OST
Ostschweizer
Fachhochschule

Python for Machine Learning Übungen

Part 2

OST
Ostschweizer Fachhochschule

Oberstrasse 10
8640 Rapperswil
Switzerland

T +41 79 350 24 36
michael_inden@hotmail.com
ost.ch

Michael Inden

3 Aufgaben zu Klassen

Aufgabe 1: SuperHero

In dieser Aufgabe soll eine Klasse `SuperHero` implementiert werden, die Superhelden mit ihren Namen, ihren Superkräften und ihrer Stärke modelliert sowie eine lesbare Konsolenausgabe produziert. Zudem soll eine Methode `is_stronger_than(other)` prüfen, ob ein Superheld stärker als ein anderer ist. Schließlich ist eine allgemeingültige Prüfung zu realisieren, die den stärksten Superhelden aus einer Menge zurückgibt. Dazu soll eine Methode `strongest_of(*heros)` mithilfe von Var Args erstellt werden. Folgendes Hauptprogramm zeigt die Verwendung und mögliche Aufrufe – variieren Sie gern die Eigenschaften.

```
def main():
    superman = SuperHero("Superman", "Kryptonite-Power", 1_000)
    batman = SuperHero("Batman", "Techno-Power", 100)
    ironman = SuperHero("Ironman", "Techno-Power", 500)

    print("Superman stronger than Batman?",
          superman.is_stronger_than(batman))
    print(SuperHero.strongest_of(superman, batman, ironman))


if __name__ == "__main__":
    main()
```

Aufgabe 2: Counter

Nachdem die Grundbegriffe beim objektorientierten Entwurf mit Python bekannt sind, soll nun ein Zähler als Klasse entworfen werden, der folgende Anforderungen erfüllt:

1. Er lässt sich auf den Wert 0 zurücksetzen.
2. Er lässt sich um eins erhöhen.
3. Der aktuelle Wert lässt sich abfragen.

Dabei existiert bereits das folgende, rudimentäre Grundgerüst:

```
class Counter:
    def __init__(self, count):
        self.count = count
```

Das Attribut `count` speichert den Zähler und kann von überall abgefragt und verändert werden. Das Rücksetzen erfolgt durch Übergabe von 0. Die obige Implementierung sollen Sie nun erweitern und verbessern, indem Sie zu den Anforderungen passende Methoden implementieren und als Kür das Attribut „verstecken“.

Aufgabe 3: Vererbung

Erstellen Sie zwei Basisklassen `ExcludeFilter` und `IncludeFilter` sowie basierend darauf einen SPAM-Filter und einen Namensfilter, die jeweils eine `filter()`-Methode anbieten und wie folgt aufgerufen werden:

```
spamfilter = SPAMFilter()
print(spamfilter.filter(["SPAM", "DAS", "IST", "SPAM", "SPAM", "DETECTED"]))

namefilter = NameFilter()
print(namefilter.filter(["DAS", "SIND", "MICHAEL", "UND", "SOPHIE"]))
```

Das gewünschte Ergebnis ist wie folgt:

```
['DAS', 'IST', 'DETECTED']
['MICHAEL', 'SOPHIE']
```

BONUS: Führen Sie nun noch eine abstrakte Basisklasse `AbstractFilter` ein, die lediglich die Signatur der Methode vorgibt. Modifizieren Sie die Konstruktion und Aufrufe passend.

Aufgabe 4: Vererbung & Duck-Typing

Warum macht die Definition von abstrakten Klassen oder Interfaces in Python weniger Sinn als in anderen Programmiersprachen? Stichwort: ***Duck-Typing***. Google'n Sie ein wenig zu dem Thema.

4 Aufgaben zu Collections

Aufgabe 1: Tennisverein-Mitgliederliste

Nehmen wir an, wir verwalten die Mitglieder des Tennisvereins in Form einer Liste. Implementieren Sie folgenden Registrierungsprozess: Zunächst melden sich Michael, Tim und Werner an. Prüfen Sie, ob sich Jana schon angemeldet hat. Nun registriert sich Andreas. Schließlich melden sich Lili, Jana und Natalija an. Geben Sie zum Abschluss die Anzahl der Mitglieder aus.

Aufgabe 2: Person als Named Tuple und Erwachsene extrahieren

Nehmen wir an, es wir modellieren Personen in Form eines Named Tuple mit den Attributen `name` und `age`. Zudem werden einige Personen als Liste wie folgt bereitgestellt:

```
persons = [Person("Mike", 37), Person("Tim", 49), Person("Tom", 5), Person("Michael", 50),
           Person("Jim", 7), Person("James", 17)]
```

Filtern Sie die Erwachsenen mithilfe einer List Comprehension heraus.

Aufgabe 3: Teiler als List Comprehension

Vereinfachen Sie die vorgegebene Implementierung der Funktion `find_proper_divisors(n)`, durch den Einsatz von List Comprehensions.

```
def find_proper_divisors(value):
    divisors = []
    for i in range(1, value // 2 + 1):
        if value % i == 0:
            divisors.append(i)
    return divisors
```

Aufgabe 4: Contains Subsequence

Mit dem Operator `in` kann man leider nur einzelne Elemente auf Enthaltensein prüfen, nicht aber Abfolgen (nicht unbedingt zusammenhängende). Dazu hatten wir auf den Folien mit einer Stringumwandlung etwas getrickst. Schreiben Sie dies hier mit passenden Methoden von Listen selbst, ohne Strings einzusetzen.

Dabei benötigt man Exception Handling der folgenden Form:

```
try:
    do_something()
except ValueError:
    handle_exception()
```

Aufgabe 5: Check Magic Triangle

Schreiben Sie eine Funktion `is_magic_triangle(values)`, die prüft, ob eine Folge von Zahlen ein magisches Dreieck bildet. Ein solches ist definiert als ein Dreieck, bei dem die jeweiligen Summen der Werte der drei Seiten alle gleich sein müssen. Beginnen Sie mit einer Vereinfachung für die Seitenlänge 3 und einer Funktion `is_magic6(values)`.

Beispiele Nachfolgend ist das für je ein Dreieck der Seitenlänge drei und der Seitenlänge vier gezeigt:

```
  1          2
 6 5        8 5
2 4 3      4   9
          3 7 6 1
```

Damit ergeben sich folgende Seiten und Summen:

Eingabe	Werte 1	Werte 2
Seite 1	1 + 5 + 3 = 9	2 + 5 + 9 + 1 = 17
Seite 2	3 + 4 + 2 = 9	1 + 6 + 7 + 3 = 17
Seite 3	2 + 6 + 1 = 9	3 + 4 + 8 + 2 = 17

Aufgabe 6: Suche mit `rindex()` und `rfind()`

Für Zeichenketten (Strings) gibt es die Funktionen `rindex()` und `rfind()`, um die Position eines gewünschten Elements ab dem Ende der Zeichenkette zu finden. In den Folien hatten wir eine Variante für Listen gesehen, die auf einer Kopie der Daten arbeitet. Das ist für umfangreiche Datenbestände suboptimal. Entwickeln Sie eine Abwandlung, die diese Probleme nicht besitzt.

KÜR: Als Kür schreiben Sie das Ganze so um, dass die Suche ab einer gewissen Position gestartet werden kann.

Aufgabe 7: Einfügen und Löschen mit Slicing

In dieser Aufgabe sollen Sie sich mit Slicing vertraut machen. Fügen Sie am Anfang und am Ende sowie in der Mitte Werte ein. Nun löschen Sie einen Bereich in der Mitte, etwa die Positionen 4 bis 7. Schließlich löschen Sie noch alles was zwischen Position 1 und der vorletzten Position ist.

Aufgabe 8: Mengen und ihre Operationen

Zunächst ist es Ihre Aufgabe, Duplikate aus einer Liste mit Städtenamen zu entfernen:

```
cities = ["Kiel", "Hamburg", "Zürich", "Bremen",
          "Hamburg", "Zürich", "Kiel", "Bremen"]
```

Nun sollen Sie für folgende Zahlen die Mengenoperationen berechnen:

```
numbers1 = {1, 1, 2, 3, 5, 8, 13, 21}
numbers2 = {1, 2, 3, 4, 5, 6, 7}
```

Aufgabe 9: Duplikate entfernen

Aus einer Liste sollen Sie die doppelten Einträge entfernen. Dabei gilt die Randbedingung, dass die ursprüngliche Reihenfolge bestehen bleibt. Schreiben Sie dazu eine Funktion `remove_duplicates(values)`.

Eingabe	Resultat
[1, 1, 2, 3, 4, 1, 2, 3]	[1, 2, 3, 4]
[7, 5, 3, 5, 1]	[7, 5, 3, 1]
[1, 1, 1, 1]	[1]

Aufgabe 10: Häufigkeiten von Namen

Stellen Sie sich vor, es wäre eine Liste mit Namen gegeben:

```
names = ["Tim", "Tom", "Mike", "Jim", "Tim", "Mike", "James", "Mike"]
```

Nun wollen Sie wissen, welcher Name am häufigsten vorkommt bzw. genauer, für alle Namen deren Anzahl ermitteln. Bereiten Sie das Ergebnis als Dictionary etwa wie folgt auf:

```
{'Tim': 2, 'Tom': 1, 'Mike': 3, 'Jim': 1, 'James': 1}
```

Aufgabe 11: Zahl als Text

Schreiben Sie eine Funktion `number_as_text(n)`, die für eine gegebene positive Zahl vom Typ `int` die jeweiligen Ziffern in korrespondierenden Text umwandelt.

```
print(number_as_text(721971))
```

Das sollte folgende Ausgabe produzieren:

```
SEVEN TWO ONE NINE SEVEN ONE
```

Aufgabe 12: Morse Code

Schreiben Sie eine Funktion `to_morse_code(input)`, die einen übergebenen Text in Morsezeichen übersetzen kann. Diese bestehen aus Sequenzen von ein bis vier kurzen und langen Tönen pro Buchstabe, symbolisiert durch '.' oder '-'. Zur leichteren Unterscheidbarkeit soll zwischen jedem Ton ein Leerzeichen und zwischen jeder Buchstabentonfolge jeweils drei Leerzeichen Abstand sein – ansonsten würden sich S (...) und EEE (...) nicht voneinander unterscheiden lassen. Beschränken Sie sich der Einfachheit halber auf die Buchstaben E, O, S, T, W mit folgender Codierung

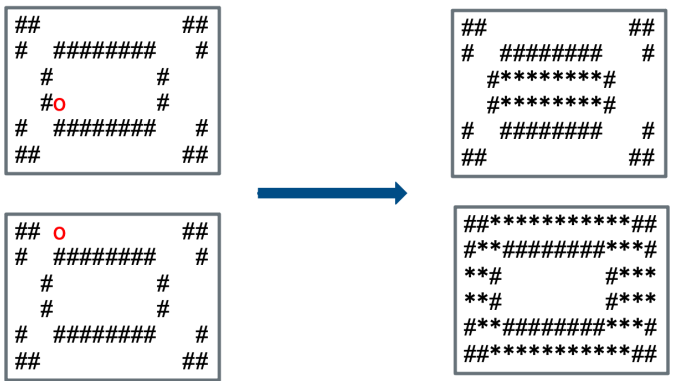
Buchstabe	Morsecode
E	.
O	- - -
S	. . .
T	-
W	. - -

Hier ein paar Beispiele für mögliche Umwandlungen:

Eingabe	Resultat
SOS	. . . - - - . . .
TWEET	- . - - . . -
WEST	. - - . . . . -

Aufgabe 13: Flood-Fill

Schreiben Sie eine Funktion `flood_fill(values2dim, start_x, start_y)`, die in einer verschachtelten Liste alle freien Felder mit einem bestimmten Wert befüllt. Nachfolgend ist der Füllvorgang für das Zeichen '*' gezeigt. Das Füllen beginnt an einer vorgegebenen Position, etwa in der linken oberen Ecke, und wird dann so lange in alle vier Himmelsrichtungen fortgesetzt, bis die Grenzen der Listen oder eine Begrenzung in Form eines anderen Zeichens gefunden wird:



Aufgabe 14: Weg aus Labyrinth

In dieser Aufgabe soll der Ausweg aus einem Labyrinth gefunden werden. Dabei sei ein Irrgarten in Form eines zweidimensionalen Arrays bzw. verschachtelter Listen mit Mauern, symbolisiert durch '#', und Zielpositionen (Ausgängen) in Form von 'X' gegeben. Von einer beliebigen Position aus sollen die Wege zu allen erreichbaren Ausgängen ermittelt und mit '.' markiert werden – liegen zwei hintereinander, so wird nur der erste von beiden geliefert. Dabei kann man sich nur in die vier Himmelsrichtungen bewegen, nicht aber diagonal. Schreiben Sie eine Funktion `find_way_out(values, x, y)`, die jeden gefundenen Ausgang mit `FOUND EXIT at ...` protokolliert.

```
##################################
# #         #    #     #  #   X#X#
#  ##### #### ##   ##  #  # ###  #
#  ##  #    #  ## ##  #  #     # #
#    #  ###  # ## ##   #   ### # #
# #   ####     ## ##      ###  # #
####   #     ####  ####  # ####  #
######   #########   ##   # ###  #
##     #  X X####X #  #  # ###  ##
##################################
FOUND EXIT: x: 30, y: 1
FOUND EXIT: x: 17, y: 8
FOUND EXIT: x: 10, y: 8
##################################
#.#         #....#.....#  #...X#X#
#..##### ####.##...##..#  #.###  #
# .##  #    #..##.## .#  #..   # #
# ...#  ###..#.##.## ..#...### # #
# # ..####.....##.##  ... ###  # #
#### ..#...  ####..####  # ####  #
######...#########.  ##   # ###  #
##     #..X X####X.#  #  # ###  ##
##################################
```

5 Aufgaben zu Collections Advanced

Aufgabe 1: Lambda-Basics

Gegeben seien folgende Namen als Ausgangsdaten:

```
sample_names = ["TIM", "TOM", "MIKE", "JOHN", "MICHAEL", "STEFAN"]
```

Schreiben Sie einen Lambda, der nur die Namen mit ungerader Länge als Ergebnis liefert. Schreiben Sie danach einen Lambda, der basierend darauf die Namen gerade Länge ermittelt.

Nutzen Sie beide in Kombination mit `filter()`.

Aufgabe 2: Lambda-Basics II

Gegeben seien wieder folgende Namen als Ausgangsdaten:

```
sample_names = ["TIM", "TOM", "MIKE", "JOHN", "MICHAEL", "STEFAN"]
```

Schreiben Sie einen Lambda, der als Ergebnis die Namen doppelt ausgibt und davor einen zufällig gewählten Gruß aus Moin, Grüezi, Hallo und Tach, also etwa

```
Moin Tim Tim
Grüezi Michael Michael
```

Aufgabe 3: `map()`

Verwenden Sie die Funktion `map()` und einen Lambda, um die Elemente von zwei Listen zu addieren. Verwenden Sie ein Lambda mit zwei Argumenten und bedenken Sie, dass `map()` mehrere Iterables (hier Listen) entgegennehmen kann.

```
values1 = [100, 200, 300, 400, 500]
values2 = [10, 20, 30, 40, 50]
```

Als Ergebnis ist folgendes gewünscht:

```
[110, 220, 330, 440, 550]
```

Aufgabe 4: `map()`

Verwenden Sie die Funktion `map()` und `reversed()`, um die Texte in einer Listen umzudrehen:

```
countries = ["USA", "Schweiz", "Deutschland", "Frankreich", "Italien"]
```

Das Ergebnis sollte folgendermaßen aussehen:

```
['ASU', 'ziewhcS', 'dnalhcstueD', 'hcierknarF', 'neilatI']
```

Aufgabe 5: `filter()`, `map()` und Lambda

Gegeben seien die ersten Primzahlen wie folgt:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Sie sollten für die Werte jeweils den kubischen Wert (x^3) bestimmen und dann eine Filterung auf alle Werte im Bereich größer als 20 und kleiner als 500 vornehmen.

Das Ergebnis sollte folgendermaßen aussehen:

```
[27, 125, 343]
```

Aufgabe 6: `sorted()`

Gegeben sei eine Liste mit Tupel aus der Einwohnerzahl und dem Städtenamen:

```
inhabitants_and_cities = [(15_000_000, "New York"), (400_000, "Zürich"),
                          (250_000, "Kiel"), (550_000, "Bremen"),
                          (2_000_000, "Hamburg"), (250_000, "Aachen")]
```

Verwenden Sie die Funktion `sorted()` und geeignete Lambdas, um die Elemente in Form von Tupeln zu sortieren.

a) Die Tupel in der Liste sollen anhand des jeweiligen ersten Elements (Einwohnerzahl) aufsteigend sortiert werden.
b) Die Tupel in der Liste sollen anhand des jeweiligen zweiten Elements (Stadt) absteigend sortiert werden.
c) Die Tupel in der Liste sollen nach dem letzten Zeichen der zweiten Elemente in absteigend sortiert werden.

Aufgabe 7: Sortierung verschachtelter Listen

In dieser Aufgabe soll ein Datenbestand jeweils nacheinander nach allen Indizes sortiert werden. Gegeben seien etwa folgende Daten:

```
values = [[5, 1, 2],
          [1, 3, 5],
          [2, 5, 1],
          [3, 2, 3],
          [4, 4, 4]]
```

Dann erwarten wir folgende Ergebnisse:

```
[[1, 3, 5], [2, 5, 1], [3, 2, 3], [4, 4, 4], [5, 1, 2]]
[[5, 1, 2], [3, 2, 3], [1, 3, 5], [4, 4, 4], [2, 5, 1]]
[[2, 5, 1], [5, 1, 2], [3, 2, 3], [4, 4, 4], [1, 3, 5]]
```

Aufgabe 8: Sortierung tabellarischer Daten

In dieser Aufgabe soll ein Datenbestand nach mehreren Kriterien sortiert werden.

1) Sortiere die Daten absteigend nach Alter und aufsteigend nach Name
2) Nach Stadt, absteigend PLZ
3) Nach Stadt, absteigend PLZ, aufsteigend Name

Nutze etwa folgenden Datenbestand

```
# "Werte einer Tabelle"
values = [["Micha", 50, 8047, "Zürich"],
          ["Lili", 42, 8047, "Zürich"],
          ["Micha", 50, 52070, "Aachen"],
          ["Tim", 50, 24107, "Kiel"],
          ["Micha", 50, 24106, "Kiel"],
          ["Michael", 50, 24106, "Kiel"],
          ["Andreas", 50, 21012, "Hamburg"],
          ["Barbara", 48, 22395, "Hamburg"],
          ["Marianne", 83, 28816, "Stuhr"]]
```

Aufgabe 9: Every2nd-Iterator

In dieser Aufgabe soll ein einfacher Iterator implementiert werden, der ausgehend von der Startposition, dann jeweils das 2. Element einer Datenstruktur liefert.

Für die Ausgangsdaten

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

erwarten wir folgendes Ergebnis:

[1, 3, 5, 7, 9]

Aufgabe 10: EveryNth-Iterator

Implementieren Sie einen Iterator namens `EveryNth`, der jedes n-te Element durchläuft. Also für die Eingabe `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]` beispielsweise folgende Werte ausgibt:

- Mit Schrittweite 3: `1, 4, 7, 10`
- Mit Schrittweite 5: `1, 6, 11`

Aufgabe 11: Fibonacci-Iterator

In dieser Aufgabe soll ein Iterator implementiert werden, der die ersten n Fibonacci-Zahlen erzeugt. Nochmals zur Erinnerung:

> Die ***Fibonacci-Zahlen*** lassen sich rekursiv wie folgt definieren:
>
> $$fib(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ fib(n-1) + fib(n-2), & \forall n > 2 \end{cases}$$
>
> Es ergibt sich für die ersten n folgender Werteverlauf:
>
> | **n** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
> |---|---|---|---|---|---|---|---|---|
> | **fib(n)** | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 |
>
> Eine natürliche, wenn auch nicht performante Umsetzung ist die rein rekursive wie folgt:

```
def fib(n):
    if n == 1 or n == 2:
        return 1

    return fib(n-1) + fib(n-2)
```

Tipp: Auf den Folien haben wir bei Listen schon einen Berechnungstrick kennengelernt. Den kann man hier für den Iterator leicht abgewandelt nutzen.

Aufgabe 12: Even-Odd-Generator

Schreiben Sie einen Generator, der alle geraden oder ungeraden Zahlen als unendliche Folge produziert.

Aufgabe 13: Fibonacci-Generator

In dieser Aufgabe soll die bereits als Iterator entwickelte Berechnung der Fibonacci-Zahlen nun in Form eines Generators realisiert werden. Was sind die Vorteile? Wie sieht es mit dem Hinzufügen des Quadrierens der Werte aus?