Inheritance, Encapsulation and Polymorphism

We hebben de modelleringskracht van OOP al gezien door de klasse en objectfuncties te gebruiken door gegevens en methoden te combineren. Er zijn nog drie belangrijke concepten, **inheritance**, die de OOP-code meer modulair maakt, gemakkelijker te hergebruiken en een relatie op te bouwen tussen klassen. **Encapsulatie** kan sommige privé-details van een klasse verbergen voor andere objecten, terwijl **polymorfisme** ons in staat kan stellen een gemeenschappelijke operatie op verschillende manieren te gebruiken. In deze sectie zullen we ze kort bespreken.

Inheritance

Met overerving kunnen we een klasse definiëren die alle methoden en attributen erft van een andere klasse. De conventie noemt de nieuwe klasse **kindklasse**, en de klasse waarvan ze erft heet **ouderklasse** of **superklasse**. Als we teruggaan naar de definitie van klassenstructuur, zien we dat de structuur voor basisovererving **klasseNaam(superklasse)** is, wat betekent dat de nieuwe klasse toegang heeft tot alle attributen en methoden van de superklasse. Inheritance bouwt een relatie op tussen de child class en de parent class, meestal op een manier dat de parent class een algemeen type is, terwijl de child class een specifiek type is. Laten we een voorbeeld proberen.

TRY IT! Definieer een klasse genaamd `Sensor` met attributen `naam`, `locatie`, en `record_datum`, die doorgegeven worden bij de creatie van een object en een attribuut `data` als een leeg woordenboek om gegevens in op te slaan. Maak een methode *add_data* met `t` en `data` als invoerparameters om timestamp en data arrays in te nemen. Wijs in deze methode `t` en `data` toe aan het `data` attribuut met `tijd` en `data` als sleutels. Bovendien moet het een `clear_data` methode hebben om de gegevens te verwijderen.

```
class Sensor():
    def __init__(self, name, location, record_date):
        self.name = name
        self.location = location
        self.record_date = record_date
        self.data = {}

    def add_data(self, t, data):
        self.data['time'] = t
        self.data['data'] = data
        print(f'We have {len(data)} points saved')

    def clear_data(self):
        self.data = {}
        print('Data cleared!')
```

Now we have a class to store general sensor information, we can create a sensor object to store some data.

EXAMPLE: Create a sensor object.

```
import numpy as np

sensor1 = Sensor('sensor1', 'Berkeley', '2019-01-01')
data = np.random.randint(-10, 10, 10)
sensor1.add_data(np.arange(10), data)
sensor1.data
```

Inherit and extend new method

Stel dat we een ander type sensor hebben: een versnellingsmeter. Deze deelt dezelfde attributen en methoden als de klasse `Sensor`, maar heeft ook andere attributen of methoden die moeten worden toegevoegd of gewijzigd dan de oorspronkelijke klasse. Wat moeten we doen? Maken we een andere klasse vanaf nul? Dit is waar overerving kan worden gebruikt om het leven gemakkelijker te maken. Deze nieuwe klasse erft van de klasse `Sensor` met alle attributen en methoden. We kunnen zelf bepalen of we de attributen of methoden willen uitbreiden. Laten we eerst deze nieuwe klasse, `Accelerometer`, maken en een nieuwe methode, `show_type`, toevoegen om te melden wat voor soort sensor het is.

```
class Accelerometer(Sensor):

    def show_type(self):
        print('I am an accelerometer!')

acc = Accelerometer('acc1', 'Oakland', '2019-02-01')
acc.show_type()
data = np.random.randint(-10, 10, 10)
acc.add_data(np.arange(10), data)
acc.data
```

Het maken van deze nieuwe klasse `Accelerometer` is heel eenvoudig. We erven van `Sensor` (aangeduid als superklasse), en de nieuwe klasse bevat in feite alle attributen en methoden van de superklasse. We voegen dan een nieuwe methode toe, `show_type`, die niet bestaat in de `Sensor` klasse, maar we kunnen met succes de child klasse uitbreiden door de nieuwe methode toe te voegen. Dit toont de kracht van overerving: we hebben het grootste deel van de `Sensor` klasse hergebruikt in een nieuwe klasse, en de functionaliteit uitgebreid. Bovendien zorgt de overerving voor een logische relatie voor het modelleren van de echte entiteiten: de klasse `Sensor` als ouderklasse is algemener en geeft alle eigenschappen door aan de kindklasse `Accelerometer`.

Inherit and method overriding

Wanneer we erven van een ouderklasse, kunnen we de implementatie van een methode van de ouderklasse wijzigen, dit heet method overriding. Laten we het volgende voorbeeld bekijken.

Voorbeeld: Maak een klasse `UCBAcc` (een specifiek type versnellingsmeter dat gemaakt is bij UC Berkeley) die erft van `Accelerometer` maar vervang de `show_type` methode die de naam van de sensor uitprint.

```
class UCBAcc(Accelerometer):

    def show_type(self):
        print(f'I am {self.name}, created at UC Berkeley!')

acc_ucb = UCBAcc('UCBAcc', 'Berkeley', '2019-03-01')
acc_ucb.show_type()
```

We zien dat onze nieuwe klasse `UCBAcc` de methode `show_type` overschrijft met nieuwe eigenschappen. In dit voorbeeld erven we niet alleen kenmerken van onze bovenliggende klasse, maar we wijzigen/verbeteren ook enkele methoden.

Inherit en update attributes met super

Laten we een klasse `NieuweSensor` maken die erft van de klasse `Sensor`, maar met een update van de attributen door toevoeging van een nieuw attribuut `merk`. Natuurlijk kunnen we de hele `__init__` methode opnieuw definiëren zoals hieronder en de functie van de ouder overschrijven.

```
class NewSensor(Sensor):
    def __init__(self, name, location, record_date, brand):
        self.name = name
        self.location = location
        self.record_date = record_date
        self.brand = brand
        self.data = {}

new_sensor = NewSensor('OK', 'SF', '2019-03-01', 'XYZ')
new_sensor.brand
```

Er is echter een betere manier om hetzelfde te bereiken. We kunnen de `super` methode gebruiken om niet expliciet naar de bovenliggende klasse te verwijzen. Laten we in het volgende voorbeeld zien hoe we dit kunnen doen:

Voorbeeld: Herdefinieer de attributen in de overerving.

```
class NewSensor(Sensor):
    def __init__(self, name, location, record_date, brand):
        super().__init__(name, location, record_date)
        self.brand = brand
```

```
new_sensor = NewSensor('OK', 'SF', '2019-03-01', 'XYZ')
new_sensor.brand
```

Nu zien we dat we met de *super* methode vermijden om alle definitie van de attributen op te sommen, dit helpt om je code onderhoudbaar te houden voor de nabije toekomst. Maar het is echt nuttig wanneer je meervoudige overerving doet, wat buiten de discussie van dit boek valt.

Encapsulation

Encapsulatie is een van de fundamentele concepten in OOP. Het beschrijft het idee om de toegang tot methoden en attributen in een klasse te beperken. Dit verbergt de complexe details voor de gebruikers, en voorkomt dat gegevens per ongeluk worden gewijzigd. In Python wordt dit bereikt door private methoden of attributen te gebruiken met underscore als voorvoegsel, dus enkelvoudig "_" of dubbel "__". Laten we het volgende voorbeeld bekijken.

**VOORBEELD

```
class Sensor():
    def __init__(self, name, location):
        self.name = name
        self._location = location
        self.__version = '1.0'

    # a getter function
    def get_version(self):
        print(f'The sensor version is {self.__version}')

    # a setter function
    def set_version(self, version):
        self.__version = version

sensor1 = Sensor('Acc', 'Berkeley')
print(sensor1.name)
print(sensor1._location)
print(sensor1.__version)
```

Het bovenstaande voorbeeld laat zien hoe de inkapseling werkt. Met een enkele underscore hebben we een private variabele gedefinieerd, die niet direct benaderd mag worden. Maar dit is slechts conventie, niets houdt je tegen om dat te doen. Je kunt er nog steeds toegang toe krijgen als je dat wilt. Met dubbele underscore kunnen we zien dat het attribuut `__version` niet direct toegankelijk is of gewijzigd kan worden. Daarom moeten we, om toegang te krijgen tot de double underscore attributen, getter en setter functies gebruiken om er intern toegang toe te krijgen, zoals in het volgende voorbeeld.

```
sensor1.get_version()
```

```
sensor1.set_version('2.0')
sensor1.get_version()
```

De enkele en dubbele underscore zijn ook van toepassing op private methoden, we zullen deze niet bespreken omdat ze vergelijkbaar zijn met de private attributen.

Polymorphism

Polymorfisme is een ander fundamenteel concept in OOP, dat meerdere vormen betekent. Polymorfisme stelt ons in staat een enkele interface te gebruiken met verschillende onderliggende vormen, zoals datatypes of klassen. We kunnen bijvoorbeeld gelijknamige methoden hebben in verschillende klassen of kind-klassen. We hebben hierboven al een voorbeeld gezien, wanneer we de methode `show_type` in de `UCBAcc` overschrijven. De parent class `Accelerometer` en de child class `UCBAcc` hebben beide een methode genaamd `show_type`, maar ze hebben een verschillende implementatie. Deze mogelijkheid om één enkele naam te gebruiken met vele vormen die in verschillende situaties anders werken, vermindert onze complexiteit enorm. We zullen niet verder ingaan op polymorfisme, als je geïnteresseerd bent, kijk dan online voor een beter begrip.