Python OOP / Development

Objecten en klassen in Python

Kristof Michiels

Inhoud

- Wat zijn objecten en hoe maken we ze?
- Overerving
- Toegang tot attributen
- Soorten methods

Object-georiënteerd programmeren (OOP)?

- OOP is een werkwijze om de structuur logisch en overzichtelijk te houden
- Gebruik van klassen maakt het mogelijk om honderden variabelen en functies overzichtelijk in georganiseerde en van heldere naamgeving voorziene structuren onder te brengen
- Een klasse noemen we de blauwdruk/het plan
- Het object noemen we de instance of een afdruk van die blauwdruk

Wat zijn objecten en hoe maken we ze?

Wat zijn objecten?

- In Python is alles een object (string, integer, dictionary, ...). Wordt door Python achter de schermen geregeld
- Je krijgt zelf met objecten te maken wanneer je je eigen objecten wil maken of wanneer je het gedrag van bestaande wil wijzigen
- Onder object verstaan we een (aangepaste) datastructuur die zowel data (variabelen, noemen we attributen) en code (functies, noemen we methods) bevat
- Een object vertegenwoordigt een uniek exemplaar van een concreet ding (een auto, een level, een speler, ...)
- Zie objecten als zelfstandige naamwoorden en hun methoden als werkwoorden. Een object vertegenwoordigt een individueel ding, en zijn methods bepalen hoe het interageert met andere dingen

Klassen maken met class

- Objecten worden geïnstantieerd - tot leven gebracht - uit een klasse-definitie
- Je kan ze beschouwen als blauwdrukken die beschrijven wat een object allemaal bevat
- We gebruiken class om een klasse te beschrijven
- Voor de klassenaam is het in Python gebruikelijk met een hoofdletter te beginnen
- Je creëert een object door de klasse aan te roepen zoals je een functie aanroept

```
class Kunstwerk():
    pass

kunstwerk_a = Kunstwerk()
kunstwerk_b = Kunstwerk()
```

Object-attributen

- Een attribuut is een variabele binnen een klasse of object
- Je kan attributen toevoegen als een object is gecreëerd (zien we in wat volgt), maar ook nog daarna (zoals in onderstaand voorbeeld)
- De waarde van een attribuut kan elk ander object zijn 😉

```
kunstwerk_a.bewaarplaats = "Parijs, Louvre"
kunstwerk_a.gemaakt_in = 1503
kunstwerk_a.hangt_naast = kunstwerk_b
```

Klasse-attributen

- Met attributen worden meestal object-attributen bedoeld
- Er bestaan ook klasse-attributen: deze en hun waarden worden gedeeld door alle geïnstantieerde objecten
- Als je de waarde verandert in een object dan heeft het geen impact op het klasse-attribuut

```
class Kunstwerk:
    soort = "schilderij"

david_van_michelangelo = Kunstwerk()
print(Kunstwerk.soort) # schilderij
print(david_van_michelangelo.soort) # schilderij
david_van_michelangelo.soort = "beeld"
print(david_van_michelangelo.soort) # beeld
print(Kunstwerk.soort) # schilderij
```

Klasse-attributen

- Pas je de waarde van het klasse-attribuut aan dan heeft dit geen effect op de bestaande objecten
- Wel bij nieuwe objecten

```
Kunstwerk.soort = "gravure"
print(Kunstwerk.soort) # gravure
print(david_van_michelangelo.soort) # beeld
mona_lisa_van_leonardo = Kunstwerk()
print(mona_lisa_van_leonardo.soort) # gravure
```

Methods

- Een method is een functie binnen een klasse of object
- Een method ziet eruit als elke andere functie maar kan op meer specifieke manieren worden gebruikt (zien we verder in deze les)

```
class Kunstwerk:
    def vertel(self):
        print("Wat kan ik vertellen? Ik ben een kunstwerk, dat staat vast!")
```

Initialisatie

- Als je object-attributen wil toekennen wanneer een object wordt gecreëerd dan gebruik je de speciale Python object-initialisatie-method __init__()
- btw: de dubbele underscores noemen we dunder ;-) We spreken van dunder-methods
- __init__() initialiseert een individueel object uit de klasse-definitie: het bevat alles om het ene object te onderscheiden van het andere
- De eerste parameter is altijd self: het verwijst naar het individuele object

```
class Kunstwerk:
    def __init__(self):
        pass
```

Initialisatie

- Hier voegen we een parameter naam toe aan de __init__-method
- Wanneer we nu een object instantiëren moeten we een string meegeven voor de parameter "naam"
- Op deze manier worden hier de object-attributen *naam* en *kunstenaar* gecreëerd

```
class Kunstwerk:
    def __init__(self, naam, kunstenaar):
        self.naam = naam
        self.kunstenaar = kunstenaar

mona_lisa_van_leonardo = Kunstwerk("Mona Lisa", "Leonardo da Vinci")
```

Wat gebeurt hier bij de instantiëring?

- De beschrijving van de Kunstwerk klasse wordt opgezocht
- Een nieuw object wordt in het geheugen geïnstantieerd/gecreëerd
- De __init__-method wordt aangeroepen: het nieuwe object wordt als self meegegeven, alsook de andere argumenten naam en kunstenaar
- De waarden van naam en kunstenaar worden opgeslagen in het object
- Het nieuwe object wordt teruggegeven en toegekend aan de variabele mona_lisa_van_leonardo
- Binnen de klasse-definitie verwijs je naar de attributen als self.attribuutnaam
- Het nieuwe object is zoals elk ander object in Python

Het self-argument

Python gebruikt self om de juiste attributen en methods te kunnen vinden

```
class Kunstwerk:
    def vertel(self):
        print("Wat kan ik vertellen? Ik ben een kunstwerk, dat staat vast!")

een_kunstwerk = Kunstwerk()
een_kunstwerk.vertel() # Wat kan ik vertellen? Ik ben een kunstwerk, dat staat vast!
```

Achter de schermen gebeurt het volgende:

```
Kunstwerk.vertel(een_kunstwerk)
```

Overerving

Overerving

- Bij het coden zal je vaak beroep kunnen doen op een bestaande klasse die objecten creëert op een manier die benadert wat je zelf nodig hebt
- Je kan de klasse aanpassen, of een nieuwe creëren met dezelfde code maar dat is niet duurzaam en efficiënt
- Een oplossing is overerving: een nieuwe klasse creëren afgeleid van een andere, maar met enkele toevoegingen of wijzigingen: een goede manier om code te herbruiken

```
class AangepasteFout(Exception):
    pass

raise AangepasteFout
```

Erven van een basis-klasse

- In je afgeleide klasse definieer je enkel wat je wenst toe te voegen of te wijzigen. Dit heft (in het Engels: overrides) het gedrag van de basis-klasse op
- Voor de oorspronkelijke klasse hebben we verschillende namen: parent, superclass of base klasse. Idem voor de afgeleide klasse: child, subclass of derived klasse

```
class Dier():
    pass

class Aap(Dier):
    pass

een_dier = Dier()
een_aapje = Aap()
```

Erven van een basis-klasse

- Een afgeleide klasse is een gespecialiseerde versie van een basis-klasse
- De relatie tussen beiden is er één van "IS EEN": een Aap is-een Dier
- Je kan checken of een klasse een afgeleide klasse is door gebruik van de issubclass()-functie

```
print(issubclass(Aap, Dier)) # True
```

Erven van een basis-klasse

Een afgeleide klasse erft alle functionaliteit van de basis-klasse

```
class Dier():
    def spreek(self):
        print("Ik ben een dier")

class Aap(Dier):
    pass

een_dier = Dier()
een_aapje = Aap()
een_dier.spreek() # Ik ben een dier
een_aapje.spreek() # Ik ben een dier
```

Gedrag opheffen of overriden

We kunnen elke method overriden, ook de __init__()-method

```
class Dier():
    def __init__(self, naam):
        self.naam = naam
    def spreek(self):
        print("Ik ben een dier")

class Aap(Dier):
    def __init__(self, naam):
        self.naam = "Meneer " + naam
    def spreek(self):
        print("Ik ben een aapje")
```

Methods toevoegen aan een afgeleide klasse

Een afgeleide klasse kan ook één of meerdere nieuwe methods toevoegen

```
class Dier():
    def __init__(self, naam):
        self.naam = naam
    def spreek(self):
        print("Ik ben een dier")

class Aap(Dier):
    def __init__(self, naam):
        self.naam = "Meneer " + naam
    def spreek(self):
        print("Ik ben een aapje")
    def vlooien(self):
        print("Vlooien, zo ontspannend, mmmm")
```

super()

Gebruik super() wanneer de afgeleide klasse iets nieuws doet (bvb een method overschrijft) maar toch iets nodig heeft van de base-klasse

```
class Dier():
    def __init__(self, naam):
        self.naam = naam

class Aap(Dier):
    def __init__(self, naam, verwantschap_mens):
        super().__init__(naam)
        self.verwantschap_mens = verwantschap_mens
```

Meervoudige overerving en de mro

- Objecten kunnen erven van meerdere base-klassen
- Wanneer je klasse verwijst naar een method of attribuut dat de afgeleide klasse niet heeft, dan zal Python zoeken in alle base-klassen
- Python volgt daarbij een "method resolution order"
- Elke klasse heeft een mro()-method die een list teruggeeft met daarin alle klassen - in volgorde - die zullen doorzocht worden naar een method of attribuut. De eerste die wordt gevonden wint
- Er is een gelijkaardig attribuut __mro__ die dezelfde list teruggeeft

Meervoudige overerving en de mro

```
class A():
    def toonA(self):
        print('Je bent in klasse A')

class B():
    def toonB(self):
        print('Je bent in klasse B')

class C(A, B):
    def toonC(self):
        print('Je bent in klasse C')

klasse_c = C()
klasse_c.mro()
```

Mixins

- Je kan zonder problemen een extra basis-klasse toevoegen aan je klasse-beschrijven, die enkel als helper-klasse zal functioneren: bvb logging is een goed voorbeeld
- Deze klasse deelt dan geen gemeenschappelijk methods met de eventuele andere basis-klassen
- Dergelijke basis-klassen worden vaak mixin-klassen genoemd

```
class MijnMixin():
    def loggen(self):
        import pprint
        pprint.pprint(vars(self))

class MijnKlasse(MijnMixin):
    pass
```

Toegang tot attributen

Directe toegang tot attributen

Object attributen en methods zijn in Python (normaal gesproken) publiek. Guido Van Rossum daarover: "we zijn allemaal volwassen"

```
class Kunstwerk():
    def __init__(self, invoer_naam):
        self.naam = invoer_naam

mona_lisa_van_leonardo = Kunstwerk("Mona Lisa")
print(mona_lisa_van_leonardo.naam) # Mona Lisa
mona_lisa_van_leonardo.naam = "Mano Lasi"
print(mona_lisa_van_leonardo.naam) # Mano Lasi
```

Getter- en setter-methods

- Programmeertalen ondersteunen vaak private attributen die niet van buitenaf kunnen benaderd of gewijzigd worden. Developers moeten in dat geval getter- en setter-methods schrijven
- Python heeft geen private attributen, maar je kan wel getters en setters schrijven met aangepaste attributen-namen om wat veiligheid in te bouwen

```
class Kunstwerk():
    def __init__(self, invoer_naam):
        self.verborgen_naam = invoer_naam
    def get_naam(self):
        return self.verborgen_naam
    def set_naam(self, invoer_naam):
        self.verborgen_naam = invoer_naam
```

Toegang tot attributen via properties

De beste manier om toegang te regelen tot attributen is via properties. Het kan op 2 manieren:

```
class Kunstwerk():
    def __init__(self, invoer_naam):
        self.verborgen_naam = invoer_naam
    def get_naam(self):
        return self.verborgen_naam
    def set_naam(self, invoer_naam):
        self.verborgen_naam = invoer_naam
    naam = property(get_naam, set_naam)

mona_lisa_van_leonardo = Kunstwerk("Mona Lisa")
print(mona_lisa_van_leonardo.naam) # Mona Lisa
mona_lisa_van_leonardo.naam = "Mano Lasi"
print(mona_lisa_van_leonardo.naam) # Mano Lasi
```

Toegang tot attributen via properties

De tweede manier is door gebruik van decorators en door het vervangen van de method-namen get_naam en set_naam door naam:

```
class Kunstwerk():
    def __init__(self, invoer_naam):
        self.verborgen_naam = invoer_naam
    @property
    def naam(self):
        return self.verborgen_naam
    @naam.setter
    def naam(self, invoer_naam):
        self.verborgen_naam = invoer_naam
```

Properties voor berekende waarden

- Een property kan ook een berekende waarde (een computed value) teruggeven
- Geef je geen setter-property mee, dan kan je de property niet veranderen (handig voor read-only attributen)

```
class Rechthoek():
    def __init__(self, basis, hoogte):
        self.basis = basis
        self_hoogte = hoogte
    @property
    def oppervlakte(self):
        return self.basis * self.hoogte

mijn_rechthoek = Rechthoek(4,6)
print(mijn_rechthoek.oppervlakte) # 24
```

Naamaanpassing voor meer privacy

- Python heeft een conventie voor attributen die niet zichtbaar horen te zijn buiten hun klasse definitie
- Je laat ze beginnen met 2 underscores: __
- Biedt geen perfecte bescherming, maar overtredingen zullen altijd opzettelijk zijn :-)

```
class Kunstwerk():
    def __init__(self, invoer_naam):
        self.__naam = invoer_naam
    @property
    def naam(self):
        return self.__naam
    @naam.setter
    def naam(self, invoer_naam):
        self.__naam = invoer_naam
```

Soorten methods

Soorten methods

- Methods kunnen onderdeel van de klasse zelf zijn, terwijl andere onderdeel zijn van de objecten die uit die klasse worden geïnstantieerd (instance methods). Nog andere zijn geen van bovenstaande
- Indien de method niet wordt voorafgegaan door een decorator (@...) is het een instance method met als eerste argument self (deze hebben we tot nu toe gezien)
- Indien de method wordt voorafgegaan door de @classmethod decorator is het een class method met als eerste argument cls
- Indien de method wordt voorafgegaan door de @staticmethod decorator is het een static method: het eerste argument is dan geen object of klasse

Class methods

- Een class method heeft betrekking op de klasse in zijn geheel
- Elke wijziging die je maakt aan de klasse beïnvloedt alle geïnstantieerde objecten
- Een voorafgaande @classmethod decorator geeft aan dat de functie die volgt een class method is
- De eerste parameter is de klasse zelf. Conventie is om deze parameter cls te noemen

Class methods

```
class Kunstwerk():
    teller = 0
    def __init__(self, invoer_naam):
        self.__naam = invoer_naam
        Kunstwerk.teller += 1
    @classmethod
    def aantal(cls):
        print("Kunstwerk heeft", cls.teller, "prachtige kunstwerken."

kunstwerk_a = Kunstwerk("Mona Lisa")
kunstwerk_b = Kunstwerk("David")
print(Kunstwerk.aantal) # Kunstwerk heeft 2 prachtige kunstwerken
```

Static methods

- Static methods hebben geen invloed op de klasse of de geïnstantieerde objecten
- We gebruiken ze voor ons gemak en bundelen ze in de klasse voor organisatorische redenen
- We geven het aan met de @staticmethod decorator en gebruiken geen self of cls parameter
- We hoeven ook geen object te instantiëren om de method te gebruiken

```
class Museum():
    @staticmethod
    def open_op_zondag():
        return "Het museum is open op zondag"

print(Museum.open_op_zondag())
```

Python OOP / development: objecten en klassen in Python - kristof.michiels01@ap.be