2. Prozedurale Programmierung

Inhalt

- Basic Syntax II
- Sequenzielle **Datentypen (List, Tupel, etc.)**
- Was ist eine **Funktion**
- **Wie schreibt man Funktionen in Python**

Eigene Syntaxelemente

- **Kommentare:** beginnen mit einem Doppelkreuz-Zeichen **#**
- **Name:** erlaubt sind die Buchstaben A - Z und a - z, die Zahlen 0 - 9, sowie der Unterstrich "_"
- **Literale:** direkte Darstellung der Werte von Basistypen

```
>>> STRING = "# Dies ist kein Kommentar."
```

Grund-Datentypen

- **Integer**
  - `>>> type(1)`
  - `<type 'int'>`
- **(sehr) lange Integer**
  - `>>> type(1L)`
  - `<type 'long'>`
- **Gleitkommazahlen**
  - `>>> type(1.0)`
  - `<type 'float'>`
- **Komplexe Zahlen**
  - `>>> type(1 + 2j)`
  - `<type 'complex'>`

- **Standardoperationen**
  - Addition `+`
  - Subtraction `-`
  - Division `/`
  - Integerdivision `//`
  - Multiplikation `*`
  - Expotentieren `**`
  - Modulo `%`
- **Built-in Funktionen**
  - `round`, `pow`, etc.

Numerische Operationen

Operation	Abkürzung
`x = x + y`	`x += y`
`x = x - y`	`x -= y`
`x = x * y`	`x *= y`
`x = x / y`	`x /= y`
`x = x % y`	`x %= y`
`x = x**y`	`x **= y`
`x = x//y`	`x //= y`

Vergleichsoperation

- `x == y`
- `x != y`
- `x < y`
- `x <= y`
- `x > y`
- `x >= y`

Wahrheitswerte

- bool ist der Typ der Wahrheitswerte `True` und `False`

- Operationen:
  - not a (Negation)
  - a and b (Konjunktion)
  - a or b (Disjunktion)

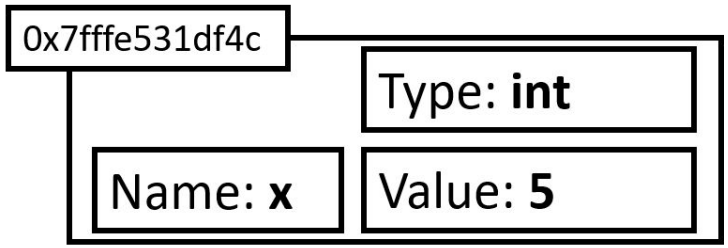
Ausdrücke und Variablen

Variable: abstrakter Behälter für eine Größe, welche im Verlauf eines Rechenprozesses auftritt

- Name
- Adresse
- Wert
- in C++: `int x;`

Python stellt keine Variablen bereit.

```
int x = 5;
```



Anweisungen

Programm: eine Abfolge von Anweisungen. Ein Programm ist dabei aus Anweisungsblöcken aufgebaut

Zuweisung: *die Verbindung* zwischen einem Namen und dem Wert

```
>>> x = 1
>>>
>>> x = x + 2
>>>
```

- x ist der Name
- 1 ist ein Objekt vom Typ-Int
- x ist mit einem NEUEN Objekt verbunden, dessen Wert x + 2 ist

Python und Variablen - C++ Beispiel

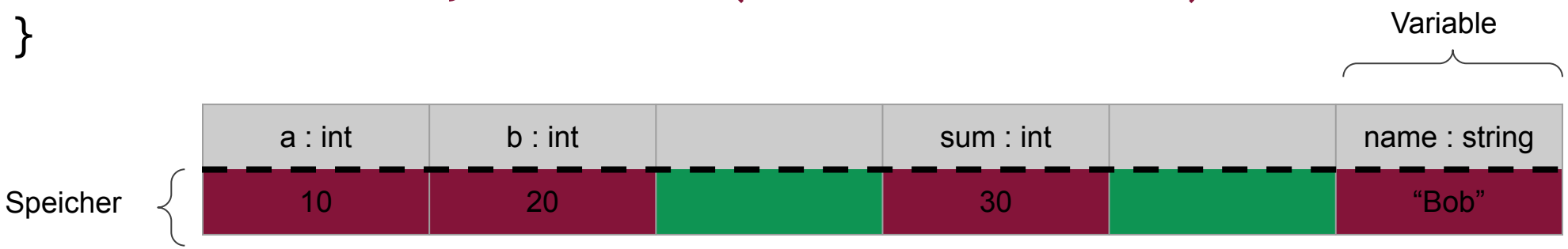
```
int main () {
    int a = 10;
    int b = 20;

    int sum = b + a;

    string name = “Bob”;

    std::cout << name;

    a = “Dob” //Fehler (a ist int)
    float a = 10.0; //Fehler (a existiert schon)
}
```



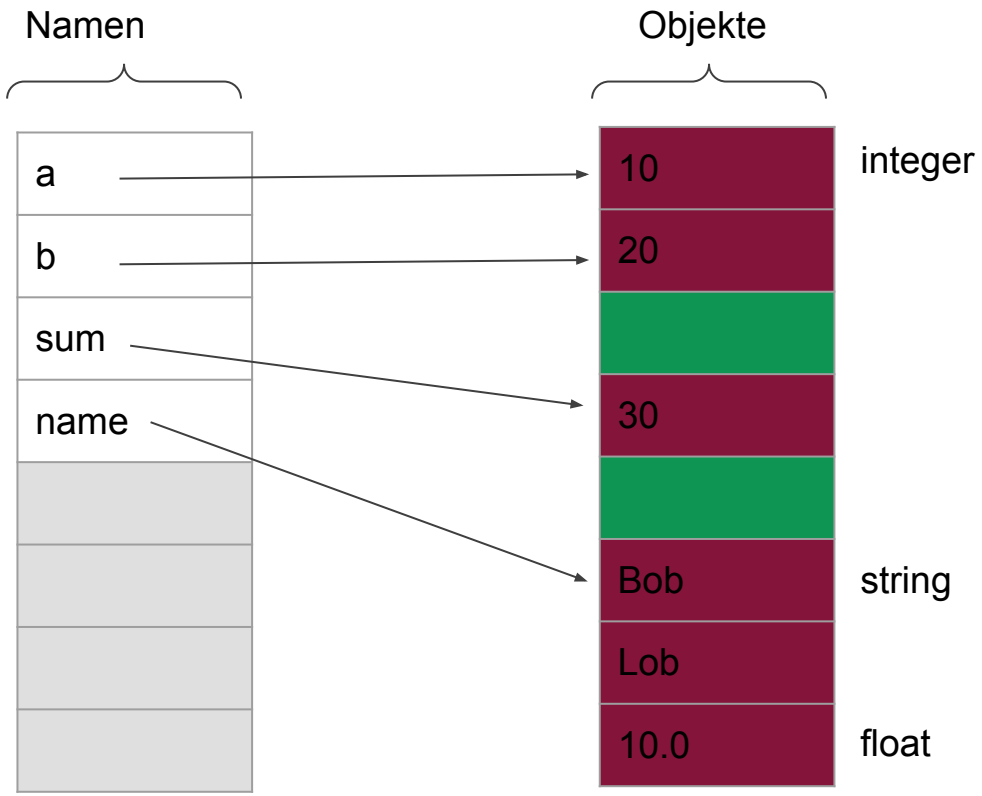
Python und Variablen

```
a = 10
b = 20

sum = b + a

name = “Bob”

print (name)
```



Python und Variablen

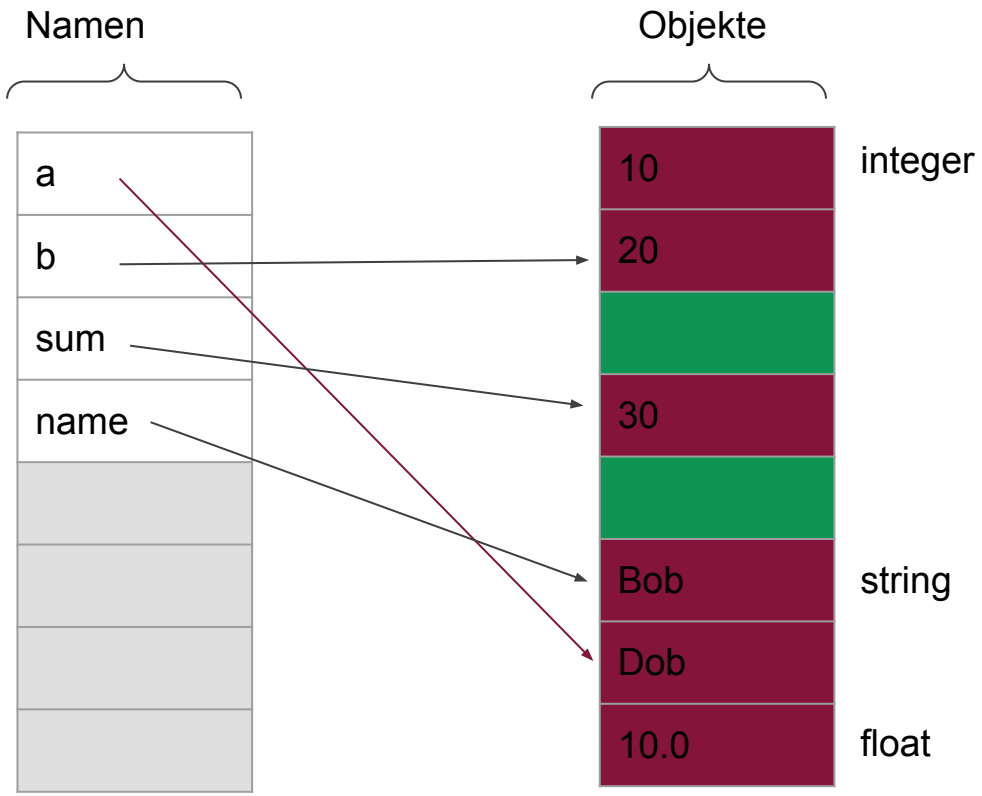
```
a = 10
b = 20

sum = b + a

name = “Bob”

print (name)

a = “Dob” #OK
```



Python und Variablen

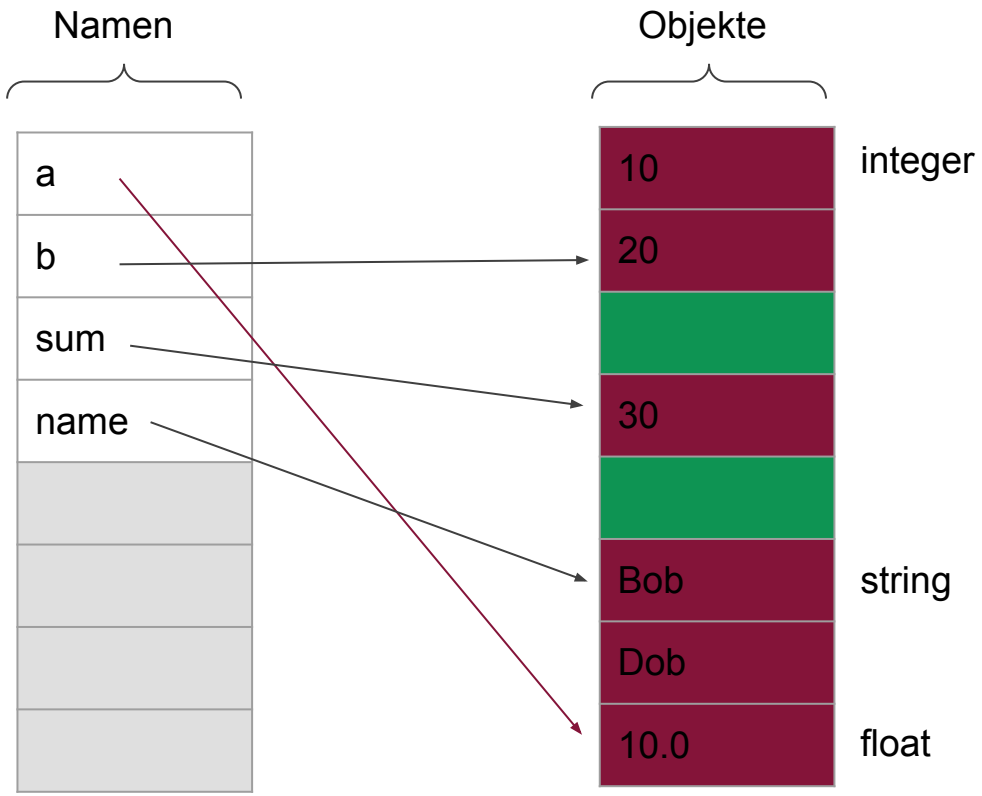
```
a = 10
b = 20

sum = b + a

name = “Bob”

print (name)

a = “Dob” #OK
a = 10.0  #OK
```



If - anweisung

```
if expression1:
   anweisung1
[elif expression2: anweisung2]
   ...
[else: anweisung]
```

- Die Ausdrücke `expression1, expression2, …` werden in angegebener Reihenfolge ausgewertet
- bis einer zutrifft
  - Dann wird die entsprechende Anweisung ausgeführt.
- Wenn keiner der Ausdrücke zutrifft, wird die else-Anweisung ausgeführt.

Einrückungen und Blöcke

- Leerzeichen sind wichtig:
  - die Anweisungen in einer if-Anweisung müssen eingerückt werden
- d.h. die Anweisungen sind zu einem Block gruppiert
- Anweisungen des gleichen Blocks müssen mit der gleichen Anzahl des gleichen Typs Leerzeichen eingerückt sein
- Leerzeichen:
  - Space, Tabulator

```
a = 10
if a == 10:
    b = 20
    print (a+b)
c = 30

print(a)
print(b) #warum funktioniert?
```

Block - if

Block - ‘main’

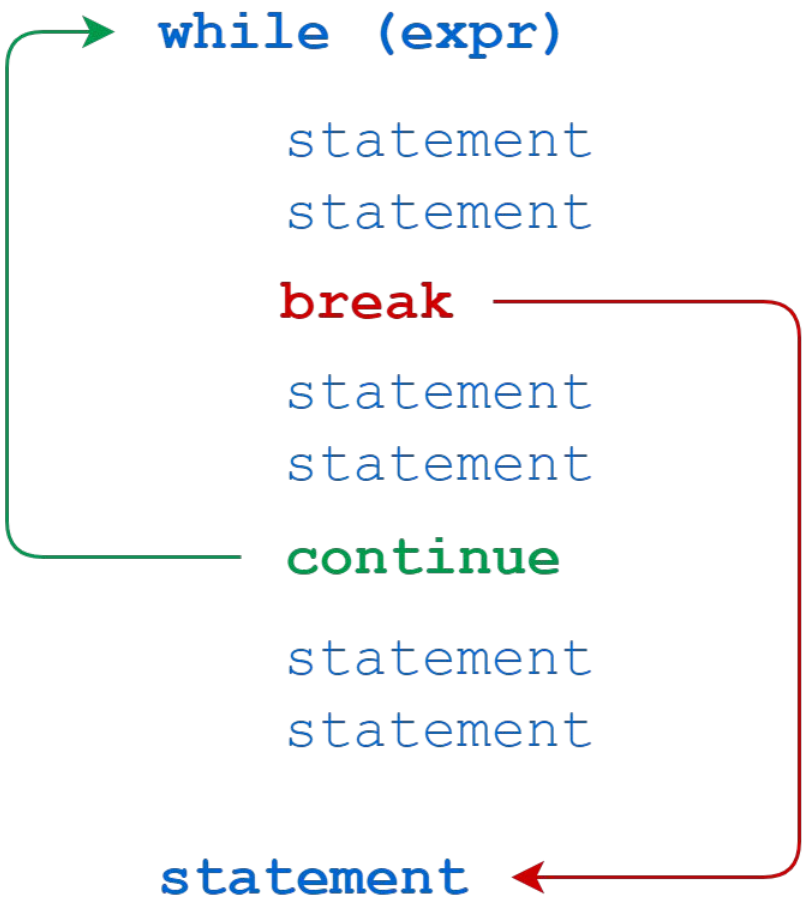
While - Schleife

```
while expression:
    block
```

1. Der Ausdruck `expression` wird ausgewertet
2. Trifft er zu, wird `block` ausgeführt
3. Danach `expression ist wieder ausgewertet (Schritt 1)`

break & continue

- Die break-Anweisung
  - verlässt die aktuelle Schleife
  - `expr` (die Bedingung) wird nicht ausgewertet
- Die continue-Anweisung
  - überspringt den Rest des Blocks
  - wertet `expr` neu aus
  - und setzt ggf. die Schleife fort



Sequenzielle Datentypen

Zur Kategorie der sequenziellen Datentypen gehören

- **str** und **unicode** für die Verarbeitung von Zeichenketten
- **list** und **tuple** für die Speicherung beliebiger Instanzen
  - eine **list** nach ihrer Erzeugung verändert werden kann (mutable)
  - ein **tuple** ist nach der Erzeugung nicht mehr veränderbar (immutable)
- **dict** für eine Zuordnung zwischen Objektpaaren
- **set** für ein ungeordneter Zusammenschluss von Elementen, wobei jedes Element nur einmal vorkommen kann

Strings

- `str1 = "abc"`
- `str2 = 'abc'`
- `str3 = """ abc """`
- `str4 = ("abc" "def")`

Escape-Sequenz

- `\a` erzeugt Signalton
- `\b` Backspace
- `\f` Seitenvorschub
- `\n` Linefeed
- `\r` Carriage Return
- `\t` horizonal Tab
- `\v` vertikal Tab
- `\" \' \\` Escaping `"'\`

```
Catalins-iMac:~ cat$ python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> name = "bob"
>>> type(name)
<class 'str'>
>>> print(name)
bob
>>> sname = name+"!!"
>>> len(sname)
5
>>> sname
'bob!!'
>>> name[0]
'b'
>>>
```

List

Eine **list** kann Elemente unterschiedlichen Datentyps enthalten

- Syntax `[Wert_1, ..., Wert_n]`
- Eine Liste kann auch nach ihrer Erzeugung verändert werden
- Die Funktion **`len()`** bestimmt die Anzahl der Elemente der Liste
- Listen können auch Listen enthalten, auch sich selbst
- Hinzugefügt werden Werte mit dem **`+-Operator`** und den Funktionen **`append()`** und **`insert()`**
- Zugriff auf Elemente mit **`[]-Operator`**

Tuple

Im Gegensatz zu **Listen** sind **Tuple** immutable

- d.h. jede Änderung erzeugt ein neues Objekt
- Syntax `(Wert_1, ..., Wert_n)`
- Sie sind damit besonders geeignet, um Konstanten zu repräsentieren
- Ein **Tupel** wird in runde Klammern geschrieben (packing)
- **min()** bestimmt das Minimum eines Tupels, **max()** das Maximum
- Nesting
- unpacking

```
x, y = (1,2)  # x← 1 und y ← 2
```

Tuple vs List

```
Catalins-iMac:~ cat$ python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> l = [1,2,3,4]
>>> l.append(10)
>>> l
[1, 2, 3, 4, 10]
>>> l[2] = 33
>>> l
[1, 2, 33, 4, 10]
>>> v = l + [101]
>>> v
[1, 2, 33, 4, 10, 101]
>>> a = l[2]
>>> a
33
>>> t = (1,2,3)
>>> t.append(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> t[0] = 101
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

Dictionary

Mit dict wird eine Zuordnung zwischen Objektpaaren hergestellt

- Syntax `{ Key_1: Value1, Key_2: Value2, ... }`
- müssen die Keys nicht ganze Zahlen (aber Liste?)
- Dictionaries sind iterierbare Objekte
- Die Länge eines Dictionaries `d` kann über `len(d)` abgefragt werden
- Mit del `d[k]` wird das Element mit Schlüssel `k` gelöscht
- mit `k in d` kann geprüft werden, ob sich der Schlüssel `k` in `d` befindet

Dictionary

```
Catalins-iMac:~ cat$ python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> d = {}
>>> d['a'] = 2
>>> d
{'a': 2}
>>> d['a']
2
>>> d.keys()
dict_keys(['a'])
>>> d.values()
dict_values([2])
>>> d['b'] = [1,2,3]
>>> d
{'a': 2, 'b': [1, 2, 3]}
>>> d['b'][1]
2
>>>
```

Set

Eine Menge ist ein ungeordneter Zusammenschluss von Elementen, wobei jedes Element nur einmal vorkommen kann

- Syntax `{Wert_1, ..., Wert_n}`
- gibt es für mutable Mengen den Typ **set**
- für immutable Mengen den Typ frozenset
- `len(m)` liefert die Anzahl der Elemente in m
- `x in m` ist True, wenn x in m enthalten ist
- `m<=t` ist True, wenn m eine Teilmenge von t ist

Set

```
Catalins-iMac:~ cat$ python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> s = {1,2,3}
>>> s[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
>>> l = [1,2,3,4,3
... ]
>>> l
[1, 2, 3, 4, 3]
>>> ns = set(l)
>>> ns
{1, 2, 3, 4}
>>> 2 in ns
True
>>> 5 in ns
False
>>>
```

List ,Dict, Tuple

- weitere Beispiele

LIST	TUPLE	DICTIONARY	SET
Allows duplicate members	Allows duplicate members	No duplicate members	No duplicate members
Changeble	Not changeable	Changeable, indexed	Cannot be changed, but can be added, non -indexed
Ordered	Ordered	Unordered	Unordered
Square bracket [ ]	Round brackets ( )	Curly brackets{ }	Curly brackets{ }

Listen

Operation	Erklärung
`s in x`	prüft, ob s in x ist
`s not in x`	prüft, ob s nicht in x ist
`x + y`	Verkettung von x und y
`x[n]`	liefert das n-te Element von x
`x[n:m]`	liefert eine Teilsequenz von n bis m
`x[n:m:k]`	liefert eine Teilsequenz von n bis m, aber nur jedes k-te Element wird berücksichtigt
`len(x)`	liefert die Anzahl von Elementen
`min(x)`	liefert das kleinste Element
`max(n)`	liefert das größte Element

Listen

```
1   myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2   print(myList[:2])
3   print(myList[2:])
4   myList[5:] = ['a', 'b', 'c']
5   print(myList)
6
7   myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
8   myList[1:9] = 'x'
9   print(myList)
10
11
```

Listen

```
1   myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2   print(myList[:2])
3   print(myList[2:])
4   myList[5:] = ['a', 'b', 'c']
5   print(myList)
6
7   myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
8   myList[1:9] = 'x'
9   print(myList)
10
11
```

[1, 2, 3, 4, 5, 'a', 'b', 'c']

[1, 'x', 10]

Tupel

```
1   tup = 1, 2, 'a'
2   print(tup)
3   print(tup[1])
4
5   for e in tup:
6       print(e)
7
8   '''
9       Was ist die Ausgabe, wenn man diese Zeile auskommentiert?
10  '''
11  #tup[1] = 'x'
12
13
```

Dictionaries

```
1   d = {'num':1,'den':2}
2   print(d)
3   print(d['num'])
4   d['num'] = 99
5   print(d['num'])
6
7   if 'num' in d:
8     print('We have num!')
9
10  del d['num']
11
12  if 'num' in d:
13    print('We have num!')
14
15
```

Initialisierung

List

```
l = []
for i in range(5):
    l.append(0)                    → [0,0,0,0,0]


l = [0]*5                          → [0,0,0,0,0]
```

Dict

```
d = {}
for i in range(5):                 → {0: 0, 1: 0, 2: 0, 3: 0, 4: 0}
    d[i] = 0

d = dict.fromkeys(range(5), 0)     → {0: 0, 1: 0, 2: 0, 3: 0, 4: 0}
```

Zustand, Verhalten, Identität

- in der realen Welt
- wir verwenden täglich viele verschiedene Objekte

Ein Objekt: Laptop

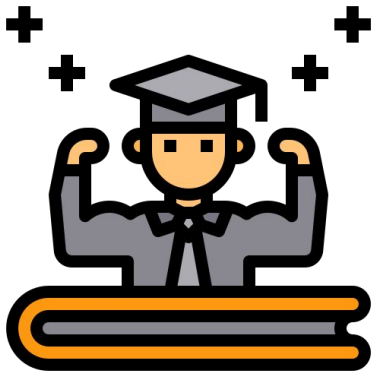
- Zustand (state): Dell (Hersteller), 14” (Bildschirm), Intel (CPU)
  - Eigenschaften
- Verhalten (behavior): einschalten, reset
  - Methoden
- Identität: 8FG89W2 (Serial Number)

Zustand, Verhalten, Identität

- in der realen Welt
- wir verwenden täglich viele verschiedene Objekte

Ein Objekt: Student

- Zustand (state): Bob (Name), 19 (Alter), UBB (Uni)
  - Eigenschaften
- Verhalten (behavior): prüfung_ablegen, studieren, schlafen
  - Methoden
- Identität: 02304 (Matrikelnummer)



Zustand, Verhalten, Identität

Python: alle sind Objekte

Ein Objekt:

- Zustand (state)
- Verhalten (behavior)
- Identität

- unveränderlichen Grund-Datentypen (Zahlen, Strings, Tupel)...
- und veränderlichen Objekte Listen, Dictionaries...

- `id(objekt)`
- `type(objekt)`
- `isinstance(objekt,typ)`

Zustand, Verhalten, Identität - in Python

```
1   l = [1,2,3]
2
3   print (id(l)) # zB: 4566092872
4
5   v = [1,2,3,4]
6   print (id(l)) # zB: 4566829256
7
8   for el in l:
9     print(el) # 1,2,3
10
11  l.append(33)
12  l.pop()
13
14
```

Identität

Zustand

verhalten

unveränderlichen und veränderlichen Objekte


```
1   s = "abc"
2   print(id(s)) #4566030184
3
4   s = s + "d"
5   print(id(s)) #4566832720
6
7
8   l = [1,2,3]
9   print(id(l)) #4566832720
10
11
12  l.append(4)
13  print(id(l)) #4566832720
14
```

unveränderlichen und veränderlichen Objekte

```
1   myList = [1, 2, 3]
2   print(myList)
3   print(myList[1])
4
5   print('Die Liste enthält', len(myList), 'Elemente')
6   print('Das erste Element ist ', myList[0], 'und das letzte ist ', myList[len(myList) - 1])
7
8   x = myList
9   print(myList , x)
10
11  '''
12      Das output?
13  '''
14  x[1] = '?'
15  print(myList , x)
16
17
```

unveränderlichen und veränderlichen Objekte

```
1   myList = [1, 2, 3]
2   print(myList)
3   print(myList[1])
4
5   print('Die Liste enthält', len(myList), 'Elemente')
6   print('Das erste Element ist ', myList[0], 'und das letzte ist ', myList[len(myList) - 1])
7
8   x = myList
9   print(myList , x)
10
11  '''
12      Das output?
13  '''
14  x[1] = '?'
15  print(myList , x)
16
17
```

- [1, '?', 3] [1, '?', 3]
- die beiden Listen wurden geändert
- myList und x sind unterschiedliche Namen für das gleiche Objekt
- id(x) == id(myList)

unveränderlichen und veränderlichen Objekte

- Unveränderliche Objekte können nach der Erstellung nicht mehr geändert werden
  - d.h. jede Änderung erzeugt ein neues Objekt

- Zugriff auf unveränderliche ist im Prinzip schneller
- Veränderlichen Objekte sind nützlich, wenn die Größe des Objekts geändert werden muss
- Unveränderliche Objekte werden verwendet, wenn man sicherstellen muss, dass das Objekt immer unverändert bleibt

- Unveränderliche Objekte sind grundsätzlich teuer zu „ändern“, da dazu eine Kopie erstellt werden muss.
- Das Ändern veränderlicher Objekte ist billig.

Prozedurale Programmierung

Ein **Programmierparadigma** = ein fundamentaler Programmierstil

Imperative Programmierung: das Programm wird als eine Reihe von Anweisungen geschrieben, die den Zustand des Programms ändern.

Zuweisung: `a = 10`

Prozedurale Programmierung: Programme werden aus eine oder mehreren Prozeduren bzw. Funktionen aufgebaut

Prozedurale Programmierung

Gemäß des **prozeduralen Paradigmas**

- wird der Zustand eines Programms mit Variablen beschrieben
- werden die möglichen Systemabläufe algorithmisch formuliert
- bilden Prozeduren/Funktionen das zentrale Strukturierungs- und Abstraktionsmittel

Prozedurale Programmierung. Wieso ist es wichtig?

```python
import time

letters = "we gonna divide some stuff"
n1="type first number: "
n2="type second number to divide by: "

print(letters)

a=float(input(n1))
b=float(input(n2))

# ##### DONT TOUCH ANYTHING BELOW LINE #####
# #####  IT WORKS AND I DONT KNOW WHY  #####
add_used = 0

# define add
def add(a, b):
    global add_used
    add_used += 1
    return a + b

# dont know why this works but it does.
def divide(a, b):
    quotient = 0
    c = 0
    d = 0
    while add(d, b) <= a:
        c = add(c, 1)
        d = add(d, b)
    return c

print("the answer is: ",divide(a, b))

time.sleep(3)
```

code_carte.cpp

```
{
  int Val_Inters;
  int nbr,i, codeessai;
  const int code= 3 ;

  int h = OpenDevice(0);

  for (;;)
  {
    Val_Inters = ReadAllDigital();
    if(( Val_Inters &  0x10) == 0x10)
    {
       cout<<"\n Détection code";
       cout<<"\n Attente inp4";
       do
       {
         Val_Inters = ReadAllDigital();
       } while (( Val_Inters &  0x08) != 0x08);

       codeessai= Val_Inters &  0x07;
       cout<<"\n Attente inp5";
       do
       {
         Val_Inters = ReadAllDigital();
       } while (( Val_Inters &  0x10) != 0x10);

       if ( codeessai == code )
       {
         cout<<"\n code ok";
         for(i=0;i<5;i++)
         {
           WriteAllDigital(0xFF);
           Sleep(500);
           WriteAllDigital(0x00);
           Sleep(500);
         }
       }
       else
       {
          cout<<"\n code non ok";
          WriteAllDigital(0xFF);
          Sleep(5000);
       }
    }
```

```
46: 27  Modifié  Insertion  Code
[C++ Avertissement] code_carte.cpp(59): W8066 Code inatteignable
[C++ Avertissement] code_carte.cpp(60): W8004 'h' est affecté à une valeur qui n'est jamais utilisée
```

```python
# svc model
ml_model = SVC()
hyper_parameter_candidates = {"C": [1e-4, 1e-2, 1, 1e2, 1e4],
    "gamma": [1e-3, 1e-2, 1, 1e2, 1e3],
    "class_weight": [None, "balanced"],
    "kernel":["linear", "poly", "rbf", "sigmoid"]}
scoring_parameter = "accuracy"
cv_fold = KFold(n_splits=5, shuffle=True, random_state=1)
classifier_model = GridSearchCV(estimator=ml_model,
    param_grid=hyper_parameter_candidates,
    scoring=scoring_parameter, cv=cv_fold)
classifier_model.fit(X_train, y_train)


# ann model
ml_model = MLPClassifier()
hyper_parameter_candidates = {"hidden_layer_sizes":[(20), (50),
   (100)], "max_iter":[500, 800, 1000],
   "activation":["identity", "logistic", "tanh", "relu"],
   "solver":["lbfgs", "sgd", "adam"]}
scoring_parameter = "accuracy"
cv_fold = KFold(n_splits=5, shuffle=True, random_state=1)
classifier_model = GridSearchCV(estimator=ml_model,
   param_grid=hyper_parameter_candidates,
   scoring=scoring_parameter, cv=cv_fold)
classifier_model.fit(X_train, y_train)
```

```python
# Spaghetti Code ##############################
def PRINTME(me):print(me)
import tkinter
x=y=z=1
PRINTME(z)
from tkinter import *
scrolW=30;scrolH=6
win=tkinter.Tk()
if x:chVarUn=tkinter.IntVar()
from tkinter import ttk
WE='WE'
import tkinter.scrolledtext
outputFrame=tkinter.ttk.LabelFrame(win,text=' Type into the scrolled text
                                              control: ')
scr=tkinter.scrolledtext.ScrolledText(outputFrame,width=scrolW,height=scrolH,
e='E'
scr.grid(column=1,row...
```

Prozedurale Programmierung. Wieso ist es wichtig?

- man kann nicht verstehen, wie es funktioniert und wieso
- schwer zu verstehen, zu erweitern, zu warten
- viel Copy-Paste (Code Reuse)
- ziemlich traurig

- jedes Teil hat ein klar definiertes Ziel
- leicht zu erweitern
- man kann alles verstehen
- Code Reuse durch Funktionen
- Lasagna ist einfach besser :)

Funktionen

Funktion: etwas, das einen oder mehrere Werte nimmt und einen oder mehrere Werte zurückgibt

- Hat einen Namen
- Kann eine Liste von (formalen) Parametern haben
- Kann einen Rückgabewert
- Hat eine Spezifikation

Syntax

```
def <name>([P1, ..., Pn]):
    #anweisungen
    [return <ergebnis>]
```

- Definition mit dem Keyword `def`
- man muss mit dem `()-Operator` die Funktion aufrufen
- `return` gibt den Wert zurück

Funktionen - Beispiel

```
def absolute_value(num):
    """
      Diese Funktion gibt den absoluten Wert
         einer eingegebenen Zahl zurück
    """

    if num >= 0:
        return num
    else:
        return -num

def main():
   print(absolute_value(2))
   print(absolute_value(-4))

main()
```

eine Funktion ohne Spezifikation ist nicht vollständig

```
1   def f (k):
2       v = 2
3       while v < k and k%v:
4           v += 1
5       return v>=k
```

- Könnt ihr bestimmen, was der Code ausgibt?
- Hat es länger als ein paar Sekunden gedauert?
- Jede Funktion hat eine Spezifikation, die besteht aus:
  - Eine kurze Beschreibung
  - Typ und Beschreibung aller Parameter
  - Bedingungen für Eingabeparameter
  - Typ und Beschreibung für den Rückgabewert
  - Bedingungen, die nach der Ausführung erfüllt sein müssen
  - Ausnahmen

Funktionen


```
1   def maximum (x,y ):
2       """
3       Gibt das Maximum von zwei Werten zurück
4       input: x,y - die Parameter
5       output: der größte der Parameter
6       Error: TypeError die Parameter dürfen nicht vergleichen werden
7       """
8       if x>y:
9           return x
10      return y
11
```

Funktionen

Jede Funktion **muss** enthalten:

- sinnvolle Namen (für Parameter und Namen)
- Kommentare
- Eine spezifikation

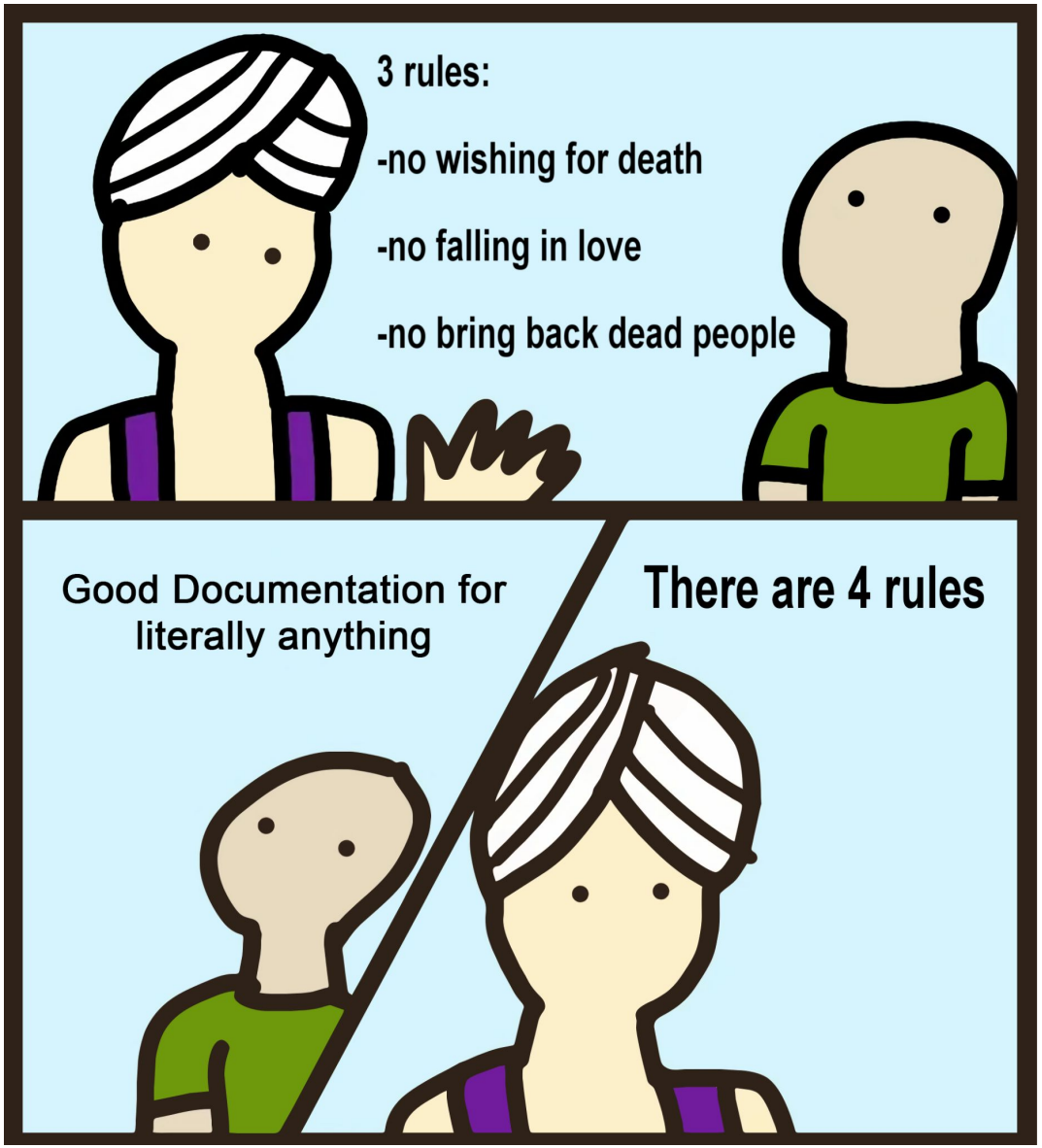
Testen!

- **man muss jede non-UI Funktion testen (kommt später)**

Funktionen

das bedeutet:
Dokumentation ist wichtig

- sinnvolle Namen
  - für Parameter und Namen
- Kommentare
- Eine spezifikation



Optionale Parameter

```
1   def test(param = 'Hallo'):
2       print (param)
3
4
5   def main():
6       test()
7       test('World!')
8
9   main()
10
11  '''
12  output:
13  Hallo
14  World!
15  '''
16
```

Sichtbarkeit und Blöcke. Teil II

- **Block:** ein Programmabschnitt, der als eine Einheit ausgeführt wird
- Blöcke sind durch einen Einrückungslevel definiert bzw. markiert
- eine Funktion ist ein Block
- ein Block wird innerhalb eines Execution Frame (Aufrufrahmen) ausgeführt
- Wenn eine Funktion aufgerufen wird, wird ein neuer Execution Frame erstellt

Aufrufen einer Funktion (Execution Frame)

Ein Execution Frame enthält:

- Einige administrative Informationen (zum Debugging verwendet)
- Informationen über, wo und wie die Ausführung fortgesetzt wird
- Definiert zwei Namespaces, den lokalen und den globalen Namespace, die sich auf die Ausführung des Codeblocks auswirken
- Ein Namespace ist eine Zuordnung von Namen zu Objekten.
- Ein bestimmter Namespace kann von mehr als einem Execution Frame referenziert werden

Aufrufen einer Funktion (Execution Frame)

```python
global_name = 10

def funktion ():
    local_name = 100

    print (global_name)
    print (local_name)

    print (locals(), globals())

funktion()
```

```
10
100
{'local_name': 100} {'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x7f700c0e7970>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__': 'main.py', '__cached__': None, 'global_name': 10, 'funktion': <function funktion at 0x7f700c0cc3a0>}
```

global vs local

```python
global_name = 10

def funktion ():
    local_name = 100

    global global_name

    global_name = 101

    print (global_name)
    print (local_name)



funktion()
print (global_name)

'''
Output

101
100
101
'''
```

```python
global_name = 10

def funktion ():
    local_name = 100

    global_name = 101

    print (global_name)
    print (local_name)



funktion()
print (global_name)

'''
Output

101
100
10
'''
```