Algorithmische Methoden 02: Elementare Datentypen

Dr. Sven Naumann

WS 20/21

Erste Schritte

Wir werden uns heute mit einigen für die Programmierung grundlegenden Fragen auseinandersetzen und uns anschließend mit zwei grundlegenden Datentypen von Python vertraut machen:

- Zahlen und
- Zeichenketten.

Abschließend werden wir uns dann mit *regulären Ausdrücken* (kurz: RE) auseinandersetzen. RE ermöglichen es uns, textuelle Daten effizient zu verarbeiten: So können wir mit ihrer Hilfe z.B. nach Wörtern oder Zeichenfolge suchen, gezielte Ersetzungen vornehmen, etc.

Was ist ein Computerprogramm? Woraus besteht es?

Wenn wir also ein Programm als eine Folge von Anweisungen auffassen, die sequentiell ausgeführt werden, dann können diese Anweisung schriftlich in einer Textdatei gespeichert und bei Bedarf ausgeführt werden.

- Dateien, die *Python* -Programme enthalten, verwenden typischerweise die Extension `.py` .

Wie in anderen Skriptsprachen, ist es möglich, Programme, die in konventionellen Programmiersprachen viele Zeilen Code erfordern, stark zu kondensieren.

Das klassische `Hello-World` -Programm wird so zu einem Einzeiler:

```
>>> print("Hello World!“)
Hello World!
```

Dieses Programm besteht aus einer einzigen Anweisung, die die Funktion `print()` mit einer Zeichenkette als Argument aufruft. Mit Hilfe der `printf()`-Funktion lassen sich einfache Ausgaben gestalten:
Wie man an diesem Beispiel schon sieht, werden in Python die Argumente von Funktionen durch Klammern eingeschlossen.

- Nimmt eine Funktion mehrere Werte, werden sie durch Kommata voneinander getrennt:

```
>>> print("Hello", " World!“)
Hello World!
```

Die Funktion `print()` akzeptiert eine beliebige Zahl von Argumenten.
Das Standardverhalten der Funktion lässt sich folgendermaßen beschreiben:
Die Argumente werden durch ein Leerzeichen voneinander getrennt ausgegeben. Die Ausgabe wird mit einem Zeilenvorschub abgeschlossen und erfolgt normalerweise auf den Bildschirm, kann aber bei Bedarf z.B. in eine Datei oder auf einen Drucker umgeleitet werden.

Eine Folge von ein oder mehreren Anweisungen, die eine inhaltliche Einheit bilden, wird als *Block* bezeichnet. Anders als in JAVA oder JavaScript werden Blöcke nicht durch geschweifte Klammern oder Semikola markiert; stattdessen verwendet *Python* **Einrückungen** :

```
äußerer Block: Anfang
...
    innerer Block: Anfang
    ...
    innerer Block: Ende
...
äußerer Block: Ende
```

Falsche Einrückungen können zu Fehlermeldungen oder nicht-intendiertem Programmverhalten führen.

Eine Anweisung endet normalerweise mit dem Ende einer Zeile. Um lange Anweisungen, die über mehrere Zeilen gehen, formulieren zu können, kann man das Backslash-Zeichen verwenden:

```
......\ # mehrzeilige
... # Anweisung
```

Wenn man umgekehrt mehrere Anweisungen in einer Zeile schreiben möchte, dann müssen sie durch Semikola voneinander getrennt werden:

```
Anweisung1; Anweisung2; ...
```

Kommentare

Wie wir auf der vorangegangenen Folie gesehen haben, verwendet Python das `#` -Zeichen als Kommentarzeichen:
Alles, was in derselben Zeile auf das `#` -Zeichen folgt, wird als Kommentar aufgefasst und vom *Interpreter* ignoriert. Neben einzeiligen Kommentaren ist es auch möglich, längere Kommentare zu verfassen. Sie werden durch jeweils drei doppelte Anführungszeichen am Ende und Anfang eingeschlossen:

```
# Einzeiliger Kommentar

"""
Mehrzeiliger
Kommentar
"""
```

Leere Zeilen oder Zeilen, die ausschließlich *whitespace* -Zeichen (Leerzeichen, Tabulator-Zeichen, ...) enthalten, werden vom Interpreter grundsätzlich ignoriert.

→ Empfehlung

Auch wenn es schwer fällt:

Kommentieren/Dokumentieren Sie Ihre Programme systematisch und ausführlich.

Variablen

- Zum temporären Speichern von Informationen verwendet man *Variablen*. Wie in vielen anderen Skriptsprachen (Pearl, Ruby z.B.) ist es auch in Python nicht erforderlich, den Typ einer Variablen explizit zu deklarieren. Um einer Variablen einen Wert zuzuweisen, verwendet man den `=`-Operator:

```
>>> x = 4
>>> print(x)
4
>>> x = "Hello World!"
>>> print(x)
Hello World!
```

- Es ist möglich, in einer Anweisung mehreren Variablen einen oder auch verschiedene Werte zuzuweisen:

```
>>> x = y = z = 2 # x, y und z erhalten denselben Wert
>>> print(x * y * z)
8
>>> x, y, z = 2, 3, "Tesla"
>>> print(x * y, z)
6 Tesla
```

- Derselben Variablen können nacheinander Werte unterschiedlichen Typs zugewiesen werden. Den Typ einer Variablen kann man mit Hilfe der `type()` -Funktion erfragen:

```
>>> x = 77
>>> type(x)
int
>>> x = "Auto"
>>> type(x)
str
```

Für die Bildung und Verwendung von Variablennamen sollte man folgende Regeln beachten:

- Variablennamen dürfen nur Buchstaben, Zahlzeichen und Unterstriche enthalten.
- Sie müssen mit einem Buchstaben oder einem Unterstrich beginnen.
- Variablennamen dürfen keine Leerzeichen enthalten: `EvaKuhn`, `Eva_Kuhn`, aber **nicht**: `Eva Kuhn`
  - Stattdessen wird ein `_` verwendet um Lesbarkeit zu erhalten: `Eva_Kuhn`
- Groß- und Kleinbuchstaben werden unterschieden: *alt* und *Alt* sind zwei unterschiedliche Variablennamen!
- Namen von Python-Sprachelementen (Funktionen, Schlüsselwörter) sollten nicht als Variablennamen verwendet werden.

Mehr dazu im offiziellen Styleguide auf python.org

Python gehört zu den objektorientierten Programmiersprachen. Sie unterscheidet verschiedene Objekttypen. Wir werden uns hier besonders auch *Zahlen* und *Zeichenketten* konzentrieren:

2 Zahlen

Python kennt drei Zahlentypen:

1. ganze Zahlen/Integer (`int`);
2. Fließkomma-/Dezimalzahlen (`float`) und
3. komplexe Zahlen (`complex`).

`int` (Integer)

Natürlich gibt es die üblichen numerischen Funktionen um natürliche Zahlen

- zu addieren, zu subtrahieren, zu multiplizieren und zu dividieren:

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1
```

- Der `**`-Operator wird zur Bezeichnung von Exponenten verwendet:

```
>>> 3 ** 2
9
>>> 3 ** 3
27
>>> 10 ** 6
1000000
```

- Es ist auch möglich, natürliche Zahlen in *hexidezimaler* oder *oktaler* Form darzustellen:

```
>>> 0xA0F
2575
>>> 0o37
31
```

`float` (Dezimalzahlen)

Objekte vom Typ float können als Gleit-/Fließkommazahlen oder in Exponentialschreibweise notiert werden:

- `0.0`, `3.14`, `-2.794`, `32.3e-18`, `9.`, `-32.54e100`, ...

In den meisten Fällen gibt es bei der Verarbeitung von float-Zahlen keine Überraschungen:

```
>>> 0.1 + 0.1
0.2
>>> 2 * 0.2
0.4
```

→ kurzer Exkurs

- Der `+`-Operator kann je nach Kontext für die Addition oder die Konkatenation (Verknüpfung zweier Zeichenketten) stehen:

```
>>> meldung = "Das Ende der Welt " + "ist nahe!"
>>> print(meldung)
Das Ende der Welt ist nahe!
```

Was geschieht nun, wenn dieser Operator zwischen Argumenten unterschiedlichen Typs steht?

- Wenn es unterschiedliche numerische Typen sind, wird für das Ergebnis der allgemeinere Typ verwendet:

```
>>> 1 + 2.0
3.0
```

Wenn aber ein numerisches Argument mit einer Zeichenkette kombiniert wird, führt das zu einem *Type*-Fehler:

```
>>> 17 + ' Kühe'
Traceback (most recent call last):
File "<pyshell#23>", line 1, in <module>
17 + ' Kühe'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Fehler dieser Art lassen sich durch Verwendung der `str()`-Funktion vermeiden, die versucht, ihr Argument in eine Zeichenkette zu konvertieren:

```
>>> x = str(17) + ' Kühe'
>>> print(x)
17 Kühe
```

`complex` (Komplexe Zahlen)

Komplexe Zahlen haben die Form `a + bJ`, wobei `a` und `b` Fließkommazahlen sind und `J` die Quadratwurzel von -1, eine imaginäre Zahl, repräsentiert:

- `3.14j`, `9.322e3-36j`, `-.6545+0j`, ...

Wir werden in diesem Kurs keine komplexen Zahlen verwenden!

Mathematische Funktionen

Natürlich sind in *Python* alle wichtigen mathematischen Funktionen verfügbar:

- `abs()`, `exp()`, `log()`, `log10()`, `max()`, `min()`, `pow()`, `round()`, `sqrt()`, ...

Gleiches gilt für die zur Formulierung von Bedingungen erforderlichen Operatoren:

- `>`, `<`, `>=`, `<=`, `==`, `!=`

Wir verzichten an dieser Stelle darauf, weitere numerische Funktionen einzuführen, da für komplexe Berechnungen meistens spezielle Bibliotheken verwendet werden, auf die wir später in diesem Kurs eingehen werden.

Zahlenkonversion

- Die Funktionen `int()` , `float()` , `complex()` können verwenden werden, um Objekte in Zahlen des entsprechenden Typs zu konvertieren:

```
>>> float(1)
1.0
>>> int(3.99)
3
```

- Auch geeignete Zeichenketten lassen sich in Zahlen konvertieren:

```
>>> int("-1")
-1
>>> float("-1")
-1.0
>>> float("1 gr")
ValueError: could not convert string to float: '1 gr'
>>> int("1.3")
ValueError: invalid literal for int() with base 10: '1.3'
```

Variablenupdate

- Besonders in iterativen Konstrukten (Schleifen/Loops) kommt es häufig vor, dass die Werte von Variablen systematisch um einen bestimmten Betrag verändert werden:

```
var = var + x
var = var - x
var = var * x
var = var / x
```

- Die Operatoren `+=` , `-=` , `*=` , `/=` , ... erlauben es, Ausdrücke dieser Art kürzer zu formulieren:

```
>>> x = 1
>>> x += 4
>>> print(x)
5
>>> x *= 2
>>> x -= 1
>>> x
9
```

3 Zeichenketten

Der für uns wahrscheinlich wichtigste Datentyp in *Python* ist der `String`-Typ. In vielen Programmiersprachen werden Zeichenketten als Folgen von Zeichen behandelt. **Da es in *Python* keinen eigenen Zeichentyp gibt, werden umgekehrt einzelne Zeichen als Zeichenketten der Länge 1 behandelt.** Zeichenketten werden in *Python* von einfachen oder doppelten Anführungszeichen begrenzt. Anders als etwa in *Ruby* gibt es keine funktionalen Unterschiede zwischen beiden Typen von Anführungszeichen:

```
'Eine einfache Zeichenkette'
"Eine einfache Zeichenkette"
```

- Es ist daher möglich, Zeichenketten zu bilden, die selbst Anführungszeichen enthalten:

```
"Ich sage: 'Nichts ist unmöglich'"
'Ich sage: "Nichts ist unmöglich"'
```

`python3` verwendet *Unicode* um Zeichenketten zu verwalten; in `python2` wurden Zeichenketten intern noch als 8-Bit-ASCII-Sequenzen gespeichert.

- Natürlich können auch *Steuerzeichen* wie `\t` oder `\n` in Zeichenketten eingebettet werden:

```
>>> print("Python")
Python
>>> print("\tPython")
  Python
>>> print("A\nB\nC")
A
B
C
```

Zu den wichtigsten Steuerzeichen gehören:

- `\b` **Backslash**
- `\f` **Formfeed**
- `\n` **Newline**
- `\t` **Tab**ulator

Operatoren

Es gibt in Python eine kleine Zahl äußerst hilfreicher Operatoren zur Verarbeitung von Zeichenketten:

- Der `+`-Operator, der wie in Ruby je nach Kontext die Operation der Addition oder der Verkettung (Konkatenation) bezeichnet (genannt Operator Overloading), ermöglicht es, zwei oder mehr Zeichenketten miteinander zu kombinieren:

```
>>> vorname = "ada"
>>> nachname = "lovelace"
>>> name = vorname + " " + nachname
>>> print(name)
ada lovelace
>>> print("Hello, " + name.title() + "!")
Hello, Ada Lovelace!
>>> message = "Hello, " + name.title() + "!"
>>> print(message)
Hello, Ada Lovelace!
```

- Mit `*` lassen sich *n* Kopien einer Zeichenkette erzeugen:

```
>>> 'Ha' * 3
HaHaHa
```

- Mit den Operatoren `in` und `not in` kann man überprüfen, ob ein Zeichen in einer Zeichenkette enthalten ist oder nicht:

```
>>> "a" in "hallo"
True
>>> "x" not in "xerox"
False
```

- Nicht-leere Zeichenketten bestehen aus einer Folge von (elementaren) Objekten. Mit der aus aus der letzten Sitzung bekannten Funktion `len()` kann man auch die Länge von Zeichenketten bestimmen und die Operatoren `[]` und `[:]` erlauben es, ein Zeichen oder einen Abschnitt aus einer Zeichenkette zu extrahieren:

```
>>> len("Hallo")
5
>>> "Hallo"[0]
'H'
>>> "Hallo"[1:4]
'all'
```

- Wir erhalten in Python eine Zeichenkette, in der mit dem Backslash eingeleitete Teilausdrücke nicht als Steuerzeichen interpretiert werden, indem wir der Zeichenkette ein `r` oder `R` (`raw`) voranstellen:

```
>>> print("h\nallo")
h
allo
>>> print(r"h\nallo")
h\nallo
```

Besondere Beachtung sollte dem `%`-Operator gewidmet werden:
Er ermöglicht es, mit geringem Aufwand komplexe Formatierungsaufgaben zu lösen (vgl. `format()`, `printf()`, etc.).
Er fungiert dabei als Platzhalter und wird bei der Ausgabe durch einen konkreten Wert ersetzt. Eine Zeichenkette kann beliebig viele Platzhalter enthalten, die mit '%' beginnen und mit verschiedenen Angaben kombiniert werden können.

Andere Methoden und generell mehr Informationen dazu auf realpython.com

Es gilt dabei folgende Syntax, `type` ist obligatorisch:

```
%[flags][width][.precision]type
```

flags	**width**	**precision**	**type**
`-` linksbündig	*Breite*	*Präzision*	`s` String
`+` mit Vorzeichen	Anzahl der Zeichen	Anz. der Nachkommastellen	`i` Integer
`0` mit vorangestellten Nullen			`f` Float
`#` mit Zahlenpräfix (oktal, hexad.)			`e` Exponetial

```
>>> print("Ich habe %s Katzen und %s Hunde." % (3, -7))
Ich habe 3 Katzen und -7 Hunde.
>>> print("%13.3e" % (356.08977)) # Breite: 13; Nachkommastellen: 3; als Exponentialschreibweise
00003.561e+02
>>> print("%010.3f" % (356.08977)) # Breite: 10; Nachkommastellen: 3; als Dezimalzahl; mit Nullen auffüllen
0356.090
```

Methoden

Python stellt eine ganze Reihe von Methoden zur Bearbeitung von Zeichenketten zur Verfügung. ***Methoden*** sind, vereinfacht ausgedrückt, Funktionen, die auf Objekte einer Klasse angewendet werden können. Sie werden mit Hilfe des `.`-Operators aufgerufen: `objekt.methode()`

```
>>> name = "eva kuhn“
>>> print(name.title())
Eva Kuhn
>>> print(name.upper())
EVA KUHN
>>> print(name.lower())
eva kuhn
```

Mit den Methoden `strip()`, `lstrip()` und `rstrip()` kann man whitespace-Zeichen am Anfang und/oder am Ende einer Zeichenkette entfernen:

```
>>> skript_sprache = ' python '
>>> skript_sprache
' python '
>>> skript_sprache.rstrip()
' python'
>>> skript_sprache.lstrip()
'python '
>>> skript_sprache.strip()
'python'
```

Die `split()`-Methode zerlegt eine Zeichenkette und liefert als Wert eine Liste (von Zeichenketten), die das Ergebnis dieser Zerlegung repräsentiert. Wenn man auf weitere Angaben verzichtet, werden dabei die `Whitespace`-Zeichen als Trennzeichen verwendet:

```
>>> "Der alte Hase rennt ins Feld".split()
['Der', 'alte', 'Hase', 'rennt', 'ins', 'Feld']
```

`split()` akzeptiert bis zu zwei Argumente: Das erste legt fest, welche Zeichen oder `RegEx` als Begrenzer interpretiert werden und das zweite, wieviele Zerlegungen durchgeführt werden sollen:

```
>>> "Der alte Hase rennt ins Feld".split('a')
['Der ', 'lte H', 'se rennt ins Feld']
>>> "Der alte Hase rennt ins Feld".split(None, 1)
['Der', 'alte Hase rennt ins Feld']
```

Mit `join()` kann aus einer Folge von Strings eine neuer String gebildet werden, der alle Elemente der Folge enthält und zwischen zwei Elemente immer einen bestimmten Trenner setzt: `Trenner.join(Folge)`

```
>>> ':-'.join(['a','b','c'])
'a:-b:-c'
```

4 Reguläre Ausdrücke

auch **Regular Expression** `RegEx`

Für viele computerlinguistische Anwendungen ist notwendig, sprachliche Daten nach bestimmten *Mustern* zu durchsuchen. Dazu kann man u.a. ***reguläre Ausdrücke*** verwenden, die von vielen Programmiersprachen unterstützt werden. Ein regulärer Ausdruck ist eine Zeichenkette, die eine (potentiell: abzählbar unendliche) Menge von Zeichenketten bezeichnet.
Wir werden diese zunächst vielleicht eher Verwirrung stiftende Aussage auf den folgenden Seiten erläutern.

Natürlich ist es bereits mit den uns bekannten sprachlichen Mitteln möglich zu überprüfen, ob eine Zeichenkette in einer anderen Zeichenkette enthalten ist:

```
>>> s = "Aller Anfang ist schwer."
>>> "Anfang" in s
True
>>> "fang" in s
True
```

Jede einfache Zeichenkette α (wie z.B. `"Anfang"`) kann als ein trivialer regulärer Ausdruck betrachtet werden. Wenn ein regulärer Ausdruck für eine Menge von Zeichenketten steht, für welche Zeichenketten steht dann eine einfache Zeichenkette α? Nun, nur für α selbst:

$$\alpha := \{\alpha\}$$

Interessant wird es, sobald α bestimmte Sonderzeichen, die eine bestimmte Funktion besitzen, enthält. Dazu kommen wir gleich.

Reguläre Ausdrücke in Python

Um in Python mit regulären Ausdrücken arbeiten zu können, ist es sinnvoll bzw. erforderlich, mit `import re` die Bibliothek `re` zu laden. In dieser Bibliothek werden unter anderem die Funktionen `search()` und `match()` definiert, mit denen man überprüfen kann, ob ein durch einen regulären Ausdruck definiertes Suchmuster in einer Zeichenkette realisiert ist. Beide Funktionen haben folgende Syntax:

```
re.match(REGEX, ZEICHENKETTE, flags=0)
re.search(REGEX, ZEICHENKETTE, flags=0)
```

Beide Funktionen nehmen als erstes Argument einen regulären Ausdruck und als zweites Argument eine Zeichenkette. **Sie prüfen, ob das durch den regulären Ausdruck spezifizierte Muster in der Zeichenkette vorkommt oder nicht.** Anders formuliert: Sie prüfen, ob es in der Menge von Zeichenketten, die durch den regulären Ausdruck bezeichnet wird, Zeichenketten gibt, die in der Zeichenkette enthalten sind, die das zweiten Argument der `match()` - oder `search()` -Funktion bildet. Das Suchverhalten der Funktionen kann durch sogenannte Flags modifiziert werden, die als optionales drittes Argument verwendet werden können. Mehrere dieser Flags können mir dem oder-Operator `|` kombiniert werden.

Als Wert liefern die Funktionen ein sogenanntes `Match object` oder bei erfolglose Suche `None` zurück. Der Unterschied zwischen beiden Funktionen liegt darin, dass `match()` nur den Anfang der Zeichenkette überprüft, während `search()` die ganze Zeichenkette untersucht:

```
>>> import re
>>> x = re.match("Katze", "Katzen sind klüger als Hunde.")
>>> print(x)
<re.Match object; span=(0, 5), match='Katze'>
>>> x = re.match("als", "Katzen sind klüger als Hunde.")
>>> print(x)
None
>>> x = re.search("als", "Katzen sind klüger als Hunde.")
>>> print(x)
<re.Match object; span=(19, 22), match='als'>
>>> x = re.search("katze", "Ich mag Hauskatzen, aber keine Raubkatzen.")
>>> print(x)
<re.Match object; span=(12, 17), match='katze'>
```

Zu den wichtigsten `flags` gehören die folgenden:

- `re.I` - Groß- & Kleinschreibung wird ignoriert.
- `re.S` - Der Punkt kann auch für ein Zeilenumbruchzeichen (`newline`-Zeichen) stehen.
- `re.M` - Jeder Zeilenanfang und jedes Zeilenende wird wie der Anfang oder das Ende einer Zeichenkette behandelt (siehe: `$` und `^`)
- `re.X` - Es können Kommentare in die regulären Ausdrücke eingebettet werden.

```
>>> x = re.search("katzen","Katzen sind klüger als Hunde.", re.I)
>>> print(x)
<re.Match object; span=(0, 6), match='Katzen'>
>>> x = re.search("zen#wirklich?","Katzen sind klüger als Hunde.", re.I|re.X)
>>> print(x)
<re.Match object; span=(3, 6), match='zen'>
```

Die bislang betrachteten Fälle sind grundsätzlich einfach und fast schon trivial zu nennen, da sie in ganz ähnlicher Form mit dem `in` -Operator abgehandelt werden könnten.

Wir werden nun auf den folgenden Folien eine ganze Reihe von Sonderzeichen einführen, die diese Situation grundlegend verändern.

Sprechweise

Wenn es in einer Zeichenkette α eine Teilkette α' gibt, die in der Menge aller durch einen regulären Ausdruck r bezeichneten Zeichenketten enthalten ist, dann sagt man:

> *r matched α'*

Sonderzeichen

Beliebiges Zeichen

Viele reguläre Ausdrücke enthalten Zeichen, die in diesem Kontext eine besondere Bedeutung besitzen. Zu diesen Zeichen gehört z.B. der Punkt `.`, der in einem regulären Ausdruck für ein beliebiges Zeichen steht:

- Der Ausdruck `de.` matched Zeichenketten der Länge 3, die mit den Buchstaben `d` gefolgt von `e` beginnen: `der`, `des`, `den`, `dem`, `det`, ...

Durch das `escape`-Zeichen `\` kann diese Funktion des Punktes aufgehoben werden:

- `Dr\.` matched **nur** `Dr.`, **nicht aber** `Dre` usw.

Auswahl

Wenn an einer Position zwei oder mehr verschiedene Zeichen auftreten können, dann werden diese Zeichen in eckige Klammern gesetzt: Ein Ausdruck der Form `…` `[z1 z2 … zn]` `…` legt fest, dass an dieser Position eines der in der Klammer genannten Zeichen vorkommen muss:

- `de[rmsn]` matched `der`, `dem`, `des`, `den`

Statt einzelner Zeichen ist es in vielen Fällen auch möglich, Zeichenbereiche oder -klassen anzugeben. Dazu verbindet man das erste und letzte Zeichen des gewünschten Bereichs mit Hilfe eines Bindestrichs:

`[0–7]` → `[01234567]`
`[l–r]` → `[lmnopqr]`
`[–C–H]` → `[–CDEFGH]`

Das `–` -Zeichen direkt nach der öffnenden Klammer wird als ‚normales' Zeichen interpretiert.

Wenn man eine Auswahl längerer Zeichenfolgen wünscht, kann man das `pipe` -Symbol `|` verwenden: `Katze|Hund` matched eine Zeichenkette, die die Zeichenfolge `Katze` oder die Zeichenfolge `Hund` enthält.

Es gibt ein weiteres Zeichen, dass innerhalb eckiger Klammern eine besondere Bedeutung hat, sofern es direkt auf die öffnende Klammer folgt: Das `^`-Zeichen dient in diesem Fall als Negationsoperator. Der reguläre Ausdruck `[^aeiou]` ist zu lesen als: Jedes Zeichen außer `a`, `e`, `i`, `o` und `u`.

Zeichenklassen

Um die Formulierung von regulären Ausdrücken zu einfachen, gibt es verschiedene vordefinierte Zeichenklassen auf die mit Hilfe von *Kurzbezeichnern* verwiesen werden kann:

`\d` → `[0-9]`
`\D` → `[^0-9]`
`\s` → `[\t\n\r\f\v]` # Alle `whitespace`-Zeichen (`tab`, `newline`, ...)
`\S` → `[^\t\n\r\f\v]` # Alles außer `whitespace`-Zeichen
`\w` → `[a-zA-Z0-9_]`
`\W` → `[^a-zA-Z0-9_]` # Komplement von `\w`

Da der Backslash `\` in regulären Ausdrücken, wie man sieht, eine besondere Bedeutung hat, muss ihm diese besondere Bedeutung durch Voranstellen eines weiteren Backslash genommen werden, wenn man in einer Zeichenkette nach ihm suchen möchte: `\\`

Anker

Manchmal ist ist wichtig zu überprüfen, ob eine Zeichenkette mit einem bestimmten Ausdruck beginnt oder endet. Zu diesem Zweck bietet Python zwei Anker an: Das Zeichen `^` (außerhalb eckiger Klammern!) fordert, dass der nachfolgende Ausdruck mit dem Präfix der Zeichenkette matched und das Zeichen `$` entsprechend, dass der nachfolgende Ausdruck mit dem Suffix der Zeichenkette matched.

```
>>> x = "Frühstück ist die schönste Jahreszeit.“
>>> print(re.search("ist", x))
<re.Match object; span=(10, 13), match='ist'>
>>> print(re.search("^ist", x))
None
>>> print(re.search("ist$", x))
None
>>> print(re.search("^Früh", x))
<re.Match object; span=(0, 4), match='Früh'>
>>> print(re.search("zeit\.$", x))
<re.Match object; span=(33, 38), match='zeit.'>
```

Zwei weitere wichtige Anker: Mit `\b` werden Wortgrenzen, mit `\B` alle anderen Positionen in einer Zeichenkette gematched. Als Wortgrenze wird die Position vor dem ersten / nach dem letzten Zeichen einer Zeichenkette oder das `whitespace`-Zeichen zwischen zwei Wörtern/Token innerhalb der Zeichenkette interpretiert.

```
>>> x = "Die Frist ist abgelaufen."
>>> print(re.search(r"ist", x))
<re.Match object; span=(6, 9), match='ist'>
>>> print(re.search(r"\bist\b", x))
<re.Match object; span=(10, 13), match='ist'>
>>> print(re.search(r“\Bist\B“, x))
None
>>> print(re.search(r“\Bist\b“, x)
<re.Match object; span=(6, 9), match='ist'>
```

Mit dem Präfix `r` wird die nachfolgende Zeichenkette als *rohe* Zeichenkette oder `raw string` interpretiert; d.h. das `\`-Zeichen wird als einfaches Zeichen und nicht mehr als `escape`-Zeichen interpretiert; anderenfalls würde etwa die Sequenz `\b` das `backspace`-Zeichen bezeichnen und den gewünschten Match verhindern.

Wiederholungen

Natürlich kann in einer Zeichenkette ein Ausdruck mehrfach vorkommen. Die regulären Ausdrücke, die sich in Python und anderen Programmiersprachen formulieren lassen, erlauben es, sehr genau zu spezifizieren, wie oft ein Ausdruck innerhalb der Zeichenkette vorkommen muss, damit ein Match gefunden wird. So kann man festlegen, das ein Ausdruck `x`

- gar nicht oder einmal vorkommt (*optionales* Element): `x?`
- ein- oder mehrmals vorkommt: `x+`
- beliebig oft (inkl. 0-mal) vorkommt: `x*`

```
_>>> print(re.search(r"(Dr\.)? Merkel\b", "Frau Merkel trifft Putin."))
<re.Match object; span=(4, 11), match=' Merkel'>
```

```
>>> print(re.search(r"(Dr\.)?\sMerkel\b", "Frau Dr. Merkel trifft Putin."))
<re.Match object; span=(5, 15), match='Dr. Merkel'>
```

```
>>> print(re.search(r"(Dr\.)?\sMerkel\b", "Frau Dr. Dr. Merkel trifft Putin."))
<re.Match object; span=(9, 19), match='Dr. Merkel'>
```

Durch zusätzliche Angaben kann außerdem genau spezifiziert werden, wie oft ein Ausdruck wiederholt werden kann. Dazu wird folgende Syntax verwendet: `{Untergrenze,Obergrenze}`.
Beide Angaben sind grundsätzlich optional: Bei fehlender `Untergrenze` wird `0`; bei fehlender `Obergrenze` `unendlich` als Wert angenommen.

- `{0,1}` ist gleichbedeutend mit `?`
- `{0,}` ist identisch mit `*`
- `{1,}` mit `+`.

Grundsätzlich gilt:
Es wird nach der längsten (Teil)Zeichenkette gesucht, die durch den regulären Ausdruck erfasst wird.

Angenommen, wir wollen einen regulären Ausdruck formulieren, der in Anschriften die Zeile findet, die die Postleitzahl und den Ortsnamen enthält. Jede Postleitzahl ist eine 4- bzw. 5-stellige Zahl und jeder Ortsname besteht aus mindestens drei Zeichen. Wir erhalten so folgenden Ausdruck: `r"^[0-9]{4,5} [A-Za-z]{3,}"`

Aber es gibt auch komplexe Ortsnamen, bei denen zwischen den Namensteilen eine Leerzeichen oder ein Bindestrich stehen kann. Wir gehen davon aus, dass es höchstens dreiteilige Ortsnamen gibt: `r"^[0-9]{4,5} [A-Za-z]{3,}([\s-][A-Za-z]{2,}){,2}"`

```
>>> print(re.search(r"^[0-9]{4,5} [A-Za-z]{3,}", "54290 Trier"))
<re.Match object; span=(0, 11), match='54290 Trier'>
>>> print(re.search(r"^[0-9]{4,5} [A-Za-z]{3,}([\s-][A-Za-z]{2,}){,2}", "54290 Trier Quint"))
<re.Match object; span=(0, 17), match='54290 Trier Quint’>
```

Gruppen und Rückwärtsverweise

Wie schon die vorangegangenen Beispiele gezeigt haben, können wir reguläre Ausdrücke durch die Verwendung runder Klammern strukturieren. Jeder geklammerte Teilausdruck wird als *Gruppe* bezeichnet und alle Gruppen in einem regulären Ausdruck werden aufsteigend indiziert (Index der ersten Gruppe: 1).

`search()` bzw. `match()` liefern, wie wir gesehen haben, als Wert ein `Match object`. Die Methode `.group()` extrahiert aus einem `Match object` die Teilzeichenkette, die den regulären Ausdrück erfüllt. Enthält der reguläre Ausdruck *Gruppen*, kann auf den Teilausdruck, auf den eine bestimmte Gruppe zutraf, zugegriffen werden:

- Durch den regulären Ausdruck `[0-9]+` wird nach einer Zeichenkette gesucht, die aus einer oder mehreren Ziffern besteht:

```
>>> x = re.search("[0-9]+", "Anschrift: Universität Trier 54286 Trier Universitätsring 12")
```

- `.group()` liefert die Teilzeichenkette `Z` zurück, die dem regulären Ausdruck entspricht:

```
>>> x.group()
'54286'
```

```
>>> x = re.search("([0-9]+)\s([a-zA-Z]+)", "Anschrift: Universität Trier 54286 Trier (...) ")
>>> print(x)
<re.Match object; span=(29, 40), match='54286 Trier'>
>>> x.group()
'54286 Trier'
>>> x.group(1)
'54286'
>>> x.group(2)
'Trier'
>>> x.group(3)
IndexError: no such group
```

Mit den Methoden `.span()`, `.start()` und `.end()` kann man auf die Position der gefundenen Teilzeichenkette in der Zeichenkette zugreifen:

- `.span()` liefert ein Tupel mit der Anfangs- und Endposition der Teilzeichenkette `Z`

```
>>> x = re.search("([0-9]+)\s([a-zA-Z]+)", "Anschrift: Universität Trier 54286 Trier (...) ")
>>> print(x)
<re.Match object; span=(29, 40), match='54286 Trier'>
>>> x.span()
(29, 40)
>>> x.start() # identisch mit span()[0]
29
>>> x.end() # identisch span()[1]
40
>>> x.span()[0]
29
>>> x.span()[1]
40
```

Es ist möglich, sich in einem regulären Ausdruck selbst auf den Abschnitt einer Zeichenkette zu beziehen, der durch eine bestimmte Gruppe gematched wurde:
Durch `\n` (wobei `n` = `1`, `2`, `3`, ...) verweist man auf die Zeichenfolge, die durch die `n`-te Gruppe eines regulären Ausdrucks gebunden wurde.

Der reguläre Ausdruck `([a-c])x\1x\1` steht für `{'axaxa', 'bxbxb', 'cxcxc'}`.

Weitere Methoden

Eine weitere wichtige Methode der `re`-Bibliothek ist `.sub()`, die es einem ermöglicht, alle bzw. eine bestimmte Anzahl von Vorkommen eines Ausdrucks in einer Zeichenkette zu ersetzen:

`re.sub(MUSTER, ERSATZ, ZEICHENKETTE, ANZAHL=0)`

Ein `ANZAHL` Argument ist optional: Wird auf es verzichtet oder der Standardwert 0 explizit verwendet, werden alle Vorkommen ersetzt; sonst die ersten `n` Vorkommen.

```
>>> re.sub('a', 'o', 'alle affen haben langeweile')
'olle offen hoben longeweile'
>>> re.sub('a', 'o', 'alle affen haben langeweile', 2)
'olle offen haben langeweile'
>>> re.sub('#.*', '', '54286 # Eine Postleitzahl im Raum Trier')
'54286 '
```

Die Methoden `.findall()` und `.finditer()` finden alle Zeichenketten, die einen regulären Ausdruck erfüllen und liefern als Wert eine Liste dieser Zeichenketten bzw. einen *Iterator*.

```
>>> re.findall('a', 'alle affen haben langeweile')
['a', 'a', 'a', 'a']
>>> re.finditer('a', 'alle affen haben langeweile')
<callable_iterator object at 0x10494b8d0>
```