

B&B Kurzvorstellung + Einführung in Python als Tool fürs Studium

Bits & Bäume Zweig Dresden

2025-10-08



Bits & Bäume Zweig Dresden

Wir sind eine Hochschulgruppe aus Dresden und Teil von des Bits & Bäume Netzwerkes, welches sich für die Schnittstelle von Digitalisierung und Nachhaltigkeit einsetzt.

Bits & Bäume Zweig Dresden

Wir sind eine Hochschulgruppe aus Dresden und Teil von des Bits & Bäume Netzwerkes, welches sich für die Schnittstelle von Digitalisierung und Nachhaltigkeit einsetzt.



**Unsere Website mit Slides
und Quellen**



Mastodon



Instagram*

Was erwartet euch?

- ① Nachhaltigkeit und Digitalisierung
- ② Wer sind wir?
- ③ Freie Software

Was ist eigentlich das Problem?



- Klimakrise, Verschmutzung, Biodiversitätskrise, ...
- Ressourcen ↘, Menschenrechte ↘, Autokratie ↗

Technologie

Umweltgutachten: Digitalisierung ist heute Brandbeschleuniger für Ressourcen-Raubbau und Ungleichheit

Die Digitalisierung, wie wir sie heute kennen, untergräbt den Klimaschutz und Grundrechte. Der Umweltbeirat der Bundesregierung macht Vorschläge, das zu ändern – vor allem auf der internationalen Ebene. Aber wie glaubwürdig wäre die Bundesregierung, wenn sie bei EU und UN nachhaltige Digitalisierung fordert, aber zu Hause wenig Fortschritte macht?

11.04.2019 um 18:42 Uhr - Leon Kaiser - 7 Ergänzungen



ökologisch:

- Energieverbrauch (→Klima!)
- Rohstoffverbrauch
- Produktionsbedingungen
- Schadstoffe

öffentlicher Diskurs:

- Filterblasen
- dysfunktionale Diskurse
- Anfechtung der Menschenrechte
- Delegitimation der fdGO

Freiheit:

- Abhängigkeiten
- zielgerichtete Werbung
- Überwachung
- → eingeschränkte Selbstbestimmung

sozial:

- Suchtverhalten
- Kurzsichtigkeit
- Konzentrations- und Kommunikationstörungen
- Vereinsamung

ökologisch:

- Energieverbrauch (→Klima!)
- Rohstoffverbrauch
- Produktionsbedingungen
- Schadstoffe

öffentlicher Diskurs:

- Filterblasen
- dysfunktionale Diskurse
- Anfechtung der Menschenrechte
- Delegitimation der fdGO

Freiheit:

- Abhängigkeiten
- zielgerichtete Werbung
- Überwachung
- → eingeschränkte Selbstbestimmung

sozial:

- Suchtverhalten
- Kurzsichtigkeit
- Konzentrations- und Kommunikationstörungen
- Vereinsamung

These: *Große Teile der Gesellschaft haben die Tragweite der Digitalisierung noch nicht ansatzweise erkannt.*

Umweltbewegung und Tech-Szene: ähnliche Herausforderungen

Umweltbewegung und Tech-Szene: ähnliche Herausforderungen



Umweltbewegung und Tech-Szene: ähnliche Herausforderungen



→ bits-und-baeume.org/forderungen

Wer wir sind?

Bits & Bäume Zweig Dresden

Wir sind eine Hochschulgruppe aus Dresden und Teil von des Bits & Bäume Netzwerkes, welches sich für die Schnittstelle von Digitalisierung und Nachhaltigkeit einsetzt.

Bits & Bäume Zweig Dresden

Wir sind eine Hochschulgruppe aus Dresden und Teil von des Bits & Bäume Netzwerkes, welches sich für die Schnittstelle von Digitalisierung und Nachhaltigkeit einsetzt.



**Unsere Website mit Slides
und Quellen**



Mastodon



Matrix Chat

Vier Freiheiten freier Software

Verwenden



Verstehen



Verändern



Verbreiten



Vier Freiheiten freier Software

Verwenden



Verstehen



Verändern



Verbreiten



Vorteile:

- Kontrolle behalten
- Erkenntnisgewinn
- Lizenzkosten: 0 €
- Anpassbarkeit an eigene Bedürfnisse
- Herstellerunabhängigkeit
(kein Vendor Lock-in)

Vier Freiheiten freier Software

Verwenden



Verstehen



Verändern



Verbreiten



Vorteile:

- Kontrolle behalten
- Erkenntnisgewinn
- Lizenzkosten: 0 €
- Anpassbarkeit an eigene Bedürfnisse
- Herstellerunabhängigkeit
(kein Vendor Lock-in)

Nachteile Proprietärer Software

- Intransparenz (Bsp: Wahlsoftware)
- Hintertüren? (Win10-Verbot für Dienstgebrauch)
- Abhängigkeit

Vier Freiheiten freier Software

Verwenden



Verstehen



Verändern



Verbreiten



Vorteile:

- Kontrolle behalten
- Erkenntnisgewinn
- Lizenzkosten: 0 €
- Anpassbarkeit an eigene Bedürfnisse
- Herstellerunabhängigkeit
(kein Vendor Lock-in)

Nachteile Proprietärer Software

- Intransparenz (Bsp: Wahlsoftware)
- Hintertüren? (Win10-Verbot für Dienstgebrauch)
- Abhängigkeit

- Datensparsamkeit
- Moderate Hardware-Anforderungen
- Sicherheit

- Transparenz
- Wahlfreiheit
- Souveränität

- Datensparsamkeit
- Moderate Hardware-Anforderungen
- Sicherheit

- Transparenz
- Wahlfreiheit
- Souveränität

→ Freie Software ist **nachhaltig!**

Einführung: Python als Tool fürs Studium



- Postdoc an der [Professur für Grundlagen der Elektrotechnik, TU Dresden](#)
- Forschung: Erklärbare künstliche Intelligenz, Formale Wissensrepräsentation, Regelungstechnik

- Zivilgesellschaftliches Engagement:
 - [Bits und Bäume Dresden](#)
 - [Hochschulgruppe für Freie Software und Freies Wissen](#)
 - [Konstruktive Digitale Diskussionskultur](#)

- Erste Erfahrungen mit Python seit 2004, aktive Nutzung seit 2008
- Seit 2010: <https://tu-dresden.de/pythonkurs>





<https://python-fuer-ingenieure.de>

Grundlagen

Warum Python? (1)

Python als Programmiersprache

- Klare, lesbare Syntax (wenig „Ballast“)
- Objektorientiert, prozedural, funktional programmierbar
- Nützliche eingebaute Datentypen (`list`, `tuple`, `dict`, `set`, ...)
- Einfache Modularisierung (`import this`)
- Gute Fehlerverwaltung (Exceptions)
- Umfangreiche Standardbibliothek
- Einfache Einbindung von externem Code (C, C++, Fortran)

Warum Python? (1)

Python als Programmiersprache

- Klare, lesbare Syntax (wenig „Ballast“)
- Objektorientiert, prozedural, funktional programmierbar
- Nützliche eingebaute Datentypen (`list`, `tuple`, `dict`, `set`, ...)
- Einfache Modularisierung (`import this`)
- Gute Fehlerverwaltung (Exceptions)
- Umfangreiche Standardbibliothek
- Einfache Einbindung von externem Code (C, C++, Fortran)



- Leicht zu lernen
- Problemorientiert (mächtig und flexibel)
- Motivationspotenzial ↗, Frustrationspotenzial ↘

Außerdem: Plattformübergreifend / frei und quelloffen / große u. aktive Community

Python als Werkzeug für Ingenieur:innen:

- Symbolisches Rechnen (Ableiten, Integrieren, Gl. lösen, ...)
- Numerisches Rechnen (lin. Algebra, DGLn, Optimierung, ...)
- Visualisieren (2D, 3D, in Publikationsqualität)
- Analyse tabellarischer Daten

- Kommunikation mit externen Geräten (RS232, GPIB, ...) ⇒ Laborautomatisierung
- Maschinelles Lernen

Python als Werkzeug für Ingenieur:innen:

- Symbolisches Rechnen (Ableiten, Integrieren, Gl. lösen, ...)
- Numerisches Rechnen (lin. Algebra, DGLn, Optimierung, ...)
- Visualisieren (2D, 3D, in Publikationsqualität)
- Analyse tabellarischer Daten

- Kommunikation mit externen Geräten (RS232, GPIB, ...) ⇒ Laborautomatisierung
- Maschinelles Lernen

⇒ Gestärkte „Forschungskompetenz“



Befehlsmodus (Esc zum Aktivieren)

- Shift-Return - Zelle ausführen, nächste aktivieren
- h - zeige Tastatur-Befehle
- m - Zellentyp ändern auf „markdown“
- y - Zellentyp ändern auf „code“
- a - neue Zelle oberhalb
- b - neue Zelle unterhalb

Bearbeitungs-Modus (Enter zum Aktivieren)

- Shift-Return - Zelle ausführen, nächste aktivieren
- Tab - Autovervollständigung oder Einrückung
- Shift-Tab - Einrückung entfernen
- Ctrl-Z - Rückgängig

Listing: hello-world.py

```
import math
print("Hello World")
a = 10
b = 20.5
c = a + b + 3**2
print(math.sqrt(c))

while True: # start infinite loop
    x = input("Your name? ") # returns a str-object
    if x == "q":
        break # finish loop
    print("Hello ", x)
```

Listing: hello-world.py

```
import math
print("Hello World")
a = 10
b = 20.5
c = a + b + 3**2
print(math.sqrt(c))

while True: ## start infinite loop
    x = input("Your name? ") ## returns a str-object
    if x == "q":
        break ## finish loop
    print("Hello ", x)
```

- Einrückungen haben syntaktische Bedeutung!
- DeFacto-Standard: 4 Leerzeichen
(in Editoren: Blockweise mit <TAB>(→) und <SHIFT+TAB> (←))

- Integer

```
>>> type(1)
<type 'int'>
```

- Gleitkommazahlen

```
>>> type(1.0)
<type 'float'>
```

- Komplexe Zahlen

```
>>> type(1 + 2j)
<type 'complex'>
```

- Operatoren:

Addition	+
Subtraktion	-
Division	/
Ganzzahldivision	//
Multiplikation	*
Potenzieren	**
Modulo	%

- Built-in Funktionen

- `round`, `pow`, etc.
- siehe `dir(__builtins__)`

- Modul `math`

- siehe `help(math)`

NoneType und Boolesche Werte

- None

- Universeller False-Wert

```
>>> type(None)
<type 'NoneType'>
```

- Boolesche Werte

- True und False

```
>>> type(True)
<type 'bool'>
```

Datentyp	False-Wert
NoneType	None
int	0
float	0.0
complex	0 + 0j
str	""
list	[]
tuple	()
dict	{}
set	set()

Operation	Abkürzung
<code>x = x + y</code>	<code>x += y</code>
<code>x = x - y</code>	<code>x -= y</code>
<code>x = x * y</code>	<code>x *= y</code>
<code>x = x / y</code>	<code>x /= y</code>
<code>x = x % y</code>	<code>x %= y</code>
<code>x = +x</code>	
<code>x = -x</code>	
<code>x = x ** y</code>	<code>x **= y</code>
<code>x = x // y</code>	<code>x //= y</code>

Vergleichsoperation
<code>x == y</code>
<code>x != y</code>
<code>x < y</code>
<code>x <= y</code>
<code>x > y</code>
<code>x >= y</code>

```
str1 = "abc"
str2 = 'xyzabcefg'hi'
str3 = """
    abc
    """
str4 = ("abc"
        "def")
>>> str2[0] # 0 ist erster Index
'x'
>>> str2[1:4]
'yz'abc'
>>> str2[-3:]
'ghi'
```

Escape-Sequenz	Erklärung
<code>\n</code>	Zeilenumbruch
<code>\r</code>	Carriage Return
<code>\t</code>	horizontaler Tab
<code>\"</code>	Escaping "
<code>\'</code>	Escaping '
<code>\\</code>	Escaping \

Stringformatierung (1)

- General Syntax: `"value of x={} and y={}".format(x, y)`

- Examples:

```
>>> a = 'H'  
>>> b = 'ello World'  
>>> "{0}{1}{2} {0}".format(a, b, 5)  
'Hello World5 H'
```

- Komplexere Beispiele (siehe auch: docs.python.org/3/library/string.html#format-string-syntax)

```
>>> "a={:06.2f} and b={:05.2f}".format(3.007, 42.1)  
'a=003.01 and b=42.10'
```

- Wichtige Methoden der Klasse `str`:

`index`, `replace`, `split`, `join`, `format`, `startswith`, `endswith`, ...

- Neuere Syntax (seit Python3.8): „f-Strings“
→ Nutzung von Python-Ausdrücken *innerhalb* eines Strings

```
a = "World"
f"Hello {a}" # use variables
f"value of x={x} and y={y}"

f"the sum is {x + y}" # make calculations
f"the result is {call_func(x, y, 'z')}" # call functions

f"Use {{double braces}} to render braces literals! {a}"
```

- Mehr Infos: <https://docs.python.org/3/tutorial/inputoutput.html>

- Syntax
`[Wert_1, ..., Wert_n]`
- Kann verändert werden
- Kann Elemente beliebigen Typs enthalten
- Kann sortiert werden
- Wichtige Methoden:
`append`, `count`, `index`, `insert`,
`pop`, `remove`, `reverse`, `sort`
- `sort` und `reverse` arbeiten „in place“
(Rückgabewert: `None`)

- Beispiele

```
>>> m = [7, 8, 9]
>>> n = ['a', 'z', 1, False]
>>> m.append('x')
>>> m[0]
7
>>> m[-1]
'x'
>>> m[:] # Anf. bis Ende
[7, 8, 9, 'x']
>>> m.pop(0)
7
>>> m.reverse()
>>> print(m)
['x', 9, 8]
```

- Syntax
`(Wert_1, ..., Wert_n)`
- Kann **nicht** verändert werden
- → Zugriff schneller als auf Listen
- Kann Elemente beliebigen Typs enthalten
- Wichtige Methoden
`index`

- Beispiele

```
>>> t = (7,8,9)
>>> t[0]
7
>>> t[-1]
9
>>> t[:] # Anf. bis Ende
(7,8,9)
>>> z = ('a', 'z', 1, False)
>>> t.index(8)
1
>>> z.index('a')
0
```

Betrifft Liste, Tupel, Strings, (Numpy-Arrays), ...

Operation	Erklärung
-----------	-----------

<code>s in x</code>	prüft, ob s Element von x ist
---------------------	-------------------------------

<code>s not in x</code>	prüft, ob s kein Element von x ist
-------------------------	------------------------------------

<code>x + y</code>	Verkettung von x und y
--------------------	------------------------

<code>x * n</code>	Verkettung, sodass n Kopien von x existieren (nicht für <code>numpy.array</code>)
--------------------	--

<code>x[n]</code>	gibt von x das Element zum Index n zurück
-------------------	---

<code>x[-1]</code>	letztes Element; (negativer Index zählt von hinten)
--------------------	---

<code>x[n:m]</code>	gibt die Teilsequenz von n bis (m-1) zurück
---------------------	---

<code>x[n:m:k]</code>	gibt die Teilsequenz von n bis (m-1) zurück, bei der nur jedes k-te Element berücksichtigt wurde
-----------------------	--

<code>len(x)</code>	gibt die Anzahl von Elementen zurück
---------------------	--------------------------------------

<code>min(x)</code>	gibt das kleinste Element zurück
---------------------	----------------------------------

<code>max(x)</code>	gibt das größte Element zurück
---------------------	--------------------------------

Assoziative Arrays (Dictionaries)

- Syntax

```
{Key_1: Value_1, Key_2: Value_2, ... }
```

- Schlüssel-Wert-Paare

- Schlüssel müssen unveränderlich sein („hashable“)

- Zugriff auf Werte via

- `d[key]` oder
- `d.get(key, default)`

- Wichtige Methoden:

- `keys`, `values`, `items`

- Beispiele

```
>>> d = { "Sachsen" : "Dresden",  
... "Brandenburg" : "Potsdam"}  
>>> e = {1: 'a', 2: 'b', 3: 'c'}  
>>> e[1]  
'a'  
>>> d.get("Sachsen")  
'Dresden'  
# kein Eintrag -> keine Ausgabe  
>>> d.get("Bayern") # -> None  
>>> d["Frankreich"]  
KeyError: 'Frankreich'  
>>> type(d)  
<type 'dict'>
```

- Syntax

```
set([Element_1, ..., Element_n])
```

- Kann jedes Element nur einmal enthalten
- Kann nicht sortiert werden
- Kann verändert werden
- `frozenset` ist unveränderlich

- Beispiele

```
>>> engineers = set(['John', 'Jane',  
... 'Jack', 'Janice'])  
>>> programmers = set(['Jack', 'Sam',  
... 'Susan', 'Janice'])  
>>> managers = set(['Jane', 'Jack',  
... 'Susan', 'Zack'])  
>>> union = engineers | programmers  
>>> intersect = engineers & managers  
>>> difference = managers - engineers  
>>> engineers.add('Marvin')  
>>> print engineers  
set(['Jane', 'Marvin',  
'Janice', 'John', 'Jack'])
```

- Syntax

```
if <Bedingung>:  
...  
elif <Bedingung>:  
...  
else:  
...
```

- Abkürzung

```
y = (1 if x == 'a' else 2)
```

- Beispiele

```
>>> x = 1  
>>> if x == 1:  
...     print("x=1")  
...  
x=1  
>>> x = 4  
>>> if x == 1:  
...     print("x=1")  
... elif x == 3:  
...     print("x=3")  
... else:  
...     print("x !=1 und x != 3")  
x !=1 und x != 3
```

Fallunterscheidung (2) (Structural Pattern Matching)

`match` -Schlüsselwort: eingeführt in Python 3.10

Beispiel:

```
def http_error(status):  
    match status:  
        case 400:  
            return "Bad request"  
        case 401 | 403 | 404:  
            return "Not found or not allowed!"  
        case _:  
            # This always matches  
            return "Something's wrong with the Internet"
```

Mehr Information: <https://peps.python.org/pep-0636/>

- Syntax

```
while <Bedingung>:  
    ...
```

- `break`

bricht komplette Schleife ab

```
while <Bedingung1>:  
    if <Bedingung2>:  
        break
```

- `continue`

beginnt nächsten Schleifendurchlauf

```
while <Bedingung1>:  
    if <Bedingung2>:  
        continue
```

- Beispiele

```
>>> x = 4  
>>> while x > 1:  
...     print(x)  
...     x -= 1  
...     print(f"{{x=}}")  
4  
3  
2  
x=1
```

for-Schleife (und range-Funktion)

- Syntax:

```
for <Variable> in <sequenz>:  
    ...
```

- Oft dabei benutzt: `range`-Funktion

- Gibt *Iterator*-Objekt zurück
- Syntax: `range(start, stop, step)`
- start, step sind optional
- stop ist exklusiv
- Beispiele:

```
>>> list(range(4)) # nur stop-Wert (exkl.)  
[0, 1, 2, 3]  
>>> list(range(2, 4)) # start und stop-Wert  
[2, 3]  
>>> list(range(1, 10, 2)) # start, stop, step  
[1, 3, 5, 7, 9]
```

- Beispiele:

```
>>> x = ['a', 'b', 'c']  
>>> count = 0  
>>> for a in x:  
...     print(count, a)  
...     count += 1  
0 a  
1 b  
2 c  
>>> for i in range(3):  
...     print(f"{i=}, {i**2=}")  
i=0, i**2=0  
i=1, i**2=1  
i=2, i**2=4  
>>> for i in range(5):  
...     if i > 8:  
...         break  
...     else:  
...         # schleife ohne break  
...         # beendet  
...     print(i)
```

- Syntax

```
def funcName(Param_1, ..., Param_n):  
    return <Resultat>
```

- optionale Parameter

```
def test(param = 'Hallo'):  
    print(param)
```

- Mehr zu Funktionen: später

- Beispiele

```
>>> def printSum(a, b):  
...     print(a+b)  
>>> printSum(1, 2)  
3  
>>> def printMult(a, b, c, d=0):  
...     print((a*b*c)+d)  
>>> printMult(2, 4, 3)  
24  
>>> printMult(a=3, c=4, b=3, d=3)  
39
```

- Anweisungsblöcke durch Einrückung definiert (an Stelle von z.B. { ... })
 - Leerzeichen anstatt Tabulatoren nutzen!
 - ggf. eigenen Texteditor entsprechend einstellen
(in Spyder: `TAB` zum Einrücken, `SHIFT+ TAB` zum Ausrücken)

- Kommentare:

```
# einzeilige Kommentare beginnen mit Raute (Hash-Symbol)  
""" mehrzeilige Strings lassen sich mit 3-fach-Quotes erzeugen.  
Sie können als Docstrings oder Mehrzeilige Kommentare dienen.  
Oder als ganz normale Strings die Zeilenumbrüche enthalten."""
```

- Empfohlene maximale Zeilenlänge: 100 Zeichen (siehe PEP8)
- Wenn eine Anweisung mehr benötigt:
 - Mit `\` lässt sich die Anweisung auf der nächsten Zeile fortsetzen
 - Bei Klammern erfolgt dies automatisch bis zur schließenden Klammer
 - Über Aufteilung auf mehrere Anweisungen nachdenken

- Indizierung beginnt immer bei 0 (anders als z.B. bei Matlab)
- Negative Indizes zählen von hinten
- Entpacken sequentieller Datentypen:

```
>>> x, y, z = range(3) # list(range(3)) -> [0, 1, 2]
>>> y
1
```

- `import`-Anweisungen werden nur einmal ausgeführt
- Neuladen ggf. mit `importlib.reload(mymodule)` erzwingen
- ∃ Umfangreiche Standardbibliothek („batteries included“)
 - <http://docs.python.org/3/library/>
 - Bevor man selber etwas programmiert, nachschauen ob Funktionalität nicht bereits in Standardbibliothek (oder als 3rd-Party-Paket) vorhanden ist

Schlüsselwort `open` (Details siehe docs.python.org/3/library/functions.html#open)

```
# Schreiben
fp = open('datei.txt', 'w') # fp = file pointer
fp.write('Hallo Datei.')
fp.writelines(['hallo\n', 'Datei\n'])
fp.write(str(5))
fp.close()

# Lesen
fp = open('datei.txt', 'r')
header = fp.read(10) # die ersten 10 Byte
lines = fp.readlines() # Liste der Zeilen ab Datei-Cursor
fp.close()
```

Schlüsselwort `open` (Details siehe docs.python.org/3/library/functions.html#open)

```
# Schreiben
fp = open('datei.txt', 'w') # fp = file pointer
fp.write('Hallo Datei.')
fp.writelines(['hallo\n', 'Datei\n'])
fp.write(str(5))
fp.close()

# Lesen
fp = open('datei.txt', 'r')
header = fp.read(10) # die ersten 10 Byte
lines = fp.readlines() # Liste der Zeilen ab Datei-Cursor
fp.close()
```

Binärdateien lesen/schreiben:

```
open('xy.pdf', 'rb') oder open('xy.pdf', 'wb')
```

Schlüsselwort `open` (Details siehe docs.python.org/3/library/functions.html#open)

```
# Schreiben
fp = open('datei.txt', 'w') # fp = file pointer
fp.write('Hallo Datei.')
fp.writelines(['hallo\n', 'Datei\n'])
fp.write(str(5))
fp.close()

# Lesen
fp = open('datei.txt', 'r')
header = fp.read(10) # die ersten 10 Byte
lines = fp.readlines() # Liste der Zeilen ab Datei-Cursor
fp.close()
```

Binärdateien lesen/schreiben:

```
open('xy.pdf', 'rb') oder open('xy.pdf', 'wb')
```

Empfehlenswert: [Context-Manager](#)-Syntax (mit dem Schlüsselwort `with`):

```
with open('datei.txt') as fp:
    lines = fp.readlines()
# fp.close() wird automatisch am Block-Ende aufgerufen
```

Funktionen in Python (1)

- Wiederkehrende Teilaufgaben kapseln

→ Komplexität beherrschen

Häufige Fehler:

- Doppelpunkt vergessen (Syntax-Fehler)
- return vergessen (→ None wird zurückgegeben)

```
def tutnix():  
    pass # dummy-statement (notwendig für Einrückung)
```

Aufruf einer Funktion ohne Argumente

```
print(tutnix()) # -> None
```

```
def quadrat1(z):  
    """  
    Quadrieren einer Zahl  
    """ # Docstring (=eingebaute Doku)  
    return z**2
```

```
quadrat1(7) # -> 49
```

```
quadrat1(7-12) # -> 25
```

Funktionen (2) – Globale und lokale Variablen

```
def quadrat2(z):
    x = z**2 # lokale Variable wegen Schreibzugriff
    print(x)
    return x

x, a = 5, 3 # "Entpacken" des Tupels (5, 3)
quadrat2(a) # -> 9
quadrat2(x) # -> 25
print(x) # -> 5

def quadrat3(z):
    print(x) # jetzt globales x (nur lesend)
    return z**2

def quadrat4(z):
    print(x) # Fehler (lokale Variable noch nicht bekannt)
    x = z**2 # lokale Variable (Schreibzugriff)
    return x
```

☐ Schlüsselworte `global` und `nonlocal` (Erklärung: stackoverflow.com/a/1261961/333403; Benutzung i. A. nicht zu empfehlen)

Funktionen (3) – Default-Argumente

```
def quadrat5(z=8):  
    return z**2
```

```
quadrat5() # -> 64
```

```
quadrat5(2.5) # -> 6.25
```

```
def quadrat_summe(a, b=0):  
    return a**2 + b
```

```
quadrat_summe(10) # -> 100
```

```
quadrat_summe(10, 3) # -> 103
```

```
# Explizite Benennung der Argumente ("keyword args")
```

```
quadrat_summe(b=-3, a=10) # -> 97 (hier: Reihenfolge egal)
```

```
# Hilfreich bei Funktionen mit vielen Argumenten
```

Beliebig viele Argumente (*args, **kwargs)

```
# Positionale Argumente
def mysum(*args): # args ist ein Tupel
    s = 0
    for x in args: s += x
    return s
mysum(5, 20, 3) # -> 28

numbers = [7, 4, -3, 15]
mysum(*numbers) # -> 23 # Entpacken der Liste

# Schlüsselwortargumente (keyword args)
def satz(**kwargs):
    print( type(kwargs) ) # -> dict
    for key, value in kwargs.items():
        print( f"Das {key} ist {value}." )

satz(Wasser='kalt') # -> Das Wasser ist kalt.
satz(u=8.2) # -> Das u ist 8.2.

def komplexe_funktion(a, b, c, *args, **kwargs):
    ... # Kombination möglich
```

Beliebig viele Argumente (*args, **kwargs)

```
# Positionale Argumente
def mysum(*args): # args ist ein Tupel
    s = 0
    for x in args: s += x
    return s
mysum(5, 20, 3) # -> 28

numbers = [7, 4, -3, 15]
mysum(*numbers) # -> 23 # Entpacken der Liste

# Schlüsselwortargumente (keyword args)
def satz(**kwargs):
    print( type(kwargs) ) # -> dict
    for key, value in kwargs.items():
        print( f"Das {key} ist {value}." )

satz(Wasser='kalt') # -> Das Wasser ist kalt.
satz(u=8.2) # -> Das u ist 8.2.

def komplexe_funktion(a, b, c, *args, **kwargs):
    ... # Kombination möglich
```

Funktionen als Objekt, scoping

```
def plus(a, b): return a + b

print( type(plus) ) # -> function
z = plus # Zuweisung (keine Auswertung!)
print( type(z) ) # -> function
z == plus # -> True
z is plus # -> True
z(2, 2) # -> 4

# Fabrik-Funktionen ("closures")
def fabrik(q):
    b = q          # b ist hier lokal
    def add(a):
        return b + a # b ist hier "global"
    return add # Funktionsobjekt wird zurückgegeben

# add is hier unbekannt
p1 = fabrik(7)
p2 = fabrik(1.5)
p1(3) # -> 10
p2(3) # -> 4.5
fabrik("Die Sonne ")("scheint.") # Auswertung ohne Zuweisung
```

Funktionen als Objekt, scoping

```
def plus(a, b): return a + b

print( type(plus) ) # -> function
z = plus # Zuweisung (keine Auswertung!)
print( type(z) ) # -> function
z == plus # -> True
z is plus # -> True
z(2, 2) # -> 4

# Fabrik-Funktionen ("closures")
def fabrik(q):
    b = q          # b ist hier lokal
    def add(a):
        return b + a # b ist hier "global"
    return add # Funktionsobjekt wird zurückgegeben

# add is hier unbekannt
p1 = fabrik(7)
p2 = fabrik(1.5)
p1(3) # -> 10
p2(3) # -> 4.5
fabrik("Die Sonne ")("scheint.") # Auswertung ohne Zuweisung
```

Argumenten- und Typ-Prüfung

- Funktionsaufrufe sehr flexibel
- Häufige Fehlerquelle: falsche Argumente (Werte bzw. Typen)
- Typ-Prüfung oft sinnvoll (Aufwand-Nutzen-Abwägung)

```
def eine_funktion(satz, ganzzahl, zahl2, liste):  
  
    if not type(satz) == str:          # eher ungünstig  
        print("Datentpyfehler: satz")  
        return -1  
  
    if not isinstance(ganzzahl, int):  
        print("Datentpyfehler: ganzzahl")  
        return -2  
  
    if not isinstance(zahl2, (int, float)):  
        print("Datentpyfehler: zahl2")  
        return -3  
  
    # kompakt in einer Zeile Variante (empfohlen):  
    assert isinstance(zahl2, int) and zahl2 > 0  
    # -> Assertion-Error bei Nichterfüllung  
    ...
```

- Doku und Code in gleicher Datei und (fast) gleichzeitig schreiben
- Nutzung sehr empfehlenswert
- Docstrings (aus dem Programm heraus verfügbare Doku, → `?` in IPython u. Notebooks)
- Ermöglicht automatische Erstellung übersichtlicher HTML-Doku mit [Sphinx](#)

```
def calculate(x, k, mode='normal'):  
    """  
    Short description: A functions that does something.  
  
    :param x:      main argument (float)  
    :param k:      order of approximation (int)  
    :param mode:  mode of the algorithm (str)  
                  (e.g. 'normal' or 'reverse')  
  
    :rtype:       result of complicated formula (float)  
  
    After the description of the parameters, typically  
    there is more information on the documented function.  
    """  
  
    y = x**k*some_other_function(x, mode)
```

Und Jetzt?

- Ende der Grundlagen-Folien

- Ende der Grundlagen-Folien
- Jetzt: interessanter Teil (Demo)
 - numpy, scipy → numerisches Rechnen
 - sympy → symbolisches Rechnen
 - matplotlib → Visualisierung

- Ende der Grundlagen-Folien
- Jetzt: interessanter Teil (Demo)
 - numpy, scipy → numerisches Rechnen
 - sympy → symbolisches Rechnen
 - matplotlib → Visualisierung

Empfehlung:

- Python + Pakete auf eigenem Rechner installieren z.B. mittels [Paketmanager uv](#)

- Ende der Grundlagen-Folien
- Jetzt: interessanter Teil (Demo)
 - numpy, scipy → numerisches Rechnen
 - sympy → symbolisches Rechnen
 - matplotlib → Visualisierung

Empfehlung:

- Python + Pakete auf eigenem Rechner installieren z.B. mittels [Paketmanager uv](#)

Niedrigschwellige Lösungen (webbasiert)

- [https://mybinder.org/...](https://mybinder.org/)
- <https://colab.research.google.com/> (!!)

- Ende der Grundlagen-Folien
- Jetzt: interessanter Teil (Demo)
 - numpy, scipy → numerisches Rechnen
 - sympy → symbolisches Rechnen
 - matplotlib → Visualisierung

Empfehlung:

- Python + Pakete auf eigenem Rechner installieren z.B. mittels [Paketmanager uv](#)

Niedrigschwellige Lösungen (webbasiert)

- [https://mybinder.org/...](https://mybinder.org/)
- <https://colab.research.google.com/> (!!)

→ Mehr Material unter <https://tu-dresden.de/pythonkurs>

Fragen gerne an Carsten.Knoll@tu-dresden.de oder im B&B-Chat

- Nachhaltigkeit und Digitalisierung sind existenzielle Herausforderungen
 - Nachhaltigkeit und Digitalisierung müssen zusammen gedacht werden
 - Freie Software ist ein Schritt in die richtige Richtung

 - An der TUD gibt es Gruppen und Menschen, die sich engagieren
- **Neue Leute sind immer hochwillkommen!** (+ persönliche Vernetzung ist nützlich)
- Python ist Freie Software und ein hilfreiches Werkzeug fürs Studium

