Coding Conventions

by

Dr. Günter Kolousek

Richtlinien

Allgemein

- Maximale Zeilenlänge von 79!
- Je Zeile maximal eine einfache Anweisung.
- Identifier (Bezeichner) müssen aussagekräftig sein.
  - Keine Sonderzeichen oder Umlaute!
  - In **Englisch**!!
- Variablen- und Funktionsnamen: beginnen klein
  - Varianten:
    - *snake case* - Darstellung: z.B. `get_area`
    - *camel case* - Darstellung: z.B. `getArea`
    - *pascal case* - Darstellung: z.B. `GetArea`
  - `.NET` verwendet *pascal case*…
- Klassen- und Typennamen in ”MixedCase”: beginnen groß!
- 2 Leerzeilen zwischen Funktionsdefintionen bzw. Klassen

Leerzeichen

- Links und rechts je binären Operator bzw. um ”=” genau ein Leerzeichen, z.B.:

```
c = a ** 2 + b ** 2
c = c / 100
```

- Nach einem unären Operator kein Leerzeichen, davor jedoch schon, z.B.:

```
c = -c
```

- Keine Leerzeichen unmittelbar innerhalb von runden, eckigen oder geschwungenen Klammern, z.B.:

```
c = ((a + b) / 2) - 3) * 4
d = e[0]
e = {1, 2, 3}
```

Leerzeichen – 2

- Keine Leerzeichen
  - *vor*
    - ”(” von Funktionen
    - ”,” oder ”;” oder ”.”
    - um = bei Initialisierungen, z.B.: `int a=1;`
  - *nach* ”.”
  - am Zeilenende
- Ein Leerzeichen *nach*: ”,” oder ”;”
- Einrückung immer 4 **Leerzeichen** (Editor konfigurieren)!

Lange Zeilen

- Umgebrochene Zeilen nachfolgend um 2 einrücken, z.B.:

```
if a == 3 or b == 4 \
  c == 4:
    print("...")
else:
```

- ```
  int get_really_long_name_for_sum(
    int really_very_very_long_name) {
      return really_very_very_long_name;
  }
  ```

- bei Zuweisungen auch möglich:

```
arith_mean = sum(all_important_numbers) /
             len(all_important_numbers)
```
```

Allgemein – 2

- angestrebte Reihenfolge der Kategorien innerhalb einer Klassendelaration
  1. Typedefs, Using, Enums
  2. Konstanten
  3. Konstruktoren
  4. Destruktor … C++, C#
  5. Methoden
  6. Klassenvariable (statisch)
  7. Instanzvariable
  8. `main` - Methode … Java, C#

Kommentare

- Alle Kommentare in **Englisch**
- Dateikopf am Anfang der Datei mit folgenden Informationen, z.B.:
  - author, matnr und file sind **zwingend**!
  - alles andere nur, wenn gefordert.

```
/*
author: Mustermann Maxi
matnr:  i09026
file:   quadrat.py
desc:   This module provides an impl of
        elliptic curve cryptography...
date:   2010-04-26
class:  2A
catnr:  26
*/
```

Kommentare – 2

- Blockkommentare mit einer Leerzeile davor
- Einzelzeilenkommentare direkt davor und eingerückt wie folgende Codezeile

```
from functools import reduce
from operator import __mul__

# geometric mean of the numbers in nums
geom_avg = reduce(__mul__, nums) **
           (1 / len(nums))
```

- Kurze Kommentare am Ende der Codezeile nach 2 Leerzeichen:

```
agenda = {nodes.pop()}  # arbitrary node
```

Python

- Importanweisungen am Anfang der Datei
  - jeweils in einer eigenen Zeile.
- Kein Leerzeichen vor, jedoch ein Leerzeichen nach `:`
  - z.B.: `if msg_available(): receive_msg()`
  - Ausnahme bei Slices: `numbers[1:-1]`
- Keine Leerzeichen unmittelbar innerhalb von geschweiften Klammern, z.B.:

```
c = {"mean": seq[(a + b) / 2]}
```

- Keine Leerzeichen um "=", wenn Keyword-Argument oder Defaultwert, z.B.:

```
def sum(a, b=0, c=0):
    return a + b + c

print(sum(4, c=2))
```

Python – 2

- Modulnamen zur Gänze in Kleinbuchstaben (”_” ist erlaubt).
- Docstrings verwenden
  1. Kurzer prägnanter Satz ohne Punkt
  2. Kurzbeschreibung (wenn sinnvoll) in ganzen Sätzen.
     Vorangehende Leerzeile.

  z.B.:

```
"""Return the sum of given numbers

All parameters will be converted to
numbers first.
"""
```

Java

- analog C# bzw. C++!
- Paketnamen bzw. Namespaces zur Gänze in Kleinbuchstaben (”_” ist erlaubt).
- Geschwungene Klammern am Ende der Zeile, z.B.:

```
if (elem >= max) {
    max = elem;
    ...
}
```

- Keine Leerzeichen bei spitzen Klammern!
  - `List<Element> stack;`
- je ein Leerzeichen bei `:` und `?`:
  - `i = (i > SIZE) ? SIZE : i;`
  - `for (int i : indices) {`

C++

- prinzipiell die Regeln wie in Java!
- Extensions: ”`.cpp`”, ”`.h`”
- `class` Header (wenn geht in einer Zeile):

```
class Dog : Animal {
```

- `private`, `public`, `protected`: 2 Zeichen eingerückt

```
class VerySpecialDogThatIsVerySpecial
  : public VerySpecialAnimal {
  public:
    int getId();
    int getId();
    void setId(int id);
  private:
    int id;
}
```

C++ – 2

- **kein** `using namespace X;` in Header-Dateien!
- Namespaces in Kleinbuchstaben
  - Wenn mehrere Wörter, dann durch _ getrennt
- Der `#define` Guard: `PROJEKT_PATH_FILE_H`
  - z.B. bei `foo/src/bar/baz.h`: `FOO_BAR_BAZ_H`
- Konstruktoren mit nur einem Parameter sollen `explicit` sein
- Reihenfolge der `#include`-Direktiven
  1. eigene Modulheaderdatei
  2. eigene sonstige Headerdateien
  3. Headerdateien der verwendeten Bibliotheken
  4. Systemheaderdateien

C++ – 3

- Copy-Konstruktoren und Assignment-Operatoren nur wenn notwendig!
  - detto: Move-Konstruktor und Move-Assignment-Operator

```
class Foo {
  public:
    explicit Foo(int i) : ip{new int{i}} {}
    ~Foo() { delete ip; }
    int* ip{};
  private:
    Foo(const Foo&) = delete;
    void operator=(const Foo&) = delete;
};
```

C++ – 4

- Nur eine Deklaration pro Zeile!
- Initializer
  - `int i{0};`
  - `auto j{0};`
    - auch bei `for (int i{1}; i < 10; ++i)`
  - `int[] numbers{0, 1, 2, 3, 4};`
  - bei Verwendung von ”=” $\rightarrow$ keine Leerzeichen um ”=”
- Default-Argumente
  - `void f(int i=0);`

C++ – 5

- Pointer und Referenzen
  - `int* p;`
  - `int* const p;`
  - `const int* const p;`
- Scope: { und } auf gleicher Höhe

```
{
    lock_guard<mutex> lg{mtx};
    c = a + b;
}
```

Tipps

Bezeichner - Allgemein

- sprechende oder übliche Bezeichner sind zwingend!
- Übliche Kurzbezeichner
  - i, j, k: Indizes, ganze Zahlen
  - e: Element als Schleifenvariable (besser: sprechender Name)
  - o: Objekt als Schleifenvariable (besser: sprechender Name)
- Bezeichner und Buchstaben, die zu vermeiden sind
  - l: kleines L
  - O: Großbuchstabe O
- Funktionen: Verb oder Verb/Substantiv…
  - `void write(string name)`
  - `void write_field(string name)`

Bezeichner - Beispiel

```
public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if (x[0] == 4)
            list1.add(x);
    return list1;
}
```

- Welche Dinge sind in `theList` gespeichert?
- Welche Bedeutung hat der Index `0` der Elemente von `theList`?
- Welche Bedeutung hat der Wert `4`?
- Für welchen Zweck wird `list1` verwendet?

Bezeichner - Beispiel – 2

- Neue Version:

```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
        if (cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.add(cell);
    return flaggedCells;
}
```

- Alte Version:

```
public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if (x[0] == 4)
            list1.add(x);
    return list1;
}
```

Bezeichner - Übliche Abkürzungen

Auch als Präfix, Infix oder Postfix zu verwenden!

aux	auxiliary: Behelfs… oder Neben…
bak	backup
cls	class
cnt	counter
ctx	context
curr	current: aktuell
db	database
dict	dictionary (in Python: eingebauter Typ)
elem	element
eof	end of file
exp	expected: erwartet
id	identifier (in Python: eingebaute Funktion)

Bezeichner - Übliche Abkürz. – 2

idx	index
in	input (in Python, C#: Schlüsselwort); Alternative: `inp`
is	$\rightarrow$ boolesch, als Teil von Namen (z.B. `isFull`) (in Python und C#: Schlüsselwort)
lst	list
mgr	manager
max	maximum (in Python: eingebaute Funktion, in C++ Funktion in `<algorithm>`)
min	minimum (siehe `max`)
msg	message
num	number
obj	object
oid	object identifier

Bezeichner - Übliche Abkürz. – 3

orig	original
out	output (in C# Schlüsselwort); Alternative `outp`
prev	previous: vorhergehend
pos	position
rec	record: Datensatz, aufzeichnen
ref	reference: Referenz, Bezug, Hinweis, hinweisen
res	result
seq	sequence
str	string (in Python: eingebauter Typ)
struct	structure (in C# und C++: Schlüsselwort!)
tmp	temporary
uid	unique id