CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

PRG – PROGRAMMING ESSENTIALS

Lecture 11 – Classes & Objects III

Milan Nemy

Czech Technical University in Prague,
Faculty of Electrical Engineering, Dept. of Cybernetics

https://beat.ciirc.cvut.cz/people/milan-nemy/

milan.nemy@cvut.cz

Four Principles of Object-Oriented Programming

1. **Encapsulation** – bundling data and methods that operate on the data into a single unit – the class
2. **Inheritance** – allows one class to inherit properties and behaviors of another class
3. **Polymorphism** – allows objects of different classed to be treated as objects of a common superclass
4. **Abstraction** – hiding unnecessary details from the user

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

Encapsulation

- idea of wrapping data and the methods that work on data within one unit
- restrictions on accessing variables and methods directly
- can prevent the accidental modification of data
- object's variable can only be changed by an object's method
- variables must be accessed via getter and setter methods

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

PROPERTY

@property

- Method to generate a property of an object **dynamically** (*e.g. calculating it from the object's other properties*)

- Use a method to **access a single attribute and return it**

- Use a different method to **update the value of the attribute** instead of accessing it directly

- These methods are called **getters** and **setters**, because they "**get**" and "**set**" the values of attributes, respectively

EXAMPLE – PROPERTY

```python
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    @property
    def fullname(self):
        return "%s %s" % (self.name, self.surname)

    @fullname.setter
    def fullname(self, value):
        # this is much more complicated in real life
        name, surname = value.split(" ", 1)
        self.name = name
        self.surname = surname

    @fullname.deleter
    def fullname(self):
        del self.name
        del self.surname
```

Variables in Classes – Naming Conventions

Naming conventions to signify the intended scope of variables within class:

- Instance (or internal) variables starting with _
  - Internal use within the class of module
  - Not truly private
- Private variables with the __ prefix
  - Name-mangled by Python to prevent direct access from outside the class
  - Still not absolute privacy

Variables in Classes – Naming Conventions

```python
class VariableConvention:
    def __init__(self, name, age):
        self.name = name              # Public variable
        self._internal_id = 12345     # Internal variable (by convention)
        self.__private_data = age     # Private variable (name-mangled)

    def display(self):
        print(f"Name: {self.name}")
        print(f"Internal ID: {self._internal_id}")
        print(f"Private Data: {self.__private_data}")

    def update_private_data(self, new_age):
        self.__private_data = new_age  # Modifying the private variable

# Creating an instance
obj = VariableConvention("Alice", 30)

# Accessing public and internal variables
print(obj.name)           # Output: Alice (Public)
print(obj._internal_id)   # Output: 12345 (Accessible, but conventionally private)
```

```python
# Accessing private variable directly (will raise an AttributeError)
try:
    print(obj.__private_data)
except AttributeError as e:
    print(e)  # Output: 'VariableConvention' object has no attribute '__private_data'
```

```python
# Using methods to update and access private data
obj.update_private_data(35)
obj.display()
# Output:
# Name: Alice
# Internal ID: 12345
# Private Data: 35
```

Getter and Setters

```python
class Person:
    def __init__(self, name, age):
        self._name = name                  # Internal variable (convention)
        self.__age = age                   # Private variable (name-mangled)

    # Getter for the public interface
    @property
    def name(self):
        return self._name

    # Setter for the public interface
    @name.setter
    def name(self, new_name):
        if isinstance(new_name, str) and new_name.strip():
            self._name = new_name
        else:
            raise ValueError("Name must be a non-empty string.")
```

```python
try:
    person.age = -5
except ValueError as e:
    print(e)  # Output: Age must be a positive integer.
```

```python
    # Getter for private variable __age
    @property
    def age(self):
        return self.__age

    # Setter for private variable __age
    @age.setter
    def age(self, new_age):
        if isinstance(new_age, int) and new_age > 0:
            self.__age = new_age
        else:
            raise ValueError("Age must be a positive integer.")

    # A method demonstrating internal and private variables
    def display(self):
        print(f"Name: {self._name}, Age: {self.__age}")
```

```python
# Using the property to access and modify the name
print(person.name)  # Output: Alice
person.name = "Bob"
print(person.name)  # Output: Bob

# Using the property to access and modify the private variable __age
print(person.age)  # Output: 30
person.age = 35
print(person.age)  # Output: 35
```

Inheritance

- Ability to define a new class that is a modified version of an existing class
- ADVANTAGE: add new methods without modifying existing class
- Parent class (superclass, base class) – child class (subclass, derived class)

```
class Car():
    pass
class Yugo(Car):
    pass

In [3]: give_me_car = Car()
In [4]: give_me_yugo = Yugo()
```

```
In [5]: class Car():
   ...:     def exclaim(self):
   ...:         print("I'm a Car!")
   ...:

In [6]: class Yugo(Car):
   ...:     pass
```

```
In [12]: give_me_car = Car()
In [13]: give_me_yugo = Yugo()
In [14]: give_me_car.exclaim()
I'm a Car!
In [15]: give_me_yugo.exclaim()
I'm a Car!
```

Inheritance – Override a Method

- New class inherits everything from its parent class
- How to replace or override a parent method?

```
In [16]: class Car():
    ...:     def exclaim(self):
    ...:         print("I'm a Car!")
    ...:
In [17]: class Yugo(Car):
    ...:     def exclaim(self):
    ...:         print("I'm a Yugo!")
```

```
In [22]: car = Car()
In [23]: yugo = Yugo()
In [24]: car.exclaim()
I'm a Car!
In [25]: yugo.exclaim()
I'm a Yugo!
```

Inheritance – Override a Method

- New class inherits everything from its parent class
- How to replace or override a parent method?

```python
class Person():  2 usages
    def __init__(self, name):
        self.name = name

class MDPerson(Person):
    def __init__(self, name):
        self.name = "Doctor " + name

class ProfPerson(Person):
    def __init__(self, name):
        self.name = "Professor " + name
```

```python
person = Person('Fudge')
doctor = MDPerson('Fudge')
professor = ProfPerson('Fudge')
print(person.name)
print(doctor.name)
print(professor.name)
```

```
Fudge
Doctor Fudge
Professor Fudge
```

Inheritance – Add a Method

- New class inherits everything from its parent class
- How to replace or override a parent method?

```python
class Car():  2 usages
    def exclaim(self):
        print("I'm a Car")

class Yugo(Car):  1 usage
    def exclaim(self):
        print("I'm a Yugo")
    def honk(self):  1 usage
        print("brrrROOOARRRRRP")
```

```python
yugo = Yugo()
yugo.honk()
```

```
brrrROOOARRRRRP
```

```python
car = Car()
car.honk()
```

```
Traceback (most recent call last):
  File "C:\Users\milan\PycharmProjects\CNN\tst.py", line 12, in <module>
    car.honk()
AttributeError: 'Car' object has no attribute 'honk'
```

Getting Help from Your Parent

- Child class can override a method from the parent
- What if it wanted to call that parent method?

```python
class Person():  1 usage
    def __init__(self, name):
        self.name = name


class EmailPerson(Person):  1 usage
    def __init__(self, name, email):
        super().__init__(name)
        self.email = email
```

```python
bob = EmailPerson( name: 'Bob Griffin',  email: 'bob@griffin.com')
print(bob.name)
print(bob.email)
```

↓

```
Bob Griffin
bob@griffin.com
```

Polymorphism

- It is possible to apply the same operation to different objects, regardless of their class

```python
class Quote():  2 usages
    def __init__(self, person, words):
        self.person = person
        self.words = words
    def who(self):
        return self.person
    def says(self):
        return self.words + "."

class QuestionQuote(Quote):
    def says(self):
        return self.words + "?"

class ExclamationQuote(Quote):
    def says(self):
        return self.words + "!"
```

Different versions of say() provide different behavior

__init__() of the parent class Quite called automatically!

```
In [4]: speaker1 = Quote('Charlie Brown', "Good grief")
In [5]: print(speaker1.who(), 'says:', speaker1.says())
   ...:
Charlie Brown says: Good grief.
In [6]: speaker2 = QuestionQuote('Snoopy', "Do you have any cookies")
   ...:
In [7]: print(speaker2.who(), 'says:', speaker2.says())
   ...:
Snoopy says: Do you have any cookies?
In [8]: speaker3 = ExclamationQuote('Lucy', "Get off my football")
   ...:
In [9]: print(speaker3.who(), 'says:', speaker3.says())
   ...:
Lucy says: Get off my football!
```

Polymorphism

- Python goes further and lets you run who() and says() methods of *any* objects that have them

BabblingBrook has no relation to Quote class or its descendants!

```
In [10]: class BabblingBrook():
    ...:     def who(self):
    ...:         return 'Brook'
    ...:     def says(self):
    ...:         return 'Babble'
    ...:
In [11]: brook = BabblingBrook()
```

```
In [12]: def who_says(obj):
    ...:     print(obj.who(), 'says', obj.says())
```

```
In [13]: who_says(speaker1)
Charlie Brown says Good grief.
In [14]: who_says(speaker2)
Snoopy says Do you have any cookies?
In [15]: who_says(speaker3)
Lucy says Get off my football!
In [16]: who_says(brook)
Brook says Babble
```

This principle is sometimes called *duck typing.*

Duck test:
"If it walks like a duck and it quacks like a duck, then it must be a duck“
Meaning:
An object's suitability for use is determined by its behavior rather than its explicit type.

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

Class Point

```python
class Point:
    """ Create a new Point, at coordinates x, y """

    def __init__(self, x=0, y=0):
        """ Create a new point at x, y """
        self.x = x
        self.y = y

    def distance_from_origin(self):
        """ Compute my distance from the origin """
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5
```

```python
class Point:
    # Previously defined methods here...

    def __add__(self, other):
        return Point(self.x + other.x,  self.y + other.y)
```

```python
def __mul__(self, other):
    return self.x * other.x + self.y * other.y
```

```python
def __rmul__(self, other):
    return Point(other * self.x,  other * self.y)
```

```python
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print(p1 * p2)
43
>>> print(2 * p2)
(10, 14)
```

source http://openbookproject.net/thinkcs/python/english3e/classes_and_objects_I.html

POLYMORPHISM

```python
def multadd (x, y, z):
    return x * y + z
```

```
>>> multadd (3, 2, 1)
7
```

```
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print(multadd (2, p1, p2))
(11, 15)
>>> print(multadd (p1, p2, 1))
44
```

- **Polymorphism ==** ability to process objects differently based on data type
- There are certain operations that can be applied to many types, such as the arithmetic operations …
- **EXAMPLE**: *The **multadd** operation takes three parameters: multiplies the first two and then adds the third*

source http://openbookproject.net/thinkcs/python/english3e/even_more_oop.html

POLYMORPHISM

```python
def front_and_back(front):
    import copy
    back = copy.copy(front)
    back.reverse()
    print(str(front) + str(back))
```

```
>>> my_list = [1, 2, 3, 4]
>>> front_and_back(my_list)
[1, 2, 3, 4][4, 3, 2, 1]
```

- **EXAMPLE:** ***front_and_back*** *– consider a function which prints a list twice: forward and backward*
- The reverse method is a **modifier** therefore a copy needs to be made before applying it (this way we prevent to modify the list the function gets as a parameter!)
- Function that can take arguments with different types and handles them accordingly is called **polymorphic**

source http://openbookproject.net/thinkcs/python/english3e/even_more_oop.html

POLYMORPHISM

```python
def reverse(self):
    (self.x , self.y) = (self.y, self.x)
```

```
>>> p = Point(3, 4)
>>> front_and_back(p)
(3, 4)(4, 3)
```

- **Python's fundamental rule of polymorphism** is called the **duck typing rule**: *If all of the operations inside the function can be applied to the type, the function can be applied to the type*.
- Operations in the **front_and_back** : *copy, reverse, print*
- **EXAMPLE**: What about our Point class?
  The copy method works on any object; already written a **__str__** method for Point objects for the str() conversion, only the reverse method for the Point class is needed!

source http://openbookproject.net/thinkcs/python/english3e/even_more_oop.html

Abstraction

- Abstraction focuses on hiding the implementation details and showing only the essential features of an object
- It allows the user to focus on what an object does rather than how it does it

abc module (Abstract Base Classes)

Animal inherits *ABC*

@abstractmethod decorator

```python
from abc import ABC, abstractmethod

# Abstract base class
class Animal(ABC):
    @abstractmethod
    def sound(self):
        """Abstract method, must be implemented by subclasses."""
        pass

    @abstractmethod
    def move(self):
        """Abstract method, must be implemented by subclasses."""
        pass
```

Objects based on Animal cannot be initialized!

Abstract methods MUST be implemented by derived classes.

Abstraction

- Abstraction focuses on hiding the implementation details and showing only the essential features of an object
- It allows the user to focus on what an object does rather than how it does it

```python
from abc import ABC, abstractmethod

# Abstract base class
class Animal(ABC):
    @abstractmethod
    def sound(self):
        """Abstract method, must be implemented by subclasses."""
        pass

    @abstractmethod
    def move(self):
        """Abstract method, must be implemented by subclasses."""
        pass
```

```python
# Subclass implementing the abstract methods
class Dog(Animal):
    def sound(self):
        return "Bark"

    def move(self):
        return "Runs on four legs"

# Subclass implementing the abstract methods
class Bird(Animal):
    def sound(self):
        return "Chirp"

    def move(self):
        return "Flies in the sky"
```

```python
# Using the abstraction
def animal_activity(animal: Animal):
    print(f"Animal sound: {animal.sound()}")
    print(f"Animal movement: {animal.move()}")
```

Abstraction

- Abstraction focuses on hiding the implementation details and showing only the essential features of an object
- It allows the user to focus on what an object does rather than how it does it

```python
# Subclass implementing the abstract methods
class Dog(Animal):
    def sound(self):
        return "Bark"

    def move(self):
        return "Runs on four legs"

# Subclass implementing the abstract methods
class Bird(Animal):
    def sound(self):
        return "Chirp"

    def move(self):
        return "Flies in the sky"
```

```python
# Using the abstraction
def animal_activity(animal: Animal):
    print(f"Animal sound: {animal.sound()}")
    print(f"Animal movement: {animal.move()}")
```

```
In [3]: dog = Dog()
In [4]: bird = Bird()
In [5]: animal_activity(dog)
Animal sound: Bark
Animal movement: Runs on four legs
In [6]: animal_activity(bird)
Animal sound: Chirp
Animal movement: Flies in the sky
```

The abstract class tells the user: "You can call *sound* and *move* on any *Animal*, but you don’t need to know how they are implemented."

REFERENCES

This lecture re-uses selected parts of the OPEN BOOK PROJECT

Learning with Python 3 (RLE)

http://openbookproject.net/thinkcs/python/english3e/index.html

available under **GNU Free Documentation License Version 1.3**)

- Version date: October 2012
- by Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers (based on 2nd edition by Jeffrey Elkner, Allen B. Downey, and Chris Meyers)
- Source repository is at https://code.launchpad.net/~thinkcspy-rle-team/thinkcspy/thinkcspy3-rle
- For offline use, download a zip file of the html or a pdf version from http://www.ict.ru.ac.za/Resources/cspw/thinkcspy3/

This lecture re-uses selected parts of the PYTHON TEXTBOOK

Object-Oriented Programming in Python

http://python-textbok.readthedocs.io/en/1.0/Classes.html#

(released under CC BY-SA 4.0 licence Revision 8e685e710775)