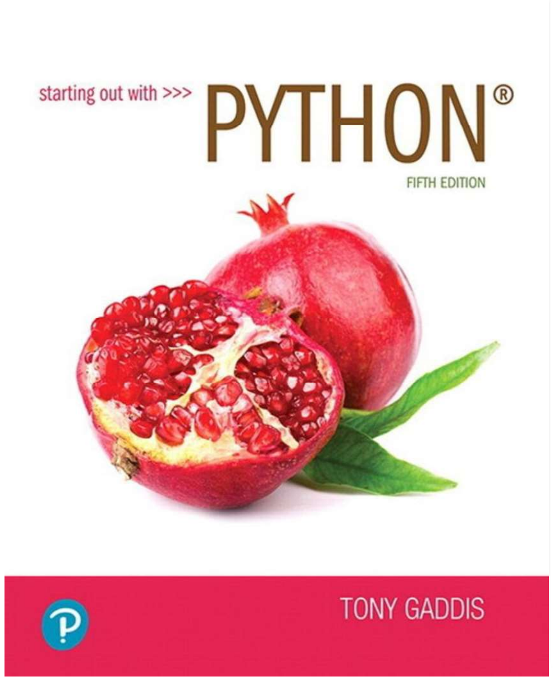
CHAPTER 10

Classes and Object-Oriented Programming



Procedural Programming

- Procedures: synonym for **functions** and sub-routines
- **Procedural programming: writing programs made of functions that perform specific tasks**
  - Functions typically operate on data items that are separate from the functions
  - Data items commonly passed from one function to another
  - Focus: On the algorithm and steps. Create functions that operate on the program's data

Object-Oriented Programming

- **Object-oriented programming: focuses on creating classes and objects**
- **Model the problem on the data involved first, not the big steps.**
- **Class: A programmer defined data type**
- **Object: entity that contains data and functions**
  - Data is known as data attributes and functions are known as methods
    - Methods perform operations on the data attributes
- **Encapsulation: combining data and code into a single object**

Object Oriented Programming

- Recall a CPU only knows how to perform on the **order of 100 operations**
- High level languages such as Python allow us to, seemingly, **create new operations** by defining new functions
- Object oriented languages allow programmers to **create new data types** in addition to the ones built into the language
  - int, float, string, list, tuple, file, dictionary, set

Object Oriented Design Example - Monopoly

If we had to start from scratch what new data types would we need to create?

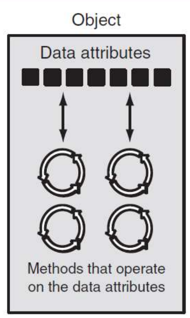
Data Types Needed:

Object Orientation

- **The basic idea of object oriented programming (OOP) is to view your problem as a *collection of objects*, each of which has certain state and can perform certain actions.**
- **Each object has:**
  - **some *data* that it maintains characterizing its current state;**
  - **a set of actions (*methods*) that it can perform.**
- **A programmer interacts with an object by calling its methods; this is called *method invocation*. That should be the *only way* that another programmer interacts with an object.**
- **Significant object-oriented languages include Python, Java, C++, C#, Perl, JavaScript, Objective C, and others.**

Object-Oriented Programming (cont'd.)

Figure 10-1 An object contains data attributes and methods

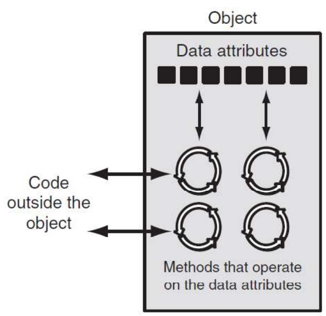


Object-Oriented Programming (cont'd.)

- **Data hiding: object's data attributes are hidden from code outside the object**
  - Access restricted to the object's methods
    - Protects from accidental corruption
    - Outside code does not need to know internal structure of the object
- **Object reusability: the same object can be used in different programs**
  - Example: 3D image object can be used for architecture and game programming

Object-Oriented Programming (cont'd.)

Figure 10-2 Code outside the object interacts with the object's methods



An Everyday Example of an Object

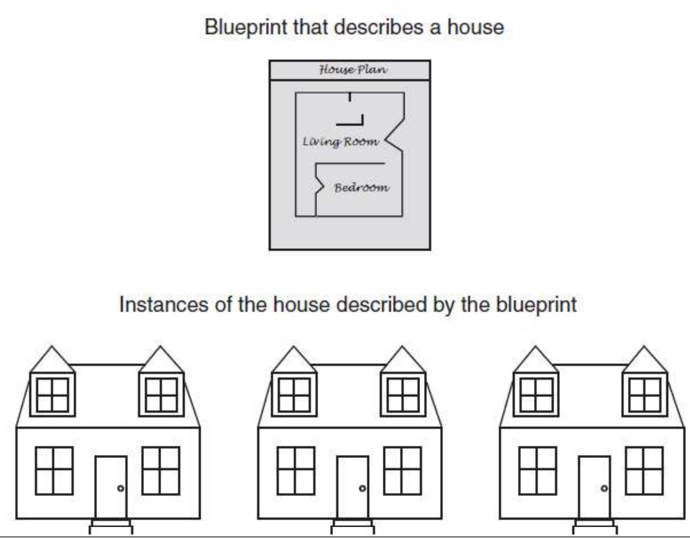
- **<u>Data attributes</u>: define the state of an object**
  - Example: clock object would have `second`, `minute`, and `hour` data attributes
- **<u>Public methods</u>: allow external code to manipulate the object**
  - Example: `set_time, set_alarm_time`
- **<u>Private methods</u>: used for object's inner workings**

Classes

- **<u>Class</u>: code that specifies the data attributes and methods of a particular type of object**
  - Similar to a blueprint of a house or a cookie cutter
- **<u>Instance</u>: an object created from a class**
  - Similar to a specific house built according to the blueprint or a specific cookie
  - There can be many instances of one class

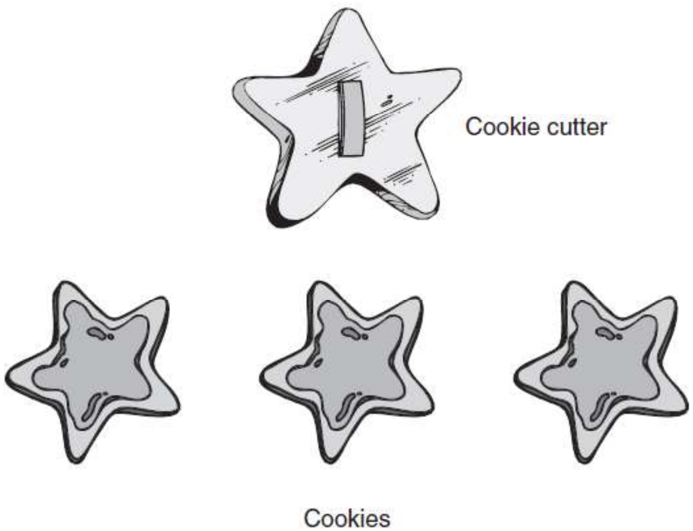
Classes

A blueprint and houses built from the blueprint



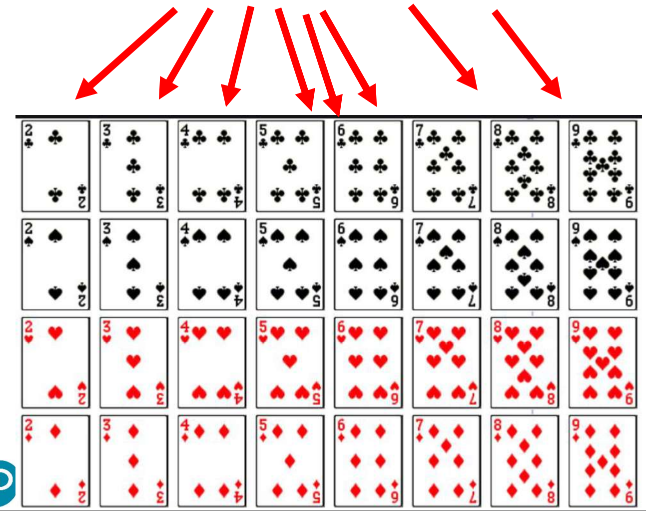
Classes

The cookie cutter metaphor



Simple Example

Class Definition for Playing cards
Playing cards have:
A Rank
A Suit



Define a PlayingCard class and then create objects of type PlayingCard to form a deck or a hand of cards.

A Concrete Example

- **Imagine that you're trying to do some simple arithmetic. You need a Calculator application, programmed in an OO manner. It will have:**
- **Some data: the current value of its**
  - *accumulator* (the value stored and displayed on the screen).
  - History of ops?
  - Memory?
- **Some methods: things that you can ask of the calculator to do:**
  - add a number to the accumulator, subtract a number, multiply by a number, divide by a number, zero out the accumulator value, etc.

Calculator Specification

- In Python, you implement a particular type of object (soda machine, calculator, etc.) with a `class`.
- Let's define a class for our simple interactive calculator.
- Data: the current value of the accumulator.
  Maybe a history of operations? Memory spots, aka variables?
- Methods: any of the following.
  - clear: zero the accumulator
  - print: display the accumulator value
  - add k: add k to the accumulator
  - sub k: subtract k from the accumulator
  - mult k: multiply accumulator by k
  - div k: divide accumulator by k

Yet Another Example

- Example: A soda machine has:
  - **Data:** products inside, change available, amount previously deposited, etc.
  - **Methods:** accept a coin, select a product, dispense a soda, provide change after purchase, return money deposited, etc.
  - Assignment 13

Class Definitions

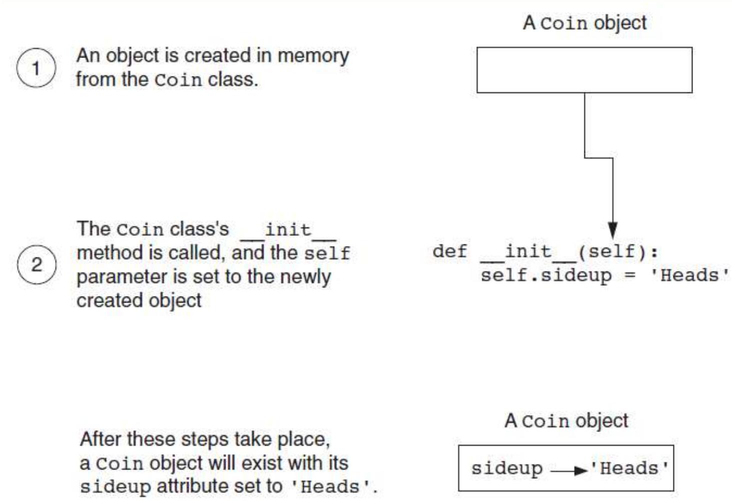
- **<u>Class definition</u>: set of statements that define a class's methods and data attributes**
  - Format: begin with `class ClassName:`
    - <u>Class names typically start with uppercase letter and internal words are capitalized, aka CamelCase</u>
  - Method definition like other Python function definitions
    - <u>`self` parameter</u>: required in every method in the class – references the specific object that the method is working on - **The object the method is working on. The object that called the method**
      name = 'Olivia'
      name.upper() # name is the argument to self

Class Definitions (cont'd.)

- **<u>Initializer method</u>: automatically executed when an instance of the class is created**
  - Initializes object's data attributes and assigns `self` parameter to the object that was just created.
  - Format: `def __init__ (self):`
  - That's two underscores before and after `init`.
  - Typically the first method in a class definition.

Class Definitions (cont'd.)

Actions caused by the `coin()` expression



Class Definitions (cont'd.)

- **To create a new instance of a class call the initializer method**
  - Format: `my_instance = ClassName()`
- **To call any of the class methods using the created instance, use dot notation**
  - Format: `my_instance.method()`
  - Because the `self` parameter references the specific instance of the object, the method will affect this instance
    - Reference to `self` is passed automatically

Hiding Attributes and Storing Classes in Modules

- **An object's data attributes (aka the internal variables) should be difficult to access**
  - To make sure of this, place two underscores (__) in front of attribute name
    - Example: `__current_minute`
- **Classes can be stored in modules**
  - Filename for module must end in .py
  - Module can be imported to programs that use the class

The Circle Class - in Circle.py

```python
import math


class Circle:
    """Model a simple circle.

    Each circle has a center point expressed as x and y coordinates
    and a radius."""

    def __init__(self, x=0, y=0, radius=0):
        self.__x = x
        self.__y = y
        self.__radius = radius

    def get_radius(self):
        return self.__radius

    def get_x(self):
        return self.__x

    def get_y(self):
        return self.__y
```

The Circle Class - in Circle.py

```python
    def get_area(self):
        return self.__radius ** 2 * math.pi

    def get_perimeter(self):
        return 2 * self.__radius * math.pi

    def contains(self, other_circle):
        """Return if other_circle is contained wholly in this Circle."""
        distance = ((self.__x - other_circle.__x) ** 2
                    + (self.__y - other_circle.__y) ** 2)
        distance = math.sqrt(distance)
        return distance + other_circle.__radius <= self.__radius

    def __str__(self):
        return ('x: ' + str(self.__x) + ', y: ' + str(self.__y)
                + ', radius: ' + str(self.__radius))
```

Client Code of Circle Class

```
c1 = Circle(1, 2, 4)
print(c1.__radius)  # causes runtime error
print(c1.__x)  # causes runtime error
c1.__radius = 5
print(str(c1))
c2 = Circle(3, 1, 1)
print(c1.contains(c2))
print(c2.contains(c1))
```

- Recall, variables prefixed with the double underscore (_ _) are hidden from clients.
- Careful, easy to create logic errors

Logic Error in Client Code

- Clients can add attributes (internal data, internal variables) to objects
- Flexible? Yes. Dangerous? You bet!

```
c2 = Circle(3, 1, 1)  # x, y, radius
c2.__x = 12
print('c2.__x in client code', c2.__x)
print('c2.get_x(), in client code', c2.get_x())
print('Result of print(c2) in client code:')
print(c2)
```

```
c2.__x in client code 12
c2.get_x(), in client code 3
x: 3, y: 1, radius: 1
```


The `BankAccount` Class – More About Classes

- Class methods can have multiple parameters in addition to `self`
  - For `__init__`, parameters needed to create an instance of the class
    - Example: a `BankAccount` object is created with a balance
      - When called, the initializer method receives a value to be assigned to a `__balance` attribute
  - For other methods, parameters may be needed to perform required task
    - Example: `deposit` method amount to be deposited

The `__str__` method

- Object's state: the values of the object's attribute at a given moment
- `__str__ method`: return a string version of the object, typically the state of its internal data
- Automatically called when the object is passed as an argument to the `print` function
- Automatically called when the object is passed as an argument to the `str` function

Working With Instances

- <u>Instance attribute</u>: belongs to a specific instance of a class
  - Created when a method uses the `self` parameter to create an attribute
  - Can be local to a method, but continues to exist after that method completes
- If many instances of a class are created, each would has its own set of attributes

Figure 10-8 The `coin1`, `coin2`, and `coin3` variables reference three `Coin` objects

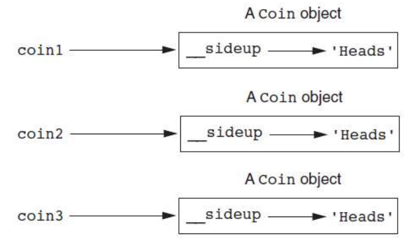
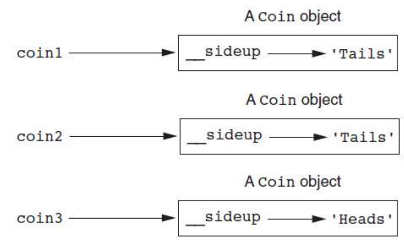


Figure 10-9 The objects after the `toss` method



Accessor and Mutator Methods

- Typically, all of a class's data attributes are private and provide methods to access and change them
- <u>Accessor methods</u>: return a value from a class's attribute without changing it
  - Safe way for code outside the class to retrieve the value of attributes
- <u>Mutator methods</u>: store or change the value of a data attribute
  - You **DO NOT** have to have mutator methods for all (or any) internal attributes

Passing Objects as Arguments

- Methods and functions often need to accept objects as arguments
- When you pass an object as an argument, you are actually passing a reference to the object
  - The receiving method or function has access to the actual object
    - Methods of the object can be called within the receiving function or method, and data attributes may be changed using mutator methods

Other methods

- generally methods with the _ _name_ _ format are not meant to be called directly
- Instead we define them and then the are called with other operators

_ _init_ _	ClassName()		
_ _len_ _	len()		
_ _str_ _	str		
_ _add_ _	+	_ _eq_ _	==
_ _lt_ _	<	_ _le_ _	<=
_ _gt_ _	>	_ _ge_ _	>=

Displaying New Classes in Data Structures

```
c1 = Circle(3, 1, 1)
c2 = Circle(5, 4, 3)
print(c1, c2)
data1 = [c1, c2]
print(data1)
```

Output of print. Great!

```
x: 3, y: 1, radius: 1 x: 5, y: 4, radius: 3
[<__main__.Circle object at 0x000001E56D308640>,
 <__main__.Circle object at 0x000001E56D308670>]
```

Output of print of list. Yuck!

_ _str_ _ and _ _ repr_ _

- print calls the _ _str_ _ method on objects sent to it
- a data structure calls the _ _repr_ _ method on the objects inside it to
- repr for representation
- Like _ _str_ _ but should display the object in a way that we could use to rebuild the object

_ _repr_ _ method for Circle

```
def __repr__(self):
    result = ('Circle(x=' + str(self.__x) + ", y=" + str(self.__y)
              + ', radius=' + str(self.__radius) + ')')
    return result
```

```
c1 = Circle(3, 1, 1)
c2 = Circle(5, 4, 3)
print(c1, c2)
data1 = [c1, c2]
print(data1)
```

```
x: 3, y: 1, radius: 1 x: 5, y: 4, radius: 3
[Circle(x=3, y=1, radius=1), Circle(x=5, y=4, radius=3)]
```

Techniques for Designing Classes

- <u>UML diagram</u>: standard diagrams for graphically depicting object-oriented systems
  - Stands for Unified Modeling Language
- General layout: box divided into three sections:
  - Top section: name of the class
  - Middle section: list of data attributes
  - Bottom section: list of class methods

Figure 10-10 General layout of a UML diagram for a class

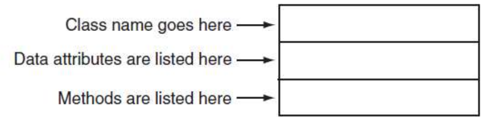
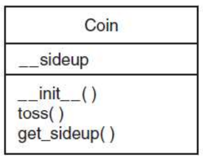


Figure 10-11 UML diagram for the coin class



Finding the Classes in a Problem

- When developing object oriented program, first goal is to identify classes
  - Typically involves identifying the real-world objects that are in the problem
  - Technique for identifying classes:
    1. Get written description of the problem domain
    2. Identify all nouns in the description, each of which is a potential class
    3. Refine the list to include only classes that are relevant to the problem

Finding the Classes in a Problem (cont'd.)

1. Get written description of the problem domain
   - May be written by you or by an expert
   - Should include any or all of the following:
     - Physical objects simulated by the program
     - The role played by a person
     - The result of a business event
     - Recordkeeping items

Finding the Classes in a Problem (cont'd.)

2. Identify all nouns in the description, each of which is a potential class

- Should include noun phrases and pronouns
- Some nouns may appear twice

Finding the Classes in a Problem (cont'd.)

3. Refine the list to include only classes that are relevant to the problem

- Remove nouns that mean the same thing
- Remove nouns that represent items that the program does not need to be concerned with
- Remove nouns that represent objects, not classes
- Remove nouns that represent simple values that can be assigned to a variable

Identifying a Class's Responsibilities

- **A classes responsibilities are:**
  - The things the class is responsible for knowing
    - Identifying these helps identify the class's data attributes
  - The actions the class is responsible for doing
    - Identifying these helps identify the class's methods
- **To find out a class's responsibilities look at the problem domain**
  - Deduce required information and actions

Summary

- **This chapter covered:**
  - Procedural vs. object-oriented programming
  - Classes and instances
  - Class definitions, including:
    - The `self` parameter
    - Data attributes and methods
    - `__init__` and `__str__` functions
    - Hiding attributes from code outside a class
  - Storing classes in modules
  - Designing classes