

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!

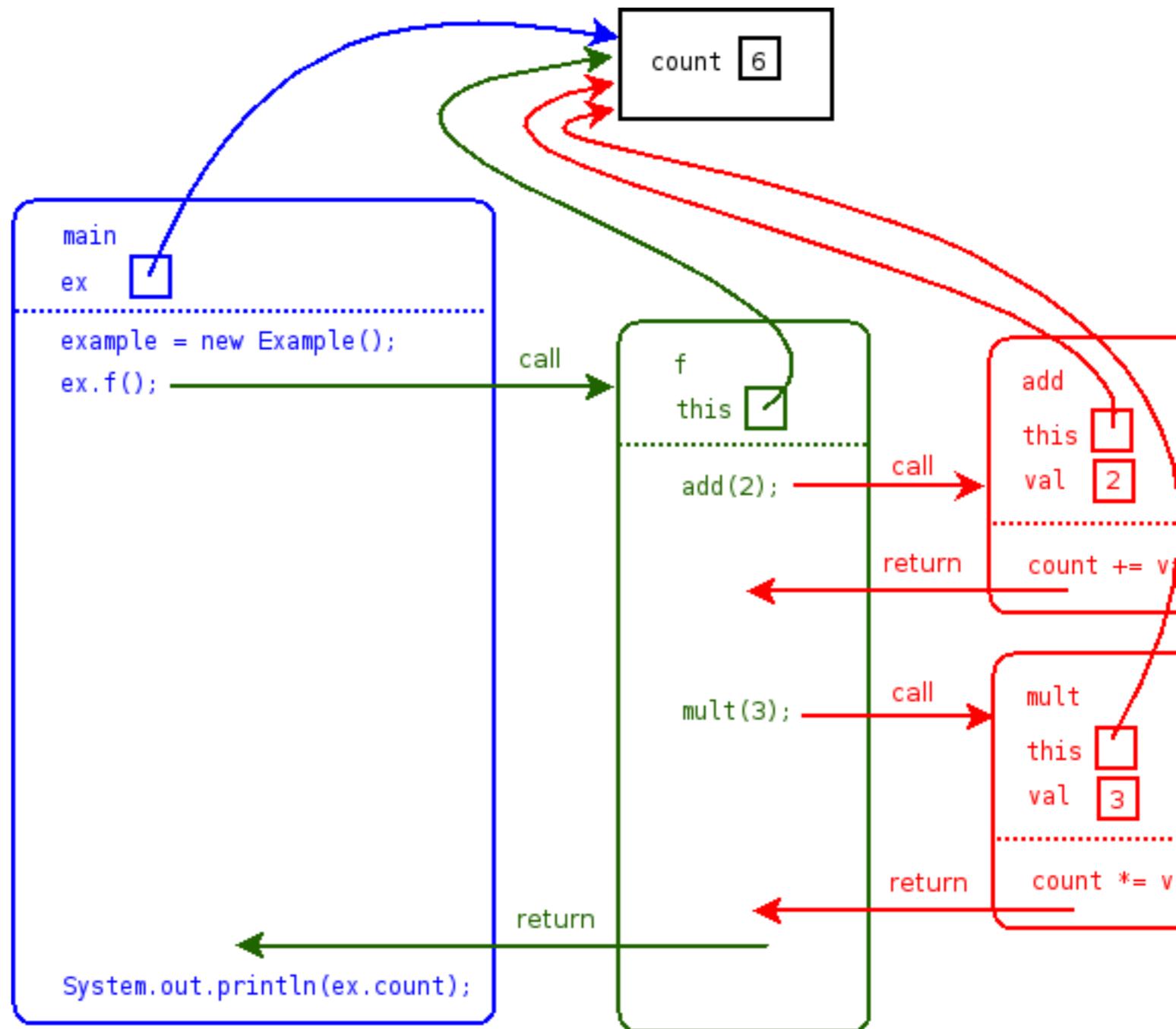
Aside: program flow and memory

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

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

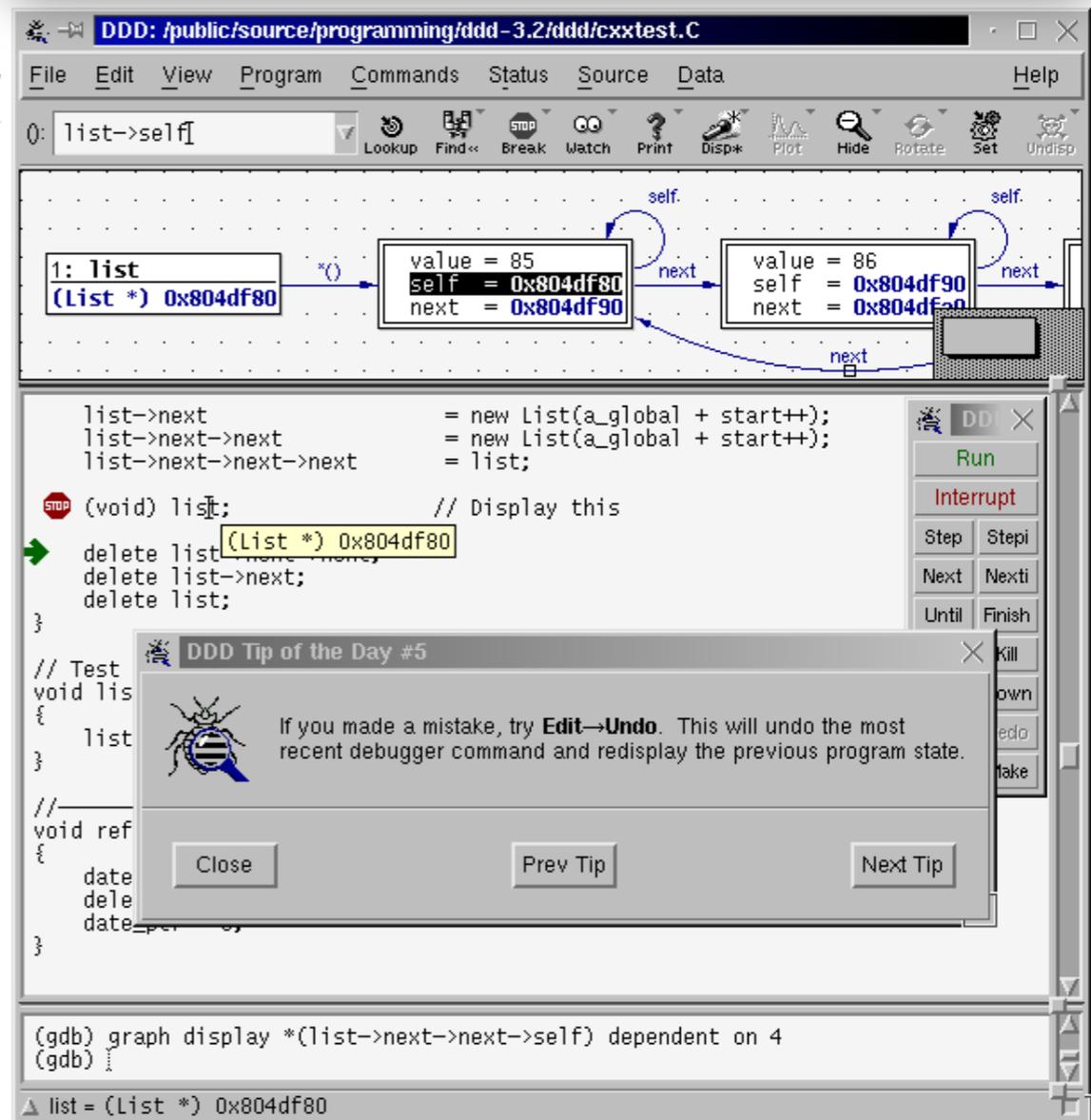
pdb

```
0.20000047, 0.34433303, 0.27002033, 0.30300303, 0.73203271,
[ 0.86932713, 0.74726936, 0.77972359, 0.88279606, 0.76825295,
0.39924089, 0.26050213, 0.82032474, 0.18800458, 0.43211861]],
'adc_sums': array([ 0.80428043, 0.81993334, 0.16511381, 0.93497246, 0.81474172,
0.32322294, 0.51430672, 0.24404024, 0.95566716, 0.52979194,
0.656204, 0.13846386, 0.38674983, 0.80887851, 0.21542999,
0.17744908, 0.19187673, 0.7651854, 0.66272061, 0.97808223,
0.09301636, 0.85309485, 0.38484974, 0.96316492, 0.75049923,
0.16777729, 0.75347307, 0.00606986, 0.36143674, 0.67134474,
0.32212175, 0.29453887, 0.02970078, 0.95121449, 0.63413519,
0.49721334, 0.72331239, 0.22943813, 0.61962722, 0.83813364,
0.55013944, 0.18937513, 0.85568434, 0.55420725, 0.08771667,
0.55564573, 0.8569015, 0.24182574, 0.35381984, 0.00141777]),
'num_samples': 10)
(Pdb) bt
/Users/kosack/anaconda/lib/python3.6/db.py(431)run()
-> exec(cmd, globals, locals)
<string>(1)<module>()
/Users/kosack/Projects/CTA/Working/ctape/ctape/io/tests/test_hdf5.py(77)<module>()
-> test_write_container("test.h5")
> /Users/kosack/Projects/CTA/Working/ctape/ctape/io/tests/test_hdf5.py(23)test_write_container()
-> r0tel.meta['test_attribute'] = 3.14159
(Pdb) 4=
```

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



3: device_name
0x27920 "fd1"

4: dev
0x27900

disk = 0x278d0
net = 0x0

name = 0x278b0 "fd1"
dev = 0x11860
total_sectors = 2880
has_partitions = 0
id = 1
partition = 0x0
read_hook = 0
data = 0x27880

name = 0x10c65 "biosdisk"
id = 0
iterate = 0xfa30 <grub_biosdisk_iterate>
open = 0xfad1 <grub_biosdisk_open>
close = 0xfc5f <grub_biosdisk_close>
read = 0xfe53 <grub_biosdisk_read>
write = 0xff19 <grub_biosdisk_write>
next = 0x0

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

```
grub_printf ("%12s", "DIR");
grub_printf ("%s%s\n", filename, dir ? "/"
return 0;
}
device_name = grub_file_get_device_name (dirname);
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)
```

```
(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)
```

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

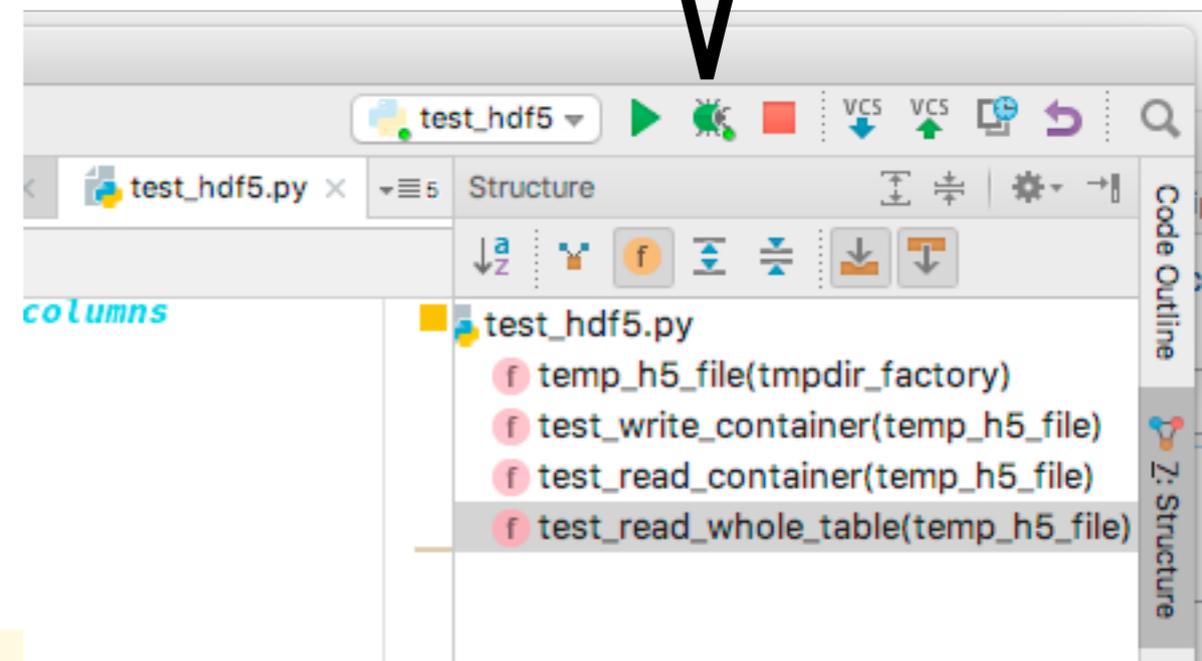
Click in margin to set a breakpoint

```
der.py
y

# 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")
    test_read_whole_table("test.h5")
```



GUI debugging

The image shows a Python IDE interface with a file explorer on the left, a code editor in the center, and a debugger at the bottom. The code editor displays Python code for reading HDF5 tables. A red dot on the left margin indicates the current execution point. A blue highlight is on the line `reader = SimpleHDF5TableReader(str(temp_h5_file))`. The debugger at the bottom shows the current stack frame with variables `mc` and `temp_h5_file`. Call stack entries are visible on the left of the debugger.

```
test.h5
test_eventfilereader.py
test_files.py
test_hdf5.py
test_hessio.py
test_serializer.py
#containers.py#
__init__.py
array.py
containers.py
eventfilereader.py
files.py
hdftableio.py
hessio.py
serializer.py
sources.py
toymodel.py
zfits.py
└─ plotting
└─ reco
└─ tests
└─ tools
└─ utils
  └─ tests
    └─ __init__.py
```

```
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):
    temp_h5_file: 'test.h5'

    mc = MCEventContainer()
    mc: {'alt': 0.0,\n 'az': 0.0,\n 'core_x': 0.0,\n 'core_y': 0.0,\n 'energy': 0.0,\n 'h_first_int': 0.0,\n 'tel': {}}

    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")
    test_read_container("test.h5")
    test_read_whole_table("test.h5")
```

values also appear right in the code!
(or on mouse-over)

currently at this line

Move up and down stack or lines

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

GUI debugging

Drill deep down into any data structure!

The image shows a GUI debugger interface with several components:

- File Explorer:** A tree view on the left showing a project structure with files like `test.h5`, `test_eventfilereader.py`, `test_files.py`, `test_hdf5.py`, `test_hessio.py`, `test_serializer.py`, `#containers.py#`, `__init__.py`, `array.py`, `containers.py`, `eventfilereader.py`, `files.py`, `hdftableio.py`, `hessio.py`, `serializer.py`, `sources.py`, `toymodel.py`, `zfits.py`, and subfolders like `plotting`, `reco`, `tests`, `tools`, and `utils`.
- Code Editor:** The central pane shows Python code with a red breakpoint at line 151. The code includes comments like `# read all 3 tables` and function definitions like `def test_read_whole...`. A variable `mc` is assigned to `MCEventContainer`.
- Variable Viewer:** A window titled `mc` displays a dictionary of variables:
 - `alt = {float} 0.0`
 - `attributes = {dict} {'energy': Monte-Carlo Energy [TeV], 'alt... View`
 - `__len__ = {int} 7`
 - `'alt' (4608341976) = {Item} Monte-carlo altitude [deg]`
 - `default = {float} 0.0`
 - `description = {str} 'Monte-carlo altitude'`
 - `unit = {Unit} deg`
 - `'az' (4415856064) = {Item} Monte-Carlo azimuth [deg]`
 - `'core_x' (4442303656) = {Item} MC core position [m]`
 - `'core_y' (4442303768) = {Item} MC core position [m]`
 - `'energy' (4442303544) = {Item} Monte-Carlo Energy [TeV]`
 - `'h_first_int' (4608895856) = {Item} Height of first interaction`
 - `'tel' (4415607728) = {Item} map of tel_id to MCCameraEventCo`
 - `az = {float} 0.0`
 - `core_x = {float} 0.0`
 - `core_y = {float} 0.0`** (highlighted in green)
 - `energy = {float} 0.0`
 - `h first int = {float} 0.0`
- Debugger:** The bottom pane shows the `Frames` and `Variables` sections. The `Frames` section shows the current stack frame `test_read_whole_tabl` and the `Variables` section shows `mc = {MCEventContainer} {'alt': 0.0, 'az': 0.0, 'core_x': 0.0, 'core_y': 0.0, 'energy': 0.0, 'h_first_int': 0.0, 'tel': {}}` and `temp_h5_file = {str} 'test.h5'`.

*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

The image shows a Python IDE interface with a file explorer on the left, a code editor in the center, and a debugger at the bottom. The code editor displays Python code for reading HDF5 tables. A red dot on the left margin indicates a breakpoint at the line `reader = SimpleHDF5TableReader(str(temp_h5_file))`. The debugger at the bottom shows the current stack frame with variables `mc` and `temp_h5_file`. Callout boxes provide instructions on how to navigate the debugger.

```
test.h5
test_eventfilereader.py
test_files.py
test_hdf5.py
test_hessio.py
test_serializer.py
#containers.py#
__init__.py
array.py
containers.py
eventfilereader.py
files.py
hdftableio.py
hessio.py
serializer.py
sources.py
toymodel.py
zfits.py
└─ plotting
└─ reco
└─ tests
└─ tools
└─ utils
  └─ tests
    └─ __init__.py
```

```
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):
    temp_h5_file: 'test.h5'

    mc = MCEventContainer()
    mc: {'alt': 0.0,\n 'az': 0.0,\n 'core_x': 0.0,\n 'core_y': 0.0,\n 'energy': 0.0,\n 'h_first_int': 0.0,\n 'tel': {}}

    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")
    test_read_container("test.h5")
    test_read_whole_table("test.h5")
```

values also appear right in the code!
(or on mouse-over)

currently at this line

Move up and down stack or lines

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

GUI debugging

```
test.h5
test_eventfilereader.py
test_files.py
test_hdf5.py
test_hessio.py
test_serializer.py
#containers.py#
__init__.py
array.py
containers.py
eventfilereader.py
files.py
hdftableio.py
hessio.py
serializer.py
sources.py
toymodel.py
zfits.py
└─ plotting
└─ reco
└─ tests
└─ tools
└─ utils
  └─ tests
    └─ __init__.py
```

```
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")
    test_read_container("test.h5")
    test_read_whole_table("test.h5")
```

Debug: test_hdf5 test_hdf5 test_hdf5

Debugger Console

Frames

- Main...
- test_read_whole_tabl
- <module>, test_hdf5
- execfile, _pydev_exec
- run, pydevd.py:1015
- <module>, pydevd.py

Variables

- mc = {MCEventContainer} {'alt': 0.0, 'az': 0.0, 'core_x': 0.0, 'core_y': 0.0, 'er
- temp_h5_file = {str} 'test.h5'

Move up and down stack or lines

You can see stack frame

use the "data view" to see values of large arrays or tables

Data View

r0tel.adc_samples x +

0	1	2	3	4
0.95997	0.98010	0.74854	0.60060	0.5954
0.68207	0.45175	0.83795	0.76688	0.6887
0.85410	0.58842	0.51559	0.36246	0.2527
0.87389	0.83798	0.14105	0.93956	0.6563
0.68928	0.53708	0.77192	0.49141	0.6709
0.38935	0.57417	0.94631	0.77080	0.4029
0.66854	0.59730	0.69974	0.93130	0.0659
0.88826	0.97069	0.04254	0.91542	0.2782
0.94109	0.56698	0.51974	0.43029	0.0505
0.90637	0.17494	0.22052	0.13475	0.4355
0.50643	0.57509	0.55480	0.49568	0.7677
0.30948	0.89409	0.15910	0.67037	0.5786
0.49066	0.41402	0.44546	0.39157	0.5963
0.95341	0.73043	0.94395	0.80189	0.2411
0.14115	0.56538	0.22046	0.22565	0.8083
0.10341	0.25694	0.95972	0.46487	0.8901
0.02162	0.65008	0.87262	0.64492	0.4582
0.70528	0.34887	0.34042	0.64684	0.3112
0.92931	0.16970	0.42819	0.47133	0.7995
0.35228	0.76336	0.39992	0.32342	0.4949
0.53163	0.72559	0.12517	0.94481	0.9549
0.20995	0.52962	0.45084	0.01140	0.1925
0.55729	0.30726	0.07956	0.75938	0.2516
0.10078	0.98490	0.34197	0.90848	0.3455
0.76712	0.46013	0.02517	0.73148	0.0315
0.20437	0.46705	0.29971	0.79643	0.8670
0.90153	0.14359	0.22539	0.23854	0.1023
0.91993	0.21435	0.75078	0.77390	0.7973
0.05615	0.96193	0.20847	0.81645	0.9192
0.01301	0.75174	0.94013	0.14905	0.8649
0.88294	0.61006	0.13029	0.88178	0.8632
0.57943	0.18664	0.32796	0.77201	0.9587
0.63643	0.94599	0.09075	0.89204	0.2995
0.07583	0.18816	0.12187	0.15590	0.0479
0.26883	0.63786	0.81847	0.48363	0.2874

r0tel.adc_samples Format: %5f

- DEMO -