# OOP: Object georiënteerd programmeren

Objectgeoriënteerd programmeren (OOP: Object Oriented Programming) is meestal een van de grootste obstakels voor beginners wanneer ze voor het eerst Python beginnen te leren.

Er zijn heel veel tutorials en lessen over OOP, dus voel je vrij om op Google te zoeken naar andere lessen, en ik heb ook enkele links naar andere nuttige tutorials online onder aan dit notitieblok geplaatst.

Voor deze les bouwen we onze kennis van OOP in Python op door voort te bouwen op de volgende onderwerpen:

- Voorwerpen
- Gebruik het *class* trefwoord
- Klassenattributen maken
- Methoden maken in een klasse
- Leren over inheritance (erfelijkheid)
- Leren over polymorfisme
- Leren over speciale methoden voor lessen

Laten we de les beginnen door te onthouden over de basis python-objecten. Bijvoorbeeld:

```
lst = [1,2,3]
```

Weet je nog hoe we methoden op een lijst konden aanroepen?

```
lst.count(2)
```

```
1
```

Wat we in deze lezing gaan doen, is onderzoeken hoe we een objecttype zoals een lijst kunnen maken. We hebben al geleerd hoe u functies maakt. Laten we objecten in het algemeen onderzoeken:

## Voorwerpen

In Python is *alles een object*. Onthoud uit eerdere lezingen dat we type() kunnen gebruiken om te controleren welk type object iets is:

```
print(type(1))
print(type([]))
print(type(()))
print(type({}))
```

```
<class 'int'>
<class 'list'>
<class 'tuple'>
```

```
<class 'dict'>
```

Dus we weten dat al deze dingen objecten zijn, dus hoe kunnen we onze eigen objecttypes maken?

## class

De gebruiker gedefinieerde (user-defined) objecten worden gemaakt met het trefwoord class. De klasse is een blauwdruk/sjabloon die de aard van een toekomstig object definieert. Van klassen kunnen we instanties construeren. Een instantie is een specifiek object dat is gemaakt op basis van een bepaalde klasse. Hierboven hebben we bijvoorbeeld het object lst gemaakt, dat een instantie was van een list-object.

Laten we eens kijken hoe we class kunnen gebruiken:

```
# Maak een nieuw objecttype aan met de naam Sample
class Sample:
    pass

# Een instantie van de klas Sample
x = Sample()

print(type(x))
```

```
<class '__main__.Sample'>
```

Volgens conventies geven we klassen een naam die begint met een **hoofdletter**. Merk op hoe x nu de referentie is naar onze nieuwe instantie van een voorbeeldklasse. Met andere woorden, we **instantiëren** de Sample-klasse.

Binnen in de klas hebben we momenteel alleen pas. Maar we kunnen klassenattributen en -methoden definiëren.

Een **attribuut** is een kenmerk (characteristic) van een object.

Een **methode** is een bewerking/operatie die we met het object kunnen uitvoeren.

We kunnen bijvoorbeeld een klasse maken met de naam Dog. Een kenmerk van een hond kan zijn ras of zijn naam zijn, terwijl een methode van een hond kan worden gedefinieerd door een .bark() -methode die een geluid retourneert.

Laten we aan de hand van een voorbeeld een beter begrip van attributen krijgen.

## Attributen

De syntaxis voor het maken van een attribuut is:

```
self.attribute = something
```

Er is een speciale methode genaamd:

`__init__()`

Deze methode wordt gebruikt om de attributen van een object te **initialiseren**. Bijvoorbeeld:

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

sam = Dog(breed='Lab')
frank = Dog(breed='Huskie')
```

Laten we opsplitsen wat we hierboven hebben. De speciale methode:

`__init__()`

wordt automatisch aangeroepen direct nadat het object is gemaakt:

`def __init__(self, breed):`

Elk attribuut in een klassendefinitie begint met een verwijzing naar het instantieobject. Het wordt volgens afspraak zelf genoemd. Het ras is het argument. De waarde wordt doorgegeven tijdens de instantie van de klasse.

`self.breed = breed`

Nu hebben we twee instanties van de klasse Dog gemaakt. Met twee rassen (breed) hebben we dan als volgt toegang tot deze attributen:

`sam.breed`

`'Lab'`

`frank.breed`

`'Huskie'`

Merk op dat we geen haakjes hebben na 'breed'; dit komt omdat het een **attribuut** is en er **geen argumenten** voor nodig zijn.

In Python zijn er ook *Class Object Attributes*. Deze klassenobjectkenmerken zijn hetzelfde voor elk exemplaar van de klasse. We kunnen bijvoorbeeld het attribuut *species* maken voor de klasse 'Dog'. Honden, ongeacht hun ras, naam of andere eigenschappen, zullen altijd zoogdieren (mammals) zijn. We passen deze logica op de volgende manier toe:

```
class Dog:

    # Class Object Attribute
    species = 'mammal'

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

```
sam = Dog('Lab','Sam')
```

```
sam.name
```

```
'Sam'
```

Merk op dat het class-object-attribuut buiten alle methoden in de klasse wordt gedefinieerd. Ook volgens conventies plaatsen we ze eerst voor de init.

```
sam.species
```

```
'mammal'
```

## Methoden

Methoden zijn **functies** die zijn gedefinieerd in de **body** van een klasse. Ze worden gebruikt om operaties uit te voeren met de attributen van onze objecten. Methoden zijn een sleutelconcept van het OOP-paradigma. Ze zijn essentieel voor het verdelen van verantwoordelijkheden bij het programmeren, vooral bij grote toepassingen.

U kunt methoden zien als functies die op een object inwerken (acting) en waarbij rekening wordt gehouden met het object zelf via het *self*-argument.

Laten we een voorbeeld bekijken van het maken van een Circle-klasse:

```
class Circle:
    pi = 3.14

    # Cirkel wordt geïnstantieerd met een straal (radius) (standaard is 1)
    def __init__(self, radius=1):
        self.radius = radius
        self.area = radius * radius * Circle.pi

    # Methode voor het resetten van Radius
    def setRadius(self, new_radius):
        self.radius = new_radius
        self.area = new_radius * new_radius * self.pi

    # Methode voor het verkrijgen van omtrek (circumference)
    def getCircumference(self):
        return self.radius * self.pi * 2


c = Circle()

print('Radius is: ', c.radius)
print('Area is: ', c.area)
print('Circumference is: ', c.getCircumference())
```

```
Radius is:  1
Area is:  3.14
Circumference is:  6.28
```

In de ___init___ methode hierboven moesten we Circle.pi aanroepen om het oppervlakteattribuut (area) te berekenen. Dit komt omdat het object nog geen eigen .pi-attribuut heeft, dus we noemen het Class Object Attribute pi in plaats daarvan. In de setRadius-methode werken we echter met een bestaand Circle-object dat wel zijn eigen pi-attribuut heeft. Hier kunnen we Circle.pi of self.pi gebruiken. Laten we nu de straal (radius) veranderen en kijken hoe dat ons Circle-object beïnvloedt (affect):

```
c.setRadius(2)

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
```

```
Radius is:  2
Area is:  12.56
Circumference is:  12.56
```

Geweldig! Merk op hoe we hiervoor 'self'. notatie gebruikten om te verwijzen naar attributen van de klasse binnen de methode-aanroepen. Bekijk hoe de bovenstaande code werkt en probeer uw eigen methode te maken.

## Inheritance (Overerving)

Overerving is een manier om nieuwe klassen te vormen met behulp van klassen die al zijn gedefinieerd. De nieuw gevormde klassen worden **afgeleide** (derived-class) klassen genoemd, de klassen waaruit we afleiden worden **basisklassen** (base-class) genoemd. Belangrijke voordelen van overerving zijn **codehergebruik** en **vermindering van de complexiteit** van een programma. De afgeleide klassen (descendants) overschrijven of breiden de functionaliteit van basisklassen (ancestors) uit.

Laten we een voorbeeld bekijken door ons eerdere werk over de Dog-klasse op te nemen:

```
class Animal:
    def __init__(self):
        print("Animal created")

    def whoAmI(self):
        print("Animal")

    def eat(self):
        print("Eating")
```

```
class Dog(Animal):
    def __init__(self):
        Animal.__init__(self)
        print("Dog created")

    def whoAmI(self):
        print("Dog")

    def bark(self):
        print("Woof!")
```

```
d = Dog()
```

```
Animal created
Dog created
```

```
d.whoAmI()
```

```
Dog
```

```
d.eat()
```

```
Eating
```

```
d.bark()
```

```
Woof!
```

In dit voorbeeld hebben we 2 klassen: Animal en Dog. Animal is de basisklasse (base), Dog is de afgeleide (derived) klasse.

De afgeleide klasse erft de functionaliteit van de basisklasse.

- Het wordt getoond door de eat() methode.

De afgeleide klasse wijzigt het bestaande gedrag (behaviour) van de basisklasse.

- getoond door de whoAmI() methode.

Ten slotte breidt de afgeleide klasse de functionaliteit van de basisklasse uit door een nieuwe bark()-methode te definiëren.

## Polymorfisme

We hebben geleerd dat hoewel functies verschillende argumenten kunnen bevatten, methoden behoren tot de objecten waarop ze werken. In Python verwijst *polymorfisme* naar de manier waarop verschillende objectklassen dezelfde methodenaam kunnen delen, en die methoden kunnen vanaf dezelfde plaats worden aangeroepen, ook al kunnen er verschillende objecten worden ingevoerd. De beste manier om dit uit te leggen is bij voorbeeld:

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

    def speak(self):
        return self.name+' says Woof!'

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

    def speak(self):
        return self.name+' says Meow!'

niko = Dog('Niko')
felix = Cat('Felix')

print(niko.speak())
print(felix.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Hier hebben we een Dog-klasse en een Cat-klasse, en elk heeft een `.speak()`-methode. Wanneer **aangeroepen**, retourneert de `.speak()`-methode van elk object een resultaat dat **uniek** is voor het object.

Er zijn een paar verschillende manieren om polymorfisme aan te tonen. Eerst met een for-lus:

```
for pet in [niko,felix]:
    print(pet.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Een andere is met functies:

```
def pet_speak(pet):
    print(pet.speak())

pet_speak(niko)
pet_speak(felix)
```

```
Niko says Woof!
Felix says Meow!
```

In beide gevallen konden we verschillende objecttypes doorgeven en kregen we objectspecifieke resultaten van hetzelfde mechanisme.

Een meer gebruikelijke (common) praktijk is om abstracte klassen en overerving te gebruiken. Een abstracte klasse is er een die **nooit verwacht** te worden **geïnstantieerd**. We zullen bijvoorbeeld nooit een Animal-object hebben, alleen Dog- en Cat-objecten, hoewel Dogs and Cats zijn afgeleid van Animals:

```
class Animal:
    def __init__(self, name):    # Constructeur van de klas
        self.name = name

    def speak(self):             # Abstracte methode, alleen gedefinieerd door conventie
        raise NotImplementedError("Subclass must implement abstract method")


class Dog(Animal):

    def speak(self):
        return self.name+' says Woof!'

class Cat(Animal):

    def speak(self):
        return self.name+' says Meow!'

fido = Dog('Fido')
isis = Cat('Isis')

print(fido.speak())
print(isis.speak())
```

```
Fido says Woof!
Isis says Meow!
```

Voorbeelden van polymorfisme uit het levensecht-applicaties zijn als de volgende: * Voor het openen van de verschillende bestandstypen: Er zijn verschillende tools nodig om Word-, pdf- en Excel-bestanden weer te geven * Voor het toevoeging van de verschillende objecten: De operator + voert rekenkundige bewerkingen (arithmetic) en aaneenschakelingen (concatenation) uit

## Speciale methoden met ___ methode-naam ___

Laten we tot slot speciale methoden bespreken. Klassen in Python kunnen bepaalde bewerkingen uitvoeren met speciale methodenamen. Deze methoden worden niet rechtstreeks aangeroepen, maar door Python-specifieke taal-syntaxis. Laten we bijvoorbeeld een Boek-klas maken:

```
class Book:
    def __init__(self, title, author, pages):
        print("A book is created")
```

```
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return "Title: %s, author: %s, pages: %s" %(self.title, self.author, self.pages)

    def __len__(self):
        return self.pages

    def __del__(self):
        print("A book is destroyed")
book = Book("Python Rocks!", "Jose Portilla", 159)

# Speciale methoden
print(book)
print(len(book))
del book
```

```
A book is created
Title: Python Rocks!, author: Jose Portilla, pages: 159
159
A book is destroyed
```

De methoden **init**(), **str**(), **len**() en **del**()

Deze speciale methoden worden gedefinieerd door het gebruik van **onderstrepingstekens** (underscore). Ze stellen ons in staat om Python-specifieke functies te gebruiken op objecten die met onze klasse zijn gemaakt.

**Geweldig! Na deze lezing zou u een basiskennis moeten hebben van hoe u uw eigen objecten kunt maken met klasse in Python. U zult hier intensief gebruik van maken in uw volgende Milestone-Project!**

Ga voor meer geweldige bronnen (resources) over dit onderwerp naar:

Mozilla's bericht

Zelfstudie-portaal

Officiële documentatie