

Objektorientierung und Funktionale Programmierung

Einführung

Kristof Van Laerhoven
kvl@eti.uni-siegen.de

Michael Moeller
Michael.moeller@uni-siegen.de

uni-siegen.de



Please write a python function that gets a number n of people as an input and returns the probability of at least two people having birthday on the same day of the year.

This is the classic **birthday paradox** problem. The easiest way to compute the probability is:

Probability(at least one shared birthday)
= 1 – Probability(all birthdays are different)

Here's a clean Python function for that:

</> Python



Run

```
def birthday_paradox_probability(n):  
    """  
    Returns the probability that at least two people  
    in a group of n share the same birthday.  
    Assumes 365 days (no leap years).  
    """  
    if n > 365:  
        return 1.0 # Pigeonhole principle  
  
    prob_all_different = 1.0  
  
    for i in range(n):  
        prob_all_different *= (365 - i) / 365  
  
    return 1 - prob_all_different
```

Generated with chatGPT

Why learn python in 2026?

Write a python function that simulates the following game: Each of two players gets a deck of cards with values from 1 to 11. Moreover, there is a deck of score cards, also ranging from 1 to 11. In each round, one of the score cards is revealed. The players then decide which of their cards they want to use to try to win the score card. After both players have chosen a card, the player that played the higher card gets as many points as the score card has. If both players played the same card, no one gets points. In the simulation both players just follow a pre-implemented strategy: One player always plays a random card, the other player always plays the same card as the score card. In the end, the python function prints the final result.

```
import random
```

```
def simulate_game():  
    # Initial decks  
    player1_cards = list(range(1, 12)) # 1 to 11  
    player2_cards = list(range(1, 12))  
    score_cards = list(range(1, 12))  
  
    # Shuffle score cards  
    random.shuffle(score_cards)  
  
    # Scores  
    p1_score = 0  
    p2_score = 0  
  
    # Play rounds  
    for score_card in score_cards:  
        # Player 1: random card  
        p1_card = random.choice(player1_cards)  
        player1_cards.remove(p1_card)  
  
        # Player 2: plays same as score card (always possible here)  
        p2_card = score_card  
        player2_cards.remove(p2_card)  
  
        # Determine winner  
        if p1_card > p2_card:  
            p1_score += score_card  
        elif p2_card > p1_card:  
            p2_score += score_card  
        # else: tie → no points  
  
    # Optional: print round info  
    print(f"Score card: {score_card} | P1: {p1_card} vs P2: {p2_card}")
```

2026?

```
Score card: 5 | P1: 10 vs P2: 5  
Score card: 10 | P1: 8 vs P2: 10  
Score card: 9 | P1: 9 vs P2: 9  
Score card: 7 | P1: 7 vs P2: 7  
Score card: 8 | P1: 4 vs P2: 8  
Score card: 1 | P1: 3 vs P2: 1  
Score card: 2 | P1: 1 vs P2: 2  
Score card: 3 | P1: 5 vs P2: 3  
Score card: 6 | P1: 2 vs P2: 6  
Score card: 4 | P1: 11 vs P2: 4  
Score card: 11 | P1: 6 vs P2: 11
```

Final Result:

Player 1 (random): 13

Player 2 (match score): 37

Player 2 wins!

PS C:\Users\Michael> █

Why learn python in 2026?

Can you implement this in a functional way, i.e., only using pure functions except one interface where all documentation (printing) happens?



Sure! Here's the same game implemented using a functional programming style in Python:



python

```
import random
from functools import reduce

def card_game():
    # Initialize the decks of cards and score cards
    player1_deck = list(range(1, 12))
    player2_deck = list(range(1, 12))
    score_deck = list(range(1, 12))

    # Shuffle the score deck
    random.shuffle(score_deck)

    # Play each round
    player1_score, player2_score = reduce(lambda score, card:
        (score[0] + card if score[1] < score[0] else score, card),
        score_deck, (0, 0))

    # Print the final result
    print("Player 1 score: ", player1_score)
    print("Player 2 score: ",
```

Notice that this implementation does not have any loops, variables are not re-assigned, and mutable data structures are avoided. This makes the code more functional in style, and it can be easier to reason about and test.

But it has mutable data structures...
It also contains re-assignments...

```
File "c:\users\michael\sciebo2\lectures\lectures
sose22\ofp\code\v01\chat_gpt_functional_cardgame.py",
line 21, in play_round
```

```
    player1_deck, player2_deck = scores[2:]
```

```
ValueError: not enough values to unpack (expected 2,
got 0)
```

Regenerate response

Yes — we can structure this in a **purely functional style** by:

- avoiding mutation (no in-place changes to lists)
- returning new state instead of modifying existing state
- isolating **all side effects (printing)** in one single function

Here's a clean implementation:

 Python



 Run

```
import random
from typing import List, Tuple

# ----- Pure helper functions -----

def draw_random_card(cards: List[int]) -> Tuple[int, List[int]]:
    card = random.choice(cards)
    new_cards = [c for c in cards if c != card]
    return card, new_cards
```

Strictly speaking, `draw_random_card` is not a pure function. It depends on the state of the current random number generator. Please avoid this.

You're absolutely right — using `random.choice` makes the function impure because it depends on hidden global state.

[...]

✓ Fully pure functional implementation

Python



Run

Why are we drawing a number between 0 and 100 in the line
"random_indices = [random.randint(0, 100) for _ in range(11)]"?

Good catch — that choice is actually arbitrary and not ideal.

✗ What's wrong with `randint(0, 100)`?

- The number **100** has no meaning in the context of the game
- We later "fix" it with:

`</>` Python



▶ Run

```
idx = rand_index % len(p1_cards)
```

- This introduces a **modulo bias** (some indices become slightly more likely than others)

In 2026

is it an alternative to make the hidden state of the random number generator an input to our function? This would make it pure again.

Yes — that's exactly the *principled* way to do it, and it's a classic functional programming pattern.

[...]

Why this is powerful

-  Fully pure: same seed → same result
-  No hidden state
-  Reproducible simulations
-  Easy to test
-  Matches how FP languages (like Haskell) model randomness

Comparison of approaches

Approach	Purity	Practicality	Elegance	
Global <code>random</code>				
Precomputed random list				
Explicit RNG state (this)				

If this solution is more practical and more elegant, why didn't you suggest it in the first place?

[...]

✓ When I *would* suggest it first

I would have led with the RNG-state approach if you had said something like:

- “make it **referentially transparent**”
- “no hidden dependencies at all”
- “model randomness functionally”
- “like in Haskell”

- LLMs können äußerst hilfreich sein und unser Programmieren stark beschleunigen.
- Wenn ich programmiere, nutze ich ständig LLMs.
- Um Probleme zu erkennen, musst du jedoch selbst programmieren können!
- Der beste Weg, Programmieren zu lernen, ist zunächst ohne das Generieren von Lösungen durch LLMs.
- Unser Übungstool bietet LLM-Unterstützung, aber keine LLM-Lösungen.

- Ich nutze <https://github.com/features/copilot>
- Für Studierende ist es kostenlos, später aber nicht mehr
- Es gibt viele Alternativen (bzgl. der Qualität kenne ich mich nicht aus)
- Mich macht es beim Coden deutlich schneller
- **Am Anfang bin ich dagegen, dass Sie diese Tools nutzen! Die Klausur wird ohne geschrieben!**

<https://www.python.org/>

Python

- ist eine universelle Programmiersprache
- existiert seit 1980
- hat in den letzten Jahren extrem an Popularität gewonnen

Vorteile

- Kostenlos, flexibel und leicht zu lernen
- Läuft auf jedem Betriebssystem
- Viele leicht zu importierende Bibliotheken erweitern die Funktionalität der Basissprache
- Ausgezeichnete Funktionalitäten im Bereich Data Science / Machine Learning

Einfachste Installation:

<https://www.python.org/downloads/>

Dann den Anweisungen folgen.

⚠️ macOS Besonderheit

- macOS hat oft schon ein System-Python vorinstalliert
- Empfehlung:
 - nutze trotzdem `python3`
 - oder installiere über `Homebrew` für eine saubere Version

🐧 Linux Besonderheit

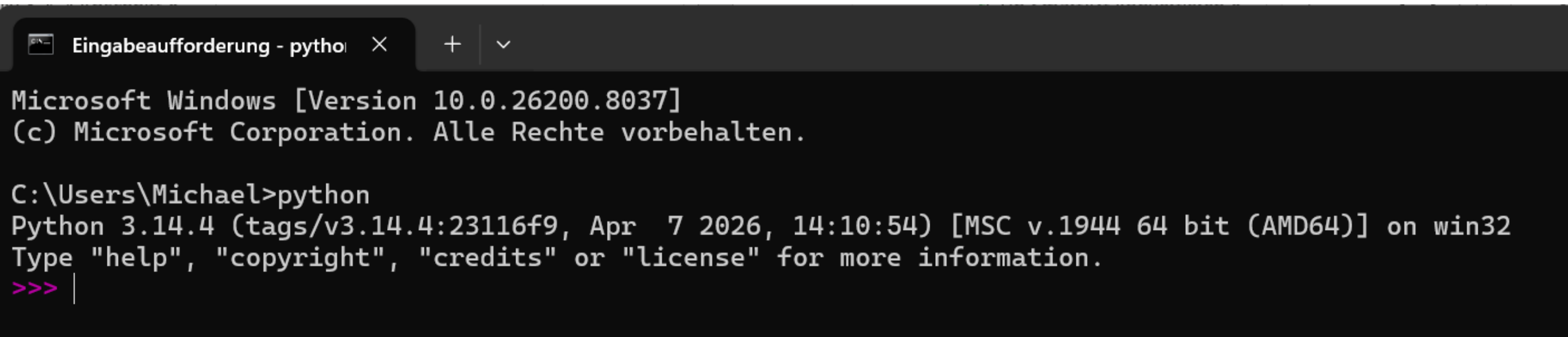
- Auf fast allen Distributionen ist `python3` standardmäßig da
- Falls nicht:

```
</> Bash
```

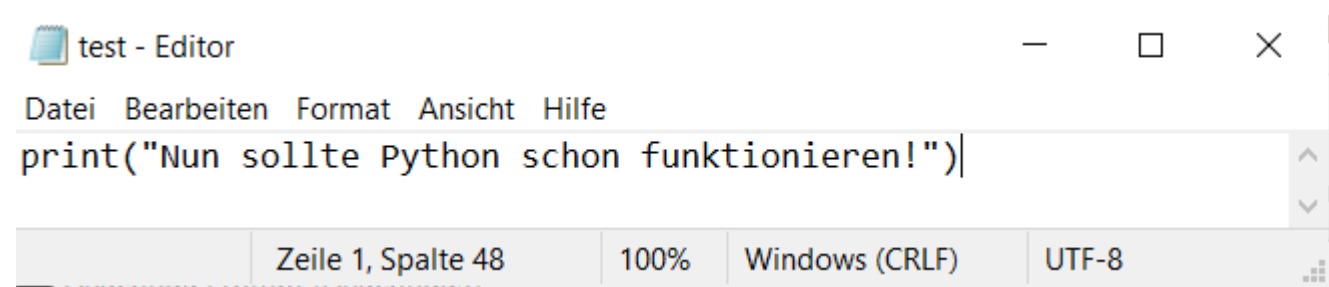
```
sudo apt install python3
```

Python - Installation

Dann sollte folgendes im Terminal funktionieren:



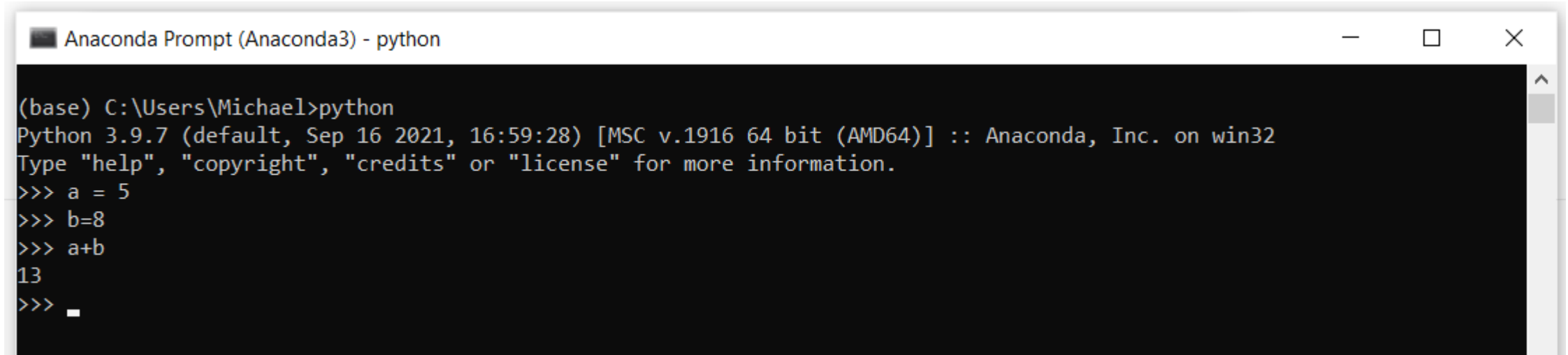
```
Eingabeaufforderung - pytho | X + v  
Microsoft Windows [Version 10.0.26200.8037]  
(c) Microsoft Corporation. Alle Rechte vorbehalten.  
  
C:\Users\Michael>python  
Python 3.14.4 (tags/v3.14.4:23116f9, Apr 7 2026, 14:10:54) [MSC v.1944 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license" for more information.  
>>> |
```



```
test - Editor  
Datei Bearbeiten Format Ansicht Hilfe  
print("Nun sollte Python schon funktionieren!")  
  
Zeile 1, Spalte 48 100% Windows (CRLF) UTF-8
```

```
C:\Users\Michael\Sciebo2\Python Scripts\OFP>python test.py  
Nun sollte Python auch schon funktionieren.  
  
C:\Users\Michael\Sciebo2\Python Scripts\OFP>|
```

Python REPL (Read Evaluate Print Loop)



```
Anaconda Prompt (Anaconda3) - python

(base) C:\Users\Michael>python
Python 3.9.7 (default, Sep 16 2021, 16:59:28) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> b=8
>>> a+b
13
>>> _
```

Möglicherweise angenehmer/benutzerfreundlicher: IDE (integrated development environment), z.B.

- Visual Studio Code
- PyCharm
- Jupyter Notebooks
- Atom
- Spyder

Eine **Variable** ist ein Name für ein Ort im Speicher der Daten eines bestimmten Typs enthält.

Im vorherigen Programm zeigt **name** auf eine Adresse im Speicher und gibt die dort gespeicherten Daten zurück, wenn **name** aufgerufen wird.

Der Wert der Variable (=welche Daten an der Stelle gespeichert sind), kann sich im Laufe eines Programms ändern

```
1 name = input("Wie heißt Du? ")
2 print("Hallo", name)
3
4
5 name = input("Wie heißt Dein(e) Sitznachbar*in? ")
6 print("Hallo", name)
7
```

```
In [14]: runfile('some_random_code_for_the_lecture.py')
Wie heißt Du? Michael
Hallo Michael
Wie heißt Dein(e) Sitznachbar*in? Theresa
Hallo Theresa
```

Der Operator **=** ist ein Zuweisungsoperator, der den Rückgabewert der Funktion **input** der Variablen **name** zuordnet.

Variablen sollten sinnvolle Namen haben! Die übliche Konvention ist es nur Kleinbuchstaben zu verwenden und Leerzeichen durch Unterstriche zu ersetzen!

Schlechte Variablennamen: aA, Aaaa, aaAAaa

Gute Variablennamen: first_name, last_name, number_of_people

Python ist eine *dynamisch typisierte* Sprache: Der Datentyp einer Variable muss nicht explizit mit angegeben werden und kann sich im Laufe eines Programms ändern!

```
1 a = 1
2 print(type(a))
3 b = 1.0
4 print(type(b))
5 c = True
6 print(type(c))
7
8 a = "Hallo"
9 print(type(a))
10
```

```
In [8]: runfile('some_random_code_for_the_lecture.py')
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'str'>
```

Wie sie später sehen werden, ist dies bei Java anders. Java ist *statisch typisiert*.

Python und Java sind beide case-sensitive: Groß- und Kleinschreibung spielt eine Rolle. **a** und **A** können unterschiedliche Variablen sein (sollten sie aber nicht – das wäre kein guter Stil!).

Variablen halten die Daten, die ein Programm verarbeitet.

Was mit den Daten geschehen soll, wird durch *Anweisungen (statements)* bestimmt.

Eine Anweisung ist eine Instruktion die vom Python-Interpreter ausgeführt werden kann, z.B. der Aufruf einer Funktion oder die Zuweisung eines Werts zu einer Variable.

Ein *Ausdruck (expression)* ist eine Anweisung, die einen Wert liefert, z.B. der arithmetische Ausdruck $2+5$.

Der Operator `=` ist der *Zuweisungsoperator*, der einen Rückgabewert eines (rechts stehenden) Ausdrucks einer (links stehenden) Variable zuordnet.

Achtung: Während große Teile der Fähigkeit zu programmieren mathematisch sind, unterscheidet sich `=` vom mathematischen Gleichheitszeichen!

```
In [9]: 6 = x
File "C:\Users\Michael\AppData\Local\Temp\ipykernel_2872\1558924665.py", line 1
      6 = x
      ^
SyntaxError: cannot assign to literal
```

```
In [10]: x = 1
In [11]: x = x + 1
In [12]: print(x)
2
```

Als nächstes werden wir einige grundlegende Datentypen (Strings, Zahlen, Booleans) und Manipulationen dieser kennenlernen, bevor wir uns Kontrollstrukturen (**if**-statements) und Schleifen (**for** und **while**) widmen.

Falls sie bereits eine andere Programmiersprache kennen, ist dies nur das Lernen einiger neuer ‚Vokabeln‘ / Notation.

Falls nicht – keine Angst – Python ist eine der einfachsten zu lernenden Programmiersprachen! Insbesondere gilt zwar

“One hour of thinking can easily save one week of programming!”
Or: programming is **not** trial-and-error!

jedoch ist die Umsetzung eines gut durchdachten Programms in einer neuen Programmiersprache durchaus etwas trial-and-error (+ Suchen der richtigen Befehle in Skript / Internet).

Nur durch intensives Üben kann man eine Programmiersprache lernen!

Wir möchten von Anfang an wissen, welche Erwartungen an mich (für die Klausur) gestellt werden!

In der Vorlesung werden Konzepte, die Syntax in Python und Beispiele dazu besprochen.

Erwartet wird, dass sie den Transfer der Konzepte (und deren Umsetzung in richtiger Syntax) auf neue Probleme beherrschen!

(Schwieriges) Beispiel:

- Wir werden den Datentyp *Liste* (Sammlung an Werten) kennenlernen
- Wir werden das Konzept einer *for-Schleife* kennenlernen und ein Beispiel von einem Sport-Spielplan machen
- Wir werden das Konzept kennenlernen, Variablen innerhalb einer Schleife zu ändern

Folgende Aufgabe ist nicht unrealistisch.

Vervollständigen sie den folgenden Code

```
symbols = ["scissors", "paper", "rock", "lizzard", "spock"]  
verbs = ["cuts", "gets crushed by", "decapitates", "gets smashed by",  
        "covers", "gets eaten by", "disproves",  
        "crushes", "gets vaporized by",  
        "poisons"]  
### Hier kommt ihr Code hin ###
```

um mittels zweier Schleifen die folgende Ausgabe zu erzeugen

```
scissors cuts paper  
scissors gets crushed by rock  
scissors decapitates lizzard  
scissors gets smashed by spock  
paper covers rock  
paper gets eaten by lizzard  
paper disproves spock  
rock crushes lizzard  
rock gets vaporized by spock  
lizzard poisons spock
```

Für größere Projekte macht es oft Sinn ein „Environment“ zu erzeugen, z.B.

```
python -m venv mein_ofp_environment
```

erzeugt eine Umgebung in der dann weitere Pakete (mit verschiedenen Versionen) installiert werden können

```
source mein_ofp_environment/bin/activate  
.\mein_ofp_environment\Scripts\activate
```

aktiviert das Environment
(oben: Linux/Mac, unten Windos)

```
deactivate
```

deaktiviert das aktuelle Environment

Wenn man Funktionalitäten aus bestimmten öffentlichen Pakten nutzen möchte, kann man einen Paketmanager verwenden, z.B. *pip*

```
pip install (paketname)
```

Mit **pip list** werden alle aktuell installierten Pakete angezeigt.

```
C:\Users\Michael\Sciebo2\Python Scripts\OFP>python -m venv mein_ofp_environment

C:\Users\Michael\Sciebo2\Python Scripts\OFP>.\mein_ofp_environment\Scripts\activate

(mein_ofp_environment) C:\Users\Michael\Sciebo2\Python Scripts\OFP>pip list
Package Version
-----
pip        26.0.1

(mein_ofp_environment) C:\Users\Michael\Sciebo2\Python Scripts\OFP>pip install numpy
Collecting numpy
  Using cached numpy-2.4.4-cp314-cp314-win_amd64.whl.metadata (6.6 kB)
Using cached numpy-2.4.4-cp314-cp314-win_amd64.whl (12.4 MB)
Installing collected packages: numpy
Successfully installed numpy-2.4.4

(mein_ofp_environment) C:\Users\Michael\Sciebo2\Python Scripts\OFP>pip list
Package Version
-----
numpy      2.4.4
pip        26.0.1

(mein_ofp_environment) C:\Users\Michael\Sciebo2\Python Scripts\OFP>deactivate
```

Variablen (Platzhalter zum Merken)

Zuweisungsoperator `=`

Rechenoperatoren `+`, `-`, `*`, `/`

Ausgabe auf der Konsole `print`

Eingabe in die Konsole `input`