Empfohlene Unterrichtsinhalte als Vorbereitung für dieses Thema: 01-03 (Variablen), 03-04 (Zufallszahlen), 05-01 (Funktionen), 05-03 (Module)

Einführung in die computerlinguistische Programmierung mit Python

05-04: Namenskonflikte 😬

Beim Programmieren können wir Namen für Variablen oder Funktionen frei vergeben - solange wir die grundlegenden Regeln befolgen, die in Thema 01-03 (Variablen) besprochen wurden. Der Interpreter trägt alle Namen, die wir im Laufe eines Programms definieren, in eine Art **Tabelle** ein, um jederzeit herausfinden zu können, welcher Wert mit einem gegebenen Namen verknüpft ist.

Es gibt drei Herausforderungen für diese Namensauflösung, die uns gelegentlich begegnen:

Herausforderung 1: Existierende Namen überschreiben

Wir haben schon oft eigene Variablen angelegt und ihre Werte später überschrieben, zum Beispiel so:

In [ ]:
```
unser_text = "Wichtiger Text"
print(unser_text)

unser_text += "????????"
print(unser_text)
```

Solange wir nur unsere eigenen Variablen überschreiben und den Überblick behalten, ist das in Ordnung. Ärgerlich ist es, wenn wir zum Beispiel vergessen, dass wir einen Namen schon verwendet haben, und zwar für etwas ganz anderes. Hier ein Beispiel:

In [ ]:
```
# eine irgendwie langweilige
# Funktionsdefinition
def test(eingabe):
    print(eingabe)

test = "Esther Seyffarth"

# hä?
test(test)
```

Das Codebeispiel oben beginnt mit der Definition einer Funktion namens `mein_name()`. Weiter unten definieren wir eine **neue Variable**, die ebenfalls `mein_name` heißt, hier aber einen String enthält. Ganz zum Schluss versuchen wir, die Funktion anzuwenden und ihr den String als Argument zu übergeben.

Was geht hier schief? Der Interpreter trägt zuerst die Funktion `mein_name` in die Namenstabelle ein. Wenn danach die Variable mit einem neuen Wert angelegt wird, **entfernt er die Funktion aus der Tabelle** und speichert stattdessen den String unter diesem Namen ab.

Wir können dann nicht mehr auf die Funktion zugreifen!

Dieses Problem können wir umgehen, indem wir **sinnvolle Bezeichnungen für Funktionen und Variablen** wählen. Eine Funktion **tut etwas**, also können wir Namen verwenden, die die Aufgabe der Funktion konkret beschreiben:

- `print_name()`
- `calculate_square()`
- `save_data_to_file()`
- `connect_to_server()`
- ...

Variablen dagegen enthalten **Informationen**. Hier sind Namen wie die folgenden sinnvoll:

- `name`
- `input_number`
- `filepath`
- `server_address`
- ...

Aussagekräftige Namen können uns helfen, Namenskonflikte zu vermeiden.

Übrigens müssen wir auch bedenken, dass Python **eingebaute Funktionen** hat! Was passiert im folgenden Beispiel?

In [ ]:
```
zahl = 5
print(str(zahl))

def str(irgendwas):
    # ergänzt eine str()-Funktion in dieser Pythondatei
    return "Haha, reingelegt!"

# Der Interpreter sucht in der Namenstabelle
# nach einem Konstrukt namens str()....
print(str(zahl))    # :( :( :(
```

Herausforderung 2: Namen aus externen Modulen

Wir können externe Pythondateien **importieren** (siehe Thema 05-03, Module). Dabei werden alle dort enthaltenen Funktionen und Variablen in die Namenstabelle des aktuellen Programms eingetragen.

Es kann passieren, dass das externe Modul Funktionen oder Variablen enthält, deren Namen **exakt in der gleichen Form** auch in unserer aktuellen Datei vorkommen! Deshalb ist es wichtig, dass wir bei jeder Verwendung eines externen Namens immer den Namen des Moduls dazuschreiben. Wir kennen das schon von Zufallszahlen:

In [ ]:
```
import random

# Wir dürfen die randint()-
# Funktion nicht einfach so
# aufrufen :(
print(randint(0,10))
```

In [ ]:
```
import random

def randint(a, b):
    # Unsere eigene, sehr schlechte
    # Definition von randint()
    return "ich bin gar keine Zahl!"

# Welche Ausgabe erwarten wir hier?
print(randint(0, 10))

# Und hier?
print(random.randint(0, 10))
```

Solange wir aktiv darauf achten, Modulnamen immer dazuzuschreiben, ist diese Herausforderung kein Problem für uns. Namen, die in der gleichen Datei definiert sind, die gerade ausgeführt wird, dürfen ohne Modulnamen verwendet werden.

Herausforderung 3: Namespaces von Funktionen

Wenn wir Funktionen definieren, vergeben wir dabei **Namen für die Argumente**. Das sind die Namen, die in der Kopfzeile der Funktion in den Klammern aufgelistet werden. Beim Aufrufen einer Funktion wird angegeben, welche Werte die Namen innerhalb der Funktion bei genau diesem Funktionsaufruf haben sollen.

Namen in Funktionsdefinitionen kann man auch als **lokale Variablen** bezeichnen. Wir können sie mit Werten füllen, sie überschreiben oder sie miteinander vergleichen, wie normale Variablen. Die Besonderheit ist, dass diese Variablen **nur innerhalb der Funktion existieren!** Hier ein Beispiel:

In [ ]:
```
def calc_square(n):
    result = n**2
    print("(wir sind gerade in der Funktion)")
    print(result)
    print(n)
    print("-----")
    return result

print("(wir sind gerade außerhalb der Funktion)")
print("-----")
print(calc_square(3))

#print(n)
print(result)
```

Wenn wir die Kommentarzeichen aus den letzten zwei Zeilen entfernen, lösen wir einen **Fehler** aus. Die beiden Variablen `n` und `result` sind **lokale Variablen**, die nur innerhalb der Funktion verfügbar sind. Beim Verlassen der Funktion werden diese Namen aus der Namenstabelle des Interpreters entfernt.

Die Variable `n` ist das **Argument** der Funktion. Die andere Variable, `result`, ist eine Variable, die innerhalb des Funktionskörpers explizit angelegt und mit einem Wert gefüllt wurde. Hier konnten wir auf den Wert von `n` zugreifen, weil die beiden Variablen im gleichen **Namensraum** existieren.

Zur Erinnerung: Die Argumente von Funktionen sind zunächst **Platzhalter-Variablen**. Funktionen definieren die Logik, aber nicht die konkreten Werte. Erst, wenn wir eine Funktion

aufrufen, geben wir konkrete Werte für die Argumente an.

Wir können uns Funktionen so vorstellen, als würden wir Teilaufgaben an eine andere Person **delegieren**. Wir wollen so wenige Informationen wie möglich austauschen. Alles, was die andere Person unbedingt wissen muss, um die Aufgabe zu erledigen, wird als Argument mitgeliefert. Die Person arbeitet separat - in einem anderen (Namens)raum - an der Aufgabe und liefert uns zum Schluss nur das Ergebnis (den return-Wert). Welche Gedanken die Person sich ansonsten gemacht hat (welche Variablen in ihrem Namensraum erzeugt wurden), wissen wir im Nachhinein nicht. Deshalb können wir darauf nicht von außerhalb zugreifen.

Zusammenfassung

- Der Interpreter speichert alle Namen und die dazugehörigen Werte in einer internen Übersetzungstabelle.
- Wir können Namen überschreiben. Dann vergisst der Interpreter, welcher Wert vorher in diesem Namen gespeichert war.
- Unsere Namen sollen eindeutig und aussagekräftig sein (Herausforderung 1).
- Wenn wir Module importieren und Namen aus diesen externen Dateien verwenden wollen, müssen wir immer dazuschreiben, in welcher Datei der jeweilige Name definiert wird (Herausforderung 2).
- Funktionen haben ihre eigenen Namensräume und lokalen Variablen. Nur die Argumente und `return` -Werte sind Schnittstellen zum restlichen Programm (Herausforderung 3).

Weitere Themen dieser Woche

- 05-01: Funktionen
- 05-02: None
- 05-03: Module