02a_python_object

May 9, 2025

1 Python for Actuaries Part 2

1.1 Agenda

In this notebook, we will cover: - Object-oriented programming - Creating objects - Attributes and functions of objects - Inheritance of objects

So far, we have been programming procedurally; now we want to program in an object-oriented way. An object is an instance of a class. A **class** is a template or blueprint for objects. It defines what attributes (data) and methods (functions) an object will have. Once we have defined a class, we can create as many objects of that class as we want.

You can think of classes somewhat like Plato's Ideas. In the class, we describe the properties (*attributes*) that all objects of this class will have and specify what can be done with the objects (through the *methods*).

A simple example from the real world would be a class Car:

```
+-------------------------+
|        Car              |
+-------------------------+
| - brand: str            |
| - model: str           |
| - color: str            |         -> Attributes
| - mileage: int          |
+-------------------------+
| + drive(km: int): void  |
| + honk(): void          |         -> Methods
+-------------------------+
```

But let's take a look at an example from the insurance industry:

[ ]:
```python
class InsuranceContract:
    def __init__(self, contract_number, insured_sum, holder): # sog. Konstruktor mit Attributen
        self.contract_number = contract_number
        self.insured_sum = insured_sum
        self.holder = holder

    def calculate_premium(self):
        # very simple premium calculation
```

```
        return self.insured_sum * 0.05
```

Now we can create insurance contracts:

[ ]:
```
# create a new contract
contract1 = InsuranceContract('12345', 100000, 'John Doe')
contract2 = InsuranceContract('12346', 200000, 'Jane Smith')

# accessing attributes
print(f"Contract number: {contract1.contract_number}")
print(f"Insured sum: {contract1.insured_sum}")
print(f"Policyholder: {contract1.holder}")

# calculating premium
praemie1 = contract1.calculate_premium()
print(f"The calculated premium is: {praemie1:.2f} euros")

praemie2 = contract2.calculate_premium()
print(f"The calculated premium is: {praemie2:.2f} euros")
```

Through object-oriented programming, we can better organize and reuse our code. Especially in the insurance industry, where many similar contracts, policies, or claims need to be managed, we can benefit from OOP. Each contract, claim, or customer can be represented as an object, which simplifies management and calculation.

1.1.1 Side Note

In Python, everything (EVERYTHING!) is an object. Thus, the base classes we have learned about so far are also objects:

[ ]:
```
def foo():
    return "bar"


print(type(42))
print(type("Hallo"))
print(type(foo))
```

Since everything is an object, you can work with all elements in a similar way in Python. Whether you are working with numbers, texts, or complex structures, each object has certain properties (attributes) and functions (methods) that you can use. We'll get to that shortly, but first:

1.2 Task: Create Class `InsuredPerson`

Create a class `InsuredPerson` that serves to store and display basic information about an insured individual.

1.2.1 Requirements:

1. The class `InsuredPerson` should have the following attributes:

- `name`: The name of the insured person (e.g., “Anna Müller”)
- `date_of_birth`: The date of birth of the insured person (e.g., “01.01.1980”)
- `address`: The address of the insured person (e.g., “Musterstraße 1, 12345 Musterstadt”)
- `insurance_number`: A unique insurance number (e.g., “VN123456”)

2. The class should have a **constructor** (`__init__`) that assigns these attributes when creating an object.
3. The class should have a method `show_contract_details()` that outputs the information about the insured person in a clear format.

1.2.2 Example:

When you create an object of the class `InsuredPerson` with the data of an insured individual, calling the method `show_contract_details()` should produce the following output:

```
Insured Person: Anna Müller
Date of Birth: 01.01.1980
Address: Musterstraße 1, 12345 Musterstadt
Insurance Number: VN123456
```

[1]:
```python
class Policyholder:
    def __init__(self, name, date_of_birth, address, policy_number):
        self.name = name
        self.date_of_birth = date_of_birth
        self.address = address
        self.policy_number = policy_number

    def display_contract_details(self):
        print(f"Policyholder: {self.name}")
        print(f"Date of Birth: {self.date_of_birth}")
        print(f"Address: {self.address}")
        print(f"Policy Number: {self.policy_number}")
```

Here you can test your code:

[2]:
```python
# Create a policyholder object
customer1 = Policyholder("Max Mustermann", "15.05.1975", "Example Street 5,␣
 ↪54321 Example City", "PN987654")

# Call the method to display contract details
customer1.display_contract_details()

print("\n---\n")  # Separator for better readability

# Test another object
customer2 = Policyholder("Julia Meier", "22.08.1990", "Sample Street 12, 65432␣
 ↪Sample City", "PN123321")

# Display contract details for the second object
```

```
customer2.display_contract_details()
```

```
Policyholder: Max Mustermann
Date of Birth: 15.05.1975
Address: Example Street 5, 54321 Example City
Policy Number: PN987654

---

Policyholder: Julia Meier
Date of Birth: 22.08.1990
Address: Sample Street 12, 65432 Sample City
Policy Number: PN123321
```

1.3 Introduction to Inheritance

With object-oriented programming, we can already modularize and make large parts of our code reusable by working with classes and objects. By encapsulating properties and behaviors in a class, we can avoid redundant code fragments and improve maintainability.

But object-oriented programming offers even more: **Inheritance** is a central concept that allows us to link classes together and establish relationships between them. With inheritance, we can define a general class from which specialized classes can be derived. This not only reduces effort but also creates a clearer structure.

1.3.1 Advantages of Inheritance:

- **Code Reusability:** Common functionalities only need to be defined once in a base class. The derived classes automatically inherit this functionality.
- **Extensibility:** New specialized classes can be easily created by inheriting from an existing class and adding additional functions or attributes as needed.
- **Maintainability:** A clearly structured inheritance hierarchy makes the code more organized and easier to maintain.

1.3.2 Example: Inheritance with Policyholders

Let's imagine we want to manage not only policyholders in our system but also other types of individuals, such as brokers or agents. All have common attributes (e.g., name, date of birth, address), but also specific differences. Instead of redefining these properties and methods in each class, we can create a general class `Person` from which all the more specialized classes inherit.

Let's try this out:

[ ]:
```
class Person():
    pass
```

Now we want the class `Policyholder` to inherit from the class `Person`:

[ ]:
```
class Policyholder(Person): # that's inheritance
```

1.4 Special Class Methods

We have already seen that there are special methods for classes in Python. In principle, every class should implement the following standard methods:

```
__init__() -> Constructor
__repr__() -> a (technical) string representation of the object
__str__() -> a (human-friendly) string representation of the object
__eq__() -> allows two objects of the class to be checked for equality with "=="
__hash__() -> returns a unique hash value for each instance
```

Other standard methods (which do not make sense for people) are:

```
__add__() -> add/concatenate two objects with "+"
__subtract__() -> subtract two objects with "-"
__getitem__(idx) -> index access returns the value at idx
__setitem__(idx, value) -> sets the element at index idx to the value value
__len__() -> returns the length of the object
__del__() -> deletes the instance of the object
__iter__() -> for iterating over the object
```

We will implement the first ones:

[ ]:
```python
class Person:
    def __init__(self, name, birth_date, adress):
        self.name = name
        self.birth_date = birth_date
        self.adress = adress

    def show_details(self):
        print(f"Name: {self.name}")
        print(f"Birthday: {self.birth_date}")
        print(f"Adress: {self.adress}")

    def __str__(self) -> str:
        return f"Name: {self.name}, Birthday: {self.birth_date}, Adress: {self.
↪adress}"

    def __repr__(self) -> str:
        return f"Person(name={self.name}, birth_date={self.birth_date},␣
↪adress={self.adress})"

    def __eq__(self, value: object) -> bool:
        return isinstance(value, Person) and self.name == value.name and self.
↪birth_date == value.birth_date and self.adress == value.adress

    def __hash__(self) -> int:
        return hash((self.name, self.birth_date, self.adress))
```

1.5 Task (Shapes)

Create a class `shape` that stores the name of the shape as an attribute and has a method `area` that returns the string "not yet implemented".
Also, implement the functions `__repr__()` and `__str__()`.

[4]:
```
class Shape:
    def __init__(self, name):
        self.name = name
    def area(self):
        raise NotImplementedError("This method should be overridden in␣
 ↪subclasses")

    # Methode area

    # repr
    def __repr__(self) -> str:
        return f"Shape(name={self.name})"
    # str
    def __str__(self) -> str:
        return f"Shape: {self.name}"
```

Circle

Create a subclass `Circle` that inherits from `Shape`. Implement the constructor that sets the name of the shape to "Circle" and stores the radius. Override the method `area` to calculate the area of the circle. Also, implement a method `__eq__()` to compare two circles.

Note: `math.pi` provides the value of π.

[5]:
```
import math

class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    def area(self):
        return math.pi * (self.radius ** 2)

    def __eq__(self, otherCircle):
        return isinstance(otherCircle, Circle) and self.radius == otherCircle.
 ↪radius
```

[6]:
```
# Example usage:
circle = Circle(5)
print(circle.name)
print(circle.area())  # output: 78.53981633974483
circle2 = Circle(6)
```

```
circle3 = Circle(5)
print(circle == circle2)
print(circle == circle3)
```

```
Circle
78.53981633974483
False
True
```

Rectangle

Create a subclass Rectangle that inherits from Shape. Implement the constructor that sets the name and stores the width and height. Override the method `area` to calculate the area of the rectangle.

[ ]:
```
class Rectangle(Shape):
    def __init__(self, width, height):
        super().__init__("Rectangle")
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def __eq__(self, otherRectangle):
        return isinstance(otherRectangle, Rectangle) and self.width ==␣
↪otherRectangle.width and self.height == otherRectangle.height
```

[11]:
```
# Beispiel:
rectangle = Rectangle(3, 5)
print(rectangle.name)  # output: Rectangle
print(rectangle.area())  # output: 15
print(repr(rectangle))  # output: Rectangle('Rectangle')
print(str(rectangle))   # output: RECTANGLE
```

```
Rectangle
15
Shape(name=Rectangle)
Shape: Rectangle
```

Square

Create a subclass `Square` that inherits from `Rectangle`. Implement the constructor to store the side length and the name, and override the method `area` to calculate the area of the square. Also, create a method `__eq__()` to test two squares for equality.

[17]:
```
class Square(Rectangle):
    def __init__(self, side_length):
        super().__init__(side_length, side_length)
        self.name = "Square"
```

[18]:
```python
# Beispiel:
square = Square(4)
print(square.name)  # output: Square
print(square.area())  # output: 16

print(square == Square(3)) # False
print(square == Square(4)) # True
```

```
Square
16
Test
False
Test
True
```