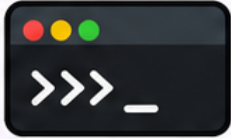


1 De interpreter

- Voert Python-code uit.
- Start via een terminal.
- Opdrachten invoeren of script uitvoeren.



2 Starten

- Open de terminal.
- Typ de startopdracht.
- Die verschilt per installatie en besturingssysteem.



3 Interactief werken

- Voer opdrachten één voor één in.
- De prompt >>> toont de interactieve modus.



4 Script uitvoeren

- Bewaar code in een Python-bestand.
- Laat de interpreter het bestand uitvoeren.



5 Code lezen

- De interpreter leest Python-code.
- Daarna voert hij de code uit.



6 Terminal

- Een tekstvenster voor opdrachten.
- Start de interpreter vanuit dit venster.



7 Twee werkwijzen

- Interactieve opdrachten.
- Uitvoering van een scriptbestand.



8 Onthoud

- De startopdracht kan verschillen.
- Volg de installatie van jouw systeem.



Python Introductie en Runtime Omgeving

8. Oefenroute: van prompt naar project



Gebruik deze korte route om de verschillen te ervaren: test eerst één regel, maak daarna een script en geef dat script een eigen virtuele omgeving.

- REPL: voer `print('Hallo!')` in en bekijk de uitvoer.
- Script: zet dezelfde regel in `hallo.py` en voer het bestand uit.
- Omgeving: maak in de projectmap `.venv` aan en activeer die.
- Sluit af met `deactivate`; je scriptbestand blijft in je projectmap staan.



Onthoud

REPL voor directe experimenten; een `.py`-bestand voor bewaarde code; `.venv` voor projectgebonden pakketten.

7. Een venv aanmaken en gebruiken



Voer de opdrachten uit in de terminal vanuit je projectmap. Activeer de omgeving voordat je projectpakketten installeert of je programma start.

- Maak de omgeving: `python -m venv .venv`.
- Windows PowerShell: `.venv\Scripts\Activate.ps1`.
- macOS of Linux (bash/zsh): `source .venv/bin/activate`.
- Typ `deactivate` om de actieve omgeving te verlaten.



Opdracht verschilt?

Als `python` niet wordt herkend, probeer op macOS/Linux `python3` of op Windows `py` voor de maakopdracht.

6. Wat is een virtuele omgeving?



Een virtuele omgeving isoleert de Python-pakketten van een project in een eigen omgeving. Zo kunnen projecten hun afhankelijkheden afzonderlijk beheren.

- Maak per project een omgeving, vaak met de mapnaam `.venv`.
- De module `venv` uit de standaardbibliotheek maakt de omgeving aan.
- De omgeving gebruikt de Python-installatie waarmee je de maakopdracht uitvoert.



Waarom gebruiken?

Projecten kunnen verschillende pakketversies nodig hebben. Een eigen omgeving helpt die afhankelijkheden gescheiden te houden.

1. De Python-interpreter



De interpreter voert Python-code uit. Je start hem vanuit een terminal; daarna kun je opdrachten invoeren of een scriptbestand laten uitvoeren.

- De interpreter leest Python-code en voert die uit.
- Een prompt zoals `>>>` geeft aan dat je in de interactieve modus zit.
- De opdracht om Python te starten kan per installatie en besturingssysteem verschillen.

2. De interactieve prompt (REPL)



In de REPL voer je een opdracht in en zie je direct het resultaat. REPL staat voor lezen, uitvoeren, afdrucken en herhalen.

- Start Python vanuit je terminal om de prompt te openen.
- Typ bijvoorbeeld `2 + 3` en druk op Enter.
- De interpreter toont `5`; voer daarna een volgende opdracht in.
- Typ `quit()` om de sessie af te sluiten.



Handig voor

Kleine experimenten, snelle berekeningen en het uitproberen van één regel code.

3. Een scriptbestand maken



Een script is een tekstbestand met Python-code. Sla het op met de extensie `.py` en voer het uit met de Python-interpreter.

- Maak bijvoorbeeld `hallo.py`.
- Zet erin: `print('Hallo!')`.
- Sla het bestand op en voer het vanuit de terminal uit met `python hallo.py` of een passende Python-opdracht voor jouw installatie.

4. Script uitvoeren: stappen



Voer een script uit vanuit de terminal in de map waar het bestand staat, of geef het bestandspad op.

1. Sla je code op als `hallo.py`.
2. Open een terminal en ga naar de map met het bestand.
3. Voer `python hallo.py` uit; gebruik zo nodig `python3 hallo.py` of `py hallo.py`.
4. Lees de uitvoer of foutmelding en pas je code aan.



Als het niet werkt

Controleer de bestandsnaam, de huidige map en of de Python-opdracht beschikbaar is in je terminal.



1 Namen en waarden



Een variabele is een naam die naar een object verwijst. Met = koppel je een naam aan een waarde. De naam zelf is geen vast getypeerd vakje: het object heeft een type.

```
aantal = 3
kleur = 'blauw'
type(aantal)
```

- `aantal = 3` koppelt `aantal` aan een geheel getal.
- `kleur = 'blauw'` koppelt `kleur` aan een tekenreeks.
- `type(aantal)` geeft het type van het object terug.

2 Dynamische typering



Python bepaalt tijdens het uitvoeren welk type een object heeft. Een naam kan later naar een object van een ander type verwijzen; de waarde verandert daardoor niet vanzelf van type.

```
waarde = 7
waarde = 'zeven'
type(waarde)
```

- `waarde = 7`
- `waarde = 'zeven'`
- Na de tweede toewijzing verwijst `waarde` naar een tekenreeks.
- Gebruik `type(waarde)` om het huidige type te bekijken.

Let op — Dynamische typering betekent niet dat Python elke combinatie van waarden en bewerkingen accepteert.

3 Toewijzen is een naam koppelen



Een toewijzing bindt een naam aan een object. Als meerdere namen hetzelfde object aanduiden, kan dat gevolgen hebben wanneer het object veranderbaar is, zoals een lijst.

```
eerste = [1, 2]
tweede = eerste
tweede.append(3)
```

- `eerste = [1, 2]` maakt een lijst en koppelt die aan `eerste`.
- `tweede = eerste` laat ook `tweede` naar diezelfde lijst verwijzen.
- `tweede.append(3)` is zichtbaar via beide namen.

Niet verwarren — Een toewijzing kopieert een veranderbaar object niet automatisch.

4 Toewijzingsoperatoren



Gebruik = voor een gewone toewijzing. Samengestelde toewijzingen combineren een bewerking met toewijzing; bij veranderbare objecten kan de werking verschillen van een nieuwe toewijzing.

```
teller = 4
teller += 1
teller -= 2
teller *= 3
teller /= 2
items += [3]
```

- `teller = 4`
- `teller += 1` is vergelijkbaar met `teller = teller + 1`.
- `teller -= 2`, `teller *= 3` en `teller /= 2` zijn andere voorbeelden.
- `items += [3]` werkt de lijst doorgaans ter plekke bij.

Vergelijk zorgvuldigheid — Bij onveranderlijke getallen levert += een nieuwe waarde op; bij lijsten kan += de bestaande lijst aanpassen.

Variabelen, toewijzingen en geheugenreferenties in Python

8 Pas het toe en controleer



Volg bij code met namen en toewijzingen telkens welke objecten worden gemaakt, welke namen ernaar verwijzen en of een bewerking een object wijzigt of een nieuwe waarde oplevert.

- Schrijf eerst de toewijzing op.
- Bepaal het type van de waarde met `type(naam)`.
- Controleer of een tweede naam hetzelfde object deelt.
- Gebruik duidelijke namen volgens de Python-stijlconventies.

Kort voorbeeld — `getal = 5`; `alias = getal`; `getal += 2`. Daarna is `getal` 7 en blijft `alias` 5: de toewijzing bindt `getal` aan een nieuwe integerwaarde.

7 Referenties en opruimen



Python beheert objecten die niet meer nodig zijn. CPython gebruikt referentietelling en aanvullende cycli-opruiming; vertrouw niet op een exact opruimmoment of op referentietelling als universeel gedrag van elke Python-implementatie.

- Een naam koppelen of verwijderen kan het aantal verwijzingen beïnvloeden.
- `del` naam verwijdert de naam, niet noodzakelijk direct het object.
- Objecten die alleen naar elkaar verwijzen kunnen een cyclus vormen.
- De garbage collector kan onbereikbare objecten opruimen.

Belangrijk onderscheid — Referentietelling is een implementatiedetail van CPython, geen belofte dat elk object onmiddellijk wordt vrijgegeven.

6 Objecten en geheugenreferenties



In Python verwijzen namen naar objecten. Als je een waarde aan een tweede naam toewijst, kan die naam hetzelfde object aanduiden. `id()` geeft tijdens de levensduur van een object een identificerende waarde terug; gebruik die vooral om te leren, niet als vaste geheugenlocatie.

```
x = [10, 20]
y = x
x is y
y = x[:]
```

- `x = [10, 20]`
- `y = x`
- `x is y` is `True`: beide namen verwijzen naar hetzelfde object.
- `y = x[:]` maakt voor deze lijst een nieuwe, ondiepe lijst.

5 Namen volgens PEP 8



Kies beschrijvende namen die in context duidelijk maken wat een waarde voorstelt. PEP 8 is de Python-stijlgids; volg de gangbare conventies voor leesbare code.

- Variabelen en functies: `snake_case`, zoals `totaal_bedrag`.
- Klassen: `CapWords`, zoals `Bestelling`.
- Constanten: hoofdletters met underscores, zoals `MAX_POGINGEN`.
- Begin een naam niet met een cijfer; gebruik geen Python-sleutelwoord als naam.

1 Int en grote gehele getallen

-42
0 7

Gebruik int voor gehele getallen zonder decimalen.

- Voorbeelden: 7, 0 en -42.
- Python-int kan zeer grote gehele getallen bevatten; de praktische grens is het beschikbare geheugen.
- Een int is niet beperkt tot een vaste 32- of 64-bits breedte.

2 Float en binaire opslag

3.14
-0.5

Gebruik float voor getallen met een fractioneel deel. Python-floats gebruiken doorgaans IEEE 754 binary64 (dubbele precisie).

- Voorbeeld: 3.14 of -0.5.
- Een float wordt binair opgeslagen; veel decimale breuken zijn daardoor benaderingen.

3 Precisie en representatiefouten



Een kleine afwijking kan ontstaan doordat de opgeslagen binaire waarde niet exact gelijk is aan de ingevoerde decimale breuk.

- 0.1 is in binair niet exact weer te geven.
- Dit is geen fout in de Python-syntaxis.
- Vergelijk benaderende waarden waar passend met `math.isclose()`.

Let op: Rond niet alleen voor weergave af als je de onderliggende nauwkeurigheid nodig hebt.

8 Keuzehulp en beginnerstips

Kies het type op basis van wat je waarde voorstelt en hoe precies je ermee moet rekenen.

Wat voor numerieke waarde verwerk je?

Een geheel aantal zonder fractie

Gebruik int; grote gehele getallen worden begrensd door beschikbaar geheugen, niet door een vaste 32- of 64-bits grens.

- Gehele aantallen of exacte gehele waarden: int.
- Metingen of berekeningen met decimalen: vaak float; houd rekening met benadering.
- Vergelijk floats benaderend met `math.isclose()` waar passend.
- `round()` verandert de afgeronde waarde; rond niet voortijdig af als nauwkeurigheid telt.

Een waarde met een fractie of meting

Gebruik vaak float en houd rekening met binaire benadering; vergelijk zo nodig met `math.isclose()`.

Primitieve datatypen: integers en floats

4 Rekenvoorbeeld



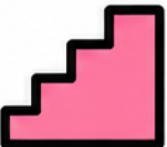
De binaire benadering kan zichtbaar worden bij een vergelijking.

`0.1 + 0.2 == 0.3` → False

`math.isclose(0.1 + 0.2, 0.3)`

Gebruik bijvoorbeeld `math.isclose(0.1 + 0.2, 0.3)` voor een benaderende vergelijking.

7 floor(), ceil() en trunc()



Deze functies behandelen het fractionele deel op verschillende manieren. Voor $x = -2.7$:

- `math.floor(x)` geeft de grootste integer $\leq x$: -3.
- `math.ceil(x)` geeft de kleinste integer $\geq x$: -2.
- `math.trunc(x)` verwijdert het fractionele deel richting nul: -2.

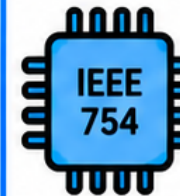
6 round() en ties-to-even



`round(x, ndigits)` rondt af naar de dichtstbijzijnde waarde. Bij een exacte gelijke afstand wordt ties-to-even gebruikt: het laatste cijfer wordt even.

- `round(2.5)` geeft 2.
- `round(3.5)` geeft 4.
- Een decimale waarde die er precies halverwege uitziet, hoeft door binaire benadering intern niet exact halverwege te liggen.

5 IEEE 754 binary64



Een eindig aantal significante binaire bits kan veel decimale waarden slechts benaderen.

- Python-floats zijn doorgaans dubbele-precisie binary64-getallen.
- De binaire opslag verklaart waarom sommige berekeningen een kleine afrondingsafwijking tonen.

Python-booleans en logische operatoren

1 Waarheidswaarden

Python gebruikt twee booleaanse waarden. Ze zijn hoofdlettergevoelig en hebben het type bool.

- True — waar
- False — onwaar
- type(True) is bool



2 Vergelijken

Een vergelijking geeft True of False terug. Gebruik == om waarden te vergelijken; = kent een waarde toe.

- $3 < 5 \rightarrow \text{True}$
- $4 == 4 \rightarrow \text{True}$
- $4 != 5 \rightarrow \text{True}$
- Ook: <, <=, > en >=



3 not

not keert de waarheidswaarde om en levert altijd True of False op.

- not True $\rightarrow$ False
- not False $\rightarrow$ True
- not ($3 < 5$) $\rightarrow$ False



4 and

and is waar als beide operanden waar zijn. Python beoordeelt van links naar rechts en stopt zodra de uitkomst vaststaat.

- True and True $\rightarrow$ True
- True and False $\rightarrow$ False
- False and (1 / 0) $\rightarrow$ False; de rechterkant wordt niet uitgevoerd



5 or

or is waar als ten minste één operand waar is. Ook hier stopt Python zodra de uitkomst vaststaat.

- False or True $\rightarrow$ True
- False or False $\rightarrow$ False
- True or (1 / 0) $\rightarrow$ True; de rechterkant wordt niet uitgevoerd



8 Veelgemaakte verwarring

and en or leveren niet altijd een bool op: ze geven een van hun operanden terug, not levert wel True of False op.

- = wijst toe; == vergelijkt
- 0 or "standaard" $\rightarrow$ "standaard", omdat 0 falsy is
- Controleer of je een waarheidswaarde of juist de operandwaarde nodig hebt



7 Kortsluiting en volgorde

and en or evalueren van links naar rechts. De rechterkant wordt overgeslagen als de linker operand de uitkomst al bepaalt.

- Falsy links bij and $\rightarrow$ stop; die operand wordt teruggegeven
- Truthy links bij or $\rightarrow$ stop; die operand wordt teruggegeven
- not heeft hogere prioriteit dan and; and heeft hogere prioriteit dan or
- Gebruik haakjes om groepering duidelijk te maken



Let op

Kortsluiting kan voorkomen dat een foutgevoelige rechterkant wordt uitgevoerd. Vertrouw daar alleen op als de volgorde van de expressie duidelijk is.

6 Truthy en falsy

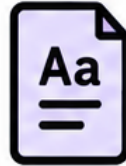
In een booleaanse context wordt een waarde als waar of onwaar behandeld. Onderstaande waarden zijn falsy; de meeste andere waarden zijn truthy.

- Falsy: False, None, 0 en lege collecties zoals [] en ""
- Truthy: bijvoorbeeld 1 en "hallo"
- Gebruik een expliciete vergelijking als je bedoelt te testen op bijvoorbeeld nul: aantal == 0



1. Tekenreeksen maken

Een tekenreeks is tekst tussen aanhalingstekens. Enkele en dubbele aanhalingstekens kunnen allebei.



- `tekst = 'Hallo'`
- `naam = "Sam"`
- `len(tekst)` geeft het aantal tekens.

2. Onveranderlijkheid

Tekenreeksen zijn onveranderlijk: een bewerking verandert de bestaande tekst niet. Het resultaat is een nieuwe tekenreeks.



- `tekst = 'hoi'`
- `nieuw = tekst.upper()`
- `tekst` blijft 'hoi'; `nieuw` is 'HOI'.

Onthoud: Sla het resultaat op in een variabele als je het later wilt gebruiken.

3. Indexering

Met een index haal je één teken op. De eerste positie is 0; een negatieve index telt vanaf het einde.

- `woord = 'Python'`
- `woord[0]` geeft 'P'.
- `woord[-1]` geeft 'n'.

P	y	t	h	o	n
0	1	2	3	4	5

8. F-strings: waarden in tekst

Een f-string voegt waarden in tekst in. Zet een **f** vóór de aanhalingstekens en plaats de variabelenaam tussen accolades.



- `naam = 'Sam'`
- `bericht = f'Hallo, {naam}!'`
- `bericht` is 'Hallo, Sam!'; `naam` blijft 'Sam'.
- Dit voegt een waarde in de tekst in; het verandert naam niet.

Kleine oefening: Maak een variabele stad en schrijf een f-string die er een begroeting mee maakt.

Tekenreeksen in Python

4. Slicing

Slicing haalt een deel van een tekenreeks op met `tekst[start:stop:step]`. De start telt mee; de stop niet.

- `'Python'[1:4]` geeft 'yth'.
- `'Python'[:2]` geeft 'Py'.
- `'Python'[-3:]` geeft 'hon'.

P	y	t	h	o	n

Tip: Laat start weg om bij het begin te starten; laat stop weg om door te gaan tot het einde.

7. Veelgebruikte methoden

Methoden leveren een resultaat terug en veranderen de oorspronkelijke tekenreeks niet.



- `' Hoi '.strip()` geeft 'Hoi'.
- `'hoi'.replace('h', 'd')` geeft 'doi'.
- `'A,B'.split(',')` geeft ['A', 'B']; `','.join(['A', 'B'])` geeft 'A,B'.
- `'Hoi'.lower()` geeft 'hoi'; `'hoi'.upper()` geeft 'HOI'.
- Gebruik bijvoorbeeld `schoon = invoer.strip()` om het resultaat te bewaren.

Werkwijze: Bewaar een gewenst resultaat in een variabele, zoals `schoon = invoer.strip()`.

6. Samenvoegen en herhalen

Met `+` voeg je tekenreeksen samen; met `*` herhaal je ze. Beide bewerkingen leveren een nieuwe tekenreeks op.

- `'zon' + 'dag'` geeft 'zondag'.
- `'ha' * 3` geeft 'hahaha'.
- `len('Python')` geeft 6.



5. Escape-tekens en raw strings

In een gewone tekenreeks introduceert een backslash sommige speciale tekens. Een raw string gebruikt een r-voorvoegsel en behandelt de meeste backslashes letterlijk.

- `'A\nB'` bevat een nieuwe regel tussen A en B.
- `'A\tB'` bevat een tab tussen A en B.
- `r'C:\map\bestand'` behandelt de backslashes letterlijk.



1. Sleutel en waarde

Een dictionary (dict) koppelt elke sleutel aan een waarde. Gebruik een dictionary als je gegevens met een herkenbare sleutel wilt terugvinden.

- Voorbeeld: naam → Noor; score → 8).
- Elke sleutel verwijst naar de bijbehorende waarde.

2. Een dictionary maken

Zet sleutel-waardeparen tussen accolades. Schrijf na elke sleutel een dubbele punt en scheid paren met een komma.

- `leerling = {"naam": "Noor", "score": 8}`
- Tekst staat tussen aanhalingstekens; het getal 8 niet.

3. Sleutels kiezen

Een sleutel moet hashbaar zijn. Veelgebruikte sleutels zijn strings en getallen. Een tuple kan ook, als al zijn onderdelen hashbaar zijn.

- Geschikt: "naam", 42, ("klas", 2).
- Niet geschikt: een lijst of dictionary als sleutel; die zijn veranderlijk.
- Een waarde mag wél veranderlijk zijn, bijvoorbeeld een lijst.

4. Waarden veilig opvragen

Gebruik vierkante haken voor een sleutel waarvan je weet dat die bestaat. Gebruik `get` als de sleutel kan ontbreken.

- `leerling["naam"]` geeft "Noor".
- `leerling.get("groep")` geeft `None` als "groep" ontbreekt.
- `leerling["groep"]` geeft een `KeyError` als die sleutel ontbreekt.

Kies bewust — Verwacht je dat de sleutel ontbreekt? Gebruik dan `get` of controleer eerst met `in`.

Python-dictionaries: sleutel-waardeparen

5. Toevoegen en wijzigen

Wijs een waarde toe aan een sleutel met vierkante haken. Een nieuwe sleutel wordt toegevoegd; bij een bestaande sleutel wordt de waarde vervangen.

- `leerling["groep"] = "B"` voegt een sleutel toe.
- `leerling["score"] = 9` vervangt de bestaande score.

6. Verwijderen

Verwijder een bestaande sleutel met `del`. Gebruik `pop` met een standaardwaarde als de sleutel eventueel ontbreekt.

- `del leerling["score"]` verwijdert de sleutel en waarde.
- `leerling.pop("groep", None)` verwijdert "groep" als die bestaat; anders geeft het `None` terug.

Let op — `del d[sleutel]` geeft een fout als de sleutel ontbreekt. Kies `pop` met een standaardwaarde als dat mogelijk is.

7. Aanwezigheid en itereren

`in` controleert of een sleutel aanwezig is, niet of een waarde voorkomt. Gebruik de views om over sleutels, waarden of paren te lopen.

- "naam" in `leerling` controleert de sleutels.
- `leerling.keys()` levert sleutels; `leerling.values()` levert waarden.
- `leerling.items()` levert sleutel-waardeparen.

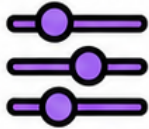
8. Werking en aandachtspunten

Dictionaries zijn veranderlijk en worden doorgaans met hashtableen geïmplementeerd. Toegang, invoegen en verwijderen zijn gemiddeld snel; constante prestaties zijn niet gegarandeerd.

- Python bewaart de invoegvolgorde van dictionary-sleutels.
- Denk bij gebruik aan ontbrekende sleutels en aan het verschil tussen sleutels en waarden.

Omvang van deze editie — 50 mindmaps is een voorlopige productieschatting, geen bevestigde verkoopbelofte. Valideer het aantal vóór finale productie aan de daadwerkelijke checkoutbelofte.

1. Wat betekent variabele?



Variabele argumenten laten een functie een wisselend aantal waarden ontvangen. Het sterretje bepaalt hoe Python ze verzamelt of uitpakt.

- Functiedefinitie: verzamelen.
- Functieaanroep: uitpakken.
- args en kwargs zijn gebruikelijke namen; de sterretjes geven het gedrag aan.

2. *args in een definitie



Een enkele ster verzamelt extra positionele argumenten in een tuple: een geordende, niet-wijzbare reeks waarden.

```
def tel_op(*getallen):  
    return sum(getallen)
```

tel_op(2, 3, 5) geeft 10.

3. **kwargs in een definitie



Een dubbele ster verzamelt extra benoemde argumenten in een dictionary: een verzameling sleutel-waardeparen.

```
def toon(**opties):  
    return opties
```

toon(kleur="blauw", maat="M") geeft
{'kleur': 'blauw', 'maat': 'M'}.

8. Veelvoorkomende fouten



Controleer wat je verzamelt of uitpakt en of de namen overeenkomen met de parameters.

- *args in een definitie verzamelt; *waarden in een aanroep pakt uit.
- **kwargs verzamelt benoemde argumenten; **opties pakt een mapping uit.
- Een onbekend argument of dubbele waarde kan een TypeError veroorzaken.
- Mini-voorbeeld:

```
def som(*getallen):  
    return sum(getallen)
```

som(4, 6) geeft 10.

Onthoud: Eén ster voor positionele waarden; twee sterren voor benoemde waarden. De namen args en kwargs zijn conventies, geen verplichte namen.

Variabele argumenten in Python: *args en **kwargs

4. Uitpakken met * bij een aanroep



Bij een functieaanroep pakt * de waarden uit een iterable, zoals een lijst of tuple, uit als afzonderlijke positionele argumenten.

```
def beschrijf(naam, leeftijd):  
    return f"{naam}: {leeftijd}"
```

waarden = ("Noor", 20)
beschrijf(*waarden) geeft "Noor: 20".

7. Combineren en volgorde



Je kunt expliciete argumenten combineren met uitgepakte waarden. Python koppelt ze aan de parameters volgens de functiedefinitie.

```
def profiel(naam, stad="Utrecht"):  
    return f"{naam} woont in {stad}"
```

gegevens = {"stad": "Leiden"}
profiel("Ava", **gegevens) geeft "Ava woont in Leiden".

Let op: Geef dezelfde parameter niet twee keer een waarde: profiel("Ava", naam="Bo") veroorzaakt een TypeError.

6. Argumenten doorgeven

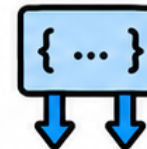


Een functie kan verzamelde argumenten doorsturen naar een andere functie. Dit heet forwarden.

```
def doorsturen(*args, **kwargs):  
    return doel(*args, **kwargs)
```

- De positionele waarden blijven positioneel; benoemde waarden blijven benoemd.

5. Uitpakken met ** bij een aanroep



Bij een functieaanroep pakt ** een mapping met string-sleutels uit als benoemde argumenten.

```
def begroet(naam, bericht):  
    return f"{bericht}, {naam}!"
```

opties = {"naam": "Sam", "bericht": "Hoi"}
begroet(**opties) geeft "Hoi, Sam!".

1 Modulebegrip



Een module is meestal een Python-bestand (.py) met code die je vanuit andere code kunt gebruiken.

- Voorbeeld: gereedschap.py bevat hulpfuncties.
- Modules groeperen code rond een taak.

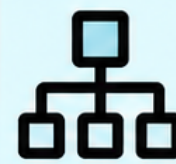
2 Importeren



Met import maak je namen uit een module beschikbaar.

- `import gereedschap` → gebruik `gereedschap.tel_op()`.
- `from gereedschap import tel_op` → gebruik `tel_op()`.
- `import gereedschap as g` → gebruik `g.tel_op()`.

3 Namen en naamruimten



Een naamruimte koppelt namen aan objecten. Een modulenaam maakt duidelijk waar een naam vandaan komt.

- `gereedschap.tel_op` verwijst naar `tel_op` in die module.
- Twee modules kunnen dezelfde functienaam hebben; de modulenaam maakt onderscheid.

8 Klein structuurvoorbeeld



In deze eenvoudige structuur bevat tekst.py de module; app.py importeert die vanuit de projectmap.

```
project/  
├── app.py  
├── hulpmiddelen/  
│   ├── __init__.py  
│   └── tekst.py
```

In app.py:

```
from hulpmiddelen import tekst
```

Modules en pakketten: structuur en imports

4 Pakketten



Een pakket organiseert verwante modules in een map.

- Voorbeeld: hulpmiddelen/tekst.py.
- Importeer bijvoorbeeld met `from hulpmiddelen import tekst`.
- Het projectpad moet zo zijn ingesteld dat Python hulpmiddelen kan vinden.

7 Veelvoorkomende importproblemen



Als Python een module niet vindt, controleer dan de naam en projectcontext.

- Controleer spelling en hoofdletters van de importnaam.
- Controleer of de module op een vindbaar pad staat, waaronder de paden in `sys.path`.
- Geef lokale bestanden niet dezelfde naam als veelgebruikte standaardbibliotheekmodules.

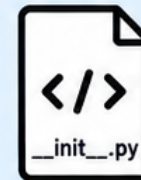
6 Projectmappen



Een overzichtelijke mapstructuur maakt imports en onderhoud eenvoudiger.

- Groepeer verwante code in duidelijke mappen.
- Kies unieke, beschrijvende bestandsnamen.
- Voer je programma uit vanuit een passende projectcontext.

5 __init__.py



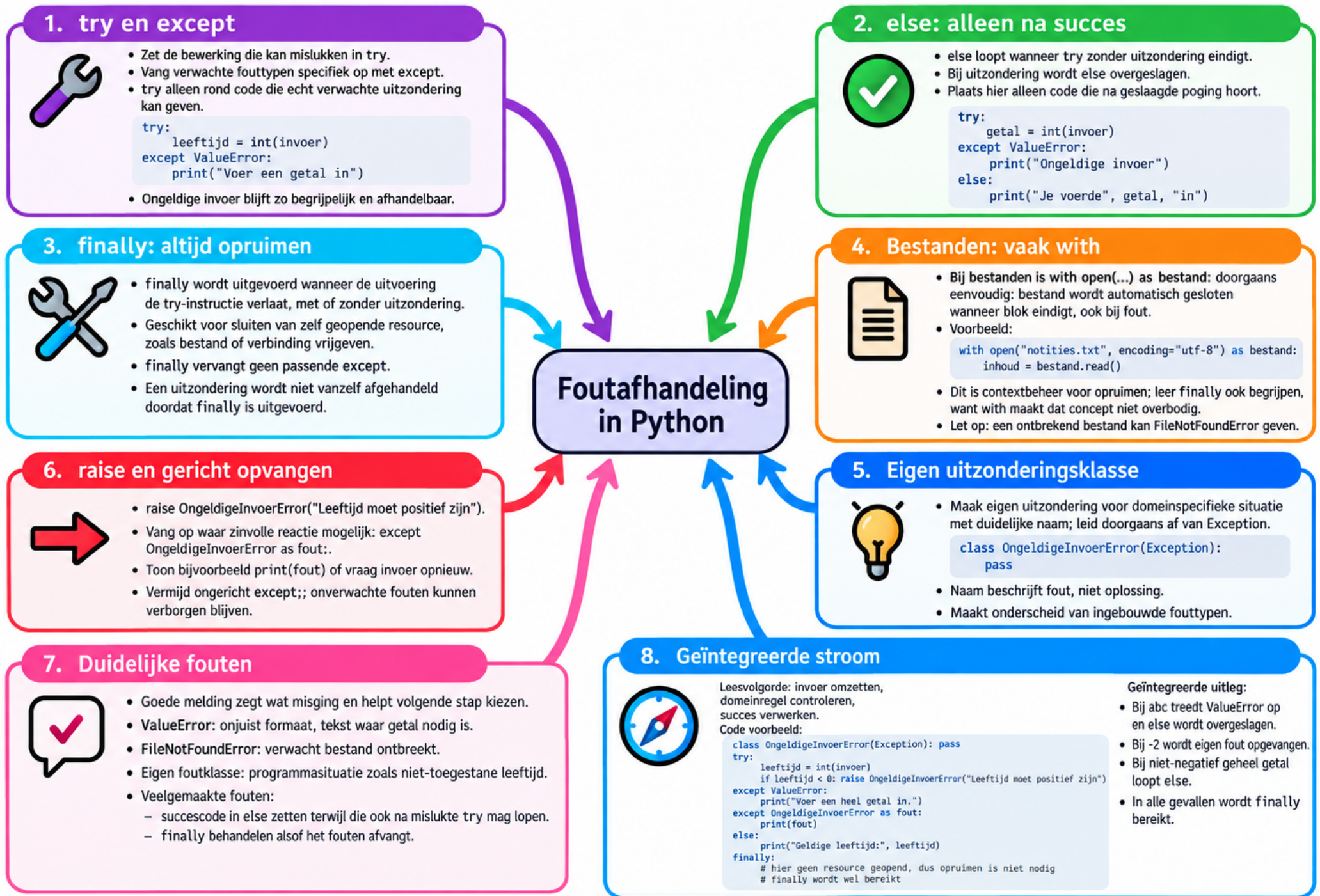
Een regulier pakket kan een bestand `__init__.py` bevatten.

- Het bestand kan leeg zijn of initialisatiecode bevatten.
- Het kan geselecteerde namen uit het pakket beschikbaar maken.
- Niet elk pakket vereist dit bestand: Python ondersteunt ook naamruimtepakketten zonder `__init__.py`.



Onthoud

Behandel `__init__.py` niet als verplicht in iedere pakketmap.



1 Wat is een contextmanager?



Een contextmanager beheert het begin en einde van tijdelijk werk. Bij het betreden van de context worden benodigde acties uitgevoerd en bij het verlaten wordt netjes afgesloten. Dit gebeurt via de protocollen `__enter__()` en `__exit__()`.

- Doel: resources gecontroleerd gebruiken en vrijgeven.
- Voorbeelden: bestanden, vergrendelingen en netwerkverbindingen.
- Resource wordt vrijgegeven, ook bij een uitzondering.

2 with-syntaxis en blokgrenzen



De dubbele punt `:` start het blok en de inspringing (indentatie) markeert de context. Na het blok worden de schoonmaakacties automatisch uitgevoerd.

Patroon:

```
with open("notities.txt", "r", encoding="utf-8") as bestand:
    inhoud = bestand.read()
# Hier is het bestand gesloten.
```

- Gebruik consistente inspringing.
- Plaats alleen werk dat van de resource afhankelijk is in het blok.
- Gebruik de resource (bijv. het bestand) niet meer na het blok.

8 Deterministisch bronbeheer en valkuilen



Een contextmanager zorgt voor expliciet en voorspelbaar openen en sluiten van resources, in plaats van te vertrouwen op garbage collection.

- Gebruik `with` voor duidelijke levensduren van resources.
- Controleer of 'w' kan overschrijven.
- Behandel fouten, codering en padvalidatie afzonderlijk.
- Sluiten maakt paden niet veilig.

Productcontext:

50 mindmaps is uitsluitend een voorlopige productieschatting voor dit afzonderlijke bumpproduct, geen bevestigde verkoopbelofte. Toets dit vóór finale productie aan de daadwerkelijke checkoutbelofte. Deze eenheid is één mindmap.

7 Zelf een contextmanager maken



Met `contextlib.contextmanager` maak je eenvoudig een contextmanager. Je voert voorbereiding op en de opruimactie erna.

- Importeer contextmanager uit `contextlib`.
- Gebruik de decorator `@contextmanager`.
- Voer voorbereiding uit en `yield` de contextwaarde.
- Doe de cleanup in een `finally`-blok.

Klein voorbeeld:

```
from contextlib import contextmanager

@contextmanager
def tijdelijke_stap():
    print("Start")
    try:
        yield
    finally:
        print("Opruimen")

with tijdelijke_stap():
    print("Werk")
```

6 Codering, bestanden en paden



Bestanden kunnen tekst (`str`) of bytes (`bytes`) bevatten. Geef bij tekst altijd expliciet een encoding op. Met `with` beheer je de levensduur van het bestand, niet de veiligheid van het pad.

- Gebruik `encoding="utf-8"`.
- Voor grote bestanden: itereren met `for regel in bestand:`.
- Gebruik `'rb'/'wb'` voor bytes, zonder encoding.
- Controleer paden en invoer apart.
- Rechten en bescherming tegen path traversal blijven belangrijk.

Let op: `with` alleen voorkomt geen path traversal, race conditions of te ruime rechten.

3 open(), alias en sluiten



De functie `open()` geeft een bestandsobject terug en met `as bestand` geef je dat een naam. Bij het verlaten van het `with`-blok wordt het bestand automatisch gesloten.

- Geef het pad, de modus en expliciet de tekenncoding op.

```
with open(pad, modus, encoding="utf-8") as bestand:
```

- Bij binaire bestanden laat je encoding weg.

4 Lezen, schrijven en toevoegen



- `'r'` leest een bestaand bestand.
- `'w'` overschrijft of maakt een nieuw bestand.
- `'a'` voegt toe aan een bestaand bestand of maakt het aan.
- `'x'` maakt exclusief een nieuw bestand en faalt als het al bestaat.
- `'b'` geeft aan dat je met bytes werkt, bijvoorbeeld `'rb'`.

Veilige keuze:

Gebruik geen `'w'` als overschrijven niet de bedoeling is. Gebruik `'x'` als een nieuw bestand vereist is. Handel een bestaand bestand af volgens je toepassing.

5 Uitzonderingen en opruimen



De opruimactie van een contextmanager wordt altijd uitgevoerd, ook als er een fout optreedt. De uitzondering wordt meestal doorgegeven aan de aanroepende code.

- Automatisch sluiten voorkomt dat resources onbedoeld open blijven.
- Een contextmanager lost de programmfout zelf niet op.
- Vang een uitzondering alleen af wanneer je er zinvol op kunt reageren.



Python-contextmanagers:
with en bronbeheer

1. Wat is een contextmanager?

Een contextmanager beheert het begin en einde van een tijdelijke werksituatie, zoals het gebruik van een bestand. Het with-statement betreedt de context en sluit die netjes af. Een contextmanager-object biedt daarvoor de protocollen `__enter__()` en `__exit__()`.



- Doel: resources gecontroleerd gebruiken en vrijgeven.
- Voorbeelden: bestanden, vergrendelingen en netwerkverbindingen.
- De resource wordt na het with-blok vrijgegeven, ook als daarin een uitzondering optreedt.

2. Begin en einde

Een contextmanager beheert het begin en einde van een tijdelijke werksituatie.



3. Tijdelijke werksituatie

Een contextmanager beheert een tijdelijke werksituatie, zoals het gebruik van een bestand.



4. with-statement

Het with-statement betreedt de context en sluit die netjes af.



5. __enter__()

Een contextmanager-object biedt het protocol `__enter__()`.



6. __exit__()

Een contextmanager-object biedt het protocol `__exit__()`.



8. Ook bij een uitzondering

De resource wordt na het with-blok vrijgegeven, ook als daarin een uitzondering optreedt.



7. Gecontroleerd vrijgeven

Doel: resources gecontroleerd gebruiken en vrijgeven.



Python-contextmanagers with en bronbeheer

1 Module re en raw strings

- `import re`
- `r'\d+'` = één of meer cijfers
- De `r` maakt backslashes leesbaar.



'bestelnummer 482' → '482'

2 Bouwstenen

- `abc` — letterlijk
- `[A-Z]` — één hoofdletter
- `+` één of meer; `*` nul of meer
- `^` begin; `$` einde
- `(...)` groep



`r'[A-Z]+'` past op 'NL'
Vorm is niet automatisch betekenis.

3 Zoeken: search

- Zoekt vanaf elke positie
- Eerste match
- `re.search(r'\d+', 'kamer 204').group()`
→ '204'
- Geen match: `None`; controleer vóór `.group()`.



8 Testen en grenzen

- Test matches én niet-matches
- `r'\d+'` op '482' → '482';
op 'geen nummer' → geen match
- `fullmatch` met `r'\d+'` op '482x'
→ geen match
- Controleer betekenis en toepassingsregels apart.



Reguliere expressies: de module re en patroonherkenning

4 Begin: match

- Controleert vanaf het begin
- `re.match(r'Kat', 'Kat en hond')`
→ 'Kat'
- `re.match(r'hond', 'Kat en hond')`
→ geen match
- De rest hoeft niet te passen.



search: overal · match: begin · fullmatch: hele tekst

7 Vervangen: sub

- `re.sub(r'\d+', '[nummer]', 'kamer 204')`
- Resultaat: 'kamer [nummer]'
- Controleer of alleen de bedoelde tekst is vervangen.



6 Vinden en groepen

- `findall` geeft gevonden tekst
- `re.findall(r'\d+', 'A12 B34')`
→ ['12', '34']
- `finditer` geeft matchobjecten;
`.group()` leest de match
- `r'([A-Z])(\d+)'` op 'A12':
groep 1 'A', groep 2 '12'



5 Volledig: fullmatch

- De hele tekst moet passen
- `r'\d+'` op '482' → match
- `r'\d+'` op '482x' → geen match
- Test passende en niet-passende invoer.



Tekstvorm bewijst niet dat een waarde geldig of toegestaan is; controleer betekenis en regels apart.

Willekeur in Python: random en secrets

1 Doel en begrippen



- **Toeval:** een uitkomst die je niet vooraf kiest.
- **Pseudo-willekeur:** een algoritme maakt een reeks die willekeurig lijkt.
- Gebruik pseudo-willekeur voor simulaties en spelletjes; niet voor geheimen.

Onthoud: Een willekeurig ogende uitkomst is niet automatisch geschikt om geheim te houden.

2 De module random



```
import random
```

- Geschikt voor dobbelsteen, spel of niet-beveiligingsgevoelige simulatie.
- Niet bedoeld voor beveiligingsgevoelige geheimen.

3 Veelgebruikte functies



- `random.random()` geeft een kommagetal vanaf 0 tot, maar niet inclusief, 1.
- `random.randint(1, 6)` kiest een geheel getal van 1 tot en met 6.
- `random.choice(keuzes)` kiest één element uit een reeks.
- `random.shuffle(kaarten)` schudt een lijst op zijn plaats.

Let op: shuffle verandert de bestaande lijst; het maakt niet automatisch een nieuwe lijst.

4 Herhaalbare resultaten met seed



- Een seed maakt een pseudo-willekeurige reeks reproduceerbaar, handig bij testen en demonstraties.
- Stel vóór de test een vaste seed in, bijvoorbeeld `random.seed(7)`.
- Voer dezelfde stappen opnieuw uit om de reeks te reproduceren.
- Gebruik een vaste seed niet wanneer onvoorspelbaarheid nodig is.

Toepassing: Als een test onverwacht faalt, helpt een reproduceerbare reeks om dezelfde situatie opnieuw te onderzoeken.

5 De module secrets



- Gebruik secrets wanneer een waarde onvoorspelbaar moet zijn, bijvoorbeeld voor een token.

```
import secrets
```
- `secrets.choice(tekens)` kiest een teken uit een opgegeven reeks.
- Bedoeld voor beveiligingsgevoelige toepassingen; geen garantie voor de veiligheid van een complete toepassing.

6 Tokens maken



- **Voorbeeld:** `import secrets; token = secrets.token_hex(16)`.
- De parameter 16 vraagt om 16 willekeurige bytes; elke byte wordt weergegeven met twee hexadecimale tekens, dus 32 tekens totaal.
- Behandel het resultaat als geheim: zet het niet in openbare code of berichten.

Voorbeeld: `token_hex(16)` levert 32 willekeurige hexadecimale tekens op.

7 Keuzehulp



Waarvoor heb je de willekeurige waarde nodig?

- Een spel, simulatie of test waarbij herhalen handig is → Gebruik random; stel eventueel een seed in voor reproduceerbare tests.
- Een geheim token of andere beveiligingsgevoelige waarde → Gebruik secrets en voorkom dat je het geheim openbaar maakt.

8 Valkuilen en mini-oefening



- **Valkuil:** random gebruiken voor een token. Kies daarvoor secrets.
- **Valkuil:** denken dat een seed extra geheimhouding biedt. Een seed maakt herhaling mogelijk.
- 1. Dobbelsteen: `worp = random.randint(1, 6)`
- 2. Tekenkeuze: `teken = secrets.choice('ABC123')`
- Leg bij elk voorbeeld uit waarom je die module kiest.

Zelf controleren: Dobbelsteenworp: gewone pseudo-willekeur. Geheim token: onvoorspelbaarheid is belangrijk, dus kies secrets.



Python
voor beginners

Decorators: functiemodificatie en wrappers



Nederlandse editie · Python voor beginners



Voorlopige productieschatting: 50 mindmaps
voor de collectie; geen bevestigde verkoopbelofte.
Controleer het aantal vóór de definitieve productie
met de checkoutbelofte.

Decorators: functiemodificatie en wrappers



1 Functie als object



Een Python-functie is zelf een object. Je kunt haar bewaren in een variabele en doorgeven aan andere code.

- Een functienaam verwijst naar de functie; haakjes roepen haar aan.
- Zonder haakjes kun je de functie doorgeven: bijvoorbeeld `taak` in plaats van `taak()`.
- Een decorator gebruikt die mogelijkheid om een functie te ontvangen.

2 Hogere-ordefunctie



Een hogere-ordefunctie neemt een functie als argument, retourneert een functie, of doet beide.

- Een decorator is doorgaans een hogere-ordefunctie.
- Hij ontvangt de oorspronkelijke functie en levert vaak een aangepaste functie terug.
- Die aangepaste functie is een callable: iets dat je kunt aanroepen.

3 Het wrapperidee



Een **wrapper** is een functie die de oorspronkelijke functie aanroept en er gedrag omheen plaatst.

- Bijvoorbeeld: toon een korte melding vóór en na de aanroep.
- De wrapper bepaalt wat er vóór of na de oorspronkelijke functie gebeurt.
- Een wrapper vervangt de functie als aanroepbaar object; de oorspronkelijke functie kan binnenin worden aangeroepen.

8 Klein voorbeeld en check



In dit **veilige voorbeeld** toont de wrapper meldingen vóór en na de functieaanroep. De functie zelf geeft een resultaat terug.

- Voorbeeld:

```
def omhul(func):  
    def wrapper(*args, **kwargs):  
        print("Start");  
        resultaat = func(*args, **kwargs);  
        print("Klaar");  
        return resultaat;  
    return wrapper;  
  
Daarna: @omhul boven def begroet(naam):  
    return f"Hallo, {naam}!".  
Bij begroet("Sam") verschijnt eerst "Start", daarna "Klaar"; de aanroep levert "Hallo, Sam!" terug.  
Let op: de meldingen verschijnen bij de aanroep; het decoreren gebeurt bij het uitvoeren van de definitie.  
Verwarring voorkomen: de decorator ontvangt de functie; de wrapper ontvangt de argumenten van de latere aanroep.
```

Onthoud

@decorator is kort voor `functienaam = decorator(functienaam)`. De wrapper roept de oorspronkelijke functie aan en geeft met `return` het resultaat door.

Decorators: functiemodificatie en wrappers

4 De @decorator-syntax



Een regel met @decorator boven een functie-definitie is een kortere schrijfwijze voor het opnieuw binden van de functienaam aan het resultaat van `decorator(func)`.

- @decorator boven `def taak(): ...` komt overeen met `taak = decorator(taak)`.
- De decorator ontvangt de functie zodra de definitie wordt uitgevoerd.
- Na het decoreren verwijst de functienaam naar het resultaat van de decorator.

7 Naam en documentatie behouden



Een wrapper kan standaard de naam en documentatie van de oorspronkelijke functie verbergen. `functools.wraps` helpt die metadata te behouden.

- Importeer `wraps` uit `functools`.
- Gebruik `@wraps(func)` boven de wrapperdefinitie.
- De wrapper blijft een wrapper; `wraps` helpt herkenbare functiegegevens over te nemen.

6 Argumenten en resultaat doorgeven



Een wrapper die verschillende aanroepen moet ondersteunen, kan positionele argumenten met `*args` en benoemde argumenten met `**kwargs` doorgeven.

- Gebruik `func(*args, **kwargs)` om de oorspronkelijke functie met de ontvangen argumenten aan te roepen.
- Gebruik `return func(*args, **kwargs)` om ook het resultaat terug te geven.
- Zonder `return` geeft de wrapper het resultaat niet automatisch door.

5 Wanneer gebeurt wat?



De decorator wordt toegepast wanneer Python de functie-definitie uitvoert, niet bij elke latere aanroep van de versierde functie.

- Definitie: Python maakt de functie en past de decorator toe.
- Aanroep: de naam verwijst doorgaans naar de wrapper; die voert de gekozen extra stappen uit.
- De oorspronkelijke functie draait wanneer de wrapper haar aanroept.





1 Lokale opslag en verbinding



SQLite is een relationele database die in de applicatie is ingebed. Python's module `sqlite3` biedt toegang tot een database, bijvoorbeeld in een lokaal bestand.

- Importeer: `import sqlite3`
- Open een verbinding:
`verbinding = sqlite3.connect("taken.db")`
- De verbinding beheert de toegang tot de database en transacties.

2 Tabellen, rijen en kolommen



Relationele gegevens staan in tabellen. Elke rij bevat één record; kolommen beschrijven de bijbehorende waarden.

- Voorbeeldtabel: `taken`
- Kolommen: `id`, `titel`, `klaar`
- Een rij kan bijvoorbeeld één taak en de status ervan voorstellen.

3 Datamodel en sleutels



Bepaal hoe gegevens bij elkaar horen voordat je queries schrijft. Sleutels helpen records te identificeren en relaties tussen tabellen weer te geven.

- Een primaire sleutel identificeert een rij, bijvoorbeeld `id`.
- Een vreemde sleutel kan verwijzen naar een rij in een andere tabel.
- Voorbeeld: `taken` kunnen met een `project_id` aan een project worden gekoppeld.



Eerst modelleren

Bepaal welke gegevens je bewaart, welke rijen uniek zijn en welke gegevens naar een andere tabel verwijzen.

8 Van vraag naar databasebewerking



Gebruik een eenvoudige route om een opslagtaak om te zetten in een passende query.

- Voorbeeldvraag: welke taken zijn klaar?
- Kies de tabel en kolommen die nodig zijn.
- Bepaal of je gegevens leest of wijzigt.
- Geef eventuele invoerwaarden als parameters door.

1 Bepaal de gegevens

Noteer wat elke taak nodig heeft, bijvoorbeeld een `id`, `titel` en `status`.

2 Kies tabellen en sleutels

Groep samenhangende gegevens in tabellen en wijs een primaire sleutel aan.

3 Formuleer de bewerking

Kies `SELECT` om gegevens te lezen of een passende wijzigingsopdracht om gegevens te veranderen.

4 Voer veilig uit

Gebruik een cursor en geef invoerwaarden afzonderlijk mee als parameters.

5 Rond wijzigingen af

Bevestig geslaagde wijzigingen met `commit()` of draai ze terug met `rollback()`.



Gegevenspersistentie met sqlite3

7 Python-bouwstenen



Verbind de basisstappen: importeer de module, wijs resultaten toe aan variabelen, roep methoden aan en gebruik de teruggegeven rijen.

- `sqlite3.connect(...)` levert een verbinding op.
- `verbinding.cursor()` levert een cursor op.
- `cursor.execute(...)` voert een opdracht uit; lees resultaten op wanneer de query rijen oplevert.
- Sluit de verbinding wanneer je klaar bent: `verbinding.close()`

1 Importeer en verbind

Schrijf `import sqlite3` en wijs `sqlite3.connect("taken.db")` toe aan een variabele.

2 Maak een cursor

Roep `.cursor()` aan op de verbinding en bewaar de cursor in een variabele.

3 Voer uit en lees

Geef de SQL-opdracht door aan `cursor.execute()`; haal resultaatrijen op als je query gegevens teruggeeft.

4 Rond af

Bevestig of draai wijzigingen terug en sluit daarna de verbinding.

6 Transacties en wijzigingen



Een transactie groepeert databasewijzigingen. Je kunt geslaagde wijzigingen bevestigen of niet-gewenste wijzigingen terugdraaien.

- Voer de wijzigingsopdrachten via de cursor uit.
- Bevestig wijzigingen met `verbinding.commit()`.
- Draai wijzigingen terug met `verbinding.rollback()`.

1 Wijziging bewaren

Voer de wijzigingsopdracht uit, controleer het resultaat en roep `verbinding.commit()` aan als je de wijziging wilt bewaren.

2 Wijziging terugdraaien

Als de wijziging niet moet worden bewaard, roep `verbinding.rollback()` aan.

4 Cursor en queryresultaten



Een cursor voert SQL-opdrachten uit en kan de opgehaalde rijen beschikbaar maken in Python.

- Maak een cursor: `cursor = verbinding.cursor()`
- Voer een query uit:
`cursor.execute("SELECT id, titel FROM taken")`
- Haal rijen op: `rijen = cursor.fetchall()`
- Een resultaatrij bevat de waarden van de geselecteerde kolommen.

5 Waarden veilig parametriseren



Gebruik placeholders voor waarden en geef die afzonderlijk mee aan `execute()`. Plak onbetrouwbare invoer niet aan SQL-tekst vast.

- Voorbeeld:
`cursor.execute("SELECT * FROM taken WHERE id = ?", (taak_id,))`
- De `?` is de placeholder; de tuple bevat de waarde.
- Parametrisering helpt SQL-injectie te voorkomen.



Niet aaneenplakken

Maak geen SQL-string door gebruikersinvoer met `+` toe te voegen. Gebruik placeholders en geef waarden apart mee.

1

Lokale opslag en verbinding



- SQLite is een relationele database die in de applicatie is ingebed.
- Python's module `sqlite3` biedt toegang tot een database, bijvoorbeeld in een lokaal bestand.
- Importeer: `import sqlite3`
- Open een verbinding:
`verbinding = sqlite3.connect("taken.db")`
- De verbinding beheert de toegang tot de database en transacties.

2

Tabellen en structuur



- Maak een tabel met SQL.
- Gebruik het datatype dat past bij je gegevens.
- Voorbeeld:

```
CREATE TABLE taken (
  id INTEGER PRIMARY KEY,
  titel TEXT,
  voltooid INTEGER
);
```

3

Primaire sleutels en relaties



- Gebruik een primaire sleutel om rijen uniek te maken.
- `INTEGER PRIMARY KEY` genereert automatisch een id.
- Relaties leg je met vreemde sleutels (`FOREIGN KEY`).
- Dit voorkomt dubbele gegevens en houdt tabellen gekoppeld.

8

Praktisch voorbeeld



- Een eenvoudige takenlijst toepassen in Python.
- Stappen: tabel maken, gegevens toevoegen, opvragen en aanpassen.
- Gebruik bovenstaande onderdelen samen in een klein script.
- Zo begrijp je hoe gegevenspersistentie met `sqlite3` in de praktijk werkt.

Gegevenspersistentie met sqlite3



4

Gegevens uitvoeren met een cursor



- Gebruik een cursor om SQL uit te voeren.
- Voorbeelden:

```
cursor = verbinding.cursor()
cursor.execute("SELECT * FROM taken")
resultaten = cursor.fetchall()
```

7

Fouten afhandelen



- Gebruik `try-except` om fouten netjes op te vangen.
- Voorbeeld:

```
try:
    cursor.execute(sql, waarden)
    verbinding.commit()
except sqlite3.Error as e:
    print("Fout:", e)
    verbinding.rollback()
```

6

Transacties en bevestigen



- Wijzigingen worden pas opgeslagen na een bevestiging (`commit`).
- Voorbeeld:
`verbinding.commit()`
- Bij een fout kun je terugdraaien met:
`verbinding.rollback()`
- Sluit de verbinding wanneer je klaar bent.

5

Parametrisering en veiligheid



- Gebruik parameters om SQL-injectie te voorkomen.
- Voorbeeld:

```
cursor.execute(
    "INSERT INTO taken (titel) VALUES (?)",
    ("Leren Python",)
)
```
- Vul waarden via `?` en geef ze apart mee.

1. Socket en eindpunt



Een socket is een software-eindpunt waarmee een programma netwerkgegevens verzendt of ontvangt.

- Verbindt een toepassing met netwerkcommunicatie.
- Gebruikt bij communicatie onder meer een IP-adres en poortnummer.
- Denk aan een contactpunt tussen je programma en het netwerk.

2. IP-adres en poort



Samen helpen het IP-adres en poortnummer gegevens naar de juiste computer en toepassing te sturen.

- IP-adres: welke netwerkinterface of computer is het doel?
- Poort: met welke toepassing of dienst wordt gecommuniceerd?
- Python-variabelen kunnen bijvoorbeeld `'host'` en `'poort'` bevatten.

3. TCP: betrouwbaar en geordend



TCP biedt toepassingen een betrouwbare, geordende bytestroom tussen verbonden eindpunten.

- Verzendt gegevens als een doorlopende stroom bytes.
- Houdt de volgorde van de bytes bij en verzorgt hertransmissie waar nodig.
- De toepassing moet zelf bepalen waar berichten beginnen en eindigen.

Onthoud

TCP levert een bytestroom, geen kant-en-klare berichten.

8. Veilige verwerking en TLS



Een gewone socket versleutelt verkeer niet vanzelf. Bescherm gegevens en communicatie passend bij de situatie.

- Valideer invoer en behandel onverwachte gegevens zorgvuldig.
- Beperk wachttijd en hoeveelheid verwerkte gegevens.
- Gebruik TLS wanneer vertrouwelijkheid en integriteit tijdens transport nodig zijn.
- Controleer ook fouten en sluit de verbinding netjes.

Belangrijk

Een socket alleen biedt geen versleuteling. TLS is relevant wanneer gegevens tijdens transport vertrouwelijk en integer moeten blijven.

Basisprincipes van sockets

4. TCP-client-server en handshake



Een server luistert op een adres en poort; een client maakt verbinding. De TCP-handshake hoort bij TCP.

- Server: socket maken → `'bind'` → `'listen'` → `'accept'`.
- Client: socket maken → `'connect'`.
- Handshake: client SYN → server SYN-ACK → client ACK.
- Na het verbinden kunnen beide kanten gegevens verzenden en ontvangen.

Handshakeroute

SYN → SYN-ACK → ACK is de klassieke TCP-handshake; beeld deze niet af als UDP-stap.

7. Fouten, time-outs en opruimen



Netwerkbewerkingen kunnen mislukken of langer duren dan verwacht. Behandel fouten en beheer sockets als tijdelijke bronnen.

- Gebruik `'try'/'except'` om passende fouten af te handelen.
- Stel een passende time-out en limieten in.
- Houd rekening met gedeeltelijke lees- en schrijfsresultaten.
- Sluit sockets netjes; gebruik waar passend `'with'` of sluit ze expliciet.

Leerstep

De eenvoudige client-servervolgorde is een leerpatroon, geen volledige productie-implementatie.

6. Python-socketlevenscyclus



De standaardbibliotheekmodule `'socket'` geeft toegang tot socketfunctionaliteit. De precieze stappen hangen af van de gekozen socketsoort.

- Bewaar host en poort in variabelen; kies een socketsoort die bij de communicatie past.
- Gebruik bij TCP aan beide kanten de passende verbindingstappen.
- `'send'` verzendt bytes; `'recv'` leest bytes. Een leesbewerking kan minder gegevens opleveren dan verwacht.
- Zet ontvangen bytes zo nodig om naar tekst met een passende codering.

Klein woordenpaar

Een Python-string is tekst; sockets verzenden en ontvangen bytes. Zet gegevens bewust om tussen beide.

5. UDP: losse datagrammen



UDP verzendt zelfstandige datagrammen zonder voorafgaande verbindingshandshake.

- Heeft geen ingebouwde garantie op aflevering of volgorde.
- Verzorgt zelf geen hertransmissie.
- Een toepassing kan zo nodig zelf controles of hertransmissies toevoegen.

Vergelijk

TCP biedt een betrouwbare, geordende stroom; UDP verstuurt losse datagrammen zonder die ingebouwde garanties.

1 Pythonic principles



Schrijf Python zo dat de bedoeling snel duidelijk is. Kies leesbare, eenvoudige oplossingen en benut ingebouwde taalconstructies.

- Eén duidelijke taak per functie.
- Gebruik itereren waar dat code eenvoudiger maakt.
- Voorbeeld: `for naam in namen:` is vaak duidelijker dan handmatig indexen.

Onthoud: Pythonic is een streven naar leesbaarheid, geen aparte syntaxregel.

2 PEP 8-opmaak



PEP 8 beschrijft conventies voor consistente Python-code. Ze maken code makkelijker te lezen, maar bepalen niet op zichzelf of Python de code syntactisch accepteert.

- Gebruik vier spaties per inspringniveau; meng geen tabs en spaties.
- Zet spaties rond operatoren: `totaal = prijs + btw`.
- Houd regels overzichtelijk en groepeer verwante code.

3 Naamgeving en structuur



Kies namen die de rol van een waarde of functie verduidelijken. Houd de conventies consequent aan.

- Variabelen en functies: `'snake_case'`, bijvoorbeeld `'bereken_totaal'`.
- Klassen: `'CapWords'`, bijvoorbeeld `'Winkelmandje'`.
- Constanten: hoofdletters met underscores, bijvoorbeeld `'MAX_POGINGEN'`.
- Splits lange taken op in kleine functies met een duidelijke naam.

8 Kwaliteitsworkflow



Combineer leesbare code, consistente opmaak en passende controles. Tools vullen elkaar aan; geen enkele controle vervangt alle andere.

1. Schrijf of wijzig een kleine, begrijpelijke eenheid code.
2. Formateer de code en bekijk de wijziging.
3. Voer linter en eventuele statische typecontrole uit; behandel meldingen.
4. Voer relevante tests uit: tests controleren gedrag door code daadwerkelijk uit te voeren.

Kort onderscheid:

Formatter: past opmaak aan.

Linters: signaleert stijl- en mogelijke codeproblemen.

Typechecker: analyseert typegebruik.

Test: voert code uit om gedrag te controleren.

Codekwaliteit, stijl en Pythonic ontwerp

4 Type hints



Type hints annoteren welke typen je voor parameters, variabelen of retourwaarden verwacht. Ontwikkelhulpmiddelen kunnen die informatie gebruiken.

- Voorbeeld: `def verdubbel(getal: int) -> int:`
- Een hint is documentatie en invoer voor hulpmiddelen; Python controleert het type niet automatisch tijdens uitvoering.
- Kies hints die passen bij het gebruik en houd ze begrijpelijk.

Niet verwarren: Een type hint vervangt geen runtime-controle wanneer je programma invoer moet valideren.

7 Statische typecontrole



Statische typecontrole analyseert type-annotaties en code zonder het programma als normale uitvoering te draaien. Dat kan inconsistenties opsporen vóór uitvoering.

- Een typechecker gebruikt hints om verwachte typen met het codegebruik te vergelijken.
- Behandel meldingen als hulp bij analyse en onderzoek ze in context.
- Statische controle is iets anders dan runtime-tests.

6 Linters



Een linter onderzoekt broncode statisch en signaleert bijvoorbeeld stijlproblemen of mogelijke fouten, zonder het programma als normale uitvoering te draaien.

- Voorbeeld: Ruff kan als linter worden gebruikt.
- Lees meldingen; pas code aan of leg bewust uit waarom een melding niet van toepassing is.
- Een lintermelding is een aanwijzing om te beoordelen, geen bewijs dat de code altijd fout is.

5 Formatters



Een formatter past codeopmaak automatisch aan volgens een gekozen stijl. Zo voorkom je veel handmatige opmaakverschillen.

- Voorbeeld: Black kan Python-bestanden formatteren.
- Draai de formatter na wijzigingen of neem hem op in je editorworkflow.
- Controleer het resultaat en stem teamafspraken op elkaar af.