Funktionale Programmierung in Python

Einführung in die funktionale Programmierung

In der funktionalen Programmierung schreibt man Funktionen, die nur Eingaben verarbeiten und einen Rückgabewert liefern – ohne etwas außerhalb zu verändern.

Sie machen keine Nebenaktionen wie print(), Dateischreiben, verändern keine Datenstrukturen außerhalb, oder globale Variablen ändern etc. Solche „reinen Funktionen“ sind einfacher zu testen, zu verstehen und wiederzuverwenden.

Funktionale Programmierung basiert auf dem Prinzip reiner Funktionen (pure functions): Sie liefern ein Ergebnis, das nur von den Eingaben abhängt – ohne dabei irgendetwas außerhalb der Funktion zu verändern.

Pure Funktion

```
def add(a, b):
    return a + b
# → hängt nur von a und b ab, macht keine Nebensache
```

Funktionen mit Nebenaktionen (Side Effects)

```
def add_and_log(a, b):
    result = a + b
    print(f"Ergebnis: {result}")  # → erzeugt eine Nebenaktion
    return result
```

Lambda-Funktionen (Anonyme Funktionen)

In Python kann man kurze Funktionen auch ohne def und ohne Namen schreiben – mit dem Schlüsselwort lambda.

Diese sogenannten anonymen Funktionen sind ideal, wenn du eine Funktion nur einmalig und direkt brauchst, z. B. beim Sortieren, Filtern oder Umwandeln von Werten.

Syntax

lambda parameter: Ausdruck

```
lambda x: x * 2  # lambda Funktion
```

```
def verdoppeln(x):  # gleiche Funktion als def-Version
    return x * 2
```

Anwendung – direkt beim Aufruf verwenden

```
ergebnis = (lambda x: x * 2)(4)   # Die Zahl 4 wird als Argument an die Lambda-Funktion übergeben.
                                  # x erhält den Wert 4, und das Ergebnis 4 * 2 wird zurückgegeben.
print(ergebnis)                   # Ausgabe: 8
```

lambda-Funktionen sind wie Mini-Funktionen für eine Zeile – kompakt und praktisch für einfache Aufgaben, aber bei komplexerer Logik ist def oft klarer.

map() – eine Funktion auf alle Listenelemente anwenden

Mit map() kannst du eine Funktion auf alle Elemente einer Liste anwenden, ohne eine Schleife schreiben zu müssen. Das Ergebnis ist ein "map-Objekt" (eine Art Liste), das du z. B. mit list() sichtbar machen kannst.

Syntax : map(Funktion, Liste)

```
def verdoppeln(x):
    return x * 2
zahlen = [1, 2, 3, 4]

ergebnis = map(verdoppeln, zahlen)
print(list(ergebnis))  # Ausgabe: [2, 4, 6, 8]
```

Was passiert hier?

- verdoppeln(x) wird für jedes Element in zahlen aufgerufen.
- **map**() gibt ein Objekt zurück, das alle Ergebnisse enthält.
- Mit list(...) oder einer for-schleife kannst du das Ergebnis aus dem map-Objekt lesen.

Mit **map**() kannst du alle Elemente einer Liste in einem Schritt verändern, ohne Schleife.

Typische Anwendung: Werte umwandeln, Strings formatieren, Daten vorbereiten

filter() – nur bestimmte Werte behalten

Mit filter() kannst du aus einer Liste nur die Elemente behalten, die eine bestimmte Bedingung erfüllen. Du gibst eine Funktion an, die für jedes Element **True** oder **False** zurückgibt.

Syntax : filter (Funktion, Liste)

Beispiel – nur gerade Zahlen behalten:

```
def ist_gerade(x):
    return x % 2 == 0
zahlen = [1, 2, 3, 4, 5, 6]
gefiltert = filter(ist_gerade, zahlen)

print(list(gefiltert))  # Ausgabe: [2, 4, 6]
```

Was passiert hier?

- ist_gerade(x) wird auf jedes Element angewendet.
- Nur die Werte, für die **True** zurückkommt, bleiben erhalten.
- Das Ergebnis ist ein Filter-Objekt, das du z. B. mit list() oder einer for-Schleife auslesen kannst.

filter() hilft dir, **nur die passenden Werte** aus einer Liste auszuwählen – ganz ohne Schleife und if.

Typische Anwendung: Nur gerade/ungerade Zahlen aus einer Liste behalten. Nur positive Werte auswählen. Zeichen aus einer Liste filtern, z. B. nur Buchstaben behalten. Aus Strings nur Wörter mit bestimmter Länge wählen. Nur gültige/erlaubte Eingaben weiterverarbeiten

reduce() – alles auf einen Wert reduzieren

Mit reduce() kannst du eine Liste auf einen einzigen Wert reduzieren, indem du wiederholt zwei Elemente kombinierst. Das ist z. B. nützlich, um eine Summe, ein Produkt oder eine Verknüpfung zu berechnen. reduce() verarbeitet alle Elemente nacheinander und reduziert sie auf einen einzigen Gesamtwert.

reduce() ist nicht automatisch verfügbar – du musst es aus dem Modul **functools** importieren.

Syntax : reduce (Funktion, Liste)

Beispiel – Produkt aller Zahlen berechnen:

```
from functools import reduce

zahlen = [1, 2, 3, 4]


def multiplizieren(x, y):
    return x * y

ergebnis = reduce(multiplizieren, zahlen)

print(ergebnis)    # Ausgabe: 24
```

Was passiert hier?

- reduce() nimmt zuerst 1 und 2, berechnet 1 * 2 = 2
- Dann 2 * 3 = 6, dann 6 * 4 = 24
- Am Ende bleibt **ein einziger Wert** übrig

```
#  Gleiche Berechnung mit lambda – in einer einzigen Codezeile statt in vier
reduce(lambda x, y: x * y, zahlen)    # Ausgabe: 24
```

Typische Anwendung : Gesamtsumme berechnen. Produkt aller Werte berechnen. Längstes Wort in einer Liste finden. Zeichenfolgen zu einem String verbinden. Werte aggregieren – z. B. aus Dictionaries, Objekten oder verschachtelten Strukturen.

Rekursive Funktionen (Rekursion)

Eine rekursive Funktion ist eine Funktion, die sich selbst aufruft, um ein Problem in kleinere Teilprobleme zu zerlegen. Das funktioniert ähnlich wie beim Denken in Schritten: "Um das große Problem zu lösen, löse ich zuerst eine kleinere Version davon."

Beispiel – Fakultät berechnen (n!):

```
def fakultaet(n):
    if n == 0:
        return 1   # Abbruchbedingung
    return n * fakultaet(n - 1)  # Selber aufrufen

print(fakultaet(5))  # Ausgabe: 120
```

Was passiert hier?

- fakultaet(5) ruft fakultaet(4) auf-- (5* Ergebnisse von fakultaet(4))
- fakultaet(4) ruft fakultaet(3) auf …(5*4 *Ergebnis von fakultaet(3))
- bis fakultaet(0) den Wert 1 zurückgibt (5*4*3*2*1 = 120)

Jede rekursive Funktion braucht:

- Jede rekursive Funktion braucht: eine Abbruchbedingung – sonst ruft sie sich unendlich oft selbst auf.
- einen Schritt, der das Problem verkleinert

Typische Anwendung : Mathematische Probleme (Fakultät, Fibonacci, Potenzieren). Durchlaufen von Ordnerstrukturen. Arbeiten mit verschachtelten Daten (z. B. Baumstrukturen, HTML)

List Comprehension

Siehe Zusatzfolie

Generator Expressions: Speicherfreundliche Alternative zur List Comprehension

Generator Expressions sehen fast aus wie List Comprehensions – aber sie geben keine Liste, sondern einen Generator zurück. Ein Generator berechnet die Werte erst bei Bedarf (lazy evaluation) – das spart Speicher.

List Comprehension

```
quadrate = [ x * x for x in range(5) ]
print(quadrate)  # Ausgabe: [0, 1, 4, 9, 16]
```

Generator Expression – fast gleich, aber mit ()

```
quadrate_gen = ( x * x for x in range(5) )
print(quadrate_gen)  # Ausgabe: <generator object ...>

for wert in quadrate_gen:
    print(wert)          # 0,1,4,9,16
```

Wenn du denselben Generator noch einmal verwendest:

```
for wert in quadrate_gen:
    print(wert)       # Ausgabe: nichts! – der Generator ist  "verbraucht"
```

Ein Generator liefert seine Werte nur einmal – danach ist er leer.

Wann sind Generator Expressions sinnvoll?

- Wenn du mit sehr großen Datenmengen arbeitest
- Wenn du nicht alle Ergebnisse auf einmal brauchst
- In Schleifen, wo du Stück für Stück verarbeitest
- Datenströme, die nur durchlaufen – nicht gespeichert – werden müssen

Zusammenfassung

Konzept	Beschreibung
lambda	Anonyme Funktion ohne def
map()	Wendet Funktion auf alle Elemente an
filter()	Filtert Elemente basierend auf Funktion
reduce()	Kombiniert alle Werte zu einem Ergebnis
Rekursion	Funktion ruft sich selbst auf
List Comprehension	Kompakte Syntax zum Erstellen von Listen
Generator Expression	Speicherfreundliche Alternative zu List Comprehension

Funktionale Programmierung in Python

Mehr erkunden:

lambda – anonyme Funktionen
https://www.w3schools.com/python/python_lambda.asp
https://docs.python.org/3/reference/expressions.html#lambda

map() – Funktion anwenden
https://docs.python.org/3/library/functions.html#map
https://www.geeksforgeeks.org/python/python-map-function/

filter() – Werte herausfiltern
https://docs.python.org/3/library/functions.html#filter
https://www.geeksforgeeks.org/python/filter-in-python/

reduce() – Liste zu einem Wert reduzieren
https://docs.python.org/3/library/functools.html#functools.reduce
https://www.geeksforgeeks.org/python/reduce-in-python/

Mehr erkunden:

Rekursion – sich selbst aufrufende Funktionen
https://docs.python.org/3/tutorial/controlflow.html#defining-functions
https://www.w3schools.com/python/gloss_python_function_recursion.asp

List Comprehension – kompakte Listenerstellung
https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions
https://www.w3schools.com/python/python_lists_comprehension.asp

Generator Expressions – speicherschonend
https://docs.python.org/3/howto/functional.html#generators
https://www.geeksforgeeks.org/python/generator-expressions/