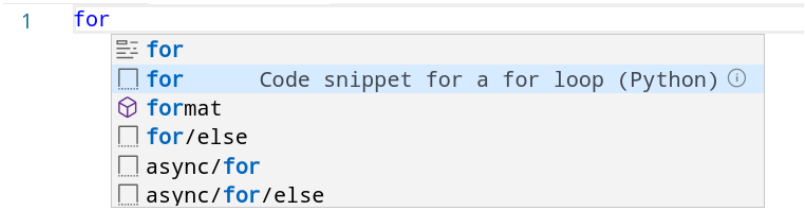
Editor-Trick des Tages: Snippets

Inzwischen können Sie Teile der Syntax von Python schon fast auswendig: `for i in range(len(liste)): print(i)` oder `if "a" in word: print(word)` sind Konstruktionen, die inzwischen schon so oft vorgekommen sind, dass es beinahe mühsam ist, immer alles selbst tippen zu müssen.

Die Python-Erweiterung von VSCode erlaubt uns, bestimmte Arten von Code-Blöcken mit Shortcuts einzufügen. Das ist hilfreich, weil Sie sich auf das Wesentliche konzentrieren können und sich nicht mehr so viel mit der äußeren Form dieser Blöcke beschäftigen müssen, wodurch z.B. die Tippfehler-Gefahr deutlich geringer wird.

Öffnen Sie VSCode und erstellen Sie eine neue Pythondatei. Wenn Sie jetzt das Wort `for` eintippen und darauf warten, dass der Tooltip eingeblendet wird, sehen Sie verschiedene Vorschläge für automatische Ergänzungen:



Das Symbol, mit dem der im Screenshot ausgewählte Eintrag der Liste markiert ist, steht für **Snippets**. Das sind Codeblöcke mit Platzhaltern, die wir mit eigenen Werten füllen können.

Probieren Sie es aus: Erzeugen Sie ein `for` -Snippet und schreiben Sie eine Schleife, die die Zahlen zwischen 0 und 10 ausgibt. Mit der Tab-Taste springen Sie vom aktuellen Platzhalter zum nächsten.

Wenn Sie möchten, können Sie auch Ihre eigenen Snippets definieren. Geben Sie dazu in der Command Palette `>configure snippets` ein und wählen Sie Python aus. In die Datei können Sie den folgenden Code einfügen:

```
    "Sign my python files!": {
            "prefix": "sign",
            "body": [
                "##########################################################################
    ##############",
                "# Aufgabe: $1
    #",
                "# Autorin: Esther Seyffarth
    #",
                "##########################################################################
    ##############"
            ]
        }
```

Ab jetzt können Sie in Ihren Pythondateien den Shortcut `sign` verwenden, um eine derartige Signatur einzufügen. Mehr Informationen zu Snippets finden Sie auf der Webseite von VSCode (https://code.visualstudio.com/docs/editor/userdefinedsnippets).

Einschub: Wiederholung zum Thema Rekursion

Rekursion ist ein Thema, das Ihnen bei fast allen Programmiersprachen begegnen wird. Seit wir Funktionen schreiben, haben wir theoretisch die Möglichkeit, Code in Endlosschleifen zu schreiben, nämlich indem eine Funktion sich selbst innerhalb ihres eigenen Funktionskörpers aufruft. Das birgt Gefahren, kann aber auch in einigen Fällen nützlich sein.

Besonders wichtig ist bei rekursiven Aufgaben, dass man eine **klar definierte Abbruchbedingung** angibt. So entsteht keine Endlosschleife, sondern es wird vor jedem rekursiven Funktionsaufruf geprüft, ob die Bedingung vielleicht erfüllt ist - falls ja, wird die Rekursion erfolgreich beendet.

Um Ihnen zu helfen, einen Ansatz für die Rekursionsaufgabe (10-02) aus der letzten Woche zu entwickeln, besprechen wir heute eine ähnliche Aufgabe. Die Herangehensweise lässt sich auf Aufgabe 10-02 übertragen.

Aufgabenstellung:

Schreiben Sie eine Funktion, die auf rekursive Weise einen String umdreht.

Herangehensweise:

1. Um einen String rekursiv umzudrehen, muss unsere Funktion diesen String als **Argument** übergeben bekommen. Nur so können wir dafür sorgen, dass der Wert, den wir betrachten, bei jedem rekursiven Funktionsaufruf immer kleiner wird - bis irgendwann die Rekursionsbedingung erfüllt ist.
2. Die Funktion muss unbedingt einen Wert zurückgeben, darf also kein leeres oder fehlendes `return` - Statement haben.

Dadurch ergibt sich als Funktionsgerüst zunächst folgendes:

In [ ]:

```
def string_umdrehen(eingabestring):
    print("?????")
    return "?????"
```

1. Jetzt ist ein guter Zeitpunkt, um über die **Abbruchbedingung dieser Rekursion** nachzudenken. Später können wir dann daran arbeiten, wie wir diese Bedingung erreichen können. Grundsätzlich wollen wir **pro Rekursionsschritt immer schrittweise kleinere Teilprobleme bearbeiten**. In diesem Fall können wir das z.B. dadurch erreichen, dass immer kleinere Ausschnitte des ursprünglichen Eingabestrings verarbeitet werden. Ab welcher Stringlänge können wir davon ausgehen, dass kein weiterer Rekursionsschritt Sinn macht?

In [ ]:

```
def string_umdrehen(eingabestring):
    if len(eingabestring) == 1:
        print("ABBRUCHBEDINGUNG!")
        return eingabestring
    else:
        return "?????"
```

1. Wir haben beschlossen, dass ein String, der aus nur einem Zeichen besteht, nicht mehr weiter zerteilt werden muss, damit wir ihn umdrehen können. Übrigens können wir ab jetzt bereits den Code ausführen und gucken, wie er sich verhält:

In [ ]:

```
def string_umdrehen(eingabestring):
    if len(eingabestring) == 1:
        print("ABBRUCHBEDINGUNG!")
        return eingabestring
    else:
        return "?????"

simple_case = "x"
complex_case = "xy"
print("+++ Simpler Fall: +++")
print("Original: {}\nUmgekehrt: {}".format(simple_case, string_umdrehen(simple_case)))
print()
print("+++ Komplexer Fall: +++")
print("Original: {}\nUmgekehrt: {}".format(complex_case, string_umdrehen(complex_case
)))
```

1. In unserer Funktion fehlt noch der eigentliche rekursive Aufruf. Wie können wir bei einem komplexen Fall (also bei Strings, auf die die Abbruchbedingung nicht zutrifft) den String so zerlegen, dass er erstens immer kleiner wird und zweitens zum Schluss der vollständige umgekehrte String zurückgegeben wird? Für diese Aufgabe bietet es sich an, **pro Aufruf ein Zeichen des Strings zu entfernen**.

In [ ]:

```
def string_umdrehen(eingabestring):
    print("Aktueller Teilstring: {}".format(eingabestring))
    if len(eingabestring) == 1:
        print("ABBRUCHBEDINGUNG!")
        return eingabestring
    else:
        return string_umdrehen(eingabestring[1:])

simple_case = "x"
complex_case = "xy"
print("+++ Simpler Fall: +++")
print("Original: {}\nUmgekehrt: {}".format(simple_case, string_umdrehen(simple_case)))
print()
print("+++ Komplexer Fall: +++")
print("Original: {}\nUmgekehrt: {}".format(complex_case, string_umdrehen(complex_case
)))
```

1. Wie Sie sehen, wird für den komplexen Fall zuerst der vollständige String `xy` ausgegeben, und dann die Funktion erneut aufgerufen, aber diesmal wird das erste Zeichen des Strings entfernt, bevor er als Argument an die Funktion übergeben wird (Slicing in Zeile 6). Leider stimmt die Rückgabe für den komplexen Fall noch nicht. Wir sind zwar mithilfe von Rekursion immer tiefer in die Aufgabe eingestiegen, haben aber die **Zwischenergebnisse noch nicht sinnvoll miteinander verknüpft**. Bisher entfernen wir pro Rekursionsschritt ein Zeichen des Strings - wo müsste dieses Zeichen wieder eingefügt werden, damit die Aufgabe korrekt gelöst ist?

In [ ]:

```
def string_umdrehen(eingabestring):
    print("Aktueller Teilstring: {}".format(eingabestring))
    if len(eingabestring) == 1:
        print("ABBRUCHBEDINGUNG!")
        return eingabestring
    else:
        anfang = eingabestring[0]
        rest = string_umdrehen(eingabestring[1:])
        print("Anfang: {}".format(anfang))
        print("Rest: {}".format(rest))
        umgedrehtes_teilwort = rest + anfang    # umgedrehte Reihenfolge!
        return umgedrehtes_teilwort

simple_case = "x"
complex_case = "xy"
print("+++ Simpler Fall: +++")
print("Original: {}\nUmgekehrt: {}".format(simple_case, string_umdrehen(simple_case)))
print()
print("+++ Komplexer Fall: +++")
print("Original: {}\nUmgekehrt: {}".format(complex_case, string_umdrehen(complex_case
)))
```

Um zu prüfen, ob die Aufgabe wirklich richtig gelöst ist, können wir jetzt ein längeres Wort ausprobieren. Ist das Ergebnis richtig?

In [ ]:

```
def string_umdrehen(eingabestring):
    print("Aktueller Teilstring: {}".format(eingabestring))
    if len(eingabestring) == 1:
        print("ABBRUCHBEDINGUNG!")
        return eingabestring
    else:
        anfang = eingabestring[0]
        rest = string_umdrehen(eingabestring[1:])
        print("Anfang: {}".format(anfang))
        print("Rest: {}".format(rest))
        umgedrehtes_teilwort = rest + anfang    # umgedrehte Reihenfolge!
        print(umgedrehtes_teilwort)
        return umgedrehtes_teilwort

complex_case = "Computerlinguistik"
print("+++ Komplexer Fall: +++")
print("Original: {}\nUmgekehrt: {}".format(complex_case, string_umdrehen(complex_case
)))
```

Zusammenfassung

Das Wichtigste beim Entwickeln rekursiver Funktionen ist die Abbruchbedingung. Sorgen Sie außerdem dafür, **dass die Abbruchbedingung erreicht werden kann**, und zwar indem Sie den ursprünglichen Wert des Arguments immer weiter zerlegen, bis die Aufgabe trivial wird (ein String mit nur einem Zeichen ist identisch zu seiner umgedrehten Variante). Die Funktion muss in jedem Fall einen **sinnvollen Rückgabewert** haben, damit die Zwischenergebnisse gut zusammengefügt werden können.

Hier noch einmal der Code für die rekursive String-umdreh-Funktion, ohne die Zwischenausgaben und etwas aufgeräumter:

In [ ]:

```
def string_umdrehen(eingabestring):
    if len(eingabestring) == 1:
        return eingabestring
    else:
        anfang = eingabestring[0]
        rest = string_umdrehen(eingabestring[1:])
        umgedrehtes_teilwort = rest + anfang    # umgedrehte Reihenfolge!
        return umgedrehtes_teilwort

complex_case = "Computerlinguistik"
print("Original: {}\nUmgekehrt: {}".format(complex_case, string_umdrehen(complex_case
)))
```

Guter Programmierstil in Python

Wir haben in den letzten Wochen schon gelegentlich darüber gesprochen, an welche Konventionen Sie sich beim Programmieren halten können, um gut lesbaren (und dadurch gut wartbaren) Code zu produzieren.

Einer der Grundsätze von Python lautet *Readability counts.* Deshalb gibt es eine formalisierte Sammlung an Empfehlungen für Stilfragen beim Programmieren, die Teil der technischen Spezifikation von Python ist. Diese Sammlung heißt **PEP8**.

PEP steht für *Python Enhancement Proposal*. Alle PEPs sind unter https://www.python.org/dev/peps/ (https://www.python.org/dev/peps/) abrufbar, sortiert nach allgemeinen Themen. Hier können Sie z.B. nachlesen, warum Python 2 nicht weiterentwickelt wird und die Migration zu Python 3 schon seit 2011 empfohlen wird: PEP 404 -- Python 2.8 Un-release Schedule (https://www.python.org/dev/peps/pep-0404/)

In den PEPs können Sie nachlesen, wie die Sprache Python sich über die Jahre hinweg weiterentwickelt, basierend auf Vorschlägen aus der Developer-Mailingliste. Als Hilfe beim Programmieren werden Sie weiterhin die Dokumentation und StackOverflow verwenden, aber wenn Sie sich mal fragen, warum bestimmte Verhaltensweisen von Python so und nicht anders konstruiert wurden, werden Sie in der PEP-Liste vielleicht fündig.

Im Text von PEPs finden Sie oft das Akronym BDFL. BDFL steht für **Benevolent Dictator for Life**, ein Titel, den Guido van Rossum - der Erfinder von Python - bis Mitte 2018 innehatte. Historisch war ein Statement des BDFL oft der entscheidende Impuls für die Annahme oder Ablehnung einzelner PEP-Vorschläge.

PEP8

In PEP8 sind Empfehlungen gelistet, wie wir Pythoncode verfassen. Da diese Empfehlungen exzessiv in der Community diskutiert wurden und inzwischen Standard sind, gibt es **Tools zum Prüfen der Einhaltung von PEP8-Richtlinien**.

Auch die Python-Extension, die wir in VSCode verwenden, kann das für uns erledigen. Um die Extension dafür einzurichten, tun wir folgendes:

1. Mit `Ctrl + Shift + P` die Command Palette öffnen
2. Eingeben: `>python select linter`
3. Kommando mit `Enter` bestätigen
4. In der Eingabezeile, die jetzt erscheint, `pycodestyle` eingeben und mit `Enter` bestätigen.

pycodestyle installieren: Falls VSCode Sie dazu auffordert, pycodestyle zu installieren, können Sie das tun. Auf den Uni-PCs werden installierte Pakete allerdings nachts wieder gelöscht.

Linter (https://en.wikipedia.org/wiki/Lint_%28software%29) sind Tools, die Ihren Code automatisch auf die Einhaltung vordefinierter Vorgaben prüfen. Dabei kann man die Fehler entweder automatisch beheben lassen oder sich einfach anzeigen lassen, wo Unstimmigkeiten gefunden wurden, um sie per Hand zu beheben.

Je nachdem, wie Ihr Editor eingestellt ist, wird Ihr Code nach der Auswahl des `pycodestyle` -Linters entweder direkt analysiert, oder Sie müssen den Linter explizit ausführen (Command Palette öffnen und `>python run linting` ausführen).

```
30  # Hier wird eine Funktion definiert, die zufällige Sätze generieren soll. Die
31  # Funktion wird ohne Argumente definiert, muss also später ohne Argumente
32  # aufgerufen werden.
33  def zufaelligen_satz_generieren():
34
35      # Hier wird eine Liste von Namen angelegt.
36      moegliche_namen = {"Kim", "Pat", "Sam", "Alex"}
37
38      # Hier wird der zusätzliche Name "John" in die Liste der Namen eingefügt.
39      moegliche_namen = moegliche_namen.append("John")
40
41      # Hier wird eine Liste mit transitiven Verben angelegt.
42      transitive_verben = ["liebt", "sieht", "kennt", "versteht"]
43
44      # Hier wird das zusätzliche Verb "fragt" in die Liste der Verben eingefügt.
45      transitive_verben.append(["fragt"])
46
47      satz = "{} {} {}."      # Hier wird ein formatierbarer String angelegt.
48                              # Unten wird er dann mit konkreten Werten gefüllt.
```

Wie Sie sehen, werden Fehler rot unterstrichen, wie man das z.B. von Word kennt.

Wenn Sie sich alle Fehler in der Datei anzeigen lassen möchten, können Sie die Liste aller Probleme entweder im Menü oben unter `View -> Problems` anzeigen lassen oder den Befehl in der Command Palette eingeben oder den Shortcut dafür verwenden (bei mir `Ctrl + Shift + M`). Sie sehen dann in etwa Folgendes:

Die Fehlermeldungen sind meist einfach zu lesen.

In meiner Liste oben sind sieben Errors - klare Verstöße gegen PEP8 - und einige Warnungen zu sehen. (Die Fehlercodes sind auf der Webseite des `pycodestyle` -Tools (http://pycodestyle.pycqa.org/en/latest/intro.html#error-codes) aufgeschlüsselt.)

Jede Zeile in der Liste der "Problems" hat folgendes Format:

- **`line too long (80 > 79 characters)`** - sprachliche Beschreibung des Problems. Meistens informativ genug, um das Problem zu beseitigen.
- **`pycodestyle`** - Name des Tools, das die Probleme entdeckt hat. (Wenn wir einen anderen Linter wählen, steht hier auch ein anderer Name.)
- **`(E501)`** - Fehlercode. Den Code können Sie googeln oder in der `pycodestyle` -Doku nachlesen.
- **`[24, 80]`** - Problem wurde in Zeile `24` an Position `80` lokalisiert. Sie können die Zeilennummer im Editor suchen und das Problem identifizieren. Oder Sie klicken auf einen Eintrag in der Liste der Probleme, dann wird die entsprechende Codezeile markiert, sodass Sie nicht suchen müssen.

Aufgabe

1. Kopieren Sie den folgenden Code in eine neue Datei in VSCode und speichern Sie das Programm als Pythondatei ab. Lassen Sie sich dann vom Linter alle Probleme des Codes anzeigen. Können Sie anhand der Hinweise das Programm so umschreiben, dass der Linter zufrieden ist?

```
BNC_PATH = "D:/corpora_and_data/english/annotated/BNC/BNC-parsed/bnc-fi-prsd.tx
t"

def collect_plural_nouns(bncpath):


    plurals = set()
    with open( bncpath, "r", encoding="utf8" ) as bncfile :
        for line in bncfile:
            if line.strip() != False:
                if "\tNNS\t" in line:
                   #print(line)
                   plurals.add(line.split()[1])

    with open("plurals.txt", "w", encoding="utf8") as outfile:
        for v in plurals:
            print("\"{}\", ".format( v.lower() ), file=outfile, end="")

    return plurals
words = collect_plural_nouns(BNC_PATH)
#print(words)
```

Nicht immer wollen wir PEP8 hundertprozentig befolgen. Die Regel, dass Zeilen nicht länger als 80 Zeichen sein sollen, ist zum Beispiel bei vielen Pythonistas unbeliebt: Wegen der Einrückungen (laut PEP8: 4 Leerzeichen pro Ebene) bleibt einem in komplexen Programmen oft nur **wenig Platz** für den eigentlichen Code.

Trotzdem ist es empfehlenswert, dass Sie sich mit den Regeln von PEP8 vertraut machen. Gewöhnen Sie sich an, in VSCode vor der Fertigstellung von Programmen immer mindestens einmal die Probleme vom Linter ausgeben zu lassen und pro Eintrag in der Liste **bewusst zu entscheiden**, ob Sie dagegen etwas unternehmen möchten.

Für Aufgaben wie 09-01 sind Linter das perfekte Tool. Falls Ihnen selbst keine Fehler oder Unstimmigkeiten auffallen, können Sie mit einem Linter prüfen, ob es noch irgendwelche **versteckten Probleme** gibt.

Debugging

Linter können Verstöße gegen Syntaxregeln oder Stilvorgaben identifizieren, aber sie helfen nicht, wenn wir Code schreiben, der einwandfrei aussieht und *trotzdem* nicht das Verhalten zeigt, das wir geplant haben.

Es gibt zum Glück auch Tools, die uns helfen, andere Formen von Programmierfehlern zu finden und zu lösen: Solche Fehler, bei denen das Programm nicht abstürzt und auch kein Linter anspringt, die aber dazu führen, dass **der Output des Programms nicht den Erwartungen entspricht**.

Hier ein typisches Beispiel aus meiner Anfangszeit als Programmiererin:

In [ ]:

```
# Aufgabe: Alle Zeichen aus einem vom User eingegebenen String
#          in einer Liste sammeln

def collect_chars_from_string(input_string):
    output_list = []
    for c in input_string:
        # Anzeige des aktuellen Zeichens
        print("aktuelles Zeichen: {}".format(c))

        # Aktuelles Zeichen an die Liste anhängen
        output_list = output_list.append(c)

    return output_list

full_string = input("Geben Sie irgendetwas ein: ")
characters = collect_chars_from_string(full_string)

print(characters)
```

Ausgabe des Programms:

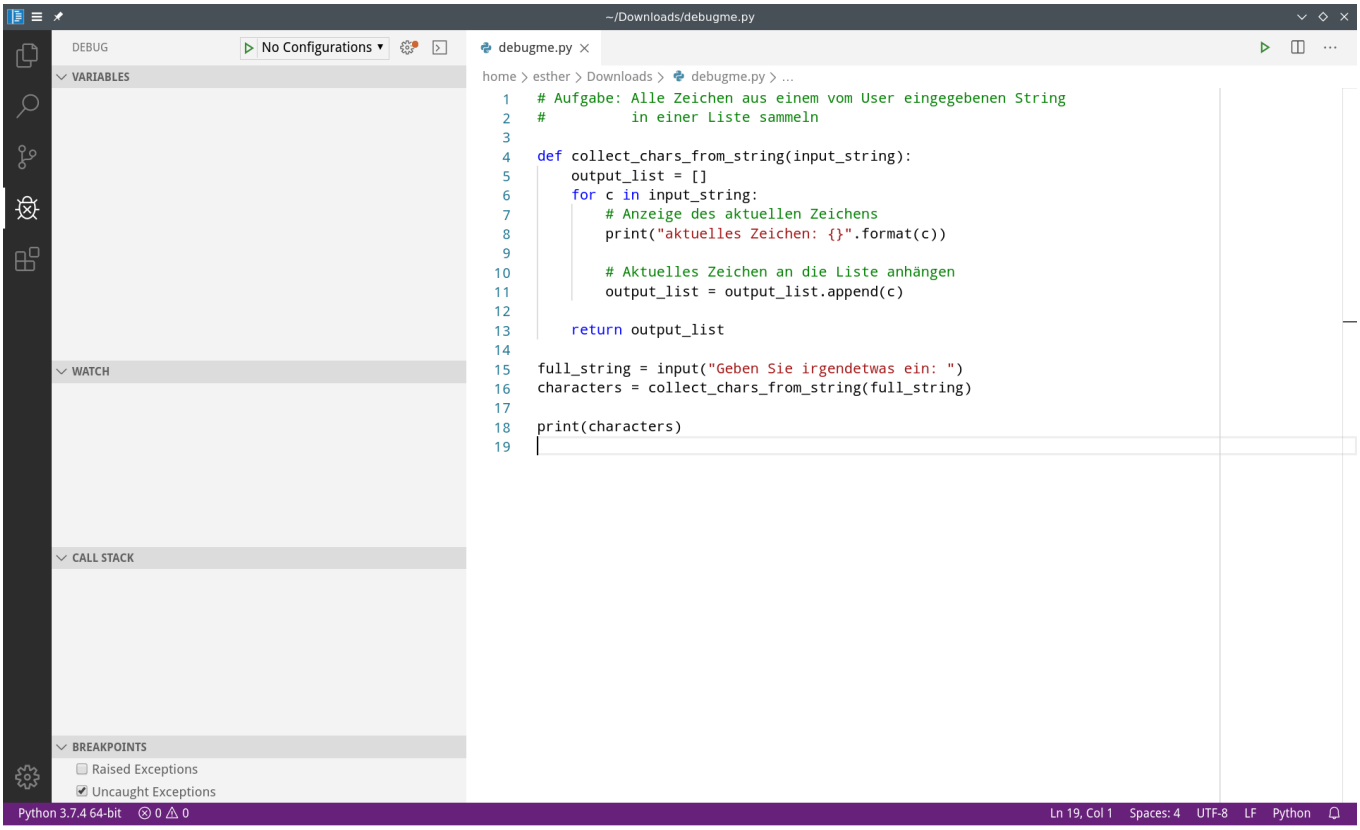
```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-1-4a3d0b9a1ba5> in <module>
     14
     15 full_string = input("Geben Sie irgendetwas ein: ")
---> 16 characters = collect_chars_from_string(full_string)
     17
     18 print(characters)

<ipython-input-1-4a3d0b9a1ba5> in collect_chars_from_string(input_string)
      9
     10         # Aktuelles Zeichen an die Liste anhängen
---> 11         output_list = output_list.append(c)
     12
     13     return output_list

AttributeError: 'NoneType' object has no attribute 'append'
```

Der Linter hilft uns hier nicht weiter: Es werden höchstens Warnungen angezeigt, aber keine Errors (das Programm ist ja auch ausführbar).

Mit dem Debugger in VSCode können wir das Programm Zeile für Zeile ausführen und prüfen, ob es in jedem Schritt wirklich das tut, was gewünscht ist. Um den Debugger zu starten, können Sie links im Menü auf das Debugger-Symbol klicken oder die Command Palette verwenden (Kommando: `>view show debug`).



Links sehen wir eine (bisher leere) Übersicht über Variablen, *Watches*, den *Call Stack* und die *Breakpoints*.

Variables wird später beim Starten des Debuggers alle Namen und ihre jeweiligen Werte auflisten.

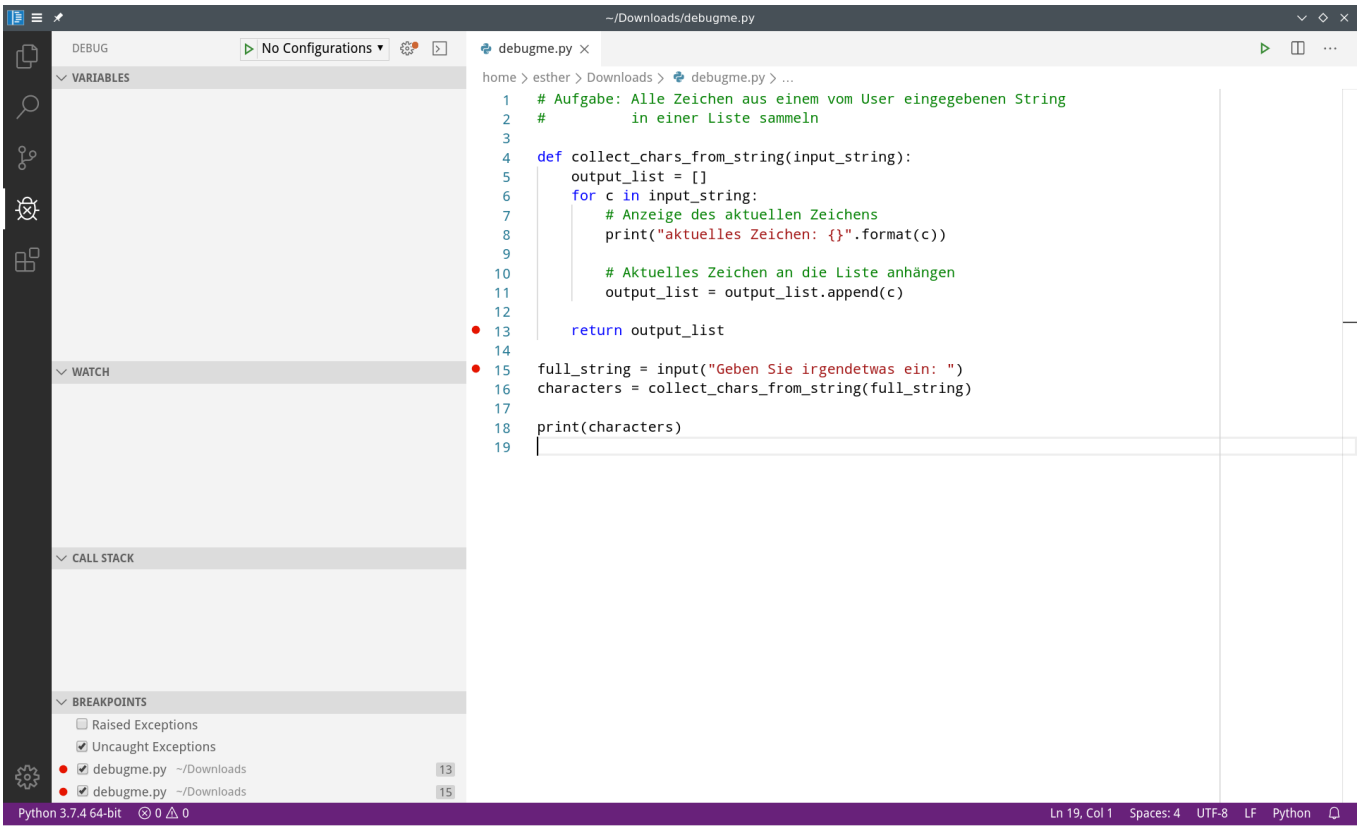
Watches können wir selbst anlegen: So können wir den Wert bestimmter Variablen bzw. komplexer Ausdrücke zu jedem Zeitpunkt auf den ersten Blick erkennen.

Der **Call Stack** ist verwandt mit dem **Traceback**, den wir schon aus unseren Fehlermeldungen kennen: Hier sehen wir, ob wir uns gerade in einem Funktionsaufruf befinden bzw. wie verschachtelt die aktuellen Funktionsaufrufe sind.

Breakpoints sind für den Start besonders wichtig: Damit legen wir fest, wann der Debugger die Ausführung des Programms pausiert, damit wir uns den Zustand des Programms näher anschauen können. Der Interpreter wird das Programm zunächst normal ausführen; beim ersten Breakpoint hält er an und wir können prüfen, wie das Programm sich verhält.

Für die aktuelle Übung setzen wir den Breakpoint einfach in die erste Zeile, die vom Interpreter ausgeführt wird. Das ist Zeile 15 im Code oben. Um den Breakpoint zu setzen, klicken wir mit der Maus **links neben die Zeilennummer**, sodass ein roter Punkt erscheint. Wir können beliebig viele Breakpoints setzen, z.B. in Zeile 15 und zusätzlich in Zeile 13 (mit der Vermutung, dass die Rückgabe der Liste ein Problem ist).

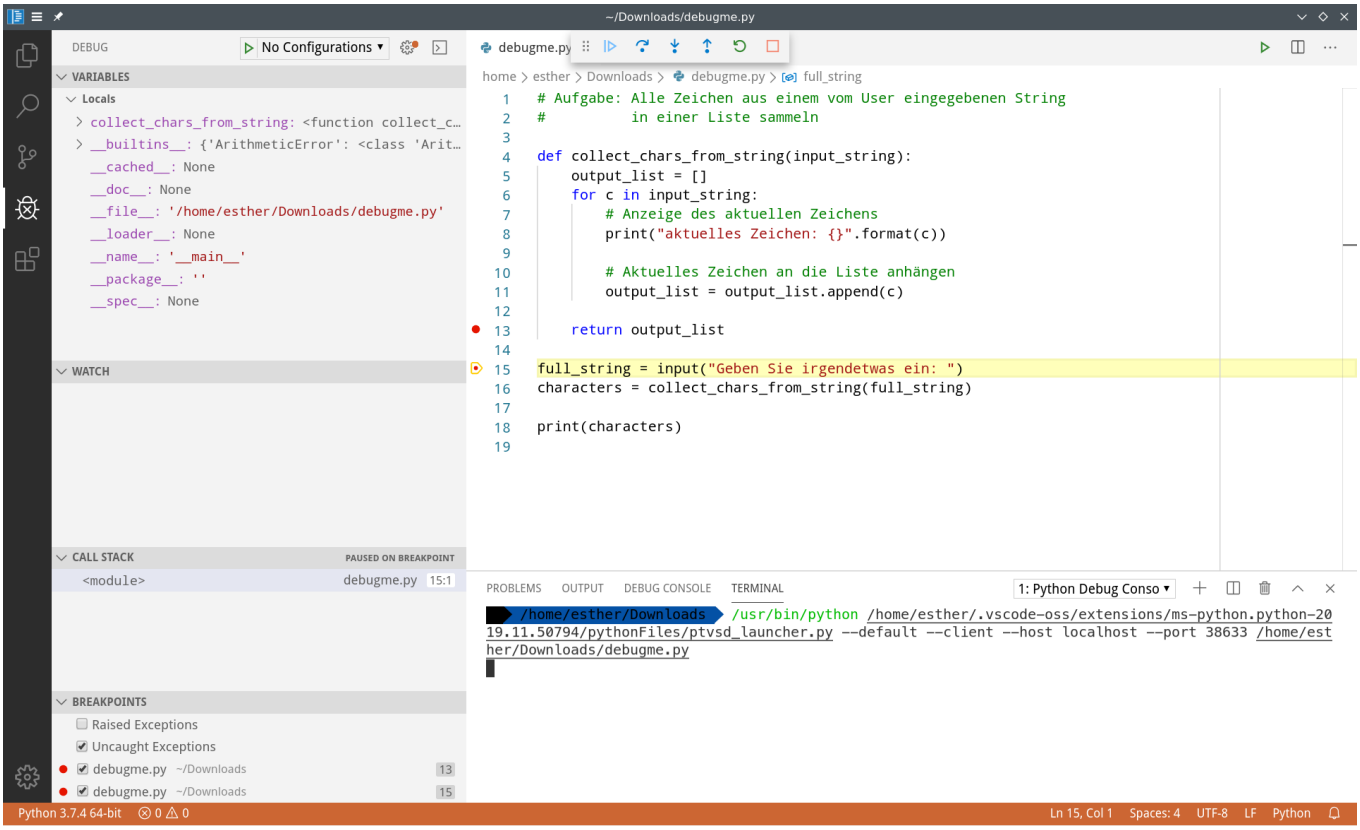
Wie Sie sehen, erscheinen unsere beiden Breakpoints sofort unten links in der Übersicht.



Es kann losgehen! Starten Sie den Debugger mit dem "Play"-Symbol oben links oder durch Auswahl von `Debug -> Start Debugging` oder durch Verwendung des Keyboard Shortcuts (bei mir: `F5`) oder mit der Command Palette. Wir müssen dem Debugger zusätzlich noch sagen, dass der Python-Interpreter verwendet werden soll (`Python` oder `Python File`).

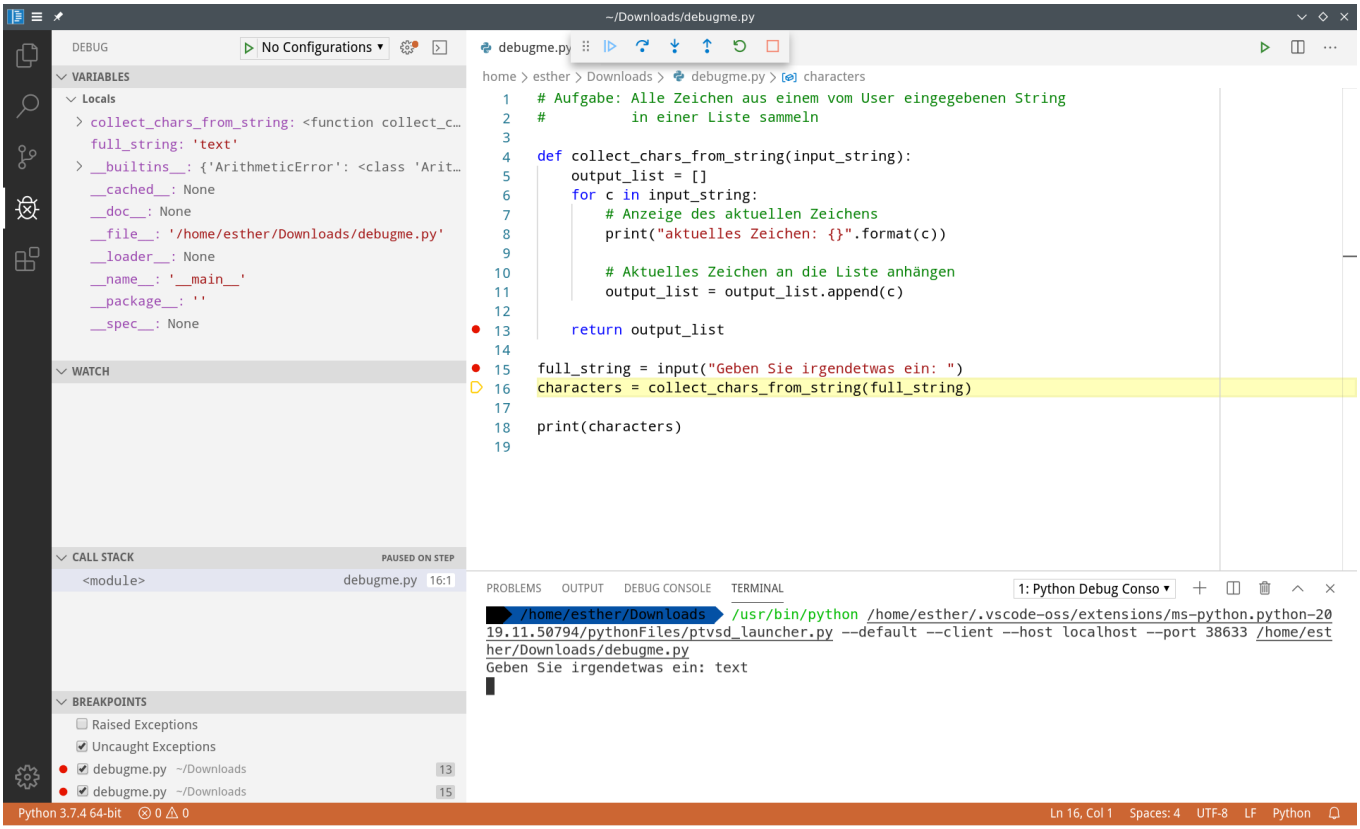
Da wir einen Breakpoint in Zeile 15 gesetzt haben, pausiert der Interpreter an diesem Punkt. Sie sehen jetzt, dass die Boxen links sich gefüllt haben: Unter **Variables** sind jetzt einige Werte verfügbar. Wie in der Sitzung vom 05.11.2019 sehen wir, dass die `__builtins__` Teil des lokalen Namespace sind. Außerdem ist die von uns definierte Funktion, `collect_chars_from_input_string()`, bekannt, weil der Interpreter sie in den Speicher gelesen hat, bevor er in Zeile 15 angekommen ist.

Im **Call Stack** sehen wir, dass wir uns momentan im `<module>` befinden - also auf der obersten Hierarchieebene des Programms. Wir erfahren hier auch, dass der Debugger in Zeile 15, Position 1 angehalten hat, und zwar in der Datei `debugme.py`.



Da dieser Teil des Programms noch nicht so interessant ist, wollen wir einen Schritt weitergehen. Dazu haben wir mehrere Möglichkeiten: Wir können in der Steuerleiste des Debuggers (oben Mitte) entweder **bis zum nächsten Breakpoint** "vorspulen" (erster Knopf); oder wir springen in die **nächste ausführbare Zeile** des Programms (zweiter Knopf); oder wir springen **in die Methode, die als nächstes ausgeführt werden soll, hinein** (dritter Knopf).

Der Interpreter hält in Zeile 15 an, bevor die Zeile tatsächlich ausgeführt wird. Da wir uns nicht dafür interessieren, was bei `input()` passiert (die Methode ist Teil der Sprache Python und mit hoher Wahrscheinlichkeit korrekt implementiert...), wählen wir Option 2 und springen in die nächste Zeile, also in Zeile 16. Zwischendurch geben wir den String unten im Terminal ein, nach dem wir gefragt werden, und bestätigen die Eingabe mit `Enter`.

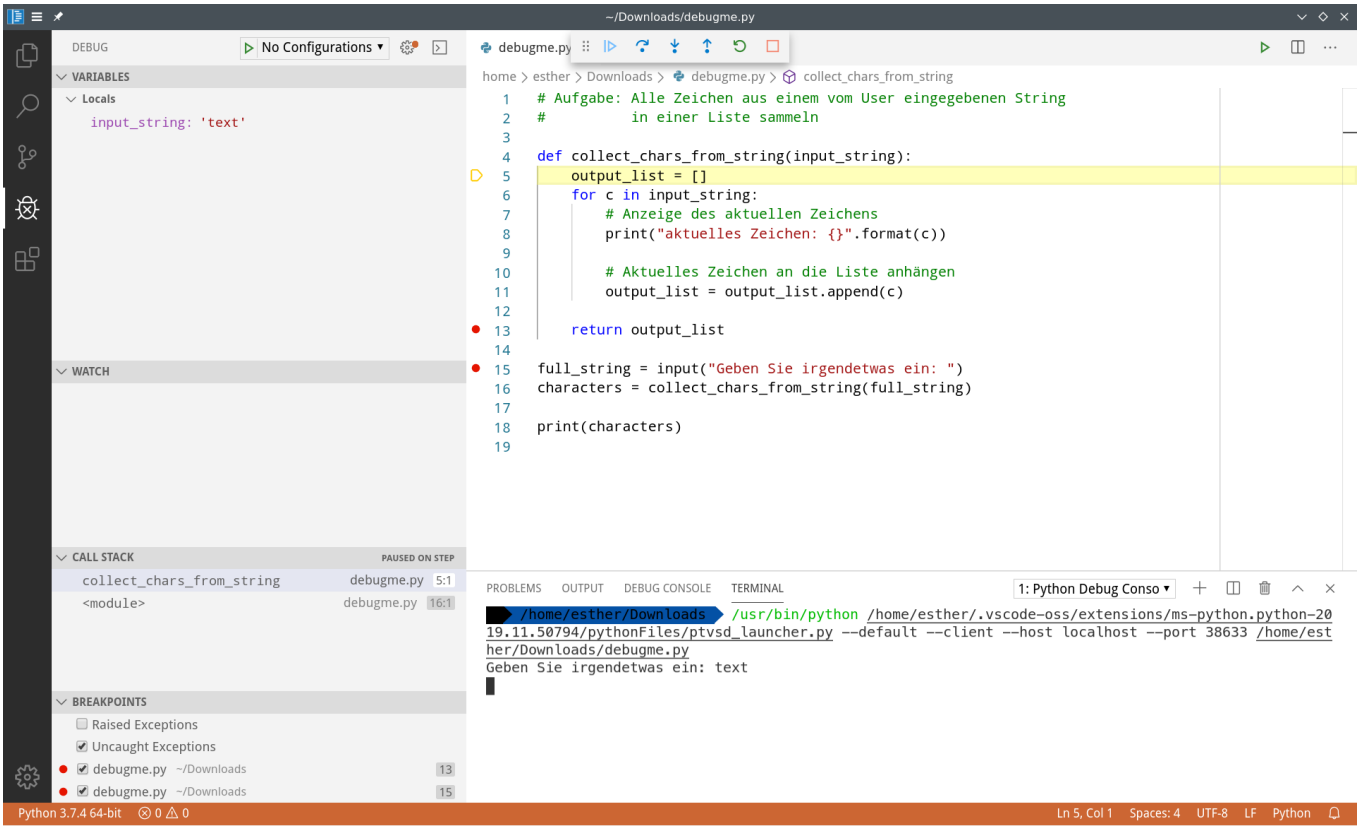


Wir sehen jetzt, dass links unter **Variables** ein neuer Eintrag erschienen ist: Der Interpreter hat nach dem Ausführen von Zeile 15 einen neuen Namen dem Namespace hinzugefügt, und zwar die Variable `full_string`, in der der Wert `text` gespeichert ist.

Als nächstes können wir in die Funktion `collect_chars_from_string()` springen, indem wir den dritten Knopf in der Steuerleiste wählen.

In der **Variables**-Box hat sich jetzt so einiges geändert: Es wird jetzt der Namespace der Funktion angezeigt. Er enthält lokale Variablen, nämlich `output_list` und `c`, die aber in Zeile 5 noch nicht belegt sind; und außerdem Argumente, also die Werte, die beim Aufruf der Funktion übergeben wurden.

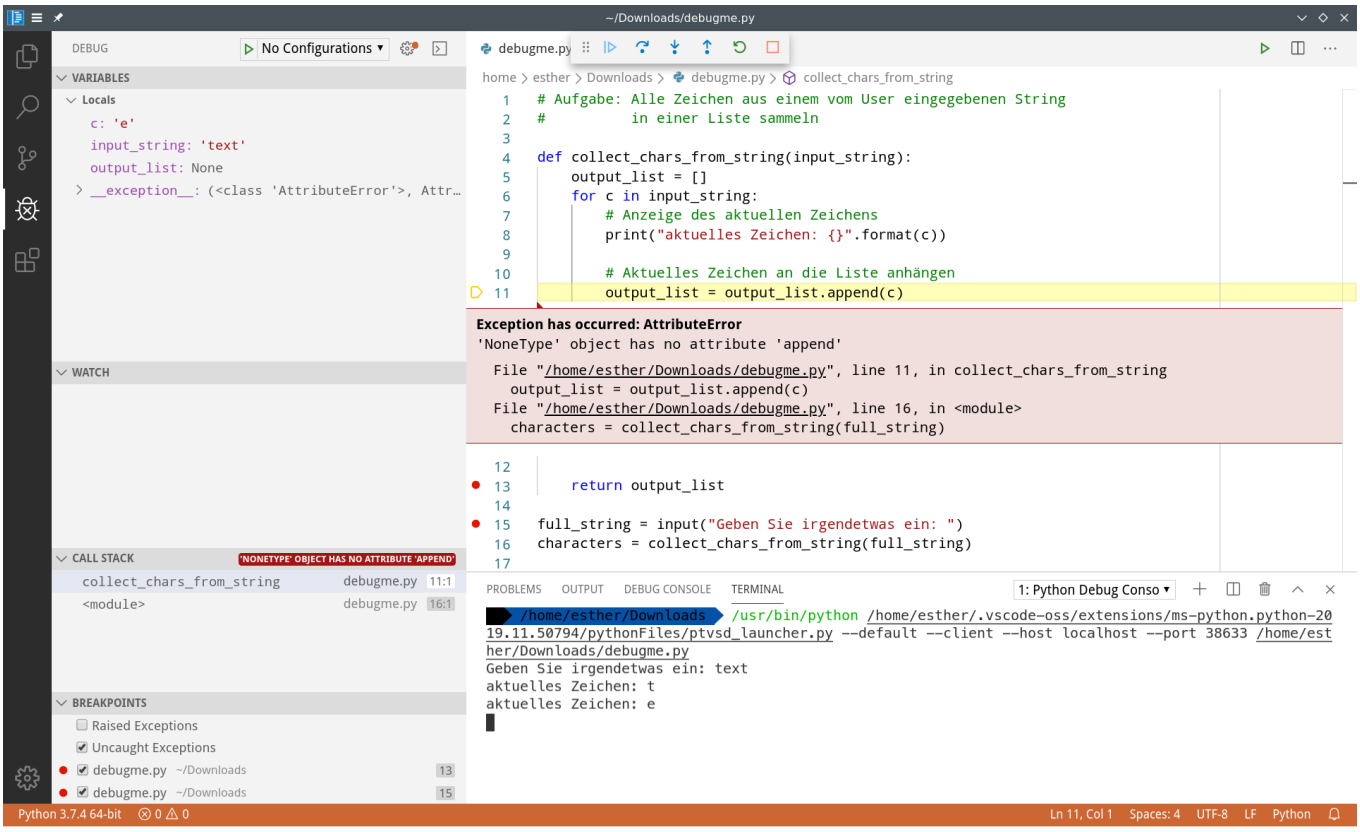
Im **Call Stack** sehen wir, dass der Stack gewachsen ist: Wir sind in die Funktion



`collect_chars_from_string()` gesprungen, also wird diese Funktion oberhalb des vorher vorhandenen Eintrags angezeigt.

Call stack bedeutet wörtlich "Aufrufstapel" und bezeichnet ein Konstrukt, das einem Stapel tatsächlich ähnelt. Der Stapel wird leer angelegt (vgl. erstes Bild vor dem Starten des Debuggers). Dann wird ein Element auf den Stapel gelegt (vgl. `<module>` im Bild zum ersten Breakpoint). Alle weiteren Elemente werden *oberhalb* der bestehenden Einträge gestapelt. Der Stack kann dann nur in der gleichen Reihenfolge wieder abgebaut werden, in der er aufgebaut wurde: Erst muss das oberste Element ordnungsgemäß beendet werden, dann das nächstobere usw.

Da wir am Schluss der Funktion einen Breakpoint gesetzt haben, können wir als nächstes mal den ersten Knopf in der Steuerleiste testen. Oder?



Schade. Der Fehler tritt also irgendwo zwischen Zeile 5 und Zeile 13 auf. (Natürlich konnten wir das eigentlich auch schon an der ursprünglichen Ausgabe sehen.)

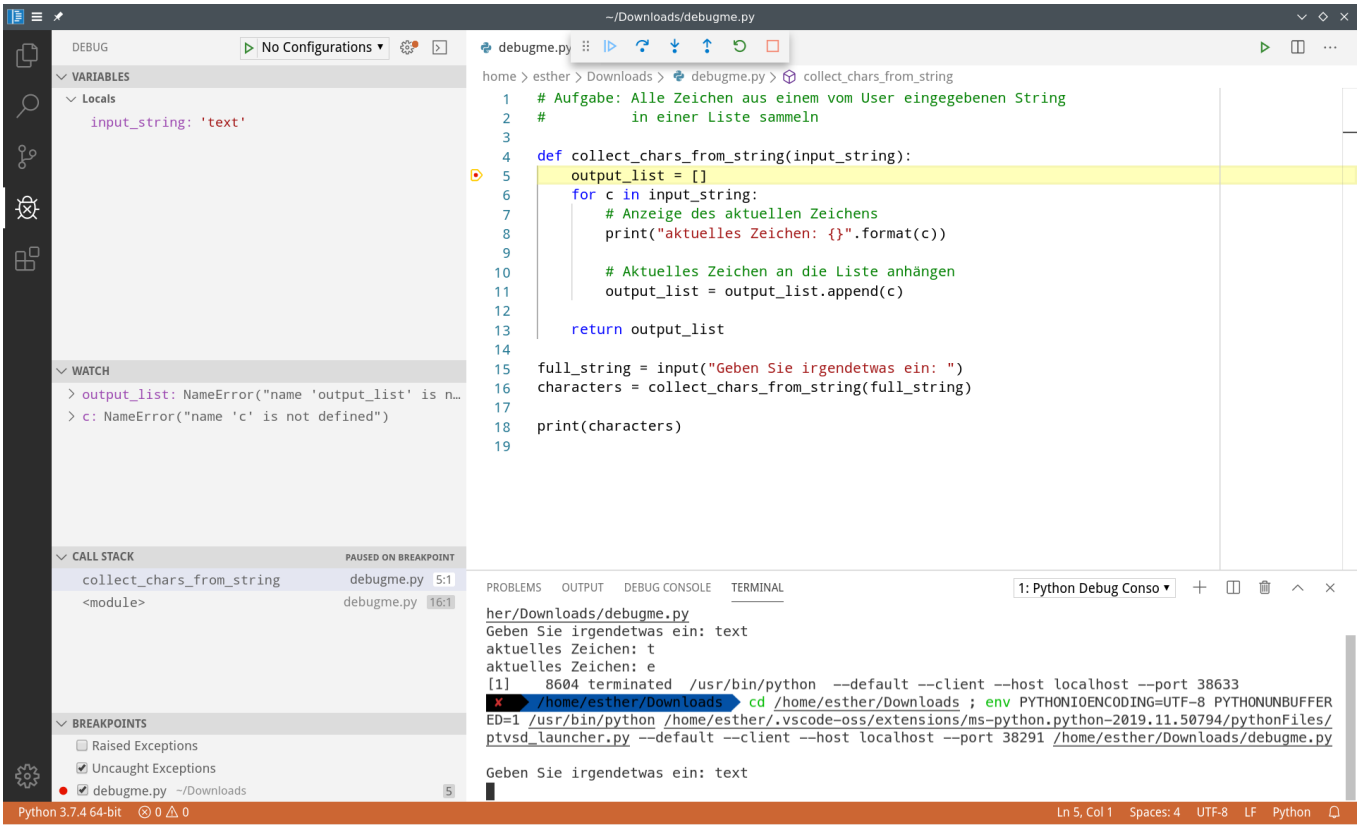
Starten wir den Debugger neu. Diesmal wollen wir genau wissen, welchen Wert die Variablen `output_list` und `c` zu jedem Zeitpunkt haben. Und wir klicken uns per Hand Schritt für Schritt durch die Funktion und durch die Schleife, um zu sehen, wo das Problem ist.

Um die Variablen im Auge zu behalten, die uns interessieren, richten wir eine **Watch** ein. Neben dem Wort "Watch" klicken wir auf das +, um einen Ausdruck einzugeben, dessen Wert immer angezeigt werden soll. Mit `Enter` bestätigen wir die Eingabe.

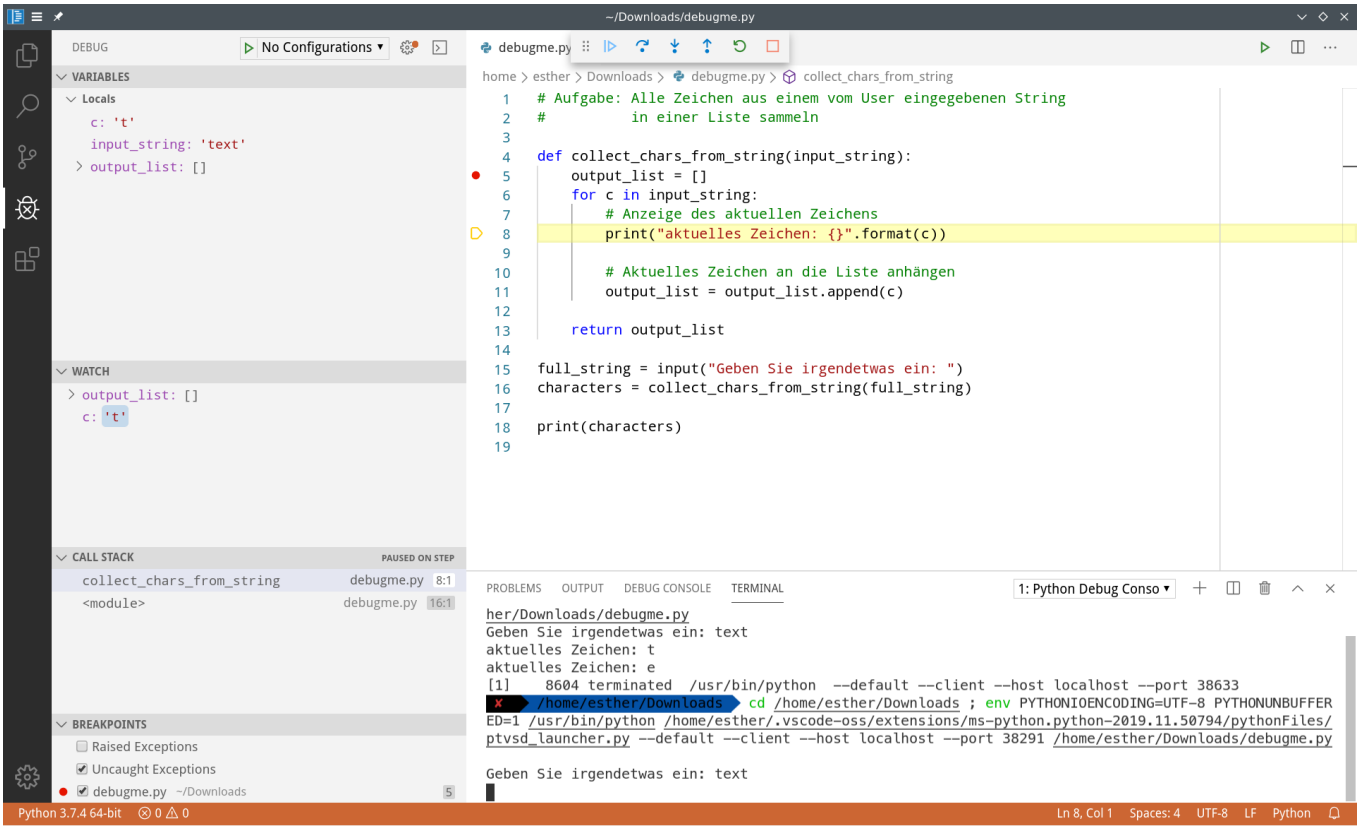
Richten Sie zwei Watches ein: Eine für `output_list` und eine für `c`.

Der Anfang des Programms schien ja ganz in Ordnung zu sein. Wir entfernen die bisherigen Break Points und setzen einen neuen Break Point, und zwar in der ersten Zeile der Funktion (Zeile 5). Von dort aus können wir uns schrittweise durchklicken ("Step Over"), bis wir das Problem finden.

Wenn wir den Debugger neu starten, sieht die Box für die Watches fehlerhaft aus: Es wird bei beiden Variablen ein `NameError` angezeigt. Das liegt daran, dass beim ersten Breakpoint noch keine Werte für diese Variablen bekannt sind. Das ist aber nicht weiter schlimm. Sobald die Variablen, die uns interessieren, angelegt werden - in Zeile 5 bzw. 6 -, verschwinden die `NameError`s.

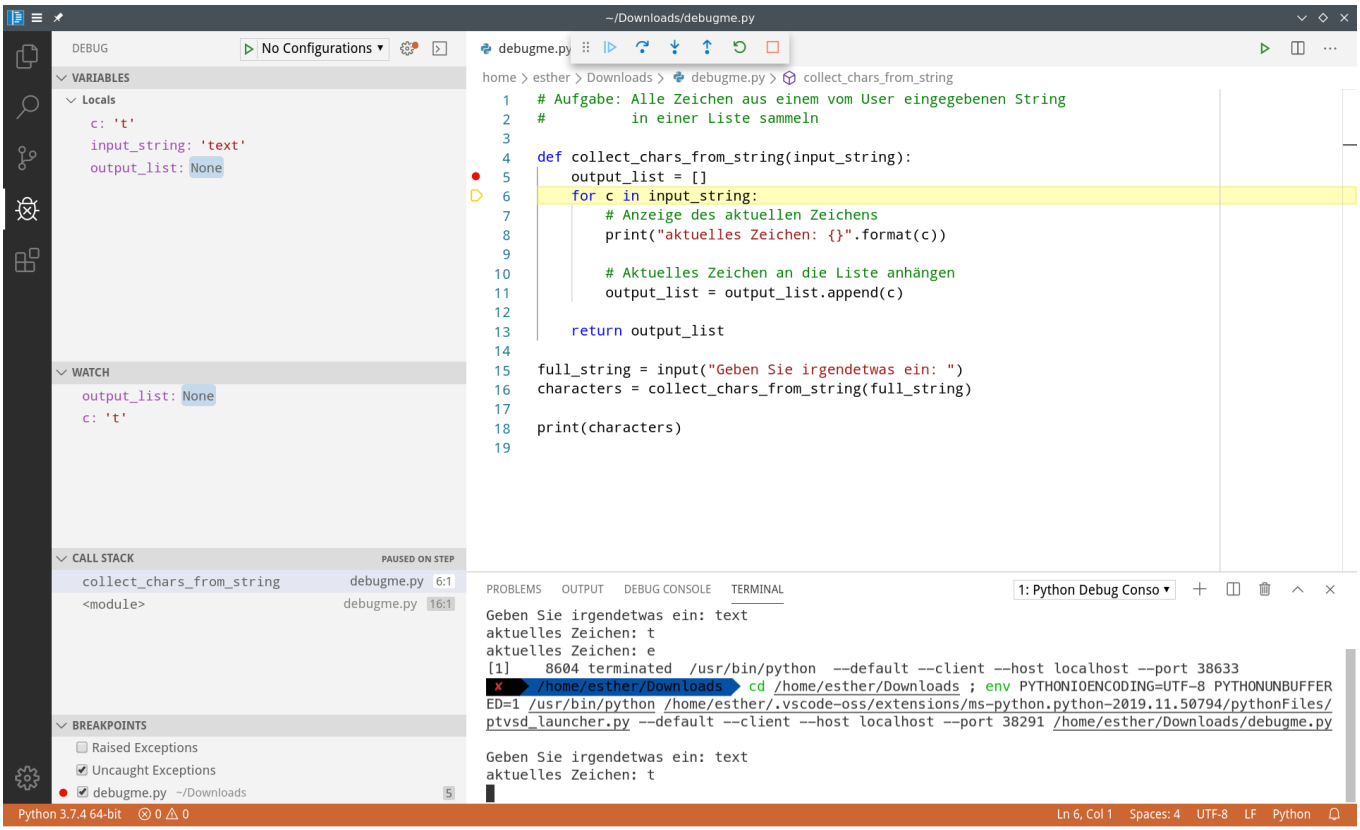


Sobald Sie z.B. Zeile 8 erreichen, sind beide `NameError` s weg. Stattdessen werden die aktuellen Werte der Variablen angezeigt. Bisher sieht alles noch so aus wie erwartet: Die Liste ist leer (weil noch kein Element eingefügt wurde), das `c` ist das erste Zeichen des Eingabe-Strings.



In der **Watch**-Box werden Werte, die sich vor kurzem geändert haben, gesondert hervorgehoben (vgl. den Wert von `c` im letzten Screenshot, bei dem der Debugger als letztes Zeile 6 ausgeführt hat, die den Wert von `c` festlegt).

Klicken Sie "Step Over", bis Sie im nächsten Schleifendurchlauf ankommen. Sie sehen folgendes:



Der Typ der `output_list` hat sich geändert. Wir wollten alle Zeichen nacheinander in einer Liste sammeln, aber jetzt hat die Variable den Typ und den Wert `None`. Damit können wir nicht operieren.

Deshalb wird der nächste Versuch, Zeile 11 auszuführen, fehlschlagen. (Überzeugen Sie sich davon selbst, indem Sie noch einige Male "Step Over" klicken, bis das Programm abstürzt.)

Der Debugger zeigt uns genau, dass zwischen Zeile 11 (im ersten Schleifendurchlauf) und Zeile 6 (wenn sie den zweiten Schleifendurchlauf einleitet) ein Problem auftritt. Wir wissen jetzt also, dass wir **in Zeile 11** ansetzen müssen, um das Problem zu beseitigen.

Möglicherweise haben einige von Ihnen bisher in den Übungen **viele Aufrufe von `print()`** in den Code eingebaut, um zwischendurch zu sehen, welchen Wert einzelne Variablen haben. Dagegen ist im Übungskontext auch nichts einzuwenden. In produktiven Systemen soll `print()` typischerweise nur dann verwendet werden, wenn es ein integraler Bestandteil der Funktionalität des jeweiligen Programms ist.

Eine weitere Funktion des Debuggers, die wir eben noch gar nicht genutzt haben, ist die `Debug Console`. Unten beim Terminal sehen Sie eine Schaltfläche zum Anzeigen dieser Konsole. Sie ermöglicht es Ihnen, **während des Debuggens beliebigen Code auszuführen**, und zwar unter Verwendung aktueller Variablenwerte. Probieren Sie die Konsole demnächst aus, wenn Sie debuggen und sich für aktuelle Werte nur einmal interessieren (Watches sind eher geeignet für Variablen, die sich oft verändern, z.B. im Verlauf von Schleifen).

Effiziente Programmierung

Inzwischen haben wir gesehen, wie wir Fehler während der Laufzeit mit `try` und `except` behandeln können und wie wir mit dem Debugger herausfinden können, an welcher Stelle unser Pythoncode fehlerhaft ist.

Um noch besser und effizienter zu programmieren, hilft es, wenn wir eine Vorstellung von der **Komplexität** einzelner Teile unseres Programms haben.

Oft gibt es mehrere Möglichkeiten, bestimmte Dinge mit Pythonbefehlen zu erreichen. Erinnern Sie sich beispielsweise an die Aufgabe, einen String umzudrehen. Wir hatten auf StackOverflow folgende Lösung dafür gefunden:

In [ ]:

```
def reverse_string_with_slicing(input_string):
    output_string = input_string[::-1]
    return output_string

print(reverse_string_with_slicing("Hallo Welt"))
```

Vorhin haben wir einen anderen Ansatz zum Umdrehen von Strings selbst entwickelt, um Rekursion zu üben:

In [ ]:

```
def string_umdrehen_rekursiv(eingabestring):
    if len(eingabestring) == 1:
        return eingabestring
    else:
        anfang = eingabestring[0]
        rest = string_umdrehen_rekursiv(eingabestring[1:])
        umgedrehtes_teilwort = rest + anfang    # umgedrehte Reihenfolge!
        return umgedrehtes_teilwort

print(string_umdrehen_rekursiv("Hallo Welt"))
```

Wie Sie sehen, verhalten beide Funktionen sich gleich. Welche Lösung **leichter zu lesen** ist, ist nicht ganz klar - die Anweisung `input_string[::-1]` ist nicht so einfach zu verstehen, aber die rekursive Lösung hat mehr Codezeilen und sieht damit komplizierter aus.

Was uns darüber hinaus interessiert, ist die **Dauer jeder Lösung**. Daran, wie lange die Funktion braucht, können wir in etwa ablesen, wie effizient sie programmiert ist.

Um die Dauer zu messen, die ein Codebeispiel benötigt, können wir `timeit` verwenden. So setzen wir `timeit` ein:

In [ ]:

```
import timeit    # Modul timeit importieren

# erste Funktion definieren
def reverse_string_with_slicing(input_string):
    output_string = input_string[::-1]
    return output_string

# zweite Funktion definieren
def string_umdrehen_rekursiv(eingabestring):
    if len(eingabestring) == 1:
        return eingabestring
    else:
        anfang = eingabestring[0]
        rest = string_umdrehen_rekursiv(eingabestring[1:])
        umgedrehtes_teilwort = rest + anfang    # umgedrehte Reihenfolge!
        return umgedrehtes_teilwort

# Befehl, um die Laufzeit der ersten Funktion zu messen
t1 = timeit.timeit('reverse_string_with_slicing("Hallo Welt")',
                   setup="from __main__ import reverse_string_with_slicing",
                   number=1000000)

print("Dauer für 1000000 Aufrufe von reverse_string_with_slicing(): {}".format(t1))

# Befehl, um die Laufzeit der zweiten Funktion zu messen
t2 = timeit.timeit('string_umdrehen_rekursiv("Hallo Welt")',
                   setup="from __main__ import string_umdrehen_rekursiv",
                   number=1000000)

print("Dauer für 1000000 Aufrufe von string_umdrehen_rekursiv(): {}".format(t2))
```

Die einzelnen Bestandteile des `timeit`-Aufrufs:

- Der Befehl, der gemessen werden soll; idealerweise genau ein Funktionsaufruf. So können wir die Dauer der einen gesamten Funktion mit der Dauer der anderen gesamten Funktion vergleichen.
- `setup`: Ein Import-Befehl der Form `from __main__ import <unsere Funktion>`.
- `number`: Anzahl der Ausführungen, die getestet werden sollen. Das ist notwendig, weil die Ausführungsdauer auch von **anderen Faktoren** abhängt als der Effizienz des Codes, sodass nicht immer das exakt gleiche Ergebnis zurückgegeben wird; wenn wir aber, wie im Beispiel oben, jede Funktion 1000000 mal ausführen, kann die Dauer dieser 1000000 Aufrufe aufschlussreich sein (vor allem, um zwei Funktionen zu vergleichen).

In meinem Test hat das Jupyter Notebook folgende Ausgabe erzeugt:

```
Dauer für 1000000 Aufrufe von reverse_string_with_slicing(): 0.18639454599997407
Dauer für 1000000 Aufrufe von string_umdrehen_rekursiv(): 2.8648401539999213
```

Wir sehen also, dass die Funktion mit Slicing **deutlich schneller** ist als die rekursive Funktion.

Im Einführungskurs ist es oft schwer, die Effizienz von Programmen einzuschätzen, weil wir uns mit vergleichsweise kleinen Aufgaben beschäftigen. Trotzdem ist es gut, wenn Sie ein Gefühl dafür entwickeln, welche von zwei Lösungen effizienter ist. Vor allem, wenn wir mit großen Datenmengen hantieren, ist Effizienz wünschenswert.

Warum ist jetzt die eine Lösung so viel langsamer als die andere? Überlegen wir mal, welche Operationen für Python aufwendig sind...

- Umwandlungen von einem Datentyp in einen anderen, allgemein: Erzeugen neuer Variablen
- Bedingungen prüfen (z.B. `len(input_string_as_list) > 0`)
- Operationen auf nicht-veränderlichen Datentypen (wie Strings)

Operationen auf nicht-veränderlichen Datentypen sind deshalb aufwendig, weil **faktisch jedesmal ein neues Objekt erzeugt wird**. Wenn wir in der rekursiven Variante nach und nach den String für die Rückgabe zusammenfügen, erzeugen wir jedesmal einen neuen String.

Übrigens kann es sein, dass die Funktionen unterschiedlich effizient für verschiedene Eingabewerte sind! Wie ist es, wenn wir **einen ganz kurzen String und einen ganz langen String** mit unseren beiden Funktionen umdrehen?

In [ ]:

```
import timeit    # Modul timeit importieren

# erste Funktion definieren
def reverse_string_with_slicing(input_string):
    output_string = input_string[::-1]
    return output_string

# zweite Funktion definieren
def string_umdrehen_rekursiv(eingabestring):
    if len(eingabestring) == 1:
        return eingabestring
    else:
        anfang = eingabestring[0]
        rest = string_umdrehen_rekursiv(eingabestring[1:])
        umgedrehtes_teilwort = rest + anfang    # umgedrehte Reihenfolge!
        return umgedrehtes_teilwort

############ KURZES EINGABEWORT: #################

input_word = "."
print("Wort: {}".format(input_word))

# Befehl, um die Laufzeit der ersten Funktion zu messen
t1 = timeit.timeit('reverse_string_with_slicing("{}")'.format(input_word),
                   setup="from __main__ import reverse_string_with_slicing",
                   number=1000000)

print("Dauer für 1000000 Aufrufe von reverse_string_with_slicing(): {}".format(t1))

# Befehl, um die Laufzeit der zweiten Funktion zu messen
t2 = timeit.timeit('string_umdrehen_rekursiv("{}")'.format(input_word),
                   setup="from __main__ import string_umdrehen_rekursiv",
                   number=1000000)

print("Dauer für 1000000 Aufrufe von string_umdrehen_rekursiv(): {}".format(t2))

print()
############ LANGES EINGABEWORT: #################

input_word = "?!?!?!?!!?!???!?!??!?!?!?!!?!???!?!??!?!?!?!!?!???!?!??!?!?!?!!?!???!?!?"
print("Wort: {}".format(input_word))

# Befehl, um die Laufzeit der ersten Funktion zu messen
t1 = timeit.timeit('reverse_string_with_slicing("{}")'.format(input_word),
                   setup="from __main__ import reverse_string_with_slicing",
                   number=1000000)

print("Dauer für 1000000 Aufrufe von reverse_string_with_slicing(): {}".format(t1))

# Befehl, um die Laufzeit der zweiten Funktion zu messen
t2 = timeit.timeit('string_umdrehen_rekursiv("{}")'.format(input_word),
                   setup="from __main__ import string_umdrehen_rekursiv",
                   number=1000000)

print("Dauer für 1000000 Aufrufe von string_umdrehen_rekursiv(): {}".format(t2))
```

So sah die Ausgabe in meinem Jupyter Notebook aus:

```
Wort: .
Dauer für 1000000 Aufrufe von reverse_string_with_slicing(): 0.16090821200032224
Dauer für 1000000 Aufrufe von string_umdrehen_rekursiv(): 0.19841613899961885

Wort: ?!?!?!?!!?!???!?!??!?!?!?!!?!???!?!??!?!?!?!!?!???!?!??!?!?!?!!?!???!?!?
Dauer für 1000000 Aufrufe von reverse_string_with_slicing(): 0.35317264500008605
Dauer für 1000000 Aufrufe von string_umdrehen_rekursiv(): 22.081919328999902
```

Aufgabe

1. Führen Sie `timeit` auf die folgenden Funktionen aus, die beide die Aufgabe haben, einer Liste alle Zahlen zwischen 0 und 1000 hinzuzufügen. Passen Ihre Beobachtungen zu dem, was Sie über veränderliche Datentypen wissen? Tipp: Wählen Sie zunächst nur 1000 Wiederholungen aus, dann ggf. 10000. Sie sollten bereits bei diesen Zahlen einen Unterschied bemerken.

In [ ]:

```python
import timeit

# erste Funktion definieren
def add_to_list_with_plus():
    my_list = []
    for i in range(1000):
        my_list = my_list + [i]
    return my_list

# zweite Funktion definieren
def add_to_list_with_append():
    my_list = []
    for i in range(1000):
        my_list.append(i)
    return my_list

# Ihre Befehle zum timen der Funktionen
```

Ein weiterer guter Grundsatz für effiziente Programmierung ist: **Vermeiden Sie Schleifen, wann immer es möglich ist.** Einige Möglichkeiten, wie Sie das erreichen können:

- Comprehensions verwenden, statt mit `for`-Schleifen eine Liste/Menge/ein Dictionary nach und nach mit Werten zu füllen.
- `join()` verwenden, statt einen String mit einer `for`-Schleife nach und nach zu füllen.
- Bevor Sie mehrere `for`-Schleifen ineinander verschachteln: Überlegen Sie, ob es eine andere Möglichkeit gibt, z.B. zwei aufeinanderfolgende `for`-Schleifen (nicht verschachtelt).
- In Ihren Schleifen sollten nur diejenigen Befehle stehen, die absolut notwendig Teil der Schleife sein müssen. Definitionen von z.B. regulären Ausdrücken sind fast immer außerhalb der Schleife besser aufgehoben.
- **`import` soll niemals in Code-Abschnitten stehen, die wiederholt ausgeführt werden!**
- Die Funktion `map()` lässt Sie eine beliebige Funktion auf alle Elemente einer Liste anwenden:

  Ohne `map`:

  ```
  oldlist = ["My", "Hovercraft", "is", "full", "of", "eels"]
  newlist = []
  for word in oldlist:
      newlist.append(word.upper())
  ```

  Mit `map`:

  ```
  oldlist = ["My", "Hovercraft", "is", "full", "of", "eels"]
  newlist = map(str.upper, oldlist)
  ```

Um einen String aus mehreren Teilstrings zusammenzusetzen, sollten Sie immer `format()` verwenden. Statt

```
output = "Das Wort " + "'und'" + " kam im Text " + str(5) + " mal vor"
```

schreiben Sie lieber:

```
output = "Das Wort {} kam im Text {} mal vor".format("und", 5)
```

Insgesamt lohnt es sich meistens, **eingebaute Methoden** zu verwenden, statt eigene Funktionen zu definieren.

Einen ausführlichen Vergleich verschiedener Lösungen für eine einzige Aufgabe finden Sie hier (https://www.python.org/doc/essays/list2str/).

Zusammenfassung

Sie haben heute gelernt,

- in VSCode zu prüfen, ob Ihr Code den Vorgaben von PEP8 entspricht
- mit dem Debugger von VSCode ein Programm Schritt für Schritt auszuführen, um fehlerhafte Codezeilen zu identifizieren
- zu messen, wie lange eine vorgegebene Anzahl Ausführungen einer Funktion brauchen, um zu vergleichen, welche Lösung effizienter ist
- Schleifen möglichst selten und möglichst kurz zu schreiben, um effizienten Code zu erzeugen
- `map()` zu verwenden, um eine Funktion auf alle Elemente einer Liste anzuwenden

Projekt: Gedichte generieren

In der letzten Übungssitzung des Jahres implementieren wir zusammen ein Programmierprojekt in Gruppenarbeit. Dazu werden Sie in drei Gruppen aufgeteilt. Am Ende der Sitzung fügen wir Ihre Teillösungen zusammen, um das Projekt abzuschließen.

Das Ziel

Wir generieren anhand deutschsprachiger Gedichte oder Balladen neue Texte. Die neuen Texte sehen fast aus wie die Originale, nur dass wir die meisten **Substantive durch andere Substantive ersetzen**, die grammatikalisch an die gleiche Stelle passen wie das jeweilige Originalwort.

```
"Wer wagt es, Kleistersmann oder Knapp,
Zu tauchen in diesen Grundsprachumfang?
Einen goldnen Versicherungsberater werf ich hinab,
Verschlungen schon hat ihn der schwarze Naturbegriff.
Wer mir den Versicherungsberater kann wieder zeigen,
Er mag ihn behalten, er ist sein eigen."
```

Das Ergebnis wird auf einer Webseite angezeigt. Jedesmal, wenn die Seite neu geladen wird, erzeugt das Programm ein neues Gedicht.

Damit wir in der Sitzung fertig werden, sind jeweils Teile des Codes schon vorgegeben, vor allem die Struktur der Funktionen und ihre Argumente. Erklärungen sind in den Dateien mit dem Stichwort `NOTE` markiert. Stellen, an denen Sie Code ergänzen sollen, sind mit `TODO` markiert. Jede Gruppe wird von eine_r Dozierenden oder unserem Tutor betreut, damit Sie jederzeit nach Tipps fragen können.

Die entstehende Webseite wird nach der Übungssitzung online gestellt. Falls alle einverstanden sind, wird auch der Code Ihrer Lösungen veröffentlicht.

Gruppe 1: Wortklassen sammeln

Gruppe 1 (mindestens 3 Personen) wird sich damit beschäftigen, mögliche **Ersetzungskandidaten für Substantive zu ermitteln**. Sie bekommen dafür eine Datei, in der für 480.131 deutsche Substantivformen morphologische Annotationen aufgelistet sind:

```
...
Adventskalender Masc_Nom_Sg Masc_Nom_Pl Masc_Gen_Pl Masc_Dat_Sg Masc_Acc_Sg Masc
_Acc_Pl
Adventskantate  Fem_Nom_Sg  Fem_Gen_Sg  Fem_Dat_Sg  Fem_Acc_Sg
Adventskerze    Fem_Nom_Sg  Fem_Gen_Sg  Fem_Dat_Sg  Fem_Acc_Sg
Adventskonzert  Neut_Nom_Sg Neut_Dat_Sg Neut_Acc_Sg
Adventskranz    Masc_Nom_Sg Masc_Dat_Sg Masc_Acc_Sg
...
```

Im Ausschnitt sehen Sie, dass die Wörter "Adventskantate" und "Adventskerze" die gleichen morphologischen Eigenschaften haben und deshalb untereinander ausgetauscht werden könnten.

Gruppe 1 wird solche Übereinstimmungen in der Datei automatisch identifizieren und ein Dictionary erstellen, in dem jeder Wortklasse (beschrieben durch eine Sequenz morphologischer Eigenschaften) alle bekannten Wortformen zugeordnet werden, die in diese Klasse gehören.

```
{   "Masc_Nom_Sg    Masc_Nom_Pl Masc_Gen_Pl Masc_Dat_Sg Masc_Acc_Sg Masc_Acc_P
l": {"Adventskalender"},
    "Fem_Nom_Sg Fem_Gen_Sg  Fem_Dat_Sg  Fem_Acc_Sg": {"Adventskantate", "Advents
kerze"},
    "Neut_Nom_Sg    Neut_Dat_Sg Neut_Acc_Sg": {"Adventskonzert"},
    "Masc_Nom_Sg    Masc_Dat_Sg Masc_Acc_Sg": {"Adventskranz"}
}
```

Benötigte Skills:

- `for`, `if`
- Dateien öffnen und verarbeiten
- Stringmethoden (z.B. `split`)
- Dictionarys anlegen
- Funktionen

Gruppe 2: Texte generieren

Gruppe 2 (mindestens 5 Personen) bekommt diese Substantivklassen von Gruppe 1 und hat die Aufgabe, Gedichttexte aus Dateien zu lesen und **neue Texte anhand der Originale und des Dictionarys zu erzeugen**.

Dazu gehört auch, dass zunächst Substantive im Originaltext gefunden werden müssen. Dann müssen diese Substantive im Dictionary gesucht und die möglichen Ersatzwörter identifiziert werden. Schließlich muss für jedes Substantiv im Gedicht ein Ersatzwort ausgewählt werden, und jedes Vorkommen des ursprünglichen Wortes muss durch das neue Wort ersetzt werden.

Benötigte Skills:

- `for`, `if`
- Reguläre Ausdrücke (`re.sub`, `re.search`)
- Dictionarys anlegen
- Dictionarys sortiert verarbeiten (`sorted`)
- Zufallsauswahl aus Mengen (`random.sample`) und Listen (`random.choice`)
- Funktionen

Gruppe 3: Webseite präsentieren

Gruppe 3 (mindestens 3 Personen) bekommt die generierten Gedichte von Gruppe 2 und hat die Aufgabe, diese **in Form einer Webseite zu präsentieren**. Die Webseite wird mit dem Pythonpaket `flask` bereitgestellt. Die Gedichte werden von Gruppe 2 im "Plain text"-Format erstellt, was in HTML nicht so gut aussieht (es fehlen zum Beispiel die Zeilenumbrüche). Gruppe 3 soll deshalb einige HTML-Tags an passenden Stellen einfügen, um das Gedicht etwas schöner darzustellen. Schließlich soll noch ein fertig vorbereiteter Disclaimer-Text aus einer Datei gelesen und unterhalb des Gedichts angezeigt werden.

Benötigte Skills:

- Reguläre Ausdrücke (`re.sub`)
- Stringoperationen (z.B. `replace`)
- Dateien lesen
- Funktionen