3 Listen von Dateien

Funktionen, Module und Dateien

Ihr Code kann nicht ewig in einem Notebook leben. Er will frei sein.

Und wenn es darum geht, Ihren Code zu befreien und mit anderen zu **teilen**, dann ist eine selbst erstellte **Funktion** der erste Schritt, auf den kurz darauf ein **Modul** folgt, mit dem Sie Ihren Code organisieren und weitergeben können. In diesem Kapitel werden Sie aus dem bisher geschriebenen Code direkt eine Funktion und auf dem Weg auch gleich ein **gemeinsam nutzbares** Modul erstellen. Ihr Modul wird sich sofort an die Arbeit machen, während Sie **for**-Schleifen, **if**-Anweisungen, Tests auf bestimmte **Bedingungen** sowie die Python-Standardbibliothek, **PSL** (*Python Standard Library*), verwenden, um die Schwimmdaten des Coachs zu verarbeiten. Außerdem werden Sie lernen, Ihre Funktionen zu **kommentieren** (was *immer* eine gute Idee ist). Es gibt viel zu tun, also an die Arbeit!

Bürogespräch

Sam: Ich habe den Coach über die aktuellen Fortschritte informiert.

Alex: Und? Ist er zufrieden?

Sam: Irgendwie schon. Er ist begeistert vom Anfang, aber wie Ihr euch vorstellen könnt, interessiert er sich eigentlich nur für das Endergebnis, nämlich das Balkendiagramm.

Alex: Und das sollte ja nicht so schwer sein, nachdem unser aktuellstes Notebook die nötigen Daten erzeugt, oder?

Mara: Zumindest ungefähr.

Alex: Wieso? Was stimmt denn nicht?

Mara: Das aktuelle Notebook, *Times.ipynb,* erzeugt Daten für Darius, der die 100 Meter Butterfly in der Altersgruppe der unter 13-Jährigen schwimmt. Wir müssen die Umwandlungen und die Durchschnittsberechnungen aber für die Dateien *aller* Schwimmer durchführen.

Alex: Das kann doch nicht so schwer sein: einfach den Dateinamen am Anfang des Notebooks durch einen anderen ersetzen, dann den *Run All*-Button drücken – und zack!, schon haben wir die Daten.

Mara: Glaubst du wirklich, der Coach hat da große Lust drauf?

Alex: Ähh … ich habe ganz vergessen, dass der Coach das ja alles selbst tun muss.

Sam: Wir sind aber auf dem richtigen Weg. Wir brauchen eine Möglichkeit, die Dateinamen aller Schwimmer zu verarbeiten. Wenn wir das hinkriegen, können wir mit dem Code für das Balkendiagramm weitermachen.

Alex: Da haben wir aber noch einiges vor uns …

Mara: Ja, aber der Weg ist nicht so weit. Wie gesagt, der gesamte nötige Code ist schon im *Times.ipynb*-Notebook enthalten.

Alex: … das du dem Coach nicht geben willst …

Mara: … jedenfalls nicht in seiner jetzigen Form.

Alex: Aber wie dann?

Sam: Wir müssten den Code so verpacken, dass er mit beliebigen Dateinamen und auch ohne Notebook funktioniert.

Alex: Ah, ja klar! Wir brauchen eine Funktion!

Sam: … die uns immerhin ein Stück weiterbringt.

Mara: Wenn sich die Funktion in einem Python-Modul befindet, kann sie an vielen Orten weiterverwendet werden.

Alex: Das klingt doch gut. Womit sollen wir anfangen?

Mara: Am besten wandeln wir den bisherigen Code im Notebook in eine Funktion um, die wir aufrufen und weitergeben können.

Sie haben den nötigen Code schon fast beisammen

Im Moment befindet er sich aber noch in Ihrem *Times.ipynb*-Notebook.

Wenn es darum geht, zu *experimentieren* und Code von Grund auf neu zu *erstellen*, sind Jupyter Notebooks kaum zu schlagen. Soll dagegen vorhandener Code *wiederverwendet* und *weitergegeben werden*, sind Notebooks nicht unbedingt die beste Wahl (und um ehrlich zu sein, wurden sie dafür auch nicht entwickelt).

Ein guter Einsatzzweck bestünde darin, eine *Kopie* Ihres Notebooks an jemanden weiterzugeben, der es dann in seiner eigenen Jupyter-Umgebung ausführen kann. Aber stellen Sie sich vor, Sie bauten eine Applikation, die einen Teil des Codes aus Ihrem Notebook verwenden muss …

Im Anhang dieses Buchs stellen wir eine Jupyter-Erweiterung vor, die Ihnen bei dieser Anforderung helfen kann. Ohne Weiteres ist die Weitergabe des Codes im Notebook tatsächlich nicht ganz einfach.

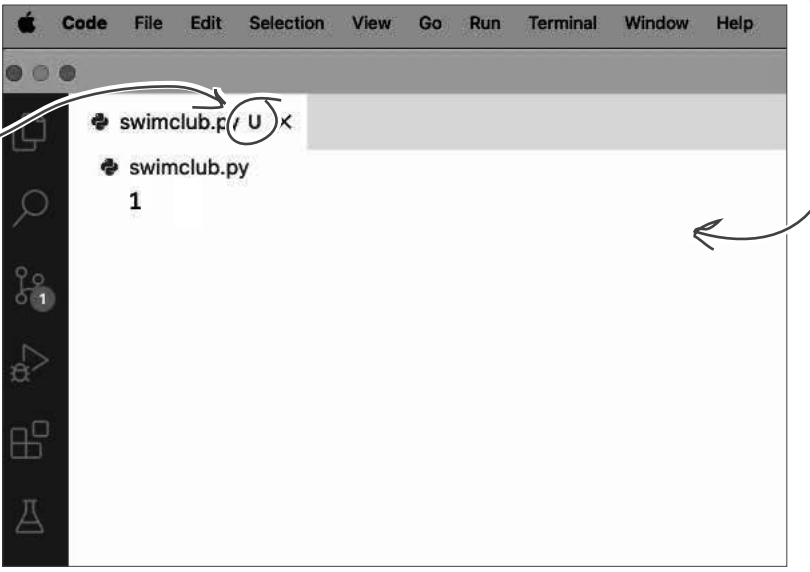
Wie können Sie *diesen* Code mit anderen teilen?

Um den Code Ihres Notebooks weiterzugeben, müssen Sie eine **Funktion** erstellen, die Ihren Code enthält. Danach können Sie diese Funktion in einem **Modul** verpacken und dieses weitergeben. Beides werden wir in diesem Kapitel tun.

Für den Anfang erstellen Sie eine neue leere Datei in Ihrem *Learning*-Ordner und nennen Sie *swimclub.py*.

Ihr Bildschirm sieht möglicherweise anders aus als in dieser Abbildung. Erstens zeigen wir hier VS Code auf einem Mac (aber unter Windows oder Linux sollte es ähnlich aussehen). Zweitens hat VS Code bemerkt, dass wir Git für die Codeverwaltung benutzen. Daher informiert uns das GUI darüber, dass es eine neue Datei gibt, die noch nicht versioniert ist.

Wir werden Git in diesem Buch nicht weiter behandeln, wollten Sie aber nicht irritieren, wenn Ihr Bildschirm sich von unserem unterscheidet. Wenn Sie >>swimclub.py<< in Ihrem >>Learning<<-Ordner erstellt haben und VS Code darauf wartet, dass Sie den leeren Bildschirm mit etwas Code füllen, dann sind Sie schon startklar.



Eine Funktion in Python erstellen

Neben dem eigentlichen Code für die Funktion müssen Sie sich auch Gedanken über die *Signatur* der Funktion machen. Hierbei gibt es drei Dinge zu beachten:

1. **Überlegen Sie sich einen schönen, aussagekräftigen Namen.**
Der Code im *Times.ipynb*-Notebook verarbeitet zuerst den Dateinamen und liest dann den Inhalt der Datei, um die vom Coach benötigten Daten zu extrahieren. Daher wollen wir die Funktion `read_swim_data` (Schwimmdaten_lesen) nennen. Ein schöner Name, ein aussagekräftiger Name … Donnerwetter, er ist fast schon perfekt!

2. **Entscheiden Sie, welche Anzahl und welche Namen mögliche Parameter haben sollen.**
Ihre Funktion `read_swim_data` übernimmt einen Parameter, der angibt, welcher Dateiname verwendet werden soll. Nennen wir ihn `filename` (Dateiname).

3. **Rücken Sie den Code der Funktion unterhalb einer def-Anweisung ein.**
Das Schlüsselwort **def** leitet die Funktion ein. Hier können Sie ihren Namen und mögliche Parameter angeben. Sämtlicher Code, der unterhalb der **def**-Zeile eingerückt ist, wird als Codeblock der Funktion verwendet.

Es kann helfen, sich >>def<< als Abkürzung für >>Definiere eine Funktion<< vorzustellen.

Anatomie einer Funktionssignatur

1 Einen schönen, aussagekräftigen Namen verwenden.

Dieser Name gibt den Nutzerinnen und Nutzern Ihrer Funktion einen guten Hinweis darauf, was sie tut.

2 Alle Parameter benennen.

Hier gibt es nur einen einzigen Parameter.

```
def read_swim_data(filename):
```

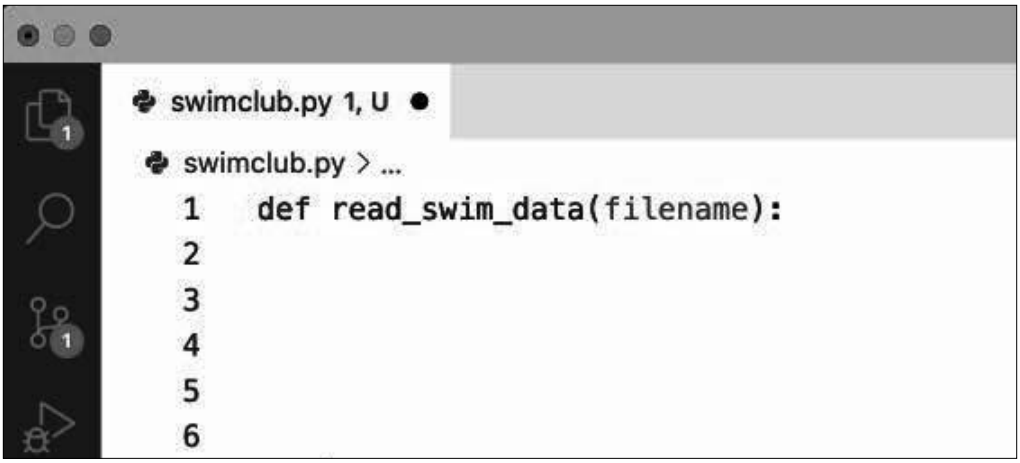
3 Beachten Sie die Verwendung von def und Ihrem besten Freund (dem Doppelpunkt).

Der Einsatz von `def` und dem Doppelpunkt ist ein klarer Hinweis darauf, dass eingerückter Code nicht weit ist.

Speichern Sie Ihren Code, so oft Sie wollen

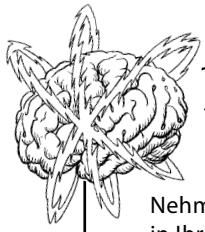
Bauen Sie nun die *Signatur* für die Funktion `read_swim_data` am Anfang der Datei *swimclub.py* ein:

Hier teilt die Benutzerschnittstelle Ihnen mit, dass Ihr Code nicht nur unversioniert, sondern auch ungespeichert ist. Sie können Ihren Code so oft speichern, wie Sie es für nötig halten.



Fügen Sie der Funktion den Code hinzu, der gemeinsam genutzt werden soll

Nachdem die Funktionssignatur der Funktion fertig ist, müssen Sie den nötigen Code aus dem Notebook kopieren und in *swimclub.py* einfügen. Dieser Code befindet sich im Notebook *Times.ipynb* aus dem vorigen Kapitel.



Kopf-Nuss

Nehmen Sie sich etwas Zeit, um den Code in Ihrem *Times.ipynb*-Notebook zu sichten. Brauchen Sie wirklich den **gesamten** Code, der hier enthalten ist?

Einfach den Code kopieren reicht nicht

Wir haben den Code, den wir für nötig halten, in unsere `read_swim_data`-Funktion eingefügt. Bei uns sieht der Code so aus:

Ein paar Mal >>Copy-and-paste<<, und der Code ist in >>swimclub.py<< gelandet. Aber reicht das wirklich aus?

```
def read_swim_data(filename):
    swimmer, age, distance, stroke = FN.removesuffix(".txt").split("-")
    with open(FOLDER+FN) as file:
        lines = file.readlines()
        times = lines[0].strip().split(",")
    converts = []
    for t in times:
        minutes, rest = t.split(":")
        seconds, hundredths = rest.split(".")
        converts.append((int(minutes)*60*100) + (int(seconds)*100) + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths  = str(round(average / 100, 2)).split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes*60
    average = str(minutes) + ":" + str(seconds) + "." + hundredths
```

Haben diese schnörkeligen Unterstreichungen unter manchen Codeteilen eine bestimmte Bedeutung?

Das haben sie tatsächlich. Gut gesehen.

Hiermit teilt VS Code Ihnen mit, dass Ihr Code Werte benutzt, die noch definiert werden müssen. Obwohl der Code syntaktisch in Ordnung ist, wird Python ihn nicht ausführen, solange diese Werte fehlen.

Diese Werte befinden sich im *Times.ipynb*-Notebook.

Sämtlicher nötiger Code muss kopiert werden

Ein Blick auf die schnörkeligen Linien auf der vorherigen Seite macht deutlich, dass `FN`, `FOLDER` und `statistics` alle *fehlen*.

`FOLDER` und `statistics` lassen sich leicht reparieren. Fügen Sie einfach die folgenden zwei Codezeilen am Anfang der *swimclub.py*-Datei ein (*außerhalb* der Funktion):

Teilt Ihrem Code mit, von wo die Funktion >>mean<< importiert werden soll.

```
import statistics

FOLDER = "swimdata/"
```

Teilt Ihrem Code mit, wo die Datendateien zu finden sind.

Wenn Sie den Code aktiv mitverfolgen (ihn beim Lesen eingeben und ausprobieren), werden Sie merken, dass die Schnörkellinien verschwinden, sobald Sie diese Codezeile in VS Code eingeben.

Berauscht von diesem Erfolg, sind Sie jetzt eventuell versucht, auch die Definition der *Konstanten* `FN` einfach hierherzukopieren. Das würde allerdings zu einem Fehler führen. Wie Sie wissen, verweist `FN` im *Times.ipynb*-Notebook auf *eine bestimmte* Datendatei, die Informationen zu Darius enthält. Wenn Sie `FN` in diesem Code weiterverwenden, wird Ihre Funktion ausschließlich diese Datei nutzen und sonst keine. Die Lösung dieses Problems besteht darin, nicht die Konstante `FN` zu verwenden, sondern den Wert, der an die Funktion `read_swim_data` übergeben wird. So kann der Coach letztlich die Dateien aller Schwimmer verarbeiten:

Ein Wert für >>filename<< wird an die Funktion übergeben.

```
def read_swim_data(filename):
    swimmer, age, distance, stroke = filename.removesuffix(".txt").split("-")
    with open(FOLDER + filename) as file:
        lines = file.readlines()
        times = lines[0].strip().split(",")
    converts = []
    for t in times:
        minutes, rest = t.split(":")
        seconds, hundredths = rest.split(".")
        converts.append((int(minutes) * 60 * 100) + (int(seconds) * 100) + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths = str(round(average / 100, 2)).split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes * 60
    average = str(minutes) + ":" + str(seconds) + "." + hundredths
```

Anstatt sich auf den Wert von >>FN<< zu verlassen, nutzt dieser Code den übergebenen Wert von >>filename<<.

Haben Sie's bemerkt? Keine schnörkeligen Linien mehr!

Probefahrt

Sobald Ihre Funktion definiert ist, sollten Sie sie speichern, bevor Sie mit dieser *Probefahrt* weitermachen.

Lassen Sie Ihren *swimclub.py*-Code in VS Code weiterhin geöffnet (wenn Sie wollen). Öffnen Sie nun ein neues Notebook, das Sie *Files.ipynb* nennen. Sie wissen bereits, dass Pythons `import`-Anweisung mit der PSL funktioniert. Wie sich zeigt, können Sie import auch für Ihre eigenen Module nutzen. Und wissen Sie was? Die Datei *swimclub.py* ist ein Python-Modul. Und das wiederum heißt, Sie können `import` verwenden, wie unten gezeigt:

Geben Sie diesen Code, wie bei Ihren anderen Notebooks auch (diesmal allerdings in »Files.ipynb«), in eine Zelle ein und drücken Sie »Shift+Enter«.

```
import swimclub
```

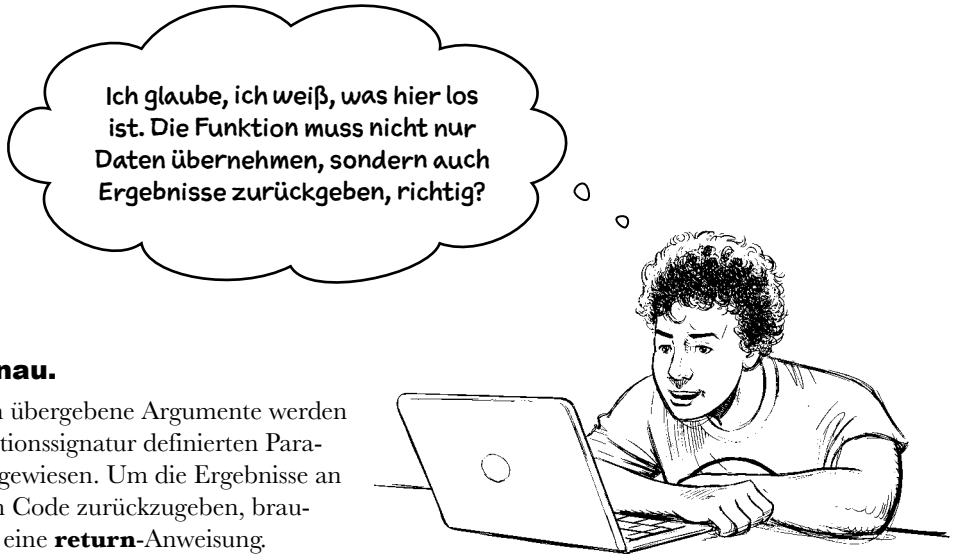
Wenn alles in Ordnung ist, wird nach Ausführung der »import«-Anweisung eine neue leere Codezelle angezeigt. Sehen Sie Fehler, sollten Sie zwei Dinge überprüfen: Stellen Sie sicher, dass Sie Ihren »swimclub.py«-Code gespeichert haben, und sorgen Sie dafür, dass sich »Files.ipynb« im gleichen Ordner befindet wie »swimclub.py« (in Ihrem »Learning«-Ordner).

Über die bekannte Punktschreibweise können Sie Ihre Funktion »read_swim_data« (importiert aus dem »swimclub«-Modul) aufrufen.

Beachten Sie, wie wir den Namen der Datendatei übergeben haben, die hier verarbeitet werden soll. Vorher war dieser Wert der Variablen »FN« zugewiesen.

```
swimclub.read_swim_data("Darius-13-100m-Fly.txt")
```

Drücken Sie nun »Shift+Enter« in Ihrer aktuellen Codezelle. Sollten Sie und wir ähnlich ticken, werden Sie sich vermutlich jetzt wundern. Wir haben erwartet, einige Daten zu sehen, aber stattdessen sehen wir, was Sie sehen, nämlich ... nichts! Was ist denn jetzt schon wieder los?



Ja, ganz genau.

An die Funktion übergebene Argumente werden den in der Funktionssignatur definierten Parameternamen zugewiesen. Um die Ergebnisse an den aufrufenden Code zurückzugeben, brauchen Sie jedoch eine **return**-Anweisung.

Spitzen Sie Ihren Bleistift

Die Änderung ist nicht groß, aber wichtig.

Nehmen Sie sich etwas Zeit, um Ihre `read_swim_data`-Funktion in der Datei *swimclub.py* zu überprüfen. Danach schreiben Sie die **return**-Anweisung auf die unten stehende Leerzeile, die Sie am Ende der Funktion einfügen würden, um Werte an den Aufrufer zurückzugeben.

Einen Vorschlag für die **return**-Anweisung finden Sie auf der folgenden Seite. Trotzdem sollten Sie vor dem Umblättern erst einmal selbst versuchen, diese einzelne Codezeile zu erstellen. (Tipp: Wir haben uns entschieden, sechs Werte aus der Funktion zurückzugeben.)

Antworten auf Seite 136

Spitzen Sie Ihren Bleistift Lösung

von Seite 135

Die Änderung ist nicht groß, aber wichtig.

Sie sollten etwas Zeit investieren, um Ihre `read_swim_data`-Funktion in der Datei *swimclub.py* zu überprüfen. Danach sollten Sie die **return**-Anweisung auf die unten stehende Leerzeile schreiben, die Sie am Ende der Funktion einfügen würden, um Werte an den Aufrufer der Funktion zurückzugeben.

Hier sehen Sie unsere **return**-Anweisung, die sechs Werte zurückgibt. Wie schneidet Ihre Anweisung im Vergleich dazu ab?

```
return swimmer, age, distance, stroke, times, average
```

Die Funktion gibt eine Sammlung von Werten an den aufrufenden Code zurück. Beachten Sie das Fehlen von runden Klammern um die Liste der Variablennamen (die sind in Python nicht nötig).

Aktualisieren und speichern Sie Ihren Code, bevor Sie weitermachen ...

Bevor Sie fortfahren, sollten Sie sicherstellen, dass Ihre `read_swim_data`-Funktion in Ihrer *swimclub.py*-Datei mit der unten stehenden Zeile endet. Achten Sie darauf, dass die Einrückung dieser Codezeile mit den Einrückungen des übrigen Codes Ihrer Funktion übereinstimmt.

Benutzen Sie VS Code, um diese Codezeile am Ende Ihrer Funktion einzufügen. Danach speichern Sie die Datei.

```
return swimmer, age, distance, stroke, times, average
```

Nachdem Sie den Code Ihres Moduls gespeichert haben, können Sie es vermutlich kaum erwarten, zu Ihrem *Files.ipynb*-Notebook zurückzukehren, um zu sehen, wie sich die Änderungen auswirken, oder?

Wir auch nicht. Trotzdem tut es uns leid, Ihnen sagen zu müssen, dass uns eine weitere *Enttäuschung* erwartet.

Probefahrt

Nachdem die `read_swim_data`-Funktion eine **return**-Anweisung besitzt und das `swimclub`-Modul gespeichert ist, kehren Sie zu Ihrem *Files.ipynb*-Notebook zurück, klicken auf die erste Codezelle und benutzen dann **Shift+Enter**, um die beiden Zellen des Notebooks erneut auszuführen.

Drücken Sie »Shift+Enter« einmal ...

```
import swimclub
```

```
swimclub.read_swim_data("Darius-13-100m-Fly.txt")
```

... dann drücken Sie »Shift+Enter« noch einmal für die zweite Zelle.

Obwohl Sie den Code des Moduls angepasst und gespeichert haben, gab es beim erneuten Ausführen der `import`-Anweisung und einem weiteren Aufruf der Funktion keinen Unterschied. Es gibt immer noch keine Ausgaben. **Was ist hier los?**

Das ist ein bisschen peinlich, wenn nicht sogar ärgerlich. Der Code ist aktualisiert und neu importiert, aber Jupyter führt trotzdem die ältere Funktion aus. Warum?!?

Ja, anscheinend ist hier etwas überhaupt nicht Ordnung ...

Tatsächlich liegt das Problem hier aber nicht bei Jupyter, sondern beim Python-Interpreter. Und (so seltsam das klingt) das ist sogar Absicht.

Offensichtlich hat hier jemand ein paar sehr ernste Fragen zu beantworten.

Import im Gespräch

Das heutige Interview führen wir mit Pythons **import**-Anweisung.

Von Kopf bis Fuß: Danke, dass Sie sich Zeit für uns nehmen, besonders so kurzfristig.

import: Es freut mich, hier zu sein.

VKbF: Zugegeben, die letzte *Probefahrt* hat mich etwas aus der Bahn geworfen. Ich habe meinen Code ergänzt und gespeichert und dann meine import-Anweisung erneut ausgeführt, aber nichts hat sich verändert. Ist dieses Verhalten wirklich Absicht?

import: Ja.

VKbF: Ernsthaft?

import: So läuft das bei mir eben …

VKbF: Aber wie kann ich dann mein Problem lösen?

import: Das ist gar nicht so schwer. Sie hätten neu starten müssen, anstatt neu zu importieren.

VKbF: Was?

import: Ich erkläre es Ihnen.

VKbF: Bitte. Ich bin ganz Ohr …

import: Als Sie Ihr neues Notebook erstellt haben, hat der Python-Interpreter eine neue Session gestartet, in der Ihr Code läuft. Die erste Aktion dieser Session war die Ausführung von mir, Ihrer freundlichen **import**-Anweisung für Ihr `swimclub`-Modul.

VKbF: Ja. Und dann habe ich meine Funktion ausgeführt. Ich habe bemerkt, dass sie keine Daten zurückgibt, sie repariert, gespeichert und dann mein Modul erneut importiert.

import: Und genau das ist eben nicht passiert.

VKbF: Jetzt haben Sie mich abgehängt …

import: Sie haben alles getan, was Sie gesagt haben, *bis auf* den letzten Schritt, den »mein Modul erneut importiert«-Teil. Wissen Sie, man sagt, ich sei etwas *schwerfällig*, was die Ressourcennutzung angeht. Daher suchen die Entwickler des Python-Interpreters ständig nach Wegen, meine Verwendung zu verbessern. Ich brauche eine Weile, um meinen Job zu erledigen.

VKbF: Em … okay …

import: Und weil der Import manchmal sehr rechenintensiv sein kann, wurde entschieden, bereits importierte Module zu *cachen* (zwischenzuspeichern). Egal, wie oft ein Modul in einer bestimmten Python-Session importiert wird, es wird immer nur die erste import-Anweisung ausgeführt. Spätere Wiederholungen werden schlicht ignoriert.

VKbF: Das heißt, wenn ich beispielsweise `import abc` in drei Codezellen eingebe und für jede **Shift+Enter** drücke, wird nur die erste Zelle ausgeführt?

import: Na ja. Es werden schon alle Zellen ausgeführt, aber nur die erste import-Anweisung wird tatsächlich berücksichtigt. Der zweite und der dritte Import werden ignoriert, weil sich das Modul schon im Cache befindet.

VKbF: Und der Python-Interpreter ignoriert spätere Importe auch dann, wenn sich der Code zwischen dem ersten und zweiten oder dem zweiten und dritten import verändert, weil er darauf optimiert ist, aus dem Cache zu lesen, richtig?

import: Ja.

VKbF: Aha! Langsam verstehe ich. Aber wie bekomme ich das Problem in den Griff? Kann ich den Cache ausleeren oder den Interpreter anweisen, ihn zu ignorieren?

import: Die beste »Lösung« besteht darin, Ihre Python-Session neu zu starten, anstatt Ihr Modul neu zu importieren. Dadurch findet der nächste Import in einer neuen Python-Session statt, deren Cache zurückgesetzt wurde.

VKbF: Okay. Das erscheint mir sinnvoll. Aber wie starte ich meine Session am besten neu?

import: Bei Jupyter Notebook gibt es einen großen, leuchtenden »Restart«-Button am oberen Rand des VS-Code-Fensters. Wenn Sie ihn anklicken, wird die vorherige Python-Session inklusive ihres Caches gelöscht und Sie können von vorne anfangen.

VKbF: Großartig. Dann werde ich das gleich mal machen. Danke für Ihre Hilfe, **import**!

import: Gern geschehen!

Probefahrt

Haben wir beim dritten Mal mehr Glück?

Klicken Sie auf den *Restart*-Button oben in Ihrem VS-Code-Fenster (bei geöffnetem *Files.ipynb*-Notebook).

Je nach Konfiguration Ihrer VS-Code-Installation werden Sie möglicherweise aufgefordert, den Neustart zu bestätigen. Kommen Sie dieser Aufforderung bei Bedarf nach.

Ihr Klick startet die Python-Session neu. Dies setzt den Modulcache zurück und entfernt alle vorhandenen Variablen und ihre Werte aus dem Arbeitsspeicher. Nach dem *Restart* klicken wir gerne noch auf diesen Button:

Ein Klick auf diesen Button setzt die Jupyter-Schnittstelle zurück. Die Zellnummerierung sowie alle früheren Ausgaben verschwinden. Sie beginnen wieder mit einer sauberen, einsatzbereiten und zurückgesetzten Python-Sitzung.

Nachdem Ihre Python-Session neu gestartet wurde, nutzen wir **Shift+Enter**, um diese beiden Codezellen noch einmal auszuführen:

```
import swimclub
```

```
swimclub.read_swim_data("Darius-13-100m-Fly.txt")
```

```
('Darius',
 '13',
 '100m',
 'Fly',
 ['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'],
 '1:26.58')
```

Nach dem Neustart der Session wird nun auch der aktualisierte Code in Ihr Modul importiert, und die Funktion gibt die sechs Einzeldaten für Darius zurück.

Module verwenden, um Code weiterzugeben

Im Moment besteht der Code in Ihrer Datei *swimclub.py* aus einer einzelnen **import**-Anweisung, einer Konstantendefinition und einer einzelnen Funktion.

Sobald Sie Code in seine eigene Datei verschieben, wird er zu einem Python-*Modul*, das Sie bei Bedarf importieren können.

```
import swimclub
```

Importieren Sie Ihr Modul und ...

```
⋮
```

```
swimclub.read_swim_data("Darius-13-100m-Fly.txt")
```

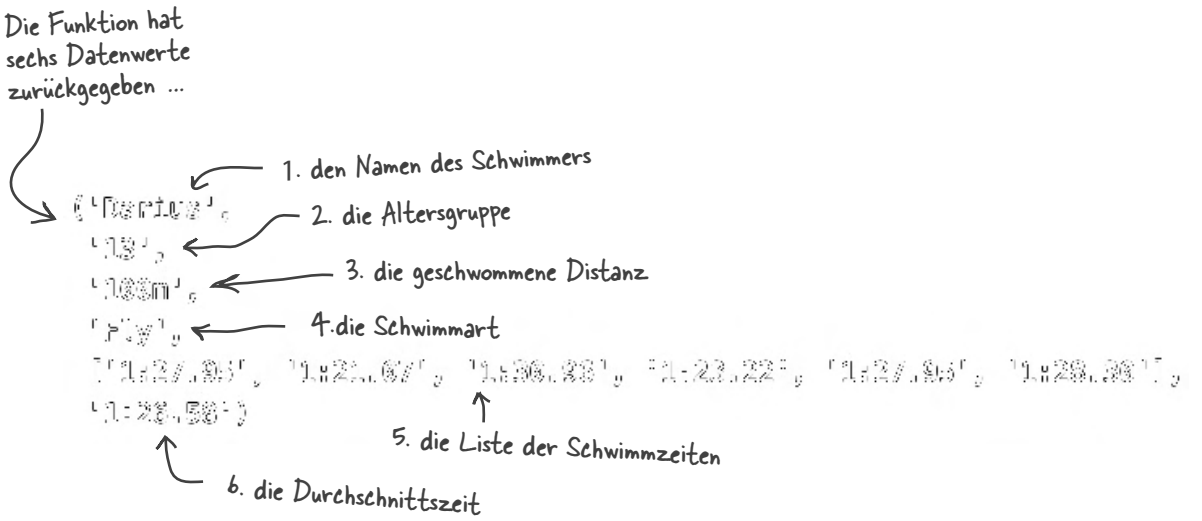
... rufen Sie Ihre Funktion auf, indem Sie dem Funktionsnamen den Namen des Moduls gefolgt von einem PUNKT voranstellen.

Dies ist ein voll qualifizierter Name.

Wenn Sie Ihre Funktion mit der »Modul PUNKT Funktion«-Schreibweise aufrufen, ergänzen (oder »qualifizieren«) Sie den Funktionsnamen mit dem Namen des Moduls, das die Funktion enthält. Neben anderen Importtechniken ist dies eine der häufigsten. Weitere Beispiele hierfür werden Sie beim Durcharbeiten dieses Buchs finden.

Erfreuen Sie sich am Glanz der zurückgegebenen Daten

Sehen wir uns noch einmal die Daten an, die von Ihrem letzten Aufruf der `read_swim_data`-Funktion zurückgegeben wurden.



Gut gesehen.

Das ist jetzt vielleicht nicht die Erklärung, die Sie erwarten, aber die runden Klammern sind Absicht.

Wir wollen hier etwas mehr ins Detail gehen, damit Sie die Vorgänge besser verstehen. Nachdem wir vorhin die **import**-Anweisung in die Mangel genommen haben, ist jetzt die **Funktion** an der Reihe.

Die Funktion im Gespräch

Eine Unterhaltung mit Pythons **Funktion**.

Von Kopf bis Fuß: Vielen Dank, dass Sie sich trotz Ihres vollen Terminkalenders die Zeit für ein Gespräch mit uns genommen haben.

Funktion: Kein Problem.

VKbF: Wie kommt es, dass Sie so beschäftigt sind?

Funktion: Ich bin immer und überall im Einsatz.

VKbF: Und Sie arbeiten mit allem?

Funktion: Wenn Sie die von mir akzeptierten Daten meinen, dann ja. Ich nehme mit Freude alles entgegen, was Sie mir geben.

VKbF: Könnten Sie das ein wenig erläutern?

Funktion: Sicher. Sie können mir eine beliebige Anzahl von Argumentwerten übergeben, die ich gern auf meine Parameter abbilde. Sie müssen nur dafür sorgen, dass die Anzahl übereinstimmt. Wenn ich zwei Parameter besitze, erwarte ich auch zwei Argumentwerte.

VKbF: Und was passiert, wenn ich Ihnen stattdessen ein oder drei Argumente übergebe?

Funktion: Dann bekomme ich schlechte Laune.

VKbF: Ich verstehe. So ist das also, hmm?

Funktion: Ja, diese Dinge nehme ich sehr genau. Außer natürlich, wenn einer meiner zwei Parameter als *optional* deklariert wurde.

VKbF: Und was passiert dann?

Funktion: Bleiben wir einen Moment bei meinem Beispiel mit den zwei Parametern. Wenn beispielsweise der zweite Parameter optional ist, übernehme ich, ohne zu murren, einen oder zwei Parameterwerte, und zwar ohne weiter nachzufragen.

VKbF: Aber was wird dem zweiten Parameter zugewiesen, wenn ich Sie nur mit einem Argument aufrufe?

Funktion: Typischerweise hat der Programmierer, der mich geschrieben hat, für diesen Fall einen Standardwert definiert, den ich dann verwende.

VKbF: Das klingt jetzt ziemlich komplex.

Funktion: Ist es aber eigentlich nicht. Und das braucht, ehrlich gesagt, auch längst nicht jede Funktion. Aber wenn Sie es brauchen, ist es ein Teil von mir. Ich bin da ziemlich flexibel.

VKbF: Und was ist mit den Rückgabewerten? Funktioniert das da genauso? Kann ich beliebig viele Werte zurückgeben?

Funktion: Nein.

VKbF: Ehrlich? Nein? Mehr haben Sie dazu nicht zu sagen?

Funktion: Nun ja. Ich dachte, das wäre klar. Stellen Sie sich mathematische Funktionen vor, die genau einen Wert zurückgeben müssen. So ist das auch bei mir. Beliebig viele Werte rein, aber nur EIN Ergebnis zurück.

VKbF: Aber, ähhm … wenn ich den Aufruf von `read_swim_data` auf der vorherigen Seite betrachte, dann sehe ich, dass doch *sechs* Ergebnisse zurückgegeben werden.

Funktion: Nein, es ist nur EIN Ergebnis.

VKbF: Was zum …

Funktion: Wenn Sie genau hinsehen, werden Sie die runden Klammern um die sechs Werte bemerken, richtig?

VKbF: Ja, aber …

Funktion: Hier gibt es kein »aber«. Diese Klammern umgeben ein einzelnes Tupel, das die sechs einzelnen Datenwerte enthält. Wie ich bereits sagte: Es wird EIN Ergebnis zurückgegeben. Entweder ein einzelner Datenwert oder ein einzelnes Tupel, das natürlich mehrere Werte enthalten kann.

VKbF: Aber der Code konvertiert die sechs Rückgabewerte doch gar nicht in ein Tupel.

Funktion: Ja ha … der Code nicht, *aber ich*. Das mache ich automatisch, wenn ich sehe, dass ein Programmierer versucht, mehr als EIN Ergebnis zurückzugeben. Sie können mir später danken.

VKbF: Nein, ich bedanke mich lieber gleich. Diese Informationen sind wirklich wichtig. Danke für das Gespräch!

Funktion: Ich helfe jederzeit gerne, die Dinge zu klären!

Funktionen geben bei Bedarf ein Tupel zurück

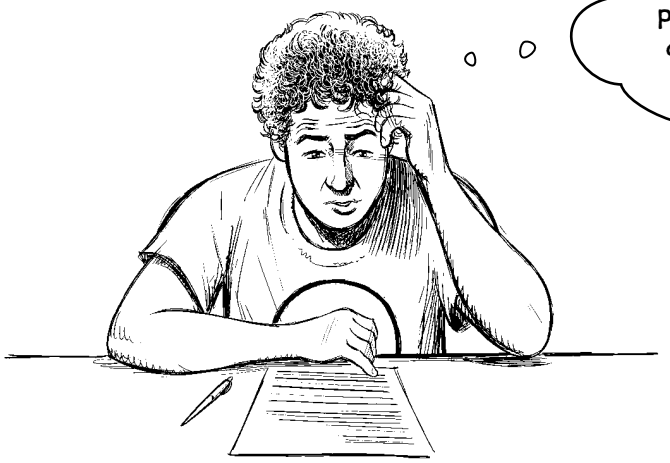
Wenn Sie eine Funktion aufrufen, die aussieht, als gäbe sie mehrere Ergebnisse zurück, sollten Sie noch einmal überlegen. Das ist nämlich nicht der Fall. Stattdessen erhalten Sie ein einzelnes Tupel zurück, das eine Sammlung von Ergebnissen enthält, unabhängig davon, wie viele einzelne Ergebnisse es gibt.

Das sieht aus, als gäbe die Funktion sechs Objekte zurück. Das ist aber nicht erlaubt, denn Funktionen haben grundsätzlich nur einen Rückgabewert. Daher verpackt Python die zurückgegebenen Objekte in einem Tupel.

```
return swimmer, age, distance, stroke, times, average
```

```
('Darius',
 '13',
 '100m',
 'fly',
 ['1:24.95', '1:25.67', '1:30.28', '1:23.22', '1:27.95', '1:28.28'],
 '1:28.56')
```

Tupel umgeben ihre Objekte mit runden Klammern (im Gegensatz zu Listen, für die eckige Klammern genutzt werden).



Ich würde mich über ein paar Zusatzinformationen dahin gehend freuen, was ein Tupel eigentlich ist …

Sehr guter Vorschlag.

Wir wollen nicht behaupten, dass hier ein bisschen Gedankenlesen im Spiel ist, aber erschreckenderweise hatten wir genau die gleiche Idee.

Hinter den Kulissen

Pythons Tupel hautnah und persönlich erleben.

Die Python-Dokumentation definiert ein **Tupel** als eine *immutable Folge (oder Sequenz)*.

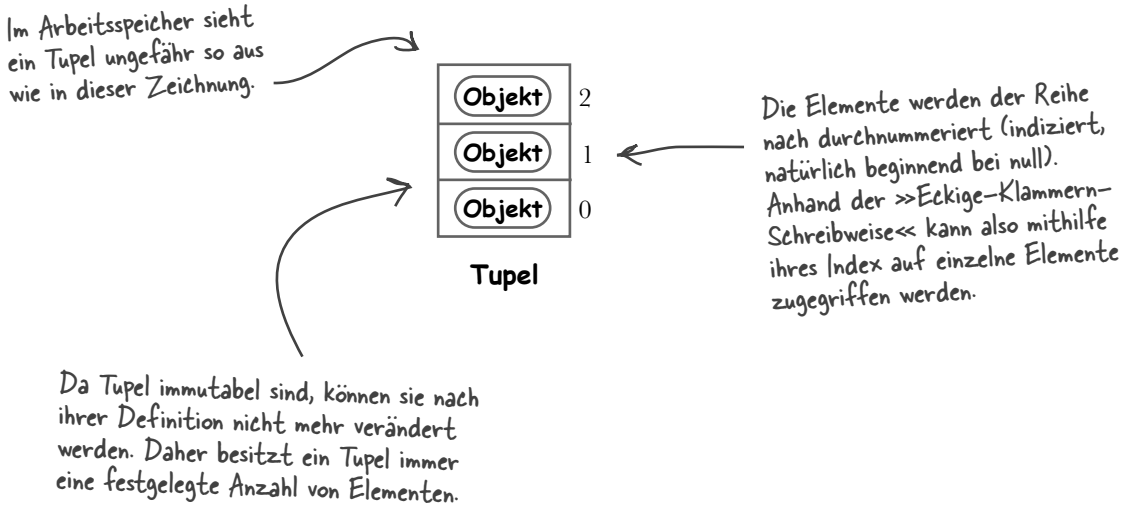
Immutabel?

Zwei *immutable* Datentypen haben Sie schon kennengelernt: Zahlen und Strings. Beiden ist gemeinsam, dass ein Wert, der einmal im Code erstellt wurde, **nicht mehr verändert** werden kann. So ist `42` immer `42`. Die Zahl kann nicht verändert werden, sie ist *immutabel.* Das Gleiche gilt für Pythons Strings: Einmal erstellt, ist »Hallo« immer »Hallo«. Der String ist unveränderlich, er ist *immutabel*.

Pythons **Tupel** übernimmt diesen Gedanken und wendet ihn auf eine Sammlung von Datenwerten an. Dabei kann es hilfreich sein, sich ein Tupel als eine Art *konstante Liste* vorzustellen. Sobald die Werte einem Tupel zugewiesen sind, kann das Tupel nicht mehr verändert werden. Es ist *immutabel.*

Folge?

Wenn Sie anhand der »Eckige-Klammern-Schreibweise« auf einzelne Elemente (oder »Slots«) einer Sammlung zugreifen können, arbeiten Sie mit einer *Folge*. Die bekannteste Art einer Folge in Python ist die Liste. Daneben gibt es aber auch noch andere, inklusive Strings und … **Tupel**. Neben der Möglichkeit, eckige Klammern verwenden zu können, behalten Tupel außerdem die *Reihenfolge* der enthaltenen Elemente bei.



Übung

Angenommen, die folgende Codezeile wurde in einer neuen, leeren Codezelle Ihres Notebooks ausgeführt (Tipp: Geben Sie diese Zeile am besten gleich in Ihr Notebook ein und führen Sie sie aus, damit Sie bei Bedarf damit experimentieren können):

```
data = swimclub.read_swim_data("Darius-13-100m-Fly.txt")
```

Verwenden Sie die »Eckige-Klammern-Schreibweise«, um auf die Schwimmzeiten des Datentupels zuzugreifen und sie einer Liste namens `times` (Zeiten) zuzuweisen. Danach lassen Sie den Inhalt der Liste `times` anzeigen. Schreiben Sie Ihren Code hier auf:

Verwenden Sie Pythons *Mehrfachzuweisung*, um alle Elemente den folgenden benannten Variablen zuzuweisen: `swimmer` (Schwimmer), `age` (Alter), `distance` (Entfernung), `stroke` (Schwimmart), `times2` (Zeiten 2) und `average` (Durchschnitt). Danach geben Sie den Inhalt der Liste `times2` auf dem Bildschirm aus.

Antworten auf Seite 146

Es gibt keine Dummen Fragen

F: Ich habe gerade ein »print dir« auf meinem `data`-Tupel ausgeführt. Dabei wurden nur zwei Dunder-Methoden ausgegeben. Kann man mit Tupeln wirklich nichts Sinnvolles anstellen?

A: Doch, selbstverständlich. Wenn Sie allerdings die Ausgaben des **print dir**-Combo-Mambos für Listen mit Tupeln vergleichen, scheinen Listen den Tupeln tatsächlich überlegen zu sein, weil Tupel so gut wie keine eigenen Methoden besitzen. Vergessen Sie aber nicht, dass Tupel immutabel sind. Sie können die enthaltenen Daten nach der Zuweisung also nicht mehr ändern (allein das reduziert die Anzahl der für Tupel nötigen Methoden bereits). Beim Durcharbeiten dieses Buchs werden Ihnen Beispiele begegnen, in denen es besser ist, Tupel anstelle von Listen zu verwenden und umgekehrt. Kurz gesagt: Beide sind wichtig und sinnvoll.

Übung, Lösung

von Seite 145

Sie sollten davon ausgehen, dass die folgende Codezeile in einer neuen, leeren Codezelle Ihres Notebooks ausgeführt wurde:

```
data = swimclub.read_swim_data("Darius-13-100m-Fly.txt")
```

Sie sollten die »Eckige-Klammern-Schreibweise« verwenden, um auf die Schwimmzeiten des Datentupels zuzugreifen und sie einer Liste namens `times` (Zeiten) zuzuweisen. Danach sollten Sie den Inhalt der Liste `times` anzeigen lassen:

Im zurückgegebenen Tupel sind die Schwimmzeiten an der fünften Position. Um die gewünschten Daten auszuwählen, greifen Sie also über den Index 4 in eckigen Klammern darauf zu.

```
times = data[4]

times
```

Geben Sie den Namen einer Variablen (allein) in eine Codezelle ein oder fügen Sie ihn am Ende einer vorhandenen Codezelle ein, um ihren Inhalt ausgeben zu lassen.

Anhand von Pythons *Mehrfachzuweisung* sollten Sie alle Elemente den folgenden benannten Variablen zuweisen: `swimmer`, `age`, `distance`, `stroke`, `times2` und `average`. Danach sollten Sie den Inhalt der Liste `times2` auf dem Bildschirm ausgeben:

Durch die Entpackungsfähigkeiten der Mehrfachzuweisung werden die Datenwerte der sechs Elemente des Tupels den einzelnen Variablennamen zugewiesen (inklusive >>times2<<).

```
swimmer, age, distance, stroke, times2, average = data

times2
```

Der Inhalt von >>times2<< kann auf die gleiche Weise auf dem Bildschirm ausgegeben werden wie beim oben gezeigten >>times<<.

Probefahrt

Hier sehen Sie unsere Ausgaben, die beim Ausführen der Codezellen der letzten *Übung* auf dem Bildschirm angezeigt wurden. Der Variablen `data` wird das gesamte von `read_swim_data` zurückgegebene (sechsteilige) Tupel zugewiesen. Die Variablen `times` und `times2` erhalten dagegen die Strings mit den Schwimmzeiten aus dem entsprechenden Element im Datentupel (wenn auch anhand verschiedener Zugriffsweisen).

```
data = swimclub.read_swim_data("Darius-13-100m-Fly.txt")
```

```
data
```

```
('Darius',
 '13',
 '100m',
 'Fly',
 ['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30'],
 '1:26.58')
```

Alle Daten.

Die Liste der Schwimmzeiten aus dem Element mit dem Index 4, also dem fünften Element im >>data<<-Tupel.

```
times = data[4]
times
```

```
['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30']
```

```
swimmer, age, distance, stroke, times2, average = data
times2
```

```
['1:27.95', '1:21.07', '1:30.96', '1:23.22', '1:27.95', '1:28.30']
```

Die einzelnen Werte aus dem >>data<<-Tupel benannten Variablen zuweisen. Hinweis: Das Tupel enthält sechs Datenwerte (auf der rechten Seite des Zuweisungsoperators) und sechs Variablennamen (linke Seite). Das passt doch wunderbar!

Ihr Code funktioniert jetzt also mit allen Schwimmerdateien? Das sind gute Neuigkeiten. Ich kann es kaum erwarten, Ihr System einzusetzen, um die Schwimmer zu finden, die nicht 100 Prozent geben. Was kommt als Nächstes?

Eine Liste mit Dateinamen wäre schön.

Ihre `read_swim_data`-Funktion übernimmt als Teil des `swimclub`-Moduls den Namen einer beliebigen Schwimmerdatei und gibt Ihnen ein Tupel mit Ergebnissen zurück.

Was jetzt benötigt wird, ist eine vollständige Liste der Dateinamen. Es sollte möglich sein, diese vom verwendeten Betriebssystem zu bekommen. Wie Sie sich schon denken können, hält die PSL auch für diesen Fall eine Lösung bereit.

Bisher haben wir nur die Datei mit den Daten von Darius genutzt. Sie können gern irgendeinen anderen Dateinamen aus dem >>swimdata<<-Ordner Ihrer >>read_swim_data<<-Funktion übergeben, um sicherzugehen, dass Ihr Code mit allen Datendateien des Coachs arbeiten kann. Denken Sie daran: Es ist der *Lebenszweck* von Jupyter Notebook, Ihnen beim Schreiben Ihres Codes das Experimentieren zu ermöglichen.

Es gibt keine Dummen Fragen

F: In meinem *Learning*-Ordner ist gerade ein neuer Ordner mit dem Namen *__pycache__* aufgetaucht. Was ist das, und wo kommt er her?

A: Dieser Ordner wird intern vom Python-Interpreter verwendet, um kompilierte Kopien aller von Ihnen erstellten und importierten Module zwischenzuspeichern (zu »cachen«). Sie müssen Ihren Python-Code nicht selbst kompilieren, um ihn auszuführen. Das erledigt Python hinter den Kulissen für Sie. Dieser interne Bytecode wird dann ausgeführt. Da dieser Prozess beim Importieren von Modulen recht kostspielig sein kann, legt der Interpreter während des Importprozesses eine Kopie des kompilierten Bytecodes im Ordner *__pycache__* ab. Beim nächsten Mal, wenn ein Modul (in einer neuen Session) importiert werden soll, vergleicht der Interpreter den Zeitstempel Ihres Moduls mit dem des Bytecodes. Sind sie gleich, wird der Bytecode wiederverwendet. Ansonsten wird die Umwandlung von Code in Bytecode erneut angestoßen. Sie können die Dateien im *__pycache__*-Ordner getrost ignorieren und es dem Interpreter überlassen, ihn zu verwalten (allerdings wollen Sie den Ordner vermutlich von Ihrem Git-Repo ausschließen).

Holen wir uns eine Liste der Dateinamen des Coachs

Geht es um die Arbeit mit Ihrem Betriebssystem (sei dies nun *Windows*, *macOS* oder *Linux*), ist die PSL für Sie da. Mithilfe des `os`-Moduls kann Ihr Python-Code plattformunabhängig mit dem Betriebssystem kommunizieren. Hier verwenden wir das `os`-Modul, um uns eine Liste der Dateien im *swimdata*-Ordner zu holen.

Geben Sie die folgenden Codebeispiele am besten direkt beim Lesen in Ihr *Files.ipynb*-Notebook ein und führen Sie sie aus.

Wie Sie sich vielleicht schon gedacht hatten, müssen wir das >>os<<-Modul vor seinem Einsatz erst importieren.

```
import os
```

Sie benötigen die Namen der Dateien im *swimdata*-Ordner. Genau für diesen Zweck enthält das `os`-Modul die superpraktische `listdir`-Funktion. Wenn Sie `listdir` den Speicherort eines Ordners übergeben, gibt die Funktion eine Liste der darin enthaltenen Dateien zurück.

Die vom >>os<<-Modul bereitgestellte Funktion >>listdir<< aufrufen.

Die Dateiliste wird einer neuen Variablen namens >>swim_files<< zugewiesen.

```
swim_files = os.listdir(swimclub.FOLDER)
```

In diesem Fall geben Sie den gewünschten Ordner über die Konstante >>FOLDER<< aus Ihrem >>swimclub<<-Modul an.

Wir verzeihen Ihnen, wenn Sie dachten, dass die Liste `swim_files` 60 Elemente enthält. Schließlich enthält der Ordner ja auch 60 Dateien. Auf unserem *Mac* bekamen wir allerdings fast einen Schock, als wir noch einmal die Größe von `swim_files` überprüften:

Die eingebaute Funktion >>len<< gibt die Anzahl der Elemente in Ihrer Liste aus.

```
len(swim_files)
```

```
61
```

Dies ist doch komisch, oder?

Zeit für etwas Detektivarbeit …

Beides optional.

Sie haben erwartet, dass die Dateiliste 60 Namen enthält. Trotzdem enthält `swim_files` laut **len** angeblich 61 Elemente.

Wir wollen herausbekommen, was hier los ist. Dafür lassen wir uns erst einmal den Wert der Liste `swim_files` auf dem Bildschirm ausgeben.

Geben Sie dies in eine leere Codezelle ein und drücken Sie dann >>Shift+Enter<<.

```
print(swim_files)
```

```
['Hannah-13-100m-Free.txt', 'Darius-13-100m-Back.txt', 'Owen-15-100m-Free.txt', 'Mike-15-100m-Free.txt',
'Hannah-13-100m-Back.txt', 'Mike-15-100m-Back.txt', 'Mike-15-100m-Fly.txt', 'Abi-10-50m-Back.txt', 'Ruth-
13-200m-Free.txt', '.DS_Store', 'Tasmin-15-100m-Back.txt', 'Erika-15-100m-Free.txt', 'Ruth-13-200m-
Back.txt', 'Abi-10-50m-Free.txt', 'Maria-9-50m-Free.txt', 'Elba-14-100m-Free.txt', 'Tasmin-15-100m-
```

⋮

Die aktuell 61 Elemente der Liste >>swim_files<<.

Die Ausgabe ist ein bisschen schwer lesbar. Am besten, wir sortieren die Liste erst einmal. Dann ist sie leichter zu untersuchen.

Das ist eine großartige Idee!

Benutzen wir unseren Combo-Mambo, um zu sehen, welche eingebauten Funktionen Listen bereitstellen.

Was können Sie mit Listen anstellen?

Hier ist die Ausgabe unseres **print dir**-Combo-Mambos für die Liste `swim_files`:

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__',
'__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

Schauen Sie mal!

Vorsicht mit den eingebauten Funktionen für Listen (besonders mit denen, die Listen verändern)!

*Wenn Sie sich jetzt bei der Aussicht, die **sort**-Methode zu verwenden, erwartungsvoll die Hände reiben, sollten Sie noch einen Moment innehalten. Die **sort**-Methode ändert die Abfolge der Listenelemente »an Ort und Stelle«, was bedeutet, dass die neue Reihenfolge den bisherigen Inhalt der Liste **überschreibt**! Die alte Sortierung ist für immer verloren, und es gibt keinen Weg zurück.*

*Wenn Sie die aktuelle Reihenfolge der Listenelemente beibehalten wollen und trotzdem sortieren müssen, kann Ihnen eine weitere eingebaute Funktion helfen. Die Built-in Function (BIF) **sorted** gibt eine geordnete Kopie Ihrer Listendaten zurück, ohne die Reihenfolge der Ausgangsliste zu verändern. Auch hier gibt es keine Möglichkeit, den Ausgangszustand wiederherzustellen, da die ursprüngliche Liste nicht verändert wurde.*

Bei der Bestie von Caerbannóg! Was ist das denn?

```
print(sorted(swim_files))
```

```
['.DS_Store', 'Abi-10-100m-Back.txt', 'Abi-10-100m-Breast.txt', 'Abi-10-50m-Back.txt', 'Abi-10-50m-
Breast.txt', 'Abi-10-50m-Free.txt', 'Ali-12-100m-Back.txt', 'Ali-12-100m-Free.txt', 'Alison-14-100m-
Breast.txt', 'Alison-14-100m-Free.txt', 'Aurora-13-50m-Free.txt', 'Bill-18-100m-Back.txt', 'Bill-18-200m-
Back.txt', 'Blake-15-100m-Back.txt', 'Blake-15-100m-Fly.txt', 'Blake-15-100m-Free.txt', 'Calvin-9-50m-

                                        ⋮

'Tarun-13-100m-Back.txt', 'Tarun-13-100m-Free.txt', 'Tarun-13-200m-Back.txt', 'Tarun-13-200m-Free.txt', 'Tarun-
13-50m-Free.txt', 'Tasmin-15-100m-Back.txt', 'Tasmin-15-100m-Breast.txt', 'Tasmin-15-100m-Free.txt',
'Tasmin-15-200m-Breast.txt']
```

Ja, das könnte ein Problem sein.

swimdata.zip wurde ursprünglich auf einem Mac erstellt. Dadurch wurde die Datei *.DS_Store* dem ZIP-Archiv automatisch hinzugefügt. Diese Art von betriebssystemspezifischen Problemen kann oft zu Schwierigkeiten führen.

Bevor wir weitermachen, ist es daher wichtig, den unerwünschten Dateinamen aus der `swim_files`-Liste zu *entfernen*.

Übung

Hier sehen Sie eine Liste der eingebauten Methoden für die Arbeit mit Listen:

```
'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort'
```

Was die Methoden **append** und **sort** tun, wissen Sie bereits. Aber was ist mit den anderen?

Benutzen Sie die BIF **help** und verbringen Sie etwas Zeit in Ihrem Notebook, um herauszufinden, was einige dieser Methoden tun. Das Ziel ist, eine Methode zu finden, mit der Sie die unerwünschte Datei *.DS_Store* löschen können. Wenn Sie glauben, eine Methode gefunden zu haben, notieren Sie den Namen auf den unten stehenden Zeilen:

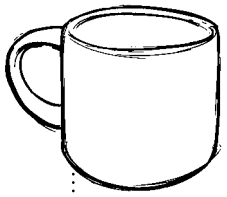
_____

_____

→ Antworten auf Seite 154

Geheimtipp des Übersetzers: Suchen Sie nach englischen Entsprechungen für das Wort »entfernen«.

Wenn Sie die eingebaute »help«-Funktion verwenden, um mehr über die eingebauten Datenstrukturen zu erfahren, stellen Sie dem Methodennamen den Namen der jeweiligen Datenstruktur voran, wie beispielsweise »help(list.append)« oder »help(set.add)«.



Entspannen Sie sich

Lassen Sie sich von der oben stehenden Übung nicht stressen, wenn Sie mehr als eine Lösung der Aufgabe finden. Das ist in Ordnung, denn manchmal kann ein Ziel auf verschiedenen Wegen erreicht werden. Es gibt äußerst selten nur einen absolut richtigen Weg, etwas zu tun. Wie in den meisten Fällen sollten Sie sich *zuerst* darauf konzentrieren, dass Ihr Code tut, was er soll, bevor Sie versuchen, ihn zu optimieren.

Übung, Lösung

von Seite 153

Wir haben Ihnen eine Liste der eingebauten Methoden für die Arbeit mit Listen gezeigt:

```
'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort'
```

Was die Methoden **append** und **sort** tun, wussten Sie bereits. Aber was ist mit den anderen?

Sie sollten die BIF **help** nutzen und etwas Zeit in Ihrem Notebook verbringen, um herauszufinden, was einige dieser Methoden tun. Ziel war es, eine Methode zu finden, mit der Sie die unerwünschte Datei *.DS_Store* aus der Liste entfernen können. Den Namen der gefundenen Methode sollten Sie auf den unten stehenden Zeilen notieren:

remove

Die eingebaute Listenmethode >>remove<< sieht aus, als sei sie hier die richtige.

Es gibt keine Dummen Fragen

F: Sehe ich die Mac-spezifische Datei *.DS_Store* auch, wenn ich *swimdata.zip* auf etwas anderem als einem Mac auspacke?

A: Leider ja. Das ZIP-Archiv wurde auf einem Apple-Gerät erstellt. Die *.DS_Store*-Datei wird also vorhanden sein, es sei denn, die Person, von der das Archiv stammt, hat ihr Zip-Werkzeug so eingestellt, dass die unerwünschte Datei ausgeschlossen wird (was in diesem Fall leider *nicht* passiert ist).

F: Können wir das Problem nicht vermeiden, wenn wir den Coach bitten, hierfür etwas anderes als einen Mac zu verwenden?

A: Wir haben den Coach gefragt, und er meinte, lieber würde er seine Steuererklärung machen …

F: Anscheinend gibt es eine Beziehung zwischen Tupeln und Listen. Sie haben eine gewisse Ähnlichkeit. Ich vermute, eine Liste ist auch eine Folge, aber ist sie auch immutabel?

A: Listen sind tatsächlich Folgen, aber sie sind nicht immutabel. Listen werden als *mutabel* bezeichnet, weil sie (im Gegensatz zu Tupeln) beim Ausführen Ihres Codes dynamisch verändert werden können.

F: Liege ich richtig mit der Vermutung, dass Listen viel nützlicher sind als Tupel?

A: Das kommt darauf an. Beide Datenstrukturen können nützlich sein. Insgesamt kommen Listen aber häufiger zum Einsatz als Tupel, weil Listen einfach für mehr Anwendungsfälle geeignet sind. Das heißt jedoch nicht, dass Listen »besser« oder »nützlicher« sind. Listen und Tupel wurden lediglich für verschiedene Zwecke entworfen.

Probefahrt

Von den elf Methoden, die in jede Python-Liste eingebaut sind, ist Ihnen **remove** besonders aufgefallen.

Hier sehen Sie, was die eingebaute Funktion **help** über **remove** zu sagen hat:

```
help(swim_files.remove)
```

```
Help on built-in function remove:

remove(value, /) method of builtins.list instance
    Remove first occurrence of value.

    Raises ValueError if the value is not present.
```

Offenbar brauchen wir hier genau diese Methode.

>>Das erste Vorkommen des Werts entfernen.<<

Natürlich *schrumpft* die Liste von 61 auf 60 Elemente, nachdem wir die **remove**-Methode auf unserer **`swim_files`**-Liste aufgerufen haben (wobei *.DS_Store* als Dateiname für das Entfernen angegeben wurde).

Die Datei angeben, die aus der Liste entfernt werden soll.

```
swim_files.remove('.DS_Store')
```

Keine Sorge, wir haben nur den Dateinamen aus der Liste entfernt. Die Datei selbst existiert weiterhin auf Ihrer Festplatte.

Die Länge der Liste nach dem Entfernen überprüfen.

```
len(swim_files)
```

```
60
```

Das ist schon besser. Die Zahl der Dateinamen in der Liste >>swim_files<< entspricht jetzt unseren Erwartungen.

Wenn Sie sich noch einmal schnell unsere sortierte Liste mit Dateinamen ansehen, werden Sie feststellen, dass sie nur eine Datei mit dem Namen >>.DS_Store<< enthält. Sollte die Liste mehrere Dateien dieses Namens enthalten, müssen Sie >>remove<< so oft aufrufen, bis alle Exemplare entfernt wurden.

Das wäre schön, was?

Wir könnten alle Vorsicht über Bord werfen und sofort mit der Erstellung der Balkendiagramme beginnen. Vielleicht ist es dafür aber noch ein bisschen zu früh.

Ihre `read_swim_data`-Funktion hat bisher ganz gut funktioniert, aber können Sie sicher sein, dass sie für wirklich *alle* Schwimmerdateien nutzbar ist? Nehmen wir uns etwas Zeit, um herauszufinden, ob `read_swim_data` mit allen übergebenen Datendateien funktioniert.

Übung

Sehen wir mal, ob wir bestätigen können, dass Ihre `read_swim_data`-Funktion wirklich mit allen Schwimmerdateien funktioniert.

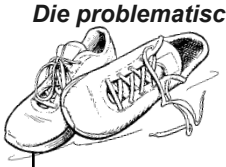
Schreiben Sie eine **for**-Schleife, um die Dateinamen in `swim_files` nacheinander zu verarbeiten. Bei jedem Durchlauf soll der Name der gerade verarbeiteten Datei ausgegeben werden. Danach sorgen Sie dafür, dass Ihre `read_swim_data`-Funktion mit dem aktuellen Dateinamen aufgerufen wird. Die von der Funktion zurückgegebenen Daten müssen in diesem Fall nicht ausgegeben werden, trotzdem müssen Sie die Funktion aufrufen.

Schreiben Sie Ihren Code hier auf und notieren Sie außerdem alles, was Sie bei der Ausführung der **for**-Schleife lernen konnten:

Hier kommt Ihr Code hin. ↳

Hier stehen Ihre Notizen. ↘

→ Antworten auf Seite 158



Übung, Lösung

von Seite 157

Sie sollten Ihre `read_swim_data`-Funktion mit allen Dateien in der `swim_files`-Liste testen.

Sie sollten eine **for**-Schleife schreiben, um die Dateinamen in `swim_files` nacheinander zu verarbeiten. Bei jedem Durchlauf sollte der Name der gerade verarbeiteten Datei ausgegeben werden. Danach sollten Sie dafür sorgen, dass Ihre `read_swim_data`-Funktion mit dem aktuellen Dateinamen aufgerufen wird. Die von der Funktion zurückgegebenen Daten mussten in diesem Fall nicht ausgegeben werden, die Funktion musste aber trotzdem aufgerufen werden.

Hier sehen Sie, welchen Code wir nach dem Experimentieren in unserem *Files.ipynb*-Notebook gefunden haben, sowie die vom Code erzeugten Ausgaben:

Die »for«-Schleife iteriert über die Daten in der Liste »swim_files« und gibt dabei den Namen der aktuell verarbeiteten Datei aus. Danach ruft sie die Funktion »read_swim_data« auf. Wenn etwas danebengeht, ist der letzte ausgegebene Dateiname derjenige, der das Problem verursacht hat.

```
for s in swim_files:
    print("Processing:", s)
    swimclub.read_swim_data(s)
```

```
Processing: Hannah-13-100m-Free.txt
Processing: Darius-13-100m-Back.txt
Processing: Owen-15-100m-Free.txt
Processing: Mike-15-100m-Free.txt
Processing: Hannah-13-100m-Back.txt
Processing: Mike-15-100m-Back.txt
Processing: Mike-15-100m-Fly.txt
Processing: Abi-10-50m-Back.txt
```

Keine dieser Dateien hat einen Fehler verursacht.

Dies ist der Name der problematischen Datei. Wenn Sie sich die unten stehende Fehlermeldung ansehen, erkennen Sie, dass der erste grüne Pfeil auf die Codezeile zeigt, die den Absturz verursacht hat ...

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/Users/barryp/Desktop/THIRD/Learning/Files.ipynb Cell 27 in <cell line: 1>()
      1 for s in swim_files:
      2     print("Processing:", s)
----> 3     swimclub.read_swim_data(s)

File ~/Desktop/THIRD/Learning/swimclub.py:15, in get_swim_data(fn)
     13 converts = []
     14 for t in times:
---> 15     minutes, rest = t.split(":")
     16     seconds, hundredths = rest.split(".")
     17     converts.append(int(minutes) * 60 * 100 + int(seconds) * 100 + int(hundredths))

ValueError: not enough values to unpack (expected 2, got 1)
```

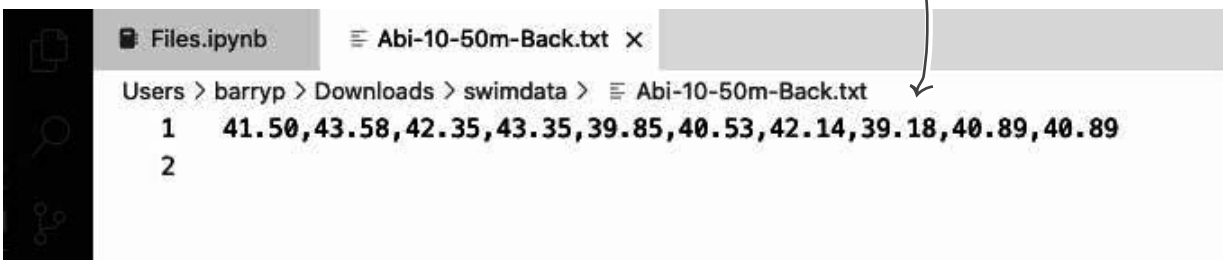
... und die Codezeile vor dem Absturz gibt den aktuellen Dateinamen aus. Hier liegt also unsere Fehlerquelle.

Das ist eine seltsame Fehlermeldung, oder?

Liegt das Problem bei Ihren Daten oder Ihrem Code?

Nachdem wir die fehlerhafte Datei gefunden haben, sehen wir uns ihren Inhalt an, um die Ursache des Problems zu finden. Hier haben wir die Datei *Abi-10-50m-Back.txt* in VS Code geöffnet:

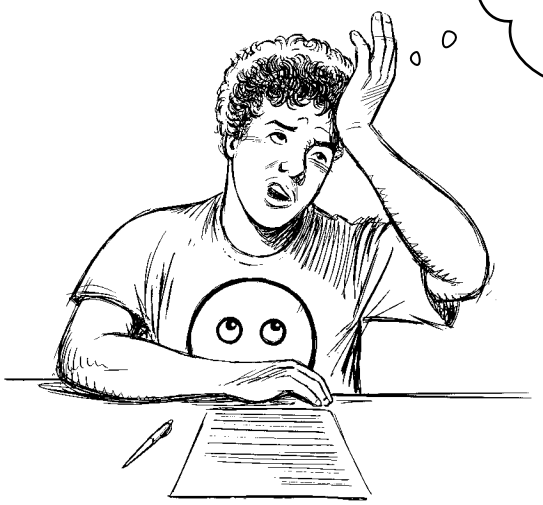
Die Daten sehen in Ordnung aus ...



Hier ist die Codezeile, die den Fehler auslöst. Sehen Sie, wo das Problem liegt?

```
--> 15     minutes, rest = t.split(':')
```

Nicht vergessen: Dieser Code beklagt sich darüber, dass es nicht genug Werte zu entpacken (>>not enough values to unpack<<) gibt.



Natürlich, jetzt sehe ich es auch. Abi ist nur 50 Meter geschwommen. Daher liegt keine der aufgezeichneten Zeiten über der Ein-Minutenmarke. Der Code geht aber davon aus, dass die Schwimmzeit einen Wert für die Minuten enthält. Mist!

Das Problem ist eine falsche Annahme.

Ihr Code in seiner jetzigen Form geht davon aus, dass alle Schwimmzeiten im Format *Minuten:Sekunden.Hundertstelsekunden* vorliegen. Das ist bei Abis Zeiten über die 50-Meter-Distanz aber offensichtlich nicht so. Und das ist auch der Grund für den **`ValueError`**.

Nachdem Sie nun wissen, worin das Problem besteht, finden Sie auch die Lösung?

Bürogespräch

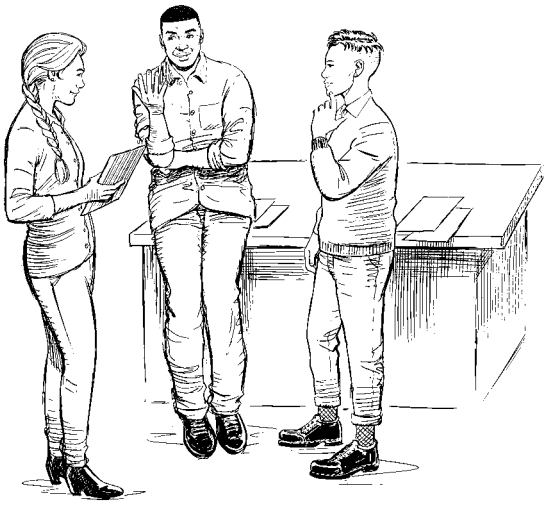
Sam: Also, welche Möglichkeiten haben wir?

Alex: Wir könnten zum Beispiel die Daten anpassen, richtig?

Mara: Und wie?

Alex: Wir könnten die einzelnen Dateien verarbeiten und sicherstellen, dass es keine fehlenden Minutenwerte gibt – zum Beispiel indem wir dem Eintrag eine Null und einen Doppelpunkt voranstellen, wenn die Minuten fehlen. So müssten wir keine Änderungen am Code vornehmen.

Mara: Das könnte funktionieren, aber …



Sam: … es wäre ziemlich unordentlich. Ich bin auch nicht wirklich scharf darauf, alle Dateien einem »Preprocessing« zu unterziehen. Schließlich müssen an der großen Mehrheit der Dateien gar keine Änderungen vorgenommen werden. Das scheint mir eine Verschwendung von Ressourcen zu sein.

Mara: Und selbst wenn wir den vorhandenen Code nicht verändern, müssten wir immer noch den Code schreiben, der das Preprocessing übernimmt, vielleicht als separates Hilfsprogramm.

Sam: Wir sollten auch nicht vergessen, dass die Daten in einem festgelegten Format vorliegen. Sie kommen von der Smart-Stoppuhr des Coachs. Ich finde, wir sollten wirklich nicht an den Daten herumpfuschen, sondern sie möglichst unangetastet lassen.

Alex: Dann müssen wir also unsere `read_swim_data`-Funktion anpassen, richtig?

Mara: Ja, ich glaube, das ist die bessere Strategie.

Sam: Ich auch.

Alex: Gut. Wie sollen wir vorgehen?

Mara: Wir müssen herausfinden, wo im Code Änderungen nötig sind …

Sam: … und wie diese Änderungen aussehen müssen.

Alex: Das klingt doch schon ganz gut. Wir untersuchen also unsere `read_swim_data`-Funktion, damit wir entscheiden können, welcher Code sich ändern muss?

Mara: Ja. Dann können wir eine **if**-Anweisung einsetzen, um eine Entscheidung zu treffen, die davon abhängt, ob die gerade verarbeitete Schwimmzeit einen Minutenwert enthält oder nicht.

Spitzen Sie Ihren Bleistift

Hier sehen Sie den aktuellen Code unseres `swimclub`-Moduls.

Schnappen Sie sich Ihren Bleistift und kreisen Sie den Codeteil ein, in den Ihrer Meinung nach die **if**-Anweisung eingebaut werden muss.

```
import statistics

FOLDER = "swimdata/"


def read_swim_data(filename):
    swimmer, age, distance, stroke = filename.removesuffix(".txt").split("-")
    with open(FOLDER + filename) as file:
        lines = file.readlines()
        times = lines[0].strip().split(",")
    converts = []
    for t in times:
        minutes, rest = t.split(":")
        seconds, hundredths = rest.split(".")
        converts.append((int(minutes) * 60 * 100) + (int(seconds) * 100) + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths = str(round(average / 100, 2)).split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes * 60
    average = str(minutes) + ":" + str(seconds) + "." + hundredths

    return swimmer, age, distance, stroke, times, average
```

Unsere Auswahl finden Sie auf der nächsten Seite. ⟶

Spitzen Sie Ihren Bleistift Lösung

von Seite 161

Auf der vorherigen Seite haben wir Ihnen den aktuellen Code unseres `swimclub`-Moduls gezeigt.

Sie sollten sich Ihren Bleistift schnappen und den Codeteil einkreisen, in den Ihrer Meinung nach die **if**-Anweisung eingebaut werden muss.

```
import statistics

FOLDER = "swimdata/"


def read_swim_data(filename):
    swimmer, age, distance, stroke = filename.removesuffix(".txt").split("-")
    with open(FOLDER + filename) as file:
        lines = file.readlines()
        times = lines[0].strip().split(",")
    converts = []
    for t in times:
        minutes, rest = t.split(":")
        seconds, hundredths = rest.split(".")
        converts.append((int(minutes) * 60 * 100) + (int(seconds) * 100) + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths = str(round(average / 100, 2)).split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes * 60
    average = str(minutes) + ":" + str(seconds) + "." + hundredths

    return swimmer, age, distance, stroke, times, average
```

Dies ist die Stelle, an der wir die >>if<<-Anweisung einbauen würden, um zu entscheiden, welches der beiden Formate für die Schwimmzeit gerade verarbeitet wird.

Entscheidungen über Entscheidungen

Genau das ist es, was **if**-Anweisungen Tag für Tag machen: Sie treffen Entscheidungen.

Ja, genau das brauchen wir hier.

Sehen wir uns die beiden möglichen Formate für die Schwimmzeiten noch einmal an:

Zuerst eine der Zeiten aus der Datei für Darius:

```
'1:30.96'
```

Und dies ist eine Zeitangabe aus Abis Daten:

```
'43.35'
```

Der Unterschied ist leicht zu erkennen: *Abis Daten enthalten keine Minuten*. Mit dieser Information im Hinterkopf sollte es möglich sein, eine Bedingung zu finden, die wir für die Entscheidung überprüfen können. Finden Sie heraus, welche das ist? (Tipp: Ihr bester Freund, der Doppelpunkt, könnte weiterhelfen.)

Suchen wir den Doppelpunkt »in« dem String

Enthält der String mit der Schwimmzeit einen Doppelpunkt, enthält er einen Minutenwert. Zwar besitzen Strings eine Vielzahl eingebauter Methoden, inklusive der für die Suche, aber dennoch wollen wir hier etwas anderes verwenden. Suchen kommen so häufig vor, dass Python hierfür den speziellen Operator **in** besitzt. Sie haben **in** schon zusammen mit **for** gesehen:

Die String-Methoden »find« und »index« können beide eine Suche durchführen.

In diesem Beispiel iteriert die »for«-Schleife über die Liste »times«.

```
for t in times:
    print(t)
```

Das Schlüsselwort »in« steht direkt vor dem Namen der Folge, über die iteriert werden soll.

Durch die Verwendung von **in** mit **for** wird die Folge angegeben, über die iteriert werden soll. Wird **in** dagegen außerhalb einer Schleife benutzt, werden seine *Suchfähigkeiten* aktiviert. Hier ein paar Beispiele für den Gebrauch von **in**:

»in« verwenden, um auf die Existenz eines Substrings innerhalb eines längeren Strings zu testen.

```
"ell" in "Hello there!"
```

```
True
```

»True« steht für Erfolg!

```
"fell" in "Hello there!"
```

```
False
```

»False« steht für einen Fehlschlag.

Nach einem Wert in einer Liste mit Werten suchen.

```
42 in ["Forty two", 42, "42"]
```

```
True
```

Dieses Beispiel ist nicht ganz so einfach. Hier wird im ersten Element der Liste nach dem String »two« gesucht. Beachten Sie, dass wir per [0] auf das erste Element zugreifen.

```
"two" in ["Forty two", 42, "42"][0]
```

```
True
```

Erscheint die Liste links des Schlüsselworts »in« innerhalb der Liste auf der rechten Seite?

```
[1, 2, 3] in ['a', 'b', 'c', [1, 2, 3], 'd', 'e', 'f']
```

```
True
```

```
[1, 2] in ['a', 'b', 'c', [1, 2, 3], 'd', 'e', 'f']
```

```
False
```

Das Schlüsselwort »in« hat es wirklich in sich! Ich konnte alle sechs Suchen durchführen, ohne eine einzige Schleife schreiben zu müssen. Das muss man einfach lieben!

Und auch wir lieben das Schlüsselwort »in«.

Es ist eine Python-Superkraft.

Spitzen Sie Ihren Bleistift

In Kapitel 0 haben wir die allgemeine Struktur einer einfachen **if**-Anweisung bereits vorgestellt:

```
if <Bedingung>:
    auszuführender Code, wenn <Bedingung> True (wahr) ist
else:
    auszuführender Code, wenn <Bedingung> False (falsch) ist
```

Fällt Ihnen jetzt, da Sie sich mit dem Schlüsselwort **in** auskennen, eine Bedingungsanweisung ein, die überprüft, ob die aktuelle Schwimmzeit (gespeichert in der Variablen `t`) einen Doppelpunkt enthält? Falls ja, schreiben Sie sie auf die unten stehende Leerzeile:

Welche Bedingung muss hier stehen?

```
if _________:
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
```

Dieser Codeblock wird ausgeführt, wenn die Bedingung »True« (wahr) ist.

Antworten auf Seite 166

Spitzen Sie Ihren Bleistift Lösung

von Seite 165

Dies ist die allgemeine Struktur einer einfachen **if**-Anweisung:

```
if <Bedingung>:
           auszuführender Code, wenn <Bedingung> True (wahr) ist
       else:
           auszuführender Code, wenn <Bedingung> False (falsch) ist
```

Jetzt, da Sie sich mit dem Schlüsselwort **in** auskennen, haben wir Sie gebeten, eine Bedingungsanweisung zu finden, die überprüft, ob die aktuelle Schwimmzeit (gespeichert in der Variablen `t`) einen Doppelpunkt enthält, und diese in der unten stehenden Leerzeile einzutragen:

Dies ist ein einfacher Test, der die Fähigkeiten des Schlüsselworts »in« nutzt.

```
if ":" in t :
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
```

Sorgen Sie dafür, dass Ihre if-Anweisungen »pythonisch« geschrieben sind!

Wenn Sie vor Python mit einer C-artigen Sprache gearbeitet haben, juckt es Ihnen vermutlich in den Fingern, die Bedingung zusätzlich mit runden Klammern zu umgeben. Oder Sie haben das Bedürfnis, ausdrücklich zu überprüfen, ob die Bedingung zu **True** *(oder* **False***) ausgewertet wird. Der Python-Interpreter wird Sie nicht davon abhalten, Code wie den unten gezeigten zu schreiben. Widerstehen Sie dieser Versuchung! Unter allen Umständen!*

Es gibt nur ein Wort, das diese beiden Versionen einer »if«-Anweisung treffend beschreibt: Pfui!

```
if (":" in t):
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
```

```
if (":" in t) == True:
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
```

Übung

Die letzte Aufgabe besteht darin, den Code auszuarbeiten, der ausgeführt wird, wenn die Bedingung als **False** ausgewertet wird. Hier ist Ihre bisherige **if**-Anweisung. Kommen Sie auf den Code, der in den **else**-Codeblock eingefügt werden muss? Experimentieren Sie in Ihrem Notebook und fügen Sie dann den unten stehenden Code ein. Unsere Lösung steht wie immer auf der nächsten Seite.

```
if ":" in t:
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
else:
```

Schreiben Sie hier den Code auf, der ausgeführt wird, wenn der »else«-Block läuft.

Antworten auf Seite 168

Es gibt keine Dummen Fragen

F: Ich vermute mal, `True` und `False` sind Pythons boolesche Werte, richtig? Kann ich stattdessen auch 1 und 0 benutzen?

A: Mehr oder weniger. In einem *booleschen Kontext* wird `1` zu `True` ausgewertet und `0` zu `False`. Allerdings verwenden Python-Programmierer `1` und `0` nur selten auf diese Weise. Das hat damit zu tun, dass in Python jeder Wert in einem booleschen Kontext verwendet werden kann.

F: Kann man herauszufinden, wozu ein Wert in einem booleschen Kontext ausgewertet wird?

A: Ja, hierfür gibt es die eingebaute Funktion mit dem naheliegenden Namen **bool**. Sie können sie interaktiv (oder in Ihrem Code) einsetzen, um beliebige Werte auf ihre boolesche Auswertung zu überprüfen.

F: Wie sieht es mit den Begriffen `true` und `false` aus, also komplett in Kleinbuchstaben geschrieben? Wie werden diese Werte ausgewertet?

A: Nicht so, wie Sie es vermutlich erwarten. Die Verwendung von `true` und `false` ist eine schlechte Idee, weil die Groß- und Kleinschreibung in Python *wichtig* ist. `True` und `False` sind boolesche Werte, `true` und `false` dagegen nicht. Beide sind gültige Variablennamen, aber *keine* booleschen Werte. Und als Variablen existieren sie, was bedeutet, dass sie *immer* zu `True` ausgewertet werden (wie kann etwas, das existiert, etwas anderes als `True` sein?). Trotzdem stimmen wir Ihnen zu, dass es ein wenig seltsam ist, wenn etwas wie der String `"false"` zu `True` ausgewertet wird. Lektion fürs Leben: Benutzen Sie auf keinen Fall `true` oder `false` als boolesche Werte, sondern immer `True` und `False`.

Übung, Lösung

von Seite 167

Abschließend sollten Sie herausfinden, welcher Code ausgeführt werden muss, wenn die Bedingung zu **False** ausgewertet wird. Wir haben Ihnen die bisherige **if**-Anweisung gezeigt, Sie sollten den Code finden, der für den **else**-Codeblock gebraucht wird. Hier sehen Sie unser Ergebnis. Sieht Ihr Code ähnlich aus, ist er gleich, oder ist er vollkommen anders?

```
if ":" in t:
    minutes, rest = t.split(":")
    seconds, hundredths = rest.split(".")
else:
    minutes = 0
    seconds, hundredths = t.split(".")
```

Wurde im Zeitstring kein Doppelpunkt gefunden, wird der Wert von >>minutes<< auf null gesetzt.

Wurden keine Minuten aufgezeichnet, müssen Sie den Zeitstring (in der Variablen >>t<<) am Punkt (>>.<<) auftrennen. So erhalten Sie die Werte für Sekunden (>>seconds<<) und Hunderstelsekunden (>>hundredths<<).

Wir haben es fast geschafft. Nur noch eine kleine Änderung.

Fügen Sie den oben gezeigten Code oberhalb Ihrer `read_swim_data`-Funktion in das `swimclub`-Modul ein und vergessen Sie nicht, die Datei zu **speichern**.

Ihr Code in *swimclub.py* sollte jetzt genau so aussehen, wie auf der gegenüberliegenden Seite gezeigt.

Dies ist Ihr >>swimclub.py<<-Code, inklusive des neu hinzugefügten >>if/else<<-Teils.

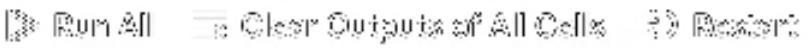
```
import statistics

FOLDER = "swimdata/"


def read_swim_data(filename):
    swimmer, age, distance, stroke = filename.removesuffix(".txt").split("-")
    with open(FOLDER + filename) as file:
        lines = file.readlines()
        times = lines[0].strip().split(",")
    converts = []
    for t in times:
        if ":" in t:
            minutes, rest = t.split(":")
            seconds, hundredths = rest.split(".")
        else:
            minutes = 0
            seconds, hundredths = t.split(".")
        converts.append((int(minutes) * 60 * 100) + (int(seconds) * 100) + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths = str(round(average / 100, 2)).split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes * 60
    average = str(minutes) + ":" + str(seconds) + "." + hundredths

    return swimmer, age, distance, stroke, times, average
```

Fügen Sie den >>if/else<<-Code am Anfang des Codeblocks ein, der zur >>for<<-Schleife gehört, und achten Sie auf die Einrückungen. Stellen Sie sicher, dass Ihr Code dem hier gezeigten entspricht.

Bevor Sie Ihren Code speichern, sollten Sie sicherstellen, dass Sie die beiden Codezeilen entfernt haben, die zuvor die Zuweisungen auf die Variablen >>minutes<<, >>seconds<< und >>hundredths<< vorgenommen haben. Stattdessen sollte hier jetzt, wie gezeigt, die >>if/else<<-Anweisung stehen.

Probefahrt

Um die letzte Version Ihres `swimclub`-Moduls auszuführen, müssen Sie zunächst Ihre Notebook-Session neu starten. Am einfachsten geht das, indem Sie alle Ausgaben Ihrer Notebook-Zellen ausleeren, das Notebook neu starten und dann die Zellen erneut ausführen. Im oberen Teil des VS-Code-Fensters sehen Sie diese drei Optionen, die Sie in der gezeigten Reihenfolge anklicken sollten (erst 1, dann 2, dann 3):



3. Dieser Button führt alle Zellen Ihres Notebooks der Reihe nach (von oben nach unten) aus.

1. Klicken Sie hier zuerst, um alle vorherigen Ausgaben aus dem Notebook zu entfernen.

2. Starten Sie das Notebook neu. Hierdurch werden alle früheren Ausführungen, Importe und Zuweisungen aus dem Speicher gelöscht.

Sobald Sie die oben gezeigten Schritte abgeschlossen haben, erzeugt Ihre **for**-Schleife (hoffentlich) 60 Ausgabezeilen. Diesmal gibt es keinen Laufzeitfehler (sofern keine anderen Codezellen einen Fehler auslösen).

```
for s in swim_files:
    print("Processing:", s)
    swimclub.read_swim_data(s)
```

```
Processing: Hannah-13-100m-Free.txt
Processing: Darius-13-100m-Back.txt
Processing: Owen-15-100m-Free.txt
Processing: Mike-15-100m-Free.txt
Processing: Hannah-13-100m-Back.txt
Processing: Mike-15-100m-Back.txt
Processing: Mike-15-100m-Fly.txt
Processing: Abi-10-50m-Back.txt
Processing: Ruth-13-200m-Free.txt
```

Diesmal wird Abis Datei fehlerfrei verarbeitet. Ju-huu!

⋮

Alles scheint in Ordnung. Aber wie können Sie sicher sein, dass wirklich alle 60 Dateien verarbeitet wurden?

Sind Sie auf 60 verarbeitete Dateien gekommen?

Vermutlich sind Sie sich jetzt sicher, dass der aktuelle Code alle Dateien im *swimdata*-Ordner verarbeitet hat. Das sind wir auch. Trotzdem kann eine zusätzliche Überprüfung sinnvoll sein. Wie immer kann auch das auf verschiedene Weise geschehen. Wir beginnen, indem wir Ihre **for**-Schleife so erweitern, dass den auf dem Bildschirm ausgegebenen Ergebniszeilen fortlaufende Nummern (beginnend mit 1) vorangestellt werden.

Für diesen Zweck gibt es eine eigene eingebaute Funktion mit dem treffenden Namen **enumerate** (aufzählen):

Die >>enumerate<<-BIF versieht jede Iteration von >>swim_files<< mit einer fortlaufenden Nummer.

Die >>for<<-Schleife verwendet jetzt zwei Schleifenvariablen: >>s<< (für >>swimfile<<) enthält den Dateinamen der aktuellen Iteration, und >>n<< (für >>Nummer<<) enthält den aktuellen Wert des Zählers.

```
for n, s in enumerate(swim_files, 1):
    print(n, "Processing:", s)
    swimclub.read_swim_data(s)
```

```
1 Processing: Hannah-13-100m-Free.txt
2 Processing: Darius-13-100m-Back.txt
3 Processing: Owen-15-100m-Free.txt
4 Processing: Mike-15-100m-Free.txt
5 Processing: Hannah-13-100m-Back.txt
6 Processing: Mike-15-100m-Back.txt
7 Processing: Mike-15-100m-Fly.txt
8 Processing: Abi-10-50m-Back.txt
9 Processing: Ruth-13-200m-Free.txt
```

⋮

```
58 Processing: Blake-15-100m-Fly.txt
59 Processing: Erika-15-100m-Breast.txt
60 Processing: Katie-9-100m-Back.txt
```

Standardmäßig beginnt die Zählung von >>enumerate<< bei 0. Hier weisen wir sie ausdrücklich an, mit 1 zu beginnen.

Das sieht gut aus: Die >>print<<-Anweisung der Schleife enthält nun zusätzlich (dank der eingebauten Funktion >>enumerate<<) den aktuellen Wert von >>n<<. Die Ausgaben bestätigen, dass tatsächlich 60 Dateien verarbeitet werden.

Es gibt keine Dummen Fragen

F: Warum müssen wir die Session denn noch einmal neu starten? Eigentlich reicht es doch, `import swimclub` **in eine leere Codezelle einzugeben, damit der Code des zuvor geladenen Moduls aktualisiert wird, oder?**

A: (Wir hoffen, Sie sitzen stabil!) Nein, das tut es nicht. Wie wir bereits besprochen haben, verwendet der Python-Interpreter ein aggressiv implementiertes Caching für importierte Module. Es verbietet kategorisch den erneuten Import eines bereits geladenen Moduls, selbst wenn sich der Code des Moduls in der Zwischenzeit geändert hat. Es gibt zwar einige Techniken, um das standardmäßige Caching von Modulen zu unterbinden, aber unserer Erfahrung nach ist es am sichersten, das Notebook nach Änderungen am Modulcode *grundsätzlich* neu zu starten. So ist garantiert, dass Sie wirklich den gewünschten Code ausführen.

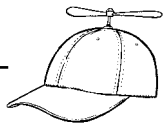
Der Code für den Coach nimmt langsam Form an …

Ihr `swimclub`-Modul ist jetzt bereit. Wenn Sie ihm den Namen einer Datei übergeben, die eine Reihe von Strings mit Schwimmzeiten enthält, kann Ihr Modul daraus nutzbare Daten erzeugen. Der Coach erwartet, dass aus diesen Daten Balkendiagramme erzeugt werden. Diese Funktionalität wollen wir im folgenden Kapitel implementieren.

Wie immer sollten Sie sich zuerst die Zusammenfassung dieses Kapitels ansehen und dann Ihr Glück mit unserem Kreuzworträtsel versuchen, bevor Sie mit dem nächsten Kapitel weitermachen.

Punkt für Punkt

- Das Schlüsselwort **def** definiert eine neue, selbst erstellte Funktion.
- Wenn Sie Ihren Code in seine eigene Datei schreiben (mit der Dateiendung *.py*), erstellen Sie ein **Modul**.
- Mit der **import**-Anweisung, zum Beispiel `import swimclub`, können Sie ein Modul wiederverwenden.
- Verwenden Sie einen **voll qualifizierten Namen**, um eine Funktion aus einem Modul aufzurufen, zum Beispiel `swimclub.read_swim_data`.
- Mit einer **return**-Anweisung kann eine selbst erstellte Funktion ein Ergebnis zurückgeben.
- Versucht eine Funktion, mehr als ein Ergebnis zurückzugeben, werden die Rückgabewerte zu einem einzelnen **Tupel zusammengefasst**. Der Grund ist, dass Python-Funktionen grundsätzlich nur ein Ergebnis zurückgeben.
- Die Datenstruktur eines Tupels ist eine **immutable Folge (Sequenz)**. Sobald einem Tupel Werte zugewiesen wurden, kann es nicht mehr verändert werden.
- Listen verhalten sich wie Tupel. Der einzige Unterschied ist, dass Listen **mutabel** sind, also verändert werden können.
- Mit dem `os`-Modul (Teil der PSL) kann Ihr Code mit dem zugrunde liegenden Betriebssystem kommunizieren.
- Obwohl Listen eine eigene **sort**-Methode mitbringen, sollten Sie bei ihrer Verwendung vorsichtig sein, denn die Sortierung findet *an Ort und Stelle* statt. Wollen Sie die aktuelle Reihenfolge einer Liste beibehalten, sollten Sie stattdessen die eingebaute Funktion **sorted** verwenden (die immer eine sortierte Kopie Ihrer Daten zurückgibt).
- Listen besitzen eine Vielzahl eingebauter Methoden (nicht nur **sort**), inklusive der hilfreichen **remove**-Methode.
- Der Operator **in** ist einer unserer Favoriten, und das sollte er auch für Sie sein. Er eignet sich sehr gut, um Dinge zu suchen (beziehungsweise die *Überprüfung auf Mitgliedschaft*).
- Wenn Sie eine Entscheidung treffen müssen, ist die Kombination aus **if** und **else** einfach unschlagbar.
- Eine oft übersehene und trotzdem wunderbare BIF ist **enumerate**. Mit ihr können die Iterationen einer **for**-Schleife nummeriert werden.



Geek-Tipp

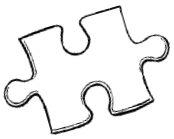
Der Code im `swimclub`-Modul funktioniert. Durch die sinnvolle Platzierung von hilfreichen Kommentaren kann er noch verbessert werden. Hier sehen Sie eine weitere Version von `swimclub.py`, in der wir genau das getan haben. Sie können selbst entscheiden, ob Sie Ihren Code mit diesen (oder ähnlichen) Kommentaren versehen. Sie sollten aber wissen, dass der gesamte auf der GitHub-Seite zu diesem Buch verfügbare Code in englischer Sprache kommentiert ist.

Diese Art von Kommentar wird als »Docstring« bezeichnet und wird oft eingesetzt, um einen mehrzeiligen Kommentar am Anfang einer Funktion einzufügen. Beachten Sie die drei doppelten Anführungszeichen, die den Kommentar umgeben. Mehr zu Docstrings finden Sie unter dieser Adresse: https://peps.python.org/pep-0257.

```
import statistics

FOLDER = "swimdata/"


def read_swim_data(filename):
    """Return swim data from a file.

    Given the name of a swimmer's file (in filename), extract all the required
    data, then return it to the caller as a tuple.
    """
    swimmer, age, distance, stroke = filename.removesuffix(".txt").split("-")
    with open(FOLDER + filename) as file:
        lines = file.readlines()
        times = lines[0].strip().split(",")
    converts = []
    for t in times:
        # The minutes value might be missing, so guard against this causing a crash.
        if ":" in t:
            minutes, rest = t.split(":")
            seconds, hundredths = rest.split(".")
        else:
            minutes = 0
            seconds, hundredths = t.split(".")
        converts.append((int(minutes) * 60 * 100) + (int(seconds) * 100) + int(hundredths))
    average = statistics.mean(converts)
    mins_secs, hundredths = str(round(average / 100, 2)).split(".")
    mins_secs = int(mins_secs)
    minutes = mins_secs // 60
    seconds = mins_secs - minutes * 60
    average = str(minutes) + ":" + str(seconds) + "." + hundredths

    return swimmer, age, distance, stroke, times, average  # Returned as a tuple.
```

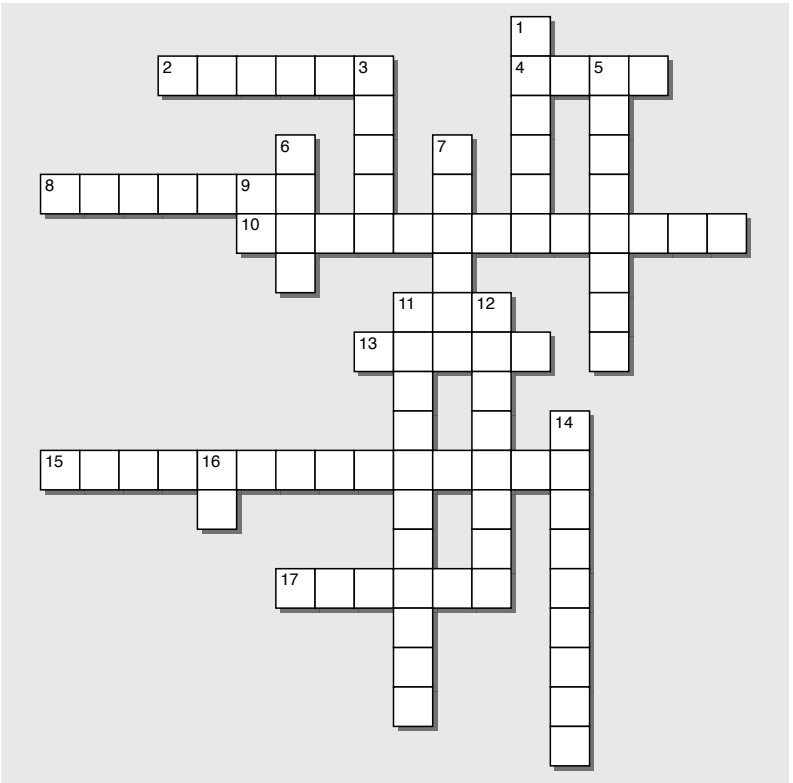
Einzeilige Kommentare beginnen mit einem Doppelkreuz (»#«) und gehen bis zum Ende der aktuellen Zeile.

Kommentare können auch ans Ende einer Codezeile gesetzt werden.



Das Python-Kreuzworträtsel

Die Antworten zu den Hinweisen finden Sie auf den Seiten dieses Kapitels, die Lösung erhalten Sie wie immer auf der folgenden Seite.



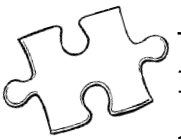
Waagerecht

2. Lädt den Code aus einer Datei.
4. Der »unwahr«-Teil von 9 senkrecht.
8. Aus dem os-Modul: Gibt eine Liste des Ordnerinhalts zurück.
10. Was Sie erhalten, wenn es nicht genug Werte zum Entpacken gibt.
11. Dieses Schlüsselwort leitet 12 senkrecht ein.
13. Ein Ort, an dem 12 senkrecht abgelegt werden kann.
15. Seien Sie explizit mit einem voll _____ Namen.
17. Kann die letzte Zeile von 12 senkrecht einleiten.

Senkrecht

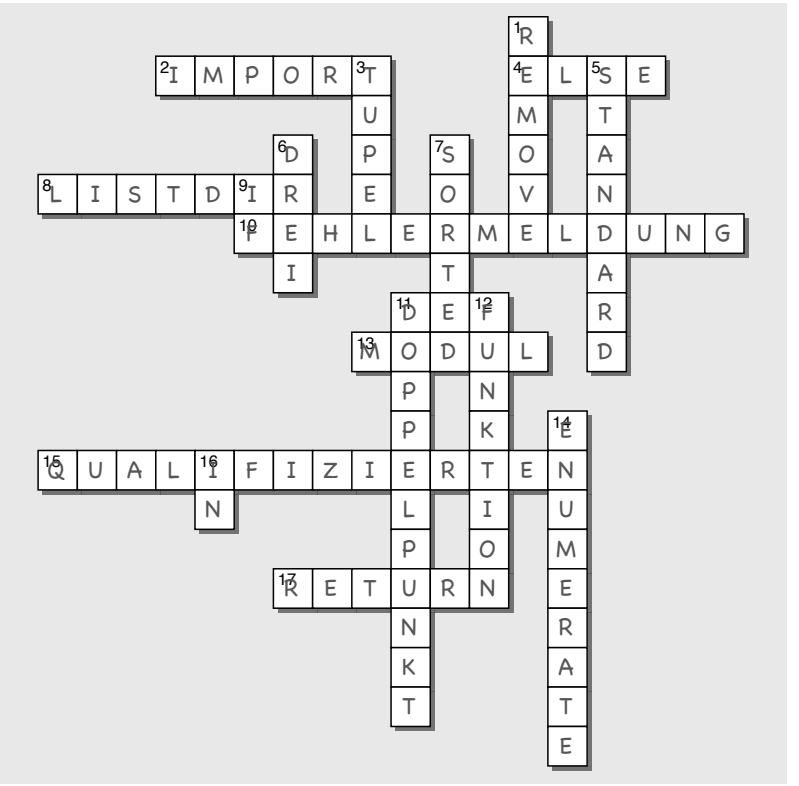
1. Findet und löscht einen Wert aus einer Liste.
3. Verhält sich wie eine immutable Liste.
5. Der Unterstrich steht für die _____-Variable.
6. Docstrings werden von _____ Paaren doppelter Anführungszeichen umgeben.
7. Eingebaute Funktion zum Ordnen Ihrer Daten (ohne die aktuelle Reihenfolge zu ändern).
9. Wird verwendet, um Entscheidungen zu treffen.
11. Ihr neuer bester Freund.
12. Ein benanntes Codestück.
14. Eine BIF, die Iterationen nummerieren kann.
16. Ein mächtiger, kleiner Operator.

Antworten auf Seite 176



Das Python-Kreuzworträtsel, Lösung

von Seite 176



Der Inhalt (im Überblick)

Der Inhalt (jetzt ausführlich)

Intro

Ihr Gehirn und Python. *Sie* versuchen, etwas zu *lernen*, und Ihr *Hirn* tut sein Bestes, damit das Gelernte nicht *hängen bleibt*. Es denkt nämlich: »Wir sollten lieber ordentlich Platz für wichtigere Dinge lassen, z. B. für das Wissen darüber, welche Tiere einem gefährlich werden könnten, oder dass es eine ganz schlechte Idee ist, nackt Snowboard zu fahren.« Tja, wie schaffen wir es nun, Ihr Gehirn davon zu überzeugen, dass Ihr Leben davon abhängt, wie man in Python programmiert?

0 Warum Python?
Ähnlich und doch anders

Wie Sie vermutlich schon wissen, beginnt Python mit dem Zählen bei null. Python hat eine Menge mit anderen Programmiersprachen **gemeinsam**. Es gibt **Variablen**, **Schleifen**, **Bedingungen**, **Funktionen** und so weiter. In diesem Eröffnungskapitel nehmen wir Sie mit auf eine **kleine Besichtigungstour**, die Ihnen einen **Überblick** über die Grundlagen von Python vermitteln soll. Das heißt, wir zeigen Ihnen die Sprache, aber ohne zu sehr ins Detail zu gehen. Sie werden lernen, mit Jupyter Notebook (innerhalb von VS Code) eigenen Code zu **schreiben** und **auszuführen**. Dabei werden Sie staunen, wie viel Programmierfunktionalität direkt in Python **eingebaut** ist. Das werden Sie **nutzen**, um verschiedene Aufgaben zu erledigen. Außerdem erfahren Sie, dass Python viele Konzepte mit anderen Programmiersprachen gemeinsam hat, diese aber **etwas anders** umsetzt. Verstehen Sie uns hier nicht falsch: Wir meinen hier die **guten** Unterschiede, nicht die *schlechten*. Lesen Sie weiter, um mehr zu erfahren.

1 Eintauchen
Sprung ins kalte Wasser

Eine neue Sprache lernt man am besten, indem man Code schreibt.
Und wenn Sie Code schreiben wollen, brauchen Sie ein **echtes** Problem, das es zu lösen gilt. Wie der Zufall es will, gibt es in diesem Kapitel eins. Hier beginnen Sie Ihre Karriere in der Applikationsentwicklung mit Python, indem Sie zusammen mit unserem freundlichen **Schwimmcoach** einen Sprung ins kalte Wasser wagen. Sie beginnen mit Pythons **Strings** und wie Sie sie nach Herzenslust **manipulieren** können. Unterwegs erstellen Sie eine Python-basierte Lösung für das Problem des Coachs. Außerdem erfahren Sie mehr über Pythons eingebaute **Listen**-Datenstruktur. Sie lernen, wie **Variablen** funktionieren und was Pythons **Fehlermeldungen** bedeuten, ohne dass Sie hierfür gleich einen Tauchschein brauchen. Und das alles, während wir ein *echtes* Problem mit *echtem* Python-Code lösen. Also, nichts wie rein – und zwar kopfüber!

2 Listen aus Zahlen
Listendaten verarbeiten

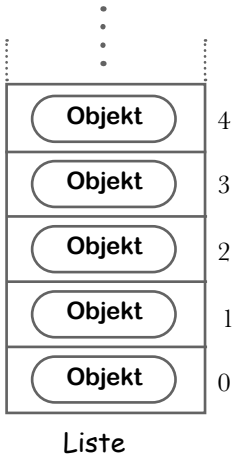
Je mehr Code Sie schreiben, desto besser werden Sie. Ganz einfach.

Auch in diesem Kapitel schreiben Sie Python-Code, um dem Coach zu helfen. Sie lernen, wie Sie Daten aus der **Datei** des Coachs **lesen** und die enthaltenen Zeilen in einer **Liste**, einer von Pythons eingebauten **Datenstrukturen**, speichern können. Neben der Erstellung von Listen aus Daten in einer Datei lernen Sie auch, Listen von Grund auf neu zu erstellen und bei Bedarf **dynamisch wachsen** zu lassen. Außerdem werden Sie Listen mit einer von Pythons beliebtesten Schleifenkonstrukten, der **for**-Schleife, verarbeiten. Sie werden Daten aus einem Datenformat in ein anderes **konvertieren,** und Sie werden einen neuen besten Freund (Ihren eigenen Python-**BFF**) kennenlernen. Nach Kaffee und Kuchen ist es jetzt Zeit, die Ärmel hochzukrempeln und sich wieder an die Arbeit zu machen.

3 Listen von Dateien
Funktionen, Module und Dateien

Ihr Code kann nicht ewig in einem Notebook leben. Er will frei sein.

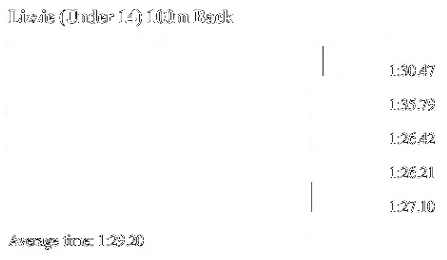
Und wenn es darum geht, Ihren Code zu befreien und mit anderen zu **teilen**, dann ist eine selbst erstellte **Funktion** der erste Schritt, auf den kurz darauf ein **Modul** folgt, mit dem Sie Ihren Code organisieren und weitergeben können. In diesem Kapitel werden Sie aus dem bisher geschriebenen Code direkt eine Funktion und auf dem Weg auch gleich ein **gemeinsam nutzbares** Modul erstellen. Ihr Modul wird sich sofort an die Arbeit machen, während Sie **for**-Schleifen, **if**-Anweisungen, Tests auf bestimmte **Bedingungen** sowie die Python-Standardbibliothek, **PSL** (*Python Standard Library*), verwenden, um die Schwimmdaten des Coachs zu verarbeiten. Außerdem werden Sie lernen, Ihre Funktionen zu **kommentieren** (was *immer* eine gute Idee ist). Es gibt viel zu tun, also an die Arbeit!



4 Formatierte String-Literale
Tabellen aus Daten

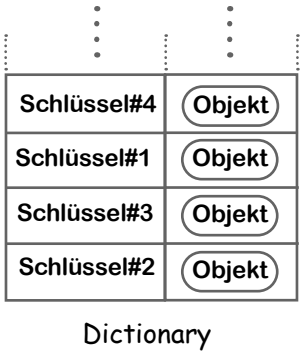
Manchmal ist die einfachste Lösung die beste.

In diesem Kapitel kommen wir endlich dazu, die Balkendiagramme für den Coach zu erstellen. Hierfür werden Sie nichts als **Strings** verwenden. Wie Sie schon wissen, besitzt Python bereits eine Menge **eingebauter Funktionalität**. Pythons **formatierte String-Literale**, auch *f-Strings* genannt, erweitern diese Möglichkeiten noch einmal auf ziemlich clevere Weise. Es klingt vielleicht seltsam, dass wir vorschlagen, Balkendiagramme mit Text zu erstellen, Sie werden aber sehr bald merken, dass es längst nicht so *absurd* ist, wie es scheint. Unterwegs werden Sie Python einsetzen, um **Dateien** zu erstellen und einen Webbrowser zu starten – und das alles mit nur wenigen Zeilen Code. Am Ende bekommt der Coach endlich, was er sich so lange gewünscht hat: die **automatische Erzeugung** von Diagrammen aus seinen Schwimmdaten. Also, nichts wie los!



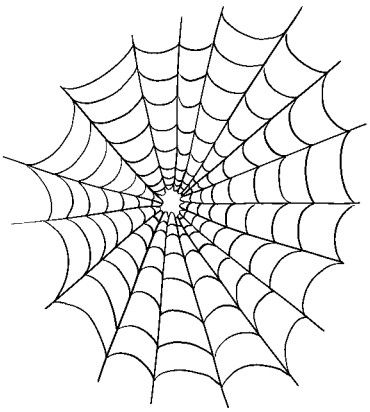
5 Daten organisieren
Die richtige Datenstruktur

Ihr Code muss seine Daten im Arbeitsspeicher ablegen können.
Und bei der Anordnung von Daten **im Arbeitsspeicher** entscheidet oft die Wahl der Datenstruktur darüber, ob es eine unordentliche Lösung gibt, die *irgendwie* funktioniert, oder eine **elegante** Lösung, die *gut* funktioniert. In diesem Kapitel lernen Sie eine weitere Python-Datenstruktur kennen, das **Dictionary**. Es wird oft mit der allgegenwärtigen Liste kombiniert, um **komplexe** Datenstrukturen zu schaffen. Der Coach muss auf einfache Weise die Daten eines beliebigen Schwimmers auswählen können. Mit einem Dictionary wird das zum Kinderspiel!



6 Eine Web-App erstellen
Webentwicklung

Fragen Sie zehn Programmierer, welches Web-Framework Sie nehmen sollen …
… und Sie bekommen wahrscheinlich elf Antworten! 😉 Bei der Webentwicklung mangelt es Python wirklich nicht an technischen *Wahlmöglichkeiten*. Jede von ihnen hat eine eigene loyale und unterstützende Entwicklergemeinschaft. In diesem Kapitel tauchen Sie ein in die Welt der **Webentwicklung**. Sie werden schnell eine Web-App für den Coach bauen, in der man die Balkendiagramme für beliebige Schwimmer betrachten kann. Unterwegs lernen Sie die Verwendung von HTML-**Templates** (Schablonen), Funktions**dekoratoren**, **GET-** und **POST**-HTTP-Methoden und vieles mehr. Es gilt keine Zeit zu verschwenden: Der Coach will endlich sein neues System vorführen. Also an die Arbeit!



7 Bereitstellung
Code überall ausführen

Code auf dem eigenen Rechner auszuführen, ist eine Sache …

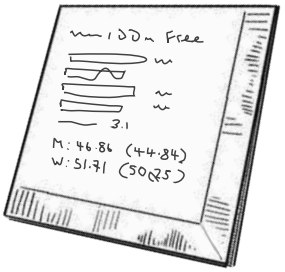
Aber eigentlich wollen Sie Ihren Code so **bereitstellen**, dass auch andere Nutzer ihn einfach weiterverwenden können. Je *geringer* der Aufwand, desto besser. In diesem Kapitel werden Sie die Web-App des Coachs fertigstellen, sie mit etwas **Stil** versehen und sie dann in der **Cloud bereitstellen**. Trotzdem wird die Komplexität nicht ins Unermessliche steigen. In einer früheren Auflage dieses Buchs haben wir damit angegeben, dass die Bereitstellung »etwa zehn Minuten« dauert, und das ist bereits ziemlich schnell … In diesem Kapitel werden Sie für die Bereitstellung nur noch zehn **Schritte** benötigen. Die Cloud wartet schon darauf, die Web-App des Coachs zu hosten. Zeit für die Bereitstellung!

8 Mit HTML arbeiten
Web-Scraping

In einer perfekten Welt wären alle benötigten Daten leicht zugänglich. Das ist aber nur selten der Fall. So werden Daten beispielsweise im Web veröffentlicht. In HTML eingebettete Daten müssen von Webbrowsern **gerendert** und von Menschen **gelesen** werden können. Was aber, wenn Sie diese Daten mit Code **verarbeiten** müssen? Geht das überhaupt? Glücklicherweise ist Python so etwas wie ein Champion, wenn es um das maschinelle Auslesen – das sogenannte ***Scraping*** – von Daten aus Webseiten geht, und in diesem Kapitel werden wir Ihnen zeigen, wie das funktioniert. Sie werden außerdem lernen, wie die ausgelesenen HTML-Seiten **geparst** werden, um nutzbare Daten zu **extrahieren**. Unterwegs werden Ihnen **Slices** (»Scheiben«) und **Soup** (»Suppe«) begegnen, aber keine Sorge, das hier ist immer noch *Python von Kopf bis Fuß* und nicht *Kochen von Kopf bis Fuß*.

9 Mit Daten arbeiten
Datenmanipulation

Manchmal sind die Daten nicht so angeordnet, wie sie gebraucht werden. Vielleicht haben Sie eine *verschachtelte Liste (List of Lists)*, brauchen aber eigentlich ein *verschachteltes Dictionary*. Oder vielleicht müssen Sie einen Wert in einer Datenstruktur mit einem Wert in einer anderen Datenstruktur verbinden, ohne dass sie richtig zusammenpassen. Das kann schnell ziemlich frustrierend werden. Aber keine Sorge: Die Macht von Python steht Ihnen auch hierbei zur Seite. In diesem Kapitel benutzen Sie Python, um Ihre Daten **durch die Mangel zu drehen**. Das Ziel ist es, die von der *Wikipedia*-Website ausgelesenen Daten vom Ende des vorherigen Kapitels in etwas wirklich *Nützliches* umzuwandeln. Sie werden lernen, wie Sie Pythons Dictionary als **Lookup**-Tabelle **nutzen** können. Hier geht es um Umwandlungen, Integrationen, Aktualisierungen, Bereitstellungen und mehr. Und ganz am Ende wird die Spezifikation, die der Coach auf der Papierserviette notiert hat, *Wahrheit*. Wetten, Sie können es kaum abwarten? Wir auch nicht … also los!



9½ Mit ~~Elefanten~~ Dataframes arbeiten

Tabellarische Daten

Manchmal wollen anscheinend alle Daten der Welt tabellarisch sein.
Tabellarische Daten gibt es *überall*. Die Schwimmweltrekorde aus dem vorherigen Kapitel liegen als **tabellarische** Daten vor. Wenn Sie alt genug sind, um sich an Telefonbücher zu erinnern, wissen Sie, dass auch diese Daten tabellarisch sind. Kontoauszüge, Rechnungen, sogar Tabellenkalkulationen sind – Sie haben es natürlich geahnt – tabellarische Daten. In diesem *kurzen* Kapitel lernen Sie einige Dinge über **pandas**, die beliebteste Python-Bibliothek zur Datenanalyse. Leider werden wir hier nur die Spitze des Eisbergs betrachten können. Trotzdem werden Sie genug lernen, um das nächste Mal, wenn Sie mit tabellarischen Daten arbeiten müssen, die am häufigsten verwendete pandas-Datenstruktur nutzen zu können, den **Dataframe**.

10 Datenbanken
Dinge ordnen

Irgendwann müssen Sie die Daten Ihrer Applikationen verwalten.
Und wenn Sie Ihre Daten besser **verwalten** müssen, reicht Python (für sich genommen) möglicherweise nicht aus. In solchen Fällen hilft der Griff zur **Datenbank**-Engine Ihrer Wahl. Um nicht den Überblick zu verlieren, beschränken wir uns an dieser Stelle auf Datenbank-Engines, die das gute alte **SQL** unterstützen. Sie werden hier nicht nur eine Datenbank **erstellen** und darin ein paar **Tabellen** anlegen, Sie werden auch Daten zur Datenbank **hinzufügen**, **daraus auswählen** und **löschen**. Für alle diese Aufgaben werden Sie SQL-Abfragen verwenden, die von Ihrem Python-Code koordiniert werden.

11 Listenabstraktionen
Datenbankintegrationen

Zeit für die Integration Ihrer Datenbanktabellen.

Mithilfe der Tabellen in der Datenbank kann Ihre Web-App die vom Coach benötigte **Flexibilität** erreichen. In diesem Kapitel erstellen wir ein Modul mit verschiedenen **Hilfsfunktionen** (Utilities), über die Ihre Web-App die Datenbank-Engine **nutzen** kann. Außerdem werden Sie eine echte Python-Superkraft kennenlernen, die Ihnen hilft, mit weniger Code mehr zu erreichen: **Listenabstraktionen**. Daneben werden Sie eine Menge Ihres schon vorhandenen Codes auf neue und interessante Weise **wiederverwenden**. Wir haben jede Menge **Integration** vor uns. Also los!


```
[sql for sql in dir(queries) if not sql.startswith("__")]
```

12 Bereitstellung in neuem Licht
Der letzte Schliff

Das Ende vom Anfang Ihrer Python-Reise ist fast erreicht.

In diesem letzten Kapitel passen Sie Ihre Web-App so an, dass sie nicht mehr SQLite, sondern **MariaDB** als Datenbank-Backend verwendet. Danach gibt es noch ein paar kleinere Änderungen, um die neueste Version Ihrer Web-App auf PythonAnywhere bereitzustellen. Dadurch unterstützt das System des Coachs eine **beliebige** Zahl von Schwimmern, die an einer **beliebigen** Zahl von Trainingseinheiten teilnehmen. In diesem Kapitel gibt es zu Python selbst nicht viel Neues. Die meiste Zeit werden Sie damit verbringen, den schon vorhandenen Code für die Zusammenarbeit mit MariaDB und PythonAnywhere anzupassen. Ihr Python-Code existiert nie **isoliert**, sondern **interagiert** mit seiner Umgebung und dem System, auf dem er läuft.


```
mysql> select count(*) from events;
+----------+
| count(*) |
+----------+
|       14 |
+----------+
1 row in set (0.00 sec)

mysql> select count(*) from swimmers;
+----------+
| count(*) |
+----------+
|       23 |
+----------+
1 row in set (0.00 sec)

mysql> select count(*) from times;
+----------+
| count(*) |
+----------+
|      467 |
+----------+
1 row in set (0.00 sec)
```

Anhang

Die zehn wichtigsten Dinge, die wir nicht behandelt haben

Wir sind fest davon überzeugt, dass man wissen muss, wann es reicht.
Besonders wenn Ihr Autor aus Irland stammt, einem Land, das dafür berühmt ist, Menschen mit der Gabe hervorzubringen, zu ignorieren, wann sie besser mit dem Reden aufhören sollten. 😉 Dabei genießen wir es, über unsere Lieblingsprogrammiersprache Python zu sprechen. In diesem Anhang zeigen wir Ihnen die zehn wichtigsten Themen, die wir Ihnen gerne »von Kopf bis Fuß« erzählt hätten. Leider hatten wir aber keine weiteren 400 Seiten zur Verfügung. Hier finden Sie Informationen zu Klassen, Exception-Handling, zum Testen eines Walrosses (Ernsthaft? Ein Walross? Ja, ein Walross), zu Switches, Dekoratoren, Kontextmanagern und Nebenläufigkeit sowie Tipps zu Typen, virtuellen Umgebungen und Programmierwerkzeugen. Wie gesagt: Es gibt immer etwas, über das man noch reden kann. Also, blättern Sie um – und haben Sie Freude an den nächsten zwölf Seiten!