Objects and Classes

- Functions and Modules Revisited
- Introduction to Classes
- Object Variables and Object References
- Instantiating Objects
- Using Methods in Objects
- Class definitions
- Scope of Data
  - Instance data
  - Local data
- The `self` Reference
- Encapsulation and visibility

- Reading for this Lecture:
  - ***Dawson, Chapter 8***
  - http://introcs.cs.princeton.edu/python/31datatype/
  - http://introcs.cs.princeton.edu/python/32class/

Objects and Classes

- As you may remember, Python is an *object-oriented* programming language
  - An object is a program entity with state and behaviors
  - Objects in a program may represent (and model) real-world entities
  - All data in a Python program are objects
- Objects belong to *classes*...
  - A class can be seen as a blueprint for an object
  - It represents the object's larger category
  - It defines the object's attributes and behaviors
- To clarify, consider functions and modules...

Functions

- **Recall:** A function is a named chunk of code, representing a program behavior, that does one or more of these:
  - Sends back a value, possibly calculating or generating something first
  - Performs operations on data
  - Carries out a set of related commands
- Using functions lets us...
  - break code up into smaller chunks
  - keep parts of the program conceptually separate
  - engage in greater code reuse

Modules

- We can enhance code reuse even more by grouping functions and constants into *modules*.
- If there is a function we find ourselves using in multiple programs, then we can put it into a module with related functions and constants
- When we want to use it, we simply import the module and access what we want via the module name.
- We do not have to rewrite the function because it is already defined within the module, which we can import into as many other code files as we like.
- For example:

```
import math
import random
```

Functions as "messages"

- You can think of a function as a kind of "message" that you send somewhere:
  - To the Python interpreter directly
  - To a module
  - To an object
- This code...

```
str_var = "Hello, world!"
print (str_var)
```

- ...is like saying "Hey, ***Python interpreter***, go print <u>**str_var**</u> to the screen!"

Functions as "messages"

- In contrast, this code...

```
import math
x = math.sqrt(9)
```

- ...is like saying "Hey, ***math module***, go calculate the square root of **<u>9</u>** and give it back!"
- This, of course, brings us back to the principle of *<u>abstraction</u>*, where we do not concern ourselves with behind-the-scenes details
- Objects and classes, then, provide us with another variety of abstraction

Introduction to Classes

- A class defines the <u>attributes and functions</u> (representing the state and behavior) of a specific type of object

- Normally, we access an object by calling a function defined by its class

- We may sometimes access object data directly, via an attribute defined by its class, but this is ***discouraged***

"Classifying" into Classes

- To understand the context of the word "class" in Python, think about the word "classify"
- A class will "classify" a set of "objects" based on similar attributes and behaviors
- The furniture in this room can be classified as `"Furniture"` class objects because of common attributes and behaviors they share
- Entities like `"Hello, world!"` and `"goodbye"` are both classified as `"str"` class objects – strings of characters

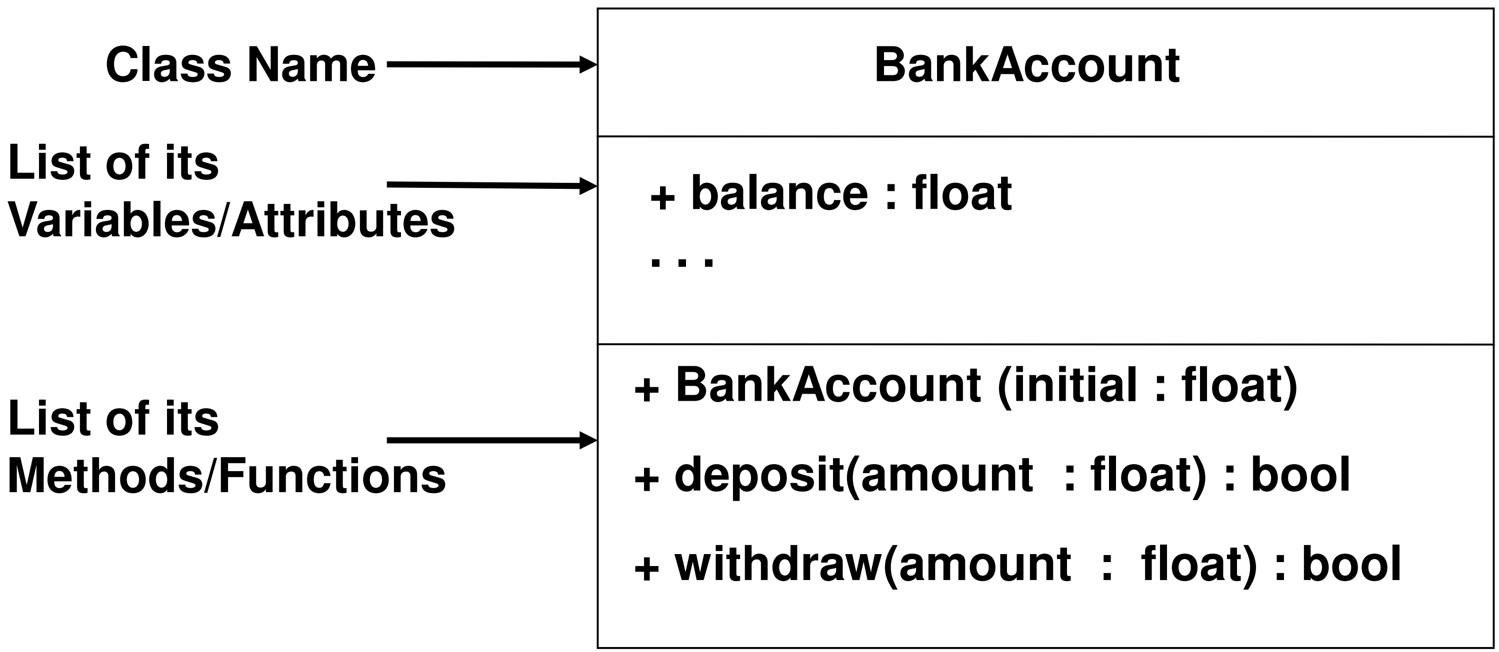
Accessing Class Members

- When we create a variable and assign it a value, we are creating a reference to an object
- The object is what contains the data
- The "reference" is the location of the object in program memory
- We access an object's "members" (i.e., functions and attributes) using the reference variable name and the **"."** operator:

```
object_name = "Hello" #ref. variable
print (object_name.upper()) # upper function
>>> HELLO
```

Example of a Class Definition

- We can draw a diagram of a class to outline its important features before writing code – its name, attributes, and methods



Example of a Class Definition

```
class BankAccount (object):

    # the constructor function
    def __init__(self, initial):
        self.balance = float(initial)

    # the deposit function
    def deposit (self, amount):
        if amount > 0:
            self.balance += amount
            return True
        else:
            print ("Must be greater than zero!")
            return False

   # rest of code..
```

Example of a Class Definition

```
class BankAccount (object):

    # previous...

    # the withdraw function
    def withdraw (self, amount):
        if amount > 0:
            self.balance -= amount
            return amount
        else:
            print ("Must be greater than zero!")
            return False

    # any additional code..
```

Defining a class

- First, you need the class **header** line:

```
class ClassName (object):
```

- After that, all the class code will be *indented* relative to the class header.
- Next you will have your **constructor function**. This contains the code that executes *when you first create a new object* of this type. Start with the header:

```
def __init__ (self):
```

- You may also include some extra parameters:

```
def __init__ (self, name, number):
```

- It will *at least* have the parameter **self**.

Defining a class

- This can be any code you want to execute when the object is first created.
- It is also where you define the attributes for that type of object:

```
def __init__ (self, name, number):
    self.name = name
    self.number = number
```

- Here, the constructor establishes that every object of this type will have the attributes **name** and **number**.
- The **self.** part distinguishes the attribute

Defining a class – string conversion

- In most cases, you will want to define a special *string* function for the class.
- This function determines how your type gets translated to the string type, when you call for a data conversion on it:

```
def __str__ (self):
    return self.name + ", " + str(self.number)
```

- If you have a variable **my_object**, referring to an object of this type, where the values of **name** and **number** are "John" and 27, then this code...

```
print (str (my_object))
```

... will print: `John, 27`

Defining a class - comparisons

- You may also define special *rich comparison* functions for the class, corresponding to the standard comparison and equality operators:

```
__lt__     ->    <
__le__     ->    <=
__eq__     ->    ==
__ne__     ->    !=
__gt__     ->    >
__ge__     ->    >=
```

- These functions determine how two objects of this type are ordered – for example, for sorting.

Defining a class - comparisons

- For example, you may decide two objects of that type should be ordered by their "name" attributes:

```
def __eq__ (self, other):
    return self.name == other.name

def __lt__ (self, other):
    return self.name < other.name
```

- After defining `__eq__` and `__lt__`, you could define the other four in terms of the previous two. For example...

```
def __gt__ (self, other):
    return not (self < other or self == other)
```

- Or, you may choose to do it your own way

Defining a class

- You can also define other functions for your class:

```
def my_function (self):

    print ("Hello, my name is:", self.name)
    print ("And my number is:", self.number)
```

- Your function *must* have the parameter **self**. However, you can also include others:

```
def my_function2 (self, day_of_week):

    print ("Hello, my name is:", self.name)
    print ("And today is:", day_of_week)
    return True
```

Creating and Using Objects

- Creating a BankAccount object:

```
my_account = BankAccount(100) # constructor
```

- Accessing BankAccount methods:

```
my_money = my_account.balance
print ("My balance is $" + str(my_money))
my_account.deposit(50.0)
print ("My balance is now $", end="")
print (my_account.balance)
```

- Of course, we could just do this...why don't we?

```
my_account.balance += 50.0
```

Prototype for a Class Definition

- We make an attribute/function *private* when we want to **prevent** access to it from code written outside the class
- Conversely, we let it be *public* when we want to **allow** access from code written *outside* the class
- `balance` in the BankAccount class is public
- Normally, we declare **attributes** to be *private* and **functions** to be *public*
- We will see some valid exceptions later

Creating Objects

- We use the *class name*, along with *parameters*, to create an object

```
my_account = BankAccount(100)
```

Variable

This calls the BankAccount *constructor*, which is a special function that initializes the object. Notice "self" is *not* a parameter here!

- Creating an object is called *instantiation*
- An object is an *instance* of a particular class
- `my_account` is assigned a ***reference*** to an object of type **`BankAccount`** that encapsulates it's data – the balance

Invoking Functions

- Once an object has been instantiated, we can use the *dot operator* to invoke, or ***call***, any of the object's functions

```
success = my_account.deposit(33.45)
```

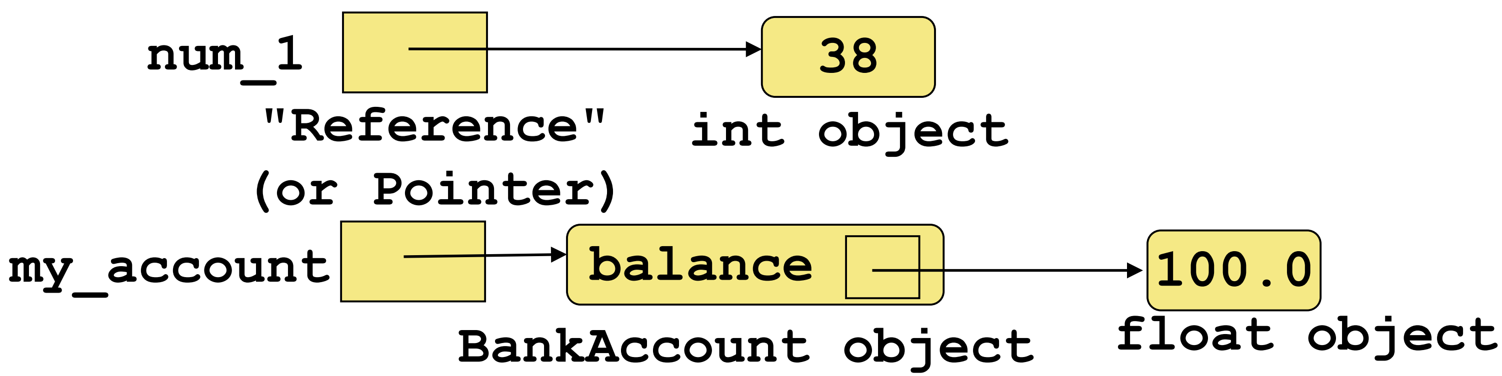
- Notice we only supply the balance, **not self**
- A function call on an object might:
  - Ask the object for some information OR
  - Ask the object to perform a service OR
  - Doing something to the state of the object
- We send the object a *message*, and we may get back a reply (as *data*)

Leveraging OOP

- Classes and objects allow us to ***encapsulate*** data and procedures, useful for (among other things):
  - Making code neater and easier to use
  - Security of object data
  - Maintaining logical structure
  - Making program easier to understand
- Example: **Address** class, later in lecture

References

- As mentioned earlier, a variable does not hold the actual data; instead, it holds a "reference" to the data object.
- An object reference can be thought of as a "pointer" to the location of the object in memory
- Rather than dealing with arbitrary address values, we often depict a reference *graphically*



References

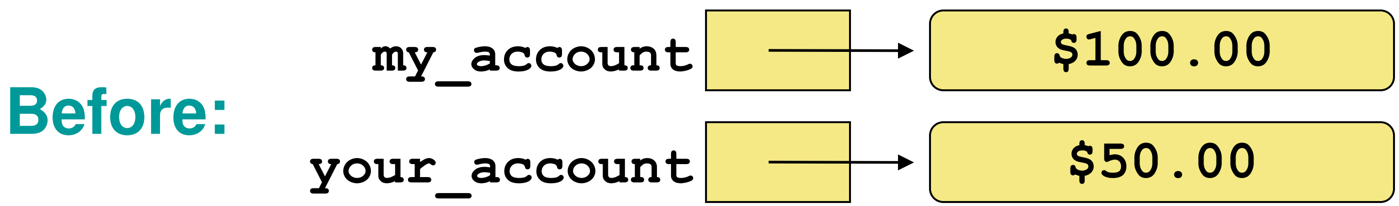
- Because the variable holds the *reference*, not the actual object, we can do things like this:

```
strings = ["", "", "", "", ""]
strings[0] = "foo"
strings[1] = "Hello World"
strings[2] = "To be or not to be, "
strings[3] = ""
strings[4] = (the entire text of Tolstoy's War and Peace)
```

- The data sizes are not a problem!

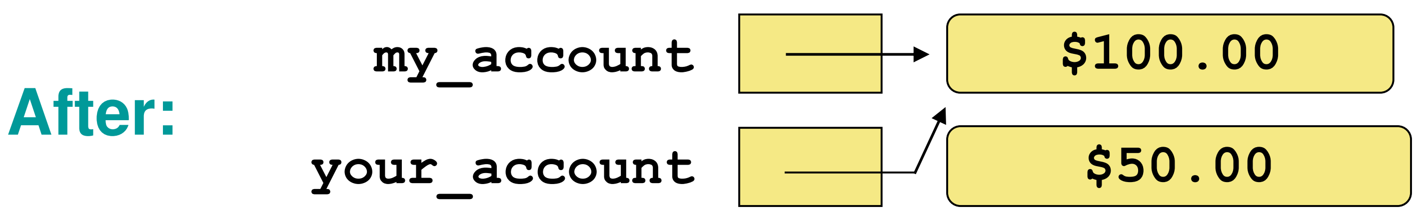
Reference Assignment

- When we re-assign a variable, we are storing a new **reference**:




```
if my_account == your_account:
    print ("The Same")   # no!

your_account = my_account
```



Garbage: See later slide

```
if my_account == your_account:
    print ("The Same")   # yes!
```

Aliases

- Two or more variables that refer to the same object are ***aliases*** of each other
- One object can be accessed using more than one variable
- Changing an object via one variable changes it for all of its aliases, because there is really only one object
- Aliases can be useful, but should be managed carefully (*Do you want **me** to be able to withdraw money from **your** account? I doubt it!*)

The `None` object

- Some languages will allow a variable to be empty, or **`null`**
- We cannot do this in Python, but we *can* point the variable to the **`None`** value
- This is good for cases where we want a variable to exist but not have a definite value yet

```
my_var = None
```

- Later, we can assign the variable another value

```
my_var = "Hello World"
```

- The **`None`** value is considered "false"

Garbage Collection

- When there are no longer any variables containing a reference to an object (e.g. the $50.00 on the earlier slide), the program can no longer access it

- The object is useless and is considered *garbage*. Lots of garbage objects can consume program memory.

- Therefore, these objects must be *garbage collected*.

- Some languages, like Java and Python, perform *automatic garbage collection* and returns an object's memory to the system for future use

- In other languages such as C/C++, the programmer must write explicit code to perform the garbage collection

Garbage Collection

- Reassigning the variable’s value makes the object garbage (unavailable):

Before: `my_account` → `$100.00`

```
my_account  = None
```

After: `my_account` → `None`

Garbage now

`$100.00`

Also: `print ("Hey!")`

Garbage now

`"Hey!"`

Garbage Collection

- If a variable is not pointing to a compatible object, any call to an attribute or function of that object will cause your program to fail.

```
my_account = BankAccount(100.00)

print (my_account.balance)  # OK

my_account = None

print (my_account.balance)  # Fails
```

Writing Classes

- True object-oriented programming is based on classes that represent objects with ***well-defined attributes and functionality***
- The programs we've written in previous examples have used classes from the standard Python types
- Now we will begin to design programs that rely on classes that we write ourselves

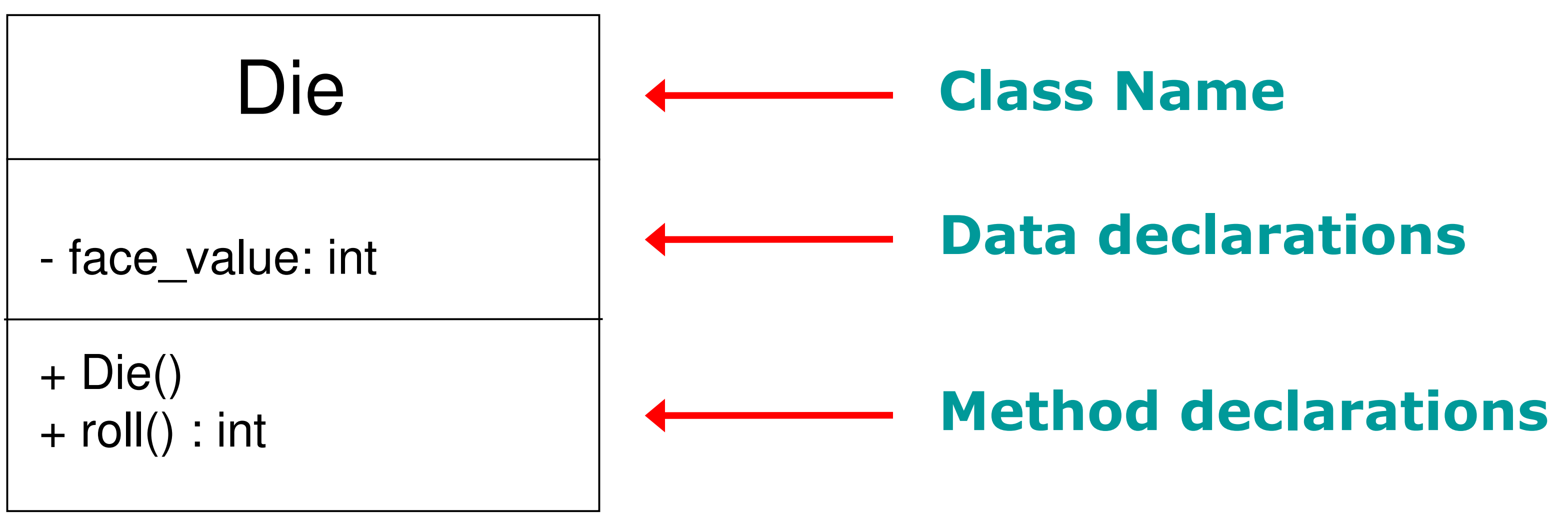
Classes and Objects

- An object has *state* and *behavior*
- Consider a 6-sided die (singular of dice)
  - It's state can be defined as ***the face showing***
  - It's primary behavior is that it ***can be rolled***
- We can represent a die in software by designing a class called `Die` that models this state and behavior
  - The class is the **<u>blueprint</u>** for a die **<u>object</u>**
- We can then instantiate as many die objects as our program needs: 2, 3, 100, etc.

Classes

- A class has a name and can contain data declarations and/or method declarations
- A UML class diagram shows it as follows:

A way of expressing info about, and relationships among, classes. More to come...



Classes

- The **values of the attributes** define the ***state*** of an object created from the class

- The **functionality of the methods** define the ***behaviors*** of an object created from that class "blueprint"

- For our `Die` class, an integer represents the current value showing on the face - its ***state***

- One of the methods represents a ***behavior*** of "rolling" the die by setting its face value to a random value between one and six

Constructors

- A *constructor* is a special method that is used to set up an object when it is initially created
- Its name will be **__init__**
- It will always have the parameter **self**, plus others
- For **`Die`**, constructor is used to set the initial face value of each new die object to one

```
my_die = Die()
```

- In Python, the **constructor** defines the class data: its attributes

Constructors

- To create an attribute inside your constructor, you will need two things. The **<u>self</u>** reference and the attribute name (i.e., variable name)

```
def __init__ (self):
    self.first_attribute = 1
    self.second_attribute = True
    self.third_attribute = "Hello World"
```

- What this does is create a variable (for the object) which holds that value
- Parameters to a constructor are usually used for setting these initial values

The __str__ Function

- All classes that represent objects should define a `__str__` function
- The `__str__` function returns a string that represents the object in some way
- It is called *automatically* when a reference to an object is is passed to the **`print`** or **`str`** functions

```
print (my_die)
s = str (my_die)
```

class declaration

```
class Address (object):
```

constructor

```
    def __init__ (self, first, last, st_add, city,
state, zip):
        self.__first = first
        self.__last = last
        self.__st_add = st_add
        self.__city = city
        self.__state = state
        self.__zip = zip
```

class attributes defined

__str__ function

```
    def __str__(self):
        line_1 = self.__first + " " + self.__last + "\n"
        line_2 = self.__stAdd + "\n"
        line_3 = self.__city + ", " + self.__state + " "
+ self.__zip + "\n"
        return line_1 + line_2 + line_3
```

Data Scope

- **<u>Recall:</u>** The *scope* of data is the area in a program in which that data can be referenced (used)
- Instance data is declared at the class level (inside the constructor) and it exists for as long as the object exists
- The instance data can be used elsewhere within the class code.
- Data declared within a function, called *local data*, can be used only within that function and exists only for as long as that function is executing

Data Scope

- Instance and local data

```
class SomeClass (object):

  def __init__ (self):
    self.__value = 10

  def __str__ (self):
    return "Value: " + str(self.__value)

  def some_function (self):
    local_value = 5
    return local_value ** 2
```

Class-level scope for **self.__value**

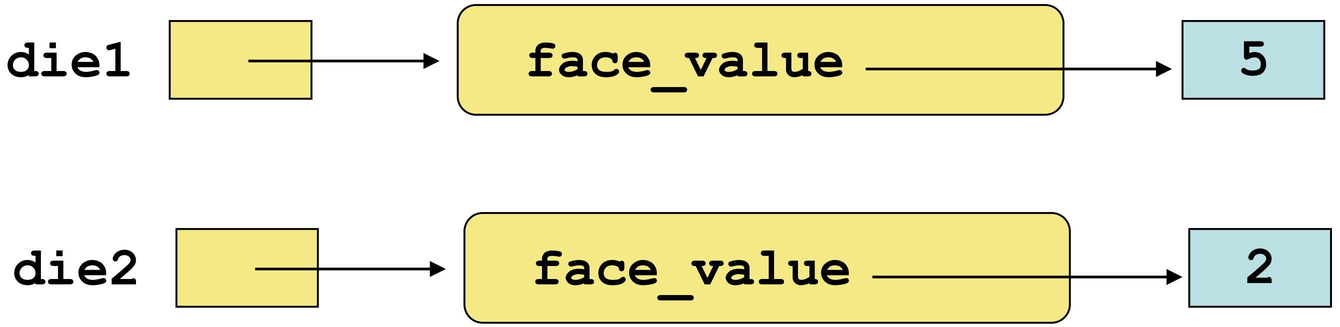
Scope for **local_value**

Instance Data

- The `face_value` attribute in the `Die` class is called *instance data* because each instance (object) created has a corresponding face value
- A class declares the type of the data, but it *does not reserve any memory space for it*
- Every time a new `Die` **<u>object</u>** is created, a new `face_value` variable is created as well
- The objects of a class share the code in the method definitions, but each object ***has its own data space in memory*** for instance data
- The instance data goes out of scope when the last reference to the object is set to null

Instance Data

- We can depict the two `Die` objects from the `DicePlayer` class as follows:



Each object maintains its own `face_value` variable, and thus its own state

Local Data

- Local data, then, is any variable defined inside the function body:

```
def some_function (self):
    local_value = 5
    return local_value ** 2
```

- The variable named **`local_value`** is accessible **only** inside **`some_function`**
- We could use the name **`local_value`** in a different function, but it wouldn't be the same variable

The `self` Reference

- `self` allows an object to refer to itself
- The `self` reference used inside a function refers to the object for which the function is executed
- Suppose `self` is used in the Die class `__str__` function as follows:

```
return str(self.__face_value)
```

- For each of the Die objects, `self` refers to and returns:

```
str(die_1)  → 5
str(die_2)  → 2
```

The `self` Reference

- The `self` reference can be used to ***distinguish*** the instance variable names of an object from local function parameters with the same names
- Without the `self` reference, we need to invent and use two different names that are synonyms
- The `self` reference lets us use the same name for instance data and a local variable or function parameter and ***resolves ambiguity***
- Using the same name, with **self** for distinguishing, makes programming more straight-forward

Static class elements

- Most of the code for a class will be geared towards serving as a blueprint for objects of that type – attributes for object *data* and functions for object *behaviors*.
- However, you can also have *static* attributes and functions, which will belong to the class as a whole.
- This is because they are relevant not to specific objects but, rather, to all objects or something else.
- These elements, you will not access via an object. Rather, you will access them via the class itself.

Static class elements

- Here are some examples of static elements in a class:

```
class Student (object):

    next_student_id = 100

    def __init__(self, name):
        self.__id = Student.next_student_id
        Student.next_student_id += 1
        self.__name = name

    def __str__ (self):
        return "Name: " + self.__name + ", ID: " +
str(self.__id)

    def num_students():
        return Student.next_student_id - 100
```

Static class elements

- Once the **Student** class is defined, you can access its static elements using the class name. Example:

```
>>> s = Student("Bob")
>>> print (s)
Name: Bob, ID: 100
>>> print (Student.num_students())
1
>>> s2 = Student("Susie")
>>> print (s2)
Name: Susie, ID: 101
>>> print (Student.num_students())
2
>>> print (Student.next_student_id)
102
>>>
```

Visibility Modification

- In Python, we accomplish encapsulation (where an object handles its own data) through the appropriate use of *visibility syntax*
- Members of a class that are declared with ***public*** *visibility* are accessible outside the class code
- Members of a class that are declared with ***private*** *visibility* can be referenced only within the class code itself.
- They cannot be accessed directly from outside, only indirectly through other class functions

Visibility Modification

- Public variables violate the spirit of encapsulation because they allow the client to "reach in" and modify the object's internal values directly
- Therefore, instance variables ***should not*** be declared with public visibility
- Instead, you should make the instance variables *private* and allow access only through special **getters** and **setters**.
- This protects instance data and preserves the spirit of encapsulation.

Visibility Modification

- Functions can also be public or private
- Functions that provide the object's services are declared with public visibility so that they can be invoked by clients
- Public functions are also called *service functions*
- Functions that simply to assist other functions are called *support or helper functions*
- Since a support function is not intended to be called by a client, it should be *private*, not public

Controlling Visibility

- To make an attribute or function **<u>public</u>**, simply create the (attribute or function) name like you normally would.
- Here, there is no need for anything special
- Create a public class <u>attribute</u>:

```
def __init__ (self):
    self.public_attribute = 1
```

- Create a public class <u>function</u>:

```
def public_function (self):
    # function code...
```

Controlling Visibility

- To make an attribute or function **private**, begin the (attribute or function) name with two underscores.
- This tells Python that they should be private
- Create a private class attribute:

```
def __init__ (self):
    self.__private_attribute = 1
```

- Create a private class function:

```
def __private_function (self):
    # function code...
```

Accessing a Private Attribute

- When an attribute is private, the object can still make it available, on its own terms.
- We will use an example from the Die class
- Create a getter for a private attribute:

```
@property
def face_value(self):
    return self.__face_value
```

- Create a setter for a private attribute:

```
@face_value.setter
def face_value(self, new_value):
    if 1 <= new_value <= Die.MAX:
        self.__face_value = new_value
    else:
        print ("Error!")
```

Accessing a Private Attribute

- Once you have done this, and you have a reference to an object, you can use the getter or setter **as if** it were a public attribute.
- Get the attribute value:

```
print ("Face value:", my_die.face_value)
```

- Set the attribute value:

```
my_die.face_value = 5
```

- But, if we try to use an invalid value...

```
my_die.face_value = 10
>> Error!
```

Visibility Types - Summary

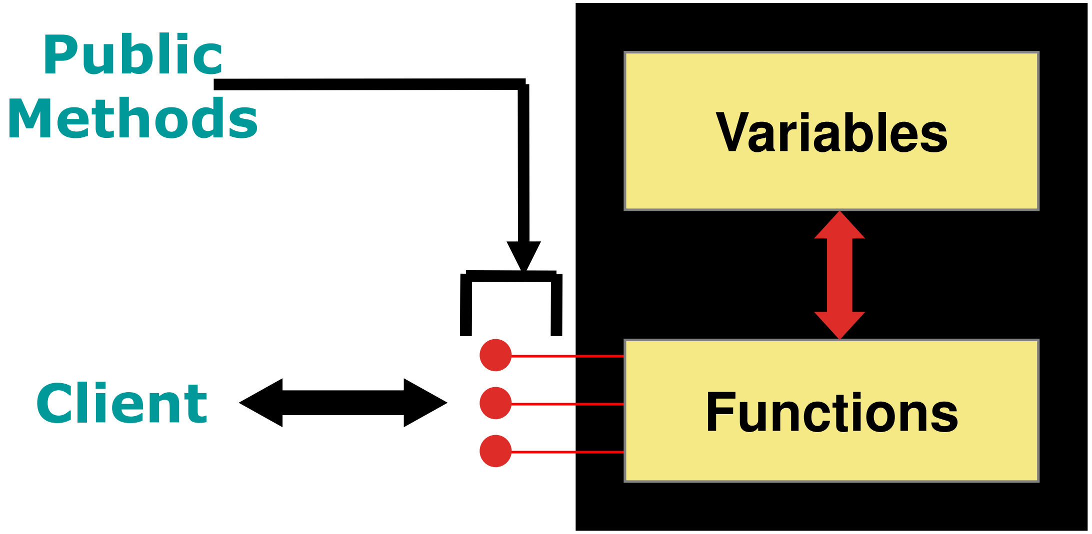
	`public`	`private`
Variables	**Violate encapsulation**	**Enforce encapsulation**
Functions	**Provide services to clients**	**Support other functions in the class**

Interface of an object

- We can take one of two views of an object:
  - internal - the details of the variables and methods of the class that defines it
  - external - the services that an object provides and how the object interacts with the rest of the system
- From outside, the object is an *encapsulated* entity providing a set of specific services
- These services define the object's *interface* - the manner in which we ("we" being other parts of the program) are able to interact with that object.

Black Box Metaphor

- An object can be thought of as a *black box* -- its inner workings are encapsulated or hidden from the client
- The client invokes the interface functions of the object, which manages the instance data



A Class: From Inside and Out

```
class X (object):

  def __init__ (self):
    self.__a = 15
    self.__c = 'c'

  def public_f (self):
    return self.__c

  def __private_f (self):
    print (self.__a**2)
```

How it looks on the inside, from the inside-class point of view.

```
class X (object):

  def __init__ (self):
    self.__a = 15
    self.__c = 'c'

  def public_f (self) :
    return self.__c
```

How it looks to other objects, from other classes. The outside-class point of view.