Modularisierung und Pakete in Python

Einführung in die Modularisierung

Modularisierung bedeutet, ein Programm in mehrere unabhängige Bausteine (Module) zu unterteilen. Jedes Modul enthält zusammengehörige Funktionen, Variablen oder Klassen.

Vorteile:

- Bessere Struktur und Übersicht
- Einfacheres Testen und Warten
- Wiederverwendbarkeit
- Vermeidung von doppeltem Code

Eigene Module erstellen und nutzen

Ein Modul ist einfach eine .py-Datei mit Funktionen oder Variablen.

Beispiel: mathe_tools.py

```
def quadrat(x):
    return x * x
def addiere(a, b):
    return a + b
```

Damit ein selbst erstelltes Modul importiert werden kann,
muss es sich im gleichen Verzeichnis wie das ausführende Skript befinden
oder in einem Verzeichnis liegen, das im sys.path enthalten ist.

In anderem File importieren:

```
import mathe_tools
print(mathe_tools.addiere(3, 5))
print(mathe_tools.quadrat(4))
```

Mehr dazu in den Codebeispielen

Verschiedene Importarten

Standardimport:

```
import mathe_tools
```

Import mit Alias:

```
import mathe_tools as mt
print(mt.quadrat(3))
```

Mehr dazu in den Codebeispielen

Nur bestimmte Funktionen importieren:

```
from mathe_tools import quadrat
print(quadrat(5))
```

Alle Inhalte importieren (nicht empfohlen):

```
from mathe_tools import *                # Hinweis: *-Import ist fehleranfällig bei Namenskonflikten.
```

Globale und lokale Variablen

Globale Variable: außerhalb aller Funktionen definiert – überall sichtbar
Lokale Variable: innerhalb einer Funktion definiert – nur dort sichtbar

```
x = 10  # global
def zeige():
    y = 5  # lokal
    print("x:", x)  # Gibt den globalen Wert von x aus
    print("y:", y)  # 5
print(x) #10
print(y) # Name error
# Verwendung von global:
z = 0
def erhoehen():
    global z
    z += 1
erhoehen()
print(z) #1
```

Mehr dazu in den Codebeispielen

Lokale Variablen sind nur innerhalb der Funktion sichtbar.

Globale Variablen sind im gesamten Modul verfügbar – können aber innerhalb einer Funktion nur mit dem Schlüsselwort global verändert werden.

Namensräume verstehen (LEGB-Regel)

Ein Namensraum (engl. namespace) ist ein Bereich, in dem ein Bezeichner (z. B. Variablenname) gültig ist. Python verwendet Namensräume, um zu unterscheiden, woher ein Name kommt und ob er zugreifbar ist.

Wenn Python einen Namen (z. B. x) auflöst, durchsucht es folgende Namensräume in dieser Reihenfolge:

Ebene	Bedeutung	Beispiel
Local	Innere Funktionsebene	Variable innerhalb einer Funktion
Enclosing	Umfassende (verschachtelte) Funktionsebene	Variable in äußeren Funktionen
Global	Modul-Ebene (außerhalb von Funktionen)	Variable, die oben im Skript steht
Built-in	Python-eigene Funktionen und Konstanten	len(), print(), True, Exception

Namensräume verstehen (LEGB-Regel)

Beispiel – Verschachtelte Namensräume:

```
x = "global"

def außen():
    x = "enclosing"

    def innen():
        x = "local"
        print("x =", x)  # Lokale Variable → Ausgabe: local

    innen()

außen()
print("x global =", x)  # Globale Variable → Ausgabe: global
```

```
zahl = [1, 2, 3]
print(len(zahl))

# Built-in-Funktion len() → Ausgabe: 3

# len ist nicht lokal, nicht global, nicht enclosing
– aber Python kennt len() als eingebaute Funktion,
weil sie Teil des Built-in-Namensraums ist.
```

Eigene Pakete erstellen

Ein Paket ist ein Ordner, der mehrere Python-Module enthält und durch eine Datei __init__.py als Paket gekennzeichnet ist.

Module = einzelne .py-Dateien mit Funktionen/Klassen

Paket = Sammlung dieser Module in einem Ordner

__init__.py = sagt Python: „Das ist ein echtes Paket“ und wird beim Import einmalig ausgeführt – ideal zum Initialisieren von Modulen oder Voreinstellungen.

- Die Datei __init__.py muss vorhanden sein, auch wenn sie leer ist – sonst erkennt Python den Ordner nicht als Paket (außer ab Python 3.3 bei Namespace-Packages).
- In größeren Projekten helfen Pakete, Funktionen themenbezogen zu gruppieren, z. B. datenbank/, tools/, analyse/
- Pakete können auch verschachtelt sein (Pakete innerhalb von Paketen)

Modul = Datei (.py)

Paket = Ordner mit Modulen + __init__.py

importlib – Module dynamisch laden

Das Modul importlib ist Teil der Python-Standardbibliothek und erlaubt dir, Module zur Laufzeit (dynamisch) zu importieren – also z. B. basierend auf Benutzereingaben oder Konfigurationsdateien.

Ideal, wenn du den Namen des Moduls erst zur Laufzeit kennst (z.B. durch Benutzereingaben)

```
import importlib

modul_name = "math"

modul = importlib.import_module(modul_name) # Python lädt hier zur Laufzeit das math-Modul und gibt dir Zugriff darauf – als Objekt.

print(modul.sqrt(25))  # Ausgabe: 5.0
```

Dynamisches Laden ist hilfreich in Situationen wie:

- Plugin-Systeme: Du lädst nur, was der Benutzer aktiviert.
- Werkzeuge für Konfiguration: Benutzer wählt, welche Funktionen geladen werden sollen.
- Testframeworks oder Skriptloader: Automatisch Module durchgehen und ausführen.
- So wird nur geladen, was wirklich benötigt wird – das spart Speicher und beschleunigt den Start des Programms.

Zusammenfassung

Thema	Beschreibung
Modul	Eine .py-Datei mit Funktionen, Klassen oder Konstanten
Importvarianten	import, from ... import, Alias, *
Namensräume	LEGB-Regel: Local, Enclosing, Global, Built-in
Global vs. Lokal	Gültigkeit von Variablen
Paket	Ordner mit __init__.py und mehreren Modulen
importlib	Dynamisches Laden von Modulen

Modularisierung und Pakete in Python

Mehr erkunden:

https://docs.python.org/3/tutorial/modules.html

https://docs.python.org/3/library/importlib.html

https://www.w3schools.com/python/python_modules.asp