Programmierhandbuch

(Style Guide)

PRG 1 / EPR / PS2

für (Python 3.x)

Version 2017/18

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, bis zu 25%. 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, Ihr Detlef Krömker

Inhalt

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 Zeile in der nur `"""` stehen.

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

2. Nach dem **docstring** stehen die import-Statements in folgender Reihenfolge (je eine Zeile pro import):

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

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

Tipp: Erstellen Sie sich Ihr persönliches Template. Diese Werte werden den angegebenen internen Variablen zugewiesen und können damit in dem Modul abgefragt werden.

So sollte ein Modul aussehen. Sie können dies als Rohversion Ihres Templates benutzen:

```
----snip----
"""Docstring: A very short sentence explaining the function. < 79 characters."""

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

from math import sqrt, log, e  #an example for builtins
#from numpy import array       #another example for third party module
#import meinmodul              #example for your own module

__author__ = "123456: John Cleese, 654321: Terry Gilliam"  #put your data here
__copyright__ = "Copyright 2017/2018 – EPR-Goethe-Uni"
__credits__ = "If you would like to thank somebody \
               i.e. an other student for her code or leave it out"
__email__ = "your email address"

def yourroutine_1():
    """Docstring: Every subroutine and function has a docstring."""

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


def yourfunction_2():
    """Docstring for yourfunction_2(). """

    pass  #To be programed later.


def main():
    print('''Hier könnte z.B. Initialisierungscode für das Programm main
         stehen, wenn es als selbstständiges Programm aufgerufen wird.''')

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 L) 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 sind erlaubt, wie

`curr`	für	current,
`eof`	für	end of file,
`eol`	für	nd of line,
`in`	für	input,
`max`	für	maximum,
`min`	für	minimum,
`out`	für	output,
`num oder no`	für	number,
`prev`	für	previous,
`ref`	für	reference,
str	für	string,
`struct`	für	structure,
`temp oder tmp`	für	temporar.

Schreibweisen von Namen:

Typ	**Konvention**	**Beispiel**
Variable	nomen_mit_unterstrich - Kleinbuchstaben	`curr_index`
Funktion	aktion_mit_unterstrich - Kleinbuchstaben	`find_all`
Prozedur	CamelCase	`InverseMapping`
Konstante	NOMEN_ALLES_GROSS - Großbuchstaben	`ALLOWED_RANGE`
Globale Variable	gCamelCase (mit führendem g), sehr selten nutzen Globale Variablen sind bei uns VERBOTEN!	`gGeneralRefNo`
Modul Bezeichnung	Kleinbuchstaben (ggf. mit_Unterstrich)	`unit_test`
Paket Bezeichnung	NUR Kleinbuchstaben (KEIN Unterstrich)	`pygame`

Für später:

Klassen	CamelCase (wie Prozeduren) Immer `self` als erstes Argument für Instanz-Methoden benutzen Und `cls` als erstes Argument für Klassen-Methoden.
Exceptions	wie Klassen mit dem Suffix `Error` (wenn es so ist)

Es gibt folgende Sonderformen von Bezeichnern/Namen:

- `_single_leading_underscore` : Dies ist ein „schwacher“ Hinweis unter Programmierer*innen, das 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 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 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 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 und das folgende # 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
				
				x
					
					x	

`+x, -x,` `~x`	Einstellige Operatoren Invertiere x		x	x x		
`x ** y`	Exponential-Bildung x^y (Achtung: rechts-assoziativ)		x	x		x

`x * y`
`x / y`
`x % y`
 	
`x // y`
	Multiplikation (Wiederholung)
	Division
	Modulo (-Division) = (Ganzzahliger) Rest
	Restlose 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`** [1])
	Test auf Identität
	Tests auf Enthaltensein in Sequenzen
		x
		
		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 höchster Priorität.
[1]) Vergleichsoperatoren dürfen verkettet werden: `x < y < z` ist im Ergebnis identisch mit `(x < y) and (y < z)`, nur in der Ausführung etwas schneller.

[2]) `x // y` schneidet den Rest bei einer Division ab (in Python bis Version 2.6 ist dies bei einer Ganzzahldivision identisch zu `x / y`. Ab Python 3 wird im Falle eines Restes ≠ 0 automatisch in eine Gleitpunktzahl konvertiert.

7. Vergleichsoperatoren

Operator	Beschreibung
`X < Y`	echt kleiner als
`X <= Y`	kleiner oder gleich
`X > Y`	echt größer als
`X >= Y`	größer oder gleich
`X == Y`	gleicher Wert
`X != Y`	Ungleicher Wert
`X is Y`	Gleiches Objekt (Variable)
`X is not Y`	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 ≠ 0 bei numerischen Datentypen oder dem „leeren String“ liefert 'TRUE'
String (str)	**`str(O)`** **`repr(O)`**oder **`` `I` ``** (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
`s.count(x)`	total number of occurences of `x` in `s`	
`s.index(x[, i[, j]])`	index of the first occurence of `x` in `s` (at or after index `i` and before index `j`)	(8)
`max(s)`	largest item of `s`	
`min(s)`	smallest item of `s`	
`len(s)`	length of `s`	
`s[i:j:k]`	slice of `s` from `i` to `j` with step `k`	(3)(5)
`s[i:j]`	slice of `s` from `i` to `j`	(3)(4)
`s[i]`	ith item of `s`, origin 0	(3)
`s * n` or `n * s`	`n` shallow copies of `s` concatenated	(2)(7)
`s + t`	the concatenation of `s` and `t`	(6)(7)
`x not in s`	`False` if an item of `s` is equal to `x`, else `True`	(1)
`x in s`	`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 byrearray aber auch auf Subseuenzen:

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

(2) Werte von *n* kleiner als `0` werden als `0` behandelt (also wie eine leere Sequenz). Beachten Sie, doe Kopien sind shallow (= flach), d.h. geschachtelte Strukturen warden 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
`s[i] = x`	item `i` of `s` is replaced by `x`	
`s[i:j] = t`	slice of `s` from `i` to `j` is replaced by the contents of the iterable `t`	
`del s[i:j]`	same as `s[i:j] = []`	
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of `t`	
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list	
`s.append(x)`	appends `x` to the end of the sequence (same as `s[len(s):len(s)] = [x]`)	
`s.clear()`	removes all items from `s` (same as `del s[:]`)	
`s.copy()`	creates a shallow copy of `s` (same as `s[:]`)	
`s.extend(t)`	extends `s` with the contents of `t` (same as `s[len(s):len(s)] = t`)	
`s.insert(i, x)`	inserts `x` into `s` at the index given by `i` (same as `s[i:i] = [x]`)	
`s.pop([i])`	retrieves the item at `i` and also removes it from `s`	(1)
`s.remove(x)`	remove the first item from `s` where `s[i] == x`	(2)
`s.reverse()`	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.