Algemeen

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 klassemethoden 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. Elk van die eigenschappen komt dan overeen met een variabele die voorafgegaan wordt door de prefix `self.`. Deze variabelen beschrijven de interne toestand van de individuele objecten en kunnen in alle methoden van de klasse gebruikt worden. 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.

National Register Number

Gegevenstype controleren met `isinstance`

Om na te gaan of een gegeven object een bepaald gegevenstype heeft, kan je natuurlijk gebruik maken van de ingebouwde functie `type(o)` die het gegevenstype van het object `o` teruggeeft. Het is echter beter om hiervoor de ingebouwde functie `isinstance(o, t)` te gebruiken. Deze functie geeft een Booleaanse waarde terug die aangeeft of het object `o` al dan niet behoort tot het type `t`.

```
>>> type(3) == int
True
>>> isinstance(3.14, int)
False
>>> isinstance(3.14, float)
True
>>> isinstance([1, 2, 3], list)
True
```

Als je wil controleren of een gegeven object 1 van meerdere types is, kun je de volgende syntax gebruiken. Hierbij lijst je de mogelijke toegelaten types op in een tuple.

```
>>> isinstance(3, (str, int))
True
>>> isinstance('a', (str, int))
True
>> isinstance(['a'], (str, int))
False
```

Mad Libs

Stringmethode `capitalize()`

De stringmethode `capitalize` kan gebruikt worden om het eerste karakter van een string om te zetten naar een hoofdletter (als het een letter is), en alle overige karakters naar kleine letters (als het letters zijn).

```
>>> 'aBcD'.capitalize()
'Abcd'
```

Specifieke info

Om de methode `invullen` te implementeren kun je gebruik maken van de `split` functie. Als je `tekst.split('_')` uitvoert met de gegeven tekst, zul je een lijst terugkrijgen met afwisselend een stuk tekst dat niet vervangen moet worden en een stuk tekst dat wel wel vervangen moet worden.

```
>>> tekst = '_Naam_ created _ding_ so that _INWONERS_ would learn _discipline_.'
>>> tekst.split('_')
['', 'Naam', ' created ', 'ding', ' so that ', 'INWONERS', ' would learn ', 'discipline', '.']
```

Quipu

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.