Python für Linguisten

Dozentin: Wiebke Petersen & Co-Dozentin: Esther Seyffarth

Fehlerbehandlung und Fehlervermeidung

Wer findet die meisten Fehler im Code?

```
1  name = input("Name eingeben: ")
2  lese_visitenkarte(name+".txt")
3
4  def lese_visitenkarte(datei):
5      # oeffne Visitenkartendatei
6      vk = open(datei, "r", encoding="utf8")
7      daten = {}
8      info = vk.readlines()
9        # erste Zeile: Vorname und Nachname
10       vorname = info[0].split()[0]
11       nachname = info[1].split()[1]
12       # zweite Zeile: Telefonnummer
13       telnr = info[1]
14       print(telnr - 2)
15       print(telnr.reverse)
16       print(int(telnr))
17       daten["vorname"] = vorname
18       daten["nachname"] = nachname
19       print(daten["telnr"])
```

(Mögliche) Fehler im Code

- NameError
- FileNotFoundError
- IndentationError
- IndexError
- TypeError
- AttributeError
- ValueError
- KeyError

Fehlerarten

- Es gibt auch noch weitere häufige Fehler in Python. Man kann sie grob unterscheiden in...
  - Fehler, die verhindern, dass das Programm ausgeführt werden kann
  - Fehler, die verhindern, dass das Programm seine Aufgabe richtig erfüllt
- Wir lernen heute eine Methode kennen, die zweite Gruppe von Fehlern zu vermeiden bzw. zu behandeln.
- Schauen wir uns die eben gelisteten Fehler mal näher an...

NameError

- Bedeutung: Variable wird nicht gefunden.
- Mögliche Ursachen:
  - Tippfehler beim Variablennamen.
  - Variablen werden außerhalb ihres Gültigkeitsbereichs verwendet, z.B. außerhalb der Funktion, in der sie als Parameter definiert werden.
  - Beim Überarbeiten des Codes wurde die Zeile, in der die Variable angelegt wurde, gelöscht.
  - An dem Punkt während des Programmablaufs, an dem die Variable referenziert wird, ist sie noch nicht definiert (Definition steht erst nach dem Aufruf).
  - Bei der Verwendung von Funktionen aus importierten Modulen fehlt der Verweis auf das jeweilige Modul, Beispiel: `search(pattern, string)` statt `re.search(pattern, string)`.
- Behandeln/Vermeiden: Am Anfang von Funktionsblöcken alle Variablen anlegen, ggf. mit `None` als Wert.

IndentationError

- Bedeutung: Die Einrückung im Programmcode passt nicht zur logischen Struktur.
- Mögliche Ursachen:
  - Code wird innerhalb des Programms an eine Stelle verschoben, die anders eingerückt ist.
  - Eine Anweisungszeile wird so lang, dass sie auf dem Bildschirm nicht mehr lesbar ist, und wird mit einem Zeilenumbruch aufgeteilt. (Kann manchmal funktionieren...)
  - Beim Kopieren/Abschreiben aus anderen Quellen wurden Leerzeichen/Tabs durch Tabs/Leerzeichen ersetzt: IDLE sieht einen Tab und vier Leerzeichen nicht als gleichwertige Einrückung an!
- Behandeln/Vermeiden: Im Zweifelsfall einen Editor mit Anzeige von Formatierungssymbolen verwenden, um zu prüfen, wo Tabs stehen und wo Leerzeichen.

IndentationError bei langen Anweisungen vermeiden

- Es kann vorkommen, dass eine Programmzeile viel zu lang wird und damit nicht mehr auf einmal auf den Bildschirm passt.
- Beispiel: Funktionsaufruf mit vielen Parametern, Berechnung mit langer Formel, unglaublich langer regulärer Ausdruck...
- Best Practice ist eine Zeilenlänge von 79 Zeichen.
- Bei den oben angeführten Beispielen kann man diese Zeilenlänge erreichen und den IndentationError vermeiden, indem man zwischen Argumente einen Backslash setzt.

```
1 with open(json_dir+"/"+tree.family+"/"+tree.name+".json", \
2                       "w", encoding="utf8") as outfile:
3         json.dump(allInfos, outfile, indent=4)
```

AttributeError

- Bedeutung: Das Objekt besitzt nicht das gewünschte Attribut (oder Methode).

```
1 s = "Python"
2 p = re.compile("yt")
3 m = re.search(p, s)
4 print(p.group(0))
```

- Mögliche Ursachen:
  - Tippfehler beim Attribut.
  - Variable hat einen anderen Typ als erwartet. (Häufig `None`, wenn eine Funktion fehlschlägt und statt eines Ergebnisses `None` zurückgibt.)
- Behandeln/Vermeiden: Durchdachte Zuweisung von Werten an Variablen; unbedingt auf die Rückgabewerte von Funktionen achten!

Exkurs: Rückgabe aus Funktionen

- Wichtig: Stelle immer sicher, dass die Funktion in jedem Fall den gleichen Rückgabetyp hat, auch wenn sie nicht erfolgreich ist.
- So kann man im Programm an der Stelle, an der die Funktion aufgerufen wird, Fehler vermeiden.

```
1  def findDaughters(dict_tree_containing_node):
2      """
3      dict -> list
4      Takes a (part of a) tree and returns the list of all
5      daughters of the given mother node. Each
6      daughter is itself a dictionary.
7      """
8      if "children" in dict_tree_containing_node.keys():
9          daughters = dict_tree_containing_node["children"]
10     else:
11         daughters = []
12     return daughters
```

TypeError

- Bedeutung: Die Anwendung einer Funktion auf eine Variable/ein Objekt schlägt fehl, weil das Objekt den falschen Typ hat.
- Mögliche Ursachen:
  - Versehentliche Verwendung von `None` wegen fehlgeschlagener Funktionen.
  - Parameter beim Funktionsaufruf stehen in falscher Reihenfolge.
  - Verwechseln von String-, Listen- und Mengenmethoden, z.B. `append()`, `add()`
- Behandeln/Vermeiden: Durchdachte Zuweisung von Werten an Variablen; die richtigen Methoden für die unterschiedlichen Typen aufs Reference Sheet schreiben/auswendig lernen; unbedingt auf die Rückgabewerte von Funktionen achten!

IndexError

- Bedeutung: Der angegebenene Index (von einer Liste oder einem String) existiert nicht.
- Mögliche Ursachen:
  - Index ist "hardcoded", die eingelesenen Daten haben aber nicht die erwartete Form (z.B. Datei mit weniger Zeilen als erwartet).
  - `for`-Schleifen mit Bezug auf einen Index werden nicht rechtzeitig abgebrochen:

```
1  s = "Python"
2  for i in range(len(s)):
3      print(s[i], s[i+1])
```

  - Index-Eingabe durch Benutzer
- Behandeln/Vermeiden: Durch `if`-Abfrage, durchdachte Schleifenkonstruktion, durchdachte Behandlung von Usereingaben, und ggf. mit `try`/`except`-Block.

KeyError

- Bedeutung: Der angegebenene Schlüssel existiert im Dictionary nicht.
- Mögliche Ursachen:
  - Nutzereingaben ohne weitere Prüfung verwendet.
  - Tippfehler beim Schlüssel, z.B. `dict[name]` statt `dict["name"]`
- Behandeln/Vermeiden: Durch `if`-Abfrage, durchdachte Behandlung von Usereingaben, und ggf. mit `try`/`except`-Block.

ValueError

- Bedeutung: Funktionen schlagen fehl, weil die Variable zwar den richtigen Typ, aber nicht den richtigen Inhalt hat:

```
1 print(int("sieben"))
```

- Mögliche Ursachen:
  - `int("sieben")`
  - `"Hallo Welt".index("y")`
  - `[1,2,3].remove(4)`
  - `range(1,10,0)`
- Behandeln/Vermeiden: Mit `try`/`except`-Block.

FileNotFoundError

- Bedeutung: Datei existiert nicht und kann deshalb nicht geöffnet werden.
- Mögliche Ursachen:
  - User gibt nichtexistenten Dateinamen ein.
  - Verzeichnis stimmt nicht.
- Behandeln/Vermeiden: Mit `try`/`except`-Block.

try und except

- Das Abstürzen des Python-Interpreters beim Auftreten von Fehlern kann man mithilfe von `try` und `except` verhindern.
- Allgemeine Verwendung:

```
1  while True:
2      x = input("Erste Zahl eingeben: ")
3      y = input("Zweite Zahl eingeben: )
4      try:
5          print(int(x)/int(y))
6      except:
7          print("Division nicht moeglich!")
```

- Das Programm wird nun im Falle eines Fehlers die Anweisungen im `except`-Block ausführen und danach normal weiterlaufen.

try und except

- Man kann im `except`-Block auch angeben, welche Art von Fehler behandelt werden soll.

```
1  while True:
2      x = input("Erste Zahl eingeben: ")
3      y = input("Zweite Zahl eingeben: )
4      try:
5          print(int(x)/int(y))
6      except ZeroDivisionError:
7          print("Division nicht moeglich!")
```

try und except

- Wenn die Fehlermeldung ausgegeben werden soll, kann man das ebenfalls im `except`-Block angeben:

```
1  while True:
2      x = input("Erste Zahl eingeben: ")
3      y = input("Zweite Zahl eingeben: )
4      try:
5          print(int(x)/int(y))
6      except ZeroDivisionError as e:
7          print("Division nicht moeglich! Grund: " + str(e))
```

try und except

- Oder es werden mehrere Fehlerarten gleichzeitig abgefangen:

```
1  while True:
2      x = input("Erste Zahl eingeben: ")
3      y = input("Zweite Zahl eingeben: )
4      try:
5          print(int(x)/int(y))
6      except (ZeroDivisionError, ValueError) as e:
7          print("Division nicht moeglich! Grund: " + str(e))
```

- Hier sind die Klammern wichtig, damit der Ausdruck akzeptiert wird.

Grundregeln im Umgang mit try und except

- Ein Fehler sollte nur abgefangen werden, wenn er eine unerwartete Situation darstellt. Gute Hintergrundlektüre: `http://stackoverflow.com/questions/77127/when-to-throw-an-exception`
- Zitat: *Exceptions should be a truly rare thing, UserHasDiedAtKeyboard type situations.*
- Vor allem Nutzereingaben können **ohne** Verwendung von `try`/`except` behandelt werden: Gehe immer davon aus, dass der Nutzer falsche Eingaben machen könnte!
- Die Nutzung von `except ... as e` ermöglicht das Erstellen eines minutiösen Fehlerprotokolls (logfile). Bei umfangreichen Anwendungen ist das sehr hilfreich!
- Die `try`/`except`-Blöcke sollten so klein wie möglich gehalten werden, da sie sonst fast ihren ganzen Informationsgehalt verlieren.

Negativbeispiel zu try und except

```
1  def login(username, pw, url, debug):
2      try:
3          if debug == False:
4              login_page = urllib2.urlopen(url, username, pw).read()
5              print("Login erfolgreich!")
6          else:
7              print("DEBUG - Login entfaellt")
8          return login_page
9      except:
10         print("Login fehlgeschlagen :-(")
```

- Der `try`-Block enthält zuviele Anweisungen.
- Falls der `except`-Block ausgeführt wird, wissen wir nicht, an welcher Stelle der Code fehlgeschlagen ist.
- Die Rückgabe der Funktion ist nicht sauber gelöst. Im Debug-Fall kann keine login_page zurückgegeben werden. Es sind Rückgaben von verschiedenen Typen (string oder None) möglich.

try, except, else, finally

```
1  def datei_lesen(pfad):
2      try:
3          f = open(pfad, "r", encoding="utf8")
4      except FileNotFoundError:
5          print("Datei nicht vorhanden!")
6          text = ""
7      else:
8          text = f.read()
9          f.close()
10     finally:
11         return text
```

- Wenn die Datei geöffnet werden kann, wird der `try`-Block ausgeführt.
- Wenn ein FileNotFoundError auftritt, wird der `except`-Block ausgeführt.
- Wenn der Fehler nicht auftritt, wird der `else`-Block ausgeführt.
- In jedem Fall wird der `finally`-Block ausgeführt.