

Cyber Security

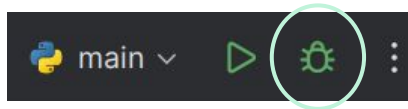
Grundlagen der Programmierung

Debugging von Projekten



Was ist der Debugger?

- Werkzeug zum verbessern/korrigieren von Code
- Debugger erlaubt Schritt für Schritt ausführung.
- Programm wird im Debugger bis zum Breakpoint/Fehler ausgeführt.
- Erlaubt das Untersuchen von Variablen
- Erlaubt das schrittweise nachvollziehen des Codes



Warum Debuggen?

- Erwartungen die man an den Code stellt werden oft nicht eingehalten.
- Einsicht nur durch draufstarren oft nicht hilfreich.
- Tieferer Einblick in den Programm ablauf nötig.
- Verifiziert das Code an der stelle auch das tut was man erwartet.



Basics

Im Allgemeinen geht es um das Finden von Fehler.

- Simpleste Form ist `print()`-Statements einbauen um:
 - Zustand mit Erwartungen abzugleichen
 - Variablen zu überwachen
 - Loop Counts zu betrachten
- Besser:
 - Programm anhalten und alle Variablen sehen
 - Manuell Ablauf kontrollieren
 - Welche Funktionen ausgeführt wurden angezeigt bekommen.
- Technisch ist das schwieriger, aber wird uns von der Entwicklungsumgebung abgenommen



Breakpoints

- Breakpoints erlauben das Programm an bestimmter Stelle anzuhalten
- Breakpoints können im eigenen oder in Bibliothekscode (importierter Code) eingesetzt werden.
- Meistens reicht ein Breakpoint direkt vor dem Fehler.
- Mit **Strg+F8** setzt man einen Breakpoint in die aktive/ausgewählte Zeile.

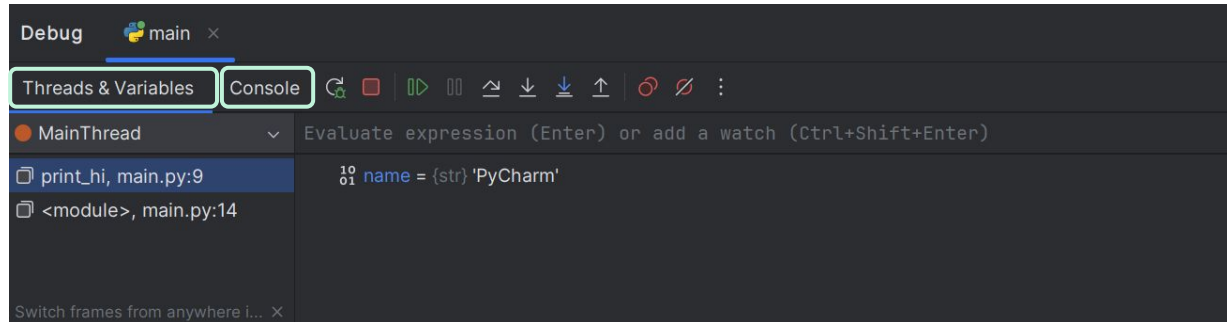
Jetzt kann der Debugger gestartet werden, und sobald er bei dem Breakpoint ankommt wird das Programm pausiert, und Variablen aus dem lokalen Scope werden angezeigt.



Interface

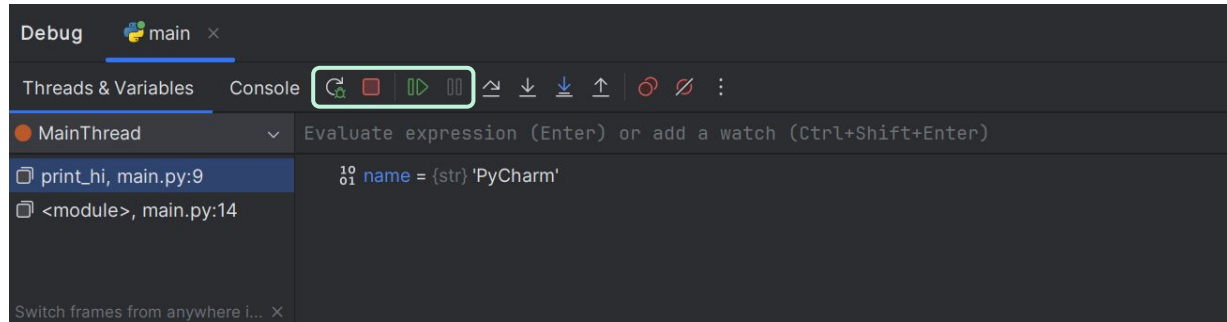
Mit dem Erreichen eines Breakpoints, wird das Debug Menü geöffnet. Hier sehen wir die relevanten Variablen und haben Zugriff auf weitere wichtige Funktionen:

- **Console:** Betrachten des Outputs des Programms
- **Thread/Modul Auswahl:** Hier kann gewählt werden Variablen aus welchem Modul wir betrachten wollen



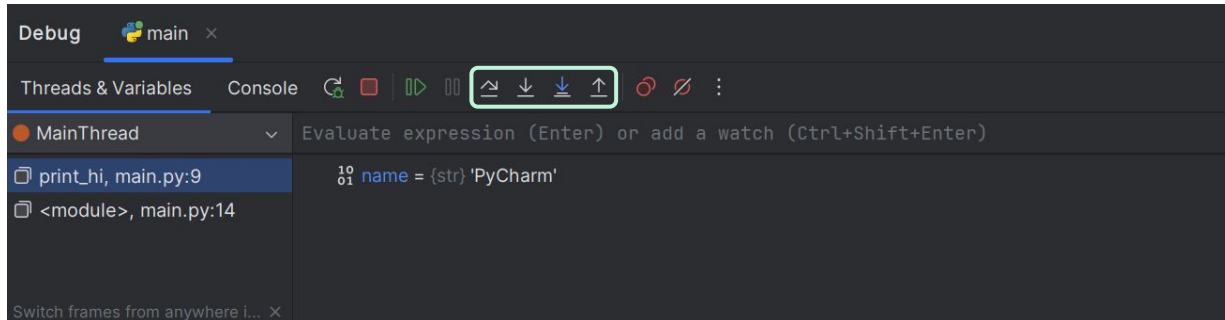
Interface

- Debugger **neustarten**
- Debugger **stoppen**
- **Weiterlaufen** zum nächsten Breakpoint
- Aktiv **pausieren**(benutzt man effektiv nicht)



Interface

- **Step Over:** Läuft bis zur nächsten Zeile des aktuellen Skripts
- **Step Into:** Folgt Code in aufgerufene Funktionen
- **Step Into My Code:** Wie step Into, aber folgt nur in eigenen Code (nicht in Bibliotheken)
- **Step Out:** Springt auf das nächst höhere Aufruf Level, also z.B. aus einer Funktion an die Stelle wo sie aufgerufen wurde.



Watches

- Definiert Statements die bei jeder Pausierung evaluiert werden
- Erlaubt das überwachen von komplexeren Datenstrukturen ohne die jedes mal anklicken zu müssen
- Kann helfen die Veränderungen einer Variable im Gesamtprogramm zu verstehen



Conditional Breakpoints

Spezifische Kondition die wahr sein muss.

- Beispiel:
 - Fehler tritt in while-Loop an bestimmter Stelle auf
 - Man setzt den Breakpoint in die Schleife
 - Man setzt die Kondition auf die Iteration vor dem Fehler
 - Betrachtet von dort den Zustand
- Erlaubt deutlich schnelleres Debuggen



Beispiel Code mit Fehler

Wir sind uns nicht ganz sicher wo der Fehler liegt, aber wissen dass das Ergebnis falsch ist.

```
1  from math import pi
2
3  pi, radius, r = 3, 1, 4
4  list_of_circle_radii = [1, 4, 8, 9, 22]
5
6  usage
7  def calculate_circumference(radius):
8      return 2 * pi * r
9
10 usage
11 def calculate_area(radius):
12     return pi * pi * radius
13
14 usage
15 def calculate_circle_info(input_radii):
16     for radius in input_radii:
17         print("Kreis mit Radius ", radius)
18         print("Umfang ", calculate_circumference(radius))
19         print("Fläche ", calculate_area(radius))
20
21 if __name__ == "__main__":
22     calculate_circle_info(list_of_circle_radii)
```



Beispiel Code mit Fehler

Gehen wir davon aus das Problem nicht direkt lösen zu können
Breakpoint setzen. **Aber wo?**

- Erste Zeile geht immer
- Idealerweise vor dem Fehler
- For-Loop eignet sich hier



Beispiel Code mit Fehler

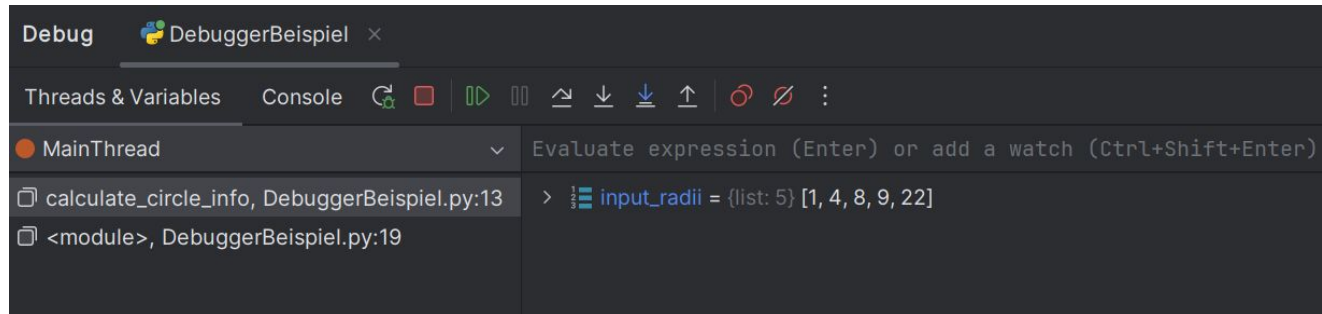
```
1  from math import pi
2
3  pi, radius, r = 3, 1, 4
4  list_of_circle_radii = [1, 4, 8, 9, 22]
5
6  1 usage
7  def calculate_circumference(radius):
8      return 2 * pi * r
9
10  1 usage
11  def calculate_area(radius):
12      return pi * pi * radius
13
14  1 usage
15  def calculate_circle_info(input_radii):
16      for radius in input_radii:
17          print("Kreis mit Radius ", radius)
18          print("Umfang ", calculate_circumference(radius))
19          print("Fläche ", calculate_area(radius))
20
21  18 ► if __name__ == "__main__":
22      calculate_circle_info(list_of_circle_radii)
```



Beispiel Code mit Fehler

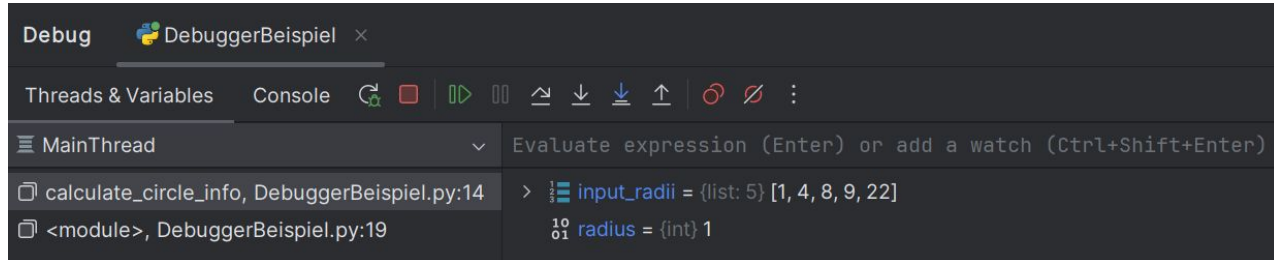
Debugger starten mit dem Bug-Icon oben rechts.
Programm sollte zum Breakpoint laufen und das Problem anhalten.
Wir sehen

- unsere aktive Funktion links
- unsere aktiven Variablen rechts



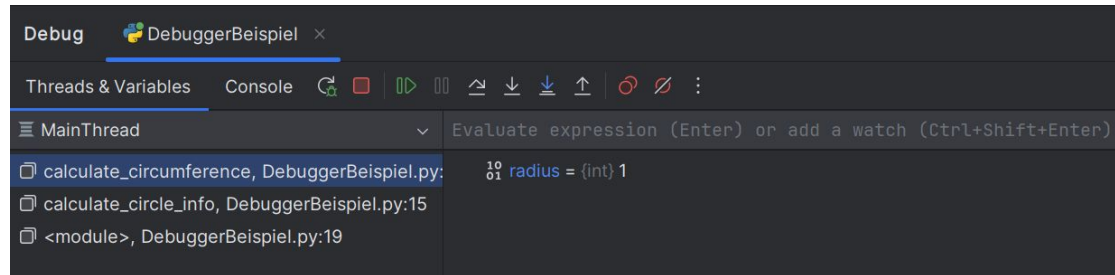
Beispiel Code mit Fehler

- Mit Step Over (F8) in die nächste Zeile gehen
- Lokale Variablen ändern sich. Radius ist jetzt 1
 - Das erste Element der Liste
- Erneut Step Over, da wir hier nur den Radius printen



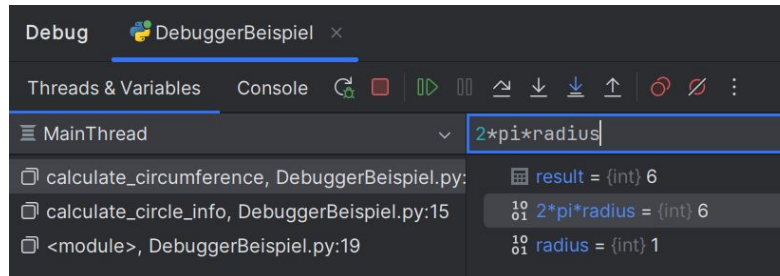
Beispiel Code mit Fehler

- Wir sind in einer Zeile die mit dem Fehler zu tun hat(Kreisumfang wird falsch berechnet)
- Step Into (F7) um der Funktion zu folgen
- Lokale Variablen sind jetzt nur die, die in der Funktion definiert werden oder ihr als Argument mitgegeben werden (hier nur radius = 1)
- Aktive Funktion ist jetzt calculate_circumference



Beispiel Code mit Fehler

- Wir sind in einer Zeile die mit dem Fehler zu tun hat (Kreisumfang wird falsch berechnet)
- Wir können jetzt eine Watch anlegen um den Wert zu überprüfen
- **$2 \cdot \pi \cdot \text{radius}$** in das Expression Feld eintragen
- Ctrl+Shift+Enter um aus der Expression eine Watch zu machen
- Ergebnis schon Falsch! (Kreis mit Radius 1 hat eine Umfang von $2\pi = 6.283...$)



Beispiel Code mit Fehler


- Weitere Expression testen indem wir den Code aus unserem Programm kopieren
- Auch hier ist das Ergebnis falsch. Beim Vergleich fällt auf:
 - Funktion erhält **radius**, nutzt aber **r**!

```
1 usage
6  def calculate_circumference(radius):  radius: 1
7  return 2 * pi * r
8
```



Beispiel Code mit Fehler

- Zur Definition von `r!`
 - **Ctrl** drücken und auf das **r** klicken
- Gepackte Struktur aus dem Testen scheint hier genutzt zu werden
- Erster Gedanke: **Kann das weg?**
 - Testweise entfernen
 - Wirft Fehler in unserer Funktion, also dort zu radius ändern
 - Erstes Problem gelöst
- Zweiter Gedanke wäre gewesen: **Was macht das?**

```
2        
3      pi, radius, r = 3, 1, 4  
4      list_of_circle_radii = [1, 4, 8, 9, 22]
```



Beispiel Code mit Fehler

- Bis auf die Fläche stimmt jetzt alles
- Also Breakpoint in die Flächen Funktion eingesetzt
- Scharfes Draufschauen
- Mit Formel für die Fläche vergleichen ($\pi * \text{radius}^2$)
- **Fehler:** Formel ist falsch

```
9      def calculate_area(radius):  
10         return pi * pi * radius  
11
```



Beispiel Code mit Fehler

Und damit ist das Programm fertig gedebuggt!

```
1  from math import pi
2
3  # radius, r = 3, 1, 4
4  list_of_circle_radii = [1, 4, 8, 9, 22]
5
6  1 usage
7  def calculate_circumference(radius):
8      return 2 * pi * radius
9
10 1 usage
11  def calculate_area(radius):
12      return pi * radius * radius
13
14 1 usage
15  def calculate_circle_info(input_radii):
16      for radius in input_radii:
17          print("Kreis mit Radius ", radius)
18          print("Umfang ", calculate_circumference(radius))
19          print("Fläche ", calculate_area(radius))
20
21  if __name__ == "__main__":
22      calculate_circle_info(list_of_circle_radii)
```



Refactor - Warum?

- Automatische Erkennung zusammenhängender Variablen
- Erlaubt das Updaten von Variablennamen im ganzen Projekt
- Nicht nur Search+Replace, sondern im richtigen Scope
- Warnung zu potenziellen Fehlern



Refactor - Wie?

- Rechtsklick auf die Variable
- Refactor: Rename... auswählen
- Neuen Namen angeben
- Validierung/Tests anschauen
- Refactor durchführen

- **Alternative:** Wort auswählen und **Shift+F6** drücken





CloudCommand