Programmierung mit Python

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2024-09 Dozent: Karsten Flügge

Themen des Tages

- Operatoren
- Listen, Tupel
- list, tuple

Operatoren

Vorrang von Operatoren, Gegebenenfalls Operationen Klammern oder in mehrere Zeilen teilen.

'Präzedenz' ≈ Bindung (Gruppierung Reihenfolge Ordnung)

```
(...) [] {}          # Klammern     ALS ERSTES
x[i], x(...), x.y  # Zugriff, Aufruf  z.B. .real index
**                 # Potenz a^b a**b
+x, -x, ~x         # Vorzeichen, bitweise Negation  (-2**2 = -4)
*, /, //, %        # ÷ Punktrechnung   3 * True == 3
+, -               # Strichrechnung
<<, >>             # Bitverschiebung
~                  # bitweises NOT
&                  # bitweises UND
^                  # bitweises XOR
|                  # bitweises ODER
< <= > >= in not is      # Vergleiche
== !=
not                # logisches NICHT  True, False
and                # logisches UND  (&& gibts nicht)
or                 # logisches ODER
in, ... ?
=  :=  += ...      # Zuweisung; kein Ausdruck   ALS LETZTES
```

Operatoren

Operatoren werden i.d.r von links nach rechts ausgewertet
10 - 4 - 2 == (10 - 4) - 2

Ausnahme: ** Potenzrechnung von rechts nach links "assoziativ"
(2 ** 3) ** 2 == 2 ** (3 ** 2)
2 ** 3 ** 2 == 2 ** (3 ** 2)

Achtung: % und // sind nicht 'assoziativ' (Klammer ist wichtig)

```
>>> (14 % 5) % 3
1
>>> 14 % (5 % 3)
0

>>> 10 * 3 // 10
3
>>> 10 * ( 3 // 10 )
0
```

Operatoren

Zuweisung = seit 3.8 :=

⚠️ Vorsicht, NIE Vergleich == mit Zuweisung = verwechseln!

```
>>> x = 1
>>> y = 2
>>> x == y
False / True  "Boolean" Wahrheitswert
>>> x = y
>>> x == y
True
```

`nil` ist null aus anderen Sprachen

gibt keine Pointer

Logisch: and, or, not, ^ (xor)

*= /= %= += -= # in place arithmetic / compound arithmetic zB compound addition

Operatoren

Mathematisch: + - / * % (modulo rest) ** (hoch, potenz, ^, ^^)
werden abgebildet auf Funktionen mit speziellem Namen
__add__

Vergleich: == , >, <
__eq__ __gt__ __lt__ equal greater / less than

Logisch: and, or, not, ^ (xor)

Operatoren

% (modulo = Rest von Division) 5 % 2 == 1

// (ganzzahlige Teilung) 5 // 2 = 2 abgerundete Teilung

// Floor Division bzw. **Ganzzahldivision mit Abrundung**.

```
>>> 5/3  normal Teilung
1.6666666666666667
>>> 5//3 abgerundetet auf int
1
```

Nützlichkeit von %:

ist x gerade <> x%2 ==0

alle 100 Schritte etwas ausgeben:

if step % 100 == 0 : print("Fortschritt!")

Operatoren

Operatoren + - / * % ... werden abgebildet auf Methoden mit speziellem Namen

```
'__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
'__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__',
'__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__',
'__getstate__', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__',
'__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__',
'__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__',
'__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
'__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__',
'__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__',
'__xor__', 'as_integer_ratio', 'bit_count', 'bit_length', 'conjugate', 'denominator',
'from_bytes', 'imag', 'is_integer', 'numerator', 'real', 'to_bytes']
```

Ungleichheit

Ungleichheit

3 != 4 heisst "3 ungleich 4"

Hintergrund in c etc darf man !x schreiben fuer "nicht x"

⚠️ in python schreibt man **not x** für logische Negierung!

not x **==** y ist das selbe wie x **!=** y

Syntaktischer Zucker

Oft möchte man eine Variable um eins erhöhen

x = x + 1

dafür gibt es spezielle Zuweisung

x += 1 # für hochzählen sehr nützlich

Der 'Trick' geht mit verschiedenen Operatoren:

*= /= %= += -= # in place arithmetic / compound arithmetic operators zB compound addition

Aufgaben

Finde Beispiele von zwei Operatoren, wo die Reihenfolge wichtig ist, also wo mit und ohne Klammer ein anderes Ergebnis herauskommt
z.B. 2 + 3 * 4 != (2 + 3) * 4

```
**                  # Potenz = hoch
+x, -x              # Vorzeichen
*, /, //, %         # Punktrechnung ÷    3 * True == 3
+, -                # Strichrechnung
== != < <= > >=         # Vergleiche
and, or, not
```

Operatoren Ausdruck

Die Anwendung von Operatoren sind ein Beispiel für einen sogenannten Ausdruck (expression)

"Alles was bei einer Variablen Zuweisung rechts stehen kann"

x = 3 * 3
x = print('hallo') # liefert None 'nichts'

Ein Ausdruck welcher nicht ausgegeben oder zugewiesen wird macht normalerweise wenig Sinn!

```
len('hallo') # ⚠ use it!?
print(len('hallo')) # ok, verwendet
laenge = len('hallo') # ok, verwendet (später…)
```

String Operatoren

```
"a" in "Hallo"   # __contains__

Modulo zum Werte in Text einsetzen:

"Hello %s, you are %d years old" % ("friend", 99)   # __mod__
# 'Hello friend, you are 99 years old'

Nutze besser format string f"Hello {name}"

plus ist klar
mal : Wiederholung
Vergleich: Lexikalische Sortierung "a" < "b"

Achtung: Groß vor klein!
>>> 'a' < 'B'
False

Vergleich klar
```

Vorschau Funktionen vs Methoden vs Operatoren

Führen Aktionen aus / holen Informationen
Rückgabe **Typ** kann unterschiedlich sein

Funktionen
stehen unabhängig von Objekten (ohne Punkt .)
print(x)
help()
dir(x) **#** listet alle Methoden vom Objekt x
len(x) # int

Methoden
hängen an Objekten (Dinge mit Eigenschaften), werden mit '.' Punkt aufgerufen, weiter mit **()**
z.B.
"HELLO"**.**lower() => "hello"
Alle Methoden zwei Unterstrichen rufen wir NICHT direkt auf, sondern:

Operatoren
hängen an Objekten aber werden ohne . aufgerufen
"a" + "b" und werden **intern** auf Methoden abgebildet z.B. __add__ "häßlich by design"

Kettenvergleich

In Mathe darf man schreiben 1 < 2 < 3 ... in python auch!

generell werden lange Vergleichsketten aufgespalten
a < b < c => (a < b) and (b < c)

aber:

5 > 3 == True wird aufgedröselt zu
(5 > 3) and (3 == True) ⚠️⚠️⚠️ warum??? generell werden lange Vergleichsketten aufgespalten

5 > 3 is True wird aufgedröselt zu
(5 > 3) and (3 is True) ⚠️⚠️⚠️

```
>>> 3 < 4 is True
False

>>> (3 < 4) is True
True
```

syntactic diabetes

Logische Operatoren

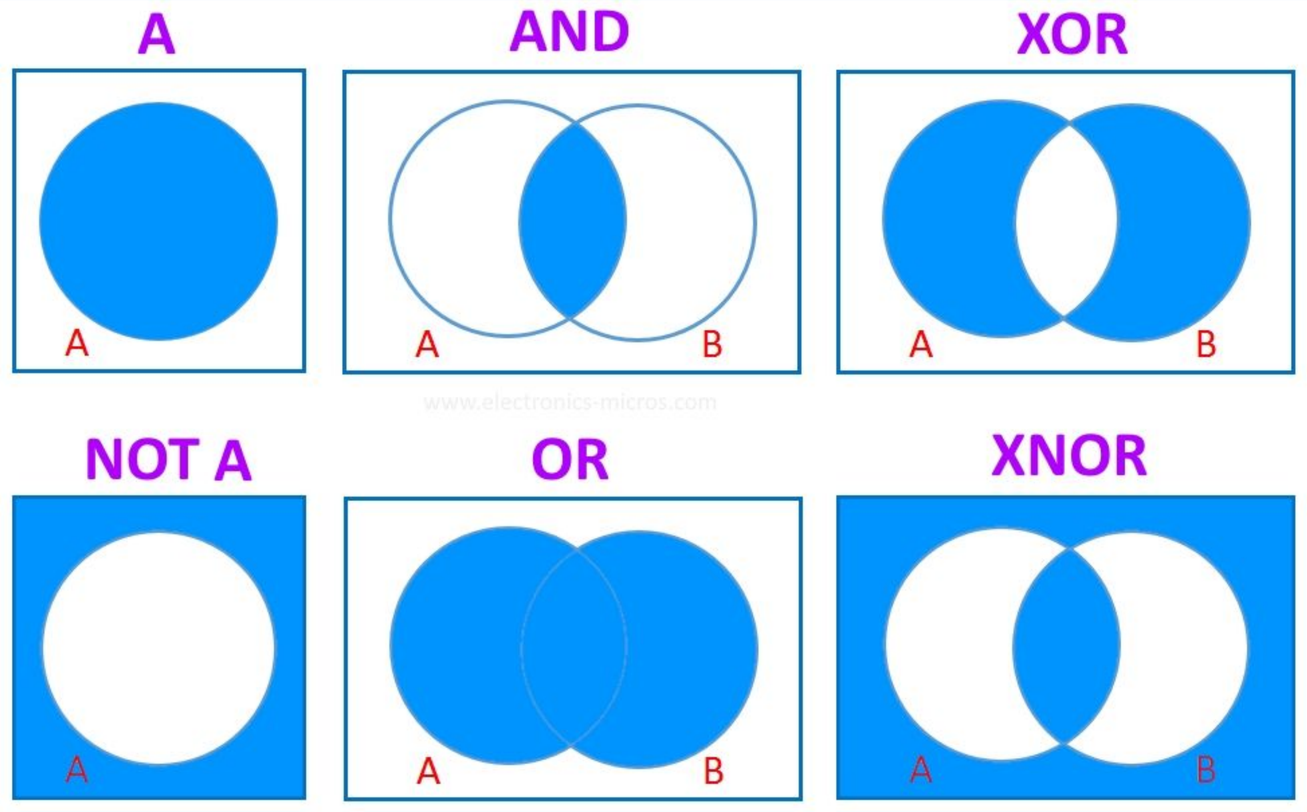
not vor **and** vor **or** (universell in Mathe und anderen Sprachen)
"Boolsche Algebra"
not ≈ **-**negation , ***** ≈ and , **+** ≈ oder
1*1 = 1
1*0 = 0 <> True and False

True or False and False # True == True or (False and False)
(True or False) and False # False

not **# logisches NICHT True, False**
and **# logisches UND** (&& gibts nicht)
or **# logisches ODER**

Logische Operatoren

einschliessendes or via Betonung "Käse oder Butter? ja beides! ok"
ausschliessendes or = "xor" entweder oder "Käse *oDer* Butter? beides. gibts nicht."



Logische Operatoren

A	B	UND `and`	ODER `or`	NICHT-UND `NAND`	XOR
False	False	False	False	True	False
False	True	False	True	True	True
True	False	False	True	True	True
True	True	True	True	False	False

Listen

Listen sind container (≈Arrays, Felder) zum sammeln von verschiedensten Objekten

list eckige klammer und komma

```
wochentage = ["Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag", "Samstag", "Sonntag"]

for i in wochentage:
  print(i)

print(wochentage[0]) # "Montag"

so wie bei Zeichen in Text fangen wir bei 0 an zu zählen!
```

Listen

Listen können verschiedene Dinge enthalten:

```
mischSack = ["Montag", 1, 4.4]

for i in mischSack:
  print(i)
```

Zugriff aufs n+1te Element über list[n] **index :**

```
print(mischSack[2]) # 4.4
```

Zugriff aufs **letzte Element** über list[**-1**] index :

```
print(mischSack[-1]) # 4.4
```

Listen

Vergleich

```
>>> [1,2,3]==[1,2,3]
True
>>> [1,2,3]==[1,3,2] Reihenfolge ist wichtig!
False
>>> [1,2,3]==[1,2,3.0] Darstellung kann abweichen
True
>>>
```

solange für jedes Element a_i == b_i gilt

Listen Hauptfunktionen

```
wochentage = ["Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag", "Samstag", "Sonntag"]

erweitern
wochentage + ["Feiertag"]

löschen
wochentage.remove("Feiertag")
del wochentage[-1]  # löscht letztes

abfragen
wochentage[0] # "Montag"

setzen
wochentage[0]="Monday" #jetzt Englisch!

prüfen
"Dienstag" in wochentage
```

Arbeiten mit Listen

- **__contains__** Liste enthält… ?

```
1 in [1,2,3] # True   x in xs == "x ∈ xs"
```

- __add__ Liste **erweitern**

```
>>> [1,2] + 3
TypeError: can only concatenate list (not "int") to list  … AHA:

>>> [1,2] + [3]
[1, 2, 3]
```

- len __len__ Länge

```
len(1,2,3) # 3
```

- **remove** Eintrag entfernen:

```
>>> [1,2,3]-[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'list' and 'list'

>>> x=[1,2,3]
>>> x.remove(2) # ⚠️ EIN objekt, was gelöscht werden soll, nicht der index
>>> x
[1, 3]

del x[1]  # löschen an index !
```

Arbeiten mit Listen

⚠️ **Modifizierende** funktionen (vs pure)

• **pop() letztes Element herausploppen (lesen und entfernen)**

```
>>> x = [1,2,3]
>>> y = x.pop()
3
>>> x
[1,2]
```

pop(n) zum aktiven entfernen des n+1 ten Elementes

*externer **operator del** bildet auf __delitem__ ab*
***del** a[1]*

• ***.append()** Liste erweitern (vs __add__ !)*

```
>>> x = [1,2,3]
>>> x.append(4)
>>> x
[1, 2, 3, 4]  == [1,2,3] + [4]
```

• *reverse() umdrehen*

```
>>> x = [1,2,3]
>>> x.reverse()
>>> x
[3, 2, 1]
```

dir(list) überschaubar

⚠️ Modifizierende Funktionen

⚠️ **Modifizierende** Funktionen "**Prozedur**" Methoden "**prozedural**" (vs pure)

Einige Funktionen **verändern** das Objekt, **andere** liefern das Ergebnis zurück.

```
>>> [1,2,3].sort()
>>>    KEIN ERGEBNIS!     aber ein Effekt:
>>> a=[3,2,1]
>>> a.sort()   # verändert a!!!
>>> a
[1, 2, 3]   AHA, a wurde sortiert

>>> sorted([3,2,1])
[1, 2, 3]
>>> sorted(a)  # a bleibt unverändert
```

Leider kann man den Funktionen nicht am Namen ansehen wie sie sich verhalten!!
Aber ein Hinweis ist: Kein Rückgabewert => Prozedur,
Aber ein Hinweis ist: Globale Funktionen ...ed (keine Methode) z.B. **sorted()**

Aber Rückgabewert kann trotzdem Objekt verändert haben, z.B. pop()

Arbeiten mit Listen

⚠ **Modifizierende** funktion (vs pure)
⚠ anders als bei str kann man in einer Liste elemente direkt **setzen / überschreiben**

```
>>> x = [1,2,3]
>>> x[1]=5
>>> x
[1, 5, 3]
```

allgemeine Syntax liste[index-n]=wert

dir(list) überschaubar

Slicing

💡 wie str kann man in einer Liste Elemente-Bereiche (**Teil-Listen**) auslesen

```
>>> x=["a","b","c","d"]

>>> x[:2] # first 2 elements
['a', 'b']

>>> x[2:] # after first 2 elements
['c', 'd']

>>> x[2:4] # from index 2 to index 4 exklusive ⚠️
['c', 'd']

x[2:-1] # von 2 bis zum letzten Element exklusive

allgemein
liste[start:ende]   start=0 end-index exklusive

Aufgabe jetzt: mit Listen experimentieren
gib mir ["b","c"] via slicing

default ende = len(x)
```

Fortgeschrittenes Slicing

Slicing ist eine Methode in Python, um Elemente aus Sequenzen wie **Listen**, **Tupeln** und **Strings** herauszugreifen. Fortgeschrittenes Slicing erweitert die Grundkonzepte und bietet mehr Flexibilität beim Zugriff auf Teile einer Sequenz.

```
liste[start:end:step]
```

```
liste = [1, 2, 3, 4, 5, 6, 7, 8, 9]
jedes_zweite = liste[::2] # "Schrittlänge 2"
umgekehrt = liste[::-1] # "Schrittlänge -1 : gehe rückwärts"
```

Kombination von Slicing-Parametern

- Kombination von Start-, End- und Schrittindizes zur präzisen Kontrolle über die Auswahl.
- Beispiel: Auswahl jedes dritten Elements in einem Teilstück der Liste:

```
liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
teilstueck = liste[1:8:3]
[1, 4, 7]
```

Aufgabe jetzt: mit Listen [::] slicing experimentieren
gib mir [2,4,6,8] via slicing
gib mir [2,5,8] via slicing
gib mir [8,6,4,2] via slicing # "Schrittlänge -2"

Sortieren

⚠️ *Modifizierend!*

```
>>> [1,3,2].sort()
>>>
>>> x
[1, 2, 3]
```

Später fortgeschrittene Sortierkriterien z.B.

nach Länge:

```
woerter = ["Banane", "Apfel", "Kirsche"]
woerter.sort(key=len)
print(woerter)  # ['Apfel', 'Banane', 'Kirsche']
```

Umdrehen

```
⚠️ Nicht Modifizierend!
liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> umgedreht = reversed(liste)  # bildet ab auf liste.__reversed__()
>>> list(umgedreht)
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
⚠️ Modifizierend!
>>> liste.reversed()
>>>    # kein Rückgabewert!
>>> liste
>>> [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
# liste[1,7]
```

```
PRO: "flachmachen" von verschachtelten Listen
sum kann anscheinend auch flaten:
doublelist =[[1, 2, 3], [5, 3]]
sum(doublelist,[])
```

Aufgaben für Listen

- Aufgaben für Listen

- a) Erstelle Liste mit 3 Namen, füge einen weiteren hinzu und lösche den zweiten
- b) Verdoppele alle Namen indem die Liste an sich selbst angefügt wird
- c) lösche letzten Namen und zeige ihn gleichzeitig an
- d) sortiere liste und zeige sie an
- e) drehe sie um
- Zusatzaufgaben wenn jemand vorzeitig fertig ist:
- f) prüfe ob "Bob" in der Liste vorkommt
- g) an welcher Stelle kommt "Alf" vor?
- h) lösche "Alf". Was passiert wenn er nicht existiert hat?
- i) zähle, wie oft "Ute" vorkommt
- j) was ist die Länge der finalen Liste?

Extra Aufgaben für Listen

- Erstelle eine Funktion, die zwei Listen erhält und prüft, ob in beiden die Zahl 42 vorkommt
- Erstelle eine Funktion, die prüft, ob alle Elemente einer Liste in einer anderen Liste **vorhanden** sind. (nicht notwendig an der selben Position)
- gemeinsamen Elemente zweier Listen:
  - Schreibe eine Funktion, die die Schnittmenge zweier Listen zurückgibt.
- Definiere **pure** sortier Funktion sort([2,3,1]) oder sort(3,1,2) => [1,2,3]
- Entferne Duplikate aus einer Liste (ohne Set-Verwendung, modifizierend oder per Konstruktion)
- Identifiziere das Element, das in einer Liste am häufigsten vorkommt. (später: via dict)
- Funktion Permutation / Rotieren von Elementen in einer Liste z.B. [1,2,3,4] => [2,3,4,1]
- IM KOPF: welche Elemente werden hier ausgewählt:

```
liste = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
teilstueck = liste[2:6:3]
```

Pause

- x

tuple

Tupel sind wie **unveränderliche** Listen (aber mit runden statt eckigen klammern)

```
leeres_tupel = ()
tupel = (1, 2, 3, 4)
```

Alles (bis auf Modifikationen!) was wir mit Listen können, können wir auch mit Tupeln

```
for i in tupel:
  print(i)

print(tupel[0])  # Ausgabe: 1
len(tupel)  # Ausgabe: 4
…
```

neue Tupel

Tupel (in englisch / python: **tuple**) sind wie **unveränderliche** Listen

```
leeres_tupel = ()  # leeres dict = {}  leere liste = []
tupel_mit_einem_element = (1, )
tupel = (1, 2, 3, 4)
```

Jedwelche Modifikation scheitert aber oder ist nur durch Kreation eines neuen Tupels erlaubt:

```
# tupel[0] = 5  # TypeError: 'tuple' object does not support item assignment
# tupel.append(5)  # AttributeError: 'tuple' object has no attribute 'append'

# Kreation eines neuen Tupels
neues_tupel = tupel + (5,)  # OK
tupel2 = (5, 6)
kombiniertes_tupel = tupel + tupel2  # (1, 2, 3, 4, 5, 6)
wiederholtes_tupel = tupel2 * 2  # (5, 6, 5, 6)
```

PS: anders als in anderen Sprachen hat python nativ keine benamten Tupel (x=1, y=2)
Wer struktur-sichere dictionaries möchte kann diese aber als modul importieren:

```
from collections import namedtuple
Person = namedtuple("Person", ["name", "age", "city"])
```

keine first class citizen

Aufgaben Einfach

Aufgaben Nachmittag

a) Finde letzte Ziffer einer Zahl (Tipp: modulo)

b) Setze Klammern so, dass das Ergebnis stimmt, überprüfe in Python
20 / 5 * 2 == 2
2 ** 3 ** 2 == 64
10 - 4 - 1 == 7
2 + 3 * 4 == 20

c) Aufgaben von Gestern noch mal angehen, falls noch nicht gelöst

d) print(10 - 3 - 2)
print(10 - (3 - 2))
Ist das ein Präzedenz 'Problem? (sondern... ?)

e) a == b * (a // b) + (a % b)
e1) was soll die Formel sagen?
e2) Stimmt sie? Am Beispiel?
e3) Kann man klammern weglassen?

die letzte ja, die erste: `3 * 10 // 3` vs `3 * (10 // 3)`

Aufgaben Operator Priorisierung

IM KOPF: Was ist das Ergebnis der folgenden Berechnungen in Python und in der realen Welt?

17 % 5 + 2 * 3

25 // 4 + 3 ** 2

30 // 4 % 3

2 + 18 // 4 * 3

7 + 20 % 6 // 2

50 // 6 % 4 + 1

3 * 7 % 5 + 2

100 // 9 // 2

5 + 2 ** 3 % 3

8 % 3 * 4 // 2

9 + 24 // 5 % 2

15 % 4 + 6 // 2 ** 2

Aufgaben Operator Priorisierung

```
17 % 5 + 2 * 3        # 8
17 % (5 + 2) * 3      # 9

25 // 4 + 3 ** 2      # 15
(25 // 4 + 3) ** 2    # 81

30 // 4 % 3           # 1
30 // (4 % 3)         # 30

2 + 18 // 4 * 3       # 14
(2 + 18) // 4 * 3     # 15

7 + 20 % 6 // 2       # 8
(7 + 20) % 6 // 2     # 1

50 // 6 % 4 + 1       # 1
50 // (6 % 4 + 1)     # 16

3 * 7 % 5 + 2         # 3
3 * (7 % 5 + 2)       # 12

100 // 9 // 2         # 5
100 // (9 // 2)       # 25

5 + 2 ** 3 % 3        # 7
(5 + 2) ** (3 % 3)    # 1

8 % 3 * 4 // 2        # 4
8 % (3 * 4) // 2      # 4

9 + 24 // 5 % 2       # 9
(9 + 24) // 5 % 2     # 0

15 % 4 + 6 // 2 ** 2    # 4
(15 % 4 + 6) // 2 ** 2  # 1
```

Datenaustausch als Ordner

WebDAV

https://d.alfanetz.de/remote.php/dav/files/D65263TQF

https://docs.nextcloud.com/server/stable/user_manual/en/files/access_webdav.html

Grenzen von Zahlen

Integer haben beliebig viele Ziffern
Float bis 10^310

super für Wissenschaft und Zahlentheorie

Vokabeln Tag 3

- Operator
- 'Präzedenz' ≈ Vorrang Bindung (Gruppierung Reihenfolge Ordnung) von Operatoren
- Liste
- Tupel
- Modifizierend
- Prozedur
- pur (kein Seiteneffekt)

Vokabular Tag 3

list,
dict (map)
tuple
set

Zustand (Menge aller Variablen und ihrer Werte)
Gültigkeitsbereich

>>> ctrl+l zum reinigen der Console (geht auch in System Console)

Vorschau Kontrollstrukturen

Kontrollstrukturen sind wichtige Bausteine in der Programmierung, insbesondere in Python. Sie ermöglichen es dem Programm, Entscheidungen zu treffen, Wiederholungen auszuführen und somit auf verschiedene Eingaben und Situationen flexibel zu reagieren. Hier sind die grundlegenden Kontrollstrukturen in Python:

Bedingte Ausführung "IF"

```
temperatur = 22
if temperatur > 25:
   print("Es ist warm")
```

"Wenn Temperatur größer 25 ist, sag es ist warm"

```
if Bedingung :
   Dann-führe-aus
```

Bedingte Ausführung "IF ELSE"

```
temperatur = get_sensor_data()
if temperatur > 25:
   print("Es ist warm")
else:
   print("Es ist kühl")
```

"Wenn Temperatur größer 25 ist, sag es ist warm, ANSONSTEN sag es ist kühl" **otherwise**

Vorschau Kontrollstrukturen

Kontrollstrukturen und Funktionen haben Programmcode in einem sogenannten Block

```
temperatur = 22
if temperatur > 25:
   print("Es ist warm")           # Block Befehl 1
   print("Es ist wirklich warm")  # Block Befehl 2
```

Dieser wird durch **Einrückung** von anderem Code abgegrenzt

```
temperatur = 22
if temperatur > 25:
   print("Es ist warm")           # Block Befehl 1
   print("Es ist wirklich warm")  # Block Befehl 2
print("nicht Teil des if Befehles")
```

Space oder Tab \t **Einrückung**