Algemeen

Vergelijkingsoperatoren en zelfgedefinieerde gegevenstypes

Om twee objecten van een zelfgedefinieerd gegevenstype te kunnen vergelijken, moet een specifieke implementatie voor de vergelijkingsoperatoren voorzien worden. Dit kan door de volgende magische methodes te overladen:

methode	operator
___lt___	$<$
___le___	$\leq$
___gt___	$>$
___ge___	$\geq$
___eq___	$=$
___ne___	$\neq$

Let er op dat het in de meeste gevallen volstaat om sommige van deze vergelijkingsoperatoren te implementeren op basis van andere vergelijkingsoperatoren. Zo zijn twee objecten doorgaans verschillend als ze niet gelijk zijn aan elkaar.

De format specifier `!r`

Python heeft twee verschillende ingebouwde functies waarmee een object kan omgezet worden naar een string: `str()` en `repr()`. Python gebruikt standaard de ingebouwde functie `str()` als het een object moet omzetten naar een string in een f-string of bij het gebruik van de stringmethode `str.format()`. Als je in plaats daarvan echter gebruik wil maken van de ingebouwde functie `repr()`, dan kan je ofwel die functie expliciet aanroepen of kan je de *format specifier* `!r` gebruiken.

```
>>> cursus = 'programmmeren'
>>> str(cursus)                                    # str (expliciet)
'programmmeren'
>>> repr(cursus)                                   # repr (expliciet)
"'programmmeren'"
>>> cursus                                         # repr (impliciet)
'programmmeren'
>>> print(cursus)                                  # str (impliciet)
programmeren
>>> f'De naam van de cursus is {cursus}.'          # str (impliciet)
'De naam van de cursus is programmmeren.'
>>> f'De naam van de cursus is {repr(cursus)}.'    # repr (expliciet)
"De naam van de cursus is 'programmmeren'."
>>> f'De naam van de cursus is {cursus!r}.'        # repr (impliciet)
"De naam van de cursus is 'programmmeren'."
```

Operator overloading voor zelfgedefinieerde gegevenstypes

Als Python de expressie

```
o1 + o2
```

moet evalueren, dan wordt die expressie omgezet naar

```
type(o1).__add__(o1, o2)
```

Op die manier kan je voor zelfgedefinieerde types vastleggen hoe de `+`-operator moet uitgevoerd worden. Dit wordt *operator overloading* genoemd. Dit blijft echter niet beperkt tot de `+`-operator. Python vertaalt elke

ingebouwde operator (wiskundige operatoren en vergelijkingsoperatoren) naar een methode waarvan de naam is vastgelegd door de ontwikkelaars van Python (de naam begint en eindigt telkens met twee underscores). Hier is een overzicht van enkele van deze *magische* methoden die met operatoren corresponderen:

operator	methode
`+`	`__add__`
`-`	`__sub__`
`*`	`__mul__`
`/`	`__truediv__`
`//`	`__floordiv__`
`**`	`__pow__`

Bij operator overloading wordt de *magische* methode dus aangeroepen op het linker operand `o1`. Maar wat als de klasse van het linker operand `o1` de magische methode niet definieert voor objecten van `type(o2)`? In dat geval wordt er een *exception* opgeworpen, en probeert Python vervolgens een andere *magische* methode (waarbij de naam van de *magische* methode wordt voorafgegaan door de letter `r`) aan te roepen op het rechter operand `o2`.

De optelling van hierboven wordt dus in tweede instantie omgezet naar

```
type(o2).__radd__(o2, o1)
```

Let hierbij op het feit dat de naam van de methode `__radd__` geworden is, in plaats van `__add__`, en dat de volgorde van de argumenten omgekeerd is. Dat laatste is vooral belangrijk voor bewerkingen die niet symmetrisch zijn.

Een verwijzing naar het huidige object teruggeven

Sommige methoden moeten een verwijzing teruggeven naar het object waarop ze werden aangeroepen. Dit object wordt automatisch als eerste argument doorgegeven aan de methode, en wordt dus toegekend aan de parameter die we de naam `self` geven als we de Python-conventie voor de naamgeving van de eerste parameter van een methode volgen.

Stel je voor dat we het spelletje Tic-Tac-Toe willen implementeren:

```
class TicTacToe:
  def __init__(self):
    self.rooster = [
      [ None, None, None ],
      [ None, None, None ],
      [ None, None, None ]
    ]
    self.speler = 'O'

  def zet(self, i, j):
    self.rooster[i][j] = self.speler
    self.speler = 'O' if self.speler == 'X' else 'X'
    return self
```

Dit laat ons toe om het spel als volgt te spelen

```
>>> spel = TicTacToe().zet(1, 1).zet(0, 0).zet(0, 1).zet(1, 0)
>>> spel.rooster
[
  ['X', 'O', None],
  ['X', 'O', None],
```

```
  [None, None, None]
]
```

waarbij we dus verschillende aanroepen van de methode `TicTacToe.zet()` achter elkaar kunnen zetten. Dit wordt mogelijk gemaakt door het feit dat deze methode een verwijzing teruggeeft naar het object waarop ze wordt aangeroepen: `return self`.

Gebruik van `self`

Als je werkt met klassen, dan is het belangrijk dat je onderscheid maakt tussen twee soorten variabelen. Er zijn **objecteigenschappen** die kunnen aangesproken worden in alle methoden van de klasse en er zijn **lokale variabelen* die enkel toegankelijk zijn binnen de methode waarin ze gedefinieerd worden. Enkele de namen van de objecteigenschappen moeten voorafgegaan worden door `self`. De lokale variabelen van een methode moeten niet voorafgegaan worden door `self`, en het getuigt van een zeer slechte programmeerstijl als je dat toch doet.

Objecteigenschappen initialiseren in de intialisatiemethode

Voor je begint aan de implementatie van een klasse, moet je eerste bepalen welke eigenschappen de objecten van die klasse moeten hebben. Deze variabelen beschrijven de interne toestand van de individuele objecten en kunnen in alle methoden van de klasse gebruikt worden. In methoden kan je naar objecteigenschappen verwijzen door de prefix `self.` voor hun naam te zetten. Het is altijd een goed idee om deze objecteigenschappen te definiëren in de `__init__` methode en er daar een initiële waarde aan toe te kennen.