



Programmierhandbuch (Style Guide)

EPI (EPR und GPR) für (Python 3.x)

Version 2021/22

Bitte gewöhnen Sie sich einen guten Programmierstil an und halten Sie sich an die Konventionen, nur so können Sie ein vollwertiges Teammitglied sein, ist ihr Quelltext auch von anderen schnell überschaubar und gut lesbar. Wir halten die anzuwendenden Konventionen minimal aber auch in den Übungen wird darauf Wert legen, es gibt ggf. Punktabzug. Wir beziehen uns auf den „offiziellen“ „Style Guide for Python Code“ von Guido van Rossum, siehe <http://www.python.org/dev/peps/pep-0008/>. Dies sind alles nur Konventionen. Dieses Handbuch ist sehr kurz gefasst, es gibt nur wenige (kursiv gesetzte) Begründungen oder Kommentare.

Sollten (wider Erwarten ;-)) Widersprüche zwischen diesen Richtlinien und PEP 8 auftreten, so sind im Zweifel auch die Regeln von PEP 8 zugelassen.

Viel Erfolg!

... basierend auf den Style Guide von Prof. Dr. Detlef Krömker. Vielen Dank für das Überlassen.

Inhalt

Inhalt	2
1 Der Modul (-kopf)	3
2. Namenskonventionen	6
3. Imports	7
4. Source Code Formatierung	7
5. Kommentare	8
6. Operatoren und Auswertereihenfolge in Python	9
7. Vergleichsoperatoren	10
8. Funktionen zur Typwandlung	10
9. Häufig benutzte Operationen auf Sequenz-Typen	11

1. Der Modul(-kopf)

1. Die erste Zeile eines jeden Moduls (oder Programms) **muss** ein **Docstring** sein. Wenn dieser String länger als eine Zeile ist, so soll die erste Zeile für sich selbst (allein) verständlich sein, also eine Zusammenfassung sein. Dann folgt eine Leerzeile. Benutzen Sie dabei grundsätzlich die drei Anführungszeichen. Ein Docstring ist also ein string-Literal.

Ein mehrzeiliger Docstring endet mit einer Leerzeile plus einer Zeile in der nur `"""` stehen.

Es gelten die Konventionen wie in [PEP 257](#) angegeben. Schreiben Sie Docstrings für jedes Modul und nach jedem `def` für jede Funktion, und auch später beim objektorientierten Programmieren für jede Klasse und jede Methode.

2. Nach dem **docstring** steht, falls benutzt, ein `from __future__`-Import. Und zwar nur solch einer! Bsp.:
`from __future__ import barry_as_FLUFL`
Voraussichtlich benutzen wir dies nicht!–Nur zur Vollständigkeit.

3. Es folgen die Angaben zur Urheberschaft (`__author__` =) und zu `__credits__` und `__email__` nach dem unten angegebenen Beispiel.

4. Hier stehen die `import`-Statements in folgender Reihenfolge (je eine Zeile pro `import`, zwischen den Gruppen eine Leerzeile):

- `import` built-in Module
- `import` andere „third party modules“
- `import` eigene Module

Tipp: Erstellen Sie sich Ihr persönliches Template. Die sogenannten Dunder-Werte (*Double UNDERscore*) werden den angegebenen internen Variablen zugewiesen und können damit in dem Modul abgefragt werden.

Wir geben keine verpflichtende Vorschriften für die Beschreibung der Parameter bei Funktionen, weil es auch von der Art der vorgesehenen automatisierten Erstellung der Dokumentation abhängt. Wir sind uns aber hoffentlich einig, dass der **docstring genug Informationen geben sollte, um das Objekt aufrufen zu können**, ohne den Quellcode zu lesen oder weitere Dokumentationen zu benutzen.

Hierzu haben sich u.a. die folgenden Methoden etabliert: Das wahrscheinlich vorherrschende Format das **reStructuredText** (reST) -Format, das von Sphinx um Dokumentation zu generieren. Hinweis: Es wird standardmäßig in JetBrains PyCharm verwendet (geben Sie nach dem Definieren einer Methode dreifache Anführungszeichen ein und drücken Sie die Eingabetaste). Es wird auch standardmäßig als Ausgabeformat in Pymont verwendet. Beispiel:

```
"""
This is a reST style.

:param param1: this is a first param
:param param2: this is a second param
:returns: this is a description of what is returned
:raises KeyError: raises an exception
"""
```

Google hat sein eigenes [Format](#), das wird oft benutzt. Es kann auch von Sphinx interpretiert werden.

Beispiel:

```
"""
This is an example of Google style.
Args:
    param1: This is the first param.
    param2: This is a second param.

Returns:
    This is a description of what is returned.

Raises:
    KeyError: Raises an exception.
"""
```

Es sei ausdrücklich auf weitere Möglichkeiten verwiesen, siehe hierzu:
<https://realpython.com/documenting-python-code/> .

So sollte ein Modul aussehen. Sie können dies als Rohversion Ihres Templates benutzen: Bitte entfernen Sie aber auch die Inline-Kommentare, die haben in echtem Code nichts verloren.

Pssst: Jetzt meckert der PEP8 online mit: „module level import not at top of file“. Ja so ist es. -Leider ist das richtige Leben nicht immer so, wie man es mag. Sorry.

```

----snip----
"""Docstring: A very short sentence explaining the function. < 72 chars.

Additional information if required and more infos.
Complete sentences please.
"""

from __future__ import barry_as_FLUFL # Only from __future__ imports allowed.

__author__ = "123456, Cleese" # Put your data here.
__credits__ = '''If you would like to thank somebody
                i.e. an other student for her code or leave it out.'''
__email__ = "your email address" # Mandatory.

from math import sqrt, log, e # An example for builtins.

# from numpy import array      # Another example for a third party module.

# import my_modul             # An example for your own module.

def yourroutine_one():
    """Docstring: Every subroutine and function has a docstring."""

    print ("yourfunction_one was called -- not yet written or incomplete.")

def yourfunction_two():
    """Docstring for yourfunction_two(). """

    pass # To be programed later.

def main():
    """Docstring: This program does ... .

    This module does stuff.
    """

    # You may put your code for initializing your program here, in case it is
    # called as a stand-alone program.

if __name__ == '__main__':
    main()

----snip----

```

2. Namenskonventionen

Alle Namen/Bezeichner sind in Englisch. Sie dürfen nur ASCII Zeichen enthalten (keine Umlaute oder Eßzett).

Wählen Sie Namen, aus denen das Bezeichnete wahrscheinlich richtig erraten wird: `current_line` ist viel besser als `c` oder `cl`. Wählen Sie die Namen so präzise wie möglich.

Benutzen Sie den Singular für individuelle Objekte und den Plural für Kollektionen.

Ein-Buchstaben-Bezeichner sollten nur in sehr überschaubaren Zusammenhängen vorkommen (wie die Variable `i` oder `k`) und nur eine oder zwei Zeilen „überleben“, das heißt eine Bedeutung haben.

Benutzen Sie **nie** „l“ (kleines eL) oder I (großes i) oder O (großes o) als „Ein Buchstaben Bezeichner“. In einigen Fonts sind diese Zeichen nicht oder kaum unterscheidbar von den Ziffern „1“ (Eins) oder „0“ (Null).

Machen Sie den Datentyp nicht zum Teil des Namens.

Benutzen Sie so wenig wie möglich Abkürzungen, insbesondere keine selbst erfundenen. Aber allgemein übliche Abkürzungen, insbesondere in Kombination mit anderen Angaben sind erlaubt, wie

<code>curr</code>	für	<code>current</code> ,
<code>eof</code>	für	end of file,
<code>eol</code>	für	nd of line,
<code>in_value</code>	für	input,
<code>max_value</code>	für	maximum – nicht nur <code>max</code> , weil es eine Funktion gleichen Namens gibt,
<code>min_value</code>	für	minimum – wie bei <code>max</code> ,
<code>out</code>	für	output,
<code>num</code> oder <code>no</code>	für	number,
<code>prev</code>	für	previous,
<code>ref</code>	für	reference,
<code>str_value</code>	für	string,
<code>struct</code>	für	structure,
<code>temp</code> oder <code>tmp</code>	für	temporal.

Schreibweisen von Namen:

Typ	Konvention	Beispiel
Variable	nomen_mit_underscore - Kleinbuchstaben	<code>curr_index</code>
Funktion	aktion_mit_underscore - Kleinbuchstaben	<code>find_all</code>
Konstante	NOMEN_ALLES_GROSS - Großbuchstaben	<code>ALLOWED_RANGE</code>
Globale Variable	sehr selten nutzen, wenn, nur in einem Modul, deshalb Globale Variablen sind bei uns VERBOTEN!	<code>_general_ref_no</code>
Modul Bezeichnung	Kleinbuchstaben (ggf. mit_underscore)	<code>unit_test</code>
Paket Bezeichnung	NUR Kleinbuchstaben (KEIN underscore)	<code>pygame</code>
Für später:		
Klassen	CamelCase (CapWords)	
Method Arguments	Immer <code>self</code> als erstes Argument für Instanz-Methoden benutzen Und <code>cls</code> als erstes Argument für Klassen-Methoden.	
Methoden Namen		
Exceptions	wie Klassen (Camel Case) ggf. mit dem Suffix (Nachsilbe) <code>Error</code> .	

Es gibt folgende Sonderformen von Bezeichnern/Namen:

- `__single_leading_underscore` : Dies ist ein „schwacher“ Hinweis unter Programmierer*innen, dass diese Bezeichner nur intern benutzt werden. `from M import *` (bei uns sowieso untersagt) importiert diese Namen nicht.
- `single_trailing_underscore_`: wird benutzt um Konflikte mit Python Keywords (`False`, `None`, `True`, and usw.), wenn sich die Nutzung nicht vermeiden lässt, z.B. beim import von Modulen aus anderen Sprachen.
- `__double_leading_underscore`: um ein durch *name mangling* geschütztes (non public, um den Begriff *private* zu vermeiden, weil er nicht ganz richtig ist) Attribut zu kennzeichnen, siehe Objektorientierung.
- `__double_leading_and_trailing_underscore__`: im Jargon “Dunder“ für “double underscored” genannt sind "magic" Objects, wie `__author__`, `__version__` leben im User-controlled namespace. Sie haben besondere Bedeutungen für den Interpreter. Bitte nie neue erfinden, sondern nur die dokumentierten benutzen.

3. Imports

Benutzen Sie immer und nur:

```
import module, dann beim Aufruf qualifizierte Namen nutzen ... also module.function
```

```
oder: from module import Name1, Name2, Name3, ...
```

Nie benutzen:

```
from module import *
```

4. Source Code Formatierung

Benutzen Sie zur Einrückung (für die Blockstruktur/Suits) 4 Leerzeichen (Blanks) **nicht** `<tab>`. (Zwischen verschiedenen Plattformen, z.B. MacOS und Windows kann es sonst zu Problemen kommen).

Des Weiteren gelten die Intendation-Regeln der PEP 8: <https://www.python.org/dev/peps/pep-0008/#id17>

Die maximale Zeilenlänge sind 79 oder 99 Zeichen. Benutzen Sie ggf. `\` um diese Zeile „logisch“ zu verlängern.

Leerzeilen sollten genutzt werden, um für sich stehende Teile des Quellcodes zu trennen, gewissermaßen um „Absätze“ darzustellen. Vor Hauptfunktionen (top-level fuctions) stehen zwei Leerzeilen. Vor sonstigen Funktionsdefinitionen (def) steht eine Leerzeile.

Bei mehrzeiligen Docstrings stehen vor und nach dem `"""` je eine Leerzeile.

In Funktionsdefinitionen ansonsten Leerzeilen bitte sehr sparsam verwenden.

Leerzeichen (Blanks) sollen vor und nach jedem Operator stehen, zwingend bei folgenden Operatoren: Zuweisungen (`=`), erweiterte Zuweisungen (`+=`, `-=` etc.), Vergleiche (`==`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, `in`, `not in`, `is`, `is not`), Booleans (`and`, `or`, `not`).

Wenn Operatoren unterschiedlicher Priorität genutzt werden, sollen Operatoren mit der geringeren Priorität mit einem Leerzeichen versehen werden. Aber nie mehr als eins und immer einheitlich.

Bsp.: `((a+b) * (c-d)), x = x*2 - 1`

Bitte **kein** Leerzeichen um das Gleichheitszeichen für ein Schlüsselwortargument (keyword argument) oder bei Default Parametern. Bsp.

```
def complex(real, imag=0.0):  
    return magic(r=real, i=imag)
```

Kein Leerzeichen steht innerhalb direkt neben runden, eckigen oder geschweiften Klammern oder direkt vor einem Komma, Semikolon oder Doppelpunkt. Ausnahmen beim Slicing, siehe <https://www.python.org/dev/peps/pep-0008/#id26>.

Kein Leerzeichen steht vor einer öffnenden runden oder eckigen Klammer.

Um eine Zuweisung (=) herum steht jeweils genau ein Leerzeichen.

Vor einem „Inline-Kommentar“ stehen mindestens 2 Leerzeichen.

5. Kommentare

Kommentieren Sie in Englisch. Passen Sie die Kommentare immer sofort an, wenn Sie Änderungen vorgenommen haben. Kommentare immer mit # markieren – nie Strings benutzen (mit Ausnahme des Docstrings)!

Kommentare sollen ganze Sätze sein: Beginnen also mit einem Großbuchstaben und enden mit einem Punkt. Ausnahme: das erste Wort ist eine kleingeschriebener Name, dann nicht ändern!

Kommentare müssen mehr sagen, als der Code selbst! Kommentare sollen vollständige Sätze sein. Benutzen Sie Inline-Kommentare nur soweit wie nötig. Ein Inline-Kommentar steht in derselben Zeile wie eine Anweisung (ein statement) und wird durch mindestens zwei Leerzeichen, das folgende # und folgendem Blank abgetrennt.

Erklären Sie keine Namen/Bezeichner, diese sollten selbsterklärend sein. Ändern Sie ggf. die Namen und erklären Sie nie das offensichtliche. Ein Beispiel:

```
Falsch:      win_size -= 20 # decrement win_size  
Noch O.K.   win_size -= 20 #leave space for scroll bar  
Richtig:    scroll_bar_size = 20  
            win_size -= scroll_bar_size
```

Block-Kommentare bestehen aus einem oder mehreren Absätzen, bestehend aus vollständigen Sätzen (mit einem Punkt am Ende). Absätze sind durch eine Zeile, die nur ein # enthält, getrennt. Block-Kommentare sind genauso eingerückt (intended), wie der zugehörige Code.

6. Operatoren und Auswertereihenfolge in Python

Operatoren	Kurzbeschreibung	String	Float	Integer	Boolean	In erweiterter Zuweisung anwendbar
(...)	(Vorrang) Klammerung	x	x	x	x	
s[i] s[i:j] f(...)	Indizierung (bei Sequenztypen) Teilbereiche (bei Sequenztypen) Funktionsaufruf	x x x				
x ** y	Exponential-Bildung x^y (Achtung: rechts-assoziativ)		x	x		x
+x, -x, ~x	Einstellige Operatoren Invertiere x		x	x x		
x * y x / y x % y x // y	Multiplikation (Wiederholung) Division ¹⁾ Modulo = Rest Restlose Division (Floor Division) ²⁾	x	x x x x	x x x x		x x x x
x + y x - y	Addition (Konkatenation) Subtraktion	x	x x	x x		x
x << y, x >> y	Bitweises Schieben (nur bei Integer)			x		x
x & y	Bitweises Und (nur bei Integer)			x		x
x ^ y	Bitweises exklusives Oder (nur bei Integer)			x		x
x y	Bitweises Oder (nur bei Integer)			x		x
x >= y, x <= y, x == y, x != y [x <> y] x is y, x is not y x in s, x not in s	Vergleichsoperatoren liefern als Ergebnis True oder False ³⁾ Test auf Identität Tests auf Enthaltensein in Sequenzen	x x	x x	x x	(x) x	
not x	Logische Negation				x	
x and y	Logisches Und				x	
x or y	Logisches Oder				x	

Anmerkungen: Oben stehen die Operatoren mit höherer Priorität. x bedeutet zulässig für diesen Typ..

¹⁾ In Python 3 ist das Ergebnis der Division / vom Typ »float«.

²⁾ $q = x // y$ ist die Ganzzahldivision (Division mit Rest).

$r = x \% y$ nennt man die Modulo-Operation und diese liefert den Rest r und zwar so, dass

r hat dasselbe Vorzeichen wie y. Das Vorzeichen von q ergibt sich wie bei der normalen Division. q ist eine Ganzzahl z.B. 4 oder 4.0: Es wird in Richtung $-\infty$ gerundet.

Die ganzzahlige Division und Modulo ergeben int-Werte, wenn beide Operanden den Typ »int« haben, aber einen float-Wert, wenn mindestens einer der Operanden den Typ »float« hat.

³⁾ Vergleichsoperatoren dürfen verkettet werden: $x < y < z$ ist im Ergebnis identisch mit $(x < y) \text{ and } (y < z)$, nur in der Ausführung etwas schneller.

7. Vergleichsoperatoren

Operator	Beschreibung
<code>X < Y</code>	echt kleiner als
<code>X <= Y</code>	kleiner oder gleich
<code>X > Y</code>	echt größer als
<code>X >= Y</code>	größer oder gleich
<code>X == Y</code>	gleicher Wert
<code>X != Y</code>	Ungleicher Wert
<code>X is Y</code>	Gleiches Objekt (Variable)
<code>X is not Y</code>	Negierte Objektgleichheit

8. Funktionen zur Typwandlung

Typ (Kürzel)	Konvertierungsfunktionen	Anmerkungen
Integer (int)	int(O[, basis]) ord(C)	Ist O ein String, muss O ein gültiges Literal für Integer sein, optional kann eine Basis (beliebiger Integer) angegeben werden. C ist ein String der Länge 1.
Flaot (float)	float(O)	Ist O ein String, muss O ein gültiges Literal für Float sein.
Complex (complex)	complex(O)	Ist O ein String, so muss O ein gültiges Literal für complex, float oder integer sein.
Boolean (bool)	bool(O)	Jeder Wert $\neq 0$ bei numerischen Datentypen oder dem „leeren String“ liefert 'TRUE'
String (str)	str(O) repr(O) oder <code>`I`</code> (Backticks) chr(I) hex(I) oct(I)	liefert ein pretty-print (String) von O sehr ähnlich wie str(O), erzeugt ein von eval() auswertbaren String, wird später behandelt Parameter muss ein Integer im zulässigen Wertebereich sein, liefern alle Strings

9. Häufig benutzte Operationen auf Sequenz-Typen

Alle Sequenztypen (list, tuple, range, aber auch str, bytes und bytearray) unterstützen die folgenden Operationen. Die Reihenfolge der Operationen gibt ihre Auswertepriorität an (oben die höchste).

s und **t** sind Sequenztypen desselben Typs, *n*, *i*, *j* and *k* sind integers und *x* ist ein beliebiger Typ.

Operation	Result	Notes
<code>s.count(x)</code>	total number of occurrences of x in s	
<code>s.index(x[, i[, j]])</code>	index of the first occurrence of x in s (at or after index <i>i</i> and before index <i>j</i>)	(8)
<code>max(s)</code>	largest item of s	
<code>min(s)</code>	smallest item of s	
<code>len(s)</code>	length of s	
<code>s[i:j:k]</code>	slice of s from <i>i</i> to <i>j</i> with step <i>k</i>	(3)(5)
<code>s[i:j]</code>	slice of s from <i>i</i> to <i>j</i>	(3)(4)
<code>s[i]</code>	<i>i</i> th item of s , origin 0	(3)
<code>s * n</code> or <code>n * s</code>	<i>n</i> shallow copies of s concatenated	(2)(7)
<code>s + t</code>	the concatenation of s and t	(6)(7)
<code>x not in s</code>	False if an item of s is equal to x , else True	(1)
<code>x in s</code>	True if an item of s is equal to x , else False	(1)

Anmerkungen (Notes):

(1) Generell operiert diese Operation nur auf einzelnen Elementen, bei str, bytes und bytearray aber auch auf Subsequenzen:

```
>>> "gg" in "eggs"
True
```

(2) Werte von *n* kleiner als 0 werden als 0 behandelt (also wie eine leere Sequenz). Beachten Sie, die Kopien sind shallow (= flach), d.h. geschachtelte Strukturen werden nicht kopiert.

(3) Wenn *i* und *j* negativ sind, sind diese relativ zum Ende der Sequenz.

(4/5) Es gelten die Slicing-Regeln.

(6) Die Konkatenation immutable Sequenztypen erzeugt immer ein neues Objekt. Das heißt, das Erzeugen von Sequenzen durch wiederholte Konkatenation ergibt quadratische Laufzeiten.

(7) Für den Range-Typ nicht implementiert.

(8) `index` generiert einen `ValueError` when **x** nicht in **s** gefunden wird.

Die Typen str und bytearray haben noch diverse zusätzliche Typ-spezifische Operationen, siehe deren Dokumentation: für Strings: <http://docs.python.org/3/library/stdtypes.html#textseq> und für Binärtypen: <http://docs.python.org/3/library/stdtypes.html#binaryseq>.

Mutable Sequence Typen (list, bytearray) unterstützen auch verändernde Operationen dieses Typs:

Operation	Result	Notes
<code>s[i] = x</code>	item i of s is replaced by x	
<code>s[i:j] = t</code>	slice of s from i to j is replaced by the contents of the iterable t	
<code>del s[i:j]</code>	same as <code>s[i:j] = []</code>	
<code>s[i:j:k] = t</code>	the elements of <code>s[i:j:k]</code> are replaced by those of t	
<code>del s[i:j:k]</code>	removes the elements of <code>s[i:j:k]</code> from the list	
<code>s.append(x)</code>	appends x to the end of the sequence (same as <code>s[len(s):len(s)] = [x]</code>)	
<code>s.clear()</code>	removes all items from s (same as <code>del s[:]</code>)	
<code>s.copy()</code>	creates a shallow copy of s (same as <code>s[:]</code>)	
<code>s.extend(t)</code>	extends s with the contents of t (same as <code>s[len(s):len(s)] = t</code>)	
<code>s.insert(i, x)</code>	inserts x into s at the index given by i (same as <code>s[i:i] = [x]</code>)	
<code>s.pop([i])</code>	retrieves the item at i and also removes it from s	(1)
<code>s.remove(x)</code>	remove the first item from s where <code>s[i] == x</code>	(2)
<code>s.reverse()</code>	reverses the items of s in place	

Anmerkungen (Notes):

(1) Der optionale Argumentwert ist -1, so dass der letzte Eintrag zurückgegeben und gelöscht wird.

(2) `remove` generiert einen `ValueError` when `x` nicht in **s** gefunden wird.