# **Programming with Python**

# Chapter 8:

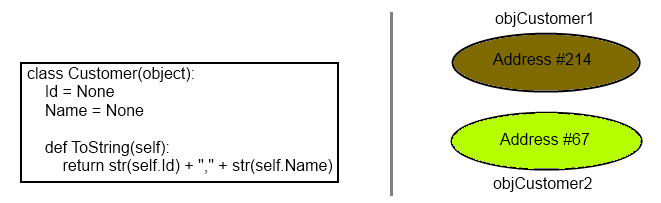


* Classes
* Fields
* Constructors
* Attributes/Properties
* Methods
* Inheritance

## Classes

Classes are a way of grouping data and processing code. The data in a class is defined as variables (called Fields) and the processing code as functions (called Methods).

Here is one example:



Here is similar example:

#--- Make the class ---

class Person(object):

FirstName = ""

def ToString(self):

return self.FirstName

#End of class

# --- Use the class ----

# by making an object!

objP1 = Person()

objP1.FirstName = "Bob"

objP2 = Person()

objP2.FirstName = "Sue"

print(objP1.ToString())

print("-------------")

print(objP2.ToString())

### A Standard Class Pattern

Class are typically made of Fields, Constructors, Properties, and Methods. Like scripts, class code follows a general design pattern.

class Person(object):

"""Doc String """

#-------------------------------------#

#Desc:

#Dev:

#Date:

#ChangeLog: (When,Who,What)

#-------------------------------------#

#--Fields--

#--Constructor--

#Attributes

#--Properties--

#--Methods--

#--End of class--

Next, we will look at each of the items (Fields, Constructors, Attributes, Properties, and Methods).

## Fields

Fields are the data members of a class.

#--- Make the class ---

class Person(object):

""" Base Class for Personal data """

#-------------------------------------#

#Desc: Holds Personal data

#Dev: RRoot

#Date: 12/12/2012

#ChangeLog:(When,Who,What)

#-------------------------------------#

**#--Fields--**

**FirstName = ""**

#--Constructor--

#Attributes

#--Properties--

#--Methods--

def ToString(self):

return **self.FirstName**

#--End of class--

# --- Use the class ----

# by making an object!

objP1 = Person()

objP1.FirstName = "Bob"

objP2 = Person()

objP2.FirstName = "Sue"

print(objP1.ToString())

print("-------------")

print(objP2.ToString())

Next, you will try creating your own fields in a class.



## LAB 8-1:

1. Create a class called Person (using the code from the last example)
2. Add a LastName field to the Person class in addition to the FirstName field.
3. Add code that returns the first and last name in the ToString() method



## Constructors

Constructors are special methods that run automatically when an object is created from the class. They are often used to set the initial values of Field/Attribute data.

#--- Make the class ---

class Person(object):

""" Base Class for Personal data """

#-------------------------------------#

#Desc: Holds Personal data

#Dev: RRoot

#Date: 12/12/2012

#ChangeLog:(When,Who,What)

#-------------------------------------#

#--Fields--

**FirstName = ""**

**#--Constructor--**

**def \_\_init\_\_(self, FirstName):**

**#Attributes**

self.FirstName = FirstName

#--Properties--

#--Methods--

def ToString(self):

return self.FirstName

#--End of class--

# --- Use the class ----

# by making an object!

**objP1 = Person("Bob")**

#objP1.FirstName = "Bob"

**objP2 = Person**("Sue")

#objP2.FirstName = "Sue"

print(objP1.ToString())

print("-------------")

print(objP2.ToString())

Next, you will try creating your own Constructor in a class.



## LAB 8-2:

1. Create a constructor for your Person class
2. Add FirstName and LastName attributes to the constructor of your Person class



## Attributes

Attributes are "virtual" fields that hold internal data. A field will be created for you invisibly when you use the following syntax in the constructor.

#--- Make the class ---

class Person(object):

""" Base Class for Personal data """

#-------------------------------------#

#Desc: Holds Personal data

#Dev: RRoot

#Date: 12/12/2012

#ChangeLog:(When,Who,What)

#-------------------------------------#

**#--Fields--**

**~~#FirstName = ""~~**

#--Constructor--

def \_\_init\_\_(self, **FirstName = ""**):

#Attributes

***self.FirstName = FirstName***

#--Properties--

#--Methods--

def ToString(self):

return self.FirstName

#--End of class--

# --- Use the class ----

# by making an object!

objP1 = Person("Bob")

objP2 = Person()

objP2.FirstName = "Sue" ***# This works since the Field still exist!***

print(objP1.ToString())

print("-------------")

print(objP2.ToString())

You often hide these "virtual" fields by using (2) underscores before the attributes name. This makes the fields ***private***. However the data can accessed through methods and properties (in the abstract). By convention, programmers should respect these hidden fields, like \_\_FirstName, as being private, but Python does not strongly enforce this.

#--- Make the class ---

class Person(object):

""" Base Class for Personal data """

#-------------------------------------#

#Desc: Holds Personal data

#Dev: RRoot

#Date: 12/12/2012

#ChangeLog:(When,Who,What)

#-------------------------------------#

#--Fields--

#FirstName = ""

#--Constructor--

def \_\_init\_\_(self, FirstName = ""):

#Attributes

**self.\_\_FirstName = FirstName # Now it's private**

#--Properties--

#--Methods--

def ToString(self):

return self.\_\_FirstName

#--End of class--

# --- Use the class ----

# by making an object!

objP1 = **Person("Bob")**

objP2 = Person()

**objP2.\_\_FirstName = "Sue"** ***#This will be ignored.***

print(objP1.ToString())

print("-------------")

print(objP2.ToString())

Next, we will look at adding more flexibility to your attributes by creating Properties.

### Properties

You should provide access to the private attributes using public property procedures. This allows you to add code for both validation and error handling.

class Person(object):

""" Base Class for Personal data """

#-------------------------------------#

#Desc: Holds Personal data

#Dev: RRoot

#Date: 12/12/2012

#ChangeLog:(When,Who,What)

#-------------------------------------#

#--Fields--

#FirstName = ""

#--Constructor--

def \_\_init\_\_(self, FirstName = ""):

#Attributes

self.\_\_FirstName = FirstName

#--Properties--

**#FirstName**

**@property # (getter or accessor)**

**def FirstName(self):**

**return self.\_\_FirstName**

**@FirstName.setter # (setter or mutator)**

**def FirstName(self, Value):**

**self.\_\_FirstName = Value**

#--Methods--

def ToString(self):

return self.FirstName

#--End of class--

# --- Use the class ----

# by making an object!

objP1 = Person("Bob")

objP2 = Person()

objP2.FirstName = "Sue"

print(objP1.ToString())

print("-------------")

print(objP2.ToString())

Next, you will try creating your own Attributes and Properties in a class.



## LAB 8-3:

1. Make the FirstName and LastName attributes private in the constructor of your Person class
2. Add a Accessors and Mutators property procedures for the FirstName and LastName attributes



## Methods

Functions inside of a class are called Methods. These methods allow you to organize your processing statements into named groups (which is exactly what functions in scripts do).

The ToString() method is one of the most common methods you will find in any class. So much so, that in Python has a built-in method that preforms this same task (\_\_str\_\_).

You can "Override" this build-in method, by including your own definition like so…

#--Methods--

def ToString(self):

"""*Explictly* returns field data"""

return self.FirstName

**def \_\_str\_\_(self): # NOTE both double underscores in the name**

**"""*Implictly* returns field data"""**

**return self.FirstName**

# --- Use the class ----

# by making an object!

objP1 = Person("Bob")

objP2 = Person()

objP2.FirstName = "Sue"

**print(objP1.ToString()) # Explicit**

**print("-------------")**

**print(objP2) # Implicit**

There are many special methods that are included in every class. For example, there is one called \_\_class\_\_ that is part of the parent class (object).

# --- Use the class ----

# by making an object!

objP1 = Person("Bob")

objP2 = Person()

objP2.FirstName = "Sue"

print(objP1)

print("-------------")

print(objP2)

print(objP2***.\_\_class\_\_***)  **# Prints out <class '\_\_main\_\_.Person'>**

Another special method is the destructor \_\_del\_\_.

***NOTE: Destructors are considered an advanced feature and should be used with care. We will not go into them here.***

**def \_\_del\_\_(self):**

"""Implictly called when object is destroyed"""

*# Add cleanup code as needed*

### Static Methods

If you want to include a method that can be called from the class without making an object first you can do so as follows:

class Math(object):

**@staticmethod**

**def Add(Value1, Value2):***# You do not need the self keyword*

**return Value1 + Value2**

#--End of class--

#-- Test the class --

Sum = **Math.Add(5, 6)**

print(Sum)

You can even have classes with both instance methods and static methods. One common example, used in older languages without "Garbage collection", is to add a static object counter to a class. Here is an example:

#--- Make the class ---

class Person(object):

""" Base Class for Personal data """

#-------------------------------------#

#Desc: Holds Personal data

#Dev: RRoot

#Date: 12/12/2012

#ChangeLog:(When,Who,What)

#-------------------------------------#

#--Fields--

#FirstName = ""

**\_\_Counter = 0 # Hey Devs, please consider this a private field!**

#--Constructor--

def \_\_init\_\_(self, FirstName = ""):

#Attributes

self.\_\_FirstName = FirstName

**Person.\_\_Counter += 1** # You do not use the self keyword

#--Properties--

#FirstName

@property #getter(accessor)

def FirstName(self):

return self.\_\_FirstName

@FirstName.setter #(mutator)

def FirstName(self, Value):

self.\_\_FirstName = Value

#--Methods--

def ToString(self):

"""Explictly returns field data"""

return self.FirstName

def \_\_str\_\_(self):

"""Implictly returns field data"""

return self.FirstName

**@staticmethod**

**def GetObjectCount():** # You do not need the self keyword

**return Person.\_\_Counter**

# --- Use the class ----

objP1 = Person("Bob")

**print(Person.GetObjectCount())**

objP2 = Person()

objP2.FirstName = "Sue"

**print(Person.GetObjectCount())**

print(objP1)

print("-------------")

print(objP2)

**print(Person.\_\_Counter) # This stops the *obvious* access with an error!**

**# AttributeError: type object 'Person' has no attribute '\_\_Counter'**

**# But…**

**# if you set the \_\_Counter to a value Explicitly you do not receive and error! So, your dev's are really on the Honor system!**

### Private methods

If you want a method to be for internal processing only, in other words private to the object or class, you can name it with (2) underscores to indicate this.

#--- Make the class ---

class Person(object):

""" Base Class for Personal data """

#--Fields--

#FirstName = ""

\_\_Counter = 0 #Hey Devs, please consider this a private field!

#--Constructor--

def \_\_init\_\_(self, FirstName = ""):

#Attributes

**self.\_\_FirstName = FirstName # Private Attribute**

**Person.\_\_SetObjectCount() # Private Method**

#--Properties--

#FirstName

@property #getter(accessor)

def FirstName(self):

return self.\_\_FirstName

@FirstName.setter #(mutator)

def FirstName(self, Value):

self.\_\_FirstName = Value

#--Methods--

def ToString(self):

"""Explictly returns field data"""

return self.FirstName

def \_\_str\_\_(self):

"""Implictly returns field data"""

return self.FirstName

@staticmethod

def GetObjectCount(): # You do not need the self keyword

return Person.\_\_Counter

@staticmethod

def **\_\_SetObjectCount**(): # This is a private and static method

Person.\_\_Counter += 1

# --- Use the class ----

# by making an object!

objP1 = Person("Bob")

print(Person.GetObjectCount())

objP2 = Person()

objP2.FirstName = "Sue"

print(Person.GetObjectCount())

Next, you will try creating your own Methods in a class.



## LAB 8-4:

Add code to return the FirstName and LastName attribute in the \_\_str\_\_ method of your Person class



Next, we will look at inheritance.

## Inheritance

One of the main advantages for OOP programming is the ability to inherit code from a parent class. This is known as extending the parent class through the mechanism of inheritance.

This can be documented like this:

Child class ->Parent class

People also use these terms for the same thing!

Derived class -> Base class

Sub class -> Super class

Here is an example:

**class Employee(Person):**

""" Class for Employee data """

#-------------------------------------#

#Desc: Holds Employee data

#Dev: RRoot

#Date: 12/12/2012

#ChangeLog:(When,Who,What)

#-------------------------------------#

#--Fields--

#--Constructor--

**def \_\_init\_\_(self, Id = ""):**

**#Attributes**

**self.\_\_Id = Id**

#--Properties--

**# Id**

**@property #getter(accessor)**

**def Id(self):**

**return self.\_\_Id**

**@Id.setter #(mutator)**

**def Id(self, Value):**

**self.\_\_Id = Value**

#--Methods--

**def ToString(self): # This overrides the original method (*it's polymorphic*)**

**"""Explictly returns field data"""**

**strData = super().ToString()**

**return str(self.Id) + ',' + strData**

**def \_\_str\_\_(self): # This overrides the original method as well**

**"""Implictly returns field data"""**

**strData = super().\_\_str\_\_()**

**return str(self.Id) + ',' + strData**

#--End of Class --

# --- Use the class ----

# by making an object!

**objE1 = Employee()**

**objE1.Id = 1**

**objE1.FirstName = "Earl"**

**print(objE1.ToString())**

**print(objE1)**

Next, you will try creating your own child class based on the Person class.



## LAB 8-5:

Create a Customer Class that has a Customer Id, FirstName, and LastName properties by inheriting code from your Person class.

