Python Gotchas

* Fließkommazahl mit "," statt "." schreiben --> Tupel mit 2 int statt 1 float:
```
  f = 1,5
  print(f, type(f)) --> (1, 5) <class 'tuple'>
```

* Verwechslung von "+=" und "=+" (analog C/C++/Java):
```
  v =  5
  v += 1  -->  6
  v =+ 1  -->  v = +1  -->  v = 1  -->  1
```

* "," vergessen --> "aa" "bb" wird zu String zusammengezogen
```
  var = ("aa" "bb")  --> var hat Stringwert "aabb" (kein Tupel)
  var = ("aa","bb")  --> var hat Tupelwert ("aa", bb")
```

* "," 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;
```

* assert ist ein Schlüsselwort, keine Funktion:
```
  assert(TEST,MSG) --> IMMER OK (da 2-elem Tupel) --> assert TEST, MSG
  assert(TEST)     --> KEIN 1-elem Tupel
  assert TEST      --> Korrekte Schreibweise
  assert TEST, MSG --> Korrekte Schreibweise
```

* 1-elem Tupel schreiben
```
  ("abc",)    --> OK
  ("abc")     --> Falsch da kein Tupel sondern ein Wert (geklammerter Ausdruck)
```
  GRUND: Doppelbedeutung von (...) = Geklammerter Ausdruck, Funktionsaufruf, Tupel

* Leere Menge schreiben
```
  set()       --> OK
  set({})     --> OK
  {}          --> Falsch da kein Set sondern Dictionary
```
  GRUND: Doppelbedeutung von {...} = dict, set, frozenset

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

* Auf Klassenvariable kann per Objekt zugegriffen werden,
  solange die Klassenvariable nur gelesen wird.
  Sobald 1x per Objektzugriff in Klassenvariable geschrieben wird,
  hat Objekt eigene gleichnamige Variable mit eigenem Wert.

```
    class C:
        var = 111
    c1 = C()
    c2 = C()
    print(C.var)    # --> 111
    print(c1.var)   # --> 111
    print(c2.var)   # --> 111
    c1.var = 222
    print(C.var)    # --> 111
    print(c1.var)   # --> 222
    print(c2.var)   # --> 111
    C.var = 333
    print(C.var)    # --> 333
    print(c1.var)   # --> 222
    print(c2.var)   # --> 333
```

* Klassenattribut nur über Klassennamen ansprechen, nicht über Objekt
  (ware aber prinzipiell OK)
  GRUND: Solange nur gelesen wird identisches Objekt,
         sobald geschrieben wird --> verschiedene Objekte (COW)

```
  class K:
      attr = "klasse"
      pass

  o1 = K()
```

```
    o2 = K()
    print("ATTR1:", K.attr)     # --> "klasse"
    print("ATTR1:", o1.attr)    # --> "klasse"
    print("ATTR2:", o2.attr)    # --> "klasse"
    o1.attr = "objekt"
    print("ATTR1:", K.attr)     # --> "klasse"
    print("ATTR1:", o1.attr)    # --> "objekt"
    print("ATTR2:", o2.attr)    # --> "klasse"

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

* Python konvertiert Datentypen NIE automatisch, immer manuell zu tun:
    int(STR) <-> str(INT/FLOAT)
    int("123.4") --> PENG!
  Ausnahmen:
    Zahlen können in Ausdrücken gemischt werden
      bool <-> int <-> float <-> complex
    Bei if/while wird Bedingung automatisch nach bool(...) konvertiert
      if BED:
      while BED:
    print(...) wandelt alle Werte nach str(...) um --> Kann jeden Wert ausgeben

* Python verhält sich bei Input/Output folgendermaßen:
  + Zeilenenden bleiben erhalten
  + Zeilende "\r\n" (Windows-OS) <-> "\n" (UNIX/Linux)    # Nur im Textmodus!
  + Verzeichnistrenner "\" (windows-OS) <-> "/" (UNIX/Linux)

* Zeichensatz (Codierung) von Source-Code, Eingabe- und Ausgabe
  Daten berücksichtigen (Python 3 geht standardmäßig von Unicode
  in UTF-8 Codierung aus). Möglichst nur ASCII-Zeichen verwenden
  (Code 0-127, ist Teil von Unicode und hat UTF-8 Codierung).

* Dateiname von Modul muss ein BEZEICHNER sein, damit es per "import" geladen
  werden kann (nur "a-zA-Z_0-9", insbesondere kein Bindestrich "-" im Namen!)

* Typen werden zur Laufzeit geprüft, nicht zur Übersetzungszeit
  (generelle Skript-Sprachen-Eigenschaft)
  --> Fehlermeldung TypeError, ValueError, ... erst zur Laufzeit

* Überschreiben von Namen von Variable, Funktionen, ...
  jederzeit ohne Warnung möglich (inkl. Typänderung)
  Ausnahme: 37 Schlüsselworte (if, else, ...)

* Namenskonventionen und Reihenfolgevorgaben einhalten (PEP8):
    cls        # Klasse in Klassenmethode
    self       # Objekt in Methode
    other      # 2. Objekt in Methode
    *args      # Positions-Argumentsammler in Funktions-Definition
    **kwargs   # Schlüsselwort-Argumentsammler in Funktions-Definition
    _          # Temporare Variable (z.B. for _)
    _NAME      # Protected-Attribut/Methode in Klasse
    __NAME     # Private-Attribut/Methode in Klasse
    KEYWORD_   # Schlüsselworte als Bezeichner (z.B. if_  while_)
    funk_tion  # Funktions-Bezeichner
    Klasse     # Klassen-Bezeichner
    KONSTANTE  # Konstanten-Bezeichner
  GRUND: Programme sehen einheitlich aus für nächsten Programmierer

* from modul import *
  GRUND: Uberschreibt evtl. viele bereits vorhandene Namen
         Welche Namen werden überhaupt importiert? viele?

* try:
     ...
  except:
     pass
  GRUND: Fehlermeldungen werden vollständig unterdrückt!
         Erschwert Fehlersuche stark!

* Test auf True oder False "over-engineered" und zu spezifisch:
    if var == True:
    if var != True:
    if var == False:
    if var != False:
```

```
  BESSER: if var:             # if --> Boolescher Kontext
          if not var:
  GRUND: "Wahr" sind fast alle Werte, identisch mit "True" ist fast keiner
         Ausdruck nach if/while wird automatisch nach bool konvertiert

* Eigenes Python-Skript so nennen wie Builtin-Modul
  oder Variable oder Funktion oder Klasse oder ...
  GRUND: Namen können jederzeit neu definiert werden
         (kommentarlos, ohne Fehlermeldungen)
  TIP: Unterstrich hinten dranhängen (z.B. "if_")
  KANDIDATEN: dir len list int dict sum min max any all (Built-in Funktionen)

* def funk(var=[]):
  BESSER: def funk(var=None):
              if var is None: var = []
  GRUND: Objekt [] wird nur 1x erzeugt bei Definition
         und wird bei jedem Aufruf ohne Parameter wiederverwendet,
         d.h. Änderungen daran bleiben zw. Funktionsaufrufen erhalten

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

* Test auf Datentyp einer Variablen nicht so:
    if type(var) == int:
    if type(var) in (int, float):
  sondern BESSER so:
    if isinstance(var, int):
    if isinstance(var, (int, float)):
  GRUND: Vererbung wird sonst nicht berücksichtigt

* Nicht selber zählen (indizieren), sondern Python zählen lassen
  (OK sind enumerate, zip, sum, min, max, any, all, ...)
  GRUND: "Verzählen" vermeiden, Performance, Speicherersparnis, Iterator, ...

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

* Quellcode per Copy-and-Paste aus Webseiten/PDF/... kopieren ist gefährlich!
  GRUND: Einrückung (Blockstruktur) geht verloren (meist) und muss
         manuell korrigiert werden (es gibt keinen Automatismus dafür
         da die Einrückung die Programm-Semantik beeinflusst)

* "raise" vor FehlerKlasse vergessen
     AssertionError("Problem")
  --> Anweisung wird ignoriert

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

* range(1,10) --> 0..9
  var[0:10] --> var[0] .. var[9]
  random.randint(1,10) --> 1..10
  INKONSISTENT!

* Auf Wert "None" testen geht auf 2 Arten:
    if var == None:
    if var is None:
  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

* Vergleich == statt Zuweisung = verwendet wird ohne Fehlermeldung ignoriert:
```

```
    var == 123
  GRUND: Wertet Vergleich aus, verwirft Ergebnis True/False, Variable bleibt gleich

* Walrus-Operator := ist keine Zuweisung, verhält sich aber wie eine Zuweisung.
    print(erg := 1 + 2 * 3 / 4)
  Nur in einem Rechenausdruck erlaubt.

* "Copy-and-Paste" von Code transferiert oft die Einrückung nicht korrekt.
  Leerzeichen fehlerhaft übertragen --> Statt Syntaxfehler andere Bedeutung möglich
  Einrückung zeilenweise manuell überprüfen (kein autom. Formatierung möglich)

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

* Gleicher Typ + Wert --> Meist nur 1x angelegt und mehrfach referenziert
    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 Speicherstelle wird evtl. für 2 verschiedene Objekte verwendet
  (wenn das 1. Objekt nicht mehr referenzierbar ist und das 2. Objekt den
  gleichen Typ hat und genauso viel Speicher benötigt).
    t = "hallo"
    print(t, id(t))
    del h
    u = "zorro"
    print(z, id(z))

* Die Objekte -5 .. 256 sowie 0.0, 1.0, -1.0, True, False, None  werden von
  Python standardmäßig vor dem Programmlauf erzeugt (da oft benötigt).

* Direkt hintereinanderstehende Strings werden von Python zu
  einem String zusammengezogen (analog Verkettung per "+")
  D.h. ein "vergessenes" Komma in einer Sequenz kann dazu führen,
  dass versehentlich 2 Elemente zu einem Element zusammengezogen werden.
    ("a", "b", "c" "d", "e" "f") == ("a", "b", "cd", "ef")
```