Object Oriented Programming
DRE7053

NORA Summer School 2024

Rogelio A Mancisidor
Assistant Professor
Department of Data Science and Analytics
BI Norwegian Business School

June 12, 2024

Contents

- Implementing Methods
- Class Variables: Public & Constant
- Testing a Class
- Object References

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

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

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
   . . .
```

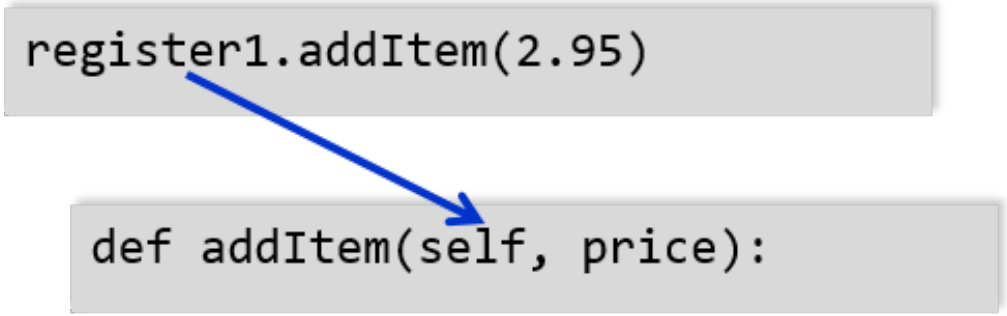
Every method must include the special `self` parameter variable. It is automatically assigned a value when the method is called.

Instance variables are referenced using the `self` parameter.

Local variable

Invoking Instance Methods

- 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:

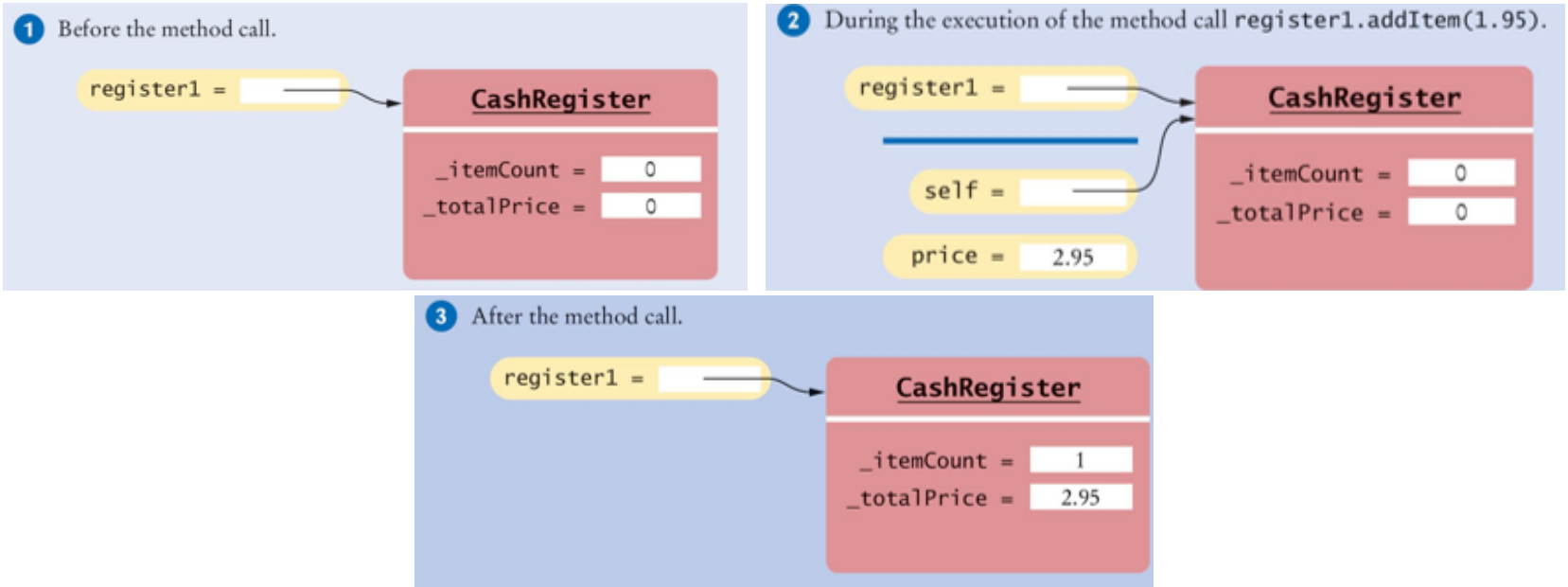


So basically something like this happens `addItem(register1, 2.95)`

Tracing The Method Call

```
register1 = CashRegister()      #1 New object
register1.addItem(2.95)         #2 Calling method
                                #3 After method
```

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



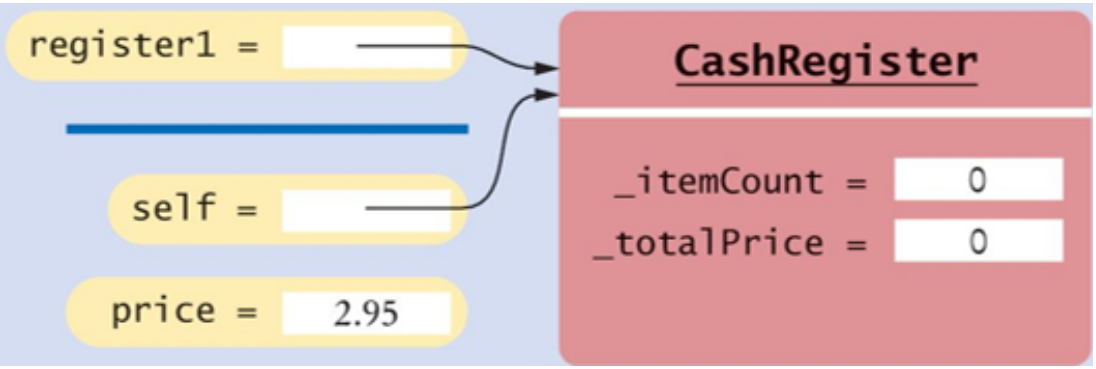
Accessing Instance Variables

- 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

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

Accessing Instance Variables

- 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`

```
## Cash register class with cleared item count and total

class CashRegister :
   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
```

Programming Tip 9.2

- 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

```
1 class BankAccount :
2     _lastAssignedNumber = 1000 # A class variable
3     def _ _init_ _(self) :
4         self._balance = 0
5         BankAccount._lastAssignedNumber =
6             BankAccount._lastAssignedNumber + 1
7         self._accountNumber =
8             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

```
1 class BankAccount :
2     OVERDRAFT_FEE = 29.95
3     . . .
```

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

Testing a Class

- In the long run, your class may become a part of a larger program that interacts with users, stores data in files, and so on
- You should always test your class in isolation integrating a class into a program
- Testing in isolation, outside a complete program, is called **unit testing**

Choices for Testing: The Python Shell

- Some interactive development environments provide access to the Python shell in which individual statements can be executed
- You can test a class simply by constructing an object, calling methods, and verifying that you get the expected return values

```
>>> from cashregister import CashRegister
>>> reg = CashRegister()
>>> reg.addItem(1.95)
>>> reg.addItem(0.95)
>>> reg.addItem(2.50)
>>> print(reg.getCount())
3
>>> print(reg.getTotal())
5.4
>>>
```

Choices for Testing: Test Drivers

- Interactive testing is quick and convenient but it has a drawback
  - When you find and fix a mistake, you need to type in the tests again
- As your classes get more complex, you should write tester programs
  - A tester program is a driver module that imports the class and contains statements to run methods of your class

Steps Performed by a Tester Program

- Construct one or more objects of the class that is being tested
- Invoke one or more methods
- Print out one or more results
- Print the expected results
- Compare the computed results with the expected

Example Test Program

- It runs and tests the methods of the `CashRegister` class

```
1  #  This program tests the CashRegister class.
2
3  from cashregister import CashRegister
4
5  register1 = CashRegister()
6  register1.addItem(1.95)
7  register1.addItem(0.95)
8  register1.addItem(2.50)
9  print(register1.getCount())
10 print("Expected: 3")
11 print("%.2f" % register1.getTotal())
12 print("Expected: 5.40")
```

Program Run

```
1 3
2 Expected: 3
3 5.40
4 Expected: 5.40
```

Test Drivers: Plan Beforehand

- Thinking about the sample program:
  - We add three items totaling $5.40
  - When displaying the method results, we also display messages that describe the values we expect to see
- This is a very important step. You want to spend some time thinking about what the expected result is before you run a test program
- This thought process will help you understand how your program should behave, and it can help you track down errors at an early stage

Test Drivers: Using Modules

- You need to import the class you are testing (here, the `CashRegister` class) into the driver module:

```
1 from cashregister import CashRegister
```

- The specific details for running the program depend on your development environment

Steps to Implementing a Class

Write down all steps to implement a `ChasRegister` class that can

- add multiple items of the same price
- add multiple items with different prices

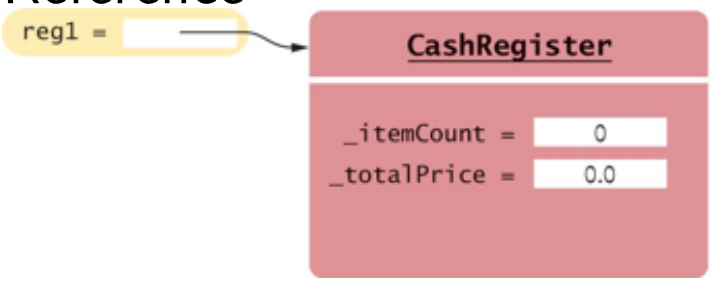
Take a look at sections 9.8 and 9.9 in PfE!

Object References

- In Python, a variable only holds the *memory location* of an object
- The object itself is stored in another location:

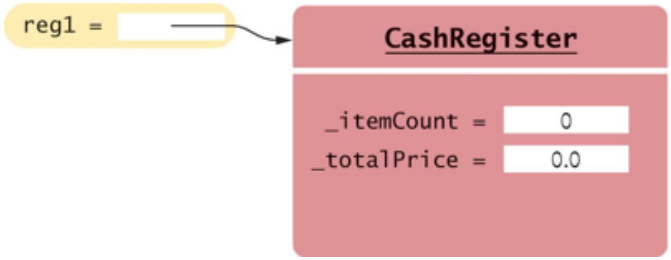
```
1 reg1 = CashRegister()
```

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



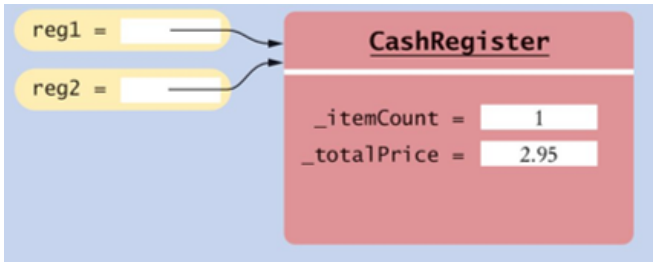
Shared References

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




```
1 reg1 = CashRegister
```

- Shared References




```
1 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.")
```

Object Lifetimes: Creation

- When you create an object the constructor 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

```
1 self._itemCount = 0
```

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

```
1 reg1 = CashRegister()
```