Addendum

December 5, 2019

1 Addendum

1.1 Namen in Funktionen

Wie findet man in einer Funktion den Wert für einen Namen?

1.1.1 Ist ein Name lokal?

- Prüfen, ob im Scope der Funktion an den Namen *zugewiesen* wird: steht links von einem =

- Wenn ja: *lokaler Name*
- Wenn nein: *nichtlokaler Name*

1.1.2 Werte suchen

- Für lokalen Namen:
- **NUR** im lokalen Namensraum des Scopes suchen
- **NICHT** im umgebenden, globalen Namensraum
- Für nicht-lokalen Namen:
- **ZUERST** im lokalen Namensraum suchen
- **DANN** im umgebenden, globalen Namensraum

1.1.3 Details

- https://docs.python.org/3/faq/programming.html#id9
- https://docs.python.org/3/faq/programming.html#id8

1.2 for-Schleifen - Ersetzung

Wie würde man eine for-Schleife durch Zuweisungen ersetzen?

Beispiel:

[ ]:
```
# Ausgangsbeispiel:
```

```
Liste = [42, 24, 99, 102]
summe = 0
for val in Liste:
    summe += val
print(summe)
```

Könnte man grob umschreiben zu:

[ ]:
```
Liste = [42, 24, 99, 102]
summe = 0

# for - Schleife auflösen
# zwei Namen:
# 1.) Namen für die Aufzählung ünber die iteriert wird
for_aufzaehlung = Liste
# 2.) Name für Index in die Aufzaehlung
for_index = 0
# Erster Schleifendurchlauf
# Vorbeitung:
val = for_auzaehlung[for_index]
# Eigentlicher Schelifenrumpf - stupide kopieren!
Liste[1] = 17
summe += val
# Ende des ersten Durchlaufs: Index erhöhen
for_index += 1

# Zweiten Durchlauf?
# JETZT prüfen: ist for_index kleiner als Länge von for_aufzaehlung!!
# Zweiter Schleifendurchlauf
# Vorbeitung:
val = for_auzaehlung[for_index]
# Eigentlicher Schelifenrumpf - stupide kopieren!
summe += val
# Ende des ersten Durchlaufs: Index erhöhen
for_index += 1



# Nach Schleife: Pseudo_Namen zerstören:
# delete for_aufzaehlung
```

1.2.1 Beispiel: pop in for

Was passiert, wenn man in `pop` auf die Aufzählung anwendet, über die iteriert wird?

[ ]:
```
L = [1,2,3,4,5]
for l in L:
    print("l: ", l)
    l.pop()
    print("L: ", L)

print(L)
```

1.3 Sieb des Erathostenes

Wir entwickeln den Code durch schrittweise Verfeinerung von einer umgangssprachlichen Formulierung!

[ ]:
```
# Beispiel-Code
def era(n):
    """Gib Liste aller Primzahlen kleiner gleich n

    Idee: Sieb des Erathos...
    Annahme: n >= 2
    """

    # Liste mit Wahrheitswerten
    sieb = [False] * (n+1)  # von 0 bis n (also n+1 viele)

    for z in range(2, n+1):   # 2, ... n:

        if sieb[z] == False:
            # neue Primzahl gefunden!
            # für Beispil z == 17: x in (34, 51, 68, ... irgendwas kleiner als
 ↪n )
            for vielfache in range(2*z, n, z):
                sieb[vielfache] = True


    # Ergebnis produzieren

    ergebnis = []
    index = 2
    while index < n:
        if sieb[index] == False:
            ergebnis.append(index)
        index += 1

    return ergebnis

primzahlen = era(10)
```

```
print(primzahlen)
```

2 Selection Sort

Sortiere eine Liste aufsteigend.

[ ]:
```
def finde_index_des_kleinsten_Elementes(Liste, startindex):
    """Finde ab startindex den Index des kleinsten Elementes;
    an Aufrufer zurückgeben
    """

    index_kleinstes = startindex # zeigt auf kleinstes bis jetzt gefundes␣
↪Element
    # Alle Werte von startindex bis Lände der Liste angucken!
    # Wenn kleiner als bis jetzt kleinstes: neues kleisntes Element; index␣
↪aktualisiern!
    # Wenn größer oder gleich: tue nichts
    for kandidaten_index in range(startindex, len(Liste)):
        if Liste[index_kleinstes] > Liste[kandidaten_index]:
            index_kleinstes = kandidaten_index

    return index_kleinstes


L = [45, 22, 17, 99, 105]
assert finde_index_des_kleinsten_Elementes(L,0) == 2
assert finde_index_des_kleinsten_Elementes(L,3) == 3

L = ["abc", "qvw", "def"]
assert finde_index_des_kleinsten_Elementes(L,0) == 0

L = ["qvw", "abc", "def", "aaa", ]
assert finde_index_des_kleinsten_Elementes(L,0) == 3
```

[ ]:
```
"""Selection Sort"""


def selection_sort(L):
    """Sortiere die Liste L in aufsteigender Reihenfolge"""

    # Schaue alle Listenplätze i an:
    for aktueller_index in range(len(L)):
        # BEstimme welches Element auf Platz i kommt
        # Also: das kleinste Element von i bis länge(L)
        # kompliziert!!! -> Funktion dafür bauen
```

```
        # index_kleinstes_Element = ... ??
        index_kleinstes = finde_index_des_kleinsten_Elementes(L,␣
↪aktueller_index)

        # in index_kleinstes: steht der index des kleinstes Elements
        L[aktueller_index], L[index_kleinstes] = L[index_kleinstes],␣
↪L[aktueller_index]


# Test-Code:
liste = [1, 175, 4, 32, 19]
selection_sort(liste)
assert liste == [1, 4, 19, 32, 175]

# Test-Code:
liste = []
selection_sort(liste)
assert liste == []


# Test-Code:
liste = [17]
selection_sort(liste)
assert liste == [17]


# Test-Code:
liste = [1, 2, 3, 4]
selection_sort(liste)
assert liste == [1, 2, 3, 4]

# Test-Code:
liste = ["abc", "qvw", "def"]
selection_sort(liste)
assert liste == ["abc", "def", "qvw"]


print(liste)



# Test-Code:
liste = [1+3, 4+6, 2+3]
selection_sort(liste)
assert liste == [4, 5, 10]
```

```
print(liste)


# Test
liste = [1, 3, 3, 2 ]
selection_sort(liste)
assert liste == [1, 2, 3, 3]
print(liste)
```

3 Merge zweier Listen

Gegeben zwei aufsteigend sortierte Listen. Produziere eine neue Liste, die die Elemente der beiden Listen in aufsteigend sortierter Form beinhaltet.

[ ]:
```
"""Merge"""

def mische_listen(La, Lb):
    """Erzeuge eine neue aufsteigende sortierte Liste
    mit den gleichen Elementen in der gleichen Anzahl! wie La und Lb

    Voraussetzung: La und Lb sind bereits aufsteigend sortiert

    Beispiel:
    La = [1, 2, 2, 2, 5, 7]
    Lb = [3, 4, 5, 8, 9]

    """

    Lresultat = []

    while ( len(La)>0 and len(Lb)>0 ):
        if La[0] < Lb[0]:
            Lresultat.append(La[0])
            La = La[1:]
        else:
            Lresultat.append(Lb[0])
            Lb = Lb[1:]

    Lresultat.extend(La)
    Lresultat.extend(Lb)

    return Lresultat

L1 = [1, 2, 3, 9 ]
L2 = [2, 6, 7, 11]
```

```
%timeit Lgemischt = mische_listen(L1, L2)
print(Lgemischt)
print(L1)
```

[ ]:
```
def mische_via_index(La, Lb):

    aindex = 0
    bindex = 0
    r = []

    while (aindex < len(La)) and (bindex < len(Lb)):
        if a[aindex] < b[bindex]:
            r.append(a[aindex])
            aindex += 1
        else:
            r.append(b[bindex])
            bindex += 1

    if aindex == len(a):
        r.extend(b[bindex:])
    else:
        r.extend(a[aindex:])

    return r

L1 = [1, 2, 3]
L2 = [2, 6, 7]

Lgemischt = mische_listen(L1, L2)
print(Lgemischt)
```

[ ]:
```
# Zeitmessung: welches Mischen ist schneller?
# Vergleiche %timeit hier: https://ipython.readthedocs.io/en/stable/interactive/
 ↪magics.html

def messe_mischen(a, b):
    print("Mischen mit Slicing:")
    %timeit r = mische_listen(a, b)

    print("Mischen mit Index:")
    %timeit r = mische_via_index(a, b)


print("Messung mit range:")
a = list(range(10000))
b = list(range(10000))
```

```
messe_mischen(a, b)

print("Messung mit zufälliger Liste: ")
import random
a = [random.randrange(10000) for i in range(10000)]
b = [random.randrange(10000) for i in range(10000)]
a.sort()
b.sort()
messe_mischen(a, b)
```

4 Binäre Suche

Gegebene eine Liste und ein zu suchender Wert. Schreibe eine Funktion, die den Wert mit binärer Suche in der Liste sucht und zurückgibt: (False, None), wenn der Wert nicht in der Liste ist, (True, Index), wenn der Wert an der Stelle Index zum ersten Mal auftaucht

[ ]:
```
"""binary search"""

def binaere_suche(liste, key):
    """Suche key in liste
    Wenn vorhanden: gibt (True, position des keys )  zurück
    Wenn nicht vorhanden: gib (False, None) zurück

    Annahme: liste muss aufsteigend sortiert sein

    Beispiel:
    binaere_suche([1, 2, 3, 4, 5], 7) == (False, None)
    binaere_suche([1, 2, 4, 8, 16, 32, 64], 32) == (True, None)
    binaere_suche(["abc", "def", "ghj", "klm"], "def") == (True, 1)
    """

    gefunden = False
    unten = 0 # kleinster INDEX ds zu betrachtenden Intervals
    oben = len(liste)-1 # größter Index (!!!) des zu betrachtenden Intervals␣
↪der Liste

    while (not gefunden
           and
           (unten <= oben)
           ):

        # schaue in die Mitte zwischen unten und oben
        mitte = int((unten+oben)/2)

        print("Vor if: ")
```

```
        print(unten, oben, mitte)

        if (liste[mitte] == key):
            # HURRA!
            gefunden = True
        elif (liste[mitte] < key):
            unten = mitte + 1  # ??
        else:
            oben = mitte - 1

        print("Nach if: ")
        print(unten, oben)
        print("-------------------")


    print("DONE ")
    print("=?=========================")
    if gefunden:
        return (gefunden, mitte)
    else:
        return (False, None)

assert binaere_suche([], 17) == (False, None)
assert binaere_suche([1, 2, 3, 4, 5], 7) == (False, None)
assert binaere_suche([1, 2, 4, 8, 16, 32, 64], 32) == (True, 5)
assert binaere_suche([1, 2, 4, 8, 16, 32, 64], 1) == (True, 0)
assert binaere_suche([1, 2, 4, 8, 16, 32, 64], 64) == (True, 6)
assert binaere_suche(["abc", "def", "ghj", "klm"], "def") == (True, 1)
```

5 Türme von Hanoi

[ ]:
```
def hanoi(hoehe, von, nach, ablage):
    """Bestimme Liste der Bewegungsanweisungen (als Tupel von->nach)
    um Turm der Höhe hoehe von von nach nach zu bewegen
    unter Benutzung von ablage """

    if hoehe == 0:
        return []
    if hoehe == 1:
        return [(von, nach)]
    if hoehe > 1:
        teil1 = hanoi(hoehe-1, von=von, nach=ablage, ablage=nach)
        zwischenschritt = hanoi(1, von=von, nach=nach, ablage=ablage)
        teil2 = hanoi(hoehe-1, von=ablage, nach=nach, ablage=von)
```

```
        return teil1 + zwischenschritt + teil2

assert hanoi(0, "A", "C", "B") == []
assert hanoi(1, "A", "C", "B") == [("A", "C")]
assert hanoi(2, "A", "C", "B") == [("A", "B"), ("A", "C"), ("B", "C")]

hanoi(6, "A", "C", "B")
```

6 Methodenaufruf - Was passiert wirklich?

- Methoden sind Funktionen, die im Namensraum einer Klasse vereinbart wurden
- Methodenaufrufe sind also Funktionsaufrufe
- Namensräume sind - für die Diskussion hier - fast das gleiche wie dictionaries
- Klassen haben Namensraum

Frage: **WELCHE** Funktion wird mit *welchen* Paramtern aufgerufen?

6.1 Fall 1: Aufzurufende Methode wird mittels Klasse identifiziert

[8]:
```
# Definition der Klasse:
class A:
    def m(x, y, z):
        # Mache irgendwas sinnvolles mit x, y, z
        print(x, y, z)

# Aufruf der Methode a in A:
A.m(5, 6, 7)

# d.h:
# 1.) A: Suche den Namen A. Diesen Namen gibt es im globen Namensraum, er
#       verweist auf eine Klasse, also gibt es einen zugehörigen  Namensraum
# 2.) A.m: Im gerade gefundenen Namensraum A, suche nach einem Namen a. Den␣
 ↪gibt es. Der verweist auf eine Funktion
# 3.) A.m(...): Rufe die gerade gefundene Funktion A.a auf
# 4.) A.m(5, 6, 7): Übergebe beim Aufruf die Werte 5, 6, 7 an die Parameter x,␣
 ↪y, z der Funktion A.a.
#     Also passiert: x=5, y=6, z=7.
# 5.) Führe den Block von A.a aus (in diesem Beispiel passiert natürlich nichts␣
 ↪)
```

```
5 6 7
```

6.2 Fall 2: Aufzurufende Methode wird mittels Objekt identifziert

Fast das gleiche wie oben, mit zwei Änderungen: * Die Suche nach der richtigen Methode ist etwas komplizierter * Beim Aufruf werden etwas andere Parameter übergeben

[ ]:
```
# Klassendefinition wie oben


# Objekt instantiieren:

o1 = A()
o2 = A()
# o1, o2 sind jetzt Namen auf ZWEI verschiedene Instanzen der Klasse A

# Aufruf:
o1.m(10, 11)
# gedanklic umgeschrieben:
o1.__class__.m(o1, 10, 11)
# Dabei passiert folgendes:
# 1.) o1.m(...) : Offenbar soll eine Methode aufgerufen werden. Ein Objekt hat␣
 ↪aber - streng genommen - keine Methode; nur
#     Klassen haben Methoden. Deswegen müssen wir das umschreiben:
# 2.) Umschreiben von o1.m(10, 11) zu
#     o1.__class__.m(o1, 10, 11)
#     ALLGEMEINE REGEL:
#         OBJEKT.METHODE(PARAMETERLISTE) wird umgeschrieben zu:
#         OBJEKT.__class__.METHODE(OBJEKT, PARAMETERLISTE)
#     Dieser Ausdruck wird vom Interpreter nun ausgewertet.
# 3.) o1: Suche den Namen o1. Der wird im globalen Namensraum gefunden (wir␣
 ↪sind ja nicht im Scope einer Funktion).
#     Der Name existiert, verweist auf ein Objekt. Ein Objekt hat einen␣
 ↪Namensraum (denke: Disctionary)
# 4.) o1.__class__ : Im Namensraum von o1 wird nach einen Eintrag __class__␣
 ↪gesucht. Der existiert (wird
#    automatisch beim Instantiieren eines Objektes angelegt). o1.__class__ ist A
# 5.) Also haben wird jetzt: A.m(o1, 10, 11)
#     Das kann aber genau wie oben behandelt werden
# 6.) Der Aufruf ist also A.m mit x=o1, y=10, z=11

# Zweites Beispiel
o2.m(20, 21)
# wird umgeschrieben zu:
# o2.__class__.m(o2, 20, 21)
# und dann entsprechend behandelt
```

6.3 Beobachtungen

- Die Methode A.m hat drei formale Parameter. Sie muss also mit drei Werten aufgerufen werden. Welche das sind, wo die herkommen, das ist der Methode (eine Funktion!) ganz egal
- Die Methode A.m hat keinen Parameter `self` als explizitem Namen. Dennoch wird im zweiten Beispiel `o1` an `x` übergeben. Das ist kein guter Stil, das ist unüblich, aber es illustriert hoffentlich, dass es ein **ganz normaler Funktionsaufruf** mit drei Parametern ist, bei denen halt das erste Arguemt ein Name für ein Objekt ist.
- Das ist der Methode an sich erstmal egal. Es hängt davon ab, was die Methode damit macht.

7 Objekt-Instantiierung

Oder: Wo kommt der erste Parameter von `__init__()` her?

Details für Liebhaber: https://eli.thegreenplace.net/2012/04/16/python-object-creation-sequence

[ ]:
```
class C:
    # eine KLASSEN-Methode, die implizit vorhanden ist:
    def __new__(XX, *args, **kwargs):
        # (über den ersten Parameter müssen wir später noch reden)
        neues_Objekt = erzeuge_neues_Objekt_der_Klasse_C()  # neu, aber leer!
        C.__init__(neues_Objekt, *args, **kwargs)
        return neues_Objekt

    # eine ganz normale INSTANZ-Methode:
    def __init__(zu_initialisierendes_Objekt, *args, **kwargs):
        # mache etwas sinnvolles mit zu_initialiserendes_Objekt, args, kwargs
        pass
        zu_initialisierendes_Objekt.irgendwas = irgendwas
        # KEIN return - das macht __new__


# Instantiiere ein Objekt:

# Das hier:
c = C(1,2,3)
# bedeutet mehr oder minder (ein paar Details fehlen):
c = C.__new__(1,2,3)
```

8 Decorator: Cache

[7]:
```
def cache(fct):
    """Dekorator für Cache für Funktionen mit EINEM Parameter"""
    cachedata = {}

    def _cache(arg):

        print("cache: arg: {}, id: {}".format(arg, id(cachedata)))

        if arg in cachedata:
            return cachedata[arg]
        else:
            r = fct(arg)
            cachedata[arg] = r
            return r

    return _cache

@cache
def fib(n):
    print("fib: {}".format(n))
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

@cache
def fib2(n):
    print("fib2: {}".format(n))
    if n == 0 or n == 1:
        return 2
    else:
        return fib2(n-1) + fib2(n-2)

print(fib(6))
print(fib2(5))
```

```
cache: arg: 6, id: 4565651960
fib: 6
cache: arg: 5, id: 4565651960
fib: 5
cache: arg: 4, id: 4565651960
fib: 4
cache: arg: 3, id: 4565651960
fib: 3
cache: arg: 2, id: 4565651960
```

```
fib: 2
cache: arg: 1, id: 4565651960
fib: 1
cache: arg: 0, id: 4565651960
fib: 0
cache: arg: 1, id: 4565651960
cache: arg: 2, id: 4565651960
cache: arg: 3, id: 4565651960
cache: arg: 4, id: 4565651960
13
cache: arg: 5, id: 4566508096
fib2: 5
cache: arg: 4, id: 4566508096
fib2: 4
cache: arg: 3, id: 4566508096
fib2: 3
cache: arg: 2, id: 4566508096
fib2: 2
cache: arg: 1, id: 4566508096
fib2: 1
cache: arg: 0, id: 4566508096
fib2: 0
cache: arg: 1, id: 4566508096
cache: arg: 2, id: 4566508096
cache: arg: 3, id: 4566508096
16
```

9 Dependency injection

Oder: wie vermeidet man Abhängigkeiten zwischen Klassen; wie sorgt man dafür, dass eine Klasse nur einen Zweck hat?

9.1 Beispiel: Pizzeria und Teigfabrik

Teig kneten, Pizza Bestellung annehmen, backen, ...

9.1.1 Erste Idee: Eine Klasse, TeigPizzeria

- Alles in einer Klasse
- Problem: Was mache ich, wenn ich die Art der Teigproduktion ändern will? Ganze Klasse ändern? ...?

[ ]:
```python
class TeigPizzeria():
    def get_teig(self): pass
```

```
    def bestelle(self, belag="Käse"):
        t = self.get_teig()
        p = t + belag
```

9.1.2 Besser: Zwei Klassen

- Klasse `Teigfabrik`: wie macht man aus Mehl Teig?
- Klasse `Pizzeria` : wie macht man aus Teig eine Pizza?

9.2 Option 1: Pizzeria has-a Teigfabrik

9.2.1 Option 1.a: Durch Konstruktor

[ ]:
```
class Teigfabrik():
    def get_teig(self): pass

class Pizzeria():

    def __init__(self, tf):
        self.teigfabrik = tf

    def bestelle(self, belag="Käse"):
        t = self.teigfabrik.get_teig()
        p = t + belag

p = Pizzeria(tf = Teigfabrik())

class Bioteigfabrk(Teigfabrik):
    pass

bio_p = Pizzeria(tf = Bioteigfabrik())
```

9.2.2 Option 1.b: Setter!

[ ]:
```
class Teigfabrik():
    def get_teig(self): pass

class Pizzeria():

    def set_teigfabrik(self, tf):
        self.teigfabrik = tf

    def bestelle(self, belag="Käse"):
        t = self.teigfabrik.get_teig()
```

```
        p = t + belag

p = Pizzeria()
p.set_teigfabrik(tf = Teigfabrik())

class Bioteigfabrk(Teigfabrik):
    pass

bio_p = Pizzeria()
bio_p.set_teigfabrik(tf = Bioteigfabrik())
```

9.3 Option 2: Mittels multiple inheritance

Details und Beispiel siehe Kapitel 12