Python

* Interpretersprache (Zwischendateien (mpc), Zwischencode)
* Plattformunabhängigkeit (weil der Code wird in Zwischencode und dann in Maschinencode umgewandelt wird) 🡪 wir brauchen einen plattformunabhängigen Interpreter!
* umfangreiche Standardbibliotheken
* PSF-Lizenz (open source, Python Code überall einbauen, frei zum Weiterbearbeiten), offener als GNU
* Rapid Prototyping: schnell und sofort Prototypen bauen, weil man kann auch funktional und iterativ programmieren (man muss kein Klassen schreiben, sondern man kann alles auf eine Seite schreiben, …)
* Garbage Collection: Daten, die für eine längere Zeit nicht verwendet werden, werden automatisch gelöscht; Verfahren zur Speicherbereinigung und Wiederfreigabe von belegtem Speicherplatz, der aber vom jeweiligen Computerprogramm nicht mehr benötigt wird
* Instanz/Referenz/GarbageCollection/del/Referenzzähler
* keine Datentypen (type): während der Laufzeit (dynamisch) werden Datentypen klar
* funktionale und objektorientierte Programmierung möglich
* Python2/Python3 (in P3 mit Abwärtskompabilität gebrochen) 🡪 immer beachten ob Python 2 oder 3
* <https://www.learnpython.org/>
* <https://www.programiz.com/python-programming>

**Unterschiede zu Java:**

* Datentypen dynamisch vergeben
* Einrückungen (4 Leerzeichen (Empfehlung von Prof. und die ganze Welt macht das auch so), wenn man z.B. mit Editor arbeitet, keine Tabulatoren 🡪 man muss aber immer die gleiche Einrückung verwenden, aslo wenn man 4 hat immer 4 verwenden, sonst Fehler) und ‘:‘ statt geschweifte, … Klammern
* keine main-Methode 🡪 man kann alles in einer Wurst schreiben (ob es gut istm ist eine andere Frage)
* Java ist streng typisiert; python hat typen hat, ist aber nicht streng 🡪 typen können bzw. ändern sich zur Laufzeit
* In python kann man funktional programmieren (keine Klassen notwendig)
* Alle Variablennamen, Methodennamen,… mit Underscore für Trennung

**Binary Search:**

* <https://www.programiz.com/dsa/binary-search>

**Allgemein:**

* Referenz = Addresse die auf den Value zeigt
* Call by Reference: wir übergeben nur die Referenz zum Value
* Call by Value: wir übergeben wirklich den Wert bzw. Inhalt

Steuerzeichen wie \n oder \t (Tabulator) 🡪 Betriebssystem wertet das Steuerzeichen aus und gibt es dann je nachdem am Bildschirm aus

(int)variable … nennt man parsen oder casting

Unterschied zwischen Fehler und Exception:

Systemfehler sind Fehler im System und Exceptions sind Programmfehler (im Code, die durch den Entwickler erzeugt werden)

In Python sagt man zu den eckigen Klammern-Listen 🡪 sind im Hintergrund arrays, die wie ArrayLists in Java funktionieren

‘ ‘ … einfache Anführungszeichen sind immer mächtiger als “ “-Anführungszeichen

**Variablen sind case sensitive**

Merke: Nie fixe Zahlen in Methoden verwenden 🡪 als Parameter übergeben

**Interaktiver Modus = Shell 🡪 ‘python‘ eingeben (https://www.python-kurs.eu/python3\_interaktiv.php)**

* Wenn man z.B. hallo ich bin ‘xy‘ ausgeben lassen möchte 🡪 print(‘hallo ich bin \‘xy\‘)
* Z.B. a=5 🡪 id(a) … Speicheradresse der Zahl; a referenziert auf die Speicheradresse:

>>> id(a)

140708733244328

>>> a=6

>>> id(a)

140708733244360

**Alle primitiven Datentypen und tupl‘s sind immutable**

Das ist ein **immutable (Speicherallokation)** Datentyp (im Stack) 🡪 wenn sich der Wert einer Variable ändert dann wird ein neuer Speicherplatz (neue Speicheradresse) festgelegt (aufgrund von wenig Speicherplatz wird auf eine bestehendes Speicherplatz referenziert verwendet (siehe Pooling = wenn der gleiche Wert in einem Speicherplatz ist wird nicht ein neuer Speicherplatz erzeugt sondern es wird auf den Speicherplatz mit dem gleichen Wert referenziert) und die Referenz zeigt nicht mehr auf die alte Speicheradresse, sondern auf die neue Speicheradresse, aber die alte Speicheradresse mit dem alten Wert bleibt auch bestehen

z.B. man legt test = 5 an 🡪 mit dem Namen test wird auf die Speicheradresse referenziert, indem auch der Wert gespeichert ist, sund wenn man test ändert, dann passiert das was ich beschrieben habe bei immutable

* **mutable** Datentypen (im Heap) 🡪 am selben Speicherplatz (Speicheradresse) wird verändert 🡪 es wird keine neue Speicheradresse bzw. Speicherplatz erzeugt 🡪 alle Objekte bzw. Klassen und auch Listen,… sind mutable

>>> l = [1,3]

>>> id(l)

1878821282560

>>> l2 = l

>>> id(l2)

1878821282560

>>> l2[0]=99

>>> l

[99, 3]

>>> l3=l2

>>> l3

[99, 3]

* **Pooling** (Pooling = wenn der gleiche Wert in einem Speicherplatz ist wird nicht ein neuer Speicherplatz erzeugt sondern es wird auf den Speicherplatz mit dem gleichen Wert referenziert 🡪 es wird auf die gleiche Speicheradresse referenziert (aus Effektivitätsgründen)):

>>> a

6

>>> b=5

>>> b=6

>>> id(a) == id(b)

True

>>> b=7

>>> id(a) == id(b)

False

>>> a= ‘fünf’

>>> b= ‘fünf’

>>> id(a) == id(b)

False

>>> a= ‘fuenf’

>>> a= ‘fuenf’

>>> id(a) == id(b)

True

**Was ist besser mutable oder immutable?**

Um **Seiteneffekt (-Bugs) (entstehen bei mutablen Datentypen)** (googln!)vermeiden 🡪 immutable Datentypen sind „besser“ (in der idealen Welt verwenden wir immutable Datentypen)

Seiteneffekt (ist schlecht) laut Prof.: wenn man ein mutablen Datentyp verändert, dann wird er überall verändert 🡪 Seiteneffekt ist in den meisten Fällen negativ

**Datentypen (ganze Zahlen (Division?), Gleitkommazahl, Zeichenketten(addieren), Listen, Dictionarys**

>>> l3

[99, 3]

>>> l2

[99, 3]

>>> l2 + l3

[99, 3, 99, 3]

>>> l3\*10

[99, 3, 99, 3, 99, 3, 99, 3, 99, 3, 99, 3, 99, 3, 99, 3, 99, 3, 99, 3]

>>> s1 = 'hallo1'

>>> s2 = 'hallo2'

>>> s1+s2

'hallo1hallo2'

>>> tup1 = (1,2)

>>> type(tup1)

<class 'tuple'>

>>> tup1[0] = 99

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

**Floor-Division** (<https://www.codingninjas.com/studio/library/floor-division-in-python>):

Floor-Division sliced (schneidet ab), nicht runden!!! 🡪 z.B. 9.99 🡪 wird zu 9

10//3 = 3

Variablen sind case sensitive

>>> simma = 5

>>> SIMMA = 6

>>> simma

5

>>> SIMMA

6

>>>

a / b … Fließkommadivision und a // b … Floor-Division

**Schlüsselwörter** (<https://openbook.rheinwerk-verlag.de/python/A_001.html>) 🡪 es gibt erfreulich wenige davon

**Dateiendungen: \*.py**

sind nur für uns Menschen, denn das Betriebssystem weiß es schon im Hintergrund

**Shebang:** <https://www.delftstack.com/de/howto/python/python-shebang/>

#!/usr/bin/env python3 (MagicLine UNIX) … wenn der Interpreter nicht angegeben wurde dann kann man das so machen

**Kommentare:** # oder mehrzeilig: “““ … “““

**pass-Anweisung:** nur für Compiler ein Hinweis, dass an dieser Stelle eine Codestelle sein sollte 🡪 kann man dann später bearbeiten

**Auswertungsreihenfolge der Operatoren:** <https://cito.github.io/byte_of_python/read/operator-precedence.html>

* Jede Sprache hat eine eigen Auswertungsreihenfolge

**Referenzierung:** <https://openbook.rheinwerk-verlag.de/python/07_001.html>

* Referenz/Instanz(besteht aus Identiẗat/Typ/Wert)
* Identität ist Ganzzahl und wie ein Fingerabdruck vergleich mit is
* type benutzen

**Instanzen freigeben (Garbagge collector mit im Spiel laut Prof.):** <https://openbook.rheinwerk-verlag.de/python/07_002.html#u7.2>

* id(ref1) == id(ref2) oder ref1 is ref2I
* Instanzen freigeben mit del
* del … mit Vorsicht zu genießen!

**Garbage collector:**

* wenn es uns interessiert 🡪 anschauen, kommt nicht zum Test
* speichert in einer Tabelle die Variablen mit den Referenzen 🡪 nach dem Kompilieren weiß der Garbage collector, ob auf eine Variable wieder unten referenziert wird im Code, dann wird dieser Speicher freigegeben 🡪 alle nicht erreichbaren Speicher werden freigegeben
* <https://www.geeksforgeeks.org/garbage-collection-python/>
* <https://stackify.com/python-garbage-collection/>

**mutable/immutable:**

* <https://openbook.rheinwerk-verlag.de/python/07_003.html#u7.3>

**tuple/range: tuple => unveränderbare Liste**

* <https://openbook.rheinwerk-verlag.de/python/13_001.html#u13>

<https://www.data-science-architect.de/gruende-fuer-python/>

**Compiler-Interpreter** <https://www.data-science-architect.de/python-compiliert-interpretiert/> (Achtung das Diagramm Entwicklungszeit,… ist nicht richtig); es gibt mehrere Interpreter mit je nach Anwendung gibt es Vor- und Nachteile

**Mutable/immutable** <https://www.data-science-architect.de/>

**Deep copy** <https://www.python-kurs.eu/python3_deep_copy.php> ... Sollen wir unbedingt anschauen

Copy kopiert nur die erste Ebene!!!, wenn man dreidimensionales Array dann funktioniert das untere Beispiel nicht, also Seiteneffekte entstehen 🡪 bei mehrdimensionen Array deep copy verwenden!!!

Eine Liste beinhaltet nur die Referenzen! Die Liste besteht aus Referenzen zu diesen Speicherstellen. Wenn wir kopieren, dann kopieren wir nur die Referenzen (Pooling entsteht hier wieder)

>>> l = [1,2,[1,2]]

>>> l

[1, 2, [1, 2]]

>>> l2 = l

>>> l2

[1, 2, [1, 2]]

>>> l2[0] = 99

>>> l2

[99, 2, [1, 2]]

>>> l

[99, 2, [1, 2]] … **Seiteneffekt**

>>> l3 = l.copy() … **deep copy**

>>> l3

[99, 2, [1, 2]]

>>> l3[0] = 666

>>> l

[99, 2, [1, 2]]

<https://www.python.org/dev/peps/pep-0008/> … Hinweise wie wir Code formatieren können 🡪 zur Lesbarkeit, Bedienbarkeit, … (jede Firma hat eine Styleangabe)

**Praxisbezogene Beispiele** <https://www.delftstack.com/de/howto/python/> 🡪 auf Python (rechte Seite) draufklicken

**Eingebaute Funktionen (Build-In-Funktionen)** <https://openbook.rheinwerk-verlag.de/python/19_008.html#u19.8> … Super-Nachschlagewerk (man kann vieles hier nachschauen)

**DUNDER (double underscore) Variablen, kommen bei Objektorientierung**

**Unittests** <https://docs.python.org/3/library/unittest.html>

<https://realpython.com/python-testing/>

<https://machinelearningmastery.com/a-gentle-introduction-to-unit-testing-in-python/>

**Global Variablen sind generell schlecht und unsicher!!!**

**Main:**

**z.B.**

z.B. in Mallaun:

def main():

return 1+2

import mallaun

import peer

def main():

mallaun\_ergebnis = mallaun.main()

peer.rechnen(mallaun\_ergebnis)

if \_\_name\_\_ == “\_\_main\_\_“:

main()

Dateien importieren und in main einbauen 🡪 Seiteneffekte werden dadurch verhindert, weil alles abgesichert ist 🡪 es kann z.B. eine Variable x in mallaun und in peer verwendet werden und gleich lauten

**Debugging scripts with pdp** [**https://docs.python.org/3/library/pdb.html**](https://docs.python.org/3/library/pdb.html)

[**https://realpython.com/python-debugging-pdb/**](https://realpython.com/python-debugging-pdb/)

**Ternärer Operator** [**https://www.geeksforgeeks.org/ternary-operator-in-python/**](https://www.geeksforgeeks.org/ternary-operator-in-python/)

Ternärer Operator ist eine kurze Schreibweise für if und else 🡪 kein Unteschied zum normalen if und else, weil immer gleich kompilliert wird

* Aus Lesbarkeitsgründen verwendet man den ternären Oprator

**Achtung**: Auswertungsreihenfolge der Operatoren bei ternären Operatoren beachten (z.B. Klammern setzen!)

**Anonyme Funktionen bzw. Methoden:** werden dynamisch zur Laufzeit im Stack gespeichert wenn diese ausgeführt und dann wieder vom Stack gelöscht 🡪 Speicher sparen

Test: Slicing

Unittest <https://realpython.com/python-testing/>:

Man weiß was passiert und man weiß war herauskommt im Gegensatz zu Integration Test.

Lambda:  
Anonyme Funktion

[Map funktion](https://www.digitalocean.com/community/tutorials/how-to-use-the-python-map-function-de) :

Map-object ist ein iterable object

Iterable:

Pointer -> bekommen Ergebnis jedoch wird es nicht in den Speicher eingegeben sondern nur das erste Element.

Kochrezept für Programm schreiben:

Methode nur eine Aufgabe, Unittest, keine globalen Variablen, so viel wie möglich kapseln

[Virtuell Enviroments:](https://realpython.com/python-virtual-environments-a-primer/)

Dafür da, um uns eine ganz exakte Python Version zu liefern mit Vorgaben. Egal welche Version die Person davor gehabt hat.

Soft-Link:

Ein verweis innerhalb einer z.B. Tabelle

A = 5 Speicherstelle Hardlink

B = A 🡪 wäre ein Softlink. 🡪 nur ein verweis auf einen verweis

Wen ein Softlink gelöscht wird, gibt es die Datei immer noch

Wenn ein Hardlink gelöscht wird, gibt es die Datei zwar immer noch aber ist nichtmehr aufrufbar

Python wird teilweise verlinkt, teilweise kopiert.

Wenn man laufend an einem Projekt arbeitet: Pipreqs

Java und Python sind Interpreter sprachen,   
  
List Comprehensions:

New\_list = [expression for member in iterable (if conditional)]

* Nur für Lesbarkeit

Nachteile:

* Keine Breakpoints, bei großen Datenmengen evtl. Stackoverflow

Dict Comprehensions:

* Es kommen bspw. Nur unique Elemente zurück

Vererbung:

Objekte: gekapselte Struktur, besteht aus Inhalt und Werkzeug

OOP: bessere Organisation

Self: das gehört zu einer Instanz

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Python: man ist immer static, java nicht

Man kann immer auf eine Instanz zugreifen/bearbeiten

\_\_str\_\_ => toString

Mehrfachvererbung problematisch

Super 🡺 Aufruf auf Oberklassen, temporäres Objekt zu einer Klasse somit kann man Methoden dieser Klasse aufrufen

Isinstance() um überprüfen, ob es vom Typ einer Instanz ist

Super deepdive

Super(Square, self) ist das gleiche wie super() jedoch kann man angeben wo er suchen beginnt. Besonders wichtig bei Mehrfachvererbung

Gebundene und ungebundene Methode: statisch oder nicht statisch

Gebundene methode 🡪 gekoppelt mit einer Instanz

Multiple Inheritance

Class RightPyramdi(Triangle, Square):

Es nimmt immer die erste Methode

\_\_mro\_\_: sucht die erste vererbte klasse, welche die gesucht methode findet also Triangel(nicht gefunden) -> Square

Kompositionen: dass eine kann es ohne dem anderen nicht geben

Aggregation: das eine kann auch ohne dem anderen existieren

Package besteht aus mehreren Modulen. Es sind class files drinnen

Jede python datei ist ein Modul.

Welche Module gibt’s es ?

* Cs datei
* Python
* Mix

Namespace 🡪 wo sehe ich was?

Import mod 🡪 zuerst im gleichen verzeichnis, danach im python env.

In python kann man zur laufzeit den Pfad erweitern sollte man aber nicht machen.

From mod import s 🡺 s wird überschrieben falls schon im code vorhanden

Dir(): was gibt es für variablen

Wenn mod importiert, es wird immer interpretiert und abgearbeitet

Reloading mod: es lässt mich es nur einmal ausführen, außer man macht ein reload(mod)

ARGS / KWARGS:

z.B. man will Methoden haben, die einmal 3, 5, 7 variablen brauchen -> normalerweise drei Methoden.

\* Operator: alles, was ich einer Methode übergebe (z.B. einzelne Werte oder Liste) wird alles in Tuple interpretiert. Also: Stern sagt, ich entpacke alles was von der Methode kommt interpretiere ich als Tuple.

\*args: kann auch anders als args heißen, der Stern ist wichtig

\*\*Operator: akzeptiert schlüsselwörter 🡺 a=“Real“, b=“Python“ ->fügt es zusammen wird wie ein dictionary verwendet

Bei namend keywords ist reihenfolge egal

Iterable: Sequenz die man iteriert

Iterator: braucht man, wenn etwas keinen Iterator hat

Iterator 2 Bedingung:

1. Magic Methode \_\_iter\_\_: Iterator muss auf die Sequenz zeigen also auf sich selbst.
   1. Was passiert, was bei next kommt. Man braucht intern immer einen Index
2. Magic Methode \_\_next\_\_ ausgeben welche die Verwaltung übernimmt.
3. Man muss eine Exception werfen. Raise Exception

Sobald man in in der for schleife, verwendet braucht man etwas welches einen Iterator hat.

2. Art von Iterator:

Manipulation von Daten, wenn man z.B. Daten quadrieren, somit muss man es nicht extra eingeben sondern man kann es direkt im Iterator eingeben

3te Art von Iterator:

Multiplikatoren: Er erzeugt neue Daten. Also man bekommt die Daten, die man zur Laufzeit benötigt.

Achtung auf Infinite Loops!!

Erleichterung