4 Strings

Strings modellieren Zeichenketten und besitzen den Typ `str`, der eine Vielzahl an Funktionen bietet. In diesem Kapitel werden wir diese Thematik anhand verschiedener Übungsaufgaben kennenlernen.

4.1 Einführung

Strings bestehen aus einzelnen Zeichen und bilden wie Listen sequenzielle Datentypen (vgl. Abschnitt **??**), weshalb viele Aktionen analog dazu ausführbar sind, etwa Slicing. Im Gegensatz zu anderen Sprachen besitzt Python keinen Datentyp für einzelne Zeichen, sondern diese werden einfach als String der Länge 1 repräsentiert.

Strings lassen sich als Zeichenkette in doppelten oder einfachen Anführungszeichen erzeugen, wie dies folgende zwei Zeilen zeigen:

```
str1 = "DOUBLE QUOTED STRING"
str2 = 'SINGLE QUOTED STRING'
```

Praxisrelevante Funktionen

Für Strings gehe ich vor allem auf die gängigsten und in der Praxis nützlichsten Funktionen ein. Nehmen wir an, die Variable `str` wäre ein String. Dann können wir folgende Funktionen aufrufen:

- `len(str)` – Ermittelt die Länge des Strings. Dies ist eine allgemeine Python-Funktion zum Abfragen der Länge von sequenziellen Datentypen wie Listen oder Tupeln usw., aber auch Strings.
- `str[index]` – Ermöglicht den indexbasierten Zugriff auf einzelne Buchstaben.
- `str[start:end]` / `str[start:end:step]` – Extrahiert die Zeichen zwischen den beiden Positionen `start` und `end` - 1. Als Besonderheit lässt sich eine Schrittweite angeben. Interessanterweise kann sogar die Bereichsangabe entfallen und mit `[::-1]` nur eine negative Schrittweite zum Einsatz kommen, wodurch ein neuer String mit umgekehrter Buchstabenreihenfolge des Originals entsteht.
- `str[:end]` – Extrahiert die Zeichen zwischen Anfang und der Position `end` - 1.
- `str[start:]` – Extrahiert die Zeichen zwischen der Position `start` und dem Ende des Strings.

- `str.lower()` / `str.upper()` – Erzeugt einen neuen String, der aus Klein- bzw. Großbuchstaben besteht – Ziffern und Satzzeichen werden nicht umgewandelt.
- `str.strip()` – Entfernt Leerzeichen am Textanfang und -ende und gibt dies als neuen String zurück. Als Besonderheit kann man ein Zeichen übergeben, das dann statt Whitespace entfernt wird.
- `str.isalpha()` / `str.isdigit()` / ... – Prüft, ob alle Zeichen des Strings alphanumerisch, Ziffern usw. sind.
- `str.startswith(other)` / `str.endswith(other)` – Prüft, ob der String mit dem übergebenen String startet bzw. endet.
- `str.find(other)` / `str.rfind(other)` – Sucht nach dem übergebenen String und gibt den Index des ersten Vorkommens zurück oder -1 bei Nichtexistenz. Die Funktion `rfind()` sucht vom Ende. Als Besonderheit kann man in beiden Fällen noch einen Indexbereich angeben.
- `str.index(other, start, end)` / `str.rindex(other, start, end)` – Liefert den Index des ersten bzw. letzten Vorkommens von `other`. Im Gegensatz zu `find()` wird bei Nichtvorhandensein eine Exception ausgelöst.
- `str.count(text)` – Zählt, wie häufig `text` im String vorkommt.
- `str.replace(old, new)` – Erzeugt einen neuen String, in dem alle Vorkommen von `old` durch `new` ersetzt sind.
- `str.split(delim)` – Liefert eine Liste mit Teilstrings, die sich durch Aufteilung des ursprünglichen Strings ergeben. Der Delimiter ist *kein* regulärer Ausdruck[1]. Ohne die Angabe eines Delimiters wird ein Text bezüglich Whitespace aufgespalten.
- `str.join(list)` – Bewirkt das Gegenteil von `split()`. Konkret: Die als Liste übergebenen Elemente werden mit dem String als Delimiter verbunden.
- `str.capitalize()` / `str.title()` – Wandelt das erste Zeichen in Großbuchstaben um. Mit `title()` wird zusätzlich innerhalb eines Strings der Anfang jedes neuen Worts in Großbuchstaben umgewandelt.

Beispiel Umwandlungen und Extraktionen

Schauen wir uns einführend einfache Aktionen auf Strings wie die Umwandlung in Klein- oder Großbuchstaben sowie das Aufspalten an:

```
name = "Karl Heinz Müller"
print(name.lower())
print(name.upper())

print(name.split())

time = '20:26:45'
hrs, mins, secs = time.split(':')
print(hrs, mins, secs)
```

[1] `https://de.wikipedia.org/wiki/Regulärer_Ausdruck`

Das führt zu folgenden Ausgaben:

```
karl heinz müller
KARL HEINZ MÜLLER
['Karl', 'Heinz', 'Müller']
20 26 45
```

Zudem können wir Texte mit * wiederholen und mit `strip()` Textbestandteile, oftmals Whitespace, an den Rändern entfernen:

```
print("-repeater-" * 3)

with_whitespace = "  --CONTENT--  "
stripped1 = with_whitespace.strip()
stripped2 = stripped1.strip("-")
print("strip1:", stripped1, "length:", len(stripped1))
print("strip2:", stripped2, "length:", len(stripped2))
```

Das führt zu folgenden Ausgaben:

```
-repeater--repeater--repeater-
strip1: --CONTENT-- length: 11
strip2: CONTENT length: 7
```

Gleichheit

Nun schauen wir uns die Definition zweier Strings und deren Vergleich an, insbesondere die Auswirkungen von == (inhaltliche Gleichheit) und `is` (Referenzgleichheit):

```
str1 = 'String with same contents but different quotes'
str2 = "String with same contents but different quotes"
str3 = "String with same contents but XXX quotes".replace("XXX", "different")
print("str1:", str1)
print("str2:", str2)
print("str3:", str3)

if str1 == str2:
    print("Inhaltlich gleich")
if str1 is str2:
    print("Referenz str1 / str2 gleich")
if str1 == str3:
    print("Inhaltlich gleich")
if str1 is str3:
    print("Referenz str1 / str3 gleich")
```

Man erhält folgende Ausgaben:

```
str1: String with same contents but different quotes
str2: String with same contents but different quotes
str3: String with same contents but different quotes
Inhaltlich gleich
Referenz str1 / str2 gleich
Inhaltlich gleich
```

Die Ausgabe, dass die Referenzen von `str1` und `str2` gleich sind, mag zunächst überraschen. Wie kommt das? Als Optimierung fasst Python gleiche Objekte manchmal

zusammen. Dieses Verhalten ist jedoch nicht garantiert. Spätestens wenn Aktionen auf den Strings erfolgen, wie oben das `replace()` sind die Referenzen nicht mehr gleich, der Inhalt aber in diesem Fall natürlich schon.

Zugriff auf einzelne Zeichen und Substrings – Slicing

Betrachten wir noch die mächtigen Slicing-Operationen, um einzelne Zeichen, ganze Bestandteile sowie sogar nicht zusammenhängende Bereiche extrahieren zu können. Danach zählen wir Vorkommen und simulieren ein Suchen und Weitersuchen. Schließlich ersetzen wir einen Textbestandteil:

```
strange_message= "a message containing only a message"

mid_chars = strange_message[10:20]
last_seven_chars = strange_message[-7:]
print("mid_chars:", mid_chars, " / last_seven_chars:", last_seven_chars)

first_char = strange_message[0]
print(first_char, "count:", strange_message.count(first_char))
print(last_seven_chars, "count:", strange_message.count(last_seven_chars))

# Suchen und weitersuchen
print("find message:", strange_message.find("message"))
print("find next message:", strange_message.find("message", 3))

# (alle) ersetzen
print("replace by info:", strange_message.replace("message", "info"))
```

Das führt zu folgenden Ausgaben:

```
mid_chars: containing  / last_seven_chars: message
a count: 5
message count: 2
find message: 2
find next message: 28
replace by info: a info containing only a info
```

String in Liste von Zeichen wandeln

Mitunter möchte man Texte als einzelne Zeichen verarbeiten. Dabei kann ein Aufruf von `list()` behilflich sein:

```
print(list("Text als Liste"))
```

Man erhält folgende Ausgabe:

```
['T', 'e', 'x', 't', ' ', 'a', 'l', 's', ' ', 'L', 'i', 's', 't', 'e']
```

Iteration

Beim Durchlaufen der einzelnen Zeichen eines Strings gibt es verschiedene Varianten. Zunächst kann man indiziert mit einer `for`-Schleife und `len()` in Kombination mit `range()` arbeiten. Das ist jedoch in Python der am wenigsten adäquate Weg. Besser ist es mit `enumerate()` zu arbeiten, was sowohl Zugriff auf den Index als auch den Wert bietet. Teilweise benötigt man gar keinen Zugriff auf den Index, dann empfiehlt sich die dritte Variante mit `in`:

```
message = "Python has several loop variants"
for i in range(len(message)):
    print(i, message[i], end=',')

for i, current_char in enumerate(message):
    print(i, current_char, end=',')

for current_char in message:
    print(current_char, end=',')
```

Formatierte Ausgabe

Die folgenden Aufrufe von `capitalize()` und `title()`

```
text = "this is a very special string"
print(text.capitalize())
print(text.title())
```

ergeben diese Ausgaben:

```
This is a very special string
This Is A Very Special String
```

Auch beim Aufbereiten einer Ausgabe mit Platzhaltern bietet Python verschiedene Formen. Im einfachsten Fall gibt man die Werte kommasepariert in `print()` an. Alternativ kann man Platzhalter per `{}` im Text angeben, die dann mit den Werten des Aufrufs von `format()` befüllt werden. Zudem gibt es noch die Variante mit Platzhaltern wie `%s` und `%d` sowie dem Modulo-Operator in Kombination mit einem Tupel, das die Werte bereitstellt. Schließlich lässt sich noch ein explizit formatierter String mit `f"text"` und benannten Parametern nutzen:

```
product = "Apple iMac"
price = 3699

# Varianten der formatierten Ausgabe
print("the", product, "costs", price)
print("the {} costs {}".format(product, price))
print(f"the %s costs %d" % (product, price))
print(f"the {product} costs {price}")
```

Das führt zu folgenden Ausgaben:

```
the Apple iMac costs 3699
the Apple iMac costs 3699
the Apple iMac costs 3699
the Apple iMac costs 3699
```

Zeichenverarbeitung

Manchmal muss man einzelne Zeichen verarbeiten, dann können die Funktionen `ord()` und `chr()` nützlich sein. Dabei konvertiert `chr()` einen `int`-Wert in eine Zeichenfolge der Länge 1 und `ord()` eine solche Zeichenfolge in einen `int`-Wert:

```
>>> ord("A")
65
>>> chr(65)
'A'
>>> ord("0")
48
>>> chr(48)
'0'
```

Beispiel: Stringverarbeitung

Als abschließendes Beispiel zur Stringverarbeitung wollen wir jeweils die Anzahl der Vorkommen von Buchstaben in einem String zählen und dabei Klein- und Großbuchstaben gleich behandeln. Für den Text »Otto« erwarten wir durch die Umwandlung in Kleinbuchstaben 2 x t und 2 x o. Eine solche Aufbereitung wird auch ***Histogramm*** genannt. Darunter versteht man eine Darstellung der Verteilung von Objekten, oftmals sind das numerische Werte. Man kennt es etwa aus der Fotografie für die Helligkeitsverteilung eines Bildes. Nachfolgend geht es um die Verteilung bzw. Ermittlung der Häufigkeiten von Buchstaben für einen Text. Dazu wandeln wir die Eingabe zunächst mit `lower()` in Kleinbuchstaben und durchlaufen diese Zeichenfolge. Durch Aufruf von `isalpha()` stellen wir sicher, dass wir nur Buchstaben in unsere Zählung aufnehmen:

```
from operator import itemgetter


def generate_character_histogram(word):
    char_count_map = {}

    for current_char in list(word.lower()):
        if current_char.isalpha():
            if current_char in char_count_map:
                char_count_map[current_char] += 1
            else:
                char_count_map[current_char] = 1

    return dict(sorted(char_count_map.items(), key=itemgetter(0)))
```

Probieren wir das Ganze einmal in der Python-Kommandozeile aus:

```
>>> generate_character_histogram("Otto")
[('o', 2), ('t', 2)]

>>> generate_character_histogram("Hallo Micha")
[('a', 2), ('c', 1), ('h', 2), ('i', 1), ('l', 2), ('m', 1), ('o', 1)]

>>> generate_character_histogram("Python Challenge, Ihr Python-Training")
[('a', 2), ('c', 1), ('e', 2), ('g', 2), ('h', 4), ('i', 3), ('l', 2), ('n', 5),
      ('o', 2), ('p', 2), ('r', 2), ('t', 3), ('y', 2)]
```

4.2 Aufgaben

4.2.1 Aufgabe 1: Zahlenumwandlungen (★★☆☆☆)

Implementieren Sie basierend auf einem String eine Prüfung auf eine Binärzahl, eine Umwandlung in eine solche sowie beides auch für Hexadezimalzahlen.

Hinweis Die Umwandlung lässt sich mit `int(value, radix)` und der Basis 2 für Binärzahlen sowie der Basis 16 für Hexadezimalzahlen lösen. Nutzen Sie diese explizit nicht, sondern programmieren Sie dies selbst.

Beispiele

Eingabe	Funktion	Resultat
"10101"	is_binary_number()	True
"111"	binary_to_decimal()	7
"AB "	hex_to_decimal()	171

Aufgabe 1a (★☆☆☆☆)

Schreiben Sie eine Funktion `is_binary_number(number)`, die prüft, ob ein gegebener String nur aus den Zeichen 0 und 1 besteht, also eine Binärzahl repräsentiert.

Aufgabe 1b (★★☆☆☆)

Schreiben Sie eine Funktion `binary_to_decimal(number)`, die eine als String dargestellte (gültige) Binärzahl in die korrespondierende Dezimalzahl umwandelt.

Aufgabe 1c (★★☆☆☆)

Schreiben Sie das Ganze erneut, aber diesmal für Hexadezimalzahlen.

4.2.2 Aufgabe 2: Joiner (★☆☆☆☆)

Schreiben Sie eine Funktion `join(values, delimiter)`, die eine Liste von Strings mit der übergebenen Trennzeichenfolge verbindet und als einen String zurückliefert. Implementieren Sie dies ohne Verwendung spezieller Python-Funktionalität selbst.

Beispiel

Eingabe	Separator	Resultat
["hello", "world", "message"]	" +++ "	"hello +++ world +++ message"

4.2.3 Aufgabe 3: Reverse String (★★☆☆☆)

Schreiben Sie eine Funktion `reverse(text)`, die die Buchstaben in einem String umdreht und als Ergebnis liefert. Realisieren Sie dies selbst, also ohne Verwendung spezieller Python-Funktionalität, etwa `[::-1]`.

Beispiele

Eingabe	Resultat
"ABCD"	"DCBA"
"OTTO"	"OTTO"
"PETER	"RETEP"

4.2.4 Aufgabe 4: Palindrom (★★★☆☆)

Aufgabe 4a (★★☆☆☆)

Schreiben Sie eine Funktion `is_palindrome(input)`, die überprüft, ob ein gegebener String unabhängig von Groß- und Kleinschreibung ein Palindrom ist. Erinnern wir uns: Ein Palindrom ist ein Wort, das sich von vorne und von hinten gleich liest.

Beispiele

Eingabe	Resultat
"Otto"	True
"ABCBX"	False
"ABCXcba"	True

Aufgabe 4b (★★★☆☆)

Schreiben Sie eine Erweiterung, die auch Leerzeichen und Satzzeichen nicht als maßgeblich ansieht, wodurch sich dann ganze Sätze prüfen lassen, etwa dieser:

```
Dreh mal am Herd.
```

4.2.5 Aufgabe 5: No Duplicate Chars (★★★☆☆)

Finden Sie heraus, ob ein gegebener String keine doppelten Buchstaben enthält. Dabei sollen Groß- und Kleinbuchstaben keinen Unterschied machen. Schreiben Sie dazu eine Funktion `check_no_duplicate_chars(input)`.

Beispiele

Eingabe	Resultat
"Otto"	False
"Adrian"	False
"Micha"	True
"ABCDEFG"	True

4.2.6 Aufgabe 6: Doppelte Buchstaben entfernen (★★★☆☆)

Schreiben Sie eine Funktion `remove_duplicates(input)`, die in einem gegebenen Text jeden Buchstaben nur einmal behält, also alle späteren doppelten unabhängig von Groß- und Kleinschreibung löscht. Dabei soll aber die ursprüngliche Reihenfolge der Buchstaben beibehalten werden.

Beispiele

Eingabe	Resultat
"bananas"	"bans"
"lalalamama"	"lam"
"MICHAEL	"MICHAEL"

4.2.7 Aufgabe 7: Capitalize (★★☆☆☆)

Aufgabe 7a (★★☆☆☆)

Schreiben Sie eine Funktion `capitalize(input)`, die einen gegebenen Text in das Format eines englischen Titels überführt, bei dem jedes Wort mit einem Großbuchstaben beginnt. Dabei dürfen Sie explizit nicht die eingebaute Funktion `title()` der Strings nutzen.

Beispiele

Eingabe	Resultat
"this is a very special title"	"This Is A Very Special Title"
"effective java is great"	"Effective Java Is Great"

Aufgabe 7b: Abwandlung (★★☆☆☆)

Nehmen wir an, als Eingabe diene nun eine Liste von Strings und es soll ein Liste von Strings zurückgegeben werden, wobei die einzelnen Wörter dann mit einem Großbuchstaben anfangen. Beginnen Sie mit folgender Signatur:

```
capitalize\_words(words)
```

Aufgabe 7c: Sonderbehandlung (★★☆☆☆)

In Überschriften findet man in der Regel eine Sonderbehandlung von Wörtern wie »is« oder »a«, die nicht großgeschrieben werden. Implementieren Sie dies in einer Methode `capitalize_special_2(words, ignorable_words)`, die als zweiten Parameter die Wörter erhält, die von der Umwandlung auszuschließen sind.

Beispiele

Eingabe	Ausnahmen	Resultat
["this", "is", "a", "title"]	["is", "a"]	["This", "is", "a", "Title"]
["java", "is", "great")	["is", "a"]	["Java", "is", "Great"]

4.2.8 Aufgabe 8: Rotation (★★☆☆☆)

Gegeben seien zwei Strings `str1` und `str2`, wobei der erste String länger als der zweite sein soll. Finden Sie heraus, ob der erste den anderen enthält. Dabei dürfen die Buchstaben innerhalb des ersten Strings auch rotiert werden: Vom Anfang oder Ende können Zeichen an die jeweils gegenüberliegende Position verschoben werden (auch wiederholt). Erstellen Sie dazu eine Funktion `contains_rotation(str1, str2)`, die Groß- und Kleinschreibung bei der Prüfung außer Acht lässt.

Beispiele

Eingabe 1	Eingabe 2	Resultat
"ABCD"	"ABC"	True
"ABCDEF	"EFAB"	True ("ABCDEF" $\hookleftarrow$ x 2 $\Rightarrow$ "CDEFAB" enthält "EFAB")
"BCDE"	"EC"	False
"Challenge"	"GECH"	True

4.2.9 Aufgabe 9: Wohlgeformte Klammern (★★☆☆☆)

Schreiben Sie eine Funktion `check_braces(input)`, die prüft, ob die als String übergebene Folge von runden Klammern jeweils passende (sauber geschachtelte) Klammerpaare enthält.

Beispiele

Eingabe	Resultat	Kommentar
"(())"	True	
"()()"	True	
"(()))(((())"	False	zwar gleich viele öffnende und schließende Klammern, aber nicht sauber geschachtelt
"((()"	False	keine passende Klammerung

4.2.10 Aufgabe 10: Anagramm (★★☆☆☆)

Als Anagramm bezeichnet man zwei Strings, die dieselben Buchstaben in der gleichen Häufigkeit enthalten. Dabei soll Groß- und Kleinschreibung keinen Unterschied machen. Schreiben Sie eine Funktion `is_anagram(str1, str2)`.

Beispiele

Eingabe 1	Eingabe 2	Resultat
"Otto"	"Toto"	True
"Mary"	"Army"	True
"Ananas"	"Bananas"	False

4.2.11 Aufgabe 11: 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	. - -

Beispiele

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

Bonus Experimentieren Sie ein wenig und ermitteln Sie für alle Buchstaben des Alphabets die korrespondierenden Morsezeichen, etwa, um Ihren Namen umzuwandeln. Dazu finden Sie unter `https://de.wikipedia.org/wiki/Morsezeichen` die notwendigen Hinweise.

4.2.12 Aufgabe 12: Pattern Checker (★★☆☆☆)

Schreiben Sie eine Funktion `matches_pattern(pattern, text)`, die einen mit Leerzeichen separierten String (2. Parameter) auf die Struktur eines Musters hin untersucht, das in Form einzelner Zeichen als erster Parameter übergeben wird.

Beispiele

Eingabe Muster	Eingabe Wörter	Resultat
"xyyx"	"tim mike mike tim"	True
"xyyx"	"tim mike tom tim"	False
"xyxx"	"tim mike mike tim"	False
"xxxx"	"tim tim tim tim"	True

4.2.13 Aufgabe 13: Tennis-Punktestand (★★★☆☆)

Schreiben Sie eine Funktion `tennis_score(score, player1_name, player2_name)`, die auf Basis eines textuell vorliegenden Punktestands für zwei Spieler PL1 und PL2 im Format <PL1-Punkte>:<PL2-Punkte> eine Ansage im bekannten Stil wie »Fifteen Love«, »Deuce« oder »Advantage Player X« macht.

Dabei gelten für ein Spiel im Tennis folgende Zählregeln:

- Ein Spiel ist gewonnen (»Game <PlayerX>«), wenn ein Spieler 4 oder mehr Punkte erreicht und dazu einen Vorsprung von mindestens 2 Punkten hat.
- Punkte von 0 bis 3 werden mit den Bezeichnungen »Love», »Fifteen«, »Thirty« und »Forty« benannt.
- Bei mindestens 3 Punkten und Gleichstand heißt es »Deuce«.
- Bei mindestens 3 Punkten und einem Punkt Unterschied heißt es »Advantage <PlayerX>«, für denjenigen der einen Punkt mehr hat.

Beispiele

Eingabe	Punktestand
"1:0", "Micha", "Tim"	"Fifteen Love"
"2:2", "Micha", "Tim"	"Thirty Thirty"
"2:3", "Micha", "Tim"	"Thirty Forty"
"3:3", "Micha", "Tim"	"Deuce"
"4:3", "Micha", "Tim"	"Advantage Micha"
"4:4", "Micha", "Tim"	"Deuce"
"5:4", "Micha", "Tim"	"Advantage Micha"
"6:4", "Micha", "Tim"	"Game Micha"

4.2.14 Aufgabe 14: Versionsnummern (★★☆☆☆)

Schreiben Sie eine Funktion `compare_versions(version1, version2)`, die es ermöglicht, Versionsnummern im Format $MAJOR.MINOR.PATCH$ miteinander zu vergleichen – dabei ist die Angabe von $PATCH$ optional. Das Ergebnis soll in Form der Zeichen <, = und > dargestellt werden.

Beispiele

Version 1	Version 2	Resultat
1.11.17	2.3.5	<
2.1	2.1.3	<
2.3.5	2.4	<
3.3	3.2.9	>
7.2.71	7.2.71	=

4.2.15 Aufgabe 15: Konvertierung `str_to_number` (★★☆☆☆)

Wandeln Sie einen String in eine Ganzzahl. Schreiben Sie dazu eine Funktion `str_to_number(input)`.

Hinweis Die Umwandlung lässt sich mit `int(value)` problemlos lösen. Nutzen Sie diese explizit nicht, sondern implementieren Sie das Ganze selbst.

Beispiele

Eingabe	Resultat
"+123"	123
"-123"	-123
"7271"	7271
"ABC"	`ValueError`
"0123"	83 (für Bonusaufgabe)
"-0123"	-83 (für Bonusaufgabe)
"0128"	`ValueError` (für Bonusaufgabe)

Bonus Ermöglichen Sie das Parsing von Oktalzahlen.

4.2.16 Aufgabe 16: Print Tower (★★★☆☆)

Schreiben Sie eine Funktion `print_tower(n)`, die als ASCII-Grafik einen Stab der Höhe $n + 1$ und n aufeinander gestapelte Scheiben, symbolisiert durch das Zeichen `#`, darstellt sowie eine untere Begrenzungslinie zeichnet.

Beispiel Ein Turm der Höhe 3 sollte in etwa wie folgt aussehen:

```
    |
   #|#
  ##|##
 ###|###
---------
```

4.2.17 Aufgabe 17: Gefüllter Rahmen (★★☆☆☆)

Schreiben Sie eine Funktion `print_box(width, height, fillchar)`, die als ASCII-Grafik ein Rechteck der angegebenen Größe zeichnet und mit dem übergebenen Füllzeichen ausfüllt.

Beispiele Nachfolgend sehen wir zwei unterschiedlich gefüllte Rechtecke:

```
+-----+      +-------+
|*****|      |$$$$$$$|
|*****|      |$$$$$$$|
|*****|      |$$$$$$$|
+-----+      |$$$$$$$|
             |$$$$$$$|
             +-------+
```

4.2.18 Aufgabe 18: Vokale raten (★★☆☆☆)

Schreiben Sie eine Funktion `translate_vowel(text, replacement)`, die in einem gegebenen Text alle Vokale durch ein Zeichen bzw. eine Zeichenfolge ersetzt. Das kann man etwa für ein kleines Ratequiz nutzen oder aber um Wortähnlichkeiten nur basierend auf den Konsonanten zu ermitteln.

Eingabe	Ersatzzeichen	Resultat
"Reiseführer"	"?"	"R??s?f?hr?r"
"Rasenmäher"	"-"	"R-s-nm-h-r"
"Ratespiel"	"_"	"R_t_sp__l"
"Rasenmäher"	""	"Rsnmhr"

4.3 Lösungen

4.3.1 Lösung 1: Zahlenumwandlungen (★★☆☆☆)

Implementieren Sie basierend auf einem String eine Prüfung auf eine Binärzahl, eine Umwandlung in eine solche sowie beides auch für Hexadezimalzahlen.

Hinweis Die Umwandlung lässt sich mit `int(value, radix)` und der Basis 2 für Binärzahlen sowie der Basis 16 für Hexadezimalzahlen lösen. Nutzen Sie diese explizit nicht, sondern programmieren Sie dies selbst.

Beispiele

Eingabe	Funktion	Resultat
"10101"	is_binary_number()	True
"111"	binary_to_decimal()	7
"AB "	hex_to_decimal()	171

Lösung 1a (★☆☆☆☆)

Schreiben Sie eine Funktion `is_binary_number(number)`, die prüft, ob ein gegebener String nur aus den Zeichen 0 und 1 besteht, also eine Binärzahl repräsentiert.

Algorithmus Eine Brute-Force- und indexbasierte Variante durchläuft den String zeichenweise vom Anfang bis zum Ende und prüft dabei, ob das aktuelle Zeichen 0 oder 1 ist. Wird ein anderes Zeichen erkannt, so bricht die Schleife ab und die Rückgabe ist `False`:

```
def is_binary_number(number):
    is_binary = True
    i = 0

    while i < len(number) and is_binary:
        current_char = number[i]
        is_binary = (current_char == "0" or current_char == "1")
        i += 1

    return is_binary
```

Das kann man auch als Suchproblem formulieren, muss hier jedoch bei der Rückgabe etwas nachdenken:

```
def is_binary_number_v2(number):
    i = 0
    while i < len(number) and number[i] in ["0", "1"]:
        i += 1

    return i >= len(number)
```

Python-Shortcut Das Ganze kann man mit Python-Spezifika noch etwas einfacher und verständlicher wie folgt realisieren:

```
def is_binary_number_short_cut(word):
    for current_char in word:
        if current_char not in ["0", "1"]:
            return False

    return True
```

Tipp: Python-Stil »Don't ask for permission, ask for forgiveness«

Es gibt – wie in der Aufgabenstellung schon angedeutet – noch die Möglichkeit, `int()` zu nutzen. Dann folgt man dem Python-Motto »Don't ask for permission, ask for forgiveness«. In diesem Fall heißt es, potenziell gefährliche Aktionen auszuprobieren, wie Indexzugriffe mit falschem Index, und bei Fehlschlag geeignet zu reagieren. Diese Gewohnheit sehe ich mit einem starken Java-Background doch eher kritisch – sicherlich ist das Vorgehen oftmals praktisch, manchmal aber doch ein wenig riskant. Doch schauen uns diese stilistisch vollkommen adäquate Variante der Prüfung an:

```
def is_binary_number_v3(number):
    try:
        int(number, 2)
        return True
    except ValueError:
        return False
```

Lösung 1b (★★☆☆☆)

Schreiben Sie eine Funktion `binary_to_decimal(number)`, die eine als String dargestellte (gültige) Binärzahl in die korrespondierende Dezimalzahl umwandelt.

Algorithmus Man durchläuft den String zeichenweise von links nach rechts und verarbeitet jedes Zeichen als binäre Ziffer. Dabei errechnet sich der Wert auf Basis des aktuellen Zeichens, indem man den zuvor umgewandelten Wert mit 2 multipliziert und den aktuellen Wert addiert. Der Algorithmus lässt sich klar und ohne Sonderbehandlungen formulieren, weil durch Aufruf der zuvor implementierten Funktion `is_binary_number(number)` eine gültige Eingabe sichergestellt ist:

```
def binary_to_decimal(number):
    if not is_binary_number(number):
        raise ValueError(number + " is not a binary number")

    decimal_value = 0
    for i, current_char in enumerate(number):
        value = int(current_char)
        decimal_value = decimal_value * 2 + value

    return decimal_value
```

Lösung 1c (★★☆☆☆)

Schreiben Sie das Ganze erneut, aber diesmal für Hexadezimalzahlen.

Algorithmus Für Hexadezimalzahlen muss der Faktor auf 16 geändert werden. Zudem sind nun die Buchstaben A bis F sowohl klein- als auch großgeschrieben erlaubt. Deren Wert wird durch eine Subtraktion `ord(current_char) - ord("A") + 10` ermittelt – dadurch bildet man 'A' bis 'F' auf die Werte 0 bis 5 ab und addiert 10, was dann den korrekten Wert ergibt:

```
def hex_to_decimal(number):
    if not is_hex_number(number):
        raise ValueError(number + " is not a hex number")

    decimal_value = 0
    for i, current_char in enumerate(number):

        if current_char.isdigit():
            value = int(current_char)
        else:
            value = ord(current_char.upper()) - ord("A") + 10

        decimal_value = decimal_value * 16 + value

    return decimal_value
```

Die Prüfung auf gültige Hexadezimalzahlen nutzt eine trickreiche Prüfung mit `in` unter der Angabe aller möglichen Ziffern und Buchstaben für Hexadezimalzahlen:

```
def is_hex_number(number):
    for current_char in number:
        if current_char not in "0123456789ABCDEFabcdef":
            return False

    return True
```

Prüfung

Zum Test nutzen wir folgende Eingaben, die die korrekte Funktionsweise zeigen:

```
@pytest.mark.parametrize("value, expected",
                         [("10101", True), ("222", False), ("12345", False)])
def test_is_binary_number(value, expected):
    assert is_binary_number(value) == expected


@pytest.mark.parametrize("value, expected",
                         [("111", 7), ("1010", 10), ("1111", 15), ("10000", 16)])
def test_binary_to_decimal(value, expected):
    assert binary_to_decimal(value) == expected


@pytest.mark.parametrize("value, expected",
                         [("7", 7), ("A", 10), ("F", 15), ("10", 16)])
def test_hex_to_decimal(value, expected):
    assert hex_to_decimal(value) == expected
```

4.3.2 Lösung 2: Joiner (★☆☆☆☆)

Schreiben Sie eine Funktion `join(values, delimiter)`, die eine Liste von Strings mit der übergebenen Trennzeichenfolge verbindet und als einen String zurückliefert. Implementieren Sie dies ohne Verwendung spezieller Python-Funktionalität selbst.

Beispiel

Eingabe	Separator	Resultat
["hello", "world", "message"]	" +++ "	"hello +++ world +++ message"

Algorithmus Durchlaufe die Liste der Werte von vorne nach hinten. Füge jeweils den Text in einen String ein, füge dann die Trennzeichenfolge hinzu und wiederhole dies bis zum letzten Wert. Als Spezialbehandlung darf nach diesem keine Trennzeichenfolge mehr hinzugefügt werden:

```
def join(values, delimiter):
    result = ""
    for i, current_value in enumerate(values):
        result += current_value
        # Kein Trenner nach letztem Vorkommen
        if i < len(values) - 1:
            result += delimiter

    return result
```

Python-Shortcut Das Zusammenfügen von Strings lässt sich mit der geeigneten Funktion `join()` schön kompakt, verständlich und ohne Spezialbehandlung schreiben:

```
result = delimiter.join(values)
```

Ein Variante mit `reduce()` sieht wie folgt aus:

```
result = reduce(lambda str1, str2: str1 + delimiter + str2, values)
```

Übrigens: Die Funktion `join()` ist auch praktisch, wenn man die Werte einer Liste in einen String wandeln möchte. Dazu verwendet man einen Leerstring als Delimiter:

```
"".join(values)     # Trick: Liste in String wandeln
```

Prüfung

Zum Test nutzen wir folgende Eingaben, die die korrekte Funktionsweise zeigen:

```
@pytest.mark.parametrize("values, delimiter, expected",
                         [(["hello", "world", "message"], " +++ ",
                           "hello +++ world +++ message")])
def test_join(values, delimiter, expected):
    assert join(values, delimiter) == expected
```