Honkbal

None als standaardwaarde

Als je een functie wil definiëren met een optionele parameter, dan moet je aan die parameter altijd een standaarwaarde toekennen. Hierbij kan het gebeuren dat deze standaardwaarde niet gekend is op het ogenblik dat de functie gedefinieerd wordt, maar dat die pas kan bepaald worden op het ogenblik dat de functie wordt aangeroepen. Dat is bijvoorbeeld het geval als je een standaardwaarde wil berekenen op basis van argumenten die aan andere parameters doorgegeven worden. In dat geval is het een goed idee om als standaardwaarde de waarde `None` te gebruiken bij het definiëren van de functie, en in de body van de functie een waarde te berekenen indien aan de parameter de waarde `None` werd doorgegeven (betekenis: er werd niet expliciet een argument aan deze parameter doorgegeven).

```
def functie(eerste, tweede=None):
    if tweede is None:
        # ken dezelfde waarde toe aan de parameter tweede als het argument dat
        # werd doorgegeven aan de parameter eerste, in het geval dat er niet
        # expliciet een tweede argument werd doorgegeven aan de functie
        tweede = eerste

    ...
```

Dit kan je dus niet op de volgende manier oplossen

```
def functie(eerste, tweede=eerste):

    ...
```

omdat op het moment dat je de functie aan het definiëren bent, de parameter `eerste` nog niet naar een waarde verwijst. Dit is net alsof je een variabele gebruikt die nog niet gedefinieerd werd.

Je gebruikt best ook `None` als standaardwaarde, als de standaardwaarde die je eigenlijk wil toekennen een veranderlijk (*mutable*) gegevenstype heeft. Je schrijft dus beter niet

```
def functie(eerste, tweede=[]):

    # OPMERKING: in dit geval wordt er één enkele lege lijst aangemaakt op het
    #            ogenblik dat de functie gedefinieerd wordt; omdat lijsten
    #            mutable zijn, is het mogelijk dat deze lijst wordt aangepast
    #            telkens als de functie wordt aangeroepen; doorgaans is dit niet
    #            het gewenste gedrag

    ...
```

maar

```
def functie(eerste, tweede=None):

    # lege lijst instellen als standaardwaarde
    # OPMERKING: in dit geval wordt er bij elke aanroep van de functie een
    #            nieuwe lege lijst aangemaakt, specifiek voor die functie-aanroep
    if tweede is None:
        tweede = []

    ...
```

Als je aan een parameter een standaardwaarde wil toekennen die vastligt op het ogenblik dat je een functie definieert, en als die standaardwaarde een onveranderlijk (*immutable*) gegevenstype heeft (`int`, `bool`, `string`, `tuple`, ...) dan kan je die zonder problemen op de klassieke manier toekennen aan de parameter.

```
def functie(eerste, tweede=42):

    ...
```

Koprolkalender

De module `datetime`

De module datetime uit The Python Standard Library definieert een aantal nieuwe gegevenstypes die datums (`datetime.date` objecten) en tijdsintervallen (`datetime.timedelta` objecten) voorstellen. Hieronder alvast enkele voorbeelden.

```
>>> from datetime import date
>>> geboortedatum = date(1990, 10, 3)
>>> geboortedatum = date(day=3, month=10, year=1990)
>>> geboortedatum.day          # day is een eigenschap
3
>>> geboortedatum.month        # month is een eigenschap
10
>>> geboortedatum.year         # year is een eigenschap
1990
>>> geboortedatum.weekday()    # weekday is een methode !!
2
>>> vandaag = date.today()
>>> vandaag
datetime.date(2015, 11, 10)    # uitgevoerd op 11 oktober 2015
>>> from datetime import timedelta
>>> morgen = vandaag + timedelta(1)
>>> morgen
datetime.date(2015, 11, 11)
>>> verschil = morgen - vandaag
>>> type(verschil)
datetime.timedelta
>>> verschil.days
1
```

Specifieke info

In de voorstelling van `datetime.date` objecten worden de maanden chronologisch genummerd van 1 tot en met 12, waarbij januari de eerste maand (maand 1) is en december de laatste maand (maand 12). In deze opgave voegen we daaraan toe dat de dertiende maand terug de maand januari is, de veertiende maand de maand februari, enzoverder. Onderstaande tabel geeft een overzicht van wat we in deze opgave bedoelen met de maanden 12 tot en met 25.

origineel	nieuw	origineel	nieuw
12	12	19	7
13	1	20	8
14	2	21	9
15	3	22	10
16	4	23	11
17	5	24	12
18	6	25	1

Zoals je kunt vaststellen hebben we hier dus terug cyclisch gedrag: na de laatste maand komt terug de eerste maand. Daarbij komt de modulo operator (`%`) altijd handig van pas. Maar aangezien de maanden genummerd worden vanaf 1, en niet vanaf 0, kunnen we de modulo operator niet rechtstreeks gebruiken. Want `12 % 12` is gelijk aan `0`, terwijl de eerste maand (januari) genummerd is als `1`.

De oplossing bestaat er dus in om de nummering 1 tot en met 12 eerst om te zetten naar de nummering 0 tot en met 11, dan de modulo operator toepassen op deze nieuwe nummering, en vervolgens de nieuwe nummering terug omzetten naar de nummering 1 tot en met 12. Dit kan op de volgende manier gebeuren:

```
>>> maand = 15
>>> (maand - 1) % 12 + 1
3
>>> maand = 24
>>> (maand - 1) % 12 + 1
12
```

Coole serienummers

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
```

Functies toekennen aan variabelen

In Python zijn functies zelf objecten van het gegevenstype `function`, waardoor je ze net als andere objecten kunt toekennen aan een variable. Dat kan handig zijn als je twee stukken programmacode hebt die zeer gelijkaardig zijn, en enkel verschillen door het feit dat er in het ene stuk code een andere functie moet aangeroepen worden dan in het andere stuk code.

Denk bijvoorbeeld aan een programma waarin voor een gegeven woordenlijst eerst alle woorden moeten afgedrukt worden die de letter `a` bevatten, en daarna ook alle woorden die de letter `b` bevatten. Dit zou je op de volgende manier kunnen oplossen

```
>>> woorden = ['appel', 'banaan', 'bes']
>>>
```

```
>>> def bevat_a(woord):
...     return 'a' in woord
...
>>> def bevat_b(woord):
...     return 'b' in woord
...
>>> for woord in woorden:
...     if bevat_a(woord):
...         print(woord)
...
appel
banaan
>>> for woord in woorden:
...     if bevat_b(woord):
...         print(woord)
banaan
bes
```

De twee `for`-lussen in bovenstaand codefragment kunnen nu als volgt herschreven worden

```
>>> functie = bevat_a
>>> for woord in woorden:
...     if functie(woord):
...         print(woord)
...
appel
banaan
>>> functie = bevat_b
>>> for woord in woorden:
...     if functie(woord):
...         print(woord)
banaan
bes
```

Waardoor we twee keer exact dezelfde lus krijgen. Om duplicatie van code te vermijden (we programmeren liever niet twee keer dezelfde code), kunnen we dit herschrijven door een nieuwe lus te introduceren die over de twee functie loopt

```
>>> functies = [bevat_a, bevat_b]
>>> for functie in functies:
...     for woord in woorden:
...         if functie(woord):
...             print(woord)
appel           # genereerd door eerste iteratie (functie == bevat_a)
banaan
banaan          # genereerd door tweede iteractie (functie == bevat_b)
bes
```

Zelfbeschrijvende reeksen

Functies met een variabel aantal argumenten

Bij het definiëren van een functie leg je in principe vast hoeveel argumenten er aan de functie moeten doorgegeven worden. Dit aantal correspondeert met het aantal parameters die opgegeven worden bij de definitie van de functie. Aan onderstaande functie moeten bijvoorbeeld twee argumenten doorgegeven worden,

en zal de functie het resultaat van de optelling van de twee doorgegeven objecten teruggeven.

```
>>> def som(term1, term2):
...     return term1 + term2
...
>>> som(1, 2)
3
```

Stel echter dat we een functie willen schrijven waaraan men nul of meer argumenten moet kunnen doorgeven, waarop de functie het resultaat van de som van alle doorgegeven objecten moet teruggeven. Dit kan door een parameter te laten voorafgaan door een asterisk (`*`). Aan deze parameter zal een tuple toegekend worden dat alle argumenten bevat die positioneel werden doorgegeven bij het aanroepen van de functie en die niet aan andere parmaters werden toegekend.

```
>>> def som(*termen):
...     totaal = 0
...     for term in termen:
...         totaal += term
...     return totaal
...
>>> som(1, 2)
3
>>> som(1, 2, 3)
6
>>> som(1, 2, 3, 4)
10
```

In dit geval zullen alle argumenten die aan de functie `som` doorgegeven worden, gebundeld worden in een tuple dat wordt toegekend aan de lokale variabele `termen`. Het is ook mogelijk om andere parameters op te geven bij de definitie van de functie, zolang de parameter die voorafgegaan wordt door een asterisk als laatste parameter gedefinieerd wordt. Er kan ook slechts één parameter voorkomen die voorafgegaan wordt door een asterisk.

Hieronder definiëren we bijvoorbeeld een functie `som` waaraan minstens twee argumenten moeten doorgegeven worden. De functie geeft nog altijd de som van alle argumenten terug die aan de functie worden doorgegeven.

```
>>> def som(term1, term2, *termen):
...     totaal = term1 + term2
...     for term in termen:
...         totaal += term
...     return totaal
...
>>> som(1, 2)
3
>>> som(1, 2, 3)
6
>>> som(1, 2, 3, 4)
10
```