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

06-02: Debugging 

Inzwischen schreiben wir schon einigermaßen komplizierte Python-Programme. Leider bringen komplexe Aufgaben auch komplexe Probleme mit sich: Je umfangreicher ein Programm ist, umso schwieriger ist es, herauszufinden, warum es nicht so funktioniert, wie es soll. Heute geht es um Fehler, bei denen das Programm nicht abstürzt, die aber dazu führen, dass **der Output des Programms nicht den Erwartungen entspricht**.

Hier ein typisches Beispiel. Wir erhalten eine Liste von Wörtern und wollen ein Dictionary erstellen, in dem die Häufigkeit jedes Buchstaben innerhalb der Liste gespeichert ist. Dazu schreiben wir eine Funktion und eine Schleife, die diese Funktion für jedes Wort aufruft. Übrigens wollen wir hier nicht zwischen Groß- und Kleinschreibung unterscheiden.

In [ ]:
```
text = ["Nennt", "mich", "Ishmael"]
# gewünschtes Dictionary:
# {"n": 3, "e": 2, "t": 1,
#  "m": 2, "i": 2, "c": 1,
#  "h": 2, "s": 1, "a": 1,
#  "l": 1}

##########################################

# Funktion zum Zählen von Buchstaben
# innerhalb eines Worts
def count_letters_in_word(word):
    letters = {}
    # durch alle Buchstaben iterieren
    for l in word:
        # prüfen, ob das aktuelle Zeichen
        # schon mal vorkam
        if l.lower() not in letters:
            # falls nicht: Wert auf 0 setzen
            letters[l.lower()] = 0
        # Wert um 1 erhöhen
        letters[l.lower()] += 1
    return letters

##########################################

# durch alle Wörter iterieren
for current_word in text:
    # Funktion auf jedes Wort anwenden
    letters = count_letters_in_word(current_word)

##########################################

# entspricht das Dictionary dem
# gewünschten Wert?
print(letters)
```

Das Dictionary, das wir erzeugen, hat zu wenige Einträge und die Zahlen sind auch zu klein. Es gibt also offensichtlich ein Problem in unserem Code, obwohl das Programm nicht abgestürzt ist. Wir sollten uns das näher ansehen.

Bisher haben wir oft `print()` -Aufrufe im Code eingefügt, um zu sehen, welchen Wert bestimmte Variablen zu einem bestimmten Zeitpunkt haben. Heute verwenden wir eine schönere und informativere Methode: Den eingebauten Debugger, den VSCode uns zur Verfügung stellt.

Dazu speichern wir zunächst den Code in einer eigenen Pythondatei, die wir dann einzeln öffnen.

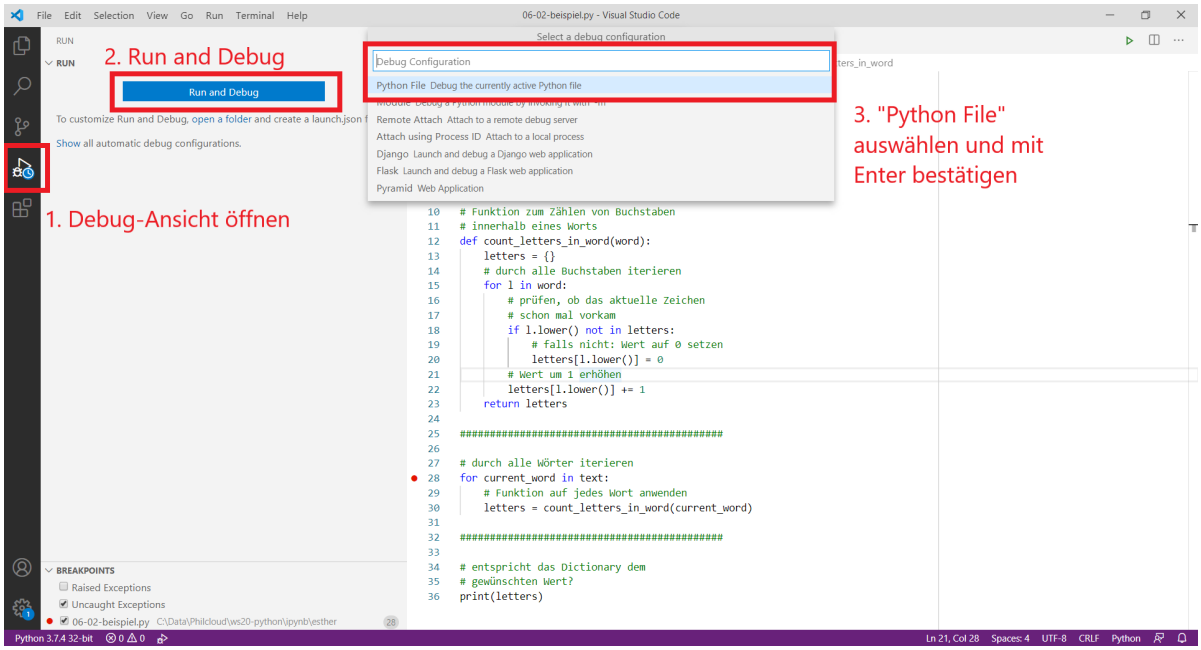
Breakpoints setzen und Debugger starten

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. Ein wichtiges Konzept hierfür sind **Breakpoints**. Das sind Markierungen, die wir im Code setzen können und an denen der Interpreter pausiert, damit wir uns den Zustand des Programms ansehen können.

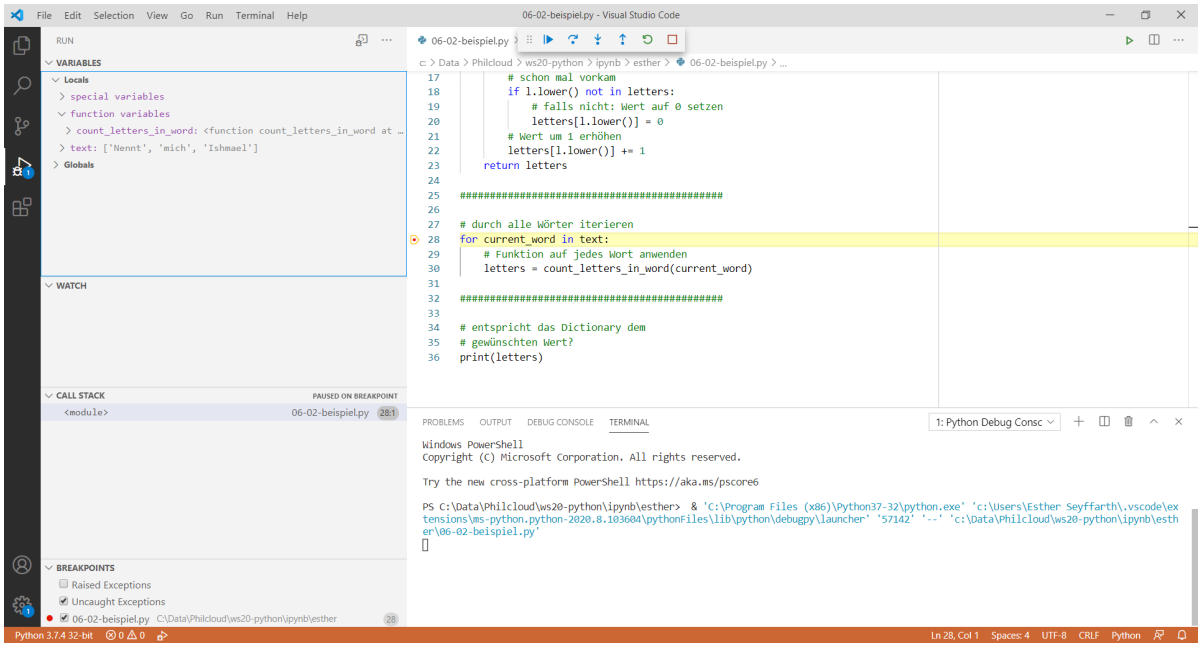
Breakpoints können wir jederzeit setzen, indem wir links neben die Zeilennummer im Editor klicken, sodass ein roter Punkt erscheint. Solange wir keine Idee haben, was das Problem in unserem Programm ist, können wir den Breakpoint in die erste Zeile setzen, die der Interpreter ausführen wird. Hier ist das Zeile 28: Der Kopf der `for` -Schleife. So sieht der Breakpoint aus:

```
27    # durch alle Wörter iterieren
28    for current_word in text:
29        # Funktion auf jedes Wort anwenden
30        letters = count_letters_in_word(current_word)
```

Um die Debug-Ansicht anzuzeigen, können wir links im Menü auf das Debugger-Symbol klicken oder die Command Palette verwenden (Kommando: `>run and debug`). Nachdem wir auf "Run and Debug" klicken, müssen wir noch auswählen, dass die aktuelle Pythondatei ausgeführt und debuggt werden soll:



Der Interpreter wird jetzt das Programm ausführen, und dabei hält er an jedem Breakpoint einmal an. Wir sehen dann verschiedene Informationen über das Programm:



Variables (links, oben) listet alle bekannten Variablen auf. Dabei ist unsere Variable `text` direkt in der Liste zu sehen, während unsere Funktion `count_letters_in_word()` unter dem Eintrag `function variables` untergeordnet ist.

Watch (links, Mitte) lässt uns bestimmte Variablen im Blick behalten. Wir können beliebige Variablennamen hier eingeben und sehen dann auf den ersten Blick, welcher Wert jeweils in der Variable gespeichert ist.

Der **Call Stack** (links, Mitte) 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 sagen dem Interpreter, an welchen Stellen die Ausführung des Programms pausiert werden soll. Wir können hier auch einzelne Breakpoints deaktivieren, wenn wir sie später doch nicht mehr brauchen. Oder wir setzen während des Debuggens neue Breakpoints, die dann sofort in dieser Liste auftauchen.

Der **Ausgabebereich** (unten, Mitte/rechts) sieht ähnlich aus wie der Ausgabebereich beim normalen Ausführen von Pythondateien. Hier ist noch nichts interessantes zu sehen, weil das Programm pausiert wurde, bevor eine Ausgabe mit `print()` passiert ist.

Stepping over!

Unser erster Breakpoint zeigt uns, dass der Interpreter die Namen `text` und `count_letters_in_word()` kennt - aber richtig viel ist bisher noch nicht passiert.

Wir können in unterschiedlich großen Schritten weiter durch das Programm gehen. Dazu benutzen wir die Knöpfe **am oberen Rand**.

Der **erste Knopf links** ("Continue", Play-Symbol) spult bis zum nächsten Breakpoint vor. Das ist aktuell noch nicht so sinnvoll, weil wir erst einen Breakpoint gesetzt haben.

Der **zweite Knopf** ("Step Over", Pfeil über einem Punkt) lässt uns zum Anfang der nächsten ausführbaren Zeile im Code springen. Dabei wird die aktuelle Zeile vollständig ausgeführt und nicht pausiert, egal wie komplex sie ist (anders als der nächste Knopf).

Der **dritte Knopf** ("Step Into", Pfeil nach unten) springt **in die Methode, die als nächstes ausgeführt wird**. Wenn in der aktuellen Zeile zum Beispiel eine Funktion aufgerufen wird, springen wir mit diesem Pfeil in die erste Zeile der Funktionsdefinition. Der **zweite Knopf** würde stattdessen die Funktion im Hintergrund durchlaufen lassen und direkt zur folgenden Zeile springen.

Der **vierte Knopf** ("Step Out", Pfeil nach oben) lässt uns aus der aktuellen Funktion herausspringen, er ist also eine Art Abkürzung, wenn wir wieder zurück zur übergeordneten Ebene springen wollen. Wir landen bei der nächsten Zeile nach der Zeile, die die eben noch aktuelle Funktion **aufgerufen** hat.

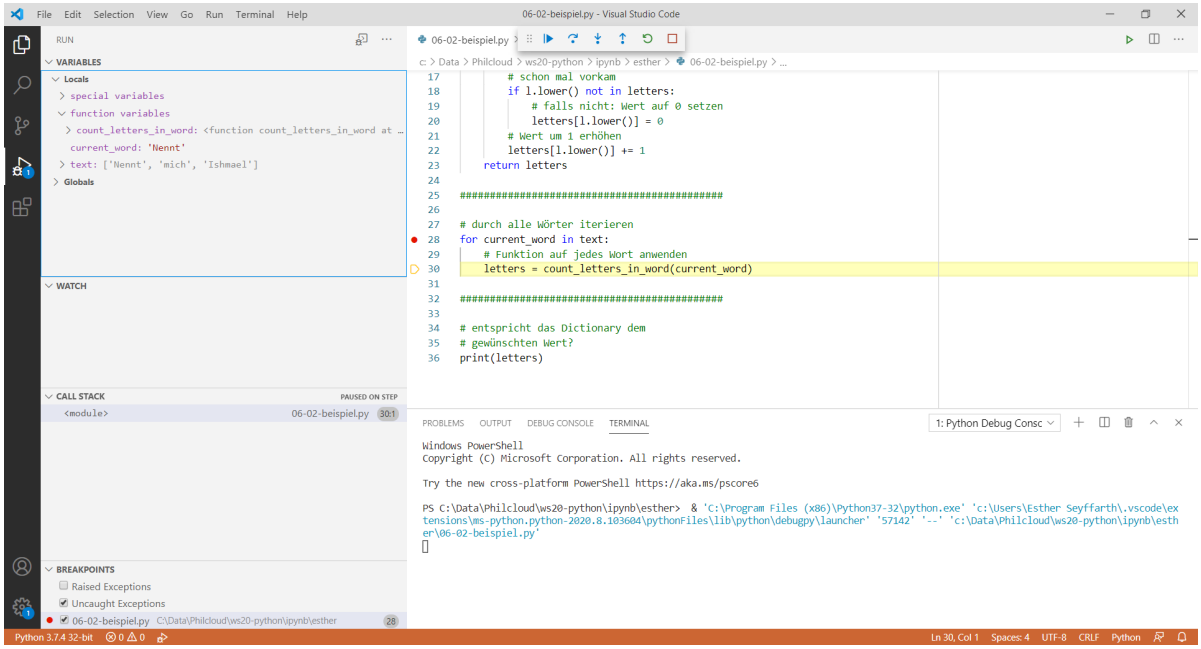
Der **fünfte und sechste Knopf** ("Restart", "Stop") sind zum Neustarten oder Abbrechen des Debuggers zuständig.

Zurück zu unserem Beispiel! Der Debugger befindet sich gerade im Kopf der `for`-Schleife. Machen wir einfach mal einen Schritt in die nächste Zeile, indem wir den "Step over"-Knopf (2. von links) benutzen.

Folgende Veränderungen beobachten wir in der Debugger-Ansicht:

- Wir stehen nicht mehr bei Zeile 28, sondern jetzt bei Zeile 30 (erkennbar im Bereich "Call Stack").

- Offensichtlich wurde die Kommentarzeile (29) übersprungen. Sie spielt für den Interpreter keine Rolle und ist nur für uns da.
- Im Bereich "Variables" ist eine neue Variable erschienen, nämlich `current_word`. Das ist die Laufvariable der `for`-Schleife, in der wir uns befinden.
- Im Code-Bereich ist die gelbe Markierung in Zeile 30 gewandert (vorher war Zeile 28 markiert).

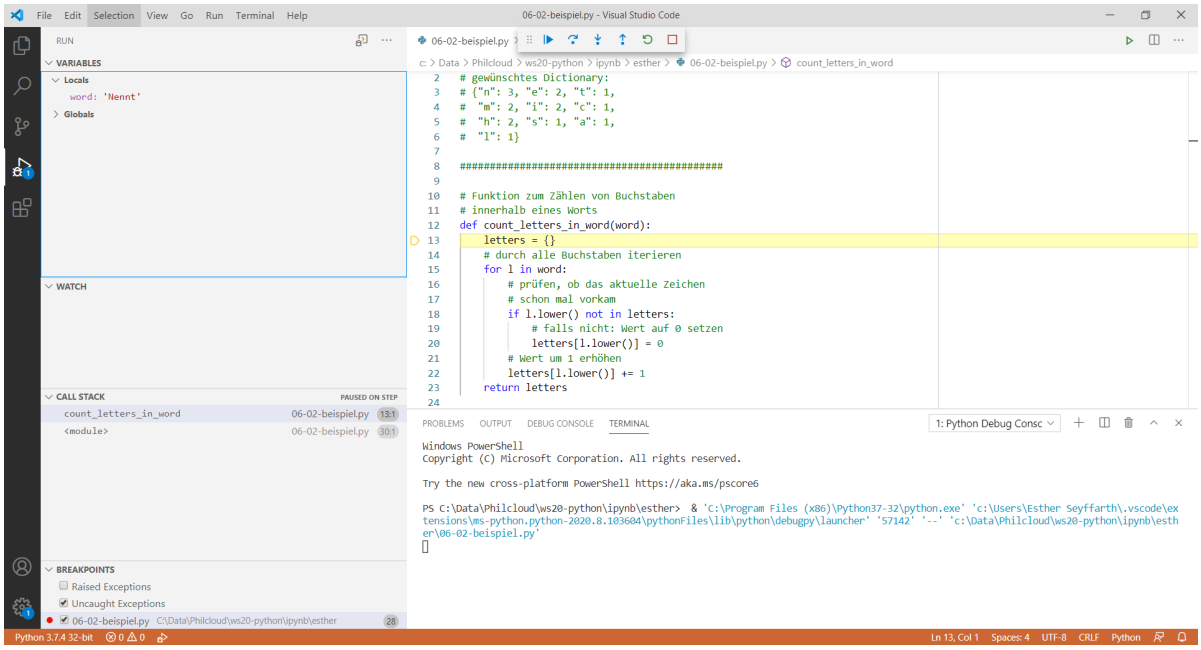


Stepping into!

Zu dem Zeitpunkt, an dem der Debugger in Zeile 30 anhält, wurde die Funktion `count_letters_in_word()` noch nicht ausgeführt. Wir interessieren uns dafür, was innerhalb dieser Funktion passiert. Also wählen wir als nächstes "Step Into" (3. Knopf von links). Wieder verändert sich einiges in der Debugger-Ansicht. Besonders auffällig ist, dass alle bisher sichtbaren Einträge im Bereich "Variables" scheinbar verschwunden sind und jetzt nur noch die Variable `word` bekannt ist. Tatsächlich sind die alten Variablen immer noch da, aber wir haben einen neuen **lokalen Namensraum** betreten, als wir in die Funktion gesprungen sind. (Das Konzept von Namensräumen wird in Thema 05-04 diskutiert.)

Unter "Globals" sind unsere alten Variablen weiterhin zu finden. Die neue Variable `word` gehört zum lokalen Namensraum der Funktion, in der wir uns jetzt befinden.

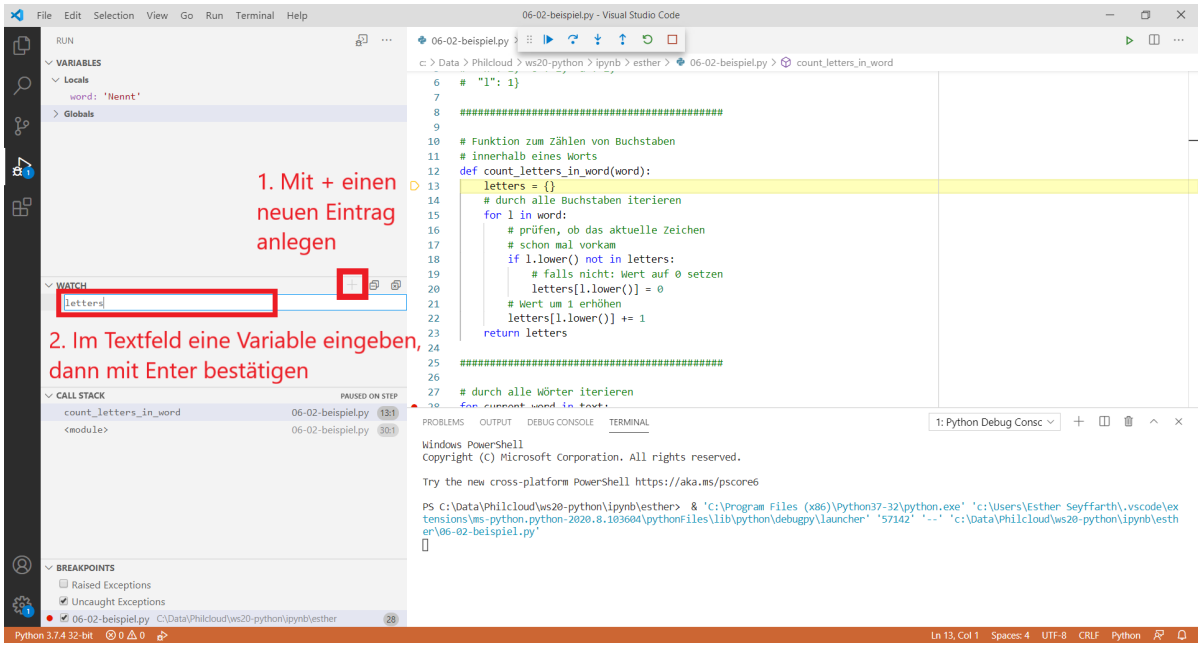
Im Bereich "Call Stack" ist auch zu sehen, dass wir die Funktion `count_letters_in_word` betreten haben. Der Call Stack sagt uns, wo wir landen würden, wenn wir als nächstes "Step Out" (4. Knopf von links) auswählen würden: Wir kämen in die nächste auszuführende Zeile nach Zeile 30, die ja den Funktionsaufruf enthält, in dem wir uns gerade befinden.



Watching!

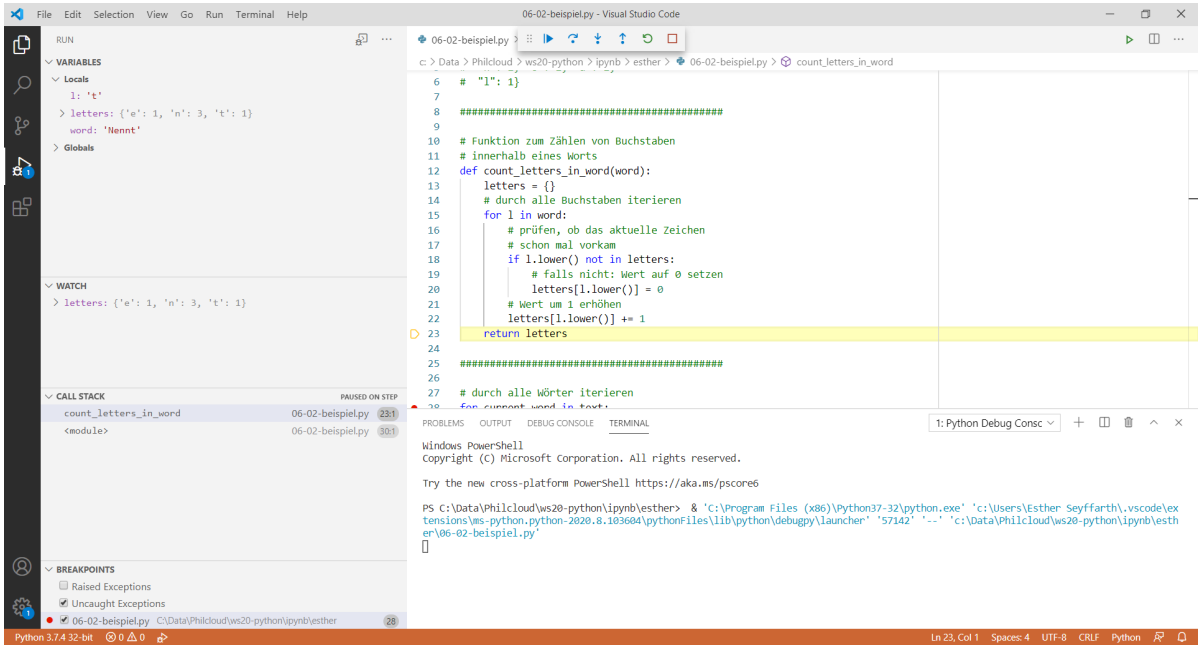
Um ein Gefühl dafür zu haben, was innerhalb der Funktion passiert, können wir mehrmals den "Step Over"-Knopf benutzen. Damit es übersichtlicher wird, können wir eine Watch für die Variablen, die uns interessieren, anlegen. Es wäre zum Beispiel gut, wenn wir den Wert der Variable `letters` im Blick behalten.

Im Bereich "Watch" können wir einen neuen Eintrag anlegen, indem wir auf das "+"-Symbol in der oberen rechten Ecke dieses Bereichs klicken. Wir geben den Namen der Variable `letters` ein und bestätigen den Eintrag mit Enter.

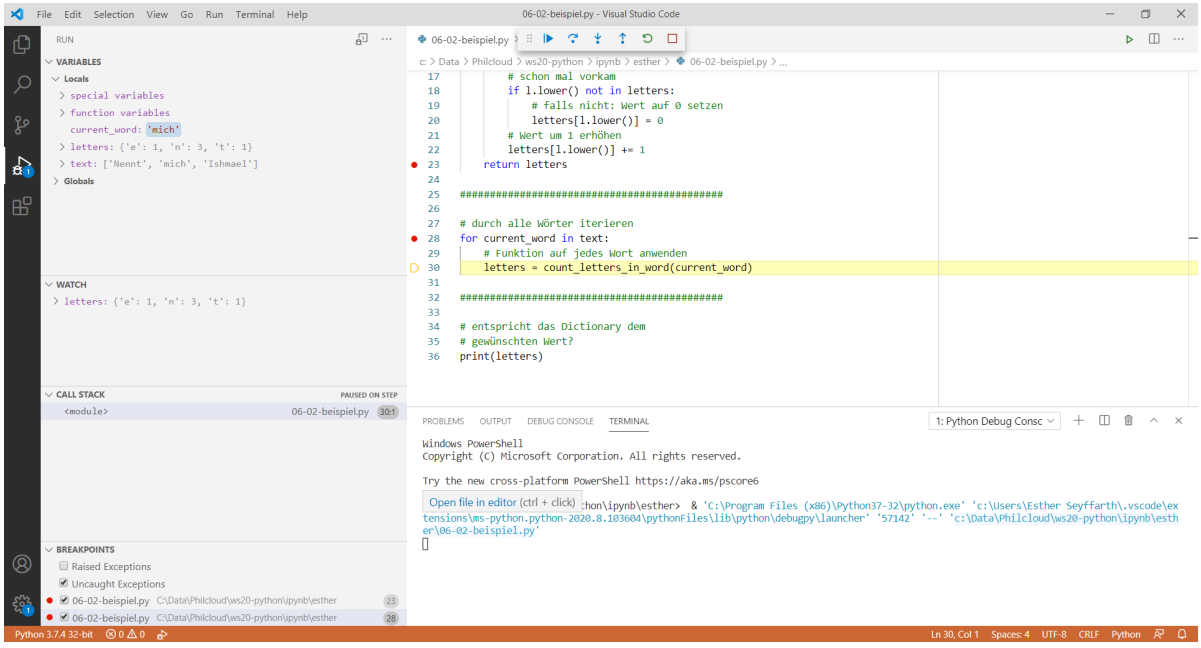


Hier im Beispiel haben wir die Watch definiert, bevor Zeile 13 ausgeführt wurde. Der Interpreter weiß also bisher noch nichts von der Variable. Sobald wir "Step Over" benutzen, wird ein Wert für die Variable angezeigt. Wir können so lange "Step Over" benutzen, bis wir am Ende der Funktionsdefinition ankommen. Bei `for` -Schleifen springt der Debugger immer wieder in den Schleifenkopf, bis die Sequenz, durch die wir iterieren, vollständig bearbeitet wurde.

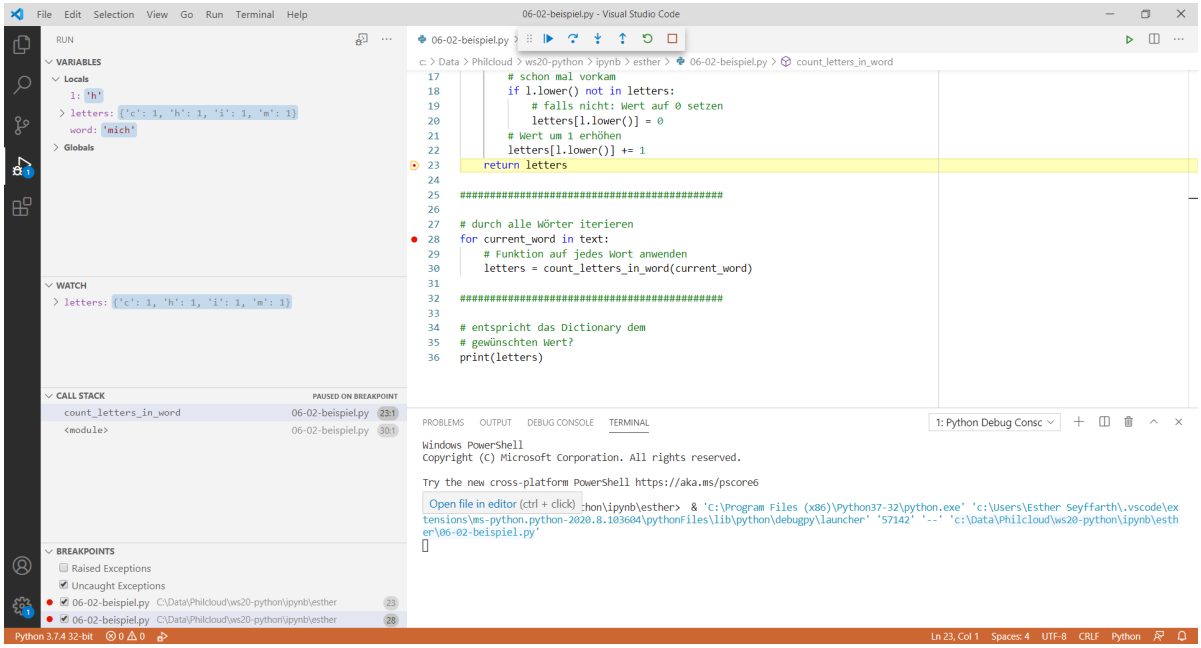
Beim Steppen beobachten wir den Wert von `letters` und sehen, wie das Dictionary sich nach und nach mit Werten füllt. Bisher sieht alles gut aus.



In der Funktion scheint es kein Problem zu geben. Um die nächsten Runden (die nächsten Aufrufe der Funktion) etwas abzukürzen, können wir einen Breakpoint in die `return` -Zeile setzen. Dann können wir den "Continue"-Knopf (1. von links) benutzen, um zum jeweils nächsten Breakpoint durchzulaufen. Wenn wir diesen Knopf direkt als nächstes nutzen, springen wir in Zeile 28, zu unserem ersten Breakpoint. Das ist der Beginn des zweiten Schleifendurchlaufs. Die Variable `current_word` hat immer noch ihren ersten Wert, "Nennt", aber mit einem Schritt in die nächste Zeile sehen wir, dass das nächste Wort aus der Liste jetzt bearbeitet wird.



Auch unsere Watch, die Variable `letters` mit dem Dictionary, sieht aktuell noch gut aus. Wir erwarten, dass als nächstes die Buchstaben im jetzt aktuellen Wort "mich" gezählt werden und dass die Zahlen im Dictionary aktualisiert bzw. ergänzt werden. Ob das passiert, prüfen wir, indem wir mit "Continue" zum Breakpoint in Zeile 23 weiterlaufen. Mal schauen...



Oh nein! Unsere bisherigen Dictionary-Inhalte sind verlorengegangen. Unser Programm zählt also die Buchstaben in jedem Wort, vergisst dabei aber immer die Buchstaben des jeweils letzten Wortes. Deshalb hat das Programm beim ersten Ausführen so ein kleines und unvollständiges Dictionary erzeugt.

Fixing it!

Durch die Pausen beim Ausführen und die Informationen zu den Variablenwerten konnten wir sehen, wo im Programm Daten verlorengegangen sind. Jetzt wissen wir mehr und können anfangen, den Code zu reparieren.

In diesem Fall gibt es mindestens zwei Möglichkeiten, wie wir den Fehler beheben könnten. Wie muss der Code verändert werden, damit das gewünschte Dictionary herauskommt?

Die einfachste Möglichkeit wäre es, das Dictionary außerhalb der Funktionsdefinition zu initialisieren und es als zusätzliches Argument zu übergeben. Dann gibt es nur ein Dictionary und es wird bei jedem Funktionsaufruf etwas mehr gefüllt.

Zusammenfassung

- Manchmal enthalten Programme Fehler, stürzen aber nicht ab. Solche Fehler sind trotzdem ärgerlich und sollten gefunden und behoben werden.
- Mit dem Debugger können wir Programme Schritt für Schritt ausführen und uns jederzeit die Werte von Variablen ansehen.
- Der Debugger wird an jedem Breakpoint anhalten. Wir können so viele Breakpoints setzen, wie wir wollen. Es macht Sinn, Breakpoints an besonders interessante oder zweifelhafte Stellen im Code zu setzen und so das Problem "einzukreisen".
- Wir können verschieden große Schritte durch das Programm machen, je nachdem, ob wir "Continue", "Step Over", "Step Into" oder "Step Out" auswählen.
- Wir können beliebig viele Variablenwerte genau beobachten, indem wir eine Watch für sie anlegen (eine Watch pro Variable oder Ausdruck).

- Nachdem der Debugger uns hilft, das Verhalten des Programms zu verstehen, können wir die identifizierten Probleme beheben.

Weitere Themen dieser Woche

- 06-01: Fehlertypen, Fehlermeldungen lesen und verstehen