3 Strings

Wir haben schon ein paar Mal Strings verwendet, ohne viel darüber nachzudenken. Als kurze Erinnerung: Variablen vom Typ `str` modellieren Zeichenketten und dienen zur Verwaltung von textuellen Informationen. In diesem Kapitel schauen wir uns die Thematik genauer an und beginnen unsere Entdeckungsreise zu Zeichenketten.

3.1 Schnelleinstieg

Strings bestehen aus einer Folge einzelner Zeichen und bilden wie Listen sequenzielle Datentypen (vgl. Abschnitt 7.1), weshalb viele Aktionen analog dazu ausführbar sind. 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'
```

3.1.1 Gebräuchliche Stringaktionen

Stringkonkatenation

Ein sehr gebräuchlicher Anwendungsfall ist das Zusammenfügen von Strings, auch Konkatenieren genannt. Dazu dient bekanntlich der Operator '+'. Nachfolgend kombinieren wir den Vor- und Nachnamen des Autors mit einem Abstand von einem Leerzeichen:

```
>>> first_name = "Michael"
>>> last_name = "Inden"
>>> print(first_name + " " + last_name)
Michael Inden
```

Besonderheiten Beim Erstellen von Texten aus einzelnen Teilbausteinen ist der Operator '+' ziemlich nützlich. Allerdings sollte man dabei ein paar Besonderheiten kennen. Im Gegensatz zu vielen anderen Sprachen, die bei der Stringkonkatenation auch die Angabe und Verknüpfung von Zahlen erlauben, ist das in Python so nicht möglich:

```
>>> "Bitte " + 2 + " Mal klingeln"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Stattdessen benötigt es eine Typumwandlung (einen Cast) mit einem Aufruf von `str` wie folgt:

```
>>> "Bitte " + str(2) + " Mal klingeln"
'Bitte 2 Mal klingeln'
```

Groß- und Kleinschreibung

In der Praxis benötigt man immer mal wieder eine Umwandlung von Groß- in Kleinbuchstaben oder andersherum. Dabei helfen die beiden Methoden `upper()` und `lower()`.

Zuerst wandeln wir einen Text vollständig in Großbuchstaben um und danach dann in Kleinbuchstaben.

```
>>> message = "IMPORTANT: Please consult the doctor"
>>> message.upper()
'IMPORTANT: PLEASE CONSULT THE DOCTOR'
>>> message.lower()
'important: please consult the doctor'
```

Besonderheiten: Keine Modifikation erlaubt Bitte beachten Sie, dass Strings in Python als unveränderliche Objekte realisiert sind. Eine Modifikation ist daher nicht möglich. Als Abhilfe kann man einen neuen String mit verändertem Inhalt erzeugen:

```
>>> hint = "Immutable String"
>>> result = hint.upper()
>>> result
'IMMUTABLE STRING'
>>> hint
'Immutable String'
```

Tipp: Was sind Methoden?

Bislang haben wir Funktionen zum Bereitstellen von Funktionalität kennengelernt. Methoden sind sehr ähnlich zu Funktionen, jedoch sind Methoden einem bestimmten Typ, hier `str`, zugeordnet. Das wird später noch in Kapitel 4 genauer besprochen.

Leerzeichen entfernen

Mitunter sollen am Anfang oder Ende oder an beiden Seiten unerwünschte Leerzeichen entfernt werden. Dazu kann man die Methoden `strip()` sowie `lstrip()` und `rstrip()` aufrufen. Beginnen wir mit der beidseitigen Entfernung:

```
>>> value_with_blanks = "  This text has blanks at the beginning and the end  "
>>> print("strip():  '", value_with_blanks.strip(), "'", sep="")
strip():  'This text has blanks at the beginning and the end'
```

Nachfolgend werden noch vorne bzw. hinten die Leerzeichen entfernt:

```
>>> print("lstrip(): '",value_with_blanks.lstrip(), "'", sep="")
lstrip(): 'This text has blanks at the beginning and the end  '
>>> print("rstrip(): '",value_with_blanks.rstrip(), "'", sep="")
rstrip(): '  This text has blanks at the beginning and the end'
>>>
>>> value_with_blanks
'  This text has blanks at the beginning and the end  '
```

Beachten Sie bitte auch hier, dass durch die Funktionsaufrufe der ursprüngliche String nicht verändert wird, sondern ein neuer String als Ergebnis entsteht. Dies folgt aus den Ausgaben, insbesondere der letzten, die einen unveränderten Inhalt der Variablen `value_with_blanks` zeigt.

Länge ermitteln

Auch die Gesamtlänge eines Strings ist des Öfteren eine wichtige Information. Die Gesamtzahl der Zeichen liefert ein Aufruf von `len()`:

```
>>> content = "This is a short message"
>>> len(content)
23
```

Auf leeren String prüfen

Zwar kann man mithilfe der zuvor vorgestellten Methode auch prüfen, ob ein String leer ist:

```
>>> no_content = ""
>>> len(no_content)  == 0
True
```

Das ist jedoch nicht so schön lesbar und wird eher als schlechter Stil angesehen. In Python prüft man gerne auf folgende Weise:

```
>>> if not no_content:
...     print("empty")
...
empty
```

Auf einzelne Zeichen zugreifen

Auf die einzelnen Buchstaben eines Strings kann man positionsbasiert mit eckigen Klammern zugreifen, wobei der erste Buchstabe den Index 0 besitzt. Nachfolgend lesen wir die ersten drei Zeichen des Strings »`This is a short message`« aus:

```
>>> content = "This is a short message"
>>> content[0]
'T'
>>> content[1]
'h'
>>> content[2]
'i'
```

Keine Modifikation erlaubt Bitte beachten Sie, dass keine korrespondierenden Zuweisungen erlaubt sind, um Zeichen in einem String zu modifizieren:

```
>>> content[2] = "X"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Wie schon erwähnt, sind Strings in Python als unveränderliche Objekte realisiert. Eine Möglichkeit, doch zeichenbasiert ändern zu können, zeige ich Ihnen später.

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=',')
```

Teilbereiche extrahieren – Slicing

Für einige Anwendungsfälle muss man auf Bestandteile eines Texts zugreifen, die durch eine Anfangs- und optional eine Endposition bestimmt sind. Dazu bietet sich das sogenannte Slicing an. Dabei übergibt man entweder Start- (inklusiv) und Endposition (exklusiv) oder aber nur den Start, wodurch dann der Text ab dieser Position bis zum Ende als neuer String geliefert wird.

Betrachten wir die mächtigen Slicing-Operationen anhand eines Beispiels, um einzelne Zeichen, ganze Bestandteile oder sogar nicht zusammenhängende Bereiche extrahieren zu können. Im Beispiel sehen wir zudem die Methode `count()`. Diese zählt, wie oft die übergebene Zeichenkette in einem String vorkommt:

```
>>> 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)
mid_chars: containing / last_seven_chars: message
>>>
>>> first_char = strange_message[0]
>>> print(first_char, "count:", strange_message.count(first_char))
a count: 5
>>> print(last_seven_chars, "count:", strange_message.count(last_seven_chars))
message count: 2
```

Wissenswert ist noch folgende Besonderheit mit einer negativen Schrittweite. Das liefert einen String von hinten gelesen, also in umgekehrter Reihenfolge:

```
>>> teile = "Dies ist ein String. Rest ABC"
>>> print(teile[::-1])
CBA tseR .gnirtS nie tsi seiD
>>> print(teile[19::-1])
.gnirtS nie tsi seiD
```

Strings wiederholen

Manchmal ist es notwendig, einen Text n-mal zu wiederholen. Das ist zwar problemlos mit einer `for`-Schleife möglich, jedoch existiert die recht praktische Multiplikation auch für Strings, womit die Aufgabe erleichtert wird:

```
>>> greeting = "MOIN"
>>> greeting * 2
'MOINMOIN'
>>> nonsens = "BLA"
>>> nonsens * 3
'BLABLABLA'
```

3.1.2 Suchen und Ersetzen

Suchen und Enthaltensein

Ab und an möchte man feststellen, ob ein gewisser Text oder Buchstabe in einem String enthalten ist. Mithilfe der Methoden `index()` und `find()` erhält man die entsprechende Position bzw. den Wert -1 für »nicht gefunden«. Eine Prüfung vom Ende des Strings ermöglicht die Methode `rindex()`. Soll lediglich geschaut werden, ob der gesuchte Teilstring enthalten ist, bietet sich der `in`-Operator an:

```
>>> maintext = "This is a secret message. Please do not distribute"
>>> maintext.index("This")
0
>>> maintext.index("Please")
26
>>> maintext.index("o")
34
>>> maintext.rindex("o")
37
>>> maintext.find("not")
36
>>> "not" in maintext
True
>>> "MICHAEL" in maintext
False
```

Weitersuchen

Auf eine praktische Besonderheit möchte ich im Zusammenhang mit `index()` und `find()` noch eingehen. Zunächst einmal sei angemerkt, dass die Methoden `index()` und `find()` nach dem ersten Vorkommen suchen. Auch wiederholtes Aufrufen ändert die Fundstelle nicht und man kann so keine anderen Vorkommen finden. Praktischerweise gibt es aber eine Variante der Methode, der man eine Startposition übergeben kann. Auf diese Weise lässt sich problemlos die Funktionalität »Suchen und Weitersuchen« realisieren, indem man immer nach der Fundstelle weitersucht. Dazu übergibt man einfach die gelieferte Position + 1 wie folgt:

```
>>> info = "one second, one hour and one day"
>>> info.index("one")
0
>>> info.index("one", 1)
12
>>> info.index("one", 13)
25
>>> info.find("one")
0
>>> info.find("one", 1)
12
>>> info.find("one", 13)
25
```

Ersetzen von Inhalten

Neben dem Suchen und dem Test auf Enthaltensein möchte man manchmal auch Teile eines Strings ersetzen. Dabei ist die Methode `replace()` hilfreich. Diese ersetzt eine gewünschte Zeichenfolge durch die als zweiten Parameter übergebene Zeichenfolge:

```
>>> greeting = "MOIN MOIN"
>>> greeting.replace("MOIN", "GRÜEZI")
'GRÜEZI GRÜEZI'
```

Weil Strings unveränderlich sind, wird auch hier als Ergebnis wieder ein neuer String mit dem veränderten Inhalt erzeugt.

Komplexeres Ersetzen von Inhalten

Ergänzend gibt es die Funktion `sub()` aus dem Modul `re`, die als Suchzeichenfolge einen sogenannten regulären Ausdruck nutzt – um beispielsweise Texte, die mit einem A starten, gefolgt von einem beliebigen Buchstaben und einem C oder D, durch einen Leerstring zu ersetzen. Für dieses Buch reicht es, zu wissen, dass der '.' in einem regulären Ausdruck einen Platzhalter für ein einzelnes, beliebiges Zeichen darstellt und dass man eine Menge von Alternativen in eckigen Klammern angeben kann. Weitere Details finden Sie unter `https://docs.python.org/3/library/re.html`.

```
>>> import re
>>> info_regex = "ACC_AEC_MUSIC_ABC_AWARD"
>>> re.sub(r"A.[CD]", "", info_regex)
'__MUSIC__AW'
```

Gibt man als regulären Ausdruck lediglich einen konkreten Text an, so wirkt wieder die exakte Übereinstimmung des Musters – hier um alle Vorkommen von "`ABC`" durch "`--`" zu ersetzen:

```
>>> info_plain = "ABCABC_MUSIC_ABCAWARD"
>>> re.sub(r"ABC", "--", info_plain)
'----_MUSIC_--AWARD'
```

3.1.3 Informationen extrahieren und formatieren

Informationen extrahieren

Immer mal wieder enthält ein Text mehrere Informationsbestandteile, etwa eine Uhrzeitangabe mit Stunden, Minuten und Sekunden jeweils durch '`:`' getrennt:

```
>>> timestamp = "11:22:33"
```

Unser Ziel ist es, die Einzelbestandteile als Zahlen zu ermitteln. Wie kann man dazu vorgehen? Um die Informationen aufzuspalten und auszulesen, bietet sich die Methode `split()` an. Dieser übergibt man ein Trennzeichen und erhält als Ergebnis eine Abfolge (genauer: eine Liste) von Strings – Listen werden wir dann in Kapitel 5 genauer kennenlernen:

```
>>> timestamp.split(":")
['11', '22', '33']
```

Das Extrahieren von Informationen mit `split(":")` scheint einfach. Versuchen wir uns an der Extraktion der Werte einer Datumsangabe. Probieren wir das mit `split()` wie zuvor aus:

```
>>> dateInfo = "23.11.2020"
>>> dateInfo.split(".")
['23', '11', '2020']
```

Formatierte Ausgabe

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 Platzhaltern nutzen:

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

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

Beim Einsatz von `format()` können Sie mit dem Formatbezeichner `:d` als Argument einen beliebigen ganzzahligen Typ verwenden. Es gibt eine Vielzahl weiterer Platzhalter. Nachfolgend sind einige für Strings und Zahlen gezeigt. Die gesamte Liste ist so ausführlich, dass sich ein Blick in die Onlinedokumentation `https://docs.python.org/3/library/string.html?highlight=string#module-string` oder aber ein Klick auf den Link `https://realpython.com/python-f-strings/` lohnt.

Auf einige Besonderheiten möchte ich noch eingehen: Es ist recht einfach möglich, etwa die Länge einer Ganzzahl mit `:<Anzahl>d` zu beschränken sowie für Gleitkommazahlen die Anzahl an Nachkommastellen mit `:.<Anzahl>f`. Nachfolgend sehen wir eine Begrenzung auf 10 Stellen bei einem Gehalt und zwei Nachkommastellen für PI.

```
>>> "{:s}'s salary is {:10d} a year. What about PI? {:.2f}".format("Tom", 123
      _456_789, math.pi)
"Tom's salary is  123456789 a year. What about PI? 3.14"
```

Formatierung mit `capitalize()` und `title()`

Mitunter soll der Anfang des Strings bzw. jedes Worts großgeschrieben werden. Dazu bieten sich die folgenden Aufrufe von `capitalize()` und `title()` an:

```
>>> text = "this is a very special string"
>>> text.capitalize()
'This is a very special string'
>>> text.title()
'This Is A Very Special String'
```

3.1.4 Praxisrelevante Funktionen im Kurzüberblick

Zum Abschluss des Schnelleinstiegs wollen wir noch einmal die wichtigsten und in der Praxis nützlichsten Funktionalitäten rekapitulieren. 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 und somit 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 das Ergebnis als neuen String zurück.
- `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.

- `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 ein Fehler ausgelöst.
- `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. 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.

3.2 Nächste Schritte

3.2.1 Zeichenverarbeitung

Manchmal muss man einzelne Zeichen verarbeiten, dann können die Funktionen `chr()` und `ord()` 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'
```

3.2.2 Strings und Listen

Umwandlung in eine Liste

Mitunter ist es praktisch, den Inhalt eines Strings mit `list()` in eine geordnete Liste von Einzelzeichen vom Typ `str` zu überführen. Listen werden wir in Kapitel 5 genauer besprechen, hier reicht das Verständnis, dass es sich dabei um eine über die Position geordnete Abfolge von Elementen, hier Zeichen, handelt.

Möchte man Texte in einzelne Zeichen wandeln, nutzt man `list()`:

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

Man erhält folgende Ausgabe:

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

Modifikation in einer Liste

Zuvor erwähnte ich, dass es praktisch ist, einen String in eine Liste zu wandeln. Warum? Das gilt immer dann, wenn man an gewissen Stellen den String ändern möchte. Strings erlauben das ja nicht. Wenn wir nun aber beispielsweise jedes dritte Zeichen großschreiben wollen (oder für Sie als kleine Fingerübung durch ein Leerzeichen ersetzen), dann ist dies auf Basis einer Liste möglich. Die gewünschten Modifikationen müssen dazu auf einzelnen Zeichen erfolgen. Danach soll dann die Liste gewöhnlich wieder in einen String gewandelt werden. Das lernen wir im Anschluss kennen.

Schauen wir uns vorab noch die Modifikation innerhalb einer Liste für jedes dritte Zeichen an. Zunächst definieren wir die Ausgangsdaten:

```
>>> alphabet = "abcdefghijklmnopqrstuvwxyz"
```

Nun erstellen wir eine passende Methode:

```
>>> def modify_every_3rd_to_upper(values):
...     for i in range(0, len(values), 3):
...         current = values[i]
...         values[i] = current.upper()
...
```

Wir wandeln unseren String in eine Liste und rufen diese gerade erstellte Methode einmal auf. Als Ergebnis ist jeder 3. Buchstabe großgeschrieben:

```
>>> as_list = list(alphabet)
>>> modify_every_3rd_to_upper(as_list)
>>> as_list
['A', 'b', 'c', 'D', 'e', 'f', 'G', 'h', 'i', 'J', 'k', 'l', 'M', 'n', 'o', 'P',
      'q', 'r', 'S', 't', 'u', 'V', 'w', 'x', 'Y', 'z']
```

Einen String aus einer Liste erzeugen

Gerade haben wir gesehen, wie man aus einem String eine korrespondierende Liste erhält. Manchmal möchte man auch aus einer Liste (wieder) einen String erzeugen. Dazu dient ein Aufruf von `join()` wie folgt:

```
>>> "".join(as_list)
'AbcDefGhiJklMnoPqrStuVwxYz'
```

Als weiteres Beispiel sehen wir einen Gruß in Form einer Liste, der dann in einen `str` gewandelt wird mit jeweils einem Minuszeichen zwischen den Buchstaben:

```
>>> message = [ 'H', 'o', 'i', ' ', 'S', 'o', 'p', 'h', 'i', 'e' ]
>>> "-".join(message)
'H-o-i- -S-o-p-h-i-e'
```

3.2.3 Mehrzeilige Strings

Python unterstützt mehrzeilige Strings. Diese werden durch drei Anführungszeichen eingeleitet und abgeschlossen:

```
>>> multi_line_string = """
...   This is line 1
...   Second line with "quotes"
...   Last line with 'single quotes'
...   """
>>> multi_line_string
'\n  This is line 1\n  Second line with "quotes"\n  Last line with \'single
     quotes\'\n  '
```

Nach der mehrzeiligen Definition kann man wie zuvor gezeigt auf dem String agieren, also dessen Länge bestimmen, Bausteine ersetzen usw. Das liegt daran, dass ein solcher mehrzeiliger String ebenfalls ein normaler String ist und damit alle zuvor beschriebenen Aktionen auch unterstützt werden.

```
>>> type(multi_line_string)
<class 'str'>
>>> len(multi_line_string)
81
>>> multiLineString.replace("line", "LINE")
'\n  This is LINE 1\n  Second LINE with "quotes"\n  Last LINE with \'single
     quotes\'\n  '
```

Besonderheit Platzhalter Außerdem kann man Platzhalter definieren und durch Aufruf von `format()` mit Werten befüllen:

```
>>> placeholders = """
...        Michael {} hat am "{}"
...        {:d} Bücher in '{}' gekauft.
...        """.format("Inden", "20.1.2020", 7, "Bremen")
```

Schauen wir uns an, was durch diese Anweisungen an Ausgaben produziert wird:

```
>>> print(placeholders)

       Michael Inden hat am "20.1.2020"
       7 Bücher in 'Bremen' gekauft.

>>>
```

Beispiel: HTML-Code

Auch wenn man HTML-Code in Python aufbereiten möchte, sind die mehrzeiligen Strings sehr hilfreich. Es ist in der Tat angenehm, dies wie folgt anzugeben:

```
>>> hello_world_html = """
...                       <html>
...                           <body>
...                               <p>Hello World</p>
...                           </body>
...                       </html>
...                       """
>>>
```

Tipp: Behandlung der führenden Leerzeichen

Eine Sache noch zur Ausrichtung bzw. des führenden Leerraums: Während Java die Leerzeichen am Anfang eines mehrzeiligen Strings ignoriert und den gesamten Text an den unteren drei Anführungszeichen ausrichtet, bleiben die Leerzeichen vor den Zeilen in Python erhalten!

3.3 Aufgaben und Lösungen

3.3.1 Aufgabe 1: Länge, Zeichen und Enthaltensein

In dieser ersten Aufgabe geht es darum, grundlegende Funktionalität der Strings anzuwenden. Sie sollen die Länge eines Texts abfragen, dann ein Zeichen an einer beliebigen Position, etwa der 13., ermitteln und schließlich prüfen, ob ein gewünschtes Wort im String enthalten ist.

Lösung

Die geforderten Aktionen lassen sich direkt mit den passenden Funktionen umsetzen, nämlich mit `len()`, `[]` und `in`:

```
>>> nachricht = "Hallo lieber Leser! Viel Spaß mit Python!"
>>> len(nachricht)
41
>>> nachricht[13]
'L'
>>> "Python" in "Hallo lieber Leser! Viel Spaß mit Python!"
True
```

3.3.2 Aufgabe 2: Zeichen wiederholen

Bei dieser Aufgabe sollen die Buchstaben eines Worts gemäß ihrer Position wiederholt werden, aus ABC wird dann ABBCCC. Schreiben Sie dazu eine Funktion `repeat_chars(input)`.

Lösung

Die Grundidee besteht darin, zeichenweise von vorne nach hinten durch den String zu laufen. Dazu nutzen wir eine `for`-Schleife mit `enumerate()`, weil wir damit sowohl den Index als auch das aktuelle Zeichen direkt im Zugriff haben. Strings besitzen die Multiplikation per `*`, die eine n-malige Wiederholung erlaubt. Die damit berechnete Wiederholung eines Buchstabens addieren wir zu unserem Resultat. Die Umsetzung in Python sieht wie folgt aus:

```
def repeat_chars(input):
    result = ""

    for i, ch in enumerate(input):
        result += ch * (i + 1)

    return result
```

Rufen wir dies exemplarisch auf:

```
print(repeat_chars("ABCD"))
print(repeat_chars("ABCDEF"))
```

Dann erhalten wir folgende Ausgabe:

```
ABBCCCDDDD
ABBCCCDDDDEEEEEFFFFFF
```

3.3.3 Aufgabe 3: Vokale raten

Bei dieser Aufgabe werden einige etwas ältere Leser sich vielleicht an die Sendung »Glücksrad« erinnern, bei der es genau um das Erraten von Wörtern, Sätzen oder Redewendungen ging, in denen Vokale fehlten.[1]

Als Erstes sollen in einem gegebenen String mithilfe der selbst geschriebenen Funktion `remove_vowels(input)` alle Vokale entfernt werden. Als Zweites soll eine Funktion `replace_vowels(input)` implementiert werden, die einen Vokal durch ein '_' ersetzt, damit wir für ein kleines Ratespiel einen Hinweis auf einen entfernten Vokal bekommen.

Lösung: Vokale entfernen

Zum Entfernen der Vokale durchlaufen wir die Eingabe zeichenweise. Für die Prüfung auf Vokal nutzen wir folgenden Trick: Wir modellieren die Vokale als String `"AÄEIOÖUüaäeioöuü"` und prüfen dann mit `in`, ob der aktuelle Buchstabe in dieser Zeichenfolge enthalten ist. Falls nein (daher hier die Negation mit `not`), übernehmen wir den Buchstaben in unsere Ergebnisvariable `result`:

```
def remove_vowel(text):
    result = ""

    for letter in text:
        if letter not in "AÄEIOÖUüaäeioöuü":
            result += letter

    return result
```

Zum Nachvollziehen beginnen wir mit der Definition eines beliebigen Texts als Ausgangsbasis und rufen dann die gerade erstellte Funktion auf:

```
text = "Es gibt viel zu entdecken!"
print(remove_vowel(text))
```

Wir erhalten wie erwartet folgendes Ergebnis:

```
s gbt vl z ntdckn!
```

[1] Es fehlten auch Konsonanten. Im Gegensatz zu den Vokalen konnte man diese aber kaufen.

Lösung: Vokale ersetzen

Wie gesagt, wäre es für ein kleines Ratespiel besser, wenn wir einen Hinweis auf einen entfernten Vokal durch ein '_' bekommen würden. Basierend auf der obigen Lösung sollte das Ganze ziemlich direkt möglich sein.

Wir wollen in diesem Buch nicht nur das Programmieren lernen, sondern insbesondere auch die Ideen guten Programmierstils befolgen. Im obigen Beispiel ist die Vokalprüfung in der Schleife »versteckt«. Diese Aufgabe kann man als unabhängige Funktionalität ansehen. Deshalb schreiben wir dafür eine eigene Funktion:

```
def is_vowel(letter):
    return letter in "AÄEIOÖÜüaäeioöuü"
```

Schauen wir doch einmal, wie sich deren Nutzung positiv auf das Verständnis und die Erweiterung auswirkt. Beim Auffinden eines Vokals müssen wir jetzt ein '_' einfügen, ansonsten das ursprüngliche Zeichen. Die `for`-Schleife und die anderen Aktionen sind analog zu zuvor. Schreiben wir ergänzend eine neue Funktion `replace_vowels(input)` – durch die Hilfsfunktion ist der Ablauf deutlicher erkennbar:

```
def replace_vowel(text):
    result = ""

    for letter in text:
        if is_vowel(letter):
            result += '_'
        else:
            result += letter

    return result
```

Mithilfe der folgenden Kurzschreibweise können wir das Ganz noch kompakter schreiben:

```
def replace_vowel(text):
    result = ""

    for letter in text:
        result += '_' if is_vowel(letter) else letter

    return result
```

Überprüfen wir noch, ob das Ganze auch wirklich wie gewünscht funktioniert:

```
print(replace_vowel("Es gibt viel zu entdecken!"))
print(replace_vowel("PYTHON INTRO BY MICHAEL"))
```

Damit erhalten wir folgende Ausgaben:

```
_s g_bt v__l z_ _ntd_ck_n!
PYTH_N _NTR_ BY M_CH__L
```

3.3.4 Aufgabe 4: String Merge

Schreiben Sie eine Funktion `string_merge(input1, input2)`, die die beiden Texte Buchstabe für Buchstabe (»wie ein Reißverschluss«) miteinander verbindet. Wenn der eine Text ACE lautet und der andere BDFGH, dann soll ABCDEFGH als Ergebnis geliefert werden. Es wird also immer abwechselnd ein Buchstabe aus dem ersten und dann einer aus dem zweiten Text genommen. Ist ein Text vollständig verarbeitet, werden alle Buchstaben aus dem verbliebenen anderen Text übernommen.

Lösung

Um die Texte Zeichen für Zeichen miteinander zu verweben, gibt es viele Varianten. Eine ganz besonders ausgefeilte ist die folgende: Wir bestimmen zunächst die Längen der beiden Texte und zudem auch die größere von beiden, die wir in der Variablen `max_length` speichern.

Nun durchlaufen wir mit einer `for`-Schleife die Texte bis zu diesem Maximalwert. Wir können per Indexzugriff ein einzelnes Zeichen extrahieren – vorab prüfen wir aber, ob der aktuelle Index noch im Bereich der Länge des jeweiligen Strings ist:

```
def string_merge(input1, input2):
    length1 = len(input1)
    length2 = len(input2)
    max_length = max(length1, length2)

    result = ""
    for i in range(max_length):
        if i < length1:
            result += input1[i]
        if i < length2:
            result += input2[i]

    return result
```

Probieren wir diese schon etwas komplexere Funktionalität mit ein paar Eingaben aus:

```
print(string_merge("ACE", "BDFGH"))
print(string_merge("ACEGH", "BDF"))
print(string_merge("First Try", "Second Attempt"))
```

Es kommt zu folgenden Ausgaben:

```
ABCDEFGH
ABCDEFGH
FSiercsotn dT rAyttempt
```

Interessant ist auch der Fall, wenn einer von beiden Texten leer ist – das führt uns fast zum Thema Testen und Auswahl zu prüfender Wertebelegungen (siehe Abschnitt 14.3).

```
print(string_merge("", "Not Empty"))
```

Als Ergebnis erhält man:

```
Not Empty
```