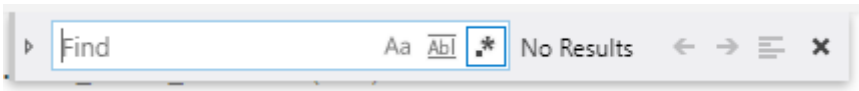
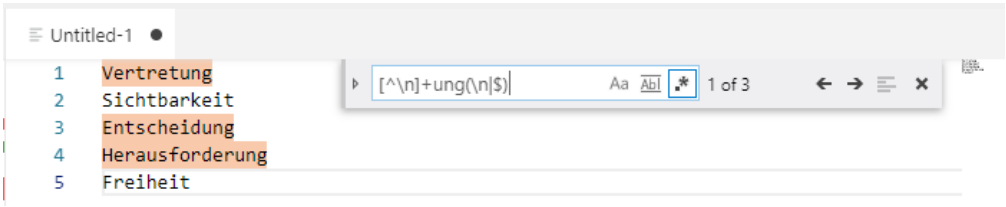
Editor-Trick des Tages: Volltextsuche mit regulären Ausdrücken, `Select All Occurrences of Find Match`

Jetzt, wo Sie reguläre Ausdrücke beherrschen, können Sie sie verwenden, um auch im Editor nach beliebigen Stringmustern zu suchen. Sie kennen sicher bereits die Volltextsuche, die sich öffnet, wenn Sie `Ctrl + F` drücken oder in der Command Palette `>Find` eingeben.

Sie können im Suchfeld auf den Button mit der Beschriftung `.*` klicken:



Jetzt können Sie als Suchbegriff reguläre Ausdrücke eingeben. Wie Sie sehen, werden alle Matches des regulären Ausdrucks (hier: `[^\n]+ung(\n|$)`) gefunden:



Um die Suchergebnisse weiterzuverarbeiten, können Sie jetzt alle Treffer auf einmal markieren, indem Sie in der Command Palette `>select all occurrences of find match` eingeben oder `Ctrl + Shift + L` drücken. Dadurch wird ein Multicursor erzeugt, der alle Suchergebnisse markiert.

Wenn Sie die markierten Treffer kopieren (mit `Ctrl + C`) und in einer anderen Datei einfügen (mit `Ctrl + V`), können Sie mit den gefundenen Elementen bequem weiterarbeiten.

Sie können außerdem die Regex-Suche verwenden, um Sequenzen, die bestimmten Mustern folgen, durch andere Muster zu ersetzen.

Probieren Sie es selbst: Tippen Sie eine Liste wie die im Beispiel in Ihren Editor und markieren Sie alle Vorkommen des angegebenen regulären Ausdrucks. Wenn Sie die Ergebnisse in eine neue Datei einfügen, werden einige Zeilenumbrüche mit übernommen. Verwenden Sie reguläre Ausdrücke und die `Replace` -Funktion von VSCode (`Ctrl + H`), um alle Vorkommen von mehreren Zeilenumbrüchen (`\n`) durch jeweils nur einen einzigen Zeilenumbruch zu ersetzen.

Programmierfehler und wie wir sie lesen

In [1]:

```
%%html
<blockquote class="twitter-tweet"><p lang="en" dir="ltr">Debugging is like being the de
tective in a crime movie where you are also the murderer.</p>&mdash; Filipe Fortes (@fo
rtes) <a href="https://twitter.com/fortes/status/399339918213652480?ref_src=twsrc%5Etf
w">November 10, 2013</a></blockquote>
<script async src="https://platform.twitter.com/widgets.js" charset="utf-8"></script>
```

> Debugging is like being the detective in a crime movie where you are also the murderer.
> — Filipe Fortes (@fortes) November 10, 2013 (https://twitter.com/fortes/status/399339918213652480?ref_src=twsrc%5Etfw)

Formale oder logische Fehler in Ihrem Pythoncode können dazu führen, dass Ihr Programm abgebrochen wird, statt ordnungsgemäß ausgeführt zu werden - oder dass es zwar ausgeführt wird, die Ergebnisse aber nicht korrekt sind. Wir beschäftigen uns heute mit häufig auftretenden Fehler und einigen Methoden, um solche Fehler zu vermeiden bzw. zu behandeln, wenn sie doch auftreten.

Hilfe, mein Programm ist abgestürzt!

Keine Panik! Wenn Sie beim Aufruf eines Pythonskriptes eine Fehlermeldung erhalten, ist das nicht schlimm. Im Gegenteil: Sie können den Text der Fehlermeldung genau lesen und so die Stelle im Code 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 mit regulären Ausdrücken arbeiten und versuchen, auf eine nicht existierende Gruppe zuzugreifen:

In [ ]:

```
import re

drink = 'warm tea'
description = re.compile('(hot|warm|cold)\s(milk|coffee|water|tea)')
m = re.search(description, drink)
print(m.group(0))
print(m.group(1))
print(m.group(2))
print(m.group(3))
```

Wir erhalten in etwa die folgende Ausgabe:

```
warm tea
warm
tea
---------------------------------------------
IndexError                Traceback (most recent call last)
<ipython-input-2-09d945422595> in <module>()
      7 print(m.group(1))
      8 print(m.group(2))
----> 9 print(m.group(3))

IndexError: no such group
```

Zuerst werden die Aufrufe von `print()`, die unproblematisch sind, ausgeführt. (Erinnern Sie sich, 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.

Beachten Sie, 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: `m`) 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 [ ]:

```
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:

```
2.5
1.0
---------------------------------------------------
ZeroDivisionError         Traceback (most recent call last)
<ipython-input-3-3c0d95283e6c> in <module>()
      5 print(division(5,2))
      6 print(division(10,10))
----> 7 print(division(2,0))

<ipython-input-3-3c0d95283e6c> in division(x, y)
      1 def division(x,y):
----> 2     div = x/y
      3     return div
      4
      5 print(division(5,2))

ZeroDivisionError: division by zero
```

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

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

Häufige Errors

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

SyntaxError

In [ ]:

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

Ausgabe:

```
  File "<ipython-input-4-c4aacdbc743a>", line 1
    print len([1, 2, 3])
            ^
SyntaxError: invalid syntax
```

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 [ ]:

```
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). Beachten Sie, dass die erste definierte Funktion nie ausgeführt wird. Wenn das Programm ausgeführt wird, sehen wir folgenden Output:

```
  File "<ipython-input-5-a464b218d2a0>", line 2
    print arg1
            ^
SyntaxError: Missing parentheses in call to 'print'
```

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.

Beachten Sie, dass hier **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 [ ]:

```
def indentationerror(number):
    print(number + 2)
     return number * 2
```

Ausgabe:

```
    File "<ipython-input-6-a0a69aa616ef>", line 3
        return number * 2
        ^
    IndentationError: unexpected indent
```

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 [ ]:

```
def taberror(word1, word2):
        print(word1)
    print(word2)
```

Ausgabe:

```
    File "<ipython-input-7-fc121b7a4c9b>", line 3
        print(word2)
                    ^
    IndentationError: unindent does not match any outer indentation level
```

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. Sie müssen sich also zwischen Tabs und Leerzeichen entscheiden.

VSCode hat eine Funktion zum Anzeigen von Whitespace-Zeichen. Öffnen Sie dazu mit `Ctrl + Shift + P` die Command Palette und geben Sie `>render whitespace` ein. Die Aktion `View: Toggle Render Whitespace` bestätigen Sie 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 [ ]:

```
fullname = "Esther Seyffarth"
print(full_name)
```

Ausgabe:

```
    ---------------------------------------------------------
    NameError                 Traceback (most recent call last)
    <ipython-input-8-efa811bd3b65> in <module>()
          1 fullname = "Esther Seyffarth"
    ----> 2 print(full_name)

    NameError: name 'full_name' is not defined
```

Versuchen Sie, 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 Sie versuchen, auf Variablen außerhalb ihres Gültigkeitsbereichs (Name Space) zuzugreifen. Erinnern Sie sich, dass z.B. Variablen, die Sie in einem Funktionskörper definieren, nur innerhalb dieser Funktion existieren und nicht außerhalb der Funktion referenziert werden können.

AttributeError

In [ ]:

```
print(' Titel '.stip())
```

Ausgabe:

```
---------------------------------------------------------
AttributeError                Traceback (most recent call last)
<ipython-input-9-72e2bcae39ec> in <module>()
----> 1 print(' Titel '.stip())

AttributeError: 'str' object has no attribute '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 [ ]:

```
import re
s = "Python"
p = re.compile("th")
m = re.search(p,s)
print(p.group(0))
```

Es macht Sinn, dass zwischen `NameError` und `AttributeError` unterschieden wird. Ersterer bezieht sich auf Variablen, die Sie definiert haben und über die Sie also die Kontrolle haben. Wenn Sie jedoch versuchen, auf ein Attribut zuzugreifen, das für einen Objekttyp nicht definiert ist, fehlt Ihnen diese Kontrolle. In dem Fall müssen Sie sich 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 [ ]:

```
print(1 + "2")
```

In [ ]:

```
print("Ergebnis: " + 100)
```

Ausgabe:

```
---------------------------------------------------------
TypeError                 Traceback (most recent call last)
<ipython-input-11-b88986c5ffd8> in <module>()
----> 1 print(1 + "2")

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

und

```
---------------------------------------------------------
TypeError                 Traceback (most recent call last)
<ipython-input-12-3fc5ce4a7c11> in <module>()
----> 1 print("Ergebnis: " + 100)

TypeError: must be str, not int
```

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 [ ]:

```
d = {"the": 200, "of": 134}
print(d("the"))
```

Ausgabe:

```
---------------------------------------------------------------
TypeError                       Traceback (most recent call last)
<ipython-input-13-7eda602a7d54> in <module>()
      1 d = {"the": 200, "of": 134}
----> 2 print(d("the"))

TypeError: 'dict' object is not callable
```

Und schließlich kann es vorkommen, dass Sie in Ihrem eigenen Code Variablennamen wählen, mit denen Funktionen von Python überschrieben werden (vgl. Unterrichtsmaterial vom 29.10.2019). Versuchen Sie unter allen Umständen, solche Situationen zu vermeiden! Sonst passiert folgendes:

In [ ]:

```
print(str(5))

str = "Hello world"
print(str)

print(str(5))
```

Ausgabe:

```
5
Hello world
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-40bb0a533c7f> in <module>()
      4 print(str)
      5
----> 6 print(str(5))

TypeError: 'str' object is not callable
```

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 Sie 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 [ ]:

```
print(int("5"))
print(int("5.1"))
```

In [ ]:

```
int("twenty")
```

Ausgabe:

```
5
---------------------------------------------------------------
ValueError                    Traceback (most recent call last)
<ipython-input-15-7e40f02ad68c> in <module>()
      1 print(int("5"))
----> 2 print(int("5.1"))

ValueError: invalid literal for int() with base 10: '5.1'
```

und

```
---------------------------------------------------------------
ValueError                    Traceback (most recent call last)
<ipython-input-16-895535f411a3> in <module>()
----> 1 int("twenty")

ValueError: invalid literal for int() with base 10: '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 aber nicht korrekt verarbeitet werden, sodass der `ValueError` entsteht.

KeyError

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

In [ ]:

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

Ausgabe:

```
---------------------------------------------------------------
KeyError                      Traceback (most recent call last)
<ipython-input-17-f2ab499ecb38> in <module>()
      1 noten = {"Mathe": 1.3, "Deutsch": 2.3, "Englisch": 1.7}
----> 2 print(noten["Latein"])

KeyError: 'Latein'
```

FileNotFoundError

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

In [ ]:

```
open("thisfiledoesnotexist.txt", "r")
```

Ausgabe:

```
---------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-18-e60da35c07c7> in <module>()
----> 1 open("thisfiledoesnotexist.txt", "r")

FileNotFoundError: [Errno 2] No such file or directory: 'thisfiledoesnotexist.tx
t'
```

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.

try und except

Sie kennen nun einige Fehlerarten in Python und können am Traceback ablesen, wie ihr Code korrigiert werden muss. In den Fällen, wo ein Error durch einen Tipp- oder Denkfehler Ihrerseits zustande gekommen ist, können Sie einfach die fehlerhafte Zeile verbessern. Es kann aber auch vorkommen, dass Fehler erst während der Programmausführung entstehen, zum Beispiel weil bestimmte Eingaben vom Benutzer nicht verarbeitet werden können.

Der Code im Beispiel unten hat die Funktion, Rechenterme umzustellen, sodass der Operator nicht mehr zwischen den Operanden steht, sondern davor. Die Funktion erwartet Eingaben der Form `[arg1] [operator] [arg2]`, beispielsweise `5 + 5`. Sehen wir uns einmal an, wie das während der Laufzeit aussieht:

In [ ]:

```
def polish_notation(calc_input):
    calc_input = calc_input.split()
    arg1 = calc_input[0]
    op = calc_input[1]
    arg2 = calc_input[2]
    result = "{} {} {}".format(op, arg1, arg2)
    return result


original_calc = input("Bitte Term eingeben: ")
polish_notation_calc = polish_notation(original_calc)
print("Polnische Notation Ihrer Eingabe: {}".format(polish_notation_calc))
print("")
```

Solange wir uns an das erwartete Eingabeformat halten, ist alles gut.

```
Bitte Term eingeben: 3 + 6
Polnische Notation Ihrer Eingabe: + 3 6
```

Wenn wir aber eine Eingabe machen, die anders aussieht, stürzt das Programm ab:

```
Bitte Term eingeben: 3+6
---------------------------------------------------------------
IndexError                    Traceback (most recent call last)
<ipython-input-1-8248f2c92376> in <module>()
      9 while True:
     10     original_calc = input("Bitte Term eingeben: ")
---> 11     polish_notation_calc = polish_notation(original_calc)
     12     print("Polnische Notation Ihrer Eingabe: {}".format(polish_notation_
calc))
     13     print("")

<ipython-input-1-8248f2c92376> in polish_notation(calc_input)
      2     calc_input = calc_input.split()
      3     arg1 = calc_input[0]
----> 4     op = calc_input[1]
      5     arg2 = calc_input[2]
      6     result = "{} {} {}".format(op, arg1, arg2)

IndexError: list index out of range
```

Die Funktion geht davon aus, dass Operanden und Operator durch Leerzeichen voneinander getrennt sind. Ist das nicht der Fall, kann das Programm seine Aufgabe nicht erfüllen.

Wenn absehbar ist, welche Fehlerart in einem Teil des Programmcodes auftreten wird, können Sie eine Fehlerbehandlung in Ihr Programm einbauen. Dazu umgeben Sie den "verdächtigen" Teil des Programms mit einer Struktur, die im Fehlerfall das Problem abfängt und eine Lösung implementiert, die sinnvoll ist. Das kann zum Beispiel so aussehen:

```
def polish_notation(calc_input):
    calc_input = calc_input.split()
    try:
        arg1 = calc_input[0]
        op = calc_input[1]
        arg2 = calc_input[2]
        result = "{} {} {}".format(op, arg1, arg2)
    except IndexError:
        result = "Kann nicht berechnet werden. Prüfen Sie Ihre Eingabe."
    return result
```

Der `IndexError` trat ursprünglich dort auf, wo auf bestimmte Indizes in `calc_input` zugegriffen wurde. Diese Zeilen sind nun in einen `try` -Block eingebettet. Befehle, die in einem solchen `try` -Block stehen, werden zunächst ganz normal nacheinander ausgeführt. Der `except` -Block gibt an, welche Befehle ausgeführt werden sollen, falls der Code im `try` -Block einen Fehler auslöst.

Der Effekt von `try` und `except` ist, dass das Programm im Fehlerfall nicht mehr abstürzt. Stattdessen wird, falls ein `IndexError` auftritt, der Code aus dem `except` -Block ausgeführt und das Programm läuft weiter.

In [ ]:

```
def polish_notation(calc_input):
    calc_input = calc_input.split()
    try:
        arg1 = calc_input[0]
        op = calc_input[1]
        arg2 = calc_input[2]
        result = "{} {} {}".format(op, arg1, arg2)
    except IndexError:
        result = "Kann nicht berechnet werden. Prüfen Sie Ihre Eingabe."
    return result


original_calc = input("Bitte Term eingeben: ")
polish_notation_calc = polish_notation(original_calc)
print("Polnische Notation Ihrer Eingabe: {}".format(polish_notation_calc))
print("")
```

Die Zeile, die den `except`-Block einleitet, enthält die Information, welche Fehlerart abgefangen werden soll. Das bedeutet, dass Ihr Code auch mehrere `except`-Blöcke nacheinander enthalten kann.

Sie können diese Angabe auch weglassen; dann wird Ihr Programm sämtliche Fehler, die es gibt, abfangen. Das wird allerdings als schlechter Stil angesehen, weil Sie so verstecken, welche Fehlerarten tatsächlich auftreten können. Außerdem ist für verschiedene Fehlerarten eine unterschiedliche Behandlung sinnvoll, und wenn Sie nicht zwischen Fehlerarten unterscheiden, ist Ihre Fehlerbehandlung zu allgemein, um wirklich nützlich zu sein.

Validierung von Benutzereingaben: *Look before You Leap* vs. *It Is Easier to Ask for Forgiveness than Permission*

Ungültige Eingaben des Benutzers können Programme abstürzen lassen. Um das zu verhindern, müssen Eingaben *validiert* werden.

Hierzu gibt es zwei Strategien: Entweder prüft das Programm erst ganz genau, ob alle Eingaben die richtige Form haben. Wenn nicht, gibt es eine Fehlermeldung aus. Wenn ja, verarbeitet es die Eingaben, wobei garantiert keine Exceptions auftreten. Diese Strategie wird mit dem Slogan *Look before You Leap* bezeichnet.

Oder das Programm fängt einfach an, die Eingaben zu verarbeiten. Erst wenn dabei Exceptions auftreten, bemerkt das Programm, dass die Eingabe falsch war, fängt die Exception ab und reagiert mit einer entsprechenden Fehlermeldung. Für diese Strategie gibt es den Slogan *It's Easier to Ask for Forgiveness than Permission*.

Hier ist noch einmal die Funktion `polish_notation`. Diesmal prüft sie sowohl, ob die Eingabe drei Elemente enthält, als auch, ob das erste und dritte Zahlen sind, und zwar nach der Strategie *It's Easier to Ask for Forgiveness than Permission*:

In [ ]:

```
def polish_notation(calc_input):
    calc_input = calc_input.split()
    try:
        arg1 = calc_input[0]
        op = calc_input[1]
        arg2 = calc_input[2]
    except IndexError:
        return "Bitte geben Sie drei durch Leerzeichen getrennte Elemente ein."
    try:
        arg1 = int(arg1)
        arg2 = int(arg2)
    except ValueError:
        return "Das erste und dritte Element müssen Zahlen sein."
    return "{} {} {}".format(op, arg1, arg2)
```

Hier ist eine Version, die dasselbe macht, allerdings mit der Strategie *Look before You Leap*. Sie versucht nicht einfach, auf die Indizes 0-2 zuzugreifen und die Elemente an den Indizes 0 und 2 in Integers umzuwandeln, sondern überprüft erst, ob das überhaupt möglich ist:

In [ ]:

```
def polish_notation(calc_input):
    calc_input = calc_input.split()
    if len(calc_input) != 3:
        return "Bitte geben Sie drei durch Leerzeichen getrennte Elemente ein."
    arg1 = calc_input[0]
    op = calc_input[1]
    arg2 = calc_input[2]
    int_pattern = re.compile('^\d+$')
    if not (int_pattern.match(calc_input[0]) and
            int_pattern.match(calc_input[2]):
        return "Das erste und dritte Element müssen Zahlen sein."
    arg1 = int(arg1)
    arg2 = int(arg2)
    return "{} {} {}".format(op, arg1, arg2)
```

Welche Strategie man anwendet, ist teils eine Geschmacksfrage. *Look before You Leap* ist oft klarer, da man sofort sieht, bis wohin die Funktion im Falle eines Fehlers ausgeführt wird, wohingegen das bei mehrzeiligen `try` -Blöcken nicht auf den ersten Blick klar ist.

TLDR

Keine Sorge, falls Sie nicht alle Unterschiede in den einzelnen Codebeispielen oben wahrgenommen haben. Die korrekte Behandlung von Exceptions ist ein Thema, in dem auch professionelle Programmierer_innen oft etwas unsicher sind. Das liegt daran, dass Fehler so schwer vorhersehbar sind.

Beispielsweise haben wir oben unterschiedliche Fehlerarten (`IndexError` , `ValueError`) mit expliziten Fehlerbehandlungen versehen. Da das Programm aber Nutzereingaben verarbeitet, ist es leicht vorstellbar, dass ganz andere Fehlerarten auftreten, an die wir vorher gar nicht gedacht haben...

Die Übungsaufgaben diese Woche sollen Ihnen helfen, sich mit Exceptions vertraut zu machen.

Hier noch einige Hinweise, die Ihnen möglicherweise helfen, Fehler von vornherein zu vermeiden:

1. Verwenden Sie sprechende Variablennamen: Nur Listen/Sets/Dictionarys/etc. in Pluralform, einzelne Strings/Zahlen/etc. in Singularform. Wenn Sie Wahrheitswerte in Variablen speichern, können Sie Namen wählen wie `is_a_verb` oder `has_suffix` .
2. Schreiben Sie möglichst kurze Funktionen, möglichst kurze Schleifen, möglichst kurze `if/elif/else` -Blöcke. Je weniger Code Sie schreiben, desto weniger Fehler machen Sie... Außerdem hilft das Isolieren einzelner Schritte dabei, sicherzustellen, dass der Code genau das tut, was er soll.
3. Gewöhnen Sie sich an, Prüfungen bestimmter Bedingungen an den Anfang Ihrer Funktion zu setzen: Ist der übergebene String lang genug? Ist die aktuelle Zeile der Datei evtl. leer, sodass Sie sie nicht weiterverarbeiten können?

In der nächsten Sitzung werden wir eine Funktionalität von VSCode kennenlernen, mit der wir den Zustand von Variablen während der Laufzeit analysieren können.

Aufgaben

1. Verändern Sie den Code der Divisionsfunktion von vorhin so, dass beim Teilen durch 0 das Programm nicht abstürzt, sondern eine Information ausgegeben wird. Wählen Sie selbst, ob Sie die Eingabe validieren oder auftretende Fehler mit try und except abfangen.

In [ ]:

```
def division(x,y):
    ######################
    # Hier Code einfügen #
    ######################
    div = x/y
    return div

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

1. Fallen Ihnen Aufgaben aus den letzten Übungssitzungen ein, bei denen eine explizite Fehlerbehandlung mit try/except hilfreich gewesen wäre? Diskutieren Sie mit Ihren Kommiliton_innen.

Zusammenfassung

Sie haben heute gelernt,

- wie Sie die Fehlerprotokolle von Python (`tracebacks`) lesen, um herauszufinden, an welcher Stelle im Code ein Fehler aufgetreten ist
- was die häufigsten Fehlerarten in Python sind
- wie Sie mit `try` und `except` erwartete Fehler abfangen können, damit Ihr Programm nicht abstürzt