Informatik I: Einführung in die Programmierung

11. Veränderliche Daten

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Peter Thiemann

29. November 2023

1 Veränderliche Daten

Veränderliche Daten

- Bisher haben wir Objekte als unveränderlich betrachtet.
- Manche Objekte können allerdings während des Programmablaufs verändert werden.
- Veränderliche Objekte:
  - Listen (Typ `list`)
  - Instanzen von Datenklassen

Weitere Operationen auf Listen

Zuweisung an Listenelement


```
>>> xs, ys = ["prolific", "proselyte"], ["prolific", "proselyte"]
>>> zs = xs
>>> print(" ".join(xs))
prolific proselyte
>>> xs[1] = "procrastinator"
>>> print(" ".join(xs)); print(" ".join(zs))
prolific procrastinator
prolific procrastinator
>>> print(" ".join(ys))
prolific proselyte
```

- Zuweisung an `xs[i]` ändert Position `i` in der Liste `xs`. Dabei muss `-len(xs) <= i < len(xs)` ein legaler Index sein.
- `xs` und `zs` sind **Aliase** (sie verweisen auf das selbe Listenobjekt); daher schlägt jede Änderung an `xs` auf `zs` durch und umgekehrt.
- `ys` ist ein separates Listenobjekt und ist von Änderungen an `xs` nicht betroffen.

Weitere Operationen auf Listen

Löschen von Listenelement


```
>>> print(" ".join(xs))
prolific procrastinator
>>> del zs[0]
>>> print(" ".join(xs)); print(" ".join(zs))
procrastinator
procrastinator
>>> print(" ".join(ys))
prolific proselyte
```

- `del zs[i]` entfernt Position `i` aus der Liste `zs`. Dabei muss `i` ein legaler Index sein.

Weitere Operationen auf Listen

Anhängen von Listenelementen


```
>>> print(" ".join(xs))
procrastinator
>>> xs.append("predator") # add a single element at the end
>>> print(" ".join(xs))
procrastinator predator
>>> xs.extend(ys) # add all elements at the end
>>> print(" ".join(xs))
procrastinator predator prolific proselyte
```

- `append()` und `insert()` sind **Methoden** der Klasse `list` und können daher mit allen Listenobjekten verwendet werden.
- Weitere Methoden: `insert()`, `remove()`, `pop()`, `reverse()`, `sort()`, `clear()` usw.
- Siehe Dokumentation.

Veränderliche Instanzen von Datenklassen

- Erinnerung: Binärbaum

```
>>> from dataclasses import dataclass
>>> from typing import Optional
>>> @dataclass
... class INode:
...   mark  : int
...   left  : Optional['INode'] = None
...   right : Optional['INode'] = None
...
```

Veränderliche Instanzen von Datenklassen


```
>>> n1, n2 = INode(42), INode(42)
>>> print(n1)
INode(mark=42, left=None, right=None)
>>> n1.mark = 0
>>> print(n1); print(n2)
INode(mark=0, left=None, right=None)
INode(mark=42, left=None, right=None)
```

- Zuweisung an Attribute
- `n1` und `n2` sind unterschiedliche Instanzen. Daher wirkt die Zuweisung `n1.mark = 0` nur auf `n1`.

Veranderliche Instanzen von Datenklassen


```
>>> n2.left = n1
>>> print(n1); print(n2)
INode(mark=0, left=None, right=None)
INode(mark=42, left=INode(mark=0, left=None, right=None), right=None)
```

```
>>> n2.right = n1
>>> print(n1); print(n2)
INode(mark=0, left=None, right=None)
INode(mark=42, left=INode(mark=0, left=None, right=None), right=INode(mark=0
```

Erinnerung: Unveränderliche Daten

Tupel


```
>>> xt = (1, 2, 3)
>>> xt[1] = -2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Erinnerung: Unveränderliche Daten

frozen Datenklassen

```
>>> @dataclass(frozen=True)
... class FNode:
...   mark  : int
...   left  : Optional['FNode'] = None
...   right : Optional['FNode'] = None
...
>>> fn1 = FNode(42)
>>> print(fn1.mark)
42
>>> fn1.mark = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 4, in __setattr__
```

2 Parameter mit Standardwert und Namen

Einschub: Parameter mit Standardwert

Die letzten Parameter einer Funktion können einen **Standardwert** haben (im Beispiel der Parameter `y`). Die entsprechenden Argumente dürfen beim Funktionsaufruf weggelassen werden; in dem Fall erhalten die Parameter den Standardwert.

```
>>> def f (x: int, y: int = 0) -> int:
...   return x - y
...
>>> assert f (3, 5) == -2
>>> assert f (3) == 3
```

Einschub: benannte Parameter

Argumente können auch über den Namen des Parameters übergeben werden, die Reihenfolge der benannten Argumente spielt dann keine Rolle.

```
>>> def f (x: int, y: int) -> int:
...   return x - y
...
>>> assert f (3, 5) == -2
>>> assert f (x=3, y=5) == -2
>>> assert f (y=3, x=5) == 2
```

Regel für den Funktionsaufruf

- Erst die unbenannten Argumente (positional arguments)
- dann die benannten Argumente
- Fehlende Argumente müssen einen Standardwert haben.