Python

Zusammenfassung

Dario Scheuber / Quelldateien

Inhaltsverzeichnis

Variablen

Ausgabe

Eine einfache Variante um Ausgaben zu machen

```
print("Text")             #Text
print("Text1","Text2")    #Text1 Text2
print(10)                 #10
print("Text: ",10)        #Text: 10

name = "Hans"
alt = 18
print(f"Der {name} ist {alt} Jahre alt")
#Der Hans ist 18 Jahre alt
```

Variablen

Zuweisungen mit dem = Zeichen

```
x = 10                    #in x Wert 10 gespeichert
y = x                     #in y wird x gespeichert Wert 10
print(x,y)                #10 10
print("Text: ",10)        #Text: 10
```

! Variablennamen

Können nur Buchstaben, Zahlen oder Unterstriche im namen verwendet werden. Zusätzliche Sonderzeichen sind nicht erlaubt.

```
x = 10                    #Integer (int) gerade Zahlen <class 'int'>
x = 11.5                  #Floating Point (float) gleitkomma Zahlen
                          #<class 'float'>
x = "Text"                #Zeichenketten (String) <class 'str'>
x = True                  #Boolscher Wert(Boolean) True/False <class 'bool'>
type(x)                   #gibt Typ zurück: <class 'bool'>
```

Typen Umwandlung

```
x = 10                    #int: 10
int(x)                    #int: 10
float(x)                  #float : 10.0
str(x)                    #String: 10
bool(x)                   #bool: True

int(10.6)                 #int: 10 --> Kommawert wird abgeschnitten
bool(-4)                  #bool: True
```

```
bool(0)                 #bool: False
bool(0.0)               #bool: False
```

Operatoren

Operation	Befehl
Addition	+
Subtraktion	-
Multiplikation	*
Division	/
Exponent	**
Modulo (Restberechnung)	%
Floor Division	//
Bitwise XOR	^

```
print(2*2)              #4
print(2**3)             #8
print(1.0/2.0)          #0.5
print(1/2)              #0.5
print(16%5)             #1
print(7//2)             #3
print(1^3)              #0b1 ^ 0b11 = 0b10 = 2
```

> **Hinweis**
>
> Alle Division wurden in Python mit Float realisiert

Zahlenformate

```
print(bin(10))          #Binär: 0b1010
print(oct(10))          #Oktal: 0o12
print(hex(10))          #Hexadezimal: 0xa
```

Strings

In Python sind Strings mit Doppelten- `"Text"` und Einfachen-Anführungszeichen `'Text'` realisiert.

```
print("Text1"+"Text2")  #Text1Text2
print("Text1","Text2")  #Text1 Text2
print("Text"*5)         #TextTextTextTextText
print("Text\tText")     #Text Text
print("Text\nText")     #Text
                        #Text
print("\U0001f600")     #Unicode: :) (Smile)
```

Sonderzeichen	Befehl
Tab	`\t`
NewLine	`\n`

Sonderzeichen	Befehl
Backslash	\\
“-Zeichen	"
’-Zeichen	'

Raw Text

Mit dem r Vorzeichen wird der nachfolgende String genau so Interpretiert.

```
print(r'C:\some\name')    #C:\some\name
print('C:\some\name')     #C:\some
                          #ame
```

Zeichenliterale

Zeichenliterale sind Strings, welche die Einrückungen etc. übernehmen.

```
print("""
Usage: Text[options]
      -t
      -h Host
""")                      #
                          #Usage: Text[options]
                          #      -t
                          #      -h Host
```

Indexieren von Strings

> **!** Wichtig
>
> Strings werden mit folgender notations Indexiert: <Stringname>[<start>:<stop>:<step>]. Negative Indexe wird von hinten gezählt, sowie bei einem negativen Step-Index wird die Zeichenkette rückwärts gezählt.

```
word = "Hello this is a Test"
print(word[0])            #H
print(word[1])            #e
print(word[2])            #l
print(word[3])            #l
print(word[4])            #o

print(word[0:5])          #Hello
print(word[5:9])          # thi
print(word[5: ])          # this is a Test
print(word[ :10])         #Hello this

print(word[::2])          #Hloti saTs
print(word[0:10:2])       #Hloti

print(word[-5:])          # Test
print(word[5:-1])         # this is a Tes
print(word[::-1])         #tseT a si siht olleH
```

```
print("Hello" in word)    #True
print(leng("Test"))       #4
```

String Funktionen

Alle Strings unterstützen auch einige **String – eigene Funktionen**.

```
name = "Wolfgang"
# Gross und Kleinschreiben
print(name.upper())         #WOLFGANG
print(name.lower())         #wolfgang

string_path = "//path//to//a//file//deep//in//a//folder"
string_path.split("//")
#['', 'path', 'to', 'a', 'file', 'deep', 'in', 'a', 'folder']
last_folder = string_path.split("//")[-1]
#'folder'
string_path.count("//")     #8
```

Schleifen

IF-Statement

Das `if`-Statement wird ausgeführt soblad das die condition true ist

```
if condition == 10:           #Wenn condition gleich
                              #10 ist dann
  print("Text1","Text2")      #Text1 Text2
```

Vergleichsoperator	Bedeutung
==	ist gleich?
!=	nicht gleich?
>	grösser als
<	kleiner als
>=	grösser gleich
<=	kleiner gleich

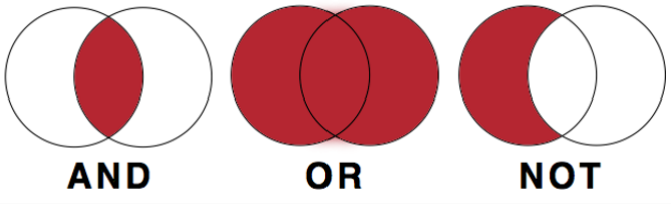
if - elif - else Statement

Bei mehreren `if`-Statements wird jedes einzelne überprüft auch wenn eines `True` war. Das `elif`- oder `else`-Statement wird nur ausgeführt, wenn das `if`-Statement `false` war.

```
if x < 10:
  print("x kleiner als 10")
elif x > 10:
  print("x grösser als 10")
else:                         #Wenn kein Statement True ist
  print("x ist 10")
```

Verknüpfen von Bedingungen

```
x = True
y = False
z = x and y                    #False
z = x or y                     #True
z = not y                      #True
```



Ablaufdiagramm / Flussdiagramm

Ein Ablauf- oder Flussdiagramm kann verwendet werden den Prozess /Ablauf eines Algorithmus, Programms, etc. graphisch zu beschreiben.

Baublock	Diagrammzeichen
Terminal/Terminator	
Prozess	
Entscheidung	
Datensatz oder Input/Output	
Flussrichtungspfeil	
Off-Page-Konnektor/Referenz	

While-Schleife

Die `while` Schleife kann verwendet werden, um einen Code Block mehrfach auszuführen, bis eine bestimmte Bedingung eingetreten ist.

```
i = 0
while i < 10:
  print(i)                     #0\n 1\n 2\n 3\n 4\n ...
  i += 1

while True:                    #Endlosschleife
  print("Endlosschleife")
```

continue

'continue': Wenn continue aufgerufen wird, wird der **Rest** des Code Blockes **übersprungen** und es beginnt ein neuer Durchlauf der Schleife

break

'break': Wenn break aufgerufen wird, wird die **Schleife** sofort **beendet**.

For-Schleife

For in Python wird etwas anders behandelt als in anderen Programmiersprachen. In Python wird for verwendet, um **alle Elemente einer Sequenz** zu bearbeiten.

```
for elem in range(10):
  print(elem)                  #0\n 1\n 2\n 3\n 4\n ...

for c in "Word":
  print(c)                     #W\n o\n r\n d\n

for i in [1,3,5]:
  print(i)                     #1\n 3\n 5\n
```

range

'range': produziert eine Sequenz von (default 0) bis zu dem eingegebenen Wert (10).

```
list(range(10))                #0,1,2,....9
```

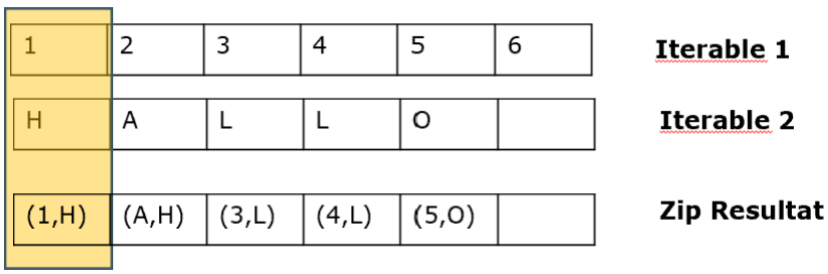
enumerate

'enumerate' gibt einen Index einer Iterable. `enumerate` liefert dazu ein Tuple (zwei Werte). Diese Werte sind index und Element von einer Iterablen.

```
for i,elem in enumerate("word"):
    print(f"{i}th letter is a {elem}")
    #0th letter is a w
    #1th letter is a o
    #2th letter is a r
    #3th letter is a d
```

zip

'zip' um gleichzeitig zwei oder mehr gleichlange Listen zu iterieren. `zip` aggregiert Element für Element von mehreren Iterables und gibt jeweils ein Tuple mit einem Element von allen Iterables zurück.



```python
for elem1,elem2 in zip([1,2,3,4,5,6],"hello"):
    print(elem1,elem2)
    #1 h
    #2 e
    #3 l
    #4 l
    #5 o
```

> **!** zip-Länge
>
> zip gibt nur so lange Tuples aus, wie alle Iterables Elemente haben.

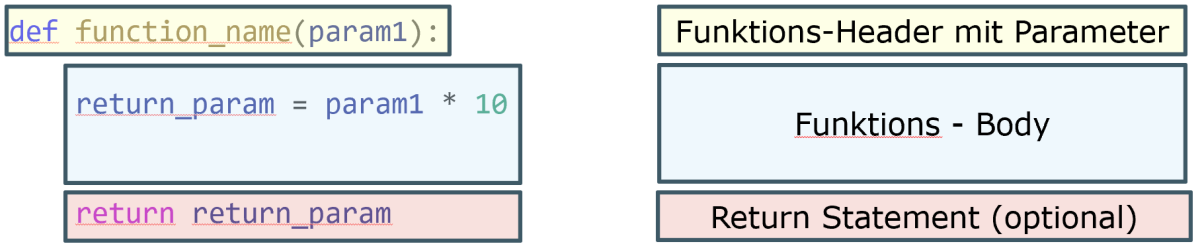
zip und enumerate

```python
for idx, tup in enumerate(zip(word_1,word_2)):
    print(f"{idx}th letters of words are: {tup[0]} \t {tup[1]}")
```

> **!** zip und enumerate
>
> Wenn zip und enumerate gleichzeitig angewendet werden, werden leider NICHT 3 Werte zurückgegeben, sondern 1 Index und 1 Tuple (2 Werte) zurückgegeben.

Funktionen

Funktionen sind Unterprogramme, die aus dem Programm aufgrufen werden können, um danach das Programm fortzusetzen.



Der Gültigkeitsbereich, auch **Scope** genannt, beschreibt in welchem Bereich Variablen, Funktionen, etc. ersichtlich sind.

Funktionsaufruf

```python
    #Main programm
    x = 10
    y = foo(x)

    #Funktion
    def foo(n):
        return n *10
```

Funktionsaufruf mit default Values

Parameter können auch mit Default Werten versehen werden. Dies macht die betroffenen Parameter **optional**. Heisst wenn sie nicht beim Aufruf angegeben werden, wird der Default – Wert verwendet.

```
    #Main programm
    x = 10
    y = foo(x)          #gleiche auswirkung wie unten
    y = foo(n=x)        #

    #Funktion
    def foo(n,word = "hello"):
        print(word)
        return n *10
```

Funktionsaufruf mit Typehints

Zudem ist es möglich, dem Interpreter zu sagen, was man für einen Datentyp man bei den verschiedenen Funktionsparametern erwartet. Dies kann man mit **:datatype** definieren

```
    #Main programm
    x = 10
    y = foo(x)          #gleiche auswirkung wie unten
    y = foo(n=x)        #

    #Funktion
    def foo(n:int,word:str = "hello"):
        print(word)
        return n *10
```

> **!** NICHT FORCIEREND
>
> Diese Type Hints sind NICHT FORCIEREND. Es sind jeweils nur Hinweise für den Programmierer. Ich kann den Variablen trotzdem noch andere Datentypen zuweisen, ohne dass es Fehler angezeigt wird. Der Code kann sogar funktionieren.

Rückgabewerte können auch mit einem Type Hint `->int` versehen werden.

```
def add_binary(a:int, b:int) -> str:
    binary_sum = bin(a+b)[2:]
    return binary_sum           #Wird als String zurückgegeben
```

Docstring

Funktionen, welche man definiert, sollten **immer dokumentiert** sein, damit man auch später noch weiss, was die Funktion machen sollte. Dies ist vor allem dann wichtig wenn man komplexere Programme mit vielen Funktionen schreibt. Generell dokumentiert man Funktionen mit einem DocString nach dem Header, nach folgender Konvention.

```
def calc_rect_area(l:float, w:float):
    """
    Calculate the Rectangle Area from a rectangle with width w and length l.
       Parameters:
```

```
        l (float) : Lenght of the Rectangle
        w (float) : Width of the Rectangle
    Returns:
        rect_area (float) : rectangle area (l*w)
    """
    rect_area = l * w
    return rect_area
```

Docstring

Mit `help(my_func)` kann die Docstring abgefragt werden, zusätzlich sind diese mit dem Mouse hover sichtbar

Rekursion

Es ist möglich eine Funktion von innerhalb derselben Funktion aufzurufen. Dies nennt sich **Rekursion**. (sich selbst aufrufen)

```
def recursion(x):
    # End Condition
    if condition:
        return fixed_value

    # Recursive Condition
    if condition:
        return recursion(x-1)
```

Datentypen

F-Strings

Wenn Sie bei normalen Strings Variablen einfügen wollen müssen Sie dies mühsam mit der String-Concats machen. Zusätzlich erlauben F-Strings das spezifizieren des gewünschten Formates. Die definition von F-Strings ist auch **schneller**.

```
normal_string = "string plus"+str(variabel)
print("string plus ",variabel)


f_string = f"string plus {variabel}"
print(f"string plus {variabel}")
```

Tipp

Das kann auch mit mehreren Variablen gemacht werden

```
#runden (4 stellen nach dem Komma)
pi = 3.141592653589793
print(f"{pi:.4f}")      #3.1415
#0-padding (5 stellen vor dem Komma)
print(f"{12:05}")       #00012
```

```
#binar
print(f"{3:b}")         #11
#hex
print(f"{11:x}")        #b
#oktal
print(f"{11:o}")        #13
```

Sequenzen

unveränderbare Sequenzen (non-mutable)

- Zeichenketten (Strings)
- Tuples

Eigenschaften:

- keine neuen Elemente einfügbar
- Werte von Elementen nicht veränderbar

Veränderbare Sequenzen (mutable)

- Listen

Eigenschaften:

- neue Elemente können eingefügt werden
- Elemente können entfernen werden
- Werte von Elementen können verändert werden

Tuples

Tuples sind eine geordnete Sequenz von Elementen und verfügt deswegen über alle standard Sequenz-Funktionalitäten. Ein Tuple wird mit normalen Klammern ( ) definiert. ::: callout-note Ein Tuple Elemente von verschiedenen Datentypen besitzen. Klammern sind optional. :::

```
x = 1
my_tuple = (1,2,4,"word",x)
print(my_tuple[0])              #1
for elem in my_tuple:
    print(elem)                 #1
                                #2
                                #4
                                #...
1 in my_tuple                   #True
(1,2) in my_tuple               #False: (1,2) ist ein Tuple in einem Tuple
len(my_tuple)                   #5

my_tup = 1,2                    #Klammern optional
x,y = my_tup                    #x=1 y=2
```

> **!** Wichtig
>
> Aber Achtung Tuples sind nicht veränderbar!

Falls jedoch **nicht alle Werte eines Tuples benötigt** werden, kann die **Tuple Entpackung** angewendet werden.

```
elem1 , *rest = my_tuple
print(elem1)                    #1
print(rest)                     #[2, 4, 'word', 1]

first, *middle, last = my_tuple
print(first)                    #1
print(middle)                   #[2, 4, 'word']
print(last)                     #1
```

! Wichtig

Hierbei sind die **Variabel Namen frei wählbar**, eine muss lediglich mit * beginnen.

Listen

Eine Liste ist eine **veränderbare, geordnete Sequenz von Elementen.** Eine Liste kann einfach mit `[]` definiert werden.

```
my_list = [1,2,"String",2.0]
my_list[0]                      #1
for elem in my_list:            #1 2 String 2.0
1 in my_list                    #True
#Sind veränderbar
my_list[0] = "a"                #['a', 2, 'String', 2.0]
my_list[0:2] = [30,34]          #[30, 34, 'String', 2.0]
```

! Wichtig

Die Elemente in einer Liste können auch verschiedene Datentypen aufweisen.

Listen Funktionen

```
my_list = [1,2,3,4]

#Element Löschen
elem = my_list.pop(0)           #elem=1 my_list=[2,3,4]

#Element Hinzufügen am Ende
my_list.append(5)               #my_list=[2,3,4,5]

#Hinzufügen an einem Spezifischen Index
my_list.insert(0,"a")           #my_list=["a",2,3,4,5]

#Liste am Ende einer Liste anfügen
my_list.extend([1,2,3])         #my_list=["a",2,3,4,5,1,2,3]

#entfernt die erste Instanz von 1
my_list.remove(1)               #my_list=["a",2,3,4,5,2,3]

#alle Einträge einer Liste löschen
my_list.clear()
```

> **!** Wichtig
>
> Wenn value bei `.remove()` nicht in der Liste ist, gibt es einen Value_Error

Wenn eine **Liste nur Zahlen** (`floats` und `ints`) beinhaltet kann diese ebenfalls mit `.sort()` der Grösse nach aufsteigen sortiert werden. Oder eine Liste **(Datentyp unabhängig)** kann mit `.reverse()` umgekehrt werden.

```
my_list = [5,3.0,2,1,10,30]
my_list.sort()              #[1, 2, 3.0, 5, 10, 30]
my_list.reverse()           #[30, 10, 5, 3.0, 2, 1]
```

List Copy by Reference

```
my_list = [1,2,3]
my_list_2 = my_list
my_list_2[2] = 4
print(my_list)              #[1, 2, 4]
```

> **!** Copy by Reference
>
> Mutables werden mit Copy by Refernce kopiert das heisst das `my_list2` mithilfe eines Zeigers auf die Speicherstelle von `my_list` zeigt. Wenn `my_list2` geändert wird, ändert sich auch `my_list` und umgekehrt!

Um richtig zu Kopieren gibt es die `.copy()` Methode.

```
my_list_2 = my_list.copy()
my_list_2[2] = 5
```

> **i** Hinweis
>
> Die `.copy()` Methode erzeugt ein neues Objekt im Speicher und weist die neue Variabel darauf zu.

Mutable Default Values

```
def add_to_list(item, list_to_add:list=None):
    if list_to_add is None:
        list_to_add = []
    ist_to_add.append(item)

    return list_to_add
```

> **!** Wichtig
>
> Wenn keine Liste übergeben wird, wird dies Detektiert. Darauf wird eine neue (leere) Liste erstellt.

Listen und Schleifen

Python unterstützt die sogenannte **List-Comprehension**, welche es erlaubt in einem 1-Zeiler Listen nach bestimmten Regeln zu erstellen.

```
#List erstellen mit Schleifen
my_list_2 = [i for i in range(10)]
#List erstellen mit Schleifen und einer Funktion
my_list = [my_func(elem) for elem in my_list_2]
#List erstellen mit Bedingungen
my_list = [elem**2 if elem%2==0 else elem for elem in my_list_2]
```

[f(x) if condition else g(x) for x in sequence]

Heisst es wird Funktion f(x) auf jedes Element der Sequenz angewendet wenn die Bedingung wahr ist. Ansonsten wird die Funktion g(x) angewendete.

Sets

Sets sind **ungeordnete** Listen, welche **jedes Element nur einmal beinhalten.** Die Items in einem Set sind **unveränderbar**, jedoch kann man Elemente **hinzufügen und entfernen**.

Syntax: `{}`

```
my_set = {"apple","banana", "pear"}

#Hinzufügen .add()
my_set.add("pineapple")
#{'banana', 'pineapple', 'pear','apple'}

#Bei mehrfach hinzufügen passiert nichts!
my_set.add("pineapple")

#Entfernen .remove()
my_set.remove("apple")
#{'banana', 'pear', 'pineapple'}
```

> **!** Wichtig
>
> Wenn das zu entfernende Item nicht vorhanden ist, gibt es eine KeyError. Bei mehrfach hinzugefühen des gleichen Elementes, passiert nichts.

Vergleich von zwei Sets

B ist ein **Subset** von A ($B \in A$)

A ist ein **Superset** von B ($A \subset B$)

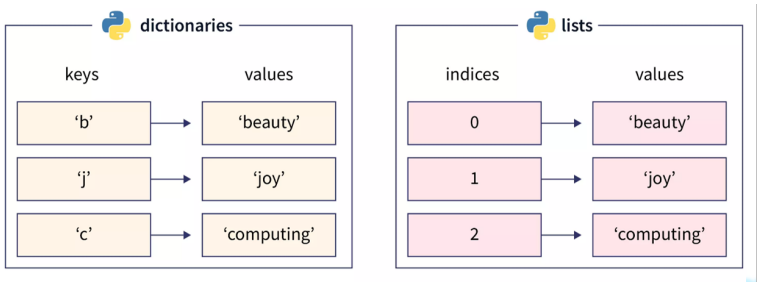
```
my_set_A.issuperset(my_set_B)
my_set_B.issubset(my_set_A)
```

Dictionaries

Ein Dictionary, kurz dict, ist eine **veränderbare (mutable)** Datenstruktur, welche das **Speichern** von **Key-Value-Pairs** erlaubt. Anders als bei Listen, werden hier die Werte (Values) nicht mit Indices sondern mit **Keys angewählt.** Beim Dictionary wie auch beim Set sind die Keys **einzigartig (unique).**

Syntax: `{:}`, `my_dict = {"key1":"value"}`

```
my_dict = {"key1":"value"}
my_dict["key1"]             #'value'
```



Hinweis

Die Keys sowohl Strings wie auch Ints oder Floats sein. Jedoch sollten in der Regel Strings als Keys verwendet werden.

Hinzufügen von Elementen in Dicts

```
my_dict["a"] = "value1"
my_dict["b"] = "value2"
my_dict["a"] = "value3"     #überschreiben von key a
```

nested dict

```
my_dict = {"John":{
    "Age": 48,
    "Gender": "m",
    "Profession":"Carpenter"
    }
}

for key in my_dict:
    print(key)                  #John

for key,value in my_dict.items():
    print(f"{key}: {value}")    #John: {'Age': 48, ...}

"John" in my_dict.keys()        #True
"Age" in my_dict["John"].keys() #True
```

Falls gleichzeitig geprüft werden soll: ob ein Key existiert, was für einen Wert dieser hat, und falls der Key nicht existiert dieser mit einem Default Wert erstell werden soll. Gibt es dafür die `set_default()` Methode:

`print(my_dict.setdefault("Robert",{}))`

`set_default()`

Gibt den Wert von «Robert» zurück, erstellt dieses Key-Value-Pair mit Default Value falls nicht vorhanden

.pop(key, default_return)

Die .pop(key) Methode kann verwendet werden, um den Wert eines Keys zu erhalten und diesen Key zu löschen.

```
my_dict.pop("Robert")
my_dict.pop("Klaus","Key nicht vorhanden")
#'Key nicht vorhanden'
```

Pretty Print

In Python gibt es Module (vorgeschriebener Code) welcher importiert werden kann. Eines dieser Module ist das PrettyPrint Modul (pprint), welches eigene print() Funktionen zur Verfügung stellt.

Einer dieser Funktionen ist das «schöne» Printen von Dictionaries. So kann zB genau spezifiziert werden:

- **Wie viele Levels** eines Dictionaries überhaupt geprinted werden sollen (depth)
- **Wie viel Leerzeichen Abstand** zwischen Levels geprinted werden soll (indent)

```
import pprint
pp = pprint.PrettyPrinter(depth=1, indent = 3)
pp.pprint(my_dict)
```

Dict-Comprehension

`{key:value for elem in iterable}` Hierbei ist oft so dass sowohl key wie auch value eine **Funktion von f(elem)** sind.

```
my_dict_2 ={f"Key_{elem}":f"Value_{elem}" for elem in range(10)}

"""
{
'Key_0': 'Value_0', 'Key_1': 'Value_1', 'Key_2': 'Value_2',
. . .
'Key_9': 'Value_9'
}
"""
```

JSON

Was wenn man nun ein Dictionary in einem File abspeichern will? Dafür gibt es das **JSON Modul.** JSON steht für JavaScript Object Notation und ist ein standardisiertes File Format, um unter anderem Dictionaries abspeichern zu können. Um ein Dict abzuspeichern hat das json Modul die `.dump()` Methode.

```
# JSON Modul importieren
import json
# öffnen eines Files
with open(file="dict.json",mode = "w+") as fp:
# speichern (schreiben) des Dicts in das File
    json.dump(obj = my_dict, fp = fp, indent= 4)
```

Ein Dict kann natürlich auch von einem JSON File erstellt werden. Dafür muss wiederum das JSON Modul verwendet werden. Hierzu hat das json Modul die `.load()` Methode.

```
# öffnen des JSON files
with open(file="dict.json", mode = "r+") as fp:
    # auslesen des Dicts
    my_read_dict = json.load(fp=fp)
pp.pprint(my_read_dict)
```

Generatoren

In Python ist ein **Generator** eine Funktion, **die einen Iterator zurückgibt**, der erst dann eine Folge von Werten erzeugt, wenn man darüber iteriert. Generatoren sind nützlich, wenn wir eine **grosse Folge von Werten** erzeugen wollen, aber nicht alle auf einmal im Speicher ablegen wollen. Ein Generator kann in Python als eine Funktion mit einem yield Keyword definiert werden.

```
def my_generator(n):
    """
    Function to Return Generator to count to n
    """
    i = 0
    while i < n:
        i += 1
        yield i

gen = my_generator(10)
#<generator object count at 0x000001C167494580>
list(gen)
#[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

i yield

Wenn yield aufgerufen wird, wird der **aktuelle Zustand der Funktion gespeichert** und ein Wert zurückgegeben.