Introduction

Key idea

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

- classical Python exceptions, as introduced previously
- inconsistent scientific results that sugget 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 occured at the end of the previous instruction.

Howto

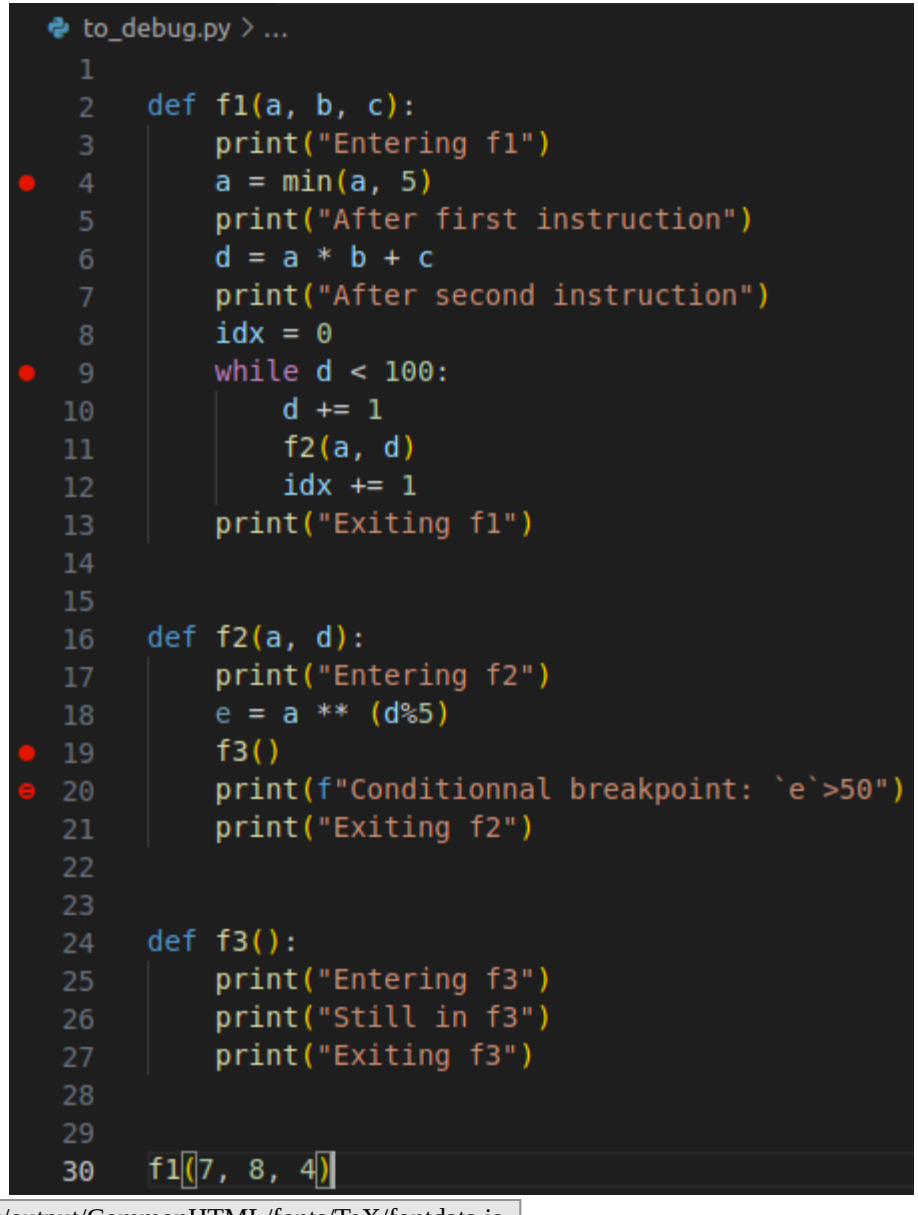
The native debugger of Python is a library called `pdb`. Whenever the `set_trace` method is used to declare breakpoints, execution falls back to debug mode.

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.

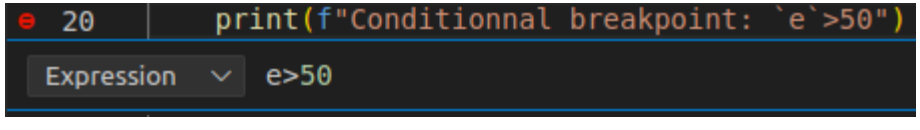
Debugging using VSCode

Set some *breakpoints*

Let's focus on the following code:

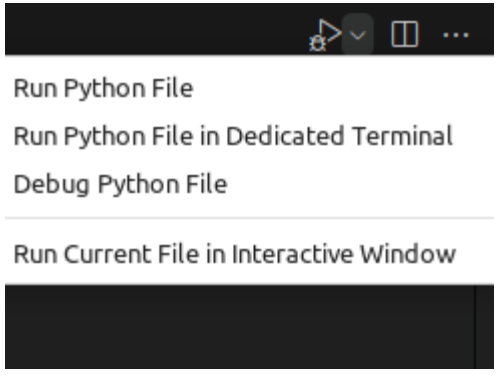


By a left clic in the margin, 4 breakpoints were defined (red dots). At lines 4, 9 and 19 are normal breakpoints. Line 20 is a conditional breakpoint: a breakpoint that is activated is and only if `e>50`.



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:

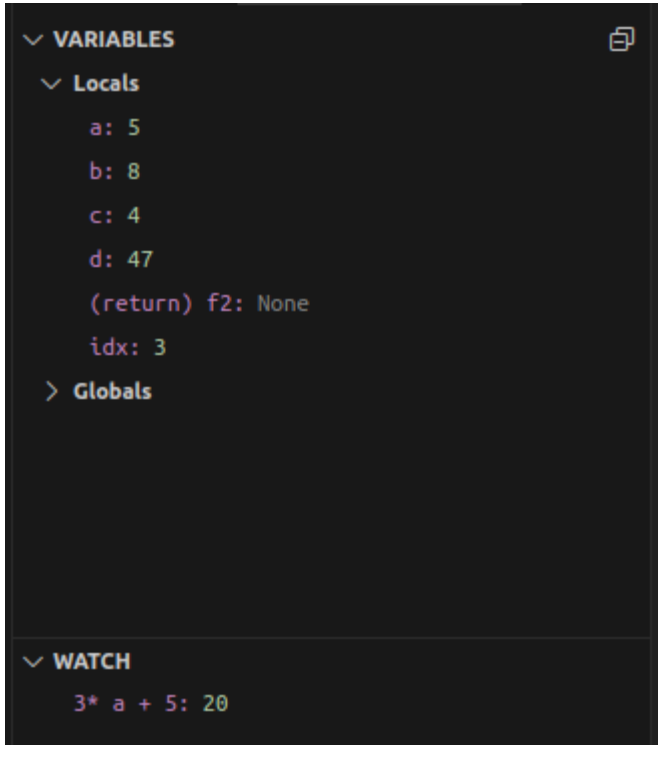
From left to right:

- *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.

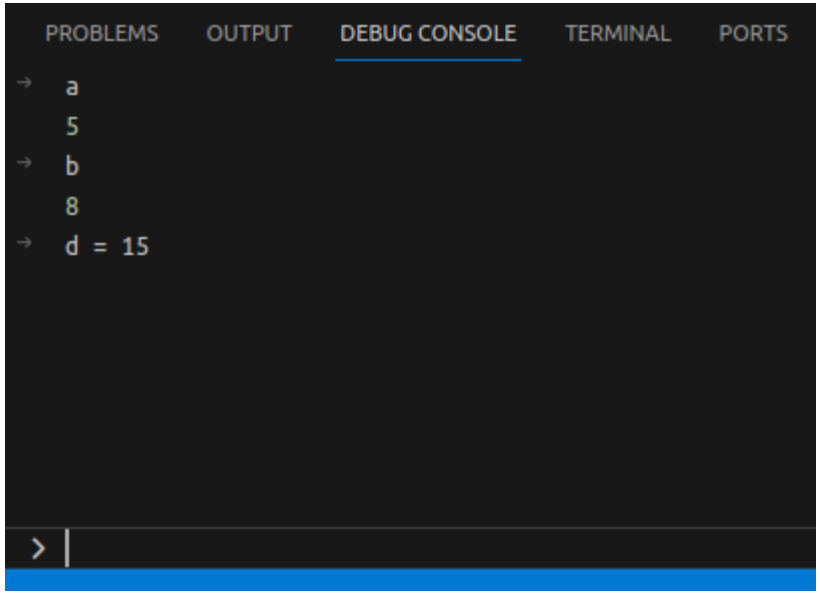
Similar to panel **variables** is panel **watch**: one can define custom Python expressions involving variables. These expressions are also updated as the execution goes on.



A right clic on a variable shows options to:

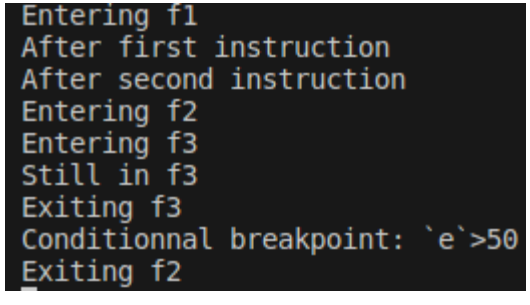
- copy the variable value
- explicitely set this value

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**.



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.