**Inhoudsopgave**

1. [Basis Print Functie]

2. [Variabelen en Gegevenssoorten]

3. [Operators]

4. [Lists, Tuples en Sets]

5. [Dictionaries]

6. [If-Else-verklaringen]

7. [For- en While-lussen]

8. [Functies](#8-functies)

9. [Lambda-functies]

10. [Modules en Import]

11. [Exception Handling]

12. [Bestandsinvoer en -uitvoer]

13. [Object-georiënteerd programmeren (OOP)]

14. [Inheritance (Overerving)

15. [Modules en Pakketten]

16. [List Comprehensions]

17. [Generators]

18. [Decorators]

19. [Context Managers]

20. [Date en Time]

**1. Basis Print Functie**

**\*\*Inleiding:\*\***

De `print()`-functie is een fundamentele functie in Python die wordt gebruikt om informatie naar de console te sturen. Het is essentieel voor het debuggen en communiceren met de gebruiker.

**\*\*Voorbeeld:\*\***

print("Hallo, wereld!")

**\*\*Uitvoer:\*\***

Hallo, wereld!

**\*\*Toepassingen:\*\***

**- \*\*Debugging:\*\***

Helpt bij het controleren van de waarden van variabelen tijdens de uitvoering van het programma.

**- \*\*Gebruikersinteractie:\*\***

Geeft instructies of informatie aan de gebruiker.

**\*\*Oefening:\*\***

- Schrijf een programma dat je naam en favoriete hobby afdrukt.

**2. Variabelen en Gegevenssoorten**

**\*\*Inleiding:\*\***

Variabelen zijn namen die verwijzen naar gegevens in het geheugen. Python is dynamisch getypt, wat betekent dat je geen type hoeft te declareren bij het toewijzen van een waarde.

**\*\*Gegevenssoorten:\*\***

**- \*\*String (`str`):\*\* Tekstuele data.**

naam = "Alice"`

**- \*\*Integer (`int`):\*\* Gehele getallen.**

leeftijd = 25`

**- \*\*Float (`float`):\*\* Decimale getallen.`**

pi = 3.1415`

**- \*\*Boolean (`bool`):\*\* Logische waarden `True` of `False`.**

is\_student = True`

**\*\*Voorbeeld:\*\***

naam = "Alice"

leeftijd = 25

pi = 3.1415

is\_student = True

print(f"Naam: {naam}")

print(f"Leeftijd: {leeftijd}")

print(f"Waarde van pi: {pi}")

print(f"Is student: {is\_student}")

**\*\*Uitvoer:\*\***

Naam: Alice

Leeftijd: 25

Waarde van pi: 3.1415

Is student: True

**\*\*Oefening:\*\***

- Declareer variabelen van elk type en druk ze af met behulp van `print()` en f-strings.

**\*\*Samenvatting:\*\***

- Variabelen slaan gegevens op voor later gebruik.

- Python ondersteunt verschillende gegevenssoorten die bepalen wat je met de gegevens kunt doen.

**3. Operators**

**\*\*Inleiding:\*\***

Operators worden gebruikt om bewerkingen uit te voeren op variabelen en waarden.

**\*\*Rekenkundige Operators:\*\***

- Optellen (`+`):\*\* “a + b”

- Aftrekken (`-`):\*\* “a - b”

- Vermenigvuldigen (`\*`):\*\* “a \* b”

- Delen (`/`):\*\* “a / b”

- Modulus (`%`):\*\* Rest van de deling “a % b”

- Exponentiatie (`\*\*`):\*\* Machtsverheffing “a \*\* b”

**\*\*Voorbeeld:\*\***

x = 10

y = 3

print(f"{x} + {y} = {x + y}") # Uitvoer: 10 + 3 = 13

print(f"{x} - {y} = {x - y}") # Uitvoer: 10 - 3 = 7

print(f"{x} \* {y} = {x \* y}") # Uitvoer: 10 \* 3 = 30

print(f"{x} / {y} = {x / y}") # Uitvoer: 10 / 3 = 3.333...

print(f"{x} % {y} = {x % y}") # Uitvoer: 10 % 3 = 1

print(f"{x} \*\* {y} = {x \*\* y}") # Uitvoer: 10 \*\* 3 = 1000

**\*\*Vergelijkingsoperators:\*\***

- Gelijk aan (`==`):\*\* `a == b`

- Niet gelijk aan (`!=`):\*\* `a != b`

- Groter dan (`>`):\*\* `a > b`

- Kleiner dan (`<`):\*\* `a < b`

**\*\*Voorbeeld:\*\***

a = 5

b = 10

print(a == b) # Uitvoer: False

print(a != b) # Uitvoer: True

print(a < b) # Uitvoer: True

print(a > b) # Uitvoer: False

**\*\*Oefening:\*\***

- Schrijf een programma dat twee getallen vraagt aan de gebruiker en alle rekenkundige en vergelijkingsoperators daarop toepast.

**\*\*Samenvatting:\*\***

- Operators zijn essentieel voor het uitvoeren van berekeningen en logische beslissingen in je code.

**4. Lists, Tuples en Sets**

**\*\*Inleiding:\*\***

Deze datastructuren worden gebruikt om meerdere items in één variabele op te slaan.

**\*\*Lists:\*\***

- Kenmerken: Geordend, veranderbaar, en toestaan van duplicaten.

- Syntax: `mijn\_list = [1, 2, 3]`

**\*\*Voorbeeld:\*\***

fruits = ["appel", "banaan", "kers"]

fruits.append("druif")

print(fruits)

**\*\*Uitvoer:\*\***

['appel', 'banaan', 'kers', 'druif']

**\*\*Tuples:\*\***

- \*\*Kenmerken:\*\* Geordend, onveranderbaar, en toestaan van duplicaten.

- \*\*Syntax:\*\* `mijn\_tuple = (1, 2, 3)`

**\*\*Voorbeeld:\*\***

coordinates = (10.0, 20.0)

print(coordinates)

**\*\*Uitvoer:\*\***

(10.0, 20.0)

**\*\*Sets:\*\***

- \*\*Kenmerken:\*\* Ongestructureerd, geen duplicaten.

- \*\*Syntax:\*\* `mijn\_set = {1, 2, 3}`

**\*\*Voorbeeld:\*\***

unique\_numbers = {1, 2, 3, 2, 1}

print(unique\_numbers)

**\*\*Uitvoer:\*\***

{1, 2, 3}

**\*\*Oefening:\*\***

- Maak een list van je favoriete films, voeg er een toe, en verander een bestaande titel.

**\*\*Samenvatting:\*\***

- Lists zijn ideaal voor geordende collecties die je kunt wijzigen.

- Tuples zijn handig voor gegevens die niet mogen veranderen.

- Sets zijn efficiënt voor het opslaan van unieke items.

**5. Dictionaries**

**\*\*Inleiding:\*\***

Dictionaries slaan gegevens op in key-value paren, wat snelle toegang tot waarden mogelijk maakt op basis van een unieke sleutel.

**\*\*Voorbeeld:\*\***

student = {

"naam": "Bob", "leeftijd": 20, "studie": "Informatica"

}

print(student["naam"]) # Uitvoer: Bob

# Waarde bijwerken

student["leeftijd"] = 21

print(student["leeftijd"]) # Uitvoer: 21

# Nieuwe key-value paar toevoegen

student["hobby"] = "Programmeren"

print(student)

**\*\*Uitvoer:\*\***

Bob

21

{'naam': 'Bob', 'leeftijd': 21, 'studie': 'Informatica', 'hobby': 'Programmeren'}

**\*\*Oefening:\*\***

- Maak een dictionary van jezelf met ten minste drie eigenschappen en druk deze af.

**\*\*Samenvatting:\*\***

- Dictionaries zijn krachtig voor het opslaan en beheren van gerelateerde gegevens.

**6. If-Else-verklaringen**

**\*\*Inleiding:\*\***

Met if-else-verklaringen kun je beslissingen nemen in je code op basis van voorwaarden.

**\*\*Voorbeeld:\*\***

leeftijd = int(input("Voer je leeftijd in: "))

if leeftijd >= 18:

print("Je bent een volwassene.")

elif leeftijd >= 13:

print("Je bent een tiener.")

else:

print("Je bent een kind.")

**\*\*Uitvoer (als de gebruiker '16' invoert):\*\***

Je bent een tiener.

**\*\*Oefening:\*\***

- Schrijf een programma dat de gebruiker vraagt om een getal en bepaalt of het positief, negatief of nul is.

**\*\*Samenvatting:\*\***

- If-else-structuren zijn essentieel voor het controleren van de stroom van een programma op basis van voorwaarden.

**7. For- en While-lussen**

**\*\*Inleiding:\*\***

Lussen worden gebruikt om herhaalde taken uit te voeren.

**\*\*For-lus Voorbeeld:\*\***

for i in range(5):

print(f"Iteratie {i}")

**\*\*Uitvoer:\*\***

Iteratie 0

Iteratie 1

Iteratie 2

Iteratie 3

Iteratie 4

**\*\*While-lus Voorbeeld:\*\***

count = 0

while count < 5:

print(f"Count is {count}")

count += 1

**\*\*Uitvoer:\*\***

Count is 0

Count is 1

Count is 2

Count is 3

Count is 4

**\*\*Oefening:\*\***

- Gebruik een for-lus om de letters van een woord afzonderlijk af te drukken.

**\*\*Samenvatting:\*\***

- Lussen automatiseren repetitieve taken en zijn onmisbaar in programmeren.

**8. Functies**

**\*\*Inleiding:\*\***

Functies zijn blokken code die hergebruikt kunnen worden om specifieke taken uit te voeren, wat je code georganiseerder en efficiënter maakt.

**\*\*Voorbeeld:\*\***

def bereken\_oppervlakte(breedte, hoogte):

return breedte \* hoogte

oppervlakte = bereken\_oppervlakte(5, 10)

print(f"De oppervlakte is {oppervlakte}")

**\*\*Uitvoer:\*\***

De oppervlakte is 50

**\*\*Oefening:\*\***

- Schrijf een functie die de faculteit van een gegeven getal berekent.

**\*\*Samenvatting:\*\***

- Functies bevorderen codehergebruik en verbeteren de leesbaarheid.

**9. Lambda-functies**

**\*\*Inleiding:\*\***

Lambda-functies zijn anonieme, inline functies die handig zijn voor eenvoudige bewerkingen.

**\*\*Voorbeeld:\*\***

# Normale functie

def vermenigvuldig(a, b):

return a \* b

# Lambda-functie

vermenigvuldig\_lambda = lambda a, b: a \* b

print(vermenigvuldig(5, 6)) # Uitvoer: 30

print(vermenigvuldig\_lambda(5, 6)) # Uitvoer: 30

**\*\*Oefening:\*\***

- Gebruik een lambda-functie met `filter()` om alle even getallen uit een lijst te halen.

**\*\*Samenvatting:\*\***

- Lambda-functies zijn beknopt en nuttig voor eenvoudige bewerkingen, vooral in combinatie met functies als `map()` en `filter()`.

**10. Modules en Import**

**\*\*Inleiding:\*\***

Modules zijn bestanden met Python-code (functies, klassen, variabelen) die je in andere Python-programma's kunt importeren om functionaliteit te hergebruiken.

**\*\*Voorbeeld met `math`-module:\*\***

import math

# Gebruik van functies uit de math-module

print(math.sqrt(16)) # Uitvoer: 4.0

print(math.pi) # Uitvoer: 3.141592653589793

# Specifieke functie importeren

from math import factorial

print(factoriaal(5)) # Uitvoer: 120

**\*\*Oefening:\*\***

- Importeer de `random` module en genereer een willekeurig getal tussen 1 en 10.

**\*\*Samenvatting:\*\***

- Modules vergroten de mogelijkheden van je programma zonder dat je alle code zelf hoeft te schrijven.

**11. Exception Handling**

**\*\*Inleiding:\*\***

Exception handling wordt gebruikt om fouten (exceptions) in je code op te vangen en af te handelen. Hierdoor voorkom je dat je programma crasht en kun je fouten op een gecontroleerde manier afhandelen.

**\*\*Basisconcepten:\*\***

- \*\*Try-except-blokken:\*\* Gebruik `try` om een blok code te markeren waar een fout kan optreden, en `except` om de fout af te handelen.

- \*\*Specifieke exceptions:\*\* Je kunt specifieke fouten afvangen, zoals `ZeroDivisionError` of `ValueError`.

**\*\*Voorbeeld:\*\***

try:

getal = int(input("Voer een getal in: "))

resultaat = 10 / getal

print(f"Resultaat is: {resultaat}")

except ZeroDivisionError:

print("Kan niet delen door nul.")

except ValueError:

print("Dat is geen geldig getal.")

**\*\*Mogelijke Uitvoer:\*\***

- Als de gebruiker '0' invoert:

Kan niet delen door nul.

- Als de gebruiker 'abc' invoert:

Dat is geen geldig getal.

**\*\*Oefening:\*\***

- Schrijf een programma dat twee getallen van de gebruiker vraagt en het resultaat van de deling weergeeft. Voeg exception handling toe om fouten zoals delen door nul en ongeldige invoer af te handelen.

**\*\*Samenvatting:\*\***

- Exception handling maakt je code robuuster en gebruikersvriendelijker door fouten netjes af te handelen.

**12. Bestandsinvoer en -uitvoer**

**\*\*Inleiding:\*\***

Met bestandsinvoer en -uitvoer kun je gegevens opslaan in en lezen uit bestanden, wat essentieel is voor het bewaren van gegevens tussen sessies.

**\*\*Basisoperaties:\*\***

- \*\*Openen van bestanden:\*\* Met `open()` kun je bestanden openen in verschillende modi, zoals lezen ('r'), schrijven ('w') en toevoegen ('a').

- \*\*Automatisch sluiten:\*\* Gebruik `with` om bestanden automatisch te sluiten na gebruik.

**\*\*Schrijven naar een bestand:\*\***

with open('voorbeeld.txt', 'w') as bestand:

bestand.write("Dit is een voorbeeld.")

**\*\*Lezen van een bestand:\*\***

with open('voorbeeld.txt', 'r') as bestand:

inhoud = bestand.read()

print(inhoud)

**\*\*Uitvoer:\*\***

Dit is een voorbeeld.

**\*\*Oefening:\*\***

- Schrijf een programma dat namen van gebruikers vraagt en deze opslaat in een tekstbestand. Laat het programma vervolgens de namen uit het bestand lezen en afdrukken.

**\*\*Samenvatting:\*\***

- Bestandsinvoer en -uitvoer zijn cruciaal voor het persistent opslaan van gegevens en het werken met grotere datasets.

**13. Object-georiënteerd programmeren (OOP)**

**\*\*Inleiding:\*\***

OOP is een programmeerparadigma dat gebruikmaakt van objecten en klassen om code te structureren. Het helpt bij het modelleren van complexe systemen door middel van objecten met eigenschappen (attributen) en gedrag (methoden).

**\*\*Basisconcepten:\*\***

- \*\*Klasse:\*\* Een blauwdruk voor objecten.

- \*\*Object:\*\* Een instantie van een klasse.

- \*\*Attributen:\*\* Eigenschappen van een object.

- \*\*Methoden:\*\* Functies die horen bij een object.

**\*\*Voorbeeld:\*\***

class Auto:

def \_\_init\_\_(self, merk, model, bouwjaar):

self.merk = merk

self.model = model

self.bouwjaar = bouwjaar

def beschrijving(self):

return f"{self.merk} {self.model} ({self.bouwjaar})"

# Object aanmaken

mijn\_auto = Auto("Toyota", "Corolla", 2020)

print(mijn\_auto.beschrijving())

```

**\*\*Uitvoer:\*\***

Toyota Corolla (2020)

**\*\*Oefening:\*\***

- Maak een klasse `Student` met attributen `naam`, `leeftijd` en `studie`. Voeg een methode toe die de studentgegevens afdrukt.

**\*\*Samenvatting:\*\***

- OOP bevordert hergebruik en modulariteit in code, waardoor complexe programma's beter beheersbaar worden.

**14. Inheritance (Overerving)**

**\*\*Inleiding:\*\***

Overerving stelt je in staat een nieuwe klasse te maken die eigenschappen en methoden van een bestaande klasse overneemt. Dit bevordert codehergebruik en maakt het mogelijk om hiërarchieën te creëren.

**\*\*Basisconcepten:\*\***

**- \*\*Basisklasse (Superklasse):\*\***  De klasse waarvan wordt overgeërfd.

**- \*\*Subklasse (Afgeleide klasse):\*\*** De klasse die overerft van de basisklasse.

**\*\*Voorbeeld:\*\***

class Dier:

def \_\_init\_\_(self, naam):

self.naam = naam

def geluid\_maken(self):

pass

class Hond(Dier):

def geluid\_maken(self):

return "Woef!"

# Object aanmaken

mijn\_hond = Hond("Rex")

print(f"{mijn\_hond.naam} zegt: {mijn\_hond.geluid\_maken()}")

```

**\*\*Uitvoer:\*\***

Rex zegt: Woef!

**\*\*Oefening:\*\***

- Maak een klasse `Vogel` die overerft van `Dier` en implementeer de methode `geluid\_maken` met een passend geluid.

**\*\*Samenvatting:\*\***

- Overerving maakt het mogelijk om code te delen tussen klassen en hiërarchieën te creëren.

**15. Modules en Pakketten**

**\*\*Inleiding:\*\***

Modules en pakketten helpen je code te organiseren en hergebruiken door het in afzonderlijke bestanden en mappen te plaatsen. Hierdoor blijft je codebase overzichtelijk en onderhoudbaar.

**\*\*Modules:\*\***

- Een module is een Python-bestand met extensie `.py` dat functies, klassen en variabelen kan bevatten.

**\*\*Pakketten:\*\***

- Een pakket is een map met modules en een `\_\_init\_\_.py` bestand.

**\*\*Voorbeeld:\*\***

**1. \*\*Module aanmaken (`hulpfuncties.py`):\*\***

def groet(naam):

print(f"Hallo, {naam}!")

**2. \*\*Module importeren en gebruiken:\*\***

import hulpfuncties

hulpfuncties.groet("Alice")

**\*\*Uitvoer:\*\***

Hallo, Alice!

**\*\*Oefening:\*\***

- Maak een module `wiskunde.py` met functies voor optellen en aftrekken, en importeer deze in je hoofdscript.

**\*\*Samenvatting:\*\***

- Modules en pakketten bevorderen codehergebruik en zorgen voor een betere organisatie van je code.

**16. List Comprehensions**

**\*\*Inleiding:\*\***

List comprehensions bieden een beknopte en expressieve manier om nieuwe lijsten te creëren op basis van bestaande lijsten.

**\*\*Syntax:\*\***

[uitdrukking for item in iterabele if voorwaarde]

**\*\*Voorbeeld:\*\***

**- \*\*Kwadraten van getallen 1 tot 5:\*\***

kwadraten = [x\*\*2 for x in range(1, 6)]

print(kwadraten)

**\*\*Uitvoer:\*\***

[1, 4, 9, 16, 25]

**- \*\*Even getallen uit een lijst:\*\***

getallen = [1, 2, 3, 4, 5, 6]

even\_getallen = [x for x in getallen if x % 2 == 0]

print(even\_getallen)

**\*\*Uitvoer:\*\***

[2, 4, 6]

**\*\*Oefening:\*\***

- Gebruik een list comprehension om alle woorden met meer dan vier letters uit een lijst van woorden te filteren.

**\*\*Samenvatting:\*\***

- List comprehensions maken je code korter en vaak beter leesbaar.

**17. Generators**

**\*\*Inleiding:\*\***

Generators zijn speciale functies die waarden één voor één opleveren met het `yield`-statement. Ze worden gebruikt voor het efficiënt itereren over grote datasets zonder al die data in het geheugen te laden.

**\*\*Voorbeeld:\*\***

def getallen\_generator(n):

i = 0

while i < n:

yield i

i += 1

for getal in getallen\_generator(5):

print(getal)

**\*\*Uitvoer:\*\***

0

1

2

3

4

**\*\*Oefening:\*\***

- Schrijf een generatorfunctie `oneven\_getallen(n)` die de eerste `n` oneven getallen oplevert.

**\*\*Samenvatting:\*\***

- Generators zijn efficiënt voor het werken met grote of oneindige reeksen, omdat ze waarden produceren op het moment dat ze nodig zijn.

**18. Decorators**

**\*\*Inleiding:\*\***

Decorators zijn een krachtige tool in Python om de functionaliteit van functies of methoden aan te passen zonder de oorspronkelijke code te wijzigen.

**\*\*Syntax:\*\***

@decorator

def functie():

pass

**\*\*Voorbeeld:\*\***

def groet\_decorator(functie):

def wrapper(\*args, \*\*kwargs):

print("Welkom!")

return functie(\*args, \*\*kwargs)

return wrapper

@groet\_decorator

def groet(naam):

print(f"Hallo, {naam}!")

groet("Bob")

**\*\*Uitvoer:\*\***

Welkom!

Hallo, Bob!

**\*\*Oefening:\*\***

- Schrijf een decorator `tijdsmeting` die de uitvoeringstijd van een functie meet en afdrukt.

**\*\*Samenvatting:\*\***

- Decorators zijn handig voor het toevoegen van cross-cutting concerns zoals logging, authenticatie of timing.

**19. Context Managers**

**\*\*Inleiding:\*\***

Context managers worden gebruikt om bronnen veilig te beheren. Het `with`-statement zorgt ervoor dat belangrijke opruimacties worden uitgevoerd, zoals het sluiten van een bestand.

**\*\*Voorbeeld:\*\***

with open('voorbeeld.txt', 'w') as bestand:

bestand.write("Dit is een voorbeeld.")

# Het bestand wordt hier automatisch gesloten.

**\*\*Eigen Context Manager Maken:\*\***

class MijnContextManager:

def \_\_enter\_\_(self):

print("Context geopend")

return self

def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):

print("Context gesloten")

with MijnContextManager() as manager:

print("Binnen de context")

**\*\*Uitvoer:\*\***

Context geopend

Binnen de context

Context gesloten

**\*\*Oefening:\*\***

- Maak een context manager die een databaseverbinding opent en sluit.

**\*\*Samenvatting:\*\***

- Context managers zorgen voor nettere en veiligere code bij het werken met bronnen die moeten worden opgeruimd.

**20. Date en Time**

**\*\*Inleiding:\*\***

Met de `datetime` module kun je werken met datums en tijden in Python, wat essentieel is voor applicaties die tijdsgerelateerde gegevens verwerken.

**\*\*Voorbeeld:\*\***

import datetime

# Huidige datum en tijd

nu = datetime.datetime.now()

print("Nu:", nu)

# Datum formatteren

geformatteerd = nu.strftime("%d-%m-%Y %H:%M:%S")

print("Geformatteerde datum en tijd:", geformatteerd)

# Datum parsing

datum\_string = "01-01-2022 12:00:00"

parsed\_datum = datetime.datetime.strptime(datum\_string, "%d-%m-%Y %H:%M:%S")

print("Parsed datum:", parsed\_datum)

**\*\*Uitvoer (voorbeeld):\*\***

Nu: 2023-10-05 14:30:00.123456

Geformatteerde datum en tijd: 05-10-2023 14:30:00

Parsed datum: 2022-01-01 12:00:00

**\*\*Oefening:\*\***

- Schrijf een programma dat het aantal dagen berekent tussen vandaag en een door de gebruiker ingevoerde datum.

**\*\*Samenvatting:\*\***

- De `datetime` module is essentieel voor het verwerken en manipuleren van datums en tijden in Python.

**21. \*\*Regular Expressions (Reguliere Expressies)\*\***

Regular expressions (regex) zijn patronen die je kunt gebruiken om tekst te doorzoeken of te bewerken. De `re`-module bevat functies om regex te gebruiken.

**\*\*Voorbeeld\*\*:**

import re

tekst = "Het nummer is 123-456-7890."

patroon = r"\d{3}-\d{3}-\d{4}"

match = re.search(patroon, tekst)

if match:

print("Gevonden:", match.group())

```

**22. \*\*JSON Parsing\*\***

JSON (JavaScript Object Notation) wordt vaak gebruikt om gegevens uit te wisselen. Met de `json`-module kun je JSON-gegevens inlezen en schrijven.

**\*\*Voorbeeld\*\*:**

import json

# JSON-string parsen naar Python-dictionary

json\_data = '{"naam": "Alice", "leeftijd": 30}'

data = json.loads(json\_data)

print(data["naam"]) # Uitvoer: Alice

# Python-dictionary omzetten naar JSON-string

python\_data = {"stad": "Amsterdam", "land": "Nederland"}

json\_string = json.dumps(python\_data)

print(json\_string)

```

**23. \*\*Exception Hierarchy en Custom Exceptions\*\***

Met custom exceptions kun je eigen fouttypes definiëren, handig voor specifieke foutafhandeling.

**\*\*Voorbeeld\*\*:**

class MijnFout(Exception):

pass

def deel(a, b):

if b == 0:

raise MijnFout("Delen door nul is niet toegestaan.")

return a / b

try:

resultaat = deel(10, 0)

except MijnFout as e:

print(e)

**24. \*\*Unit Testing\*\***

Unit testing is het testen van afzonderlijke onderdelen van je code. In Python wordt hiervoor vaak de `unittest`-module gebruikt.

**\*\*Voorbeeld\*\*:**

import unittest

def vermenigvuldig(a, b):

return a \* b

class TestVermenigvuldig(unittest.TestCase):

def test\_vermenigvuldig(self):

self.assertEqual(vermenigvuldig(2, 5), 10)

self.assertEqual(vermenigvuldig(-1, 1), -1)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**25. \*\*Virtual Environments\*\***

Een virtual environment creëert een geïsoleerde Python-omgeving voor een project, handig voor afhankelijkheden en versies.

**\*\*Commando's\*\*:**

python -m venv mijn\_omgeving # Aanmaken

mijn\_omgeving\Scripts\activate # Activeren op Windows

source mijn\_omgeving/bin/activate # Activeren op Unix/MacOS

deactivate # Deactiveren

**26. \*\*Werken met Databases (SQLite)\*\***

SQLite is een lichtgewicht databasesysteem dat vaak wordt gebruikt in kleine toepassingen. Python heeft een ingebouwde module, `sqlite3`, voor interactie met SQLite-databases.

**\*\*Voorbeeld\*\*:**

import sqlite3

conn = sqlite3.connect('voorbeeld.db')

c = conn.cursor()

# Tabel aanmaken

c.execute('''CREATE TABLE IF NOT EXISTS studenten

(id INTEGER PRIMARY KEY, naam TEXT, leeftijd INTEGER)''')

# Gegevens invoegen

c.execute("INSERT INTO studenten (naam, leeftijd) VALUES (?, ?)", ('Alice', 22))

conn.commit()

# Gegevens ophalen

c.execute("SELECT \* FROM studenten")

print(c.fetchall())

conn.close()

**27. \*\*Threading en Multiprocessing\*\***

Met `threading` en `multiprocessing` kun je taken parallel uitvoeren, wat handig kan zijn voor het versnellen van processen.

**\*\*Voorbeeld (Threading)\*\*:**

import threading

def print\_getallen():

for i in range(5):

print(i)

thread = threading.Thread(target=print\_getallen)

thread.start()

thread.join()

**28. \*\*Web Frameworks (Flask)\*\***

Flask is een micro webframework waarmee je snel webapplicaties kunt maken in Python.

**\*\*Voorbeeld\*\*:**

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def home():

return "Welkom op de homepagina!"

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**29. \*\*Data Science Libraries\*\***

Python bevat krachtige bibliotheken voor data-analyse, zoals `pandas` voor datamanipulatie en `numpy` voor numerieke berekeningen.

**\*\*Voorbeeld met Pandas\*\*:**

import pandas as pd

data = {

'Naam': ['Alice', 'Bob', 'Charlie'],

'Leeftijd': [25, 30, 35]

}

df = pd.DataFrame(data)

print(df)

**30. \*\*Matplotlib voor Visualisatie\*\***

Matplotlib is een veelgebruikte bibliotheek voor data-visualisatie. Met `plt.plot()` kun je grafieken maken.

**\*\*Voorbeeld\*\*:**

import matplotlib.pyplot as plt

x = [1, 2, 3, 4]

y = [10, 20, 25, 30]

plt.plot(x, y)

plt.xlabel('X-as')

plt.ylabel('Y-as')

plt.title('Voorbeeld Grafiek')

plt.show()

**## Algemene Tips**

**- \*\*Consistente Indentatie:\*\* Zorg ervoor dat je code netjes is ingesprongen; Python gebruikt indentatie om codeblokken te definiëren.**

**- \*\*Commentaar Toevoegen:\*\* Gebruik `#` om commentaar in je code te schrijven, wat helpt bij het begrijpen en onderhouden van je code.**

**- \*\*Leesbaarheid:\*\* Gebruik duidelijke variabelenamen en houd je code overzichtelijk.**

**---**

**\*\*Extra Oefeningen:\*\***

**1. \*\*Gebruikersinvoer:\*\***

**- Vraag de gebruiker om zijn naam en begroet hem met een persoonlijke boodschap.**

**2. \*\*Berekeningen:\*\***

**- Schrijf een programma dat de som, het verschil, het product en het quotiënt van twee ingevoerde getallen berekent.**

**3. \*\*Gegevensstructuren:\*\***

**- Maak een dictionary van drie studenten met hun cijfers en bereken het gemiddelde cijfer.**