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.

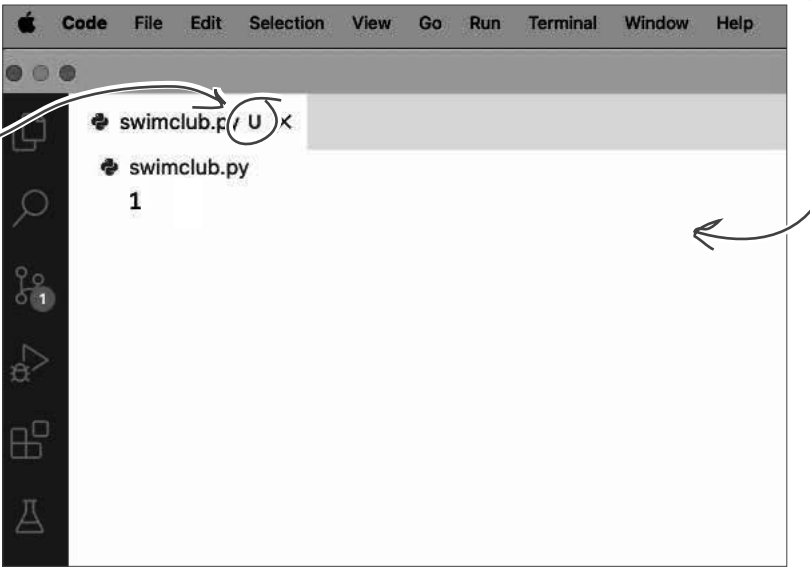
Wie können Sie *diesen* Code mit anderen teilen?

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):
```

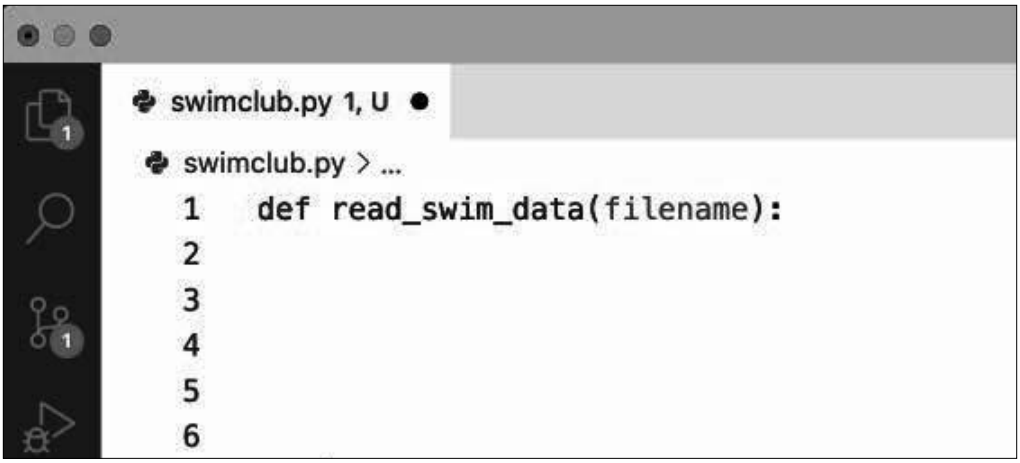
3 Beachten Sie die Verwendung von def und Ihrem besten Freund (dem Doppelpunkt).

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

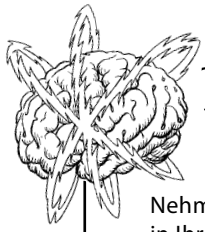
Bauen Sie nun die *Signatur* für die Funktion `read_swim_data` am Anfang der Datei *swimclub.py* ein:

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
```

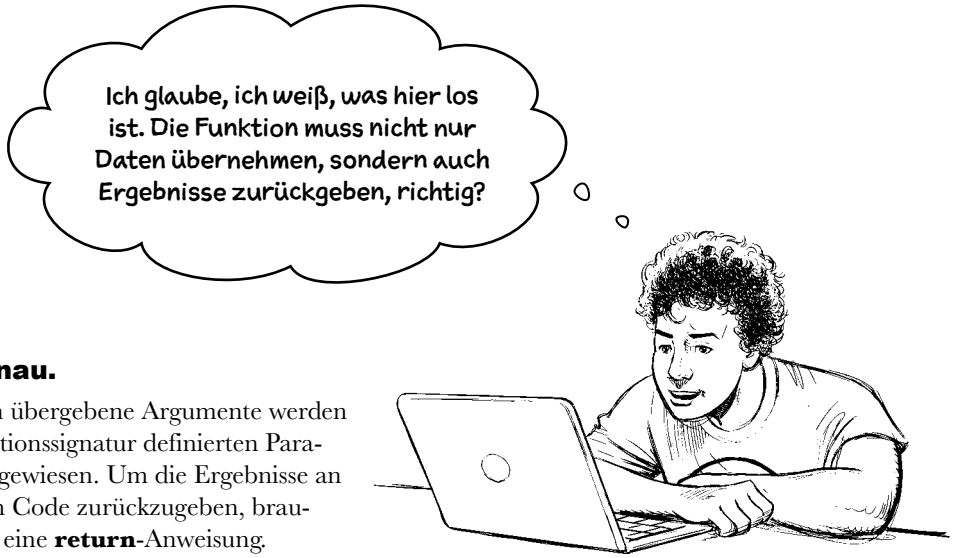
Wenn alles in Ordnung ist, wird nach Ausführung der »import«-Anweisung eine neue leere Codezelle angezeigt. Sehen Sie Fehler, sollten Sie zwei Dinge überprüfen: Stellen Sie sicher, dass Sie Ihren »swimclub.py«-Code gespeichert haben, und sorgen Sie dafür, dass sich »Files.ipynb« im gleichen Ordner befindet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

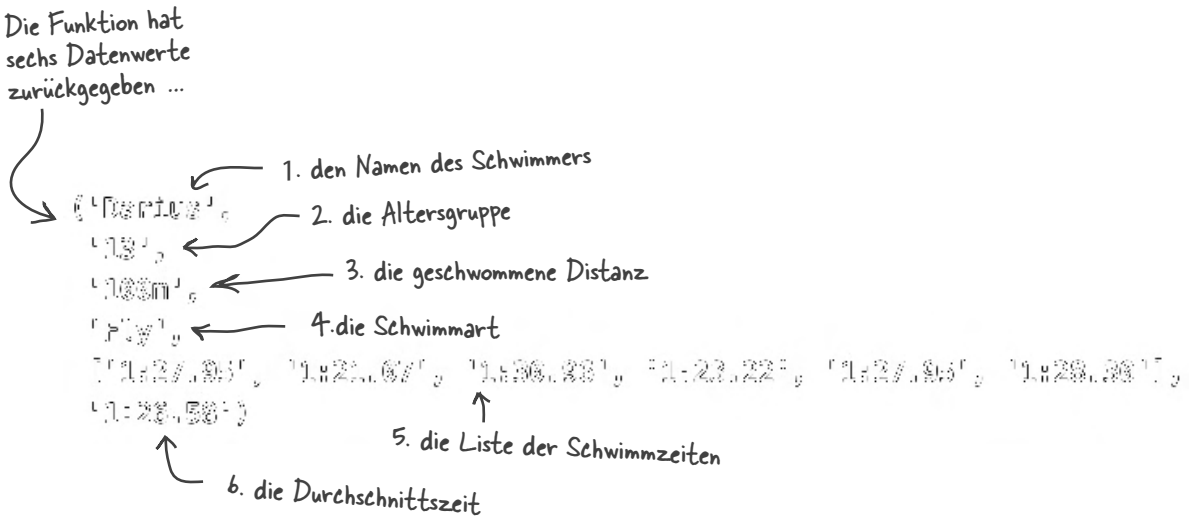
... rufen Sie Ihre Funktion auf, indem Sie dem Funktionsnamen den Namen des Moduls gefolgt von einem PUNKT voranstellen.

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

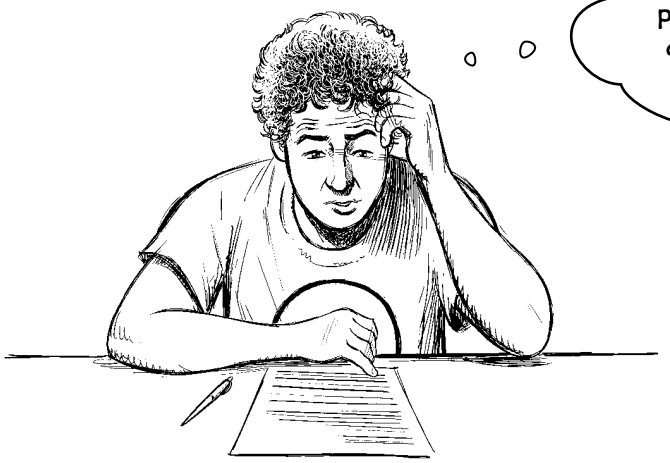
Wenn Sie eine Funktion aufrufen, die aussieht, als gäbe sie mehrere Ergebnisse zurück, sollten Sie noch einmal überlegen. Das ist nämlich nicht der Fall. Stattdessen erhalten Sie ein einzelnes Tupel zurück, das eine Sammlung von Ergebnissen enthält, unabhängig davon, wie viele einzelne Ergebnisse es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