O'REILLY

8TH ANNUAL OSCON™ OPEN SOURCE CONVENTION

Object-Oriented Scripting

Avi Kak
kak@purdue.edu

JULY 24-28, 2006

Contents:

Part A: Object-Oriented Notions in General (slides 3 --- 10)

Part B: Object-Oriented Scripting in Perl (slides 11 --- 71)

Part C: Object-Oriented Scripting in Python (slides 72 –- 143)

Part D: Some Applications of OO Scripting (slides 144 --- 155)

Part A of this Tutorial

Object Oriented Notions in General

Slides 3 – 10

The Main OO Concepts

The following fundamental notions of object-oriented programming in general apply to object-oriented scripting also:

- Class
- Encapsulation
- Inheritance
- Polymorphism

What is a Class?

- At a high level of conceptualization, a class can be thought of as a category. We may think of “Cat” as a class.
- A specific cat would then be an **instance** of this class.
- For the purpose of writing code, a class is a data structure with **attributes**.
- An instance constructed from a class will have specific values for the attributes.
- To endow instances with behaviors, a class can be provided with **methods**.

Methods, Instance Variables, and Class Variables

- A method is a function you invoke on an instance of the class or the class itself.
- A method that is invoked on an instance is sometimes called an **instance method.**
- You can also invoke a method directly on a class, in which case it is called a **class method** or a **static method**.
- Attributes that take data values on a per-instance basis are frequently referred to as **instance variables**.
- Attributes that take on values on a per-class basis are called **class attributes** or **static attributes** or **class variables**.

Encapsulation

- Hiding or controlling access to the implementation-related attributes and the methods of a class is called **encapsulation.**
- With appropriate data encapsulation, a class will present a well-defined **public interface** for its **clients**, the users of the class.
- A client should only access those data attributes and invoke those methods that are in the **public interface**.

Inheritance and Polymorphism

- **Inheritance** in object-oriented code allows a subclass to inherit some or all of the attributes and methods of its superclass(es).

- **Polymorphism** basically means that a given category of objects can exhibit multiple identities at the same time, in the sense that a **`Cat`** instance is not only of type Cat, but also of type **`FourLegged`** and **`Animal`**, *all at the same time*.

Polymorphism (contd.)

- As an example of polymorphism, suppose we make an array like

```
@animals = (kitty, fido, tabby, quacker, spot);
```

of cats, dots, and a duck --- *instances made from different classes in some* Animal *hierarchy* --- and if we were to invoke a method **`calculateIQ()`** on this list of animals in the following fashion

```
foreach my $item (@animals) {
    $item.calculateIQ();
}
```

polymorphism would cause the correct implementation code for **`calculateIQ()`** to be automatically invoked for each of the animals.

Regarding the Previous Example on Polymorphism

- In many object-oriented languages, a method such as **`calculateIQ()`** would need to be declared for the root class `Animal` for the control loop shown on the previous slide to work properly.

- All of the public methods and attributes defined for the root class would constitute the public interface of the class hierarchy and each class in the hierarchy would be free to provide its own implementation for the methods declared in the root class.

- Polymorphism in a nutshell allows us to manipulate instances belonging to the different classes of a hierarchy through a common interface defined for the root class.

Part B of this Tutorial

Object Oriented Scripting in Perl

Slides 11 – 71

Defining a Class in Perl

We need to pull together the following three concepts in order to understand the notion of a class in Perl:

- Packages
- References
- The notion of **blessing an object into a package**

Using a Perl Package as a Class

- A class in Perl is a package that encloses the methods that characterize the behavior of the class and of the instances constructed from the class.

- Since, strictly speaking, a package is merely a namespace, it does not allow us to directly incorporate the attributes that can be used on a per-instance basis in the same sense that a C++ or a Java class does. (Although, as we will see later, a package does allow us to use attributes on a per-class basis, meaning like the static data members in C++ and Java.)

- This limitation, however, does not prevent us from creating a Perl instance with state.

Creating Stateful Objects in Perl

- Perl creates stateful objects through the expedient of packaging the state variables (meaning the per-instance variables) inside a standard data structure like a hash and having an object constructor return such a hash as an instance of the class.

- It is important to realize that just as much as Perl can use a hash for a class instance, it can also use a scalar or an array for the same purpose.

- The convenience of a hash is obvious --- it gives us named placeholders for the instance variables of a class, just like the data members in a C++ or a Java class.

Hashes, Arrays, and Scalars as Instance Objects! How Can That be?

- Hashes, arrays, and scalars are free-standing objects in Perl, meaning that they don't ordinarily belong to any particular package.
- It is true that, in general, a variable inside a package would need to be accessed with its package qualified name. But if the variable is holding a reference to, say, a hash, that hash itself has no particular package association.
- So how does Perl establish the needed association between a data object that is to serve as an instance of a class and the class itself (meaning the package that will be used as a class)?

So How Does Perl Acquire the Notion that a Data Object is of a Certain Type?

- The type labeling is needed if the behavior of an object must correspond to what is specified for that class through its methods.

- In Perl, the type association between a data object and a package that is to serve as a class is established through the mechanism of **blessing**.

- When an object is **blessed** into a package, the object becomes tagged as belonging to the package.

- Subsequently, the object can be considered to be of the type that is the name of the package.

Blessing an Object into a Class

- Note the type of reference held by **`$ref`** before and after it is blessed:

```
my $ref = {name=>“Trillian”, age=>35};
print ref( $ref );              # Hash
bless $ref, Person;
print ref( $ref );              # Person
```

- Also note that the above call to **`bless`** will also create a class named **`Person`** by the autovivification feature of Perl
- It is the object to which **`$ref`** is pointing that is getting blessed and not the variable **`$ref`** itself

What do we get by Blessing an Object into a Class?

- After an object has been blessed into a class, a method invoked on a reference to the object using the arrow operator will try to call on the subroutine of that name from the package corresponding to the class:

```
package Person;

sub get_name {
    my $self = shift;
    $self->{name}
}

package main;

my $ref = {name=>“Trillian”, age=>35};
bless $ref, Person;
print $ref->get_name();      # Trillian
```

Special Syntax for Methods

- From the call syntax on the previous slide, it looks like we are calling **`get_name()`** without any arguments.

- But because, we are invoking **`get_name()`** on a blessed reference, **`get_name()`** is implicitly supplied with one argument, which is the reference on which the subroutine is invoked.

- So it would be correct to say that the invocation of **`get_name()`** on the previous slide is translated by Perl into the following function call

```
Person::get_name( $ref )
```

which agrees with how the subroutine expects itself to be called.

Can Any Reference to Any Sort of an Object by Blessed ?

- Any reference whatsoever can be blessed. Here we are blessing an array:

```
package StringJoiner;

sub wordjoiner {
    my $self = shift;
    join “”, @$self;
}

package main;
my $ref = [‘hello’, ‘jello’, ‘mello’, ‘yello’];
bless $ref, StringJoiner;
print $ref->wordjoiner();          # hellojellomelloyello
```

Providing a Class with a Constructor

- A constructor's job is to create instance objects from the class definition

- A constructor must do the following:

  --- select a storage mechanism for the instance variables

  --- obtain a reference to the data object created (which will serve as a class instance) for the values provided for the instance variables

  --- bless the data object into the class and return the blessed reference to the object

How Should a Constructor be Named?

- While there are constraints on how a constructor is named in Python and in mainstream OO languages like C++ and Java, there are no such constraints in Perl.

- In Perl, if a class has a single constructor, it will typically be named **new**. But a class is allowed to have any number of constructors, a feature that Perl has in common with C++ and Java.

Example of a Class with a Constructor

```
package Person;

sub new {
  my ($class, $name, $age) = @_;
  bless {
    _name => $name;
    _age  => $age;
  }, $class;
}
```

- The constructor shown above is typically invoked with the following arrow-operator based syntax:

```
my $person = Person->new( “Zaphod”, 114 );
```

- Subsequently, the variable **`$person`** will hold a reference to an instance of class **`Person`**.

Constructor Example (contd.)

- In the constructor call on the previous slide, we used the arrow operator to invoke a method on the class itself, as opposed to on a reference to an instance object. Perl translates this call into:

```
  my $person = Person::new( Person, “Zaphod”, 114 );
```

- If we so wanted, we could use this syntax directly for constructing a **`Person`** instance.

- But that is not a recommended style for Perl OO because of its ramifications in constructing instances of the subclasses of a class.

Data Hiding and Data Access Issues

- In the **`Person`** class on slide 23, the names we used for the attributes started with an underscore.

- This is just a **convention** in Perl (and also Python) for denoting those names that are internal to a class.

- Ideally, it should be nobody's business how the various attributes are represented inside a class, not even how they are named.

- However, unlike in C++ and Java, there is no way to enforce such privacy aspects of how data is stored in a class in Perl (and in Python). Instead, we resort to conventions.

Convention Regarding Data Hiding

- Convention regarding data hiding states that the names used for the instance and the class variables begin with an underscore and that these attributes of a class be only accessible through the methods designated specifically for that purpose.

```
package Person;
sub new {
  my ($class, $name, $age) = @_;
  bless {
    _name = $name; _age = $age;
  }, $class;
}
sub name { $_[0]->{_name} }
sub age {
  my ($self, $age) = @_;
  $age ? $self->{_age} = $age : $self->{_age};
}
```

Packaging a Class into a Module

- A package in Perl is merely a namespace and it is indeed possible to have multiple packages in the same script file.

- But when it comes to using a package as a class in the object-oriented sense, a common practice is to have a single package --- and therefore a single class --- in a file, thus creating a module file.

An Example of a Module File for a Class

```
package Person;
###  filename: Person.pm
use strict;

sub new {
  my ($class, $name, $age) = @_;
  bless {
    _name = $name; _age = $age;
  }, $class;
}
sub get_name { $_[0]->{_name} }
Sub get_age { $_[0]->{_age} }
sub set_age {
  my ($self, $age) = @_;
  $self->{_age} = $age;
}
1
```

Notes:

1. The module file for the class name ends in the suffix '.pm'
2. The last statement in the file is just the number '1'. It can also be 'return 1'.
3. Usually a class file name will contain documentation before the 'package' statement.

Importing a Class File into a Script

Here is an example of how to use the class file shown on the previous slide:

```
#!/usr/bin/perl -w
### filename: TestPerson.pl
use strict;
use Person;

my ($person, $name, $age);
$person = Person->new(“Zaphod”, 114 );
$name = $person->get_name;
$age  = $person->get_age;
print “name: $name  age: $age\n”;   # name: Zaphod  age: 114

$person->set_age( 214 );
$age = $person->get_age;
print “name: $name  age: $age\n”;   # name: Zaphod  age: 214
```

Constructors with Named Parameters

- When a function takes a large number of arguments, it can be difficult to remember the position of each argument in the argument list of a function call.

- Perl scripts can take advantage of the built-in hash data structure so that functions can be called with named arguments.

- In addition to the convenience provided by attaching a name with an argument, the name—argument pairs can be specified in any positional order.

- The same can be done for a constructor, as shown next.

Example of a Constructor with Named Args

```
package Employee
### filename: Employee.pm
use strict;
sub new {
  my ($class, %args) = @_;
  bless {
    _name           =>     $args{name},
    _age            =>     $args{age},
    _gender         =>     $args{gender},
    _title          =>     $args{title},
  }, $class;
}
sub get_name { $_[0]->{_name} }
sub get_age  { $_[0]->{_age} }
sub get_gender { $_[0]->{_gender} }
sub get_title { $_[0]->{_title} }
sub set_age{ $_[0]->{_age} = $_[1] }
sub set_ title{ $_[0]->{_title} = $_[1] }
1
```

Calling a Constructor with Named Args

- The constructor of the class shown on the previous slide can be called with the following more convenient syntax:

```
my $emp = Employee->new( name    => “Poly”,
                         title   => “boss”,
                         gender  => “female”,
                         age     => 28, );
```

Default Values for Instance Variables

- If desired, it is possible to provide a constructor with default values for one or more of the instance variables named in the body of the constructor.

- When the constructor is meant to be called with the arguments in a specific positional order, the default values can only be specified for what would otherwise be the trailing arguments in a normal constructor call.

- With a constructor that expects to be called with named arguments, any arguments left unspecified can be taken care of by its default value, as shown on the next slide.

Ex. of a Constructor with Defaults for Args

```
package Flower;
### filename: Flower.pm
use strict; use Carp;
sub new {
  my ($class, $name, $season, $frag) = @_;
  bless {
    _name       =>    $name      || croak(“name required”),
    _season     =>    $season    || _ask_for_season($name),
    _fragrance  =>    $frag      || ‘unknown’,
  }, $class;
}
sub get_name { $_[0]->{_name} }
sub get_season  { $_[0]->{_season} }
sub get_fragrance { $_[0]->{_fragrance} }
sub set_season{ $_[0]->{_season} = $_[1] }
sub set_ fragrance{ $_[0]->{_fragrance} = $_[1] }
sub _ask_for_season {
  my $flower = shift;
  print STDOUT “enter the season for $flower: “;
  chomp( my $response = <> );
  $response;
}
1
```

Hiding Free-Standing Functions

- The **`Flower`** class shown on the previous slide suffers from one limitation: it does not make it difficult for a programmer to try to use the subroutine **`_ask_for_season()`** directly even though that subroutine is meant for just internal use by the class.

- A client of the **`Flower`** class could make the following invocation

```
my $flower = Flower->new( “rose” );
$flower->_ask_for_season();
```

- While the result of this external invocation of **`_ask_for_season()`** would produce a meaningless result in this case (since the instance object as opposed to the name of the flower will be passed as the argument to the subroutine), in general anything could happen, including the injection of a difficult to locate bug in a large script.

Hiding Free-Standing Functions (contd.)

- Unfettered access to private subroutines can be controlled by making such subroutines anonymous and having private variables hold references to them:

```
package Flower;
###  filename: Flower.pm
use strict;
my $ask = sub {
  my $fl = shift;
  print “enter season for $fl:”;
  chomp( my $response = <> );
  $response;
}
sub new {
  my ($cls,$nam,$seas,$frag)=@_;
```

```
bless {
   _name=>$nam || croak(“nam req”),
   _seasn=>$seas || $ask( $nam ),
   _fragrance=>$frag || ‘unknown’,
  }, $cls;
}
sub get_name { $_[0]->{_name} }
Sub get_season { $_[0]->{_season} }
sub get_fragrance { ……… }
sub set_season {………}
sub set_fragrance { ………… }
1
```

Object Destruction

- Perl comes with an automatic garbage collector for reclaiming memory occupied by unreferenced objects.

- The garbage collector is invoked by the system automatically through the mechanism of reference counting that is associated with the objects.

- In the same spirit as a destructor in C++ and the finalize method in Java, Perl allows a programmer to define a special method named DESTROY() that is called automatically for cleanup just before the system reclaims the memory occupied by the object.

Object Destruction (contd.)

- A programmer-provided **`DESTROY()`** can be important in situations where an object contains open filehandles, sockets, pipes, database connections, and other system resources. The code to free up these resources can be placed in **`DESTROY()`**.

- In addition to being invoked when the reference count for an object goes down to zero, **`DESTROY()`** would also be called if the process or the thread in which the Perl interpreter is running is shutting down.

Class Variables and Methods

- Except for the constructors and a few other functions embedded in definitions, the subroutines shown in the previous class definitions have mostly been those that are meant to be invoked on a per instance basis. A constructor is obviously intended to be invoked directly on a class.

- Methods that are meant to be invoked directly on a class are commonly referred to as either *class methods* or as *static methods.*

- Just like class methods, we can also have *class variables* or *class attributes* or *static attributes*. These variables are global with respect to the class, meaning global with respect to all the instances made from that class.

Class Variables and Methods (contd.)

- Shown next is a **`Robot`** class that allows us to assign a unique serial number to each **`Robot`** instance.
- The class has been provided with a class-based storage (as opposed to instance-based storage) for keeping track of the serial numbers already assigned so that the next **`Robot`** instance would get the next serial number.
- The class has also been provided with a class method for returning the total number of robots already made.

Class Variables and Methods (contd.)

```
#!/usr/bin/perl -w
use strict;
### filename: ClassData.pl

package Robot;

my $_robot_serial_num = 0;
my $_next_serial =
    sub {++$_robot_serial_num };
my $_total_num =
    sub { $_robot_serial_num };

# Constructor:
sub new {
 my ($class, $owner) = @_;
 bless {
  _owner => $owner,
  _serial_num=>$_next_serial->();
 }, $class;
}
```

```
# for both set and get (instance)
sub owner {
  my $self = shift;
  @_ ? $self->{_owner} = shift
     : $self->{_owner};
}

# an instance method
sub get_serial_num {
  my $self = shift;
  $self->{_serial_num};
}

# a class method
sub how_many_robots {
  my $class = shift;
  die “illegal call to static”
     unless $class eq ‘Robot’;
  $_total_num->();
}
```

Class Variables and Methods (contd.)

- Shown below is the test code for the **Robot** class that demonstrates the workings of class variables and methods.

```
my $bot = Robot->new( “Zaphod” );
my $name = $bot->owner();
my $num = $bot->get_serial_num();
print “owner: $name  serial number: $num”;
              # owner: Zaphod  serial number: 1

$bot = Robot->new( “Trillian” );
$name = $bot->owner();
$num = $bot->get_serial_num();
print “owner: $name  serial number: $num”;
              # owner: Trillian  serial number: 2

# invoke class method:
my $total_production = Robot->how_many_robots();
print $total_production;                   # 2

# my $x = Robot::how_many_robots();  # ERROR
```

Inheritance and Polymorphism in Mainstream OO

- Inheritance in mainstream OO languages such as C++ and Java means that a derived class inherits the attributes and the methods of all its parent classes.

- What the word "*inherits*" means in the above sentence is tighter than what would be suggested by, say, a child class just acquiring the attributes and the methods of its parent class.

- In the mainstream OO languages, the memory allocated to an instance of a child class contains slots for all the instance variables in all the parent classes.

Inheritance and Polymorphism in Mainstream OO (contd.)

- Therefore, in C++ and Java, an instance of a child class has built into it “sub-instances” of the parent classes.

- It is for this reason that in C++ and Java the constructor of a child class must explicitly or implicitly call the constructor of its parent classes before it does anything else.

- The calls to the constructors of the parent classes are needed for the initialization of that portion of the memory of a child class instance that is meant to hold the parent-class slices.

Inheritance and Polymorphism in Perl

- Inheritance in Perl (and in Python also) works very differently compared to how it works in mainstream OO languages such as C++ and Java.

- Perhaps the biggest difference is caused by the fact that when memory is allocated for a subclass instance in Perl (and in Python also), it does **not** contain slots for the base class attributes.

- In other words, a subclass instance in Perl is a completely separate data object and it does **not** contain “slices” for the data objects that could be formed from the parent classes.

Inheritance and Polymorphism in Perl (contd.)

- Inheritance in Perl (and also in Python) means only that if a method invoked on a subclass instance is not found within the subclass definition itself, it will be *search for* in the parent classes.

- This manner of search for a method in the parent classes automatically allows class instances to behave polymorphically, *albeit with a subtle twist vis-à-vis how polymorphism works in the mainstream OO languages.* This point is explained in further detail on the next slide.

Inheritance and Polymorphism in Perl (contd.)

- The polymorphism that results from Perl's (and also Python's) search-based approach to inheritance differs in a subtle way from how polymorphism works in mainstream OO languages.

- Whereas a child class in mainstream OO is *equally polymorphic* – if one could use that characterization – with respect to all its parent classes, the polymorphic behavior in Perl is weighted toward the parent classes that appear earlier in the recursive depth-first left-to-right search order through the parent classes of a child class.

- Perl (and Python) shares with the mainstream OO languages many of the other benefits derived from inheritance. This includes incremental development of code, using abstract classes for defining interfaces that the rest of the code can be programmed to, etc.

The ISA Array for Specifying the Parents of a Class

- The `ISA` array is fundamental to how inheritance works in Perl.
- A derived class is provided with a list of its parent classes through the `ISA` array.
- The script shown on the next slide presents the following class hierarchy

$$\begin{tikzcd}
\mathbf{X} & & \mathbf{Y} \\
& \mathbf{Z} \arrow[lu] \arrow[ru] &
\end{tikzcd}$$

where the child class **Z** is derived from the parent classes **X** and **Y**.

An Example of a Class Hierarchy

```
#!/usr/bin/perl –w
use strict;
###  InheritanceBasic.pl

package X;
  sub new { bless {}, $_[0] }
  sub foo { print “X’s foo\n” }

package Y;
  sub new { bless {}, $_[0] }
  sub bar { print “Y’s bar\n” }

package Z;
  @Z::ISA = qw( X Y );
  sub new { bless {}, $_[0] }
```

```
package main;

print join ‘ ‘, keys %Z::; #ISA new

my $zobj = Z=>new();
$zobj->foo();     # X’s foo
print join ‘ ‘, keys %Z::;
                  # ISA new foo
$zobj->bar();     # Y’s bar
print join ‘ ‘, keys %Z::;
                  # bar ISA foo new
print join ‘ ‘, values %Z::;
  # *Z::bar *Z::ISA *Z::foo *Z::new
```

How a Method is Searched For in a Class Hierarchy

- In the example on the previous slide, if a method invoked on a `Z` instance does not exist in class `Z` itself, the method is searched for in the parent classes in the `ISA` array defined for class **`Z`**. *In other words, such a method call would be dispatched up the inheritance tree.*

- In general, when not found in the class itself, the search for a method definition in the inheritance tree takes place *through a recursive left-to-right depth-first fashion.*

Left-to-Right Depth-First Search for a Method Definition

- Let's say that the inheritance tree that converges on a class **Z** looks like

$$\begin{tikzcd}
A & & B & \\
& P \arrow[ul] \arrow[ur] & & Q \\
& & Z \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

- When a method called on a **Z** instance is not found in **Z**'s definition, it is searched for in the namespace of class **P**. If not found there, the search goes to class **A**, and then to class **B**, and finally to class **Q**.

- This search takes place only once. Subsequently, a reference to that method is cached in the namespace of the class itself.

An Exercise in Class Derivation

- Write a script that implements the three level hierarchy:

$$\begin{tikzcd}
\texttt{Person} & & \\
\texttt{Employee} \arrow[u] & & \texttt{ExecutivePerks} \\
& \texttt{Manager} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

- For this simple exercise, it is more convenient to place all of your code in a single file.

An Exercise in Class Derivation (contd.)

- Use the hash as the storage mechanism for the instance objects.

- Place instance variables **`_name`** and **`_age`** in the **`Person`** class. Accessor methods for these variables, **`get_name()`**, **`get_age()`**, **`set_age()`**, will be inherited by all classes derived from **`Person`**.

- Provide the derived class **`Employee`** with one additional instance variable, **`_position`**, and the associated **`get_position()`** and **`set_position()`** methods.

- Also provide the **`Employee`** class with a hash **`%_promotion_table`** whose key-value pairs show the current **`_position`** and the next **`_position`** when an **`Employee`** is promoted. This class should also come with a method called **`promote()`** that changes the value of the **`_position`** of an **`Employee`**. This method should be inheritable by all child classes of **`Employee`**.

An Exercise in Class Derivation (contd.)

- Define another base class **`ExecutivePerks`** with a class variable **`_bonus`** and the **`get_bonus()`** and **`set_bonus()`** methods.
- Provide the **`Manager`** class with an instance variable called **`_department`**, which stands for the department that a manager is in charge of, and the associated get and set methods.
- For tutorial participants needing extra help, a starter file for this exercise will be provided. The starter file includes the code for **`Person`** and **`Employee`**, and a partial implementation of **`Manager`**.

Demonstration of Polymorphism for the **Person** Class Hierarchy

- By placing **`Person`** in the **`ISA`** array of **`Employee`** and **`Employee`** in the **`ISA`** array of **`Manager`**, an **`Employee`** instance inherits the methods of **`Person`** and a **`Manager`** instance inherits the methods of both **`Person`** and **`Employee`**. This allows an **`Employee`** to act like a **`Person`** and a **`Manager`** to act like an **`Employee`** and a **`Person`**. *That is polymorphism in its most basic form.*

- For strongly typed OO languages like C++ and Java, a particular consequence of polymorphism is the following property: *A derived class type can be substituted wherever a base class type is expected.*

Demonstration of Polymorphism in Perl OO (contd.)

- So in C++ and Java, if you write a function that needs a base class argument, by virtue of polymorphism you are allowed to call this function with a derived class object for the argument. In other words, suppose you write a function in Java

```
void foo( Person p );
```

we are allowed to invoke this function in the following manner

```
Manager man = new Manager( …… );
foo( man );
```

Demo of Polymorphism in Perl OO (contd.)

- This manifestation of polymorphism is also exhibited by Perl OO (and Python OO also).

- Suppose you write the following stand-alone function in Perl:

```
sub foo {            # expects a Person arg
  my ($arg) = @_;
  my $nam = $arg->get_name();
  print $nam;
}
```

- If you call this function with a **`Manager`** argument, it will do the right thing even though **`get_name()`** is only defined for the parent class.

A Derived-Class Method Calling a Base-Class Method

- A derived-class method can call a base-class method of the same name for doing part of the work by using the keyword **SUPER**

- To illustrate, consider the **Employee–Manager** portion of the class hierarchy shown previously. We could provide **Employee** with the following **print()** method:

```
sub print {
  my $self = shift;
  print "$self->{_name}  $self->{_position}  ";
}
```

- The **print()** for the derived class **Manager** could now be written as:

```
sub print {
  my $self = shift;
  $self->SUPER::print();
  print "$self->{_department}";
}
```

The Keyword **SUPER**

- The keyword **`SUPER`** causes Perl to search for **`print()`** in the direct and indirect superclasses of **`Manager`.**

- This search proceeds in exactly the same manner as for any regular method invocation, except that it starts with the direct superclasses.

- It is also possible to ask Perl to confine its search to a particular superclass (and all the superclasses of that class). For example, if we want **`Manager`**’s **`print()`** to specifically use **`Employee`**’s **`print()`**, then we replace the **`SUPER`** statement by:

```
        $self->Employee::print()
```

The **UNIVERSAL** Class

- Every class in Perl inherits implicitly from a base class called **`UNIVERSAL`**.

- We can therefore say that **`UNIVERSAL`** is implicitly at the root of every class hierarchy in Perl.

- **`UNIVERSAL`** plays the same role in Perl that **`object`** plays in Python and **`Object`** in Java.

- Every Perl class inherits the following methods from **`UNIVERSAL`**:

```
--- isa( class_name )
--- can( method_name )
--- VERSION( need_version )
```

The **UNIVERSAL** Class (contd.)

- Of the three methods, **`isa()`**, **`can()`**, and **`VERSION()`**, that every class inherits from **`UNIVERSAL`**, only the first two can be invoked by a programmer. The third, **`VERSION()`**, is invoked automatically by the system if the programmer requests a particular version of a Perl class.

- The **`isa()`** method that every class inherits from **`UNIVERSAL`** can be used to test whether a given object is an instance of a particular class.

- For example, for the **`Employee—Manager`** class hierarchy:

```
$man = Manager->new(“Trillian”, “manager”, “sales”);
print $man->isa(‘UNIVERSAL’) ? “yes” : “no”;  # yes
print $man->isa( ‘Manager’ ) ? “yes” : “no”;  # yes
print $man->isa(‘Executive’) ? “yes” : “no”;  # no
print $man->isa(‘Employee’)  ? “yes” : “no”;  # yes
```

The **`UNIVERSAL`** Class (contd.)

- The other programmer-usable class that every class inherits from **`UNIVERSAL`** is **`can().`** This method can be used to test whether a given class supports a particular method, either directly or through inheritance.

- A call to **`can()`** returns a reference to the supported method.

- On the next slide, the variable **`$which_func`** holds a reference to the first available function from the alternatives listed on the right hand side of the assignment.

The can() Method of UNIVERSAL

```
#!/usr/bin/perl –w
use strict;
###  CanMethod.pl

package X;
  sub new { bless {}, $_[0] }
  sub foo { print “X’s foo\n” }

package Y;
  sub new { bless {}, $_[0] }
  sub bar { print “Y’s bar\n” }

package Z;
  @Z::ISA = qw( X Y );
  sub new { bless {}, $_[0] }
  sub baz { print “Z’s baz\n” }
```

```
package main;

my $obj = Z->new();

print $obj->can(“foo”) ? “yes”:“no”;
                                # yes
print Z->can(“foo”) ? “yes” : “no”;
                                # yes
my $which_func = $obj->can(“boo”) ||
                 $obj->can(“bar”) ||
                 $obj->can(“baz”);
&$which_func;          # Y’bar
```

The **VERSION()** Method of **UNIVERSAL**

- This method is invoked directly by the system when a programmer requests that a specified version of a certain class file be loaded in.
- You can associate a version number with a class file by using the special variable **$VERSION** as follows

```
    $class_name::VERSION = 1.2;
```

- Subsequently, when the same class file is loaded in through the **use** directive, the load request can be customized to a special version of the class by

```
    use  class_name 1.2;
```

The **VERSION()** Method (contd.)

- The **use** directive shown at the bottom of the previous slide causes the invocation of the **VERSION()** method inherited from **UNIVERSAL** through the following command:

```
    class_name->VERSION( 1.2 )
```

  which makes sure that the version number is met by the class file.

Summary of How a Method is Searched for in a Class Hierarchy

- Earlier we talked about how a method is searched for in a class hierarchy when the method is invoked on a derived-class object. There our discussion focused on the direct and indirect superclasses of a derived-class that are searched for a given method.

- Now we will generalize that discussion to include the **`UNIVERSAL`** class in the search process.

- We will also address the issue of when exactly a method call is shipped off to **`AUTOLOAD()`** of the derived class and the **`AUTOLOAD()`** of the various direct and indirect superclasses of a derived class.

Method Resolution Order Summary (contd)

- Let's say we have

```
package Z;
@Z::ISA  =  qw( X Y );
```

- Now a method call like on a **`Z`** instance **`zobj`**:

```
zobj->foo();
```

will cause a search to be conducted for the method **`foo()`** in the order shown on the next slide.

MRO Summary (contd.)

1. Perl will first look for **`foo()`** in the class **`Z`**. If not found there, continue to the next step.

2. Perl will look for **`foo()`** in the parent class **`X`**. If not found there, proceed to the next step.

3. Perl will look for **`foo()`** – and do so recursively in a depth-first manner – in the direct and indirect superclasses of **`X`**. If not found in any of those superclasses, proceed to the next step.

4. Perl will look for **`foo()`** in the next base class, **`Y`**, declared in the **`ISA`** array shown on the previous slide. If not found there, proceed to the next step.

MRO Summary (contd.)

5. Perl will look for **`foo()`** – and do so recursively in a depth-first manner – in the parent classes of **`Y`**. If not found there, proceed to the next step.

6. After searching through the programmer-specified superclass hierarchy in the manner specified above, Perl will search for the method in the root class **`UNIVERSAL`**. If not found there, proceed to the next step.

7. Search for an implementation of **`AUTOLOAD()`** in exactly the same manner as outlined in the previous six steps. Dispatch the call to **`foo()`** to the first **`AUTOLOAD()`** implementation found. If no **`AUTOLOAD()`** implementation is found, throw a run-time exception.

Abstract Classes and Methods

- Abstract classes and methods play a very important role in OO

- An abstract class can represent the root interface of a class hierarchy. The concrete classes in the hierarchy can then be the implementations of the interface, each implementation designed for a different (although related) purpose.

- As an example, we can have a **`Shape`** hierarchy. A **`Shape`** represents an abstract notion. The root class **`Shape`** may only *abstractly declare* methods like **`area()`** and **`circumference()`.** The concrete classes such as **`Circle`** and **`Rectangle`** would then provide specific implementations for these methods.

Abstract Classes and Methods (contd.)

- A Perl class can be made abstract by having its constructor throw an exception if it should get invoked by a client of the class. This would prevent a client from constructing an instance of what is supposed to be an abstract class. The same can be done for a method if it is supposed to be abstract. (This can be done in Python also.)

- When using the above approach, you have to watch out for the fact that should a client inadvertently try to create an instance of an abstract class, your code will throw a run time error.

- A measure of protection against such run-time issues related to the use of abstract classes can be addressed by providing the package file containing the abstract class with its own **`import()`** method that ensures that the concrete child classes have fulfilled their contract with regard to providing the promised implementation code for the methods declared in the abstract class.

Part C of this Tutorial

Object Oriented Scripting in Python

Slides 72 – 143

OO Terminology in Python

- Python literature refers to everything as an object since practically all entities in Python are objects in the sense of possessing retrievable data attributes and invocable methods using the dot operator that is commonly used in object-oriented programming for such purposes.

- A user-defined class (for that matter, any class) in Python comes with certain **pre-defined** attributes.

OO Terminology in Python (contd.)

- The **pre-defined attributes** of a class are not to be confused with the **programmer-supplied attributes** such as the **class and instance variables** and the **programmer-supplied methods**.

- By the same token, an instance constructed from a class is an object with certain **pre-defined attributes** that again are not be confused with the **programmer-supplied instance and class variables** associated with the instance and the **programmer-supplied methods** that can be invoked on the instance.

OO Terminology in Python (contd.)

- What are commonly referred to as **data attributes** of a class in Perl are frequently called **instance variables** and **class variables** in Python. In Python, the word **attribute** is used to describe any property, variable or method, that can be invoked with the dot operator on either the class or an instance constructed from a class.

- Obviously, the attributes available for a class include the programmer-supplied class and instance variables and methods. This usage of attribute makes it all encompassing, in the sense that it now includes the pre-defined data attributes and methods, the programmer-supplied class and instance variables, and, of course, the programmer-supplied methods.

OO Terminology in Python (contd.)

- Our usage of **method** remains the same as before; these are functions that can be called on an object using the object-oriented call syntax that for Python is of the form **`obj.method(),`** where **`obj`** may either be an instance of a class or the class itself.

- Therefore, the **pre-defined** functions that can be invoked on either the class itself or on a class instance using the object-oriented syntax are also methods.

- The **pre-defined attributes**, both variables and methods, employ a special naming convention: *the names begin and end with two underscores.*

OO Terminology in Python (contd.)

- You may think of the **pre-defined** attributes as the **external properties** of classes and instances and the **programmer-supplied** attributes (in the form of instance and class variables and methods) as the **internal properties**.

- Python makes a distinction between **function objects** and **callables**. While all function objects are callables, not all callables are function objects.

OO Terminology in Python (contd.)

- A **function object** can only be created with a **`def`** statement.
- On the other hand, a **callable** is any object that can be called like a function.
- For example, a class name can be called directly to yield an instance of a class. Therefore, a class name is a callable.
- An instance object can also be called directly; what that yields depends on whether or not the underlying class provides a definition for the **system-supplied `__call__()`** method.

Defining a Class in Python

- We will present the full definition of a Python class in stages.
- We will start with a very simple example of a class to make the reader familiar with the pre-defined `__init__()` method whose role is to initialize the instance returned by a call to the constructor.
- First, here is the simplest possible definition of a class in Python:

```
class SimpleClass:
    pass
```

An instance of this class may be constructed by invoking its pre-defined default constructor:

```
x = SimpleClass()
```

Defining a Class in Python (contd.)

- Here is a class with a user-supplied constructor initializer in the form of **`__init__().`** This method is automatically invoked to initialize the state of the instance returned by a call to **`Person():`**

```
#!/usr/bin/env python
#--------------  class Person --------------
class Person:
    def __init__(self, a_name, an_age ):
        self.name = a_name
        self.age  = an_age
#--------- end of class definition --------

#test code:
a_person = Person( “Zaphod”, 114 )
print a_person.name      # Zaphod
print a_person.age       # 114
```

Pre-Defined Attributes for a Class

- Being on object in its own right, every Python class comes equipped with the following pre-defined attributes:

`__name__`	string name of the class
`__doc__`	documentation string for the class
`__bases__`	tuple of parent classes of the class
`__dict__`	dictionary whose keys are the names of the class variables and the methods of the class and whose values are the corresponding bindings
`__module__`	module in which the class is defined

Pre-Defined Attributes for an Instance

- Since every class instance is also an object in its own right, it also comes equipped with certain pre-defined attributes. We will be particularly interested in the following two:

  `__class__` : string name of the class from which the instance was constructed

  `__dict__` : dictionary whose keys are the names of the instance variables

- It is important to realize that the namespace as represented by the dictionary **`__dict__`** for a class object is **not** the same as the namespace as represented by the dictionary **`__dict__`** for an instance object constructed from the class.

`__dict__` vs. `dir()`

- As an alternative to invoking **`__dict__`** on a class name, one can also use the built-in global **`dir()`**, as in

```
dir( MyClass )
```

  which returns a tuple of just the attribute names for the class (both directly defined for the class and inherited from a class's superclass).

Illustrating the Values for System-Supplied Attributes

- This extension of the previous script illustrates the values for the pre-defined attributes for class and instance objects:

```
#!/usr/bin/env python
#------  class Person --------
class Person:
    ‘A very simple class’
    def __init__(self,nam,yy ):
        self.name = nam
        self.age  = yy
#-- end of class definition --

#test code:
a_person = Person(“Zaphod”,114)
print a_person.name    # Zaphod
print a_person.age     # 114
```

```
print Person.__name__   #Person
print Person.__doc__
          # A very simple class
print Person.__module__   # main
print Person.__bases__    # ()
print Person.__dict__
 # {‘__module__’ : ‘__main__’,
 #  ‘__doc__’ : ‘A very simp..’,
 #  ‘__init__’:<function __init..’,
print a_person.__class__
             # __main__.Person
print a_person.__dict__
   #{‘age’:114, ‘name’:’Zaphod’}
```

Class Definition: More General Syntax

```
class MyClass :
  ‘optional documentation string’
  class_var1
  class_var2 = var2

  def __init__( self, var3 = default3 ):
    ‘optional documentation string’
    attribute3 = var3
    rest_of_construction_init_suite

  def some_method( self, some_parameters ):
    ‘optional documentation string’
    method_suite

  ………
  ………
```

Class Definition: More General Syntax (contd.)

- Regarding the syntax shown on the previous slide, note the class variables **`class_var1`** and **`class_var2`**. Such variables exist on a per-class basis, meaning that they are static.

- A class variable can be given a value in a class definition, as shown for **`class_var2`**.

- In general, the header of **`__init__()`** may look like:

```
def __init__(self, var1, var2, var3 = default3):
  body_of_init
```

This constructor initializer could be for a class with three instance variables, with the last default initialized as shown. The first parameter, typically named **`self`**, is set implicitly to the instance under construction.

Class Definition: More General Syntax (cond.)

- If you do not provide a class with its own **`__init__()`,** the system will provide the class with a default **`__init__()`.** You override the default definition by providing your own implementation for **`__init__()`.**

- The syntax for a user-defined method for a class is the same as for stand-alone Python functions, except for the special significance accorded the first parameter, typically named **`self`.** It is meant to be bound to a reference to the instance on which the method is invoked.

New-Style Versus Classic Classes in Python

- Python 2.2 introduced new-style classes while retaining the old-style classes for backward compatibility.

- The old style classes are referred to as the classic classes.

- The basic motivation for new style classes is to allow subclassing of built-in classes. It was not previously possible to extend, say, the string class **`str`** to create a more customized string class. But now you can do that with ease.

- All new style classes are subclassed, either directly or indirectly from the root class **`object`**.

New Style vs. Classic Classes (contd.)

- The **`object`** class defines a set of methods with default implementations that are inherited by all classes derived from **`object`**.

- A case in point is the **`__getattribute__()`** method that gets invoked whenever a class method is invoked. Its implementation in the **`object`** class is a do-nothing implementation.

- A class that inherits from **`object`** can provide an override implementation for **`__getattribute__()`** if something special needs to be done because a method was invoked.

What does a New Style Class get From the Root Class **object**

- The list of attributes defined for the **object** class can be seen by printing out the list returned by the built-in **dir()** function:

```
                    print dir( object )
```

  This call returns

```
['__class__','__delattr__','__doc__','__getattribute__'
  '__hash__','__init__','__new__',__reduce__',
  '__reduce_ex__','__repr__','__setattr__','__str__']
```

- We can also examine the attribute list available for the **object** class by printing out the contents of its **__dict__** attribute by

```
                    print object.__dict__
```

  This will print out both the attribute names and their bindings.

How Python Creates an Instance from a New Style Class

Python uses the following two-step procedure for constructing an instance from a new-style class:

STEP 1:

- The call to the constructor creates what may be referred to as a generic instance from the class definition.

- The generic instance's memory allocation is customized with the code in the method **`__new__()`** of the class. This method may either be defined directly for the class or the class may inherit it from one of its parent classes.

Creating an Instance for New-Style Classes (contd.)

- The method **`__new__()`** is implicitly considered by Python to be a static method. Its first parameter is meant to be set equal to the name of the class whose instance is desired and it must return the instance created.

- If a class does not provide its own definition for **`__new__()`,** a search is conducted for this method in the inheritance tree that converges on the class (more on that later).

STEP 2:

- Then the method **`__init__()`** of the class is invoked to initialize the instance returned by **`__new__()`.**

Example Showing `__new__()` and `__init__()` Working Together for Instance Creation

- The script shown on slide 95 defines a class **`X`** and provides it with a static method **`__new__()`** and an instance method **`__init__()`.**

- We do not need any special declaration for **`__new__()`** to be recognized as static because this method is special-cased by Python.

- Note the contents of the namespace dictionary **`__dict__`** created for class **`X`** as printed out by **`X.__dict__.`** This dictionary shows the names created specifically for class **`X`**. On the other hand, **`dir(X)`** also shows the names inherited by **`X`**.

Instance Construction Example (contd.)

- Also note that the namespace dictionary **`xobj.__dict__`** created at runtime for the instance **`xobj`** is empty --- for obvious reasons.

- As stated earlier, when **`dir()`** is called on a class, it returns a list of all the attributes that can be invoked on a class and on the instances made from that class. The returned list also includes the attributes inherited from the class's parents.

- When called on a instance, as in **`dir( xobj ),`** the returned list is the same as above plus any instance variables defined for the class.

Instance Construction Example (contd.)

```
#!/usr/bin/env python
#---------- class X -----------
class X (object):   # X derived
                    # from object
  def __new__( cls ):
    print "__new__ invoked"
    return object.__new__( cls )

  def __init__( self ):
    print "__init__ invoked"

#----------- Test Code ---------
xobj = X()    # __new__ invoked
              # __init__ invoked
print X.__dict__
  #{'__module__': '__main__',
  # '__new__': <static method ..>,
  # ………
```

```
print xobj.__dict__    # {}

print dir(X)
   #['__class__','__delattr__',
   # '__getattribute__',
   # '__hash__','__init__'
   # '__module__','__new__'
   # …………….]

print dir( xobj )
   #['__class__','__delattr__',
   # '__getattribute__',
   # '__hash__','__init__',
   # '__module__',__new__',
   # ……………]
```

How Python Creates an Instance from a Classic Class

- There does not exist a separate **`__new__()`** method for constructing an instance from a **classic** class.

- The call to the class itself results in the construction of an instance object that is subsequently (and automatically) initialized by the class's **`__init__()`** method if the class is provided with such a method.

- The script shown on the next slide defines a classic Python class **`X`**. It is classic because **`X`** is not subclassed from the root class **`object`**.

- The class is not provided with a **`__new__()`** method because it does not need one for instance construction.

Constructing an Instance of a Classic Class (contd.)

```
#!/usr/bin/env python
#------------------ class X --------------------
class X :
  def __init__( self ):
    print "__init__ invoked"

#----------------- Test Code --------------------
xobj = X()      # __init__ invoked
print X.__dict__    # {'__module__' : '__main__',
                   #  '__doc__' : None,
                   #  '__init__': <function ……> }
print xobj.__dict__      # {}

print dir( X )   # ['__doc__','__init__','__module__']

print dir(xobj)  # ['__doc__','__init__','__module__']
```

The Syntax for Defining a Method

- A method defined for a class must have special syntax that reserves the first parameter for the object on which the method is invoked. This parameter is typically named **`self`** for instance methods, but could be any legal Python identifier.

- In the script shown on the next slide, when we invoke the constructor using the syntax

  ```
          xobj = X( 10 )
  ```

  the parameter **`self`** in the call to **`__init__()`** is set implicitly to the instance under construction and the parameter **`nn`** to the value 10.

- A method may call any other method of a class, but such a call must always use class-qualified syntax, as shown by the definition of **`bar()`** on the next slide.

Defining a Method (contd.)

- One would think that a function like **baz()** in the script below could be called using the syntax **X.baz(),** but that does not work. (We will see later how to define a class method in Python).

```
#!/usr/bin/env python
#----------- class X ------------
class X:
  def __init__(self, nn):
    self.n = nn
  def getn(self):
    return self.n
  def foo(self,arg1,arg2,arg3=1000):
    self.n = arg1 + arg2 + arg3
  def bar( self ):
    self.foo( 7, 8, 9 )
  def baz():
    pass
#--- End of Class Definition ----
```

```
xobj = X(10)
print xobj.getn()     # 10

xobj.foo(20,30)
print xobj.getn()     # 1050

xobj.bar()
print xobj.getn()     # 24

# X.baz()             # ERROR
```

A Method Can be Defined Outside a Class

- It is not necessary for the body of a method to be enclosed by a class.

- A function object created outside a class can be assigned to a name inside the class. The name will acquire the function object as its binding. Subsequently, that name can be used in a method call as if the method had been defined inside the class.

- In the script shown on the next slide, the important thing to note is that is that the assignment to **`foo`** gives **X** an attribute that is a function object. As shown, this object can then serve as an instance method.

Method Defined Outside a Class (contd.)

```
#!/usr/bin/env python

def bar(self,arg1,arg2, arg3=1000):
  self.n = arg1 + arg2 + arg3

#---------- class X ------------
class X:
  foo = bar

  def __init__(self, nn):
    self.n = nn

  def getn(self):
    return self.n

#--- End of Class Definition ----
```

```
xobj = X(10)
print xobj.getn()      # 10

xobj.foo( 20, 30 )
print xobj.getn()      # 1050
```

Only One Method for a Given Name Rule

- When the Python compiler digests a method definition, it creates a function binding for the name of the method.

- For example, for the following code fragment

```
class X:
  def foo(self, arg1, arg2):
    implemention_of_foo

  rest_of_class_X
```

the compiler will introduce the name **`foo`** as a key in the namespace dictionary for class **X**. The value entered for this key will be the function object corresponding to the body of the method definition.

Only One Method Per Name Rule (contd.)

- So if you examine the attribute **`X.__dict__`** after the class is compiled, you will see the following sort of entry in the namespace dictionary for **`X`**:

```
    'foo' : <function foo at 0x805a5e4>
```

- Since all the method names are stored as keys in the namespace dictionary and *since the dictionary keys must be unique*, this implies that there can exist only one function object for a given method name.

- If after seeing the code snippet shown on the previous slide, the compiler saw another definition for a method named for the same class, then *regardless of the parameter structure of the function*, the new function object will replace the old for the value entry for the method name. (This is unlike what happens in C++ and Java where function overloading plays an important role.)

Method Names can be Usurped by Data Attribute Names

- We just talked about how there can only be one method of a given name in a class --- regardless of the number of arguments taken by the method definitions.

- As a more general case of the same property, a class can have only one attribute of a given name.

- What that means is that if a class definition contains a class variable of a given name after a method attribute of the same name has been defined, the binding stored for the name in the namespace dictionary will correspond to the definition that came later.

Bound and Unbound Methods

- To understand how you can endow a Python class with static methods, it is important to understand what is meant by bound and unbound methods.

- In general, when a method is invoked on an instance object or on the class itself, Python creates a method object and associates with it the following attributes: **`im_self`**, **`im_func`**, and **`im_class`**.

- When the method object is first initialized, the **`im_self`** attribute is set to **`None`**.

- Subsequently, if the first argument supplied to the method call is an instance object, the **`im_self`** attribute is set to a reference to that instance. **In this case, we say that method object is bound.**

Bound and Unbound Methods (contd.)

- Since, in general, a method can be called on any object, what if a method is called directly on the class itself? In this case, the **`im_self`** attribute is set to **`None`** and **the method object is said to be unbound.**

- In both cases, the **`im_class`** attribute would be set to the name of the class. Again in both cases, the **`im_func`** attribute would be set to the function object in question.

- As shown on the next slide, a method that would ordinarily be called as a bound method on an instance object may also be invoked as an unbound method directly on the class.

- As we will see later, calling a method as an unbound method is particularly useful when a subclass needs to call a particular base class method.

Bound and Unbound Methods (contd.)

```
#!/usr/bin/env python
#---------- class X ------------
class X:
  def foo(self, mm):
    print “mm = “, mm
#--- End of Class Definition ---

xobj = X()
print X.foo                # <unbound method X.foo>
print xobj.foo
    # <bound method X.foo of <__main__.X instance at 0x403b51cc>

#call foo() as a bound method:
xobj.foo(10)               # mm = 10

#call foo() as an unbound method:
X.foo( xobj, 10 )          # mm = 10
```

Using `__getattr__()` as a Catch-All for Non-Existent Methods

- The role played by **`AUTOLOAD()`** in Perl OO is played by the system-supplied **`__getattr__()`** method for a Python class.

- If a non-existent method is invoked on an instance object, Python farms out that call to the **`__getattr__()`** method provided a class possesses, either directly or through inheritance, a definition for this method.

- By a non-existent method call we mean a method call whose definition cannot be found either in the class or through a search in the inheritance tree that converges on the class.

Trapping Non-Existent Method Calls (contd.)

- To trap calls with **`__getattr__(),`** this method must be defined with two parameters. The system would set the first to the instance object on which a non-existent method is called and set the second to the name of the non-existent method.

- Additionally, **`__getattr__()`** must return a **callable object**. As mentioned previously, an object is considered callable if it can be called with a function-call syntax, that is with the '( )' operator, with or without arguments.

`__getattr__()` vs. `__getattribute__()`

- For new style classes, Python also makes available the **`__getattribute__()`** method that is called whenever a method is invoked.

- The **`__getattribute__()`** method is defined for the root class **`object`** with a do-nothing implementation. It can however be overridden in your own class to provide any set-up operations before the code in a method is actually executed.

- As for the difference between **`__getattr__()`** and **`__getattribute__()`**, the former is called only when a non-existent method is invoked on an instance object, the latter is called whenever a method is referenced.

Subjecting a Callable to a Function Call Operation

- Here is an interesting difference between Perl and Python: the difference between just accessing a callable defined for a class and subjecting the callable to a function call operation with the ‘()’ operator.

- This distinction also applies to any callable object, whether or not it is defined for a class.

- For a class **`X`** with method `foo`, calling just **`X.foo`** returns a result different from what is returned by **`X.foo().`** The former returns the method object itself that **`X.foo`** stands for and the latter will cause execution of the function object associated with the method call.

Destruction of Instance Objects

- Just like Perl, Python also comes with an automatic garbage collector. The basic principle on which the Python garbage collector works is the same as in Perl.

- Each object created is kept track of through reference counting. Each time an object is assigned to a variable, its reference count goes up by one, signifying the fact that there is one more variable holding a reference to the object.

- And each time a variable whose referent object either goes out of scope or is changed, the reference count associated with the object is decreased by one. When the reference count associated with an object goes to zero, it becomes a candidate for garbage collection.

- Python provides us with __**del**__ () that works the same way as **DESTROY()** in Perl.

Encapsulation Issues for Classes

- Encapsulation is one of the cornerstones of OO. How does it work in Python?

- The same as in Perl OO. All of the attributes defined for a class are available to all.

- So the language depends on programmer coopration if software requirements, such as those imposed by code maintenance and code extension considerations, dictate that the class and instance variables be accessed only through `get` and `set` methods.

- As with Perl, a Python class and a Python instance object are so open that they can be modified after the objects are brought into existence.

Defining Static Attributes for a Class

- A class definition usually includes two different kinds of attributes: those that exist on a per-instance basis and those that exist on a per-class basis. That latter, as we mentioned in the Perl portion of the tutorial, are also called **static**.

- In Python, a variable becomes static if it is declared outside of any method in a class definition.

- For a method to become static, it needs the **`staticmethod()`** wrapper.

- Shown on the next slide is a class with a class variable (meaning a static data attribute) **`next_serial_num`**

Static Attributes (contd.)

```
#!/usr/bin/env python
#---------- class Robot ------------
class Robot:
  next_serial_num = 1

  def _init_(self, an_owner):
    self.owner = an_owner
    self.idNum = self.get_next_idNum()

  def get_next_idNum( self ):
    new_idNum = Robot.next_serial_num
    Robot.next_serial_num += 1
    return new_idNum

  def get_owner(self):
    return self.owner
  def get_idNum(self):
    return self.idNum
#----- End of Class Definition -----
```

```
robot1 = Robot(“Zaphod”)
print robot1.get_idNum()  # 1

robot2 = Robot(“Trillian”)
print robot2.get_idNum()  # 2

robot3 = Robot(“Betelgeuse”)
print robot3.get_idNum()  # 3
```

Static Attributes (contd.)

- A static method is created by supplying a function object to **`staticmethod()`** as its argument. For example, to make a method called **`foo()`** static, we'd do the following

```
    def foo():
       print “foo called”

     foo = staticmethod( foo )
```

  The function object returned by **`staticmethod()`** is static.

- In the above example, when **`foo`** is subsequently called directly on the class using the function call operator ‘()’, it is the callable object bound to **`foo`** in the last statement above that gets executed.

- The same idea works for static methods with args and for static methods that need to call other static methods in the same class.

An Instance Variable Hides a Class Variable of the Same Name

- When the **`self.attribute`** notation is used to access a data attribute of the class, an instance variable of a given name will hide a class variable of the same name.

- When a class variable gets hidden in this manner, it can still be accessed with the **`class.attribute`** notation.

Private Attributes in a Class

- Python provides a mechanism for endowing a class with private data and method attributes.
- This is done through name mangling in such a way that a private attribute becomes "inaccessible" outside the class.
- Any data-member name or a method name that has **at least** two leading underscores and **at most** one trailing underscore is private to that class.
- However, it is important to bear in mind that the "privateness" achieved in this manner still depends on programmer cooperation. Since the result of name mangling is predictable, the names can still be reached outside the class through their mangled versions.

Private Attributes (contd.)

- A name that has at least two leading underscores and at most one trailing underscore is renamed by the compiler by attaching to the name an underscore followed by the class name.

- For example, an attribute name such as

  `__m`

  in a class called **`X`** will be mangled into

  `_X__m`

Defining a Class with Slots

- New style classes allow the **`__slots__`** attribute of a class to be used to name a list of instance variables. Subsequently, no additional variables can be assigned to the instances of such a class.

- In the example on the next slide, we try for **`__init__()`** to declare an instance variable **`c`**. This is in addition to the instance variables already defined through the **`__slots__`** attribute. However, this becomes a source of error at runtime.

- Accessing an instance variable that is declared as a slot but that has not yet been initialized will evoke the **`AttributeError`** from Python. Doing the same for a classic class returns **`None`**.

A Class with Slots (contd.)

```
#!/usr/bin/env python

class X( object ):
  __slots__ = [‘a’, ‘b’]

  def _init_(self, aa, bb, cc):
    self.a = aa
    self.b = bb
    self.c = cc       # Will cause error

#----- Test Code -----

xobj = X( 10, 20, 30 )
      # AttributeError: ‘X’ object has no attribute ‘c’
```

Descriptor Classes in Python

- A **descriptor class** is a new-style class with override definitions for at least one of the special system-supplied methods: **`__get__()`**, **`__set__()`**, and **`__delete__()`**.

- When an instance of such a class is used as a **static attribute** in another class, accesses to those values are processed by the **`__get__()`** and the **`__set__()`** methods of the descriptor class.

- The class whose static attributes are the instances of a descriptor class is known as the **owner class**.

Descriptor Classes (contd.)

- Slide 125 shows a simple descriptor class, a new style class obviously since it is derived from **`object`**, that stores a single data value.

- The print statements in the override definitions of **`__get__()`** and **`__set__()`** are merely to see these methods getting invoked automatically.

- The calls to the **`__get__()`** and **`__set__()`** methods of a descriptor class are orchestrated by the **`__getattribute__()`** method of the owner class. (It is therefore possible for **`__get__()`** and **`__set__()`** to not get called depending on the override definition for **`__getattribute__()`** of the owner class.)

Descriptor Classes (contd.)

- Next, the script on the next slide defines an **owner** class, **`UserClass`**, again a new style class, with three **static** attributes, **`d1`**, **`d2`**, and **`d3`**. The first two of these are set to instances of the **`DescriptorSimple`** class.

- Finally, the script constructs an instance of **`UserClass`**. When we try to retrieve the value of the **`d1`** attribute of the class, the following message is displayed on the screen:

```
 Retrieving with owner instance: <__main__.UserClass
object at …> and owner type: <class __main__.User Cl
ass’>
 100
```

Descriptor Classes (contd.)

```
#!/usr/bin/env python

#---- class DescriptorSimple -----
class DescriptorSimple( object ):
  def __init__(self, initVal=None):
    self.val = initVal

  def __get__(self, owner_inst, \
                  owner_type):
    print “Retrieving with owner \
       instance: “, owner_inst, \
     “ and owner type”, owner_type
    return self.val

  def __set__(self,owner_inst,val):
    print “Setting attribute for \
      owner instance: “, owner_inst
    self.val = val
```

```
# An owner class:
class UserClass( object ):
  d1 = DescriptorSimple(100)
  d2 = DescriptorSimple(200)
  d3 = 300
#---------- Test Code --------
u = UserClass()
print u.d1     # 100
print u.d2     # 200
print u.d3     # 300
print UserClass.d1   #100
u.d1  =  400  # does the
u.d2  =  500  # expected thing

UserClass.d1 = 600  #This does
      # NOT do what you think
      # it does.
```

Extending a Class in Python

- An inheritance chain in Python is constructed by including the name of the superclass in the header of the subclass, as in

```
    class SomeSubClass( SomeSuperClass ):
```

- A **derived-class** method **overrides** a base-class method of the same name.

- Sometimes it is useful for a derived-class method to get a part of its work done by a base-class method of the same name.

- When a derived-class method makes a call to a base-class method of the same name, we say that the derived class is extending the base-class method. Doing so is straightforward in a single-inheritance chain.

Extending a Class (contd.)

- Method extension for the case of single-inheritance is illustrated in the **`Employee-Manager`** class hierarchy on the next slide. Note how the derived-class **`promote()`** calls the base-class **`promote()`**, and how the derived-class **`myprint()`** calls the base-class **`myprint()`**.

- Extending methods in multiple inheritance hierarchies requires calling **`super()`**. To illustrate, suppose we wish for a method **`foo()`** in a derived class **`Z`** to call on **`foo()`** in **`Z`**'s superclasses to do part of the work:

```
class Z( A, B, C, D):
   def foo( self ):
          ….do something….
          super( Z, self).foo()
```

Extending a Class (contd.)

```
#!/usr/bin/env python
#------- base class Employee ---------
class Employee:
  def __init__(self, nam, pos):
    self.name = nam
    self.position = pos

  promotion_table = {
      ‘shop_floor’ : ‘staff’,
      ‘staff’ : ‘management’,
      ‘manager’ : ‘executuve’
  }

  def promote(self):
    self.position = \
Employee.promotion_table[self.position]

  def myprint(self):
    print self.name, “ “, self.position,
```

```
#----- derived class Manager ------
class Manager( Employee ):
  def __init__(self, nam, pos, dept):
    self.name = nam
    self.position = pos
    self.dept = dept

  def promote(self):
    if self.position == ‘executive’:
      print “not possible”
      return
    Employee.promote( self )

  def myprint(self):
    Employee.myprint(self)
    print self.dept

#---------- Test Code -----------
emp = Employee(“Orpheus”, “staff”)
emp.promote()
print emp.position     # management
```

New Style Classes Revisited

- As previously mentioned, Python now supports two different kinds of classes: **classic classes** and **new-style classes**
- A new style class is subclassed from the system-supplied **`object`** class or from a child of the **`object`** class
- All of the built-in classes are new style classes. That is, the built-in types such as **`list, tuple, dict`**, etc., are now new style classes. This allows them to be subclassed for defining more specialized utility classes.
- An instance of a new style class is created by the static method **`__new__().`** If a user-defined class is not provided with an implementation for this method, its inherited definition from a superclass is used.

New Style Classes Revisited (contd.)

- An instance returned by **`__new__()`** is automatically initialized by the class's **`__init__()`** method. If a class does not directly provide a definition for this method, its inherited definition from a superclass is used.

- For new style classes, an instance **`xobj`**'s class can be ascertained by calling **`xobj.__class__.`** For classic classes, the same is accomplished by calling **`type(xobj).`** By the way, **`type(xobj)`** also works for new style classes.

- To ascertain whether an instance **`xobj`** is of a certain specific type, we can use the function **`isinstance()`** as in

```
isinstance(xobj, className)
```

New Style Classes Revisited (contd.)

- Using multiple inheritance, a subclass derived from a mixture of classic and new-style classes is treated like a new style class.
- You cannot multiply inherit from the different built-in types. For example, you cannot construct a class that inherits from both the built-in **`dict`** and the built-in **`list`**.
- When a constructor of a built-in type is called without arguments, it results in an instance with an appropriate default state. For example, **`str()`** returns an empty string and **`int()`** returns 0.
- The built-in types **`staticmethod, super, classmethod`**, and **`property`** have special roles in Python OO. The calls to their constructors return function objects.

Extending the Built-In Types

- As mentioned already, all of the built-in classes are now new-style classes. That allows for them to be extended for creating more specialized classes.

- Let's say you want a variation on the built-in **`dict`** class that would allow us to construct a dictionary from two arguments, one a list of keys and another a separate list of the corresponding values. That can done easily by extending **`dict`** into, say, **`MyDict`**, and providing an appropriate override for the **`__init__()`** method.

- In general, extending a built-in type may involve overriding just the **`__new__()`**, or just the **`__init__()`**, or both. If the extension requires customization of memory allocation, you'd need to override **`__new__()`**.

Extending the Built-In Types (contd.)

- The next slide shows us subclassing the built-in **`int`** class. We call the new class **`size_limited_int`**.

- We want the **`size_limited_int`** constructor to raise an exception if an attempt is made to construct an integer instant whose integer value is outside the range permitted by the class.

- We also want to provide the class with an override definition for the '+' operator that would allow us to add two of **`size_limited_int`**'s, with the summation being of type **`size_limited_int.`**

- For the override for the '+' operator through the implementation for **`__add__(),`** note the call to **`super().`** This causes the parent class's **`__add__()`** to be invoked for the addition operation. (Call to **`super()`** returns a subclass type object if it is supposed to return anything at all.)

Extending the Built-In Types (contd.)

```
#!/usr/bin/env python

class size_limited_int( int ):

  maxSize = 100

  def __new__(cls, intVal, size = 100):
    cls.maxSize = size
    if intVal < -cls.maxSize or
                  intVal > cls.maxSize:
      raise MyException(“out of range”)
    return int.__new(cls, intVal )

  def __add__( self, arg ):
    res = super( size_limited_int,
                  self ).__add__( arg )
#   res = int.__and__( self, arg )
    return size_limited_int( res,
                         self.maxSize )
```

```
#----------   Test Code  ---------------

n1 = size_limited_int( 5 )

print n1           # 5

print isinstance(n1,size_limited_int)
                                 # True
print isinstance(n1, int)        # True
print isinstance(n1, object)     # True

try:
  n2 = size_limited_int( 1000 )
except Exception, error:
  print error          # out of range

n3 = size_limited_int( 10 )
n4 = n1 + n3
print n4             # 15
print isinstance( n4, size_limited_int)
                            # True
```

Abstract Classes and Methods in Python

- The comments made earlier in the Perl section with regard to the general importance of abstract classes and methods apply to Python also.

- The **`NotImplementedError`** exception is used in Python to designate abstract classes and methods.

- A Python class is made abstract by having its constructor raise the **`NotImplementedError`** exception.

- A method is made abstract by having its body do the same. When a method is made abstract in this manner, there is the expectation that the full implementation of the method will be supplied in a derived class.

Multiple Inheritance in Python OO

- Like Perl, Python allows a class to be derived from multiple base classes. The header of such a derived class would look like

```
class MyDerivedClass( Base1,Base2,Base3,……):
   …body of the derived class…
```

- Suppose we invoke a method on a derived-class instance and the method is not defined directly in the derived class, in what order will the base classes be searched for a definition for the method?

- The order in which the class and its bases are searched for an applicable definition is called the **Method Resolution Order** (MRO).

- MRO are different for classic classes and for new style classes

MRO for Classic Classes

- The search for a method is carried out in a left-to-right depth-first fashion.
- Since this is the same as for Perl, we will not go further into this MRO here, except for showing some of its shortcomings in the next few slides.

The LRDF based MRO

- We will refer to the left-to-right depth-first MRO for classic classes as the **LRDF** lookup algorithm.

- Although straightforward, LRDF has some serious shortcomings for complex class hierarches, especially when hierarchies contain inheritance loops:

$$\begin{tikzcd}
& A & \\
B \arrow[ur] & & C \arrow[ul] \\
& D \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

This sort of an inheritance loop is also referred to as **diamond inheritance.**

A Shortcoming of LRDF

- With regard to the diamond inheritance loop shown on the previous slide, let's say that both **`A`** and **`C`** provide separate definitions for a method named **`foo().`**
- When we call **`foo()`** on a **`C`** instance, we will obviously invoke **`C`**'s **`foo().`**
- Now let's say that **`B`** and **`D`** do not provide their own definitions for **`foo().`**
- If we call **`foo()`** on a **`D`** instance, the LRDF rule will invoke **`A`**'s **`foo()`** even though **`D`** is "closer" to **`C`** than to **`A.`**

LRDF Shortcoming (contd.)

- The behavior of LRDF described on the previous slide is counterintuitive in the sense that after you get used to **`C`** exhibiting its **`foo(),`** you'd expect **`D`** -- since it is derived from **`C`** – to exhibit the same behavior.

- If **`B`**'s insertion in the superclass list of **`D`** --- knowing fully well that **`D`** does not possess **`foo()`** --- causes **`D`** to acquire its **`foo()`** behavior from a different class, you'd be perplexed.

- This limitation of LRDF-based MRO can be formalized by saying that LRDF lacks **monotonicity**.

Superclass Linearization

- For a more precise definition of **monotonicity** in name lookup in inheritance graphs, we need to introduce the notion of **superclass linearization.**

- A **superclass linearization** (*of the inheritance graph*) for a class **`C`**, denoted **`L[C]`**, is the list of all the classes, starting with **`C`**, that should be searched sequentially from left to right for given name.

- Here are the superclass linearizations produced by LRDF-based MRO for the ABCD diamond hierarchy

  `L[A] = A     L[B]=BA    L[C] = CA   L[D] = DBACA`

  When we examine L[C], C's methods get priority over A's. But when we examine L[D], exactly the opposite is true.

Desirable Properties for MRO

- A good MRO algorithm must be **monotonic**.
- A good MRO algorithm must also preserve **local precedence ordering.**
- **Preserving local precedence** ordering means that the order in which the immediate bases of a class P appear in the linearization for P or for any of its subclasses must not violate the base class order of P.
- So whereas monotonicity refers to the priority accorded to the names in a class vis-à-vis the names in a superclass, local precedence order deals with the priority to be accorded to the names in one base class vis-à-vis the names in another base class at the same level of inheritance.

Desirable Properties for MRO (contd.)

- For some class hierarchies, it can be shown trivially that they would not admit good superclass linearizations regardless of what MRO rules are used for the purpose.
- This happens particularly if two different classes inherit from two separate bases but in opposite order.
- Python uses the C3 algorithm to derive superclass linearizations for new-style classes.

Part D of this Tutorial

Some Applications of OO Scripting

Slides 144 – 155

Writing GUI Scripts

- If you want to create a graphical user interface with Perl or Python, a commonly used toolkit for that is Tk.
- With Perl, Tk is used through its object-oriented wrapper module Perl/Tk. The Perl module file for the wrapper is **`Tk.pm`**
- With Python, Tk is used through the object-oriented wrapper module Tkinter. The Python module file for the wrapper is **`Tkinter.py`**

Writing GUI Scripts

- Write Perl and Python scripts for the following GUI:

  The GUI should display two windows side-by-side, one for text entry and the other for displaying geometric figures. As the user enters text in text-entry window, certain words should automatically cause the drawing of some related geometric figure in the other window.

  For example, if the user entered the following phrase in the text window:

  The red herring became green with envy
  when it saw the orange fox jump over a blue
  fish ….

  That should cause the appearance of red, green, orange, and blue tiles to appear at random locations in the draw window.

Writing GUI Scripts

```
#!/usr/bin/perl -w
###  CrazyWindow.pl
use strict;
use Tk;

my $mw = MainWindow->new( -title
                       => “CrazyWindow”);
my $textWin = $mw->Scrolled(‘Text’,
                     -width => 45,
                     -scrollbars => ‘e’,
               )->pack(-side => ‘left’);

my $drawWin = $mw->Canvas()
                 ->pack(-side => ‘right’,
                        -fill => ‘y’,
                        -expand => 1);
$textWin->focus;
$textWin->bind(‘<KeyPress>’ =>
               \&word_accumulator);
…………
…………
```

```
#!/usr/bin/env python
###  CrazyWindow.py
import random
from Tkinter import *

mw = Tk()
mw.title( “CrazyWindow” )
scrollbar = Scrollbar(mw, \
                  orient = ‘vertical’)
textWin = Text(mw, width = 45,
               yscrollcommand =  \
                 scrollbar.set )
scrollbar.config( command = \
               textWin.yview )
scrollbar.pack(side = ‘left’,fill=‘y’)
textWin.focus()
textWin.pack( side = ‘left’ )
drawWin = Canvas( mw )
drawWin.pack(side = ‘right’,fill=‘y’)
…………
…………
```

Scripting for Network Programming

- Scripts for network programming are based on the client-server model of communications.
- Central to the client-server model is the notion of a port and the notion of communicating over a socket through a port.
- A server monitors a designated port for incoming requests from clients.
- A client wishing to communicate with a server sends the server its socket number that is a combination of the client's IP address and the port number on which the client expects to hear back from the server.

Scripting for Network Programming (contd.)

- There are two fundamentally different types of communication links one can establish through a port:

- A one-to-one open and continuous connection that uses handshaking, sequencing, and flow control to ensure that all the information packets sent from one end are received at the other. (The TCP Protocol)

- And a simpler and faster one-shot messaging link that may be operating in one-to-many or many-to-many modes. (The UDP Protocol)

Scripting for Network Programming (contd.)

- In Perl, an internet socket (TCP or UDP) is made by constructing an instance of the **`IO::Socket::INET`** class.

- In Python, the **`socket`** module that comes with the standard distribution of Python provides support for socket programming. One typically constructs a socket object by invoking the function **`socket()`** with appropriate arguments.

- A socket constructor (in both Perl and Python) usually takes the following three arguments (all of them are provided with defaults): the socket address family, the socket type, and the protocol number.

Client Side Sockets for Fetching Docs

```
#!/usr/bin/perl -w
###  ClientSocketFetchDocs.pl
use strict;
use IO::Socket;
die “usage: $0 host doc” unless @ARGV >1;
my $host = shift @ARGV;
my $EOL = “\r\n”;
my $BLANK = $EOL x 2;
foreach my $doc (@ARGV) {
  my $sock = IO::Socket::INET->new(
                 Proto => “tcp”,
                 PeerAddr => $host,
                 PeerPort=>“http(80)”,
           ) or die $@;
  $sock->autoflush(1);
  print $sock “GET $doc HTTP/1.1”.$EOL .
           “Host: $host” . $EOL .
           “Connection: closed” . $BLANK;
  while ( <$sock> ) {print};
  close $sock;
}
```

```
#!/usr/bin/env python
###  ClientSocketFetchDocs.py
import sys
import socket
if len( sys.argv ) < 3:sys.exit(“error” )
host = sys.argv[1]
EOL = “\r\n”
BLANK = EOL * 2
for doc in sys.argv[2:]:
  try:
    socket=socket.socket(socket.AF_INET,
                 socket.SOCK_STREAM)
    socket.connect(host, 80)
  except socket.error, (value,message):
    print message;sys.exit(1)
  sock.send( str(“GET %s HTTP/1.1 %s” +
   “Host: %s%s Connection: closed %s”)
        % (doc, EOL, host, EOL, BLANK)
  while 1:
    data = sock.recv(1024)
    if data = ‘’: break
    print data
```

Interacting with a Flat-File Database

- Small databases (for personal or small-business use) can be efficiently stored as fixed-length or variable-length records in flat files.
- For variable length records, the information in each record is usually stored in either comma-separated or tab-separated form.
- For a flat-file database to be useful, you need scripts for reading from and writing into the database. Although GUI based frameworks are the most convenient for such purposes, command-line based can also do the job. (Even for the GUI front-ends, the user choices must be translated into commands under the hood.)

Interacting with a Flat-File Database (contd)

The next slide shows us creating a Perl and a Python class with the following functionalities:

- It allows a user to view a specific column of the database by the name of that column
- It allows for the flat file to be read into the script in its entirely and for the updated/modified database to be written back out to the file.
- It allows specific entries to be changed by the user in any of the records.
- etc.

Interacting with a Flat-File Database (contd)

```
package CSV;
###  CSV.pm
use strict;
sub new {
  my ($class, $db_file) = @_;
  bless {
    _dbfile => $db_file,
    _db    =>  [],
  }, $class;
}
sub show_schema {
  my $self = shift;
  my @schema = @{$self->{_db}[0];
  print join “  “, @schema;
}
sub retrieve_row { ……… }
sub retrieve_column { ……… }
sub populate_database_from_disk_file {…}
sub write_database_to_file { ……… }
sub show_row_for_last_name { ………… }
sub interactive { ……… }
```

```
###  CSV.pm
import re
class CSV( object ):
  def __init__(self, db_file):
    self._dbfile = db_file
    self._db     = []

  def show_schema(self):
    schema = self._db[0]
    print “  “.join( schema )

  def retrieve_column(self,col_index):
    for i in range(1, len(self._db):
    print self._db[i][col_index]

  def retrieve_row(self, row_index): ……
  def populate_db_from_file(…):…
  def write_database_to_file(……): ………
  def show_row_for_last_name(……):………
  def interactive(self): ………
```

For further information:

For further information on OO modules for Perl and Python, visit

- For Perl modules: http://www.cpan.org
  This is the web site for Comprehensive Perl Archive Network, a repository for all information related to Perl.

- Many of the Python modules you are likely to use for GUI, network, database, and other applications come with the standard distribution. More specialized modules are available from
  http://cheeseshop.python.org/pypi/

- Also visit http://www.activestate.com for comprehensive and easily navigable information regarding both Perl and Python. This web site also includes Perl and Python compilers for the Windows platform.

END OF THE TUTORIAL

- Any comments, feedback, suggestions, etc., regarding this tutorial would be most welcome. Please send them to

  Avi Kak
  kak@purdue.edu

- This tutorial was excerpted from a forthcoming book "Scripting With Objects" by Avinash Kak.