

Table of Contents

Cheat Sheet 2.0



Choose Your Poison!

- Mengen
- String Formatting
- Reguläre Ausdrücke
- Tupel
- Comprehensions
- Rekursion
- Fehlerbehandlung

Mengen

- Wie Dictionarys werden Sets durch "{}" markiert.
- Sets/Mengen sind Sequenzen, die wohl-unterschiedene Elemente enthalten. D.h., in einem Set können keine Elemente wiederholt auftreten.
- Sets sind unsortiert.
- Sets sind veränderbar (Mutability)

In [1]:

```
set1 = {1,2,3,3,3}
print(set1)
```

In [1]:

```
set1 = {1,2,3,3,3} # Nur wohlunterschiedene Elemente
print(set1)
print("-----")
set1_anders = set1 # set1_anders zeigt auf den selben Wert wie set1
set1_anders.add(4) # Wir fügen dem Set etwas hinzu
print("-----")
print("Das ist ist das veränderte Set:\n {}".format(set1_anders))
print(set1 is set1_anders) # Sets sind veränderbar
```

Was passiert hier?

In [1]:

```
set2 = {}
print(set2)
print(type(set2))
```

Ein leeres Set kann mit Hilfe der Set-Funktion erstellt werden.

In [1]:

```
set3 = set()
print(type(set3))
print(set3) # Auch hier keine Verwechslungsgefahr mit Dictionarys
```

Auch andere Sequenzen können in Sets umgewandelt werden

In [1]:

```
set4 = set([1,1,2,3,4,5,5])
print(set4) # Keine Wiederholungen
set5 = set({"one": 1, "two": 2, "three": 3})
print(set5) # Die Keys sind wohlunterschieden
set6 = set("Mengen in Python")
print(set6)
print(set6)
```

Ab Python 3.7 bleibt die Einfügereihenfolge erhalten. Sets sind aber weiterhin unsortiert; es gibt keine Indizierung und deshalb auch kein Slicing bzw. andere damit verbundene Operationen.

Set-Methoden

- Anzahl der Elemente einer Sets bestimmen

In [1]:

```
set7 = set([1,1,2,3,4,5,5])  
  
number_of_elements_set4 = len(set7)  
print(number_of_elements_set7)
```

- Element hinzufügen

In [1]:

```
set7 = set([1,1,2,3,4,5,5])  
  
set7.add("four")  
print(set7)
```

- Elemente entfernen

In [1]:

```
set7 = set([1,1,2,3,4,5,5])  
  
set7.discard("four")  
set7.discard(6)  # Alles gut, obwohl 6 nicht in der Menge ist  
print(set7)  
print("-----")  
set7.remove(5)  
#set7.remove(6)  # Hier passiert's: remove versucht ein Element zu entfernen, das es ni  
cht gibt, und wirft deshalb einen Fehler  
print(set7)
```

Discard entfernt ein Element **falls** es vorhanden ist. Remove ist hartnäckig und versucht das angegebene Element zu entfernen und **wirft einen Fehler** falls das Element nicht existiert.

- Ein Set um ein weiteres Set erweitern

In [1]:

```
set7 = set([1,1,2,3,4,5,5])  
set8 = set([4,5,6,7,8])  
  
set7.update(set8)  
print(set7)
```

Auch hier wird `set7` verändert. Es wird kein neues Objekt erstellt.

- Schnitt von Mengen

In [1]:

```
set6 = set([2,3,4])
set7 = set([1,1,2,3,4,5,5])
set8 = set([4,5,6,7,8])

set9 = set6.intersection(set7)
set10 = set6 & set7
print(set9)
print(set10)
print("-----")
print(set9 is set10)
print("-----")
set11 = set6 & set7 & set8
set12 = set6.intersection(set7, set8)
print(set11)
print(set12)
```

Beide Varianten `intersection` und `&` erfüllen den gleichen Zweck. Beide können mehrere Sets miteinander schneiden. Analog verhält es sich auch bei:

- Vereinigung

In [1]:

```
set6 = set([2,3,4])
set7 = set([1,1,2,3,4,5,5])

set12 = set6.union(set7)
set13 = set6 | set7
print(set12)
print(set13)
```

- Differenzmenge

In [1]:

```
set6 = set([2,3,4])
set7 = set([1,1,2,3,4,5,5])

set14 = set7.difference(set6)
print(set14)
```

- Teilmengentest

In [1]:

```
set6 = set([2,3,4])
set7 = set([1,1,2,3,4,5,5])

is_it = set6.issubset(set7)
print(is_it)
```

Out[3]:

String Formatting

- Mit Hilfe der Format-Methode können wir in einen String, der {} Platzhalter enthält, Daten verschiedenen Typs an den entsprechenden Stellen in den String einfügen.
- Die einzufügenden Daten werden dabei automatisch zu Strings umgewandelt.

In [1]:

```
my_fillable_string = "Hier kommt das erste Datum: {}. Jetzt noch eins: {}. Alle guten Dinge sind {}."
print(my_fillable_string)
print("-----")
print(my_fillable_string.format("08.01.2019", [8, 1, 2019], 3))
print("-----")
print(my_fillable_string)  # Der String mit den Platzhaltern wird nicht verändert
```

- Die Daten werden in der Reihenfolge in den String eingefügt, in der sie der Methode als Argument bereitgestellt werden, FALLS nicht anders definiert.
- Die Reihenfolge kann aber auch explizit angegeben werden. Innerhalb der Platzhalter {} muss dafür der Index des gewünschten Arguments genannt werden.

In [1]:

```
another_format_string = "Hier kommt erst das zweite Datum: {1}. Und nun das erste: {0}"
print(another_format_string.format("Das Erste", "Das Zweite"))
```

Die Format-Methode eignet sich auch sehr gut um Daten strukturiert auszugeben

In [1]:

```
daten_matrix = [{"c1", "c2", "c3"}, [1,0,0], [0,1,0], [0,0,1]]
#daten_matrix = [[1,0,0], [0,1,0], [0,0,1]]
struktur = "{} {:>5} {:>5}"

for l in daten_matrix:
    print(struktur.format(*l))
```

In Zeile 9 des letzten Code Schnipsels wird der Asterisk-Operator (*) benutzt. Der Format-Methode wird eine Liste übergeben, deren Elemente mit Hilfe des * einzeln als Argumente verfügbar gemacht werden.

(<https://docs.python.org/3.7/tutorial/controlflow.html#unpacking-argument-lists>

(<https://docs.python.org/3.7/tutorial/controlflow.html#unpacking-argument-lists>)).

Reguläre Ausdrücke

Zeichenarten (Auswahl)

- `.` => Repräsentiert ein beliebiges Zeichen
- `\d` => Ziffern
- `\D` => alle Zeichen außer Ziffern
- `\w` => Whitespace (Zeilenumbrüche("\n"), Tabs("\t"), Leerzeichen(" "))
- `\W` => Alle Zeichen außer Whitespace
- `^` => Beginn eines Strings
- `$` => Ende eines Strings
- `\b` => Wortgrenze (Whitespace am Anfang oder am Ende einer alphanumerischen Zeichenkette)
- `\B` => Kein Wortanfang oder -Ende
- `\` => Escape Character (Zum Beispiel kann man mit `\&` die Et/Und-Zeichen finden)

Gruppen und Mengen (Auswahl)

- `[...]` => Menge von Zeichen
- `(...)` => Gruppe von Zeichen
- `^` innerhalb eines Sets (`[^ ...]`) => keine Zeichen aus dem Set
- `(A|B|C)` => A oder B oder C
- `\1` , `\2` , ... , `\n` => Referenz auf Gruppen: `(dum)(di)\1\2` matcht auf "dumdidumdi"

Quantifizierer (Auswahl)

- `+` => das vorherige Element tritt ein- oder mehrmals auf
- `*` => das vorherige Element tritt 0 oder mehrmals auf
- `?` => das vorherige Element tritt 0- oder einmal auf

Suchen mit Regulären Ausdrücken

Um reguläre Ausdrücke benutzen zu können müssen wir erst das entsprechende Modul importieren. Dann können wir

Regulären Ausdruck definieren/entwerfen

In [1]:

```
import re
objekt_string = "Dr. Angela Dorothea Merkel"
# objekt_string = "Annegret Kramp-Karrenbauer"
# objekt_string = "Johnnie Walker"
# objekt_string = "Dr. Oetker"
# objekt_string = "Ben"
re_name = "((Dr.|Prof.|Prof. Dr. )?(([A-Z][a-z]*) ([A-Z][a-z]*)?)? ([A-Z][a-z]*(-[A-Z][a-z]*)?)?)$"
```

- Im Muster sind die akademischen Grade optional: "((Dr.|Prof.|Prof. Dr.))?". Dann folgt eine (verschachtelte) Gruppe für Vornamen und eine Gruppe für Nachnamen.
- Die Vornamen Gruppe ist deshalb verschachtelt, weil wir davon ausgehen, dass Personen mehrere Vornamen haben können: "([A-Z][a-z]*)([A-Z][a-z]*)?". Das Leerzeichen in der zweiten Subgruppe ist notwendig um mehrere Vornamen verbinden zu können.
- Bei Nachnamen erwarten wir nur einen einzelnen zusammenhängenden String (ohne Leerzeichen), aber wir erhalten die Option für Doppelnamen, die mit einem Bindestrich verbunden sind: "[A-Z][a-z]*(-[A-Z][a-z]*)?\$".

Muster kompilieren

In [1]:

```
muster = re.compile(re_name)
```

Suchen

Die Search-Methode gibt den ersten Match im Objekt-String zurück. Falls kein Match gefunden wird, wird None zurückgegeben.

In [1]:

```
erster_treffer = re.search(muster, objekt_string)
print(erster_treffer)
print(erster_treffer.group(4))
if erster_treffer != None: # Hier wird überprüft ob ein Match gefunden wurde.
    for i in range(7): # Hier werden alle 7 Gruppen ausgegeben.
        if erster_treffer.group(i): # Hier wird überprüft, ob die jeweilige Gruppe einen nicht-leeren String enthält.
            print(erster_treffer.group(i).rstrip()) # Per group(<index>) kann auf Gruppen zugegriffen werden.
```

Mit Hilfe der Methode findall() können Sie alle Strings finden auf die Ihr regulärer Ausdruck matcht.

```
alle_treffer = re.findall(muster, objekt_string)
print(alle_treffer)
```

- Rückgabe der Findall-Methode: Liste von Tupeln, Jedes Tupel entspricht einem Treffer.
- Die Tupel Elemente entsprechen den gematchten Gruppen.
- Alle optionalen Gruppen die nicht gematcht wurden, werden als leerer String ausgegeben.

Ersetzen

- Mit Hilfe von `re.sub()` können sie nach einem regulären Ausdruck suchen und diesen einen beliebigen String ersetzen
- Der optionale Parameter `count` bestimmt wieviele Matches ersetzt werden sollen. Wird `count` auf 0 gesetzt oder nicht angegeben, werden alle Matches ersetzt.
- Sie können bei der Definition des einzusetzenden Strings auch gematchte Gruppen wieder aufgreifen.

In [1]:

```
string_mit_ersetzungen1 = re.sub(muster, "test", objekt_string, count = 0 )
print(objekt_string)
print(string_mit_ersetzungen1)
print("-----")
string_mit_ersetzungen2 = re.sub(muster, "\\4" , objekt_string, count = 0 )
print(string_mit_ersetzungen2)
```

Beachten Sie: Wenn Sie auf eine gematchte Gruppe referenzieren dann müssen sie den Escape Character `\` escapen, d.h. Sie müssen `\\` benutzen damit der Python-Interpreter diesen Ausdruck als einfachen Backslash auffasst. Andernfalls wird beispielsweise `\\4` durch die zweistellige Hexadezimaldarstellung des Integer 4 ersetzt

Aufgabe

1. Finden Sie mit Hilfe von VS Code und einem regulären Ausdruck alle Code-Schnipsel in diesem Org-Dokument und fügen Sie diese dann in ihr Cheat Sheet aus der letzten Wiederholungssitzung ein.
2. Do you even REPL?
 - Die REPL. Was ist das?
 - Wo/Wie kann ich eine REPL benutzen?
 - Wofür sollte ich eine REPL benutzen?

Tupel

- Tupel, markiert durch `()`, sind ein Sequenz-Datentyp.
- Sie sind nicht veränderbar.
- In Tupeln bleibt die Reihenfolge der Elemente erhalten
- Tupel sind indiziert. Man deshalb einzelne Elemente mit Hilfe ihres Index aufrufen.
- Slicing funktioniert wie bei Listen
- Die Verarbeitung von Tupeln ist effizienter als die von Listen, da Sie nicht veränderbar sind.

In [1]:

```
tuple1 = ("es", "sg", "3.", "n")
print(tuple1)
print(tuple1[-2])
print("-----")
tuple2 = tuple1[1:3]
print(tuple2)
```

Comprehensions

List Comprehensions

- Basierend auf einer Sequenz von Daten (Liste, Menge, Tupel,...)
- Jedes Element der Ausgangssequenz wird verarbeitet und dann einer neuen Liste hinzugefügt
- Die Aufnahme in die neue Liste kann mit Bedingungen beschränkt werden.

In [1]:

```
list1 = ["Element" + str(i) for i in range(3) if i < 2]
print(list1)
```

Set Comprehensions

Set-Comprehensions funktionieren ähnlich wie List-Comprehensions, mit den Besonderheiten, die für Sets gelten.

In [1]:

```
my_string1 = "Set comprehensions ähneln von der Form her bis auf die Klammern den List  
comprehensions Beachten Sie dass Sets ungeordnet sind und jedes Element nur einmal ent  
halten ist"  
set1 = {c for c in my_string1.replace(" ", "") if c not in "äöü"}  
print(set1)
```

In dieser Comprehension wird die Eingangssequenz zuerst durch eine String-Methode verändert und dann verarbeitet.

Dict Comprehensions

- Sowohl die Wahl des Schlüssels als auch die Wahl des zugehörigen Werts kann an Bedingungen geknüpft werden.
- Darüber hinaus können auch noch weitere Bedingungen als Filter benutzt werden.

In [1]:

```
list2 = ["car", "Cars", "Office", "offices", "Actor", "actors", "Actress", "actresses"]
print(list2)

print("-----")
sg_pl_dict1 = {item : "pl" if item.endswith("s") and not item.endswith("ss") else "sg"
for item in list2}
print(sg_pl_dict1)

print("-----")
sg_pl_dict2 = {item.lower() if item[0].isupper else item: "pl" if item.endswith("s") and
not item.endswith("ss") else "sg" for item in list2}
print(sg_pl_dict2)

print("-----")
sg_pl_dict3 = {item.lower() if item[0].isupper else item: "pl" if item.endswith("s") and
not item.endswith("ss") else "sg" for item in list2 if not item.startswith(("a", "A"
))}
print(sg_pl_dict3)
```

Tuple-Comprehension?

- Für Tuple gibt es keine Comprehension-Syntax
- Die Generator-Expression: Nach und nach werden die Elemente aus der Range-Funktion bereitgestellt. (Von durch Generator-Expressions wird nach der Verarbeitung kein Objekt in den Speicher geschrieben (ressourcen-schonend, <https://www.python.org/dev/peps/pep-0289/> (<https://www.python.org/dev/peps/pep-0289/>)).
- Die Tuple-Funktion nimmt die Elemente entgegen und erstellt ein Tupel.

In [1]:

```
no_tuple_comprehension = tuple(i for i in range(10))
print(no_tuple_comprehension)
```

Rekursion

- Rekursive Funktionen rufen sich nach dem erstmaligen Aufruf im Programmcode immer wieder selbst auf um eine Aufgabe zu lösen
- Es ist entscheidend, dass eine Abbruchbedingung den zirkulären Aufruf der Funktion verhindert
- Die Funktion unten löst das Potenzieren mit ganzzahligen positiven Potenzen rekursiv.

In [1]:

```
def potenzierer(basis, potenz):  
    if potenz > 1: # Rekursionsschritt  
        return basis * potenzierer(basis, potenz-1)  
    elif potenz == 1: # Abbruchbedingung  
        return basis  
    elif potenz == 0: # Regel falls die Potenz 0 ist  
        return 1  
    else:  
        return "I'm sorry, I can't do that Dave"  
  
print(potenzierer(2,3))  
print(potenzierer(2, 0.5))
```

Try & Except (& Else & Finally)

Fehler Typen:

- SyntaxError
- IndentationError
- NameError
- AttributeError
- TypeError
- ValueError
- KeyError
- FileNotFoundError

Fehlerbehandlung

- "It is Easier to Ask for Forgiveness than Permission"
- Versuchen Sie immer einen Fehlertyp zu antizipieren, wenn Sie mit `try` und `except` Fehler abfangen.
- Sie können mehrere Except-Statements nutzen um verschiedene Fehlertypen abzufangen

In [1]:

```
Misch_Masch = ["123", "drei", "456", None, True]
Fehlermeldung = "Das folgende Element wurde aus der Liste auf Grund eines {} entfernt: {}"

for element in Misch_Masch:
    try:
        element = int(element)
    except ValueError:
        Misch_Masch.remove(element)
        print(Fehlermeldung.format(ValueError, element))
        print("-----")
    except TypeError:
        Misch_Masch.remove(element)
        print(Fehlermeldung.format(TypeError, element))
        print("-----")
    else:
        print("Das folgende Element wurde in einen Integer-Wert umgewandelt: {}".format(element))
        print("-----")
    finally:
        print("So sieht die Liste im Moment aus: {}".format(Misch_Masch))
```