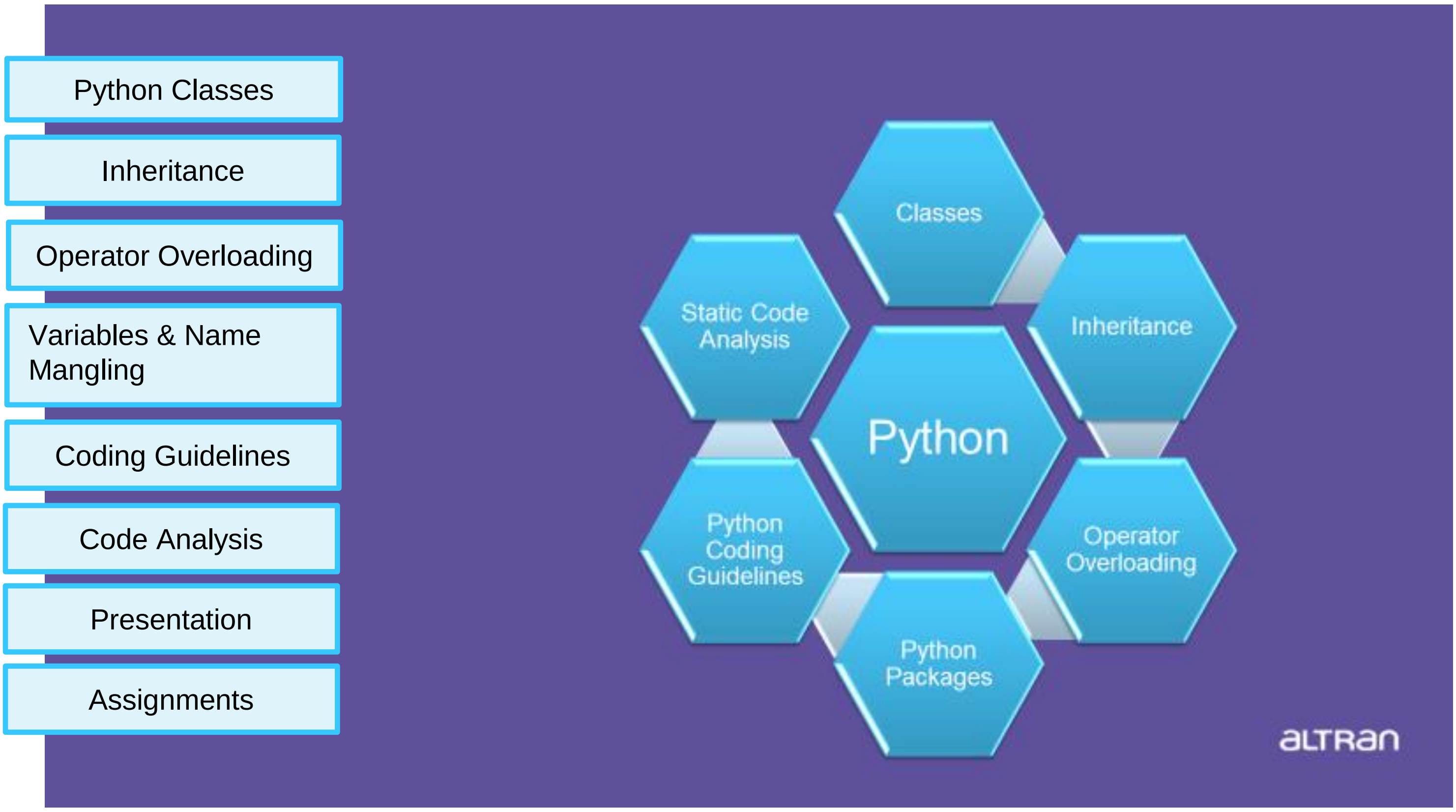


Python Classes

Inheritance

Operator Overloading

Variables & Name Mangling

Coding Guidelines

Code Analysis

Presentation

Assignments

Python Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Refer the conceptual video on Classes & objects, Inheritance, Abstract Class. CLICK HERE

Key-Points to Note:

a) A new class is created using the “class” statement and name of the class

b) This is followed by an indented block of statements (usually function definitions) which form the body of the class.

c) Class definitions must be executed before they have any effect.

d) When a class definition is interpreted, a “class object” is created.

e) Instance variables (here “name”) are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

Click here to know more

f) When a class defines an __init__() method, class instantiation automatically invokes __init__() for the newly-created class instance.

g) A method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. Often, the first argument of a method is called self. This is just a convention.

Inheritance

- In most class-based object-oriented languages, an object created through *inheritance* (a "child object") acquires all the properties and behaviors of the parent object.
- One of the major benefits of object-oriented programming is reuse of code and one of the ways this is achieved is through the inheritance mechanism.

The Python syntax for a derived class definition looks like this:

```
 class DerivedClassName(BaseClassName):
<statement-1>
.
.
<statement-N>
```

Conceptual Video on Inheritance

More about Inheritance

Key-Points to Note:

- The name *BaseClassName* must be defined in a scope containing the derived class definition
- When the class object is instantiated, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.
- Derived classes may override methods of their base classes

- An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name.

Operator Overloading

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names.

This is Python’s approach to *operator overloading*, allowing classes to define their own behavior with respect to language operators.

Conceptual Video on Method Overloading

CLICK HERE

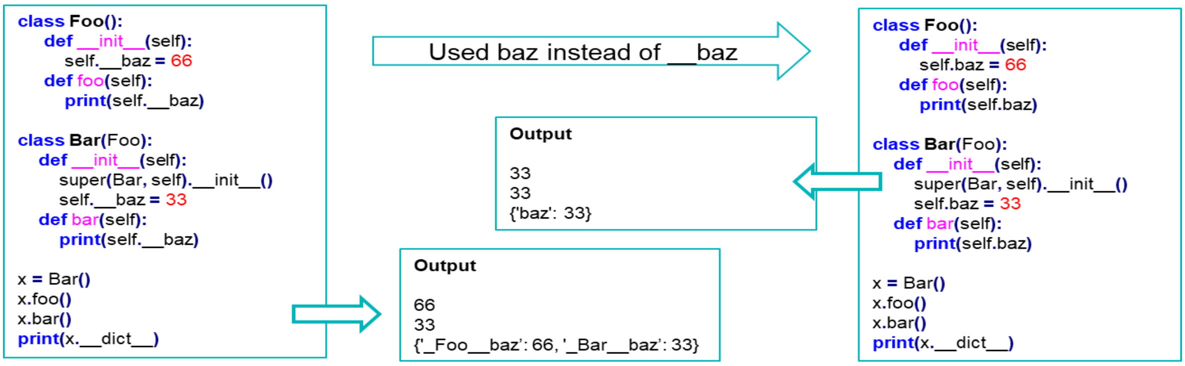
Variables & Name Mangling

Private Variables

Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore should be treated as a
non-public part of the API (whether it is a function, a method or a data member).
It should be considered an implementation detail and subject to change without notice.

Name mangling

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses),
there is limited support for such a mechanism, called *name mangling*.
Any identifier of the form __*var* (at least two leading underscores, at most one trailing underscore) is textually replaced with
_classname__var, where classname is the current class name with leading underscore(s) stripped.



```
class Foo():
    def __init__(self):
        self.__baz = 66
    def foo(self):
        print(self.__baz)

class Bar(Foo):
    def __init__(self):
        super(Bar, self).__init__()
        self.__baz = 33
    def bar(self):
        print(self.__baz)

x = Bar()
x.foo()
x.bar()
print(x.__dict__)
```

```
Output

66
33
{'_Foo__baz': 66, '_Bar__baz': 33}
```

Used baz instead of __baz

```
class Foo():
    def __init__(self):
        self.baz = 66
    def foo(self):
        print(self.baz)

class Bar(Foo):
    def __init__(self):
        super(Bar, self).__init__()
        self.baz = 33
    def bar(self):
        print(self.baz)

x = Bar()
x.foo()
x.bar()
print(x.__dict__)
```

```
Output

33
33
{'baz': 33}
```

Python Packages

Packages are a way of structuring Python's module namespace by using "dotted module names"

(For example, the module name *A.B* designates a submodule named *B* in a package named *A)*

The use of dotted module names saves the authors of multi-module packages from having to worry about each other's module names.

(*Modules like html.parser and email.parser and dateutil.parser all have unique names though the authors of the modules used the same name "parser")*

More about Packages CLICK HERE

Coding Guidelines

Why do we need Coding Guidelines?

- Code is read much more often than it is written!!
- A well written code improves the readability and make it consistent across the wide spectrum of codes
- A style guide is about consistency
  - ➢ Consistency with the style guide is important
  - ➢ Consistency within a project is more important
  - ➢ Consistency within one module or function is the most important

Conceptual Video on Coding guidelines CLICK HERE

Please go through the references below for a detailed discussion.

1. https://www.python.org/dev/peps/pep-0008/
2. https://github.com/google/styleguide/blob/gh-pages/pyguide.md

Key-Points to Note:

- Use 4-space indentation, and no tabs.
- Wrap lines so that they don't exceed 79 characters.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: a = f(1, 2) + g(3, 4).
- Name the classes and functions consistently
- The convention is to use *CamelCase* for classes and *lower_case_with_underscores* for functions and method
- Always use *self* as the name for the first method argument.

Code Analysis

Static Code Analysis is the analysis of computer software that is performed without executing programs. Usually a tool (generally called linter) is used for this. A Lint or a Linter is a program that supports linting (verifying code quality)

Following reports are typically provided by Static Analyzers

- Warnings about syntax errors
- Non-Adherence to Coding Standard
- Duplicated Code
- Suspicious constructs

Static Analyzers are especially useful for interpreted languages like Python. Because such languages lack a compiling phase that displays a list of errors prior to execution.

- PyLint (https://www.pylint.org/)
- Pyflakes (https://pypi.org/project/pyflakes/)
- Pychecker (http://pychecker.sourceforge.net/) Note: *Works for Python 2.7 and below.*

Video on Static Code Analysis CLICK HERE

Presentation CLICK HERE

Assignments CLICK HERE