* Fließkommazahl mit "," statt "." schreiben --> Tupel mit 2 int statt 1 float:
```
  f = 1,5
  print(f, type(f)) --> (1, 5) <class 'tuple'>
```
  GRUND: Tupel entsteht sobald ein Komma vorkommt, runde Klammern außenrum sind
         nicht notwendig (aber optisch leichter erkennbare Form)
  GRUND: Fließkommazahlen sind immer mit "." zu schreiben (jede Sprache).

* "," am Zeilenende --> Tupel statt Wert:
```
  var = 123      --> var hat Wert 123
  var = 123,     --> var hat Wert (123,) d.h ein Tupel
```

* ";" am Zeilenende ist überflüssig (Statement-Abschluß):
```
  var = 123
  var = 123;
```
  GRUND: In Python gilt das Zeilenende als Ende einer Anweisung.

* Ein vergessenes Komma "," in einer Sequenz von Strings kann dazu führen,
  dass versehentlich 2 Elemente zu einem Element zusammengezogen werden:
```
  var = ("aa" "bb")  --> var hat Stringwert "aabb" (kein Tupel)
  var = ("aa","bb")  --> var hat Tupelwert ("aa", bb")
  ("a", "b", "c" "d", "e" "f") --> ("a", "b", "cd", "ef")
  ("a",
   "b",
   "c"
   "d",
   "e"
   "f") --> ("a", "b", "cd", "ef")
```
  GRUND: Direkt hintereinanderstehende Strings (ohne Operator dazwischen)
  werden von Python zu einem String zusammengezogen (analog Verkettung per "+")
  (auch auf mehreren Zeilen verteilte Strings falls sie in Klammern stehen)

* Verwechslung von "+=" und "=+" (analog C/C++/Java):
```
  v =  5
  v += 1  -->  6
  v =+ 1  -->  v = +1  -->  v = 1  -->  1
```
  Verwechslung von "-=" und "=-" (analog C/C++/Java):
```
  v =  5
  v -= 1  -->  4
  v =- 1  -->  v = -1  -->  v = -1  -->  -1
```
  GRUND: Bei "=+" / "=-" zählt Plus/Minuszeichen als (überflüssiges) Vorzeichen.

* Nutzung von "++" oder "--" zum Inkrementieren/Dekrementieren von Werten
  (wie in C/C++/Java üblich):
```
  erg = ++v        # OK, aber identisch zu: erg = v
  erg = --v        # OK, aber identisch zu: erg = v
  erg = ++a * --b  # OK, aber identisch zu: erg = a * b
  erg = v++        # --> SyntaxError: invalid syntax
  erg = v--        # --> SyntaxError: invalid syntax
  erg = a++ * b--  # --> SyntaxError: invalid syntax
```
  GRUND: Die Operatoren "++" und "--" gibt es in Python nicht (ABSICHTLICH!).
  Präfix-Form ++v, --v als DOPPELTES Vorzeichen interpretiert --> wirkungslos.
  Suffix-Form v++, v-- stellt einen Syntaxfehler dar.

* "assert" ist ein Schlüsselwort, keine Funktion, es erwartet 1 oder 2 durch
  Komma getrennte Werte (typischerweise eine Bedingung + einen String)
  (Klammern der Parameter führt zu Syntaxfehlern oder merkwürdigen Ergebnissen):
```
  assert TEST      # --> Korrekte Schreibweise
  assert TEST, MSG # --> Korrekte Schreibweise
  assert(TEST,MSG) # --> FEHLER!
  assert(TEST)     # --> FEHLER!
```

* "del" ist ein Schlüsselwort, keine Funktion
  (Klammern sind zwar erlaubt, aber unüblich):

```
    del V1           # --> Korrekte Schreibweise
    del V1, V2, ...  # --> Korrekte Schreibweise
    del(V1)          # --> OK, aber ungewöhnlich
    del(V1, V2, ...) # --> OK, aber ungewöhnlich

* Ein 1-elementiger Tupel sind mit einem "überflüssigen" Komma zu schreiben:
    ("abc",)    --> OK
    ("abc")     --> Falsch da kein Tupel sondern ein Wert (geklammerter Ausdruck)
  GRUND: Mehrdeutige Klammer (...) muss eindeutig gemacht werden:
    a) Rechenausdruck (Expression):   (1+2)*3   (123)    (1+2+3)
    b) Tupel (Komma notwendig):       ()        (123,)   (1, 2, 3)
    c) Funktions-Aufruf:              f()       f(123)   f(1, 2, 3)

* Eine LEERE MENGE ist speziell zu schreiben:
    set()    frozenset()     # --> OK
    set({})  frozenset({})   # --> OK
    {}                       # --> FALSCH da Dictionary
  GRUND: Mehrfachbedeutung von {...} = dict, set, frozenset

* Vergessene schließende Klammer (z.B. bei print(...), [...], {...}) oder
  vergessenes schließendes Stringende "..." '...' """..."""  '''...'''
  --> Fehlermeldung "SyntaxError" evtl. irreführend erst viele Zeilen später,
      da Aufteilung auf mehrere Zeilen innerhalb Klammern/String erlaubt ist.

* Methoden müssen in Klasse eingeschachtelt sein (korrekte Einrückung):
    class KLASSE:
        def METHODE1(self, ...):   # Methode Gehört zur Klasse
            ...
    def METHODE(self, ...):        # Funktion von Klasse unabhängig
        ...
  Nicht eingerückte Methoden nach Klasse sind Klassen-unabhängige Funktionen.

* Auf KLASSEN-VARIABLE kann per OBJEKT zugegriffen werden,
  solange die Klassenvariable NUR GELESEN wird.
  Sobald 1x per OBJEKT in die Klassenvariable GESCHRIEBEN wird,
  hat Objekt EIGENE gleichnamige Variable mit eigenem Wert.
    class C:
        var = 111
    c1 = C()
    c2 = C()
    print(C.var, c1.var, c2.var)    # --> 111, 111, 111
    c1.var = 222
    print(C.var, c1.var, c2.var)    # --> 111, 222, 111
    C.var = 333
    print(C.var, c1.var, c2.var)    # --> 333, 222, 333

* KLASSEN-VARIABLE nur über den Klassennamen ansprechen, nicht über das Objekt
  (wäre nur zum Lesen OK)
  GRUND: Solange nur gelesen wird identisches Objekt,
         sobald geschrieben wird --> verschiedene Objekte (COW)
    class K:
        attr = "klasse"
    o1 = K()
    o2 = K()
    print(K.attr, o1.attr, o2.attr)     # --> "klasse", "klasse", "klasse"
    o1.attr = "objekt"
    print(K.attr, o1.attr, o2.attr)     # --> "klasse", "objekt", "klasse"

* Klammern nach Bezeichner setzen oder nicht ist ein Riesenunterschied:
    var = name()   # --> Funktions-Aufruf/Objekt-Konstruktor/...
    var = name     # --> Referenz auf Funktion/Klasse/...

* Python konvertiert Datentypen NIE automatisch, das ist immer manuell zu tun:
    int(STR) <-> str(INT/FLOAT)
    int("123.4") --> PENG!
  Ausnahmen:
```

```
    1) Alle Zahlentypen können in Ausdrücken gemischt werden:
         bool <-> int <-> float <-> complex <-> Decimal <-> Fraction
    2) Bei if/while wird Bedingung automatisch nach bool(...) konvertiert
         if BED: ...
         while BED: ...
    3) print(...) konvertiert alle angegebenen Werte/Objekte per str(...)
       --> Kann garantiert jeden Wert/jedes Objekt ausgeben!

* Python verhält sich bei Datei-Input/Output wie ein UNIX/Linux-System:
  + Zeilenenden bleiben erhalten
  + Zeilenende "\r\n" (Windows-OS) <-> "\n" (UNIX/Linux)
  + Verzeichnistrenner "\" (Windows-OS) <-> "/" (UNIX/Linux)

* Die Zeichensatz-Codierung von Source-Code, Eingabe- und Ausgabe-Daten
  berücksichtigen (Python 3 geht standardmäßig von Unicode in UTF-8 Codierung
  aus).

* Für Bezeichner möglichst nur ASCII-Zeichen verwenden (Code 32-127, Teil von
  Unicode und hat UTF-8 Codierung).
    KONSTANTE = 3.141529
    variable = 123
    class KlassenName: ...
    def funktions_name

* Der Dateiname eines Moduls muss ein BEZEICHNER sein, damit es per "import"
  geladen werden kann (nur die Zeichen "a-zA-Z_0-9" verwenden, insbesondere
  kein Leerzeichen und kein Bindestrich "-" im Modulnamen nutzen!)

* Unterverzeichnis mit Modulen und Modul NICHT GLEICH nennen:
    Programm-Code
      # programm.py
      import verz.unter.modul as vum
      --> ModuleNotFoundError: No module named 'verz.unter'; 'verz' is not a package
    Verzeichnis + Dateistruktur:
      +-- programm.py
      +-- verz
      |   +--- unter
      |        +--- modul.py
      +-- verz.py

* Datentypen werden erst zur Laufzeit (Run-Time) überprüft, nicht während der
  Übersetzungszeit (Compile-Time), generelle Eigenschaft von Skript-Sprachen.
  --> Fehlermeldung TypeError, ValueError, ... erst zur Laufzeit

* Überschreiben von bereits vorhandenen Namen von Variablen, Funktionen, ...
  ist jederzeit ohne Warnung möglich (inkl. Typänderung).
  Ausnahme: Die 37 Schlüsselworte (if, else, ...) sind nicht umdefinierbar
  Typische Kandidaten (versehentlich verwendete eingebaute/interne Namen):
    len   = 0
    sum   = 12345
    min   = 0
    max   = 999
    any   = True
    all   = []
    str   = "hallo"
    int   = 123
    float = 123
    list  = [1, 2, 3]
    tuple = (1, 2, 3)
    dict  = {"a": 1, "b": 2, "c": 3}
    type  = "int"
    dir   = "C:\\Users\\Administrator\\Documents"

* Eigenes Python-Skript nicht so nennen wie ein Builtin-Modul oder eine
  bereits vorhandene interne Variable, Funktion, Klasse oder ...
  GRUND: Namen können jederzeit neu definiert werden
```

```
       (kommentarlos, ohne Fehlermeldungen)
  TIP: Unterstrich hinten dranhängen (z.B. "if_")
  KANDIDATEN: len sum min max any all str int list ... (Built-in Funktionen)

* Namenskonventionen und Reihenfolge-Vorgaben einhalten (PEP8):
    cls        # Klasse in Klassenmethode
    self       # Objekt in Methode
    other      # 2. Objekt in Methode
    *args      # Sammler für Positions-Argument in Funktions-Definition
    **kwargs   # Sammler für Schlüsselwort-Argument in Funktions-Definition
    *_         # Sammler für zu ignorierende Elemente in Tupel-Zuweisungs
    _          # Temporäre Variable (z.B. for _)
    NAME       # Public    Attribut/Methode in Klasse
    _NAME      # Protected Attribut/Methode in Klasse
    __NAME     # Private   Attribut/Methode in Klasse
    __NAME__   # Python-interne Variable (nie für eigene Zwecke nutzen!)
    KEYWORD_   # Schlüsselworte als Bezeichner (z.B. if_  while_)
    funk_tion  # Funktions-Bezeichner (Snake Case)
    Klasse     # Klassen-Bezeichner (Camel Case)
    KONSTANTE  # Konstanten-Bezeichner (Trump Case)
  GRUND: Programm für nächsten (durchschnittlichen) Programmierer schreiben!

* Vollständigen Import aller Elemente aus Modul per * nicht benutzen:
    from modul import *
  GRUND: Überschreibt alle gleichnamigen bereits vorhandenen Namen!
         Welche Namen werden überhaupt importiert und wie viele?

* Fehler NICHT UNTERDRÜCKEN, sondern abfangen und behandeln:
    try:       # Start
        ...    # "Überwachter" CODE: Fehler --> except angesprungen
    except:    # Fängt ALLE Fehler ab --> zu viele (z.B. Syntax, Name, Type, Value,
Assert, ...)!
        pass   # Unterdrückt ALLE Fehler --> Programmierer bekommt nichts mit!
  GRUND: Alle Fehler-Meldungen werden abgefangen + vollständig unterdrückt!
         Keine Fehler-Behandlung sondern Fehler-UNTERDRÜCKUNG!
         Erschwert Fehlersuche stark!

* Maximal "Exception" abfangen, nicht "BaseException"!
    except: ...   # entspricht "except BaseException: ..."
  GRUND: Programm mit exit()-Funktion und "Strg-C" (Cancel) nicht abbrechbar!

* Programm übersetzbar, aber Exception löst "NameError" aus
  GRUND: Exception-Namen werden erst im Fehlerfall gesucht ob vorhanden
         (nicht zur Übersetzungszeit)
  Analog bei Funktionen und Klassen: Erst bei Nutzung werden Namen darin geprüft

* Exception-Fall unwirksam
  GRUND: Vorher steht in der Kaskade eine allgemeinere Exception --> greift zuerst
         Python überprüft nicht, ob Exception-Kasakade disjunkt ist

* Test auf True/False ist "over-engineered" und zu spezifisch
  (Datentyp von "var" MUSS "bool" sein, sonst Ergebnis "False"):
    if var == True:       # --> if var:
    if var != True:       # --> if not var:
    if var == False:      # --> if not var:
    if var != False:      # --> if var:
    while var == True:    # --> while var:
    while var != True:    # --> while not var:
    while var == False:   # --> while not var:
    while var != False:   # --> while var:
  BESSER: if var:         # if --> Boolescher Kontext
          if not var:     # if --> Boolescher Kontext
  GRUND: "Wahr" oder "Falsch" sind ALLE WERTE,
         IDENTISCH mit "True" oder "False" ist fast KEIN Wert.
         Der Ausdruck nach if/while wird automatisch nach bool() konvertiert.
```

* Test auf leeren/gefüllten CONTAINER ist "over-engineered" und zu spezifisch
  (Datentyp MUSS "tuple", "list", "dict" sein, sonst Ergebnis "False"):
    if tpl == ():   # --> if not tpl:
    if tpl != ():   # --> if tpl:
    if lst == []:   # --> if not lst:
    if lst != []:   # --> if lst:
    if dct == {}:   # --> if not dct:
    if dct != {}:   # --> if dct:
    if mng == set():   # --> if not mng:
    if mng != set():   # --> if mng:
  GRUND: Leere Container ergeben "False" in booleschem Kontext
         Gefüllte Container ergeben "True" in booleschem Kontext
         {} ist kein set, sondern ein dict --> Datentyp verschieden, Wert nicht!

* Defaultwert eines Funktions-Parameters mit MUTABLE Objekt ist problematisch:
    def funk(var=[]): ...
  BESSER: def funk(var=None):
              if var is None:
                  var = []
  GRUND: Objekt [] wird nur 1x erzeugt bei Funktions-Definition
         und bei jedem Aufruf ohne Parameter wiederverwendet,
         d.h. Änderungen daran bleiben zw. Funktions-Aufrufen erhalten.

* Eigene Funktion liefert "None" als Ergebnis zurück (unabhängig vom Aufruf).
  GRUND: "return" ohne Wert oder "return ERGEBNIS" am Funktions-Ende vergessen!
         Dann findet automatisch ein "return None" am Funktions-Ende statt!

* Folgende Initialisierungen sind gefährlich:
    a = b = {}
    a = [[] * 10]
  GRUND: Gleiches Dictionary- oder Listen-Objekt wird mehrfach verwendet
  BESSER: (a, b) = ({}, {})
          (a, b) = ([], [])
          a = list(map(lambda x: [], range(10))

* Test auf Datentyp einer Variablen nicht so durchführen:
    if type(var) == int:
    if type(var) == int or type(var) == float:
    if type(var) in (int, float):
  sondern BESSER so:
    if isinstance(var, int):
    if isinstance(var, (int, float)):
  GRUND: Vererbung bei "type()" NICHT berücksichtigt, bei "isinstance()" schon!

* Selber zählen und indizieren vermeiden:
    arr = ["hallo", "welt", "hier", "bin", "ich"]
    for i in range(len(arr)):   # --> for elem in arr:
        print("ELEM", arr[i])   #   print("ELEM", elem)
  GRUND: "Verzählen" möglich, Performance, Speicherersparnis, Iterator, ...
  ZITAT: "In Python zählt man nicht selber, sondern lässt Python zählen."
         (mit enumerate, zip, sum, min, max, any, all, ...)

* Mit Iteratoren oder Generatoren arbeiten,
  anstatt Listen/Tupel/... zu verwenden
  GRUND: Performance + Speicherersparnis + ...

* Quellcode per "Copy-and-Paste" aus Webseiten/PDF/... kopieren ist gefährlich!
  "Copy-and-Paste" von Code transferiert oft die EINRÜCKUNG nicht korrekt.
  GRUND: Einrückung (Blockstruktur) geht (meist) verloren (teilweise oder
         vollständig) und muss manuell korrigiert werden (es gibt keinen
         Automatismus dafür, da die Einrückung Teil der Programm-Semantik ist)
  GRUND: Leerzeichen/Tabulator falsch übertragen --> Andere Bedeutung möglich
  --> Einrückung unbedingt Zeile für Zeile manuell überprüfen!

* "raise" vor FehlerKlasse vergessen:
     AssertionError("Problem")

```
  --> Anweisung wird ignoriert

* Raw- und Format-Strings falsch herum geschrieben sind normaler String:
    "R..."   # Korrekt: R"..."
    "F..."   # Korrekt: F"..."

* Es gibt 3 Funktionen zum Vergleich von REGEX mit String:
    re.search()       # Kein automatischer Anker
    re.match()        # Anker ^ automatisch vorne
    re.fullmatch()    # Anker ^ $ automatisch vorne + hinten
  --> IMMER re.search und bei Bedarf die Anker ^ $ verwenden!

* Python verzögert Auswertungen möglichst lange:
    print(range(10))       # --> "range(10)"
  --> Muss daher manchmal dazu gezwungen werden:
    print(list(range(10))) # --> "[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]"

* Grenzen sind INKONSISTENT (obere Grenze erreicht oder nicht?):
    range(10)               # --> 0..9 (10 Werte!)
    range(1, 10)            # --> 1..9 ( 9 Werte)
    var[0:10]               # --> var[0] .. var[9] (9 Elemente)
    random.randint(1, 10)   # --> 1..10
  TIP: Obere Grenze bei Range immer mit Addition/Subtraktion von 1 schreiben:
    range(1, 10+1, 1)       # --> 1..10
    range(10, 0-1, -1)      # --> 10..0

* Auf Wert "None" testen geht auf 2 Arten:
    if var == None:   # OK (un-pythonic)
    if var is None:   # OK + empfohlen (pythonic)
  GRUND: Bedeutet das Gleiche (None ist SINGLETON, identisch mit Klasse)

* Datentyp "NoneType" gibt es ab Python 3.6 nicht mehr
  --> Statt dessen type(None) verwenden falls notwendig

* Vergleich "==" statt Zuweisung "=" verwendet wird ohne Fehlermeldung ignoriert:
    var == 123
  GRUND: Wertet Vergleich aus und verwirft Ergebnis True/False.
         --> Variablenwert bleibt gleich.

* Ausdrücke (Expressions) für sich ohne Zuweisung oder Funktions-Aufruf außenrum
  werden ignoriert (sind aber kein Fehler).
    """docstring ..."""     # Für mehrzeilige Kommentare + Docstrings benutzt
    var == 123              # Vergleich (statt Zuweisung)
    (1 + 2) * 3 ** 4        # Rechenausdruck (ohne Zuweisung)
    AssertionError("...")   # "raise" davor vergessen

* Walrus-Operator := ist keine Zuweisung, sondern ein Operator.
  Verhält sich aber wie eine Zuweisung:
    print(erg := 1 + 2 * 3 / 4)   # OK: erg == 2.5
    erg := 1 + 2 * 3 / 4          # FEHLER
    (erg := 1 + 2 * 3 / 4)        # OK
  GRUND: Nur in geklammertem oder weiterverwendetem Rechenausdruck erlaubt!

* Zirkuläre Struktur ist möglich --> Entscheidung über Freigabe per Reference
  Counting funktioniert nicht mehr --> Garbage Collector notwendig:
    a = []        #
    b = [a]       #
    a.append(b)   #
    print(a, b)   # --> [[[...]]] [[[...]]]

* Vergleich "==" und "!="     vergleicht die WERTE     von 2 Objekten.
  Vergleich "is" und "is not" vergleicht die IDENTITÄT von 2 Objekten.
    (OBJ1 is OBJ2 ist ABKÜRZUNG für id(OBJ1) == id(OBJ2)

* Bei "==" muss der Datentyp der Gleiche sein, sonst kommt IMMER "False" heraus.
  (außer bei Zahlen bool <-> int <-> float <-> complex <-> Decimal <-> Fraction)
```

```
  True == 1               # --> True
  False == 0              # --> True
  123 == 123.0            # --> True
  123 == (123.0+0j)       # --> True
  123.0 == (123.0+0j)     # --> True
  123 == "123"            # --> False
  (1, 2, 3) == [1, 2, 3]  # --> False
  0 == "0"                # --> False
  "" == ()                # --> False
  [] == ()                # --> False
```

* Gleicher Typ + Wert --> Meist nur 1x angelegt und mehrfach referenziert
  --> Objekt nicht nur gleich, sondern identisch

```
  a = "abc"
  b = "abc"
  print(a, b, a == b, a is b)    # --> abc abc True True
```

Aber nicht immer:

```
  c = "ab"
  c += "c"
  print(a, c, a == c, a is c)    # --> abc abc True False
```

* Gleiche ID heißt auch gleicher WERT!
  Gleicher WERT heißt aber nicht unbedingt gleiche ID!

```
  id(OBJ1) is id(OBJ2)   # --> OBJ1 == OBJ2
  s1 = "hallo welt"      # --> s1 = "hallo welt"
  s2 = "hallo"           # --> s2 = "hallo"
  s2 += " welt"          # --> s2 = "hallo welt"
  s1 == s2               # --> True
  s1 is s2               # --> False
```

* Gleiche Speicherstelle wird evtl. nacheinander für 2 verschiedene Objekte
  ALTERNIEREND verwendet (wenn 1. Objekt nicht mehr referenzierbar ist und
  2. Objekt den gleichen Typ hat und genauso viel Speicher benötigt).
  a) String:

```
  t = "hallo"
  print(t, id(t))
  while True:
      del t
      t = "zorro"
      print(t, id(k))
```

  b) Ganze Zahl:

```
  i = 123456;
  while True:
      i += 1; print("I", i, id(i))
      i += 1; print("I", i, id(i))
```

* Die Objekte -5 .. 256 sowie 0.0, 1.0, -1.0, True, False, None werden von
  Python vorab beim Programmstart erzeugt (da oft benötigt).
  --> Haben ungewöhnlich kleine + feste ID.

* Ganze Zahlen haben als ID ihren Wert.

```
  var = 123
  print(var, id(var))   # --> 123 123
```

* Type Annotation/Hint wird in Python verglichen mit anderen Programmiersprachen
  "verkehrt herum" geschrieben und mit einem ":" (Doppelpunkt) zwischen dem
  Bezeichner und dem zugehörigem Datentyp getrennt:

```
  var: int = 123
  def funk(a: int, b: str, c: float) -> bool:
```

  In C/C++ schreibt man den Typ "anders herum" und ohne ":" (Doppelpunkt)

```
  int var = 123
  bool funk(int a, str b, float c)
```

  GRUND: Erst sehr spät hinzugefügt (ab Python 3.5)

* Type Annotation/Hint wird von Python nur akzeptiert, aber komplett ignoriert.
  --> Erst von Tools wie "mypy", "pylint", ... werden die Datentypen überprüft.