# Numerische Methoden in der Physik (Teil I)

## Professur Simulation Naturwissenschaftlicher Prozesse

Dr. Janett Prehl

Institut für Physik

Wintersemester 2023/24
Stand: 1. Februar 2024

# Organisatorisches

## Kontakt

Dr. Janett Prehl | C60.212 | 531 35612 | janett.prehl@physik.tu-chemnitz.de

## Vorlesungs-/ Übungszeit

| | | |
|---|---|---|
| Do | 11:00 - 12:30 Uhr | C60.205 |
| Do | 12:30 - 13:15 Uhr | C60.205 |

## Unterlagen

**Opal:** `bildungsportal.sachsen.de/opal/auth/RepositoryEntry/21346615297?421`

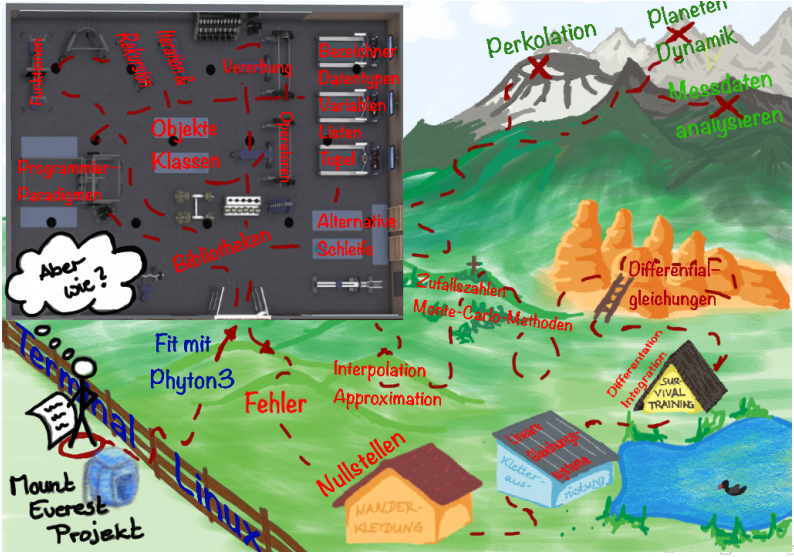
# Organisatorisches

## Wissensstand (Umfrage)

- Wer hat schon mal mit Linux/Unix und der Console in Linux/Unix gearbeitet?
- Haben Sie bereits eine Programmiersprache gelernt?
- Wie gut schätzen Sie Ihre eigenen Programmierkenntnisse in Schulnoten von 1 (kann nicht programmieren) bis 6 (mir kann man nichts mehr beibringen) ein?

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| | | | | | |

- Was möchten Sie am Ende dieser Veranstaltung können?


Funktionen
Rekursion
Iteration
Vererbung
Objekte
Klassen
Operatoren
Programmier-Paradigmen
Bibliotheken
Skalierbarer Datentypen
Variablen
Listen
Typen
Alternative
Schleife
Aber wie?
Perkolation
Planeten-Dynamik
Messdaten analysieren
Zufallszahlen
Monte-Carlo-Methoden
Differential-gleichungen
Differentation
Integration
SUR-VIVAL-TRAINING
Fit mit Phyton3
Fehler
Interpolation
Approximation
Nullstellen
Terminal
Linux
Mount Everest Projekt
WANDER-KLEIDUNG
Klettern
Trekking

## Geplante Themen: Wintersemester

- Einführung Linux (Kurzversion)
- Einführung Programmiersprachen
  - Iteration, Rekursion
  - Prozedurales- und objektorientiertes Programmieren (OOP)
- Einführung `python3`
  - Eingaben, Variablen, Listen, Tupel
  - Ablaufsteuerung: Alternativen, Schleifen
  - Bibliotheken: Fehlerbehandlung, Mathematik, Graphik
  - OOP: Klassen, Objekte, Operatoren, Vererbung
- Extra-Wünsche? `SymPy`, Animation (Basic!)

## Geplante Themen: Sommersemester

Einführung Numerik

- Nullstellenberechnung
- Lineare Gleichungssystem
- Differentiation und Integration
- Differentialgleichungen – Planetenbewegung
- Zufallszahlen/ Monte-Carlo Methoden – Brownsche Bewegung/ Perkolation
- Interpolation/ Approximation – Messdaten analysieren

**Modulprüfung im Sommersemester:**

30 min mündliche Prüfung (über Inhalte beider (!!!) Semester)

## Verhalten in der Veranstaltung

- Keine Anwesenheitspflicht: Wer nicht möchte, kann gehen.
- Bitte Ruhe, wenn Vorlesender spricht.
- Dezentes Sprechen erlaubt.
- Diskussion & Fragen (auch außerhalb der Übung) ausdrücklich erwünscht!
- Bei Problemen mit dem Vorlesungsstoff zeitnah zum Vorlesenden gehen.

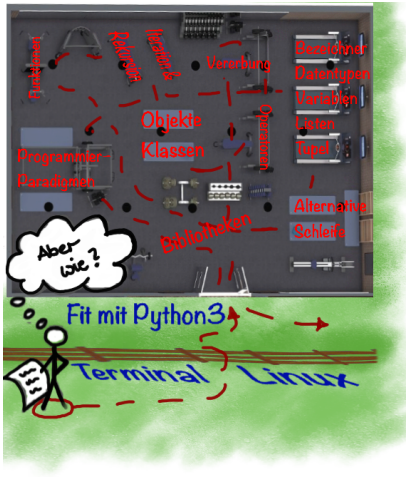
# 1. Einheit

## Ziel

Erster Kontakt / Umgang mit

- dem Arbeitsplatz,
- dem Betriebssystem Linux und
- der Shell, dem Terminal *.

Einleitung python-Umgebung



# Computerpool

## Zugang

- Zugang mittels Magnetkarte (TU-Card, DACS-System)
- ***Nutzungszeit:***
  *Werktags von 7:00 – 21:30 (6:00-24:00)*
- Login über URZ-Kürzel

## Hardwareausstattung

- 23 Linux Rechner
- 2 GPGPU-Knoten (lgpgpu1 & lgpgpu2)
- Drucker, nutzbar mit eigenem Papier
- **Jede Betriebsstörung melden (janett.prehl@physik.tu-chemnitz.de)!**
  Nicht am Gerät *rumbasteln*!

## Verhaltensregeln

- *Benutzung nur zum Zwecke von Forschung & Lehre*
- Jeder kann Raum nutzen (in Absprache mit Übungsleiter)
- Essen, Trinken, Rauchen, Zweckentfremdung von Geräten **verboten**
- **Nach Benutzen der Rechner: Bildschirm & Rechner anlassen!**
- Einhaltung des aktuellen Hygienekonzepts

# Organisation der Rechentechnik an der TUC

## Vor Ort (lokal)

Hardware
Software

## Dezentral

(eigene) Daten

## → PC

Einfache Wartung
Netzentlastung

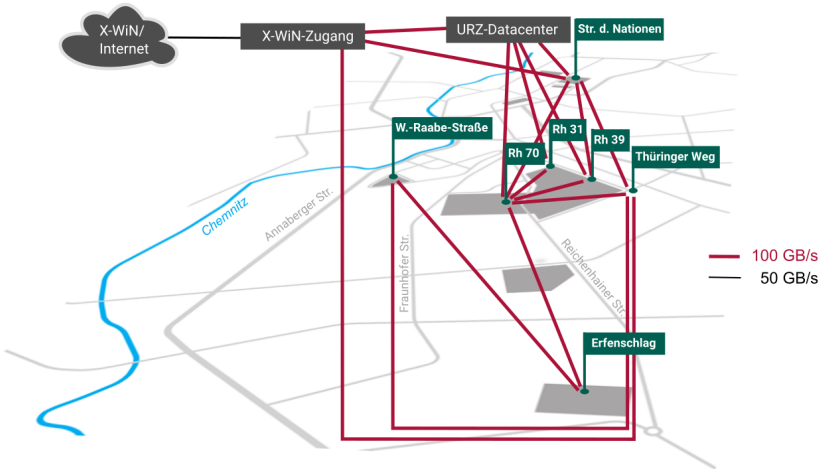
## → afs (Andrew File System)

Ausfallsicherheit
Daten von überall abrufbar

## GreenIT

Rechner schalten sich im Anmeldebildschirm nach einigen Minuten selbstständig aus.
Eigenhändiges Anschalten per Knopfdruck notwendig.

# Dezentral: Infrastruktur



# An- & Abmelden

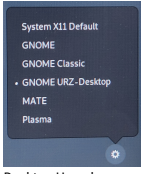
## Anmelden

- Falls Bildschirm aus: Rechner anschalten (GreenIT)
- Bei Anmeldung **Groß- und Kleinschreibung beachten!**
- Benutzername als Login-Name eingeben
- *Desktop-Umgebung auswählen*
- Passwort im extra verschlüsselten Feld eingeben


Bild von https://de.linuxcapable.com

## Abmelden

- Möglichst alle gestarteten Programme schließen
- Entsprechend der Benutzeroberfläche abmelden
- Nach Erscheinen des **Anmeldebildschirms** den Rechner verlassen
- **Tipp:** Wenn Person angemeldet → rechts unten Personenlogo zum Ummelden




Desktop-Umgebungen

# Schnittstelle Betriebssystem – Nutzer

## Vor Ort (lokal): Betriebssystem (BS)

Schnittstelle zwischen Hardware und Anwendungen

## Unix, Linux & Distributionen

| | |
|---|---|
| Unix | Eines der ersten Mehrbenutzer-BS (1969 entwickelt)<br>Etablierte u.a. Filesystem Hierarchy Standard |
| Linux | Unabhängig von Unix entwickeltes BS<br>Ebenfalls Mehrbenutzer-BS<br>Folgt Filesystem Hierarchy Standard<br>$\Rightarrow$ Unixoides Betriebssystem |
| Distribution | Zusammenstellung von Betriebssystem-Komponenten und aufeinander abgestimmten Anwendungen<br>z.B.: **Debian**, Fedora, Ubuntu, LinuxMint, Trisquel, ... |

# Verwaltung von Daten

## Speicherung von Daten

- Ziel: Sinnvolles abspeichern von Daten mit schnellen Zugriffszeiten
- Problem beim "Hinter einander weg speichern"
  - Lücken durch Löschen von Daten
  - Lücken teilweise nicht groß genug für neue Daten
  - $\rightarrow$ Teilen der Daten $=$ Anstieg der Ausführungszeit
- Lösung: Getrennte Speicherung der Verzeichnisstruktur und Daten

## Eine Datei (File)

- Jedes „Ding“ im Rechner was Daten erzeugt oder empfängt.
- Identifiziert mittels INODE

## INODE (Index Node)

- stellt Metadaten bereit: Seriennummer, Zugriffsrechte, Besitzer, Größe, letzter Zugriff, usw.

## Verzeichnis – DENTRY (directory entry)

Verzeichnis besteht aus Verzeichniseinträgen (directory entries).
DENTRY:

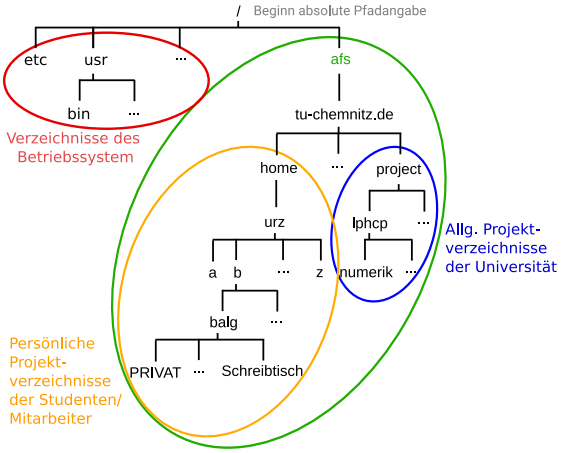
- INODE
- Namen des Objekts
- Elternobjekt

## Kodierung durch Shell

`Elternobjekt/Kindobjekt` **Trennzeichen ist der Slash (/)**

## Dateisystem

- Verzeichnisse des Betriebssystems
- afs - Verzeichnisstruktur (Andrew File System)
  - Persönliche Verzeichnisse der Studenten/ Mitarbeiter
  - Projektverzeichnisse einzelner Gruppen/ Organisationen der TUC
- Wichtige Verzeichnisse:
  - Root-Verzeichnis /
  - Aktuelles Verzeichnis in dem man sich befindet .
  - Elternverzeichnis des aktuellen Verzeichnisses ..

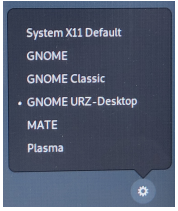


# Getting Started

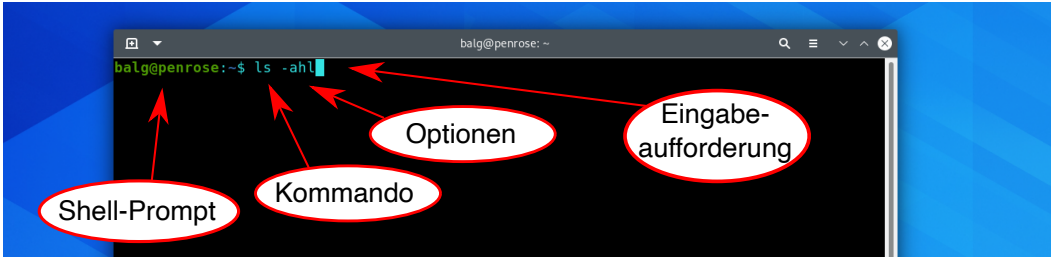
## Aufgaben

- Loggen Sie sich ins System ein!

  **Hinweis:** Zahnrad-Symbol rechts unten im Anmeldebildschirm (nach Eingabe Benutzername) zum Wechseln der Nutzeroberfläche
- Suchen Sie Firefox/ GoogleChrome, Dateien/ Nautilus, Terminal/ Konsole, gedit/emacs/geany, libreOffice!
- Starten Sie ein Terminal/ Konsole!



# Konsole (Terminal) – Shell

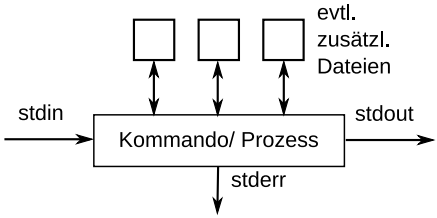



## Interpreter

- Jede Anweisung wird direkt interpretiert und die entsprechende Aktion (Prozess) ausgeführt
- Anweisungsstruktur (Kommando): `Kommando [optionen] <Argumente>`

## Datenströme bei Prozessen

0 Standardeingabe (stdin)
1 Standardausgabe (stdout)
2 Standardfehlerausgabe (stderr)
+ evtl. zusätzliche Dateien




## Umleiten von Datenströmen

```
cmd1 > Ausgabedatei
cmd1 >> Ausgabedatei
cmd1 < Eingabedatei
```

## Leitungen (Pipes) – Verknüpfung Datenströmen

Standardausgabe wird Standardeingabe eines weiteren Programms:

```
cmd1 | cmd2 | ...
```

## Vermeiden von Sonderzeichen

Auf Grund der Sonderbedeutung vieler Sonderzeichen im Unix/Linux-System und bei diversen Programmiersprachen ist es nicht sinnvoll Lehrzeichen, Umlaute und Sonderzeichen in Dateinamen zu verwenden.

## Typische Kommando-Struktur:
**Kommando - Leerzeichen - [Optionen] - Leerzeichen - <Argumente>**

```
#Hilfe zur Bash-Shell anzeigen
man bash

#alle Dateien des eigenen Backup-Ordners auflisten
ls -la ~/BACKUP

#Lege leere Datei an und editiere sie via Datenstromumleitung
touch test.txt
echo "Das ist ein test." > test.txt

#Wechsel in den Ordner PRIVAT und lege ein Verzeichnis fuer NumMethPhy an
cd PRIVAT
mkdir numerik

#Kopiere Datei von ~/ nach ~/PRIVAT/numerik
cp ../../test.txt .
```


balg@penrose:~$ ls -hl
insgesamt 1,3M
-rw-r--r-- 1 balg urz_users 2,7K 23. Jun 14:41 20220623_SimNatPro_Pruefungsinfo.txt
-rw-r--r-- 1 balg urz_users 838K 25. Sep 2019 appel.m.96.nmr.483.pdf
drwxr-xr-x 4 balg urz_users 2,0K 13. Apr 2021 Arduino
drwxr-xr-x 2 balg urz_users 2,0K 8. Dez 2014 autosave
drwxr-xr-x 2 balg root 2,0K 2. Jun 2003 BACKUP
drwxr-xr-x 3 balg urz_users 2,0K 15. Jul 14:01 Bilder
drwxr-xr-x 5 balg urz_users 4,0K 9. Feb 2021 bin
drwxr-xr-x 2 balg urz_users 2,0K 7. Jul 2020 build
-rw------- 1 balg urz_users 1,9K 4. Okt 2019 DEADJOE
drwx------ 6 balg urz_users 4,0K 31. Aug 14:35 Desktop
drwxr-xr-x 3 balg urz_users 2,0K 7. Jul 2020 Dokumente
drwxr-xr-x 2 balg urz_users 2,0K 17. Jan 2022 Downloads
root 2,0K 1. Aug 2018 EigeneDateien
urz_users 16K 17. Mär 2022 Hardware_212015.xlsx_0.od
urz_users 52K 3. Mär 2020 Hoffmann-ControlledDynamic
drwxr-xr-x 2 balg urz_users 4,0K 26. Nov 2018 home
2 balg urz_users 2,0K 17. Mai 2011 include
2 balg urz_users 2,0K 7. Jul 2020 install
3 balg urz_users 2,0K 1. Okt 2008 java
-rw------- 1 balg root 65 16. Okt 2020 jupyterhub_cookie_secret
-rw-r--r-- 1 balg urz_users 0 9. Okt 2020 jupyterhub.error.txt
-rw-r--r-- 1 balg root 4 16. Okt 2020 jupyterhub-proxy.pid
-rw-r--r-- 1 balg root 100K 16. Okt 2020 jupyterhub.sqlite
-rw-r--r-- 1 balg urz_users 6,0K 30. Jul 2020 LeMonADE_Ex0.cpp
-rw-r--r-- 1 balg urz_users 5,3K 30. Jul 2020 LeMonADE_Ex1_RNG.cpp
-rw-r--r-- 1 balg urz_users 9,5K 30. Jul 2020 LeMonADE_Ex2_classMol.cpp
-rw-r--r-- 1 balg urz_users 12K 30. Jul 2020 LeMonADE_Ex3_Feat.cpp
-rw-r--r-- 1 balg urz_users 8,0K 30. Jul 2020 LeMonADE_Ex4_UpdAnal.cpp
-rw------- 1 balg urz_users 9,3K 15. Jun 2020 letterToEditors_CKHS.txt_0.odt
-rw-r--r-- 1 balg urz_users 4,9K 16. Mai 11:01 logfile.txt
drwx------ 6 balg urz_users 4,0K 3. Mär 2020 Mail
drwxr-xr-x 2 balg urz_users 2,0K 16. Jun 2011 Musik
-rwxr-xr-x 1 balg urz_users 7,6K 23. Mär 2019 newbib
Datei-/ Verzeichnis-größe
Letztes Änderungs-datum
afs-Zugriffsrechte
Ersteller
Datei-/ Verzeichnis-name

## Nützliche Befehle & Programme

| | | |
|---|---|---|
| cd <Ziel> | *change directory* | Verzeichnis wechseln |
| cp <Ursprung> <Ziel> | *copy* | Datei von <Ursprung> nach <Ziel> kopieren |
| cp [-r] <Ursprung> <Ziel> | *copy [rekursive]* | Ordner von <Ursprung> nach <Ziel> kopieren |
| ctrl + c | *cancel* | Abbruch des aktuell ausgeführten Programms |
| ctrl + r | | Rückwärtssuche in ausgeführten Kommandozeilen |
| ls [-ahl] <Pfad> | *list [all human list]* | Auflisten von Dateien und Ordnern des Verzeichnisses |
| man <Befehl> | *manual* | Hilfe zu Befehl (cmd) anzeigen |
| mkdir <Pfad> | *make directory* | Ordnern anlegen |
| pwd | *print workin directory* | Anzeigen des aktuellen absoluten Verzeichnispfades |
| rm <Pfad> | *remove* | Datei löschen |
| rm [-r] <Pfad> | *remove [rekursive]* | Ordner löschen |
| tree | | Verzeichnisstruktur betrachten |
| clear | | Bereinigt Terminaldarstellung |
| | | |
| tab | *Tabulator* | Befehlsvervollständigung |
| up/down | *Pfeiltasten* | Zugriff auf ausgeführte Kommandozeilen |

## Kennlernübungen - Dateisystem

1. In welchem Verzeichnis bzw. Ordner befindet Ihr Euch?
2. Welche Standard-Ordner sind vorhanden?
3. Was bedeutet relative bzw. absolute Pfadangabe und wie sieht sie jeweils aus?
4. Findet die Zeichen/Symbole ~ , /, * auf der Tastatur.
5. Welche Pfade/Dateien verstecken sich hinter den Zeichen ~ , / , . , .. und *?
6. Wiederholung: Starten Sie über Terminal/ Konsole die Programme Firefox/ GoogleChrome, Dateien/ Nautilus, gedit, LibreOffice in der Konsole.

   **Tipp:** Verwenden Sie & am Ende jeder Befehlszeile, alternativ Strg+Z → bg+Enter

## Lösung

1. Homeverzeichnis
   `/afs/tu-chemnitz.de/home/urz/[urz-Kürzel-Initial]/[urz-Kürzel]/`
3. Pfadangaben:

   | | |
   |---|---|
   | Absolute Pfadangabe: | Vollständiger Verzeichnispfad beginnend mit / |
   | Relative Pfadangabe: | Verzeichnispfad ausgehend vom aktuellen Ordner |

5. Symbole $\leftrightarrow$ Pfade/Dateien:

   | | |
   |---|---|
   | ~ | Homeverzeichnis |
   | / | Wurzelverzeichnis |
   | . | aktuelles Verzeichnis |
   | .. | Elternverzeichnis |
   | * | beliebige Zeichen (keine bis alle) |

6. `firefox &`, `nautilus &`, `gedit &`, `libreoffice &`

## Umgang mit der Konsole der Kommandostruktur

1. Legen Sie eine leere Datei an.
2. Überprüfen Sie, ob das Anlegen erfolgreich war.
3. Legen Sie ein Verzeichnis (z.B. `NumMethPhy`) für diese Lehrveranstaltung in Ihrem Homeverzeichnis an.

   Es sinnvoll für jede Einheit ein neues Subverzeichnis anzulegen.
4. Wechseln sie in das neu-angelegte Verzeichnis.
5. Kopieren Sie `numerik.txt` aus dem Elternverzeichnis in das aktuelle Verzeichnis unter dem Namen `testNeu.txt`
6. Lassen Sie sich das aktuelle Arbeitsverzeichnis anzeigen.
7. Üben Sie den Umgang mit Datenstromumlenkung (Bsp.-Aufgaben auf Folgefolien).
8. *Zusatz:* Probieren Sie alle zuvor gelisteten Kommandos aus.

## Kennlernübungen zum Umleiten von Datenströmen

Was bewirken die folgenden Befehle?

1. `echo "Numerik"`
2. `echo "numerik" > numerik.txt`
3. `cat numerik.txt`

Wie verändert sich der Inhalt von `numerik.txt`, wenn Sie folgende Befehle durchführen:

1. `echo "Numerik" > numerik.txt`
2. `echo "Numerik" >> numerik.txt`

**Tipp 1:** Benutzt die Pfeiltasten zum Wiederaufruf bereits abgeschickter Befehle.

**Tipp 2:** Benutzt die `tab`-Taste zur automatischen Wortverlängerung bei Befehlen und Pfadeingaben.

# Zusatz

## Kennlernübungen II zum Umleiten von Datenströmen

Probieren Sie ausserdem noch folgende Befehle aus:

1. `cat >hello.txt`, dann beliebige Eingabe, Beenden mit `strg+d`
2. `cat hello.txt`
3. `echo "Hello World" > helloWorld.txt` oder
   `cat >helloWorld.txt`, Hello World
4. `cat numerik.txt hello.txt helloWorld.txt`
5. `cat numerik.txt hello.txt helloWorld.txt | sort [-r] [> final.txt]`

# Python-Terminal

## Starten - Eingabe - Beenden der Python-Umgebung

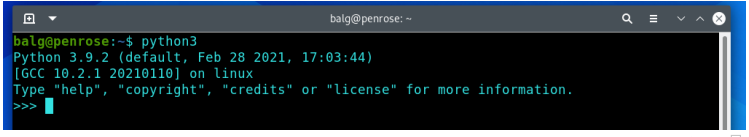
Starten des interaktiven Python-Terminals (Python3) im Terminal:

- `python` (oder `python3`)
- `/bin/python3`
- `/afs/tu-chemnitz.de/ubc/PYTHON/python` (Hier sind verschiedene Versionen verfügbar)

Zen of Python: `import this`

Beenden des interaktiven Python-Terminals:

- `strg+d`
- `exit()`



## Programm - Ausführen mit Python

Schreibe Minimal-Python-Programm:

- Lege Datei an: `touch ZenOfPython.py`
- Schreibe in Datei: `import this` (Editor, Datenstromumleitung, etc.)

**Ausführen des Python-Programms im Terminal**: `python3 ZenOfPython.py`

**Hinweis:** Ausgaben sind Datenströme (`stdout, stderr`), die umlenkbar und dadurch speicherbar sind.

## Einfache Benutzung (z.B. via SSH)

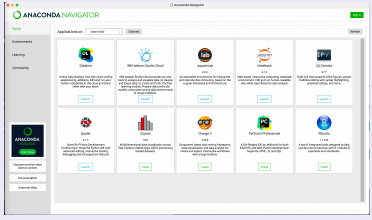
- Texteditor (z.B. `gedit`, `emacs`, anderer Texteditor) + **Python-Interpreter**
- Interaktives Python-Terminal (im Terminal)
  **Nachteil:** Enthält nur eine abgespeckte Version der Standardbibliotheken
- Python-IDEs bzw. Python-Distriubtionen

# Python – Editoren & Entwicklungsumgebungen

## Komfortabler: Entwicklungsumgebungen

Enthalten einen umfangreichen Satz an Standard-bibliotheken, automatische Erkennung von Funktionen, Wortverlängerung, . . . und (meist) ein interaktives Python-Terminal zum Testen.

**Beispiele:** Mu, Thonny, PyCharm, **Spyder**, **Jupyter-Notebooks** , Eric, . . .




## Anaconda: Spyder, Jupyter-Notebooks

Python-Distribution Anaconda enthält Sypder und Jupyter-Notebooks.
Ist für alle Plattformen/ Betriebssysteme verfügbar.

# Zugang zu Ressourcen

## Login von Außen

- Allgemein: `login.tu-chemnitz.de`
- Dateimanager: `wfm.hrz.tu-chemnitz.de/swfm/index.html`
- E-Mail: `mail.tu-chemnitz.de`

## Unterlagen - Opal

`bildungsportal.sachsen.de/opal/auth/RepositoryEntry/21346615297?42`

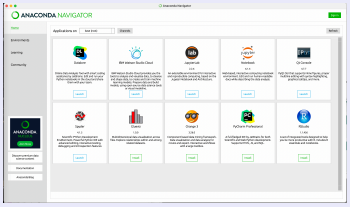
## Bash-Hilfe (Unsere Standardshell)

`http://www.selflinux.org/selflinux/html/shellprogrammierung.html`

## Software für zu Hause

**python3** : Mindestanforderung interaktives Terminal + Standardbibliotheken

→ Empfehlung anaconda-Paket (https://www.anaconda.com/products/individual)
→ Installation anaconda (inkl. Spyder)
https://docs.anaconda.com/anaconda/install/




## Empfohlen Software für zu Hause

- VPN-Client (https://www.tu-chemnitz.de/urz/network/access/vpn.html)
- X2Go-Client (https://www.tu-chemnitz.de/mathematik/mrz/x2go.php)
- SSH-Client 'PuTTY' (https://www.tu-chemnitz.de/informatik/friz/windows/putty.php)

# 2. Einheit

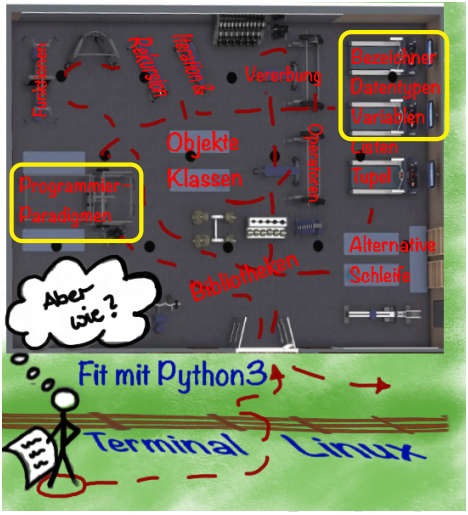
## Ziel

- Python allgemein
- Programmiersprache Python

→ Daten-/ Objekttypen

→ Bezeichner/ Variablen

→ Rechen-/ Vergleichsoperationen



# Einführung zur Programmiersprache

## Allgemeine Begriffe

- Algorithmus:
  geordnete Folge von Handlungsvorschriften, die mit endlich vielen Schritten das gewünschte Resultat liefert (und automatisch auf technischen Gerät abgearbeitet werden kann)
- Programm:
  für den Computer aufbereiteter Algorithmus
- Programmiersprache:
  formale Sprache zur Beschreibung von Algorithmen
- Software:
  Oberbegriff für Programme (in verschiedenen Repräsentationen) und beschreibende Dokumentation

# Programmiersprache Python


## Allgemeines

- Anfang der 1990er Jahre von Guido van Rossum entwickelt
- Name basiert auf britischer Komikertruppe *Monty Python* (Monty Python's Flying Circus), die Schlange wird aber oft als Symbol verwendet
- Moderne, universelle, höhere Programmiersprache
- Freie Open Source Software der Python Software Foundation: www.python.org/psf
- Selbstständige und einbettbare Interpreter- bzw. Skriptsprache (Übersetzung zur Laufzeit nicht zur Compile-Zeit) → **Nachteil:** Probleme bei der Rechenzeitperformance (Schnelligkeit)
- Interpretative Abarbeitung → viele Aktionen werden erst zur Laufzeit ausgeführt (z.B. Namenauflösung, Typprüfung, ...)
- Viele Literatur und Online-Dokumentation verfügbar

# Programmiersprache Python

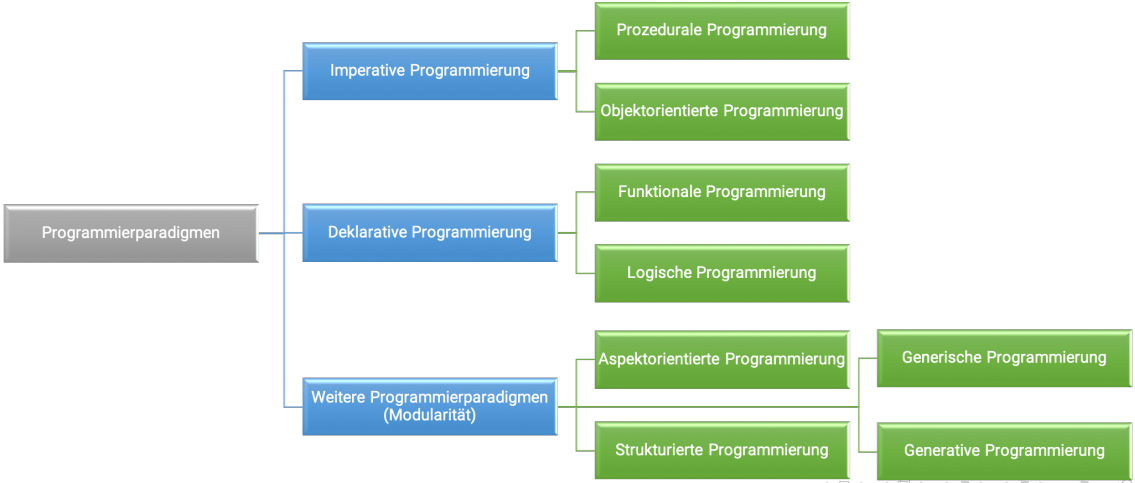
## Allgemeines

- Unterstützt mehrere Programmierparadigmen
  - objektorientiert
  - imperativ / prozedural
  - funktional
  - aspektorientiert

## Was ist ein Programmierparadigma?

Ein Programmierparadigma ist der zugrundeliegende Ansatz für die Programmierung. Es beschreibt den grundlegenden Stil, wie das Programm entworfen wird. Er soll die Entwicklung von “gutem Code” unterstützen.

# Programmierparadigmen



# Hauptprogrammierparadigmen

## Imperative Programmierung

Ein Programm besteht aus einer Folge von Befehlen, die durch den Rechner nacheinander abgearbeitet werden um ein Ergebnis erreichen.

- Prozedural

  Die zu lösende Aufgabe wird in Teilaufgaben zerlegt, die in Funktionen/Prozeduren zusammengefasst werden und innerhalb des Programms wiederverwendet werden können.

- Objektorientiert

  Die zu lösenden Aufgabe wird in Objekte zerlegt, die durch Eigenschaften (Attribute) und Funktionen (Methoden) charakterisiert werden. Die Daten können nur über Methoden verändert werden. Diese Objekte werden typischerweise an der Realität angelehnt.

# Hauptprogrammierparadigmen

## Deklarative Programmierung

Ein Programm beschreibt hierbei nicht die Vorgehensweise um zum Ergebnis zu gelangen, sondern welche Ergebnisse am Ende stehen. Die Problemlösung wird dem Programm mit Hilfe entsprechender Algorithmen überlassen.

- Funktional

  Es können mathematische Funktionen definiert werden. Im Gegensatz zu den prozeduralen Funktionen können diese sofort auf Datensätze und Prozesse angewendet werden.

- Logisch

  Es werden eine Menge an Axiomen (Ansammlung von Fakten, Annahmen) aufgestellt. Bei einem Logikprogramm versucht der Interpreter die Aufgabe anhand der Axiome zu berechnen.

# Programmierparadigmen

## Weitere Paradigmen – Aspekt: Modularität

- Aspektorientiert

  Man unterscheidet zwischen Komponenten (Systemeigenschaften, die in verallg. Funktionen “abgekapselt” werden können) und komponentenübergreifende Aspekte (können nicht abgekapselt werden). Erhöht die Modularität der objektorientierten Programmierung.

- Strukturiert

  Ist ein Verfahren zur Programmentwicklung. Dabei wird Angenommen, dass die Programmbestandteile einheitlich und selbstständig sind und es werden die Bedingungen dargestellt, wie diese zusammenwirken.

# Programmierparadigmen

## “Guter Code” ?!

- KISS = Keep it simple and stupid
  trickreiches Programmieren vermeiden, Tricks erhöhen Fehlerrate,
  Faustregel: 10 % des Programms benötigen 90% der Rechenzeit
  Hintergrund: Rechenzeit ist billiger als Arbeitszeit
- Guter Code ist klar strukturiert und sorgfältig dokumentiert/ kommentiert.
  Klare Struktur vereinfacht Fehlersuche bei Modifikationen.
  Minimum: Schnittstellen dokumentieren (Funktionsübergaben, . . . )
- Portabilität hat einen sehr hohen Wert: Kontrolle der Zuverlässigkeit $\rightarrow$ effektives Mittel der Fehlersuche; Vermeidet Probleme bei Hardwareausfällen, da auf anderer Hardware weiterverwendbar
- Prüfen Sie die einzelnen Programmteile vor dem Zusammenbau sorgfältig. Bsp.: 1% Ausfallwahrscheinlichkeit, 100 Komponenten $\rightarrow$ Wahrscheinlichkeit Ergebnis korrekt: 36.6%

# Programmierparadigmen

## “Guter Code” ?! - Literatur

- Robert C. Martin *“Clean Code – A Handbook of Agile Software Craftsmanship”* Robert C. Martin Series. Prentice Hall, Upper Saddle River, NJ, 2009
(PDF downloadbar `https://github.com/jriall/clean-code-notes`)
- Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Pumbley, Ben Waugh, Ethan P. White, and Paul Wilson. *“Best practices for scientific computing”* PLoS Biology, **12(1)**:e1001745, 2014, **DOI 10.1371/journal.pbio.1001745** (PDF downloadbar *im Uninetz*)

# Programmiersprache Python

## Python Versionen – Rückwärtskompatibilität

Python bemüht sich um hohe Rückwärtskompatibilität durch evolutionäre Weiterentwicklung der Sprache, aber gezielter Bruch bei Umstieg von Python 2 auf Python 3:

- Modernisierung der Sprache und Befreiung von “Altlasten”
- Python-2-Programme sind in der Regel unter Python 3 erst nach einer Quelltextänderung lauffähig
- Versionen 2.6/2.7 wird noch gepflegt, in 2.7 sogar Rückportierungen von Python 3 enthalten

**Hinweis:** Wir verwenden Version 3.9.2 (`python3 -V`) und höher.

# Programmiersprache Python

## Einsatzgebiete I

- Breites Spektrum da umfangreiche Standardbibliotheken sowie Vielzahl freier Zusatz-Module
  → **Gefahr** der "Block-Box-Denkweise"
  - NumPy, SciPy (große multi-dimensionale Felder/Matrizen, "high-level"Mathefunktionen)
  - MathPlotLib, PyPlot, Animation (Graphische Darstellung, Diagramme)
  - TensorFlow (Maschinen- und Deep-Learning Algorithmen)
  - StatsModels (statische Auswertungen)
  - random (Zufallszahlen)

# Programmiersprache Python

## Einsatzgebiete II

- Automatische Speicherverwaltung inkl. -bereinigung und Ausnahmebehandlung (exception handling) $\rightarrow$ robuste Programmiersprache
- Erweiterbar durch reine Python-Module und z.B. einbettbar durch CPython in c/c++-Programme
- Steuerung von Programmen, wie LibreOffice, GIMP, ... durch Python-Skripte

# Programmiersprache Python

## Literatur – Bücher (engl. downloadbar)

1. A. Downey *"Think Python – How to Think Like a Computer Scientist"* Green Tea Press, Needham/Massachusetts, 2015
   (PDF downloadbar `www.thinkpython2.com` *im Uninetz*)
2. H. P. Langtangen *"A Primer on Scientific Programming with Python"* Springer, Dordrecht, 2009, ISBN 978-3-642-02474-0, **DOI 10.1007/978-3-642-02475-7**
   (PDF downloadbar *im Uninetz*)

## Literatur – Online Tutorials

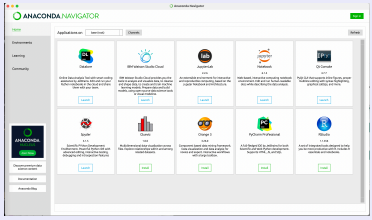
- Python Software Foundation, *"python"* 2001–2019, `https://www.python.org`
- Bernd Klein *"Python Kurse – Python 3 Tutorial"*, 2011–2018, `https://www.python-kurs.eu/python3_kurs.php`

# Python – Editoren & Entwicklungsumgebungen

## Komfortabler: Entwicklungsumgebungen

Enthalten einen umfangreichen Satz an Standard-bibliotheken, automatische Erkennung von Funktionen, Wortverlängerung, . . . und (meist) ein interaktives Python-Terminal zum Testen.

**Beispiele:** Mu, Thonny, PyCharm, **Spyder**, **Jupyter-Notebooks** , Eric, . . .




## Anaconda: Spyder, Jupyter-Notebooks

Python-Distribution Anaconda enthält Sypder und Jupyter-Notebooks.
Ist für alle Plattformen/ Betriebssysteme verfügbar.

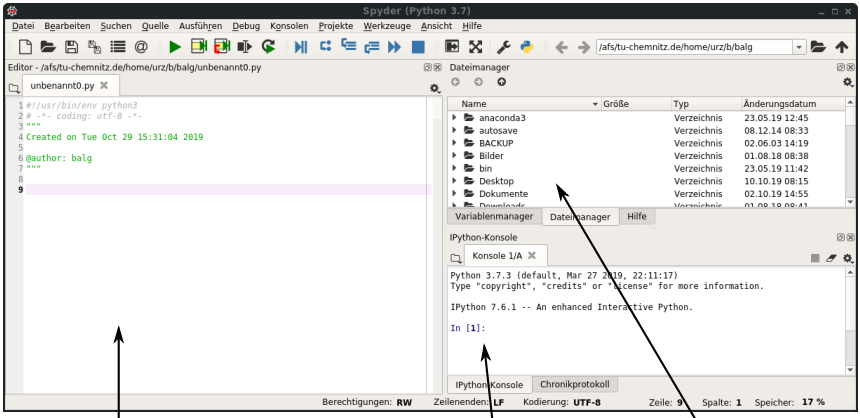
Zugriff auf URZ-Administrierten PCs via `/afs/tu-chemnitz.de/global/capp/anaconda/`

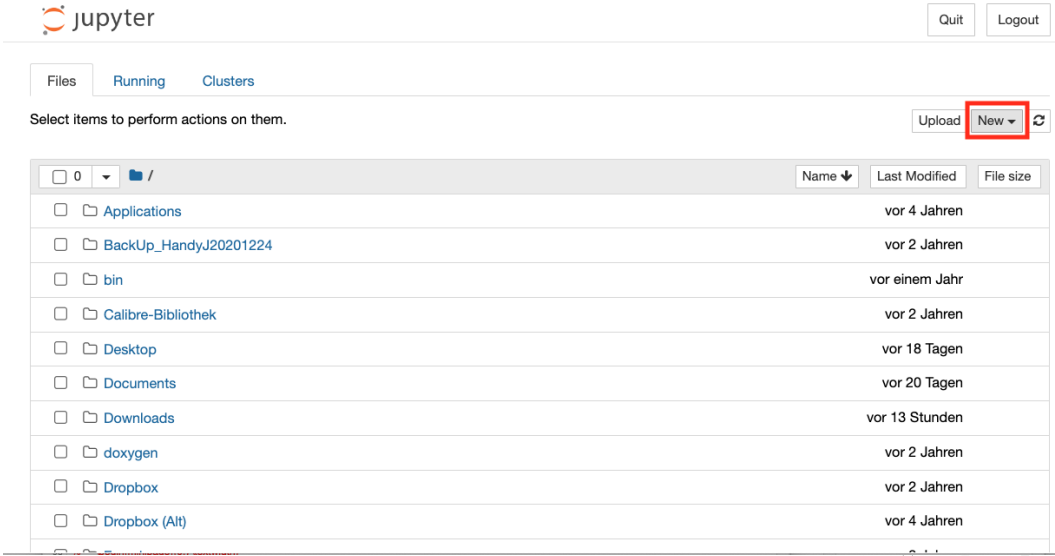
## Vereinfachung für Aufruf von Programmen

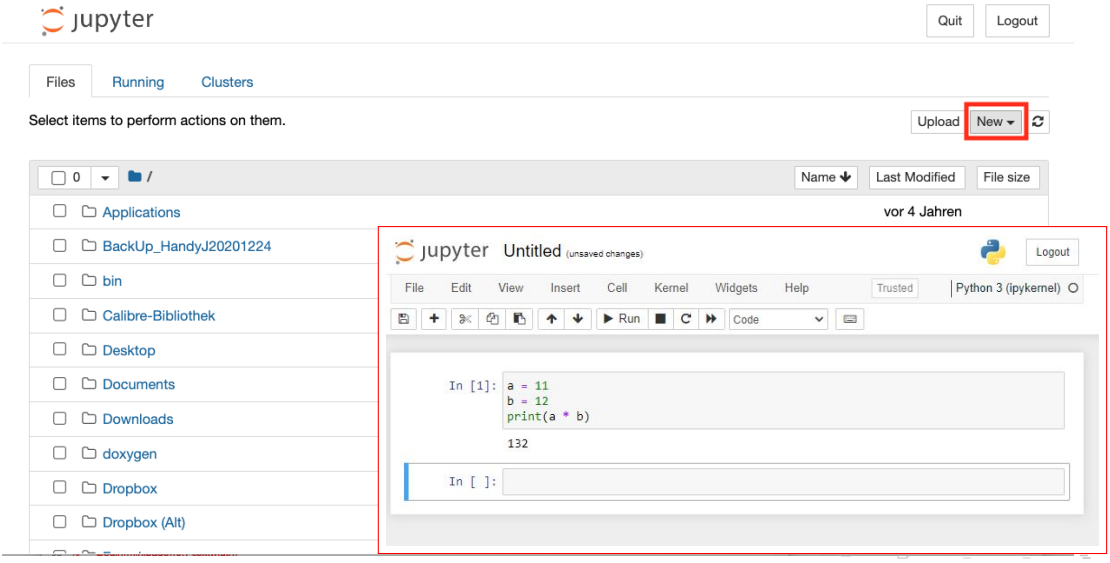
- Einrichten eines Anaconda-Profils im eigenen Home-Verzeichnis

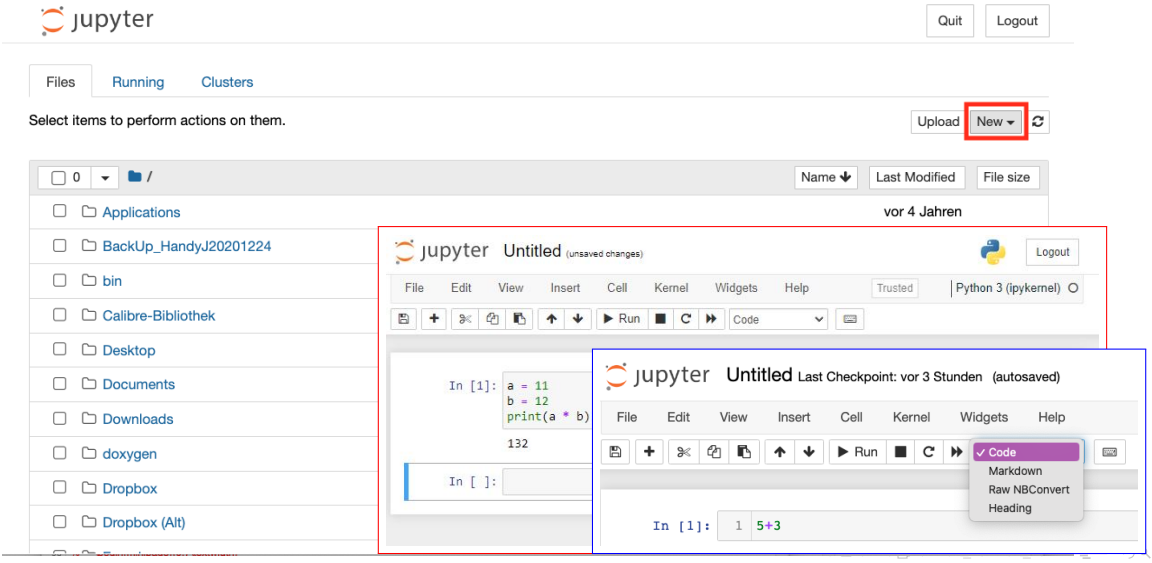
```
. /afs/tu-chemnitz.de/global/capp/anaconda/etc/profile.d/conda.sh &&
    conda activate base
sypder &
jupyter-notebook &
```

- Dauerhafte Vereinfachung für den Aufruf über die Konsole
  → Alias für Funktionsaufruf definieren (wirkt sich nur in der Konsole aus, einmaliger Vorgang!)
  1. Öffne `~/.bashrc`
  2. Füge hinzu: `alias anaconda=". <Verzeichnis> && conda activate base"`
  3. Starte Konsole neu.
- Führe nun erst `anaconda` dann nach Wahl `spyder&` oder `jupyter-notebook&` aus


Spyder (Python 3.7)
Datei Bearbeiten Suchen Quelle Ausführen Debug Konsolen Projekte Werkzeuge Ansicht Hilfe
/afs/tu-chemnitz.de/home/urz/b/balg
Editor - /afs/tu-chemnitz.de/home/urz/b/balg/unbenannt0.py
unbenannt0.py
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Oct 29 15:31:04 2019
@author: balg
"""
Dateimanager
Name
Größe
Typ
Änderungsdatum
anaconda3 Verzeichnis 23.05.19 12:45
autosave Verzeichnis 08.12.14 08:33
BACKUP Verzeichnis 02.06.03 14:19
Bilder Verzeichnis 01.08.18 08:38
bin Verzeichnis 23.05.19 11:42
Desktop Verzeichnis 10.10.19 08:15
Dokumente Verzeichnis 02.10.19 14:55
Variablenmanager
Dateimanager
Hilfe
IPython-Konsole
Konsole 1/A
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
Type "copyright", "credits" or "license" for more information.
IPython 7.6.1 -- An enhanced Interactive Python.
In [1]:
IPython-Konsole
Chronikprotokoll
Berechtigungen: RW
Zeilenenden: LF
Kodierung: UTF-8
Zeile: 9
Spalte: 1
Speicher: 17 %
Python-Editor
Interaktive Python-Konsole
Hilfsfenster (Datei- oder Variablen-manager, Hilf)


jupyter
Quit
Logout
Files
Running
Clusters
Select items to perform actions on them.
Upload
New
0
/
Name
Last Modified
File size
Applications
vor 4 Jahren
BackUp_HandyJ20201224
vor 2 Jahren
bin
vor einem Jahr
Calibre-Bibliothek
vor 2 Jahren
Desktop
vor 18 Tagen
Documents
vor 20 Tagen
Downloads
vor 13 Stunden
doxygen
vor 2 Jahren
Dropbox
vor 2 Jahren
Dropbox (Alt)
vor 4 Jahren


jupyter
Quit
Logout
Files
Running
Clusters
Select items to perform actions on them.
Upload
New
0
/
Name
Last Modified
File size
Applications
vor 4 Jahren
BackUp_HandyJ20201224
bin
Calibre-Bibliothek
Desktop
Documents
Downloads
doxygen
Dropbox
Dropbox (Alt)
jupyter
Untitled
(unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3 (ipykernel)
Run
Code
In [1]:
a = 11
b = 12
print(a * b)
132
In [ ]:


jupyter
Quit
Logout
Files
Running
Clusters
Select items to perform actions on them.
Upload
New
0
/
Name
Last Modified
File size
Applications
vor 4 Jahren
BackUp_HandyJ20201224
bin
Calibre-Bibliothek
Desktop
Documents
Downloads
doxygen
Dropbox
Dropbox (Alt)
jupyter Untitled (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code
In [1]:
a = 11
b = 12
print(a * b)
132
In [ ]:
jupyter Untitled Last Checkpoint: vor 3 Stunden (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Run
Code
Markdown
Raw NBConvert
Heading
In [1]:
1 5+3

# Einführung zur Programmiersprache

## Digitale Funktionsweise

- Informationsdarstellung mit diskreten Zuständen
- Elementare Informationseinheit: 1 Bit $\rightarrow$ 0, 1

## Eigenschaften einer Sprache

- **Lexik:** Vorschrift zur Bildung von Wörtern
- Syntax: Vorschrift zur Bildung von Sätzen
- Semantik: Bedeutung ("Was ist gemeint")

# Einführung zu Python

## Elementare Daten und Anweisungen

- Wertzuweisung:
  `diff = temp2 - temp1`
- Eingabe: `b = input("b?␣")`
- Ausgabe: `print("b␣", b)`
- Daten
  - Datentyp: Definiert
    - Speicherplatzbelegung
    - Wertevorrat
    - zulässige Operationen
  - Standard-Datentypen:
    - `int` - ganze Zahlen
    - `float` - reelle Zahlen
    - `char` - Textzeichen

## Lexikalische Einheiten in Python

1. Schlüsselworte: `if, else, ...`
2. vordefinierte Datentypen: `int, ...`
3. Bezeichner (Variablen)
4. Sonderzeichen: `; , : . + -` ...
5. Trennzeichen:
   Leerzeichen, Tabulatoren, Zeilenschaltung
6. Zahlenwerte: dezimal, binär, ...
7. Textzeichen: `'a', '#', '\n'`
8. Zeichenkette: `"Test 1\n"`
9. Kommentare: # bis Zeilenende

# Schlüsselwörter und Sonderzeichen

## Schlüsselwörter

```
and, as, assert, break, class, continue, def, del, elif, else,
except, False, finally, for, from, global, if, import, in, is,
lambda, None, nonlocal, not, or, pass, raise, return, True,
try, while, with, yield
```

## Besondere Zeichen

```
_ __ @ , ; : # $
' ''' " """
( ) { } [ ]
```

## (Erweiterte) Zuweisung

```
=
+= -= *= /= **= //= %=
|=  ^=  &= <<= =>> @=
```

## (Rechen-, Bit-, Vergleichs-) Operatoren

```
+ - * ** / // %
| ^ & << >> ~
== < > != <= >=
```

## Verwendung des interaktiven Python-Terminals

Starten Sie ein interaktives Python-Terminal (python3):

- im Terminal (siehe F. 28) oder
- in spyder (siehe F. 50) oder
- nutze Jupyter-Notebook (siehe F. 51 ff.)

## Aufgabe – “Hello World” und “Zen of Python”

Geben Sie “Hello World” mit Python aus.

Lassen sich das “Zen of Python” ausgeben: `import this`

## Zen of Python

Beautiful is better than ugly. Explicit is better than implicit.
**Simple is better than complex.** **Complex is better than complicated.**
Flat is better than nested. Sparse is better than dense.
**Readability counts.**
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently. Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
**There should be one—and preferably only one—obvious way to do it.**
Although that way may not be obvious at first unless you're Dutch.
Now is better than never. Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
**If the implementation is easy to explain, it may be a good idea.**
Namespaces are one honking great idea—let's do more of those!

# Bezeichner

## Namensgebung in Python

Ein Bezeichner ist ein Name um Variablen, Funktionen, Klassen, Module oder andere Objekte **eindeutig** zu benennen.

- Benutzbare Zeichen:
  - Groß- und Kleinbuchstaben von A,a bis Z,z
  - Unterstrich _
  - Zahlen 0 bis 9 (aber nicht an erster Stelle)
- Ausnahmen - Python Schlüsselwörter
- Namenskonvention für Bezeichner sinnvoll
  z.B. maximum_height, MaximumHeight (CamelCase-Schreibweise), HeightMax, ...

# Python-Umgebung

## Aufgabe – Operatoren

Verwenden Sie Python (interaktives Terminal) als Taschenrechner und berechnen Sie zunächst 2+2.

Was bewirken die folgenden Rechenoperatoren? `+` `-` `*` `**` `/` `//` `%`

Was erhalten Sie bei den folgenden Vergleichsoperatoren? `4>0.37`, `5==2`

Welche Vorrangregeln gibt es bei Operatoren? `-t**2*g/2`, `-(t**2)*(g/2)`, `-t**(2*g)/2`
`a/b+c+a*c`, `a/(b+c)+a*c`, `a/(b+c+a)*c`

**Hinweis:** Operatoren sind Symbole die Operationen auf Variablen und Werte anwenden.

## Allgemeine Benutzung

- Standard Kodierung ist UTF-8 (Windows: `latin1` = ISO-8859-1)
- Groß- und Kleinschreibung wird unterschieden (case-sensitive)

# Daten- und Objekttypen

## Daten- und Objekttypen

Datentypen definieren Speicherplatzbelegung, Wertevorrat und zulässige Operationen, d.h. über Datentypen wird festgelegt, wie die Bits im Speicher interpretiert werden.

Unter Datentypen werden meist die “Standard-Datentypen” (mit kleinen Variationen) und unter Objekttypen komplexere, meist zusammengesetzte Datentypen

### Standard Daten- und Objekttypen enthalten in Python3

- None : leeres Objekt
- Zahlen
- Sequenzen / Folgen (Zeichenketten, Listen, Bytes, . . . )
- Mengen
- Klassen, Klassen-Typen, Spezial-Objekte, . . .
- Abbildungen (Mappings)
- Rufbare Typen (Funktionen, Methoden, . . . )
- Module
- . . .

**Hinweis:** Liste ist nicht vollständig.

# Daten- und Objekttypen

## Zahlen

| | |
|---|---|
| `int` | : Ganze Zahl (unbegrenzte Genauigkeit, in Python2 `long`, 8 Byte) |
| `float` | : Reelle bzw. Gleitkommazahl (32 Byte) |
| `decimal.Decimal` | : Reelle Zahl ohne Rundungsgenauigkeit |
| `fraction.Fraction` | : Rational Zahlen (Brüche) |
| `complex` | : Komplexe Zahlen |
| `bool` | : Boolean (Wahrheitswert, 1 Byte) – Unterklasse von `int` mit `False`: Falsch, 0; `True`: Wahr, 1 |

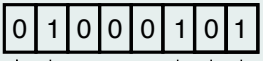

## Byte-Strings und Zeichenketten

| | |
|---|---|
| `str` | : Unveränderbar Zeichenkette |
| `bytes` | : Unveränderbar Byte-String (Python3) |
| `bytearray` | : Veränderbarer Byte-String (Python3) |

# Daten- und Objekttypen

## Beispiel: Wie die gleiche Bitreihenfolge zu verschiedenen Ergebnissen führt.

Bedeutung der Bits




`int` (32 Bits)

zahl=0*(-1)+1*$2^{30}$+0*$2^{29}$+...+1*$2^2$+0*$2^1$+1*$2^0$
=1073741829

`float` (32 Bits)

-1 $2^7$ $2^0$ $2^{22}$ $2^1$ $2^0$
Exponent Matisse

zahl=0*(-1)*(0*$2^{22}$+...+1*$2^2$+0*$2^1$+1*$2^0$)*$10^{1*2**7+...+0*2**0}$
= 5*$10^{128}$

`char` (8 Bits)

ASCII-Code

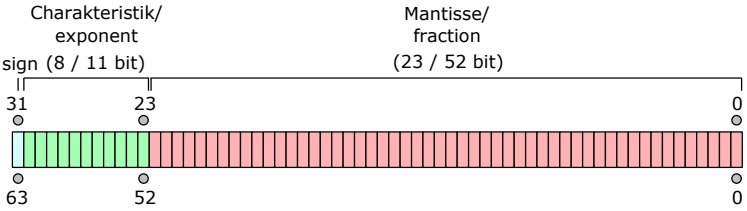
zeichen=E

*Bemerkung:* Speicherplatzbelegung entspricht der Bitreihenfolge und die Umrechnung zur Zahl dem zugehörigen Wertevorrat.

# Fehler durch Zahlendarstellung

IEEE Standard 754-2008 für `float`/ `double`

Früher: `float` vielgestaltig: $\pm$ Mantise $\cdot \beta^{\text{exp}}$ mit $\beta = \{2, 8, 10, 16\}$
Heute: IEEE-Standard (IEEE : Institute of Electrical and Electronics Engineers)




$$\mathbf{x} = \mathbf{s} \cdot \mathbf{m} \cdot \mathbf{b}^{\mathbf{e}} = (-1)^{\mathbf{S}} \cdot (1 + \sum_{\mathbf{i}=1}^{\mathbf{p}} \mathbf{M}_{\mathbf{i}} / 2^{\mathbf{i}}) \cdot 2^{\mathbf{E}-\mathbf{B}}$$

| | |
|---|---|
| $\mathbf{s} \in \{-1, 1\}$ | : Vorzeichen |
| $\mathbf{m}$ | : Mantisse |
| $\mathbf{b} = 2$ | : Basis (Standard) |
| $\mathbf{e}_{32} \in [-126, 127]$ | : Exponent, B$= 127$ |
| $\mathbf{e}_{64} \in [-1022, 1023]$ | : Exponent, B$= 1023$ |

# IEEE Standard 754-2008 für `float`/ `double`

`float` – 32 Bit: $\mathbf{x} \in [-3.4 \cdot 10^{38}, -1.2 \cdot 10^{-38}] \cup [1.2 \cdot 10^{-38}, 3.4 \cdot 10^{38}]$

`double` – 64 Bit: $\mathbf{x} \in [-1.8 \cdot 10^{308}, -2.2 \cdot 10^{-308}] \cup [2.2 \cdot 10^{-308}, 1.8 \cdot 10^{308}]$

## Sonderfälle:

- $\mathbf{E} = 0, \mathbf{M} = 0 \rightarrow$ 0 (Null)
- $\mathbf{E} = \mathbf{E}_{\text{max}} = \mathbf{B}, \mathbf{M} = 0 \rightarrow$ Unendlich (Inf für “infinity”)
- $\mathbf{E} = \mathbf{E}_{\text{max}} = \mathbf{B}, \mathbf{M} > 0 \rightarrow$ keine Zahl (NaN für “not a number”) $\rightarrow 0/0, (\pm\text{Inf})/(\pm\text{Inf}), 0 \cdot (\pm\text{Inf})$

*Bemerkung:* Für weitere Infos siehe *Wikipedia – IEEE Standard 754 (engl.)*

# Konsequenzen der Zahlendarstellung

## Folgerung 1:

Nicht alle Zahlen lassen sich exakt durch die Maschinenrepresentation $\mathrm{fl}(\cdot)$ darstellen.
Wenn Zahl z durch $\mathrm{fl}(z)$ angenähert dann:

- $|z| < 2^{e_{min}}(e_{min} = -B + 1) \Rightarrow \mathrm{fl}(z) = 0$
- $|z| > 2^{e_{max}}(e_{max} = B) \Rightarrow$ Zahl nicht darstellbar oder “overflow”
- sonst: $\mathrm{fl}(z) = z(1 + \epsilon)$ (Maschinengenauigkeit) mit $|\epsilon| < \epsilon_{Mach}$
- “Maschinen-Epsilon”:
  - $\epsilon_{mach}$ kleinste positive Zahl für die $\mathrm{fl}(1 + \epsilon) > 1$ gilt.
  - $\epsilon_{Mach} = (1+\text{Mantisse})2^0$ mit Mantisse$= 0 \ldots 01$ (23 / 52 Bits lang)
  - $\epsilon_{Mach} = \{2^{-23}, 2^{-52}\} = \{1/8 \cdot 10^{-6}, 1/4 \cdot 10^{-15}\}$

# Konsequenzen der Zahlendarstellung

## Folgerung 2:

Bei der Anwendung jeder Maschinenarithmetik $\oplus \in \{+, -, \cdot, /\}$ gilt:

$$\mathbf{fl}(\mathbf{a} \oplus \mathbf{b}) = (\mathbf{a} \oplus \mathbf{b})(1 + \epsilon)$$

mit $|\epsilon| \leq \epsilon_{\text{Mach}} \mathbf{h}$.

$\Rightarrow$ Maschinenarithmetik ist weder assoziativ noch distributiv.

# Besonderheit: Subtraktion = Stellenauslöschung

Bsp.-Rechnungen mit Mantisse der Länge 4

**Zahlen unterschiedlicher Größe:**

Mathematik

$$\begin{array}{rr} & .1000 \cdot 10^{2} \\ - & .1000 \cdot 10^{-3} \\ \hline & .9999 \cdot 10^{1} \end{array}$$

Maschinenarithmetik

$$\begin{array}{rr} & .1000 \cdot 10^{2} \\ - & .000001 \cdot 10^{2} \\ \hline & .1000 \cdot 10^{2} \end{array}$$

Effekt:
Folgerung 2 – Fehler durch Zahlendarstellung und Anwendung Maschinenarithmetik

**Zahlen fast gleicher Größe:**

$$\begin{array}{rr} & .1234 \cdot 10^{5} \\ - & .1233 \cdot 10^{5} \\ \hline & .0001 \cdot 10^{5} \end{array}$$

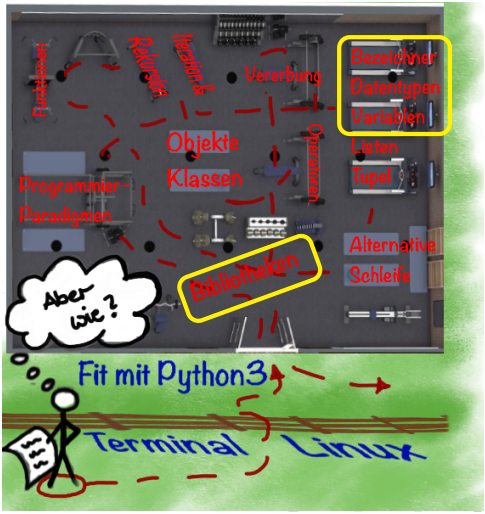
Interne Speicherung des Ergebnissen
$\rightarrow 0.1 \cdot 10^{2}$

Effekt:
Exaktes Ergebnis für diese Operation, **aber** – Fehler durch Zahlendarstellung und Anwendung Maschinenarithmetik

# 3. Einheit

## Ziel

- Programmiersprache Python (Lexik) II
- Programme
- Ausnahmebehandlung



# Beispiele für Fehler

## Aufgabe: Genauigkeitsfehler in python

Testen Sie folgende Beispiele in Python aus. Was beobachten Sie? Woran liegt diese Beobachtung?

$$0.1 + 0.1 + 0.1 == 0.3$$
$$\mathbf{sum}([0.1] * 10) == 1$$
$$\mathbf{import math; math.fsum}([0.1] * 10) == 1$$
$$\mathbf{round}(2.5) = ???$$
$$\mathbf{round}(3.5) = ???$$

*Hinweis:* Für alle komplexeren Mathematischen Funktionen benötigen Sie eine extra Bibliothek (z.B. `math`, `numpy`, `scipy`), die über `import` eingebunden werden müssen. (Details siehe Kap. "Bibliotheken und Module")

# Beispiele für Fehler

## Aufgabe: Genauigkeitsfehler in `python` – Lösung

Testen Sie folgende Beispiele in Python aus. Was beobachten Sie? Woran liegt diese Beobachtung?

$$0.1 + 0.1 + 0.1 == 0.3 \quad \rightarrow \text{ False}$$

$$\text{sum}([0.1] * 10) == 1 \quad \rightarrow \text{ False}$$

$$\text{math.fsum}([0.1] * 10) == 1 \quad \rightarrow \text{ True}$$

$$\text{round}(2.5) = 2$$

$$\text{round}(3.5) = 4$$

Implementierte Methode : “Banker’s rounding”

# Beispiele für Fehler

## Aufgabe: Stellenauslöschungseffekte

Testen Sie folgende Eingaben und vergleichen Sie die Ergebnisse. Was sind Ihre Schlussfolgerungen für sich selbst?

1. Für $\mathbf{x} = 1.2 * 10^{-5}$

$$\frac{1-\cos(\mathbf{x})}{\mathbf{x}^2} = \frac{1}{2}\left(\frac{\sin(\mathbf{x}/2)}{\mathbf{x}/2}\right)^2$$

2. **Zusatz:** Für $\mathbf{x} = 1.2345678910^{-8}$

$$\frac{\sqrt{1+\mathbf{x}}-1}{\mathbf{x}} = \frac{\sqrt{1+\mathbf{x}}-1}{\mathbf{x}}\frac{\sqrt{1+\mathbf{x}}+1}{\sqrt{1+\mathbf{x}}+1} = \frac{1+\mathbf{x}-1}{\mathbf{x}(\sqrt{1+\mathbf{x}}+1)} = \frac{1}{\sqrt{1+\mathbf{x}}+1}$$

$$\frac{\sqrt{1+\mathbf{x}}-1}{\mathbf{x}^2} = \frac{\sqrt{1+\mathbf{x}}-1}{\mathbf{x}^2}\frac{\sqrt{1+\mathbf{x}}+1}{\sqrt{1+\mathbf{x}}+1} = \frac{1+\mathbf{x}-1}{\mathbf{x}^2(\sqrt{1+\mathbf{x}}+1)} = \frac{1}{\mathbf{x}\,(\sqrt{1+\mathbf{x}}+1)}$$

# Beispiele für Fehler

## Aufgabe: Stellenauslöschungseffekte – Lösung

Testen Sie folgende Eingaben und vergleichen Sie die Ergebnisse. Was sind Ihre Schlussfolgerungen für sich selbst?

1. Für $\mathbf{x} = 1.2 * 10^{-5}$

$$0.499,999,732,974 = 0.499,999,999,994 \quad \text{exakter}$$

2. **Zusatz:**Für $\mathbf{x} = 1.2345678910^{-8}$

$$0.499,999,991,963,3465 = 0.499,999,998,456,7902 \quad \text{exakter}$$

$$40,499,999.71758106 = 40,500,000.24355002 \quad \text{exakter}$$

$\rightarrow$ Differenzen von Zahlen gleicher Größe vermeiden

Sinnvolle Eingabe von Formeln (besonders C/C++, Fortran, Java, ...) wichtig.

# Variablen

## Variablen in Python

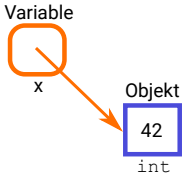
- Variablennamen sind Bezeichner
- Variablen enthalten nur Referenzen (Verweise, Zeiger) auf Objekte
- dynamische Typisierung: Datentyp ist an Objekt und nie an Variable gekoppelt
- Variablen werden nicht definiert, sondern über erste Referenzzuweisung erzeugt

→ Referenzzuweisung kann jederzeit geändert werden (d.h. Typänderung)

- Zuweisung erfolgt von Rechts nach Links: `x = 42`

Beispiel:

```
x = 42
# Zuweisung
```

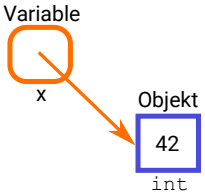


# Datentyp ↔ Variablen

## Verständnisbeispiel

```
x = 42
id(x)
type(x)
```

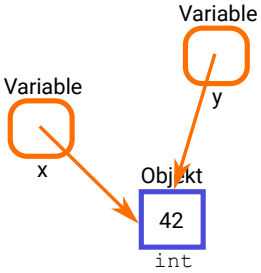
`id(x)` : Funktion zur Ermittlung der Referenz
`type(x)` : Funktion zur Ermittlung des Datentyps



# Datentyp ↔ Variablen

## Verständnisbeispiel

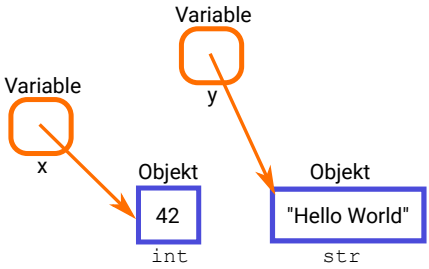
```
x = 42
id(x)
type(x)
y = 42F
id(y)
type(y)
```



# Datentyp ↔ Variablen

## Verständnisbeispiel

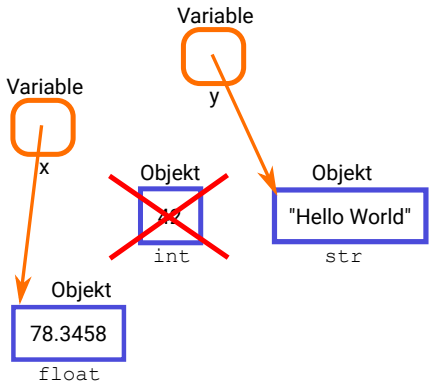
```
x = 42
id(x)
type(x)
y = 42
id(y)
type(y)
y = "Hello World"
id(y)
type(y)
```



# Datentyp ↔ Variablen

## Verständnisbeispiel

```
x = 42
id(x)
type(x)
y = 42
id(y)
type(y)
y = "Hello World"
id(y)
type(y)
x = 78.3458
id(x)
type(x)
```



# Gleichheit $\leftrightarrow$ Wahrheit

## Operatoren zum

`==, !=` : Test auf wertmäßige Gleichheit
`is, is not` : Test auf Identität:

## Beispiel

```
a = True
b = 1
print(a == b)
print(a != b)
print(a is b)
print(a is not b)
```

**Hinweis:** Beachte den Unterschied zu `a = b`

# Gleichheit $\leftrightarrow$ Wahrheit

## Operatoren zum

- `==, !=` : Test auf wertmäßige Gleichheit
- `is, is not` : Test auf Identität:

## Beispiel

```
a = True
b = 1
print(a == b)      # True
print(a != b)      # False
print(a is b)      # False
print(a is not b)  # True
```

**Hinweis:** Beachte den Unterschied zu `a = b`!

# Besondere Zahlen

## Wie kann man in Python ...

- eine Hexadezimal-, Oktal- oder Binärzahl in eine Variable eingeben?
- eine Integerzahl in Hexadezimal-, Oktal- oder Binärdarstellung ausgeben lassen?
- eine Gleitkommazahlen in Potenzdarstellung (z.B. $17.3 \times 10^2$) speichern?
- eine komplexe Zahlen speichern und darauf zugreifen?
- das vorangegangene Ergebnis in der Folgeeingabe nochmal verwenden?
- den Wert einer Variable `i` z.B. um 3 erhöhen?

# Besondere Zahlen

## Wie kann man in Python ...

- eine Hexadezimal-, Oktal- oder Binärzahl in eine Variable eingeben?

```
x = 0xB4
x = 0o11
x = 0b101011
```

- eine Integerzahl in Hexadezimal-, Oktal- oder Binärdarstellung ausgeben lassen?

```
hex(19)
oct(65)
bin(42)
```

- eine Gleitkommazahlen in Potenzdarstellung (z.B. $17.3 \times 10^2$) speichern?

```
3.14159
17.3e-02
17.3*10**-2
```

# Besondere Zahlen

## Wie kann man in Python ...

- eine komplexe Zahlen speichern und darauf zugreifen?

```
d = 4 + 1j
d.real
d.imag
```

- das vorangegangene Ergebnis in der Folgeeingabe nochmal verwenden?

```
78
_
_ + 5
```

- den Wert einer Variable `i` z.B. um 3 erhöhen?

```
i = 42
i = i + 3
i += 3
```

# Einführung zur Programmiersprache

## Eigenschaften einer Sprache

- Lexik: Vorschrift zur Bildung von Wörtern
- **Syntax:** Vorschrift zur Bildung von Sätzen
- Semantik: Bedeutung ("Was ist gemeint")

# Einführung zur Programmiersprache

## Darstellung von Syntax

- Nassi-Shneiderman-Diagramm:
  Struktogramm bei dem verschiedenen Programmierelemente über Strukturblöcke dargestellt werden, die auch ineinander geschachtelt werden können (DIN 66261)
- Flow-Chart:
  Programmablaufpläne (Flussdiagramm) bei dem einfache geometrische Objekte typische Programmierelemente widerspiegeln, die über Pfeile (flows) miteinander verbunden werden (DIN 66001)
- Pseudo-Code:
  Programmcode, der nicht zur maschinellen Interpretation, sondern lediglich zur Veranschaulichung eine Algorithms dient; orientiert sich meinst an höheren Programmiersprachen

# Einführung zur Programmiersprache

## Darstellung von Syntax

### Quellcode:

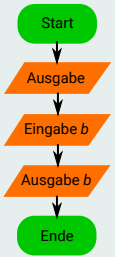
```
print("Hello World")
b = input("b? ")
print("Variable b ",b)
```

### Nassi-Shneidermann-Diagramm:

helloWorld.py

| Ausgabe: ”Hello World!” |
|---|
| Eingabe: b |
| Ausgabe: b |

### Flow-Chart:




### Pseudo-Code:

Ausgabe "Hello World"
Eingabe b
Ausgabe b

# Einführung in Python

## Programme in Python

- Textdatei, die die zu auszuführenden Befehle enthält (Quellcode)
- Dateiendung ist typischerweise `.py`
- In Python gibt es Formatierungsvorgaben für den Quellcode
- Ausführung im Terminal: `python3 <dateiname.py>`
- Ausgabe direkt in das Terminal (die Console)
- Ausführung im Terminal: `python3 -i <dateiname.py>` mit anschließendem interaktivem Terminal zur Weiterbearbeitung
- Bearbeitung, Speicherung in Entwicklungsumgebung mit integriertem Terminal (z.B. spyder)

# Einführung in Python

## Aufgabe – Erster Fehler

Schreiben Sie ein Programm `Fehler.py` für Python in dem Inhalt

```
myvar = 5.2
prinnt(Myvar)  # Funktionsname & Variablenname falsch
```

und führen Sie das Programm anschließend aus. Was passiert?

Wie können Sie mögliche Syntax-Fehler finden, bevor Sie das Programm ausführen?

Es wird immer nur der erste Fehler angezeigt!
Es können also immer noch mehr Fehler im Code versteckt sein.

## Formatierungsvorgaben in Python

- **Zen of Python:** Readability counts.
- Eine logische Zeile (Befehl) entspricht einer Zeile
- Endekennzeichen eines Befehls ist Zeilenwechsel
  **Hinweis:** Einfache Befehle können mit Semikolon ; getrennt in einer logischen Zeile stehen
- Backslash \ am Zeilenende sorgt für Fortsetzung der logischen Zeile über Zeilenende hinaus

```
i = 10 + 20 + 30 +\
 40
```

  **Hinweis:** Bei Konstruktion mit ( ), { }, [ ] kann der Backslash entfallen

```
print(10 + 20 + 30 +
 40)
```

- Blockbildung durch Einrückung (≙ Abbilden der Quellcode-Hierarchien)
  **Hinweis:** Indent (Einrücken) und Dedent (“Herausrücken”) sind Syntax-Elemente
- Kommentare beginnen mit # und gehen bis Zeilenende

# Nutereingaben in Python

## Nutzereingaben via `input()`

- `input("...")` interpretiert alle Eingaben als Zeichenketten
  **Hinweis:** In Python können Zeichenketten addiert werden, führt aber mathematisch nicht zu richtigen Ergebnissen.
- Typkonvertierung zum Einlesen von Zahlen notwendig: “Datentypfunktionen” wie `int(), float(),...`

```
var = input("Kommazahl: ")
f = float(var)
# oder in einem Schritt
var = int(input("Ganze Zahl: "))
```

**Hinweis:** Kommentare (#) sind Textelemente im Quellcode, die vom Interpreter/Compiler ignoriert werden. Sie dienen der Verständlichkeit des Quellcodes.

# Nutereingaben in Python

## Aufgabe - Mittlere Geschwindigkeit

velocity.py

Ein Auto fährt $t_1$ Sekunden mit der Geschwindigkeit $v_1 \frac{m}{s}$ und $t_2$ Sekunden mit $v_2 \frac{m}{s}$.

Schreiben Sie ein Programm, das die mittlere Geschwindigkeit $v$ berechnet.

- Erstellen Sie zunächst ein/e Struktogramm/ Flow-Chart/ Pseudo-Code.
- Schreiben Sie Ihren Quelltext (z.B. `velocity.py`) und kommentieren Sie diesen.
- Führen Sie Ihr Programm aus.
- Testen Sie verschiedene Eingabewerte. Nutzen Sie gegebenenfalls die Dateiumleitung mit einer Eingabedatei: `python3 velocity.py < velocity.in`

**Fragen:**

- Was sind sinnvolle Test-Eingabewert?
- Was passiert wenn man Texte eingibt oder $t_1 + t_2 = 0$ ist?

# Nutereingaben in Python

## Aufgabe - Mittlere Geschwindigkeit

velocity.py

Ein Auto fährt $t_1$ Sekunden mit der Geschwindigkeit $v_1 \frac{m}{s}$ und $t_2$ Sekunden mit $v_2 \frac{m}{s}$.

Schreiben Sie ein Programm, das die mittlere Geschwindigkeit $v$ berechnet.

**Fragen:**

- Was sind sinnvolle Test-Eingabewert?
  Zunächst Werte die man im Kopf überprüfen kann und “Grenzfälle”.
- Was passiert wenn man Texte eingibt oder $t_1 + t_2 = 0$ ist?
  Fehlerbehandlung notwendig

# Fehler- und Ausnahmebehandlung

## Fehler- und Ausnahmebehandlung

- Ausnahme (exception): Zustand der im Programm nicht vorgesehen ist.
- Ausnahmesituation (Fehler): Wenn Ausnahme während der Ausführung auftritt.
- Ausnahmebehandlung (exception handling): Verfahren um mit Fehlern automatisch während der Ausführungszeit umgehen zu können. Im Idealfall kann man den Fehler “reparieren” ansonsten wird das Programm typischerweise mit Fehlercode (Hinweis an den Nutzer) beendet.

# Fehler- und Ausnahmebehandlung

## Ausnahmebehandlung in Python

- Ausnahmebehandlung erfolgt durch Schlüsselwörter:
  - **`try`**-Block, d.h. `try:` auf den eingerückter Block mit Befehlen folgt
  - **`except`**-Block mit abzufangenden Fehler-Code, wobei mehrere `except`-Blöcke zu einem `try`-Block gehören können
    Auch diese werden jeweils eingerückt.
  - Erweiterbar durch **`else`**- und **`finally`**-Block
- Je nach Algorithmus kann es verschiedene Ausnahmen (Fehler-Codes) geben, z.B.
  - `ValueError` (Falscher Datentyp), `OSError` (Fehler bei systembezogenen Fehlern, z.B. Lesen/Schreiben von Dateien), `ZeroDivisionError`, ...
- Detaillierte Infos `docs.python.org/3/library/exceptions.html`

# Fehler- und Ausnahmebehandlung

## Ausnahmebehandlung in Python

```
try:
  <Zu testende Befehle>
except <FehlerCode>:
  <Fehlerausgabe1>
# except (<FehlerCode1>,<FehlerCode2>,...)
#    <Fehlerausgabe2>
# except:   ## Alternativ aus else denkbar
#    <Fehlerausgabe3>
#   raise   ## raise erzwingt hier Fehlerausgabe
# finally:
#   <Befehle zum Clean-Up>   ## werden zwingend ausgefÃ¼hrt
```

# Fehler- und Ausnahmebehandlung

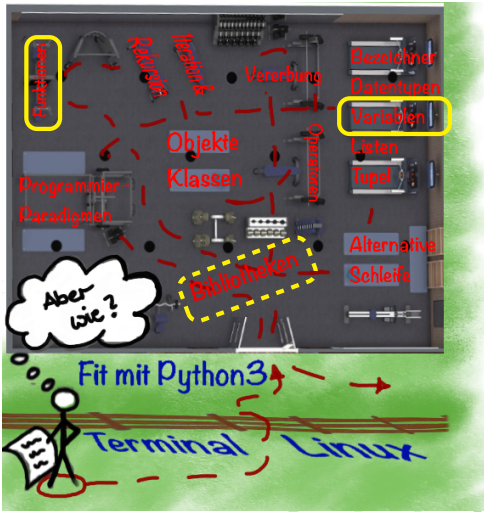
## Beispiel für Ausnahmebehandlung in Python

```
import sys     # fuer sys.exit()
try:
  x = float(input("Kommazahl: "))
  inverse = 1/x
except ValueError as e:     # e ist Alias fuer ValueError
  print("Fehler: Bitte Kommazahl eingeben.", e)     # e ist optional
  sys.exit()
except ZeroDivisionError as zde:
  print("Fehler: Division 0")
  sys.exit()
except:
  print("Unerwarteter Fehler ist aufgetreten.")
  raise     # raise erzwingt hier Fehlerausgabe
```

# 4. Einheit

## Ziel

- Ausnahmebehandlung II
- Bibliotheken
- Funktionen
- Sichtbarkeit von Variablen
- Docstrings



# Fehler- und Ausnahmebehandlung

## Aufgabe – Mittlere Geschwindigkeit mit Ausnahmebehandlung velocityError.py

Erweitern Sie Ihr Programm `velocity.py` so zu `velocityError.py`, dass Sie

- falsche Benutzereingaben für $t_1, t_2, v_1$ und $v_2$,
- Division durch 0 und
- unerwartete Fehler

abfangen.

In welchem Fall wäre es sinnvoll bei längeren Programm, dass Programm zu beenden? Welche Alternativen zum Beenden des Programms können Sie vorstellen?

## Aufgabe – Diskriminante

diskriminante.py

Schreiben Sie ein Programm, dass die Diskriminante

$$\mathtt{D} = \mathtt{b}^2 - 4\,\mathtt{a}\,\mathtt{c}$$

einer quadratischen Gleichung

$$\mathtt{a}\,\mathtt{x}^2 + \mathtt{b}\,\mathtt{x} + \mathtt{c} = 0$$

berechnet. Der Nutzer soll die Parameter a, b, c der Gleichung eingeben können. Beachten Sie hier die möglichen Fälle von Nutzerfehleingaben.
Anschließend soll die Gleichung mit eingesetzten Parametern und der Wert der Diskriminante ausgegeben werden.

**Hinweis:** Gehen Sie wie bei der Bearbeitung von `velocityError.py` vor.

# Bibliotheken und Module

## Module

Module ermöglichen die Aufteilung von Quelltext. Es gibt zwei Arten von Module:

- Bibliotheken (Libraries)
  Standardbibliothek, eigene Bibliotheken, Bibliotheken von Drittanbietern
- lokale Module
  Nur für ein Programm zugreifbar

Module sind typischerweise Extra-Dateien mit den entsprechenden Funktionen. Sie haben die Endung `.py` wenn sie in Python geschrieben sind.

## Einbinden von Funktionalitäten aus existierende Bibliotheken und Modulen

1. Die ganze Bibliothek einbinden:

```
import <bibliothek1>, <bibliothek2>, ...
<bibliothek>.<objekt>
```

```
import math
math.sqrt(2)
```

2. Die Bibliothek als Ganzes unter einem Alias einbinden:

```
import <bibliothek> as <alias>
<alias>.<objekt>
```

```
import math as m
m.sqrt(2)
```

3. Einzelne Objekte der Bibliothek laden:

```
from <bibliothek> import <objekt1>, <objekt2>
<objekt1>
```

```
from math import sqrt, exp
sqrt(2)
```

4. Alle Objekte der Bibliothek laden:

```
from <bibliothek> import *
<objekt>
```

```
from math import *
sqrt(2)
```

**Hinweis:** Beachte Eindeutigkeit der Module muss erhalten bleiben.

# Bibliotheken und Module

## Einbinden der sinnvollen Bibliothek

Beispiel: Verwendung der Wurzelfunktion

1.
```
from math import sqrt
sqrt(4)   # -> 2.0
sqrt(-1)  # -> Fehler
```

2.
```
from cmath import sqrt
sqrt(4)   # -> 2.0 + 0j
sqrt(-1)  # -> 1j
```

3.
```
from numpy.lib.scimath import sqrt
sqrt(4)   # -> 2.0
sqrt(-1)  # -> 1j
```

# Bibliotheken und Module

## Tipps im Umgang mit Bibliotheken

- Wo sind die Bibliotheken im Verzeichnis abgelegt? (interaktiver Modus)
  `<bibliothek>.__file__`

```
>>> import math
>>> math.__file__
```

- Welche built-in-Funktion sind im Module definiert? (interaktiver Modus)
  `dir(<bibliothek>)`

```
>>> import math
>>> dir(math)
```

# Bibliotheken und Module

## Aufgabe – Mathematische Funktionen

kugel.py

Ziel ist es den Umgang mit der `math`, `numpy` oder `scipy`-Bibliothek zu üben! Nutzen Sie also so viele Funktionen wie möglich.

Gegeben ist der Punkt $(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (1, 2, 3)$.

- Berechnen/ Bestimmen Sie für diesen Punkt die Kugel- und Zylinderkoordinaten.
- Nehmen wir an der Punkt liegt auf einer Kugeloberfläche und das Zentrum der Kugel befindet sich nicht im Koordinaten Ursprung, sondern bei $\vec{\mathbf{r}}_0 = (1, 0, 1)^{\mathrm{T}}$. Wie groß ist das Kugelvolumen der zugehörigen Kugel?

## Funktionen

- Gruppierung und “abkapseln” von logisch zusammenhängenden Anweisungen
- Deklaration definierter Schnittstellen zwischen mehreren Programmierern

## Verwendung von Funktionen: Vorteile – Nachteile

Vorteile:

- Verständlichkeit des Quelltextes erhöht
- Schreibaufwand verringert
- Fehler des Programms besser lokalisierbar
- Gruppenarbeit vereinfacht

Nachteile:

- Aufruf der Funktion notwendig
- zusätzlicher Schreibaufwand wegen Deklaration der Schnittstelle

**Zen of Python:** Readability counts.

# Funktionen

Beispiel aus der Mathematik: $f(x) : \mathbb{R} \to \mathbb{R},\ x \mapsto x^2$

## Syntax für Funktionen: Definition

- Funktionen beginnen mit Schlüsselwort **def** <funktionsname> (...)
- Alle Anweisungen innerhalb der Funktion (Funktionskörper) werden eingerückt.
- Funktionen können (optional) via **return** Objekte an die aufrufende Stelle zurückgeben.
- Definition: Angabe von Name, Parameterliste und "Inhalt"

```
def <funktionsname> (<Parameterliste>):
  <Anweisung(en)>
  return <value>   # Rueckgabewert optional
```

```
def fprint():
  print(x*x)

def f(x):
  return x*x
```

- **Wichtig:** Beachte Zugriffsrechte-/ Sichtbarkeit von Variablen

# Funktionen

## Syntax für Funktionen: Rückgabewert und Funktionsaufruf

- **return**-Anweisung = Rückgabewert:
  - Beendet Funktion und gibt den Wert `<value>` an die aufrufende Stelle zurück
  - Wird `<value>` nicht angegeben, dann wird `None` zurückgeben
  - Kann mehrfach an jeder beliebigen Stelle im Funktionskörper stehen
- Funktionsaufruf: Verwendung des Funktionsnamen und der zu übergebenden Informationen in Form von Variablen oder Konstantenwerten.

```
<funktionsname> (<Argumentenliste>)  # Aufruf
```

```
fprint()
f(3)
x=4.5
f(x)
```

- Funktionsaufruf kann erst **nach** Funktionsdefinition erfolgen, wobei Funktionsdefinition innerhalb einer anderen Funktion geschehen kann
- **Wichtig:** Reihenfolge der Parameter- und Argumentenliste muss identisch sein

# Funktionen

## Syntax für Funktionen: Parameterliste

- Parameterliste:
  - Keinen, einen oder mehrer Bezeichner, wobei zuerst obligatorische dann optionale Parameter aufgezählt werden
  - Optionale Parameter = Default-Parameter, werden in der Definition mit default-Werten definiert `<Parametername>=<default-value>` und müssen beim Aufruf nicht angegeben werden.

```
def f(x,a=1,b=2):
  return a*x*x+b*x
```

```
f(x)
f(x,2,1)
```

  - Schlüsselwortparameter: Wenn man bei mehreren optionalen Parametern nicht alle auf dem default-Wert lassen möchte, kann man beim Funktionsaufruf über `<Parametername>=<value>` diese direkt adressieren

```
f(x,b=1)
```

  - Beliebig viele Parameter können mit `*args` und Keyword-Parameter mit `**kwargs` übergeben werden (wichtig ist `*` oder `**`, `args`/`kwargs` sind frei wählbare Bezeichner)

# Funktionen

## Aufgabe – Ausgabe-Funktion

fprint.py

Schreiben Sie ein Funktion, die eine personalisierte “Hello World!”-Ausgabe nach dem Schema

“<Name> Hello World funktioniert!”

ermöglicht. Versuchen Sie die folgenden zwei Versionen:

1. Es werden keine Parameter übergeben und sowohl das Einlesen des Namens als auch die Ausgabe geschieht in der Funktion.
2. Es wird der Name als Parameter übergeben und nur die Ausgabe passiert in der Funktion.

**Hinweis:** Überlegen Sie sich jeweils sinnvolle Funktionsbezeichner.
**Hinweis:** Aufgabe dieser Funktion ist die Ausgabe auf die Konsole, daher wird kein Rückgabewert benötigt.

**Frage:** Ist hier eine Ausnahmebehandlung notwendig?

# Funktionen

## Aufgabe – Kelvin ↔ Grad Fahrenheit

umrechnungKF.py

Schreiben Sie jeweils eine Funktion mit der Sie Kelvin in Grad Fahrenheit und Grad Fahrenheit in Kelvin umrechnen können. Die Formel dazu lautet

$$T_F = (T_K - 273.15) * 9/5 + 32$$

Überlegen Sie sich sinnvolle Funktionsbezeichner, wie viele Übergabeparameter Sie benötigen und welchen Wert sie zurückgeben sollen.

## Aufgabe – Diskriminante II

diskriminante2.py

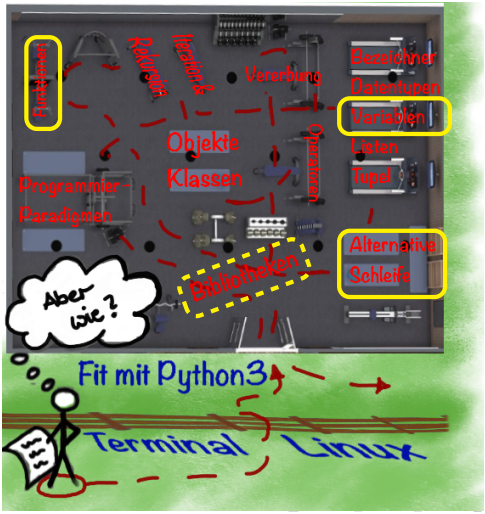
Schreiben Sie Ihr Programm `diskriminante.py` so um, dass (nur) die Berechnung der Diskriminante in einer eigenen Funktion durchgeführt wird.

Überlegen Sie sich sinnvolle Funktionsbezeichner, wie viele Übergabeparameter Sie benötigen und welchen Wert sie zurückgeben sollen.

# 5. Einheit

## Ziel

- Sichtbarkeit von Variablen
- Docstrings
- Ablaufsteuerung
  - Alternativen
  - Schleifen



# Sichtbarkeit von Variablen

## Globale Variablen

Variablen, die ausserhalb von Funktionen oder im globalen Zusammenhang definiert/ deklariert werden. Man kann von überall auf sie zugreifen.

## Beispiel – Globale Variable

VarGlobal.py

```
x = "global"

def test():
  print("x inside: ", x)

test()
print("x outside:", x)
```

# Sichtbarkeit von Variablen

## Globale Variablen

Variablen, die ausserhalb von Funktionen oder im globalen Zusammenhang definiert/ deklariert werden. Man kann von überall auf sie zugreifen.

## Beispiel – Globale Variable

VarGlobal.py

```
x = "global"

def test():
  print("x inside:", x)

test()
print("x outside:", x)
```

```
x inside: global
x outside: global
```

# Sichtbarkeit von Variablen

## Lokale Variablen

Variablen, die innerhalb einer Funktion oder eines Blockes definiert/ deklariert werden. Man auf sie nur innerhalb der Funktion/ des Block zugreifen.

## Beispiel – Lokale Variable

VarLokal.py

```
x = "global"

def test():
  x = "local"
  print("x inside: ", x)

test()
print("x outside:", x)
```

# Sichtbarkeit von Variablen

## Lokale Variablen

Variablen, die innerhalb einer Funktion oder eines Blockes definiert/ deklariert werden. Man auf sie nur innerhalb der Funktion/ des Block zugreifen.

## Beispiel – Lokale Variable

VarLokal.py

```
x = "global"

def test():
  x = "local"
  print("x inside: ", x)

test()
print("x outside:", x)
```

```
x inside: local
x outside: global
```

# Sichtbarkeit von Variablen

## Schlüsselwörter `global` und `nonlocal`

Um auf Variablen ausserhalb ihres Block zuzugreifen kann man sie mit den Schlüsselwörtern `global` und `nonlocal` erweitern und verknüpfen.

- `global`: Bindet den darauffolgenden Bezeichner an die zugehörige globale Variable
- `nonlocal`: Ermöglicht den schreibenden Zugriff auf die am nächsten höhergelegenen gleichnamige Variable innerhalb ineinander geschachtelter Funktionen (Namensuche erfolgt von inne nach außen)

## Sicheres Programmieren

Die Verwendung von globalen Variablen in Funktionen, Methoden, . . . sollte so weit wie möglich eingeschränkt werden um den Quellcode portable, flexibel und lesbar zu halten.

# Sichtbarkeit von Variablen

## Beispiel – Blockübergreifende Variablendefinition

VarBlock1.py–VarBlock3.py

```
def outer():
  x = "outer"
  def inner():
    x = "inner"
    print("inner:␣", x)

  inner()
  print("outer:␣", x)


x = "global"
outer()
print("global:␣", x)
```

```
def outer():
  x = "outer"
  def inner():
    global x
    x = "inner"
    print("inner:␣", x)

  inner()
  print("outer:␣", x)

x = "global"
outer()
print("global:␣", x)
```

```
def outer():
  x = "outer"
  def inner():
    nonlocal x
    x = "inner"
    print("inner:␣", x)

  inner()
  print("outer:␣", x)

x = "global"
outer()
print("global:␣", x)
```

# Sichtbarkeit von Variablen

## Beispiel – Blockübergreifende Variablendefinition

VarBlock4.py, VarBlock4a.py

```
 1  def outer():
 2    def inner():
 3      def innermost():
 4        nonlocal a, b
 5        print(a)
 6        print(b)
 7        a=40
 8        b=50
 9
10      a = 30
11      innermost()
12      print(a)
13      print(b)
14
15    a = 10
16    b = 20
17    inner()
18    print(a)
19    print(b)
20
21  outer()
```

# Sichtbarkeit von Variablen

## Beispiel – Blockübergreifende Variablendefinition

VarBlock4.py, VarBlock4a.py

```
 1 def outer():
 2   def inner():
 3     def innermost():
 4       nonlocal a, b
 5       print(a)  # Ausgabe: 30
 6       print(b)  # Ausgabe: 20
 7       a=40      # aendert inner() 'a'
 8       b=50      # aendert outer() 'b'
 9
10     a = 30
11     innermost()
12     print(a)    # Ausgabe: 40
13     print(b)    # Ausgabe: 50
14
15   a = 10
16   b = 20
17   inner()
18   print(a)    # Ausgabe: 10
19   print(b)    # Ausgabe: 50
20
21 outer()
```

# Variablen in Funktionen

## Take Home Message! – Guter Programmierstil!

Alle Variablen, die ihr in Funktionen verwendet, werden entweder

- als Parameter an die Funktion übergeben oder
- werden innerhalb der Funktion definiert.

Es wird nicht angenommen, dass die Variable “schon irgendwo” bekannt/ definiert ist!

# Docstring

## Dokumentation von Quellcode

Ziel ist es den Quellcode auch nach längere Zeit noch zu verstehen und anderen Programmieren das Lesen und Verstehen des Quellcodes zu vereinfachen.

## Dokumentation in Python

- Kommentare (#): Sind in der Quellcode-Datei für den Programmierer lesbar
- **Docstring**
  - Sind besondere Zeichenketten im Quellcode, die über bestimmte Befehle durch Python aufgerufen werden
  - Sollten zumind. in Paketen (thematisch zusammenpassende Module), allen Module, allen Funktionen und Klassen die von Modulen exportiert werden und allen öffentlichen Methoden sollte Docstrings haben.

# Docstring

## Docstring

- Definition:
  - `"""triple double quotes"""`,
  - `r"""raw triple double quotes"` (wenn Backslash enthalten),
  - `u"""unicode triple double quotes"` (für Unicode)
- Sie werden entsprechend ihres zugehörigen Blocks eingerückt
- Aufruf:
  - Über die `__doc__`-Methode des Objekts, d.h. hier `<funktionsname>.__doc__` (liefert eine Zeichenkette zurück)
  - Über die `help`-Funktion: `help(<funktionsname>)`
- Erste Anweisung in Modulen, Funktionen, Klassen oder Methoden
- Können überall in den Quelltext geschrieben um automatische Dokumentationen zu erstellen mittels Zusatzwerkzeugen wie Doxgen, PyDoc, Epydoc, ...

# Docstring

## Arten von Docstring

- One-line Docstrings:
  - Ein Satz mit Punkt der inkl. der An-/Ausführungszeichen eine Zeile lang ist.
  - Bestimmt die Funktion/ Methode/ Effect via Anweisung (“Do this”, “Return that”)
  - Keine Beschreibung
- Multi-line Docstrings:
  - Ausführliche mehrzeilige Beschreibung des Objekts, die mit One-line Docstring beginnt, dann Leerzeile und dann ausführlichere Beschreibung inkl. Parameterliste, Rückgabewert, etc.
  - Es existieren unterschiedliche Formatierungsstandards (Numpydoc, google, reST, Epytext (javadoc style), . . . )

# Docstring

## Aufgabe – Docstrings

Erweitere alle fünf geschriebenen Funktionen (Personalisiertes HelloWorld, Fahrenheit↔Kelvin, Diskriminante) um einen One-line Docstring und rufe diesen auf. Benutze jede Version des Aufrufs min. ein Mal.

fprintDoc.py, umrechnungKFDoc.py, diskriminante2Doc.py

# Ablaufsteuerung

## Ablaufsteuerung

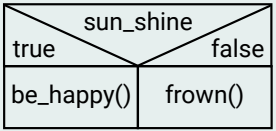
- Bedingte Ausführung – Alternative
  - Unvollständige Alternative
  - Vollständige Alternative
  - Mehrfachverzweigte Alternative
  - Bedingte Ausdrücke
- Wiederholte Ausführung
  - `while`-Schleife
  - `for`-Schleife

# Bedingte Ausführung

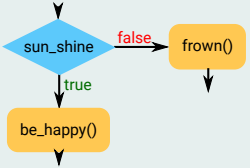
## Alternative – if-else

- Entsprechend der Auswertung eines Wahrheitswertes (boolean) werden verschieden Programmzweige ausgeführt, die gleichrangig nebeneinander stehen
- Allgemeine Formulierung: Wenn ..., dann ..., ansonsten ...
- Beispiel: **Wenn "☼?" wahr ist, dann ☺, ansonsten halt ☹.**

### Struktogramm




### Flow-Chart




### Pseudocode

```
if sun_shining
  be_happy()
else
  frown()
```

# Bedingte Ausführung

## Alternative – if-else

- Definition:

```
if <ausdruck1>:
    <anweisungen>
elif <ausdruck2>:
    <anweisungen>
elif <ausdruck3>:
...
else:
    <anweisungen>
```

Unvollständig: nur `if`
Vollständig: `if-else`
Mehrfachverzweigt: `if-elif-else`
(es gibt kein `switch-case`-Scenario in Python)

- Bedingte Ausdrücke = Kurzschreibweise (ternärer Operator):
  `<anweisung1> if <ausdruck> else <anweisung2>`

# Bedingte Ausführung – <ausdruck>

## Logische Operatoren (boolesche Arithmetik)

```
and   falsch, wenn eine Seite falsch
or    wahr, wenn eine Seite wahr
not   negieren
```

## Beispiel

```
a = True        ==> a hat den Wert?
b = not a       ==> b hat den Wert?
c = a and b     ==> c hat den Wert?
d = a or c      ==> d hat den Wert?
```

## Operatoren (Vergleiche)

```
< > == != >= <=
```

## Beispiel

```
n = 2; m = 3
a =  (n <  m)  ==> a?
a =  (n == m)  ==> a?
a =  (n != m)  ==> a?
a = !(n == m)  ==> a?
```

- short-circuit evaluation: Bei **and** und **or** wird einer der Operanden zurückgeliefert und die Auswertung abgebrochen, sobald das Ergebnis klar ist

# Bedingte Ausführung – <ausdruck>

## Logische Operatoren (boolesche Arithmetik)

```
and   falsch, wenn eine Seite falsch
or    wahr, wenn eine Seite wahr
not   negieren
```

## Beispiel

```
a = True          ==> True
b = not a         ==> False
c = a and b       ==> False
d = a or c        ==> True
```

## Operatoren (Vergleiche)

```
< > == != >= <=
```

## Beispiel

```
n = 2; m = 3
a =  (n <  m)  ==> True
a =  (n == m)  ==> False
a =  (n != m)  ==> True
a = !(n == m)  ==> True
```

- short-circuit evaluation: Bei **and** und **or** wird einer der Operanden zurückgeliefert und die Auswertung abgebrochen, sobald das Ergebnis klar ist

# Bedingte Ausführung

## Hinweis: Umgang mit Antworten auf Ja/Nein-Fragen

1. Die Antwort in einer eigenen Variable abspeichern und dann den gespeicherten Wert im `if-else` überprüfen.
2. Die Ja/Nein-Frage direkt als Bedingung einsetzen

Version 1:

```
n = 2
m = 3
a = (n < m)
if a:
    n=m
```

Version 2:

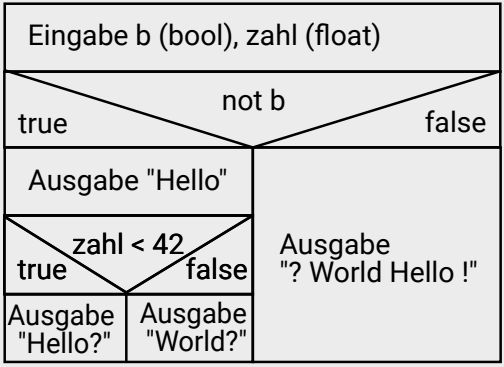
```
n = 2
m = 3

if n < m:
    n=m
```

# Bedingte Ausführung

## Aufgabe – Hello World

alternative.py

Programmieren Sie diese `if-else`-Version von “Hello World”.



# Bedingte Ausführung

## Beachte: Nutzereingabe von booleschen Werten

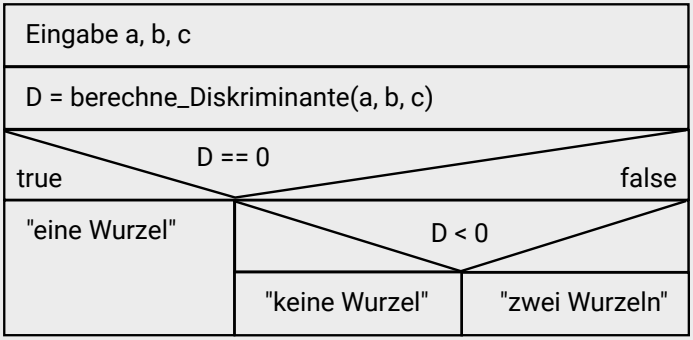
- Bei Umkonvertierung einer Zeichenkette in `bool` werden die Zeichenketten alphanumerisch ausgewertet und sind $\neq 0$ außer die leere Zeichenkette
- Besser `bool(int(input(...)))`, dann wird "0" $\rightarrow$ 0 $\rightarrow$ `False`

# Bedingte Ausführung

## Aufgabe – Diskriminante III

diskriminante3.py

Erweitern Sie Ihr Programm zur Berechnung der Diskriminante um folgende Auswertung.



# Bedingte Ausführung

## Aufgabe – Quadratische Gleichung

quadratischeGleichung.py

Schreiben Sie ein Programm, dass die Lösung einer Gleichung

$$\mathrm{a}x^2 + \mathrm{b}x + \mathrm{c} = 0$$

für *ALLE* möglichen Werte von $\mathrm{a}, \mathrm{b}, \mathrm{c}$ berechnet. Der Nutzer soll die Parameter $\mathrm{a}, \mathrm{b}, \mathrm{c}$ der Gleichung eingeben können. Anschließend soll die Gleichung mit eingesetzten Parametern und die Lösungsmenge ausgegeben werden.

---

Beispiel:

```
Eingabe von a,b,c: 4 [enter] 2 [enter] 0.25 [enter]
4x^2+2x+0.25=0
==> x={-0.25}
```

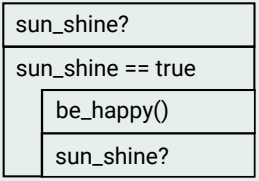
# Wiederholte Ausführung

## Schleifen

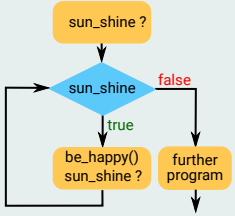
- Entsprechend einer Lauf- oder Abbruchbedingung wird der Schleifenkörper wiederholt ausgeführt
- 3 wesentliche Aspekte:
  - Initialisierung: Wie beginnt die Wiederholung
  - Schleifenkörper: Was wird wiederholt
  - Lauf-/Abbruchbedingung: Kriterium, wann Wiederholung endet
- Beispiel: **Wie ist das Wetter? Solange wie "☼?" wahr ist, dann ☺ und teste Wetter.**

# Wiederholte Ausführung

## Struktogramm




## Flow-Chart




## Pseudocode

```
sun_shining?
while(sun_shining == true)
  be_happy()
  sun_shining?
```

## Typische Arten

1. Abweis-Schleife: **Lauf**bedingung am Anfang — while
2. Nichtabweis-Schleife: **Lauf**bedingung am Ende — do-while
3. Zählschleife — for
4. Schleife mit **Abbruch**bedingung im Schleifenkörper — while-if, do-while-if

# Wiederholte Ausführung

## Initialisierung

```
i=0
```

## Schleifenkörper

```
1 i=i+1
2 print(i)
3 i=i-2
```

## Aufgabe

Welche Ausgabe erscheint, wenn Sie den Schleifenkörper wiederholt ausführen?

## Schleifentypen in Python

- `while`-Schleife
- `while-if`-Schleife
- `for`-Schleife

# Wiederholte Ausführung

## while-Schleife

Definition:

```
<Initialisierung der Laufbedingung>
while <Laufbedingung>:
    <Anweisungen>
    <Weiterfuehrung der Laufbedingung>
```

Beispiel:

```
i=0
while i <= 0:
    i=i+1
    print(i)
    i=i-2
```

# Wiederholte Ausführung

## while-if-Schleife

Definition:

```
<Initialisierung der Laufbedingung>
while <Laufbedingung>:
    <Anweisungen>
    if <Bedingung>:
        <AnweisungenIf>
        break   #bricht die Schleife ab
    <Anweisungen>
    <Weiterfuehrung der Laufbedingung>
else:       #optional nur in Python
    <Anweisungen>
```

Beispiel:

```
i=0
while i <= 0:
    i=i+2    #i=i+1
    if i > 3:
        print("if break:", i)
        break
    print(i)
    i=i-1    #i=i-2
else:
    print("else test")
```

Schlüsselwörter:

- **break**: Bricht die Schleife ab (vollständig, else-Zweig inbegriffen)
- **continue**: Beendet den aktuellen Schleifendurchlauf

# Wiederholte Ausführung

## for-Schleife (Zählschleife)

Definition:

```
for <Laufvariable> in <Sequenz>:
    <Anweisungen>
```

Beispiel:

```
for i in range(6):
    print(i)
```

- Schlüsselwörter `break`, `continue` funktionieren auch in der Zählschleife
- <Sequenz> können Listen, Tupel, Zeichenketten, etc. sein

# Wiederholte Ausführung

## Hinweis

Mit jeder Schleifenart kann jede Art an Schleife programmiert werden!

## Aufgabe – Schleifentest

schleifentest.py

Geben Sie die natürlichen Zahlen von 1 bis einschließlich 1234 aus.
Probieren Sie *alle* 3 Schleifenarten.

## Aufgabe – Eingabetest

eingabetest.py

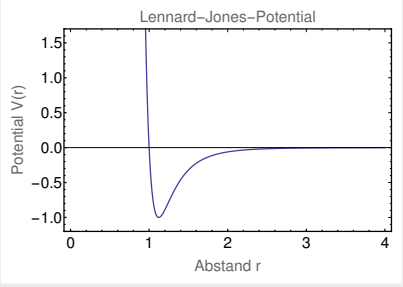
Schreiben Sie ein Programm, bei dem Sie solange ganzzahlige Werte eingeben können wie diese durch 3 teilbar sind. Ist eine Eingabe nicht durch 3 teilbar, soll das Programm mit einer entsprechenden Information beendet werden.

**Zusatz:** Behandeln Sie auch Fehleingaben wie Zeichenketten und Gleitkommazahlen.

## Aufgabe – Lennard-Jones-Potential

Das Lennard-Jones-Potential ist ein einfaches zur Beschreibung von Wechselwirkungskräften zwischen zwei Atomen. Das Lennard-Jones-Potential ist wie folgt definiert:

$$\mathrm{V}(\mathrm{r}) = 4\,\epsilon\,\left\{\left(\frac{\sigma}{\mathrm{r}}\right)^{12} - \left(\frac{\sigma}{\mathrm{r}}\right)^{6}\right\},$$




wobei der der erste Term die Pauli-Abstoßung und der zweite Term die van-der-Waals- Anziehung repräsentiert. Der Parameter $\epsilon$ repräsentiert die Potentialtiefe und $\sigma$ ist der Teilchenabstand, an dem das Lennard-Jones-Potential eine Nullstelle besitzt.

## Aufgabe - Lennard-Jones-Potential

lennardjones.py

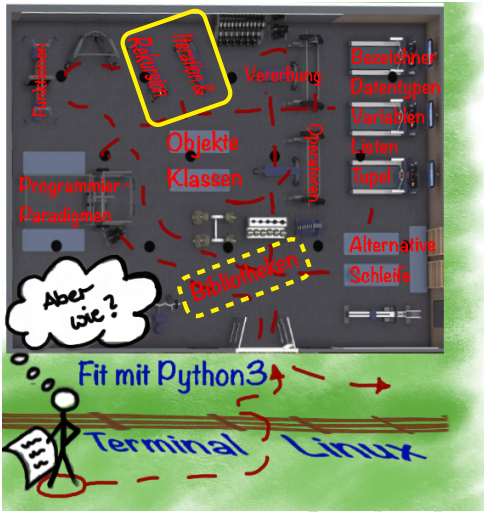
Schreiben Sie ein Programm, dass das Lennard-Jones-Potential für ein Interval $[\mathrm{r}_1, \mathrm{r}_2]$ berechnet und tabellarisch (d.h. $\mathrm{r}$, Tabulator, Wert LJP) ausgibt. Der Nutzer soll die Werte $\epsilon$, $\sigma$, $\mathrm{r}_1$, $\mathrm{r}_2$, und $\Delta\mathrm{r}$ eingeben können.
Verwenden Sie eine Funktion zur Berechnung des Lennard-Jones-Potentials für einen Abstand $\mathrm{r}$.

# 6. Einheit

## Ziel

Programmierkonzepte

- Iteratives Programmieren
- Rekursives Programmieren



# Programmierkonzepte

## Divide et impera

- Schwierige Aufgaben können in einfache Teilaufgaben zerlegt werden.

# Iteratives Programmieren

## Iteration

- Aufgabe wird so zerlegt, dass EINE einfache Aufgabe WIEDERHOLT gelöst wird.
- Meist mit Rückkopplung, so dass das Ergebnis eines Iterationsschrittes als Ausgangswert für den Nächsten dient.

## Beispiel: Addiere alle Zahlen von 1 bis n

Man definiere eine Zahl $\mathrm{i}$ als 1 und eine Summe $\mathrm{s} = 0$. In jedem Schritt addiere man $\mathrm{i}$ zu $\mathrm{s}$ dazu und erhöhe $\mathrm{i}$ um 1. Dies wiederholt man bis $\mathrm{i} == \mathrm{n}$ ist und fertig ist die Summe.
($\mathrm{n}$ mal wiederholen, bzw. wiederhole bis)

## Beispiel: Hochhaus mit $\mathrm{N}$ Etagen bauen

Eine Baufirma hat $\mathrm{N}$ Personen und die jeweils eine Etage bauen können. Der erste fängt an und wenn er fertig ist, gibt er den Auftrag weiter, bis keiner mehr übrig ist. Dann ist das Haus fertig.

# Iteratives Programmieren

## Beispiel: Iteratives berechnen von $n!$

1. Wie kann man die Fakultät im Sinne der Iteration umschreiben? Was kann uns helfen?
   ???
2. Welche kleinen Teilaufgaben kann man daraus ableiten?
   ???
3. Grundschema der resultierenden Aufgabenlösung?
   ???
4. Wie lautet der resultierende Algorithmus?
   ???

# Iteratives Programmieren

## Beispiel: Iteratives berechnen von $\mathbf{n}!$

1. Wie kann man die Fakultät im Sinne der Iteration umschreiben? Was kann uns helfen?
   $\mathbf{n}! = \mathbf{n} * (\mathbf{n}-1) * \cdots * 2 * 1 = \mathbf{n} * [(\mathbf{n}-1) * [\cdots * [2 * [1]]]]$
2. Welche kleinen Teilaufgaben kann man daraus ableiten?
   $\mathbf{n}! = [\mathbf{n} \cdot [(\mathbf{n}-1) \cdot [(\mathbf{n}-2) \cdot [\ldots]]]]$
3. Grundschema der resultierenden Aufgabenlösung?
   $(\mathbf{a} \cdot \mathbf{b}) \leftrightarrow \text{res} = \mathbf{n} * \text{res}$ mit der Bed. das n immer kleiner wird
4. Wie lautet der resultierende Algorithmus?
   Initialisiere $\text{res}, \mathbf{n}$
   Wiederhole Grundschema $\mathbf{n}$ mal $\text{res} = \mathbf{n} * \text{res}$ wobei $\mathbf{n} = \mathbf{n} - 1$ oder $\mathbf{n} = \mathbf{n} + 1$

| Initialsiere res, n |
| --- |
| n > 0 |
| res = res * n |
| n = n - 1 |

# Iteratives Programmieren

## Berechnung der Fakultät $\mathbf{n}!$

fakultaet.py,fakultaetFunk.py

Schreiben Sie ein Programm, dass die $\mathbf{n}$-te Fakultät einer natürlichen Zahl iterative berechnet.

Lagern Sie anschließend die Berechnung in eine Funktion aus. Ein- und Ausgabe erfolgen weiterhin ausserhalb der Funktion.

## Berechnung des Logarithmus über Reihenapproximation

naeherungLog.py

Berechnen Sie den Wert von $\log(\mathbf{x}+1)$ durch die Näherungsformel

$$\log(\mathbf{x}+1) = \lim_{\mathbf{n}\to\infty} \mathbf{L}(\mathbf{x},\mathbf{n}) \approx \sum_{\mathbf{i}=1}^{\mathbf{n}} \frac{1}{\mathbf{i}} \left(\frac{\mathbf{x}}{\mathbf{x}+1}\right)^{\mathbf{i}},$$

wobei der Nutzer die Genauigkeit der Näherungslösung eingeben kann.
*Version 1:* Die Genauigkeit wird zwischen der Näherungslösung und dem exakten Wert bestimmt.
*Version 2:* Da die Reihe monoton zur Lösung konvergiert, kann man auch die $|\mathbf{L}(\mathbf{x},\mathbf{n}+1) - \mathbf{L}(\mathbf{x},\mathbf{n})|$ als Genauigkeitswert verwenden.

# Rekursives Programmieren

## Rekursion

- Aufgabe wird so zerlegt, dass EINE schwere Aufgabe SICH SELBST als einfachere Teilaufgabe enthält. Die Lösung ergibt sich, sobald die einfachste Teilaufgabe gelöst wird.
- Aufgabe wird so zerlegt, dass EINE einfache Aufgabe als Funktion definiert wird, die sich selbst wieder aufruft.

## Beispiel: Addiere alle Zahlen von 1 bis n

Diese Problem lösst eine Funktion **BerechneSumme(**$\mathbf{k}$**)**.
*Rekursionsstart:* Im Fall $\mathbf{k} == 1$, weiß man, dass man nur 1 zurückgeben muss, weil fertig.
*Rekursionsschritt:* Im Fall $\mathbf{k} > 1$ kennt man "nur" $\mathbf{n}$ und weiß dass man $\mathbf{n}$ zur Summe von $\mathbf{n} - 1$ hinzuaddieren muss. Daher ruft man die Funktion **BerechneSumme(**$\mathbf{n} - 1$**)** auf und addiert eigenen Wert $\mathbf{n}$ dem Rückgabewert hinzu.
Aufruf der Funktion mit **BerechneSumme(**$\mathbf{n}$**)** löst dann das Problem.

# Rekursives Programmieren

## Rekursion

- Aufgabe wird so zerlegt, dass EINE schwere Aufgabe SICH SELBST als einfachere Teilaufgabe enthält. Die Lösung ergibt sich, sobald die einfachste Teilaufgabe gelöst wird.
- Aufgabe wird so zerlegt, dass EINE einfache Aufgabe als Funktion definiert wird, die sich selbst wieder aufruft.

## Beispiel: Hochhaus mit $\mathbb{N}$ Etagen bauen

Eine Baufirma soll ein Haus mit $\mathbb{N}$ Etagen bauen, hat aber nur Material um die $\mathbb{N}$te Etage zu bauen. Das Bauen der vorhergehenden Etagen wird an ein Subunternehmen ausgelagert und man startet wenn das Subunternehmen fertig ist. Dies passiert immer weiter, bis man das Subunternehmen ist, das das Fundament baut und seinem Auftraggeber bescheid gibt, dass man fertig ist.

# Rekursives Programmieren

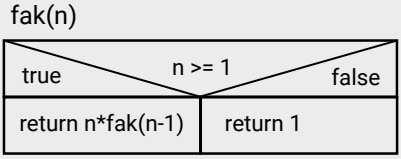
## Beispiel: Rekursives berechnen von `n`!

1. Wie kann man die Fakultät im Sinne der Rekursion umschreiben? Was kann uns helfen?
   ???
2. Welche kleinen Teilaufgaben kann man daraus ableiten?
   ???
3. Grundschema der resultierenden Aufgabenlösung?
   ???
4. Wie lautet der resultierende Algorithmus?
   ???

# Rekursives Programmieren

## Beispiel: Rekursives berechnen von $\mathbf{n}!$

1. Wie kann man die Fakultät im Sinne der Rekursion umschreiben? Was kann uns helfen?
   $\mathbf{n}! = \mathbf{n} * (\mathbf{n}-1) * \cdots * 2 * 1 = \mathbf{n} * (\mathbf{n}-1)!$
2. Welche kleinen Teilaufgaben kann daraus ableiten?
   $\mathbf{n}! = \mathbf{n} * (\mathbf{n}-1)!$
3. Grundschema der resultierenden Aufgabenlösung?
   $\mathbf{f}(\mathbf{n}) = \mathbf{n} * \mathbf{f}(\mathbf{n}-1)$
4. Wie lautet der resultierende Algorithmus?
   Definiere $\mathbf{f}(\mathbf{n}) = \mathbf{n} * \mathbf{f}(\mathbf{n}-1)$
   mit f(0) = 1



# Rekursives Programmieren

## Berechnung der Fakultät n! — fakultaetRek.py

Schreiben Sie ein Programm, dass die n-te Fakultät einer natürlichen Zahl rekursiv berechnet. Die Ein- und Ausgabe soll im Hauptteil des Programms erfolgen.

# Iteratives vs. rekursives Programmieren

## Aufgaben

Überlegen Sie sich jeweils einen iterativen und einen rekursiven Algorithmus zum Lösen folgender Aufgaben. Überlegen Sie sich zuerst eine Programmablauf!

- Wenn sich jede Bakterie einmal pro Sekunde teilt und so eine neue Bakterie erzeugt; wie viele Bakterien gibt es nach t Sekunden? bakterien.py
- Berechnen Sie die Summe der natürlichen Zahlen von 1 bis $\mathbb{N}$. sumN.py
- Nehmen wir an, jede Person begrüßt jede andere Person genau ein Mal per Handschlag. Wieviele Handschläge gibt es, wenn sich eine Gruppe von $\mathbb{N}$ Personen zusammenfindet und sich begrüßt. handschlag.py
- Auf einen Parkplatz passen $\mathbb{M}$ Autos, wieviele Möglichkeiten gibt es $\mathbb{N}$ Autos ($\mathbb{N} < \mathbb{M}$) auf diesen Parkplatz ordnungsgemäß zu parken? parkplaetze.py

**Hinweis:** Verwenden Sie nur Grundrechenarten (+, -, *, /) zum Lösen dieser Aufgaben!

# Komplexere rekursive Beispiele

## Typische Rekursionsaufgaben (oft auch iterativ lösbar):

- Berechnung mathematischer Funktionen (Potenz, Fakultät, Summe, Summe aller Quadrate, Binomische Formel, Ackermanns Funktion, Fibonacci, …)
- Umrechnung von Dezimal- (pos., ganzzahlig) in Dualzahl $\rightarrow$ Modulomethode
- Berechnung von Reproduktionstabellen
- Türme von Hanoi

# Komplexere rekursive Beispiele

## Kaninchenpopulation nach Leonardo Fibonacci (1202) kaninchen.py

Modell zur Beschreibung des Wachstums einer Kaninchenpopulation:

1. jedes Kaninchenpaar gebärt jeden Monat ein neues Kaninchenpaar
2. ein neu geborenes Kaninchenpaar kann erst im 2. Lebensmonat ein neues Paar gebären
3. die Kaninchen leben ewig

Beginnend mit einem trächtigen Pärchen, wieviele Kaninchen gibt es nach $\mathbb{N}$ Monaten?

Überlegen Sie sich einen rekursiven Algorithmus!

# Komplexere rekursive Beispiele

## Türme von Hanoi (1883, Édouard Lucas) – rekursiv

hanoi.py

In einem Tempel in der indischen Stadt Benares liegen 64 kostbare Scheiben zu einem Turm aufgeschichtet. Jede Scheibe ist ein wenig kleiner als die Scheibe auf der sie ruht. Eine Priesterorden hat nun die Aufgabe erhalten den Turm unter Beachtung heiliger Regeln zu einer anderen Stelle im Tempel zu bewegen. Die Scheiben sind so kostbar, dass sie nur auf einem von drei Plätzen im Tempel liegen dürfen, dem Platz wo der Turm zu Beginn stand, dem wo er aufgebaut werden soll und einem Platz dazwischen. Die Scheiben sind schwer und zerbrechlich, daher darf immer nur eine der Scheiben bewegt werden, niemals mehrere zur gleichen Zeit. Die letzte Regel besagt, dass zu keiner Zeit eine Scheibe auf einer kleineren Scheibe liegen darf. Wenn der Turm an einer Stelle abgebaut und an andere Stelle wieder ganz aufgebaut wurde, wird der Tempel und mit ihm die ganze Welt zu Staub zerfallen.

# Komplexere rekursive Beispiele

## Türme von Hanoi (1883, Édouard Lucas) – rekursiv

hanoi.py

Spielregeln:

1. Die $\mathtt{N}$ Scheiben dürfen nicht außerhalb der markierten Felder {1,2,3} abgelegt werden.
2. Pro Zug darf nur eine Scheibe $\mathtt{i}$ verschoben werden.
3. Eine Scheibe darf niemals auf eine kleinere Scheibe gelegt werden.

Schreiben Sie ein Programm, das ausgibt in welcher Reihenfolge welche Scheibe $\mathtt{n}$ von wo $\mathtt{i}$ nach wo $\mathtt{j}$ gelegt werden muss um einen Turm von $\mathtt{N}$ Scheiben zu verschieben.

`n : i -> j`

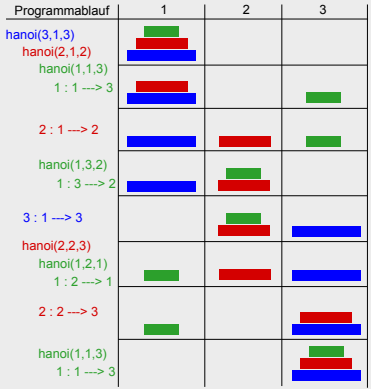
Bitte probieren Sie zunächst KLEINE Werte von $\mathtt{N}$, da die Schrittanzahl$= 2^{\mathtt{N}} - 1$ ist.

# Komplexere rekursive Beispiele

## Türme von Hanoi (1883, Édouard Lucas) – rekursiv

hanoi.py

Ablaufplan bei $\mathbf{n} = 3$:




Struktogramm:

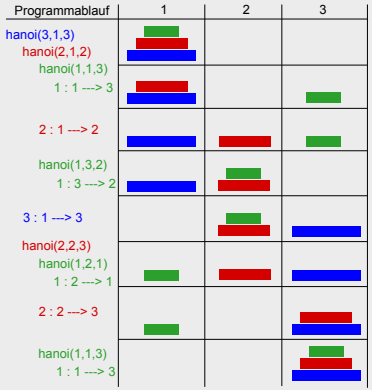
???

# Komplexere rekursive Beispiele

## Türme von Hanoi (1883, Édouard Lucas) – rekursiv

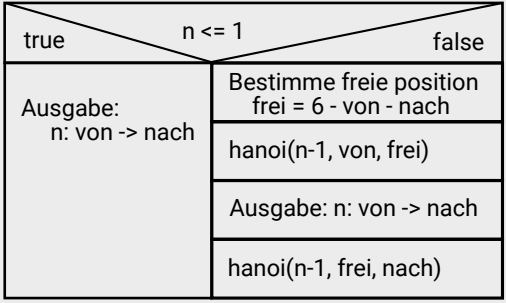
hanoi.py

Ablaufplan bei $\mathbf{n} = 3$:




Struktogramm:

hanoi(n, von, nach)

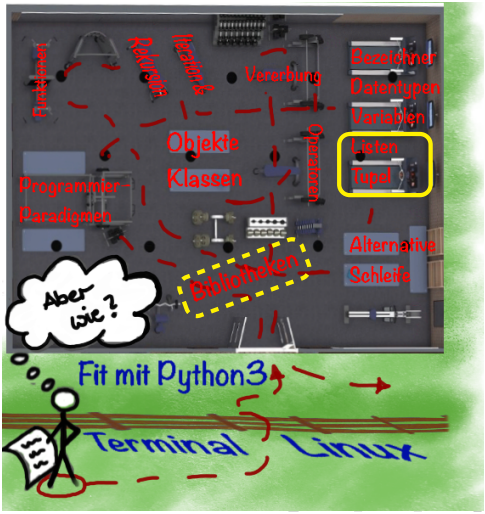


# 7. Einheit

## Ziel

Nichtelementare Datentypen

- Sequentielle Datentypen
- Nicht-sequentielle Datentypen



# Nichtelementare Datentypen

## Sequentielle Datentypen

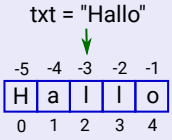
Eine Folge gleich- oder verschiedenartiger Elemente mit definierter Reihenfolge, so dass man über Indizes darauf zugreifen kann

- Veränderbare (mutable) Typen :
  Daten können während des Programmablaufs jederzeit geändert werden.
  - Listen
  - Binärdaten vom Typ "bytearray"
- Unveränderbare (immutable) Typen :
  Daten werden beim Initialisieren festgelegt und können anschließend nicht mehr geändert werden.
  - Strings
  - Tuple
  - Binärdaten vom Typ "bytes"

# Sequentielle Datentypen

## Index-Zugriff

- Zugriff auf einzelne Elemente eines sequentiellen Datentyps via `[<i>]`
- `<i>` ist Index (Positionswert) beginnend von 0 bis Länge der Folge
  Bestimmung Länge eine sequentiellen Variable: `len()`
- Erweiterter Zugriff möglich:
  - Ausschneiden: Angabe von Index `i` bis `j` via `[<i>:<j>]` (Slicing), `[:<j>]` oder `[<i>:]`
  - Erweiterung um Schrittweiten d: `[<i>:<j>:<d>]`




## Was passiert bei . . . ?

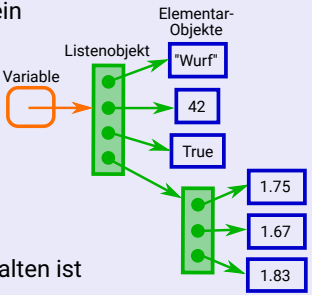
```
test = "Hello World"
test[-2]
test[5]
```

```
test[3:8]
test[:4:2]
test[::-1]
```

# Listen (veränderbare Typen)

## Listen anlegen

- Definition: `<liste> = [<wert1>, <wert2>, ...]`
- Werte können gleiche oder verschiedene Datentypen, oder auch Listen sein
  → Verschachtelte Listen
- Beispiel: `versuch = ["wurf", 42, True, [1.75,1.67,1.83]]`
  → `versuch[0]`=?, `versuch[3][2]`=?




- Verwendbare Operatoren (i.A. für sequentielle Datentypen):
  - + Operator: Verketten von Listen
  - * Operator: Wiederholte Anwendung von Listen
  - `<wert> in <liste>`: Prüft ob `<wert>` in der Liste `<liste>` enthalten ist
  - `<wert> not in <liste>`

# Listen (veränderbare Typen)

## Listen manipulieren

- `<liste>.append(<wert>)`-Methode: Fügt `<wert>` als letztes Element der Liste hinzu (= “push”-Methode).

  **Hinweis:** append und + führen zum gleichen Resultat, aber
  - bei `<liste>.append(<wert>)` und `<liste> += <wert>` wird `<wert>` an Liste direkt angehängt
  - `<liste> = <liste> + <wert>` erzeugt Kopie von `<liste>`, hängt dann `<wert>` an und muss anschließend Speicherplatz freigeben
- `<liste>.insert(<i>,<wert>)`: Fügt `<wert>` an `i`. Stelle ein.
- `<liste>.extend(<iterierbares unveränderbares Objekt>)`: Erweitert Liste um mehrere Einträge

**Bemerkung:** Iterator ist ein Objekt, das eine zählbare Anzahl an Werten enthält, über die iteriert werden kann.

# Listen (veränderbare Typen)

## Listen manipulieren

- `<liste>.pop()`-Methode: Gibt das letzte Element der Liste zurück und entfernt es aus der Liste
  Erweiterung `<liste>.pop(<i>)`: Gibt das `<i>`. Element der Liste zurück und entfernt es aus der Liste
- `<liste>.remove(<wert>)`: Entfernt Eintrag an der ersten gefunden Stelle

# Listen (veränderbare Typen)

## Listen an Funktionen übergeben

- Übergabe an Funktionen als Referenzen
- Direkte Veränderung der Listenelemente über Manipulationsmethoden möglich (append, insert, etc.)
- Beachte: Bei neuer Referenzzuweisung sind die “neuen” Werte nur lokal bekannt

```
def testListe(l):
    l.append(4)
#    l=[1,2,3]
#    print(l)
#    l.insert(2,5)

my_list = [42,12,2]
testListe(my_list)
print(my_list)
```

# Listen (veränderbare Typen)

## Aufgabe – Skalarprodukt

skalarprod.py

1. Schreiben Sie ein Programm, das zwei (Spalten-)Vektoren $\vec{\mathbf{x}} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_\mathbf{N})$ und $\vec{\mathbf{y}} = (\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_\mathbf{N})$ gleicher Länge $\mathbf{N}$ vom Nutzer einliest und anschließend das Skalarprodukt $\mathrm{sp}$ beider Vektoren berechnet und ausgibt. Die allg. Formel für das Skalarprodukt lautet:
$$\mathrm{sp} = \sum_{\mathrm{j}=1}^{\mathbf{N}} \mathbf{x}_\mathrm{i} * \mathbf{y}_\mathrm{i}$$
2. Überlegen Sie sich einfache Beispielfälle um Ihr Programm zu testen.
3. Lagern Sie anschließend die vollständige Berechnung in eine selbstgeschriebene Funktion aus.
4. Welche Funktion in welcher Bibliothek könnte diese Berechnung für Sie übernehmen? Testen Sie es aus.

# Arrays (veränderbare Typen)

## NumPy – Arrays

- Arrays sind ein NumPy-interner Datentyp
- Arrays vergleichbar mit mathematischen Vektoren
- Definition: `<array>=numpy.array(<liste>)` mit `<liste>=[...]`
- Analog zu Listen: Verschiedene Datentypen kombinierbar (mehrdimensional), indizierbar (slicing), iterierbar
- Unterschied: Funktionalität der Objekte!

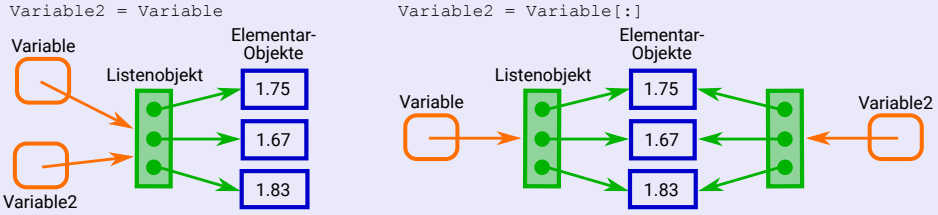
```
liste=[1,2,3,4]
yl=3*liste
print(yl)
```

```
import numpy as np
array=np.array([1,2,3,4])
ya=3*array
print(ya)
```

# Listen (veränderbare Typen)

## **Beachte:** Listen kopieren/wiederholen/verketten

- Kopieren/... von 1-dimensionale (flachen) Listen ist kein Problem, wenn man die Slicing-Version verwendet: `list2 = liste1[:]`



# Listen (veränderbare Typen)

## **Beachte:** Listen kopieren/wiederholen/verketten

- Beachte zusätzlich: Beim Kopieren/... verschachtelter Listen via Slicing, werden nur Referenzen auf die Unterlisten erstellt

```
v1 = ["wurf",42,True,[1.75,1.67,1.83]]
v2 = v1[:]
v2[3] = 43
v2[3][1] = 1.43
print(v1,v2)
```

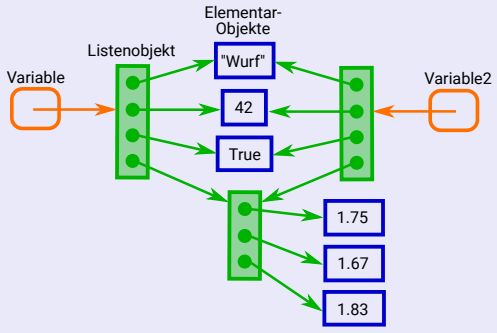
**Abhilfe:** `deepcopy()` aus der Bibliothek copy

```
x = ["a","b","c"]
y = 4 * [x]
y[0][1] = "p"
print(y)
```

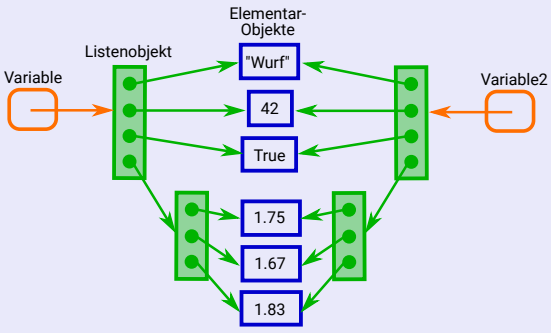
# Listen (veränderbare Typen)

## **Beachte:** Listen kopieren/wiederholen/verketten (Erklärung)

```
Variable2 = Variable[:]
```




```
import copy
Variable2 = copy.deepcopy(Variable)
```



# Tupel (unveränderbare Typen)

## Tupel

- Definition: `<tupel> = (<wert1>, <wert2>, ...)`
- Werte können gleiche oder verschiedene Datentypen sein
- Verwendbare Operatoren: + Operator, * Operator, `<wert> in <tupel>`, `<wert> not in <tupel>`
- Bestimmung der Länge über `len(<tupel>)`
- Tupel werden als lokale Kopie an Funktionen übergeben
  $\rightarrow$ Lokale Kopie, Speicherintensiv, langsam (Rechenzeit), “sicher”
  $\Rightarrow$Vgl. Listen $\rightarrow$ Referenzen, Speicherneutral, schnell (Rechenzeit), “unsicher”

# Tupel (unveränderbare Typen)

## Tuple - Packing

- Tuple-Packing: `x = (a, "Hallo", 3.45)`
- Tuple-Unpacking: `(wert1, text, wert2) = x`
- Tuple-Unpacking mit Wildcards `*` (beliebig viele Parameter)

```
items = range(5)          # (0,1,2,3,4)
a, *rest = items          # a=0, rest=[1,2,3,4]
a, *rest, b = items       # a=0, rest=[1,2,3], b=4
*rest, b = items          # rest=[0,1,2,3], b=4
```

# Tupel (unveränderbare Typen)

## Tuple - Vergleiche

- Vergleich zweier Tuple (Listen) ist möglich. Was passiert?

```
a = (5,6)      # (5,6)      ## (5,6)
b = (1,4)      # (5,4)      ## (6,4)
if a>b :
    print ("a ist groesser.")
else :
    print ("b ist groesser.")
```

# Tupel (unveränderbare Typen)

## Tuple - Vergleiche

- Vergleich zweier Tuple (Listen) ist möglich. Was passiert?

```
a = (5,6)      # (5,6)      ## (5,6)
b = (1,4)      # (5,4)      ## (6,4)
if a>b :
    print ("a ist groesser.")
else :
    print ("b ist groesser.")
```

  Ausgaben sind:
  - `a ist groesser`
  - `# a ist groesser`
  - `## b ist groesser`

# Erweiterung

## Tauschen der Inhalte von Variablen

- “Per Hand”: ?
- “Per Tuple”: ?

# Erweiterung

## Tauschen der Inhalte von Variablen

- “Per Hand”: ?
- “Per Tuple”: ?
- “Per Hand”: `a=1,b=2, a=c, a=b, b=c`
- “Per Tuple”: `a=1,b=2, (a,b)=(b,a)` (Höherer Speicherbedarf)

# Strings (unveränderbare Typen)

## Methode zur formatierten Ausgabe von Variablen via %

Der String-Modulo % erlaubt die formatierte Ausgabe von Variablen.

- Definition: `"bla %<code> blub %<code> ..." % (<var1>, ..., <varN>)`
- `<code>` : setzt sich zusammen `[Breite][.Genauigkeit]Datentyp`

  `[Breite]` : Anzahl an Ziffern die min. für die Zahl reserviert werden (Leerzeichen vor der Zahl)
  `[.Genauigkeit]` : Anzahl der Kommastellen
  `Datentyp` : d $\leftrightarrow$ Ganze Zahl, f oder e $\leftrightarrow$ Gleitkommazahl, ...
- Hinweis: Anzahl an %-Symbolen im Text "..."müssen der Anzahl der Variablen in der Klammer entsprechen!
- Typische Codes: siehe `https://www.python-course.eu/python3_formatted_output.php`

# Strings (unveränderbare Typen)

## Beispiel - Zuweisung der Positionscode

```
"Text1: %5d, Text2: %8.2f" % (453, 59.058)
```

→ Resultierende Zeichenkette:

```
"Text1:   453 , Text2:     59.06"
```

```
"Für N = %5d ist e = %10.8f" % (nmax,resE1)
```

→ Resultierende Zeichenkette:

```
"Für N =  3688 ist e = 2.71791329"
```

# Strings (unveränderbare Typen)

## `string`-Methode `format`

`string`-Methode `format` erlaubt das formatierte ausgeben von Variablen in Kombination mit Zeichenketten

- Definition: `<"Zeichenkette">.format(p0,p1,..k0=v0,k1=v1,...)`
- `<"Zeichenkette">` enthält Ersetzungsregeln zur formatierten Ausgabe von Variablen `p0,p1,...`
- Ersetzungsregeln innerhalb der Zeichenkette sind mit {<code>} gegeben

# Strings (unveränderbare Typen)

## `string`-Methode `format`

- `<code>` setzt sich zusammen aus `<pos/key>:<format><type>`
  - `<pos/key>`: Entweder Angabe der Position innerhalb der `.format()`-Klammer (pos$= 0, 1, \ldots$!); oder Angabe des Keys entsprechend der `.format()`-Klammer (`key=k0,k1,...`)
  - `<format>`: Anzahl der Zeichen die für Ausgabe benutzt werden soll, erweitert durch Anzahl Kommastellen (wenn der Datentyp so etwas hergibt)
  - `<type>`: Definiert den Datentyp der Ausgabevariable (z.B. `d` für Integer, `f` für float)
  - Typische Codes: siehe `https://www.python-course.eu/python3_formatted_output.php`

# Strings (unveränderbare Typen)

## Beispiel - Zuweisung der Positionscode

```
"Text1: {0:5d}, Text2: {1:8.2f}".format(453, 59.058)
```
→ Resultierende Zeichenkette:
```
"Text1:   453 , Text2:     59.06"
```

```
"Für N = {0:5d} ist e = {1:10.8f}".format(nmax,resE1)
```
→ Resultierende Zeichenkette:
```
"Für N =  3688 ist e = 2.71791329"
```

## Beispiel - Zuweisung der Variablenname

```
"Art: {a:7d}, Price: {p:6.3f}".format(a=453, p=59.058)
```
→ Resultierende Zeichenkette:
```
"Art:     453 , Price:  59.058"
```

# Alternative: Formatierte Ausgabe

## print()

- Standarddefinition:
  `print(<wert1>[,..., sep=' ', end='\n', file=sys.stdout, flush=False)`
- `sep`: Trennzeichen zwischen `<wert1>` und Co. (alternativ: `'\n',' :) ',...`)
- `end`: Schlusszeichen am Ende der print-Ausgabe (alternativ: `' ',...`)
- `file`: Gibt an wohin die print-Ausgabe geschrieben wird.
  Möglichkeit:

```
fh = open("daten.txt", "w")
print("#hallo\n", <werte>, file=fh)
fh.close()
```

- `flush`: Ob print-Ausgaben sofort oder gesammelt ausgegeben sollen.

# Erweiterung: Einlesen von Werten

## `input()`

- Bisher `input()` → String
- `eval(input())` → Interpretiert Eingabe in vermutete Datentypen

## Testaufgaben – `eval()`

Test Sie die folgenden Befehle aus um `eval(input())` besser zu verstehen:

```
a = eval(input("a,b␣"))      # 3,4
b,c = eval(input("b,c" ))    # 1,2
print(a, b + c, end='\n')
staedteEval = eval(input("Staedteliste?␣"))
staedteList = list(input("Staedteliste?␣"))
    # ["Chemnitz","Dresden", "Leipzig"]
print(staedteEval, staedteList, sep='\n')
```

# Nichtelementare Datentypen

## Nicht-Sequentielle Datentypen

Eine Folge gleich- oder verschiedenartiger Elemente (unveränderbarer Datentyp) ohne definierte Reihenfolge

- Menge (set) :
  - Definition: `<set> = {<wert1>,<wert2>,...}` (`<wertX>` – unveränderbarer Datentyp)
  - Umwandlung von Listen in Mengen via `set(<liste>)` und umgekehrt `list(<set>)`
  - Methoden: `.add()`, `.clear()`, `.copy()`, `.difference()`, `.union()`, `.intersection()`,...
- Wörterbuch (Dictionary) :
  - Definition: `<dict> = {<schl1> : <wert1>, <schl2> : <wert2>,...}` (`<schlX>` – Schlüsselwort ist unveränderbarer Datentyp)
  - Zugriff auf Elemente via `[<schli>]`
  - Operatoren: `len(<dict>)`, `del <dict>[<schli>]`, `<schli> in <dict>`, `<schli> not in <dict>`

# Nichtelementare Datentypen

## List-, Set-, oder Dict-Comprehension

- Kompakte Möglichkeit eine Liste, Menge oder ein Dictionary zu erzeugen
- Definition:
  - Liste: `<liste> = [ <werti> for <i> in <Sequenz> if <Filter> ]`
  - Set: `<set> = { <werti> for <i> in <Sequenz> if <Filter> }`
  - Dict: `<dict> = { <schli> : <werti> for <i> in <Sequenz> if <Filter> }`
- Bedeutung:

```
x = [i*i for i in range(5) if i%2 == 0]
```

```
x = []
for i in range(5)
  if i%2==0:
    x.append(i*i)
```

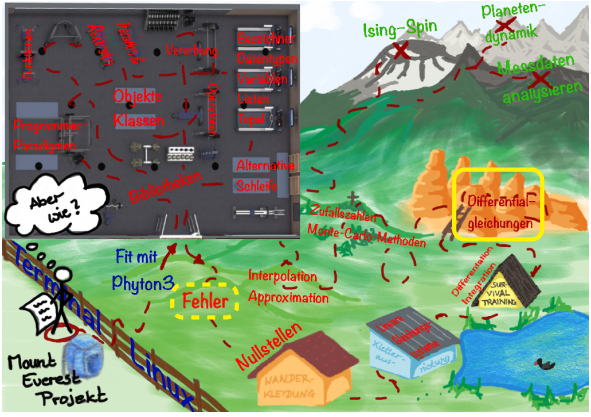
- Typischer Anwendungsfall:

```
y = [float(input("x[{0:1d}]?".format(i))) for i in range(10)]
```

# 8. & 9. Einheit

## Ziel

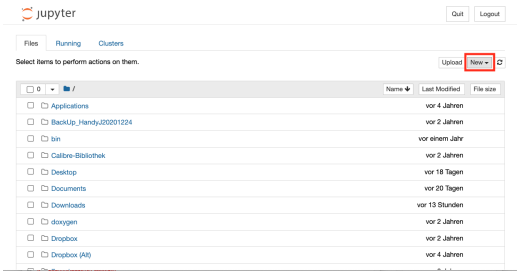
- Graphikbibliothek `matplotlib`
- Computeralgebra-System `sympy`

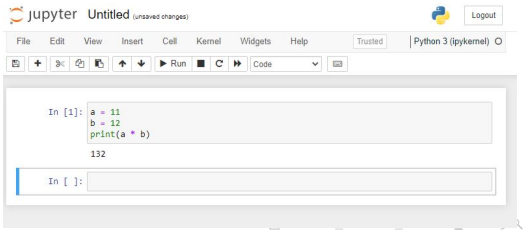


# IDE: Jupyter-Notebook

## Jupyter Notebooks (z.B. via Anaconda)

- Grafische Oberfläche zur Entwicklung von formatierten python-Skripten
- Besondere Anwendungsmöglichkeiten:
  - Symbolisches Programmieren (Computeralgebra-System) via `sympy`
  - Weitergabe von formatierten Skripten und Aufgaben
- Konsolenaufruf: `jupyter-notebook &` startet im Browser einen Dateimanager
- (Leeres) Notebook starten →
- Dateityp: `<Dateinamen>.ipynb`

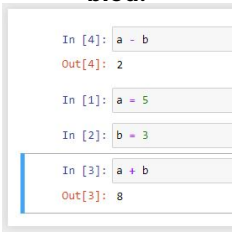


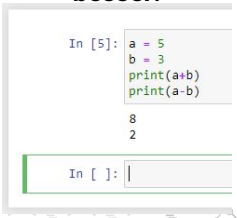
# IDE: Jupyter Notebooks

- Zellenbasiert für python-Code, praktisch fürs python -Modul `sympy` (kommt später)
- Zellen können in beliebiger Reihenfolge ausgeführt werden
- Ausführung des Codes mittels `[shift]+[enter]`
- Achtung: Die Reihenfolge der Ausführung der Zellen ist wichtig, da Definitionen von Variablen/Funktionen persistieren
- Hinweis: Alles was wir bisher gelernt haben können wir auch hier nutzen. (Bibliotheken, Variablen, Funktionen, Schleifen, Alternativen)

**blöd:**



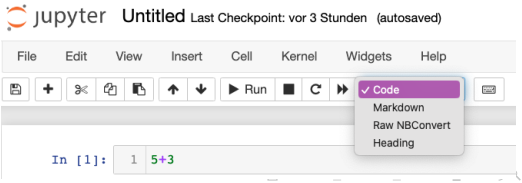

**besser:**



# IDE: Jupyter Notebooks

## Formatieren von Jupyter-Notebooks

- **Code:** python Quellcode
- **Markdown:** Formatierter Text
  Formatierungserklärungen zu Jupyter-Notebooks von IBM
- **Raw NBConvert:** Konvertieren von verschieden Code-Formaten in HTML
- **Heading:** siehe Markdown # . . .
- Reihenfolge der Zellen änderbar via $\uparrow, \downarrow$



# IDE: Jupyter Notebooks

## Umgang mit IDE üben

quadFunk.ipynb

- Definiere eine Funktion mit 4 Parametern $(\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{c})$ in einer eigenen Zelle, die den Funktionswert $\mathbf{f}(\mathbf{x})$ einer quadratischen Funktion berechnet und zurück gibt.
- Lese die Parameter $\mathbf{a}, \mathbf{b}, \mathbf{c}$ vom Nutzer ein. Erstelle eine Liste der $\mathbf{x}$-Koordinaten im Intervall $[-2, 2]$ mit Schrittweite $\Delta\mathbf{x} = 0.2$ und erstelle eine Liste der zugehörigen Funktionswerte. Lasse dir beide Listen kommentiert ausgeben.
- Erweitere das Notebook um eine Überschrift und die Aufgabenstellung.

# Grafische Darstellung

## matplotlib.pyplot

`matplotlib` ist eine Python Bibliothek zur Darstellung von 2D-Diagrammen. Typische Befehle und eine umfangreiche Galerie an Möglichkeiten der Bibliothek findet man auf `https://matplotlib.org`.

Ausgewählte Grundbefehle um 2D-Kurven darzustellen:

- Erstellen eines Diagramms mittels Variable `<plot>` via `import matplotlib.pyplot as <plot>`
- Anzeige einer Kurve:
  `<plot>.plot([x],y,[<format>],[<schli>=...])`
  → `<format>` = `'g-'` (grüne Linie), `'ro'` (rote Punkte)
  → `<schli>=...`, z.B. `label='linie1', linewidth=2`
- Anzeige des Diagramms:`<plot>.show()`

# Grafische Darstellung

## matplotlib.pyplot

- Achsenbeschriftung:
  `<plot>.xlabel(<label>)`, `<plot>.ylabel(<label>)`
- Legende anzeigen:
  `<plot>.legend(handles=[<curve1>,...])`
- Anzeige mehrere Kurven in einem Diagramm:
  - `<plot>.plot([x1],y1,[<format1>],[x2],y2,[<format2>] ,...)` oder
  - Mehrmaliges aufrufen von `<plot>.plot([x],y,[<format>], [<schli>=...])`
    $\rightarrow$ `<schli>=...`, z.B. `label='linie1', linewidth=2`
- Als PDF speichern:
  `<plot>.savefig(<pfad>)`
- **Zusatz - Jupyter-Notebook:** Einbinden der Bilder im Notebook (1$\times$ im Notebook)
  `%matplotlib inline`

# Grafische Darstellung

## matplotlib.pyplot

*Hinweis:* Oft wird die Variable `<plot>` in zwei Subvariablen geteilt, vor allem wenn man mehrere Diagramme in einem kombinieren möchte.
`<fig>, <ax> = <plot>.subplots()` (Variable `<ax>` wird dann für `plot()`, etc. verwendet)

# Grafische Darstellung

## Minimalbeispiel

GrafikMin.py/GrafikMin.ipynb

**Version 1:**

```
%matplotlib inline
# nur in Jupyter-Notebook
import matplotlib.pyplot as plt
listx = range(-10,10)
listy = [i**2-3 for i in listx]
plt.plot(listx, listy,'ro',label='
    test1')
plt.ylabel('f(x)')
plt.legend()
plt.show()
```

**Version 2:**

```
%matplotlib inline
# nur in Jupyter-Notebook
import matplotlib.pyplot as plt
listx = range(-10,10)
listy = [i**2-3 for i in listx]
fig,ax = plt.subplots()
ax.plot(listx, listy,'ro',label='
    test1')
ax.set_ylabel('f(x)')
ax.legend()
plt.show()
```

# Grafische Darstellung

## Aufgabe – Grafische Darstellung der quadratischen Fkt. quadFunk.ipynb

Erweitern Sie Ihr Notebook so, dass die Daten in einem $\mathtt{f}(\mathtt{x})$-$\mathtt{x}$-Diagramm dargestellt werden. Benutzen Sie hierzu die Python-Bibliothek `mathplotlib.pyplot` und machen Sie sich mit einigen Grundbefehlen vertraut.
Das Diagramm sollte eine Achsenbeschriftung, Legende und einen Titel haben.

## (Zusatz)Aufgabe – Grafische Darstellung des Lennard-Jones-Potentials lennardJonesPlot.py

Erweitern Sie Ihr Programm `lennardJones.py` so, dass Sie das Potential $\mathtt{V}(\mathtt{x})$ in einem Diagramm darstellen und in eine PDF-Datei ausgeben lassen.

# Symbolisches Programmieren – sympy

- Open-Source Bibliothek für python zum symbolischen Rechnen
- Dokumentation: https://docs.sympy.org/latest/index.html

```
In [1]: import sympy
        sympy.init_session()
```

Laden des Moduls

führt folgende relevante Kommandos aus

```
IPython console for SymPy 1.9 (Python 3.8.12-64-bit) (ground types: python)

These commands were executed:
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)
>>> init_printing()

Documentation can be found at https://docs.sympy.org/1.9/
```

legt Art der Division fest (*true division*)

importiere alles aus sympy, ⇒ Benutzung von Methoden als wären sie vordefiniert

Definieren von Symbolen für Gl.

Vorbereiten der Ausgabe, damit alles hübsch aussieht

# Symbolisches Programmieren – sympy

## Mathematische Funktionen

Durch `sympy.init_session()` werden alle gängigen mathematischen Funktionen und Konstanten importiert, z.B.:

- `exp(...), log(...), sqrt(...)`
- `cos(...), sin(...), tan(...)`, etc.
- `pi, I, E`, etc.

Symbolische Variablen/ Funktionen werden mit `Display(<ausdruck>)` “schön” dargestellt.

# Symbolisches Programmieren – sympy

## Gleichungen

- (Symbolische) Funktionen können wie normale Variablen gespeichert werden:
  `f = sin(x) * cos(x)**3 + pi`
- Die als **Veränderliche** genutzte Variablen **müssen vorab** definiert werden (siehe `sympy.init_session()`)
- *Hinweis:* Dies sind die **symbolischen** Repräsentationen der Funktionen, nicht mit den Numerischen verwechseln!
- Gleichungen werden mittels `Eq(<lhs>, <rhs>)` dargestellt.
- Vereinfachung von komplexen Darstellungen: `simplify(<ausdruck>)`

# Symbolisches Programmieren – sympy

## Graphische Darstellung von Funktionsverläufen

- `matplotlib` → Diskretisieren der analytischen Lösung für gegebene Argumente und Parameter
- Generieren einer aufrufbaren Funktion:

```
alpha, beta = symbols('alpha_beta', real=True)
f = sin(alpha x) * cos(alpha x)**3 + beta
# bei Gl. nur die rechte Seite: gl = Eq(...); f_repl=gl.rhs
f_repl = f.subs([ (alpha,1.5), (beta, 42) ]) # Ersetzen aller
    Parameter durch Werte
# generiere aufrufbare Fkt.
f_func = lambdify(x, f_repl, 'numpy')
f_func(1.5)   #Verwendung
```

→ Anschließend Datenpunkte (x,y) generieren und mit `matplotlib` darstellen

# Symbolisches Programmieren – sympy

## Vereinfache Gleichungen

Vereinfache die Funktion

$$\mathbf{f}(\mathbf{x}) = \sin(\mathbf{x})\ \cos^3(\mathbf{x}) + \sin^3(\mathbf{x})\ \cos(\mathbf{x})$$

und lasse beide Versionen in Form einer Gleichung symbolisch darstellen.
Lasse dir auch den grafischen Verlauf via `matplotlib` ausgeben.

# Definition von Gleichungen

## Lösung:

In [7]:

```
1 f = sin(x) * cos(x)**3 + sin(x)**3 * cos(x) # Definition
2 display(f) # Gleichung anzeigen
3 display(simplify(f)) # Gleichung vereinfachen & anzeigen
4
5 display( Eq(f, simplify(f)) ) # Anzeige mit Gleichheitsz.
```

$$\sin^3(x)\cos(x) + \sin(x)\cos^3(x)$$

$$\frac{\sin(2x)}{2}$$

$$\sin^3(x)\cos(x) + \sin(x)\cos^3(x) = \frac{\sin(2x)}{2}$$

# Symbolisches Programmieren – sympy

## Differentiation

- Funktion: `Derivative(<ausdruck>).doit()`

```
f = sin(x) * exp(x)                  # Gleichung definieren
f_derivative = Derivative(f, x, 2)   # 1. Moeglichkeit
display( f_derivative.doit() )
display( f.diff(x,2) )               # 2. Moeglichkeit
```

- Ableitung ist nach verschiedenen Variablen möglich

Leite folgende Funktionen nach $\mathbf{x}$ ab und vereinfache:

$$\mathbf{f(x)} = \frac{1}{1+\mathbf{e}^{-\mathbf{x}^2}} \qquad \mathbf{f(x)} = \sqrt{\mathbf{x}+\sqrt{\mathbf{x}+\sqrt{\mathbf{x}}}} \qquad \mathbf{f(x)} = \ln\frac{\mathbf{x}}{3\mathbf{x}^4+\mathbf{x}+2}$$

# Differentiation

## Lösung:

Berechne die folgenden Ableitung:

a) $y = \frac{1}{1+e^{-x^2}}$ $\quad y' = \frac{x}{2\cosh^2(x^2/2)}$

b) $y = \sqrt{x+\sqrt{x+\sqrt{x}}}$ $\quad y' = \frac{4\sqrt{x}\sqrt{x+\sqrt{x}}+2\sqrt{x}+1}{8\sqrt{x}\sqrt{x+\sqrt{x}}\sqrt{x+\sqrt{x}+\sqrt{x}}}$

c) $y = \ln\frac{x^6}{2+x+3x^4}$ $\quad y' = \frac{2-9x^4}{x(3x^4+x+2)}$

# Symbolisches Programmieren – sympy

## Integration

- Funktion: `integrate(...).doit()`
- bestimmtes Integral: `integrate(f, (x, x0, x1))`
- unbestimmtes Integral: `integrate(f, x)`

```
f = sin(x) * exp(x)                          # Gleichung
x0 = 0                                       # Grenzen definieren
x1 = 2
f_integral = integrate(f, (x, x0, x1))  # 1. Moeglichkeit
display( f_integral.doit() )
display( f.integrate((x, x0, x1)) )     # 2. Moeglichkeit
```

Integriere folgende Funktionen nach $\mathbf{x}$ und vereinfache:

$$\mathrm{F}(\mathbf{x}) = \int \cos^2(\mathbf{x})\mathrm{d}\mathbf{x} \qquad \mathrm{F}(\mathbf{x}) = \int \mathbf{x}\sin(\alpha\mathbf{x})\mathrm{d}\mathbf{x} \qquad \mathrm{F} = \int_3^4 \frac{\mathbf{x}^2}{\mathbf{x}^3 - 13}\mathrm{d}\mathbf{x}$$

# Differentiation

## Lösung:

Berechne die folgenden Integrale:

a) $F(x) = \int \cos^2(x)dx$ $\quad F(x) = x/2 + \sin(2x)/4 = \frac{x+\sin(x)\cos(x)}{2}$

b) $F(x) = \int x\sin(\alpha x)dx$ $\quad F(x) = \frac{-\alpha x\cos(\alpha x)+\sin(\alpha x)}{\alpha^2}$

c) $F = \int_3^4 \frac{x^2}{x^3-13}dx$ $\quad F = 1/3(\ln(51) - \ln(14) = 1/3\ln(51/14)$

# Symbolisches Programmieren – sympy

## Nullstellen

- Reele Nullstellen

```
f = x**3 - 7*x + 6   # Gl. def.
display(solveset(f)) # Nullstelle anzeigen
```

- Komplexe Nullstellen

```
f = x**4 + x**3 - 7 * x + 6     # Gleichung definieren
f_solutions_exact = solveset(f) # Nullstellen berechnen
f_solutions_num = [element.evalf() for element in f_solutions_exact]
display( f_solutions_num )      # Ausgabe der numerisch evaluierten Nullst.
```

Berechne die Nullstellen der folgenden Gleichungen (Wo könnten diese Gleichungen auftrauchen?):

$$0 = \lambda^2 + 3\,\lambda - 4 \qquad\qquad 0 = \lambda^3 + 3\lambda^2 + 3\lambda + 1$$

# Nullstellen

## Lösung:

Berechne die Nullstellen der folgenden Gleichungen:

a) $\lambda^2 + 3\lambda - 4 = 0$
$\lambda_1 = -4, \lambda_2 = 1$

b) $\lambda^3 + 3\lambda^2 + 3\lambda + 1 = 0$
$\lambda_{1,2,3} = -1$

# Symbolisches Programmieren – sympy

## Lineare Gleichungssysteme (LGS)

- LGS - Darstellung in Matrix-Vektor-Form:

$$\begin{array}{rcrcrcl} \mathbf{x} & + & \mathbf{y} & + & \mathbf{z} & = & 1 \\ \mathbf{x} & + & 2\mathbf{y} & + & 3\mathbf{z} & = & 2 \end{array} \quad \Rightarrow \quad \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \Rightarrow \quad \mathbf{A} \cdot \vec{\mathbf{r}} = \vec{\mathbf{b}}$$

```
A = Matrix(([1,1,1], [1,2,3]))      # definiere Matrix
b = Matrix(([1], [2]))              # definiere rechte Seite
r = Matrix(([x], [y], [z]))         # definiere Variablen
display( Eq(MatMul(A, r), b) )      # Ausgabe Gleichunssystem
display(linsolve((A,b), *r))        # Ausgabe Loesung
```

Was wird bei dem obigen Beispiel ausgegeben? Bestimme die Koeffizienten der folgenden Gleichung via “Koeffizientenvergleich”:

$$3\mathrm{C}\mathbf{x}^2 + (3\mathrm{D} - 4\mathrm{C})\mathbf{x} + 2/3\mathrm{C} - 2\mathrm{D} + 3\mathrm{E} = 1/3\mathbf{x}^2$$

# Symbolisches Programmieren – sympy

## Differenzialgleichungen

- Beispiel - Harmonischer Oszillator: $\ddot{\mathbf{x}} = -\omega^2 \mathbf{x}$ mit $\omega = \sqrt{\mathbf{k}/\mathbf{m}}$

```
# definiere extra allg. Parameter
omega = symbols('omega', real=True, positive=True)
# stelle Gleichung auf
diffeq = Eq( f(t).diff(t, 2) + omega**2 * f(t), 0 )
display(diffeq)
diffsol = dsolve(diff_eq, f(t)) # loese DGL
display(diffsol)
# Grafische Darstellung (Integrationskonstanten auch ersetzten)
diffsol_eq = diffsol.rhs
diffsol_repl = diffsol_eq.subs([ (omega, 2*pi), ('C1', 0.0), ('C2',0.1) ])
diffsol_func = lambdify(t, diffsol_repl, 'numpy')
# -> matplotlib.pyplot ...
```

# Symbolisches Programmieren – sympy

## Allg. Beispiel

Löse folgende DGL 1. Ordnung) und stelle sie grafisch dar:

$$\mathrm{x}\mathrm{y}' + \mathrm{y} = 4\mathrm{x}^3 - 2\mathrm{x}^2$$

## Harmonischer Oszillator + AddOn

Die Differenzialgleichung des harmonischen Oszillators $\mathrm{m}\ddot{\mathrm{x}} = -\mathrm{k}\mathrm{x}$ ($\omega = \sqrt{\mathrm{k}/\mathrm{m}}$) lässt sich um verschieden Kräfte “erweitern”:

| | |
|---|---|
| Stokes’sche Reibung: | $\mathrm{F}_\mathrm{R} = -\alpha\dot{\mathrm{x}}$ |
| Externe Anregung: | $\mathrm{F}_\mathrm{ext} = \mathrm{F}_0 \sin(\omega_0 \mathrm{t})$ |
| Externe Anregung: | $\mathrm{F}_\mathrm{ext} = \mathrm{F}_0 \mathrm{e}^{-\beta \mathrm{t}}$ |

Löse die entsprechende DGL 2. Ordnung und stelle die Lösung für selbstgewählte Parameter das. Welches Verhalten zeigt sich? Beachten sie die Fallunterscheidung im Fall der Reibung in Abhängigkeit von $\alpha$ zu $\omega$.

# Was bisher geschah . . .

Python

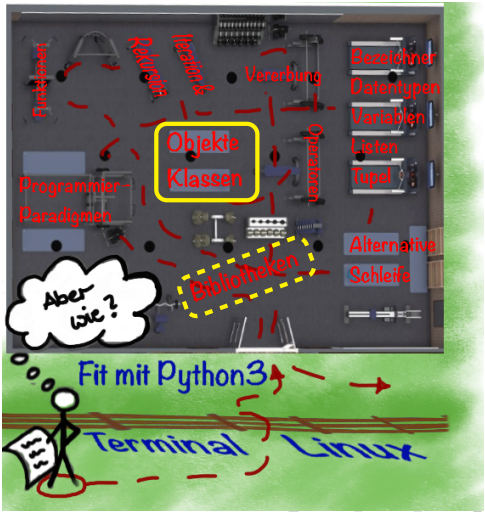
- Elementare-, Nichtelementare Datentypen
- Funktionen
- Alternativen
- Schleifen
- Grafische Darstellung (matplotlib)
- Algebraisches Rechnen (sympy)

# 10. Einheit

## Ziel

Erweiterung bisheriger Befehle

Einführung Objektorientiere Programmierung (OOP)



# Grundlagen

Wiederholung: Gültigkeitsdauer (scope) von Variablen

## Namensraum

- Zuordnung von Namen zu Objekten, in Python typ. als Dictionary intern hinterlegt
- Innerhalb eines Namensraum, Bezeichner eindeutig wählen
- Ausserhalb eines Namensraum werden Bezeichner und Namensraum kombiniert über das Symbol “`.`”
- Beispiele von bereits bekannten Namensräumen:
  - Menge aller eingebauter Funktionen (existiert sobald Interpreter startet)
  - globale Namen eines Moduls (existiert sobald Modul geladen wird)
  - lokale Name eines Funktionsaufrufs (existiert sobald Funktion aufgerufen bis sie verlassen wird)
- Anweisungen der obersten Ebene sind Teil des Moduls `__main__`

# Objektorientiertes Programmieren (OOP)

- Relativ neues Programmierkonzept aus den 60iger Jahren
- Erste OOP-Sprache: “Simula 67” von Ole-Johan Dahl und Kristen Nygaard (1967)

## Grundkonzept

- Zusammenfassen von Eigenschaften (Attributen) und Verhaltensweisen (Methoden) eines betrachteten Gegenstands (Objekts) in einer Datenstruktur (Klasse)
- Abkapseln dieser Attribute und Methoden, so dass Nutzer der Klasse oder Methoden fremder Objekte die Attribute nicht “falsch” manipulieren können

**Hinweis:** OOP ist eine Bottom-Up-Methode, während iteratives programmieren (schrittweises Verfeinern) eine Top-Down-Methode ist.

# Objektorientiertes Programmieren (OOP)

## Begriffe I

- **Objekte** – Instanzen, Exemplar:
  Abbildung realer Gegenstände mit Eigenschaften und Verhaltensweisen
- **Klassen** – Typ, Gruppierung:
  Abstraktion von Objekten mit gemeinsamen Eigenschaften und Verhaltensweisen
- **Attribute** – Eigenschaften:
  Kennzeichnende Eigenschaften eines Objekts
- **Methoden** – Funktionen, Operationen:
  Verhaltensweisen/ Aktionen eines Objekts

# Objektorientiertes Programmieren (OOP)

## Begriffe II

- **Abkapslung**:
  Schutz der Attribute / Methoden eines Objekts vor falschem Zugriff
- **Vererbung**:
  Übernahme von vorhandenen Attribute / Methoden einer Klasse in eine andere
- **Polymorphismus** – Vielgestaltigkeit:
  Bei gleichlautenden Aufforderungen werden durch Objekte verschiedener Klassen unterschiedlich verarbeitet

# Klassen allgemein

## Klassen $\leftrightarrow$ Objekt

- Klasse:
  - Implementation von “Abstrakten Datentypen”
  - Vereinen Daten (Attribute) und dazugehörige Operationen (Methoden)
  - Vorlagen (=Baupläne) nach denen Objekte während der Laufzeit angelegt werden können
    $\rightarrow$ Erst durch Ausführen (Aufrufen) haben sie Wirkung
  - Klassen definieren einen eignen Namensraum
- Objekt:
  - Instanzierung einer Klasse
  - Elemente der Klasse werden mit `.` adressiert

# Klassen allgemein

## Klassen ↔ Objekt

- (Minimal) Definition Klasse (“New Style”):

```
class <KlassenName> (object[, ...])]:
    <anweisung1>
    ...
    <anweisungN>
```

```
class Musterklasse (object):
    pass
```

- Python “New Style”: Wenn die Klasse nicht von einer Superklasse abgeleitet wird, sollte eine Basisklasse verwendet werden (typ. Basisklasse in Python ohne eigene Eigenschaften ist `object`)
- Instanzierung benutzt die Funktionsnotation:

```
<objekt> = <KlassenName>()
<objekt>.<anweisung1>
```

```
mo = Musterklasse()
```

**Hinweis:** In Python sind jeder Wert ein Objekt und hat daher eine Klasse → `<objekt>.__class__`

# Klassen allgemein

## Methoden

- Funktion, die innerhalb einer Klasse definiert werden
- Aufruf erfolgt über Objekt (`<objekt>.<methode>([...])`)
- Direkter Zugriff auf **alle** Daten und Methoden der eigenen Klasse möglich unabhängig vom Schutzniveau
- Stellen Verhaltensweisen dar, wie man korrekt auf die Attribute zugreift. Sie enthalten Berechnungen und Plausibilitätsprüfungen, stellen Datenumwandlungen zur Verfügung, testen Richtigkeit, und dienen der Fehlervermeidung (Abkapslung)

# Klassen allgemein

## Methoden

Es gibt drei Kategorien von Methoden:

- (Dynamische) Methoden: Instanzgebunden, daher benötigen sie Referenz auf Instanz `self`
- Klassenmethoden: An Klasse nicht Instanz gebunden, benötigen daher Klassen-Referenz `cls` → `@classmethod` (in Zeile vor der Methodendef.)
- Statische Methoden: An (Basis-) Klasse nicht Instanz gebunden, benötigen keine Referenz, → `@staticmethod` (in Zeile vor der Methodendef.)

# Klassen allgemein

## (Dynamische) Methoden

- Besonderheit: Erster Übergabeparameter `self` ist die Referenz auf das Objekt, dass die Methode aufruft. Dieser Übergabeparameter erscheint nicht in der Argumentenliste des Methodenaufrufs (Namenskonvention in Python legt den Bezeichner `self` fest)
- Beispiel:

```
class Musterklasse (object):
    def f(self):
        return "Hello World"

musterobjekt = Musterklasse()
print(musterobjekt.f())
```

# Klassen allgemein

## (Dynamische) Methoden – Magische Methoden I

- Es gibt einige besondere Methoden (magische Methoden), die bereits durch Python vorgesehen sind: `__<PythonBegriff>__`
- Für Klassen sollten immer eine Initialisierungsmethode definiert werden (vlg. Konstruktoren in anderen Programmiersprachen): `__init__`-Methode

```
class <klasse> (object):
    def __init__(self [, <parameter>, ...]):
        <anweisungen>
```

# Vgl. classmethod ↔ staticmethod

**Beispiel für die Nutzung von Klassen- und/oder statischen Methoden:**

```
from datetime import date
class student(object):
    def __init__(self, name='None', age=0):
        self.name = name
        self.age = age

    @classmethod
    def from_Birthyear(cls, name, year):
        study1 = cls(name, date.today().year - year)
        return study1

    @staticmethod
    def is_age_valid(age):
        return age >=16

study = student.from_Birthyear('Neo',1999)  # Erzeugung von Objekten der Klasse
valid = student.is_age_valid(study.age)     # Allg. Validitaetstest
```

# Klassen allgemein

## Attribute

- Instanzattribute
  - Bezeichner, die in Methoden via `self.<name>` verwendet werden, sind Instanzattribute dieser Klasse
  - Je nach Schutzniveau sind sie unterschiedlich erreichbar
- Klassenattribute
  - Bezeichner, die direkt in der Klasse definiert sind
  - Alle Instanzen greifen auf die gleiche Variable innerhalb der Methoden via `type(self).<name>`, d.h. sie habe für alle Instanzen den gleichen Wert
  - Zugriff von außen über Klasse `<klasse>.<name>` (empfohlen) oder <objekt>.<name> (nicht empfehlenswert)
- Je nach Schutzniveau sind sie unterschiedlich erreichbar

# Klassenattribute

## Beispiel für die Nutzung von dynamischen- und Klassenattributen

```
class student(object):
    anzStud=0                                   # Klassenattribut

    def __init__(self, name='None', age=0):
      type(self).anzStud += 1
        self.name = name                        # Dyn. Attribut
        self.age = age                          # Dyn. Attribut
        self.matrikel = type(self).anzStud      # Dyn. Attribut

study1 = student('max', 18)
print(study1.matrikel, student.anzStud)
  # student.anzStud = type(study1).anzStud
study2 = student('nora',19)
print(study1.matrikel, study2.matrikel, student.anzStud)
  # student.anzStud = type(study1).anzStud = type(study2).anzStud
```

# Klassen allgemein

## Dynamische Attribute

- Es können Objekten (auch Funktionen) jederzeit Attribute extern hinzugefügt werden
  $\rightarrow$ diese gehören dann nur zu diesem Objekt $\neq$ Instanzattributen
- Leere Klassen in Kombination mit dynamischen Attributen können als Ersatzkonstrukt für Datensätze (C/C++) verwendet werden
  $\rightarrow$ Clustern von Bezeichner in einem Namensraum, die thematisch zusammengehören

# Klassen allgemein

## Aufgabe – Punktmasse I

Der beliebteste Gegenstand in der Physik ist die Punktmasse ohne Reibung im Vakuum. Stellen Sie sich vor, dass Sie eine geradlinig gleichförmige Bewegung einer Punktmasse auf einer Linie simulieren möchten. Dafür soll eine Klasse programmiert und verwendet werden.
Ihr Objekt Punktmasse soll über den Aufruf mit einer vom Nutzer definierten Masse, einem Ort und einer Geschwindigkeit initialisiert werden.
Lassen Sie sich alle Attribute der Klasse anzeigen.

Wie können bei der `__init__`-Methode Standardwerte für Masse ($\mathtt{m} = 1$), Ort ($\mathtt{x} = 0$) und Geschwindigkeit ($\mathtt{v} = 0$) definieren?

# Klassen allgemein

## Aufgabe – Punktmasse II

Erweitern Sie Ihr Klasse Punktmasse um eine Methode, die die Koordinaten der Punktmasse entsprechend seiner Geschwindigkeit und einer gegeben (oder eingegeben) Zeitschrittweite aktualisiert.
Lassen Sie sich für $\mathbb{N}$ Zeitschritte (Schrittweite ist selbst wählbar) die Orte ausgeben.

## Aufgabe – Punktmasse III

Erweitern Sie Ihr Klasse Punktmasse um ein Klassenattribut, das die Anzahl der Objekte der Klasse Punktmasse zählt und eine Klassenmethode, die den aktuellen Wert des Klassenattributs ausgibt.
Legen Sie nun mehrere Objekte dieser Klasse an und überprüfen Sie den Inhalt des Klassenattributs.

# Zusammenfassung

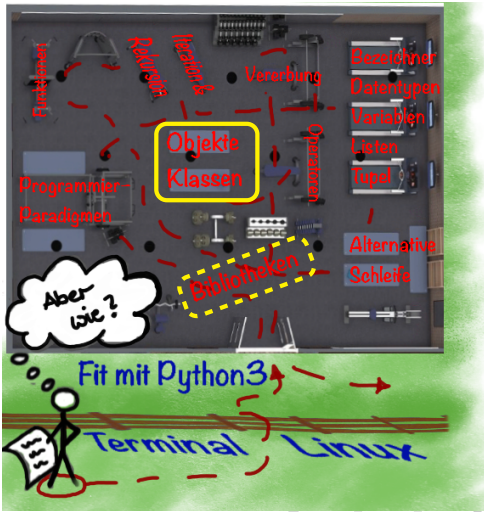
Erste Schritte in OOP

# 11. Einheit

## Ziel

Modularisieren (z.B. in OOP)

Abkapseln (in OOP)



# Abkapslung

## Ziel der Abkapslung

1. Auslagern von Funktionen und Klassen in externe Dateien (Modularisieren) → Dient auch der Wiederverwendbarkeit
2. Durch Verwendung von Schutzniveaus, soll der direkte Zugriff auf Attribute und damit eine fehlerhafte Verwendung verhindert werden
3. Einführen von `get`- und `set`-Methoden, die Plausibilitätsprüfung, Richtigkeitstests, Datenumwandlung, ... durchführen
4. Erweiterungsmöglichkeit in Python: Definition von `properties`

# Modularisieren: Module & Pakete

## Module

- Module sind ‘externe’ `.py`-Dateien, die Funktions- und Klassendefinitionen enthalten
- Module müssen sich
  - in einem der Python-Suchpfade befinden:

```
import sys
for dir in sys.path
    print(dir)
```

  - im dem selben Verzeichnis wie das aufrufende Python-Skript
- Einbinden von eigenen Modulen:

```
import <EigModul>
<EigModul>.<Funktion>
var = <EigModul>.<Klasse>()
```

# Modularisieren: Module & Pakete

## Module entwickeln

```
if __name__ == "__main__":
```

$\longrightarrow$ Trick um Module gleichzeitig als alleinstehendes Skript und als importierbares Module verwenden zu können

- `__main__`: Name des Top-Level-Programms, das ausgeführt wird
- `__name__`: Entspricht bei importierten Modulen dem Dateinamen und bei dem ausgeführten Skript `"__main__"` unabh. vom Dateinamen

# Modularisieren: Module & Pakete

## Pakete

- Zusammenfassung von mehreren Modulen
- Ordner innerhalb des Python-Suchpfades oder im Projektverzeichnis
- Bestehen aus:
  - `__init__.py`: Leere Datei oder Datei mit Initialisierungscode für das Paket
  - `<Module>.py`: Modul-Dateien die zum Paket dazugehören
  - Weitere Subordner mit Paketen

# Modularisieren: Module & Pakete

## Pakete – `__init__.py`

| Version A: | Version B: |
|---|---|
| `__init__.py`<br>leer | `__init__.py`<br>`from <EigPaket> import <EigModul>` |
| `<Program>.py`<br>`from <EigPaket> import <EigModul>`<br>`<EigModul>.<...>` | `<Program>.py`<br>`import <EigPaket>`<br>`<EigPaket>.<EigModul>.<...>` |

# Modularisieren: Module & Pakete

## Pakete – Verschachtelt

Verzeichnisstruktur

```
|- project
    |- package1
    |   |- module1.py
    |   |- module2.py
    |-package2
        |- __init__.py
        |- module3.py
        |- subpackage1
            |- module4.py
```

Verwendung in Python

```
from package1 import module1
from package1.module2 import function1
from package2 import class1
from package2.subpackage1.module5 import \
    function2
```

# Modularisieren: Module & Pakete

## Übersichtlichkeit im Quellcode

- Einbinden der Standard-Bibliotheken (dann Leerzeile)
- Einbinden von ‘Third-Party’-Bibliotheken (dann Leerzeile)
- Einbinden von eigenen Bibliotheken (dann Leerzeile)

## Aufgabe – Module/Paket Punktmasse

test.py & punktmasse/*

1. Wandeln Sie ihr bisheriges Programm mit der Klasse Punktmasse in ein ‘externes’ Modul um, dass Sie trotzdem noch als alleinstehendes Programm verwenden können. Testen Sie das einbinden Ihres Module in eine weitere Datei zu Testzwecken.
2. Erweitern Sie das Module zu einem Paket. Testen Sie die Einbindung des Pakte zunächst unter Verwendung einer leeren `__init__.py` und im Anschluss unter einer erweiterten `__init__.py`.

# Abkapslung

## Schutzniveaus

- Konzept zur Abkapslung von Daten vor “falschem/fremden” Zugriff
- In Klassen gibt es typ. drei mögliche Schutzniveaus:
  - `public`: Auf öffentliche Elemente kann programmweit zugegriffen werden (Member der Klasse, Nutzer der Klasse, Fremde Objekte, . . . )
  - `protected`: Auf geschützte Elemente kann lesend/schreibend zugegriffen werden, was jedoch nicht gemacht werden sollte. Sie sind für die Vererbung wichtig.
  - `private`: Auf private Elemente kann nur innerhalb der eigenen Klasse zugegriffen werden. Sie sind von außen nicht sicht- und benutzbar.
- Zum auf private Attribute zugreifen zu können werden `set`- (Setter) und `get`-Methoden (Getter) – Namenskonvention – eingeführt
- Getter und Setter regeln die korrekt zugriff auf private Attribute und sind typ. public

# Abkapslung

## Schutzniveaus

- Per se ist alles `public`
- Werden über Bezeichnerkonvention dargestellt:
  - `public:` `<varPub>`
  - `protected:` `_<varProt>`
  - `private:` `__<varPriv>`

```
class test:
    """ Zeigt Schutzniveaus. """
    def __init__(self):
        self.pub = "public"
        self._prot = "protected"
        self.__priv = "private"

t = test()
print(t.pub)
print(t._prot)
print(t.__priv) # Fehlermeldung!
```

# Abkapslung

## Schutzniveaus – Getter und Setter → Allg.

- Namenskonvention:

```
class <EigKlasse> (object):
  ...
  def <getX>(self):
    [<tests>]
    return self.__<x>

  def <setX>(self, <x>):
    [<tests>]
    self.__<x> = <x>
```

- Vorteil: Sicherere Verwendung der Attribute
- Nachteil: ‘Unintuitive’ Benutzung für Nutzer der Klasse

# Abkapslung

## Schutzniveaus – Properties

- Idee: Der Nutzer verwenden Attribute, die intern aber die Getter und Setter verwenden um die privaten Attribute zu verändern
- Entsprechend Zen of Python sollten entweder Getter/Setter oder Properties verwendet werden. (“There should be one – and preferably only one – obvious way to do it.”)
- Definition:
  - Allg.: `property(<fget>,<fset>,<fdel>,<"docString">)`
  - Definition privater Getter/Setter via `def __<getX>(self):` und `def __<setX>(self, <x>):`
  - Lesender und schreibender Zugriff auf Property x: `<x> = property(__<getX>, __<setX>)`
  - Lesender Zugriff: `<x> = property(__<getX>)`
- Benutzung:`<objekt>.<x>`, $\rightarrow$ z.B. `<var> = <ob1>.<x>+<ob2>.<x>` (Es wirkt wie ein direkter Attributszugriff.)
- **Hinweis:** Properties können auch komplexere Zusammenhänge darstellen und mehrere private Attribute verwenden/verändern.

# Einschub: Dekoratoren

## Dekoratoren

- Element des aspektorientiertes Programmieren
- Idee: Kernfunktionalität einer Funktion wird mit einer anderen Funktionalität erweitert
- Aufrufbares Objekt, das als Argument eine Funktion annimmt
- Mittels “@” referenzierbar
- Es gibt built-in Dekoratoren in Python (z.B.: @x.setter, @classmethod, . . . ), man kann aber auch selbst welche entwickeln.
- Zwei Arten von Dekoratoren:
  - Funktionsdekoratoren (Closures): Funktionen mit internen Funktionen
  - Klassendekoratoren (Funktoren): Objekte

# Abkapslung

## Definitionsbeispiel für Properties . . . via Dekoratoren

```
class punkt (object):
    def __init__(self,x=0):
        self.__x=x

    def getX(self):
        return self.__x
    def setX(self,x):
    # hat nur 2 Parameter
        self.__x=x
    def delX(self):
        del self.__x

    x = property(getX, setX, delX, "I'm␣
        property␣of␣X")
```

```
class punkt (object):
    def __init__(self,x=0):
        self.__x=x

    @property
    def x(self):
        return self.__x
    @x.setter
    def x(self,x):
        self.__x=x
    @x.deleter
    def x(self):
        del self.__x
```

# Abkapslung

## Aufgabe – Punktmasse V (alternativ: IV) punktmasseV.py

Verändern Sie Ihre Klasse (Modul) Punktmasse so, dass Ihre Attribute im privaten Schutzbereich definiert sind. Führen Sie sinnvolle Properties (alternativ: Getter/Setter) für alle privaten Attribute, um diese trotzdem "direkt" über das Objekt verändern/ausgeben zu können. Welche Properties sind sinnvoll?
Testen Sie Ihre neue Klasse.

**Erweiterungsoption:** Die Geschwindigkeit ihrer Punktmasse soll zufällig in jedem der $\mathbf{N}$ Zeitschritte in einem vom Nutzer vorgegebenen Geschwindigkeitsinterval $[\mathbf{v}_{\text{min}}, \mathbf{v}_{\text{max}}]$ geändert werden. Informieren Sie sich selbst, wie die `random`-Bibliothek für Zufallszahlen zu verwenden ist.

# Was bisher geschah …

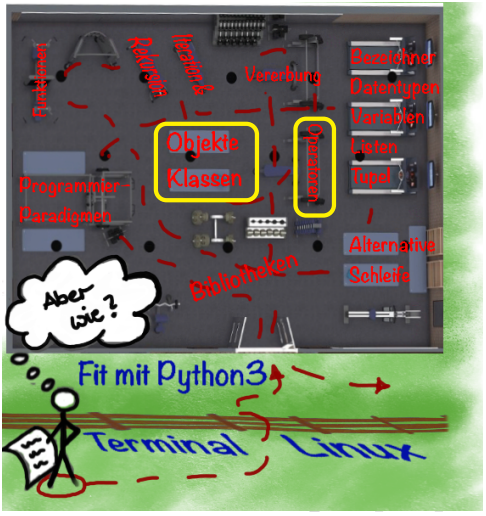
OOP

- Begriffe (Klasse, Objekte, Attribute, Methoden)
- Modularisieren
- Schutzniveaus
- Getter, Setter, Properties

# 12. Einheit

## Ziel

Magische Methoden



# Magische Methoden und Operatoren

## Besondere (magische) Methoden/ Operatoren

- Methoden, die man nicht direkt aufruft sondern die indirekt Verwendung finden
- Typische Beispiele:

| Magische Funktion | : Wirkung |
|---|---|
| `__init__` | : Direkt nach Erzeugung eines Objekts |
| `__str__` | : Erzeugung eines String eines Objektes (`str(...)`) |
| `__repr__` | : Erzeugung einer Objektrepräsentation in String-Form (`repr(...)`) |
| `__iter__` | : Erzeugt einen Objektiterator (`iter(...)`) |
| `__next__` | : Liefert den nächsten Eintrag bei einem Iterator |
| `__del__` | : Löscht ein Element |
| ... | : |
| `__add__` | : + Operator |
| `__sub__` | : - Operator |
| ... | : |

# Magische Methoden und Operatoren

## Überladung von Methoden/Operatoren

- Überlagerung, Erweiterung bzw. Anpassung bereits existierender Funktionen/ Methoden/ Operatoren für eine neue bzw. eigene Klasse
- Ursprüngliche Sinn der Funktionen/ Methoden/ Operatoren soll dabei erhalten bleiben

→ Magische Methoden

# Magische Methoden

## `__str__` ⇔ `__repr__`

Sind die Methoden `__str__` und `__repr__` für eine Klasse definiert, erzeugen sie eine String für das entsprechende Objekt.

- `__str__`
  - Aufruf durch `print`-Funktion, aber nicht bei direkter Ausgabe in der interaktiven Konsole
  - Ziel: **Lesbarkeit der Objektdaten für den Endnutzer**
  - Anwendung `eval` auf `str(<objekt>)` → String
- `__repr__`
  - Aufruf durch `print`-Funktion (wenn `__str__` nicht definiert) und bei direkter Ausgabe in der interaktiven Konsole
  - Ziel: Eindeutigkeit und Unmissverständlichkeit der Objektdaten (interne Darstellung), d.h. Quellcodezeile um Objekt mit diesen Parametern anzulegen.
  - Anwendung `eval` auf `repr(<objekt>)` → `<objektKlasse>` (wenn `__repr__` einen Objektaufruf als String ausgibt)

# Magische Methoden

## __str__ ⇔ __repr__

```
1  class punkt (object):
2      def __init__(self, x=0):
3          self.x = x
4      def __str__(self):
5          return "Punkt x={0:f}\n".format(self.x)
6      def __repr__():
7          return "punkt({0:f})".format(self.x)
8
9  pkt1 = punkt(1.5)
10 print(pkt1)        # = print(str(pkt1)) ->Ausgabe: Punkt x=1.5
11 print(repr(pkt1))  # Ausgabe: punkt(1.5)
12
13 # Anwendung fuer repr, funkt. nicht mit str
14 pkt3 = eval(input("Punkt?"))  # Wenn Eingabe: punkt(3.4)
15 print(pkt3)                   # Ausgabe: Punkt x=3.4
```

# Magische Methoden

## Aufgabe – Punktmasse VI

punktmasseVI.py

Erweitern Sie Ihre Klasse (Modul) Punktmasse um eine `__str__` und `__repr__`-Methode und testen Sie deren Anwendungsmöglichkeiten aus.
Ziel:

- Lassen Sie den Nutzer **direkt** ein Objekt der Klasse Punktmasse mit den Parameter $\mathbf{m} = 3.5, \mathbf{r} = 1, \mathbf{v} = -0.5$ eingeben.
- Diese Eingabe soll mittels `eval()` ausgewertet werden und in eine Variable gespeichert werden.
- Überprüfen Sie die korrekte Speichung des Objektes in der Variable via `print()`. (Warum reicht das als Test aus?)

# Magische Methoden

Verschiedene magische Methoden sind bei unterschiedlichen Problemen relevant.

## Bsp. für Iterator

Enthält die Klasse ein Feld, ist es sinnvoll einen Iterator, etc. zu definieren

```
 1 class feld (object):
 2     def __init__(self,data):
 3         self.data=data
 4         self.index=len(data)
 5
 6     def __iter__(self):
 7         return self
 8
 9     def __next__(self):
10         if self.index==0:
11             raise StopIteration
12         self.index=self.index-1
13         return self.data[self.index]
```

```
14 if __name__ == "__main__":
15     ob=feld([1,2,3,4,5])
16     for i in iter(ob):
17         print(i)
```

# Was bisher geschah …

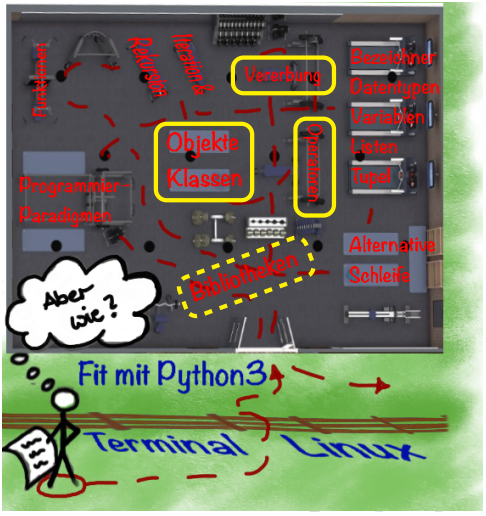
OOP

- Objekt, Klasse
- Modularisieren
- Schutzniveaus
- Getter, Setter, Properties
- Magische Methoden

# 13. Einheit

## Ziel

- Operatoren
- Vererbung



# Operatoren

## Operatorüberladung

- Binäre Operatoren, Erweiterte Zuweisungen und Vergleich:
  `def __<operator>__(self, other):`
  - `a + b` mit a $\in$ Klasse A und b $\in$ Klasse B $\rightarrow$ `a.__add__(b)`
  - Das aufrufende Objekt a definiert die Klasse, aus der `__add__` gerufen wird
  - Meist ändert der Operator nicht einen der Operanden, sondern gibt die Ergebnisse in einem neuen Objekt zurück. Dies ist mit dem Befehl `type(self)(<var-Liste>)` ($\rightarrow$ Konstruktor-Aufruf für Objekt vom Type `self`)

```
def __add__(self, other):
  tmp = self.m+other.m
  return type(self)(tmp)   # erzeugt Objekt vom Typs self
```

# Operatoren

## Operatorüberladung

- Binäre Operatoren, Erweiterte Zuweisungen und Vergleich:
  `def __<operator>__(self, other):`
  - **Hinweis:** Ist die aufrufende Klasse eine Standardklasse (int, float, ...), kann darin kein neuer Operator für das eigene Probleme definiert werden. Dann kann in der eigenen Klasse `__r<operator>__`-Operatoren (reverse) definieren, die die Reihenfolge der Operanden tauschen. Das geht nur, wenn die Operation kommutativ ist.

    ```
    def __radd__(self, other):
      return <EigKlasse>.__add__(self, other)
    ```
- Unäre Operatoren: `def __<operator>__(self):`
  (pos, neg, abs, invert (=not), complex, int, float, long, oct, hex, bin)

# Operatoren

## Aufgabe – Punktmasse VII

Erweitern Sie Ihre Klasse (Modul) Punktmasse um folgende Operatoren:

- Addition zweier Punktmassen: Massen und Geschwindigkeiten werden addiert, Wenn Punktmasse unterschiedlichen Orte haben, wird der mittlere Ort beider Punktmassen, der neue Ort
- Multipliziere/Dividiere Punktmasse mit einem Skalar und umgekehrt: Vergrößert Masse und Geschwindigkeit der Punktmasse um Faktor Skalar

Recherchieren Sie typische Implementationsmöglichkeiten für Operatoren selbstständig.
Sie testen Sie alle implementierten Operatoren.

# Magische Methoden und Operatoren

## Aufgabe – Komplexe Zahlen

Entwerfen und Implementieren Sie eine Klasse Komplex zur Benutzung und einfachen Verwendung von komplexen Zahlen. Schreiben Sie Ihre Klasse so, dass Sie das Skript auch jederzeit als Modul verwenden könnten.

- ?
- ?
- ?
- ?

# Magische Methoden und Operatoren

## Aufgabe – Komplexe Zahlen

Entwerfen und Implementieren Sie eine Klasse Komplex zur Benutzung und einfachen Verwendung von komplexen Zahlen. Schreiben Sie Ihre Klasse so, dass Sie das Skript auch jederzeit als Modul verwenden könnten.

- Anlegen
- Ausgabe
- Properties
- Operationen: binär (z.B.: +, -, *, /), revers (z.B.: `int*complex`), unär (z.B.: =, not, $|z|$)

# Magische Methoden und Operatoren

## Aufgabe – Komplexe Zahlen

Details:

- Der Nutzer sollte Objekte mit keinem und mit zwei Übergabeparameter (Realteil, Imaginärteil) anlegen können. Übergibt er keine Werte sollte das Objekt $0 + \mathrm{i}0$ liefern.
- Es sollte eine sinnvolle Ausgabe (im mathematischen Stil) für die Klasse erzeugt werden, wenn der Nutzer `print()` auf ein Objekt der Klasse Komplex anwendet. Außerdem soll der Nutzer soll `eval()` für diese Klasse verwenden können.
- Der Nutzer soll einfach auf den Real- und Imaginärteil zugreifen können, jedoch soll er diese nicht direkt ändern können.
- Der Nutzer möchte die Grundrechenoperationen (+, -, *,/) zwischen zwei Objekten der Klasse Komplex anwenden können. Außerdem soll die Multiplikation mit einem Skalar und die Berechnung des Betrags einer komplexen Zahl möglich sein.

Testen Sie (nach und nach) alle von Ihnen implementierten Methoden.

# Vererbung

## Vererbung/ Ableiten

- Ableiten einer spezialisierten Subklasse (Unterklasse, abgeleitete Klasse) von einer bereits bestehenden Klasse (Oberklasse, Basisklasse)
- Für abgeleitete Klassen gilt:
  - Vorhandene Variablen und Methoden könne direkt verwendet werden (unter Berücksichtigung der Schutzniveaus)
  - Neue Variable und Methoden können hinzugefügt werden
  - Vorhandene Methoden können redefiniert werden (Überschreiben)
- “Neues” Schutzniveaus **protected**: Geschützte Membervariablen und Methoden, auf die innerhalb der eigenen Klasse oder der Vererbungslinie zugegriffen werden kann (in Python nur Namenskonvention)
- Vererbungsregeln der Schutzniveaus sind zu berücksichtigen

# Vererbung

## Vererbungsregeln

- Schutzniveau der Unterklasse können explizit für neue Attribute/ Methoden gesetzt werden
- Schutzniveaus der vererbten Methoden hängen an Vererbungsregeln
- Je nach Programmiresprache gibt es klare die **Vererbungsregeln** `privat, protected, public`, durch die die **Schutzniveaus der Oberklasse** beim Übertrag auf die Unterklasse redefiniert werden.
  In python wird nur `public` vererbt, daher betrachten wir hier nur diese.

# Vererbung

## Vererbung in Python

- Vererbung in Python public – Definition: `class <uklasse> (<oklasse> [,...]):`
- Es werden per se alle Elemente vererbt, Zugriffsrechte hängen aber an Vererbungsregel → public
- In der Unterklasse werden nur Re- oder Neudefinition “genannt” (definiert).
- Zugriff auf Methoden der Oberklassen: `<oklasse>.<Methode>()` oder `super().<Methode>()`

# Vererbung

## Vererbungsregel public

Konsequenzen der Vererbungsregel:

- `private`-Member aus `oklasse` sind in `uklasse` verborgen aber indirekt über vererbte `public`-Methoden/Properties zugreifbar
- `public`-Member aus `oklasse` bleiben `public` in `uklasse`
- `protected`-Member sind wie `public`, aber an Namenskonvention wird deutlich, dass man als Nutzer diese Attribute/ Methoden nicht benutzen soll

# Vererbung

## Allg. Beispiel Vererbung

```
class A (object):    # Oberklasse
    def __init__(self, a=0):
        self.__a=a

class B (A):         # Unterklasse
    def __init__(self, b=0):
        A.__init__(self, 2*b)  # Init der privaten Att. von A
        self.__b=b             # Erweiterung der Unterklasse

a=A()
b=B(3)
```

# Vererbung

## Besondere Effekte bei der Vererbung

- Auswirkung: Instanzmethode $\leftrightarrow$ `@classmethod` $\leftrightarrow$ `@staticmethod`
  - Instanzmethode ist an Instanz und deren Daten gekoppelt (`self`)
  - `@classmethod` ist and Klasse (`cls`) des aufrufenden Objekts und deren Eigenschaften gekoppelt, kann nicht direkt auf Daten einer Instanz zugreifen
  - `@staticmethod` ist analog zu Funktion unabhängig von Klasse

# Vererbung

## Instanzmethode ↔ @classmethod ↔ @staticmethod

```
1  class A (object):
2      def __init__(self, a=0):
3          self.__a=a
4
5      def print_ob(self, t="hallo"):
6          print(self.__a, t)
7
8      @classmethod
9      def print_class(cls, t="hallo"):
10         print(cls, t)
11
12     @staticmethod
13     def print_stat(t="hallo"):
14         print("---",t)
```

```
16 class B (A):
17     def __init__(self, b=0):
18     # Init der privaten Att. von A
19         A.__init__(self, 2*b)
20     # Erweiterung der Unterklasse
21         self.__b=b
22
23 a=A()            # self.__a=0
24 b=B(3)           # self.__b=3, self.__a=6
25 a.print_ob("test1")     # 0 test1
26 b.print_ob()            # 6 hallo
27 a.print_class()         # <class '__main__.A'> hallo
28 b.print_class("test2")  # <class '__main__.B'> test2
29 a.print_stat("test3")   # --- test3
30 b.print_stat()          # --- hallo
```

# Vererbung

## Mehrdeutigkeit – “Diamantproblem”

**Beispiel 1:**

```
class A:
    def m(self):
        print("m_von_A_ruft")
class B(A):
    def m(self):
        print("m_von_B_ruft")
class C(A):
    def m(self):
        print("m_von_C_ruft")
class D(B,C):
    pass

x = D()
x.m()   # m von B ruft
```

**Beispiel 2:**

```
class A:
    def m(self):
        print("m_von_A_ruft")
class B(A):
    pass
class C(A):
    def m(self):
        print("m_von_C_ruft")
class D(B,C):
    pass

x = D()
x.m()   # m von C ruft
```

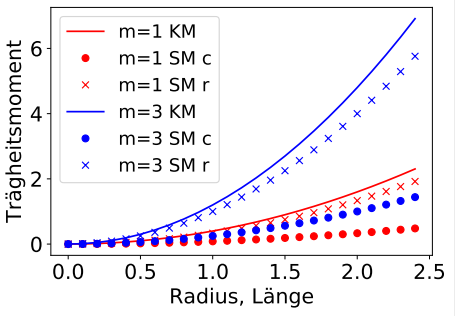
# Vererbung

## Aufgabe – Kugel vs. Stabmasse

tragMom.py, kugelmasse.py, stabmasse.py

Ziel dieser Aufgabe ist es, ein Masse-Trägheitsmoment-Diagramm zu erstellen, um den Unterschied zwischen verschiedenen Körpern und deren Trägheitsmomente graphisch zu veranschaulichen. Dazu soll der Radius bzw. die Länge des Stabs für z.B. zwei verschiedene Massen variiert werden. Programmiertechnisch ist der Fokus auf die Vererbung zu legen.

(Details nächste Folien)



# Vererbung

## Aufgabe – Kugel vs. Stabmasse

kugelmasse.py, stabmasse.py

Starten Sie von Ihrem Modul Punktmasse aus Aufgabe Punktmasse VI (oder VII). Leiten Sie von dieser Klasse zwei neue Klassen ab, die Kugelmasse und die Stabmasse. Die Subklassen sollten in einem eigenen Modul definiert werden.

1. Definieren Sie zunächst nur eine `__init__` und eine print-Methode (print, `__str__`, oder `__repr__`) um den Zugriff auf die vererbten Attribute (public, privat, protected) zu testen.
2. Erweitern Sie die print-Methode so, dass sie explizit die print-Methode der Oberklasse aufrufen.
3. Implementieren Sie ein print-Methode ein, die die korrekte Klassenbezeichnung des aufrufenden Objektes ausgibt (Programmiertechnischer Minimalaufwand).

# Vererbung

## Aufgabe – Kugel vs. Stabmasse

kugelmasse.py, stabmasse.py

4. Kugelmasse:
   - Erweitern Sie nun die Klasse Kugelmasse um das private Attribut Kugelradius (+ notwendige Methoden/Properties um mit diesem Attribut zu “arbeiten”)
   - Fügen Sie eine Methode hinzu, die das Trägheitsmoment der Kugelmasse berechnet, wenn die Rotationsachse durch den Kugelmittelpunkt verläuft.
5. Stabmasse:
   - Erweitern Sie nun die Klasse Stabmasse um das private Attribut Stablänge (+ notwendige Methoden/Properties um mit diesem Attribut zu “arbeiten”)
   - Fügen Sie eine/ zwei Methode/n hinzu, die das Trägheitsmoment der Stabmasse berechnet, wenn die Rotationsachse Senkrecht zum Stab zum einen in der Mitte des Stabs und zum Anderen am Ende des Stabs verläuft.

Testen Sie alle Attribute und Methoden aus.

# Vererbung

## Aufgabe – Kugel vs. Stabmasse (Zusatz) kugelmasse.py, stabmasse.py

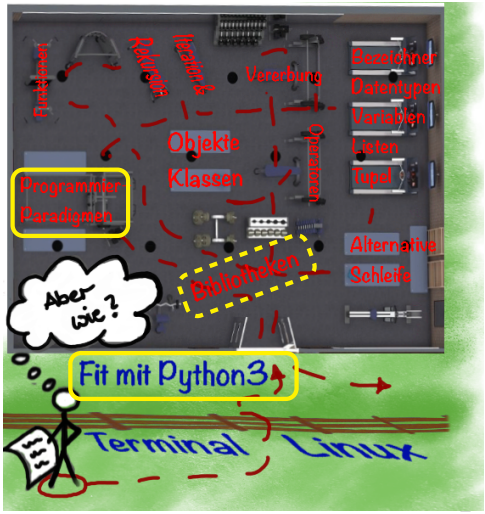
6. Untersuchen Sie selbstständig wie die Vererbung auf die Klassenattribute wirkt. Gilt das Klassenattribut auch für Subklassen oder wird es neu “angelegt”?
7. Wie wirkt das Schutzniveau `protected`?

# 14. Einheit

## Ziel

**Evaluation**

Python
→ aspektorientiert
→ funktional



# Funktionale Programmierung

## Programmierkonzepte in Python

- imperativ/prozedural
- objektorientiert
- **funktional**
- aspektorientiert

## Wdh.: Funktionale Programmierung

Programmieren mit mathematischen Funktionen. Mathematische Funktionen sind Funktionen die bei gleichem Eingabewert immer den gleichen Rückgabewert liefern. Sie können als Abfrage einer (unendl. ) großen Tabelle gesehen werden.

# Funktionale Programmierung

## Elemente einer Funktionale Programmiersprache

- **First-class functions**: Funktionen können als Argument/Parameter an Funktionen übergeben und als Rückgabewert zurückgegeben werden.
- **lambda Funktionen** (**Anonyme Funktionen**)
- **Funktionen höherer Ordnung**: `map()`, `filter()`, `reduce()`
- Closures (siehe Dekoratoren Folie 290)
- List-Comprehension (siehe Folie 187)
- Generatoren, Iteratoren

# Funktionale Programmierung in Python

## Bibliotheken für Funktionale Elemente

- `operator`: Operatoren zur Verarbeitung von Sequenzen
- `functools`: Erzeugung von Funktionen höherer Ordnung
- `itertools`: Erstellung von Iteratoren aus Sequenzen

# Funktionale Programmierung in Python

## lambda Funktionen (Anonyme Funktionen)

- lambda Funktionen bzw. anonyme Funktionen, sind Funktionen / Funktionskörper ohne Namen
- Sie haben (beliebig viele) Übergabeparameter (Argumentenliste), führen einen Ausdruck aus, der gleichzeitig Rückgabewert der Funktion ist.
- Definition: `lambda <Argument> : <Ausdruck>`
- Anwendungsbeispiel:

```
(lambda x : x*21)(2)
y = (lambda x : x*x)(3) - 10
for i in range(10):
    (lambda x : x+42)(i)
    (lambda x : x+i)(42)
f = lambda x : x*x-1  # Verwendung wie gewoehnliche Funktion
f(1)
```

# Funktionale Programmierung in Python

## Funktionen höherer Ordnung

- Dienen der Verarbeitung von Listen
- `map()`: Anwenden einer Funktion auf jedes Element der Liste
  - Definition: `r=map(<func>,<seq>)` $\leftrightarrow$ *`[<func> for <i> in <seq>]`*
- `filter()`: Anwenden einer Funktion auf jedes Element und Rückgabe des Elements, wenn Funktion `wahr` liefert
  - Definition: `r=filter(<func>,<seq>)` $\leftrightarrow$ *`[<func> for <i> in <seq> if <cond>]`*

# Funktionale Programmierung in Python

## Übersichtlichkeit durch lambda Funktion, map(), und List-Comprehension

```
1  def fahrenheit(T):
2      return ((9.0/5)*T+32)
3  def celcius(T):
4      return ((5.0/9)*(T-32))
5
6  def Fahrenheit2Celcius(F_liste):
7      erg=[]
8      for F in F_liste:
9          erg.append(celcius(F))
10     return erg
11 def Celcius2Fahrenheit(C_liste):
12     erg=[]
13     for C in C_liste:
14         erg.append(fahrenheit(C))
15     return erg
```

```
17 temp=(36.5, 37, 37.5, 39)
18
19 F_liste = Celcius2Fahrenheit(temp)
20 C_liste = Fahrenheit2Celcius(F_liste)
```

# Funktionale Programmierung in Python

## Übersichtlichkeit durch lambda Funktion, map(), und List-Comprehension

```
1  temp=(36.5, 37, 37.5, 39)
2
3  #Funktionale Programmierung - lambda, map()
4  F_liste = list(map(lambda C: (9.0/5)*C+32, temp))
5  C_liste = map(lambda F: (5.0/9)*(F-32),F_liste)
6
7  # Funktionale Programmierung - ListComprehension
8  F_liste = [(9.0/5)*C+32 for C in temp]
9  C_liste = [(5.0/9)*(F-32) for F in F_liste]
```

# Funktionale Programmierung in Python

## Aufgabe – Buchhandlung

buecher.py,buecher.txt

In einem Onlinebuchhandel gibt es folgende Liste mit Bestellungen

| Bestellnummer | Buchtitel, Autor | Anzahl | Einzelpreis (EUR) |
|---|---|---|---|
| 254 | Per Anhalter durch die Galaxis, D. Adams | 7 | 12 |
| 6133 | Mr. Tompinks seltsame Reisen, G. Gamov | 1 | 44.99 |
| 7323 | What if?, R. Munroe | 5 | 14.99 |

in einem Abrechnungsprogramm.

Schreiben Sie ein Python-Programm, dass aus dieser Liste eine Liste mit Zweier-Tupel extrahiert, die jeweils die Bestellnummer und den Gesamtpreis (Anzahl$*$Einzelpreis) besteht.wie folgt Dabei soll beachtet werden, dass der Mindestbestellwert von 50 EUR pro Bestellung einzuhalten bzw. zu zahlen ist.

# Funktionale Programmierung in Python

## Aufgabe – Buchhandlung II

buecher2.py,buecher2.txt

Erweitern Sie Ihr Programm aus der vorangegangenen Aufgabenstellung.
Dieses sieht die Struktur der Liste pro Bestellung wie folgt aus: [Bestellnummer, (Artikelnummer, Anzahl, Einzelpreis), ..., (Artikelnummer, Anzahl, Einzelpreis)] Schreibe wieder ein Programm, das eine Liste mit Zweier-Tupel (Bestellnummer, Gesamtpreis) ausgibt.

*Hinweis:* Erarbeiten Sie sich selbstständig die `reduce()`-Funktion in Python. Welche sinnvolle Alternative gibt es zu `reduce()`?

# Funktionale Programmierung in Python

## Iteratoren

- **Iterator**: Ist ein Zeiger auf Elemente einer Menge, deren Datenstruktur nicht bekannt sein muss. Er erfüllt das Iteratorprotokoll, d.h. er bietet die Methoden `__iter__` und `next()` an.
- Beispiel für Iterator:

```
todoListe = ["Praktikumsbericht", /
    "Einkaufen", "Ausdauerlauf"]
for todo in todoListe:
    print(todo)
```

```
todoListe = ["Praktikumsbericht", /
    "Einkaufen", "Ausdauerlauf"]
iterator = iter(toDo)
while True:
    try:
        todo = next(iterator)
    except StopIterator:
        break
    print(todo)
```

# Funktionale Programmierung in Python

## Generatoren

- **Generator**: Sind besondere Funktionen, die intern eine Liste erzeugen, aber nicht explizit Werte sondern einen Iterator zurück geben.
- Definition ähnlich wie Funktionen, nur `yield` statt `return`
- Definition von Generator definiert automatisch `__iter__` und `next()`
- Generatoraufruf liefert Iterator-Objekt, aber nicht explizite Werte, zurück
- `next`-Methode durchläuft Iterator-Objekt bis nächsten `yield`, danach wird Iterator unterbrochen nicht beendet (Werte werden zwischengespeichert)
- Iterator wird erst beendet, wenn Iteratorkörper vollständig abgearbeitet wurde → `IteratorStop` (Error-Handling)

*Alternative:* **Iteratorklassen** – Klasse, in der die Methoden `__iter__`, `__next__`, etc. für das Handling der internen Daten selbst definiert werden kann (siehe F. 254).

# Funktionale Programmierung in Python

## Generatoren – Beispiel

```
1  def count_gen(n):
2      count = 0
3      while n > count:
4          yield count
5          count += 1
```

```
7  gen1 = count_gen(5)
8  gen2 = count_gen(2)
9  print(next(gen1))
10
11 for i in gen1:
12     print(gen1)
13     next(gen2)
```

- Was wird innerhalb der for-Schleife ausgegeben?
- Was passiert in Zeile 13?
- Was passiert wenn man `print(gen1)` durch `print(next(gen1))` ersetzt?

# Funktionale Programmierung in Python

## Aufgabe – Generator

Programmieren Sie eine Generator, eine Iterator auf eine Liste der Werte $n!$ von 1 bis $n$ erzeugt. Testen Sie Ihren Generator.

*Zusatz:* Wie könnte ein Generator für die Erzeugung einer Reihe der ersten $n$ Fibonacci-Zahlen oder Primzahlen aussehen?

# Aspektorientierte Programmierung

## Programmierkonzepte in Python

- imperativ/prozedural
- objektorientiert
- funktional
- **aspektorientiert**

## Wdh.: Aspektorientierte Programmierung

Man unterscheidet zwischen Komponenten (Systemeigenschaften, die in verallg. Funktionen “abgekapselt” werden können) und komponentenübergreifende Aspekte (können nicht abgekapselt werden). Erhöht die Modularität der objektorientierten Programmierung.

# Aspektorientierte Programmierung in Python

## Dekoratoren

- Idee: Kernfunktionalität einer Funktion wird mit einer anderen Funktionalität erweitert
- Aufrufbares Objekt, das als Argument eine Funktion annimmt
- Mittels “@” referenzierbar
- Es gibt built-in Dekoratoren in Python (z.B.: @x.setter, @classmethod, . . . ), man kann aber auch selbst welche entwickeln.
- Zwei Arten von Dekoratoren:
  - Funktionsdekoratoren (Closures): Funktionen mit internen Funktionen
  - Klassendekoratoren (Funktoren): Objekte

# Aspektorientierte Programmierung in Python

## Typische Anwendung von Dekoratoren

- Überprüfung von Argumenten (Datentypen)
- Zählen von Funktionsaufrufen
- “Erinnern” (Memoisation) von Werten

# Aspektorientierte Programmierung in Python

## Klassendekoratoren (Funktoren)

Definition:

```
class Decorator (object):
    def __init___(self,func)
        self.__func=func

    # Neue magischen Methode
    def __call__(self, *__args, **__kw)
        print("Hello", self.__func)
        try:
            return self.__func(*__args,**__kw)
        finally:
            print("Good_Bye", self.__func)
```

Wichtig: Error-Handling verwenden

Anwendung:

```
@Decorator
def tDec(a, *b, **c):
    print(a,b,c)

tDec(1,2,l=5)
```

# Aspektorientierte Programmierung in Python

**Anwendungsbeispiel:** “Sparen” von Rechenzeit bei der Berechnung von Fibonacci-Zahlen

## Aufgabe – Fibonacci-Zahlen

fib.py

Programmieren Sie die Berechnung der $\mathbf{n}$ten Fibonacci Zahl (typ. rekursive), anhand der Rechenregel

$$\mathbf{fib}(\mathbf{n}) = \begin{cases} \mathbf{fib}(\mathbf{n}-1) + \mathbf{fib}(\mathbf{n}-2) & \mathbf{n} \geq 2 \\ 1 & \mathbf{n} = 1 \\ 0 & \mathbf{n} = 0 \end{cases} .$$

Berechnen Sie die Fibonacci-Zahlen für $\mathbf{n} = [0, 40]$ in Zweischritten. Lassen Sie sich die Rechenzeit für jede Berechnung ausgeben.
*Hinweis:* Bibliothek `time`

# Aspektorientierte Programmierung in Python

## Beispiel-Dekorator für Cachespeicherung

Was benötigen wir initial?

- ...
- ...

Was soll beim Aufruf passieren? Wert ist ...

- vorhanden: ...
- nicht vorhanden, ...
- nicht vorhanden, ...

# Aspektorientierte Programmierung in Python

## Beispiel-Dekorator für Cachespeicherung

Was benötigen wir initial?

- Namen der Funktion, auf die wir Dekorator anwenden
- Leeres Dict für Cache (`{ <key>:<val>, ...}`)

Was soll beim Aufruf passieren? Wert ist ...

- vorhanden: Wert aus Cache verwenden
- nicht vorhanden, aber speicherbar (`KeyError`): Wert berechnen, speichern und verwenden
- nicht vorhanden, aber nicht speicherbar (`TypeError`): Wert berechnen und verwenden

# Aspektorientierte Programmierung in Python

## Aufgabe – Dekorator `memorized` fibDekorator.py

Implementieren Sie eine Dekorator (z.B. `memorized`) für die Cachespeicherung bereits berechneter Funktionswerte einer Funktion. Wenden Sie diesen Dekorator anschließend auf ihre Fibonacci-Funktion an. Was stellen Sie bei den Rechenzeiten fest?

Weitere Details und Erweiterungsmöglichkeiten z.B. unter `www.grimm-jaud.de/index.php/private-vortraege/39-dekaratoren-in-pyhton` (Abschnitt Beispiel)

# Zusammenfassung

- Aspektorientierte Programmierung
- Funktionale Programmierung

# Ende des ersten Semesters

Vielen Dank
für Eure Mitarbeit
und euer Engagement.

Ich freue mich
auf das nächste Semester
mit Euch!