Add the price of an item	`addItem(price)`
Get the total amount owed	`getTotal()`
Get the count of items purchased	`getCount()`
Clear the cash register for a new sale	`clear()`

Syntax: Instance Methods

- Use instance variables inside methods of the class
  - Similar to the constructor, all other instance methods must include the self parameter as the first parameter.
  - You must specify the `self` implicit parameter when using instance variables inside the class.

Syntax

```
class ClassName :
   . . .
   def methodName(self, parameterName1, parameterName2, . . .) :
      method body
   . . .
```

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

Every method must include the special `self` parameter variable. It is automatically assigned a value when the method is called.

Instance variables are referenced using the `self` parameter.

Local variable

Invoking Instance Methods

- As with the constructor, every method must include the special `self` parameter variable, and it must be listed first.
- When a method is called, a reference to the object on which the method was invoked (`register1`) is automatically passed to the `self` parameter variable:

```
register1.addItem(2.95)
```

```
def addItem(self, price):
```

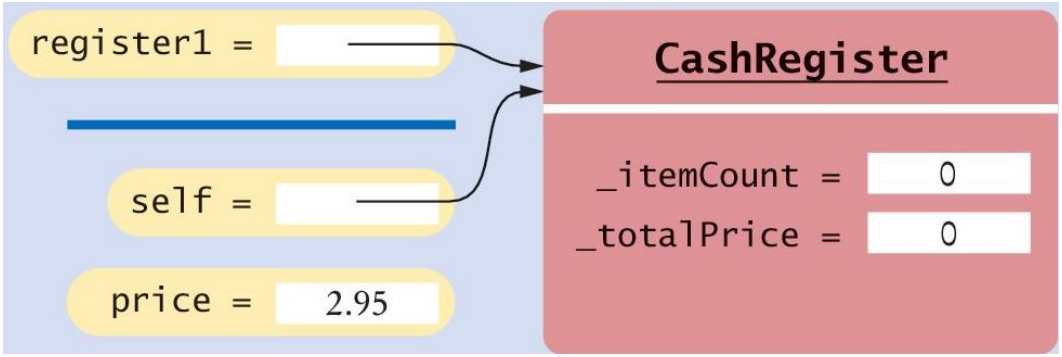
Accessing Instance Variables (1)

- To access an instance variable, such as `_itemCount` or `_totalPrice`, in a method, you must access the variable name through the `self` reference.
  - This indicates that you want to access the instance variables of the object on which the method is invoked, and not those of some other `CashRegister` object.
- The first statement in the `addItem()` method is

```
self._itemCount = self._itemCount + 1
```

Accessing Instance Variables (2)

- Which `_itemCount` is incremented?
  - In this call, it is the `_itemCount` of the `register1` object.



Calling One Method Within Another

- When one method needs to call another method on the same object, you invoke the method on the self parameter.

```
def addItems(self, quantity, price) :
    for i in range(quantity) :
        self.addItem(price)
```

Example: `CashRegister.py` (1)

```
 7  class CashRegister :
 8     ## Constructs a cash register with cleared item count and total.
 9     #
10     def __init__(self) :
11        self._itemCount = 0
12        self._totalPrice = 0.0
13
14     ## Adds an item to this cash register.
15     #  @param price the price of this item
16     #
17     def addItem(self, price) :
18        self._itemCount = self._itemCount + 1
19        self._totalPrice = self._totalPrice + price
20
21     ## Gets the price of all items in the current sale.
22     #  @return the total price
23     #
24     def getTotal(self) :
25        return self._totalPrice
```

```
27     ## Gets the number of items in the current sale.
28     #  @return the item count
29     #
30     def getCount(self) :
31        return self._itemCount
32
33     ## Clears the item count and the total.
34     #
35     def clear(self) :
36        self._itemCount = 0
37        self._totalPrice = 0.0
```

Programming Tip

- Instance variables should only be defined in the constructor.
- All variables, including instance variables, are created at run time.
  - There is nothing to prevent you from creating instance variables in any method of a class.
- The constructor is invoked before any method can be called, so any instance variables that were created in the constructor are sure to be available in all methods.

Class Variables

- They are a value properly belongs to a class, not to any object of the class.
- Class variables are often called “static variables”.
- Class variables are declared at the same level as methods. (In contrast, instance variables are created in the constructor.)

Class Variables: Example (1)

- We want to assign bank account numbers sequentially: the first account is assigned number 1001, the next with number 1002, and so on.
- To solve this problem, we need to have a single value of `_lastAssignedNumber` that is a property of the *class*, not any object of the class.

```
class BankAccount :
    _lastAssignedNumber = 1000 # A class variable
    def _ _init_ _(self) :
        self._balance = 0
        BankAccount._lastAssignedNumber =
           BankAccount._lastAssignedNumber + 1
        self._accountNumber =
           BankAccount._lastAssignedNumber
```

Class Variables: Example (2)

- Every `BankAccount` object has its own `_balance` and `_account-Number` instance variables, but there is only a single copy of the `_lastAssignedNumber` variable.
- That variable is stored in a separate location, outside any `BankAccount` object.
- Like instance variables, class variables should always be private to ensure that methods of other classes do not change their values. However, class *constants* can be public.

Class Variables: Example (3)

- For example, the `BankAccount` class can define a public constant value, such as

```
class BankAccount :
    OVERDRAFT_FEE = 29.95
    . . .
```

- Methods from any class can refer to such a constant as `BankAccount.OVERDRAFT_FEE`.

Steps to Implementing a Class

1) Get an informal list of responsibilities for your objects

Deposit funds.
Withdraw funds.
Add interest.

– There is a hidden responsibility as well. We need to be able to find out how much money is in the account.

Get balance.

Steps to Implementing a Class

- 2) Specify the public interface
  - Constructor

```
def __init__(self, initialBalance = 0.0) :
```

  - Mutators

```
def deposit(self, amount) :
def withdraw(self, amount) :
def addInterest(self, rate) :
```

  - Accessors

```
def getBalance(self) :
```

- 3) Document the public interface

```
## Constructs a bank account with a given balance.
#  @param initialBalance the initial account balance (default = 0.0)
#
```

Steps to Implementing a Class

4) Determine the instance variables

```
self._balance = initialBalance
```

5) Implement constructors and methods

```
def getBalance(self) :
   return self._balance
```

6) Test your class

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
```

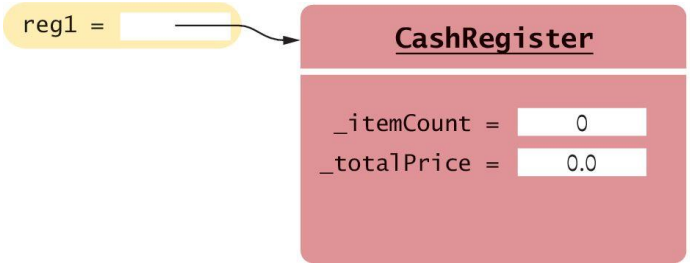
Object References

- 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.

Reference

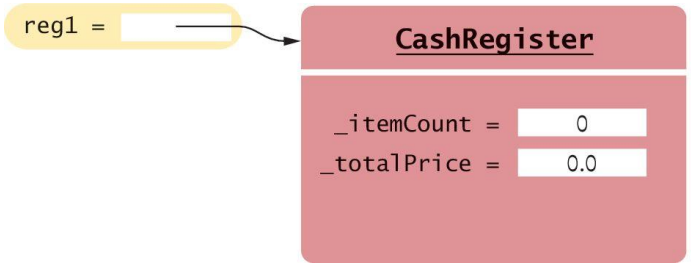


Object

Shared References

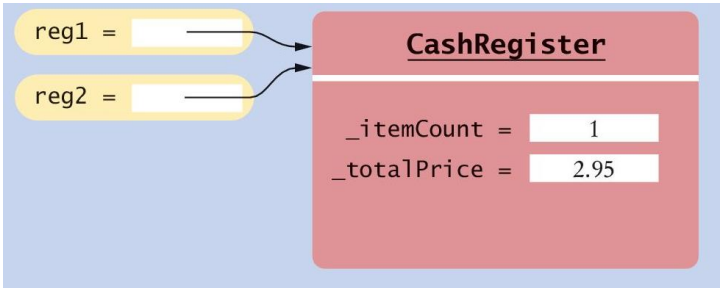
- Multiple object variables may contain references to the same object ('aliases')
  - Single Reference

```
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 `is not` 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

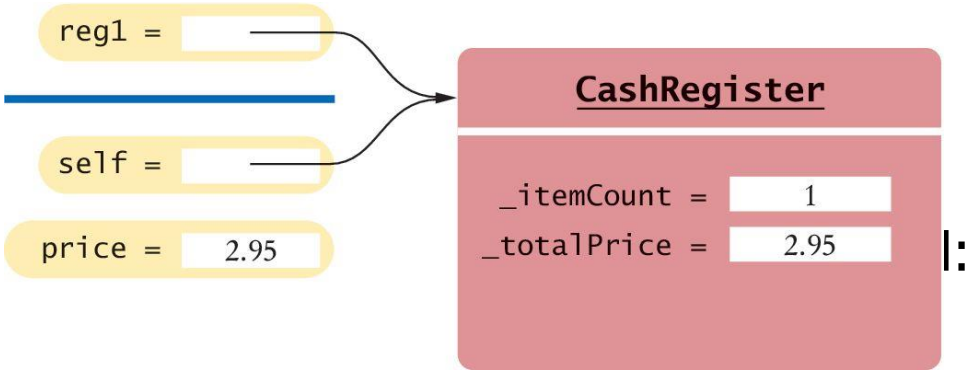
- A reference may point to 'no' object
  - You cannot invoke methods of an object via a None reference – causes a run-time error

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

```
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 

:

```
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
```

Summary: Classes and Objects

- A class describes a set of objects with the same behavior.
  - Every class has a public interface: a collection of methods through which the objects of the class can be manipulated.
  - Encapsulation is the act of providing a public interface and hiding the implementation details.
  - Encapsulation enables changes in the implementation without affecting users of a class

Summary: Variables and Methods

- An object's instance variables store the data required for executing its methods.
- Each object of a class has its own set of instance variables.
- An instance method can access the instance variables of the object on which it acts.
- A private instance variable should only be accessed by the methods of its own class.
- Class variables have a single copy of the variable shared among all of the instances of the class.

Summary: Method Headers, Data

- Method Headers
  - You can use method headers and method comments to specify the public interface of a class.
  - A mutator method changes the object on which it operates.
  - An accessor method does not change the object on which it operates.
- Data Representation
  - For each accessor method, an object must either store or compute the result.
  - Commonly, there is more than one way of representing the data of an object, and you must make a choice.
  - Be sure that your data representation supports method calls in any order.

Summary: Constructors

- A constructor initializes the object's instance variables
- A constructor is invoked when an object is created.
- The constructor is defined using the special method name: `_ _init_ _()`.
- Default arguments can be used with a constructor to provide different ways of creating an object.

Summary: Method Implementation

- The object on which a method is applied is automatically passed to the `self` parameter variable of the method.
- In a method, you access instance variables through the `self` parameter variable.

Summary: Patterns for Classes

- An instance variable for the total is updated in methods that increase or decrease the total amount.
- A counter that counts events is incremented in methods that correspond to the events.
- An object can collect other objects in a list.
- An object property can be accessed with a getter method and changed with a setter method.
- If your object can have one of several states that affect the behavior, supply an instance variable for the current state.
- To model a moving object, you need to store and update its position.

Summary: Object References

- An object reference specifies the location of an object.
- Multiple object variables can contain references to the same object.
- Use the `is` and `is not` operators to test whether two variables are aliases.
- The `None` reference refers to no object.

Final example

Rational Numbers

- **Rational number** consists of two integer parts, a numerator and a denominator
  - Examples: 1/2, 2/3, etc.
- Python has no built-in type for rational numbers
  - We will build a new class named **`Rational`**

```
>>> oneHalf = Rational(1, 2)
>>> oneSixth = Rational(1, 6)
>>> print oneHalf
1/2
>>> print oneHalf + oneSixth
2/3
>>> oneHalf == oneSixth
False
>>> oneHalf > oneSixth
True
```

Operators need to be **overloaded**

Rational Number Arithmetic and Operator Overloading

OPERATOR	METHOD NAME
+	`__add__`
-	`__sub__`
*	`__mul__`
/	`__div__`
%	`__mod__`

- Object on which the method is called corresponds to the left operand
  - For example, the code **`x + y`** is actually shorthand for the code **`x.__add__(y)`**

Rational Number Arithmetic and Operator Overloading (continued)

- To overload an arithmetic operator, you define a new method using the appropriate method name
- Code for each method applies a rule of rational number arithmetic

TYPE OF OPERATION	RULE
Addition	$n_1/d_1 + n_2/d_2 = (n_1d_2 + n_2d_1) / d_1d_2$
Subtraction	$n_1/d_1 - n_2/d_2 = (n_1d_2 - n_2d_1) / d_1d_2$
Mutiplication	$n_1/d_1 * n_2/d_2 = n_1n_2 / d_1d_2$
Division	$n_1/d_1 / n_2/d_2 = n_1d_2 / d_1n_2$

Rational Number Arithmetic and Operator Overloading (continued)

```
def __add__(self, other):
    """Returns the sum of the numbers."""
    #Self is the left operand and other is the right operand
    newNumer = self._numer * other._denom + \
               other._numer * self._denom
    newDenom = self._denom * other._denom
    return Rational(newNumer, newDenom)
```

- **Operator overloading** is another example of an abstraction mechanism
  - We can use operators with single, standard meanings even though the underlying operations vary from data type to data type

Equality and the `__eq__` Method

- Not all objects are comparable using < or >, but any two objects can be compared for == or !=

  `twoThirds < "hi there"` should generate an error

  `twoThirds != "hi there"` should return **`True`**

```
def __eq__(self, other):
    """Tests self and other for equality."""
    if self is other:                   # Object identity?
        return True
    elif type(self) != type(other):     # Types match?
        return False
    else:
        return self._numer == other._numer and \
               self._denom == other._denom
```

- Include **`__eq__`** in any class where a comparison for equality uses a criterion other than object identity

Comparisons and the `__cmp__` Method

- **`__cmp__`** is called whenever you use the comparison operators: ==, !=, <, >, <=, and >=
- Returns 0 if operands are equal, -1 if left operand is < right one, 1 if left operand > right one

```
>>> cmp(1, 1)     # Equal
0
>>> cmp(1, 2)     # Less than
-1
>>> cmp(2, 1)     # Greater than
1
```

```
def __cmp__(self, other):
    """Compares two rational numbers."""
    extremes = self._numer * other._denom
    means = other._numer * self._denom
    return cmp(extremes, means)
```

Thank you very much!