

Introduction

Key idea

In a development process, the developer can encounter some unpredictable and unwanted errors. These can be:

- classical Python exceptions, as introduced previously
- inconsistent scientific results that suggest a code error

The **debugger** is a tool that makes solving these problems an easier task. Using the debugger, one can pause running code at some given instructions, called **breakpoints**. Whenever a breakpoint is encountered, the user can:

- observe the some variable values (local or global variables)
- evaluate some new expressions using these variables
- enter the details of the breakpoint and run these instructions one after another
- resume the execution until a new breakpoint is met

Execution is **always** paused **before** the breakpoint, as if breakpoint occurred at the end of the previous instruction.

Howto

The native debugger of Python is a library called `pdb`. Whenever the `set_trace` method is used to declare breakpoints, execution falls back to debug mode.

Yet, `pdb` is not handy, and a much better debugger integration is done within recent IDE. In this part, the debugger of *Visual Studio Code* is introduced.

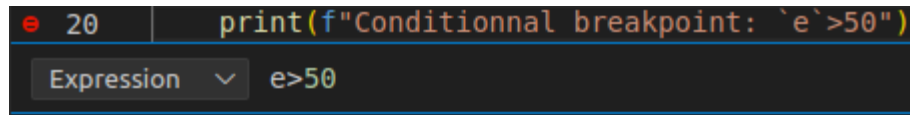
Debugging using VSCode

Set some *breakpoints*

Let's focus on the following code:

```
to_debug.py > ...
1
2 def f1(a, b, c):
3     print("Entering f1")
4     a = min(a, 5)
5     print("After first instruction")
6     d = a * b + c
7     print("After second instruction")
8     idx = 0
9     while d < 100:
10         d += 1
11         f2(a, d)
12         idx += 1
13     print("Exiting f1")
14
15
16 def f2(a, d):
17     print("Entering f2")
18     e = a ** (d%5)
19     f3()
20     print(f"Conditionnal breakpoint: `e`>50")
21     print("Exiting f2")
22
23
24 def f3():
25     print("Entering f3")
26     print("Still in f3")
27     print("Exiting f3")
28
29
30 f1(7, 8, 4)
```

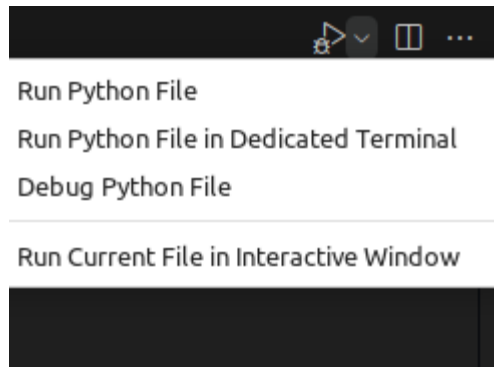
By a left clic in the margin, 4 breakpoints were defined (red dots). At lines 4, 9 and 19 are normal breakpoints. Line 20 is a conditional breakpoint: a breakpoint that is activated is and only if `e>50`.



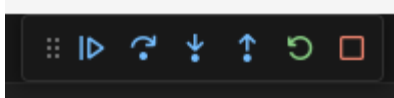
The screenshot shows a code editor with a dark background. A line of code is highlighted in blue: `print(f"Conditionnal breakpoint: `e`>50")`. To the left of the code, in the margin, there is a red dot indicating a breakpoint. Below the code line, there is a dropdown menu with the text "Expression" and a downward arrow. To the right of the dropdown, the expression `e>50` is entered.

Launch the debugger

Unfold the menu at the right hand side of the page and clic on 'Debug Python file':



A new toolbar is printed at the top:



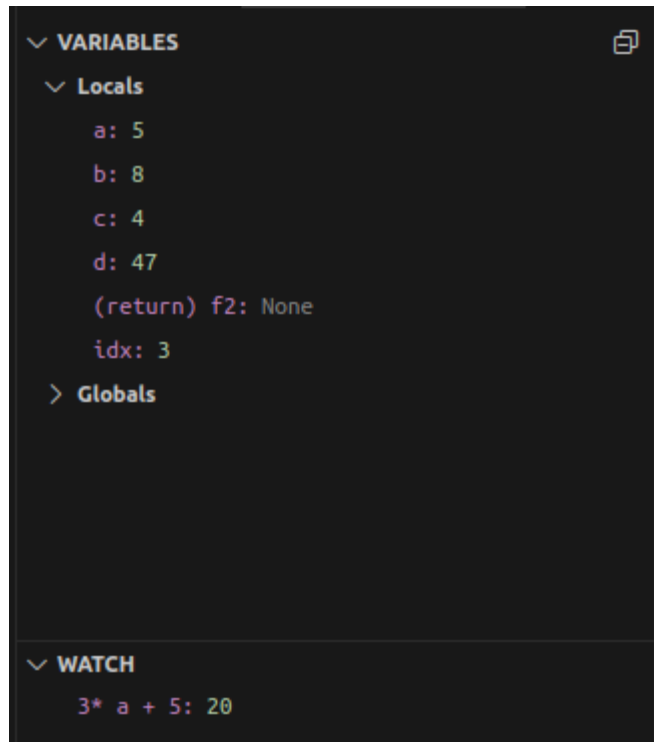
From left to right:

- *Continue*: resume execution until next breakpoint
- *Step over*: run one instruction after another. If the instruction is a function call, execution is paused only at breakpoints of the called function.
- *Step into*: run one instruction after another. If the instruction is a function call, execution is paused at every instruction of this call, no matter the presence of breakpoints or not.
- *Step out*: get out of a previously issued *step into* by executing every instruction without pausing, until the end of the function
- *Stop*: exit debug mode

Debug tools

At every moment, local **variables** (for instance, the variables defined in the function currently inspected) are showed in a pannel on the left hand side. Their value change in real time as execution goes on.

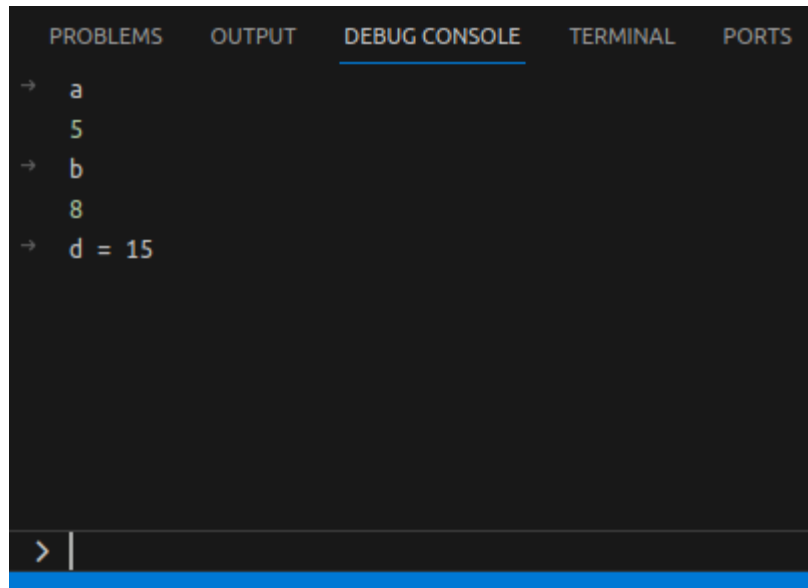
Similar to panel **variables** is panel **watch**: one can define custom Python expressions involving variables. These expressions are also updated as the execution goes on.



A right clic on a variable shows options to:

- copy the variable value
- explicitly set this value

Yet, these options are more accessible in the **DEBUG CONSOLE**, at the bottom of the window:



Execution

As usual, outputs of the execution are visible in the **TERMINAL** tab, next to **DEBUG CONSOLE**.

```
Entering f1
After first instruction
After second instruction
Entering f2
Entering f3
Still in f3
Exiting f3
Conditionnal breakpoint: `e`>50
Exiting f2
```

Important notes

There is not return trip in debug mode: the execution of an instruction cannot be undone.

For this reason, breakpoints must be carefully placed **before** the problematic section. Usually, setting less than 5 breakpoints is enough to debug, as functions *Step over* and *Step into* are pretty powerful.

