Python Programming

Control flow: beslissingsstructuren en loops

Kristof Michiels

Inhoud

- Met control flow-elementen neem je de volgorde waarin en de voorwaarden waaronder de instructies van je programma wordt uitgevoerd in handen
- De elementen die je hier tot je beschikking hebt zijn beslissingsstructuren en loops

Inhoud

- beslissingsstructuren
  - if-(else-)statements
  - expressies en operators
  - if(-elif)-else-statements
  - structurele patroonherkenning
- loops
  - for-loop
  - while-loop
  - de range()-functie

Beslissingsstructuren

if-statements

- Een if-statement bevat een voorwaarde, gevolgd door een reeks statements die mogelijk kunnen worden uitgevoerd. Dat deze statements tot de if-blok behoren wordt aangegeven door inspringingen
- De voorwaarde wordt geëvalueerd en indien *True* worden de statements effectief uitgevoerd; indien *False* worden deze statements overgeslagen
- Vervolgens gaat het programma verder met de instructies die onder het if-statement staan

```
getal = 24
if getal % 2 == 0:
    print(f"Het getal {getal} is deelbaar door 2")
    # hier kunnen nog extra statements staan
print("En het programma loopt door...")
```

De voorwaarde binnen een if-statement

- De voorwaarde, die kan gaan van eenvoudig tot complex, evalueert in *True* of *False*. We noemen die voorwaarde een Booleaanse expressie, genoemd naar wiskundige George Boole
- Een expressie is een reeks waarde-elementen in combinatie met operators, wat een nieuwe waarde oplevert
- De operators handelen op de waarde-elementen door berekeningen, vergelijkingen of andere bewerkingen uit te voeren. Er bestaan verschillende soorten operators en dus ook expressies

```
if naam == "Kristof" or naam == "Bart":  # vergelijkende én logische operators
    print("Leuk, je bent een Kristof of een Bart!")

if naam == "David" and leeftijd == 28:  # vergelijkende én logische operators
    print("Wat een toeval: een 28-jarige David!")
```

Vergelijkende operators

```
print(4 == 5) # False
print(4 < 5) # True
print(5 >= 2) # True
print(5 <= 5) # True
print(5 > 2) # True
```

- Het resultaat van een vergelijkende expressie met deze operators is een boolean
- Willen we twee waarden met elkaar vergelijken, dan gebruiken we een dubbel gelijkheidsteken

Logische operators

```
print(True and True) # True
print(True and False) # False
print(True or False) # True
print(False or False) # False
print(not True)
print(3 < 5 and 6 > 2) # True
```

- Het betreft hier de Engelse termen and, or en not
- Ze laten toe om gecombineerde expressies te evalueren
- De not-operator keert de boolean om: True wordt False en omgekeerd

Identiteitsoperators

```
string_1 = "Aap"
string_2 = "Noot"
string_3 = string_1
print(string_1 is string_2) # False
print(string_1 is not string_2) # True
print(string_1 is string_3) # True
```

- Identiteitsoperators (identity operators) worden gebruikt om objecten met elkaar te vergelijken
- Niet om te zien of ze gelijkwaardig zijn, maar of ze hetzelfde object zijn, met dezelfde geheugenlocatie
- Is: als de expressie True teruggeeft zijn x en y hetzelfde object
- Is not: als de expressie True teruggeeft zijn x en y niet hetzelfde object

Lidmaatschap operators

```
print(1 in [1,2,3,4,5]) # True
print(10 not in [1,2,3,4,5]) # True
print("Python" in "Mijn favoriete programmeertaal is Python") # True
```

- Lidmaatschap operators (membership operators) worden gebruikt om te evalueren of een reeks aanwezig is in een object
- In: is de reeks aanwezig in het object
- Not in: is de reeks niet aanwezig in het object

If-else-statements

- Een if-else-statement heeft een extra else-gedeelte (zonder voorwaarde!) met eigen blok instructies
- Wanneer de if-voorwaarde wordt geëvalueerd zal bij *True* de eerste blok code worden uitgevoerd, en bij *False* de tweede. Er zal steeds maar één blok code worden uitgevoerd
- Je gebruikt een if-else-statement ipv 2 if-statements als de voorwaarden elkaar uitsluiten. Dit is efficiënter: slechts één voorwaarde moet worden gecontroleerd

```
getal = int(input("Geef een geheel getal aub: "))
if getal > 0:
    boodschap = "groter dan 0"
else:
    boodschap = "kleiner dan of gelijk aan 0"
print(f"Het getal is {boodschap}")
```

Meerdere if-statements

```
getal = int(input("Geef een geheel getal aub: "))
if getal > 0:
    boodschap = "groter dan 0"
if getal <= 0:
    boodschap = "kleiner dan of gelijk aan 0"
print(f"Het getal is {boodschap}")
```

- Bovenstaand voorbeeld vraagt een geheel getal aan de gebruiker en gebruikt twee if-statements om na te gaan of dat getal groter of kleiner/gelijk is aan 0
- We maken hier tweemaal gebruik van een if-statement met twee voorwaarden die elkaar uitsluiten. Dit is toegelaten, maar het kan efficiënter, namelijk met een if-else statement...
- Let op! Meerdere if-statements kunnen perfect ok zijn indien de voorwaarden elkaar niet uitsluiten

If-elif-else-statements

```
getal = int(input("Geef een geheel getal aub: "))
if getal > 0:
    boodschap = "groter dan 0"
elif getal < 0:
    boodschap = "kleiner dan 0"
else:
    boodschap = "gelijk aan 0"
print(f"Het getal is {boodschap}")
```

- Gebruik if-elif-else wanneer je meer dan twee mogelijkheden wil bieden
- Begint met een if-gedeelte gevolgd door één of meer elif-gedeeltes, gevolgd door een else (zonder voorwaarde). Elk gedeelte beschikt over een door 4 spaties voorafgegaan codeblok

If-elif-else-statements

```
leeftijd = 16
if leeftijd < 4:
    ticketprijs = 2
elif leeftijd < 12:
    ticketprijs = 10
elif leeftijd < 26:
    ticketprijs = 18
else:
    ticketprijs = 26
print(f"Jouw toegang tot de zoo kost ${ticketprijs}.")
```

- De voorwaarden worden van boven naar onder één voor één geëvalueerd: van zodra een *True* is gevonden wordt de betreffende codeblok uitgevoerd en wordt de rest van het statement overgeslagen
- Indien geen *True* wordt gevonden voert de interpreter de code van het else-blok uit

If-elif-statements

```
leeftijd = 16
if leeftijd < 4:
    ticketprijs = 2
elif leeftijd < 12:
    ticketprijs = 10
elif leeftijd < 26:
    ticketprijs = 18
print(f"Jouw toegang tot de zoo kost ${ticketprijs}.")
```

- Het else-gedeelte op het einde mag worden weggelaten
- Het is dan mogelijk dan van alle codeblokken geen enkele wordt uitgevoerd, omdat geen enkele voorwaarde in *True* resulteert
- Bovenstaand voorbeeld is ok, maar wat met een bezoeker van 26 of ouder?

Geneste if-instructies

```
leeftijd, sociaal_tarief  = 45, True
if leeftijd < 4:
    ticketprijs = 0
elif leeftijd < 12:
    ticketprijs = 10
elif leeftijd < 26:
    ticketprijs = 18
else:
    if sociaal_tarief:
        ticketprijs = 12
    else:
        ticketprijs = 24
```

De codeblokken binnen de if-*-statements kunnen (zo goed als) elke Python-code bevatten. Dat kunnen ook andere if, if-else, if-elif of if-elif-else statements zijn. We noemen dit geneste if-instructies.

Structurele patroonherkenning

- Structural pattern matching is een beslissingsstructuur - nieuw sinds Python 3.10
- Hierbij wordt een bepaalde structuur vergeleken met verschillende modellen of patronen om een overeenkomst te vinden en daarop een actie uit te voeren
- Biedt 2 nieuwe keywords: match en case. Beslissingen op basis van data die matchen met bepaald patroon
- Kan gezien worden als een meer geavanceerde versie van de switch/case uit andere programmeertalen
- Is een alternatief voor lange if-else-blokken
- Er is veel geavanceerder pattern matching gebruik mogelijk!

structurele patroonherkenning

```
taal = "JS"
match taal:
    case "JavaScript" | "JS":
        print("Je koos JavaScript")
    case "Python":
        print("Python, zeer goede keuze!")
    case _:
        print(f"Je koos {taal}")
```

- Dit noemen we literal matching d.w.z. matchen met string-literals
- Je kan verschillende cases combineren met | (“or”). De _ gedraagt zich als 'default' en is optioneel
- Enkel de eerste matchende case wordt uitgevoerd. Daarna verlaten we de codeblok (dus break niet nodig)

Structurele patroonherkenning

```
getal = 13
match(getal % 3, getal % 5):
    case (0,0):
        print("fizzbuzz")
    case (_,0):
        print("buzz")
    case (0,_):
        print("fizz")
    case _:
        print(str(getal))
```

- Hier match je de elementen in een tuple gelijktijdig met mogelijke patronen. De mogelijkheden zijn krachtig!
- Voor mensen die dieper willen graven, in zaken zoals pattern grouping, pattern sequence en waarden: hier vind je de officiële documentatie rond PEP 634.

Loops

Herhalen met een while- of for-loop

- Indien je een bepaalde taak meerdere keren wil uitvoeren dan kan je gebruik maken van één van Pythons beschikbare loop-constructies: een while-loop of een for-loop
- Beiden laten toe dezelfde groep statements meerdere keren uit te voeren
- Gebruik je ze efficiënt dan kan je complexe zaken uitvoeren (zoals berekeningen) in slechts een zeer beperkt aantal statements

De while-loop

```
teller = 1
while teller < 10:
    print(teller)
    teller = teller + 1
```

- While-loops gebruik je om één of meerdere statements een aantal keer te laten uitvoeren, zo lang de Booleaanse voorwaarde die je aan de loop meegeeft True is. Wordt de voorwaarde False, dan stopt de loop
- Je moet zorgen dat in de statements binnen de loop een mechanisme aanwezig is dat de voorwaarde na verloop van tijd doet veranderen in False. Doe je dit niet dan beland je in wat we noemen een infinite loop. De loop gaat dan continue door en je programma blijft vastzitten

De while-loop: voorbeeld

```
getal = int(input("Geef een geheel getal aub (0 om de app te verlaten): "))
while getal != 0:
    if getal > 0:
        print("Dat is een positief getal")
    else:
        print("Dat is een negatief getal")
    getal = int(input("Geef een geheel getal aub (0 om de app te verlaten): "))
```

In dit voorbeeld zie je hoe de gebruikersinput binnen de loop de voorwaarde kan wijzigen na elke evaluatie.

De while-loop

- Zoals je hebt kunnen vaststellen laat een while-loop een codeblok (indentatie met 4 spaties) uitvoeren zo lang een voorwaarde resulteert in *True*
- Wanneer de laatste regel van de codeblok is bereikt wordt teruggekeerd naar het begin van de loop en wordt de voorwaarde opnieuw geëvalueerd, en indien *True*, de codeblok opnieuw uitgevoerd
- Dit duurt tot de loop-voorwaarde resulteert in *False*. Als dit gebeurt wordt de codeblok niet meer uitgevoerd en gaat het programma verder met het eerste statement na de loop
- De loop-voorwaarde kan wijzigen door gebruikersinput of door code binnen de loop-codeblok. Zorg er elk geval voor dat de voorwaarde ooit *False* teruggeeft, anders belandt je programma in een *infinite loop*

De for-loop

```
gemeten_temperaturen = [14,12,15,9]
for meting in gemeten_temperaturen:
    print(meting)
```

- For-loops doorlopen wat we in Python iterables noemen. Een list is bijvoorbeeld zo'n iterable
- Binnen een for loop kan je dus voor elk element in een lijst een bepaalde actie beschrijven
- Wanneer de voorwaarden tijdens het doorlopen van de iterable moeten kunnen wijzigen gebruik je beter een while-loop
- Elk element in de collectie wordt gekopieerd naar een variabele vooraleer de codeblok voor dit element wordt uitgevoerd. De variabele kan gebruikt worden in de codeblok zoals elke andere variabele

De for-loop: voorbeelden

```
for i in range(5):
    print(i)
```

```
lijst = ['aap', 'noot', 'peer']
for i in lijst:
    print(i)
```

```
getal = int(input("Geef een geheel getal: "))
print(f"De veelvouden van 3 tot en met {getal} zijn:")
for i in range(3, getal + 1, 3): # we gaan zodadelijk in op de range()-functie!
    print(i)
```

De range()-functie

```
veelvouden_van_drie = [3, 6, 9, 12, 15] # is hetzelfde als range(3,16,3)
for element in range(3, 16, 3):
    quotient = element / 3
    print(f"{element} gedeeld door 3 is {int(quotient)}.")
```

- Behoort tot de in Python ingebouwde functies, zoals int(), float(), set(), reverse(), filter(), map()
- Deze belangrijke Python bouwstenen laten ons toe vaak voorkomende taken uit te voeren in onze code
- Laat toe om een reeks (list) van gehele getallen op te stellen binnen een bepaald bereik (vandaar 'range')
- Het aantal argumenten die je meegeeft bepaalt de opbouw van de reeks
- De meest gebruikte use case voor range() is de for-loop

De range()-functie

- Roep je deze functie aan met 1 argument dan start de range met 0 en eindigt met het argument -1

```
for getal in range(10): # [0,1,2,3,4,5,6,7,8,9]
  print(getal)
```

- Wanneer twee argumenten zijn meegegeven dan start de range met het eerste argument en eindigt met het tweede argument -1

```
for getal in range(2,5): # [2,3,4]
  print(getal)
```

De range()-functie

- Range() kent ook nog een derde argument, de step (een geheel getal). Is deze groter dan 0 dan begint de range bij het eerste argument en loopt tot argument2 - 1, telkens in sprongen gelijk aan de stepwaarde

```
for getal in range(10,20,2): # [10,12,14,16,18]
  print(getal)
```

- Een lege range wordt teruggegeven als het eerste argument groter is dan het tweede. De codeblok in de loop wordt in dit geval nooit uitgevoerd

```
for getal in range(5,2): # []
  print(getal)
```

De range()-functie

- Een negatieve stepwaarde zorgt ervoor dat een collectie van afnemende waarden ontstaat. De step-waarde mag geen 0 zijn

```
for getal in range(0, -4, -1): # [0, -1, -2 en -3]
  print(getal)
```

- Nog meegeven dat je de elementen in een range kan benaderen via het index-getal (zoals bij een list)
- Je kan zelfs de slicing notatie gebruiken: range(6)[2:5] (zien we bij de les over lists)

Geneste loops

- De statements binnen de codeblok van een loop kunnen op hun beurt een loop bevatten. We noemen dit geneste loops
- Elk type loop kan genest zitten binnen een ander type buitenloop

```
boodschap = input("Geef een boodschap (laat leeg om te stoppen): ")
while boodschap != "":
    aantal = int(input("Hoe vaak moet dit herhaald worden? "))
    for i in range(aantal):
        print(boodschap)
    boodschap = input("Geef een boodschap (laat leeg om te stoppen): ")
```

Python OOP / development: control flow - kristof.michiels01@ap.be