Grundlagen der Programmierung

Objektreferenzen

Funktionale Programmierung: Paradigma, Funktionen höherer Ordnung, anonyme Funktionen

Objekte und Referenzen

- **Python:** Identifier (Name des Objekts) ist **Referenz** (Verweis) auf das Objekt.
- Wertzuweisung übergibt die Referenz (*keine Kopie der Werte*)
- `o1 is o2` gibt zurück, ob `o1` und `o2` identisch (Referenzen auf dasselbe Objekt) sind
- *Beispiel Liste:*

```
x = [1,2]
y = x            # Übergabe der Referenz
y is x           # True
y.append(3)
print(x)         # Ausgabe: [1,2,3]
```

Recap: Änderungen an Objekten

1. Mit Wertzuweisung zweite Referenz auf ein Objekt erzeugen

```
x = [1,2]          x = [1,2]
y = x              y = x
```

2. Objekt über zweite Referenz ändern

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

- Änderung ist auch über die erste Referenz sichtbar

```
print(x)   # [1,2,3]   # [1,2]
```

- Wird aber ein neuer Wert mit dem `=`-Operator zugewiesen, dann entsteht ein neues Objekt an einem anderen Speicherplatz
→ Änderung über erste Referenz *nicht* sichtbar

Identität und Gleichheit

```
x = [1,2]
y = x                # Übergabe der Referenz
y is x               # True (Identität)

y = [1]              # neues Objekt
y is x               # False
y.append(2)
y is x               # False (verschiedene Objekte)
y == x               # True (Gleichheit)
```

Klasse Fahrzeug

```
class Vehicle:
    def __init__(self,bez,ge):
        self.type = bez
        self.speed = ge

    def accelerate(self,ge):
        self.speed += ge
```

Identität und Gleichheit

```
x = Vehicle("Lok", 40)
y = x
y is x                  # True (Identität)

y = Vehicle("Lok", 60) # neues Objekt
y is x                  # False
x.accelerate(20)
y is x                  # False
y == x                  # sollte True ergeben (?)
                        # liefert aber False,
                        # da nicht speed und type
                        # verglichen werden
```

Identität und Gleichheit

- *ohne Definition von* `__str__()`

```
x = Vehicle("Lok", 40)
y = x
print(x)  # Vehicle object at 0x0043D70
print(y)  # Vehicle object at 0x0043D70

y = Vehicle("Lok", 60)
print(x)  # Vehicle object at 0x0043D70
print(y)  # Vehicle object at 0x0043DF0
```

Operatormethoden

- `__eq__(self,other)`
  wird aufgerufen, wenn Objekte **`self`** und **`other`** mit **`==`** verglichen werden
- `__gt__(self,other)`
  wird aufgerufen, wenn Objekte **`self`** und **`other`** mit **`>`** vergleichen werden
- `__lt__(self,other)` **`<`**
- `__ge__(self,other)` **`>=`**
- `__le__(self,other)` **`<=`**
- `__ne__(self,other)` **`!=`**

Operatormethoden (2)

- `__add__(self,other)`
  wird aufgerufen, wenn Objekte **`self`** und **`other`** mit **`+`** addiert werden
- `__sub__(self,other)` **`-`**
- `__mul__(self,other)` **`*`**
- `__truediv__(self,other)` **`/`**
- `__floordiv__(self,other)` **`//`**
- `__mod__(self,other)` **`%`**
- `__pow__(self,other)` **`**`**

Klasse Fahrzeug

```
class Vehicle:
    def __init__(self,bez,ge):
        self.type = bez
        self.speed = ge

    def __eq__(self,other):
        return (self.speed == other.speed) and
                            (self.type == other.type)

    def accelerate(self,ge):
        self.speed += ge
```

Kopieren von Objekten

- Erzeugen eines neuen Objekts mit dem Konstruktor mit den Werten des Originals

```
x = Vehicle("Lok", 40)
x.accelerate(20)
y = Vehicle(x.type, x.speed)
y is x                    # False
y == x                    # True (nach Def. __eq__)
```

- Verwenden von **`deepcopy()`** aus dem Modul **`copy`**

```
import copy
...
z = copy.deepcopy(x)
```

Funktionale Programmierung

Funktionale Programmierung – Idee

- Programme führen *mathematische* Berechnungen aus
- Berechnungen in der Mathematik:
  - Anwendung von Operationen und Funktionen auf Werte
  - Weiterrechnen mit so erhaltenen Ergebnissen
  - *Beispiel:* $\log_2(\sin(3x+1))$
- *Fundamentale Konzepte:*
  - Funktion
  - Operation
  - Werte (Ein-/Ausgabegrößen)
  - Komposition

Komposition:
Verketten von Funktionen
(*Ineinander einsetzen*)

1. $y_1 = 3x$ (mult(3,x))
2. $y_2 = y_1 + 1$ (add(y_1,1))
3. $y_3 = \sin(y_2)$
4. $y_4 = \log_2(y_3)$

Funktion

Eine *k*-stellige **Funktion** $f: M_1 \times M_2 \times \ldots \times M_k \longrightarrow N$
ist eine Relation, die jedem Tupel aus $M_1 \times M_2 \times \ldots \times M_k$
eindeutig ein Element aus *N* zuordnet: $f(x_1, x_2, \ldots, x_k) = y$.

Definitionsbereich: $M_1 \times M_2 \times \ldots \times M_k$
Wertebereich: (Teilmenge von) N

Argumente: $x_1 \in M_1, x_2 \in M_2, \ldots, x_k \in M_k$
Funktionswert: $y \in N$

Operationen

Eine k-stellige **Operation** auf einer Menge M ist eine Funktion

$$\circ : M^k \longrightarrow M$$

Beispiele:

$+ : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}$ (Addition ganzer Zahlen)

$* : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}$ (Multiplikation ganzer Zahlen)

$- : \mathbb{Z} \longrightarrow \mathbb{Z}$ (Entgegengesetztes ganzer Zahlen)

$+ : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$ (Addition reeller Zahlen)

usw.

Verschiedene Definitions- und Wertebereiche bestimmen unterschiedliche Operationen und Funktionen!

Operationen in Programmiersprachen

- Operation eines Datentyps *D* erlaubt oft Argumente verschiedener Datentypen, von denen mindestens eines *D* ist, und Funktionswerte eines weiteren Typs

- `5 / 2          # 2.5` → `/ :  int × int ⟶ float`
- `len(string) # 3` → `length : str → int`
- `L.insert(0,"GdP")`
- …

➢ oft ***mehrsortige*** Operationen

Datentypen

- Mögliche Werte gehören zu einem **Datentyp**.
- Datentypen sind bestimmt durch
  1. eine Menge ***M*** von gültigen Werten
  2. *Operationen* auf diesen Werten: $\circ_1, \circ_2, \ldots, \circ_n$

➢ Tupel **(M, $\circ_1, \circ_2, \ldots, \circ_n$)**
(*mehrsortige Algebra mit Trägermenge M;*
algebraische Struktur, falls alle Operationen auf M)

- *Beispiele:*
  $(\mathbb{Z}, +, *)$, $(\mathbb{R}, +, *)$, ...

Datentypen in Python

- Zahlentypen **int**, **float** (mit +, *, -, /, //, %, ...)
- Boolescher Typ **bool** (mit and, or, not)
- Zeichenkettentyp **str** (mit `len`, +, *, ...)
- Kollektionstyp indizierte Liste (mit ...)
- Tupel (mit ...)
- Menge (mit ...)
- Dictionary (mit ...)
- Datei (mit ...)
- **Klassen**: benutzerdefinierte Typen
  - **Trägermenge:** Menge aller Objekte dieser Klasse
  - **Operationen:** Methoden

Erinnerung: Funktionale Programmierung – Idee

- Programme führen *mathematische* Berechnungen aus
- Berechnungen in der Mathematik:
  - Anwendung von Operationen und Funktionen auf Werte
  - Weiterrechnen mit so erhaltenen Ergebnissen
  - *Beispiel:* $\log_2(\sin(3x+1))$
- *Fundamentale Konzepte:*
  - Funktion
  - Operation
  - Werte (Ein-/Ausgabegrößen)
  - Komposition

Funktionale Programmierung

- Programme: mathematische Berechnungen
- Programme bestehen (nur) aus
  - Funktionsdefinitionen
  - Funktionsaufrufen
- Funktionsaufrufe können weitere Funktionsaufrufe enthalten
  → Realisierung von **Komposition**
  - *Keine Zuweisungen der Form* `x = x + 1`
    → `f(x)` mit $f(x) = x + 1$
    und ggf. Weiterverwendung als Argument einer Funktion
- **Rekursion**
  - *Keine Berechnungen mit Hilfe von Schleifen!*

Selbstaufrufe

- Funktionen können sich selbst aufrufen
  → *z.B.* iterierte Funktionsanwendung: $f(f(f(x)))$
- <u>Beispiel</u>: f verdoppelt eine Zahl

$f(f(f(1))) = f(f(2)) = f(4) = 8$

```
def f(x):
      return 2*x

n = 3
k = 1
for i in range(1,n+1):
      k = f(k)
print(k)
```

Iteration:
Wiederholter Aufruf der Funktion in einer *Schleife*, wobei die Ausgabe des letzten Aufrufs als Eingabe des nächsten Aufrufs verwendet wird.

Rekursion

- alternative Definition mit Selbstaufruf:

$f(0) = 1$

$f(n) = 2\,f(n\text{-}1) \quad \text{für} \;\; n > 0$

```
def f(x):
    if x == 0:
        return 1
    else:
        return 2 * f(x-1)
```

$f(3) = 2\,f(2) \qquad f(3) = 2 \cdot 4 = 8$

$f(2) = 2\,f(1) \qquad f(2) = 2 \cdot 2 = 4$

$f(1) = 2\,f(0) \qquad f(1) = 2 \cdot 1 = 2$

$f(0) = 1$

Rekursion:
Definition der Funktion enthält Selbstaufruf mit anderer Parameterliste. Für einen festen Wert wird das Ergebnis vordefiniert.

Rekursive Definitionen

- Definition durch Rückgriff auf
  - die Definition atomarer Bestandteile
  - die Definition von kleineren Bestandteilen desselben Konzepts
- **Funktionen:**
  Definition von $f(n)$ durch Rückgriff auf
  - $f(a)$ für gewisse Anfangswerte a
  - $f(k)$ für $k < n$
- primitive **boolesche Ausdrücke:**
  - **True**, **False** sind boolesche Ausdrücke
  - Wenn A und B boolesche Ausdrücke sind, dann auch
    A **or** B , A **and** B , **not** A
- analog arithmetische und andere Ausdrücke

Fakultät: Iteration *versus* Rekursion

- **Iteration**
  - mathematische Definition:
    fac(*n*) = 1·2· ... ·*n*
  - Realisierung mit Schleife
    (*Iteration = Wiederholung*):

```
def fac(n):
    result = 1
    for i in range(1,n+1):
        result = result * i
    return result
```

fac(0) *korrekt berechnet?*

- **Rekursion**
  - mathematische Definition:
    fac(0) = 1
    fac(*n*) = *n* · *fac*(*n*-1) , *n*>0
  - Realisierung:

```
def fac(n):
    if n == 0:
        return 1
    return n * fac(n-1)
```

Warum braucht man kein else?

Funktionale Programmierung in Python

Name: Intervallsumme
Eingabe: zwei ganze Zahlen a und b
Ausgabe: Summe aller ganzen Zahlen i mit $a \leq i \leq b$

```
def sumInts(a, b):
    if a > b:
        return 0
    else:
        return a + sumInts(a+1, b)
```

Funktionale Realisierung eines Algorithmus (mit Rekursion)

Kubische Intervallsumme

Name: Kubiksumme
Eingabe: zwei ganze Zahlen a und b
Ausgabe: Summe aller ganzen Zahlen i^3 mit $a \leq i \leq b$

```
def cube(x):
    return x**3

def sumCubes(a, b):
    if a > b:
        return 0
    else:
        return cube(a) + sumCubes(a+1, b)
```

Funktionales Programm

```
print(sumInts(0,cube(2)))    # 0+1+2+3+4+5+6+7+8
                             # also Ausgabe von 36

L = [1,2,3]                  # Objekt (Eingabewert)
print(sumCubes(1,len(L)))    # 1³+2³+3³ = 36
```

Funktionen höherer Ordnung – Motivation

- Verallgemeinerung auf Intervallsumme von Werten beliebiger Funktionen

- **Name:** Funktionswertsumme
  Eingabe: zwei ganze Zahlen *a* und *b*, Funktion *f*
  Ausgabe: Summe der $f(i)$ mit $a \leq i \leq b$

- *Idee:* Übergabe der Funktion *f* als Argument

➢ **Funktion höherer Ordnung:** Funktion, die eine Funktion als Argument oder Funktionswert erlaubt

Höhere Funktion in Python – Beispiel

```
def sum(f, a, b):
      if a > b:
            return 0
      else:
            return f(a) + sum(f, a+1, b)

def sumCubes2(a, b): return sum(cube, a, b)

def id(x): return x

def sumInts2(a, b): return sum(id, a, b)
```

Funktionen höherer Ordnung (Funktionale)

Rekursive Definition: **Funktionen ...**

- **nullter Ordnung** keine Argumente (Konstanten/Daten)

- ***n*-ter Ordnung** mindestens ein Argument ist Funktion (n-1)-ter Ordnung (n > 0), aber kein Argument höherer Ordnung

Beispiele: cube(x) ist Funktion erster Ordnung

sum(f, a, b) ist Funktion zweiter Ordnung (wenn f erster Ordnung ist)

Beispiele aus der Mathematik

- Differenzierungsfunktion $\frac{df}{dx} = \frac{d}{dx}(f)$
  - Argument: einstellige, reelle, differenzierbare Funktion f
  - Funktionswert: erste Ableitung f'
- Unbestimmte Integralfunktion
  - Argument: einstellige, reelle, integrierbare Funktion f
  - Funktionswert: Stammfunktion von f
- Bestimmte Integralfunktion
  - Argumente: integrierbare Fkt. f, reelle Zahlen a, b
  - Funktionswert: reelle Zahl

Definitions- und Wertebereiche

- *Beispiele:*
  - Differenzierungsfunktion:
    DB: Menge aller differenzierbaren reellen Funktionen
    WB: Menge aller reellen Funktionen

$$\frac{d}{dx} : (\mathbb{R} \to \mathbb{R})_{\mathrm{d}} \to (\mathbb{R} \to \mathbb{R})$$

  - sum: $(\mathbb{Z} \to \mathbb{Z}) \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$: sum(f, a, b) = y

$$\text{mit } y = \sum_{i=a}^{b} f(i)$$

Anonyme Funktionen – Motivation

```
def sumCubes2(a, b): return sum(cube, a, b)

def sumInts2(a, b): return sum(id, a, b)
```

- Voraussetzung: **`id`** und **`cube`** sind definiert
- Verallgemeinerung auf andere Funktionen?!
- *Idee:* Einsetzen der Zuordnungsvorschrift für f
  - *statt* **`id`** *einfach* $x \mapsto x$ *oder in Python:* `x:x`
  - *statt* **`cube`** *einfach* $x \mapsto x^3$ *oder in Python:* `x:x**3`
- Kennzeichnung des Ausdrucks `x:y` als Zuordnung durch `lambda`

Anonyme Funktionen in Python

```
def sumIntsL(a, b):
      return sum(lambda x:x, a, b)

def sumCubesL(a, b):
      return sum(lambda x:x**3, a, b)

def sumSquares(a, b):
      return sum(lambda x:x**2, a, b)
```

- lambda-Ausdrücke können mehrstellige Funktionen zurückgeben!

```
lambda x,y : x+y
lambda a,b,c: a+b**c
```

`lambda` *versus* `def`

- `lambda` und `def` definieren beide eine Funktion
  - `def` mit Funktionsbezeichner (Eintrag in die Symboltabelle)
  - `lambda` ohne Funktionsbezeichner

- `def` ist Anweisung
  → darf nicht in Ausdrücken auftauchen

- `lambda x:y` ist Ausdruck (mit einer Funktion als Wert)
  → darf in Ausdrücken (z.B. als Parameter) auftauchen

- Zuweisung eines lambda-Ausdrucks an Funktionsnamen ist möglich:

`mult = lambda x,y: x*y` *entspricht*

```
def mult(x,y):
    return x*y
```