1. Lists, Tuples, Dictionaries
   1. Lists
   2. Tuples
   3. Strings als Listen
   4. Dictionaries
2. Error Handling
   1. Raising Exceptions
   2. Handling Exceptions
3. Debugging
4. Testing
   1. Assert
5. Object Oriented Programming
   1. Classes
   2. Declaring Method Attributes
   3. Special Methods
6. Inheritance
7. Design Principles and Patterns
8. S.O.L.I.D. Principles
   1. Single Responsibility Principle
   2. Open-Closed Principle
   3. Liskov’s Substitution Principle
   4. Interface Segregation Principle
   5. Dependency-Inversion Principle
9. Python Modules
10. Version Control Systems (Git)
11. **Lists, Tuples, Dictionaries**
    1. **Lists**

=An ordered set of values, where each value is identified by an index. Lists are **mutable**. Lists can have multiple types of values in it.

**Commands**

***l[x]=2*** → Ändert den Wert an der Stelle x zu 2. Die Stelle x muss bereits in der Liste vorhanden sein.

***l.append (3)***→ fügt den Wert *3* der Liste hinzu an letzter Stelle

***l.extend ([4,5])*** → fügt die Werte aus der Liste *[4,5]* der Liste *l* hinzu.

Unterschied *append / extend*: Extend hat meine eine Liste/Tuple mit Werten, welche dann der Liste hinzugefügt werden. Append fügt die Liste als ganzes als ein Element hinzu. Append für Einzelobjekte verwenden, extend für Listen, der Werte einer anderen Liste / Tuple hinzugefügt werden soll.

***x in l*** → return True wenn der Wert der Variable x in der Liste vorkommt, False wenn der Wert nicht vorkommt.

***x not in l*** → return True wenn der Wert der Variable x in der Liste nicht vorkommt, False wenn der Wert vorkommt.

***l3=l1+l2*** → Listen können mit anderen Listen addiert werden.

***l3=2\*l1*** → Listen können multipliziert werden.

***len(l)*** → Gibt die Länge / Anzahl Werte in der Liste zurück.

***l.pop(x)*** → Gibt den Wert an der Stelle x zurück und löscht das Element aus der Liste. Wenn kein Wert angegeben ist, wird das letzte Element gepoppt.

***del(l[x])*** → Löscht den Wert an der Stelle x aus der Liste.

***l.remove(x)*** → Löscht das erste Elemente mit dem Wert x aus der Liste.

***l.sort()*** → Sortiert die Liste von klein nach gross. Kann nur gemacht werden, wenn nur Zahlen in der Liste sind.

***l.index(x)*** → Gibt den Index des ersten Vorkommens des Wertes x zurück.

* 1. **Tuples**

=An ordered set of values, where each value is identified by an index. Tuples are **immutable**. Tuples can have multiple types of values in it.

**Commands**

***t=(1,2,3,4)*** → Erstellt ein Tuple mit den folgenden Werten.

***t=(1,)*** → Erstellt ein Tuple mit einem Wert. Ohne das Komma würde *t* einfach zu 1 als Integer werden.

***(a,b)=(1,2)*** → Weist a und b in einem Tupel die Werte 1 und 2 zu.

* 1. **Strings als Listen**

***s=”Ein einfacher String”*** → Erstellt einen einfachen String.

***chars=s.split(x)*** → Erstellt eine Liste mit einer Unterteilung nach jedem x, wobei das x nicht in die Liste miteinbezogen wird. Ist kein x angegeben, werden Leerzeichen als Unterteiler verwendet. Satzzeichen werden in diesem Fall auch in die Liste einbezogen. *chars* wäre hier also *[“ein”, “einfacher”, “String”]*

***str=”x”.join(chars)*** → Fügt die Elemente der Liste aneinander mit x dazwischen. Wenn kein x angegeben ist: “*eineinfacherString”*.

* 1. **Dictionaries**

Bestehen aus Keys zu einem Key zugehörigen Values. Keys sind immutable (int, float, string, tuple, bool) und dürfen nicht mehrfach vorkommen. Values können mehrfach vorkommen und können jede Art von Datentyp haben (auch lists, dictionaries)

**Commands**

***dic={“Alfred”: 123456, “Susi”: 124563}*** → Erstellt ein Dictionary mit 2 Keys (Alfred und Susi) und einer zugehörigen Nummer.

***dic[“Becky”] = 143265*** → Fügt einen Eintrag hinzu oder überschreibt den Wert, falls der Key bereits existiert.

***dic[“Alfred”]*** → gibt den zugehörigen Wert für den Key *Alfred* zurück.

***x in dic*** → Wenn *x* im Dictionary *dic* vorhanden ist, return *True*, sonst *False*. Suche findet sowohl bei Keys als auch bei Values statt.

***x in dic.keys()*** → Suche/ Iteration durch alle Keys

***x in dic.values()*** → Suche/ Iteration durch alle Values

1. **Error Handling**
   1. **Raising Exceptions**

***def do\_something\_with\_number(n):***

***if n < 0:***

***raise ValueError(“Need positive number”)***

* 1. **Handling Exceptions**

***try:***

***n=-1***

***if n<0:***

***raise ValueError(“Need positive number”)***

***except:***

***print(“Oops, there was an error!”)***

***print(“Life must go on.”)***

*Try*: Versucht den folgenden Block durchzuführen, wenn ein Fehler entsteht, wird das *except* ausgeführt.

*Except*: Dieser Block wird ausgeführt, wenn beim *try*-Block ein Fehler gefunden wurde.

Nach dem Try/Except Block wird das Programm weiter ausgeführt, auch wenn ein Fehler aufgetreten ist.

**try:**

**print("try1")**

**fun\_that\_might\_raise\_ex()**

**print("try2")**

**except ZeroDivisionError:**

**print(”specific except")**

**except:**

**print(“general except”)**

**else:**

**print("else")**

**finally:**

**print("finally")**

Try: Versucht den Block durchzuführen

except ZeroDivisionError: Wenn ein bestimmter Fehlertyp auftaucht, führe diesen Block aus

except: Wenn irgendein anderer Fehler aufgetreten ist, führe dies aus

else: Wenn keine Exception ausgeführt wurde

finally: Wird immer ausgeführt, egal ob ein Except oder das Else durchgeführt wurde.

1. **Debugging**

**Begriffe**

*Breakpoint*: Haltepunkt, bei dem das Programm anhält, wenn im Debugging-Modus

*Step Over*: Ein Befehl nicht genauer ansehen, weiter zum nächsten

*Step Into*: In eine Funktion hineingehen, also eine Funktion genauer ansehen

*Step Out*: Aus der Funktion wieder hinaustreten

*Continue*: Mit der schrittweisen Ausführung aufhören und den Rest des Programms normal ausführen

*Abort*: Debugging-Session abbrechen

1. **Testing**
   1. **Assert**

Mit assert kann eine Bedingung für das Programm forciert werden. Es wird ein Error ausgegeben, wenn diese Bedingung zum Zeitpunkt des *assert* Befehls nicht vorhanden ist.

Bsp:

***def foo(x):***

***assert x>=0***

***...***

***foo(-1)*** *#führt zu einem AssertionError*

1. **Object Oriented Programming**
   1. **Classes**

Objekttyp, der interne Daten hat und Befehle, wie mit diesen Daten umzugehen ist.

***class*** *ComplexNumber:*

*x=1 #*Wert, der von allen Instanzen der Klasse benötigt wird.

***def*** *\_\_init\_\_(self, real, imag):*

*self.real = real*

*self.\_\_imag = imag* #Versteckte Variable, sodass sie nur intern benutzt werden kann.

*c1=ComplexNumber(1,2)*

***print****(c1.real)* #prints “1”

* 1. **Declaring Method Attributes**

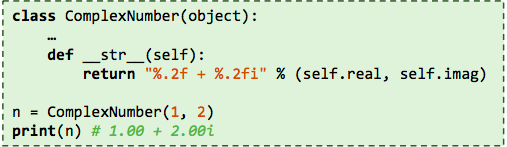
***def*** *\_\_str\_\_(self):*

*return “...”*

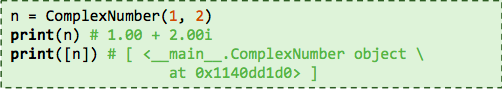
Diese Methode muss in der Klasse definiert sein, damit die Klasse geprintet werden kann.

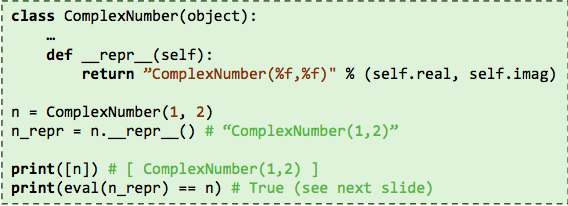
Methoden, die in einer Klasse definiert sind, aber keine ihrer Attribute benötigt (Also auch kein self), benötigen *@staticmethod* vor der Methode.

* 1. **Special Methods**

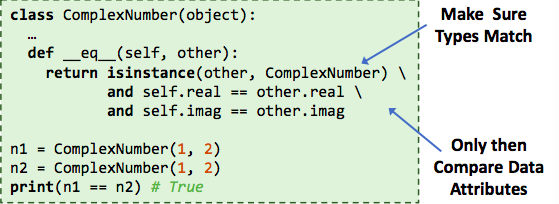


Das Ziel von \_\_str\_\_ ist es, eine lesbare Repräsentation eines Objektes zu erstellen.

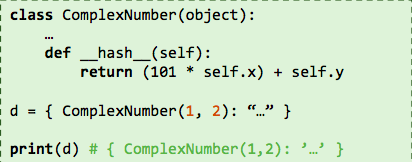




In Kollektionen (Listen, Tuples,...) muss \_\_repr\_\_ benutzt werden, um das Objekt darstellen zu können.



Um 2 Objekte vergleichen zu können, muss \_\_eq\_\_ benutzt werden.



Um eine Klasse hashen zu können, muss eine \_\_hash\_\_ Methode definiert werden.

1. **Inheritance**

1. **Design Principles and Patterns**
2. **S.O.L.I.D. Principles**
   1. **Single Responsibility Principle**
   2. **Open-Closed Principle**
   3. **Liskov’s Substitution Principle**
   4. **Interface Segregation Principle**
   5. **Dependency-Inversion Principle**
3. **Python Modules**
4. **Version Control Systems (Git)**