B1B

Back in Business
Python

Thomas Zenger
mail@thomas-zenger.de
9. April 2021

Inhaltsverzeichnis

1 Vorwort

1.1 `B1B` - Was ist das?

Dieses Skript soll als Nachschlagewerk und Cheat Sheet dienen und eine Gedächtnisstütze zu einem bereits vertiefeten Thema bieten. Es soll keine Fachliteratur ersetzen und dient auch nicht zum Erlernen der Thematik, da auf ausführliche Erklärungen größtenteils verzichtet wird.

1.2 Motivation

Diese Idee zu diesem Kompendium enstand während meines Informatikstudiums. Da bereits erlernte Techniken, wie z.B. Programmier- oder Scriptsprachen, mit dem Fortschreiten des Studiums in den Hintergrund traten, zu einem späteren Zeitpunkt jedoch wieder benötigt wurden, war es unerlässlich sich diese wieder ins Gedächtnis zu rufen. Aus diesem Grund entstand dieses Werk als eine Art "erweiterte Zusammenfassung".

1.3 Literatur und Grundlagen

Folgende Werke fanden bei der Erstellung dieses Dokuments Beachtung. An dieser Stelle soll ausdrücklich erwähnt werden, dass sich diese Arbeit nicht als Plagiat oder Kopie genannter Literatur verstanden werden soll, sondern als Lernhilfe und Zusammenfassung.

- S. Kaminski (2016) - Python 3
- P. Barry (2017) - Python von Kopf bis Fuß
- K. Garg - Script Scientific Computing with Python
- Numpy User Guide (https://numpy.org)
- https://www.python-kurs.eu/

2 Allgemeines

2.1 Ausführbares Skript

Ausführbares Skript und ermöglicht Import in andere Python Programme.

```
1 #!/usr/bin/env python
```

2.2 Kommentare

```
1 print(``Hello World!'') #Kommentar bis zum Ende der Zeile
```

2.3 Nichts tun

Mit der Anweisung `pass` wird das Programm nichts tun. Die Anweisung kann als Platzhalter für noch zu implementierende Funktionen genutzt werden.

2.4 Funktionen nachladen

Module werden in Python mithilfe des Schlüsselwortes `import` nachgeladen.

```
1 #Ganzes Modul
2 import sys
3
4 #einzelnes Element eines Moduls
5 from sys import stdout
```

2.5 Manpage & Co.

```
1  #Dokumention der Funktion
2  help(funktionsname)
3
4  #globaler Namensraum
5  dir()
6
7  #Attribute eines Objekts
8  dir(variable)
9
10 #Typ eines Attributs
11 type(variable)
```

3 Variablen

3.1 Gültige Variablennamen

Variablen müssen mit einem Buchstaben oder einem Unterstrich beginnen. Der Name selbst ist Case sensitiv und darf neben Buchstaben auch Zahlen und der Unterstrich verwendet werden.

3.2 Zuweisung

In Python erfolgt keine Typendefinition (keine statische Typisierung). Die Variablen deuten auf einen Speicherbereich. Bei Zuweisung einer Variablen auf eine andere Variable, zeigen beide auf den selben Speicherbereich.

```
1 #Einfache Zuweisung
2 a = 1;
3
4 #Mehrfachzuweisung
5 a, b = 1, 2;
6
7 #Speicherbereich einer Variablen anzeigen
8 id(a)
```

4 Objekttypen

Alle Datentypen sind Objekte. Mit dem Typ sind Funktionen verknüpft (Objektmethoden).

```
#Methodenaufruf
objekt.methode()
```

4.1 Zahlen

```
ganzzahl = 42 #int
langezahl = 43218932549085324908
fliesskomma1 = 3.12 #float
fliesskomma2 = 8.12e+10
komplex = 1+2j #complex
```

Besonderheiten

- Ganze Zahlen können Werte über den gültigen Speicherbereich annehmen
- Ganze Zahlen haben keine Begrenzung bei der Genauigkeit
- Angabe von Ganzen Zahlen kann auch binär, oktal oder hexadezimal erfolgen (Präfixe: `0b`, `0o`, `0x`)

4.1.1 Arithmetische Operatoren

`+`	Addition	`abs(x)`	absoluter Wert
`-`	Subtraktion	`int(x)`	erzeugt Ganzzahl
`*`	Multiplikation	`float(x)`	erzeugt Fließkommazahl
`/`	Division	`complex(x)`	erzeugt komplexe Zahl
`//`	Ganzzahldivision	`divmod(x)`	Ganzzahl- & Modulodivision
`%`	Modulo	`pow(x, y)`	Potenzieren
`-`	Negation	`x ** y`	Potenzieren
`+`	unveränderter Wert		

4.1.2 Bit-Operatoren

`a | b`	ODER	`a` $\ll$ `b`	n Bit nach links
`a ^ b`	XOR	`a` $\gg$ `b`	n Bit nach rechts
`a & b`	UND	$\sim$`a`	Bitkomplement

4.1.3 Vergleichsoperatoren, Boolsche Operatoren

`a < b`	kleiner	`a >= b`	größer gleich
`a <= b`	kleiner gleich	`a == b`	gleich
`a > b`	größer	`a !=b`	ungleich
`AND`	UND	`OR`	ODER
`NOT`	NICHT		

4.1.4 Kurzschreibweise

```
1 a += b
```

4.2 Zeichenketten

Python speichert Zeichenketten als einzelne Zeichen innerhalb eines String-Objekts mit UTF-8. Ein Typ für ein einzelnes Zeichen ist nicht vorhanden, alle Zeichenketten gehören der Klasse `str` an. Zeichenketten sind unveränderlich, alle Aktionen liefern eine neue Zeichenkette.

```
1 #Gleichwertig, einfaches und doppeltes Anfuhrungszeichen ohne
     Escapen
2 s = 'hello'
3
4 s = ''hello''
5
6 #Multiline String
7 s = '''hello
8 ... world
9 ...'''
```

4.3 Listen

Eine Liste ist eine geordnete Sammlung von Objekten. Eine Liste kann beliebige Objekte enthalten und ist veränderbar.

```
1 #leere Liste
2 liste = []
3
4 #Liste mit Lange n
5 liste = [n]
6
7 #direkte Zuweisung
8 liste = [1, 3.14, ''hello'']
```

4.4 Tupel

Tupel sind wie Listen eine geordnete Sammlung von Objekten. Im Gegensatz zur Liste sind Tupel nicht veränderbar

```
#leeres Tupel
tupel = ()

#direkte Zuweisung
tupel = (1, 2, 3)

#Ein-Element-Tupel
tupel = (1,)
```

4.5 Range

Ein Bereichstyp zur Erzeugung von unveränderbaren Zahlenlisten. Es können bis zu drei Parameter bei der Erzeugung angegeben werden (Startwert, Endwert, Schrittweite). Standardstartwert ist 0.

```
list(range(1, 10, 2)

r = range(10)
```

4.6 Operationen auf Sequenztypen

Gemeinsamer Satz von Operationen und Funktionen für Objekte des Typs Strings, Listen, Tupel und Range.

4.6.1 Index-Zugriff

- Abfrage erfolgt über Index (Nur bei Liste: Zuweisung)
- Index beginnt mit 0
- Letztes Element hat den Index Länge - 1
- Negativer Index: Es wird ab Ende der Sequenz gezählt

4.6.2 Slicing

```
list = [1, 2, 3, 4, 5, 6]

#Extrahieren Teilstuck druch Index, Ende exklusiv
list[start:ende:schrittweite]

#Loeschen eines Listenbereichs
list[start:ende] = []
```

4.6.3 weitere Funktionen

```
''hallo'' + ''welt''     #''hallo welt''
''hallo'' * 2            #''hallo hallo''
s = ''hallo''
''ll'' in s              #true
''oo'' not in s          #true
len(s)                   #5
s.index('a')             #1
min(s)                   #Minimum ' '
max(s)                   #Maximum 'o'
s.count('s')             #2
```

4.7 Dictionarys

Der Typ `dict` ist eine Sammlung von Schlüssel-Wert-Paaren, wobei der Schlüssel unveränderlich ist. Es existiert keine definierte Reihenfolge, ein Zugriff über einen Index ist somit nicht möglich.

4.7.1 Deklaration

```
d = {}
d = {'key1' : 'wert1', 'key2' : 'wert2', 'key2' : 'wert2', }
```

4.7.2 Zugriff, Zuweisung, Löschen

Vor dem Zugriff sollte man überprüfen, ob der Schlüssel enthalten ist (`in`).

```
'key1' in d

#Zugriff
d['key1']

#Zuweisung
d['key1'] = 42

#Entfernen Key und Value
del d['key1']
```

4.7.3 Operationen

```
#Erweitern
d.update(d2)

#Alle Keys bzw. Werte
d.keys()
d.values()

#Alle Paare
d.items()
```

Die Methoden liefern ein Objekt welches mit der List-Mehtode in eine Liste umgewandelt werden kann (`list(d.values())`).

4.8 Set und Frozenset

Veränderbare Menge von unveränderbaren Objekten, welche mathematische Mengenoperationen beherrscht. Kein Index, Sortierung und Slicing. Zusätzlich arbeitet das Frozenset wie ein Dictionary und kann nicht verändert werden.

```
1 s = {1, 2, 'a', 3.14}
2 s.add('=')
3 s.remove(2)
4 2 in s
5 len(s)
```

4.8.1 Mengenoperationen

Operator	Methode	Funktion
<=	issubset	Untermenge
>=	issuperset	Obermenge
|	union	Vereinigungsmenge
&	intersection	Schnittmenge
−	difference	Differenzmenge
^	symmetric_difference	Differenz mit einmaligen Werten

4.9 Erzeugung von Listen, Sets und Dictionarys

Außer der expliziten Deklaration steht den Objekten der Typen list, set und dict die Erzeugung durch eine Schleife zur Verfügung (“List Comprehension” bzw. “Display”).

```
1  #Explizite Erzeugung
2  l = ['a', 'b', 'c']
3  s = {1, 2, 3}
4  d = {'a' : 1, 'b' : 2}
5
6  #Allgemeine Form
7  <Ausdruck> for <Schleifenvariable> in <Sequenz> <Test>
8
9  #List
10 l = [x for x in rage(10, 15)]
11 s = {x / 2 for x in range(5) if x % 2 == 0}
12 d = {x : y for x, y in enumerate(l)}
```

5 Steuerung des Programmablaufs

5.1 Konstanten - Boolesche Werte

`True`	wahr
`False`	falsch
`None`	undefiniert

5.2 Objekt- vs. Wertevergleich

```
1  #Eindeutige ID des Objekts (Referenz)
2  id(object1)
3
4  #Identitaetsvergleich (kein Vergleich des Inhalts, sondern der
       Referenz)
5  object1 is object2
6  object1 is not object2
7
8  #Zuweisung auf selbe Referenz
9  object1 = 1
10 object2 = 2
11 object1 = object2
12 object1 is object2 #True
13
14 #Wertvergleich
15 object2 = 3
16 object1 == object2 #False
```

5.3 Boolesche Operatoren

Diese Operatoren haben die niedrgste Priorität aller bis jetzt vorgestellten Operatoren.

`or`	Logisches Oder
`and`	Logisches Und
`not`	Logische Negation

5.4 Bedingte Ausführung `if`

```
1  a = 42
2
3  #Einfache if-Abfrage
4  if a == 42:
5          print(a)
```

```
6
7  #Mehrere Bedingungen
8  if a == 0:
9          print(''0'')
10 elif a > 0:
11         print(a)
12 else:
13         print(''kleiner'')
```

5.5 Objektabhängige Ausführung `with`

Mit dem `with`-Statement wird ein Block abhängig von einem Context ausgeführt. Der Context wird durch ein Objekt dargestellt. Es werden Methoden gestellt, die zu Beginn und Ende des Blocks ausgeführt werden.

```
1 with expression [as variable]:
2     with-block
```

5.6 Bedingter Ausdruck

Ternärer Operator als Platzersparnis im Gegensatz zu einer vollständigen If-Anweisung.

```
1 #Binaerer Ausdruck
2 if True:
3    1
4 else:
5    0
6
7 #Ternaerer Ausdruck
8 1 if True else 0
```

6 Schleifen

6.1 Zählschleife `for`

Durchläuft alle Elemente eines Objekts. Bei einem Dictionary wird der key zurückgeliefert. Die Rückgabe der Schlüsselwerte erfolgt nicht in der Reihenfolge, in der das Dictionary initialisiert wurde (erzwingbar mit `sorted()`).

```
1  for a in ''Hello'':
2          print(a)                  #H e l l o
3
4  #Dictionary Zugriff Mehrfachzuweisung
5  d = {'1' : 4, '2' : 8, '3' : 16}
6  for a, b in d.items():
7          print(a, b)
8
9  #Range und (optional) Schleifenende
10 for n in range(3):
11         print(n)
12 else:
13         print(''Ende'')
```

Der Endteil kann mit der `break`-Anweisung unterbunden werden.

6.2 Bedingte Schleife `while`

Bedingung wird vor jeder Ausführung geprüft.

```
1  i = 0
2  while i < 10:
3          i += 1
4  #optional
5  else:
6          print(''Ende'')
```

6.3 Unterbrechung und Neustart

```
1  #Abbruch der Schleife
2  for n in range(10):
3          if n == 2:
4             break
5          print(n)
6
7  #neue Iteration
8  for n in range(10):
9          if n % 2 == 0:
```

```
10                      continue
11              print(n)
```

7 Funktionen

7.1 Definition & Aufruf

Eingeleitet durch Schlüsselwort `def`. Parameter sind optional. Der Funktionsname und seine Parameter nennt man Interface oder Signatur. Eine Funktion kann ein Ergebnis an seinen Aufrufer zurückgeben (liefert ansonsten none zurück). Funktionen sind in Python ebenfalls Objekte. Der Funktionsname darf mehrfach im Programmcode, beachtet wird dabei aber nur die letzte Definition. Die Funktion muss vor dem ersten Aufruf bekannt sein. Eine Funktion kann einer Variable zugewiesen werden.

```
1 def funktionsname():
2     pass
3
4 f = funktionsname
```

7.2 Sichtbarkeit von Variablen

Eine Funktion stellt einen neuen Namensraum bereit. Variablen, die übergeben oder hier erstellt werden, werden mit der Beendigung der Funktion gelöscht. Eine lokal definierte Variable überdeckt die globale Definition. Findet der Python-Interpreter im lokalen Namensraum keine Definition der Variable, versucht er diese im globalen Namensraum zu finden. Schreibenden Zugriff auf globale Variablen erhält man mit dem Schlüsselwort `global`

7.3 Funktionsparameter

Die Übergabe von Parametern erfolgt in Python als Referenz, dadurch können veränderbare Objekte durch die Funktion manipuliert werden.

7.3.1 Positionsabhängige Parameter (Positional Parameters

Die einfachste Form der Parameterübergabe, auch “formale Parameter” genannt. Trennung erfolgt durch Kommata in der Parameterliste. Der Bezeichner der Variablen in der Signatur sind in der Funktion bekannt. Die Funktion muss immer der exakten Anzahl der definierten Parameter aufgerufen werden. Eine Mehrfachdefinition mit unterschiedlichen Parameterlisten ist nicht möglich.

```
1 def func(a, b, c)
2     pass
```

7.3.2 Variable Anzahl von Parametern (Variable Arguments)

Falls die Anzahl de Parameter nicht von vorneherein bekannt ist, kann die variable Parameterliste genutzt werden. Falls diese genutzt wird, steht sie hinter den positionsabhängigen Parametern.

```
1 def func(args*)
2     pass
3
4 #Sequenzobjekt
5 def func(list)
6
7 #Objekte einer Sequenz
8 def func(*list)
```

7.3.3 Keyword Parameter (Keyword Arguments)

Mit den benannten Parametern (Keyword Parameter) kann man optionale Parameter realisieren, die beim Aufruf der Funktion einen Namen zugewiesen bekommen und in beliebiger Reihenfolge angegeben werden können. Sie werden in der Funktionsdefinition nach den variablen Parametern aufgeführt. Die Variable nimmt alle bennanten Argumente in ein Dictionary auf.

```
1 def func(**kwargs):
2     for k, v in kwargs.items():
3         print(''%s=%s'' % (k, v))
4
5 func(p1=1, p2=2)
```

7.4 Defaultwerte für Funktionsparameter

Parameter muss nur angegeben werden, wenn der Wert von der Vorgabe abweichen soll.

```
1 def func(a, b=None):
2     if b:
3         print(''a'', a, ''b'', b)
4     else:
5         print(''a'', a)
```

7.5 Rückgabewert einer Funktion

```
1 def func():
2     return 42
3
4 a = func()
```

7.6 Manipulation von Argumenten

Als Parameter übergebene, veränderbare Objekte können in einer Funktion verändert werden. Diese Änderungen sind nach der Rückkehr sichtbar.

```
1 def func(a):
2     a[0] = 0
```

7.7 Dokumentation

```
1 def func():
2     ''Dokumentation''
3
4 help(func)
```

7.8 Geschachtelte Funktionen

Eine Funktion kann lokal in einer Funktion definiert sein. Auch ist es möglich, dass die äußere Funktion die innere Funktion als Rückgabewert liefert (Factory Funktion).

```
1 def outer():
2     def inner(v):
3         return v*v
4     return inner
5
6 a = outer()
7 a(2)             #4
```

8 Funktionales

8.1 Lambda Funktionen

EIne Lambda Funktion ist eine Funktion ohne Namen und mit nur einem Ausdruck. Default Werte und variable Parameter können genutzt werden.

```
1 add = lambda x,y : x+y
2 add(1, 2)
```

8.2 Funktionen auf Sequenzen ausführen `map()`

Funktion benötigt zwei Argumente (Funktion und Sequenz-Objekt). Das Ergebnis ist ein Iterator

```
1 i = map(lambda v : v * v, rage(5))
```

8.3 Daten aus Sequenz extrahieren `filter()`

Wie der `map()`-Funktion wird der `filter()`-Funktion eine Funktion und eine Sequenz zugewiesen.

```
1 filter (lambda x: x == 25, list)
```

8.4 `reduce()`

Die Funktion `reduce()` wendet die gegebene Funktion auf die ersten zwei Werte der Sequenz an. Existiert nur noch ein Wert, wird dieser zurückgegeben.

```
1 data = [47,11,42,13]
2 functools.reduce(lambda x,y: x+y, data) #113
```

9 Ein- und Ausgabe

9.1 Eingabe `input()`

Einfachste Form mit Einlesen von der Tastatur mit dem Befehl `input()`. Beendet wird das Einlesen durch:

- Return oder Enter
- `Ctrl + C`
- `Ctrl + D`

```
1 s = input('Eingabe: \n')
```

9.2 Ausgabe `print()`

Gegenstück zu `input()`. Nimmt beliebig viele Parameter entgegen und fügt am Ende das Newline Zeichen ein.

9.2.1 Formatierung

Ausgabeformatierung mithilfe der Methode `format()`. Platzhalter werden durch die übergebenen Werte/Variablen ersetzt. Variablen können auch über ihren Index oder Namen angesprochen werden

Platzhalter	**Funktion**
{}	Universeller Platzhalter
{ name }	Variable mit Namen `name` hier einsetzen
{ :5 }	Feld 5 Zeichen breit formatieren
{ n:5 }	Variable 5 Zeichen breit einfügen
{ :10b }	Feld 10 Zeichen breit und Binärdarstellung
d	Dezimaldarstellung
f	Fließkommazahl (6 Nachkommastellen)
.2f	2 Nachkommastellen
b	Binärdarstellung
x,X	Hexadezimaldarstellung mit kleinen oder großen Buchstaben
!a	Funktion `ascii() auf Variable anwenden`
!s	Einsatz von `str()`
!r	Einsatz von `repr()`

```
1 'p2 {1:10} p1 {0:04X} k2 {key2} k1 {key1}'.format(42, 3.15, key1='a
    ', key2='b')
2 #p2       3.15 p1 002A k2 b k1 a
```

Formatierung mit `str`-Methoden

Methode	Beschreibung
`ljust(w)`	Text links ausgeben, Zeichenkette der Länge w
`rjust(w)`	Text rechts ausgeben , Zeichenkette der Länge w
`zfill(w)`	Zeichenkette der Breite w links mit 0 füllen
`center(w[, fillchar])`	Text in de Mitte einer Zeichenkette der Breite w positionieren und ggf. das Füllzeichen verwenden
`captialize()`	Erste Zeichen in Großbuchstaben, der Rest klein
`lower()`	Alle Zeichen in Kleinbuchstaben
`swapcase()`	Klein- gegen Großbuchstaben und umgekehrt
`title()`	Erster Buchstabe jedes Wortes in Großbuchstaben

9.2.2 Formatierung mit Formatstrings

Der Formatstring ist eine Zeichenkette mit Steuerzeichen als Platzhalter. Die Variablen werden nach einem `%` angegeben.

Platzhalter	Ausgabe
`%s`	Zeichenkette
`%d`	Ganzzahl
`%x`	Ganzzahl Hexadezimal Klein
`%X`	Ganzzahl Hexadezimal Groß
`%o`	Ganzzahl Oktal
`%f`	Fließkommazahl
`%2d`	Anzahl Zeichen
`%02s`	Anzahl Zeichen mit ggfls. führenden Nullen
`%.2f`	Beschränkung Nachkommastellen
`%5.2f`	Gesamtbreite mit Nachkommastellen
`%10s`	Breite 10 Zeichen rechtsbündig
`%-10s`	Breite 10 Zeichen linksbündig

9.2.3 Ausgabe auf `stdout` oder `stderr`

Ausgabe mit der `write()` Methode auf den Standard Ausgabekanal. Der Import der Kanäle erfolgt mit `sys`.

9.3 Dateien

9.3.1 Arbeiten mit Dateien

Für Textdateien reicht die Angabe des Namens. Bei anderen Dateitypen muss der Modus mit angegeben werden.

Parameter	Funktion
r	Lesen (Default)
w	Schreiben (Löscht evtl. vorhandene Datei)
a	am Ende der Datei anhängen
r+	Lesen & Schreiben
b	angehängt an anderen Modi, öffnet Datei im Binärmodus

```
1 fd = open ('foo')
2 data = fd.read()
3 fd.close
```

9.3.2 Methoden von File-Objekten

`read(n)`	Liest n Bytes aus der Datei
`readline()`	Liest eine Zeile inklusive des Zeilenendezeichens
`readlines()`	Liefert eine Liste aller Zeilen in der Datei
`write(s)`	Schreibt Zeichenkette s in Datei
`seek(p)`	Setzt Position des Zeigers auf Position p
`tell()`	Liefert die Position des Zeigers

9.3.3 Daten in Objekte ausgeben

Das Modul `io` bietet eine String-Klasse, die wie ein File-Objekt behandelt werden kann: `StringIO`. StringIO verhält sich wie eine Datei.

9.3.4 Fehlerbehandlung bei der Arbeit mit Dateien

```
1  try:
2      file = open{'robots.txt'}
3  except (FileNotFoundError, PermissionError) as err:
4      print(''Fehler {}''.format(err))
5
6  try:
7      data = file.read()
8  except: ValueError
9      pass
10 finally:
11     file.close()
```

Äquivalent zu obigen Code mit Context Manager:

```
1 with open('robots.txt', 'r') as file:
2     f.read()
```

10 Fehlerbehandlung

10.1 Fehler abfangen `try ... except`

```
1  try:
2          #do sth
3
4  #einen Fehler abfangen
5  except KeyboardInterrupt as err:
6      print(''{}''.format(error))
7
8  #Verschiedene Fehler abfangen
9  except (TypeError, SyntaxError):
10
11 #alle Fehler abfangen
12 except:
13     pass
14
15 #nach try weiter ohne Fehler
16 else:
17     #Anweisungen, die keinen Fehler ausloesen koennen
18
19 finally:
20     #Vor Beendigung ausfuehren
```

10.2 Exception auslösen

Eine Exception kann mit dem Schlüsselwort `raise` ausgelöst werden. Nach dem Schlüsselwort wird die gewünschte Exception und ggfls. Parameter in Klammern angegeben.

11 Objekte

11.1 Definition von Klassen

```
1 class MeineKlasse():
2     pass
```

11.2 Attribute

Klassen können Variablen definieren. Diese werden als Attribute bezeichnet. Im Gegensatz zu normalen Variablen werden Attribute immer mit Referenz auf das aktuelle Objekt angesprochen (`self`). Auf Attribute und Methoden kann von außen immer zugegriffen werden. Attribute können sogar dynamisch dem Objekt hinzugefügt oder entfernt werden.

```
1 class MeineKlasse():
2     def setvalue(self, v):
3         self.value = v
4     def getvalue(self):
5         return self.value
```

11.3 Methoden

```
1 class MeineKlasse():
2     def func(self):
3         pass
```

11.4 Von der Klasse zum Objekt

```
1 object = MeineKlasse()
2 object.setvalue(42)
```

11.4.1 Konstruktor

```
1 class MeineKlasse():
2     def __init__(self, v):
3         self.v = v
4
5 object = MeineKlasse(42)
```

11.4.2 Überladen von Funktionen

Ein Überladen von Funktionen mit mehreren Paramatern ist grundsätzlich möglich. Allerdings ermöglicht Python nur die Nutzung der letzten Definition der Funktion.

11.4.3 Klassenvariablen

Eine Klassenvariable ist eine Variable, die von allen Objekten einer Klasse geteilt wird. Die Klassenvariable wird, im Gegensatz zu Objektvariablen, ohne das Präfix `self` definiert. Soll innerhalb eines Objektes auf die Klassenvariable zugegriffen werden, geschieht dies über die Klasse eines existierenden Objektes.

```
1 class MeineKlasse():
2     classvar = 42
3
4     def __init__(self, v):
5         self.v = v
6
7     def func():
8         type(self).classvar =  3
```

Typ der Klassenvariable

Bei der Nutzung von Klassenvariablen muss darauf geachtet werden, ob diese veränderbar sind (list, dict) oder nicht (tuple, str). Nicht änderbare Typen werden bei einer Zuweisung als neue Variable im betreffenden Objekt angelegt und die Referenz im Namen gespeichert.

11.5 Vererbung

11.5.1 Einfache Vererbung

```
1  class Punkt:
2
3      def __init__(self, x=0, y=0):
4         self.x = x
5         self.y = y
6
7      def getX(self):
8          return self.x
9
10     def getY(self):
11         return self.y
12
13 class Ort(Punkt):
14     def __init__(self, x, y, name):
15         super(Ort, self).__init__(x,y)
16         self.name = name
17
18     def getName(self):
19         return self.name
```

Funktion	Beschreibung
`type(objekt)`	Liefert den Typ des Objekts o
`isinstance(objekt, klasse)`	Liefert True, wenn objekt ein Exemplar von Klasse
`issubclass(klasse, vklasse)`	Liefert True, wenn klasse eine Subklasse von vklasse

11.5.2 Von eingebauten Klassen erben

```
1 class MDict(dict):
2     def __init__(self):
3         super(MDict, self).__init__()
```

11.5.3 Mehrfachvererbung

```
1  class A:
2      def m(self):
3          print(''A.m()'')
4
5  class B:
6      def m(self):
7          print(''B.m()'')
8
9  class C(A,B):
10     pass
11
12 c = C();
13 c.m()
14 #A.m(), wegen Suchreihenfolge (1. Auftauchen)
```

12 Weiterführendes

12.1 Typ einer Variablen

```
1 liste = [1, 2, 3]
2 type(liste) == list    #true
```

12.2 Attribute eines Objektes

```
1 #Ausgabe aller Attribute, Methoden des Objektes
2 liste = [1, 2, 3]
3 dir(liste)
4
5 #Testen aus Existenz
6 hasattr(liste, '__doc__')
```

12.3 Standardfunktionen implementieren

Methodenname	Einsatzzweck
`__init__(self [, ...])`	Konstruktor
`__del__(self)`	Destruktor
`__repr__(self)`	Liefert Darstellung des Objekts als String
`__str__(self)`	Liefert String
`__bytes__(self)`	Byte-String des Objets
`__format__(self, f)`	ruft `format() auf`
`__hash__(self)`	ruft `hash()` auf, sollte nur zusammen mit `eq()` implementiert werden
`__bool__(self)`	bewertet das Ergebnis von `len()`

12.4 Vergleichsoperatoren

Methodenname	Einsatzzweck
`__eq__(self, o)`	equal
`__gel__(self, o)`	greater or equal
`__gt__(self, o)`	greater
`__le__(self, o)`	less or equal
`__lt__(self, o)`	less than
`__ne__(self, o)`	not equal

12.5 Attributzugriif

Methodenname	**Einsatzzweck**
`__getattr__(self, n)`	Wird aufgerufen, falls Attribut nicht in verfügbaren Namensraum gefunden
`__getattribute__(self, n)`	Wird zuerst beim Attributzugriff aufgerufen
`__setattr__(self, n, v)`	Wird aufgerufen, wenn einem Attribut ein Wert zugewiesen wird
`__delattr__(self, n)`	Wird durch `del o.n`aufgerufen
`__dir__(self)`	Aufruf von `dir()`

13 NumPy - Numerical Python

13.1 Arrays

Die zentrale Datenstruktur in NumPy ist das mehrdimensionale Array. Es ist ein mehrdimensionaler Container für homogene Daten (Ein Datentyp gilt für das gesamte Array).

```
1  import numpy as np
2
3  a = np.zeros(3)          #([0., 0., 0.])
4  type(a)                  #numpy.ndarray
5
6  b = array([1, 2, 3, 4, 5])
7
8  c = np.arange(5)         #gleichverteilte Werte
9  c[1] = 9.7               #Konvertierung zu Int-Wert (9)
10 c = c*0.5                #Konvertierung dtype=float
11
12 d = np.arange(5, dtype=np.float)    #array([0., 1., 2., 3., 4.])
13 e = np.arange(3,5,0.5)              #array([3., 3.5, 4., 4.5])
14 f = np.linspace(1, 10, 3)           #array([1., 5.5, 10.])
15
16 g = np.array([[1, 2],   #2-dimensionales Array
17               [3, 4],
18               [5, 6]])
19 g.shape                 #Zeilen und Spalten(3,2)
20
21 h = np.arange(12).reshape(4,3)      #2D
22 i = np.arange(24).reshape(2,3,4)    #3D
```

13.2 Array Indexing

```
1  import numpy as np
2  a = np.linspace(1, 2, 5)
3  a[0]             #1
4  a[0:2]  #[1, 1.25]
5  a[-1]            #2.
6
7  b = np.array([[1, 2], [3, 4]])
8  b[0,0]  #1
9  b[1, 1] #4
10 b[0,:]  #[1, 2]
```

13.3 Reshaping Arrays

```
1 import numpy as np
2
3 a = array([0, 1, 2, 3, 4, 5])
4 a = np.arange(6).reshape(3,2)        #2D Reshaping
5                                      #3D (Plane, Rows, Cols)
6 array ([[1, 2],
7         [3, 4],
8         [5, 6]])
```

13.4 Array I/O

```
1 import numpy as np
2
3 arr =  np.array([[0, 1, 2, 3] [4, 5, 6]])
4
5 np.savetxt(fname='array_out.txt', X=arr, fmt=%d)
6
7 loaded_arr = np.loadtxt(fname='arr_out.txt')
```

13.5 Array Methoden

`a.copy()`	Kopie	`a.argmax()`	Index des Maximum
`a.sort()`	Sortieren	`a.cumsum()`	Kumulative Summe
`a.sum()`	Summe	`a.cumprod()`	Kumulatives Produkt
`a.mean()`	Durchschnitt	`a.var()`	Varianz
`a.min()`	Minimum	`a.std()`	Standardabweichung
`a.max()`	Maximum	`a.transpose()`	Transponierte Matrix

`a.searchsorted()`	Index des ersten Wertes der $\geq$ der Variable ist
`np.random.rand(3,2)`	Array mit zufälligen Werten & gegebenen Shape
`np.random.randint(2, size=10)`	Array mit zufäligen Int-Werten

13.6 Array Operations

```
1 import numpy as np
2
3 a = np.arange(1, 6, 1)
4 b = np.arange(1, 6, 1)
5
6 c = a + b                  #[2, 4, 6, 8, 10]
7 d = a * b                  #[1, 4, 9, 16, 25]
8 e = a ** b                 #[1, 4, 27, 256, 3125]
9 f = a + 1                  #[2, 3, 4, 5, 6]
```

14 Pandas - Python and data analysis

Pandas ist ein Python-Modul zur Daten Manipulation und Analyse. Die Bibliothek baut auf Numpy auf. Scipy und Matplotlib stellen eine sinnvolle Ergänzung zu Pandas da.

14.1 Datenstrukturen

14.1.1 Series

Eine Series ist ähnlich dem eindimensionalen Array. Ein Array fungiert hierbei als Index (Label), in einem weiteren Array werden die Daten gespeichert (Werte).

```
1  import pandas as pd
2  import numpy as np
3
4  s = pd.Series([10, 20, 30, 40, 50])
5
6  #Indizies umbennen
7  s.index = ['START', 'QUARTER', 'HALF', 'THREE-QUARTER', 'ENDE']
8
9  #Series mit bestimmten Index erstellen
10 ind = ['START', 'QUARTER', 'HALF', 'THREE-QUARTER', 'ENDE']
11 t = pd.Series([100, 200, 300, 400, 500], index=t )
12
13 #Zugriff Index
14 s['START']
15
16 #Informationen zu Series
17 s.describe()
```

14.1.2 Data Frame

Ein Data Frame besitzt einen Zeilen- und einen Spaltenindex. Es ist vergleichbar mit einem Dictionary aus Series mit einem normalen Index. Jede Series wird über einen Index (Name de Spalte) angesprochen. Bei den meisten Operationen wird eine Kopie des Data Frames erzeugt und nicht der Data Frame selbst geändert.

```
1  import pandas as pd
2  import numpy as np
3
4  years = range(2014, 2018)
5  a = pd.Series([10.1, 10.2, 10.3, 10.4], index=years)
6  b = pd.Series([20.1, 20.2, 20.3, 20.4], index=years)
```

```
c = pd.Series([30.1, 30.2, 30.3, 30.4], index=years)

#Series aneinanderreihen (hintereinander)
d = pd.concat([a, b, c])

#Data Frame erzeugen aus Series
df = pd.concat([a, b, c], axis=1)

#Spaltennamen
df.columns                                  #Ausgabe der Range
shops_df.columns.values          #Ausgabe der Werte

df.columns = ['a', 'b', 'c']                #Spaltennamen umbenennen

#Spaltennamen vor Konkatenierung angeben
a.name = 'a_new'
b.name = 'b_new'
c.name = 'c_new'

#Zugriff auf Spalte durch Indizierung
df['a','b']
print(df.a)

#Zugriff auf bestimmte Reihen
df[0:2]
df.loc[2017]              #liefert Ergebnis als Series
df.loc[[2016, 2017]]      #liefert Ergebnis als Data Frame

#Zugriff auf Reihen und Spalten
df[0:2, ['a', 'b']]

#Bedingter Zugriff
df[df.a > 10]

#Spalten umbenennen
df2 = df.rename(columns={'a'='alt'})

#Reihen umsortieren
df.reindex(index=[2017, 2016, 2015, 2014])
```

Fehlende Werte

```
#Reihen mit fehlende Werte entfernen
df.dropna()

#Fehlende werte ersetzen
df.fillna()
```

14.2 Map and Apply

14.2.1 Map

```
#Elementweise auf Serie
df['str'].dropna().map(lambda x : 'map_' + x)
```

14.2.2 Apply

```
#Arbeitet auf Spalten-/Reihenbasis (axis=0 rows, axis=1 cols)
f = lambda x : x +1
df.apply(f, axis=0)
```

14.2.3 Applymap

```
#Elementweise
f = lambda x : x +1
df.applymap(f)
```

14.3 Vectorized Operations

```
df = pd.DataFrame(data={"A":[1,2], "B":[4,5]})
df["C"] = df["A"]+df["B"]
```

14.4 Groupings

```
#erzeugt Objekt
ef = df.groupby('Spaltenname')

#Anzeigen pro Gruppierungselement
ef.size()

#Funktionen sind anwendbar
ef.A.sum()
```

14.5 Pandas Stats

```
#Statistik zum Data frame
df.describe()

#Kovarianz
df.cov()

#Korrelation
df.corr()
```

14.6 Merge und Join

14.6.1 Left Join

nur Keys des linken Frame

```
1 pd.merge(left_frame, right_frame, on='key', how='left')
```

14.6.2 Right Join

nur Keys des rechten Frame

```
1 pd.merge(left_frame, right_frame, on='key', how='right')
```

14.6.3 Outer Join

Full Join, alle Keys, falls Key in einem Frame nicht vorhanden NaN

```
1 pd.merge(left_frame, right_frame, on='key', how='left')
```

14.6.4 Inner Join

Intersection beider Frames (nur Keys, die in beiden Frames vorhanden)

```
1 pd.merge(left_frame, right_frame, on='key', how='left')
```

15 Matplotlib

15.1 Line Plot

```
1  import numpy as np
2  import matplotlib . pyplot as plt
3
4  x = np.linspace(0, 100, 1000)
5  y = np.power(x, 2)
6  z = np.power(x, 3)
7
8  plt.plot(x,y, 'b-', x, z, 'go')
9
10 #limits
11 plt.xlim(1, 50)
12 plt.ylim(0,5000)
13
14 #labels
15 plt.xlabel("X-Achse")
16 plt.ylabel("Y-Achse")
17 plt.title("Graph")
18
19 #Legende
20 plt.legend(('$x^2$', '$x^3$'))
21
22 plt.savefig("fig1.png")
23 plt.show()
```

15.2 Scatter Plot

```
1  import numpy as np
2  import matplotlib . pyplot as plt
3
4  npoints = 100
5  x = np.random.standard_normal(npoints)
6  y = np.random.standard_normal(npoints)
7
8  plt.scatter(x,y)
9
10 plt.savefig("fig1.png")
11 plt.show()
```

15.3 Box Plot

```
1  import numpy as np
2  import matplotlib . pyplot as plt
3
4  npoints = 100
5  x = np.random.standard_normal(npoints)
6
7  plt.boxplot(x)
8
9  plt.savefig("fig1.png")
10 plt.show()
```

15.4 Mehrere Plots

Subplot erhält drei Argumente (Reihen, Spalten, Nr. des Subplots)

```
1  import numpy as np
2  import matplotlib . pyplot as plt
3
4  def f(t):
5      return np.exp(t) * np.cos(2*np.pi*t)
6
7  t1 = np.arange(0.0, 5.0, 0.1)
8  t2 = np.arange(0.0, 5.0, 0.02)
9
10 fig = plt.figure()
11
12 axis1 = fig.add_subplot(211)
13 plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
14 axis1.set_title("plot1")
15
16 axis2 = fig.add_subplot(212)
17 plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
18 axis1.set_title("plot2")
19
20 plt.savefig("fig1.png")
21 plt.show()
```

16 Changelog

V0.1 - 2019.11.14

- Start Bearbeitung

V0.1 - 2020.01.11

- Kapitel 06 bis 10 hinzugefügt

V0.1 - 2020.01.12

- Kapitel 11 & 12 hinzugefügt

V0.1 - 2020.01.13

- Kapitel 13 bis 15 hinzugefügt