**Rapid Prototyping:** Alles geht sehr schnell, man muss auch keine Klassen-Struktur und ähnliches schreiben. Python hat auch viele vorinstallierte Bibliotheken, die man verwenden kann.

**Unterschiede Python Java:**

Bei **Python** muss der Datentyp nicht angegeben werden auch eine andere Syntax. Ich muss auch keine Klassenstruktur schreiben wie bei Java. Python Programme sind im Vergleich zu Java kleiner.

Bei **Java** haben wir eine strikte Typisierung bei Python eine dynamische Typisierung -> das hat nicht immer Vorteile man hat mehr Freiheit aber auch mehr Verantwortung. In Java kann ich außerdem mit Modifikatoren einstellen wer was sieht (private, protected). Bei Python gibt es diese nicht. Java ist schneller beim Kompilieren.

**Unterschiede Listen Array Python:**

Bei einer Liste ist es egal was für einen Typ sie beinhaltet. (Man kann somit auch Listen in Listen schreiben.

Bei einem Array muss der Datentyp angegeben werden, dafür hat man mit einem Array auch mehr Leistung, es müssen alle Elemente den gleichen Typ haben, es hat eine fixe Länge

Bei einem Array stehen die Elemente nebeneinander im Speicher und bei einer Liste zeigt ein Pointer bei jedem Element auf das vorherige Element und einer aufs nächste.

*Sobald man irgendetwas hinzufügen kann ist es eine Liste.*

**Garbage Collector:** Wenn Objekte nicht mehr benötigt werden, dann werden sie zum Überschreiben freigegeben. Der GC weiß, ob ein Objekt zum Überschreiben freigegeben ist, wenn:

Auf jeder Instanz gibt es einen Referenzzähler. Wenn eine Instanz nun angelegt wird, dann zeigt eine Referenz darauf. Mit jeder neuen Referenz auf die Instanz, erhöht sich der Referenzzähler. Wenn der RZ einer Instanz auf 0 ist, dann weiß der GC dass das Objekt zum Überschreiben freigegeben ist.

Alle Interpreter Sprachen haben einen GarbageCollector.

**Lambda-Funktionen:** Die Funktion wird während der Laufzeit definiert und wenn man sie nicht mehr braucht, dann wird sie wieder gelöscht. Normale Funktionen werden immer über die ganze Laufzeit in dem Prozessspeicher abgelegt.

**Referenz:** Ist ein symbolischer Link (ein Zeiger, ein Verweis) auf einen Wert.

**Instanz:** ist ein Objekt bzw. der Wert, auf den die Referenz verweist.

**Identität:** Ist eine eindeutige Kennung für ein Objekt. Bei Immutable Datentypen ist die ID immer die gleiche, wenn die values gleich sind.

**Sequentielle Datentypen:** Listen,arrays,dictionarys, … Alle bei denen Werte hintereinander im Zusammenhang stehen

*Wenn man irgendetwas hinzufügen kann dann redet man von mutable(veränderbar) ansonsten von immutable(unveränderbar)*

**Mutable vs Immutable:**

Bei Immutable verweist die Referenz auf einen Ort im Speicher. Wenn die Referenz geändert wird dann zeigt die alte Referenz immer noch auf den alten Ort im Speicher. -> Die Ids von immutable DT sind immer gleich, wenn sie den gleichen value haben (wenn ein 6er angelegt ist, dann bleibt es bei nur einem 6er, es wird kein zweiter erstellt)

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Mutable Datentypen können nach ihrer Erzeugung andere Werte annehmen. Bei mutable zeigt die neue und auch die alte Referenz auf den gleichen Ort im Speicher. Bei mutable wird immer eine neue Instanz angelegt, das heißt auch dass die ids nicht gleich sind. Alle **Objekte** sind mutable.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Dass nennt man Seiteneffekt.

**Copy vs Deepcopy:**

Wir haben ein Objekt. Z.B. den Noel mit Schokorigel. Bei Copy wird nur der Noel kopiert und bei Deepcopy wird der Schokorigel mitgecopied.

Um eine gänzlich neue Liste zu erstellen, auf die nicht die gleiche referenz zeigt, wie auf die alte Liste, müsste theoretisch jedes Element mittels z. B. Einer for Schleife in die neue Liste gespeichert werden. Dafür bietet uns Python praktischerweise die Methode copy() welche genau dies macht

Also copy erstellt eine neue Liste, auf die aber nicht die gleiche Referenz zeigt wie auf die alte Liste.

Und eben, dass bei einer Liste aus listen ma ja dann theoretisch a schleife in einer schleife brauchen würde, und da eben copy nimmer reicht und ma deepcopy hernehmen muss

Diese Funktionen kann man gegen die oben erwähnten Seiteneffekte nutzen. Soll heißen, wenn nur mutable Datentypen in der Liste sind, dann sollte man Deepcopy verwenden.

**Compiler vs Interpreter:**

Ein Compiler wandelt den gesamten Programmcode in Maschinensprache um, erst danach wird das Programm ausgeführt. Er benötigt im Vergleich mehr Zeit und Ressourcen als der Interpreter.

Ein Interpreter verarbeitet den Programmcode zur Laufzeit, das heißt die Anweisung wird ausgelesen, interpretiert und sofort ausgeführt. Das macht er Zeile für Zeile.

**Interpretersprachen, Maschinensprachen, Programmiersprachen:**

Es gibt keine Programmiersprache, die man nicht kompilieren muss. Manche werden in Interpretersprache und andere in Maschinensprache übersetzt. Interpretersprache wird in Bytecode übersetzt, den man in Class-Dateien finden kann. Bei Bytecode sind manche Stellen lesbar und manche schon kompiliert. Ein Teil vom Programm kann Plattformunabhängig kompiliert werden und der andere nicht, der muss dann nachkompiliert werden.

Eine Interpretersprache und Bytecode ist Plattformunabhängig. -> dafür langsamer.

Maschinensprache ist nicht Plattformunabhängig dafür aber schneller als Interpretersprache.

Assembler sind CPU spezifische Anweisungen (nimm Registerbank 1 und schiebe sie auf Registerbank 2 in der CPU).

In Java wird in 2 Schritten kompiliert (1. Kompilieren, 2. Ausführen).

In Python muss man nur einen machen, in Wirklichkeit wird aber auch in 2 Schritten kompiliert.

LINKS AUF KW42 DAZU

**Map-Funktion:**

Ist eine iterable Datenstruktur. Nimmt man her, um auf jedes Element in einer Iterablen eine Funktion anzuwenden.

**Klassen und Vererbung:**

Man programmiert in Klassen aufgrund von **Kapselung**, also um Objekte und Daten zu beschützen und dass man Variablen nicht verändern kann, wenn man sie woanders verwendet.

Man vererbt, um eine Struktur zu erstellen damit Inhalte nicht doppelt vorkommen und man Code wiederverwenden kann. „So wenig wie möglich, so viel wie nötig“. Mehrfachvererbung ist möglich, sollte man aber nicht viel verwenden, da das Programm irgendwann unübersichtlich wird.

**Self-Python:**

Self ist eine Referenz die auf sich zeigt. Es wird automatisch die Referenz auf das aktuelle Objekt mitgegeben. Python kann somit eindeutig zuordnen, welches Attribut von welchem konkreten Objekt geändert werden soll. So ähnlich wie das this in Java.

**Konstruktor-Python:**

In Python erstellt man den Konstruktor als erstes wenn man eine neue Klasse erstellt. Es gibt grundsätzlich nur einen Konstruktor und zwar den \_\_init\_\_ in Java kann man mehrere Konstruktoren erstellen.

**Super in Python:**

Steht für superclass sprich Elternklasse. Wir stellen hiermit eine Verbindung zwischen der Eltern- und Kindklasse her.

**Static:**

Static variablen gehören immer zu einer Klasse -> in Python sind alle Variablen in einer Klasse automatisch static.

**Virtual environment:**

Ist immer mit Python verheiratet, man kann dadurch sofort ein sauberes system haben.

Es dient zum Einrichten von ein oder mehreren unabhängigen Python-Umgebungen. Wenn ich zum Bleistift nicht will, dass ich wegen einem importierten Projekt, irgendwelche schwindeligen Bibliotheken auf mein System installieren soll, dann kann ich das ganze in einem VENV starten, denn da ist es abgekapselt von der Umgebung.

==================================================================================

pip freeze geht in pip rein und schaut nach was alles installiert ist -> kann man in requirements.txt speichern

pip freeze eigetlich nicht so gut weil immer alle requeiremts drinnen stehen also wenn man pandas installiert wird auch numpy reingeschrieben.

* Piprequs – geht in alle python files rein und schaut nach was in diese files inportiert worden ist -> in requirements steht nur mehr pandas drinnen

Namespaces noch rein!!!!!!

Modul ist eine ausgelagerte codesammlung. Bessere wartbarkeit, modul wird an mehren stellen verwendet -> wiederverwendbarkeit von modulen. Code wird durch struktur geschützt, bester schutz. Modul kann python code sein oder eine c file. Eine python datei ist schon ein modul. Wie werden module gefunden. Er such zuerste in aktuellen ordner. aktueller ordner ist immer da wo die main gestartet wird. Wenn hier nicht gefunden geht er in python pfad und sucht dort. Mit sys.path kann man python pfad nachschauen. Man kann pfad auch zur laufzeit verändern, sollte man aber eig nicht machen -> böse

From mod import foo dann landet nur die methode foo in stack geladen und erzeut keinen neuen namespace. Es wird alles überschrieben was in foo gleicht heisst wie in der datei die es importier

From mod import \* -> böse

Man kann überall im code from import … machen, sogar in einer methode aber ist ganz ganz schlecht, da man nie weis wo ws gmeacht wird usw. -> imports immer ganz oben

Import mit \* in methode geht nicht, da man nur im local namespace ist

dir() zeigt alle definierten sachen in dem namespace an

möglichkeiten um daten zu übergeben:

* args:

\*args bedeutet dass man eine liste üder tupel übergibt und die parameter füllen sich dann von links nach rechts mit den einträgen der liste. Wenn liste größer als parameter bleibt der rest im args vorhanden

* KWARGS:

\*\*kwargs: Mit dictionary, wenn man einen eintrag mehr im dictionary hat als parameter sind, bekommt man einen typeerror

Für verkettete listen insertionsort, weil meisttens weniger rechenschritte

**\*ARGS**

\*args haben 2 Anwendungsfälle:

1. Wenn die Parameterwerte für eine Funktion in Objektform vorliegen (einer Liste oder einem Tupel) können sie eben in dieser Form an die Funktion übergeben werden.
2. Die Anzahl der Übergabeparameter an die Funktion soll variabel bleiben.

**1. ÜBERGABE VON PARAMETERN IN OBJEKTFORM**

* Übergabe der Parameter als Tupel oder Liste. Hier werden die Elemente *In-Order-Of-Appearance* den Parametern der Funktion zugeordnet.

**def** myFunction(a,b,c,\*args):

**return** **print**(a,b,c, args)

myFunction(\*(1,2,3,4)) # Übergabe als Tupel

myFunction(\*[1,2,3,4]) # Übergabe als Liste

>>1, 2, 3 (4,)

In-Order-Of-Appearence wurden die Werte 1,2,3 den Parametern a,b,c zugeordnet. Der verbleibende Wert 4, verbleibt ungenutzt als Tupel innerhalb der Funktion und wird über den args-Parameternamen ansprechbar.

Durch den Funktionsaufruf mit \*args verkürzt sich gegenüber einem Aufruf mit Parameternamen auch der Quellcode:

param = [1,2,3]

myFunction(\*param) # Aufruf mit args

myFunction(a=1,b=2,c=3) # Aufruf mit Parameternamen

**2. VARIABLE LÄNGE DER ÜBERGABEPARAMETER**

Eine weitere Option die durch den \*args-Parameter entsteht, ist die beliebige Fortschreibung von Übergaben an die Funktion. Hier werden im ersten Beispiel der Funktion myFunction 7 Werte übergeben. Die Werte 1,2,3 werden den Parametern a,b,c zugeordnet. Die weiteren 4 Werte werden (wie bereits oben gesehen) als Tupel in der Funktion ansprechbar. Im zweiten Beispiel sind die Keyword-Argumente mit Werten besetzt und das \*args-Tupel steht so wie es ist, also mit den Werten 4,5,6,7 , in der Funktion zur Verfügung.

myFunction(1,2,3,4,5,6,7)

>>1, 2, 3 (4, 5, 6, 7)

myFunction(1,2,3,(4,5,6,7))

>>1 2 3 ((4, 5, 6, 7),)

**\*\*KWARGS**

* Übergabe der Parameter als Dictionary. Über die Keys werden die Werte den Parametern der Funktion zugeordnet.

\*\*kwargs sind verglichen mit \*args eine konservative Variante an die Funktion ein Objekt zu übergeben. Da das Objekt ein Dictionary ist, verhindert man eine Falschzuordnung von Parameter zu Wert wie es durch die *Order-Of-Appearence*-Regel bei der Verwendung \*args passieren kann.

**def** myFunction(a,b,c,\*\*kwargs):

**return** **print**(a,b,c, \*\*kwargs)

myDict = {'b':2,'a':1,'c':3}

myFunction(\*\*myDict)

Bei der Nutzung von \*\*kwargs werden alle benötigten Parameter mit ihren Werten als Key-Value Paare in das Dictionary eingetragen und der Funktion übergeben. Enthält das Dictionary Keys, die nicht den Parametern der Funktion zuzuordnen sind, führt dies zu einem Fehler:

**def** myFunction(a,b,c,\*\*kwargs):

**return** **print**(a,b,c, \*\*kwargs)

myDict = {'b':2,'a':1,'c':3,'d':4}

myFunction(\*\*myDict)

TypeError: ‚d‘ is an invalid keyword argument for this function

**\*ARGS UND \*\*KWARGS GEMEINSAM IN EINER FUNKTION**

Natürlich lassen sich \*args und \*\*kwargs auch gemeinsam in einer Funktion verwenden. Wir sehen an dem Beispielcode unten 5 Funktionsaufrufe von myFunction, die in ihrer Definition sowohl \*args als auch \*\*kwargs enthält.

**def** myFunction(parameter\_1, parameter\_2, \*args ,\*\*kwargs):

**print**(parameter\_1)

**print**(parameter\_2)

**print**(\*args)

myFunction('A',\*(100,200,300))

>>A

>>100

>>(200, 300)

myFunction('A',\*\*{'parameter\_2':1000})

>>A

>>1000

>>()

myFunction(parameter\_2 = 1000, \*('A',))

>>A

>>1000

>>()

myFunction('A',\*\*{'parameter\_1':1000,'parameter\_2':1000})

TypeError: myFunction() got multiple values for argument ‚parameter\_1‘

myFunction('A',\*(1,2,3,),\*\*{'parameter\_2':1000})