

Bisherige Themen

Letzte Woche haben wir folgende Themen behandelt:

- Komplexe Aufgabenstellungen in Teilprobleme zu zerlegen, die sich leichter bearbeiten lassen
- Nutzereingaben während des Programms abzufragen
- Dateien zu lesen (Modus "r")
- Dateien anzulegen oder zu überschreiben (Modus "w")
- Dateien zu ergänzen (Modus "a")

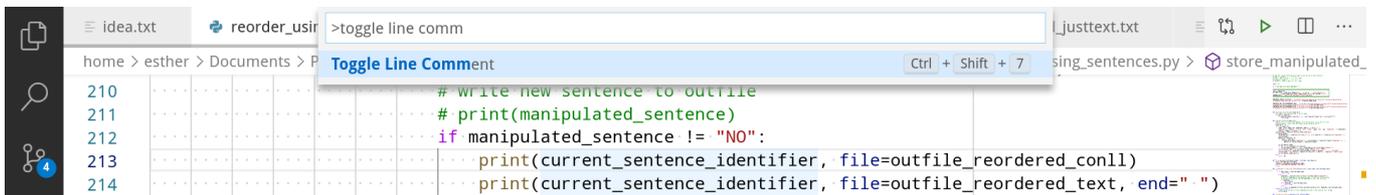
Falls Sie Fragen zu diesen Themen oder zu den Übungsaufgaben haben, sprechen Sie uns bitte an!

Und so geht es weiter:

Editor-Trick des Tages: Toggle Line Comment

In VSCode können Sie eine Tastenkombination verwenden, um die Zeile, in der der Cursor sich befindet, sofort zu kommentieren. Sie können den Shortcut auch mit dem Multicursor oder einfach der Markierung mehrerer Zeilen kombinieren.

Probieren Sie es aus! Öffnen Sie eine Ihrer letzten Pythondateien und drücken Sie `Ctrl + Shift + P`, um die Command Palette zu öffnen. Geben Sie den Befehl `>toggle line comment` ein. Während Sie den Befehl eingeben, sehen Sie auf der rechten Seite einen Shortcut, vermutlich `Ctrl + Shift + 7`. Sie können den Befehl entweder über die Command Palette ausführen oder sich den Shortcut merken und ab sofort Zeilen bequem kommentieren. Was passiert, wenn Sie den Shortcut zweimal nacheinander auf die gleiche Zeile anwenden?



Datentyp des Tages: None

Es gibt in Python einen speziellen Datentyp namens `None`, der leere Objekte bezeichnet, die keinen anderen Datentyp haben.

In `[]`:

```
print(type(None))

# Leere Listen, Strings usw. haben ihren erwarteten Datentyp:
print(type([]))
print(type(""))
```

Oft haben Methoden, die z.B. auf veränderlichen Objekten operieren, den Rückgabebetyp `None` - es muss immer einen Rückgabebetyp geben, und `None` bekommen wir, wenn nichts anderes zurückgegeben wird. Wenn Sie also eine Fehlermeldung wie die folgende sehen...

In [1]:

```
for i in [1,2,3].append(4):
    print(i)          # :(
```

```
-----
-
TypeError                                Traceback (most recent call las
t)
<ipython-input-1-186a39154164> in <module>
----> 1 for i in [1,2,3].append(4):
      2     print(i)          # :(
```

TypeError: 'NoneType' object is not iterable

... liegt das mit hoher Wahrscheinlichkeit daran, dass das Ergebnis der Methode `append()` vom Typ `None` ist und nicht - wie im Codebeispiel sinnvoll wäre - eine Liste.

Damit der Code ohne Fehler ausgeführt wird, können wir beispielsweise folgendes schreiben:

In []:

```
for i in [1,2,3] + [4]:
    print(i)          # :)
```

Hier übrigens der Grund, warum der erste Versuch schiefgegangen ist: Das Ergebnis der `append()`-Methode hat den Rückgabety `None` .

In []:

```
print(type([1,2,3].append(4)))
```

`None` kann auch als Boolean ausgewertet werden und hat dann den Wert `False` . Das ist auch bei leeren Strings, leeren Listen und der Zahl `0` der Fall:

In []:

```
print(bool(None))
print(bool([]))
print(bool(""))
print(bool(0))

print("-----")

print(bool([5,3,1]))
print(bool("String mit Inhalt"))
print(bool(-10))
```

Wenn Sie mitten in Ihrem Programm den Wert einer Variable ausgeben lassen und `None` angezeigt wird, wissen Sie, dass etwas nicht geklappt hat. Prüfen Sie in so einem Fall alle Stellen, an denen der Wert der Variable verändert wurde, daraufhin, ob eventuell keine Rückgabe geliefert wurde und daher `None` der Wert der Variable ist.

Funktionen



Inzwischen haben wir schon eine Menge Programmiergrundlagen behandelt und Sie sind bereit, komplexere Aufgaben zu bearbeiten. **Funktionen** bieten Ihnen die Möglichkeit, Sinnabschnitte Ihrer Programme vom Rest abzukapseln und so den Überblick zu bewahren.

Stellen Sie sich beispielsweise vor, dass Sie Quadratzahlen berechnen wollen. Bisher haben wir das etwa so gemacht:

In []:

```
input_number = 6                                # Variable erzeugen und mit dem Wert 3 belegen
result = input_number * input_number            # Quadratzahl berechnen
print(result)                                   # Ergebnis ausgeben
```

Wenn wir genau eine Quadratzahl berechnen wollen, ist gegen den Code oben nichts einzuwenden. Wenn wir die Berechnung aber mehrmals für unterschiedliche Werte durchführen wollen, lohnt es sich, den Code dafür allgemeiner zu schreiben. Dann können wir die Funktionalität "Quadratzahl berechnen" vom restlichen Code abkapseln und jederzeit aufrufen, wenn wir sie brauchen.

Eine Funktion hat in Python die folgende Form:

```
def calculate_square(input_number):           # Funktionsdefinition eröffnen
    result = input_number * input_number     # beliebig viele Befehle
    return result                             # Funktionsdefinition beenden
```

Die Einrückung zeigt, welche Zeilen zur Funktionsdefinition gehören. Der **Kopf** der Funktion enthält den Namen (hier: `calculate_square`) und eine Angabe aller **Parameter**, die wir in der Funktion brauchen. (Erinnern Sie sich an die Parameter in der Sitzung am 16.10., z.B. `s.split(".", 2)`)

In der Beispielfunktion heißt der einzige Parameter `input_number`.

Im **Körper** der Funktion folgen beliebig viele Befehle, in denen das **Argument** (der konkrete Wert, der als Parameter verwendet wird) verarbeitet wird. Beispielsweise berechnen wir oben das Quadrat der übergebenen Zahl. Das Ergebnis wird hier in der Variable `result` gespeichert.

Schließlich gibt die Funktion mit `return` einen Wert zurück. Im Beispiel ist das der Wert, der in der Variable `result` gespeichert ist. Da `return` die Funktion sofort beendet, werden keine eingerückten Zeilen unterhalb von `return` mehr ausgeführt.

Wir können uns Funktionen wie eine Art Rezept vorstellen, nach dem bestimmte Aufgaben gelöst werden. Die Funktionsdefinition, die wir oben sehen, braucht noch einen konkreten Eingabewert (ein Argument für den Parameter `input_number`), damit ein Ergebnis berechnet werden kann.

Erst, wenn die Funktion im Code mit einem konkreten Argument **aufgerufen** wird, werden die Befehle innerhalb des Funktionskörpers ausgeführt. Das Ergebnis - der Wert, der in der letzten Zeile der Funktion zurückgegeben wird - kann dann an der Stelle im Code, wo die Funktion aufgerufen wurde, verwendet werden.

Probieren Sie es aus: Führen Sie den Code in der nächsten Zelle aus und beachten Sie, in welcher Reihenfolge die `print()`-Befehle ausgeführt werden.

In []:

```
print("1")

def calculate_square(input_number):
    print("2")
    result = input_number * input_number
    return result

print("3")

my_square = calculate_square(4)   # das Ergebnis des Funktionsaufrufs calculate_square
                                  # mit dem Argument 4
                                  # wird in der Variable my_square gespeichert

print("Ergebnis der Berechnung: " + str(my_square))

print("4")
```

Im Gegensatz zu z.B. Schleifen sind Variablen, die in Funktionskörpern definiert werden, außerhalb der Funktion nicht mehr verfügbar. Probieren Sie es aus: Ergänzen Sie im Beispiel unterhalb des bisherigen Codes einen `print()`-Aufruf für `result` oder `input_number`. Das Programm stürzt ab.

Mithilfe von Funktionen können wir Programmcode, der vorher unübersichtlich war, auslagern. Wir geben Funktionen dafür sprechende Namen und lassen jede Funktion genau eine Aufgabe erfüllen.

Beachten Sie, dass der Interpreter eine Funktion *gelesen* haben muss, bevor sie *ausgeführt* werden kann. Eine Funktion muss also immer oberhalb ihres ersten Aufrufs im Hauptprogramm definiert werden.

Don't Repeat Yourself

Mit Funktionen können wir **Redundanz** (unnötige Wiederholungen) in unserem Code vermeiden. Jedesmal, wenn wir später im Programm eine Quadratzahl berechnen wollen, können wir die eben definierte Funktion verwenden, statt immer wieder die gleichen Codezeilen an verschiedenen Stellen einzufügen. Die Beispielfunktion ist sehr übersichtlich, aber wenn eine Funktion eine komplexere Aufgabe erfüllt - z.B. Daten nach bestimmten Kriterien zu filtern - , wollen wir vermeiden, Codezeilen mehrfach zu schreiben.

Codewiederholungen sind eine typische Fehlerquelle. Außerdem machen Sie unser Programm länger und dadurch unübersichtlicher. Mit Funktionen vermeiden wir solche Wiederholungen.

Aufgabe

1. Können Sie den Code unten "aufräumen"? Der Funktionskörper steht schon da. Verschieben Sie die Befehle, die in die Funktionen gehören, und ersetzen Sie sie im Hauptprogramm durch einen Funktionsaufruf. Achten Sie auf die Benennung der Variablen im Hauptprogramm und in der Funktionsdefinition.

In []:

```
##### Funktionsdefinition #####  
  
def filter_dictionary_only_keep_nouns(input_dict):  
    """  
    Diese Funktion filtert das eingegebene Dictionary so, dass zum Schluss  
    ein Dictionary zurückgegeben wird, das nur die Nomen aus dem ursprünglichen  
    Dictionary enthält  
    """  
    output_dict = {}  
  
    ##### - Hier Code einfügen - #####  
    # ...  
    #####  
  
    return output_dict  
  
##### Hauptprogramm #####  
  
words = {"Terminal": "N", "for-Schleife": "N", "einrücken": "V", "mutable": "A", "zurückgeben": "V", "redundant": "A", "Funktion": "N"}  
filtered_words = {}  
  
for w in words:  
    if words[w] == "N":  
        filtered_words[w] = "N"  
  
print("gefiltertes Dictionary (nur Nomen): " + str(filtered_words))
```

Versuchen Sie ab jetzt, Übungsaufgaben durch den sinnvollen Einsatz von Funktionen zu lösen. Dadurch erhalten Sie Zwischenergebnisse, können Programmierfehler besser finden, und bei Änderungen im Code sehen Sie leichter, welche Funktionen angepasst werden müssen und welche gleich bleiben können.

Zusammengefasst: Tipps zum Schreiben von Funktionen

1. Jede Funktion soll **eine** Aufgabe erfüllen.
2. Verändern Sie innerhalb einer Funktion **nur die Argumente und die Output-Werte**, nicht die Variablen außerhalb der Funktion.
3. Funktionsdefinitionen müssen im Code immer **oberhalb** der dazugehörigen Funktionsaufrufe stehen.
4. Sie können auch innerhalb einer Funktionsdefinition eine oder mehrere andere Funktionen aufrufen.
5. Sie können mehrere `return`-Statements schreiben, die in unterschiedlichen Fällen greifen. Achtung: Die Funktion soll in jeder möglichen Verwendung eine Rückgabe liefern.

Module

Ab jetzt empfehlen wir Ihnen, Sinnabschnitte unserer Python-Programme in Funktionen auszulagern, wann immer das sinnvoll ist.

Wenn Sie möchten, können Sie Ihre Funktionsdefinitionen sogar in eine separate Datei auslagern. Diese Datei können Sie dann - wie letzte Woche mit dem Modul `random` - am Anfang Ihres Codes importieren. Dazu muss die Datei mit den Funktionen eine `.py`-Dateiendung haben und im gleichen Verzeichnis liegen wie die andere Pythondatei.

```
# Datei utils.py
def calculate_square(input_number):
    print("2")
    result = input_number * input_number
    return result
```

```
# Datei main.py
import utils

print(utils.calculate_square(3))
```

Ihre eigenen Module zu definieren kann sinnvoll sein, wenn Sie an größeren Projekten arbeiten. Ein wichtiger Teil der Philosophie von Python ist, dass Code für Menschen lesbar sein soll:

Readability counts!

Jedes Programm muss früher oder später gewartet werden, wenn sich Teile des Systems, auf dem das Programm ausgeführt wird, ändern. Die Wartbarkeit eines Programms steigt mit der Lesbarkeit: Wenn jede Teilaufgabe von einer Funktion erledigt wird, und wir auf den ersten Blick sehen, welche Funktion welche Teilaufgabe erfüllt, können wir viel besser mit dem vorhandenen Code arbeiten.

Ein Beispiel dafür, warum Lesbarkeit und Wartbarkeit nützlich sind:

Stellen Sie sich vor, Ihr Programm liest Informationen aus einer Datenbank. Nach drei Jahren stellen Sie fest, dass die Datenbank nicht groß genug ist, und ersetzen sie durch eine andere Datenbank, deren Struktur ganz anders aufgebaut ist.

Um das Pythonprogramm zu warten, bearbeiten Sie jetzt *nur* die Funktion, die für das Lesen der Informationen aus der Datenbank zuständig ist. Der Rückgabewert der Funktion, z.B. ein Dictionary, wird wie vorher an das restliche Programm weitergereicht; alle folgenden Funktionen können weiterarbeiten wie vorher. Dadurch, dass Sie die Funktionalität "Datenbank lesen" vom restlichen Code isoliert haben, müssen Sie nur einen Bruchteil des Programms ändern.

Namespaces

Dass Variablen, die in Funktionen definiert werden, nicht außerhalb der Funktion verfügbar sind, liegt daran, dass sie sich in einem anderen **Namespace** befinden. Ein Namespace ist die Zuordnung von Namen (z.B. Variablen) zu Werten, oder genauer: Zu Zellen im Arbeitsspeicher, in denen die Werte abgelegt wurden. Ein Pythonprogramm hat bis zu 3 Namespaces:

1. Built-in Namespace: Enthält die Ausdrücke, die Teil der Sprache Python sind, z.B. `print`, `str()`, `for`, `if`, `def` etc.
2. Globaler Namespace: Enthält Ausdrücke, die innerhalb eines Moduls (einer Pythondatei) existieren.
3. Lokaler Namespace: Enthält Ausdrücke, die innerhalb einer Funktion existieren.

Informationen über Namespaces können wir erhalten, wenn wir uns das Ergebnis der Methode `dir()` ausgeben lassen. `dir()` enthält bei jedem Aufruf die Namen, die im aktuellen Namespace existieren. Das folgende Codebeispiel verdeutlicht den Unterschied: `dir()` wird einmal außerhalb der Funktionsdefinition aufgerufen und einmal innerhalb der Funktion.

Achtung: Falls Sie das Programm hier im Jupyter Notebook mehrfach ausführen, merkt der Interpreter sich die Variablen, sodass die Ausgabe beim zweiten Mal anders aussieht als beim ersten Mal. Um alles zurückzusetzen, können Sie oben im Menü das Kommando `Kernel -> Restart and Clear Output` ausführen. Oder Sie kopieren den Code in Ihren Editor und führen ihn dort aus.

In []:

```
print("Programmstart:")
print(dir())

print("-----")

mein_name = "Esther Seyffarth"

print("nach Variablendefinition:")
print(dir())

print("-----")

def calc_square(n):
    result = n**2
    print("in der Funktionsdefinition:")
    print(dir())
    print(mein_name)
    print("-----")
    return n**2

print("Nach Funktionsdefinition:")
print(dir())
print("-----")

print(calc_square(3))

print("Programmende:")
print(dir())
```

Der Aufruf in Zeile 1 zeigt uns den Namespace des Moduls. Darin enthalten sind zunächst nur die Namen, die zur Sprache Python gehören.

Als nächstes definieren wir eine eigene Variable und führen `print(dir())` erneut aus. Wir sehen, dass unsere eigene Variable jetzt Teil des Modul-Namespace (globaler Namespace) geworden ist.

Die Funktionsdefinition sorgt dafür, dass der Name der Funktion ebenfalls im Namespace erzeugt wird. Das sehen wir an der Ausgabe von Zeile 21.

Nun folgt ein Aufruf unserer selbstgeschriebenen Funktion mit dem Argument 3 (Zeile 24). Der Funktionskörper wird ausgeführt, wobei der Parameter `n`, eine lokale Variable im Funktions-Namespace, den Wert 3 hat.

Die Ausgabe in Zeile 16 zeigt uns, dass wir hier nicht mehr im globalen Namespace sind, sondern im lokalen Namespace der Funktionsdefinition. Nur zwei Variablen sind bekannt: `n`, der Parameter der Funktion, und `result`, eine Variable, die wir selbst in Zeile 14 innerhalb der Funktion erzeugt haben.

Am Ende des Programms wird schließlich noch einmal `print(dir())` ausgeführt. Wir sehen, dass weder `n` noch `result` in diesem Namespace enthalten sind. Daran liegt es, dass weiter oben im Notebook unser Programm abgestürzt ist, als wir versucht haben, Variablen aus der Funktionsdefinition auszugeben.

Aufgabe

1. Können wir innerhalb der Funktionsdefinition auf die zuvor definierte Variable `mein_name` zugreifen? Warum/warum nicht? Prüfen Sie Ihre Antwort, indem Sie einen Aufruf von `print(mein_name)` in der Funktion ergänzen.

Wenn wir im Code auf Variablen oder Funktionen zugreifen, hält der Interpreter sich an eine feste Reihenfolge: Er sucht den angegebenen Namen erst im lokalen Namespace (falls vorhanden), dann im globalen Namespace und dann im Builtin-Namespace.

Beim Importieren von Modulen, z.B. `import random`, werden Namen im globalen Namespace ergänzt:

In []:

```
print(dir())  
  
print("-----")  
  
import random  
  
print(dir())
```

Erinnern Sie sich, dass wir die Funktionen von `random` verwendet haben, indem wir geschrieben haben:

```
zufallszahl = random.randint(1,5)
```

Dabei haben wir den Interpreter informiert, dass die Funktion `randint()` im Namespace des Moduls `random` zu finden ist.

Das war notwendig, weil Namenskonflikte immer möglich sind: Es kann sein, dass wir im eigenen Code Funktionen definiert haben, die genauso heißen wie die Funktionen in importierten Modulen.

Führen Sie das folgende Codebeispiel aus und beachten Sie den Unterschied zwischen den `print()` - Anweisungen.

In []:

```
import random

def randint(min, max):
    # soll aussehen wie die Zufallszahlenfunktion
    result = "ätsch, das ist gar keine Zufallszahl"
    return result

print(random.randint(1,5))
print(randint(1,5))
```

Wir können den Python-Interpreter richtig verwirren, indem wir Namenskonflikte herbeiführen:

In []:

```
zahl = 5
print(5)
print(str(5))

def str(irgendwas):
    # ergänzt eine str()-Funktion im globalen Namespace dieser Pythondatei
    return "Haha, reingelegt"

# Der Interpreter sucht erst im globalen Namespace; nur falls dort der Name str nicht g
# efunden wird,
# sucht er im Builtin-Namespace
print(str(5))

### ACHTUNG ###
# Nach dem Ausführen dieser Zelle sollte am besten
# der Kernel neugestartet werden - sonst können wir
# in diesem Notebook nicht mehr die echte str()-Funktion
# verwenden :(
```

Die Take-Home-Message hier lautet: **Achten Sie auf Ihre Variablennamen!** Vermeiden Sie außerdem, innerhalb von Funktionen auf Variablen außerhalb des lokalen Namespace zuzugreifen. Es ist zwar möglich (siehe Beispiel oben mit der Variable `mein_name`), macht den Code aber schlecht nachvollziehbar und damit schwerer lesbar.

Stattdessen können Sie Variablen, die in der Funktion verfügbar sein sollen, als Parameter mit übergeben. Zum Schluss der Funktion geben Sie dann alle Werte, die weiter verwendet werden sollen, zurück - da wir Funktionen schreiben, um jeweils genau eine Aufgabe zu erfüllen, haben wir meist nur einen einzigen Rückgabewert. Haben wir einmal mehr, können wir z.B. ein Dictionary erstellen, das alle Rückgabewerte im gewünschten Format enthält und das dann nach dem Ausführen der Funktion "entpackt" werden kann.

for -Schleifen und Namespaces

Achtung! Weder Schleifen noch `if`-Blöcke erstellen ihre eigenen Namespaces. Das wird problematisch, wenn Sie z.B. Schleifen verschachteln und Variablennamen dabei mehrfach belegen.

Verwenden Sie also immer eindeutige Variablen.

Rückblick: Python-Syntax

Inzwischen haben wir eine Reihe von Werkzeugen zur Verfügung, um eigenen Pythoncode zu schreiben. Sie können jederzeit in den Kursmaterialien nachschauen, wie bestimmte Befehle oder Konstrukte verwendet werden. Hier eine kleine Übersicht über die verschiedenen Schreibweisen und die Bedingungen, unter denen jede Schreibweise verwendet wird:

Anweisung	Bedeutung
<code><name> = <wert></code>	Variablendefinition bzw. Überschreiben des Werts in bestehender Variable.
<code>for <element> in <sequenz>: <anweisung></code>	Schleife, die pro Element in der Sequenz einmal ausgeführt wird.
<code>if <bedingung>: <anweisung> elif <bedingung2>: <anweisung2> else: <anweisung3></code>	If-Block: Prüft, ob die erste Bedingung erfüllt ist. Falls nein, wird die zweite Bedingung geprüft. Falls alle alternativen Bedingungen nicht erfüllt sind, werden die Anweisungen im <code>else</code> -Block ausgeführt.
<code>def <funktion> (<parameter1>, <parameter2>): <anweisungen> return <rückgabewert></code>	Funktionsdefinition: Beliebig viele Parameter, ein Rückgabewert.
<code><name> = <funktion> (<argument>)</code>	Funktionsaufruf: Der Rückgabewert der Funktion (ausgeführt mit dem angegebenen Argument) wird als Wert in der Variable <code>name</code> gespeichert.
<code><name> = <objekt>. <methode>(<argument>)</code>	Methodenaufruf: Eingebaute Operationen z.B. für Strings. Das <code>objekt</code> ist der String, auf den die <code>methode</code> angewendet wird.

Zusammenfassung

Sie haben heute gelernt,

- wie Sie wiederkehrende Aufgaben in Funktionen umsetzen, die dann mit beliebigem Input aufgerufen werden können, um zum gewünschten Ergebnis zu kommen
- wie Sie Teile Ihres Codes in eigene Module auslagern, um Ihr Programm lesbarer, effizienter und wartbarer zu machen
- wie Sie Module importieren können
- wie Python die Namen von Variablen, Methoden und Funktionen auflöst und welche Probleme dabei auftreten können

In []:

```
#####
#                               Übungsaufgabe zu Funktionen                               #
#####
#
# Dieses Programm soll für beliebige User-Eingaben prüfen,
# ob sie im "Spongebob-Case" geschrieben sind. Das Ergebnis
# soll in Form eines Boolean (True oder False) zurückgegeben
# werden.
# Falls das Ergebnis "True" lautet, soll der ASCII-Spongebob
# ausgegeben werden. Andernfalls soll der Originaltext
# ausgegeben werden.
#
# Der Code funktioniert bereits. Können Sie die relevanten
# Zeilen so umbauen, dass eine Funktionsdefinition und ein
# Aufruf der Funktion entstehen? Die User-Eingabe soll außerhalb
# der Funktionsdefinition stehen.
#
# Infos zu "Mocking Spongebob": https://knowyourmeme.com/memes/mocking-spongebob

your_input = input("Please enter a word: ")

# Bis wir das Gegenteil feststellen, gehen wir davon aus,
# dass der Text im Spongebob-Case vorliegt:
sponge_case = True

last_character_is_upper = your_input[0].isupper()

# Schleife: Alle Zeichen ab Index 1 prüfen und mit dem
# jeweils vorigen Zeichen vergleichen
for c in your_input[1:]:

    # SpongeCase liegt nur dann vor, wenn die Großschreibung
    # jedes Zeichens sich von der des Zeichens davor unterscheidet!
    if c.isupper() == last_character_is_upper:

        # Zwei aufeinanderfolgende groß- bzw. kleingeschriebene
        # Zeichen sind nicht erlaubt!
        sponge_case = False

    # die letzte beobachtete Großschreibung merken wir uns,
    # um das nächste Zeichen damit zu vergleichen
    last_character_is_upper = c.isupper()

if sponge_case:
    print("""#>          *
#>          *
#>          ----//-----
#>          \..C/--...--/ \   `A
#>          (@ ) ( @) \   \// |w
#>          \          \   \---/
#>          HGGGGGGG   \   /`
#>          V `-----`-`-`
#>          <<    <<
#>          ###   ###""")
else:
    print(your_input)
```

In []:

```
#####  
#           Übungsaufgabe zu Funktionen           #  
#####  
#  
# Schreiben Sie jetzt eine Funktion, die einen String als Argument  
# bekommt und diesen String im Spongebob-Case zurückgibt.  
# Sie können hier die Stringmethoden .upper() und .Lower() verwenden.  
  
##### hier Ihren Code einfügen! #####
```