

Cloud Style Guide

30. März 2016

1 Quick-Start: how to

Regel Nummer eins: viel Inhalt, wenig Bla Bla! Dies wollen wir hier auch einhalten. **Regel Nummer zwei:** sinnvolle Kommentare anbringen, Variablen+Dateien nachvollziehbar benennen, insgesamt auf Nachvollziehbarkeit achten. Hierzu unbedingt das Kapitel 2 beachten! **Regel Nummer drei:** eigene nützliche Erkenntnisse den Anderen zur Verfügung stellen durch Aktualisieren des Kautschuk Repos. Die verwendeten Programme:

- Python 3.x
- LuaLaTeX
- make
- atom o.ä. Texteditoren (z.B. VIM, notepad++)

Der generelle Ablauf zum Erstellen eines Protokolls sollte sich wie folgt gestalten. Zunächst wird das verwendete Template (also die Vorlage) in einen vernünftig benannten Ordner (vorzugsweise /Praktikum/xxx mit xxx=Versuchsnummer) kopiert. Das Template ist im Kautschuk Repo zu finden und sollte bei sinnvollen Änderungen stets gepusht werden, sodass alle dieselbe Vorlage verwenden. Dann solltet ihr schauen, ob es den Versuch bereits gibt im Repo. Falls nicht, solltet ihr gesteigerten Wert auf die Nachvollziehbarkeit eures Codes legen. In diesem Fall seid ihr es, die diesen Versuch nämlich im Repo erstellen werdet.

Die Daten aus den Messungen müssen ins Digitale übertragen werden. Dazu zwei Möglichkeiten:

- Manuelles Anlegen von .txt Dateien im Ordner "/Praktikum/xxx/Messdaten".
- Erstellen von .txt mit Hilfe von Python (der berühmte Datenersteller.py). Aber auch hierbei sollen die Daten in dem "/Praktikum/xxx/Messdaten" Ordner landen.

Die .txt Dateien sollten ein sinnvolles Format haben. Zum Beispiel könnten die Dateien nach Aufgabenteilen benannt werden, z.B. "a.txt". Die erste Zeile einer jeden .txt Datei im Ordner Messdaten muss eine Beschreibung der Spalten samt Einheiten beinhalten, natürlich ist diese Zeile auszukommentieren. Z.B.

```
|| # Wert [-]    R_2[Ohm]    R_3/R_4[-]
```

Jeder soll wissen, was genau ihr dort eingetragen habt, ggf. müssen weitere Kommentare helfen, um eindeutig anzuzeigen, welche Daten vorliegen.

Der nächste Schritt ist die Bearbeitung der Datei "/Praktikum/xxx/PythonSkript.py", welche sich durch das Kopieren des Templates bereits an richtiger Stelle befindet. In diesem Skript sind die gängigen Code Examples als Kommentare hinterlegt, sodass diese direkt an die benötigte Stelle kopiert werden können. Diese .py ist per default die einzige python Datei, die von make ausgeführt wird. Nun zum generellen Vorgehen - Beispiel-Code hierzu findet sich in PythonSkript.py wieder.

- Nach den imports werden ggf. benötigte Funktionen deklariert mit einer sprechenden Namensgebung, z.B.

```
||     def phi(param_a, Temperatur)
```

- Nun werden globale Konstanten deklariert und in einem Kommentar beschrieben, z.B.

```
||     R = const.physical_constants["molar gas constant"]      #Array of value,  
||         unit, error
```

- Erst jetzt werden die Aufgabenteile behandelt. Dazu wird eine kommentierte Abgrenzung verwendet und es folgt die Kommentarzeile

```
|| ##### Aufgabenteil a) #####
```

- Variablen werden deklariert, häufig durch Einlesen der Messwerte, und in SI umgerechnet, z.B.

```
|| c_W = np.genfromtxt('Messdaten/spezifische_Waermekapazitaet.txt', unpack=
|| True)
|| c_W *= 1e-3 # J / (kg*K)
```

- Häufig erfolgt dann ein Curve-Fit. Dieser kann mit den bereit gestellten Funktionen durchgeführt werden, z.B.

```
|| params_max = ucurve_fit(reg_linear, t_extern, U_pos_ges_log)
```

In der Datei "regression.py" sind verschiedene Fit-Funktionen vorgehalten.

- Die Ergebnisse der Regression werden in neue Variablen geschrieben, z.B.

```
|| R_a, Offset_a = params_max
```

Dann werden sie über die Funktionen

```
|| write(string filename, string content)
|| make_SI(double value, unit, figures)
```

zur Wiedergabe im Tex-Dokument in den build-Ordner geschrieben. Die Funktion *make_SI* ermöglicht die vernünftige Formatierung zur Ausgabe im PDF, vor Allem durch die Wahl der Nachkommastellen mit der Angabe von *figures*.

- Häufig werden Datensätze auch in Tabellen gespeichert. Hierzu empfiehlt sich für eine gute Formatierung die Nutzung der Funktion *make_full_table*, welche die Verwendung von *make_table* voraussetzt. Ein beschriebenes Beispiel hierzu findet sich in der PythonSkript Datei wieder. Es gibt außerdem eine kleine Doku dazu hier im Kautschuk Repo. Wichtig ist, dass *make_full_table* eine komplett fertig formatierte Tabelle ausgibt, die in Latex nur noch durch z.B.

```
|| \input build/Tabelle_b_texformat.tex
```

eingebunden werden muss. Label und Caption sind also jetzt nicht mehr in Latex, sondern in Python einzustellen.

- Ebenso finden sich für das Plotten von Daten eine Reihe von immer wieder verwendeten Funktionen in unserer Vorlage. Es ist z.B. möglich, die Arraygrenzen automatisch zu setzen. Hier gibt es im Rahmen der matplotlib sicher eine ganze Reihe von speziellen Anwendungen. Findet sich bei der Recherche etwas Nützliches, so sollte dies durch entsprechende Ergänzungen in dem Beispielblock allen im Kautschuk Repo verfügbar gemacht werden. Wir nutzen im Übrigen matplotlibrc, was eine vernünftige Formatierung von Grafiken ermöglicht, aber die Rechenzeit bei der Ausführung des Python Skripts deutlich erhöht. Zum schnellen Testen empfiehlt es sich, die Plots zunächst auszukommentieren. Wie die Plots ins tex Dokument eingebunden werden, kann den Beispielen in "/Praktikum/xxx/content/auswertung.tex" entnommen werden. Es gibt verschiedene Anordnungsoptionen.

Das Tex-Dokument soll dann jeder so erstellen, wie er es für richtig hält. Relevant für uns ist in erster Linie "PythonSkript.py". Trotzdem gibt es sicher den ein oder anderen, der euren gesamten Code gerne ausführen würde. Dazu gilt als **Regel Nummer vier**, dass das Klonen des Repos es einer beinhalteten MAKE-file ermöglicht, zu bauen, OHNE dass man noch irgend etwas tun muss! Die Standard-make file ist in der Vorlage enthalten. Schließlich wäre es noch nett, in den Ordner "/Praktikum/xxx/final/" die abtestierten Protokolle in PDF-Form zu legen. Zu guter letzt pusht ihr den xxx-Ordner an die entsprechende Stelle im Kautschuk Repo, sodass nun jeder Zugriff darauf hat und es von jedem Versuch möglichst schnell eine Referenz gibt.

2 Coding Style

Hier einige Leitlinien für einen sauberen und auch von anderen gut lesbaren Python-Code.

- Grundsätzlich orientieren wir uns an PEP8, einem der am weitesten verbreiteten Standards für Python-Code. Die Quintessenz davon ist:

- `import`-Statements sollten sich am Anfang der Datei befinden, es sollte nur importiert werden, was auch wirklich gebraucht wird.
- Operatoren sind sinnvoll mit Whitespace zu umgeben:

```
# Sinnvoller Gebrauch von Whitespace
```

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

```
# Aber so nicht:
```

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

- Sinnvolles setzen von neuen Zeilen: logisch getrennte Blöcke innerhalb einer Funktion/Aufgabe mit einer, Aufgaben, Klassen, etc. mit zwei neuen Zeilen voneinander trennen.
- Am einfachsten lässt sich das mit einem guten Plugin einhalten; für Atom gibt es *atom-beautify*, das einen den Großteil der Arbeit abnimmt.
- Sprechender Code ist besser als Kommentare! Es ist einfacher und schneller (für den Leser des Codes) den Sinn von einigen Zeilen Code zu erkennen, wenn dieser gut benannte Variablen, Funktionen, etc. besitzt. Kommentare sind sparsam zu verwenden und kurz zu halten; nur wenn sich die Funktionsweise/der Sinn nicht aus dem Code ergibt oder eine Zeile nicht ersichtliche 'Nebenwirkungen' hat, ist ein Kommentar angebracht.
- Wenn möglich *docstrings* verwenden statt Kommentaren! Zum Beispiel kann man eine Methode so kommentieren:

```
def example_method(param1, param2):
    """Class methods are similar to regular functions.

    Note:
```

```

        Special cases, side effects, etc.

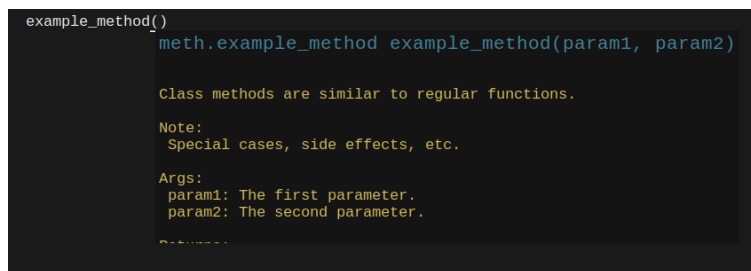
    Args:
        param1: The first parameter.
        param2: The second parameter.

    Returns:
        True if successful, False otherwise.

    """
    pass
example_method()

```

Der Vorteil liegt darin, dass ein 'Kommentar' in dieser Form dann immer erscheint, wenn man diese Methode aufruft (wenn man seinen Texteditor richtig eingestellt hat, duh).



```

example_method()
meth.example_method example_method(param1, param2)

Class methods are similar to regular functions.

Note:
    Special cases, side effects, etc.

Args:
    param1: The first parameter.
    param2: The second parameter.

Returns:
    True if successful, False otherwise.

```

Insbesondere die Datentypen, die die Methode annimmt und ausgibt sollten spezifiziert werden, wenn dies nicht aus der Benennung hervorgeht (`IsWindowOpen()` gibt mit Sicherheit ein boolean zurück, bei `GenerateInfoGraphic()` ist das aber nicht mehr klar). Docstrings können auch für Variablen genutzt werden.

- Eine für Physiker sinnvolle naming convention ist folgende:
 - Variablennamen und Funktionen benutzen Kleinschreibung für Wörter, physikalische Bezeichner (T für Temperatur, `phi` für einen Winkel) werden so gut es geht in gültige Variablennamen übertragen.
 - Zum Trennen von Wörtern sind Unterstriche zu verwenden.
 - Meistens ist anders als in einer Rechnung auf Papier nicht sofort ersichtlich, was für ein Winkel, etc. gemeint ist; es schadet selten, einen Variablennamen zu wählen, der auch noch 15 Zeilen später aussagekräftig ist.
 - Hilfsvariablen sind sinnvoll zu benennen; `m2` und `bla` sind sehr schlechte Namen für Variablen, auch wenn man sie nur für ein paar Zeilen braucht. Es erleichtert das Lesen des Codes ungemein, diesen aussagekräftige Namen zu geben. Nebenbei findet man so auch Fehler im eigenen Code schneller.
Einen einfachen counter in einer for-Schleife, dessen Sinn auch ohne langen Namen ersichtlich ist, muss man aber nicht umändern. Hier sollten *i*, *j* oder *cnt* verwendet werden.
 - Wenn eine Variable einheitenbehaftet ist, wird die Einheit hinter der Deklaration als Kommentar angegeben. Es sind immer Grundeinheiten zu verwenden (also etwa immer Meter, nie Kilometer).

– Beispiel:

```
def generate_fibonacci_number(target_elem):
    previous_elem = 0
    current_elem = 1
    for i in range(target_elem):
        next_elem = current_elem + previous_elem
        previous_elem = current_elem
        current_elem = next_elem
    return current_elem

T_cold_bucket = 230 # K
phi_mirror = 1.764 # rad
distance_microphone = 10 # m
```

Diese Leitlinien sind nur ein erster Vorschlag; bitte übt Kritik! Ein solcher Standard hilft enorm, aber wir müssen ihn alle als akzeptabel befinden, damit ihn auch jeder gerne nutzt.