Classes and Objects

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#python-advanced

Table of Contents

Classes and Instances

Class Objects

- Classes support two kinds of operations:
  - **attribute references** – **access** them using the " **.** " operator
  - **instantiation** - uses **function notations**

```
class Example:
    text = 'Hello'

    def print_text(self):
        return 'SoftUni'

Example.text          # attribute reference
Example.print_text    # attribute reference
x = Example()         # instantiation
```

Instantiation

- It is known as "**calling**" the class
- Creates an **empty object** – a new instance of the class
- Assign the object to a **local variable**

```
class Person:
    name = "George"
    age = 25

person = Person()
print(person.name) # George
print(person.age)  # 25
```

__init__()

- Creates objects with instances, customized to a **specific initial state**
- **Automatically invoked** for the newly created class instance

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

my_laptop = Laptop("Inspiron 15", "Dell")
```

self Parameter

- `self` is used to represent **the instance of the class**
- It **binds the attributes** with the given arguments
- It is **not a keyword**, but using it increases the readability of the code

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

my_laptop = Laptop("Inspiron 15", "Dell")
```

Instance Objects

- Instances support only one kind of operation:
  - **attribute references** - **access** them using the "**.**" operator

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

my_laptop = Laptop("Inspiron 15", "Dell")
print(my_laptop.name)  # Inspiron 15
print(my_laptop.model)  # Dell
```

Problem: Vehicle

- Create a class called `Vehicle`
- Upon initialization, it should receive `max_speed` and `mileage`
- The `max_speed` should be a default argument = 150
- Create an instance variable called `gadgets` – empty list

```
car = Vehicle(20)
print(car.max_speed)
print(car.mileage)
print(car.gadgets)
car.gadgets.append('Hudly Wireless')
print(car.gadgets)
```

```
150
20
[]
['Hudly Wireless']
```

Solution: Vehicle

```
class Vehicle:
    def __init__(self, mileage,
max_speed=150):
        self.max_speed = max_speed
        self.mileage = mileage
        self.gadgets = []
```

Attributes

Attributes

- **Data** and **procedures** that "belong" to the class
- Valid attribute names are the ones **in the class's namespace**
- There are two kinds of attribute references:
  - **Methods**
  - **Data attributes**

Methods

Instance Methods

- Define the **behavior** of the object
- The **instance object** is passed as a **first argument** of the method - using "`self`" by convention

```
class MyClass:

    def say_hello(self):
        return 'Hello'

x = MyClass()
x.say_hello()          # conventional way
MyClass.say_hello(x) # equivalent
```

Special / Dunder Methods

- **Built-in methods** that you can define to add "**magic**" to your classes
- Surrounded by **double underscores** e.g., **__init__**
- **Enrich** the class design and **enhance** the readability

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

x = Dog("Max")
print(x.__dict__) # {"name": "Max"}
```

Methods

- We could **change the state** of the object using methods

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

    def change_name(self, new_name):
        self.name = new_name

x = Dog("Max")
x.change_name("Rex")
print(x.name) # Rex
```

__str__() Method

- __str__() - returns a printable **string representation** of any user-defined class

```
class MyClass:
    def __str__(self):
        return 'This is My Class'

my_instance = MyClass()

print(str(my_instance))       # This is My Class
print(my_instance.__str__()) # This is My Class
print(my_instance)            # This is My Class
```

__repr__() Method

- __repr__() - returns a **machine-readable representation** of any user-defined class

```
class MyClass:
    def __repr__(self):
        return 'This is My Class'

my_instance = MyClass()

print(repr(my_instance))
print(my_instance.__repr__())
print(my_instance)                  # This is My Class
# use print() only when __repr__() returns string
```

Problem: Point

- 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

```
p = Point(2, 4)
print(p)
p.set_x(3)
p.set_y(5)
print(p)
```

```
The point has coordinates (2,4)
The point has coordinates (3,5)
```

Solution: Point

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def set_x(self, new_x):
        self.x = new_x

    def set_y(self, new_y):
        self.y = new_y

    def __str__(self):
        return f'The point has coordinates ({self.x},{self.y})'
```

Data Attributes

Data Attributes

- Values that are **stored internally** and are **unique** to that object
- They define the **state** of the object
- There are two types of data attributes:
  - **Instance variables**
  - **Class variables**

Instance vs Class Variables

- **Instance variables** are unique to each instance
- **Class variables** are shared by all instances of the class

```
class Laptop:
    brand = "Dell"                  # class variable

    def __init__(self, name):
        self.name = name            # instance variable

first_laptop = Laptop("Latitude 5300")
second_laptop = Laptop("Inspiron 15")
print(first_laptop.brand == second_laptop.brand) # True
print(first_laptop.name == second_laptop.name)   # False
```

Example: Bad Practice

- It is **not** a good practice to **declare** or **remove** data attributes **outside the class**

```
class Laptop:
    def __init__(self, model):
        self.model = model

my_laptop = Laptop("Swift")

my_laptop.ram = 8
Laptop.brand = "Dell"
del my_laptop.model
```

Instance vs Class Variables

- **Instance variables** are independent from one instance to the other
- Modifying a **class variable** affects all object instances at the same time

```
class Dog:
    tricks = []     # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

poodle = Dog("Bella")
beagle = Dog("Max")
poodle.tricks.append('roll over')
print(beagle.tricks) # shared by all dogs ['roll over']
```

Example: Good Practice

```
class Dog:
    kind = 'canine' # class variable shared by all instances

    def __init__(self, name):
        self.name = name
        self.tricks = []   # creates empty list for each dog

poodle = Dog("Bella")
beagle = Dog("Max")
print(poodle.name, poodle.kind) # Bella canine
print(beagle.name, beagle.kind) # Max canine
poodle.tricks.append('roll over')
beagle.tricks.append('play dead')
print(poodle.tricks)              # ['roll over']
print(beagle.tricks)              # ['play dead']
```

Problem: Circle

- 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

```
circle = Circle(10)
circle.set_radius(12)
print(circle.get_area())
print(circle.get_circumference())
```

```
452.16
75.36
```

Solution: Circle

```
class Circle:
    pi = 3.14

    def __init__(self, radius):
        self.radius = radius

    def set_radius(self, new_radius):
        self.radius = new_radius

    def get_area(self):
        return Circle.pi * self.radius ** 2

    def get_circumference(self):
        return 2 * Circle.pi * self.radius
```

Problem: Glass

- 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

```
glass = Glass()
print(glass.fill(100))
print(glass.fill(200))
print(glass.empty())
print(glass.fill(200))
print(glass.info())
```

```
Glass filled with 100 ml
Cannot add 200 ml
Glass is now empty
Glass filled with 200 ml
50 ml left
```

Solution: Glass

```
class Glass:
    capacity = 250

    def __init__(self):
        self.content = 0

    def fill(self, ml):
        if self.content + ml <= Glass.capacity:
            self.content += ml
            return f"Glass filled with {ml} ml"
        return f"Cannot add {ml} ml"

    def empty(self):
        self.content = 0
        return "Glass is now empty"

    def info(self):
        return f"{Glass.capacity - self.content} ml left"
```

Special Data Attributes

the __doc__ Attribute

- Provides **documentation** of the object as a **string**

```
class MyClass:
    """This is MyClass."""

    def example(self):
        """This is the example module of MyClass."""

print(MyClass.__doc__) # This is MyClass.
print(MyClass.example.__doc__)
# This is the example module of MyClass.
```

the __dict__ Attribute

- This is a dictionary containing a **module's symbol table**

```
class MyClass:
    class_variable = 1

    def __init__(self, instance_variable):
        self.instance_variable = instance_variable

first = MyClass(2)
second = MyClass(3)

print(MyClass.__dict__) # {'__module__': '__main__', ... }
print(first.__dict__)   # { 'instance_variable': 2 }
print(second.__dict__)  # { 'instance_variable': 3 }
```

Problem: Smartphone

- 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

```
smartphone = Smartphone(100)
print(smartphone.install("Facebook", 60))
smartphone.power()
print(smartphone.install("Facebook", 60))
print(smartphone.install("Messenger", 20))
print(smartphone.install("Instagram", 40))
print(smartphone.status())
```

```
Turn on your phone to install
Facebook
Installing Facebook
Installing Messenger
Not enough memory to install
Instagram
Total apps: 2. Memory left: 20
```

Solution: Smartphone

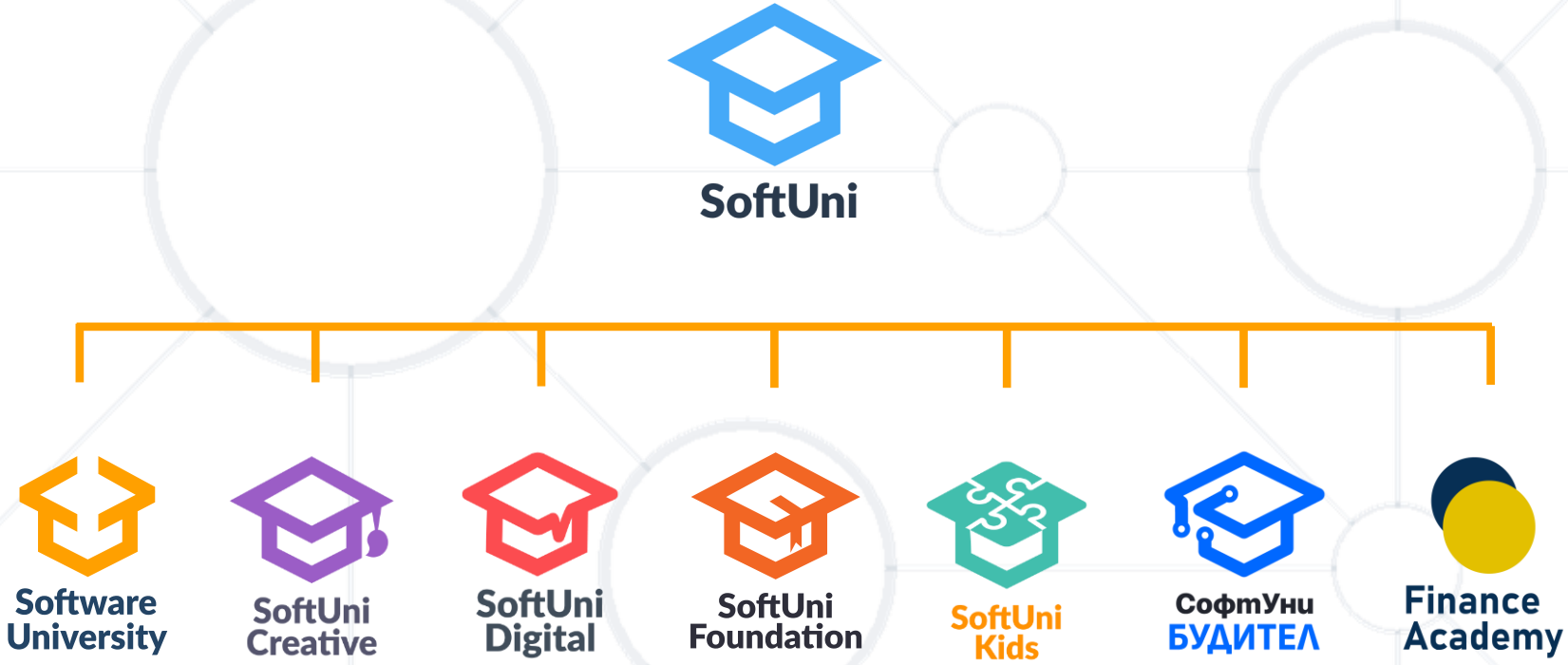
```
class Smartphone:
    def __init__(self, memory):
        # TODO: create memory, apps, and is_on attributes

    def power(self):
        self.is_on = not self.is_on
    def install(self, app_name, memory):
        if self.memory - memory <= 0:
            return f"Not enough memory to install {app_name}"
        if not self.is_on:
            return f"Turn on your phone to install {app_name}"
        # TODO: add the new app and decrease the memory
        return f"Installing {app_name}"
    def status(self):
        return f"Total apps: {len(self.apps)}. Memory left: {self.memory}"
```

Summary

- **Instance objects** are individual objects of a class
- **Methods** are functions that belong to an object
- **Instance variables** are unique to each instance
- **Class Variables** are shared by all instances

Questions?



SoftUni Diamond 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

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