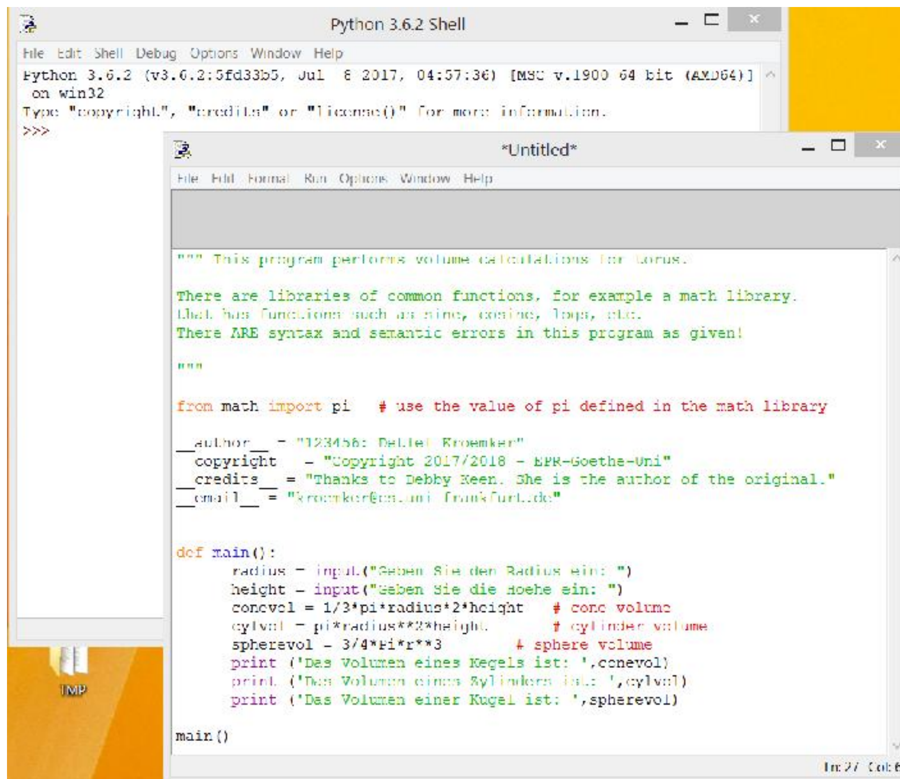
# Ein IDLE Debugger Tutorial[1] (Version 1.0, 24.07.2017)

*Im englischen Original von Debby Keen (übersetzt, modernisiert und ergänzt von Detlef Krömker)*

IDLE hat einen eingebauten Debugger. Dieser kann sehr hilfreich sein, um schrittweise durch ein Programm zu gehen und zu beobachten, wie die Variablen ihre Werte verändern.

Starten Sie IDLE und öffnen Sie das Programm volume[2] in einem Editor-Fenster. (Achtung: Dieses File ist in der Moodle-Plattform unter dem angegebenen Link ‚volume' gespeichert.) Ihre Arbeit sollte dann so aussehen:

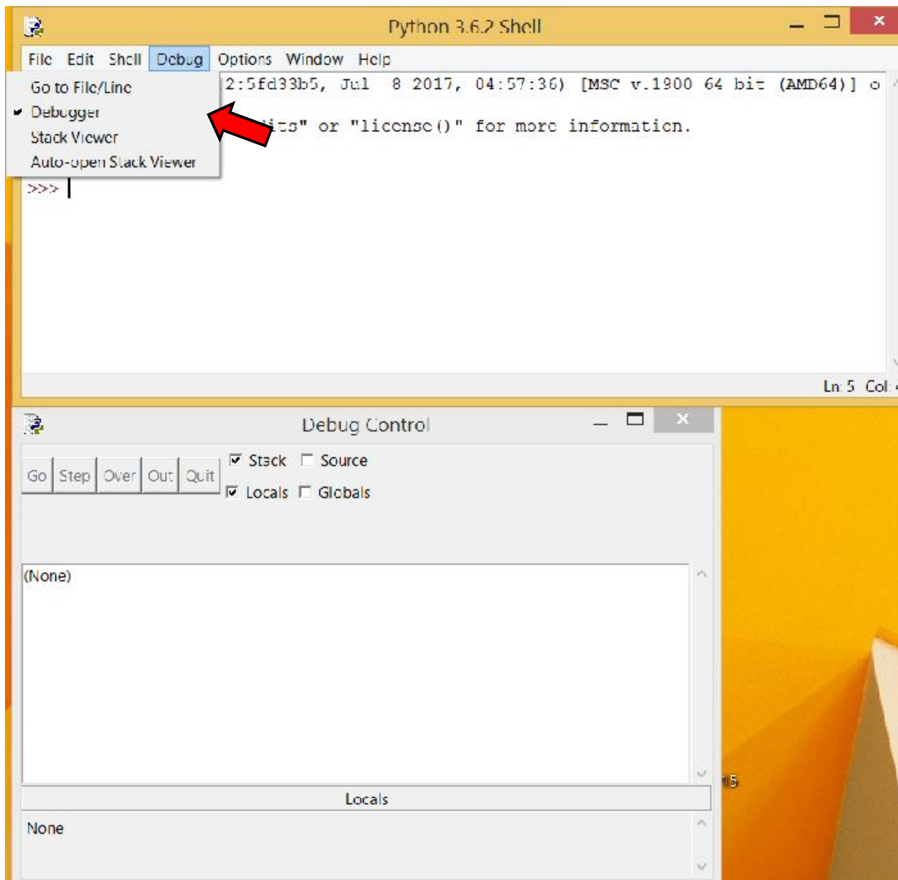


Es ist eine gute Idee, das Programm jetzt zu speichern, z.B. als volume.py.

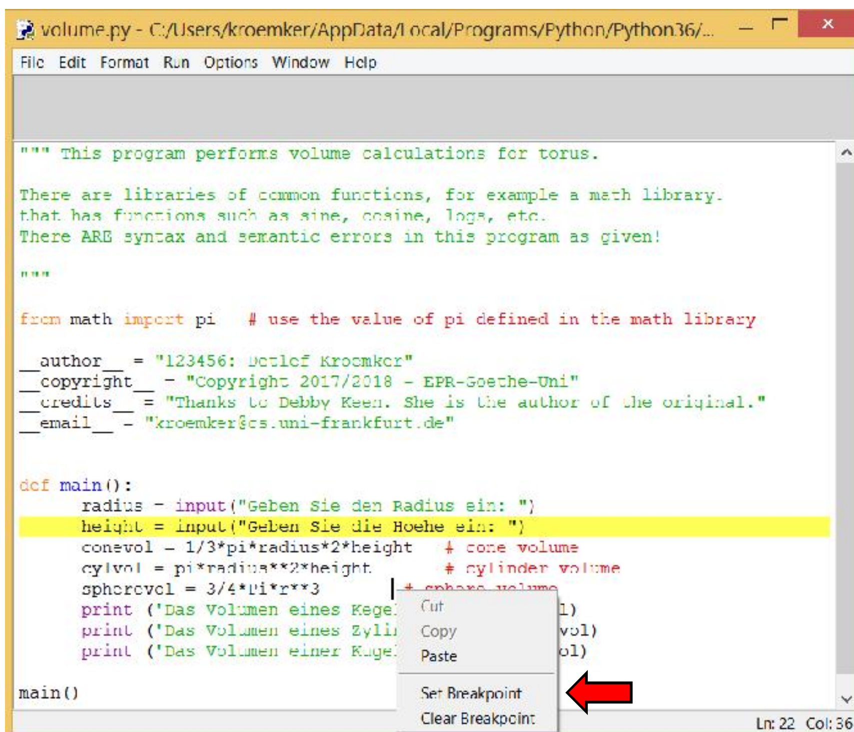
[1] *Dieser Text beruht auf einer Website von Dr. Debby Keen, University of Kentucky, https://www.cs.uky.edu/~keen/help/debug-tutorial/debug.html* (abgerufen am 22.07.2017). Das Material wurde übersetzt und gründlich überarbeitet.

[2] https://moodle.studiumdigitale.uni-frankfurt.de/moodle2/mod/resource/view.php?id=16520 (Sie müssen eingelogged sein um den

In dem zugehörigen Shell-Fenster von IDLE klicken Sie bitte den Menuepunkt **Debug** und dann im Drop-Down Menue **Debugger**. Sie sollten jetzt Fenster ‚Debug Control' sehen. Beachten Sie auch das [DEBUG ON] im zugehörigen Shell-Fenster.




Sehr nützliche Funktionen des Debuggers sind Breakpoints, die man im Source Code setzen kann. Breakpoints sind „Marken" in Ihrem Code, die dem Debugger anzeigen, dass er mit normaler Geschwindigkeit bis zu diesem Punkt laufen und dann anhalten soll und dann dem Benutzer die Kontrolle (über das weitere Vorgehen) übergibt.



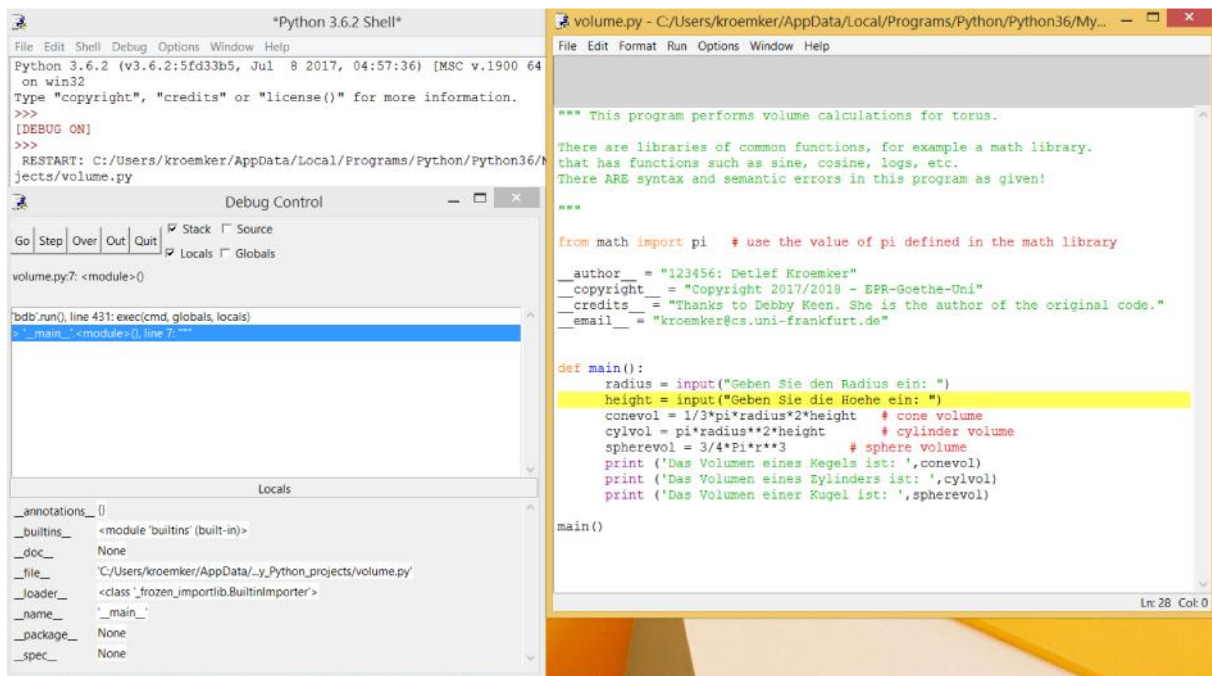
Einen Breakpoint setzt man im Editor-Fenster mit „Rechts-Klick“ auf die Zeile in der der Breakpoint sitzen soll und dann ‚Set Breakpoint‘ klickt. Die Programmzeile wird gelb markiert. Im Beispiel oben ist dies die Zeile

```
    height = …
```

der Fall. Mit **Clear Breakpoint** kann man einen gesetzten Breakpoint wieder löschen. Klickt man in einer anderen Zeile ein weiteres Mal **Set Breakpoint**, so wird ein weiterer Breakpoint gesetzt. Man kann in größeren Programmen „beliebig“ viele solcher Breakpoints haben. Achtung: Breakpoint zu setzen geht nur, wenn das Programm gespeichert ist. Die Fehlermeldung erfolgt leider nur mit einem akustischen Signal ‚beep‘.

Beachte: Die genaue Bedeutung eines Breakpoints ist: Halte **vor** der Ausführung dieser Zeile an.

Um die Untersuchung des Programms zu starten, betätigen Sie im Menue **Run** den Befehl **Run Module** oder als Shortcut **F5**.
Das Programm hält dann vor der Ausführung der ersten ausführbaren Codezeile an. Stellen Sie die drei Fenster von IDLE (Editor, Shell und Debug Control) so ein, dass sie alle drei Fenster überwachen können.

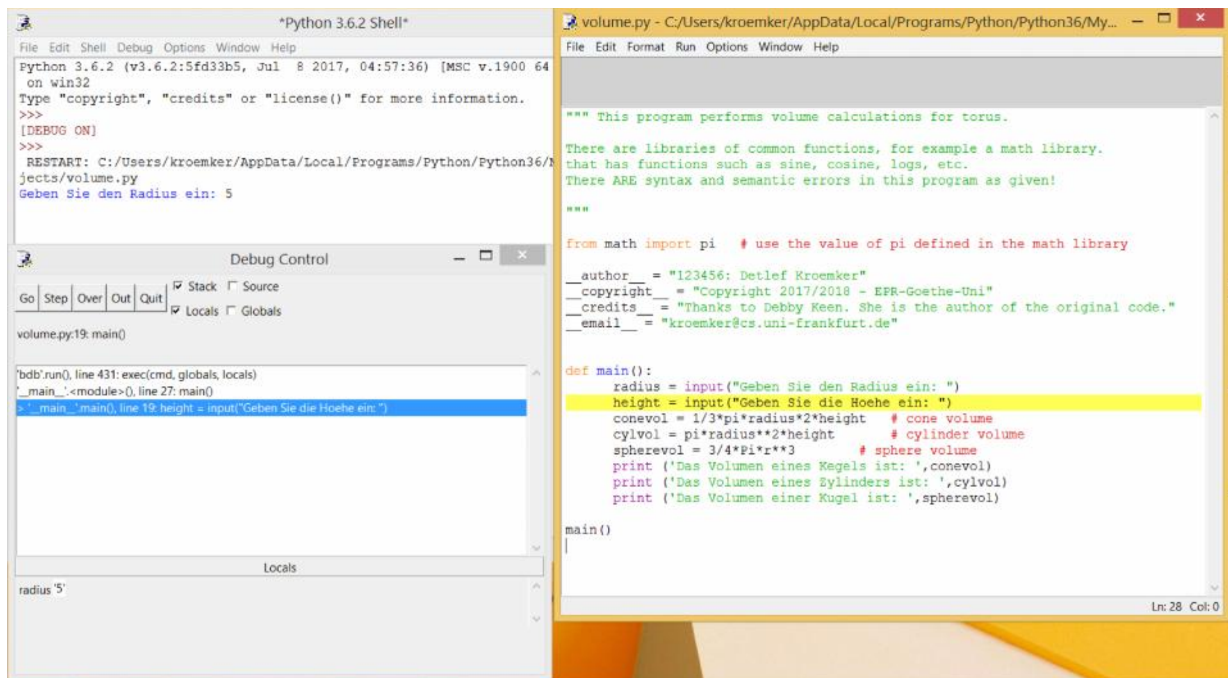



Sie sehen, der Debugger hält vor der Ausführung der ersten ausführbaren Programmzeile in line 7: """" (die Leerzeile wird nicht gezählt), also dem `from math import pi` Statement an. Jetzt können Sie mit **Go** die Fortsetzung des Programms bis zum ersten Breakpoint veranlassen.

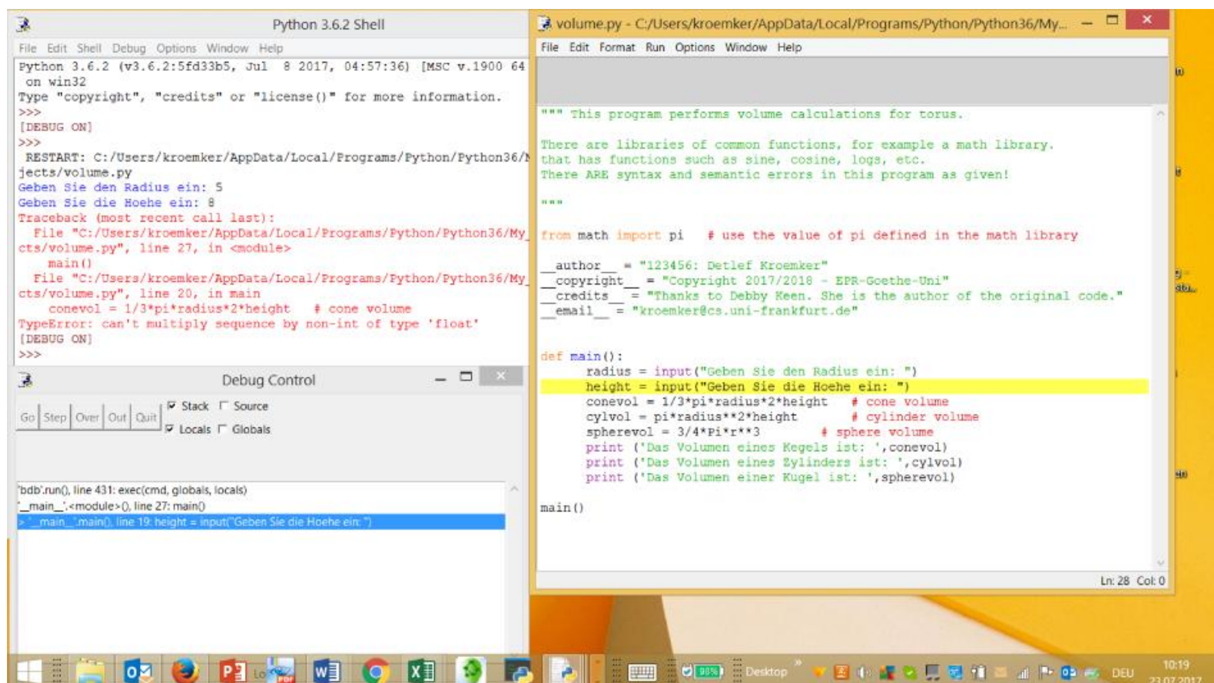
Natürlich müssen Sie das Programm z.B. in der Shell (der Konsole) bedienen. Für den Radius hatte ich nach der Aufforderung ‚5‘ eingegeben. Der Debugger hält vor der Ausführung des Breakpoints an: also (siehe Debug Control) vor:

>‘__main__‘.main(), line 19: height = input("Geben Sie die Hoehe ein: ").

Sie sehen weiterhin (unten), dass der Debugger den Programmkontext gewechselt hat, von Modul volume zum Unterprogramm main() und dort jetzt die lokale Variable radius den String `'5'` als Wert enthält. Beachten Sie, dass durch die Schreibweise als Literal auch der Typ angegeben wird.




Wir wählen jetzt, mit **Go** fortzufahren und werden (wie erwünscht) aufgefordert, die Höhe einzugeben, was auch mit dem Wert 8 erfolgt. Da kein weiterer Breakpoint gesetzt ist, sollte das Programm nun bis zum Ende durchlaufen. Leider erfolgt folgendes:




Der Interpreter wirft einen Fehler, der deutlich rotmarkiert im Shell-Fenster angezeigt wird: `Traceback …`

Die letzte Zeile der Fehlermeldung meldet uns die Ursache:

`TypeError: can't multiply sequence by non-int of type 'float'`

Tatsächlich haben wir vergessen, die eingelesenen Werte radius und height in ‚int‘ oder ‚float‘ zu wandeln. Wir wählen dazu `eval()` und korrigieren das Programm volume im Editor-Fenster. Gleich setzen wir auch noch den Breakpoint um, jetzt so, dass wir die Durchführung der Berechnungen beobachten können.

Natürlich müssen wir wieder alle Schritte von vorn ausführen: Ein Restart (F5) des Programms volume.py; diese hält vor der Ausführung des 1. Kommandos an; Go … und dann:

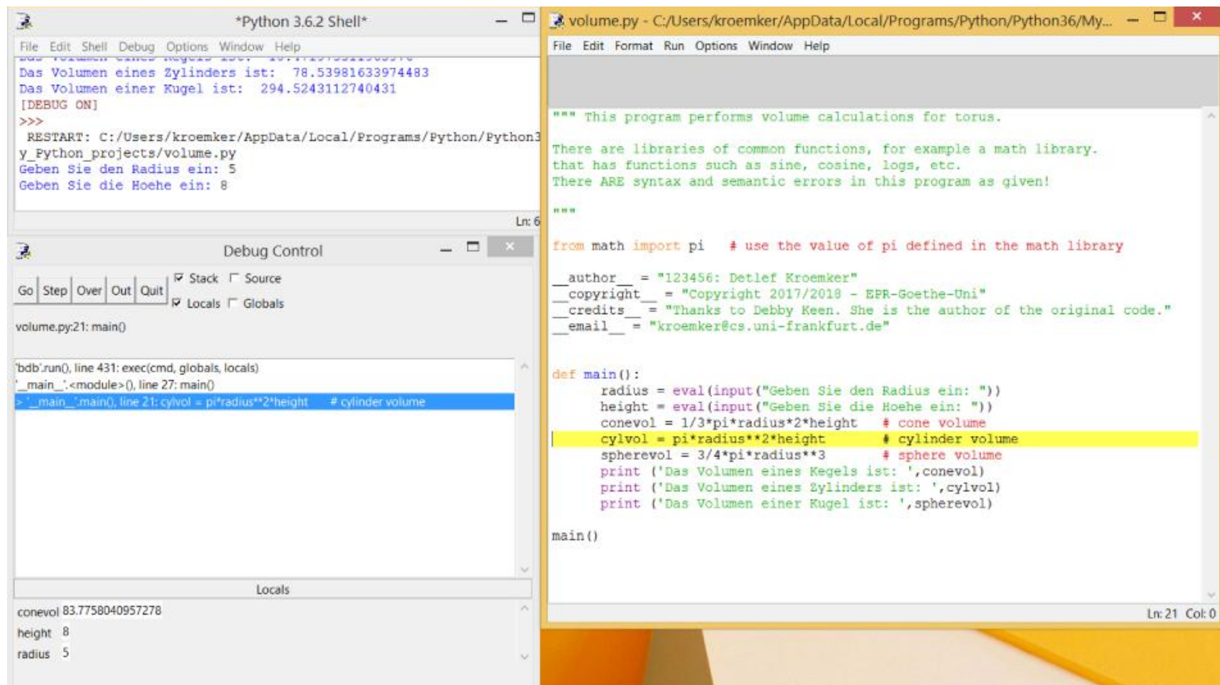
Leider läuft das Programm wieder nicht bis zum Breakpoint, wir bekommen eine Fehlermeldung:

```
    spherevol = 3/4*Pi*r**3         # sphere volume
NameError: name 'Pi' is not defined
```

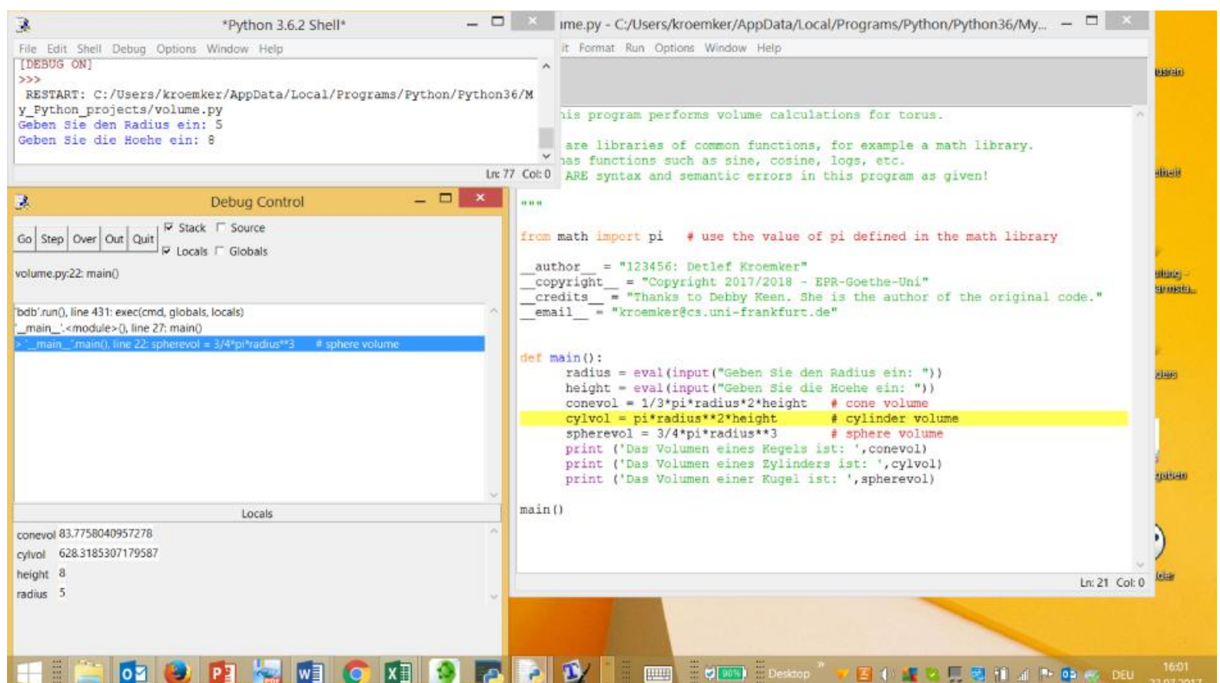
Klar, nur ein Tippfehler: ‚Pi' ist hier groß geschrieben und damit nicht bekannt. Außerdem bemerken wir auch noch einen weiteren Fehler: r ist auch nicht bekannt: r muss `radius` heißen.

Wir bemerken aber an dieser Stelle, dass `height` und `radius` nunmehr vom Typ Integer sind.

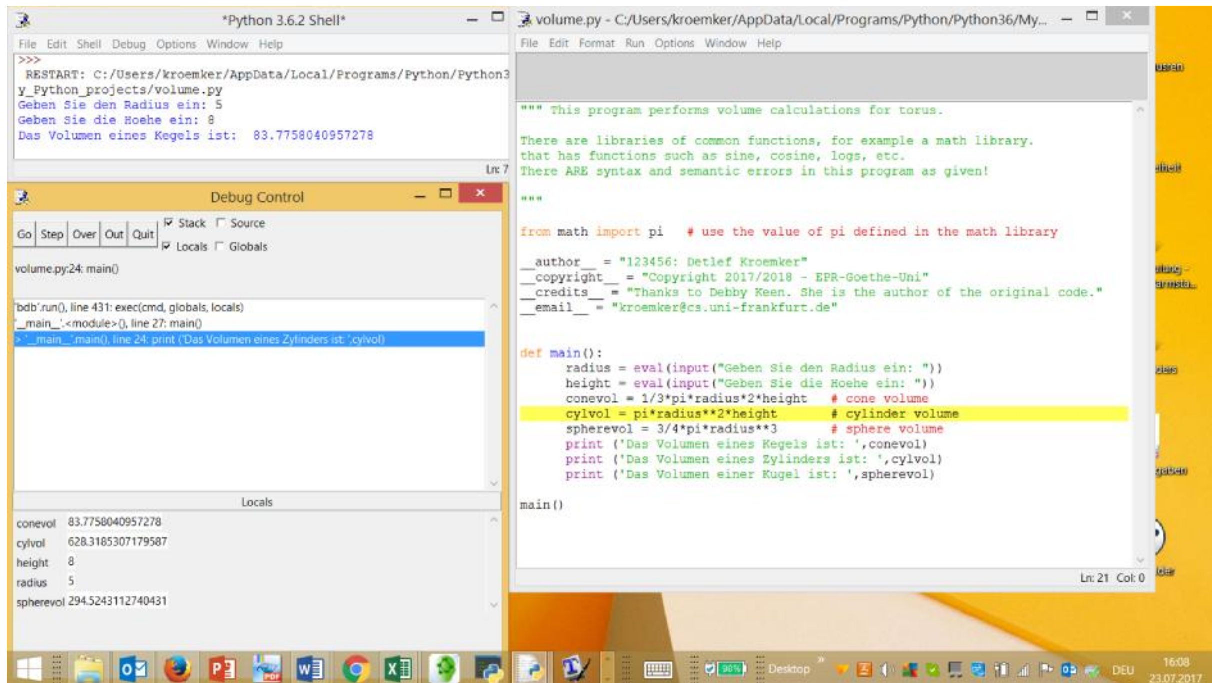
Schnell korrigiert und dasselbe Procedere wieder von vorn.



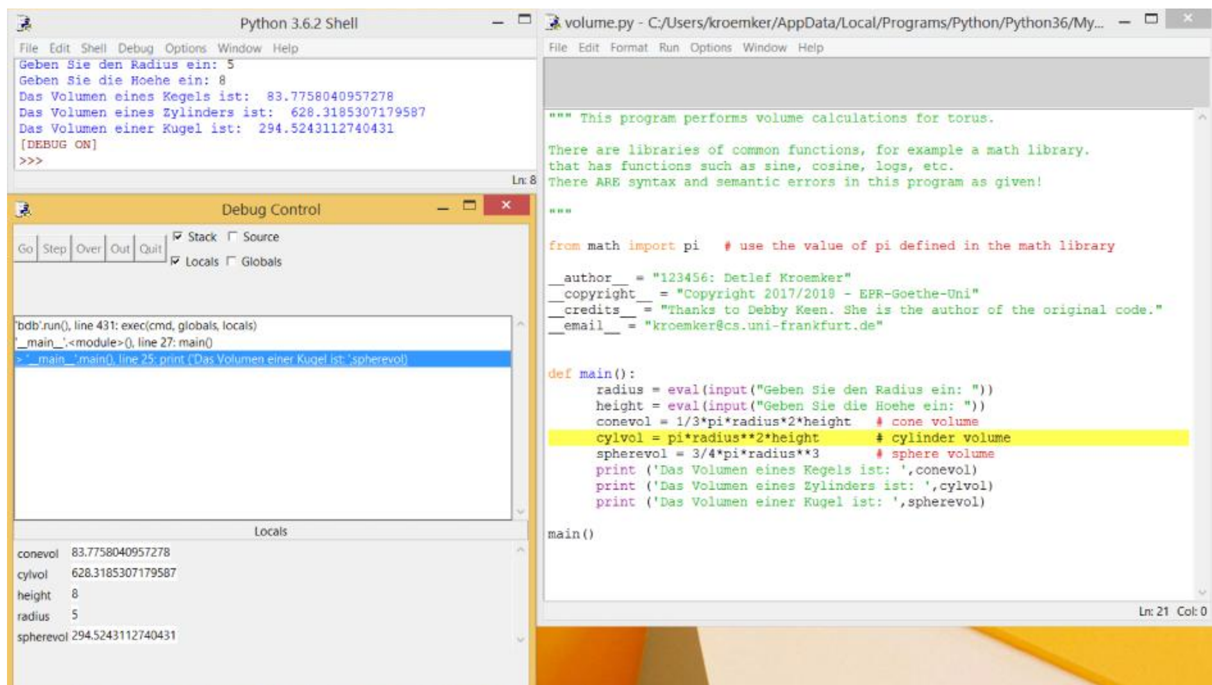

Jetzt ist unser Programm wie erwünscht bis zum nächsten Breakpoint durchgelaufen: Im Debug Control Fenster sehen wir auch, dass die Eingabewerte und der Berechnungswert richtig sind und können jetzt fortfahren: Wir versuchen jetzt einmal den Button **Step**.



Tatsächlich sind wir eine Zeile weiter geschritten. Noch zweimal weitersteppen.
Ooops! Jetzt sind wir irgendwo in der Ausführung der `print()`- Routine gelandet. Meist ist dies unsinnig, da dort die Fehler nicht zu suchen sind. Wenn wir einmal in eine nicht zu untersuchende Funktion gelandet sind, kommen wir durch den **Out**-Button wieder heraus, jetzt direkt vor die zweite `print()`- Routine.




Ein **Over** überspringt die weitere Ausführung der externen Funktion. Ein weiteres **Over** bringt das Programm volume zum Abschluss, siehe die Shell, in der alle drei Ausgaben gemacht wurden und erneut die Eingabeaufforderung ‚>>>' steht.




Also zusammengefasst umfasst das Arbeiten mit dem Debugger folgende Schritte:

- Breakpoint(s) setzen.
- Programmausführung mit **F5** im Editorfenster starten.

- Mit **Go** bis zum nächsten Breakpoint das Programm ausführen oder mit **Step** das nächste Statement ausführen, mit **Over** (wenn ein Unterprogramm ausgeführt werden soll, dessen Korrektheit man unterstellt) oder mit **Out** bis zum Ende des externen Function Calls ausführen (‚step out oft he function‘).
- Inspizieren der Werte und Typen der Variablen und Ihre Änderungen im Debug Control Fenster und des Programmverhaltens z.B. in der Shell.
- Programm-Korrekturen im Editor-fenster ausführen.
- Wiederausführung des Programmes mit **F5**, usw..

**Den Debugger zu benutzen ist kein Allheilmittel.** Bei Syntaxfehlern und einfachen Laufzeitfehlern (insbesondere solche, die mit einem `Traceback` enden) hilft er kaum. Diese debugged man lieber ohne Debugger. Erst wenn man **keine** Idee mehr hat, woher der Fehler kommen könnte oder wenn man Programmergänzungen (z.B. print(<Locals>) durchführen müsste, lohnt sich der Einsatz des Debuggers. Wichtig ist, dass sie bei der Ausführung immer alle drei Fenster: (Shell, Debug Control und Editor) im Auge haben.

Es fehlen noch ein paar Bedienmöglichkeiten:

- Der Button **Quit** halt den aktuellen Programmlauf an.

Die Checkboxen::

- **Globals** bewirkt, dass unterhalb der lokalen Variablen des aktuellen Geltungsbereichs (**Locals**) zusätzlich auch die globalen Variablen Globals angezeigt werden.
- **Source** bewirkt, dass die aktuelle Programmzeile (gerade ausgeführte) grau markiert wird: Es ist also etwas einfacher zu verfolgen, wo der Interpreter aktuell arbeitet.

Bitte auch beachten: In der Dokumentation des Debuggers[3] ist vermerkt: „This feature is still incomplete and somewhat experimental.“ Er ist aber hilfreich – sonst hätten wir ihn nicht behandelt.

[3] The Python Standard Library – 25.5. IDLE https://docs.python.org/3.6/library/idle.html (zuletzt abgerufen am 24.07.2017).