Debugging

Karl Kosack
CEA Paris-Saclay

ASTERICS-OBELICS International School
Annecy, June 2017

H2020-Astronomy ESFRI and Research Infrastructure Cluster
(Grant Agreement number: 653477).

Debugging

When you run a piece of code and:

- get an error/crash/exception
- encounter an unexpected result
- want to know what the code is doing

Do you:

- Add a bunch of *print/cout/printf* statements and try to track down the issue?
- Run the code in a debugger?

If you said "print statements", you have some learning to do!

Debugging

When you run a piece of code and:

- get an error/crash/exception
- encounter an unexpected result
- want to know what the code is doing

Do you:

- Add a bunch of *print/cout/printf* statements and try to track down the issue?
- Run the code in a debugger?

If you said "print statements", you have some learning to do!

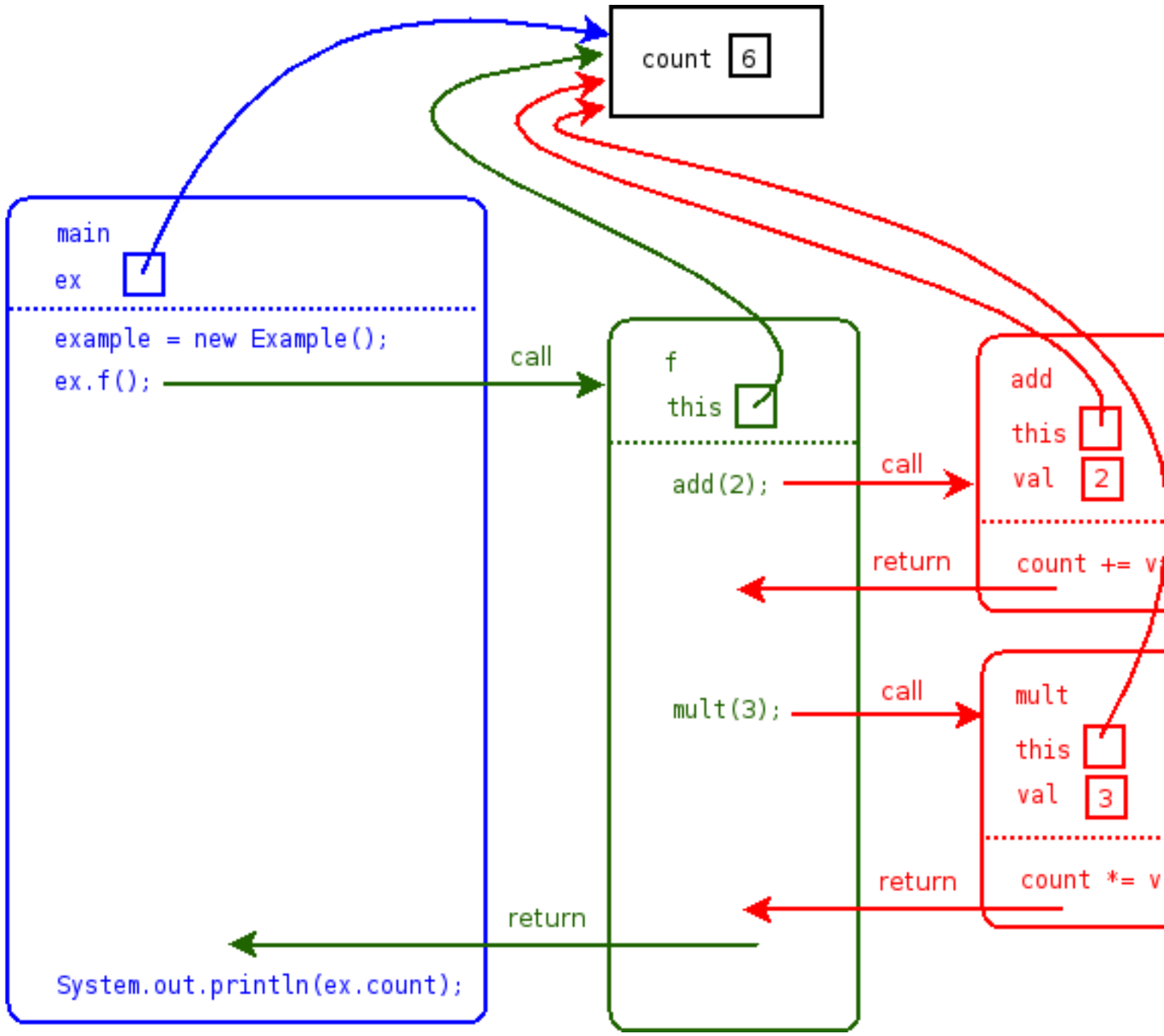
Aside: program flow and memory

Heap:

- all global variables, dynamic memory

Stack:

- All functions currently being executed and their local variables
- Single function's data is stored in a "**Stack Frame**",
- Frames are *stacked* on top of each other to represent hierarchy (bottom of stack =outermost)



caveat: python's memory management and stack is at a higher level of abstraction than this, but conceptually is the same

diagram from: http://faculty.ycp.edu/~dhovemey/spring2007/cs201/info/exceptionsFileIO

What is a debugger?

A debugger:

- **runs** or **attaches** to a *running* piece of code or a program that has just crashed or had an exception
- allows you to **view the value** of any variable
- allows you to **move through the execution** of the code and **inspect data**!
  - ➤ go to next line
  - ➤ step into function
  - ➤ go up or down one level of function calls (the "call stack")
  - ➤ watch a variable for change
  - ➤ keep running until a condition occurs

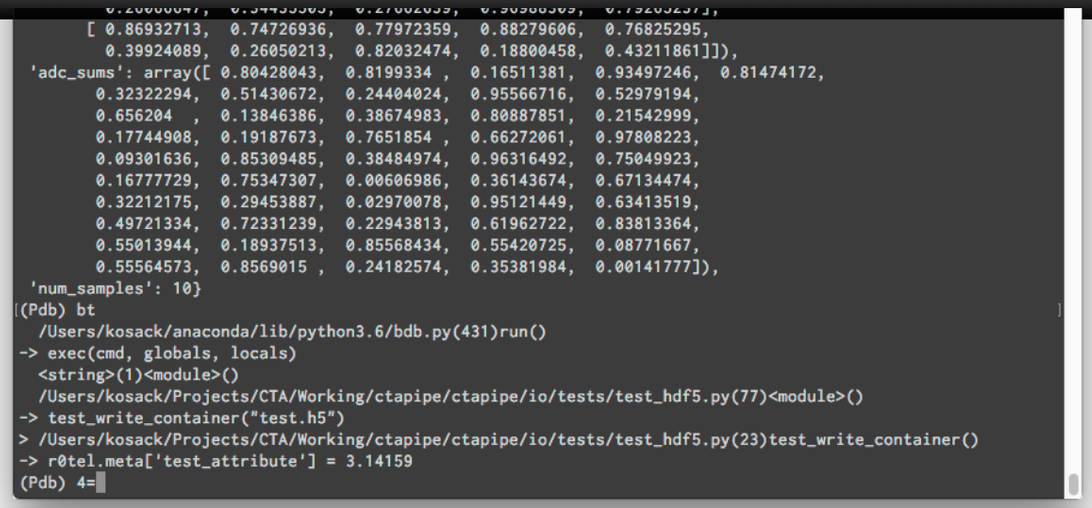
The basic use/concepts of debuggers is independent of language (a C++ debugger works the same as a python debugger)

Two levels of debugging interface

Text-mode debuggers:

- gdb (c/c++), pdb (python)
- simple command-line interface, with text commands
- good for quick debugging

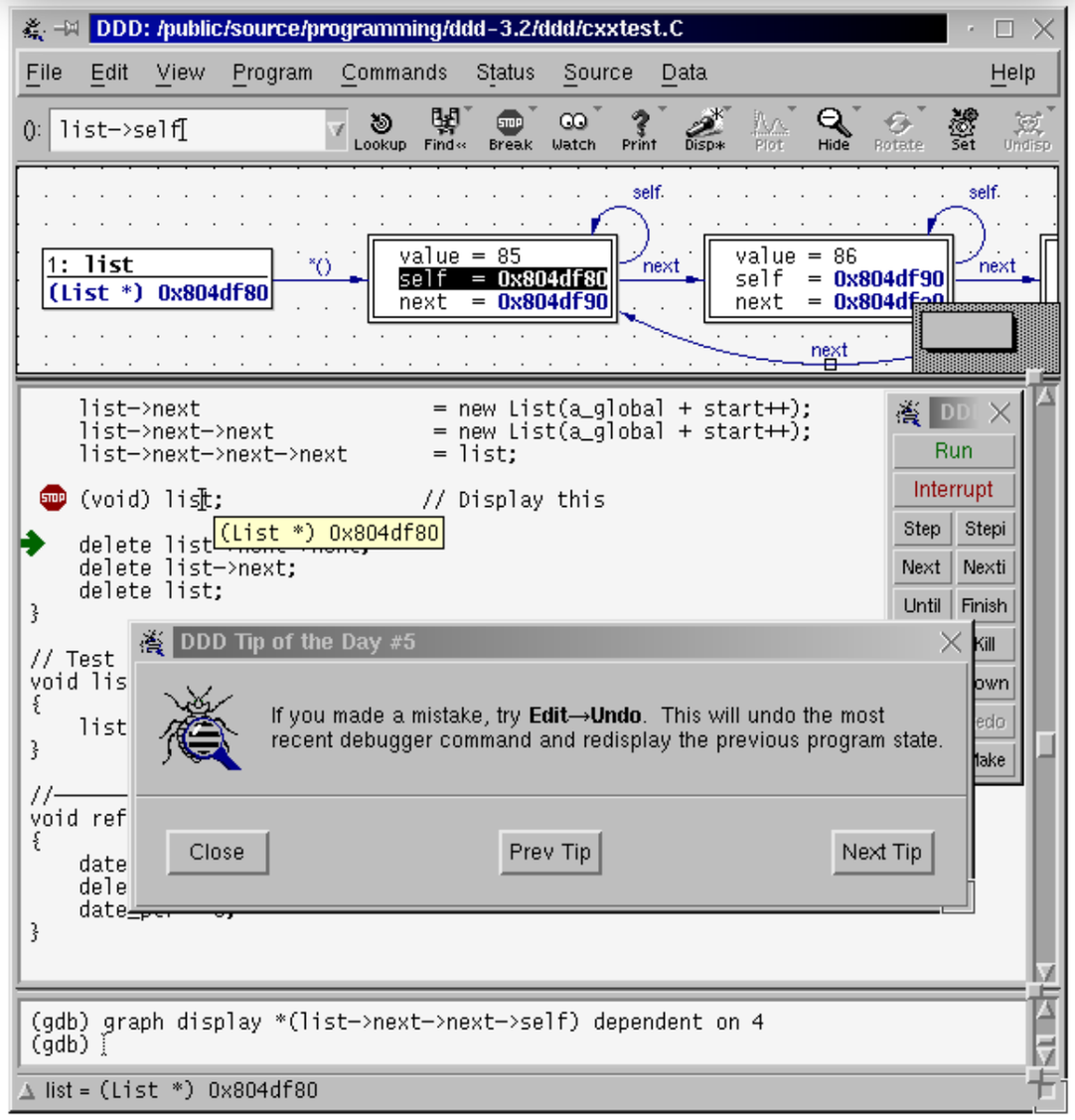
pdb



GUI Debuggers:

- often integrated with nice interactive development environments (IDEs)
- Allow point-and-click inspection of code and variables
- Examples:
  - ddd [Data Display Debugger] (c/c++)
  - PyCharm's debugger (python)

GNU ddd

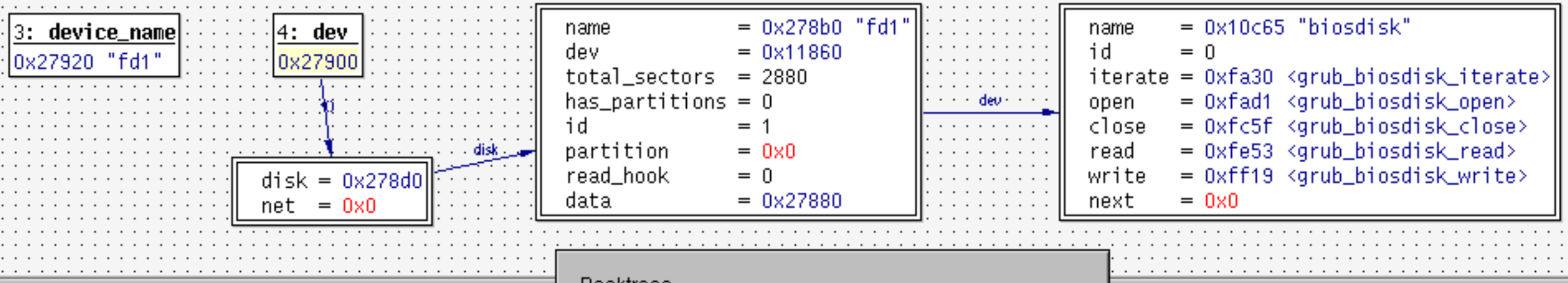


File Edit View Program Commands Status Source Data Help

0: dev

Lookup Find» Break Watch Print Display Plot Hide Rotate Set Undisp

Run Interrupt Step Stepi Next Nexti Until Finish Cont Kill Up Down Undo Redo Edit Make




```
      grub_printf ("%-12s", "DIR");

    grub_printf ("%s%s\n", filename, dir ? "/"

    return 0;
  }

  device_name = grub_file_get_device_name (dirna
  dev = grub_device_open (device_name);
  if (! dev)
    goto fail;

  fs = grub_fs_probe (dev);
  path = grub_strchr (dirname, ')');
  if (! path)
    path = dirname;
  else
    path++;

  if (! path && ! device_name)
```

Backtrace

```
#7  0x0009776e in grub_command_execute (
#6  0x00098a15 in grub_script_execute ()
#5  0x000985f0 in grub_script_execute_cm
#4  0x0009890e in grub_script_execute_cm
#3  0x000985f0 in grub_script_execute_cm
#2  0x00098847 in grub_script_execute_cm
#1  0x0009322f in grub_cmd_ls () at ls.c
#0  grub_ls_list_files () at ls.c:145
```

Up Down Close Help

```
(gdb) graph display *dev dependent on 4
(gdb) graph display *(dev->disk) dependent on 5
(gdb) graph display *(dev->disk->dev) dependent on 6
(gdb) graph display *(dev->disk->data) dependent on 6
(gdb) Attempt to dereference a generic pointer.
Disabling display 8 to avoid infinite recursion.
(gdb) graph undisplay 8
(gdb)
```

Display 4: dev (enabled, scope grub_ls_list_files, address 0x67cc4)

Debugging python code

There are many ways to enter the text-mode debugger PDB:

DEBUGGING AFTER AN EXCEPTION (my most common use case)

- 1) run a python program in *ipython*
- 2) it crashes with an exception
- 3) type **%debug** to enter PDB and jump to where the exception occurred!
- (alternately run "ipython —pdb <script.py>")

common PDB commands (and the same for gdb!):

- **u**(p), **d**(own) (move in the stack)
- **bt** (backtrace) == where
- **cont**(inue) running program
- **n**(ext) [next line]
- **s**(tep) into next operation (e.g. into functions)
- **l** and **ll** (list + longlist) of code at point
- **q** (quit debugging)
- any python expression

- DEMO -

Debugging python code

Use Case 2: no exception occurred, but you want to see what is happening inside a function

- **Brute-force**: place this line where you want to halt the program and start debugging:

```
import pdb; pdb.set_trace()
```

- More work, but more flexible: run the script inside the debugger:

```
python -m pdb  myscript.py
```

  - the script will not run, but rather start at the first statement and then wait for you to type commands
  - use *next, step, cont* to step through program
  - set a breakpoint! (break <linenumber>) and *continue* to it!

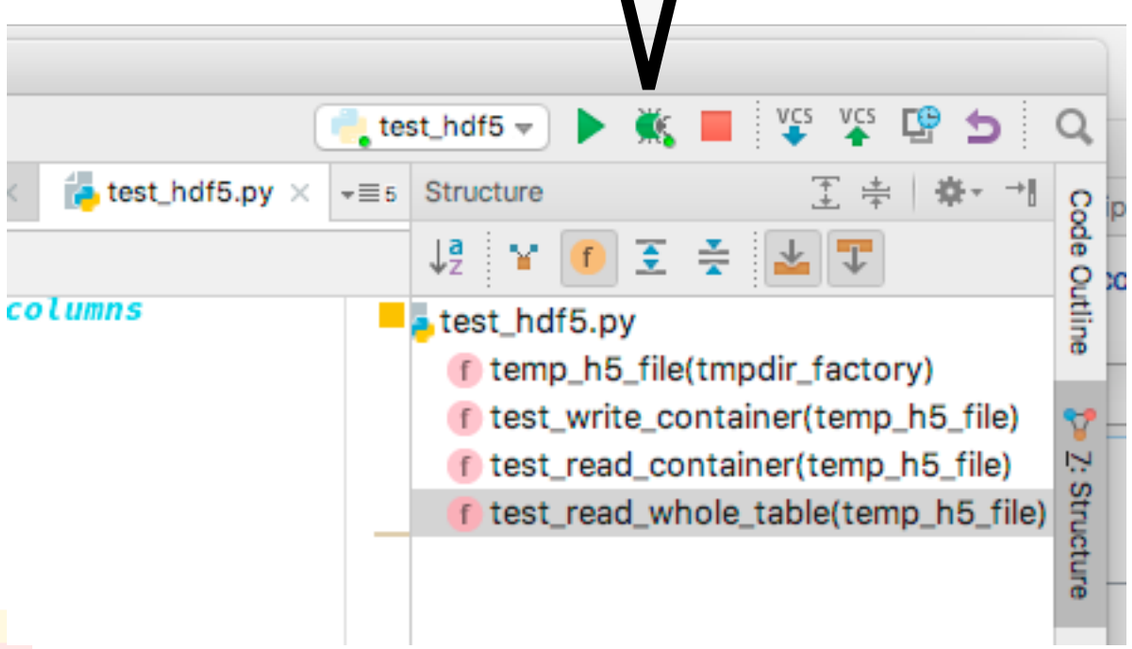
- DEMO -

GUI Debugging

This is all nice and good, but it gets tedious for more than simple debugging…

Solution: use a GUI debugger!

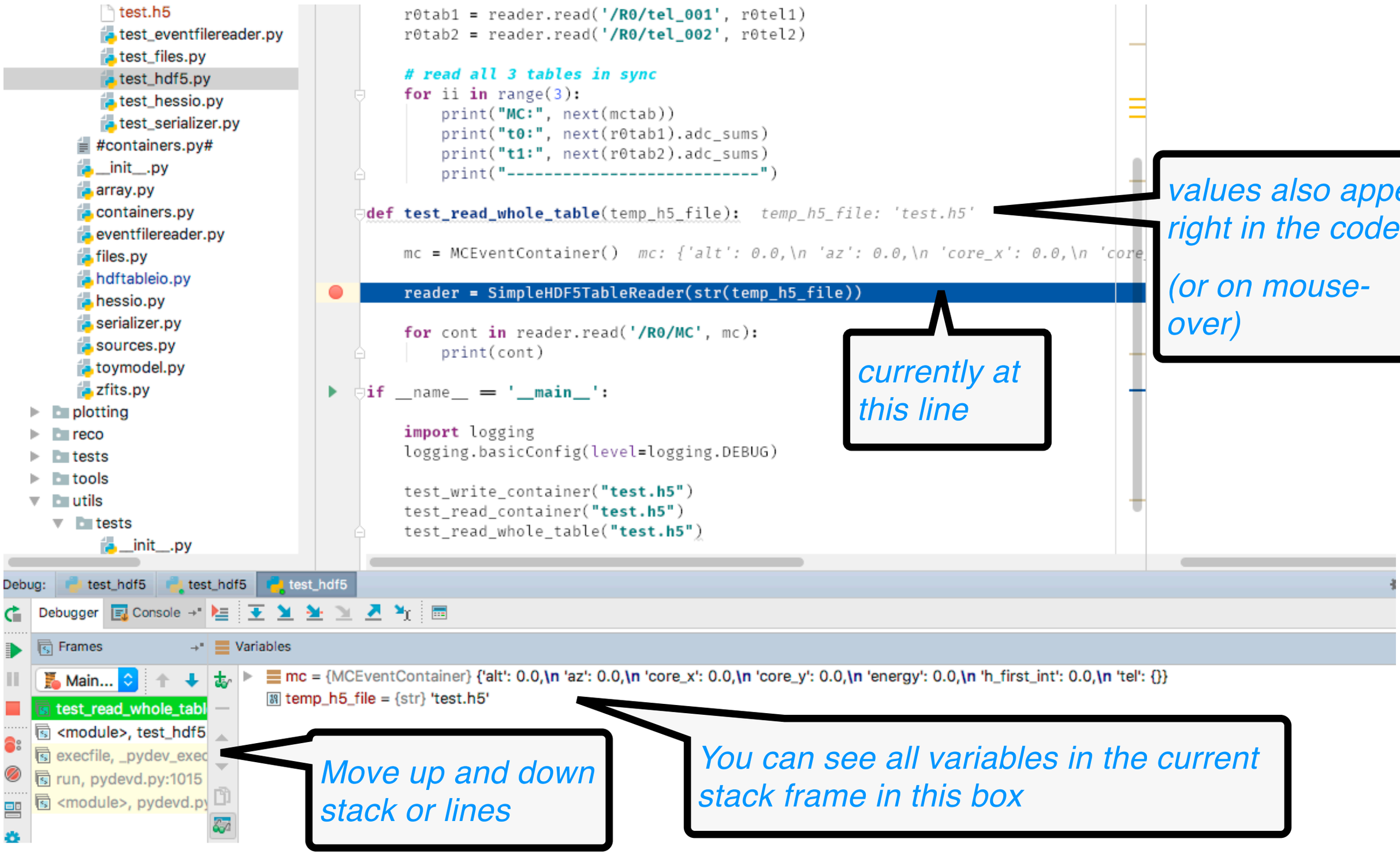
Open the "executable" part of the script and click the "debug" icon in the top-right corner



Click in margin to set a breakpoint

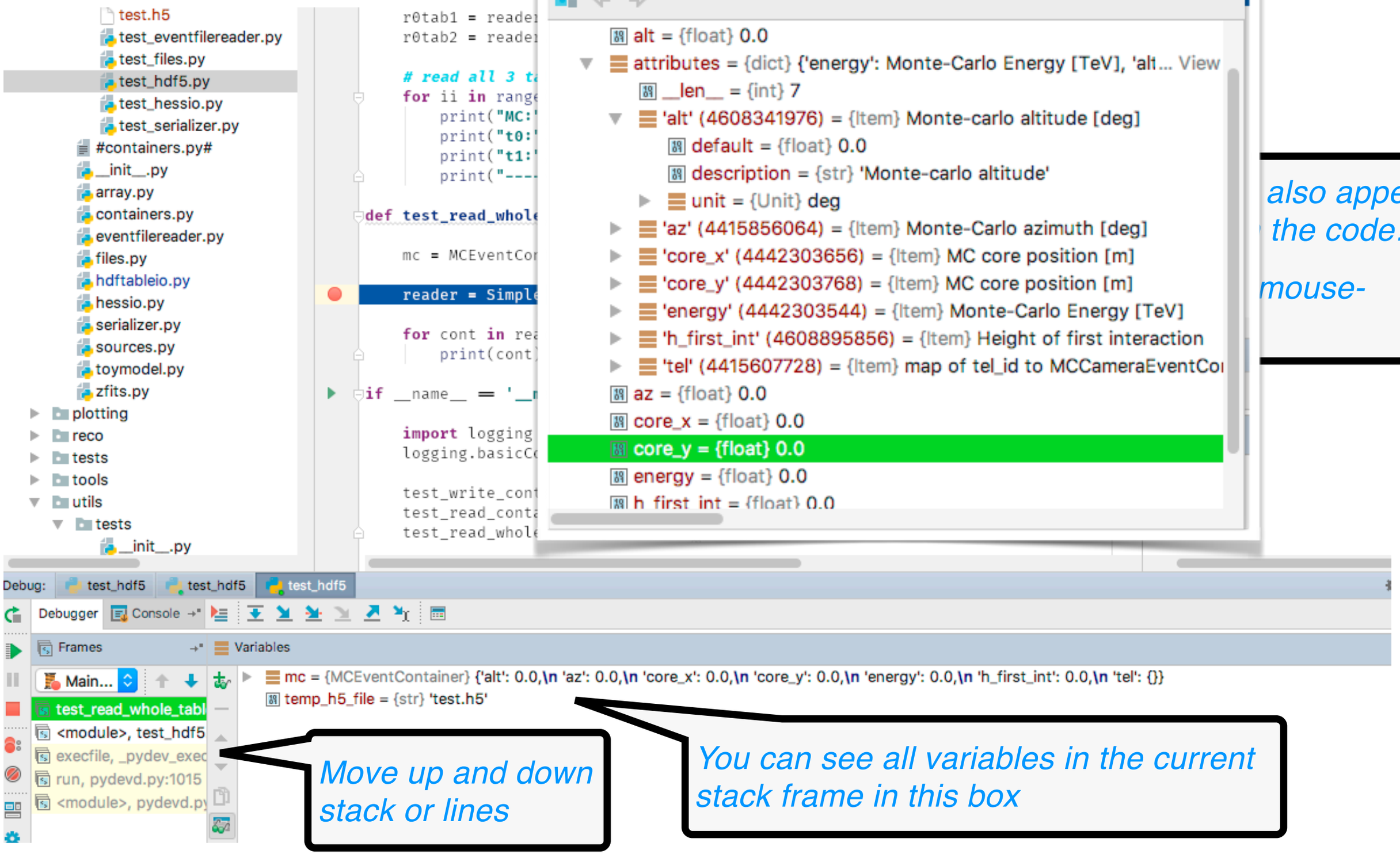
```
der.py
                mctab = reader.read('/R0/MC', mc)
                r0tab1 = reader.read('/R0/tel_001', r0tel1)
                r0tab2 = reader.read('/R0/tel_002', r0tel2)

                # read all 3 tables in sync
                for ii in range(3):
                    print("MC:", next(mctab))
                    print("t0:", next(r0tab1).adc_sums)
                    print("t1:", next(r0tab2).adc_sums)
                    print("---------------------------")

def test_read_whole_table(temp_h5_file):

                mc = MCEventContainer()

                reader = SimpleHDF5TableReader(str(temp_h5_file))

                for cont in reader.read('/R0/MC', mc):
                    print(cont)

if __name__ == '__main__':

                import logging
                logging.basicConfig(level=logging.DEBUG)

                test_write_container("test.h5")
asets.py
histogram.py
_2_fits.py
lg.py
ss.py
m.py
```

GUI debugging



GUI debugging

Drill deep down into any data structure!



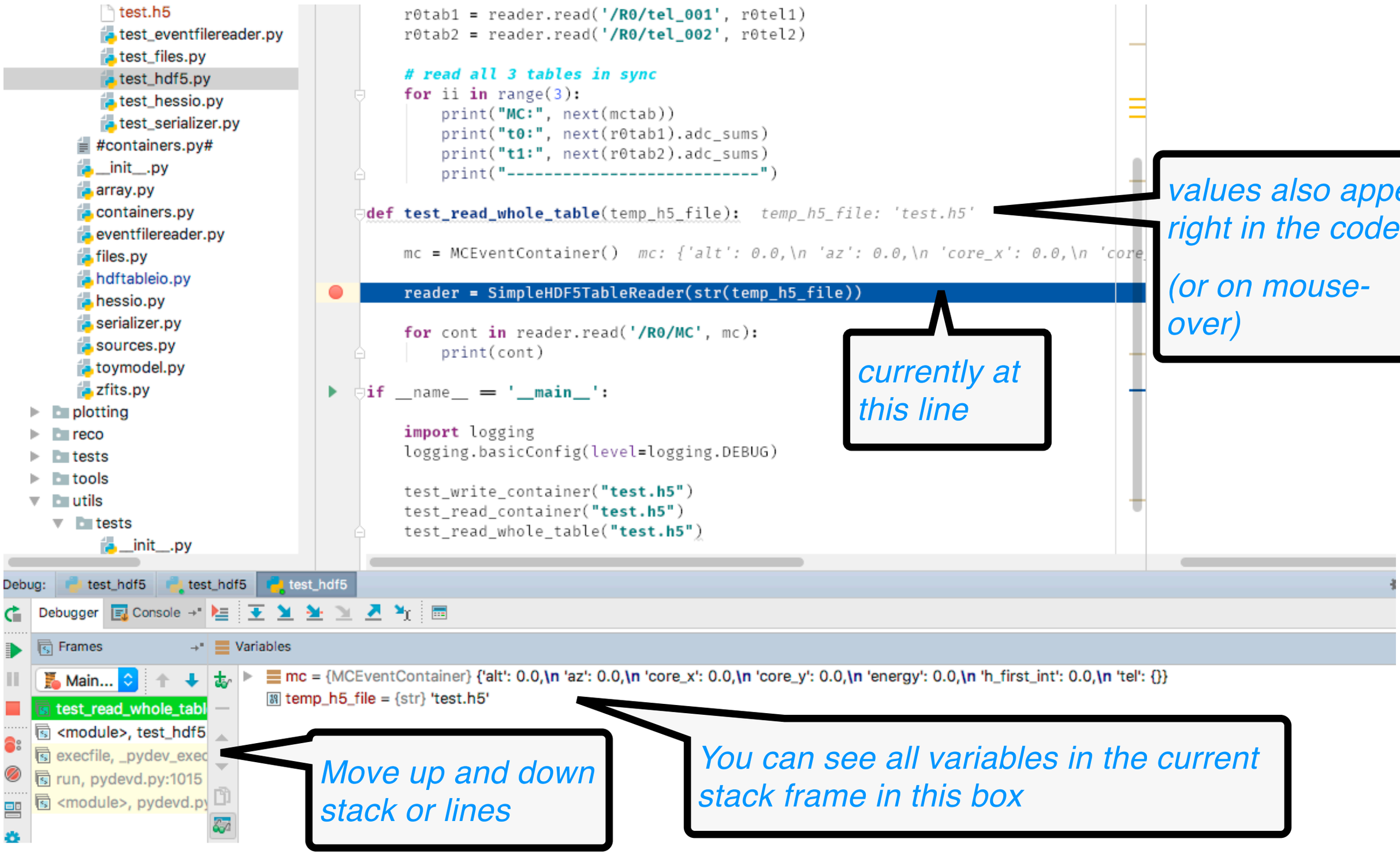
also appear the code!

mouse-

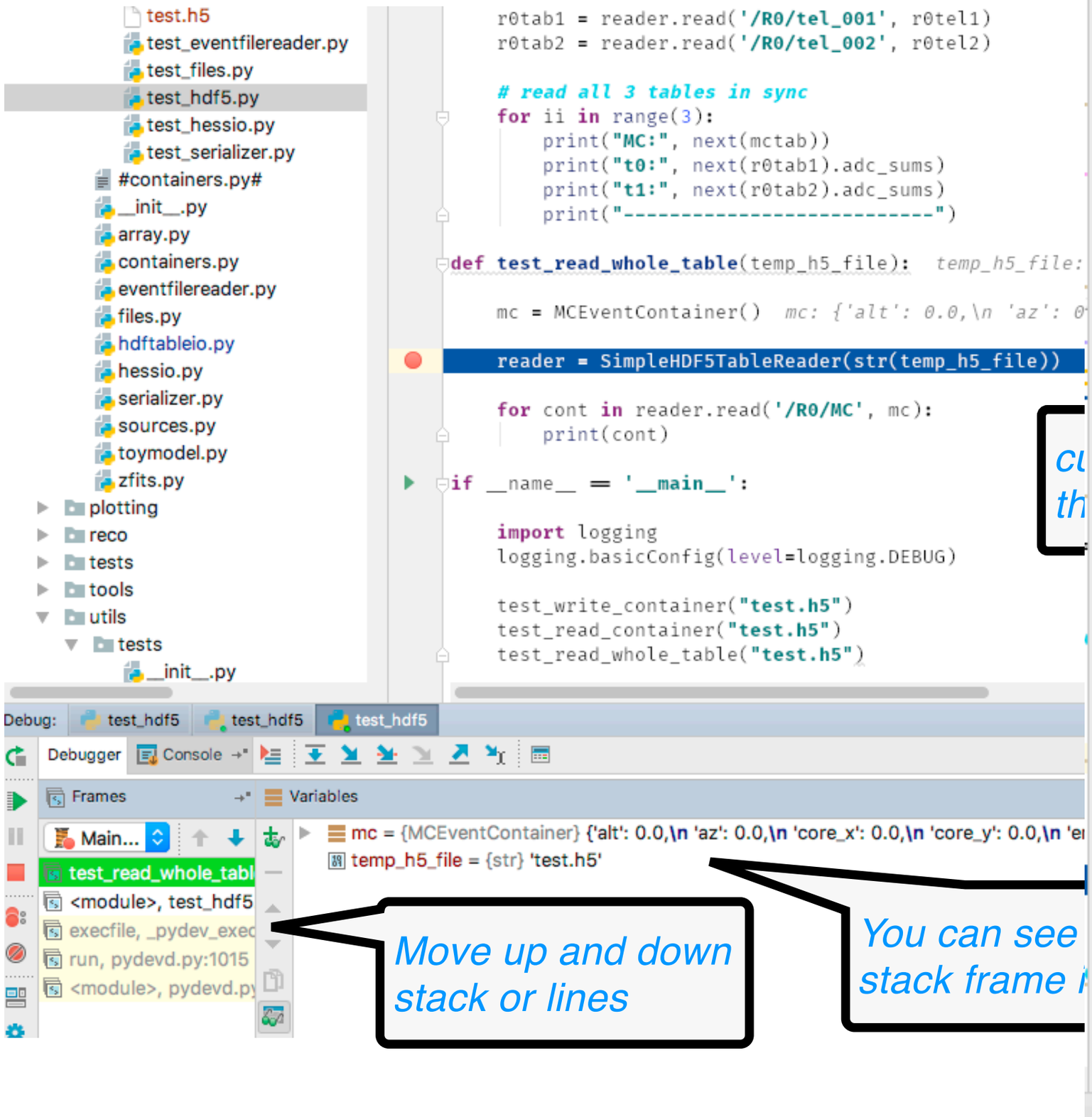
Move up and down stack or lines

You can see all variables in the current stack frame in this box

GUI debugging



GUI debugging



- DEMO -