Kommandozeilenargumente

- `import sys` ← immer !!!

Wie ü.gibt man Argumente ?

`<var> = sys.[i]`

```
import sys

if len(sys.argv) < 3: #guckt, ob ich mind. 3 Elemente habe (Dateiname, liter, distanz)
    print('Bitte geben Sie Liter und die Distanz an!.')
    sys.exit() #beendet d. Programm sofort

liter = float(sys.argv[1])
distanz = float(sys.argv[2])

verbrauch = (liter *100)/distanz

print(round(verbrauch,2))
```

```
# Terminal
python3 verbracuh.py 45.5 600

# Output
7.58
```

"Call by Value" vs. "Call by Reference"

- Call by Value (Werte-Kopie):
    - Wie ein Foto v. einem Blatt Papier
        - Wenn ich auf dem Foto herummale, bleibt das Original-Blatt sauber.

      ```
      int, float, str, bool
      ```

- Call by Reference (Adress-Verweis):
    - Wie Google-Doc-Link.
        - Wenn ich ü. den Link etw. im Dokument lösche, dann ist es ∀ anderen, d. den Link haben, auch weg.

```
list, dict, set
```

String

Man kann ein Text nicht ändern !!!

```
text = "Was machen Sachen"

text[2] = i

#Augabe: Error
```

Listen

Werkzeuge:

- `.pop()`:
  - `.pop(<index>)`
  - dann wird das Element an dieser stelle gelöscht
- `.remove()`:
  - wenn das erste Element, der dem Element in `.remove(<elem>)` entsprciht, wird entfernt
- `.copy()`:
  - **wichtig !!!!**
    - man kopirt nur d. ***oberste Ebene***

```
x = [[1,2],[3,4],5]
y = x.copy()

y[1][0] = y[2]
print('x: ', x) # [[1,2],[5,4],5]
print('y: ', y) # [[1,2],[5,4],5]

y[2] = y[0]
print('x: ', x) # [[1,2],[5,4],5]
print('y: ', y) # [[1,2],[5,4],[1,2]]
```

```
x = [[1], [2]]
y = x.copy()
y[0].append(10)
y[1] = [99]
print(x) #[[1, 10], [2]]
```

x = [[1], [2]] y = x.copy() y[0].append(10) y[1] = [99] print(x)

Wann .append und wann += ?:

- `.append()`:
  - Wenn ich ein Objekt mit den Klammern hinzufügen möchte
    - Bsp.: Ich lege d. gesamte tüte mit den Elementen in den Wagen
  - wenn ich nur den Inhalt hinzufügen möchte
    - Bsp.: Ich lege nur d. Ware in den Wagen & schmeiße die leere Tüte weg

```
haupt_liste = [1, 2]
zusatz = [3, 4]

# Szenario A: append (Das Objekt als Ganzes)
liste_a = [1, 2]
liste_a.append(zusatz)
print(liste_a)
# Ergebnis: [1, 2, [3, 4]]  <-- Liste in der Liste

# Szenario B: += (Nur der Inhalt)
liste_b = [1, 2]
liste_b += zusatz
print(liste_b)
# Ergebnis: [1, 2, 3, 4]    <-- Alles auf einer Ebene
```

Dictionaries

Wie sortieren ?:

- `sorted_indices = sorted(häufigkeiten.keys())`

```
my_dict = {'a': 2, 'c': 1, 'b': 4, 'd': 3}
sortierte_liste_tupel = sorted(my_dict.keys()) # Ausgabe: {'a': 1, 'b': 2,
'c': 3, 'd': 4}
```

Wie Elemente hinzufügen:

- `dic.upadte()`

```
dic = {}
n = 7
k = 3
```

```
dic.update({(n,k) : 45})
print(dic)
```

Mit Text arbeiten

Man muss es sich so vorstellen, dass `.readlines()` unsere Augen sind & wir jede Zeile lesen ! Das heißt, wenn ich etw. hinzufüge, dass ich dann mit den Augen unter d. letzen Zeile bin. Wenn jmd. jzt. fragen würde, was da steht, dann würde ich "nichts" sagen, weil da nichts geschrieben steht. Damit ich lesen kann, muss ich wieder mit den Augen zur ersten Zeile, welches hier mit `.seek({index})` geschieht. ES IST GANZ WICHTIG ZU WISSEN, DASS `.split()`, `.strip()`, usw. EINEN STRING ZURÜCK GEBEN !!!

Also wenn ich in der Klausur eine Aufgabe bekommen sollte, welches mit einer Datei ist, dann muss ich als ersten gucken, ob ich den gesamten Inhalt als Ganzes brauche, oder ob ich es auch zeilenweise angucken kann.

Wenn ich eine „Textanalyse“ machen muss, also den gesamten Text vor mir gedruckt haben möchte, damit ich da zum Bsp Wörter oder Buchstaben „korrigieren“ möchte, dann benutze ich readlines(). Im Computer ist d. gedruckte Blatt dann eine Liste mit den einzelnen Elementen als String mit \n & \t.

Wenn ich aber zeilenweise abschreiben möchte, v. einem Buch in meinen Heft, dann muss ich `.readline()` verwenden. Wichtig hierbei ist, dass `mein Auge immer einen Zeile weiter geht !` + ich muss dem Computer sagen, dass es etwas `n Mal` machen soll, weil er dann etwas z.B. nur 1 mal macht

Allgemein:

- `r` = read
- `w` = write
- `a` = append
- **Wie kombinieren?**
  - bestehende Datei ändern möchte → `'r+'`
  - neue Datei erstellen & konrollieren möchte → `'w+'`
  - an bestehende Datei etw. anhängen möchte → `'a+'`

Wie Datei öffnen ?:

```
with open("Probe.txt", 'r') as data:
    print(data)
```

- somit benötige ich kein `var = close()` am Ende

.readlines():

```
with open("Probe.txt", 'r') as data:
    f_inhalt = data.readlines()
    print(f_inhalt)
```

- somit wird mir eine Liste aus den Elementen ausgegeben
- Textdatei

```
    # Ausgabe
    ['1) Papa \n', '2) Mama \n', '3) Efe \n', '4) Sude \n', '5) Simge \n', '6)
Pepee \n', '7) Mayla ']
```

.readline()

.readline() ist so, als ob wir ein Buch haben und ein Heft. Unser Auge guckt im Buch n. d. ersten Zeile und wir schreiben d.as in unser Heft ab. Autom., wenn wir ins Buch gucken, beginnen wir mit d. 2. Zeile & d. schreiben wir ebenso in ins Heft. usw. Da wir dem Computer alles definieren müssen, müssen wir sagen wie viele solcher Prozesse wir durchmachen möchten, welches hier = 66 mal ist. D.h., dass wir 66 $\times$ d. Buch lesen & abschreiben

- *Übungsblatt 6, Nr.4*
- hierbei wird nur eine Zeile gelesen

```
with open("Probe.txt", 'r') as data:
    f_inhalt = data.readline()
    print(f_inhalt)
```

```
# Augabe
1) Papa
```

- wenn ich aber `print(f_inhalt, end=' ')` mache, dann wird es ohne den Absatz ausgegeben

```
# Augabe
1) Papa
```

Wie đ Zeilen iterieren ?:

- đ d. Iterieren, wird ein Memory-Problem umgangen !</span

```
with open("Probe.txt", 'r') as data:
    for line in data:
        print(line, end='')
```

```
#Ausgabe
1) Papa
2) Mama
3) Efe
4) Sude
5) Simge
6) Pepee
```

Wie Inhalt zu d. Datei hinzufügen ?:

- Text2

```
with open("Probe.txt", 'a+') as data:

    mayla = '\n7) Mayla'

    # 1. Schreiben/Anhängen
    data.write(mayla)

    # 2. Zeiger zurücksetzen auf den Anfang (Index 0)
    data.seek(0)

    # 3. Lesen der gesamten Datei ab Index 0
    inhalt = data.read()

    # 4. Ausgabe des gelesenen Inhalts
    print(inhalt)
```

```
# Veränderte Textdatei:
1) Papa
2) Mama
3) Efe
4) Sude
5) Simge
6) Pepee
7) Mayla
```

- Wie kann man denn direkt einen kompletten langen Text hinzufügen ?

```
with open("Probe.txt", 'a+') as data:

    #String schrieben
    mayla = str(input(f"Was soll hinzugefügt werden: "))

    #String zur Datei hinzufügen
    data.write('\n' + mayla)

    #Position in der Datei wieder auf die erste zeile setzen
    data.seek(0)

    lines = data.readlines()

    for line in lines:
        print(line, end="")
```

```
# Augabe
1) Papa
2) Mama
3) Efe
4) Sude
5) Simge
6) Pepee
7) Mayla
```

- w+

```
with open("Probe.txt", 'w+') as data:

    #String schrieben
    mayla = str(input(f"Was soll hinzugefügt werden: "))

    #String zur Datei hinzufügen
    data.write(mayla)

    #Position in der Datei wieder auf die erste zeile setzen
    data.seek(0)

    lines = data.readlines()

    for line in lines:
        print(line, end="")
```

```
# Ausgabe
Hallo was geht ?
# man sieht alos, dass d. gesamte Inhalt gelöscht wurde
```

Wie kann man denn gezielt leere Zeilen entfernen ?:

```
with open("Probe.txt", 'a+') as data:

    data.seek(0)

    lines = data.readlines()

    for line in lines:
        if line.strip() == '':
            continue

        print(line, end="")
```

- Somit sagen wir also, wenn wir an eine leere Zeile stoßen, dann einfach ignorieren & mit d. nächsten Zeile weiter machen zu lesen ?

```
# Text davor

1) Papa
2) Mama
3) Efe
4) Sude
5) Simge
6) Pepee
7) Mayla

# Text danach:
1) Papa
2) Mama
3) Efe
4) Sude
5) Simge
6) Pepee
7) Mayla
```

Wie kann ich denn machen, dass ich jede dritte Zeile oder 4. Zeile lesen möchte ?:

- %

Wie kann man einfach eine neue Datei erstellen ?:

```
pfad = 'My fam.txt'
with open(pfad,'w'):
    pass
```

Was ist ein optionaler Parameter ?:

- ein optinaler Parameter ist ein Parameter, dass immer autom, f, eine var, gegesetzt wird, falls hier ichts eingesetzt wird

```
def begruessung(name="Mensch"):
    print("Hallo " + name)

begruessung()
begruessung("Lisa")
```

```
#output
"Hallo Mensch"
"Hallo Lisa"
```

Wie kann man einzelne Zeichen zählen ?:

- wir verw. .count()

```
def letterCount(text, zeichen=string.ascii_letters):
    count_collection = []

    for letter in zeichen:
        count = text.count(letter)
        count_collection.append(count)

    return count_collection

text = str(input('String: ')) #"Was machen Sachen ?"
zeichen = str(input('Zeichen: ')) #'ach'

print(letterCount(text, zeichen))
```

```
#Output
[3, 2, 2]
```

- $3 \times a,\ 2 \times c,\ 2 \times h$

Rekursion

So kann mich sich die Rekursion auch vorstellen.

Bsp.: Fibonacci-Zahlen

Wenn du wissen willst, was der Wert von $F(5)$ ist, gehst du zu Person 5. Da diese d. Antwort ¬ auswendig weiß, ruft sie sofort bei ihren Kollegen in Büro 4 und Büro 3 an und bittet sie um Hilfe. D. Problem bei d. normalen Rekursion ist, dass diese Kollegen wiederum ihre Nachbarn anrufen müssen, was zu unzähligen Telefonaten führt, bei denen dieselben Fragen immer & immer wieder gestellt werden.

Hier kommt unsere Schachtel 📦 ins Spiel, d. im Flur ∀ erreichbar steht: das Memo-Dictionary. Bevor eine Person zum Hörer greift, schaut sie erst in diese Schachtel. Liegt dort schon ein Zettel mit dem Ergebnis f. ihr Büro, nimmt sie den Wert einfach heraus & gibt ihn sofort an den Fragesteller weiter. Ist d. Schachtel leer, müssen die Kollegen angerufen werden. Die Einzigen, d. niemals telefonieren müssen, sind d. Personen in Büro 0 und Büro 1. Sie kennen ihre Werte (0 und 1) sofort, legen sie als Erste in d. Schachtel und starten damit d. Kettenreaktion. Sobald eine Person d. Antw. ihrer beiden Kollegen erhalten hat, addiert sie diese, schreibt das neue Ergebnis auf einen Zettel für die Schachtel und liefert die Antwort ab. So wird jede Zahl im Gebäude nur ein einziges Mal wirklich berechnet. ✨

```
def querSumme(n:str, result:int)->int:
    if len(n) == 0:
        return result
    else:
        b = int(n[0])
        result += b
        return querSumme(n[1:],result)


n = str(input("n: "))
n_lst = []
for i in n:
    n_lst.append(i)
result = 0
print(querSumme(n_lst,result))
```

oder

```
def quersumme(n: int) -> int:
    if n < 10:
        return n

    return (n % 10) + quersumme(n // 10)
```

Dreiecke

1) Fläche deines Dreiecks begrenzen:

- Präsenzblatt
- Was wir berechnen

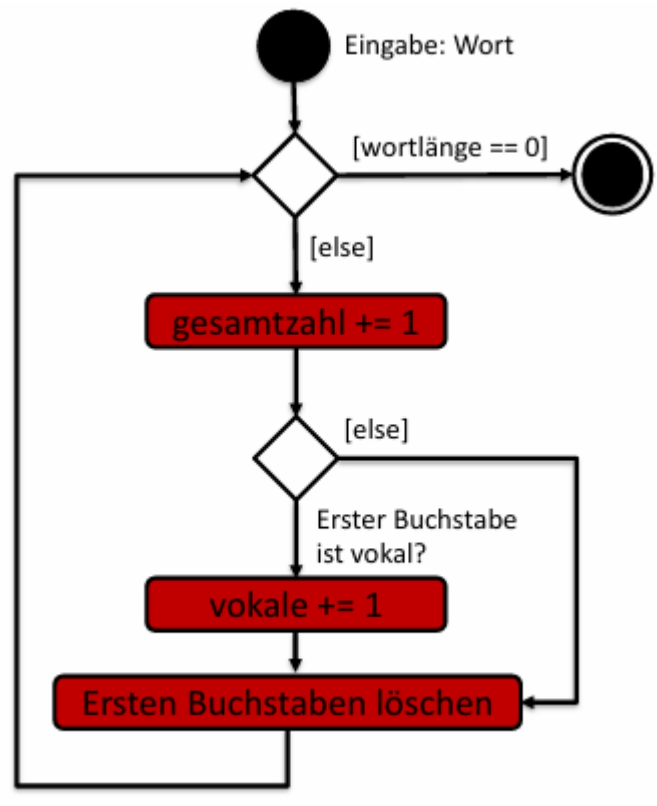
- Also $len(Höhe_{absolut})$ & je n. dem wo wir uns in dem absoluten intervall befinden, wird d. Breite demetsprechend berechnet.
- $H_{absolut} \in [1,4]$
    - wobei $H_{relativ} \in [0,3]$, weil Länge = $4-1=3$ (Gesamtlänge)
- Also, wir haben jzt. d. Basis B, aber weil wir aber d. Hälfte d. Basis benötigen = $\frac{B}{2}$
- d. absolute Höhe H_{abs} & autom. auch H_{rel}
- Die $Gesamthöhe \in [2.5, 5.5]$ & $H_{relativ}$ bewegt sich im Bereich $5.5-2.5=3 \implies H_{relativ} \in [0,3]$
    1. random Höhe wählen
    2. H_{rel} bestimmen
    3. B bestimmen
    4. x koordinaten Wählen: $r.choice[x_{mid}-w, x_{mid}+w]$.
- Rampe $B=2$ & $H_{abs}=1 \rightarrow$ Verhätnis zwischen Höhe zur Länge $= \frac{Höhe}{Länge}$
    - 100 Menschen ingesamt
    - 34 = krank
    - $34/100 = 0.34 = 34$
    - Das Verhältnis zw. den Menschen & den Kranken beträgt 34
        - wir wollen wissen, wie das Verhältnis zwischen d. Breite & der relativen Höhe ist $= \frac{\frac{B}{2}}{H} = \frac{B}{2H}$

Tests

Allgemein:

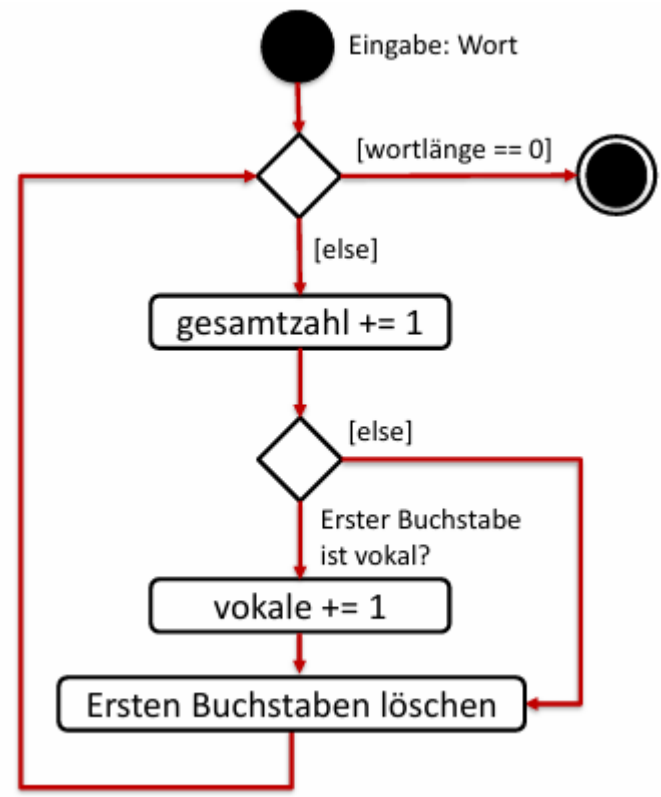
1. Anweisungsü.deckung: C_0
2. Zweigsü.deckung: C_1
3. Pfadü.deckung: C_2a, C_2b, C_2c
4. Bedingungsü.deckung: C_3a, C_3b, C_3c

Was ist Anweisungsü.deckung: C_0:




```python
def vokal_counter(wort:str)-> tuple:
    gesamtzahl = 0
    vokale = 0

    while len(wort) > 0:
        gesamtzahl += 1

        if wort[0] in "aeiou":
            vokale += 1

        wort = wort[1:]

    return gesamtzahl, vokale


wort = str(input("Wort: ")).lower().replace(" ","")
print(wort)
print(vokal_counter(wort))
```

Was ist Zweigü.deckung: C_1:




```
def c1_test()->bool:
    return vokal_counter('ab') == (2,1)
```

Storage

```
#Probe.txt
1) Papa
2) Mama
3) Efe
4) Sude
5) Simge
6) Pepee
7) Mayla
```

```
#Probe.txt
1) Papa
2) Mama
3) Efe
4) Sude
5) Simge
6) Pepee
```