Python Development

Functies

Kristof Michiels

Wat zijn functies?

Wat zijn functies?

- Functies helpen om de complexiteit in een programma beheersbaar te houden
- Ze dienen meerdere doelen:
  - ze maken het mogelijk om code op te splitsen in kleinere units
  - ze laten ons toe code eenmaal te schrijven en vanop verschillende plaatsen aan te roepen wanneer nodig
  - ze laten toe om de onderdelen van onze toepassing individueel te testen

Wat zijn functies?

- Ze doen dat door een reeks statements af te zonderen voor later meermaals gebruik
- Een functie wordt uitgevoerd door ze aan te roepen vanuit je code
- Na het uitvoeren wordt teruggekeerd naar de locatie waar de functie werd aangeroepen en gaat de uitvoering van het programma verder

Wat zijn functies?

- We hebben in de eerste week reeds functies gebruikt: bvb input(), print(), int() en float()
- Het zijn functies die Python ons standaard biedt en kunnen vanuit elk Python programma worden aangeroepen
- Bovenstaande voorbeelden hebben één en slechts één helder doel: dat is ook hoe jouw functies horen te zijn!
- We leren in dit document onze eigen functies te schrijven en aan te roepen

De basis van functies

Basisvoorbeeld

```
def groet_gebruiker():
    """Toon een eenvoudige begroeting"""
    print("Hallo daar!")

groet_gebruiker()
```

- We definiëren functies met het trefwoord *def*, gevolgd door de naam van de functie en (voorlopig lege) haakjes en tot slot een dubbelpunt
- Op deze regel volgen (met telkens een insprong van 4 spaties) een reeks statements die zullen worden uitgevoerd wanneer de functie wordt aangeroepen
- De haakjes kunnen argumenten bevatten, dat is informatie die aan de functie van buitenaf wordt meegegeven

Basisvoorbeeld

```
def groet_gebruiker():
    """Toon een eenvoudige begroeting"""
    print("Hallo daar!")

groet_gebruiker()
```

- Merk in het vb op dat je in je code de functie moet aanroepen. Doe je dit niet, dan wordt ze nooit uitgevoerd
- Je kiest steeds voor een naam die helder beschrijft wat binnen de functie gebeurt
- Een goed gekozen functienaam bevat vaak een werkwoord (actie) en zelfstandig naamwoord
- Gebruik geen hoofdletters en scheid woorden door een underscore

Basisvoorbeeld

```
def groet_gebruiker():
    """Toon een eenvoudige begroeting"""
    print("Hallo daar!")

groet_gebruiker()
```

- Elke functie hoort een korte beschrijving te hebben van wat de functie doet
- Deze beschrijving komt onmiddellijk na de functie-definitie en gebruikt het docstring format
- Een goedgedocumenteerde functie zorgt ervoor dat andere programmeurs de functie kunnen gebruiken enkel en alleen door het lezen van de docstring

De basis van functies

```
def groet_gebruiker():
    """Toon een eenvoudige begroeting"""
    print("Hallo daar!")

for _ in range(5):
    groet_gebruiker()

print("De code (en het leven) gaan verder")
```

- Een functie kan zo vaak als nodig worden aangeroepen vanop verschillende plaatsen binnen de toepassing
- In bovenstaand voorbeeld een aantal keer binnen een loop. We gebruiken hier "_" als teller omdat we de for-lus louter als loop willen gebruiken en geen variabele teller nodig hebben binnen onze code

Informatie meegeven met parameters

Functies met parameters

- Onze voorbeeldfunctie miste flexibiliteit: stel dat je een andere boodschap wil meegeven?
- We kunnen de mogelijkheden en de flexibiliteit vergroten door het voorzien van één of meer argumenten
- De functie ontvangt die argumentwaarden in de vorm van parameter-variabelen die toegevoegd worden binnen de haakjes wanneer de functie wordt gedefinieerd
- Het aantal parameter-variabelen die in de functie staan beschreven geven aan hoeveel argumenten moeten worden meegegeven wanneer de functie wordt aangeroepen

Functies met parameters

```
def groet_gebruiker(naam_gebruiker):
    """Toon een gepersonaliseerde begroeting"""
    print(f"Hallo daar {naam_gebruiker.title()}!")

groet_gebruiker("Anneleen")
```

- In bovenstaand voorbeeld werd één parameter toegevoegd: *naam_gebruiker*
- Eventuele meerdere parameters worden gescheiden door een komma
- De functie wordt hier aangeroepen met het argument "Anneleen"
- Binnen de functie kunnen de waarden van de parameter variabelen worden gebruikt wanneer nodig

Functies met parameters

```
def teken_kader(breedte, hoogte):
    """Deze functie tekent een kader, vanaf een breedte x hoogte van 2x2 """
    if breedte < 2 or hoogte < 2:
        print("Ik kan de rechthoek niet tekenen: breedte of hoogte zijn te klein")
        quit()
    print("#" * breedte)
    for i in range(hoogte - 2):
        print("#" + " " * (breedte - 2) + "#")
    print("#" * breedte)

teken_kader(8, 6)
teken_kader(4, 4)
```

- In dit voorbeeld moeten 2 argumenten worden voorzien telkens de functie wordt aangeroepen omdat de functie-definitie dit vereist

Functies met parameters

```
def teken_kader(breedte, hoogte, teken_kader, teken_vulling):
    """Deze functie tekent een kader en gebruikt vier argumenten """
    print(teken_kader * breedte)
    for i in range(hoogte - 2):
        print(teken_kader + teken_vulling * (breedte - 2) + teken_kader)
    print(teken_kader * breedte)

teken_kader(8, 6, "*", "/")
```

- Wanneer de functie wordt uitgevoerd zal de waarde van het eerste argument worden gekoppeld aan de eerste parameter, en de waarde van het tweede argument aan de tweede parameter
- Wil je de functie nog meer flexibiliteit geven, dan kan je meer parameters gaan gebruiken (in dit vb: 4)

Default-waarden

- Wil je de functie aanroepen, dan zal je telkens 4 parameters moeten voorzien
- Wanneer bepaalde waarden frequent worden gebruikt dan kan je standaard (*default*) waarden voor parameters meegeven aan de functie-definitie. Parameters met default komen steeds achteraan
- De functie kan vanaf dan aangeroepen worden met 2, 3 of 4 argumenten. Bij 3 of 4 argumenten worden de default waarden overschreven.

```
def teken_kader(breedte, hoogte, teken_kader="#", teken_vulling=" "):
    ...

teken_kader(8, 6)
teken_kader(8, 6, "@")
teken_kader(8, 6, "#", ".")
```

Positionele vs trefwoord-argumenten

```
teken_kader(breedte=8, hoogte=6)
teken_kader(breedte=8, hoogte=6, teken_kader="@")
teken_kader(breedte=8, hoogte=6, teken_kader="#", teken_vulling=".")
```

- Een trefwoord-argument is een naam-waarde-paar dat je meegeeft aan een functie
- Je koppelt daarmee onmiddelijk een waarde aan een parameter
- Je geeft explicieter aan wat bij wat hoort en moet de volgorde van de parameters niet meer respecteren

Variabelen in functies en return-waarden

Variabelen in functies

- Variabelen kunnen ook gecreëerd worden binnen een functie
- Deze bestaan dan enkel binnen de functie, wanneer de functie wordt uitgevoerd
- De variabele houdt op te bestaan wanneer de functie eindigt en kan na afloop niet meer gebruikt worden
- We noemen dit lokale variabelen en spreken van een lokale *scope* (of reikwijdte)
- In onderstaand voorbeeld kan de variabele exponent niet opgevraagd worden buiten de functie

```
def bereken_macht(grondtal)
    exponent = 3
    print(f"De {exponent}-de macht van {grondtal} is {grondtal ** exponent}")
```

Return-waarden

- De statements in de functies die we hebben gezien zorgden met de print()-functie voor een resultaat
- We konden met argumenten het resultaat van de functie beïnvloeden, maar er was geen verdere communicatie met de code buiten de functie
- Soms zal je als resultaat van een functie een resultaat berekenen dat later in het programma en buiten de functie zal worden gebruikt

Return-waarden: een voorbeeld

```
def groet_gebruiker(naam_gebruiker):
    """Toon een gepersonaliseerde begroeting"""
    return f"Hallo daar {naam_gebruiker.title()}!"

print(groet_gebruiker("Anneleen"))
```

- We doen dit door de functie een return waarde te laten teruggeven met het return trefwoord gevolgd door de waarde die wordt teruggegeven
- Wanneer het return statement uitgevoerd is eindigt de functie meteen en wordt de controle teruggegeven aan de locatie waar de functie werd aangeroepen

Return-waarden

- Functies die waarden teruggeven met return staan vaak aan de rechterkant van een toekennings-statement (bvb getal = mijn_functie())
- Ze kunnen evenwel ook in andere contexten functioneren waarin een waarde nodig is: bvb bij een voorwaarde in een if-statement of een while-loop, of als argument voor een andere functie
- Een functie die geen waarde teruggeeft moet het return trefwoord niet gebruiken
- Je kan het return trefwoord evenwel gebruiken zonder waarde om ergens een einde van een functie te forceren
- Elke functie kan meerdere return statements bevatten (al dan niet met een waarde). Van zodra de interpreter een return tegenkomt bij het uitvoeren van een functie, stopt de functie

Functies gebruiken uit andere bestanden

Functies gebruiken uit andere bestanden

- Functies helpen je je code te structuren en overzichtelijk te houden
- Bij grotere programma's worden functies vaak in afzonderlijke bestanden (we noemen deze modules) bijgehouden
- Wil je ze gebruiken dan importeer je de module of de functie in je programma
- Je maakt een functie beschikbaar binnen je programma met een import-statement
- Je gaat dit binnenkort ook vaak doen voor functies uit libraries en modules die je niet zelf geschreven hebt

Een volledige module importeren

In onderstaand voorbeeld hebben we twee bestanden: kleuren.py en gebruik_kleuren.py

```
# kleuren.py
def geef_kleur()
    return ...

def geef_complementaire_kleur(primaire_kleur)
    return ...
```

```
# gebruik_kleuren.py
import kleuren
print(kleuren.geef_complementaire_kleur("geel"))
```

Een volledige module importeren

- Een module is een bestand dat eindigt op .py. Hier wordt *kleuren.py* gebruikt als module
- Om de functionaliteit van de module beschikbaar te maken in *gebruik_kleuren.py* voegen we een import statement toe bovenaan het bestand
- We roepen de externe functies aan door eerst te verwijzen naar de module, en dan met een dot-notatie de functie te vernoemen: bvb. kleuren.geef_complementaire_kleur()

Specifieke functies importeren

- Het is ook mogelijk slechts één of enkele functie(s) te importeren. Je scheidt ze door een komma
- Met deze syntax hoef je de dot-notatie niet te gebruiken wanneer je de functie aanroept
- Dit is de beste en meest efficiënte aanpak

```
from module_naam import functie_naam
```

```
from module_naam import functie_0, functie_1, functie_2
```

```
# gebruik_kleuren.py
from kleuren import geef_complementaire_kleur
print(geef_complementaire_kleur("geel"))
```

"as" gebruiken om de functie een alias te geven

- Als de functie-naam conflicteert met een andere functie die je gebruikt (of te lang is) dan kan je gebruik maken van een alias:

```
from module_naam import functie_naam as fie
```

```
# gebruik_kleuren.py
from kleuren import geef_complementaire_kleur as ck
print(ck("geel"))
```

"as" gebruiken om de module een alias te geven

- Je kan ook een alias geven aan een module en zo tot kortere notaties komen:

```
import module_naam as md
```

```
# gebruik_kleuren.py
import kleuren as kl
print(kl.geef_complementaire_kleur("geel"))
```

Alle functies in een module importeren

- Je kan aangeven aan Python om elke functie in de module te importeren door gebruik te maken van de asterisk operator (*)
- Voordeel hier is dat je de dot-notatie niet hoeft te gebruiken
- Te vermijden bij gebruik van grote modules

```
from module_naam import *
```

```
# gebruik_kleuren.py
from kleuren import *
print(geef_complementaire_kleur("geel"))
```

Verhinderen dat bij importeren code in een module wordt uitgevoerd

- Doe je door de code in de module bvb in een main()-functie te stoppen
- Met een if-statement kan je dan checken of het bestand daadwerkelijk autonoom werd uitgevoerd, of dat het werd geïmporteerd
- Deze structuur moet toegepast worden telkens je een programma maakt dat functies bevat die mogelijk door een ander bestand zullen gebruikt worden

```
if __name__ == "__main__":
      main()
```