1.11 Listen: viele Werte unter einem Namen

Leitidee

Stell dir vor, du willst die Noten deiner letzten fünf Klassenarbeiten in einem Programm speichern. Mit dem, was du bisher kennst, brauchst du dafür fünf Variablen: `note1`, `note2` und so weiter. Bei den Noten einer ganzen Klasse oder bei den Temperaturen eines ganzen Jahres wird das schnell unübersichtlich. Programme brauchen deshalb eine Möglichkeit, **viele Werte unter einem Namen** zu speichern: die **Liste**. Vieles daran kennst du schon von den Zeichenketten – auch in einer Liste hat jeder Wert eine Position, und gezählt wird ab 0. Am Ende dieses Abschnitts programmierst du eine Einkaufsliste.

Lernziele

Nach diesem Abschnitt kannst du

- eine Liste (ein Feld) als Datenstruktur beschreiben, die mehrere Elemente der Reihe nach zu einer Einheit zusammenfasst,
- zwischen dem **Index** und dem **Element** an dieser Stelle unterscheiden,
- über den Index lesend und schreibend auf einzelne Listenelemente zugreifen,
- Elemente mit `append`, `insert`, `pop` und `remove` hinzufügen und entfernen,
- mit einer Schleife alle Elemente einer Liste durchlaufen,
- die Fehlermeldung `IndexError` erklären und den Fehler beheben.

1.11.1 Warum Listen?

Speicherst du fünf Noten in fünf Variablen `note1` bis `note5`, hat das zwei Schwächen. Kommt eine sechste Note dazu, brauchst du eine neue Variable **und** musst jede Rechnung ändern, die alle Noten benutzt. Außerdem kannst du keine Schleife über `note1` bis `note5` laufen lassen: Für Python sind das fünf völlig unabhängige Namen.

Mit einer **Liste** speicherst du alle Noten in **einer** Variablen:

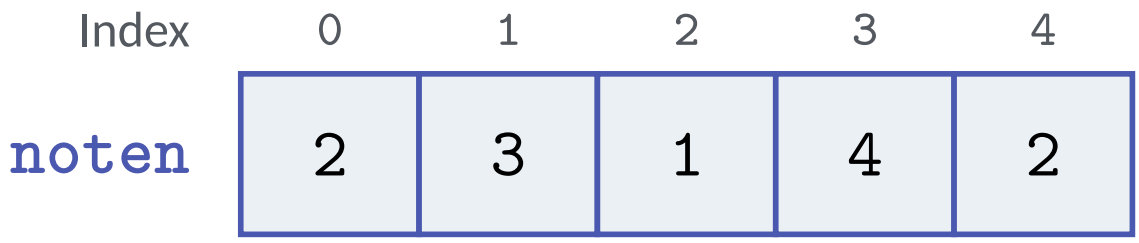
```
1 noten = [2, 3, 1, 4, 2]
2 print(noten)
3 print(type(noten))
```

AUSGABE

```
[2, 3, 1, 4, 2]
<class 'list'>
```

- **Zeile 1:** Eine Liste schreibt man in **eckigen Klammern** `[ ]`. Die einzelnen Werte werden durch Kommas getrennt. Die ganze Liste wird in der Variablen `noten` gespeichert.
- **Zeile 2:** `print` gibt die Liste mit ihren Klammern und Kommas aus.
- **Zeile 3:** Der Datentyp einer Liste heißt `list` (englisch für „Liste“).

Die Werte in einer Liste heißen **Elemente**. Sie stehen in einer festen Reihenfolge hintereinander, wie die Waggons eines Zuges: Es gibt ein erstes und ein letztes Element, und jedes andere Element hat genau einen Vorgänger und einen Nachfolger. Man sagt deshalb: Eine Liste ist eine **lineare** Datenstruktur.



Die Liste `noten` mit fünf Elementen. Über jedem Element steht seine Position, der Index

Definition: Liste bzw. Feld

Eine **Liste** (auch **Feld** genannt) ist eine Datenstruktur, die mehrere Werte – die **Elemente** – in einer festen Reihenfolge zu einer Einheit zusammenfasst. In Python schreibt man sie in eckigen Klammern, z. B. `[2, 3, 1, 4, 2]`.

Liste oder Feld?

In Python heißt diese Datenstruktur **Liste** (`list`). In anderen Programmiersprachen wie Java oder C begegnet dir dafür das Wort **Feld** (englisch *array*). Ein Feld hat dort meist eine feste Länge. Eine Python-Liste kann dagegen wachsen und schrumpfen. Die Grundidee ist dieselbe: viele Elemente in einer Reihe, durchnummeriert ab 0.

Weitere Listen

In einer Liste können auch Zeichenketten oder Kommazahlen stehen, z. B. `[1.99, 0.49, 2.5]`. Eine Liste kann sogar leer sein:

```
1 staedte = ["Saarbrücken", "Trier", "Mainz"]
2 leer = []
3 print(staedte)
4 print(leer)
```

AUSGABE

```
['Saarbrücken', 'Trier', 'Mainz']
[]
```

Beachte die Ausgabe in der ersten Zeile: Python zeigt Zeichenketten **in einer Liste** mit einfachen Anführungszeichen `'…'` an. Das ist nur die Art der Anzeige. Im Programm darfst du `"…"` oder `'…'` schreiben. In diesem Skript verwenden wir immer `"…"`.

Die leere Liste `[]` in Zeile 2 sieht nutzlos aus. Sie ist aber praktisch: Man legt sie an und füllt sie später nach und nach.

Merke

Eine **Liste** (in Python: `list`) fasst mehrere **Elemente zu einer Einheit** zusammen, z. B. `noten = [2, 3, 1, 4, 2]`.

1.11.2 Index und Element

Denk an die Schließfächer in deiner Schule. Jedes Fach hat eine **Nummer** an der Tür, und in jedem Fach liegt ein **Inhalt**. Wer „Fach 2“ sagt, meint die Nummer – nicht das, was darin liegt.

Bei einer Liste ist es genauso. Jedes Element hat eine Positionsnummer, den **Index**. Wie bei Zeichenketten beginnt die Zählung bei **0**: Das erste Element hat den Index 0, das zweite den Index 1 und so weiter.

Definition: Index und Element

Der **Index** ist die Positionsnummer eines Elements in einer Liste. Er beginnt bei 0. Das **Element** (Listenelement) ist der Wert, der an dieser Position gespeichert ist.

Lesender Zugriff

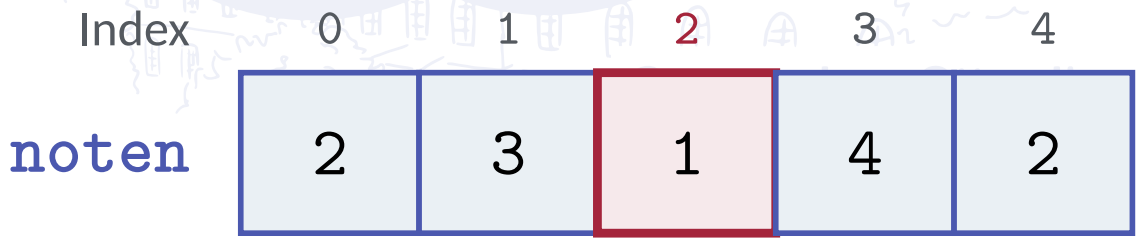
Um ein einzelnes Element zu lesen, schreibst du den Index in eckigen Klammern hinter den Namen der Liste – genau wie bei Zeichenketten:

```
1 noten = [2, 3, 1, 4, 2]
2 print(noten[0])
3 print(noten[2])
4 erste = noten[0]
5 print("Die erste Note ist", erste)
```

AUSGABE

```
2
1
Die erste Note ist 2
```

- **Zeile 2:** `noten[0]` liefert das Element mit dem Index 0. Das ist das erste Element, die `2`.
- **Zeile 3:** `noten[2]` liefert das Element mit dem Index 2. Das ist die `1` – also das **dritte** Element. Die Zahl in den eckigen Klammern ist der Index, nicht der Wert!
- **Zeilen 4–5:** Ein gelesenes Element kann man wie jeden anderen Wert in einer Variablen speichern.



Der Ausdruck `noten\[2\]` liefert das Element am Index 2. Das ist die 1, nicht die 2

Gerade bei Listen mit kleinen Zahlen verwechselt man Index und Element leicht. In der Liste `noten` kommt das Element `2` zweimal vor: am Index 0 und am Index 4. Am Index 2 steht dagegen das Element `1`.

Die Länge einer Liste

Wie viele Elemente eine Liste hat, verrät dir `len(...)` – wie bei Zeichenketten:

```
1 noten = [2, 3, 1, 4, 2]
2 print(len(noten))
3 print(noten[len(noten) - 1])
```

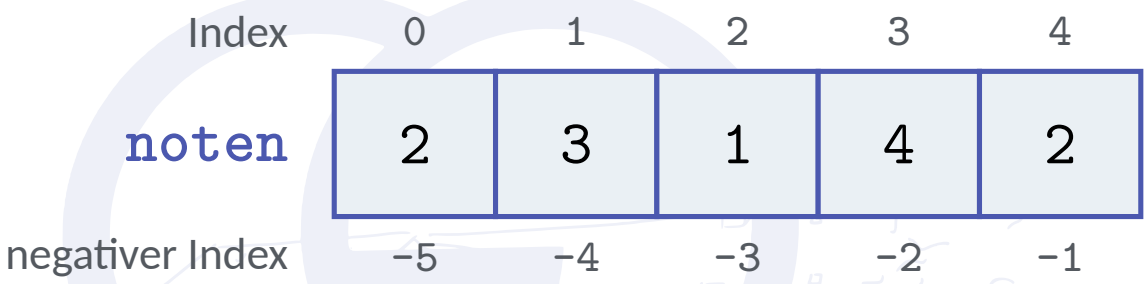
AUSGABE

```
5
2
```

Die Liste hat fünf Elemente. Weil die Zählung bei 0 beginnt, hat das letzte Element aber den Index **4**, also `len(noten) - 1`. Zeile 3 liest deshalb das letzte Element. Das funktioniert für jede Liste, egal wie lang sie ist.

Negative Indizes

Von den Zeichenketten kennst du schon eine Abkürzung: Ein **negativer Index** zählt von hinten. `noten[-1]` ist das letzte Element (die 2), `noten[-2]` das vorletzte (die 4).



Unter jedem Element steht der negative Index. Er zählt von hinten, beginnend bei -1

Merke

- Jedes Element besitzt einen **Index**, beginnend bei `0`. Der Index ist **nicht** dasselbe wie der gespeicherte Wert!
- **Lesender Zugriff:** `liste[index]` liefert das Element an dieser Position.
- Die Anzahl der Elemente erhält man mit `len(liste)`. Das letzte Element hat den Index `len(liste) - 1` (oder kurz `-1`).

Arbeitsauftrag 131: Index oder Element?

Gegeben ist die Liste `zahlen = [3, 0, 4, 1, 2]`. Bestimme **ohne Computer**, welchen Wert die Ausdrücke haben. Prüfe danach mit `print(...)` am Computer.

`zahlen[1]`	`len(zahlen)`	`zahlen[-1]`	`zahlen[zahlen[0]]`

Gib außerdem an: An welchem Index steht das Element `4`?

Das Element 4 steht am Index:

1.11.3 Schreibender Zugriff: ein Element ändern

Ein Schließfach kann man ausräumen und neu füllen – die Nummer bleibt gleich. Genauso kannst du ein Listenelement **überschreiben**. Der Zugriff mit Index steht dann **links** vom =:

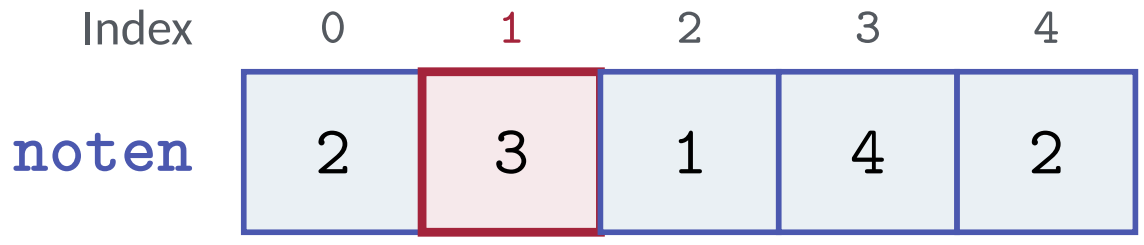
```
1 noten = [2, 3, 1, 4, 2]
2 noten[1] = 5
3 print(noten)
4 noten[2] = noten[2] + 1
5 print(noten)
```

AUSGABE

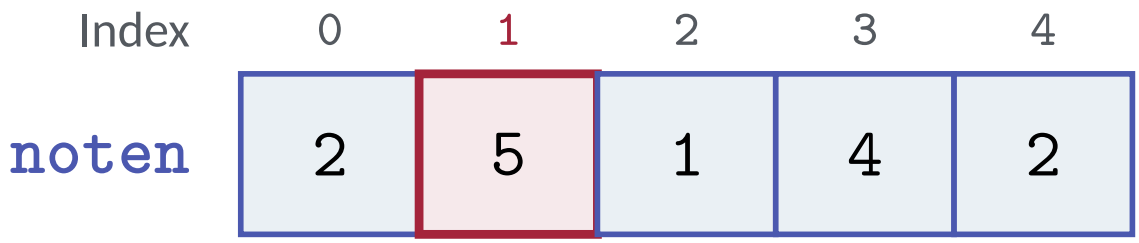
```
[2, 5, 1, 4, 2]
[2, 5, 2, 4, 2]
```

- **Zeile 2:** Das Element am Index 1 bekommt den Wert 5. Der alte Wert 3 ist weg. Das funktioniert wie eine Zuweisung an eine Variable – nur eben an **eine Stelle** der Liste.
- **Zeile 4:** Wie bei `x = x + 1` wird zuerst die rechte Seite ausgerechnet: `noten[2]` ist 1, plus 1 ergibt 2. Dieser Wert wird dann am Index 2 gespeichert.

Die Länge der Liste ändert sich dabei nicht. Es werden nur Inhalte ausgetauscht.



Vorher: Am Index 1 steht die 3



Nachher (nach `noten\[1\] = 5`): Am Index 1 steht die 5. Alle anderen Elemente sind unverändert

Merke

Schreibender Zugriff: `liste[index] = neuer_wert` überschreibt das Element an dieser Position.

Stolperstein

Bei **Zeichenketten** geht das nicht. Eine Zeichenkette kann man nur lesen, aber nicht stellenweise ändern:

```
wort = "Hallo"
wort[0] = "B"
```

FEHLERMELDUNG

```
TypeError: 'str' object does not support item assignment
```

Die Meldung heißt: „Ein `str`-Objekt unterstützt keine Zuweisung an einzelne Stellen.“ Listen sind dagegen **veränderbar**. Das ist ein wichtiger Unterschied zwischen den beiden Datenstrukturen.

1.11.4 Ungültige Position: der `IndexError`

Was passiert, wenn du auf einen Index zugreifst, den es gar nicht gibt?

```
noten = [2, 3, 1, 4, 2]
print(noten[5])
```

FEHLERMELDUNG

```
Traceback (most recent call last):
  File "noten.py", line 2, in <module>
    print(noten[5])
          ~~~~~^^^
IndexError: list index out of range
```

Lies die Meldung wieder von unten nach oben: **IndexError** heißt „Fehler beim Index“. *list index out of range* bedeutet: „Der Listenindex liegt außerhalb des gültigen Bereichs.“ Die Zeilen darüber zeigen, dass der Fehler in Zeile 2 passiert ist, und die Zeichen `^^^` zeigen auf `[5]`.

Die Liste hat fünf Elemente. Gültig sind also nur die Indizes 0 bis 4 (und von hinten −1 bis −5). Einen Index 5 gibt es nicht.

Auch beim **Schreiben** an eine Stelle, die es nicht gibt, meldet Python einen Fehler. Eine Liste wird durch eine Zuweisung nicht automatisch länger:

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

FEHLERMELDUNG

```
IndexError: list assignment index out of range
```

list assignment index out of range heißt: „Der Index für die Zuweisung liegt außerhalb des gültigen Bereichs." Um ein Element **hinzuzufügen**, brauchst du `append` aus dem nächsten Abschnitt.

Merke

Ein Zugriff auf einen ungültigen Index führt zu einer Fehlermeldung (`IndexError`). Gültig sind bei einer Liste mit `n` Elementen nur die Indizes `0` bis `n - 1`.

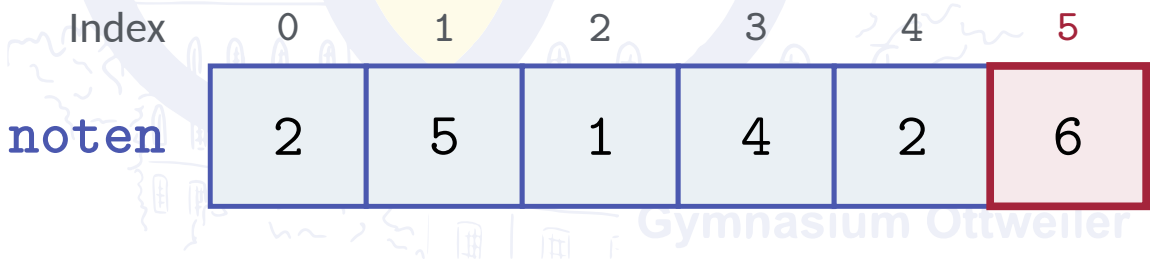
Die häufigste Ursache für einen `IndexError`: Man zählt aus Gewohnheit ab 1 und greift mit `liste[len(liste)]` auf das „letzte" Element zu. Der größte gültige Index ist aber `len(liste) - 1`. Prüfe bei einem `IndexError` immer: Wie lang ist die Liste gerade, und welchen Index habe ich benutzt?

1.11.5 Listen verändern: Elemente hinzufügen und entfernen

Eine Liste kann wachsen und schrumpfen. Dafür gibt es eigene **Methoden**, die man mit einem Punkt an den Namen der Liste hängt: `liste.methode(...)`. Methoden kennst du schon von den Zeichenketten, zum Beispiel `text.upper()`.

Ein Element anhängen: append

`append` heißt „anhängen". Die Methode fügt ein neues Element **am Ende** der Liste an. Nach `noten.append(6)` ist die Liste `noten` um ein Element länger. Das neue Element hat den Index 5:



Nach `noten.append(6)`: Das neue Element steht am Ende, am Index 5

Merke

Mit `liste.append(wert)` wird ein neues Element ans Ende der Liste angehängt.

Das Beispiel im Ganzen

Das folgende Programm fasst alles zusammen, was du bis jetzt über Listen gelernt hast – auch `append`. Überlege zuerst, was es ausgibt. Lies dann weiter.

```
1 noten = [2, 3, 1, 4, 2]
2
3 print(noten)
4 print("Anzahl Noten:", len(noten))
5 print("Erste Note (Index 0):", noten[0])
6 print("Dritte Note (Index 2):", noten[2])
7
8 noten[1] = 5
9 print("Nach Änderung:", noten)
10
11 noten.append(6)
12 print("Nach append:", noten)
```

AUSGABE

```
[2, 3, 1, 4, 2]
Anzahl Noten: 5
Erste Note (Index 0): 2
Dritte Note (Index 2): 1
Nach Änderung: [2, 5, 1, 4, 2]
Nach append: [2, 5, 1, 4, 2, 6]
```

- **Zeilen 3–6:** Lesender Zugriff: die ganze Liste, ihre Länge, das Element am Index 0 und das Element am Index 2. Beachte: Die **dritte** Note hat den Index **2**.
- **Zeile 8:** Schreibender Zugriff: Das Element am Index 1 (die 3) wird durch 5 ersetzt.
- **Zeile 11:** Die Note 6 wird ans Ende angehängt. Die Liste hat danach sechs Elemente.

Ein Element einfügen: `insert`

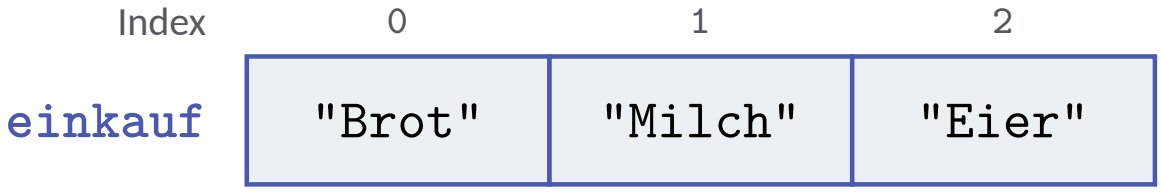
Mit `liste.insert(index, wert)` fügst du ein Element an einer **beliebigen Stelle** ein. Der erste Wert in den Klammern ist der Index, an dem das neue Element stehen soll, der zweite ist das Element selbst:

```
1 einkauf = ["Brot", "Milch", "Eier"]
2 einkauf.insert(1, "Käse")
3 print(einkauf)
```

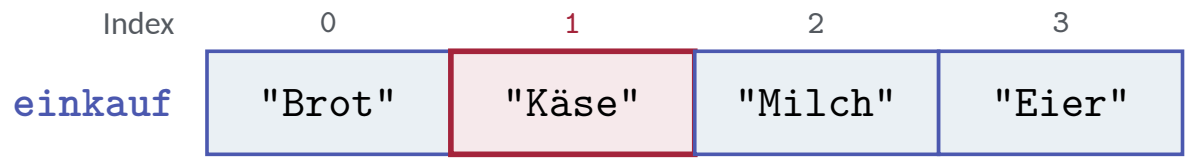
AUSGABE

```
['Brot', 'Käse', 'Milch', 'Eier']
```

Der Käse steht jetzt am Index 1. Alle Elemente, die vorher ab Index 1 standen, rücken eine Stelle nach rechts. Ihr Index wird dadurch um 1 größer: Die Milch hatte vorher den Index 1, jetzt hat sie den Index 2.



Vorher: drei Elemente mit den Indizes 0 bis 2



Nachher (nach `einkauf.insert(1, "Käse")`): Milch und Eier sind eine Stelle nach rechts gerückt

Elemente entfernen: `pop` und `remove`

Zum Entfernen gibt es zwei Methoden. Sie unterscheiden sich darin, was du in die Klammern schreibst: einen **Index** oder ein **Element**.

`liste.pop(index)` entfernt das Element **am angegebenen Index**. Ohne Index entfernt `pop()` das **letzte** Element. In beiden Fällen gibt `pop` das entfernte Element zurück – so wie eine Funktion mit `return` einen Wert zurückgibt. Du kannst es also in einer Variablen auffangen:

```
1 einkauf = ["Brot", "Käse", "Milch", "Eier"]
2 letztes = einkauf.pop()
3 print(letztes)
4 print(einkauf)
5 erstes = einkauf.pop(0)
6 print(erstes)
7 print(einkauf)
```

AUSGABE

```
Eier
['Brot', 'Käse', 'Milch']
Brot
['Käse', 'Milch']
```

- **Zeile 2:** `pop()` ohne Index entfernt das letzte Element (`"Eier"`) und gibt es zurück. Es wird in `letztes` gespeichert.
- **Zeile 5:** `pop(0)` entfernt das Element am Index 0 (`"Brot"`). Die übrigen Elemente rücken nach links. `"Käse"` hat jetzt den Index 0.

`liste.remove(wert)` sucht dagegen nach einem **Element mit diesem Wert** und entfernt es. Kommt der Wert mehrmals vor, wird nur das **erste** Vorkommen entfernt:

```
1 einkauf = ["Brot", "Käse", "Milch", "Käse"]
2 einkauf.remove("Käse")
3 print(einkauf)
```

AUSGABE

```
['Brot', 'Milch', 'Käse']
```

Die Tabelle fasst die Methoden zusammen:

Methode	Wirkung	in den Klammern steht
`liste.append(wert)`	hängt `wert` ans Ende an	ein Element
`liste.insert(i, wert)`	fügt `wert` am Index `i` ein	ein Index und ein Element
`liste.pop()`	entfernt das letzte Element und gibt es zurück	nichts
`liste.pop(i)`	entfernt das Element am Index `i` und gibt es zurück	ein Index
`liste.remove(wert)`	entfernt das erste Element mit dem Wert `wert`	ein Element

Merke dir den Unterschied: `pop` erwartet einen **Index**, `remove` ein **Element**.

Stolperstein

Ist der Wert nicht in der Liste, kann `remove` nichts entfernen und meldet einen **ValueError** („falscher Wert“):

```
einkauf = ["Brot", "Milch"]
einkauf.remove("Käse")
```

FEHLERMELDUNG

```
ValueError: list.remove(x): x not in list
```

Die Meldung heißt: „`x` ist nicht in der Liste.“ Ähnlich bei `pop`: Aus einer leeren Liste kann nichts entfernt werden. Dann meldet Python `IndexError: pop from empty list`.

So kannst du es sagen

So beschreibst du Listenoperationen in Fachsprache – vom einfachen zum genauen Satz:

- *„Die 5 ist an Stelle 1.“ → „Das Listenelement am Index 1 hat den Wert 5.“*
- *„Ich mach die 3 zur 5.“ → „Das Listenelement an der Stelle 1 wird von 3 auf 5 verändert.“*
- *„Die 6 kommt hinten dran.“ → „Mit `append` wird das Element 6 am Ende der Liste hinzugefügt.“*
- *„Da kommt ein Fehler.“ → „Beim Zugriff auf die ungültige Listenposition 5 erhalte ich die Fehlermeldung `IndexError`.“*

Arbeitsauftrag 132: Listen-Vorhersage

Lies das Programm, **ohne** es auszuführen. Zeichne die Liste `tiere` nach jeder Zeile als Reihe von Kästchen und schreibe den Index darüber. Gib dann an, was das Programm ausgibt. Überprüfe deine Vorhersage am Computer.

```
1 tiere = ["Hund", "Katze", "Maus"]
2 tiere.append("Igel")
3 tiere[0] = "Pferd"
4 tiere.insert(2, "Esel")
5 tiere.remove("Katze")
6 x = tiere.pop()
7 print(tiere)
8 print(x)
9 print(len(tiere))
```

Die Liste nach jeder Zeile und die Ausgabe:

1.11.6 Listen mit Schleifen durchlaufen

Oft soll ein Programm mit **jedem** Element einer Liste etwas tun. Dafür gibt es zwei Arten von `for`-Schleifen.

Schleife über die Elemente

Über die Zeichen einer Zeichenkette läufst du mit `for zeichen in text:`. Über die Elemente einer Liste geht es genauso:

```
1 staedte = ["Saarbrücken", "Trier", "Mainz", "Koblenz"]
2 for stadt in staedte:
3     print("Ich fahre nach", stadt)
```

AUSGABE

```
Ich fahre nach Saarbrücken
Ich fahre nach Trier
Ich fahre nach Mainz
Ich fahre nach Koblenz
```

- **Zeile 2:** Die Schleifenvariable `stadt` bekommt nacheinander jedes Element der Liste, von vorne nach hinten.
- **Zeile 3:** Der eingerückte Schleifenrumpf wird für jedes Element einmal ausgeführt – hier also viermal.

Schleife über die Indizes

Manchmal brauchst du nicht nur das Element, sondern auch seinen **Index**. Dann läufst du mit `range(len(liste))` über alle gültigen Indizes:

```
1 staedte = ["Saarbrücken", "Trier", "Mainz", "Koblenz"]
2 for i in range(len(staedte)):
3     print(i, staedte[i])
```

AUSGABE

```
0 Saarbrücken
1 Trier
2 Mainz
3 Koblenz
```

- **Zeile 2:** `len(staedte)` ist 4. `range(4)` erzeugt die Zahlen 0, 1, 2, 3 – genau die gültigen Indizes der Liste. Die Schleifenvariable `i` ist also ein **Index**.
- **Zeile 3:** Mit `staedte[i]` wird das Element am Index `i` gelesen.

Die Schleife über die Indizes brauchst du immer dann, wenn du die Position kennen musst – zum Beispiel, um Elemente zu verändern. Mehr dazu im nächsten Abschnitt.

Versuch: Fehler finden

Ändere die Schleife zu `for i in range(1, len(staedte) + 1):`. Was gibt das Programm jetzt aus, bevor es abbricht? Welche Fehlermeldung erhältst du? Erkläre: Welche Indizes erzeugt `range` jetzt, und welcher davon ist ungültig?

Eine Liste mit einer Schleife füllen

Ein sehr häufiges Muster: Man beginnt mit einer **leeren Liste** und hängt in einer Schleife ein Element nach dem anderen an.

```
1 quadratzahlen = []
2 for zahl in range(1, 6):
3     quadratzahlen.append(zahl * zahl)
4 print(quadratzahlen)
```

AUSGABE

```
[1, 4, 9, 16, 25]
```

- **Zeile 1:** Die Liste ist zuerst leer.
- **Zeilen 2–3:** Für jede Zahl von 1 bis 5 wird ihr Quadrat ans Ende der Liste angehängt.
- **Zeile 4:** Erst nach der Schleife wird die fertige Liste ausgegeben. Stünde `print` eingerückt in der Schleife, würde die Liste fünfmal ausgegeben – jedes Mal ein Stück länger.

1.11.7 Übungen

Arbeitsauftrag 133: Einkaufsliste

Lege eine leere Liste an. Frage nacheinander 5 Artikelnamen ab (mit `input()` in einer Schleife) und füge sie mit `append()` zur Liste hinzu. Gib die vollständige Liste am Ende aus.

Werkzeug

Mit `liste.index(wert)` erfährst du, an welchem **Index** ein Element steht. Kommt der Wert mehrmals vor, erhältst du den Index des ersten Vorkommens. Ist der Wert gar nicht in der Liste, meldet Python einen Fehler, z. B. `ValueError: 'Lea' is not in list`. Im nächsten Abschnitt programmierst du so eine Suche selbst (**lineare Suche**).

Arbeitsauftrag 134: Index vs. Wert

Gegeben ist die Liste `staedte = ["Saarbrücken", "Trier", "Mainz", "Koblenz"]`.

1. Gib den Wert am Index 2 aus.
2. Ändere den Wert am Index 0 zu `"Saarlouis"` und gib die Liste aus.
3. Gib den **Index** der Stadt `"Mainz"` aus (Tipp: `liste.index(wert)`).

Tipp

Zur Erinnerung: Mit einer negativen Schrittweite zählt `range` rückwärts: `range(4, -1, -1)` erzeugt 4, 3, 2, 1, 0. Die Zahl in der Mitte (`-1`) ist wie immer die Grenze, die **nicht** mehr erreicht wird.

Arbeitsauftrag 135 (Zusatz): Liste umkehren

Gegeben ist die Liste `zahlen = [1, 2, 3, 4, 5]`. Erzeuge mit einer Schleife eine **neue** Liste, die die Elemente in umgekehrter Reihenfolge enthält, und gib sie aus. Verwende dafür keine eingebaute Funktion zum Umkehren.

1.11.8 Weitere Übungen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Arbeitsauftrag 136: Wochentage ●○○

Gegeben ist die Liste `tage = ["Mo", "Di", "Mi", "Do", "Fr"]`. Bestimme **ohne Computer** die Werte der Ausdrücke. Prüfe danach mit `print(...)` am Computer.

`tage[1]`	`tage[-1]`	`len(tage)`	`tage[len(tage) - 3]`

Gib außerdem an, an welchem Index das Element `"Do"` steht.

"Do" steht am Index:

Arbeitsauftrag 137: Liste verändern ●○○

Gib **ohne Computer** an, was das Programm ausgibt. Überprüfe danach am Computer.

```
1 punkte = [12, 7, 9]
2 punkte[0] = 10
3 punkte.append(15)
4 punkte[1] = punkte[1] + 3
5 print(punkte)
6 print(len(punkte))
```

Meine Vorhersage:

Arbeitsauftrag 138: Stundenplan ●○○

Am Montag hast du die Fächer Mathe, Deutsch, Informatik und Sport.

1. Lege eine Liste `montag` mit diesen vier Fächern an.
2. Gib das erste und das letzte Fach aus.
3. Sport fällt aus, stattdessen ist Kunst. Ändere das Element mit einem schreibenden Zugriff.
4. Nach der Schule ist noch die Theater-AG. Hänge `"Theater-AG"` an die Liste an.
5. Gib die Liste und die Anzahl der Einträge aus.

Arbeitsauftrag 139: IndexError erklären ●○○

Das Programm soll die letzte Farbe der Liste ausgeben. Es bricht aber ab.

```
farben = ["rot", "gelb", "grün"]
print(farben[3])
```

1. Gib an, welche Indizes in dieser Liste gültig sind.
2. Erkläre, warum die Fehlermeldung `IndexError: list index out of range` erscheint.
3. Nenne zwei Möglichkeiten, die letzte Farbe richtig auszugeben.

Meine Antworten:

Arbeitsauftrag 140: Preiserhöhung am Kiosk ●●○

Am Schulkiosk werden alle Preise um 50 Cent teurer. Die Preise stehen in der Liste `preise = [1.5, 2.0, 0.8, 3.0]` (in Euro).

Schreibe ein Programm, das mit einer Schleife **jedes Element der Liste** um 0.5 erhöht. Gib die Liste danach aus.

Tipp

Du musst die Elemente **in der Liste** verändern. Dafür brauchst du die Schleife über die Indizes: `for i in range(len(preise)):`. Im Rumpf steht dann ein schreibender Zugriff wie im Auftrag *Liste verändern*.

Arbeitsauftrag 141: Siegerliste ●●○

Die Liste `sieger = ["Lea", "Tom", "Ada", "Ben"]` enthält die Namen nach der Platzierung beim Sportfest. Schreibe ein Programm, das mit einer Schleife diese Ausgabe erzeugt:

AUSGABE

```
Platz 1 - Lea
Platz 2 - Tom
Platz 3 - Ada
Platz 4 - Ben
```

Arbeitsauftrag 142: Zwei Mannschaften ●●○

Beim Fußball im Sportunterricht stehen alle in einer Reihe: `reihe = ["Ada", "Ben", "Cem", "Dana", "Eli", "Finn"]`. Wer an einem geraden Index steht (0, 2, 4), kommt in Mannschaft A, die anderen in Mannschaft B.

Schreibe ein Programm, das zwei neue Listen `team_a` und `team_b` mit `append` füllt und beide ausgibt.

Tipp

Denk an die Schrittweite von `range` aus dem Abschnitt über die for-Schleife: `range(0, 6, 2)` erzeugt 0, 2, 4. Statt 6 schreibst du besser `len(reihe)`.

Arbeitsauftrag 143: Plätze tauschen ●●●

Im Sitzplan `sitzplan = ["Ben", "Ada", "Tom", "Mia", "Lea"]` wollen Tom und Lea die Plätze tauschen.

1. Bestimme mit `.index()` die Indizes von Tom und Lea.
2. Tausche die beiden Elemente mit schreibenden Zugriffen. Gib den Sitzplan vor und nach dem Tausch aus.
3. Erkläre, warum du dafür eine **Hilfsvariable** brauchst.

Tipp

Stell dir zwei Gläser vor: eins mit Saft, eins mit Wasser. Um die Inhalte zu tauschen, brauchst du ein drittes, leeres Glas. Im Programm ist das dritte Glas eine Hilfsvariable, z. B. `merke = sitzplan[i]`.

Warum braucht man die Hilfsvariable?

1.11.9 Das Wichtigste in Kürze

- Eine **Liste** (ein **Feld**) fasst mehrere **Elemente** in einer festen Reihenfolge zu einer Einheit zusammen: `noten = [2, 3, 1, 4, 2]`.
- Jedes Element hat einen **Index**, beginnend bei 0. Index und Element sind verschiedene Dinge: `noten[2]` ist das Element am Index 2.
- **Lesen:** `liste[i]`. **Schreiben:** `liste[i] = wert`. Negative Indizes zählen von hinten: `liste[-1]` ist das letzte Element.
- `len(liste)` liefert die Anzahl der Elemente. Gültige Indizes sind `0` bis `len(liste) - 1`. Ein ungültiger Index führt zu einem `IndexError`.
- `append` hängt an, `insert` fügt ein, `pop` entfernt am Index, `remove` entfernt ein Element mit einem bestimmten Wert.
- `for element in liste:` durchläuft die Elemente, `for i in range(len(liste)):` durchläuft die Indizes.

Selbstcheck

- ☐ Ich kann erklären, was eine Liste ist und warum sie praktischer ist als viele einzelne Variablen.
- ☐ Ich kann bei einer Liste zwischen Index und Element unterscheiden.
- ☐ Ich kann ein Element über seinen Index lesen und überschreiben.
- ☐ Ich kann Elemente mit `append` und `insert` hinzufügen und mit `pop` und `remove` entfernen.
- ☐ Ich kann eine Liste mit einer Schleife durchlaufen und mit `append` in einer Schleife füllen.
- ☐ Ich kann erklären, wann ein `IndexError` auftritt, und den Fehler beheben.

Gymnasium Ottweiler