Encapsulation

Benefits of Encapsulation

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Table of Contents

Have a Question?

Software University

sli.do

#python-advanced

Encapsulation Definition

What is Encapsulation?

- **Packing** of data and functions into a single component
- Allows us to put **restrictions** and can **prevent the accidental modification** of data
- To do that, an object's variable can only be changed by an object's **method**

Encapsulation in Python

- Everything written within the Python class (methods and variables) are **public by default**
- However, **Python** implements weak encapsulation. This means it is **performed by convention** rather than being enforced by the language

Access Modifiers

- How to control access?
  - Using a **single underscore**
  - Using **double underscore** (making it "private")
  - Using **getter and setter methods** to access private variables
- It is **a matter of convention** to differentiate them into three terms – **public**, **protected** and **private**

Single Underscore

- Using a single leading underscore is just a **convention**
- A name prefixed with an underscore should be treated as a **non-public** method/ variable

```
class Person:
    def __init__(self, name, age=0):
        self.name = name
        self._age = age

person = Person('Peter', 25)
print(person.name)          # Peter
print(person._age)          # 25
```

Double Underscore

- When naming an attribute with **two leading underscores**, it invokes **name mangling**
- In Python, mangling is used for attributes that one class **does not want subclasses to use**
- Python **does not restrict access** to such attributes
- It is still possible to **access** or **modify** a variable that is considered "private" **from outside** the class

Example: Double Underscore

Software University

```
class Person:
    def __init__(self, name, age=0):
        self.name = name
        self.__age = age

    def info(self):
        print(f"I am {self.name}, {self.__age} years old.")

person = Person('Peter', 25)
```

```
# accessing data using the class method
person.info()      # I am Peter, 25 years old.

# accessing data directly from outside
print(person.name) # Peter
print(person.__age) # AttributeError: 'Person' object has no attribute '__age'
```

Name Mangling a Variable

Name Mangling a Variable

- Used to show that the variable **should not be accessed** outside the class

```
class Car:
    def __init__(self):
        self.__max_speed = 200

    def drive(self):
        print('driving max speed ' + str(self.__max_speed))

red_car = Car()
red_car.drive()             # driving max speed 200
red_car.__max_speed = 10    # won't change because it is name mangled
red_car.drive()             # driving max speed 200
```

Getter and Setter Methods

- Used to **access and change** the private variables as they are part of the class

```
class Person:
    def __init__(self, name, age=0):
        self.name = name
        self.__age = age

    def info(self):
        print(self.name)
        print(self.__age)

    def get_age(self):          # getter
        print(self.__age)

    def set_age(self, age):     # setter
        self.__age = age
```

```
person = Person('Peter', 25)

# accessing data using class method
person.info()       # Peter
                    # 25

# changing age using setter
person.set_age(26)
person.get_age()    # 26
```

Problem: Person

- Create a class called **Person**. Upon initialization, it should receive **name** and **age**
- Create name mangled properties - **name** and **age**
- Create **get_name** and **get_age** methods to return their values

```
person = Person("George", 32)
print(person.get_name())
print(person.get_age())
```

```
George
32
```

Solution: Person

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age
```

Problem: Mammal

- Create a class called `Mammal`. Upon initialization, it should receive a `name`, `type`, and `sound`
- Name mangle a class attribute `kingdom = "animals"`
- Create three more instance methods:
  - `make_sound()` - returns `"{name} makes {sound}"`
  - `get_kingdom()` - returns the private `kingdom` attribute
  - `info()` - returns `"{name} is of type {type}"`

Solution: Mammal

```
class Mammal:
    __kingdom = 'animals'

    def __init__(self, name, type, sound):
        # TODO: Implement

    def make_sound(self):
        return f"{self.name} makes {self.sound}"

    def get_kingdom(self):
        return self.__kingdom

    def info(self):
        return f"{self.name} is of type {self.type}"
```

Getters and Setters (1)

- The "pythonic" way of defining **getters and setters** is using **properties**
- By defining properties, you can **change** the **internal implementation** of a class without affecting the program

```
class Person:
    def __init__(self, age=0):
        self.age = age
    @property
    def age(self):
        return self.__age
    @age.setter
    def age(self, age):
        if age < 18: self.__age = 18
        else: self.__age = age
```

```
person = Person(25)
print(person.age)   # 25
person.age = 10
print(person.age)   # 18
```

Example: Getters and Setters

```
class Car:
    def __init__(self, max_speed: int):
        self.max_speed = max_speed

    def drive(self):
        print('driving max speed ' + str(self.max_speed))

    @property
    def max_speed(self):
        return self.__max_speed

    @max_speed.setter
    def max_speed(self, value):
        if value > 447:
            value = 447
        self.__max_speed = value

red_car = Car(200)
red_car.drive()              # driving max speed 200
red_car.max_speed = 512      # changes the speed to 447
red_car.drive()              # driving max speed 447
```

Implement properties only if needed

Getters and Setters (2)

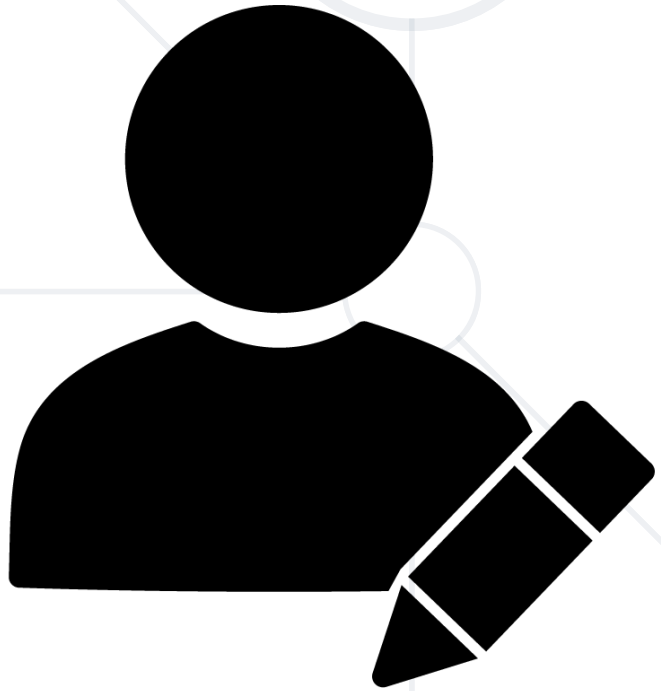
- You should use Python properties to **apply rules** to an attribute

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

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, value):
        if value <= 0:
            raise Exception("Age must be greater than zero")
        self.__age = value
```

An exception will be thrown on any attempt to violate the rule

Problem: Profile

- Read the problem description here
- Create a class as described in the problem description and test your class with your own examples
- Submit only your class in the judge system



Name Mangling a Method

Name Mangling a Method

- It is a **class method** that should only be called from **inside the class** where it is defined

```
class Person:
    def __init__(self):
        self.first_name = 'Peter'
        self.last_name = 'Parker'

    def __full_name(self):
        return f'{self.first_name} {self.last_name}'

    def info(self):
        return self.__full_name()
```

```
person = Person()
print(person.info())                  # Peter Parker
print(person.__full_name())           # AttributeError
print(person._Person__full_name())    # Peter Parker
```

Problem: Email Validator

- Create a class as described in the problem description here
- Test your class with the provided test code or with your own examples

```
mails = ["gmail", "softuni"]
domains = ["com", "bg"]
email_validator = EmailValidator(6, mails, domains)
print(email_validator.validate("pe77er@gmail.com"))
print(email_validator.validate("georgios@gmail.net"))
print(email_validator.validate("stamatito@abv.net"))
print(email_validator.validate("abv@softuni.bg"))
```

```
True
False
False
False
```

Solution: Email Validator

```
class EmailValidator:
    def __init__(self, min_length, mails,  domains):
        self.min_length = min_length
        self.mails = mails
        self.domains = domains
    def __is_name_valid(self, name):
        return len(name) >= self.min_length
    def __is_mail_valid(self, mail):
        return mail in self.mails
    def __is_domain_valid(self, domain):
        return domain in self.domains
    def validate(self, email):
        # TODO: Implement
```

Built-in Functions for Accessing Attributes

Get Attribute Function

- `getattr()` - used to access the attribute of an object
- Returns **the value** of the named attribute
- The method takes multiple parameters:
  - **Object**
  - **Name**
  - **Default (optional)**

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

person = Person('Peter')
print(getattr(person, 'name'))          # True
print(getattr(person, 'age'))           # AttributeError
print(getattr(person, 'age', 'None'))   # None
```

__getattr__()

- Called when an attribute lookup **has not found** the attribute in the usual places
- The method takes **one** parameter – **the name of the attribute**

```
class Phone:
    def __getattr__(self, attr):
        return None

phone = Phone()
print(phone.color)              # None
print(getattr(phone, 'size')) # None
```

- When accessing `phone.color`, Python calls `phone.__getattr__('color')`

Has Attribute Function

- `hasattr()` - checks if an attribute exists or not
- Returns `True` if an object has the given named attribute and `False` if it does not
- The method takes two parameters:
  - **Object**
  - **Name**

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

person = Person('Peter')
print(hasattr(person, 'name')) # True
print(hasattr(person, 'age'))  # False
```

Set Attribute Function

- `setattr()` - used to set the value of the attribute
- Returns **None**
- The method takes three parameters:
  - **Object**
  - **Name**
  - **Value**

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

person = Person('Peter')
print(setattr(person, 'name', 'George'))  # None
print(person.name)                        # George
print(setattr(person, 'age', 21))         # None
print(person.age)                         # 21
```

__setattr__()

- Called when an attribute **assignment is attempted**
- The method takes two parameters:
  - The **name** of the attribute
  - The **value** we want to assign to the attribute

```
class Phone:
    def __setattr__(self, attr, value):
        self.__dict__[attr] = value.upper()

phone = Phone()
phone.color = 'black'
print(phone.color)  # BLACK
```

Delete Attribute Function

- `delattr()` - deletes an attribute from the object
- If you are accessing the attribute after deleting, it raises `AttributeError`
- The method takes two parameters:
  - **Object**
  - **Name**

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

person = Person('Peter')
print(person.name)                  # Peter
print(delattr(person, 'name'))      # None
print(person.name)                  # AttributeError
```

__delattr__()

- Called when an attribute **deletion is attempted**
- The method takes one parameter:
  - The **name** of the attribute
- It should only be implemented if `del obj.name` is **meaningful** for the object

```
class Phone:
    def __delattr__(self, attr):
        del self.__dict__[attr]
        print(f"'{str(attr)}' was deleted")

phone = Phone()
phone.color = 'black'
del phone.color  # 'color' was deleted
```

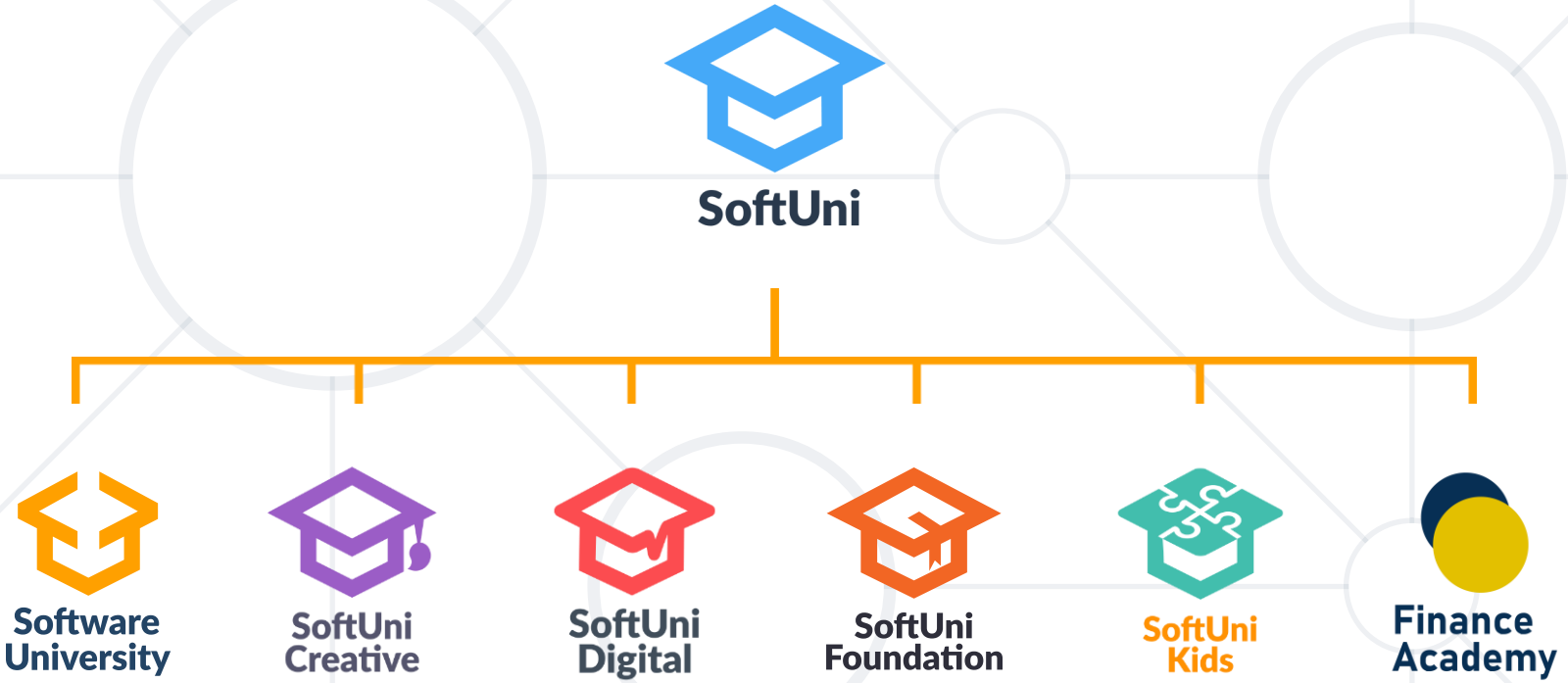
Example

```
class Employee:
    name = 'Diyan'
    salary = '25000'

    def show(self):
        print(self.name)
        print(self.salary)

employee = Employee()
print(getattr(employee, 'name'))    # Diyan
print(hasattr(employee, 'name'))    # True
setattr(employee, 'height', 152)
print(getattr(employee, 'height')) # 152
delattr(Employee, 'salary')
```

Summary

- **Encapsulation** is packing of data and functions into a single component
- **Name mangling** is used for attributes that one class does not want subclasses to use
- The **property decorator** is the pythonic way of using getters and setters
- We could use built-in functions for **accessing attributes**

Questions?



SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

createX

Postbank
Решения за твоето утре

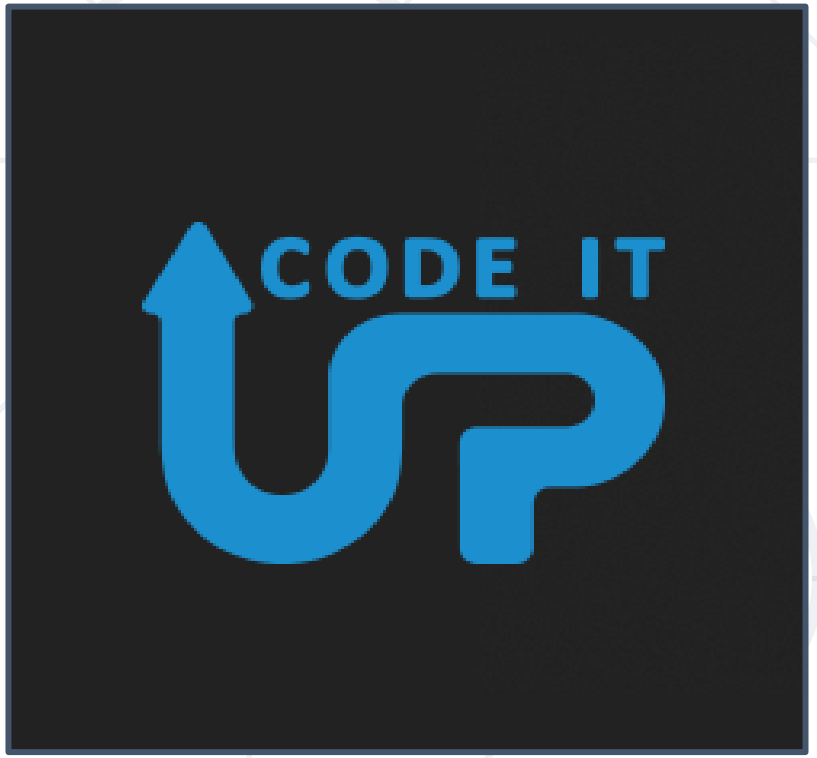
BOSCH

DXC TECHNOLOGY

SmartIT

Educational Partners

Software University



Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License

- This course (slides, examples, demos, exercises, homework, documents, videos, and other assets) is **copyrighted content**
- Unauthorized copy, reproduction, or use is illegal
- © SoftUni – https://about.softuni.bg
- © Software University – https://softuni.bg