Python OO Programming

Python OO les 2

Kristof Michiels

Topics

- Klassen en instances
  - Instance methods en attributen
  - Instance type nakijken in runtime
  - Methods: op klasse-niveau en statische methods
- Overerving en object compositie
- Magische object methods
- Labo-oefening

Klassen en instances

Ter herinnering: een eenvoudige klasse-definitie

```
class Boek:
    def __init__(self, titel):
        self.titel = titel

boek1 = Boek("Brave New World")
boek2 = Boek("World War Z")

print(boek1)
print(boek1.titel)
```

Een klasse importeren uit een module

```
#module.py

class Boek:
    def __init__(self, titel):
        self.titel = titel
```

```
import module

boek1 = module.Boek("Brave New World")
boek2 = module.Boek("World War Z")

print(boek1)
print(boek1.titel)
```

Werken met modules houdt je code georganiseerd en gestructureerd.

Instance methods & attributen

```
class Boek:
    def __init__(self, titel, lengte, auteur, prijs):
        self.titel = titel
        self.lengte = lengte
        self.auteur = auteur
        self.prijs = prijs
        self.__geheim = "Dit is een geheime eigenschap"

    def setKorting(self, hoeveelheid):
        self._korting = hoeveelheid

    def getPrijs(self):
        if hasattr(self, "_korting"):
            return self.prijs - (self.prijs * self._korting)
        else:
            return self.prijs
```

Instance methods & attributen

```
oBoek1 = Boek("Brave New World", "Aldous Huxley", 254, 26)
oBoek2 = Boek("World War Z", "Max Brooks", 433, 12)

print(oBoek1.getPrijs())
print(oBoek2.getPrijs())
oBoek2.setKorting(0.25)
print(oBoek2.getPrijs())

# Eigenschappen met dubbele underscores worden verborgen door de interpreter
print(oBoek2._korting)
# print(oBoek2.__geheim)
# print(oBoek2._Boek__geheim)
```

Instance methods & attributen

- _korting: intern attribuut. Gewoonte om met underscore te laten beginnen: "gebruik het niet in je code"
- hasattr(): testen als een attribuut aanwezig is
- dubbele underscore (__geheim):
  - attribuut kan niet bekeken worden buiten de klasse.
  - Om ervoor te zorgen dat sub-klassen het attribuut niet kunnen overschrijven
  - Andere klassen kunnen dit omzeilen door de klassenaam toe te voegen

Instance type nakijken in runtime

- Nakijken in runtime welk type of klasse een object is
- Je kan het object type nakijken met type()
- Je kan twee types vergelijken
- Je kan isinstance() gebruiken om een specifieke instance te vergelijken met een gekend type

Instance type nakijken in runtime

```
class Boek:
    def __init__(self, titel):
        self.titel = titel

class Krant:
    def __init__(self, naam):
        self.naam = naam

oBoek1 = Boek("The Walking Dead")
oBoek2 = Boek("World War Z")
oKrant1 = Krant("De Morgen")
oKrant2 = Krant("Het Laatste Nieuws")
```

Instance type nakijken in runtime

```
print(type(oBoek1))
print(type(oKrant1))
print(type(oBoek1) == type(oBoek2))
print(type(oBoek1) == type(oKrant2))
print(isinstance(oBoek1, Boek))
print(isinstance(oKrant1, Krant))
print(isinstance(oKrant2, Boek))
print(isinstance(oKrant2, object))
```

Methods: op klasse-niveau en statische methods

```
class Boek:
    BOEK_TYPES = ("HARDCOVER", "PAPERBACK", "EBOOK")
    __boeklijst = None

    @staticmethod
    def getboeklijst():
        if Boek.__boeklijst == None:
            Boek.__boeklijst = []
        return Boek.__boeklijst

    @classmethod
    def getboektypes(cls):
        return cls.BOEK_TYPES

    def setTitel(self, nieuwetitel):
        self.titel = nieuwetitel
```

Methods: op klasse-niveau en statische methods

```
    def __init__(self, titel, boektype):
        self.titel = titel
        if (not boektype in Boek.BOEK_TYPES):
            raise ValueError(f"{boektype} is geen toegelaten boektype")
        else:
            self.boektype = boektype

print("Boek types: ", Boek.getboektypes())
oBoek1 = Boek("Titel 1", "HARDCOVER")
oBoek2 = Boek("Titel 2", "PAPERBACK")
deboeken = Boek.getboeklijst()
deboeken.append(oBoek1)
deboeken.append(oBoek2)
print(deboeken)
```

Methods: op klasse-niveau en statische methods

- We hebben al gezien hoe de __init__ method werkt
- We hebben al gezien hoe de instance methods werken: setTitel
  - krijgen een specifiek object als argument
  - werken op data specifiek voor die instance
- Klasse methods worden gedeeld op klasse niveau tussen alle instances van die klasse: getBoektypes
  - krijgen een klasse als argument
  - gebruikt de @classmethod decorator
  - Je gebruikt de klassenaam om de method aan te roepen

Methods: op klasse-niveau en statische methods

- Eigenschappen gecreëerd op klasse-niveau worden gedeeld door alle instances: BOEK_TYPES
  - je verwijst bij gebruik naar de klassenaam
- Statische methods: getBoeklijst
  - ontvangen geen klasse of instance argumenten
  - werken gebruikelijk op data die niet instance of klasse-specifiek is
  - wordt niet superveel gebruikt
  - eerder voor namespacing / een method die "hoort bij" de klasse: als het ware een globale functie die je onderbrengt in de namespace van de klasse
  - gebruikt de @classmethod decorator

Overerving en object compositie

Overerving

- één van de belangrijkste concepten van OO programmeren
- Zorgt ervoor dat een klasse eigenschappen en methods kan erven van één of meer *base* klassen
- Maakt het eenvoudig om gemeenschappelijke data en functionaliteit te centraliseren in één plaats
- Resultaat: effiëntere code organisatie

Overerving

```
class Publicatie:
    def __init__(self, titel, prijs):
        self.titel = titel
        self.prijs = prijs

class Periodiek(Publicatie):
    def __init__(self, titel, prijs, uitgever, periode):
        super().__init__(titel, prijs)
        self.periode = periode
        self.uitgever = uitgever

class Boek(Publicatie):
    def __init__(self, titel, auteur, lengte, prijs):
        super().__init__(titel, prijs)
        self.auteur = auteur
        self.lengte = lengte
```

Overerving

```
class Tijdschrift(Periodiek):
    def __init__(self, titel, uitgever, prijs, periode):
        super().__init__(titel, prijs, uitgever, periode)

class Krant(Periodiek):
    def __init__(self, titel, uitgever, prijs, periode):
        super().__init__(titel, prijs, uitgever, periode)


oBoek1 = Boek("Brave New World", "Aldous Huxley", 311, 26)
oKrant1 = Krant("De Morgen", "DPG Media", 2, "Dagelijks")
oTijdschrift1 = Tijdschrift("Humo", "DPG Media", 4, "Wekelijks")

print(oBoek1.auteur)
print(oKrant1.uitgever)
print(oBoek1.prijs, oKrant1.prijs, oTijdschrift1.prijs)
```

Overerving

- Publicatie is een *base*-klasse
- *Super*-klasse __init__-functie aanroepen: super().__init__()

Abstracte basisklassen

- Je wil een basisklasse voorzien die een template biedt voor overerving naar andere klassen
- Enkel een blauwdruk, kan niet geïnstantieerd worden
- Subklassen worden dan concrete implementaties van dat idee
- Je wil afdwingen dat bepaalde methods moeten geïmplementeerd worden in de subklassen

Abstracte basisklassen

```
from abc import ABC, abstractmethod

class GrafischeVorm(ABC):
    def __init__(self):
        super().__init__()

    @abstractmethod
    def berekenOppervlakte(self):
        pass


class Cirkel(GrafischeVorm):
    def __init__(self, straal):
        self.straal = straal

    def berekenOppervlakte(self):
        return 3.14 * (self.straal ** 2)
```

Abstracte basisklassen

```
class Vierkant(GrafischeVorm):
    def __init__(self, zijde):
        self.zijde = zijde

    def berekenOppervlakte(self):
        return self.zijde * self.zijde

oCirkel1 = Cirkel(10)
print(oCirkel1.berekenOppervlakte())
oVierkant1 = Vierkant(12)
print(oVierkant1.berekenOppervlakte())
```

Abstracte basisklassen

- We maken gebruik van de ABC-module van de standard library
- We gebruiken de @abstractmethod decorator om aan te geven dat een method abstract is en elke subklasse deze method moet overschrijven

Meervoudige overerving

- Gebruik je wanneer je een klasse maakt die overerft van meer dan één basisklasse
- Kan handig zijn, indien met de nodige voorzichtigheid gebruikt

Meervoudige overerving

```
class A:
    def __init__(self):
        super().__init__()
        self.foo = "foo"
        self.naam = "Klasse A"

class B:
    def __init__(self):
        super().__init__()
        self.bar = "bar"
        self.naam = "Klasse B"
```

Meervoudige overerving

```
class C(B, A):
    def __init__(self):
        super().__init__()

    def tooneigenschappen(self):
        print(self.foo)
        print(self.bar)
        print(self.naam)

oC = C()
print(C.__mro__)
oC.tooneigenschappen()
```

Meervoudige overerving

- Indien een eigenschap met dezelfde naam aanwezig is (in het voorbeeld: naam)
- "Method resolution order": volgorde in dewelke ze gedefinieerd zijn
- Volgorde kennen? Gebruik het mro-attribuut

Interfaces

- Python geen expliciete taalondersteuning voor interfaces
- Een soort contract of belofte om een bepaald gedrag of mogelijkheden te hebben
- Hier: combinatie van meervoudige overerving en ABC
- We zouden de JSONify klasse kunnen gebruiken als interface voor een andere klasse

Interfaces

```
from abc import ABC, abstractmethod

class GrafischeVorm(ABC):
    def __init__(self):
        super().__init__()

    @abstractmethod
    def berekenOppervlakte(self):
        pass

class JSONify(ABC):
    @abstractmethod
    def toJSON(self):
        pass
```

Interfaces

```
class Cirkel(GrafischeVorm, JSONify):
    def __init__(self, straal):
        self.straal = straal

    def berekenOppervlakte(self):
        return 3.14 * (self.straal ** 2)

    def toJSON(self):
        return f"{{ \"cirkel\": {str(self.berekenOppervlakte())} }}"


oCirkel = Cirkel(10)
print(oCirkel.berekenOppervlakte())
print(oCirkel.toJSON())
```

Object compositie

- Dit is het bouwen van complexe objecten uit andere meer eenvoudige objecten
- Overerving: "is een"-relatie
- Samenstelling: "heeft"-relatie
  - Een boek: heeft een auteur, heeft hoofdstukken
- Verschillende ideeën kunnen geïsoleerd worden en in hun eigen klassen geplaatst
- Je kan beide technieken combineren

Object compositie

```
class Boek:
    def __init__(self, titel, prijs, auteur=None):
        self.titel = titel
        self.prijs = prijs

        self.auteur = auteur
        self.hoofdstukken = []

    def voeghoofdstuktoe(self, hoofdstuk):
        self.hoofdstukken.append(hoofdstuk)

    def getboekpaginatelling(self):
        resultaat = 0
        for h in self.hoofdstukken:
            resultaat += h.paginatelling
        return resultaat
```

Object compositie

```
class Auteur:
    def __init__(self, voornaam, familienaam):
        self.voornaam = voornaam
        self.familienaam = familienaam

    def __str__(self):
        return f"{self.voornaam} {self.familienaam}"


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

Object compositie

```
oAuteur1 = Auteur("Leo", "Tolstoy")
oBoek1 = Boek("War and Peace", 39.95, oAuteur1)

oBoek1.voeghoofdstuktoe(Hoofdstuk("Hoofdstuk 1", 102))
oBoek1.voeghoofdstuktoe(Hoofdstuk("Hoofdstuk 2", 91))
oBoek1.voeghoofdstuktoe(Hoofdstuk("Hoofdstuk 3", 124))

print(oBoek1.titel)
print(oBoek1.auteur)
print(oBoek1.getboekpaginatelling())
```

Magische methods

Magische methods?

- Een aantal methods die Python associeert met elke klasse definitie
- Je klassen kunnen die methoden overschrijven om bepaald gedrag aan te passen
- Er zijn er heel veel, wij gaan kijken naar de meest bruikbare en gebruikte
- Magical methods (Engels) - ook metaobjecten genoemd

String vertegenwoordiging

- De __str__ functie wordt gebruikt om een gebruiksvriendelijke string vertegenwoordiging van het object terug te geven
- De __repr__ functie wordt gebruikt om een gebruiksvriendelijke string vertegenwoordiging van het object terug te geven
- Maakt bvb debuggen gemakkelijker

String vertegenwoordiging

```
class Boek:
    def __init__(self, titel, auteur, prijs):
        super().__init__()
        self.titel = titel
        self.auteur = auteur
        self.prijs = prijs

    def __str__(self):
        return f"{self.titel} door {self.auteur}, kost {self.prijs}"

    def __repr__(self):
        return f"titel={self.titel},auteur={self.auteur},prijs={self.prijs}"
```

String vertegenwoordiging

```
oBoek1 = Boek("Brave New World", "Aldous Huxley", 26)
oBoek2 = Boek("Elementaire Deeltjes", "Michel Houellebecq", 19)

print(oBoek1)
print(oBoek2)

print(str(oBoek1))
print(repr(oBoek2))
```

Gelijkwaardigheid en vergelijken

- Objecten in Python kunnen zich normaal gezien niet met elkaar vergelijken. Maar we kunnen ze het leren.
- De __eq__ method wordt opgeroepen wanneer het object vergeleken wordt met een ander
- De __ge__ method gaat een groter dan of gelijk aan-relatie nagaan met een ander object
- De __lt__ method gaat een kleiner dan of gelijk aan-relatie nagaan met een ander object
- Als beide bovenstaande method aanwezig is worden onze objecten meteen ook sorteerbaar
- https://docs.python.org/3/reference/datamodel.html

Gelijkwaardigheid en vergelijken

```
class Boek:
    def __init__(self, titel, auteur, prijs):
        super().__init__()
        self.titel = titel
        self.auteur = auteur
        self.prijs = prijs

    def __eq__(self, waarde):
        if not isinstance(waarde, Boek):
            raise ValueError("Kan geen boek vergelijken met een niet-boek type")
        return (self.titel == waarde.titel
        and self.auteur == waarde.auteur
        and self.prijs == waarde.prijs)
```

Gelijkwaardigheid en vergelijken

```
    def __ge__(self, waarde):
        if not isinstance(waarde, Boek):
            raise ValueError("Kan geen boek vergelijken met een niet-boek type")
        return self.prijs >= waarde.prijs

    def __lt__(self, waarde):
        if not isinstance(waarde, Boek):
            raise ValueError("Kan geen boek vergelijken met een niet-boek type")
        return self.prijs < waarde.prijs
```

Gelijkwaardigheid en vergelijken

```
oBoek1 = Boek("Elementaire deeltjes", "Michel Houellebecq", 20)
oBoek2 = Boek("The Catcher in the Rye", "JD Salinger", 29)
oBoek3 = Boek("Elementaire deeltjes", "Michel Houellebecq", 20)
oBoek4 = Boek("To Kill a Mockingbird", "Harper Lee", 24)
print(oBoek1 == oBoek3)
print(oBoek1 == oBoek2)
print(oBoek2 >= oBoek1)
print(oBoek2 < oBoek1)
print(oBoek3 >= oBoek2)
boeken = [oBoek1, oBoek3, oBoek2, oBoek4]
boeken.sort()
print([boek.titel for boek in boeken])
```

Toegang tot attributen

- De magic methods geven ook complete controle over hoe een object's attributen gewijzigd of opgevraagd worden
- De __getattribute__ wordt opgeroepen telkens een attribuut wordt opgevraagd
- De __settattribute__ wordt opgeroepen telkens een attribuut wordt aangepast
- De __getattr__ wordt opgeroepen telkens een __getattribute__ opvraging faalt

Toegang tot attributen

```
class Boek:
    def __init__(self, titel, auteur, prijs):
        super().__init__()
        self.titel = titel
        self.auteur = auteur
        self.prijs = prijs
        self._korting = 0.1

    def __str__(self):
        return f"{self.titel} door {self.auteur}, kost {self.prijs}"

    def __getattribute__(self, naam):
        if (naam == "prijs"):
            p = super().__getattribute__("prijs")
            k = super().__getattribute__("_korting")
            return p - (p * k)
        return super().__getattribute__(naam)
```

Toegang tot attributen

```
    def __setattr__(self, naam, waarde):
        if (naam == "prijs"):
            if type(waarde) is not float:
            raise ValueError("Het 'prijs' attribuut moet een float zijn")
        return super().__setattr__(naam, waarde)

    def __getattr__(self, naam):
        return naam + " is hier niet!"
```

Toegang tot attributen

```
oBoek1 = Boek("War and Peace", "Leo Tolstoy", 39.95)
oBoek2 = Boek("The Catcher in the Rye", "JD Salinger", 29.95)
oBoek1.prijs = 38.95
print(oBoek1)
oBoek2.prijs = float(40)
print(oBoek2)
print(oBoek1.willekeurigemethod)
```

Oproepbare objecten

- De __call__ method zorgt ervoor dat het object kan aangeroepen worden zoals eender dewelke functie
- Zelfde aantal parameters zoals de init-functie
- Handig als je objecten hebt wiens attributen vaak wijzigen, of vaak samen worden gewijzigd
- Resulteert in meer compacte en eenvoudiger te lezen code

Oproepbare objecten

```
class Boek:
    def __init__(self, titel, auteur, prijs):
        super().__init__()
        self.titel = titel
        self.auteur = auteur
        self.prijs = prijs

    def __str__(self):
        return f"{self.titel} door {self.auteur}, kost {self.prijs}"

    def __call__(self, titel, auteur, prijs):
        self.titel = titel
        self.auteur = auteur
        self.prijs = prijs
```

Oproepbare objecten

```
class Boek:
    def __init__(self, titel, auteur, prijs):
        super().__init__()
        self.titel = titel
        self.auteur = auteur
        self.prijs = prijs

    def __str__(self):
        return f"{self.titel} door {self.auteur}, kost {self.prijs}"

    def __call__(self, titel, auteur, prijs):
        self.titel = titel
        self.auteur = auteur
        self.prijs = prijs
```

Labo-oefening

Labo-oefening

- Maak een opzet voor een eenvoudige text-based adventure game
- Werk object-georiënteerd
- Gebruik waar het kan de concepten die we deze week hebben gezien

Python OO programming - les 2 -

kristof.michiels01@ap.be