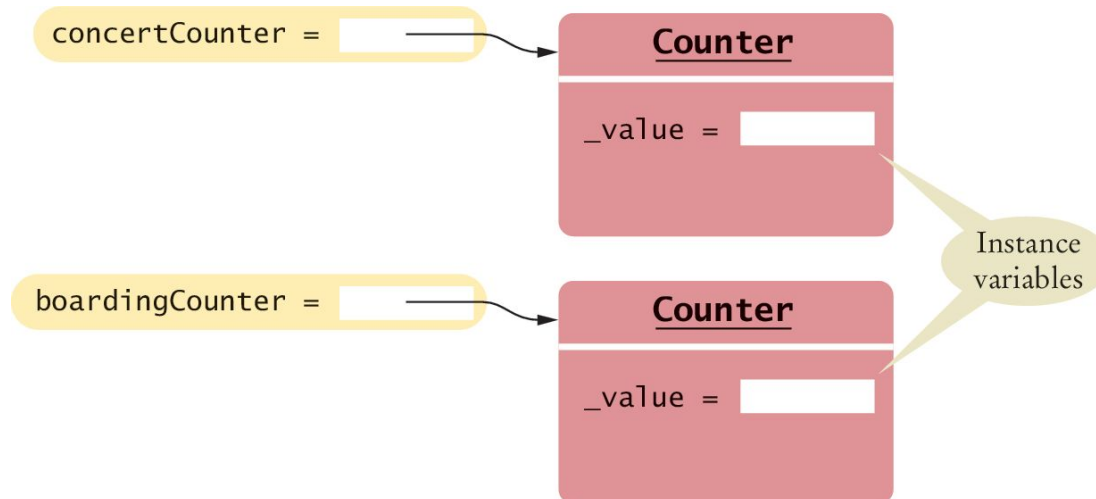


# Instance Variables

---

- 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`

# Class Methods and Attributes

---

- Note how the `click()` method increments the instance variable `_value`
- *Which* instance variable? The one belonging to the object on which the method is invoked
  - In the example below the call to `click()` advances the `_value` variable of the `concertCounter` object
  - No argument was provided when the `click()` method was called *even though the definition includes the `self` parameter* variable
- The `self` parameter variable refers to the object on which the method was invoked `concertCounter` in this example

```
concertCounter.click()
```

# 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

```
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)
```

```
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
```

5. What would happen if you didn't call `reset` immediately after constructing the `Tally` object?

► Show Answer

---

6. Consider a change to the implementation of the counter. Instead of using an integer counter, we use a string of `|` characters to keep track of the clicks, just as a human might do.

```
class Counter :
    def reset(self) :
        self._strokes = ""

    def click(self) :
        self._strokes = self._strokes + "|"
    . . .
```

How do you implement the `getValue` method with this data representation?

► Show Answer

---

7. Suppose another programmer has used the original `Counter` class. What changes does that programmer have to make in order to use the modified class from [Self Check 6](#)?

► Show Answer

---

8. Suppose you use a class `Clock` with instance variables `_hours` and `_minutes`. How can you access these variables in your program?

► Show Answer

# 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 Public Interface

---

- 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  
    #
```

Class comments document the class and the behavior of each method

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

```
        # Method body
```

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

```
    ## Gets the price of all items in the current sale.
```

```
    # @return the total price
```

```
    #
```

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

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

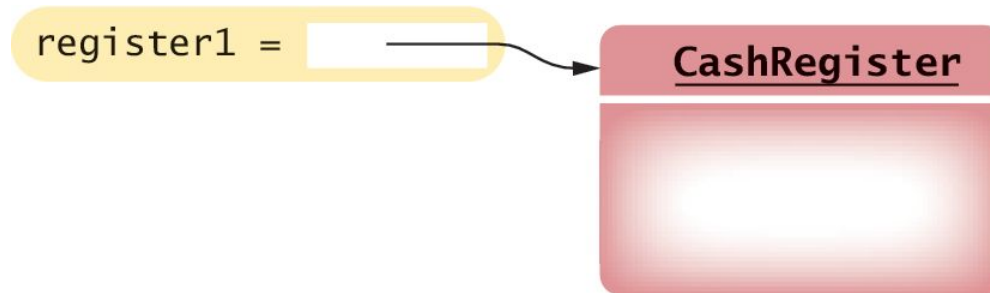
# Using the Class

---

- 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.
```

# 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):
```

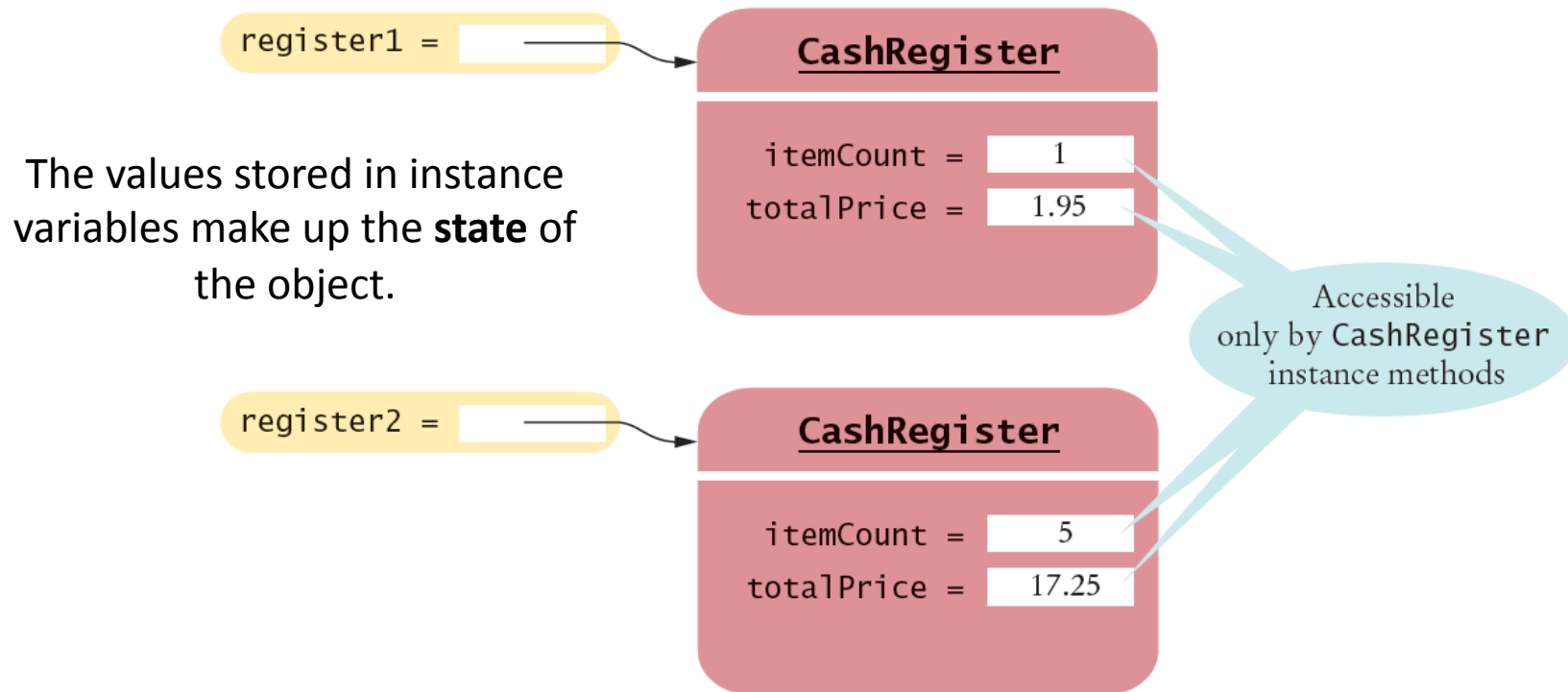
2) Mutator Methods: 'set' methods

- 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



# 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 9.1

---

- 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



# Programming Tip 9.1

---

- You should always use encapsulation, in which all instance variables are private and are only manipulated with methods
- Typically, methods are public
  - However, sometimes you have a method that is used only as a helper method by other methods
  - In that case, you should identify the helper method as private by using a name that begins with a single underscore

# 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

---

- Only one constructor can be defined per class
- But you can define a 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

---

- 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`, *parameterName*<sub>1</sub>, *parameterName*<sub>2</sub>, . . . ) :  
                  *constructor body*

The special name `__init__`  
is used to define a constructor. — `class` `BankAccount` :  
                                  `def` `__init__`(`self`) :  
  `self`.\_balance = 0.0  
  . . .

A constructor defines  
and initializes the  
instance variables. — `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

```
def __init__(self):  
    self._itemCount = 0  
    self._totalPrice = 0
```

Refers to the  
object being  
initialized



```
register = CashRegister()
```


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



# Object References

---

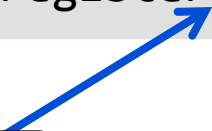
```
register = CashRegister()
```



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

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

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



Call the method through the reference

# Common Error 9.1 (1)

---

- 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 9.1 (2)

---

- 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. They 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