Empfohlene Unterrichtsinhalte als Vorbereitung für dieses Thema: 04-01/2/3 (Dictionaries, Dateien schreiben/lesen) , 05-01/2/3/4 (Funktionen, None, Module, Namenskonflikte)

Einführung in die computerlinguistische Programmierung mit Python

Fehlertypen: Erkennen, Verstehen und Vermeiden 💥

Oh Nein, mein Programm wirft einen Fehler

Keine Panik! Wenn Ihr beim Aufruf eines Pythonskriptes eine Fehlermeldung erhaltet, ist das nicht schlimm. Im Gegenteil: Der Text der Fehlermeldung genau zu lesen hilft dabei die Stelle im Code zu identifizieren, die für den Absturz verantwortlich ist.

Die Sprache Python kennt eine Reihe verschiedener Fehlerarten. Die Fehlermeldungen, die im Problemfall angezeigt werden, enthalten immer mindestens zwei Informationen: Den Namen des Fehlers und die Zeilennummer, in der der Fehler aufgetreten ist.

Folgendes passiert z.B., wenn wir in einer Schleife auf Indize Ausdrücken arbeiten und versuchen, auf eine nicht existierende Gruppe zuzugreifen:

In [1]:
```
words = ["I", "like", "coffee"]

for i in range(5):
  print(str(i+1) + "." + words[i])
```

Zuerst werden die Aufrufe von `print()` , die unproblematisch sind, ausgeführt. (Wir erinnern uns, dass der Interpreter Befehle von oben nach unten ausführt und daher erst mitten im Programm merkt, dass ein Fehler vorliegt.)

Dann tritt ein Fehler auf, der als `IndexError` bezeichnet wird. Das bedeutet, dass es nicht möglich ist, auf den angegebenen Index im angegebenen Objekt zuzugreifen. Die Zahl, auf die der Pfeil zeigt, ist die Zeilennummer: Unser Fehler ist in Zeile 9 aufgetreten.

Wir müssen beachten, dass die fehlgeschlagene Zeile sich durch nichts von der Form oder Struktur der vorher ausgeführten Zeilen unterscheidet. Ob ein `IndexError` auftritt oder nicht, hängt davon ab, wie der Inhalt des vorliegenden Objekts (hier: `words`) beschaffen ist.

Traceback (most recent call last)

Der Traceback ist die Auflistung der aufgetretenen Fehler im Programm. Dabei bedeutet most recent call last, dass die Liste von oben nach unten chronologisch sortiert ist. Das spielt immer dann eine Rolle, wenn der fehlerhafte Code sich innerhalb einer Funktionsdefinition befindet. Betrachten wir diese Funktion, die das Ergebnis der Division zweier Zahlen zurückgeben soll:

In [1]:
```
def division(x,y):
    div = x/y
    return div

print(division(5,2))
print(division(10,10))
print(division(2,0))
```

Die Funktionsdefinition sieht auf den ersten Blick korrekt aus. Die ersten zwei Aufrufe der Funktion sind auch erfolgreich. Aber beim dritten Aufruf, mit den Argumenten 2 und 0, tritt ein Fehler auf: Durch null zu teilen ist nicht möglich. Die Ausgabe des Programms sieht so aus:

Zuerst wird die Funktion definiert, die wir verwenden wollen. (Erinnern Sie sich, dass Funktionsdefinitionen erst dann tatsächlich ausgeführt werden, wenn die Funktion weiter unten im Code aufgerufen wird.)

Die zwei folgenden Zeilen enthalten die Ergebnisse der ersten beiden Funktionsaufrufe, die problemlos berechnet werden können.

Als nächstes versucht der Python-Interpreter, die Funktion mit den Argumenten 2 und 0 auszuführen. Dieser Aufruf steht in Zeile 7 der Eingabe (erster Teil des Tracebacks, mit Pfeil auf Zeile 7).

Der Fehler tritt aber erst auf, wenn im Zuge des Funktionsaufrufs Zeile 2 der Eingabe mit den Zahlen 2 und 0 als Argumenten ausgeführt wird (zweiter Teil des Tracebacks, mit Pfeil auf Zeile 2).

Der Traceback erlaubt es uns, die Auswirkungen eines Fehlers in einem Teil des Programms auf den gesamten Ablauf des Programms zu verstehen. Es könnte zum Beispiel sein, dass einfach eine falsche Funktion aufgerufen wurde und der Fehler so entstanden ist. Dann müssen wir nicht den Code der Funktion korrigieren, sondern die Stelle im Code, an der der falsche Funktionsaufruf steht.

Der Traceback hilft uns, indem er explizit auflistet, was der Interpreter in welcher Reihenfolge auszuführen versucht hat und an welcher Stelle er gescheitert ist.

Häufige Fehlermeldungen

Im Folgenden werden einige der häufigsten Arten von Fehlern aufgelistet und erläutert. In den bisherigen Übungsaufgaben habt ihr vermutlich schon einige dieser Errors beobachten können. Jetzt, wo wir in der Lage sind, den Traceback zu lesen und zu verstehen, können wir jederzeit in der Python-Dokumentation nachlesen, welche Bedeutung ein Error hat und wie wir ihn behandeln können.

SyntaxError

In [1]:
```
print len([1, 2, 3])
```

Jede Programmiersprache folgt fest definierten Regeln für die äußere Form des Codes, zum Beispiel bezüglich der notwendigen Einrückungen, Klammern, Doppelpunkten und so weiter. Fehler, die aus Verstößen gegen diese Regeln resultieren, sind für den Interpreter besonders

einfach zu finden. Er meldet sogar Fehler in Funktionen, die im Programm niemals aufgerufen werden:

In [1]:
```
def missing_parentheses(arg1):
    print arg1

def invalid_syntax(arg1):
    print len(arg1)

invalid_syntax([3, 4, 5])
```

In diesem Programm werden zwei Funktionen definiert, die jeweils einen Formfehler enthalten (print() wird ohne Klammern verwendet). Dabei wird die erste definierte Funktion nie ausgeführt wird. Die Fehlermeldung bezieht sich klar auf Zeile 2 des Programms, also auf den Funktionskörper einer Funktion, die später im Programm nicht verwendet wird. Der Pfeil in der vorletzten Zeile zeigt auf die Stelle, an der der Interpreter den Syntaxfehler entdeckt hat.

Hier wird **kein Traceback** angezeigt wird, obwohl der Fehler sich in der Funktionsdefinition befindet. Das hängt damit zusammen, dass Syntaxfehler geprüft werden, bevor das Programm ausgeführt wird. Es gibt daher keine Informationen zur "Verschachtelung" von Anweisungen.

Außerdem wird nur einer der beiden Syntaxfehler im Code identifiziert. Erst, wenn Sie Zeile 2 korrigiert haben, setzt der Interpreter beim nächsten Programmstart die Prüfung der Syntax fort und stellt fest, dass auch Zeile 5 fehlerhaft ist.

IndentationError und TabError

Bei diesen beiden Fehlern handelt es sich um Untertypen von SyntaxError. Sie werden also auch vor dem Ausführen des Programms geprüft und vom Interpreter gemeldet.

In Python wird durch verschiedene Einrückungen der Codezeilen signalisiert, auf welcher logischen Ebene jede Zeile sich befindet. Alle Zeilen, die in einer gemeinsamen Einrückungsebene stehen, werden linear von oben nach unten nacheinander ausgeführt; um Zeilen einander unterzuordnen, wird Code unterhalb von Funktionskopfzeilen, `if`-Anweisungen, `for`-Schleifenköpfen oder `with`-Statements weiter nach rechts eingerückt. Sobald der Code auf die ursprüngliche Einrückungsebene zurückkehrt, endet der untergeordnete Block.

Wenn sich die Einrückungsebene mitten im Code ändert, ohne dass sie durch eins der genannten Elemente eingeleitet wird, kann die Logik des Programms nicht interpretiert werden:

In [1]:
```
def indentationerror(number):
    print(number + 2)
     return number * 2
```

Übrigens fordert der Interpreter keine bestimmte Anzahl von Einrückungsleerzeichen. Er prüft nur, ob in aufeinander folgenden Zeilen die gleiche Anzahl von Leerzeichen am Anfang steht. Allerdings ist es guter Programmierstil, genau 4 Leerzeichen pro Einrückungsebene zu verwenden.

Im Gegensatz zum IndentationError ist der TabError fast unmöglich zu sehen:

In [1]:
```
def taberror(word1, word2):
        print(word1)
    print(word2)
```

Ein TabError entsteht dann, wenn eine Einrückungsebene sich nicht ausschließlich durch Tabs oder ausschließlich durch Leerzeichen vom restlichen Code absetzt, sondern durch eine Kombination von beidem.

Der Grund dafür ist, dass ein Tab je nach Interpretation für eine beliebige Anzahl von Leerzeichen stehen kann, z.B. 4 oder 8. Durch diesen Interpretationsspielraum ist nicht editorübergreifend klar, welche Leerzeichenzahl genauso wie ein Tab behandelt werden soll. Wir müssen uns also zwischen Tabs und Leerzeichen entscheiden.

VSCode hat eine Funktion zum Anzeigen von Whitespace-Zeichen. Wir können dazu mit `Ctrl + Shift + P` die Command Palette öffnen und `>render whitespace` eingeben. Die Aktion `View: Toggle Render Whitespace` bestätigen wir mit Enter. Jetzt werden für normale Leerzeichen besondere Symbole angezeigt und für Tabs andere Symbole, sodass Sie eine Chance haben, zu sehen, wo das Problem liegt.

NameError

In [1]:
```
fullname = "Bruce Banner"
print(full_name)
```

Versuchen wir, auf nichtdefinierte Variablen zuzugreifen, entsteht ein `NameError`. In diesem Beispiel wird die Variable `full_name` in der ersten Zeile mit Underscore angelegt, in der zweiten Zeile aber ohne Underscore referenziert.

Ein `NameError` tritt auch dann auf, wenn wir versuchen, auf Variablen außerhalb ihres Gültigkeitsbereichs (Name Space) zuzugreifen. Wir erinnern uns, dass z.B. Variablen, die wir in einem Funktionskörper definieren, nur innerhalb dieser Funktion existieren und nicht außerhalb der Funktion referenziert werden können.

AttributeError

In [1]:
```
print(' Titel '.stip())
```

Falls Sie sich beim Aufrufen einer Funktion vertippen, wie hier bei `strip()`, tritt kein NameError auf, sondern ein AttributeError. Achtung: Dieser Fehler wird nicht nur durch Verschreiben ausgelöst. Was ist das Problem im folgenden Code?

In [1]:
```
input_list = ["I'd", "like", "three", "coffee,", "please."]

print(input_list.keys())
```

Es macht Sinn, dass zwischen `NameError` und `AttributeError` unterschieden wird. Ersterer bezieht sich auf Variablen, die wir definiert haben und über die wir also die Kontrolle haben. Wenn wir jedoch versuchen, auf ein Attribut zuzugreifen, das für einen Objekttyp nicht definiert ist, fehlt uns diese Kontrolle. In dem Fall müssen wir uns eine andere Strategie zum Korrigieren des Fehlers überlegen.

TypeError

Damit eine Operation in Python erfolgreich ist, müssen alle Operanden vom richtigen Typ sein. Ist das nicht der Fall, wird ein `TypeError` ausgelöst:

In [1]:
```
print(1 + "2")
```

In [1]:
```
print("Ergebnis: " + 100)
```

Es gibt auch noch andere Situationen, in denen Sie auf einen TypeError stoßen können. Im folgenden Beispiel werden runde Klammern verwendet, wo der Interpreter eckige Klammern erwartet:

In [1]:
```
d = {"the": 200, "of": 134}
print(d("the"))
```

Und schließlich kann es vorkommen, dass wir in unserem eigenen Code Variablennamen wählen, mit denen Funktionen von Python überschrieben werden (vgl. Unterrichtsmaterial vom 30.11.2020: 05-04 Namenskonflikte). Solche Situationen sollten wir unter allen Umständen vermeiden! Sonst passiert folgendes:

In [1]:
```
print(str(5))

str = "Hello world"
print(str)

print(str(5))
```

Die eingebaute Funktion `str()` wandelt das übergebene Argument in ein Objekt vom Typ String um. Im Codebeispiel wird eine Variable namens `str` angelegt, deren Wert ein String ist. Wenn nach dieser Zuweisung nun `str` im Code vorkommt, wird immer auf diesen String verwiesen, und Sie haben keinen Zugriff mehr auf die ursprüngliche Bedeutung von `str` als Funktion.

Nach dem Ausführen dieser Code-Zelle sollte der Kernel neugestartet werden, weil sonst in dieser Sitzung die ursprüngliche `str()`-Methode nicht mehr verwendet werden kann.

ValueError

Wie wir gesehen haben, ist es wichtig, dass jedes im Code vorkommende Objekt jeweils vom richtigen Typ ist. Allerdings reicht es nicht aus, wenn der Typ stimmt: Für viele Operationen ist es wichtig, dass nicht nur der Typ korrekt ist, sondern auch bestimmte Bedingungen über den Wert des Objektes erfüllt sind:

In [1]:
```
print(int("5"))
print(int("5.1"))
```

In [1]:
```
int("twenty")
```

Im ersten Beispiel wird der String `"5"` erfolgreich in eine Integerzahl umgewandelt. Bei den Strings `"5.1"` und `"twenty"` funktioniert das allerdings nicht.

Die Umwandlungsfunktion `int()` verlangt ein Stringargument, und es werden in allen Zeilen Strings übergeben. Im zweiten und dritten Aufruf können die übergebenen Strings

KeyError & IndexError

Der `KeyError` ist verwandt mit dem oben schon erwähnten `IndexError`. Beide gehören in die Kategorie `LookupError`, treten also auf, wenn in einer Datenstruktur ein Element gesucht wird, das nicht vorhanden ist. Beim `IndexError` lag das daran, dass die angegebene Position im String oder der Liste nicht existiert, z.B. weil der Index größer ist als der größte vorhandene Index.

Löst unser Code einen `KeyError` aus, bedeutet das, dass in einem Dictionary der gesuchte Schlüssel nicht gefunden werden kann:

In [1]:
```
noten = {"Mathe": 1.3, "Deutsch": 2.3, "Englisch": 1.7}
print(noten["Latein"])
```

FileNotFoundError

Wenn wir versuchen, eine Datei zu lesen, die nicht existiert, wird ein `FileNotFoundError` ausgelöst:

In [1]:
```
open("thisfiledoesnotexist.txt", "r")
```

Achtung: Beim Öffnen von Dateien geben Sie den Modus an, in dem sie geöffnet werden sollen: `"r"` zum Lesen, `"w"` zum Schreiben. Der Error wird nur dann ausgelöst, wenn Sie als Modus `"r"` angeben. Andernfalls wird die Datei einfach vom Python-Interpreter angelegt.

Zusammenfassung

- Wir haben gelernt, wie wir den `traceback` einer Fehlermeldung von Python lesen, um herauszufinden, an welcher Stelle im Code ein Fehler aufgetreten ist.
- Wir haben die häufigsten Fehlerarten in Python kennengelernt.

Weitere Themen dieser Woche

- Debugging 🕵