Programming Basics

Heute

- Kahoot! Repetition
- Rückblick «Arbeitsfluss Git»
- KISS:
- Arbeiten im Buch
  - Bedingungen
  - Funktionen

PAUSE

- Weiter im Buch
- Module - PIP
- UML zu Code

PAUSE

- Intro Sammeldatentypen
- Arbeiten im Buch

PAUSE

- Sammeldatentypen 2
- Arbeiten im Buch
- Übungen

https://kahoot.it/

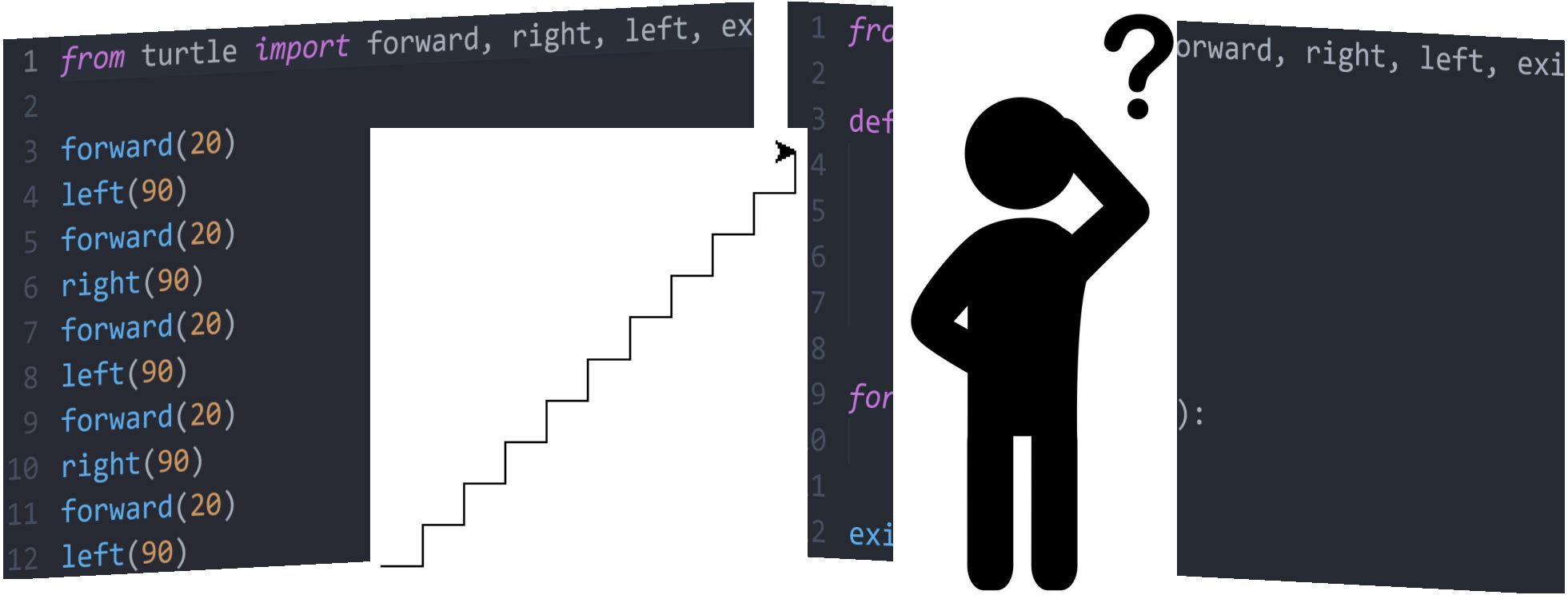
Arbeitsfluss Git + Github

- ***pull***: Hole Commits von Github (**wichtig!**)
- Änderungen an Quellcodedateien machen
- Sobald ein Feature fertig ist, alle dazu nötigen Dateien in *Stage* aufnehmen
- *Commit* mit Versionsnachricht erstellen
- ***pull***: Hole Commits von Github (**wichtig!**)
- ***push***: Lade lokale Commits auf Github hoch

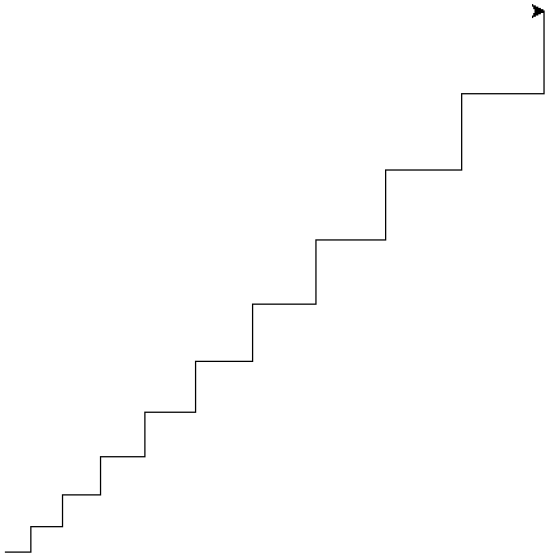
… ab jetzt fliesst die Commit-History in die Beurteilung ein

KISS

keep it simple and stupid



Modularität



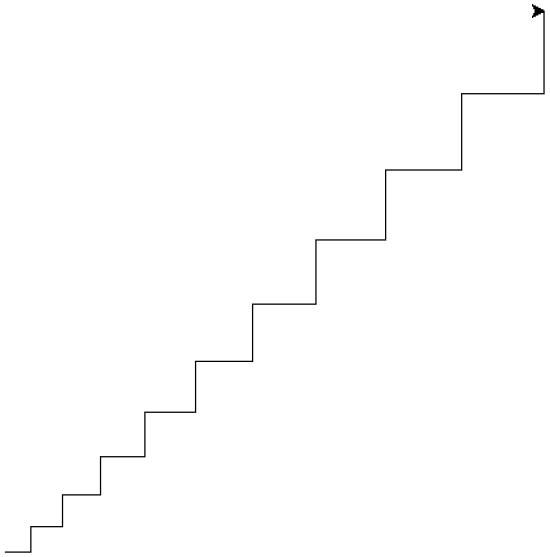
```python
from turtle import forward, right, left, exi

def step():
    forward(20)
    left(90)
    forward(20)
    right(90)

for i in range(0, 10):
    step()

exitonclick()
```

Modularität - Funktionen



```
from turtle import forward, right, left, exi

def step(size):
    forward(size)
    left(90)
    forward(size)
    right(90)

for size in range(20, 70, 5):
    step(size)

exitonclick()
```

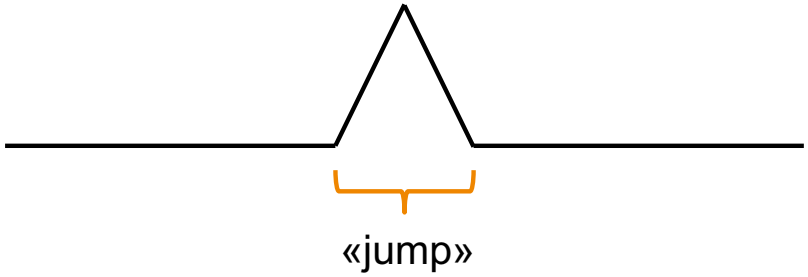
Funktion

Loop

start, end, step

Übung

- Neues Skript unter `programming-basics/code/jump.py`
- Erstelle eine Funktion `jump()`, welche ein Zickzack zeichnet. Danach soll bei `forward()` die Richtung wieder gleich sein wie zuvor.




```
forward()
jump()
forward()
```

Stagen, Committen und pushen…

Übung…

- Erzeuge folgendes Resultat:
  Committen ;)

- Modifiziere `jump()` so, dass die Sprunghöhe angegeben werden kann:
  `jump(20)`
  … und erzeuge folgendes Resultat:

```
def jump(height=20):
    left(60)
    forward(height)
    right(120)
    forward(height)
    left(60)
```

Standardwert definieren:
`jump()` setzt `height` auf `20`,
`jump(50)` setzt `height` auf `50`

Pause

Buch Kapitel 2&3: Flowcontrol & Functions

- Jetzt lesen
- Als Aufgabe zu lesen & bearbeiten
- Optional zu lesen

https://ssr-2024.github.io/programming_basics/

- Dokumentation nicht vergessen…

Module

- Wiederverwendung von existierender Funktionalität

  … im Sport als Modularitätshypothese untersucht

Module

- ... beim Programmieren ein Grundlegendes Konzept:

Baukasten *ispw_functions*

schatzsuche.py

examples > turtle > schatzsuche.py > ...

```python
from ispw_functions import forward, right, found

forward()
right()
forward()
forward()
forward()
found()
```

ispw_functions.py

examples > turtle > ispw_functions.py > ...

```python
import tkinter
# import functions from turtl
from turtle import forward as

setup(500, 500, -1, 0)

def right(angle=90): ...

def left(angle=90): ...

def forward(step=10): ...

def step_down(): ...

def step_up(): ...

def done(): ...

def found(): ...
```

Module – in Python

- In Python ist jede Python-Datei (ohne Spezialzeichen im Namen) ein Modul
- Beim Importieren wird das Modul geladen und Code, der nicht in einer Funktion ist wird ausgeführt! → durch Modul bestimmte Zustände bereits herstellen

hilfreich, wenn mehrere Module importiert

schatzsuche.py

examples > turtle > schatzsuche.py > ...

```python
from ispw_functions import forward, right, found

forward()
right()
forward()
forward()
forward()
found()
```

importiert nur die angegebenen Funktionen

schatzsuche2.py

examples > turtle > schatzsuche2.py

```python
from ispw_functions import * #

forward()
right()
forward()
forward()
forward()
found()
```

importiert alle Funktionen

schatzsuche3.py

examples > turtle > schatzsuche3.py > ...

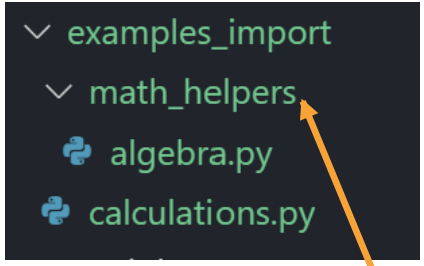
```python
import ispw_functions

ispw_functions.forward()
ispw_functions.right()
ispw_functions.forward()
ispw_functions.forward()
ispw_functions.forward()
ispw_functions.found()
```

importiert die Modul-Referenz

Python Packages

- Packages fassen mehrere Module zusammen (in einem Ordner)



package

calculations.py

examples > examples_import > calculations.py > ...

```python
from math_helpers.algebra import sqrt
print(sqrt(9))
```

package.module

auf Modul in Package zugreifen

algebra.py

examples > examples_import > math_helpers > algebra.py

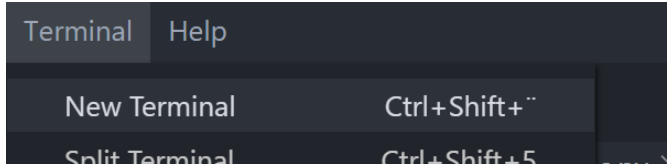
```python
def sqrt(val):
    return val**0.5
```

pip = Modulverwaltung von Python

→ ausführbares Programm (im Terminal)

PIP – Pythons Paketverwaltung

- Modularität:
  Python wird mit den essenziellen Paketen/Modulen ausgeliefert.
  Bei Bedarf werden weitere Pakete hinzugefügt – mit **pip**
- VS Code: Terminal>New Terminal




```
pip install package-name
```

To install a specific version of a package instead of the latest version:

```
pip install package-name==1.0.0
```

To search PyPI for a particular package:

```
pip search "query"
```

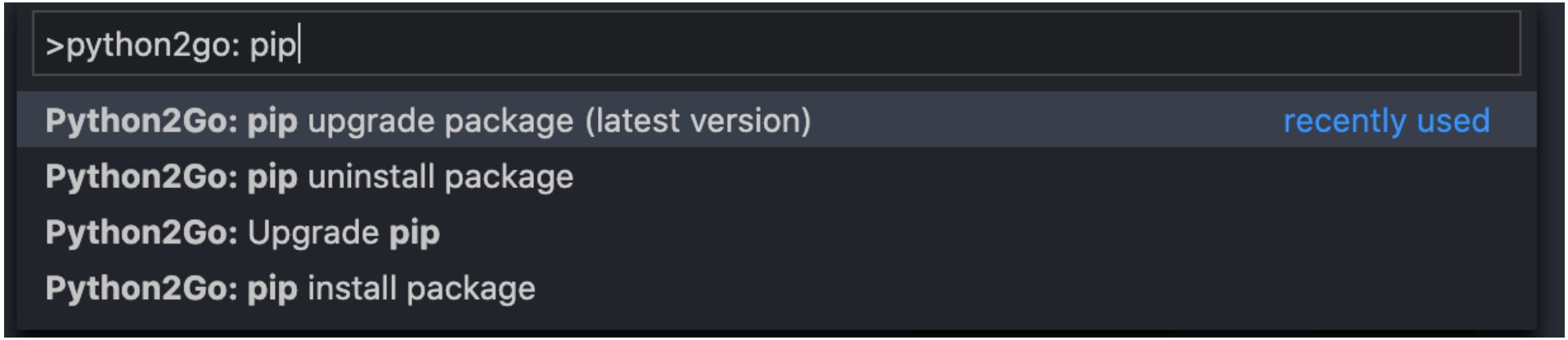
To see details about an installed package:

```
pip show package-name
```

To list all installed packages:

```
pip list
```

PIP – in VS Code



PIP Beispiel

→ über Py Pi Pakete suchen

- Suche auf Google nach «art Python»
- Installiere das Paket und erzeuge deinen Namen als ASCII-Art Text (am besten direkt im interaktiven Modus)
- Als Beispiel im wiki (101_functions.md#Funktionen-von-Bibliotheken-aufrufen) festhalten

```
 _______________
< Hello, world! >
 ---------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

→ Nachdenken

ASCII-Art Texte erstellen: https://www.devdungeon.com/content/create-ascii-art-text-banners-python

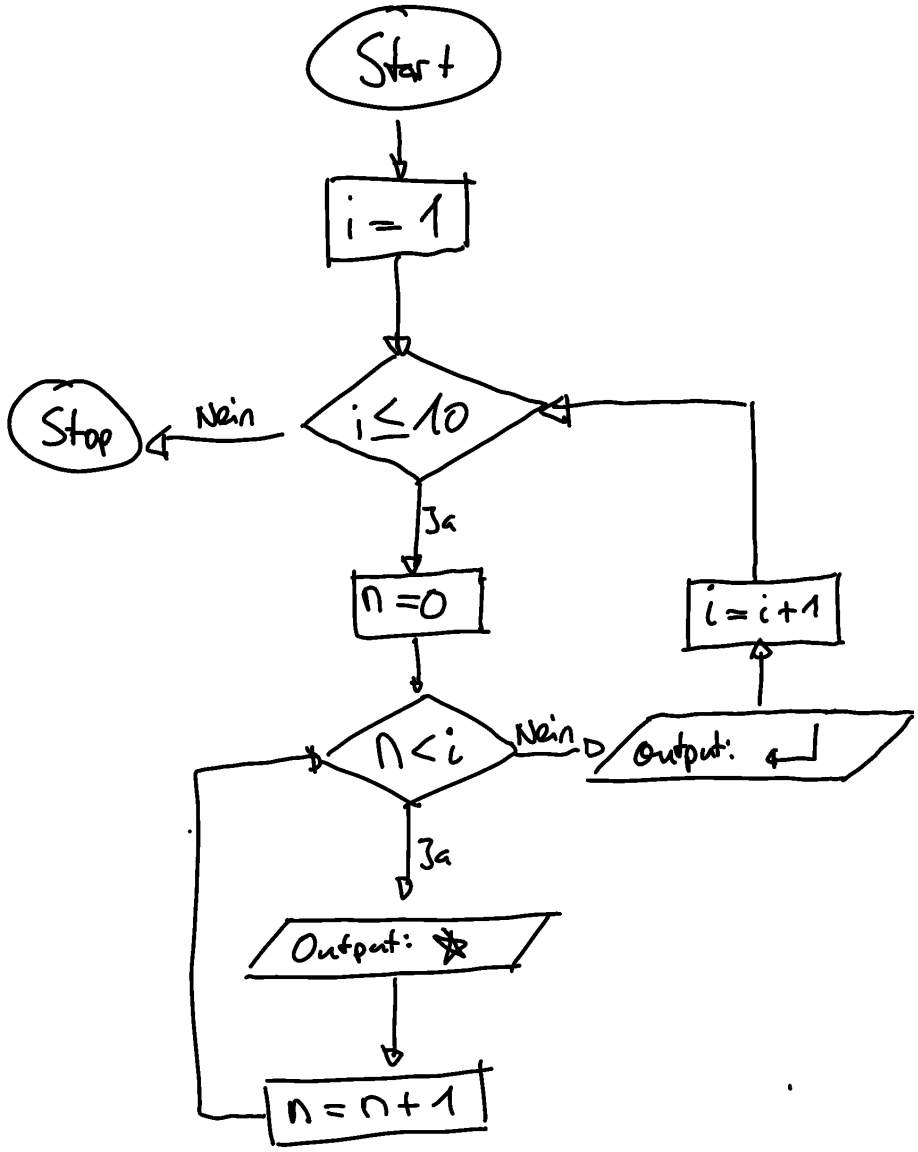
Pause

Übung: UML umsetzen….

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
* * * * * * * * * *
```

Hinweis: Denke an die Keyword-Argumente von print()

```
print('Hello', end='')
```



Kann der Code vereinfacht werden?

```python
i = 1
while i <= 10:
    n = 0
    while n < i:
        print('*', end='')
        n = n + 1
    print('')
    i = i + 1
```

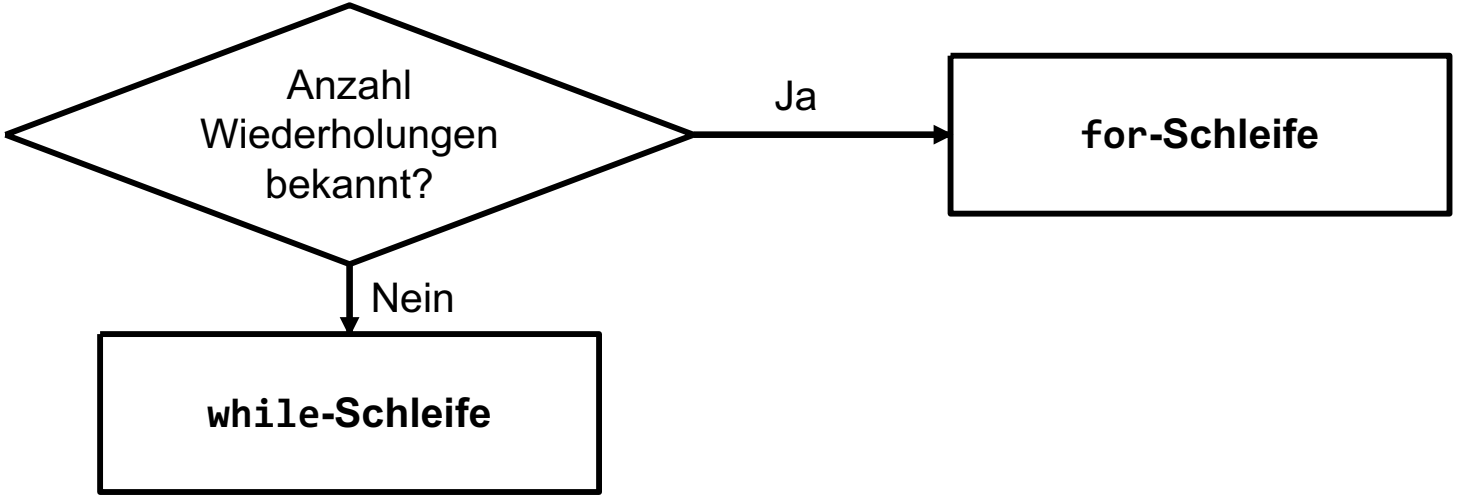
```python
i = 1
while i <= 10:
    print('*' * i)
    i = i + 1
```

```python
for i in range(1, 11):
    print('*' * i)
```

UML zu Code

- 1:1 Übersetzung möglich
- Programmiersprachen bieten oft Vereinfachungen für gängige Aufgaben an
  - Kann die Verständlichkeit erhöhen

- Wann `while`-Schleifen, wann `for`-Schleifen?



Visualisierung von Prozentzahlen

```
33%: * * *
50%: * * * * *
22%: * *
70%: * * * * * * *
12%: *
90%: * * * * * * * * *
```

Ideen?

z.B. so?

geschweifte Klammern?

```python
def visualize(percentage):
    print(f"{percentage}%: {'* ' * (percentage // 10)}")

visualize(33)
visualize(50)
visualize(22)
visualize(70)
visualize(12)
visualize(90)
```

An Zeilen Breakpoints setzen

zum debugg

bei Fehlersuche hilfreich!

Sammeldatentypen!

```
33%: ***
50%: *****
22%: **
70%: *******
12%: *
90%: *********
```

list

```
percentages = [33, 50, 22, 70, 12, 90]
for percentage in percentages:
    visualize(percentage)
```

Listen

- Kapitel 4 lesen, bearbeiten & dokumentieren

https://ssr-2024.github.io/programming_basics/docs/lecture-book-sections/chapter4.html

Committen nicht vergessen…

Für schnelle: Übung 104 beginnen

Pause

Wie könnte man sowas ausgeben?

```
Sucessful Tests:
marc:    **********
remo:    *****
lars:****
lana:    *******
elio:********
andi:    *********
```

Klassiker für Dictionaries!

```
marc:    **********
remo:    *****
lars:****
lana:    *******
elio:********
andi:    *********
```

Halte das Beispiel fest und implementiere die Funktion `print_result(name, percentage)`.

```python
successful_tests = {
    'marc': 100,
    'remo': 61,
    'lars': 42,
    'lana': 87,
    'elio': 70,
    'andi': 90
}

for name, percentage in successful_tests.items():
    print_result(name, percentage)
```

Funktionen dokumentieren:

```python
def avg(array):
    """calculates the average of a list,
    returns 0 for empty lists.

    Parameters
    ----------
    array : list of int, float
        list with numeric data

    Returns
    -------
    float
        the average of the list elements
    """

    n = len(array)
    if n == 0:
        return 0
    return sum(array) / n
```

Nach dem docstring Standard von NumPy:

https://numpydoc.readthedocs.io/en/latest/format.html

Aufgabe: Dokumentiere die Funktion `print_result(name, percentage)`.

```python
successful_tests = {
    'marc': 100,
    'remo': 61,
    'lars': 42,
    'lana': 87,
    'elio': 70,
    'andi': 90
}

for name, percentage in successful_tests.items():
    print_result(name, percentage)
```

Funktionen dokumentieren mit Typisierung

```python
from typing import List


def avg(array: List[float]) -> float:
    """calculates the average of a list,
    returns 0 for empty lists.

    Parameters
    ----------
    array : list of int, float
        list with numeric data

    Returns
    -------
    float
        the average of the list elements
    """

    n = len(array)
    if n == 0:
        return 0
    return sum(array) / n
```

Datentypen importieren

array ist eine Liste von floats

Pfeil definiert Rückgabetyp

Die Funktion gibt ein float zurück

Rückgabewerte: → werden nur Typen (ohne Namen) zurückgegeben

// → nochmal abchecken

Vorteile bei angegebenen Typen: Autocomplete!

```
def to_text(number):
    '''casts an int or a float to str
    Parameters
    ----------
    number : int, float

    Returns
    -------
    str
    '''
    return str(number)
```

```
toText(15).        You, a few
                   capitalize
                   casefold
def adjust_        center
    mean =         count
    adjuste        encode
    for val        endswith
        adj        expandtabs
                   find
EMS  1   OU        format
                   format_map
```

M

```
                   abc adjusted
    for v          abc append
        a          abc array
                   abc average
EMS  1   C         abc avg
                   abc calculates
```

ohne Annotation

Aktuell: Autocomplete schafft die Basics…

```
def to_text(number):
    '''casts an int or a float to str
    Parameters
    ----------
    number : int, float

    Returns
    -------
    str
    '''
    return str(number)
```

```
def to_text(num):
    '''casts an int or a float to str
    Parameters
    ----
    num
    Retu
    ----
    str
    '''
    retu

to_text()
```

```
(num) -> str

casts an int or a float to str
Parameters
----------
num : int, float

Returns
-------
str
```

ohne Annotation

You, a few seconds ago • Unco

Vorteile bei angegebenen Typen: Autocomplete!

```python
def to_text(num: Union[int, float]) -> str:
    '''casts an int or a float to str
    Parameters
    ----------
    num : int, float

    Returns
    -------
    str
    '''
    return str(num)
```

```python
def to_text(num: Union[int, float]) -> str:
    '''casts an int or a float to str
    Parameters
    ----------
    num : int, float

    Returns
    -------
    str
    '''
    return str(num)

to_text()
```

```
(num: int | float) -> str

casts an int or a float to str
Parameters
----------
num : int, float

Returns
-------
str
```

mit Annotation

Dictionaries & Strings

- Kapitel 5 & 6 lesen, bearbeiten & dokumentieren

https://ssr-2024.github.io/programming_basics/docs/lecture-book-sections/chapter5.html

Committen nicht vergessen…

Übungen (labs) bearbeiten