Style Guide für Programmcode

Neben den hier genannten Richtlinien besitzen die meisten Programmiersprachen auch ihre eigenen Style Guides mit mehr Sprach-spezifischen Tips und Regeln:

- Google Java Style Guide
- PEP 8 — Style Guide for Python Code
- MATLAB Style Guidelines 2.0

Hintergrund

Programmcode wird im Rahmen des Studiums verwendet, um technische Probleme zu lösen, und um Problemstellungen und Lösungen zu illustrieren. Es ist daher essentiell, dass Programme sowohl *korrekt* sind, als auch *lesbar*.

Programmcode ist in diesem Sinne wie jeder andere Text, und es muss Wert gelegt werden auf logischen Aufbau, sinnvollen Satzbau und sorgfältige Wortwahl. Schlecht geschriebener Programmcode führt genau wie schlecht geschriebener Text unnötig komplex und missverständlich. Gut geschriebener Code macht komplexe Zusammenhänge klar und einleuchtend.

Im Laufe der Programmierung eines Programmes probiert ein Programmierender oft mehrere Lösungswege aus, bis er eine funktionierende Lösung findet. Dementsprechend sind Programme anfangs oft unklar formuliert, da verschiedene Teile mit verschiedenen Zielen geschrieben wurden. Wie auch bei Text muss solcher Code umformuliert werden, um klar und verständlich zu werden.

Diese Tätigkeit des Überarbeitens und Verschönerns von Programmcode sollte nicht vernachlässigt werden und kann leicht ähnlich viel Zeit kosten kann wie das ursprüngliche Schreiben des Codes selbst. Dieser Aufwand ist aber in der Regel verschwindend gegenüber der Menge an Ärger und Frustration, die man sich durch missverständlichen Programmcode einhandelt, den man wieder und wieder neu lesen muss und der fast zwangsläufig zu Folgefehlern führt.

Gleichzeitig ist gut geschriebener Code dramatisch einfacher zu verstehen, Fehler sind leichter zu finden, und der Code kann mit weniger Aufwand an neue Anforderungen angepasst werden. Kurz: Lieber früh Aufwand in guten Style stecken als später in Fehlersuche zu ersticken und der Komplexität nicht mehr Herr werden.

Dieser Style Guide soll als Leitfaden dienen, um „Write-Only Code“ zu vermeiden, und Hilfestellungen für gute Formulierungen zu bieten. Jenseits dessen ist dieses Dokument aber *kein Gesetzbuch*. Lesbarkeit geht immer vor, und ist in jedem Fall wichtiger als sklavische Konsistenz.

Namensgebung

Man kann in Programmcode nur mit Dingen arbeiten, die man benennen kann. Jede Variable, jede Funktion, jede Klasse, jede Methode und jede Datei kann nur verwendet werden, wenn ihr Name bekannt ist. Die Wahl von geeigneten Namen ist daher von größter Wichtigkeit.

Die folgenden Abschnitte zeigen Beispiele für gute Namensgebung von Variablen, Funktionen, Dateien und Klassen.

Diese hier gezeigten Richtlinen stellen nur *eine* Möglichkeit von sinnvoller Namensgebung dar. Gerade in der Zusammenarbeit mit anderen Menschen ist es wichtiger, einem gemeinsamen Konzept zu folgen, als allein eine „bessere“ Schreibweise zu verteidigen.

Allgemeines

Für Namen gilt das selbe wie für den Rest des Programmcodes: Lesbarkeit und Verständlichkeit sind das oberste Gebot. Wenn sich die Bedeutung eines Namens klar aus dem Kontext erschließt, ist es ein guter Name.

Englisch

Programmiersprachen sind in der Regel englisch. Alle eingebauten Funktionen, Klassen, und Keywords sind in den allermeisten Programmiersprachen englisch. Darüber hinaus ist aber auch die allgemeine Sprache des Programmierens, der Ingenieure und der Wissenschaft weltweit englisch.

Demnach sollten auch alle eigenen Namen auf Englisch vergeben werden, und auch alle Kommentare auf englisch verfasst werden.

Namen mit ganzen Worten benennen

Die allermeisten Namen sollten normale englische Worte sein. Nur selten kann ein mit sehr kurzen Variablennamen Algorithmus übersichtlich formuliert werden.

Besonders in mathematischem Code kann es aber angebracht sein, Variablennamen wie `x`, `y`, `t` oder `n` zu verwenden. Das darf aber wirklich nur gemacht werden, wenn die Variablen sehr lokal verwendet werden und ihre Bedeutung aus dem Kontext absolut klar ist. In allen anderen Fällen sollten vollständige Worte als Variablennamen verwendet werden.

Ausnahmen sind fachspezifische Abkürzungen wie etwa `fs` für die Sampling-Rate oder `fft` für die Fast Fourier Transform. Andere Fachbereiche haben entsprechend eigene Abkürzungen. Selbst dann sollte man aber darüber nachdenken, etwa statt `fs` `samplerate` zu verwenden, um fachfremden Programmierer_innen zu helfen.

Grundsätzlich zu vermeiden ist aber die Verwendung von `l` und `O`, die mit `1` und `0` verwechselt werden könnten, und `i`, `j` und `e`, die je nach Programmiersprache für die imaginäre Zahle und die Eulersche Zahl reserviert sind.

Zusammengesetzte Worte in camelCase oder snake_case

Wenn ein Name aus mehr als einem Wort besteht, sollten die Unter-Worte klar trennbar sein. Das kann man beispielsweise erreichen, indem die jeweils ersten Buchstaben eines Wortes groß geschrieben werden („*camelCase*“), oder indem Unterstriche zwischen den Worten einfügt werden („*snake_case*“). Verschiedene Programmiersprachen haben hier verschiedene Konventionen:

MATLAB

```
worldDomination % lower-case camelCase for variables and functions
EvilGenius      % upper-case CamelCase for classes
```

Java

```
worldDomination // lower-case camelCase for variables and methods
EvilGenuis      // upper-case CamelCase for classes and types
```

python

```
world_domination # lower-case snake_case for variables and functions
EvilGenius       # upper-case CamelCase for classes and types
```

Üblicherweise unterscheidet man bei der Schreibweise auch zwischen Variablen- und Funktionsnamen und Klassen- und Typennamen. Um Diese im Code auseinanderzuhalten, werden oft Variablen- und Funktionsnamen *klein geschrieben* und Klassen- und Typennamen *groß geschrieben*.

Aussprechbare Namen

Da Code nicht nur auf dem Computer existiert, sondern der Programmierende sich auch regelmäßig mit Kolleg_innin und Kommiliton_innen über Programmier-Arbeit austauschen muss, ist es zweckmäßig, aussprechbare Namen zu vergeben. Das schließt auch automatisch Abkürzungen wie `strcpy` aus (gesprochen: „String-Copy“).

Ebenso sollte darauf geachtet werden, keine Namen zu vergeben die ähnlich ausgesprochen, aber unterschiedlich geschrieben werden.

Variablennamen

In der Regel beschreiben Variablen *Werte*, die von einem Programm verwendet werden. Dementsprechend sollten sie mit *Nomen* benannt werden.

Die meisten Programmiersprachen erlauben mehr oder weniger beliebige Zeichenkombinationen als Variablennamen. Insbesondere hat keine Programmiersprache ein Problem mit

```
f = m * a
```

Menschliche Leser werden es aber immer einfacher finden, mit Worten anstatt von einzelnen Buchstaben zu arbeiten:

```
force = mass * acceleration
```

Einheiten oder Typen als Suffix anhängen

Gelegentlich kann es nützlich sein, die Einheit einer Variable anzugeben. Man schreibt das üblicherweise als ausgeschriebener oder abgekürzter Suffix:

```
elevationAngleRadian
speedKPH
coordinatesLatLon
coordinates_pixels
altitude_feet
frequency_kHz
```

Dies kann auch verwendet werden, um den Datentyp von Variablen zu beschreiben, wenn das nicht aus dem Kontext ohnehin selbsterklärend ist

```
dateString = "2017-04-04"
covarianceMatrix = [ ... ]
```

Prefix **n** für Anzahlen

Häufig vorkommende Konzepte wie „Anzahl an Dateien“ kann ähnlich üblicher mathematischer Notationen mit dem Prefix `n` geschrieben werden:

```
nFiles, nBlocks, nExperiments
```

Collections im Plural benennen

Wenn eine Variable mehrere Werte auf einmal enthält, kann man dies durch einen Variablennamen im Plural verwenden, etwa:

MATLAB
```
numbers = [1, 2, 3, 4]
```

python
```
employee_names = ["Jen", "Tom", "Max"]
```

In manchen Fällen kann es nützlicher sein, die Art der Collection direkt zu benennen:

MATLAB
```
timeVector = linspace(0, 1, 44100)
```

python
```
coordinateList = [(0, 0), (0, 1), (2, 1)]
```

Konsistente Benennung von Schleifen-Variablen

In Schleifen kommen häufig Zählvariablen oder Indizes vor. Als Konvention sollten diese im Singular und gegebenenfalls mit dem Präfix `idx` benannt werden:

python
```
for file in fileList:
    # ...
```

MATLAB
```
for idxFile=1:nFiles
    % ...
end
```

Prefix `is` oder `has` für Boolean-Variablen

Variablen, die nur `wahr` oder `falsch` enthalten können, sollten dies mit einem geeigneten Variablennamen kennzeichnen, etwa

MATLAB
```
isValid
```

oder

python
```
has_attribute
```

Dabei sollten negative Bezeichnungen (`isInvalid`, `isNotValid`) vermieden werden. Statt dessen können die entsprechenden negierungs-Operatoren verwendet werden:

MATLAB
```
~isValid % instead of isNotValid
```

python
```
not has_attribute # instead of doesnt_have_attribute
```

Keine eingebauten Variablennamen überschreiben

Jede Programmiersprache besitzt einige vordefinierte Namen, die von der Programmierumgebung zur Verfügung gestellt werden. Wenn solche Namen in einem Programm neu definiert werden, überschreiben sie damit die vorhandenen Funktionalitäten mit den neuen Werten. Dies kann zu fatalen Fehlern führen

MATLAB
```
pi = 3    % don't EVER do this
true = [] % or this
```

Besonders wichtig ist das in Matlab, da dort alle Funktionen immer im globalen Workspace zur Verfügung stehen. Besonders aufpassen sollte man bei Variablennamen wie:

MATLAB
```
alpha, angle, axes, axis, balance, beta, contrast, gamma, image,
info, input, length, line, max, min, mode, power, rank, run, start,
text, type
```

Diese Namen können statt dessen durch ein zusammengesetztes Wort ersetzt werden, etwa `arrayLength` anstatt von `length`. Ebenfalls akzeptabel ist die Verwendung des Prefix `this`, etwa `thisLength`.

In Python ist es außerdem üblich, verbotene Namen mit einem Postfix-_ zu versehen, etwa `len_`, da `len` eine eingebaute Funktion ist.

Konstanten

Dynamische Programmiersprachen wie Matlab oder Python besitzen keine Möglichkeit, echte Konstanten zu definieren. Um so mehr sollte man im Programmcode Wert darauf legen, Konstanten deutlich zu kennzeichnen, üblicherweise durch konsequente Großschreibung:

```
MAX_ITERATIONS, LOG_FILE_NAME
```

Funktionen

Funktionen sollten in der Regel wie Variablen benannt werden. Anders als Variablen stellen Funktionen aber *Aktionen* dar, und sollten dementsprechend mit *Verben* benannt werden. Wie auch bei Variablennamen sollten weiterhin vollständige Worte verwendet werden.

MATLAB

```
function newData = transformData(oldData)
    % ...
end
```

Viele Programmiersprachen verwenden aus historischen Gründen teilweise noch früher gebräuchliche Abkürzungen wie

MATLAB
```
strcmp % "String Compare" from C
```

python
```
os.rmdir # "Remove Directory" from C
```

aber auch übliche mathematische Funktionen wie

MATLAB
```
sin, cos, min, max, exp, gcd, mean, std
```

Für selbst geschriebenen Code sollten solche Namen vermieden werden und in jedem Fall mit besonderer Sorgfalt dokumentiert werden.

Funktionen mit nur einem Rückgabewert nach ihrem Rückgabewert benennen

Wenn eine Funktion nur einen einzigen Rückgabewert berechnet, und keine weitere Neben-Aufgabe hat als diesen Rückgabewert zu berechnen, kann sie nach diesem Wert benannt werden, etwa

python
```
averageHeight = mean(heights)
```

Es gelten dann alle Namensregeln wie bei Variablen, also

- Zusammengesetzte Worte in camelCase oder snake_case
- Prefix `is` oder `has` für Booleans
- Plural oder Typ-Bezeichnung für Collections
- Prefix `n` oder für Längenbezeichnungen
- Prefix `idx` für für Indizes

In einigen Programmiersprachen werden Funktionen *immer* als Verb benannt. In diesem Fall kann der Prefix `compute` vor der Bezeichnung des Rückgabewerts verwendet werden:

Java
```
covariance = Covariance().computeCovarianceMatrix(dataMatrix);
```

Gelegentlich sieht man statt `compute` auch den Prefix `get`; dieser sollte aber vermieden werden, da er konventionell für Getter und Setter verwendet wird.

Getter und Setter

In Programmiersprachen, die keine expliziten Accessors[1] unterstützen, sollten die Namen von Gettern und Settern mit `get` und `set` beginnen:

Java

```java
public class Account {
    private float amount = 0;

    public float getAmount() {
        return this.amount;
    }

    public void setAmount(float newAmount) {
        if (newAmount < 0) {
            throw new Exception("Account can't hold negative money");
        } else {
            this.amount = newAmount;
        }
    }
}
```

In Matlab findet man vereinzelt noch alten Code, der von einer Zeit stammt, als Objektorientiertes Programmieren in Matlab noch nicht gut funktionierte. Dort kann man teils noch Methoden finden, die mit `get` und `set` beginnen. In modernem Code sollte man statt dessen *Dependent Properties*[2] verwenden:

MATLAB

```matlab
classdef Account < handle
    properties (Dependent)
        amount = 0
    end

    methods
        % gets called when x = account.amount
        function amount = get.amount(obj)
            amount = obj.amount;
        end

        % gets called when account.amount = x
        function set.amount(obj, newAmount)
            if newAmount > 0
                obj.amount = newAmount;
            else
                error('Account can''t hold negative money');
            end
        end
    end
end
```

[1] Ein *Accessor* verhält sich von außen wie ein normaler Member-Variablen-Zugriff, ruft aber in Wirklichkeit bei jedem Lese- und Schreibzugriff auf die Member-Variable eine Getter- und Setter-Funktion auf. Dieses Konzept wird in verschiedenen Sprachen unterschiedlich genannt.

[2] So werden *Accessors* in Matlab genannt.

In Python sollten entsprechend *Properties*[3] verwendet werden:

python

```python
class Account:
    def __init__(self):
        self._amount = 0

    # gets called when x = account.amount
    @property
    def amount(self):
        return self._amount

    # gets called when account.amount = x
    @amount.setter
    def amount(self, new_amount):
        if new_amount < 0:
            raise RuntimeError("Account can't hold negative money")
        else:
            self._amount = new_amount
```

Zusammengehörige Funktionen zusammengehörig benennen

Funktionen kommen oft im Doppelpack vor, etwa

```
get/set, add/remove, create/destroy, start/stop, insert/delete,
increment/decrement, old/new, begin/end, first/last, up/down,
min/max, width/height, lat/lon, next/previous, open/close,
show/hide, suspend/resume, ...
```

In diesem Fall sollten die oben genannten Paarungen verwendet werden, da sie konventionell zusammen gehören[4].

Struktur

Programmcode jeder Art muss klar aufgebaut sein, um leicht lesbar zu sein. Wie auch ein Buch aus mehreren Kapiteln mit einer logischen Folge von Ereignissen und Erzählungen besteht, muss auch Code logisch aufgebaut werden, und die Dinge in einer sinnvollen Reihenfolge nennen.

Wenn man die Wahl hat, empfiehlt es sich, Code von außen nach innen zu schreiben, sprich: Als erstes in einer Datei sollten die allgemeinsten Funktionen vorkommen, die den groben Ablaufplan des Programms aufzeigen; danach kommen schrittweise Unterfunktionen, die von den allgemeineren Funktionen aufgerufen werden. Auf diese Art liest sich Code annähernd wie eine Zeitungsnachricht, bei der auch erst eine allgemeine Übersicht gezeigt wird und dann schrittweise immer weiter ins Detail gegangen wird.

[3] So werden *Accessors* in Python genannt.

[4] Wann immer man eine solche Paarung sieht, sollte man aber auch kurz darüber nachdenken, ob sich dieses Konzept nicht auch anders ausdrücken ließe, etwa durch Konstrukte wie *Accessor* statt `get`/`set`, *RAII*[5] statt `create`/`destroy`, *Context Manager*[6] statt `start`/`stop`, *Iterator*[7] statt `next`/`previous`, Wertepaare statt `width`/`height`, etc.

Kryptischen Code vermeiden

Im Falle von Programmcode steckt in der Kürze selten Würze. Statt dessen sollte Code maximal lesbar sein. Wenn das bedeutet, dass einige Variablennamen etwas länger ausfallen, und manch ein Zwischenergebnis auf eigenen Zeilen in eigenen Variablen abgelegt wird, dann ist das kompakterem Code vorzuziehen.

Die zentrale Frage, die man sich als Programmierer_in immer wieder stellen sollte ist: „Werde ich diesen Code in einem Monat noch verstehen¿‘ Sollte die Antwort auf diese Frage jemals negativ ausfallen, ist das ein klares Zeichen, dass man den Code umformulieren sollte.

Klammern und Leerzeichen verwenden und Missverständnisse vermeiden

Jede Programmiersprache hat eindeutige „Punkt vor Strich“-Regeln, die die *Präzedenz* von Operatoren definiert. Dennoch ist es leicht, verwirrenden Code zu schreiben, in dem die Reihenfolge der Operationen nicht offensichtlich ist.

Um Missverständnisse zu vermeiden sollten daher Leerzeichen und Klammern verwendet werden, um zusammengehörige Ausdrücke zu kennzeichnen, etwa `2^4 * 2` oder `(2^4) * 2` anstatt von `2^4*2`.

Ausdrücke auf mehrere Zeilen aufteilen

Ähnlich wie lange mathematische Ausdrücke oft auf mehrere Zeilen verteilt werden, ist es oft auch nützlich, Programmcode innerhalb einer Zeile umzubrechen.

Um einen bestehenden Ausdruck auf mehrere Zeilen zu verteilen, müssen je nach Programmiersprache die Zeilenumbrüche markiert werden. In Java kann eine Zeile jederzeit umgebrochen werden:

Java

```java
String longString = "some long text"
                  + "some more text";
```

Matlab muss jeden Zeilenumbruch (außer innerhalb von Matrizen) mit einem `...` markieren:

MATLAB

```matlab
longString = strcat('some long text', ...
                    'some more text');
```

In Python müssen Zeilenumbrüche mit `\` markiert werden oder innerhalb von Klammern stehen:

python

```python
long_string = 'some long text' \
            + 'some more text'
long_string = ('some long text'    # no \ necessary since inside parenthesis
               + 'some more text')
```

Variablen einmalig und eindeutig verwenden

Es ist oft verlockend, Variablen nacheinander für verschiedene Zwecke wiederzuverwenden. Das macht Programmcode allerdings schwerer zu lesen, da nun die Variable zu verschiedenen Zeiten verschiedene Konzepte repräsentiert.

Es ist besser, Variablen pro Funktion einmalig und eindeutig zu verwenden, damit immer klar ist, was mit einer Variable gemeint ist.

Zwischenergebnisse in Variablen speichern

Hier ein Beispielprogramm, in dem in einer Zeile eine Audiodatei eingelesen wird, eine gemeinsame Mono-Spur berechnet wird, und deren RMS-Loudness in dB berechnet wird:

MATLAB
```
level = 20*log10(sqrt(mean(sum(audioread(filename)^2, 2))));
```

Dieses Programm wäre deutlich leichter zu verstehen, hätte man diese Schritte einzeln ausgeführt:

MATLAB
```
stereoData = audioread(filename);
monoData = sum(stereoData, 2);
rms = sqrt(mean(monoData^2));
level = 20*log10(rms);
```

Auf diese Art können Variablennamen auch schon als Dokumentation dienen, indem einzelne Zwischenergebnisse benamt werden.

Code-Abschnitte in Funktionen auslagern

Wenn man das Gefühl hat, dass ein Code-Abschnitt mit einem Überschrifts-Kommentar besser verständlich wäre, ist es oft eine bessere Lösung, diesen Abschnitt in eine eigene Funktion auszulagern:

python
```
blocklength = 1024
filename = 'audiofile.wav'


# load audio data as blocks:
blocks = []
with soundfile(filename) as file:
    blocks.append(file.read(blocklength))


# calculate spectrogram
window = hann(blocklength)
spectrogram = []
for block in blocks:
    spectrogram.append(abs(fft(block*window)))
```

besser:

python
```
blocks = read_audio_blocks(filename, blocklength)
spectrogram = calculate_spectrogram(blocks, window)
```

```
def read_audio_blocks(filename, blocklength):
    blocks = []
    with soundfile(filename) as file:
        blocks.append(file.read(blocklength))
    return blocks

def calculate_spectrogram(blocks, window):
    spectrogram = []
    for block in blocks:
        spectrogram.append(abs(fft(block*window)))
    return spectrogram
```

Im obigen Beispiel sieht man, wie Funktionsnamen als eine Art Dokumentation verwendet werden, indem sie die Bedeutung eines Programmes beschreiben.

Der große Vorteil, dies in Funktionsnamen anstatt von Kommentaren zu machen ist, dass Funktionsnamen zwangsläufig bei Änderungen des Codes mit angepasst werden müssen. Kommentare dagegen bleiben leicht unbeachtet und werden bei Änderungen des Codes nicht mitgeändert—und solcherlei *falsche Kommentare* sind schlimmer als gar keine Kommentare.

Globale Variablen vermeiden

Globale Variablen sind häufig ein Quell von schwer zu findenden Fehlern, da sie unbemerkt geändert werden können und das Verhalten von Funktionen trotz gleicher Funktionsargumente ändern können.

Statt dessen können Klassen, *persistent Variables* oder *static Variables* verwendet werden.

Formatierung

Um Programmcode leicht lesbar zu machen, muss er sinnvoll formatiert sein. Sinnvoll bedeutet in der Regel *konsistent*. Es ist wichtig, dass ähnliche Dinge ähnlich geschrieben werden, damit der Leser den Sinn des Codes leicht erfassen kann. Umgekehrt ist es wichtig, unabhängige Dinge klar von einander zu trennen, damit der Leser nicht verwirrt wird.

Die folgenden Empfehlungen sind nicht die einzige Art, Code sinnvoll zu formatieren. In manchen Umgebungen werden leicht unterschiedliche Varianten dieser Regeln befolgt. Im Zweifel ist es immer wichtiger, konsistent mit dem restlichen Code des Projekts/der Firma/der Programmiersprache zu sein, anstatt auf eigenen „richtigeren“ Regeln zu bestehen.

Leerzeilen verwenden, um Code-Blöcke voneinander zu trennen

Wie beim Schreiben von Prosa, sollte auch Code in logische Absätze unterteilt werden. *Ein* Absatz spricht von *einem* Thema, und Absätze sind mit Leerzeilen voneinander abgesetzt.

Leerzeilen sollten konsistent eingesetzt werden und es sollte nie mehr als eine oder zwei Leerzeilen auf einmal eingesetzt werden. Python hat etwa die Konvention, *innerhalb* von Funktionen einzelne Leerzeilen für Absätze zu verwenden, aber *Funktionen untereinander* mit zwei Leerzeilen zu trennen.

Leerzeichen zwischen Operatoren und Variablen verwenden

Für Code gelten prinzipiell die selben Zeichensetzungsregeln wie für Text:

- Operatoren sollten mit Leerzeichen abgesetzt sein, also `x + y` anstatt von `x+y`
- Auf jedes Satzzeichen folgt ein Leerzeichen, also `[x, y]` anstatt von `[x,y]`
- Kein Leerzeichen nach öffnenden Klammern und vor schließenden Klammern, also `[x]` anstatt von `[ x ]`

Diese Regeln sollten aber immer im Kontext betrachtet werden. So kann es beispielsweise sinnvoll sein, von diesen Regeln abzuweichen um Punkt-vor-Strich Reihenfolgen klarzustellen.

Zeilenlänge

Als Grundregel sollte keine Zeile länger als etwa 72-100 Buchstaben sein. Historisch kommt die Faustregel von alten IBM Lochkarten, die pro Lochkarte maximal 80 Zeichen speichern konnten, von denen 8 Zeichen für die Seriennummer reserviert waren. Frühe Terminals konnten ebenfalls nicht mehr als 80×24 Zeichen pro Zeile darstellen. Selbst „moderne“ Terminals wie das Windows CMD halten auch heute noch an dieser Tradition fest. Schreibmaschinen hatten meist ein ähnliches Limit.

Das mag aus heutiger Sicht anachronistisch und albern erscheinen. Es spricht allerdings einiges dafür, auch heute noch eine konsistente Zeilenlänge einzuhalten:

- Der Standard in der Typographie von Büchern ist etwa 60 Buchstaben pro Zeile für maximale Lesbarkeit. Zu lange Zeilen machen es schwierig, vom Ende der Zeile zurück zum passenden Anfang der nächsten Zeile zu springen.
- Eine begrenzte Zeilenlänge erlaubt es auch auf Laptops, zwei Fenster nebeneinander anzuordnen.
- Code mit einer begrenzten Zeilenlänge ist problemlos druckbar.
- Code mit begrenzter Länge kann mit Sehbehinderungen auch in großer Schriftgröße noch sinnvoll gelesen werden.

Dennoch sollte man die Zeilenlänge nicht sklavisch auf einen bestimmten Wert limitieren. Die Lesbarkeit steht weiterhin im Vordergrund. Es ist in der Regel kein Problem, wenn einzelne Zeilen leicht über das Limit hinausstoßen. Dennoch sind *Im Allgemeinen* 80 Zeichen eine gute Richtlinie, die man nicht ohne Grund überschreiten sollte.

Code sauber einrücken

Programmcode ist in der Regel immer eine rekursive Struktur aus ineinander verschachtelten Blöcken, etwa `if`-Statements in `for`-Loops in Funktionen. Um diese Struktur im Programmcode sichtbar zu machen, werden innere Blöcke tiefer eingerückt als äußere Blöcke.

Konventionell werden pro Einrückungsstufe 2, 4, oder 8 Leerzeichen oder ein Tab verwendet. Der Standard in Matlab und Python sind 4 Leerzeichen.

Sprachen wie Java oder Matlab stellen diese Verschachtelungsstruktur durch Zeichen wie `{`, `}`, oder `end` dar. Es gibt daher genau *eine* korrekte Einrückung, die auch durch die üblichen Texteditoren automatisch korrigiert werden kann, in Matlab etwa mittels Strg-I  .

Python  ist hier ein Sonderfall, da die Verschachtelungsstruktur allein *durch* die Einrückung angegeben wird. In Python gibt es daher keine Möglichkeit, Code falsch einzurücken, da falsch eingerückter Code auch automatisch ein Programmfehler ist.

Code kommentieren

Der Sinn von Kommentaren ist es, zusätzliche Informationen über den Code hinzuzufügen. Es versteht sich daher von selbst, dass Kommentare nicht den Code selbst beschreiben sollten, sondern darüber hinaus Mehrinformation enthalten sollten, das nicht ohnehin schon im Code steht.

Beispielsweise können Kommentare den *Sinn* oder die *Motivation* eines Code-Abschnittes beschreiben, ein Benutzungsbeispiel geben, Probleme und Einschränkungen eines Programmes nennen, Entscheidungen der Programmierer dokumentieren, Referenzen und Querverweise liefern, oder Verbesserungsmöglichkeiten dokumentieren.

In jedem Fall zeigt es sich, dass solcherlei Informationen am Besten *während des Programmierens* geschrieben werden. Versucht man, Kommentare erst später hinzuzufügen, hat man oft schon die wichtigsten Punkte vergessen.

Dabei sollte man sich Mühe geben, Kommentare in sauberem Englisch und in einfachen Worten zu halten. Nicht jeder Leser kennt komplizierte Fremdworte. Kommentare sind ein wichtiger Bestandteil des Codes, der nicht übersehen werden sollte.

Dokumentation

Je nach Programmiersprache gibt es verschiedene Konventionen, um Code außerhalb des Source Codes zu dokumentieren. Diese Dokumentation ist dann in den entsprechenden Hilfe-Browsern der Sprache verfügbar und sollte die Verwendung von Funktionen und Klassen beschreiben.

Die Dokumentation von Funktionen sollte zumindest alle Input-Argumente und Returnwerte und gegebenenfalls Seiteneffekte[8] dokumentieren. Besser sollten außerdem noch Nutzungshinweise, Querverweise, Beispiele und typische Fehler genannt werden.

Allen Dokumentations-Konventionen ist gemein, dass die erste Zeile jeweils eine Kurzzusammenfassung der Funktion geben sollte. Die folgenden Zeilen können dann die Funktion und ihre Argumente in größerem Detail beschreiben.

In Java werden hierfür nach *Javadoc*-formatierte Kommentarblöcke direkt vor Methodenbeginn verwendet:

Java

```java
/**
 * Calculates the root mean square level of a signal
 *
 * Calculates the variance of a zero-mean signal as the root of the
 * mean of the square of all signal samples.
 *
 *
 * @param signal an array of float samples
 * @return       the level as a float
 */
public int rms(Array signal) {
    // ...
}
```

In Matlab wird Dokumentation im ersten Kommentar nach der Funktionsdeklaration geschrieben, und folgt ebenfalls speziellen Formatierungsregeln:

[8] Alles was eine Funktion herbeifürt, das *nicht* als Rückgabewert zurückgegeben wird. Dazu gehören etwa Daten ausgeben, globale Variablen ändern, Dateien schreiben oder Benutzereingaben abfragen.

MATLAB

```
function value = rms(signal)
%RMS calculates the root mean square level of a signal
%   VALUE = RMS(SIGNAL) Calculates the variance of a zero-mean SIGNAL
%   as the root of the mean of the square of all signal samples.
%
%   SIGNAL is a double vector
%   VALUE is a scalar double

% ...

end
```

In Python wird Dokumentation im sogenannten *Docstring* gespeichert, einem Multi-Line String der direkt als erstes Statement einer Funktion steht. Die Formatierung des Docstrings ist dabei nicht fest vorgeschrieben. Als Konvention hat sich aber reStructuredText etabliert:

python

```
def rms(signal):
    """Calculates the root mean square level of a signal.

    Calculates the variance of a zero-mean signal as the root of the
    mean of the square of all signal samples.

    Parameters
    ----------
    signal
        An array of signal samples.

    Returns
    -------
    rms : float
        The RMS value.
    """

    # ...
```