ICS 104 - Introduction to Programming in Python and C

Objects and Classes

Reading Assignment

- Chapter 9: Sections 1 - 7.

Learning Outcomes

- 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.

Object-Oriented Programming

- You have learned how to structure your programs by decomposing tasks into functions.
    - 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

Classes

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

Media Bakery.

Objects and Programs

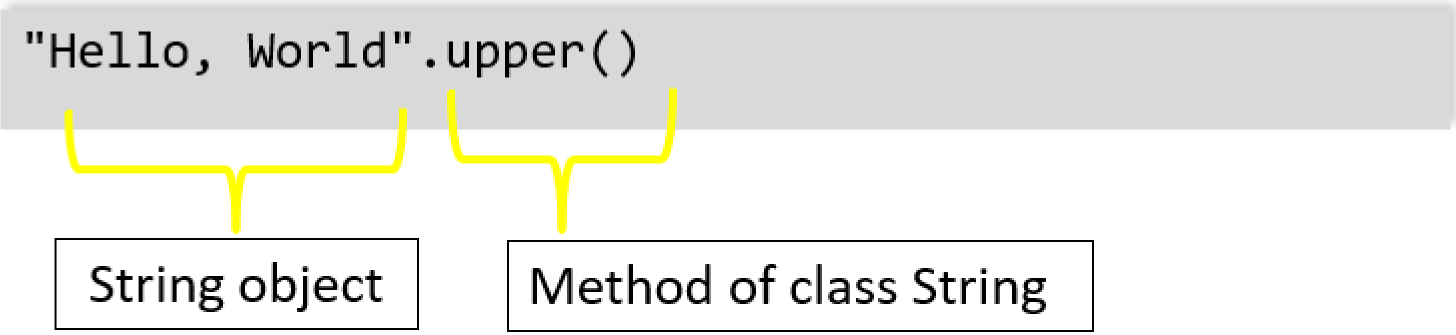
- You have learned how to structure your programs by decomposing tasks into functions
    - Experience shows that it does not go far enough
    - It is difficult to understand and update a program that consists of a large collection of functions.

- 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.

- 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 insert or remove methods to operate on list objects.

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:



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

String object

Method of class String

- 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

In [ ]:

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

- However, list has a pop() method, and the following call is legal

In [ ]:

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

In [ ]:

```
myList = ["Hello", "World"]
myList.pop()
print(myList)
```

Student Activity

- Is the method call "Hello World".print() legal? Why or Why not?

In [ ]:

```
"Hello World!".print()
```

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 need not 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

- 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

- Consider, **Tally Counter:** A class that models a mechanical device that is used to count people
    - For example, to find out how many people board a bus.
- Whenever the operator pushes a button, the counter value advances by one. The counter has a display to show the current value.

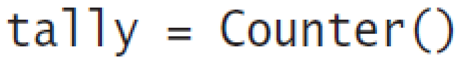
© Jasmin Awad/iStockphoto.

- What **operations** (aka methods) can you identify are needed in this class?

-
    - Increment the tally
    - Get the current total

Using the Counter Class

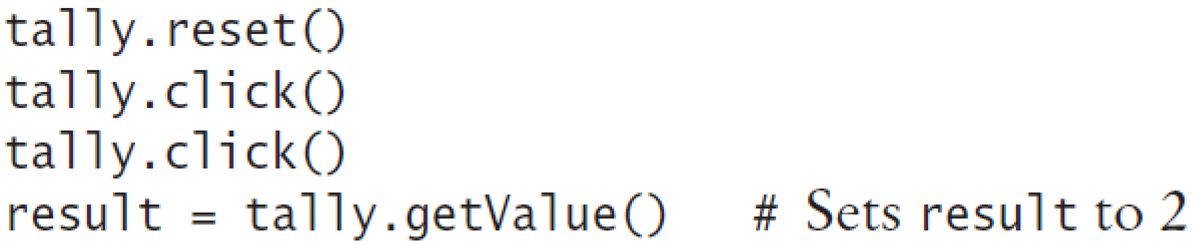
- First, note that we will show how to define the class later. Now, we are concerned with using the class.
- First, we construct an object of the class.
- In Python, you don’t explicitly declare instance variables
    - Did we declare an integer variable before using it?
- Instead, when one first assigns a value to an instance variable, that instance variable is created



```
tally = Counter()
```

- More information about constructing objects will be given later.

- Next, we invoke methods on our object



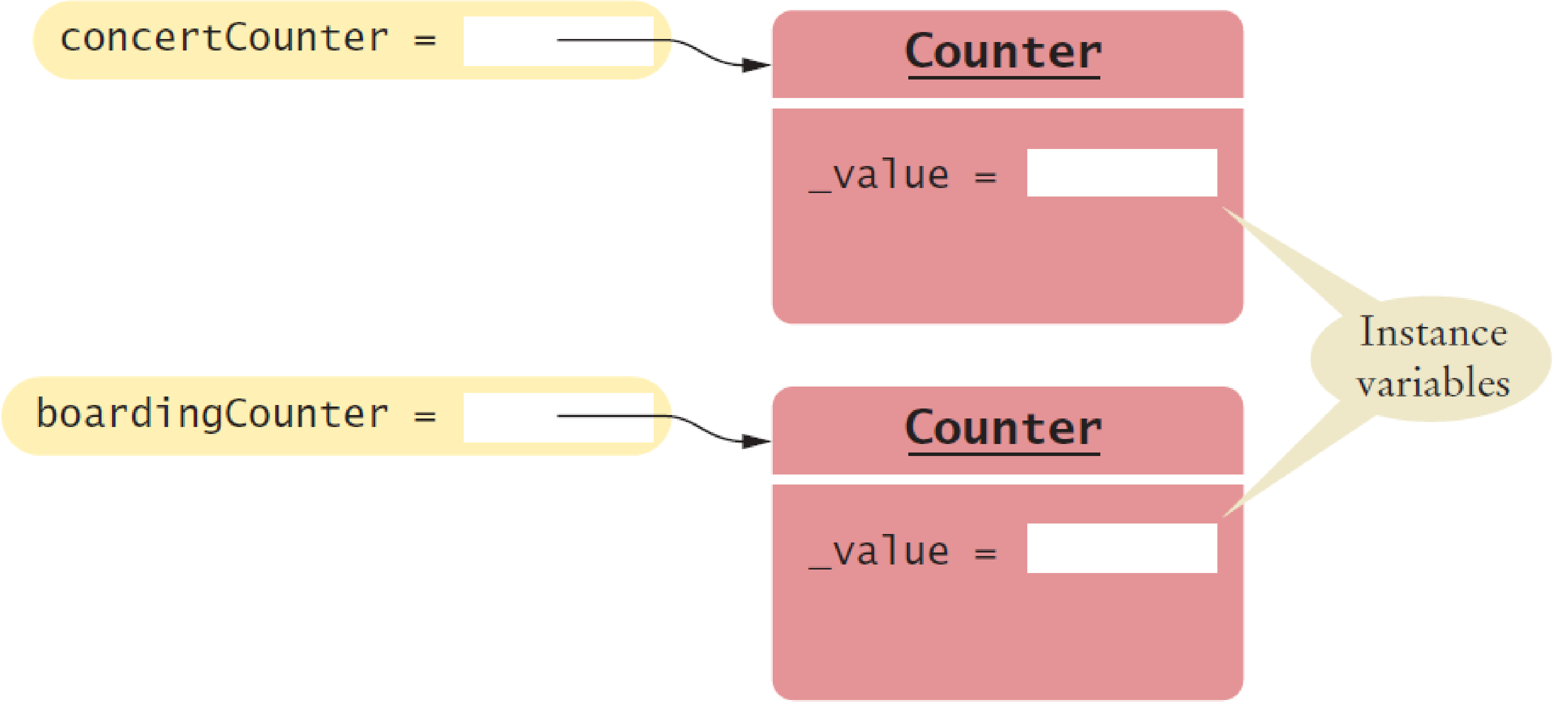
```
tally.reset()
tally.click()
tally.click()
result = tally.getValue()   # Sets result to 2
```

- We can invoke the methods again, and the result will be different:

```
tally.click()
result = tally.getValue()      # Sets result to 3
```

Instance Variables

- An instance of a class is an object of the class
- An object stores its data in **instance variables**
- 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 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

In [ ]:

```python
#  This module defines the Counter class.
#

## Models a tally counter whose value can be
#  incremented, viewed, or reset.
#
class Counter :
   ## Gets the current value of this counter.
   #  @return the current value
   #
   def getValue(self) :
      return self._value

   ## Advances the value of this counter by 1.
   #
   def click(self) :
      self._value = self._value + 1

   ## Resets the value of this counter to 0.
   #
   def reset(self) :
      self._value = 0
```

In [ ]:

```
##
#  This program demonstrates the Counter class.
#

# Import the Counter class from the counter module.
#import counter
#from counter import Counter
# The above two lines are commented since we did not
# save the Counter class in a file called counter.py.

tally = Counter()
tally.reset()
tally.click()
tally.click()

result = tally.getValue()
print("Value:", result)

tally.click()
result = tally.getValue()
print("Value:", result)
```

Student Activity

- What would happen if you did not call reset immediately after constructing the tally object?

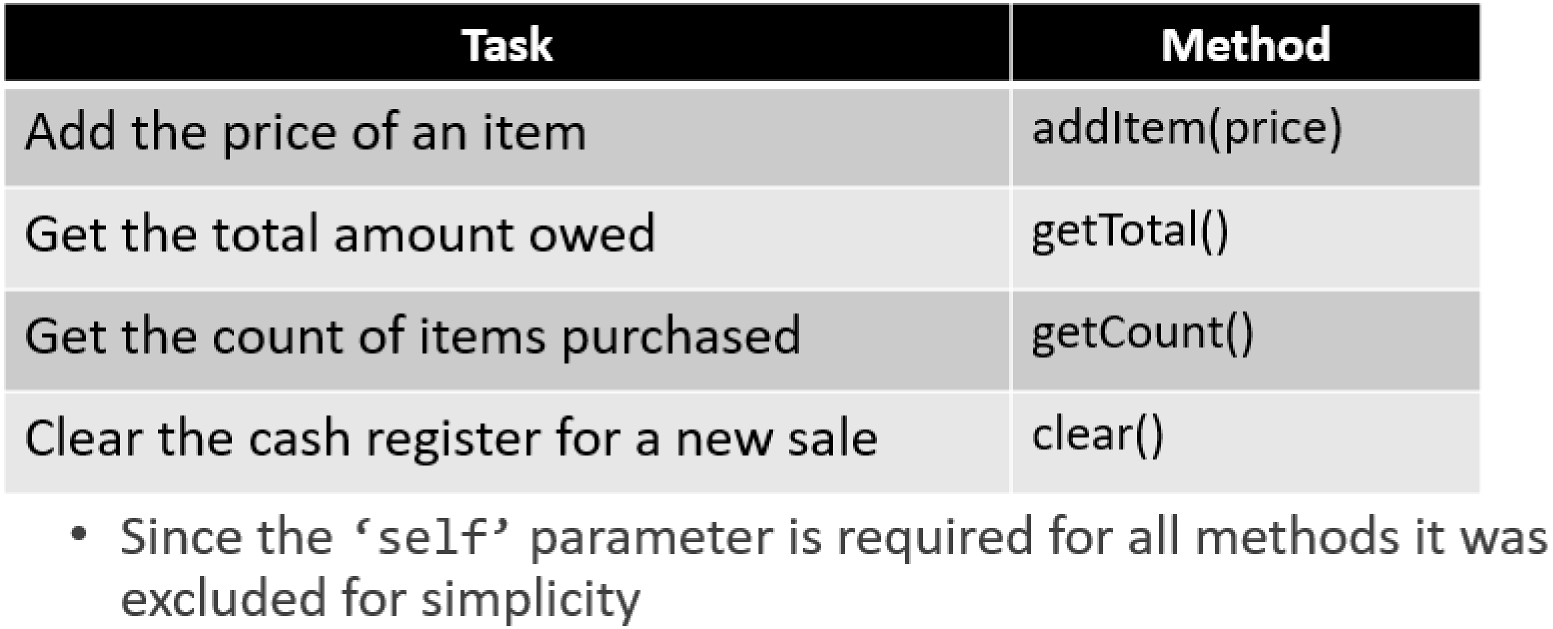
Public Interface of a Class

- 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 (A Cash Register Class)

© James Richey/iStockphoto.

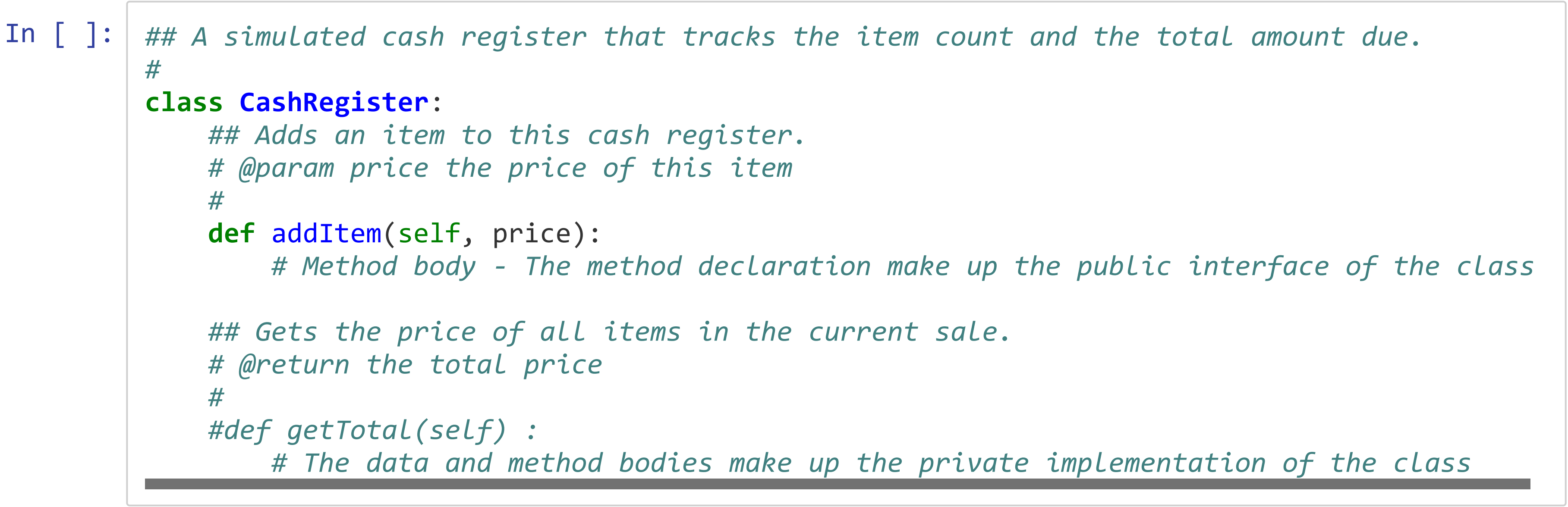
- We want to use objects that simulate cash registers.
    - A cashier who rings up a sale presses a key to start the sale, then rings up each item. A display shows the amount owed as well as the total number of items purchased.



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



In [ ]:

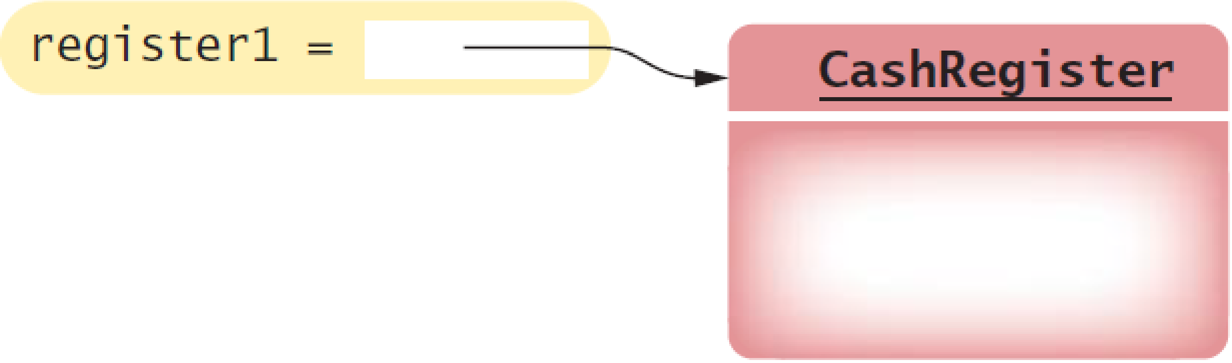
```
## A simulated cash register that tracks the item count and the total amount due.
#
class CashRegister:
    ## Adds an item to this cash register.
    # @param price the price of this item
    #
    def addItem(self, price):
        # Method body - The method declaration make up the public interface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

```
In [ ]: register1 = CashRegister()
            # Constructs a CashRegister object
```

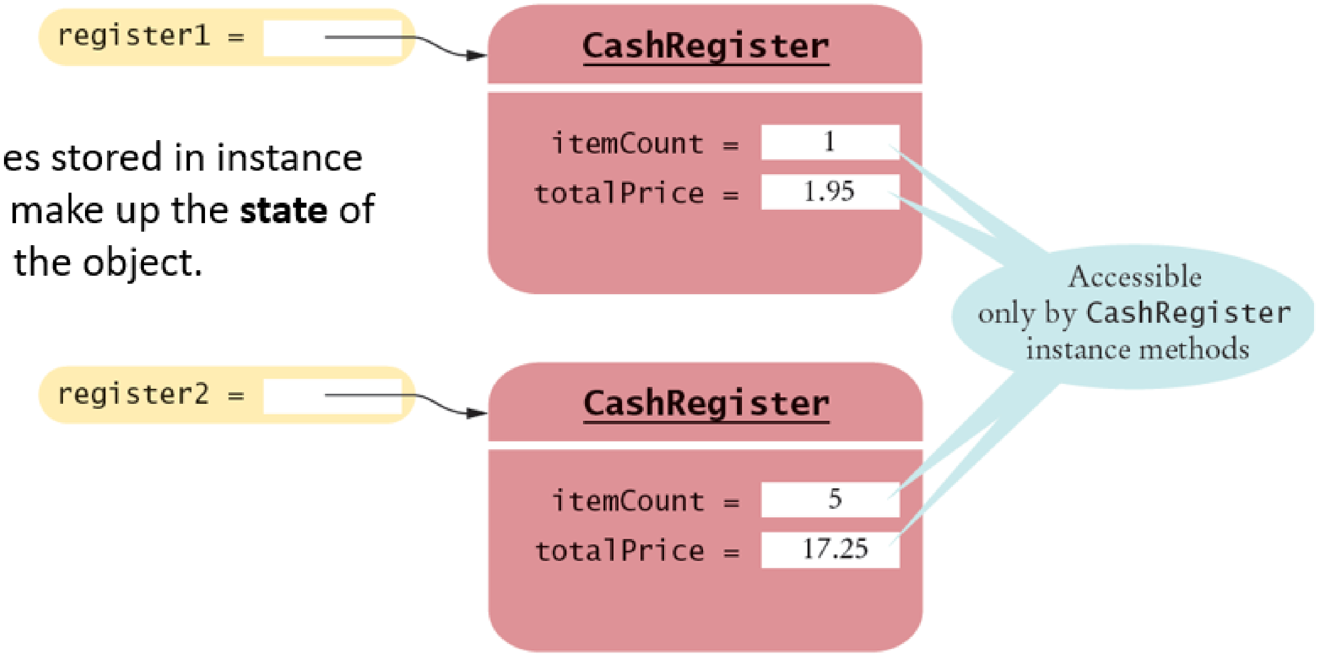


Using Methods

- Now that an object has been constructed, we are ready to invoke a method:
- `register1.addItem(1.95) # Invokes a method`

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.

Constructors

- A constructor is a method that initializes instance variable of an object.
  - It is automatically called when an object is created.
  - Python uses the special name `__init__` to define constructor.

```
# Calling a method that matches the name of the class

# Invokes the constructor

def __init__(self):

    self._itemCount = 0

    self._totalPrice = 0.0
#
```

Default and Named Arguments

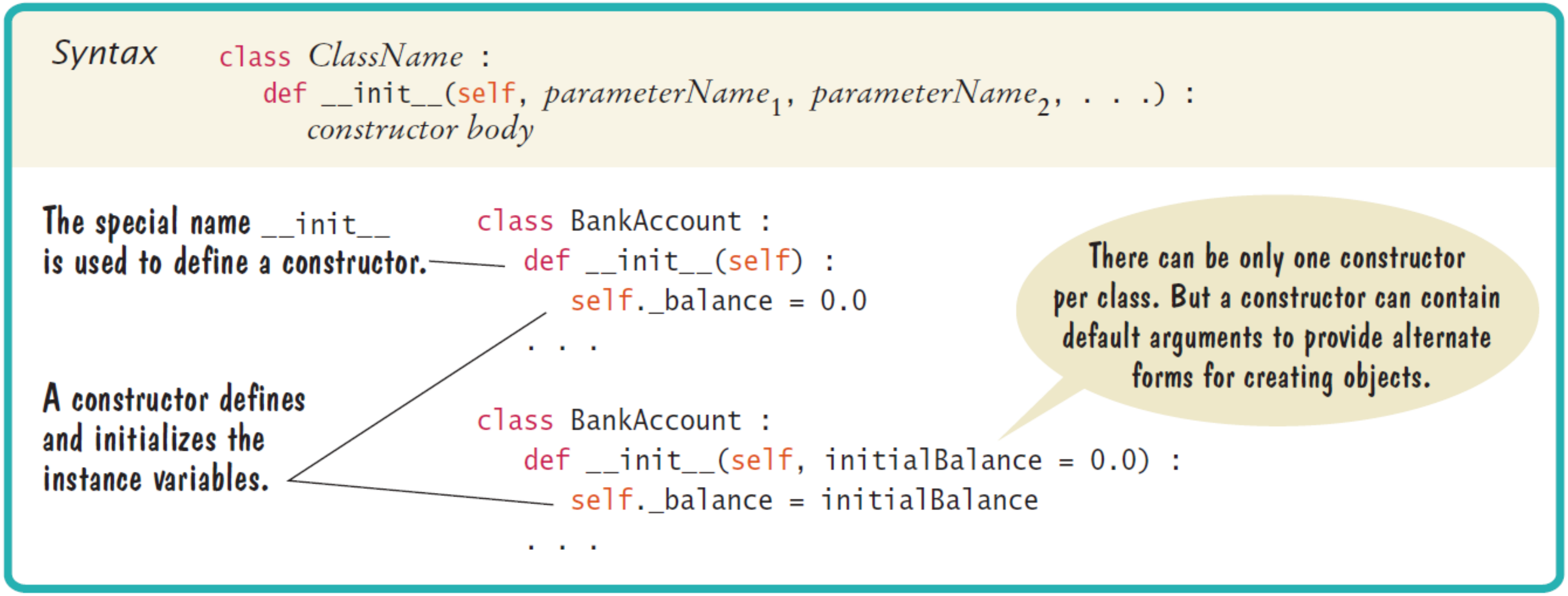
- Only one constructor can be defined per class:
- But you can define constructor with default argument values that simulate multiple definitions

```
class BankAccount:
        def __int__(self, initialBlance = 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
```

Syntax: Constructors



Syntax

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

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

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

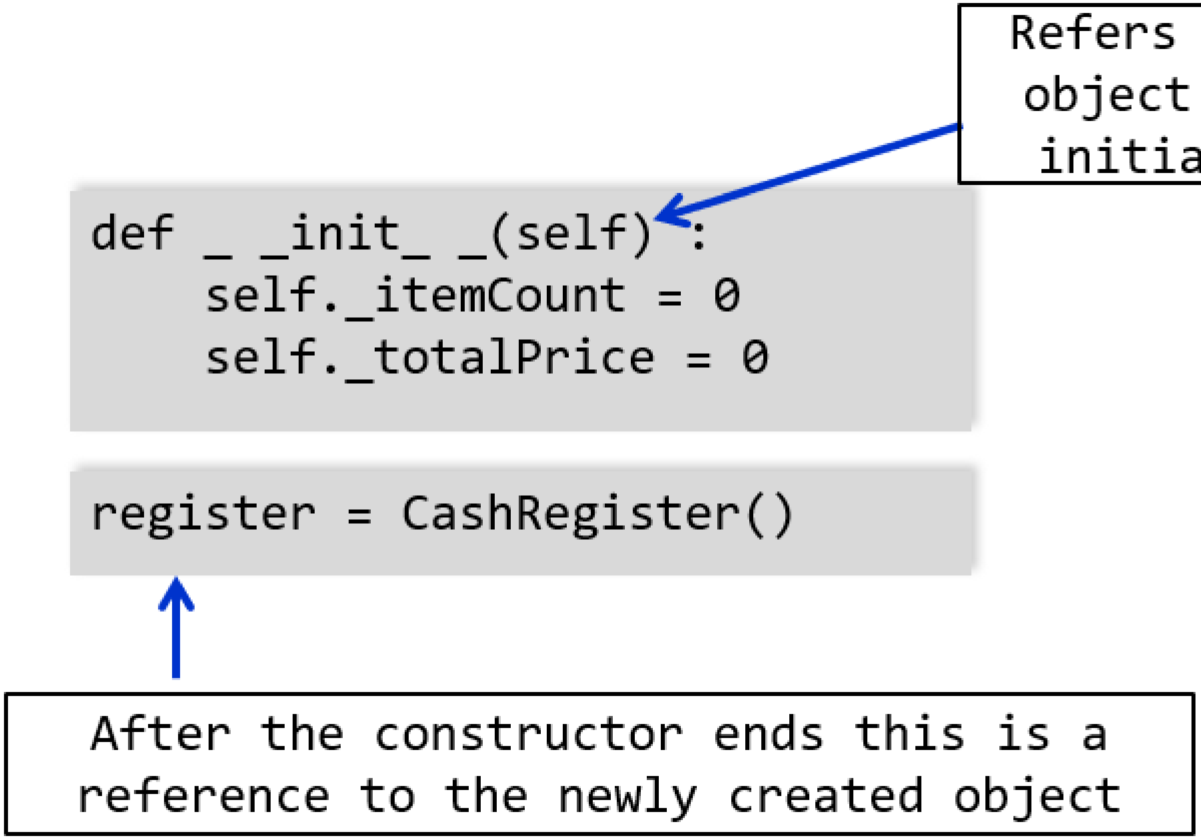
A constructor defines and initializes the instance variables.

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

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

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



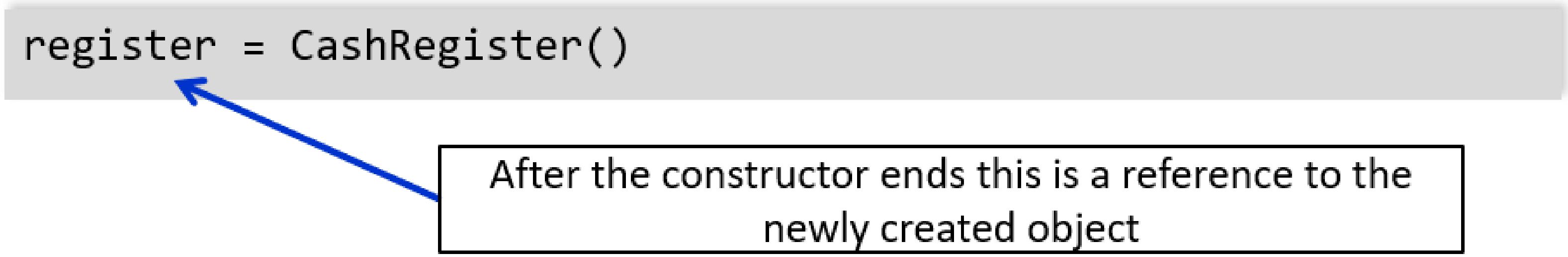
Refers to the object being initialized

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

```
register = CashRegister()
```

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

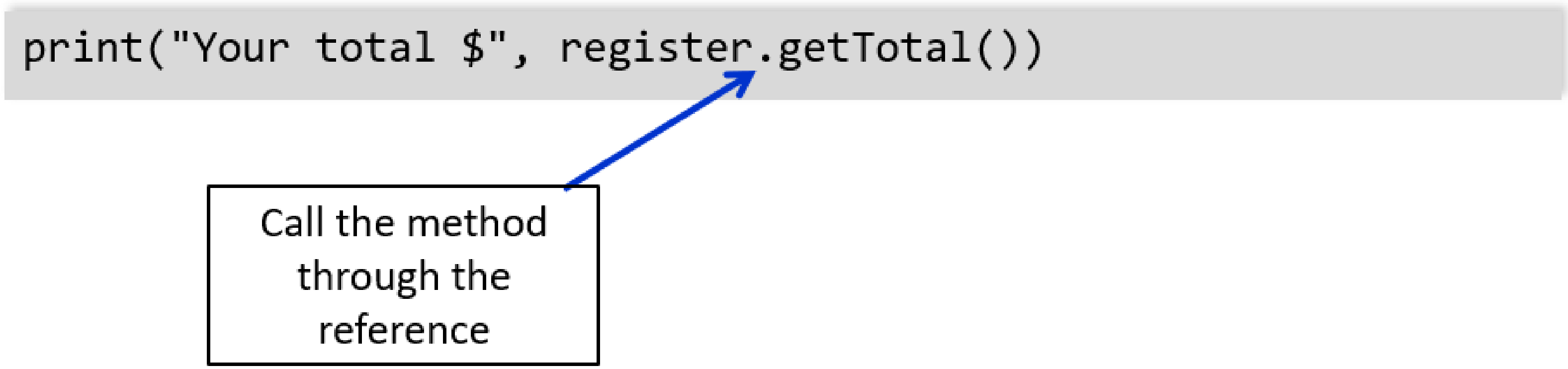
Object References



```
register = CashRegister()
```

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

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

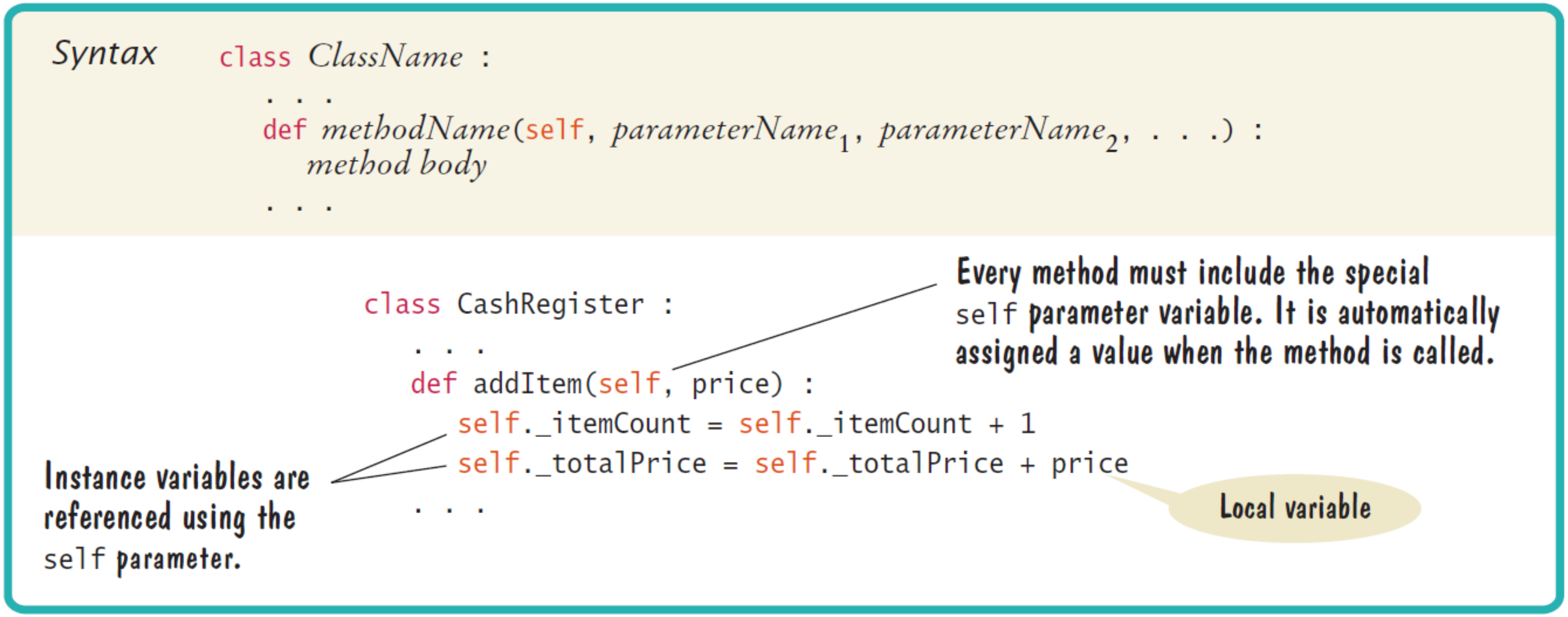


```
print("Your total $", register.getTotal())
```

Call the method through the reference

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 fist parameter
  - You must specify the `self` implicit parameter when using instance variable 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
   . . .
```

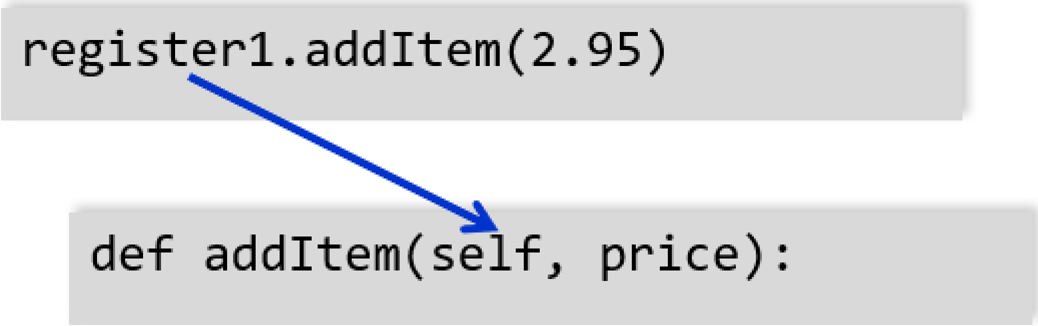
Every method must include the special self parameter variable. It is automatically assigned a value when the method is called.

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):
```

Complete Example

In [ ]:

```
#  This module defines the CashRegister class.
## A simulated cash register that tracks the item
#  count and the total amount due.

class CashRegister :
   ## Constructs a cash register with cleared item
   #  count and total.
   #
   def __init__(self) :
      self._itemCount = 0
      self._totalPrice = 0.0

   ## Adds an item to this cash register.
   #  @param price the price of this item
   #
   def addItem(self, price) :
      self._itemCount = self._itemCount + 1
      self._totalPrice = self._totalPrice + price

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

   ## Gets the number of items in the current sale.
   #  @return the item count
   #
   def getCount(self) :
      return self._itemCount

   ## Clears the item count and the total.
   #
   def clear(self) :
      self._itemCount = 0
      self._totalPrice = 0.0
```

Summary

- 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

- 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

- 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