Object Oriented Programming
DRE 7053

NORA Summer School 2024

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

June 11, 2024

OBJECTS AND CLASSES

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.

Contents

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

Classes describe objects with the same behavior, e.g., a *Car* class describes all passenger vehicles that have a certain capacity and shape. Classes *generalize* objects!

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

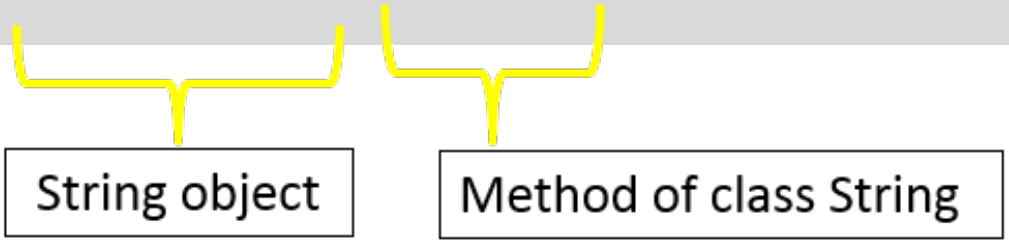
Objects and Programs

- 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 `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()
```



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:

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

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

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

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

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):
- In Python, you don't explicitly declare instance variables
  - Instead, when one first assigns a value to an instance variable, the instance variable is created

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

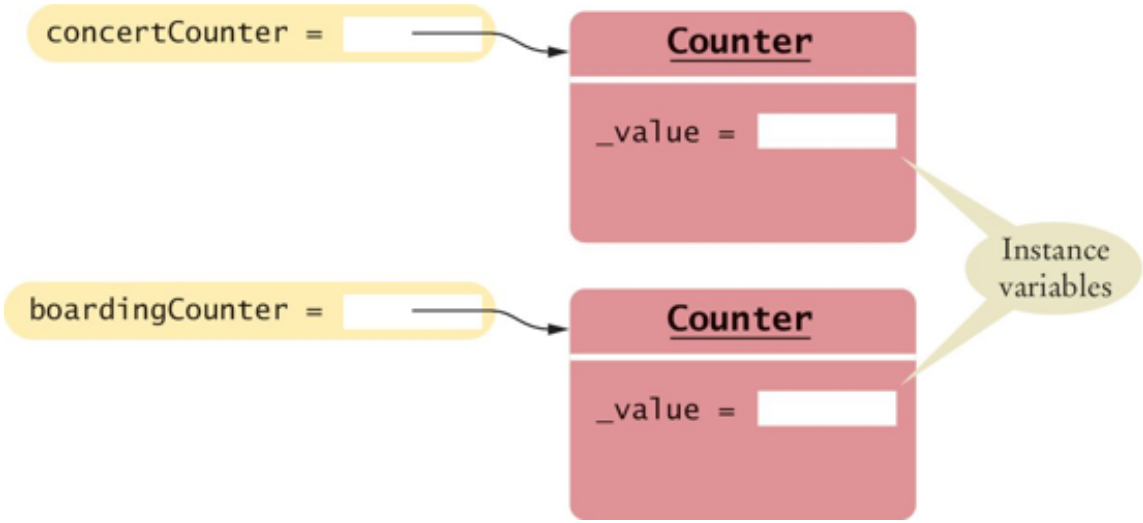
Using the Counter Class

- Next, we invoke methods on our object

```
1 tally.reset()
2 tally.click()
3 tally.click()
4 result = tally.getValue()  # Result is 2
5 tally.click()
6 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

```
1 def click(self) :
2     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

```
1 concertCounter.click()
```

Example of Encapsulation

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

```
1 def getValue(self) :
2     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

```
1  # This program demonstrates the Counter class.
2  # Import the Counter class from the counter module.
3  from counter import Counter
4
5  tally = Counter()
6  tally.reset()
7  tally.click()
8  tally.click()
9
10 result = tally.getValue()
11 print("Value:", result)
12
13 tally.click()
14 result = tally.getValue()
15 print("Value:", result)
```

Complete Simple Class Example

```
1  # This module defines the Counter class. Models a tally
2  # counter whose value can be incremented, viewed, or
       reset.
3  class Counter :
4     ## Gets and treturn the current value of this counter
5     def getValue(self) :
6        return self._value
7
8     ## Advances the value of this counter by 1.
9     def click(self) :
10       self._value = self._value + 1
11
12    ## Resets the value of this counter to 0.
13    def reset(self) :
14       self._value = 0
```

Program Run

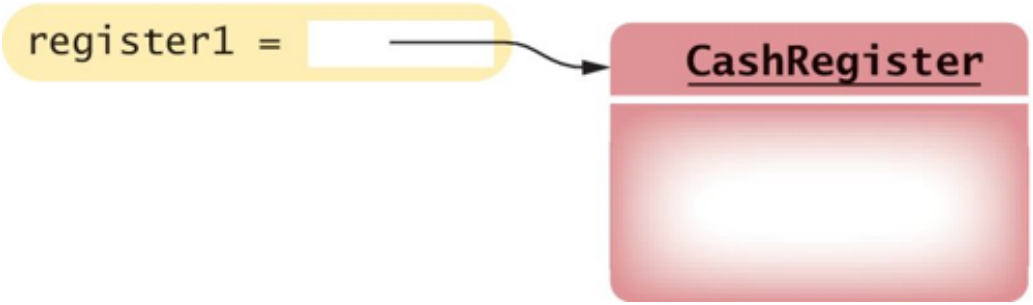
```
1  Value:2
2  Value:3
```

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:

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

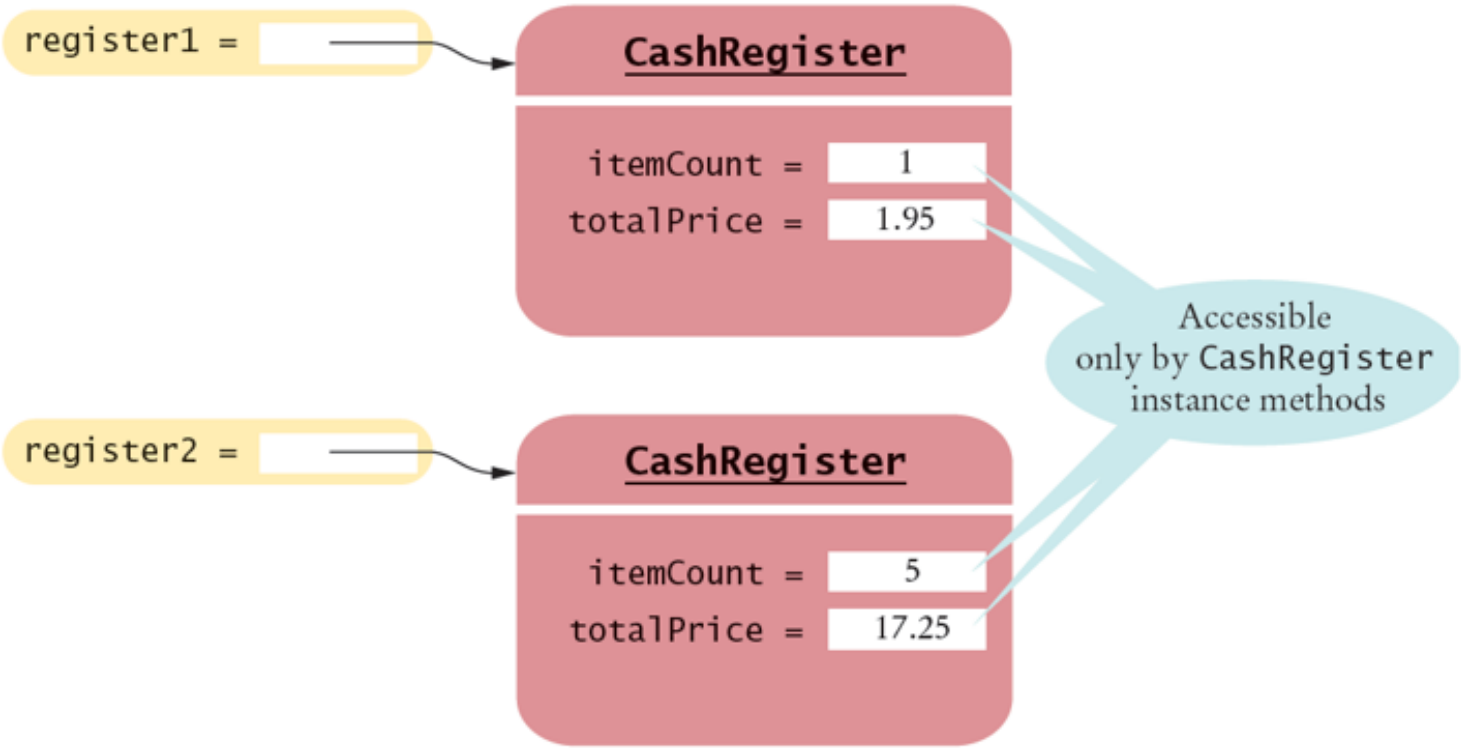
Accessor and Mutator Methods

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

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 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

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

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

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

Default and Named Arguments

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

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

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

Syntax: Constructors

Syntax

```
class ClassName :
   def __init__(self, parameterName_1, parameterName_2, . . .) :
      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
   . . .
```

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

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

```
1 register = CashRegister()
```

Object References

```
1 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

```
1 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:

```
1 register1 = CashRegister()
2 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. The are intended for specific internal purposes (in this case, to initialize a newly created object).