PCL II – Loesungen Uebung 01
Linus Manser (lmanser, 13-791-132) und Roland Benz (rolben, 97-923-163)

1 Coding-Stil und Programmierprinzipien

1.1

Lies dir den offiziellen Python Style-Guide (https://www.python.org/dev/peps/pep-0008/) und die Google-Ergänzungen dazu (https://google.github.io/styleguide/pyguide.html#Python_Style_Rules) durch. Versuche, deinen Code wenn möglich danach zu gestalten. Alle Übungen in PCL II werden auch nach Coding-Stil bewertet. Du musst nicht unbedingt in allen Einzelheiten dem PEP08 folgen, aber es ist ein guter Ansatzpunkt.

Zusammenfassung der wichtigsten Punkte:

BE CONSISTENT.

- **Main**

  In Python, `pydoc` as well as unit tests require modules to be importable. Your code should always check `if __name__ == '__main__'` before executing your main program so that the main program is not executed when the module is imported.

  ```
  def main():
      ...

  if __name__ == '__main__':
      main()
  ```

- **A Foolish Consistency is the Hobgoblin of Little Minds**

  A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important. However, know when to be inconsistent

- **Code lay-out**
  - **Indentation**

    Use 4 spaces per indentation level. Spaces are the preferred indentation method.
    Break input arguments, list inputs to make them visible.
  - **Tabs or Spaces?**

    Spaces are the preferred indentation method.
  - **Maximum Line Length**

    Limit all lines to a maximum of 79 characters. For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters.
  - **Blank Lines**

    Surround top-level function and class definitions with two blank lines. Method definitions inside a class are surrounded by a single blank line. Extra blank lines may be used (sparingly) to separate groups of related functions.
  - **Source File Encoding**

    Code in the core Python distribution should always use UTF-8 (or ASCII in Python 2).
  - **Imports**

    Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants. Absolute imports are recommended.
- **String Quotes**

  In Python, single-quoted strings and double-quoted strings are the same. This PEP does not make a recommendation for this. Pick a rule and stick to it.
- **Whitespace in Expressions and Statements**
  - **Pet Peeves**

    Yes: spam(ham[1], {eggs: 2})
    No: spam(ham[1], { eggs: 2 })
    Yes: if x == 4: print x, y; x, y = y, x
    No: if x == 4 : print x , y ; x , y = y , x
    Yes: ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]
    No: ham[1: 9], ham[1 :9], ham[1:9 :3]
    `Yes: spam(1)`
    No: spam (1)
    Yes:

    ```
    x = 1
    y = 2
    long_variable = 3
    ```

    No:

    ```
    x           = 1
    y           = 2
    long_variable = 3
    ```
  - **Other Recommendations**

Yes:

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

No:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

Yes:

```
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

Rather not:

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

- **Comments**

  Comments should be complete English sentences.

  - **Block Comments**

    Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space. Paragraphs inside a block comment are separated by a line containing a single # .

  - **Inline Comments**

    Use inline comments sparingly.

  - **Documentation Strings**

    Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the def line. Note that most importantly, the """ that ends a multiline docstring should be on a line by itself.

- **Version Bookkeeping**

  If you have to have Subversion, CVS, or RCS crud in your source file, do it as follows.

  ```
  __version__ = "$Revision$"
  # $Source$
  ```

  These lines should be included after the module's docstring, before any other code, separated by a blank line above and below.

- **Naming Conventions**

  `module_name, package_name, ClassName, method_name, ExceptionName, function_name, GLOBAL_CONSTANT_NAME, global_var_name, instance_var_name, function_parameter_name, local_var_name.`

  - **Prescriptive: Naming Conventions**

    - **Names to Avoid**
    - **Package and Module Names**

      Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

    - **Class Names**

      Class names should normally use the CapWords convention.

    - **Exception Names**

      Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix "Error" on your exception names

    - **Global Variable Names**

      The conventions are about the same as those for functions.

    - **Function Names**

      Function names should be lowercase, with words separated by underscores as necessary to improve readability. mixedCase is allowed only in contexts where that's already the prevailing style

    - **Function and method arguments**

      Always use `self` for the first argument to instance methods.

      Always use `cls` for the first argument to class methods.

      If a function argument's name clashes with a reserved keyword, it is generally better to append a single trailing

underscore rather than use an abbreviation or spelling corruption. Thus `class_` is better than `clss` .

- **Method Names and Instance Variables**
  Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability. Use one leading underscore only for non-public methods and instance variables.
- **Constants**
  Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include `MAX_OVERFLOW` and `TOTAL` .
- **Designing for inheritance**
  Public attributes should have no leading underscores. If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores.

- **Public and internal interfaces**
  Documented interfaces are considered public, unless the documentation explicitly declares them to be provisional or internal interfaces exempt from the usual backwards compatibility guarantees. All undocumented interfaces should be assumed to be internal.
  To better support introspection, modules should explicitly declare the names in their public API using the `__all__` attribute. Setting `__all__` to an empty list indicates that the module has no public API.
  Even with `__all__` set appropriately, internal interfaces (packages, modules, classes, functions, attributes or other names) should still be prefixed with a single leading underscore.

- **Programming Recommendations**
  Yes:
  if foo is not None:
  No:
  if not foo is None:

  Yes:
  def f(x): return 2*x
  No:
  f = lambda x: 2*x

1.2

Für die Lesbarkeit eines Programms ist nicht nur das Layout wichtig, sondern auch die Organisation des Codes. Ein wichtiges Prinzip ist das Prinzip der Modularität, also die Aufteilung eines Programms in kleinere, wohldefinierte Bestandteile. Es gibt aber auch andere solche “Prinzipien der Programmierung”. Recherchiere einige dieser Prinzipien im Internet (zum Beispiel auf Wikipedia: https://en.wikipedia.org/wiki/Category:Programming_principles) und notiere drei, die dir besonders einleuchten. Weshalb sind sie wichtig?

- **The DRY principle** is stated as “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.” When the DRY principle is applied successfully, a modification of any single element of a system does not require a change in other logically unrelated elements.
  Wie bei Lehrbüchern, in denen mehrmals dasselbe erzählt wird, in jedem Folgekapitel mit ein paar Zusatzinformationen. Diese sind dann meistens um die 600 Seiten zu lang, und laufen Gefahr, sich gelegentlich zu widersprechen.
- **Rule of three** is a code refactoring rule of thumb to decide when a replicated piece of code should be replaced by a new procedure. It states that the code can be copied once, but that when the same code is used three times, it should be extracted into a new procedure.
  Ein gut benannte und dokumentierte Funktionsdeklaration ist wesentlich verständlicher und einfacher zu warten, als dieselbe Codesequenzen mehrmals kopiert und in einigen Argumenten/Variablen verändert.
- ***Worse is better***, also called ***New Jersey style***, was conceived by Richard P. Gabriel to describe the dynamics of software acceptance, but it has broader application. The idea is that quality does not necessarily increase with functionality. There is a point where less functionality ("worse") is a preferable option ("better") in terms of practicality and

usability. Software that is limited, but simple to use, may be more appealing to the user and market than the reverse.
Das ist der Grund, weshalb mir viele Unix Befehle ein Rätel bleiben. Die haben teilweise gegen hundert Optionen und wenig oder keine Beispiele in der Dokumentation. Microsoft hat sich nicht durchgesetzt, weil sie besser waren, sonder weil das wesentliche intuitiv angewendet werden konnte. Optimalerweise sollte eine Software oder Programmiersprache so designt sein, dass auch der unregelmässige Anwender die Grundfunktionalitäten intuitiv und ohne viel Mühe benützen kann. Weitergehende Konzepte sollten bevorzugt anhand von Code- oder Anwendungsbeispielen erklärt werden. Google hat sich als Suchmaschine wohl auch deshalb gegenüber Yahoo durchgesetzt, weil die Eingabeseite nicht derart überladen war, sondern nur eine Funktionalität hatte.

1.3
Nimm ein altes Python-Skript von dir, zum Beispiel aus PCL I, und kritisiere es. Welche Stil- und Organisationsprobleme findest du? Das Skript sollte ungefähr 50-80 Zeilen lang sein. (Wenn du kein geeignetes Skript hast, darfst du auch eins von jemand anderem nehmen.)

Mein Programm:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

#PCL I, Übung 3, HS15
#Aufgabe 2_2
#Autor: Roland Benz
#Matrikel-Nr.: 97-923-163

'''
2.2 Buchstabensuppe
Nimm den String „gesundheitswiederherstellungsmittelmischungsverhaeltniskundiger“, das das längs-
te Wort des Text+Berg Korpus (SAC-Jahrbuch 1905) ist, und lass dieses mit Hilfe einer for-Schleife
ausgeben. In jedem Durchlauf der for-Schleife soll jeweils nur ein einziger Buchstabe des Wortes aus-
gegeben werden, und zwar so oft wie seine Position im Alphabet. Zum Beispiel tritt „g“ im Alphabet
an 6ter Stelle auf. Darum wird es 6 mal ausgegeben. Wenn beispielsweise das Programm nur die
Zeichen „gesundheit“ liest, sieht die Ausgabe so aus:
gggggg
eeee
sssssssssssssssss
uuuuuuuuuuuuuuuuuuuu
nnnnnnnnnnnnn
dddd
hhhhhhhh
eeeee
iiiiiiiii
ttttttttttttttttt
Tipp: Du erhältst die Position eines Zeichens im Alphabet mit der Funktion string.lowercase.index():
import string
letter_pos = string.lowercase.index(a_letter)
'''

#Debug:  python -m pdb <fileName.py>

#Your code
from random import choice
from string import lowercase
import string

strLongWrd="gesundheitswiederherstellungsmittel"\
                                        +"mischungsverhaeltniskundiger"
print "\n %s \n" % (strLongWrd)

#Solution 1
i=0
for char in strLongWrd:
        i+=1
        charPosABC = string.lowercase.index(char) + 1
        strToPrint = "".join(char for i in range(charPosABC))
        print " %d : <%d> \t %s \n" % (i,charPosABC,strToPrint)
        #print "%d:   " + strToPrint + "\n"              % (i)

#Solution 2
i=0
for char in strLongWrd:
        i+=1
        charPosABC = string.lowercase.index(char) + 1
        strToPrint = char * charPosABC
        print " %d : <%d> \t %s \n" % (i,charPosABC,strToPrint)

#With randomized strings
i=0
```

Kommentar:

Programmencoding. Gut

Aufgabe angegeben. Gut

Autor angegeben. Gut

Aufgabenstellung reinkopiert. Man sieht ohne zu lesen, worum es bei der Aufgabe in etwa geht. Gut.

Zusätzlich eine kurze Zusammenfassung in einem Satz, Inputs, Outputs wäre hilfreich. Schlecht

Info zum Starten mit Debugger. Gut

Import Packages am richtigen Ort. Gut. Kommentar wäre hilfreich. Schlecht.

Kommentar, wozu der String gebraucht wird, wäre hilfreich. Schlecht.

Man sieht auf den ersten Blick, dass für die Aufgabenstellung zwei Lösungen angeboten werden. Gut.

Der Code hat keinen Kommentar. Schlecht.

Es handelt sich dreimal um denselben Algorithmus. Eine Funktion wäre deshalb, eine saubere und übersichtliche Lösung. Schlecht.

PCL II – Loesungen Uebung 01
Linus Manser (lmanser, 13-791-132) und Roland Benz (rolben, 97-923-163)

for char in strLongWrd: i+=1 charPosABC = string.lowercase.index(char) + 1 strToPrint = "".join(choice(lowercase) for i in range(charPosABC)) print " %d : <%d> \t %s \n" % (i,charPosABC,strToPrint)	Eine zusätzliche Lösung, welche jedoch in der Aufgabenstellung nicht verlangt wurde, um die Methode random.choice() zu testen. Gut.

Reflexion/Feedback

a) Fasse deine Erkenntnisse und Lernfortschritte in zwei Sätzen zusammen.
b) Wie viel Zeit hast du in diese Übungen investiert?

a)Auffrischen von Vererbung, Kennenlernen von Parsern, Kennenlernen der __new__ Funktion, Auffrischen der Kommandozeilenaufrufe mit Argumenten und Optionen, Kennenlernen der sys.stdin mit codecs.getreader, Auffrischen von Encoding.
b)Zusammen 20 Stunden