

# Datastructuren

Bas Terwijn

February 9, 2026

## 1 Introductie

Dit document behandelt Python datastructuur 'set' en 'dict'.

### 1.1 Doelen

- oefenen met 'set' en 'dict'
- de relatie met equality en hashing begrijpen

## 2 Opdracht `art_heist.py`

Er heeft een kunstroof plaatsgevonden in het nationale kunstmuseum van Waddinxveen. De politie ziet een deel van de inwoners van Nederland als mogelijke dader. In bestand `assignments/inwoners.csv` vind je de id en naam van elke van deze 200.000 mogelijke daders.

Het is jouw taak om voor de politie de groep van mogelijke daders zo klein mogelijk te maken op basis van onderstaande informatie.

## 3 Buitenland

Mensen die in het buitenland waren ten tijden van de kunstroof zijn geen daders. De id van deze mensen vind je in bestand `assignments/buitenland.csv`.

**opdracht1:** Bepaal de ids van de nog mogelijke daders. Gebruik hierbij eerst een 'list' om de ids op te slaan en op te zoeken.

*tip:* Gebruik Python module 'csv' om makkelijker csv bestanden te lezen.

*test:* Er zouden nog 183.745 mogelijke daders moeten overblijven.

**opdracht2:** Het zoeken in een 'list' is traag, gebruik daarom nu een [set](#) om de ids op te slaan en op te zoeken voor hogere snelheid.

*tip:* Controleer of het type van de verzamelingen van ids echt 'sets' zijn door het type te printen, bv met:

```
print( type(dader_ids) )
```

- Hoeveel sneller is een 'set' in dit geval?
- Is het resultaat hetzelfde als bij gebruik van een 'list'?
- Waarom is een 'set' sneller?

**opdracht3:** Het is nog iets sneller, makkelijker en leesbaarder om de set operatoren in figuur 1 te gebruiken.

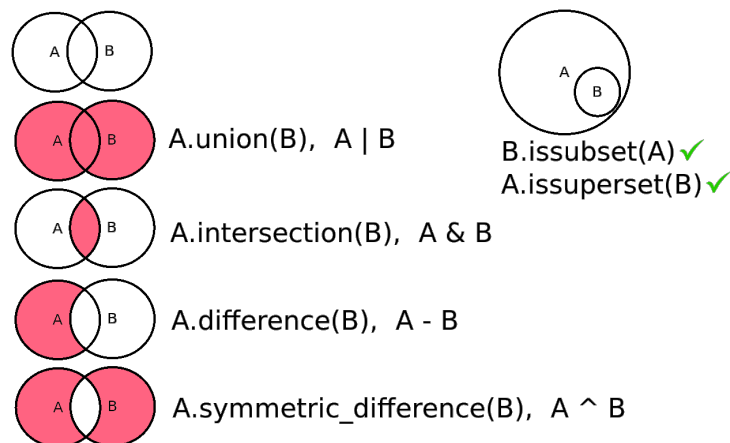


Figure 1: set operatoren

### 3.1 Lengte

Met beveiligingscamera's is vastgesteld dat de dader een lengte heeft groter of gelijk aan 150 en kleiner of gelijk aan 160 cm. De lengte van inwoners vind je in bestand `assignments/lengte.csv`.

**opdracht4:** Wat zijn nu de ids van de nog mogelijke daders?

*test:* Er zouden nog 49.130 mogelijke daders moeten overblijven.

### 3.2 Geboortedatum

Op basis van getuigenverklaringen is vastgesteld dat de dader geboren is in of na 1980 en in of voor 1990. De geboortedatum van inwoners vind je in bestand `assignments/geboortedatum.csv`.

**opdracht5:** Wat zijn nu de ids van de nog mogelijke daders?

*tip:* Probeer waar mogelijk code-duplicatie te voorkomen door herbruikbare functies te maken.

*test:* Er zouden nog 9.643 mogelijke daders moeten overblijven.

### 3.3 Beroepscrimineel of Gelegenheidsdief

De politie neemt aan dat de dader een beroepscrimineel is die al eerder in contact is gekomen met justitie, **of** dat het een gelegheidsdief is die binnen 50km van het museum woont. Het museum bevindt zich op GPS coördinaat breedtegraad:52.030592, lengtegraad:4.641464.

De ids van beroepscriminelen die al eerder in contact zijn gekomen met justitie vind je in bestand `assignments/beroepscrimineel.csv`.

De postcode van inwoners vind je in bestand `assignments/postcode.csv`.

Het GPS coördinaat van iedere postcode vind je in bestand `assignments/postcode_coordinaat.csv`.

### 3.4 GPS coördinaten

Gebruik Python module 'geopy' om afstanden tussen GPS coördinaten te bepalen. Installeer deze module met:

```
pip install geopy
```

Vervolgens kun je op deze manier bijvoorbeeld de afstand tussen Amsterdam en Parijs bepalen:

```
from geopy.distance import distance as geopy_distance
```

```

amsterdam = (52.372947, 4.893291)
parijs    = (48.857891, 2.295166)

print(geopy_distance(amsterdam, parijs).km) # distance in kilometers

```

**opdracht6:** Wat zijn nu de ids van de nog mogelijke daders?

*test:* Er zouden nog 2.700 mogelijke daders moeten overblijven.

### 3.5 Resultaat

Rapoteer je resultaat.

**opdracht7:** Maak een csv bestand 'daders.csv' van de nog mogelijke daders met 'id, naam' als header.

Test deze opdracht met:

```
pytest test_art_heist.py
```

## 4 Equality en Hashing

Als we objecten van een class aan een set of dict willen toevoegen, komen we een beetje technisch probleem tegen. We kijken hier naar een voorbeeld waar we een object van class `Coord`, genaamd `c1`, toevoegen aan set `coord_set`.

```

src/coord.py

class Coord:

    def __init__(self, values):
        self.values = values

    def __repr__(self):
        return str(self.values)

    def __setitem__(self, index, value):
        self.values[index] = value

c1 = Coord([1, 1])
c2 = Coord([1, 1])
print(id(c1), id(c2)) # 139940334891904 139940334097168 (different)

# identity-based equality and hashing is the default
print( c1 == c2 )           # False
print( hash(c1), hash(c2) ) # 7933673374520 7883269437765 (different)

coord_set = set([c1])
print(coord_set)           # {[1, 1]}
print( c1 in coord_set )   # True
print( c2 in coord_set )   # False BAD!

```

Het probleem is dat voor een class standaard:

- de equality operator `==` gebaseerd is op de identiteit en niet de waarden van een object
- de hash functie `hash()` gebaseerd is op de identiteit en niet de waarden van een object

Dus omdat de identiteit van `c1` en `c2` verschillend zijn, worden ze als verschillend gezien door de `==` operator en `hash()` functie, ook al hebben ze dezelfde waarden `[1, 1]`. Hierdoor wordt wel `c1` gevonden in `coord_set`

omdat het exact hetzelfde object is, maar `c2` niet omdat het een ander object is zoals weergegeven in figuur 2. Om ook `c2` te kunnen vinden in `coord_set` hebben we twee verschillende oplossingsmogelijkheden.

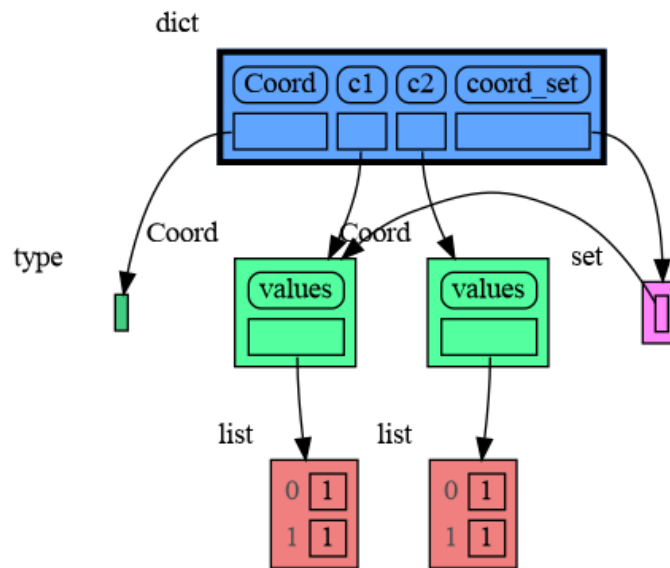


Figure 2: `coord_set` met een `Coord` object

#### 4.1 Oplossing 1: sla een immutable representatie op in de set

De eerste oplossing is om niet de `Coord` objecten in de set op te slaan, maar in plaats daarvan een builtin immutable representatie van de elk `Coord` object.

Dit helpt want:

- de equality operator `==` van een builtin type is gedefinieerd om naar de waarden en niet naar de identiteit te kijken
- een immutable type heeft een hash functie `hash()`, en ook die kijkt naar de waarden en niet naar de identiteit

We voegen hiervoor de `immu()` methode toe aan class `Coord` welke een immutable representatie van een `Coord` object teruggeeft. In dit geval kiezen we voor immutable type `tuple`, we hadden eventueel ook voor immutable type `str` kunnen kiezen.

```

src/coord1.py

class Coord:

    def __init__(self, values):
        self.values = values

    def __repr__(self):
        return str(self.values)

    def __setitem__(self, index, value):
        self.values[index] = value

    def immu(self):
        return tuple(self.values)

c1 = Coord([1, 1])
c2 = Coord([1, 1])
print(id(c1), id(c2)) # 139940334891904 139940334097168 (different)
  
```

```
# immutable types have value-based equality and hashing
print( c1.immu() == c2.immu() )           # True
print( hash(c1.immu()), hash(c2.immu()) ) # 838904819212 838904819212 (same)

coord_set = set([c1.immu()])
print(coord_set)                          # {(1, 1)}
print( c1.immu() in coord_set )           # True
print( c2.immu() in coord_set )           # True GOOD!
```

Nu wordt naast `c1` ook `c2` gevonden in `coord_set`, probleem opgelost. Een nadeel van deze oplossing is wel dat we nu niet moeten vergeten om `.immu()` te gebruiken en dat we nu geen `Coord` object maar een `tuple` in de set opslaan zoals figuur 3 laat zien.

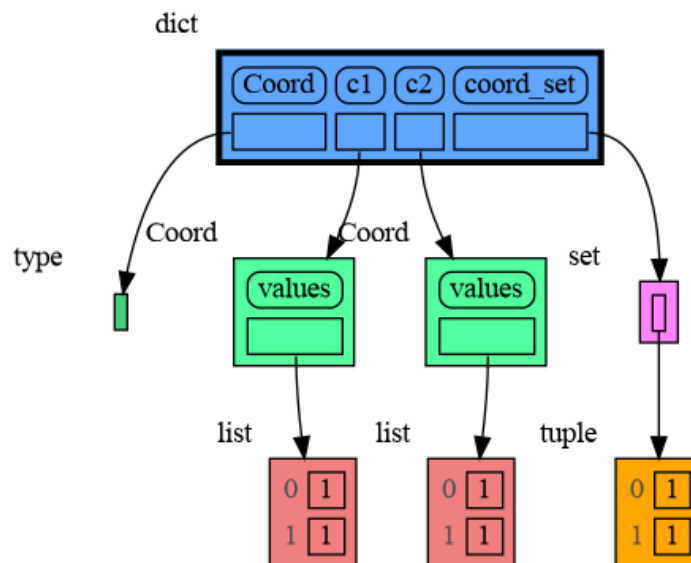


Figure 3: `coord_set` met een tuple waarden

Als we nu een element uit de set halen, krijgen we nu dus een `tuple` terug in plaats van een `Coord` object, en kunnen we dus niet de methoden van `Coord` gebruiken:

```
c = next(iter(coord_set))           # get first element in set
print( type(c) )                    # <class 'tuple'>
# c[0] = 0                           # TypeError: no item assignment BAD?
```

## 4.2 Oplossing 2: definieer zelf `==` en `hash()`

De tweede oplossing is om zelf de `==` en `hash()` te definiëren voor class `Coord` zodat deze gebaseerd zijn op de waarden van een `Coord` object in plaats van de identiteit. We voegen hiervoor de dunder methoden, `__eq__()` voor `==` en `__hash__()` voor `hash()`, toe aan class `Coord`.

```
src/coord2.py

class Coord:

    def __init__(self, values):
        self.values = values

    def __repr__(self):
        return str(self.values)
```

```

def __setitem__(self, index, value):
    self.values[index] = value

def __eq__(self, other):
    return self.values == other.values

def __hash__(self):
    return hash(tuple(self.values))

c1 = Coord([1, 1])
c2 = Coord([1, 1])
print(id(c1), id(c2)) # 139940334891904 139940334097168 (different)

# we defined value-based equality and hashing for Coord objects
print( c1 == c2 )           # True
print( hash(c1), hash(c2) ) # 838904819212 838904819212 (same)

coord_set = set([c1])
print(coord_set)           # {[1, 1]}
print( c1 in coord_set )   # True
print( c2 in coord_set )   # True GOOD!

```

Door deze methoden wordt er nu niet meer gekeken naar de identiteit maar naar de waarden want `__eq__()` vergelijkt de waarden van twee `Coord` objecten en niet de identiteit. Ook de `__hash__()` methode retourneert een hash-waarde van de waarden, we converteren hierbij wel eerst naar een tuple omdat deze een `__hash__()` functie heeft en een list niet.

Ook nu wordt naast `c1` ook `c2` gevonden in `coord_set`, probleem opgelost. We hoeven nu geen `.immut()` te gebruiken voor een immutable representatie, en we kunnen gewoon `Coord` objecten in de set opslaan zoals is weergegeven in figuur 2.

Als we nu een element uit de set halen, krijgen we nu dus wel een `Coord` object, en kunnen we de methoden van `Coord` gebruiken:

```

c = next(iter(coord_set)) # get first element in set
print( type(c) )          # <class '__main__.Coord'>
c[0] = 0                  # BAD?
print( c1 in coord_set )  # False

```

Maar, dat is ook gelijk het nadeel van deze oplossing, want met methoden kunnen we mogelijk een `Coord` object aanpassen (mutaten) nadat deze al in de set is geplaatst. Hierdoor verandert de hash-waarde van het object terwijl de set dat niet door heeft. Dan staat het object dus op de verkeerde plaats in de set en kan het niet meer gevonden worden. Dit kan dus voor bugs zorgen die erg moeilijk te vinden zijn.

Het hangt van de situatie af welke oplossing het beste is, maar geef de voorkeur aan oplossing 1 als het niet noodzakelijk is om objecten van een class in een set op te slaan, om bugs te voorkomen.

## 5 Opdracht birthday\_match.py

Wanneer we programma `birthday_match.py` uitvoeren, krijgen we een dictionary met 2000 paren van naam en random geboortedatum te zien:

```

$ python3 birthday_match.py | more # type 'Space' to scroll and 'q' to quit
people_birthdays: {'name1': 2011-03-12, 'name2': 1994-06-25, 'name3': 2011-03-12,
'name4': 1994-06-25, 'name5': 1957-12-03, ...}
matches: []

```

Sommige mensen delen dezelfde geboortedatum. Het is jouw taak om deze mensen te vinden met de functie `find_birthday_matches()` onderaan in dit programma. Deze functie moet een lijst teruggeven met daarin de lijsten van namen van alle mensen die eenzelfde geboortedatum delen. Namen van mensen die geen geboortedatum delen komen niet in deze lijst te staan. Op basis van bovenstaande voorbeelddata zou de output er dus als volgt uit moeten komen te zien:

```
$ python3 birthday_match.py
people_birthdays: ...
matches: [['name1', 'naam3'], ['name2', 'naam4'], ...]
```

omdat 'name1' en 'name3' dezelfde geboortedatum delen, en 'name2' en 'name4' ook, terwijl 'name5' een unieke geboortedatum heeft en dus wordt weggelaten.

## 5.1 Dictionary

Je kunt hiervoor een [dictionary](#) gebruiken om de mensen te groeperen op geboortedatum. Een dictionary gebruikt net als een set de `==` operator en `hash()` functie om keys te vergelijken en te hashen, maar pas op, want de `==` operator en `hash()` functie van een `Date` object is standaard op de identiteit van het object en dus niet op de waarden van het object. Dus pas één van bovenstaande oplossingsmogelijkheden toe om dit op te lossen.

Test deze opdracht met:

```
pytest test_birthday_match_obfus.py
```