Python

PR3 | IB3

History of Python

Der Schöpfer

Guido van Rossum begann die Entwicklung 1989 am CWI in den Niederlanden.

Die Anfänge

Was als Hobbyprojekt während der Weihnachtsferien startete, führte 1991 zum Release.

Inspiration

Nachfolger der Sprache ABC, mit Fokus auf einfache Syntax und Lesbarkeit.

Namensgebung

Benannt nach *"Monty Python's Flying Circus"* - nicht nach der Schlange.

Evolution of Python


```
// Syntax Vergleich
print "Hello World" # Python 2
print("Hello World") # Python 3
```

Fokus auf bessere Lesbarkeit und Modernisierung

Zen of Python

Simple is better than complex

Code soll möglichst einfach und verständlich bleiben. Komplexität wird nach Möglichkeit vermieden.

Readability counts

Gut lesbarer Code ist besonders wichtig. Code wird öfter gelesen als geschrieben.

One obvious way to do it

Für Probleme soll es möglichst eine klare, standardisierte Lösung geben.

Verfasst von Tim Peters — Ausgabe in Python durch: **import this**

Einfache & produktive Syntax

Python

Weniger Boilerplate-Code, schnelle Entwicklung und sehr einfach zu erlernen.

```
print(“Hello World”)
```

Java (Zum Vergleich)

Erfordert wesentlich mehr strukturellen Code für die exakt gleiche Ausgabe.

```
public class Main {
    public static void main(String [] args) {
        System.out.println(“Hello World”);
    }
}
```

Libraries & Frameworks

Web Development

Riesige Auswahl an leistungsstarken Frameworks wie **Django**, **Flask** und **FastAPI** für Backend & APIs.

AI & Data Science

Branchenstandard durch etablierte Bibliotheken wie **TensorFlow**, **PyTorch**, **Pandas** und **NumPy**.

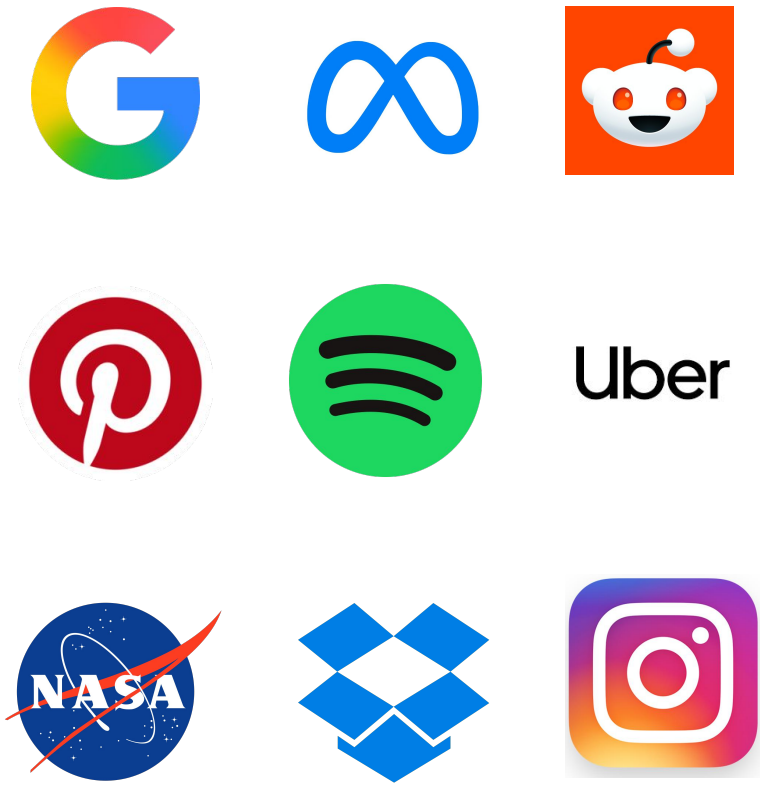
Automatisierung

Schnelle Entwicklung durch wiederverwendbaren Code mit Tools wie **Selenium** und **Requests**.

Python in der Industrie

- **KI & Data Science:** Google, Spotify, Meta
- **Web & Backend:** Instagram, Reddit, Pinterest
- **Automatisierung:** Uber, Dropbox
- **Wissenschaft:** NASA, CERN
- **Embedded/IoT:** Raspberry Pi, MicroPython

Besonders relevant für KI, Webentwicklung und Datenanalyse. Sehr hohe Nachfrage auf dem Arbeitsmarkt.



Nachteile von Python

Performance

Langsamer als kompilierte Sprachen wie C++ oder Java durch den Interpreter-Ansatz.

Weniger geeignet für sehr performancekritische Anwendungen (Execution Speed).

Ressourcen & Plattformen

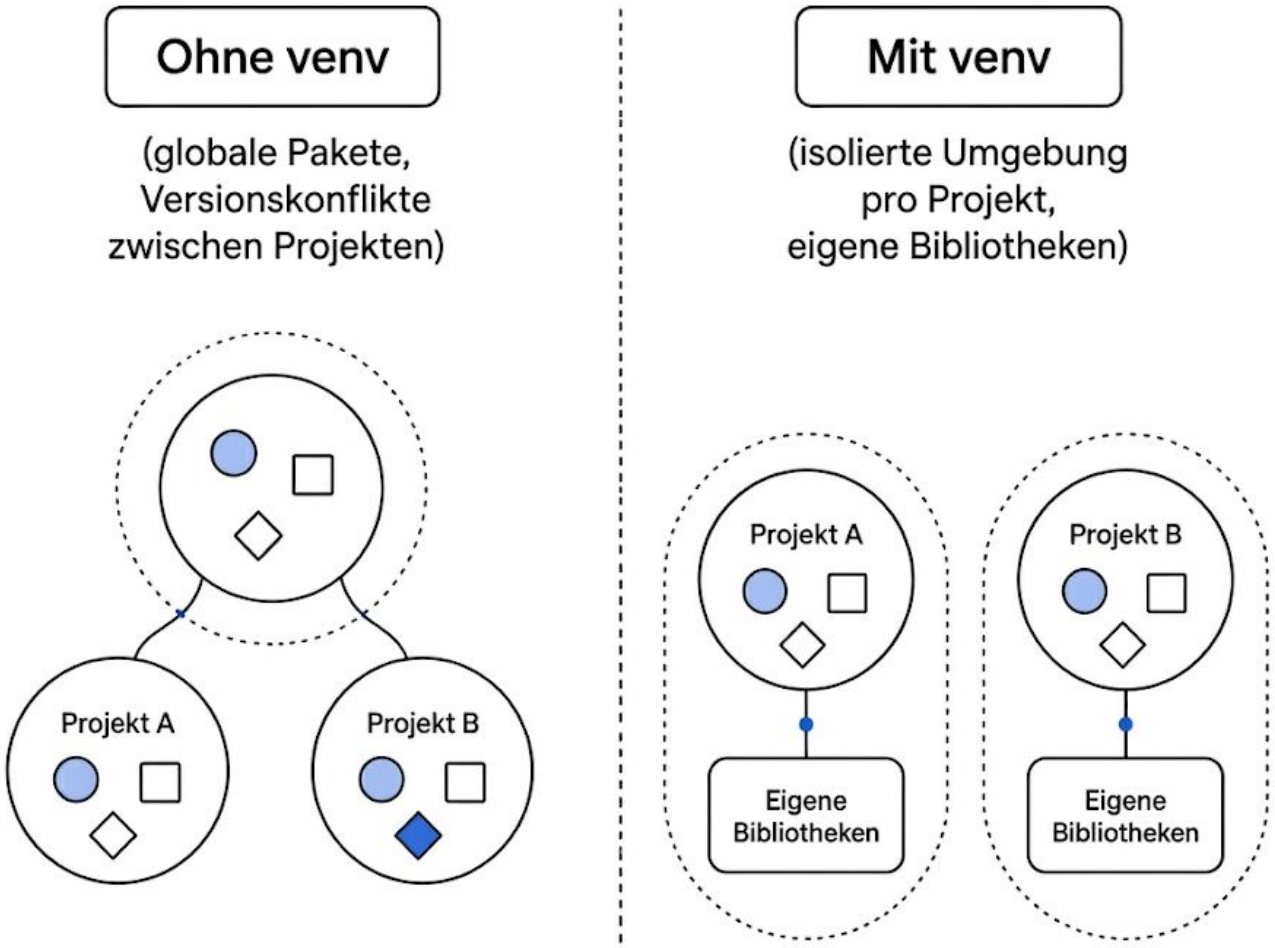
Höherer Speicherverbrauch im Vergleich zu hardwarenahen Sprachen.

Zudem weniger verbreitet in der nativen mobilen App-Entwicklung.

Tooling in Python

Definition: Tooling bezeichnet die Gesamtheit aller Softwarewerkzeuge, die den Kernprozess der Programmierung unterstützen, automatisieren und optimieren.

Venv – Virtuelle Umgebungen



Warum venv?

Abgeschlossene Umgebung pro Projekt – keine Versionskonflikte

Befehle

- `python3 -m venv .venv`
- Mac/Linux: `source .venv/bin/activate`
- Windows: `venv\Scripts\activate`

Teilen

- `pip freeze > requirements.txt`
- `pip install -r requirements.txt`

Ruff - Code Analyse und Formatierung

All-in-One

Vereint Flake8, isort u.a.

Schnell

Programmiert in Rust

Funktionen

Syntaxfehler, Stil, Formatierung

Fehlererkennung

```
ruff check .
```

Formatierung

```
ruff format .
```

Testen in Python

pytest

- Python, leichtgewichtig
- Tests als Funktionen
- Automatisch: `test_*.py`
- Einfach: `assert`
- Flexible Fehlerbehebung

```python
def test_add():
    assert add(2, 3) == 5
```

JUnit (Zum Vergleich)

- Java, stark strukturiert
- Tests meist in Klassen
- Automatisch via `@Test`
- Eigene Assertions
- Setup mit `@BeforeEach`

```java
class MathTest {
    @Test
    void testAdd() {
        assertEquals(5, add(2, 3));
    }
}
```

TOML – Konfigurationsdatei

Beispiel

```toml
[project]
name = "mein_projekt"
version = "1.0.0"

[tool.ruff]
line-length = 88
```

Syntax

- `[project]` – Metadaten des Projekts
- `[tool.*]` – Konfiguration einzelner Tools
- `[build-system]` – Build-Backend und Anforderungen

Python Debugger – pdb

Breakpoint setzen

```
breakpoint()
```

Programm wird angehalten, Variablen können untersucht werden

Wichtige Befehle

- `p variable` – Wert anzeigen
- `n` – Nächste Zeile
- `s` – In Funktion springen
- `c` – Bis nächster Breakpoint
- `q` – Beenden

Getting started

Grundlagen

- Das erste Python-Programm
- Variablen und Datentypen
- Arbeiten mit Variablen

Datenstrukturen

- Listen (Lists)
- Dictionaries
- Sets & Tuples

Anwendung

- Kontrollstrukturen & Schleifen
- Funktionen (def)

Variablen, Datentypen

Datentyp	Python Kürzel	Beschreibung & Beispiel
String	str	Zeichenketten: "Max", 'Python'
Integer	int	Ganzzahlen: 22, -5, 1000
Float	float	Gleitkommazahlen (mit Punkt): 1.80, 3.14
Boolean	bool	Wahrheitswerte: True oder False

Variablen, Datentypen und Rechenoperationen

Dynamische Typisierung

Dynamic Typing

Python erkennt den Datentyp automatisch anhand des Wertes.
Eine explizite Definition (wie in Java/C++) entfällt.

Flexibilität

Variablen können ihren Typ zur Laufzeit ändern. x = 5 kann später zu x = "Text" werden.

```
name = "Max"
age = 22
height = 1.80
student = True
```

Ausgabe:

```
Max
22
1.8
True
```

Type Casting

Häufig liegen Daten im falschen Format vor (z. B. Benutzereingaben als Text). Casting-Funktionen erlauben die Korrektur:

- int("42") → Wandelt Text in Ganzzahl um
- float("3") → Erzeugt 3.0
- str(100) → Macht aus einer Zahl Text

```
# Macht aus dem String einen echten Integer
echte_zahl = int("42")
print(echte_zahl)
```

```
# Macht aus dem String einen Float
komma_zahl = float("3")
```

```
# Macht aus der Zahl 100 den String "100"
text_wieder = str(100)
```

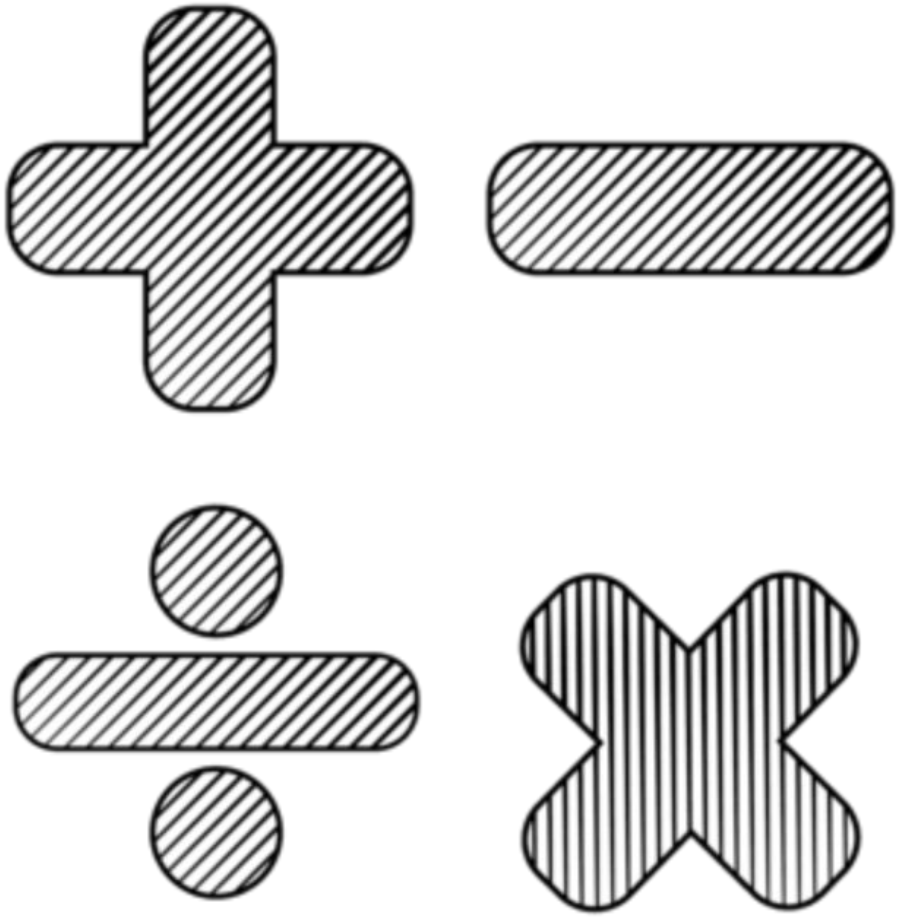
Rechenoperatoren

Neben den Standardoperationen bietet Python spezielle Operatoren:

Ganzzahldivision: 10 // 3 = 3

Exponent: 10 ** 3 = 1000

Modulo (%): Berechnet den Rest (10 % 3 = 1)



Hinweis: Eine normale Division liefert in Python immer einen float!

Arbeiten mit Listen

Lists

Mutable & Indexierung

Listen sind veränderbare Datenstrukturen. Der Zugriff erfolgt über Indizes.

0-basiert: Das erste Element hat den Index 0.

-1 : Mit [-1] greift auf das letzte Element zu.

List-Funktionen

`.append()` - Ende hinzufügen
`.insert()` - Exakt einschieben
`.remove()` - Wert löschen
`.pop()` - Letztes Element entfernen

Dictionaries

- Dictionaries speichern Daten in Paaren.
- Der Zugriff ist extrem schnell, unabhängig von der Größe.

```
auto = {
    "marke": "BMW",
    "modell": "M3",
    "baujahr": 2022
}

# Zugriff über den Schlüssel
print("Modell:", auto["modell"])
```

Ausgabe:

```
Modell: M3
```

Sets und Tupel

Sets (Mengen)

Einzigartig & Ungeordnet. Filtern Duplikate automatisch. Basieren auf Hashes, daher keine feste Reihenfolge möglich.
Werden in geschweiften {} Klammern erstellt

Tuples (Tupel)

Unveränderbar (Immutable). Ideal für Datensicherheit und Fixwerte (z.B. Koordinaten).
Werden mit runden Klammern () erstellt.

Kontrollstrukturen: if, elif, else

Kontrollstrukturen

- **if (Wenn):** Die erste Prüfung.
- **elif (Oder wenn):** Die spezifische Alternative.
- **else (Sonst):** Der automatische Notfallplan.

```
note = 2

if note == 1:
    print("Ausgezeichnet")
elif note <= 4:
    print("Bestanden")
else:
    print("Nicht bestanden")
```

Ausgabe:

```
Bestanden
```

Logik und Schleifen

Schleifen

In Python hat jedes Objekt eine "Wahrheit":

False: 0, "", [], {}, None

True: Alles, was Inhalt besitzt.

For-Schleife

Zähler- oder Elementgesteuert. Nutzt range() für definierte Wiederholungen.

```
for zaehler in range(1, 4):
    print("Durchgang Nummer:", zaehler)
```

While-Schleife

Läuft unendlich lange, solange eine Bedingung auf True steht.

```
countdown = 3
while countdown > 0:
    print("Countdown:", countdown)
    countdown = countdown - 1 # Bedingung verändern
```

Keywords

continue überspringt einen Durchlauf,
break beendet die gesamte Schleife.

```
for nummer in range(1, 6):
    if nummer == 3:
        continue # Die 3 wird übersprungen
    if nummer == 5:
        print('hello 5')
        break # Bei 5 wird die komplette Schleife beendet
```

Wie schreiben wir eine Funktion in Python?

Funktionen

Funktionen kapseln Code und vermeiden Redundanz.

Schlüsselwort: def

Syntax: def name(parameter):

Return: Schickt ein Datenobjekt zurück. Beendet die Funktion sofort.

Funktionsaufruf: Befehl um Funktion mit konkreten Argumenten auszuführen.

```python
def berechne_quadrat(zahl):
    ergebnis = zahl * zahl
    return ergebnis

#Funktion verwenden
meine_quadratzahl = berechne_quadrat(5)

# Jetzt können wir mit der Variable weiterrechnen
print("Das Doppelte der Quadratzahl ist:", meine_quadratzahl * 2)
```

```
Das Doppelte der Quadratzahl ist: 50

[Execution complete with exit code 0]
```

Covering Interesting Python Aspects

Syntax-Philosophie

Python's minimalist approach: readability and explicit code execution.

Dynamic & Duck Typing

"If it walks like a duck and quacks like a duck, it's a duck."

List Comprehensions

Compact and powerful syntax for creating and transforming sequences.

Decorators

Modifying function behavior dynamically with the elegant @ wrapper.

Magic / Dunder Methods

Unlocking Python's internal operator overloading with __double__ underscores.

Syntax-Philosophie

Warum braucht Python keine Klassen für "Hello World"?

Der Globale Scope

- Direkte Code-Ausführung auf der obersten Ebene der Datei.
- Kein erzwungenes Klassen-Korsett für einfache Skripte.

Minimalistischer Ansatz

- Der Code wird rein sequenziell von oben nach unten gelesen.
- Vollständige Eliminierung von struktuellem “Boilerplate-Code”.

Python

vs.

Java

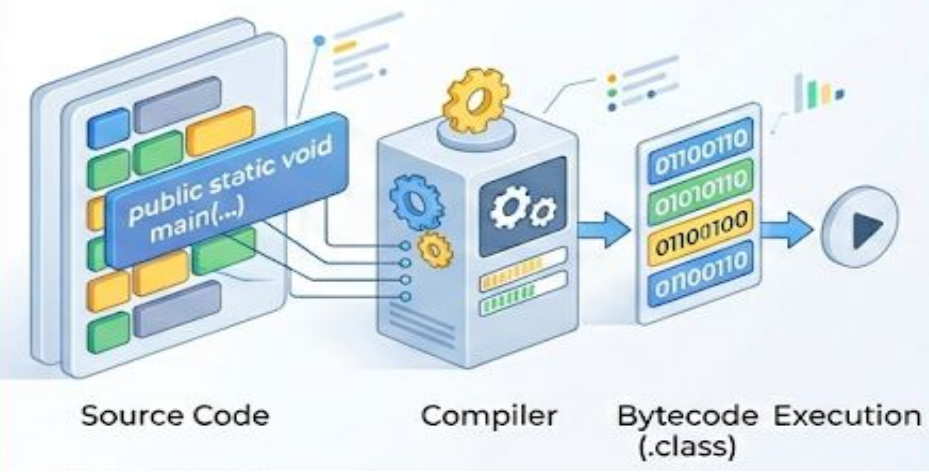
Syntax-Philosophie

Compiler vs. Interpreter im Vergleich

Java (Die Compiler-Welt)

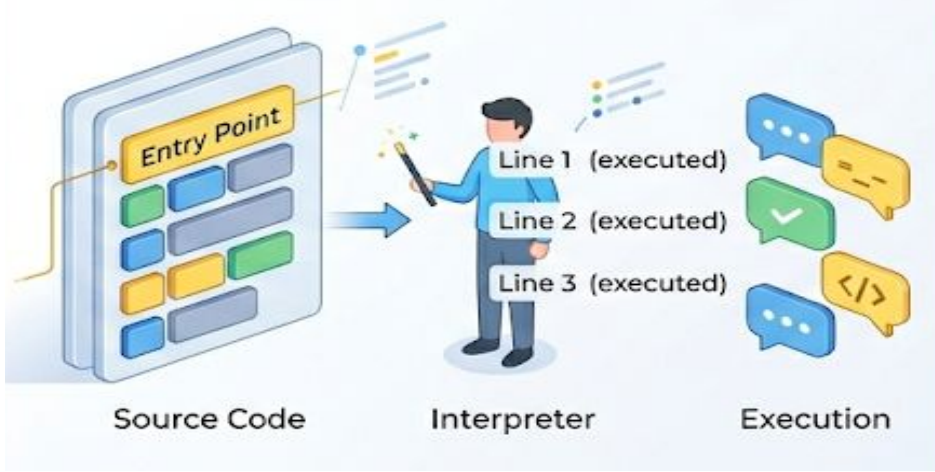
- Code-Analyse und Bytecode-Übersetzung vor dem Start.
- Zwingende Vorgabe eines festen Einstiegspunkts:

```
public static void main
```



Python (Die Interpreter-Welt)

- Zeile-für-Zeile Übersetzung und Ausführung direkt zur Laufzeit (Runtime).
- Die allererste Zeile im Skript ist automatisch der Entry Point.



Dynamische Typisierung

Warum haben Variablen keinen festen Typ?

Das Konzept der Labels

- Variablen sind in Python nur "Labels" (Etiketten), keine starren Container.
- Der Datentyp ist immer an den Wert gebunden, niemals an die Variable selbst.
- Entscheidend: Die Zuweisung erfolgt dynamisch zur Laufzeit, nicht im Voraus.

Java (Kompilierzeit)

```java
// Fehler beim Kompilieren!
int x = 5;
x = "Hello"; // Incompatible types
```

Python (Laufzeit)

```python
# Funktioniert einwandfrei!
x = 5
x = "Hello"
```

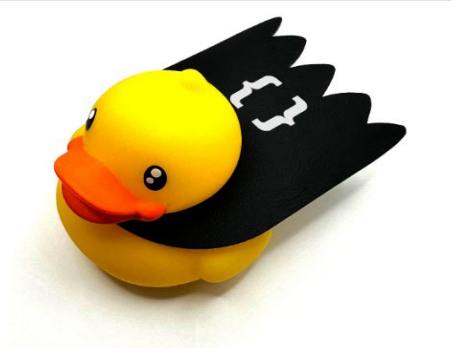
Duck Typing

Verhalten statt Herkunft

Das Prinzip

"If it walks like a duck and quacks like a duck, it's a duck."

Der Fokus liegt auf dem Verhalten eines Objekts, nicht auf seiner Klasse oder Herkunft.



Verzicht auf Interfaces

- Java erfordert explizite Interfaces (z.B. `implements Quackable`).
- Python fragt nicht nach dem Typ, sondern führt die Methode einfach aus.
- Ermöglicht hohe Flexibilität und weniger "Boilerplate"-Code.

Duck Typing

Verhalten statt Herkunft (Code-Vergleich)

Python (Dynamisch)

```python
class Duck:
    def quack(self):
        print("Quack!")

class Person:
    def quack(self):
        print("I'm a Duck! (Quack)")

# Python fragt nicht nach dem Typ, sondern nach der Methode:
def make_it_quack(obj):
    obj.quack()

make_it_quack(Duck()) #Ausgabe: Quack!
make_it_quack(Person()) #Ausgabe: I'm a Duck! (Quack)
```

Java (Statisch mit Interfaces)

```java
// 1. Interface definieren (Zwingend erforderlich in Java)
interface Quackable {
    void quack();
}
// 2. Klassen müssen das Interface explizit implementieren
class Duck implements Quackable {
    @Override
    public void quack() {
        System.out.println("Quack!");
    }
}
class Person implements Quackable {
    @Override
    public void quack() {
        System.out.println("I'm a Duck! (Quack)");
    }
}
public class Main {
    // Die Methode verlangt den Interface-Typ, nicht einfach "obj"
    public static void makeItQuack(Quackable obj) {
        obj.quack();
    }
    public static void main(String[] args) {
        makeItQuack(new Duck()); //Ausgabe: Quack!
        makeItQuack(new Person()); //Ausgabe: I'm a Duck!
    (Quack)
    }
}
```

List Comprehensions

Kompakt, Lesbar & Schnell

Elegante Syntax

Ersetzt traditionelle, mehrzeilige for-Schleifen und if-Abfragen durch einen **eleganten Einzeiler**.

Liest sich intuitiv – fast wie ein **natürlicher englischer Satz**.

Performance & Vergleich

Schneller: Höhere Ausführungsgeschwindigkeit, da die Schleife intern auf C-Ebene optimiert ausgeführt wird.

Das Java-Äquivalent: Java Streams bieten ähnliche Funktionen, sind aber **syntaktisch deutlich komplexer** mit mehr Boilerplate.

List Comprehensions

Kompakt, Lesbar & Schnell (Beispiel)

Python

```python
arr = [1, 2, 3, 4, 5]
# Operation | Loop | Condition

result = [n * n for n in arr if n % 2 == 0]

print(result) # Output: [4, 16]
```

Java (Streams API)

```java
List<Integer> arr = Arrays.asList(1, 2, 3, 4, 5);

// Gleiche Logik, deutlich mehr Struktur-Code
List<Integer> result = arr.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .collect(Collectors.toList());
```

Decorators

Funktionen elegant erweitern

Das Konzept des Wrappings

Ein Decorator nimmt eine bestehende Funktion, fügt neue Features hinzu und gibt sie zurück.

Der ursprüngliche Quellcode der Funktion bleibt unberührt (Open-Closed-Prinzip).

Python vs. Java

Java Annotations

Reine Metadaten (wie `@Override`), die oft erst durch Frameworks aktiv werden.

Python Decorators

Direkter „Syntax Sugar“. Im Hintergrund wird echter Code ausgeführt:

```
beispiel = timer(beispiel)
```

Decorators

Funktionen elegant erweitern (Beispiel)

1. Der Decorator (Die Verpackung)

```python
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        func() # Ausführung der Original-Funktion
        end = time.time()
        print(f"Executed in {end - start:.4f}
    seconds")
    return wrapper
```

2. Die Anwendung

```python
@timer
def beispiel():
    print("Das ist nur ein Beispiel...")
    time.sleep(2)

beispiel()

# Output:
# Das ist nur ein Beispiel...
# Executed in 2.0001 seconds
```

Magic / Dunder Methods

Das Konzept

Was sind "Dunder"?

Steht für **Double Underscore**.

`__init__, __str__, __add__`

Sie ermöglichen es, das **Basis-Verhalten** von Objekten komplett zu überschreiben.

Problem mit Operatoren

Java erlaubt (außer bei Strings) kein Operator Overloading. Um zwei Objekte zu addieren, wird man zu unnatürlichen Methodenaufrufen gezwungen.

```
Point p1 = new Point(1, 2);
Point p2 = new Point(3, 4);

// p1 + p2 ist verboten!
// Boilerplate-Methoden erforderlich!

Point p3 = p1.add(p2);
```

Magic / Dunder Methods

Operator Overloading in Aktion

Die Python-Lösung

Python verknüpft mathematische Operatoren direkt mit den Dunder-Methoden der Klasse.

Der Code wird so lesbar und intuitiv wie eine mathematische Gleichung.

Syntax Sugar

```
a + b a.__add__(b)

a == b a.__eq__(b)

print(a) a.__str__()
```

Magic / Dunder Methods

Operator Overloading in Aktion (Beispiel)

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  # Die Magic Method für den '+' Operator
  def __add__(self, other):
    return Point(self.x + other.x, self.y + other.y)

  # Definiert, wie das Objekt gedruckt wird
  def __str__(self):
    return f"Point({self.x}, {self.y})"

p1 = Point(1, 2)
p2 = Point(3, 4)

# ✅ Python ruft intern p1.__add__(p2) auf!
p3 = p1 + p2
print(p3) # Output: Point(4, 6)
```

Übungen

List Comprehensions

Compact and powerful syntax for creating and transforming sequences.

Decorators

Modifying function behavior dynamically with the elegant @ wrapper.

Magic / Dunder Methods

Unlocking Python's internal operator overloading with __double__ underscores.

Hausaufgabe