E.Tutorial
www.et.ethz.ch

Programmieren mit Python Modul 3

Sequenzielle Datentypen, Such- und Sortieralgorithmen, Sequenzanalyse

Theorieteil

Inhaltsverzeichnis

Begriffe

Liste	Listen-Bereich	Suchalgorithmus
Tupel	geschachtelte Liste	
Index	Listen-Abstraktion	Sortieralgorithmus
Dimension	Dictionary	
Listen-Durchlauf	Schlüssel-Wert Paar	Sequenzanalyse

Autoren:

Lukas Fässler

E-Mail:

et@ethz.ch

Datum:

18 October 2024

Version: 1.1

Hash: c539f64

Trotz sorgfältiger Arbeit schleichen sich manchmal Fehler ein. Die Autoren sind Ihnen für Anregungen und Hinweise dankbar!

1 Modulübersicht

Dieses Modul befasst sich mit folgenden drei Themenbereichen:

1. **Datenstrukturen für zusammengehörige Daten**: Bis hierhin haben Sie als Datenstruktur vor allem Variablen verwendet. Damit kann jeweils nur ein Wert gespeichert werden. Um mehrere zusammengehörige Werte unter einem gemeinsamen Bezeichner ablegen zu können, kommen in allen Programmiersprachen spezielle Datenstrukturen zum Einsatz. Einen **sequentiellen Datentyp** haben Sie bereits in Modul 1 kennen gelernt. Zeichenketten (Strings) speichern eine Folge von Einzelzeichen. In Python kommen als sequentielle Datentypen **Listen**, **Tupel** und **Dictionary** zum Einsatz, die mit diesem Modul eingeführt werden.
2. **Such- und Sortieralgorithmen**: Die Suche von einzelnen Elementen in einer Datensammlung und das Sortieren von Daten sind zwei der häufigsten Aufgaben, mit denen eine Programmiererin oder ein Programmierer konfrontiert ist. In dieser Einführung erfahren Sie, wie die populärsten Vertreter der Such- und Sortieralgorithmen funktionieren, die Sie im praktischen Teil dieses Moduls umsetzen werden.
3. **Sequenzanalyse**: In diesem Modul werden Sie Ihre Programmierkenntnisse einsetzen, um eine Hypothese aus der Biologie mittels DNA-Sequenzanalyse zu überprüfen. In diesem Abschnitt finden Sie eine kurze Einführung zum Thema Sequenzanalyse.

2 Datenstrukturen für zusammengehörige Werte

Beim Programmieren werden oft **zusammengehörige Daten** verwendet (z.B. Temperaturen, Lottozahlen, Termine, Trainingszeiten etc.). Um eine zusammengehörige Gruppe von Elementen unter einem Bezeichner abzuspeichern, verwendet man in allen modernen Programmiersprachen spezielle Datenstrukturen. **Listen** (*lists*) sind eine Datenstruktur zur Speicherung und Organisation von Daten, die in vielen Programmiersprachen vorkommen. Sie bestehen aus einer geordneten Menge von n Elementen. Die Elemente können über einen sogenannten **Index** angesprochen werden. Dieser gibt die Position eines Elements in einer Liste an. In vielen Programmiersprachen hat dabei das **erste Element** den **Index 0**, das zweite den Index 1 und das letzte den Index $n - 1$ (siehe Beispiel in Tabelle 1).

Index	0	1	2	3	4	5
Wert	12	13	15	17	23	32

Tabelle 1: Beispiel für eine Liste mit sechs Elementen.

Besteht ein Element einer Liste selbst wieder aus einer Liste, entsteht eine **geschachtelte**

Liste (*nested list*). Man kann es sich als Tabelle mit `m` mal `n` Elementen vorstellen, die jeweils über zwei Indizes angesprochen werden (siehe Beispiel in Tabelle 2).

Index	0	1	2	3	4	5
0	12	13	15	17	23	39
1	14	53	45	87	27	62
2	22	33	17	19	83	32

Tabelle 2: Beispiel für eine geschachtelte Liste mit drei mal sechs Elementen.

3 Sequenzielle Datenstrukturen in Python

Python unterscheidet die sequenziellen Datenstrukturen **Listen** (*list*), **Tupel** (*tuple*) und **Dictionaries**. In Python verwendet man als Datenstruktur auch häufig so genannte *NumPy-Arrays*. Diese werden in einem späteren Modul eingeführt.

3.1 Listen

3.1.1 Listen erstellen

In Python werden **Listen** erzeugt, indem Werte in **eckige Klammern** `[]` eingeschlossen und deren Elemente durch **Kommata** (`,`) getrennt werden.

Beispiel:

```
# Leere Liste a.

a = []

# Liste mit 3 Elementen 1, 2, 3.

a = [1,2,3]
```

Die einzelnen Elemente einer Liste können von unterschiedlichem Datentyp sein. Folgende Liste umfasst vier Elemente von drei Datentypen String, Integer und Float.

Beispiel:

```
b = ["Freitag", 13, "März", 10.30]
```

Listen können **ineinander verschachtelt** sein, indem sie andere Listen als Unterlisten enthalten (*nested lists*).

Beispiel:

```
c = [["Freitag","Samstag"], [13,14], "März", [10.30,11.30]]
```

3.1.2 Hinzufügen und Löschen von Listen-Elementen

In Python gibt es spezielle Methoden, um Listen neue Elemente hinzuzufügen oder Elemente zu löschen. Hier sollen beispielhaft `append()`, `insert()` und `pop()` erwähnt werden.

Beispiel:

```
d = ["Freitag", 14, "März"]

d.append("15:33")

# Fügt der Liste am Ende ein weiteres Element hinzu.

# Ausgabe:
# Freitag 14 März 15:33

d.pop(0)

# Löscht das erste Element der Liste.

# Ausgabe:
# 14 März 15:33

d.insert(0, "Donnerstag")

# Fügt ein neues Element an die erste Stelle
# (Index 0) der Liste.

# Ausgabe:
# Donnerstag 14 März 15:33
```

3.2 Tupel

Tupel sind den Listen ähnlich. Der Unterschied besteht darin, dass Tupel im Gegensatz zu Listen *nicht* verändert werden können.

3.2.1 Tupel erzeugen

In Python erzeugt man Tupel, indem Werte in **runde Klammern** `()` eingeschlossen und durch **Kommata** (`,`) getrennt werden.

Beispiel:

```
# Leeres Tupel e.

e = ()

# Tupel mit 3 unveränderbaren Elementen 1, 2, 3.

e = (1,2,3)
```

3.3 Zugriff auf einzelne Elemente (Indexierung)

Auf einzelne Elemente von Listen und Tupel wird über einen **Index** (z.B. `i`) zugegriffen. `x[i]` liefert somit das Element aus der Liste `x` an der Position `i`. Es gilt zu beachten, dass die Indizes bei 0 beginnen und bei `n-1` enden, wobei `n` der Anzahl der Elemente entspricht.

Beispiel:

```
# Liste f mit 3 Elementen.

f = [1,2,3]

# Aufruf des ersten Elements mit Index 0.

print(f[0])

# Resultat: 1.

# Addition zweier Elemente.

print(f[1]+f[2])

# Resultat: 5.
```

Bei Python werden beim Zugriff auf einzelne Elemente sowohl bei Listen als auch bei Tupel *immer* **eckige Klammern** `[index]` gesetzt.

Beispiel:

```
# Liste g und Tupel h mit je 3 Elementen.

g = ["Freitag", 14, "März"]
h = ("Freitag", 13, "Mai")
print(g[0], g[1], g[2])
print(h[0], h[1], h[2])

# Ausgabe:
# Freitag 14 März
# Freitag 13 Mai

# Veränderung der Werte zweier Listen-Elemente.

g[0] = "Samstag"
g[1] = 15
print(g[0], g[1], g[2])

# Ausgabe:
# Samstag 15 März

# Veränderung von Tupel-Elementen führt
# zu einer Fehlermeldung.

h[0] = "Samstag"

# TypeError: tuple object does not support item assignment
```

Mit negativen Indizes kann man bei Python auch Elemente vom Ende einer Liste her gezählt ansprechen. Das letzte Element hat dabei den Index -1.

Beispiel:

```
i = ["Freitag", 14, "März"]
print(i[-1], i[-2])

# Ausgabe:
# März 14
```

Um auf ein einzelnes Element einer **geschachtelten Liste** zuzugreifen, werden die zwei Indizes für die Zeilen- und Spaltennummer angegeben.

Beispiel:

```
# Liste j mit 2 mal 2 Elementen.

j = [[0,0],[0,0]]

# Veränderung des ersten Listen-Elements von j.

j[0][0] = 22
j[0][1] = 24
j[1][0] = 36
j[1][1] = 39

# Neue Speicherbelegung von j:
# [[22,24],[36,39]]
```

3.4 Zugriff auf Bereiche (Slicing)

Der Aufruf von **Bereichen** wird bei Python als *Slicing* (engl. für Ausschneiden) bezeichnet. Um Bereiche von Listen (Teillisten) aufzurufen, kann ein **Doppelpunkt** (`:`) verwendet werden. Alleine steht er für alle Elemente. Mit einer Zahl vor oder nach dem Doppelpunkt kann der Bereich mit `start:stop` eingegrenzt werden. `start` bezieht sich auf den Index des Elementes am Anfang des Bereichs und `stop` bezieht sich auf den Index des Elements eins *bevor* die Angabe des Bereichs endet. Negative Werte ändern die Richtung der Zählweise des Index vom Ende der Liste her.

Beispiel:

```
# Liste k mit 3 Elementen.

k = [1,2,3]

# Aufrufe von Listen-Bereichen mit Doppelpunkt.

print(k[:])

# Resultat: 1,2,3.

print(k[1:])

# Resultat: 2,3.

print(k[1:3])

# Resultat: 2,3.

print(k[1:-1])

# Resultat: 2.
```

3.5 Listen-Durchlauf mit Schleifen

Mit einer for-Schleife können **alle Elemente einer Liste durchlaufen** werden. Es gibt dabei verschiedene Möglichkeiten:

- Iteration über eine Zahlenfolge und Zugriff auf den Listen-Index
- Iteration über eine Liste ohne Zugriff auf den Listen-Index
- Iteration mit der Funktion `enumerate()`

3.5.1 Iteration über eine Zahlenfolge und Zugriff auf den Listen-Index

Bei dieser Variante wird die Funktion `range()` verwendet, um eine **Laufvariable** mit einer Zahlenfolge in einem definierten Bereich zu variieren (siehe Einführung der for-Schleife im vorherigen Modul). Beim Listenzugriff entspricht der Wert der **Laufvariablen** dem **Index-Wert** der Liste.

Beispiel:

```
# Liste m mit 5 Elementen.

m = [1,2,3,4,5]

# Aufruf aller Elemente von m unter Einsatz
# einer for-Schleife (Bereich 0 bis 4).

for i in range(0,5):
    print(m[i])

# Ausgabe der 5 Listen-Elemente.
```

Erklärung: Die for-Schleife zählt von 0 bis 4. Bei jedem Schleifendurchlauf wird die Variable `i` als Index verwendet, um das entsprechende Listen-Element auszudrucken.

Um verschachtelte Listen iterativ zu bearbeiten, sind **geschachtelte Schleifen** mit mehreren Index-Variablen notwendig.

Beispiel:

```
# Verschachtelte Liste n mit 2 mal 3 Elementen.

n = [[0,0,0],[0,0,0]]

# Zugriff auf alle 6 Listen-Elemente
# mittels geschachtelter Schleife.

for i in range(0,2):
  for j in range(0,3):
      n[i][j] = 5

# Speicherbelegung von n:
# [[5, 5, 5], [5, 5, 5]]

# Ausgabe der 2-dimensionalen Struktur in der Konsole

for i in range(0,2):
  for j in range(0,3):
      print(n[i][j], end= " ")
  print()

# Ausgabe:
# 5 5 5
# 5 5 5
```

Erklärung: Die beiden Laufvariablen `i` und `j` werden zunächst auf den Wert 0 gesetzt. Das erste Listen-Element, auf das zugegriffen wird, heisst `n[0][0]`. Danach wird die innere Schleife mit der Laufvariablen `j` durchlaufen. Die nächsten Listen-Elemente heissen `n[0][1]` und `n[0][2]`. Nach Abschluss der inneren Schleife geht es zurück in die äussere Schleife, die Laufvariable `i` wird auf den Wert 1 gesetzt und die innere Schleife beginnt wieder bei 0. Die weiteren Listen-Elemente heissen `n[1][0]`, `n[1][1]` und `n[1][2]`. Um die Listen-Elemente einer verschachtelten Liste in der Konsole in Zeilen und Spalten anzuordnen, muss nach jedem Element mit `end= " "` der automatische Zeilenumbruch unterbunden und am Ende einer Zeile mit `print()` wieder ein Zeilenumbruch eingefügt werden.

3.5.2 Iteration über eine Liste ohne Zugriff auf den Listen-Index

Bei dieser Variante durchläuft die for-Schleife *alle* Elemente einer Liste mit Hilfe einer Variablen ohne dass wir den Listen-Index benötigen.

Beispiel:

```
# Liste o mit 5 Elementen.

o = [1,2,3,4,5]

# Aufruf aller Elemente der Liste o mit der Variable var.

for var in o:
    print(var)

# Ausgabe der 5 Listen-Elemente.
```

3.5.3 Die Funktion enumerate()

Mit der Funktion `enumerate()` kann ebenfalls über eine Liste "geloopt" werden. Die Funktion enthält zusätzlich einen automatischen Zähler mit beliebigem Startwert.

Beispiel:

```
p = ["Montag", "Dienstag", "Mittwoch"]

for zaehler, wert in enumerate(p, 1):
    print(zaehler, ":", wert)

# Ausgabe:
# 1 : Montag
# 2 : Dienstag
# 3 : Mittwoch
```

3.6 Strings als Listen

Der Datentyp **String** (Zeichenketten), den Sie bereits seit Modul 1 verwenden, besteht ebenfalls aus einer geordneten Menge von Elementen; in diesem Fall beliebigen Einzelzeichen. Bei einem String kann deshalb auch auf einzelne Elemente (also einzelne Zeichen) zugegriffen werden.

Beispiel:

```
# String mit dem Namen q und der Zeichenfolge "Hallo".

q = "Hallo"

print(q[1])

# Ausgabe: a.

print(q[-1])

# Ausgabe: o.

print(q[0:3])

# Ausgabe: Hal.
```

3.7 Berechnung der Listen-Länge

Auch wenn Funktionen erst im nächsten Modul im Zentrum stehen werden, sollen an dieser Stelle die im Zusammenhang mit Listen häufig verwendete Funktion `len()` erwähnt werden, mit der die **Länge einer Liste** abgefragt werden kann.

Beispiel:

```
r = [1,2,3]

# Bestimmung der Länge von r.

print(len(r))

# Ausgabe:
# 3
```

Häufig wird die Funktion `len()` verwendet, um beim Durchlaufen der Liste die obere Grenze der Schleife auszurechnen. Dies hat den Vorteil, dass bei einer Änderung der Listen-Länge die Schleife nicht angepasst werden muss.

Beispiel:

```
s = [1,2,3]

# Der Endwert der for-Schleife wird durch die Funktion
# len() bestimmt.

for i in range(0,len(s)):
    print(s[i])

# Ausgabe der 3 Listen-Elemente.

# Aenderung der Listen-Länge.

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

# Dieser Anweisungsblock muss nicht angepasst werden.

for i in range(0,len(s)):
    print(s[i])

# Ausgabe der 10 Listen-Elemente.
```

3.8 Listen-Abstraktion

Mit **Listen-Abstraktionen** (Engl. *List Comprehension*) können mit wenig Code neue Listen erstellt und mit Daten gefüllt werden.

Eine effiziente Form, in Python grössere Listen zu erzeugen und mit einem **Einheitswert** (z.B. 0) zu initialisieren, lautet wie folgt:

```
    meineListe = [einheitswert for x in range(anzahl)]
```

- **meineListe**: Name der Liste.
- **einheitswert**: Dieser Wert wird für alle Listen-Elemente gesetzt.
- **anzahl**: Anzahl gewünschter Listen-Elemente, resp. Listen-Länge.

Beispiel:

```
# Liste t mit 100 Elementen; alle mit Wert 0 initialisiert.

t = [0 for x in range(100)]
```

Eine aufsteigende Liste in einem definierten Bereich kann wie folgt erzeugt werden:

Beispiel:

```
# Aufsteigende Liste u von 1 bis 99.

u = [x for x in range(1,100)]
```

Listen-Werte können auch durch Formeln generiert werden.

Beispiel:

```
# Durch eine Formel generierte Liste v.

v = [x**2 for x in range(10)]

# Es wird folgende Liste generiert:
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Auch geschachtelte Listen können mittels Listen-Abstraktion erzeugt werden.

Beispiel:

```
# Geschachtelte Liste w mit 4 mal 3 Elementen
# und Einheitswert 0.

w = [[0 for x in range(3)] for y in range(4)]

# Es wird folgende Liste generiert:
# [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

3.9 Dictionaries

Dictionaries ("Wörterbuch", selten mit *assoziative Liste* übersetzt) haben einen ähnlichen Aufbau wie Listen. Während bei Listen der Zugriff über den Index geschieht (Ganzzahlen beginnend bei 0 für das erste Element), bieten Dictionaries die Möglichkeit, selbst-definierte Indizes von einem beliebigen Datentyp (also auch Strings) zu verwenden. In einem Dictionary wird der Index **Schlüssel** (*key*) genannt. Dictionaries kann man sich als ungeordnete Menge von Schlüssel-Wert-Paaren (*key-value pairs*) vorstellen, wobei die Schlüssel innerhalb eines Dictionaries eindeutig sein müssen. Die Daten sind also nicht nach Index sortiert.

3.9.1 Dictionary erstellen

Dictionaries werden mit **geschweiften Klammern** `{}` erzeugt und die einzelnen Elemente durch **Kommata** (`,`) getrennt. Jedes Element enthält einen Schlüssel und einen Wert, die durch **Doppelpunkt** (`:`) voneinander getrennt sind.

```
# leere Dictionary x.

x = {}

# Dictionary mit den Schlüsseln "gelb", "rot" und "grün"
# und den zugehörigen Werten 19, 11 und 22.

x = {"gelb": 19, "rot": 11, "grün": 22}

# Zugriff auf Werte über den Schlüssel.

print(x["grün"] + x["rot"])

# Ausgabe:
# 33
```

3.9.2 Iterieren über Dictionaries

Mit for-Schleifen und den Methoden `keys()`, `values()` und `items()` können Schlüssel, Werte oder beides zusammen durchlaufen werden.

```
# Dictionary x mit 3 Schlüssel-Wert Paaren.

x = {"gelb": 19, "rot": 11, "grün": 22}

# keys() gibt eine Liste von Schlüsseln zurück.

for var in x.keys():
    print(var)

# Ausgabe:
# gelb
# rot
# grün

# values() gibt eine Liste der Werte zurück.

for var in x.values():
    print(var)

# Ausgabe:
# 19
# 11
# 22

# items() gibt die Schlüssel-Wert-Paare zurück.

for var in x.items():
    print(var)

# Ausgabe:
# ('gelb', 19)
# ('rot',  11)
# ('grün', 22)
```

4 Such- und Sortieralgorithmen

Sind Daten in Listen abgelegt, werden häufig darauf Algorithmen angewendet, die einzelne Elemente suchen oder mehrere Elemente der Reihe nach sortieren. Wir beschränken uns

hier auf eine kurze Beschreibung einiger ausgewählter Vertreter, die Sie im praktischen Teil selber implementieren werden.

4.1 Suchalgorithmen

Es gibt verschiedene **Suchalgorithmen**, die sich in ihrem Aufbau und ihrer Effizienz unterscheiden. Die bekanntesten sind die *lineare* und die *binäre Suche*.

4.1.1 Lineare Suche

Bei der *linearen Suche* wird eine Menge von Elementen nach einem bestimmten Element durchsucht. Die Suche beginnt beim ersten Element, und die Elemente werden in der Reihenfolge durchlaufen, in der sie abgespeichert sind. Entspricht das betrachtete Element dem gesuchten Element, wird die Suche beendet, ansonsten wird weiter gesucht.

So kann zum Beispiel mit der *linearen Suche* nach dem *maximalen* Wert gesucht werden:

1. Die Position des (momentanen) Maximums wird in einer Variablen (z.B. `max`) gespeichert.
2. Zuerst wird das erste Element der Liste als das Maximum angenommen.
3. Es werden nun alle Elemente der Liste (ausser des ersten) durchlaufen.
4. Ist der Wert des Feldes an der momentanen Position grösser als das bisher angenommene Maximum, dann wird diese Position in `max` gespeichert.

4.1.2 Binäre Suche

Die Voraussetzung für die *binäre Suche* ist, dass die Daten in sortierter Form vorliegen.

Die *binäre Suche* funktioniert wie folgt:

1. Es wird das mittlere Element der sortierten Datenmenge untersucht. Ist dieses grösser als das gesuchte Element, muss nur noch in der unteren Hälfte gesucht werden, andernfalls in der oberen.
2. Die Suche wird bei jedem darauf folgenden Durchlauf auf die neue Datenmenge angewendet. Der Suchraum (d.h. der Teil der Datenmenge, welcher durchsucht wird) halbiert sich dadurch bei jedem Durchgang.
3. Die Suche endet, sobald das gesuchte Element gefunden wurde oder nur noch ein Element übrig bleibt. In diesem Fall ist es entweder das gesuchte Element oder es kommt in der Datenmenge nicht vor.

4.2 Sortieralgorithmen

Das Ziel von **Sortieralgorithmen** ist es, die Elemente einer Menge nach einem bestimmten Kriterium zu sortieren. Nach dem Sortieren liegen die Elemente in aufsteigender oder absteigender Reihenfolge vor.

Es gibt verschiedene Sortieralgorithmen, die sich in ihrem Aufbau und ihrer Effizienz unterschieden. Die bekanntesten sind *Bubble-Sort*, *Insertion-Sort*, *Merge-Sort* und *Quick-Sort*. Hier soll stellvertretend *Bubble-Sort* betrachtet werden.

4.2.1 Bubble-Sort

So erreichen Sie mit *Bubble-Sort* eine aufsteigende Sortierung (siehe Abbildung 1):

1. Es werden jeweils zwei benachbarte Elemente einer Liste verglichen. Begonnen wird mit den Elementen mit dem Index 0 und 1, dann 1 und 2, dann 2 und 3 etc.
2. Wenn der Wert des linken Elements grösser ist als der Wert des rechten, werden die beiden Werte vertauscht.
3. Mit einem Listendurchlauf „wandert“ so das grösste Element ans Ende der Liste.
4. Nun werden die Schritte 1 bis 3 wiederholt, um das zweitgrösste Element an die zweitletzte Position zu bringen. Der Vergleich des zweitletzten Elements mit dem letzten entfällt, da das letzte das grösste ist.
5. Die Schritte 1 bis 4 werden so lange wiederholt, bis die zwei kleinsten Elemente miteinander verglichen werden.

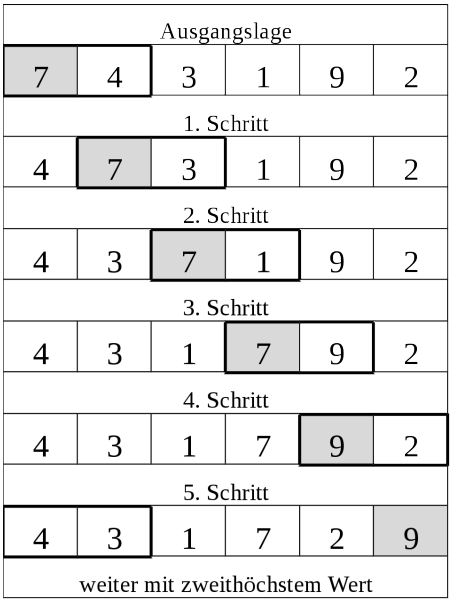

Abbildung 1: Beispiel für Bubble-Sort. Details siehe Text.

5 Sequenzanalyse

Die **Sequenzanalyse** gehört zu den wichtigsten Anwendungen der Bioinformatik. Durch die Entwicklung von Methoden zur Analyse von Gen- und Proteinsequenzen ist die Zahl

von bekannten Sequenzen, die in Bio-Datenbanken aufgenommen wurden, stark gestiegen. Um die Biologie des Organismus, von dem die Sequenzen stammen, zu verstehen, ist der automatisierte Vergleich von Sequenzdaten von DNA, RNA oder Protein mit bereits bekannten Sequenzen ein wichtiges Mittel. Untersucht wird die Abfolge und die Position einzelner Moleküle in der Sequenz mit dem Ziel, Erkenntnisse über Eigenschaften, Funktion, Strukutur und Evolution zu gewinnen.

Die Nukleotiden der **Desoxyribonukleinsäure** (engl. *DNA*) unterscheiden die **4 Basen** **A**denin, **T**hymin, **G**uanin und **C**ytosin. Bei der Sequenzanalyse wird die DNA in Form von Zeichenketten (*String*) angegeben, wie zum Beispiel `"ATCCTATC"`. Beim Vergleich zweier DNA-Sequenzen kann es vorkommen, dass wir in einer Sequenz an einer Position statt `"C"` ein `"G"` finden. Eine solche Änderung eines Moleküls wird *Single Nucleotide Polymorphism (SNP)* genannt.

Beispiel eines SNP in Sequenz 2 an Position 3:

```
     1 2 3 4 5 6 7 8
Seq1 A T C C T A T C
Seq2 A T G C T A T C
```

In anderen Fällen kann es passieren, dass ein Molekül fehlt (*Deletion*) oder wir finden ein zusätzliches Molekül (*Insertion*).

Beispiel einer Insertion in Sequenz 1 an Position 3:

```
     1 2 3 4 5 6 7 8
Seq1 A T C C T A T C
Seq2 A T - C T A T C
```

In diesen beiden Beispielen ist die Sequenzlänge kurz und die Ähnlichkeit hoch. Daher ist es einfach, die Sequenzen visuell abzugleichen. Wenn Sie jedoch hunderte oder tausende von Positionen abgleichen müssen, würden Sie wahrscheinlich nicht alles von Hand vergleichen wollen. Daher gibt es hierfür spezielle Programme, die Algorithmen zum Alignment von Sequenzen verwenden.