Algemeen

Zelfgedefinieerde vergelijkingsoperatoren

Om twee objecten van een zelfgedefinieerd type te kunnen vergelijken, moet een specifieke implementatie voor de vergelijkingsoperatoren voorzien worden. Dit kan door 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 `format`. Als je in plaats daarvan echter gebruik wil maken van de ingebouwde functie `repr`, dan kan je ofwel die functie expliciet aanroepen of je kan gebruikmaken van de *format specifier* `!r`.

```
>>> cursus = 'programmmeren'
>>> str(cursus)                                    # str (expliciet)
'programmmeren'
>>> repr(cursus)                                   # repr (expliciet)
"'programmmeren'"
>>> cursus                                         # repr (impliciet)
'programmmeren'
>>> 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 bij zelfgedefinieerde types

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:

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 het 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 objecten geven na een bewerking een verwijzing naar zichzelf terug. Stel 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', 'X', None ],
  [ 'O', 'O', None ],
  [ None, None, None ]
]
```

Het belangrijke hier is dus de `return self`.