Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Ausarbeitung

PEPs - Python Enhancement Proposals

vorgelegt von

Noah Fuhst

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang: Informatik
Matrikelnummer: 7158991

Betreuer: Michael Kuhn

Hamburg, 24.06.2019

Abstract

In dieser Ausarbeitung wird eine generelle Übersicht über die PEPs (Python Enhancement Proposals) gegeben. Dabei wird herausgestellt, was PEPs eigentlich sind, welche Arten es gibt, und einige wichtige PEPs hervorgehoben. Dabei wird insbesondere PEP 8 betrachtet, welches Quelltextkonventionen für Python beinhält, ebenso wird ein kurzer Crashkurs zu PEP 8-konformer Programmierung geboten. Weiterhin werden Tools zum einfacheren Einhalten von PEPs vorgestellt sowie der PEP Erstellungsprozess beleuchtet.

Inhalt

1 Einleitung

Dieses Kapitel dient dazu, eine kurze generelle Übersicht über Python sowie dessen Entwicklungsprozess herzustellen. Dies ist unter anderem wichtig dafür, dass verstanden werden kann, warum PEPs in Python wichtig sind.

2017 hat Python den Platz der beliebtesten Programmiersprache im IEEE-Ranking eingenommen.(vgl. [Cas17]) Dies liegt sicherlich nicht zuletzt an der Philosophie und dem Anspruch der Sprache, einen gut lesbaren Code zu erzeugen. Diese wird unter anderem in verschiedenen PEPs ("Python Enhancement Proposals") festgehalten, in diesem Fall zum Beispiel PEP 20 – "The Zen of Python" [Pet04] und PEP 8 – "Style Guide for Python Code" [vRBWNC01]. Dabei gibt es noch viele weitere PEPs, welche allesamt verschiedene Anwendungszwecke haben. Doch warum hat Python dieses System der PEPs recht zentral in der eigenen Entwicklung eingebaut? Sehen wir uns dafür einmal die Geschichte von Python an.

Python wurde von Guido van Rossum, einem holländischen Softwareentwickler, der bis 2018 den Titel des Benevolent Dictator for Life für Python innehielt, entwickelt. Bis heute gibt es mehrere verschiedene Versionen von Python, welche untereinander nicht kompatibel sind:

- Python 0.9 und 1, die erste veröffentlichte Version von Python, 1991
- Python 2, wird auch heute noch verwendet, Python 2.1 war der erste Release unter der PSF ("Python Software Foundation")(vgl. [Fou19])
- Python 3, der aktuelle Major-Release, 2008(vgl. [Fou08])

Python war dabei seit dem ersten Release 1991 eine Open-Source-Software. Dadurch beteiligen sich natürlich viele Entwickler an der Verbesserung und Erweiterung der Sprache. Entsprechend benötigt man einen standardisierten Weg für Kommunikation, Absprachen, oder zum Vorschlagen von neuen Features für die Sprache. Außerdem wird eine Dokumentationsmöglichkeit für die Arbeit an Python benötigt, ebenso wie ein Weg, Feedback von der Community einzuholen. Entsprechend sind die PEPs entstanden, und bis heute ein zentraler Bestandteil für die Entwicklung in und an Python. Für Beginner sind dabei die oben genannten PEPs 8 und 20 zu empfehlen, da diese Anweisungen geben, die dabei helfen nach Pythons Philosophie konform zu programmieren.

2 Hauptteil

Dieses Kapitel dient dazu, einen generellen Überblick über PEPs sowie ihre Struktur und Anwendungszwecke darzustellen. Ebenso werden einige wichtige PEPs vorgestellt, welche für die Arbeit in Python essentiell sind, sowie einige Tools, welche die Einhaltung der PEPs einfacher machen. Außerdem sehen wir uns an, wie ein PEP überhaupt erstellt, angenommen und implementiert wird.

2.1 Was sind PEPs?

Die offizielle Definition von PEPs besagt: "PEP stands for Python Enhancement Proposal. A PEP is a design document providing Information to the Python community, or describing a new feature for Python or its processes or environment."[Cog00]. Entsprechend stellen PEPs Informationen über das Python-Umfeld bereit (z.B. in PEP 101 - "Doing Python Releases 101" [vR01a] oder PEP 10 - "Voting Guidelines" [War02]). Es gibt aber auch PEPs, welche beim Coding helfen können, z.B. indem sie Quelltextkonventionen (PEP 8) oder Standards zur Verwendung bestimmter Keywords geben (z.B. PEP 484 - "Type Hints" [vRJL14]). Es gibt jedoch noch viele weitere Arten von PEPs, u.a. auch PEPs über PEPs (z.B. PEP 1 - "PEP Purpose and Guidelines" [Cog00]).

Generell gilt, dass die meisten PEPs freiwillig verwendet werden können - jedoch ist die Verwendung zu empfehlen, um einen gewissen Standard in dem geschriebenen Code zu gewährleisten. Es gibt allerdings auch einige PEPs, deren Einhaltung außer in wenigen Ausnahmefällen verpflichtend ist. Diese werden später in dieser Ausarbeitung weiter behandelt. Übrigens ist das Prinzip der Enhancement Proposals nicht auf Python beschränkt, so gibt es z.B. auch JEPs - "Java Enhancement Proposals" [Rei11], welche denselben Zweck für Java erfüllen - dabei jedoch keine so zentrale Rolle einnehmen wie die PEPs bei Python.

2.2 Arten von PEPs

PEPs werden nach PEP 1 in drei Arten unterschieden. Zunächst gibt es die PEPs, welche über neue Features oder Implementationen in Python berichten. Sie dienen der Dokumentation der Entwicklung, u.a. um Bugfixes zu listen, Vorschläge der Community aufzunehmen, oder auch Release Notes einer neuen Python Version zu veröffentlichen (vgl. [Cog00]). Ein Beispiel wäre dabei PEP 494, die Python 3.6 Release Schedule.

Der zweite Typ PEP behandelt Informierende PEPs. Diese hängen meistens mit der Programmiersprache selbst zusammen, und bieten Richtlinien zur Anwendung von bestehenden Implementationen (oft Quelltextkonventionen). Die Einhaltung ist dabei stets freiwillig, allerdings durchaus nahezulegen, da so der eigene Code für alle lesbarer wird - was mit der Philosophie von Python einhergeht. Ein Beispiel für ein Informierendes PEP ist PEP 257 - "Docstring Conventions" [vR01b].

Der dritte und letzte Typ PEP ist das Prozessbasierte PEP. Diese Art PEPs behandelt Prozesse um Python herum. Zu ihnen gehören auch die META-PEPs, das sind einige besonders wichtige PEPs, welche auch im PEP 0 (dem Index aller PEPs) [PD00] explizit gelistet werden. Anders als bei den anderen PEP Arten ist die Einhaltung der Prozessbasierten PEPs verpflichtend. Ein Beispiel für ein META-PEP ist PEP 8 - "Style Guide for Python Code" [vRBWNC01].

2.3 Wichtige PEPs

2.3.1 Übersicht

Nun werden wir uns einige wichtige PEPs ansehen. Zunächst betrachten wir PEP 0, den Index aller PEPs, gefolgt von PEP 1 - "PEP Purpose and Guidelines" [Cog00]. Das wohl wichtigste PEP im Bezug auf die direkte Entwicklung in Python ist vermutlich PEP 8 - "Style Guide for Python Code" [vRBWNC01], welches Quelltextkonventionen für Python beinhält. Entsprechend werde ich dieses PEP sehr ausführlich behandeln. Zum Schluss schauen wir uns PEP 20 - "The Zen of Python" [Pet04] an, welches die Philosophie von Python näherbringt, sowie einige generelle Designtipps gibt.

2.3.2 PEP 0 – Index of Python Enhancement Proposals

PEP 0 dient als Index aller PEPs. Es werden alle PEPs sowie deren Autoren und Implementationszustände aufgelistet. Dabei können PEPs bereits vollständig implementiert sein, sich aktuell noch in der Implementation befinden, aber auch abgelehnt oder mittlerweile überholt worden sein. Weiterhin lässt sich die komplette PEP History über GitHub im PEP 0 abrufen. PEP 0 ist entsprechend die erste Anlaufstelle, falls man nach PEPs zu bestimmten Themen sucht.

2.3.3 PEP 1 – PEP Purpose and Guidelines

PEP 1 kann als Fundament für alle anderen PEPs gesehen werden. Es enthält grundlegende Regeln zur Erstellung und Wartung von PEPs, ebenso wie den Genehmigungsprozess der PEPs. Außerdem werden die Adressaten der PEPs genannt, welche hauptsächlich die Core Developers des CPython Interpreters sind, sowie die Mitglieder des Steering Councils, aber auch andere Python Entwickler (vgl. [Cog00]). Weiterhin gibt es auch

Tipps, um vorgeschlagene PEPs möglichst erfolgreich zu machen, ebenso wie Richtlinien zum Design der PEPs.

2.3.4 PEP 8 – Style Guide for Python Code

PEP 8 ist das wahrscheinlich wichtigste PEP, welches direkt mit der Entwicklung in Python zusammenhängt. Es enthält Quelltextkonventionen, welche das Ziel haben, gut lesbaren Code zu erstellen (und daher mit der Philosophie von Python einhergehen). Es handelt sich bei PEP 8 um eines der META-PEPs, sodass die Verwendung verpflichtend ist. Die in PEP 8 gefundenen Quelltextkonventionen sind nur auf Python ausgelegt, generellere Designhinweise können aber in PEP 20 gefunden werden.

PEP 8 Crashkurs

Da PEP 8 eine so zentrale Stellung in der Entwicklung von Python hat, hier ein kleiner Crashkurs zur Entwicklung mit PEP 8-konformem Code. Dieser Crashkurs ist dabei zugeschnitten auf SE1-Wissen, da dies zum Zeitpunkt meines Vortrages das bei allen Zuhörern vorhandene Mindestwissen sein sollte (allerdings ist PEP 8 deutlich umfangreicher). Entsprechend ist anzumerken, dass die Einrückung in Python nicht optional ist. Anders als z.B. Java arbeitet Python nicht mit geschweiften Klammern, um Codeblöcke identifizieren zu können, sondern mit dem Whitespace (also dem Einrückungslevel). Insofern kann eine falsche Einrückung zu einer falschen Ausführung des Codes führen. Außerdem gilt, dass Konsistenz in einem Programm durchaus vor der strikten Einhaltung von PEP 8 geht.

Pro Einrücklevel sind entsprechend 4 Leerzeichen zu verwenden (vgl. [vRBWNC01]). Tabulatoren sind dabei zu vermeiden, da sie je nach IDE oder OS in der Breite variieren können. Ebenso ist ein Zeichenmaximum von 79 Zeichen pro Zeile einzustellen, damit auch in kleineren Fenstern (z.B. wenn man mehrere Fenster mit mehreren Python-Dateien nebeneinander offen hat) der Code sowie Kommentare gut lesbar bleiben (vgl. [vRBWNC01]). Eine solche Zeichenbeschränkung kann allerdings bei den meisten IDEs und Codeeditoren eingestellt werden, welche dann automatisch die Zeilenumbrüche setzen.

```
1 def method(self):
2     a = 1
3     if a == 1
4         b = 2
```

Listing 2.1: Vier Leerzeichen pro Einrücklevel

Bei Zeilenumbrüchen sind sogenannte "Hanging Indents" [vRBWNC01] zu verwenden, sollte man in einer Zeile das Zeichenlimit ausreizen. Ein Hanging Indent bedeutet, folgende Zeilen desselben Absatzes bis zur ersten öffnenden Klammer eines Statements (in der ersten Zeile) einzurücken. Alternativ kann man auch die folgenden Zeilen, die zu demselben Absatz gehören, eine Stufe weiter einrücken als den folgenden Codeblock (vgl. [vRBWNC01]).

```
1 def methode(self, variable1, variable2, variable3,
2             variable4, variable5, variable6):
3     int a = 1
4     #...
```

Listing 2.2: Beispiel für einen Hanging Indent

Weiterhin sollten binäre Operatoren immer nach Zeilenumbrüchen gesetzt werden (vgl. [vRBWNC01]), um mehrzeilige Rechnungen lesbarer zu machen. Warum, zeigt folgendes Beispiel:

```
1 int i = (var1 + var2 +
2         var3 + var4 -
3         var5 - var6 *
4         var7)
```

Listing 2.3: Operatoren vor Zeilenumbrüchen - schlecht

Es ist ersichtlich, was passiert, wenn Operatoren vor Zeilenumbrüchen gesetzt werden. Oft muss man wieder in die vorige Zeile gucken, um zu sehen, welche Operation nun mit einer Variable zu verwenden ist. Besser ist dort folgendes Beispiel:

```
1 int i = (var1 + var2
2         + var3 + var4
3         - var5 - var6
4         * var7)
```

Listing 2.4: Operatoren nach Zeilenumbrüchen - gut

Fortführend sind zwei Leerzeilen vor Klassendeklarationen und eine Leerzeile vor Methodendeklarationen zu setzen [vRBWNC01], um eine bessere Übersicht über die Blockstruktur des eigenen Codes zu gewährleisten. Konstante Werte sind in Python komplett in Großbuchstaben und mit Unterstrichen zu schreiben (vgl. [vRBWNC01]. Dies ist wichtig, da es anders als z.B. in Java kein Schlüsselwort wie "final" gibt, um Konstanten als solche für den Interpreter zu kennzeichnen. Entsprechend gibt es diese Konvention, um anderen Entwicklern anzuzeigen, dass eine Konstante nicht verändert werden soll.

```
1
2
3 class New:
4
5     def rechnen(self):
6         KONSTANTER_WERT = 30
```

Listing 2.5: Freizeilen und Konstanten

Imports sind stets in eigenen Zeilen unterzubringen (vgl. [vRBWNC01], um so eine ansehnliche Auflistung aller importierten Module zu bieten, außerdem sind sie nach First-, Second- und Third-Party Imports zu gruppieren(vgl. [vRBWNC01], dies ist allerdings eigentlich nicht relevant für SE1-Wissen. Weiterhin sollen Leerzeichen um alle Operatoren gesetzt werden (egal ob Vergleich, Zuweisung, boolisch...)(vgl. [vRBWNC01]).

```
1 import math
2 import cmath
3     def method(self, c, d):
4             a = 1
5             if(b == True)...
6             j and k
```

Listing 2.6: Imports in eigenen Zeilen und Lücken um Operatoren

Für Kommentare gelten ebenfalls Regeln nach PEP 8, u.a. sind diese stets auf Englisch zu verfassen (vgl. [vRBWNC01]), da Softwareentwicklung sehr international ist. Außerdem sind Blockkommentare Zeilenkommentaren vorzuziehen, da sie zunächst einmal einen angenehmeren Lesefluss zulassen (sollten sie über mehrere Zeilen laufen), und zweitens nicht so sehr unter dem oben erwähnten Zeichenlimit von 79 Zeichen leiden, da es bei ihnen kein Problem darstellt, wenn sie über mehrere Zeilen gehen.

```
1 #Blockcomments are superior to Linecomments. That's
2 #because they are often more readable over multiple
3 #lines. Also they often are less redundant.
4 def KommentarMethode(self):
5     a += 1 #increments a
```

Listing 2.7: Beispiel für Kommentare

Bezüglich Namenskonventionen für Klassen und Variablen gilt: Klassen werden nach der CamelCase/CapWords-Namenskonvention bennant, Variablen werden kleingeschrieben und mit Unterstrichen getrennt (vgl. [vRBWNC01]). Zu guter Letzt ist am Ende jeder Python-Datei eine Leerzeile zu setzen, um das Ende des Programms zu signalisieren (vgl. [vRBWNC01]).

```
1 class HelloThere:
2     def methode(self):
3         neue_variable = "63n3r4l K3n0b1"
4
```

Listing 2.8: Namenskonventionen und Leerzeile am Ende

Dies war nun ein kurzer Crashkurs zu PEP 8. PEP 8 beinhält allerdings noch viele weitere Konventionen, für ernsthafte Entwicklung in Python ist dieser Crashkurs entsprechend nur bedingt zu empfehlen. Dieser Crashkurs bezieht sich hauptsächlich auf Wissen aus SE1 (Softwareentwicklung 1), da das Vorstellen jeder einzelnen Konvention hier den Rahmen sprengen würde.

2.3.5 PEP 20 – The Zen of Python

Bei PEP 20 ist auf den ersten Blick ersichtlich, dass es sich sehr stark von den anderen PEPs unterscheidet. Zum einen ist es sehr kurz, zum anderen ist es sehr allgemein (nicht nur auf Python beschränkt) formuliert. PEP 20 soll dem Leser die Philosophie von Python näherbringen, gut lesbaren Code zu schreiben. PEP 20 gibt grundlegende Codedesigntipps, wie z.B. "Explicit is better than implicit" [Pet04], welche durch den Programmierer ausgelegt werden können. So kann man entsprechend sagen, dass z.B. eine explizite Variablenbenennung besser ist als eine implizite, welche nur durch die Operationen mit der Variable erahnen lässt welchen Zweck eben diese erfüllt. Anders als PEP 8, handelt es sich bei PEP 20 um ein informierendes PEP (vgl. [PD00]), es gibt also keine Verpflichtung zur Einhaltung.

2.4 Tools zum Einhalten der PEPs

Das Einhalten von Richtlinien und Regeln, wie sie z.B. von PEP 8 vorgeschrieben werden, bedeutet natürlich auch einen nicht zu unterschätzenden Mehraufwand beim Lernen einer neuen Programmiersprache (oder auch für erfahrene Entwickler). Entsprechend gibt es einige Tools, welche das Einhalten der PEPs erleichtern. Meistens behandeln diese Tools PEP 8, jedoch kann deren Funktionalität auch darüber hinausgehen, z.B. hat das Tool "pylint" [Pyl] einen eingebauten UML-Generator. Diese Programme überprüfen den Code, während er geschrieben wird und passen die Formatierung automatisch an die PEP 8 Konventionen an. Alternativ passiert die Formatierung automatisch mit jedem Speichern des Programms. Teilweise sind solche Tools auch bereits in IDE's eingebaut, so hat z.B. das IDE PyCharm eine eingebaute Überprüfung bzgl. PEP 8. Beispiele für solche Tools sind pylint [Pyl], autopep8 [Hat19] oder flake8 [fla16].

2.5 Erstellung und Aufnahme neuer PEPs

Generell darf jeder Python Entwickler ein PEP vorschlagen. Jedoch benötigt man dafür die Hilfe eines sogenannten "Core Developers" [Cog00], wecher einem beim Erstellprozess zur Seite steht. Hat man einen solchen gefunden, erstellt man sein PEP als .rst-Datei und erstellt für diese einen Pull-Request im Python-GitHub. Danach wird die Datei von anderen PEP-Autoren bezüglich Struktur und Inhalt überprüft, erst danach bekommt das PEP eine eigene Nummer. Zu diesem Zeitpunkt sollte dann bereits eine Beispielimplementation vorliegen, um zu zeigen, dass die Idee des PEPs auch umsetzbar ist. Der nächste Schritt ist, die .rst-Datei sowie die Beispielimplementation an die "python-dev"-Mailingliste [Cog00] zu schicken, um so Feedback einzuholen. Danach folgt die abschließende Prüfung durch die Core Developers oder den Steering Council (vgl. [Cog00]). Damit das PEP akzeptiert wird, ist unter anderem wichtig, dass es eine klare und vollständige Beschreibung des Vorschlags enthält und keine unnötige Verlangsamung oder Beschäftigung des Interpreters hervorgerufen wird. Außerdem muss eine erkennbare Verbesserung für Python im Vorschlag enthalten sein (vgl. [Cog00]. Ist das PEP angenommen worden, muss die Beispielimplementation vervollständigt und implementiert werden, bevor es in den Status "final" übergehen kann (vgl. [Cog00]).

3 Zusammenfassung

PEPs spielen für Python eine zentrale Rolle. Sie beinhalten Regeln und Richtlinien für alles was Python betrifft, und könnten so ebenfalls als sehr ausführliches FAQ zu verschiedensten Themengebieten betrachtet werden. Dabei sind nur einige wenige PEPs verpflichtend einzuhalten (die sog. Meta-PEPs), bei informierenden PEPs ist die Nutzung stets freigestellt.

Das wichtigste PEP für die direkte Entwicklung in Python ist wohl PEP 8, welches Quelltextkonventionen für Python enthält. PEP 8 ist ebenfalls verpflichtend einzuhalten, da es eines der Meta-PEPs ist. Die Einhaltung der PEPs lässt sich dabei mithilfe von einigen Tools, wie z.B. pylint, flake8 oder autopep8 deutlich vereinfachen, sollte noch kein solches Tool im IDE oder Codeeditor vorhanden sein. Für allgemeinere Designhinweise (auch zu anderen Programmiersprachen) kann PEP 20 konsultiert werden, welches die Philosophie von Python näher erläutert, um so das Schreiben von gut lesbarem Code zu vereinfachen.

Literatur

[Cas17] Stephen Cass. Article: The 2017 Top Programming Languages. 07 2017. Letzter Zugriff: 05.05.2019.

[Cog00] Barry Warsaw; Jeremy Hylton; David Goodger; Nick Coghlan. PEP 1 – PEP Purpose and Guidelines. 06 2000. Letzter Zugriff: 14.04.2019.

[fla16] Flake8: Your Tool For Style Guide Enforcement. 2016. Letzter Zugriff: 14.04.2019.

[Fou08] Python Software Foundation. Python 3.0 Release. 12 2008. Letzter Zugriff: 14.04.2019.

[Fou19] Python Software Foundation. History and License – Python 3.7.3 Documentation. 04 2019. Letzter Zugriff: 14.04.2019.

[Hat19] Hideo Hattori. autopep8 1.4.4. 04 2019. Letzter Zugriff: 14.04.2019.

[PD00] Python-Dev. PEP 0 – Index of Python Enhancement Proposals. 07 2000. Letzter Zugriff: 14.04.2019.

[Pet04] Tim Peters. PEP 20 – The Zen of Python. 08 2004. Letzter Zugriff: 05.05.2019.

[Pyl] Pylint - code Analysis for Python. Letzter Zugriff: 14.04.2019.

[Rei11] Mark Reinhold. JEP 0: JEP Index. 08 2011. Letzter Zugriff: 08.06.2019.

[vR01a] Barry Warsaw; Guido van Rossum. PEP 101 – Doing Python Releases 101. 08 2001. Letzter Zugriff: 12.06.2019.

[vR01b] David Goodger; Guido van Rossum. PEP 257 – Docstring Conventions. 05 2001. Letzter Zugriff: 14.04.2019.

[vRBWNC01] Guido van Rossum; Barry Warsaw; Nick Coghlan. PEP 8 – Style Guide for Python Code. 07 2001. Letzter Zugriff: 12.06.2019.

[vRJL14] Guido van Rossum; Jukka Lehtosalo; Łukasz Langa. PEP 484 – Type Hints. 09 2014. Letzter Zugriff: 08.06.2019.

[War02] Barry Warsaw. PEP 10 – Voting Guidelines. 03 2002. Letzter Zugriff: 14.04.2019.

Liste aller verwendeten Quellen

1. Stephen Cass. Article: The 2017 Top Programming Languages. 07 2017. Letzter Zugriff: 05.05.2019.
2. Barry Warsaw; Jeremy Hylton; David Goodger; Nick Coghlan. PEP 1 – PEP Purpose and Guidelines. 06 2000. Letzter Zugriff: 14.04.2019.
3. Flake8: Your Tool For Style Guide Enforcement. 2016. Letzter Zugriff: 14.04.2019.
4. Python Software Foundation. Python 3.0 Release. 12 2008. Letzter Zugriff: 14.04.2019.
5. Python Software Foundation. History and License – Python 3.7.3 Documentation. 04 2019. Letzter Zugriff: 14.04.2019.
6. Hideo Hattori. autopep8 1.4.4. 04 2019. Letzter Zugriff: 14.04.2019.
7. Python-Dev. PEP 0 — Index of Python Enhancement Proposals. 07 2000. Letzter Zugriff: 14.04.2019.
8. Tim Peters. PEP 20 — The Zen of Python. 08 2004. Letzter Zugriff: 05.05.2019.
9. Pylint - code Analysis For Python. Letzter Zugriff: 14.04.2019.
10. Mark Reinhold. JEP 0: JEP Index. 08 2011. Letzter Zugriff: 08.06.2019.
11. Barry Warsaw; Guido van Rossum. PEP 101 — Doing Python Releases 101. 08 2001. Letzter Zugriff: 12.06.2019.
12. David Goodger; Guido van Rossum. Pep 257 — Docstring Conventions. 05 2001. Letzter Zugriff: 14.04.2019.
13. Guido van Rossum; Barry Warsaw; Nick Coghlan. PEP 8 — Style Guide for Python Code. 07 2001. Letzter Zugriff: 12.06.2019.
14. Guido van Rossum; Jukka Lehtosalo; Łukasz Langa. Pep 484 – Type Hints. 09 2014. Letzter Zugriff: 08.06.2019.
15. Barry Warsaw. Pep 10 – Voting Guidelines. 03 2002. Letzter Zugriff: 14.04.2019.
16. Python Software Foundation. General Python FAQ. 04 2019. Letzter Zugriff: 14.04.2019
17. Python Software Foundation. What is Python? Executive Summary. https://www.python.org/doc/essays/blurb/, Letzter Zugriff: 14.04.2019
18. o.V. Guido van Rossum. https://de.wikipedia.org/wiki/Guido_van_Rossum, Letzter Zugriff: 14.04.2019

19. o.V. Benevolent Dictator for Life. https://de.wikipedia.org/wiki/Benevolent_-Dictator_for_Life, Letzter Zugriff: 14.04.2019

Appendix

Liste der Listings