

NPRG065: Programming in Python *Lecture 8*

<http://d3s.mff.cuni.cz>

Department of
Distributed and
Dependable
Systems



Tomas Bures

Petr Hnetynka

{bures, hnetynka}@d3s.mff.cuni.cz



CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

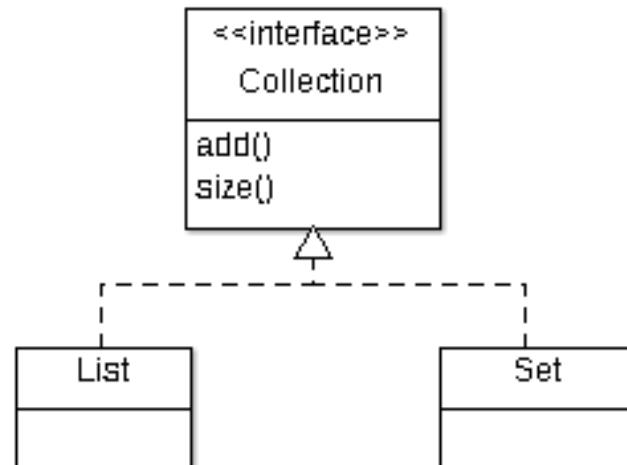
Visibility

- No visibility modifiers like in Java, C++,...
 - everything is public
- Attributes starting with `_` should be considered as private (better call them internal)
- **Name mangling** – partial support for private attributes
 - identifier `__xxx` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__xxx`

Examine and run
mangling.py

Interfaces and Polymorphism

- Structural subtyping
 - aka “Duck typing”
 - “If it walks like a duck and quacks like a duck, it must be a duck.”
- There is not language construct for an interface
 - This means that interfaces form an implicit contract that is captured by comments and documentation



Interfaces and Polymorphism

```
class List:  
    def add(item): ...  
    def size(): ...
```

```
class Set:  
    def add(item): ...  
    def size(): ...
```

```
def add_to_collection(collection, item):  
    collection.add(item)  
  
ls = List()  
st = Set()  
  
item = get_item()  
add_to_collection(ls, item)  
add_to_collection(st, item)
```

Note on Inheritance

- In statically-typed languages, inheritance combines two different features
 - Subclassing
 - Forming a class based on a previous class
 - The aim is to reuse code
 - Subtyping
 - A type may be used in places where supertype is expected
 - The aim is polymorphism
- In Python, however, inheritance brings only subclassing
 - Subtyping is handled at runtime based on method lookup

Properties

- Recommendation
 - directly access object variables
 - i.e., not to use getters and setters
- Sometimes getter and/or setters are necessary
 - e.g., read-only values, computed values, changes in subclasses,...
- Solution => properties
 - property ~ variable with getter, setter and deleter

Properties

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "'x' property.")
```

- Not all of the getter, setter and deleter are necessary
 - can be None

Properties

- Easier specification – via `@property` decorator

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

See
[properties.py](#)

staticmethod

- Methods without self
 - similar static methods in Java and C++
 - methods logically belonging to a class but do not access any object variables

```
class C:  
    @staticmethod  
    def show(msg):  
        print(msg)
```

See
[static.py](#)

Abstract Base Classes

- Module abc
- Support for “interfaces” and methods that must be implemented in subclasses

```
import abc

class PluginBase(abc.ABC):

    @abc.abstractmethod
    def process(self, input):
        pass

class ToUpperPlugin(PluginBase):
    def process(self, input):
        return input.upper()
```

Examine and run
abstract_base.py

Abstract Base Classes

- collections.abc
 - special module for collection like classes

ABC	Inherits from	Abstract Methods	Mixin Methods
Container		<code>__contains__</code>	
Hashable		<code>__hash__</code>	
Iterable		<code>__iter__</code>	
Iterator	Iterable	<code>__next__</code>	<code>__iter__</code>
Reversible	Iterable	<code>__reversed__</code>	
Generator	Iterator	<code>send</code> , <code>throw</code>	<code>close</code> , <code>__iter__</code> , <code>__next__</code>
Sized		<code>__len__</code>	
Callable		<code>__call__</code>	
Collection	Sized, Iterable, Container	<code>__contains__</code> , <code>__iter__</code> , <code>__len__</code>	
Sequence	Reversible, Collection	<code>__getitem__</code> , <code>__len__</code>	<code>__contains__</code> , <code>__iter__</code> , <code>__reversed__</code> , <code>index</code> , and <code>count</code>
MutableSequence	Sequence	<code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__len__</code> , <code>insert</code>	Inherited <code>Sequence</code> methods and <code>append</code> , <code>reverse</code> , <code>extend</code> , <code>pop</code> , <code>remove</code> , and <code>__iadd__</code>
ByteString	Sequence	<code>__getitem__</code> ,	Inherited <code>Sequence</code> methods

Enum

- Module enum
 - enum ~ a class with several named constants

```
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

a = Color.RED

if a is Color.RED:
    print("is red")

for color in Color:
    print(color)
```

Examine and run
enums.py



The slides are licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).