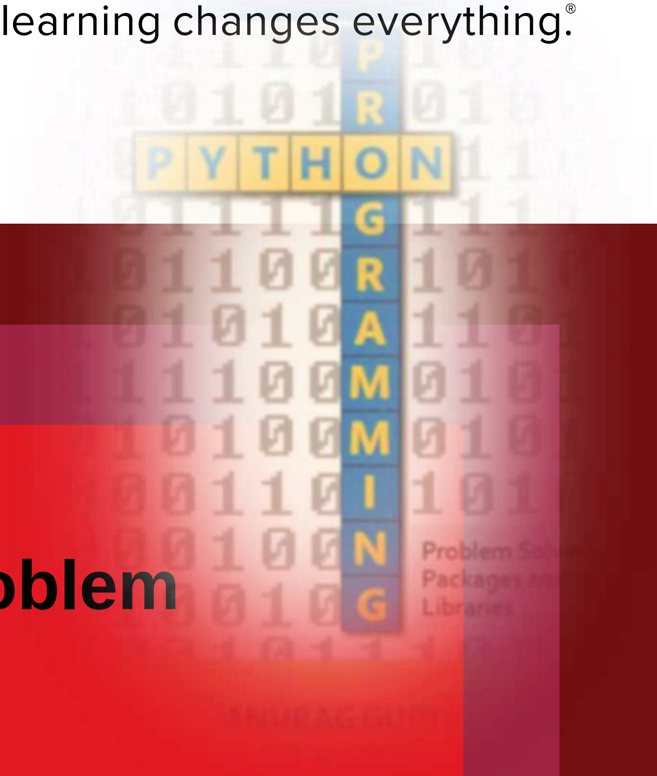
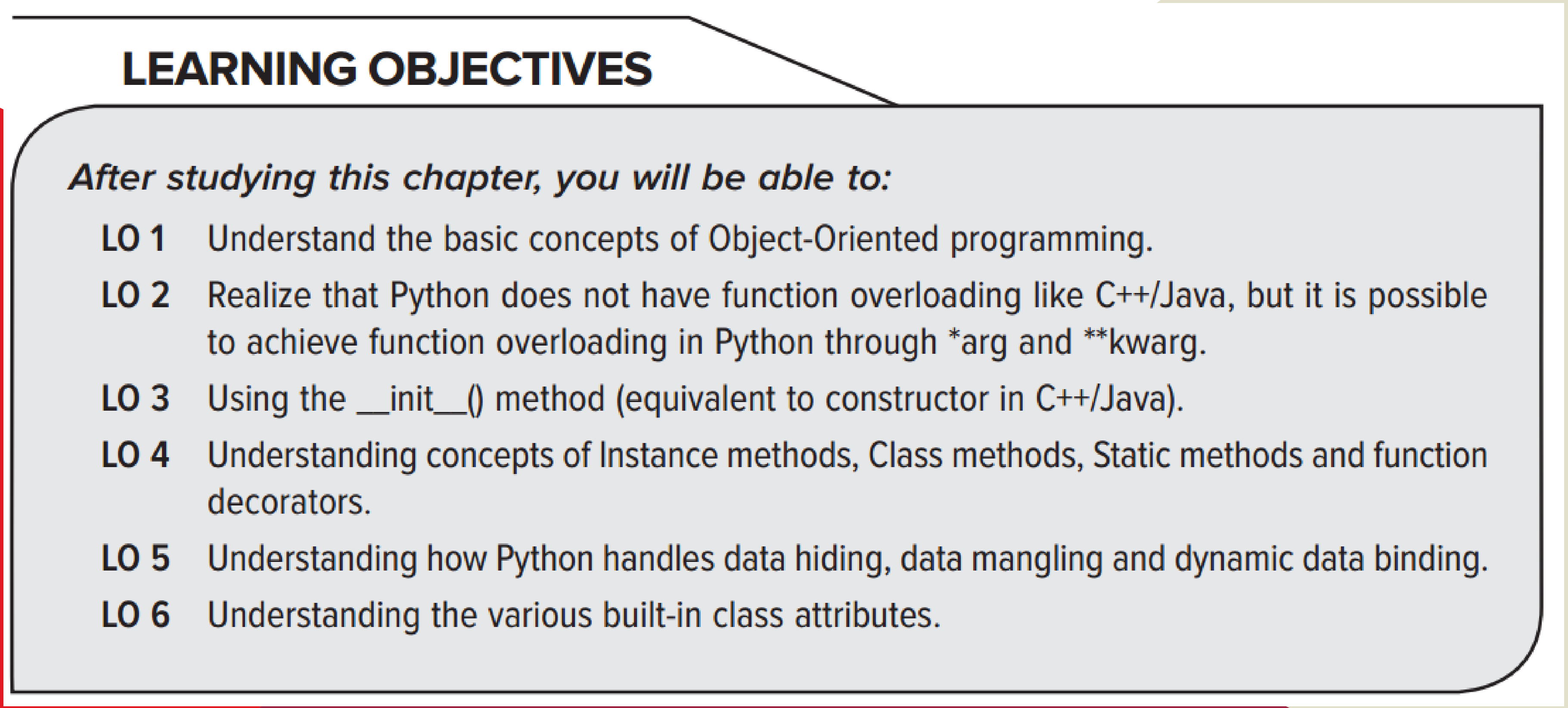
Because learning changes everything.®



TITLE: Python Programming: Problem Solving, Packages and Libraries

Lecture PPT: Chapter 13 Object Oriented Programming

Object Oriented Programming: Learning Objectives



LEARNING OBJECTIVES

After studying this chapter, you will be able to:

LO 1	Understand the basic concepts of Object-Oriented programming.
LO 2	Realize that Python does not have function overloading like C++/Java, but it is possible to achieve function overloading in Python through *arg and **kwarg.
LO 3	Using the __init__() method (equivalent to constructor in C++/Java).
LO 4	Understanding concepts of Instance methods, Class methods, Static methods and function decorators.
LO 5	Understanding how Python handles data hiding, data mangling and dynamic data binding.
LO 6	Understanding the various built-in class attributes.

OOP is introduced in two parts:

(1) Understanding OOP terminology and concepts (which are largely common to most programming languages)

(2) Knowing specifically how OOP concepts are implemented in Python.

Basic concept of object-oriented programming

The concepts of OOP are explained in the following steps:

- **Concept of class and object**
- **Abstraction**
- **Data encapsulation/ data hiding**
- **Inheritance**
- **Polymorphism**

Concept of class, object and abstraction

In computer science, one tries to model the real world. The real world consists of “objects” or “entities” which have “characteristics” and do some “action” or “behaviour”. So in effect one is trying to “model” or “represent” objects of real world through two things namely:

- Characteristics/Qualities
- Action/ Behaviour.

Data Abstraction

Abstraction is the process of keeping only the “most essential” characteristics by “removing or hiding the non-essential” ones. Through abstraction, one can “decrease complexity” making it possible to “create a model” of the object.

See the data abstraction example in the book. In the book, data abstraction is explained taking the example of a “car” and a “driving license”.

Concept of data encapsulation

- **Technically data encapsulation is: “Restricted access to methods or variables”**
- **Languages like Java have a keyword “private” by which “access” to a variable can be restricted.**
- **But Python does not have a similar arrangement.**
- **Encapsulation in Python is achieved by “convention”. The convention is that a variable inside a class which should not be directly accessed should be “pre-fixed” with an underscore (_).**
- **So in Python “private” attributes and methods are not really “hidden” in the same way as they are in Java.**
- **This scheme provides safeguard against “accidental” access to “private” variables, but if a programmer is determined to access these so called “private” variables, then he can do so. How this can be done is explained later.**

Class

The concept of class can be discussed at 2 levels:-

- As a concept
- As an entity used in programming

As a concept a class is:

- “model of an entity created through data abstraction and data encapsulation”.

However the term “class” as an “entity” in a programming language like Python is:

- a “template for constructing objects”.

Objects

Objects can also be discussed at 2 levels:

- **As a concept**
- **As an entity used in programming**

As a concept:-

- Take the example of a car.
- The concept of a “car” is what you have formed in your mind after “abstracting” some common properties of the instances of cars that you see around you.
- So you can say that “cars” don’t exist but “instances of cars” exist. You can extend the same logic to “classes” and “objects”. A “class” is like the “concept” of a car while a real car like say “Maruti 800” is an instance of a “car”.

As an entity used in programming

- Objects are “constructed” from Python classes

Inheritance

(Note that implementation of inheritance in Python is dealt in next chapter)

- **The book gives a good example of inheritance. It takes a class “ParentCar” and a derived or “sub-classed” class called “ChildCar”. The script used for the parent class is “reused” to create a child class with some “extra functionalities”.**
- **The next chapter deals with all concepts related to inheritance.**

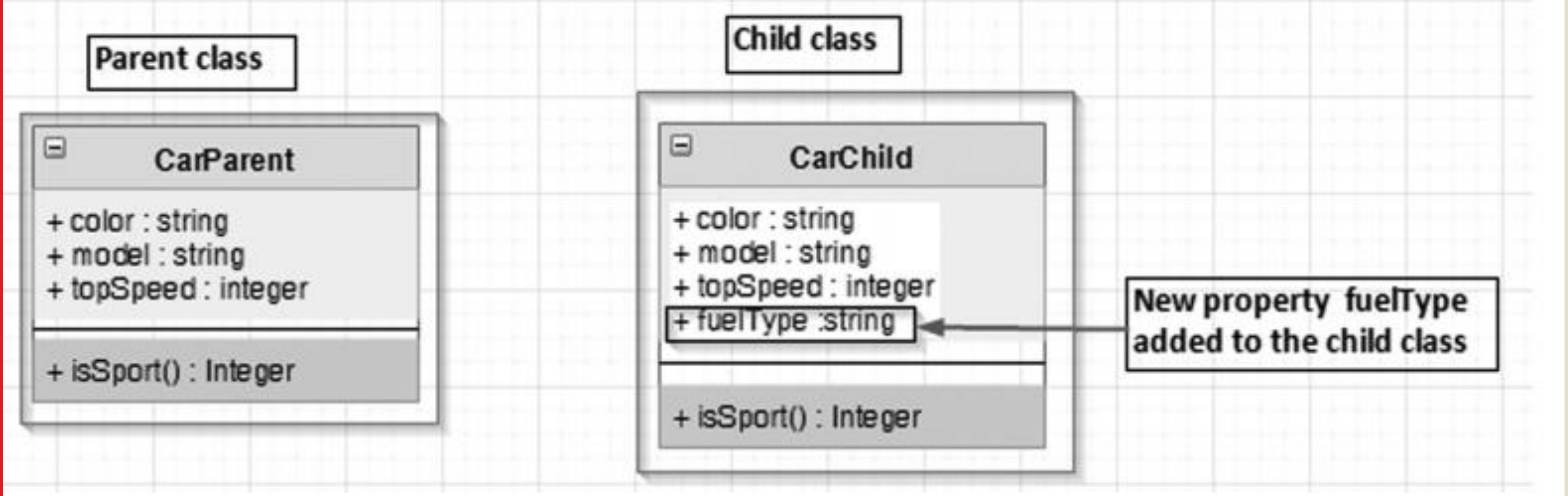


FIGURE 13.2 Code reuse by inheritance. (A new class is created from an existing class and a new attribute is added to it.)

Polymorphism

- **The concept of polymorphism is closely linked to the concept of inheritance.**
- **The book has explained the concept of polymorphism taking the "ParentCar" and "ChildCar" example.**
- **You may say that:-**

  The ability of the property or method of an object to "behave" or "act" differently depending on the "class type" of the object is called polymorphism.

- **The actual implementation of polymorphism in Python is explained in next chapter along with inheritance.**

Operator overloading

The book gives the following example:

- Suppose you have two variables x and y and you write a statement: z = x + y. Here x and y are the operands and + is the operator.
- Suppose x = 3 and y = 4, then both are of type int and the + operator will add these two int values and assign the result to z, i.e., 7.
- But suppose you have x = ‘3’ and y = ‘4’, then what? Should the result be ‘addition’ of numbers to give 7?   Or should it be joining (concatenation) of the two digits (which here are strings) leading to an answer of 34?
- Operator for addition i.e. (+) ‘sees’ the operands on both sides of the operator (i.e., x and y here) and decides accordingly what to do. If both operands are integers it ‘adds’ them. If both operands are strings it ‘concatenates’ them.
- So operator ‘+’ behaves differently depending on the operands. This is operator overloading.

Short note on function overloading

- **Function overloading is not present in Python. Why? Because function overloading is also called “compile time polymorphism” and since Python is not a compiled language but an “interpreted” language, there is no function overloading in Python. However, “function overloading” is available in compiled languages like Java and C++.**
- **It is a very simple concept: It says that in the same piece of code you may have functions with the same name but different numbers of parameters.**
- **So in Java it is possible to have same function name with different numbers (or data types) of variables and the function will act differently depending on the number or type of arguments passed.**
- **This is not possible in Python. If you use the same function name twice in Python, the second definition will prevail.**
- **However Python can implement a scheme “similar” to function overloading using *arg and **kwarg. Tgis was discussed in the chapter on functions.**

Creating classes and objects in Python

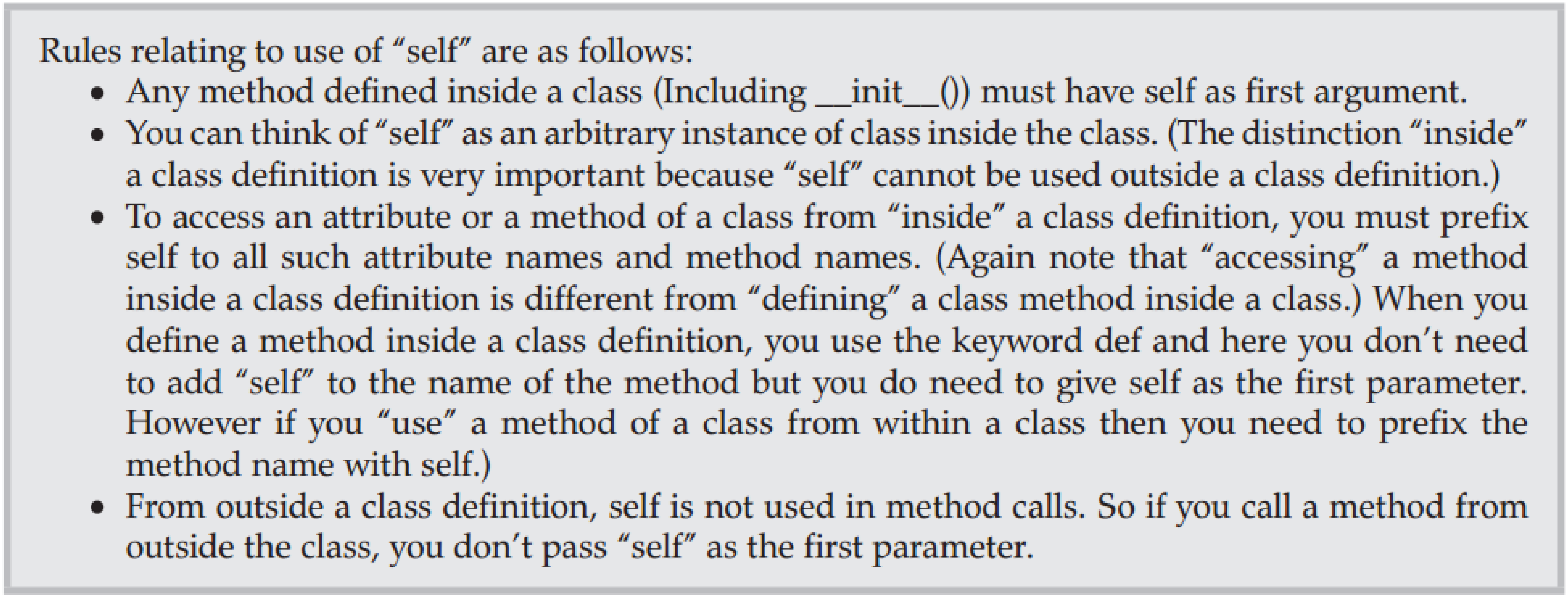
The book explains the process of writing code to create classes. The steps are:

- **Creating a simple class and simple objects (without any constructor)**
- **A class which has an `__init__()` method (ie a class with a constructor)**
- **Also explained is the concept of: self**
- **A class which has attributes, `__init__()` and also default values for `__init__()`.**
- **A class which has attributes as well as member functions or class methods**

The above concepts are explained in details in the book and can be followed from there.

Rules regarding parameter “selg”

The book gives the following rules regarding the use of self parameter:



Rules relating to use of “self” are as follows:

- Any method defined inside a class (Including __init__()) must have self as first argument.
- You can think of “self” as an arbitrary instance of class inside the class. (The distinction “inside” a class definition is very important because “self” cannot be used outside a class definition.)
- To access an attribute or a method of a class from “inside” a class definition, you must prefix self to all such attribute names and method names. (Again note that “accessing” a method inside a class definition is different from “defining” a class method inside a class.) When you define a method inside a class definition, you use the keyword def and here you don’t need to add “self” to the name of the method but you do need to give self as the first parameter. However if you “use” a method of a class from within a class then you need to prefix the method name with self.)
- From outside a class definition, self is not used in method calls. So if you call a method from outside the class, you don’t pass “self” as the first parameter.

Concepts of static methods/ class methods

The need for static methods is explained in the book with an analogy as follows:-

- **Suppose you have a class “Car”. You can think of this class as a “factory which produces Car objects”.**
- **Now the data on how many Car objects are produced are of no concern to the individual Car objects.**
- **However someone say a factory manager, may be interested to know how many Car objects are produced.**
- **So the factory manager may need some method to count the number of Car objects. This is where static methods come in.**
- **So static methods relate to a “class” rather than the “instance of a class”.**

How to create static methods in Python

- There is difference in version 2.x and 3.x in how static methods are created in Python. Both the ways of creating static methods are discussed in the book.
- In this presentation, only the technique of creating static method in Python 3.x are discussed. (For technique for Python 2.x, please refer to the book)
- To create a static method, Python 3.x uses “Function decorator @staticmethod”

What is “Function decorator @staticmethod”?

- A function decorator is placed just before the def statement.
- It starts with a @ symbol.
- For example the function decorator for declaring a method as static in Python is @staticmethod.
- Note that @staticmethod function is just a function “defined inside a class”, nothing more.
- You can “call” a static method without first “instantiating, i.e., creating an instance” of the class.

Example script shows how *@staticmethod* is used

In the following example code, the class Person has a static method called numP(). This method has a “function decorator” called *@staticmethod*.

Refer to the book for explanation

```
class Person:
    count = 0
    def __init__(self, name, sex = 'Female', age = 20):
        Person.count = Person.count +1
        self.name = name
        self.sex = sex
        self.age = age
    @staticmethod       #Function decorator
    def numP():  # No self parameter in numP() because it is staticmethod
        print('count->',Person.count)

# create 3 objects of Person class ie anita, sunita and sunil
anita = Person('Anita')
sunita = Person('Sunita')
sunil = Person('Sunil', 'Male', 19)
Person.numP()   # Call to static method numP() of Person Class
```

```
# OUTPUT
count-> 3
```

Mangling in Python -- 1

Python supports the concept of name “mangling”. The concept is very simple. Inside a class you may use a variable name which is also being used in another class. How do you differentiate between these two similar variables? Especially in case of inheritance where you are deriving child classes from parent classes, there is a chance that the variable names being used by parent or child classes may clash. Python provides a work around to this problem by using the concept of “name mangling”.

Mangling in Python -- 2

The “mangling” algorithm works as follows:

The class variables which are to be mangled are proceeded by two underscores and at most one trailing underscores. So a mangled variable can be named __var or __var_ but not _var or _var__ or __var__

Once the Python interpreter sees a class variable with double underscore, it “mangles” its name to _ClassName.__varName where ClassName is the class to which the variable belongs and __varName is the name of the variable name.

Mangling in Python -- 3

Certain things to note about “mangling” in Python:

1. “Mangling” in Python does not completely provide “private” variables (like in some other programming languages like Java).
2. “Mangling” in Python is more to avoid conflict in variable name rather than to hide the variables from the programmer.
3. In any case, any such variable say varName in class say ClassName can be accessed by _ClassName__varName.
4. Therefore, it is better to call “mangled” names as “pseudo-private” than “private.”

(Continued to next slide----)

Mangling in Python -- 4

(Continued from previous slide----)

5. In general “mangling” is not required unless the programmer is working on a large number of derived classes or working on a big team of programmers. (Since the chance of name clash increases with large class hierarchies and with a number of different programmers working on the same project.)
6. Many OOP languages like C++ and Java have concept of private members and private methods.
7. These members and methods can only be accessed from inside the class definition and not from outside.
8. However, Python does not have private attributes or private methods.

Static versus dynamic binding

- **Static binding is also called early binding while dynamic binding is also called late binding.**
- **A programming language is called “statically typed” if you know the type of a variable at the time of compilation of the program, which is the same as writing the program.**
- **A programing language is called “dynamically typed”, if you need not know the “type” of a variable at compile time, rather you come to know about the “type” of a variable only at run time.**
- **Most scripting languages like Python have this feature of dynamic typing, since there is no compiler to do static type-checking anyway.**
- **In fact in Python it is the Python interpreter which “determines” the type of an object.**

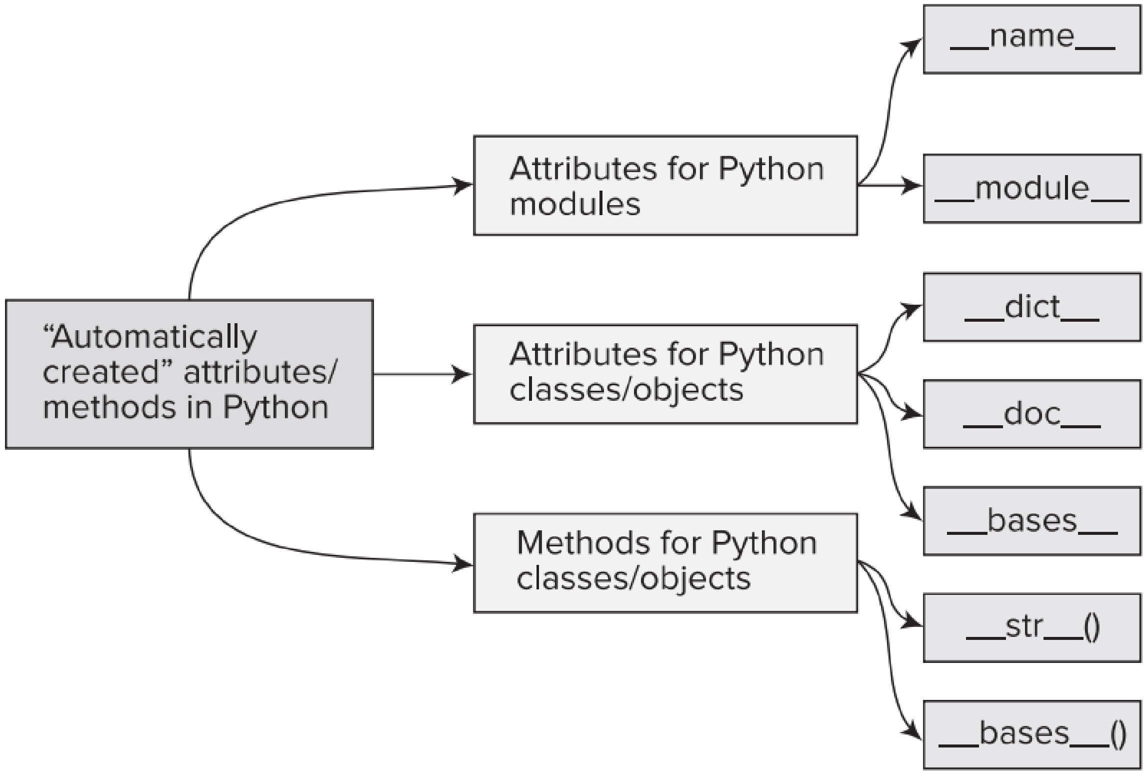
Some common "built in" attributes and methods of modules and classes

- In Python once you define classes and once you create objects of these classes, then python automatically provides certain "built-in" attributes to these classes and to objects of these classes.
- So the "built-in" attributes are not defined by the script writer. Rather they are automatically created by the Python interpreter.

The book discusses:

- Two "automatically created" module attributes namely __name__ and __module.
- Some common "automatically created" attributes for classes/ objects like: __dict__, __doc__ and __bases__.
- Some "automatically created" class/ object methods like: __del__() and __str__().

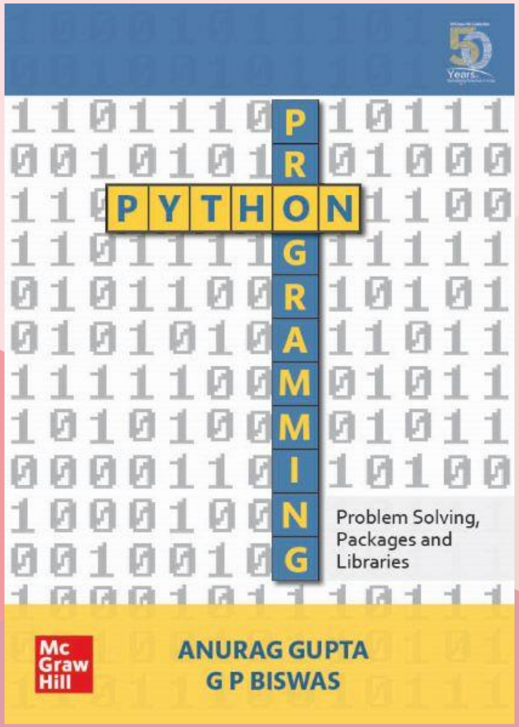
Figure shows common “built in” attributes and methods of modules and classes



“Automatically created” attributes/ methods in Python
Attributes for Python modules
Attributes for Python classes/objects
Methods for Python classes/objects
__name__
__module__
__dict__
__doc__
__bases__
__str__()
__bases__()

FIGURE 13.5 Automatically created attributes and methods of Python modules and classes/objects

These “automatically created” attributes and methods of Python modules, classes and objects are discussed in details in the book. Lot of examples are also give.
Please refer to the book for details


PYTHON
PROGRAMMING
Problem Solving, Packages and Libraries
ANURAG GUPTA
G P BISWAS
Mc Graw Hill

Because learning changes everything.®

Thank You!

For any queries or feedback contact us at:

support.india@mheducation.com

1800-103-5875

www.mheducation.co.in