Debugging & Profiling Scientific Code

Karl Kosack
CEA Paris-Saclay

ESCAPE School, June 2022

A bit about me...

https://www.mpi-hd.mpg.de/hfm/HESS/
https://www.cta-observatory.org/
https://github.org/cta-observatory/ctapipe

H.E.S.S. (Namibia)

Astrophysicist at ***CEA Paris-Saclay*** (Astrophysics Department)

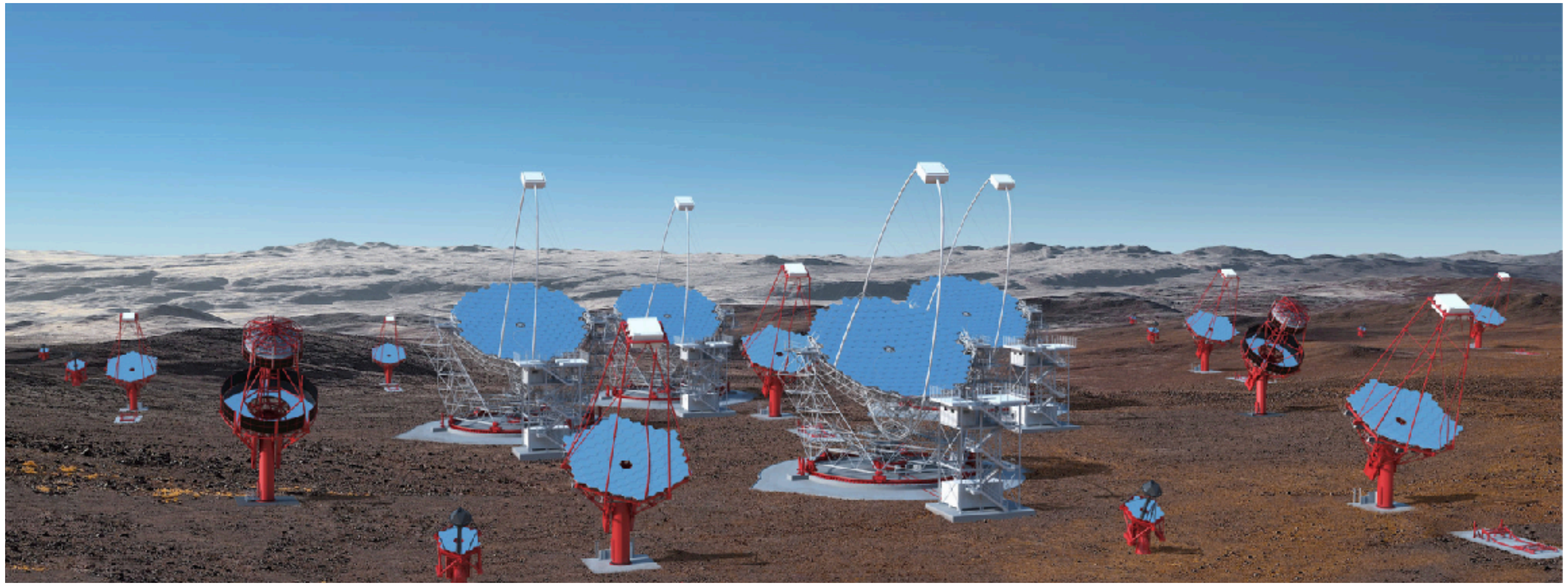
- High energy gamma rays, sources of cosmic ray acceleration
- **HESS** and **CTA** Atmospheric Cherenkov Telescope consortia
- Coordinator of *Data Processing and Preservation* for **CTA Observatory** (60% of time)
- creator and developer of **ctapipe** software for IACT low-level analysis pipeline

Other Background ***(apart from gamma-ray astro)*:**

- Computational Physics
- Data analysis, processing, statistics
- Lots of scientific software development over the years...
- Was a hard-core C/C++/Perl (!) user, now **essentially 100% python for 10+ years**!

A bit about me...

https://www.mpi-hd.mpg.de/hfm/HESS/
https://www.cta-observatory.org/
https://github.org/cta-observatory/ctapipe



Cherenkov Telescope Array - *(Canary Islands + Chile) - artist's conception*

Astrophysicist at ***CEA Paris-Saclay*** (Astrophysics Department)

- High energy gamma rays, sources of cosmic ray acceleration
- **HESS** and **CTA** Atmospheric Cherenkov Telescope consortia
- Coordinator of *Data Processing and Preservation* for **CTA Observatory** (60% of time)
- creator and developer of **ctapipe** software for IACT low-level analysis pipeline

Other Background ***(apart from gamma-ray astro)*:**

- Computational Physics
- Data analysis, processing, statistics
- Lots of scientific software development over the years...
- Was a hard-core C/C++/Perl (!) user, now **essentially 100% python for 10+ years**!

Some good advice for writing Scientific Code

Get the Code to **Work** First: (test and **Debug**)

Some good advice for writing Scientific Code

Get the Code to **Work** First:
(test and **Debug**)

Write ***tests*** to ensure it does!

Some good advice for writing Scientific Code

Get the Code to **Work** First:
(test and **Debug**)

↓

Profile to find bottlenecks

Write ***tests*** to ensure it does!

Some good advice for writing Scientific Code

Get the Code to **Work** First:
(test and **Debug**)

↓

Profile to find bottlenecks

↓

Optimize *only* what needs to be!

Write ***tests*** to ensure it does!

Some good advice for writing Scientific Code

Get the Code to **Work** First:
(test and **Debug**)

↓

Profile to find bottlenecks

↓

Optimize *only* what needs to be!

Write ***tests*** to ensure it does!

run **tests** to check

Some good advice for writing Scientific Code

Get the Code to **Work** First: (test and **Debug**)

↓

Profile to find bottlenecks

↓

Optimize *only* what needs to be!

↓

Refactor if necessary (redesign/rewrite)

Write ***tests*** to ensure it does!

run **tests** to check

Some good advice for writing Scientific Code

Get the Code to **Work** First:
(test and **Debug**)

Write ***tests*** to ensure it does!

Profile to find bottlenecks

Optimize *only* what needs to be!

run **tests** to check

Refactor if necessary
(redesign/rewrite)

run **tests** to check

Some good advice for writing Scientific Code

Get the Code to **Work** First: (test and **Debug**)

Profile to find bottlenecks

Optimize *only* what needs to be!

Refactor if necessary (redesign/rewrite)

Write ***tests*** to ensure it does!

run **tests** to check

run **tests** to check

Debugging:

- What happens when a program runs?
- What is a debugger?
- How do you use a debugger?
  - ➤ command-line
  - ➤ GUI
  - ➤ in a notebook

Profiling:

- Why profile your code?
- How to profile:
  - ➤ Using timing loops
  - ➤ Function Call Profiling with cProfile
  - ➤ Memory Profiling with memprof
  - ➤ Line profiling with lineprof

Debugging

ESCAPE School, June 2022

ESCAPE
European Science Cluster of Astronomy & Particle physics ESFRI research infrastructures

What is your current approach?

What is your current approach?

When you run a piece of code and:

What is your current approach?

When you run a piece of code and:

- get an error/crash/exception

What is your current approach?

When you run a piece of code and:

- get an error/crash/exception
- encounter an unexpected result

What is your current approach?

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 "under the hood"

What is your current approach?

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 "under the hood"

What do you usually do?

What is your current approach?

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 "under the hood"

What do you usually do?

Do you: (show of hands)

What is your current approach?

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 "under the hood"

What do you usually do?

Do you: (show of hands)

- Add a bunch of ***print* statements** and try to track down the issue?

What is your current approach?

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 "under the hood"

What do you usually do?

Do you: (show of hands)

- Add a bunch of ***print* statements** and try to track down the issue?
- Use an **interactive python interpreter**

What is your current approach?

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 "under the hood"

What do you usually do?

Do you: (show of hands)

- Add a bunch of ***print* statements** and try to track down the issue?
- Use an **interactive python interpreter**
- Use a **Jupyter notebook**?

What is your current approach?

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 "under the hood"

What do you usually do?

Do you: (show of hands)

- Add a bunch of ***print* statements** and try to track down the issue?
- Use an **interactive python interpreter**
- Use a **Jupyter notebook**?
- Write a set of **unit tests**?

What is your current approach?

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 "under the hood"

What do you usually do?

Do you: (show of hands)

- Add a bunch of ***print* statements** and try to track down the issue?
- Use an **interactive python interpreter**
- Use a **Jupyter notebook**?
- Write a set of **unit tests**?
- Run the code in a **debugger**?

What is your current approach?

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 "under the hood"

What do you usually do?

Do you: (show of hands)

- Add a bunch of ***print* statements** and try to track down the issue?
- Use an **interactive python interpreter**
- Use a **Jupyter notebook**?
- Write a set of **unit tests**?
- Run the code in a **debugger**?

First: how do programs run?

Our program

```
def function_b(n):
    x = 3.3
    return sin(n * x * RAD_TO_DEG)

def function_a(n):
    return n * function_b(n + 1)

if __name__ == "__main__":
    RAD_TO_DEG = 180.0/np.pi
    for ii in range(10):
        function_a(ii)
```

The Call Stack

Global Memory

Local Memory

First: how do programs run?

Our program

```python
def function_b(n):
    x = 3.3
    return sin(n * x * RAD_TO_DEG)

def function_a(n):
    return n * function_b(n + 1)

if __name__ == "__main__":
    RAD_TO_DEG = 180.0/np.pi
    for ii in range(10):
        function_a(ii)
```

we are here

The Call Stack

main program

Global Memory

RAD_TO_DEG = 57.29

Local Memory

First: how do programs run?

Our program

```
def function_b(n):
    x = 3.3
    return sin(n * x * RAD_TO_DEG)

def function_a(n):
    return n * function_b(n + 1)

if __name__ == "__main__":
    RAD_TO_DEG = 180.0/np.pi
    for ii in range(10):
        function_a(ii)
```

we are here

The Call Stack

main program

Global Memory

```
RAD_TO_DEG = 57.29
ii = 0
```

Local Memory

First: how do programs run?

Our program

```
def function_b(n):
    x = 3.3
    return sin(n * x * RAD_TO_DEG)

def function_a(n):
    return n * function_b(n + 1)

if __name__ == "__main__":
    RAD_TO_DEG = 180.0/np.pi
    for ii in range(10):
        function_a(ii)
```

we are here

The Call Stack

main program

Global Memory

```
RAD_TO_DEG = 57.29
ii = 0
```

Local Memory

First: how do programs run?

Our program

we are here →

```
def function_b(n):
    x = 3.3
    return sin(n * x * RAD_TO_DEG)

def function_a(n):
    return n * function_b(n + 1)

if __name__ == "__main__":
    RAD_TO_DEG = 180.0/np.pi
    for ii in range(10):
        function_a(ii)
```

The Call Stack

function_a
main program

Global Memory

```
RAD_TO_DEG = 57.29
ii = 0
```

Local Memory

```
n = 0
```

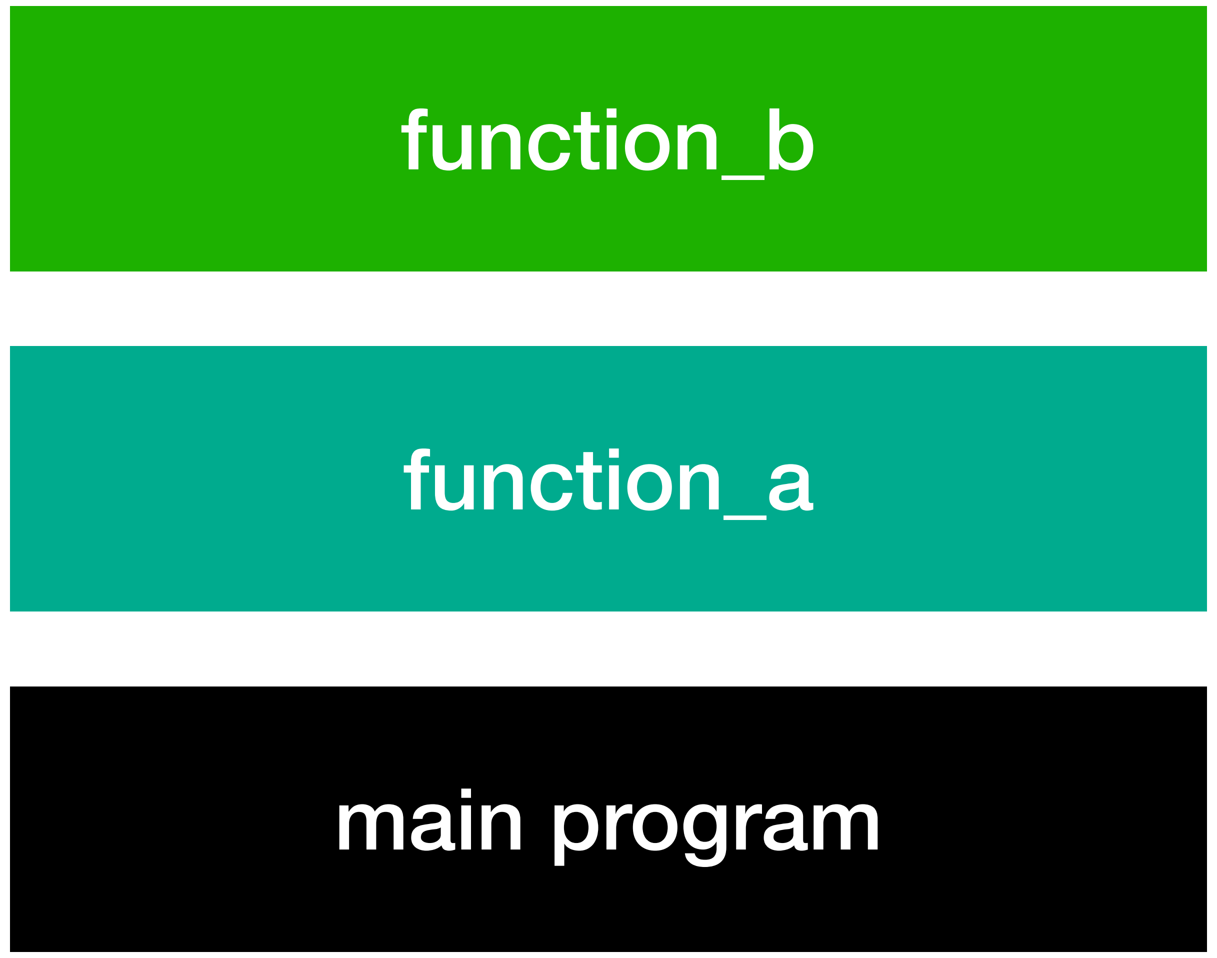
First: how do programs run?

Our program

we are here →

```python
def function_b(n):
    x = 3.3
    return sin(n * x * RAD_TO_DEG)

def function_a(n):
    return n * function_b(n + 1)

if __name__ == "__main__":
    RAD_TO_DEG = 180.0/np.pi
    for ii in range(10):
        function_a(ii)
```

The Call Stack



Global Memory

```
RAD_TO_DEG = 57.29
ii = 0
```

Local Memory

```
n = 1
x = 3.3
```

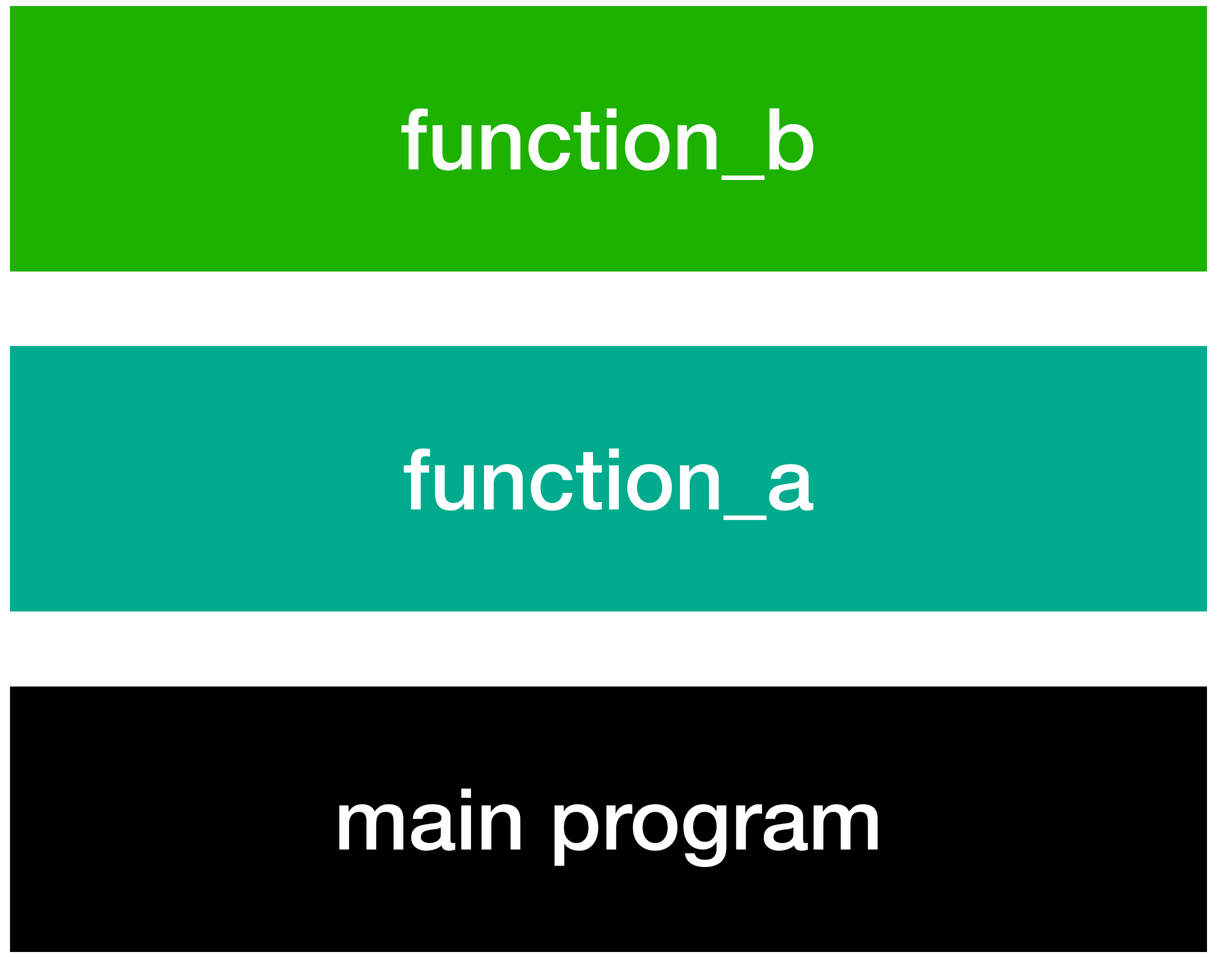
First: how do programs run?

Our program

we are here →

```python
def function_b(n):
    x = 3.3
    return sin(n * x * RAD_TO_DEG)

def function_a(n):
    return n * function_b(n + 1)

if __name__ == "__main__":
    RAD_TO_DEG = 180.0/np.pi
    for ii in range(10):
        function_a(ii)
```

The Call Stack



Global Memory

```
RAD_TO_DEG = 57.29
ii = 0
```

Local Memory

```
n = 1
x = 3.3
```

First: how do programs run?

Our program

we are here →

```python
def function_b(n):
    x = 3.3
    return sin(n * x * RAD_TO_DEG)

def function_a(n):
    return n * function_b(n + 1)

if __name__ == "__main__":
    RAD_TO_DEG = 180.0/np.pi
    for ii in range(10):
        function_a(ii)
```

The Call Stack

function_a

main program

Global Memory

```
RAD_TO_DEG = 57.29
ii = 0
```

Local Memory

```
n = 0
```

First: how do programs run?

Our program

```
def function_b(n):
    x = 3.3
    return sin(n * x * RAD_TO_DEG)

def function_a(n):
    return n * function_b(n + 1)

if __name__ == "__main__":
    RAD_TO_DEG = 180.0/np.pi
    for ii in range(10):
        function_a(ii)
```

we are here

The Call Stack

main program

Global Memory

```
RAD_TO_DEG = 57.29
ii = 0
```

Local Memory

First: how do programs run?

Our program

```python
def function_b(n):
    x = 3.3
    return sin(n * x * RAD_TO_DEG)

def function_a(n):
    return n * function_b(n + 1)

if __name__ == "__main__":
    RAD_TO_DEG = 180.0/np.pi
    for ii in range(10):
        function_a(ii)
```

we are here

The Call Stack

main program

Global Memory

```
RAD_TO_DEG = 57.29
ii = 1
```

Local Memory

Program flow and memory in e.g. C(++)

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)

python's memory scoping and stack is at a higher level of abstraction than this, but conceptually is pretty similar

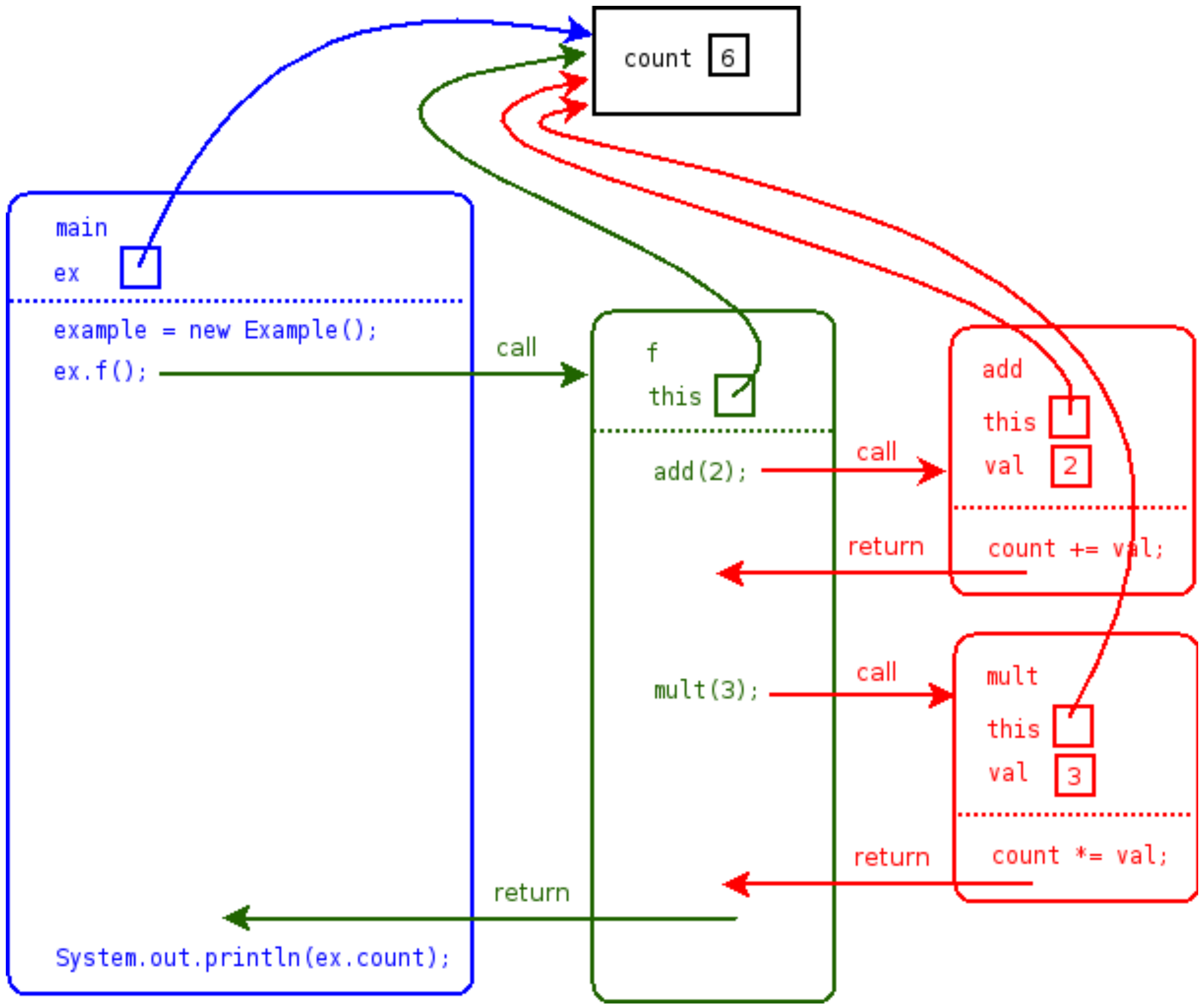


diagram from: http://faculty.ycp.edu/~dhovemey/spring2007/cs201/info/exceptionsFileIO.html

Program flow and memory in e.g. C(++)

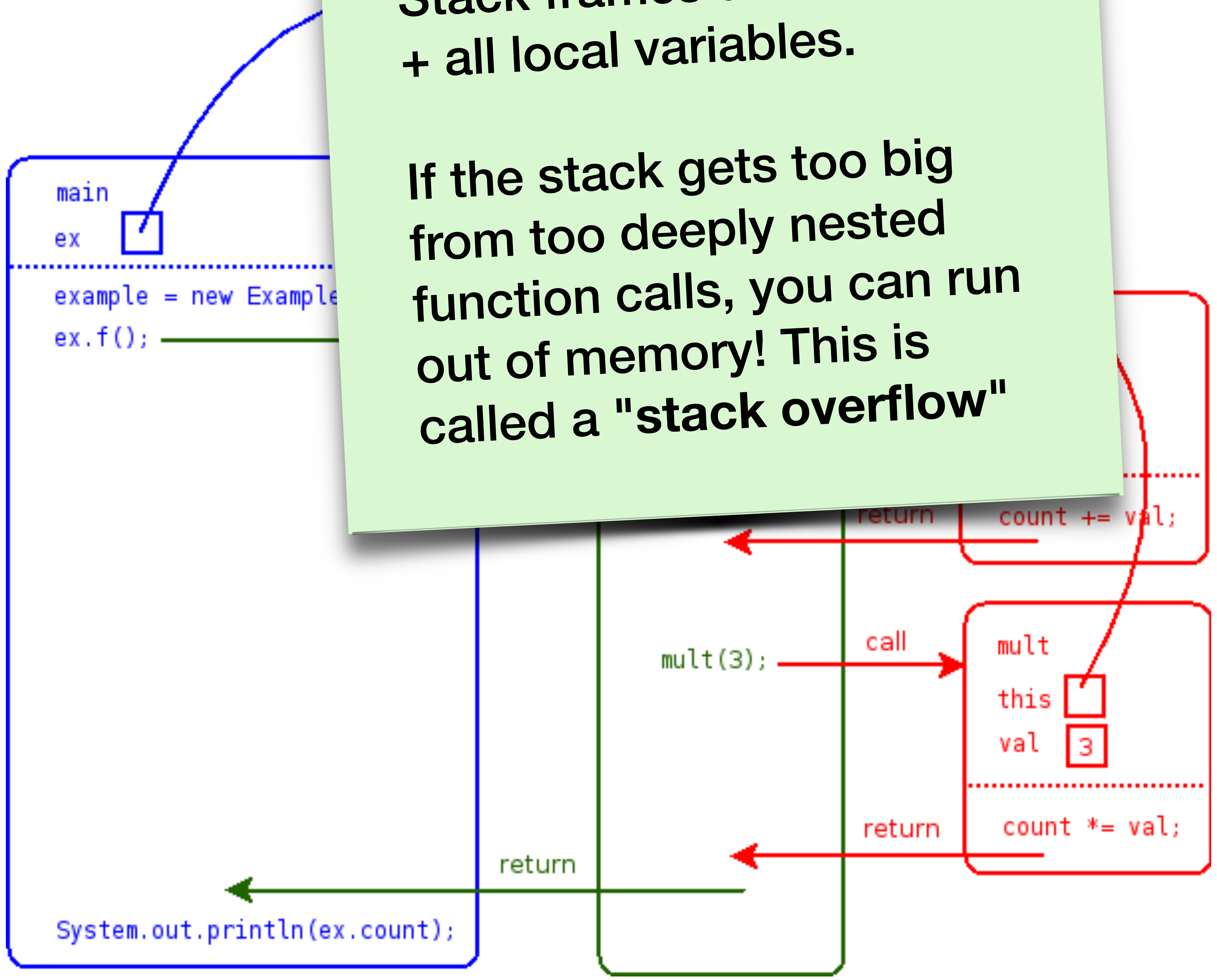
Heap:

- 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)

python's memory scoping and stack is at a higher level of abstraction than this, but conceptually is pretty similar



Stack frames use memory + all local variables.

If the stack gets too big from too deeply nested function calls, you can run out of memory! This is called a "**stack overflow**"

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

Program flow and memory in e.g. C(++)

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)

python's memory scoping and stack is at a higher level of abstraction than this, but conceptually is pretty similar

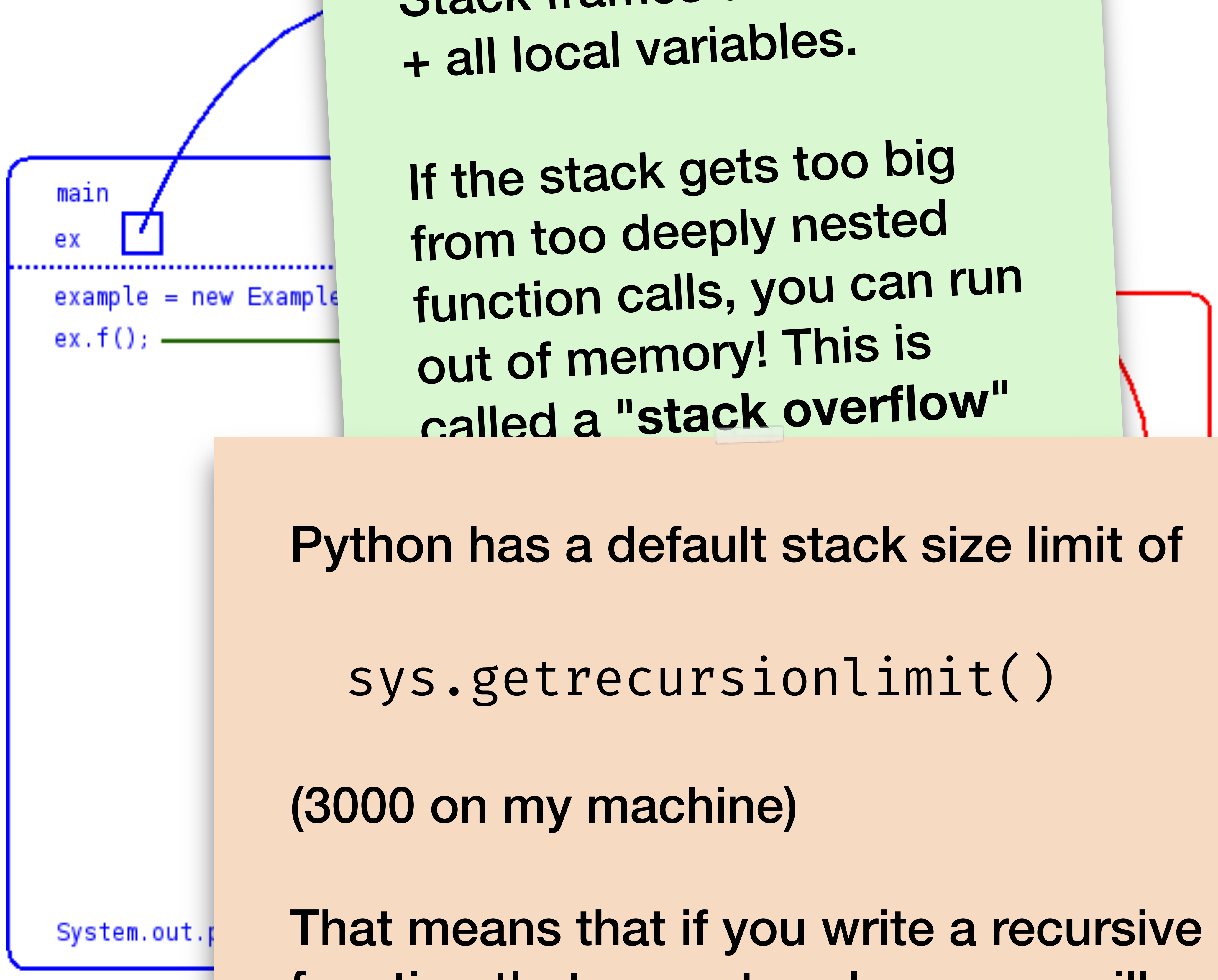


diagram from: http://facult

Stack frames use memory + all local variables.

If the stack gets too big from too deeply nested function calls, you can run out of memory! This is called a "**stack overflow**"

Python has a default stack size limit of

```
sys.getrecursionlimit()
```

(3000 on my machine)

That means that if you write a recursive function that goes too deep, you will hit this limit. It throws a **RecursionError** in that case

What is a debugger?

A debugger:

- **runs** or **attaches** to a *running* piece of code or a program or one 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 (*up and down the call stack*)
  - ➤ watch a variable for change
  - ➤ keep running until a condition occurs

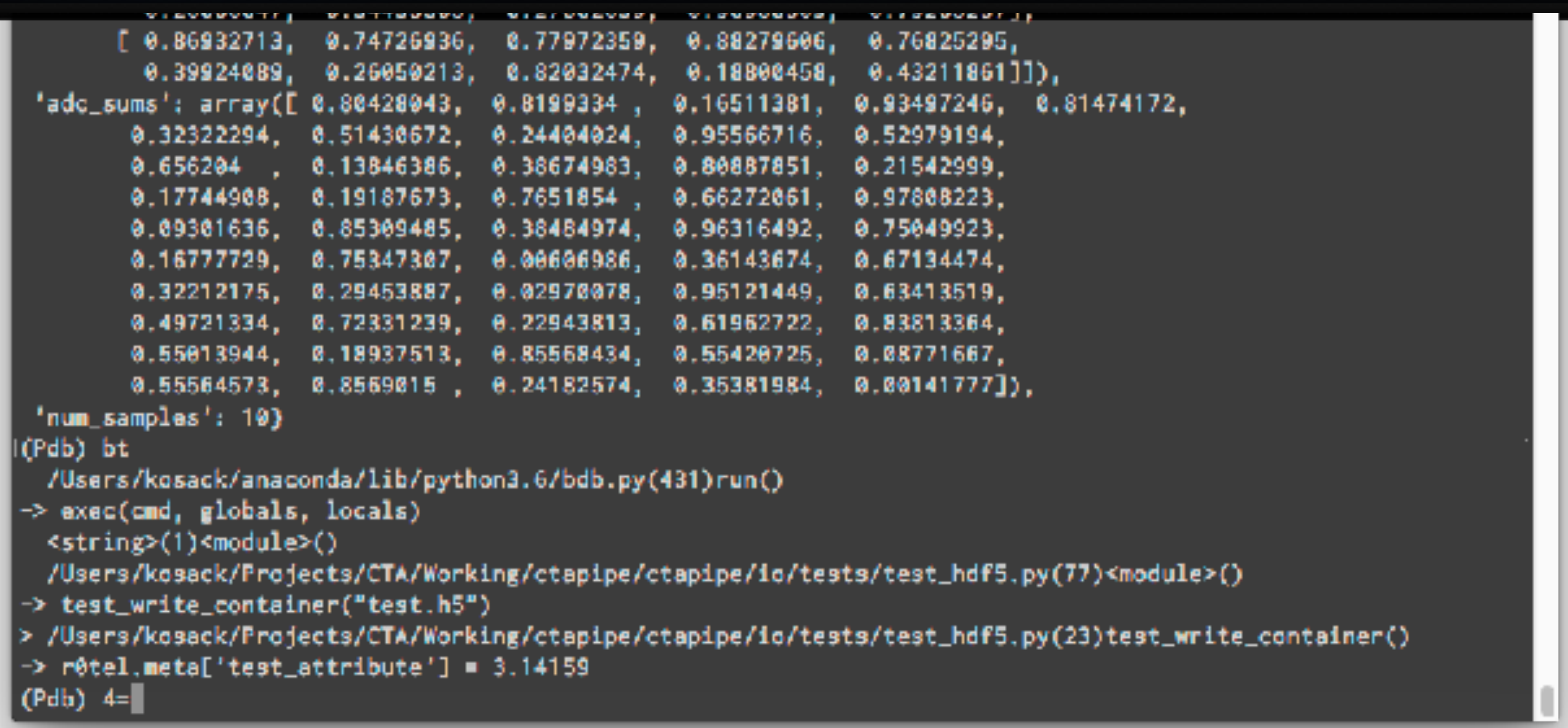
The basic use/concepts of debuggers is independent of language (a C++ debugger works the same as a python debugger)

Two levels of debugging interface

Text-mode debuggers:

- command menu interface
- good for quick debugging

pdb

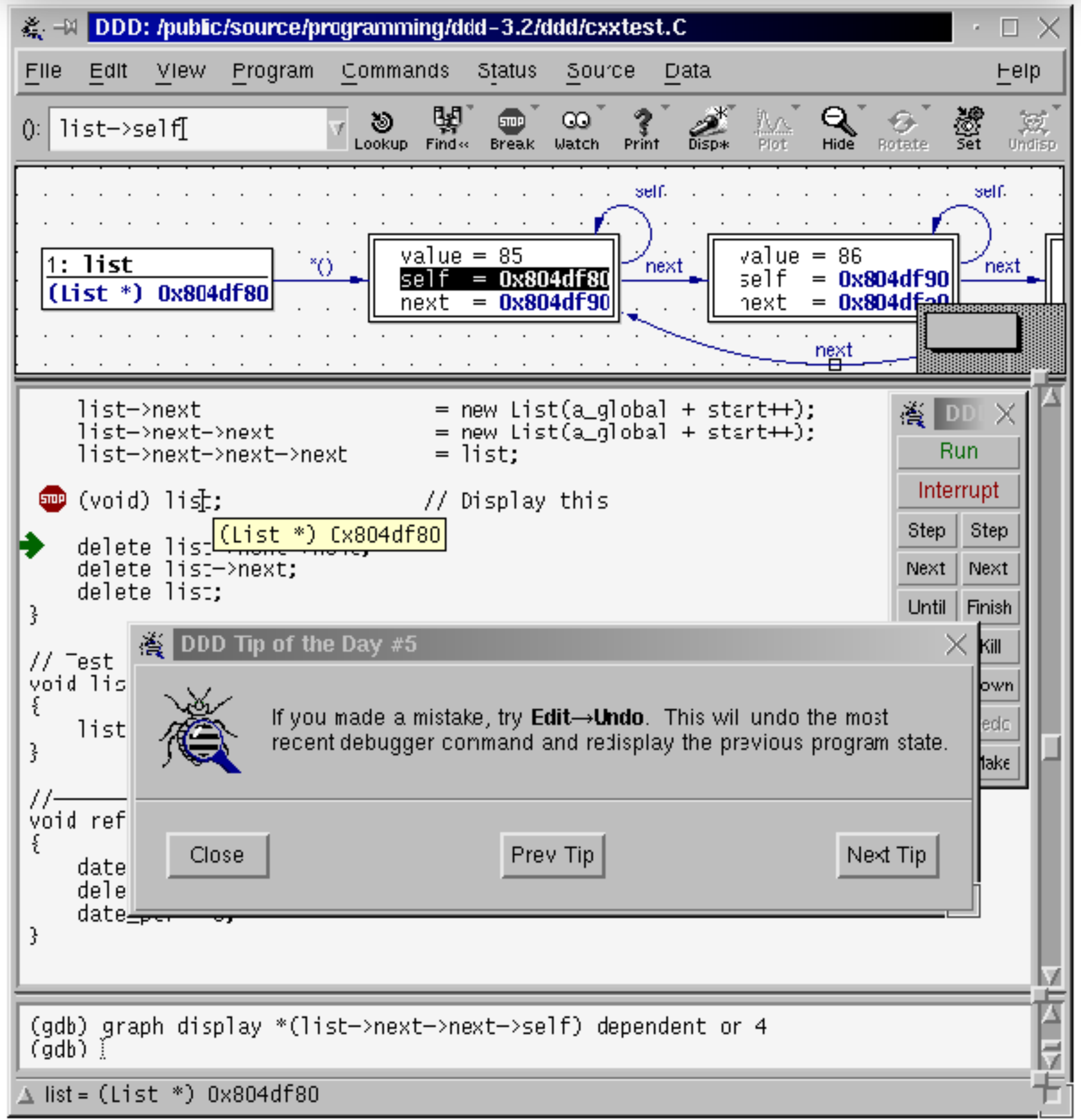


- ➤ pdb (Python debugger)
- ➤ ipydb (iPython debugger)
- ➤ gdb (GNU debugger, C/C++)

GUI Debuggers:

- often integrated with interactive development environments (IDEs)
- Allow point-and-click inspection of code and variables
- visual inspection of data

GNU ddd



- ➤ GNU **ddd** [Data Display Debugger] (c/c++)
- ➤ **PyCharm's debugger** (python)
- ➤ **VSCode's debugger** (multiple languages)
- ➤ Emacs **dap-mode** (multiple languages)

Use case 1: Your code "crashed"

My recommendation: **start with the *IPython* debugger!**

- Run your code in **ipython** *not python* to get it to work...
  - ➤ Make sure you run in INTERACTIVE python mode (-i)
  - ➤ Make sure you run with an INTERACTIVE GUI as well!

```
ipython -i --matplotlib=auto my-script.py
```

- When the exception is thrown (a bug!),
  - ➤ all you need to do is type: **%debug** at the ipython prompt and it will take you to the python debugger!

Then what?

common text-mode debugger commands (PDB or 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
- **?** to show help!

Then, once inside the debugger:

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:

```
breakpoint()      # for python version 3.7 and above
```

then run python as usual (e.g. `python myscript.py`)

- 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 -

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:

```
breakpoint()      # for python versi
```

then run python as usual (e.g. `python my`

- More work, but more flexible: ru

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

  - ➤ the script will not run, but rathe for you to type commands
  - ➤ use *next, step, cont* to step th
  - ➤ set a breakpoint! (*break* <linenumber>) and *continue* to it!

TIP: You can control which debugger is used by setting the environment variable *PYTHONBREAKPOINT*
(the default is pdb, the built-in python debugger

I prefer IPython's debugger, ipdb:

```
% mamba install ipdb
% export PYTHONBREAKPOINT=ipdb.set_trace
% python my_script_to_debug.py
```

- DEMO -

Debugging Unit Tests

Another common problem: what to do when a unit test fails?

- You can automatically enter the debugger automatically **when a test fails**:

pytest **--pdb**

- **Or even if it doesn't fail:** start pdb for every tested function**:**

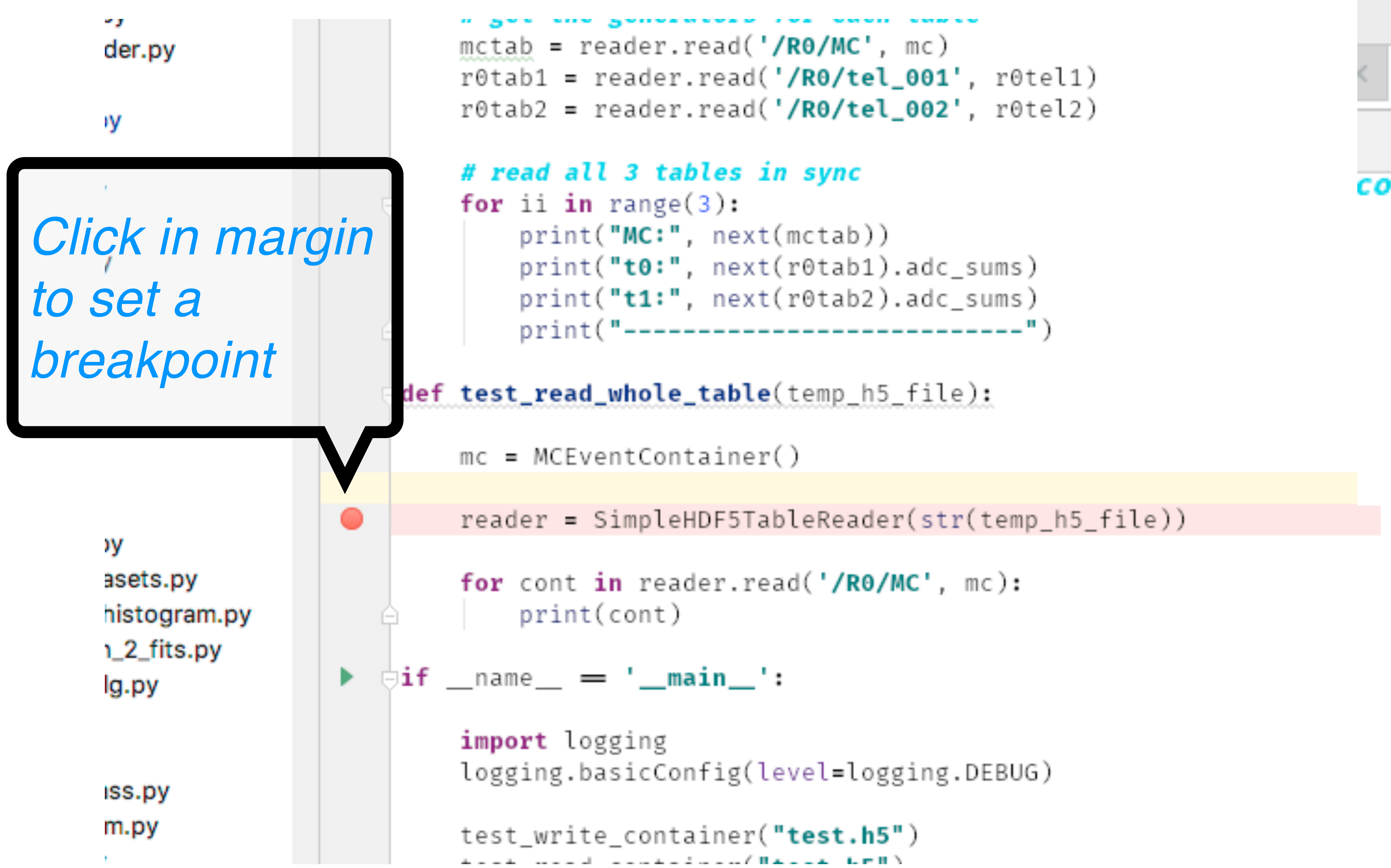
pytest **--trace**

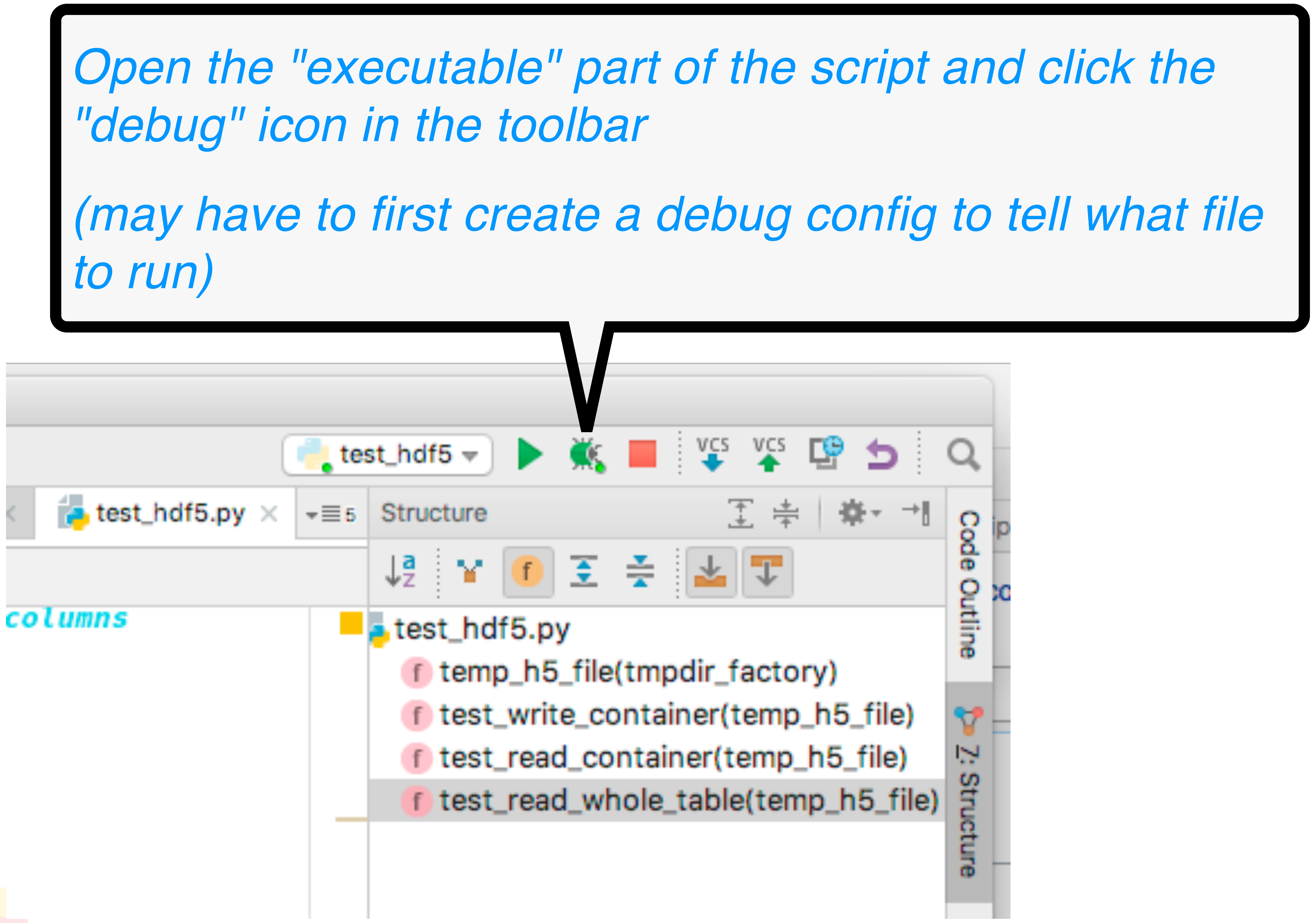
- And of course `breakpoint()` still works

GUI Debugging

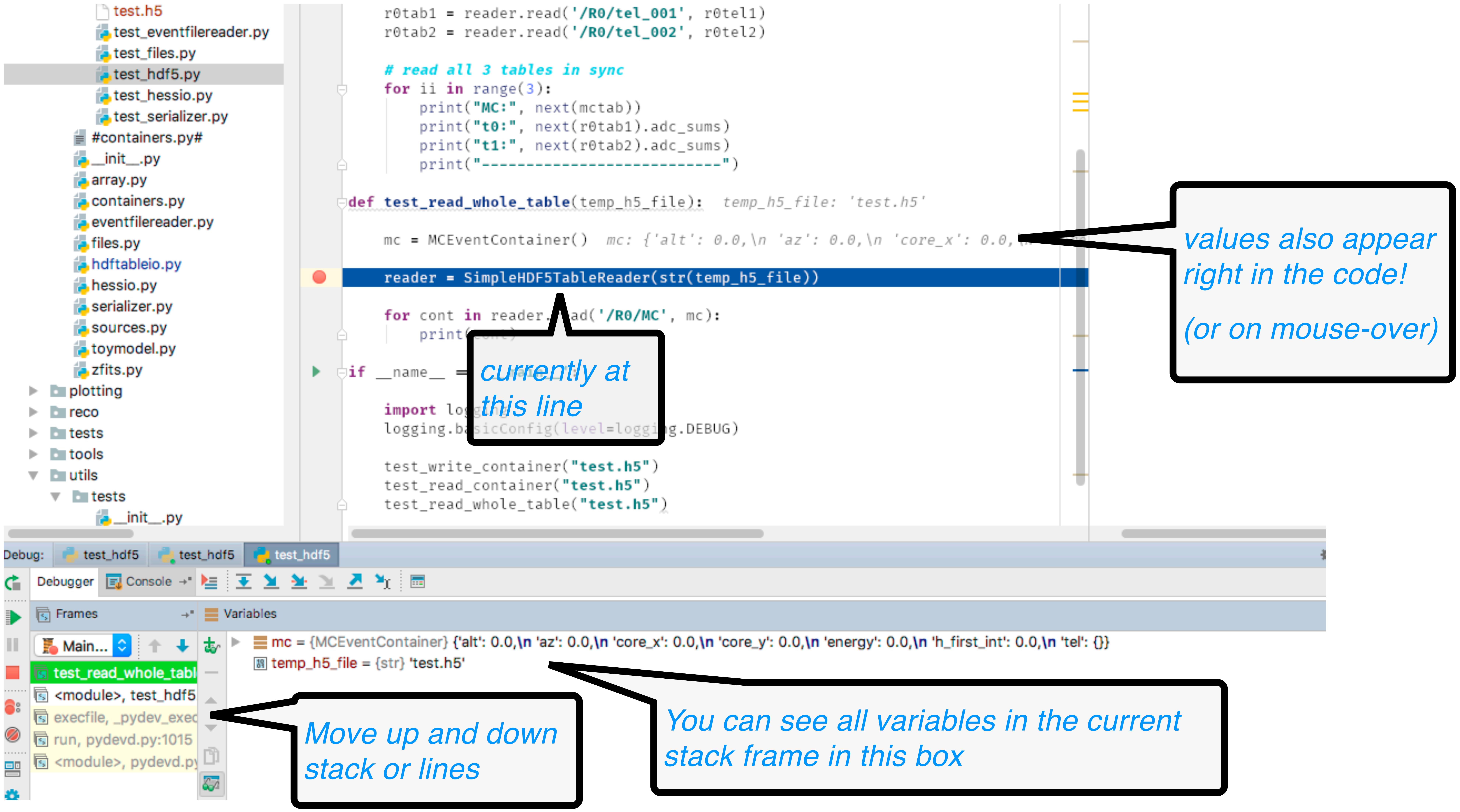
This is all nice and good, but it gets tedious for more than simple debugging…

Solution: use a GUI debugger!

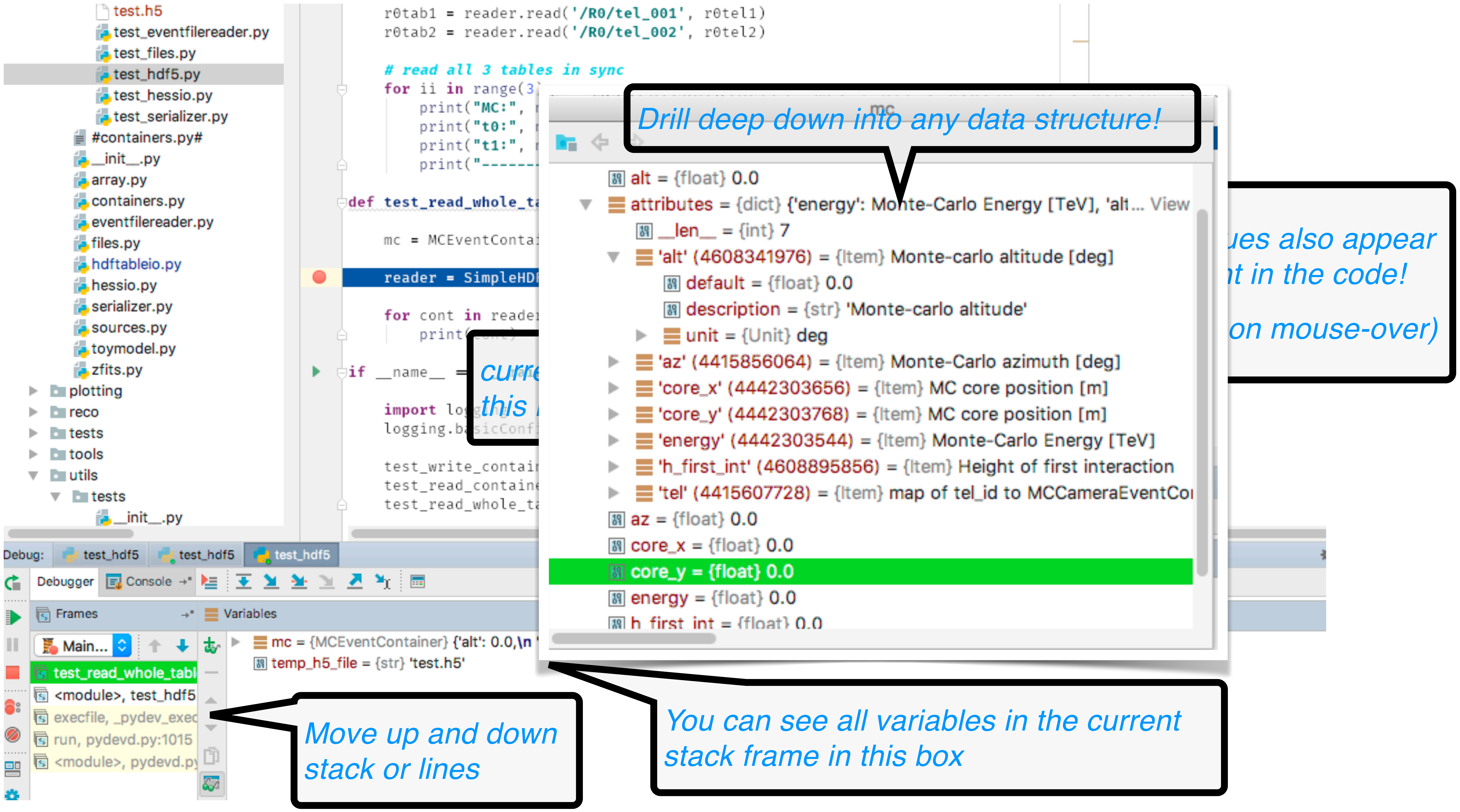




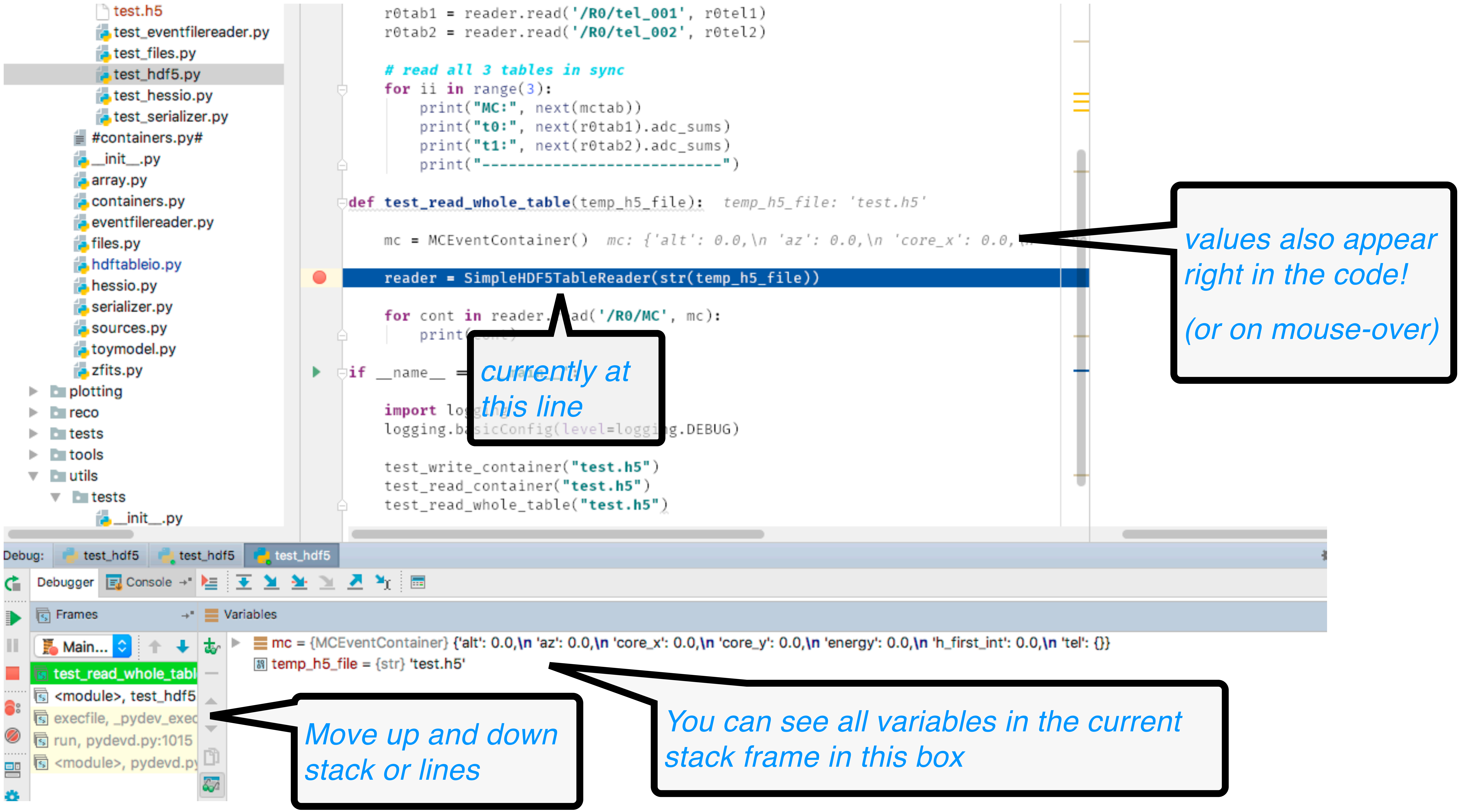
GUI debugging



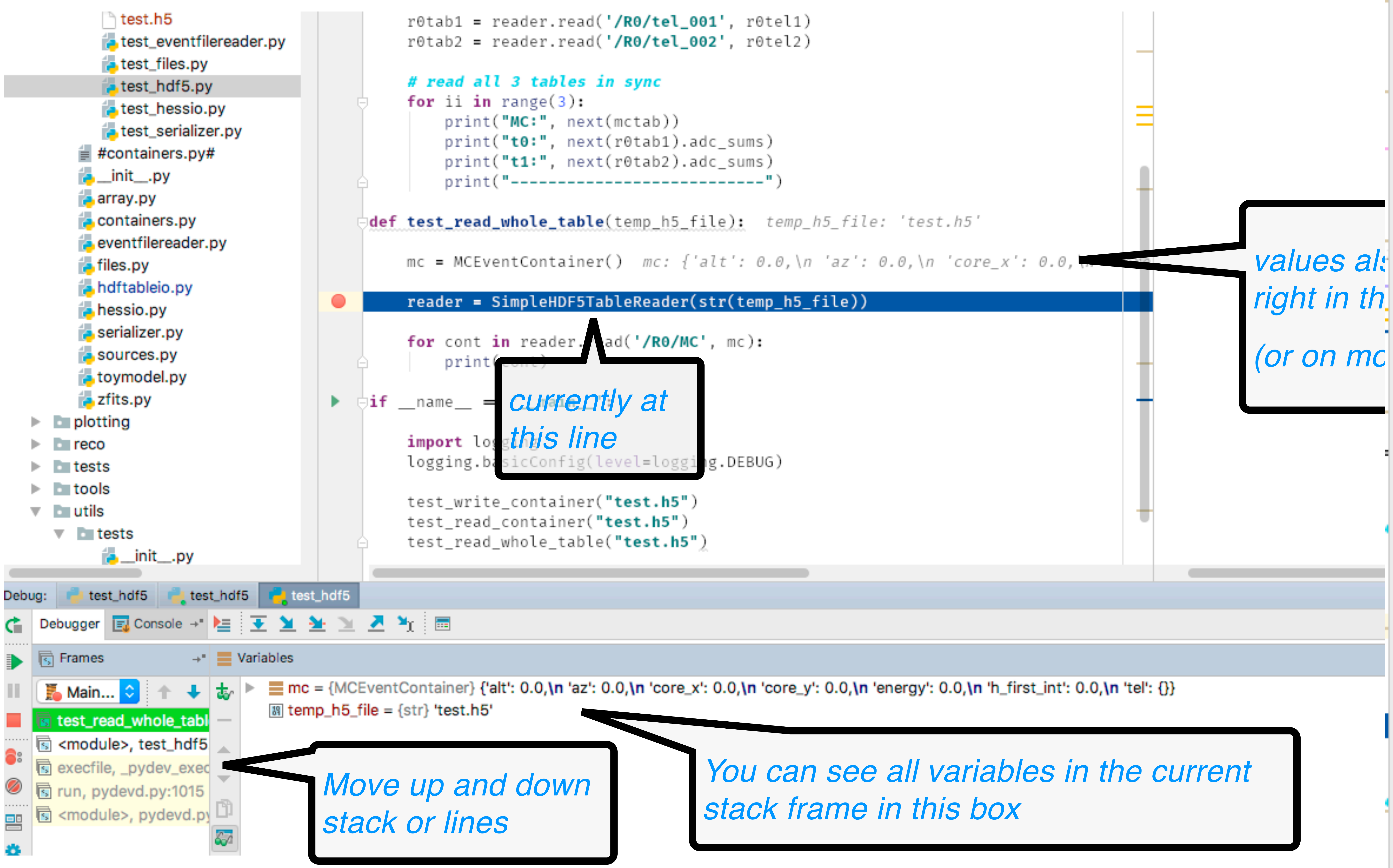
GUI debugging

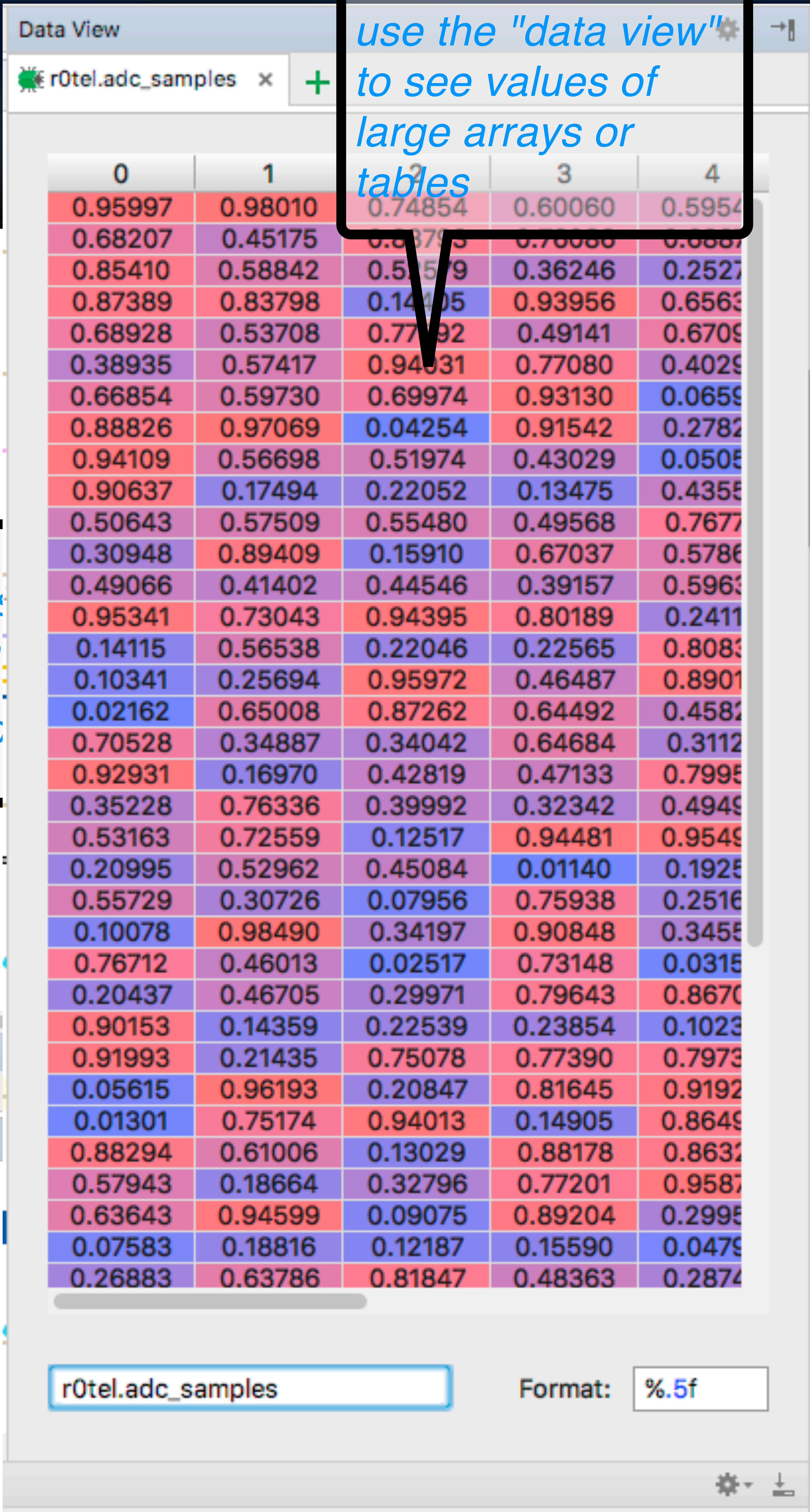


GUI debugging

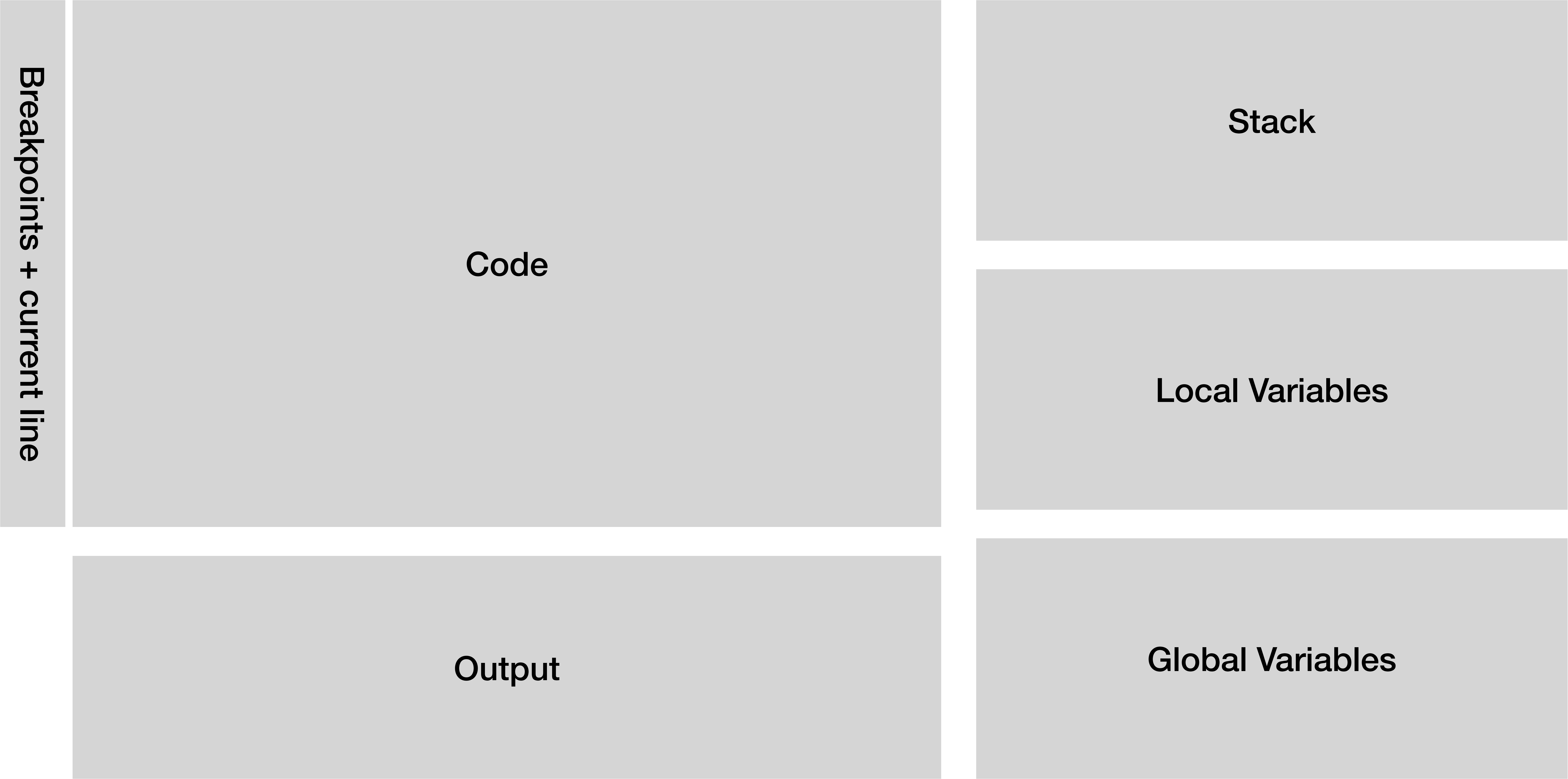


GUI debugging



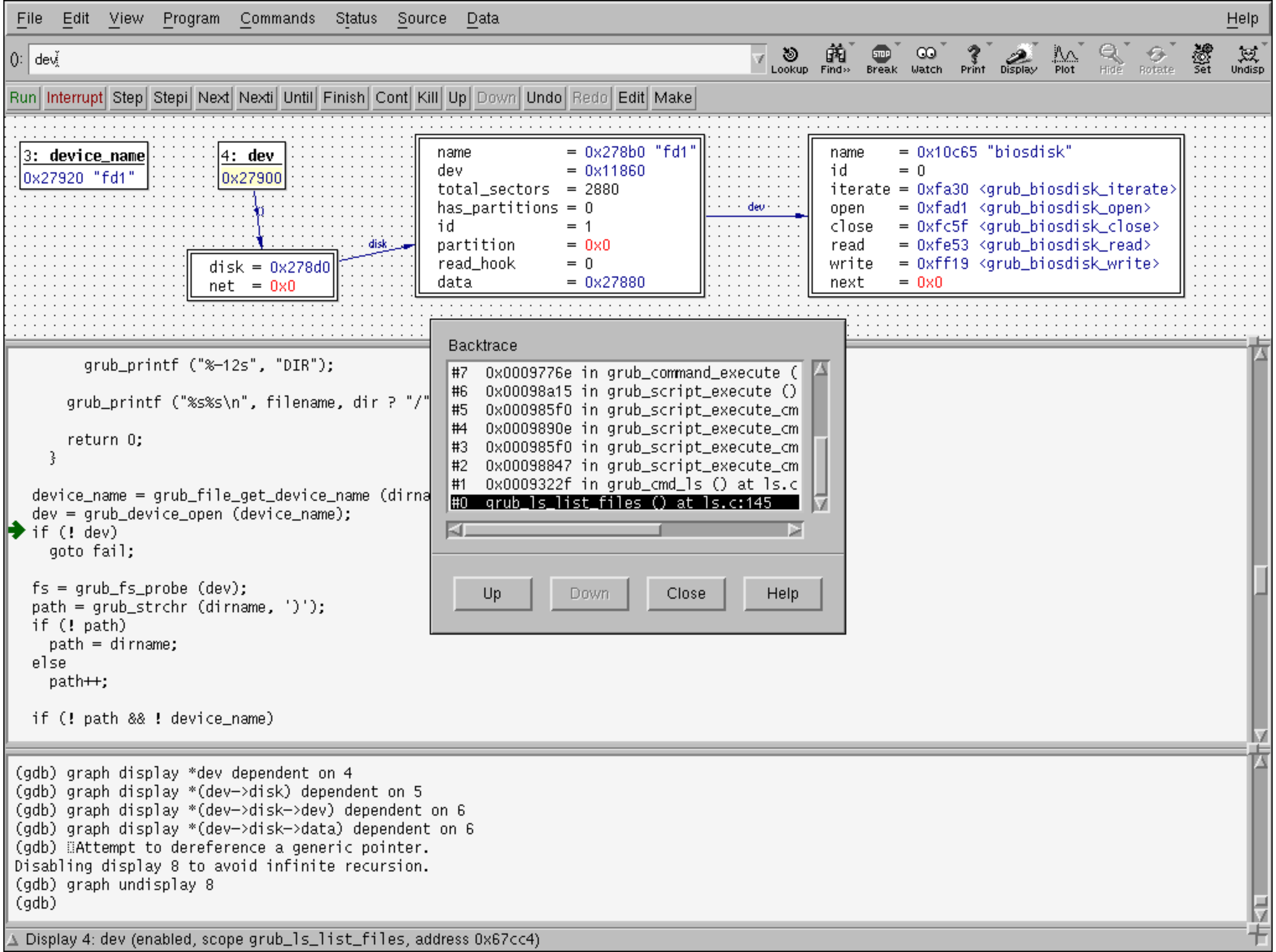


GUI Debuggers: what they usually look like



So basically like what I showed before, but fully interactive!

Sometimes also a "view" of data structures

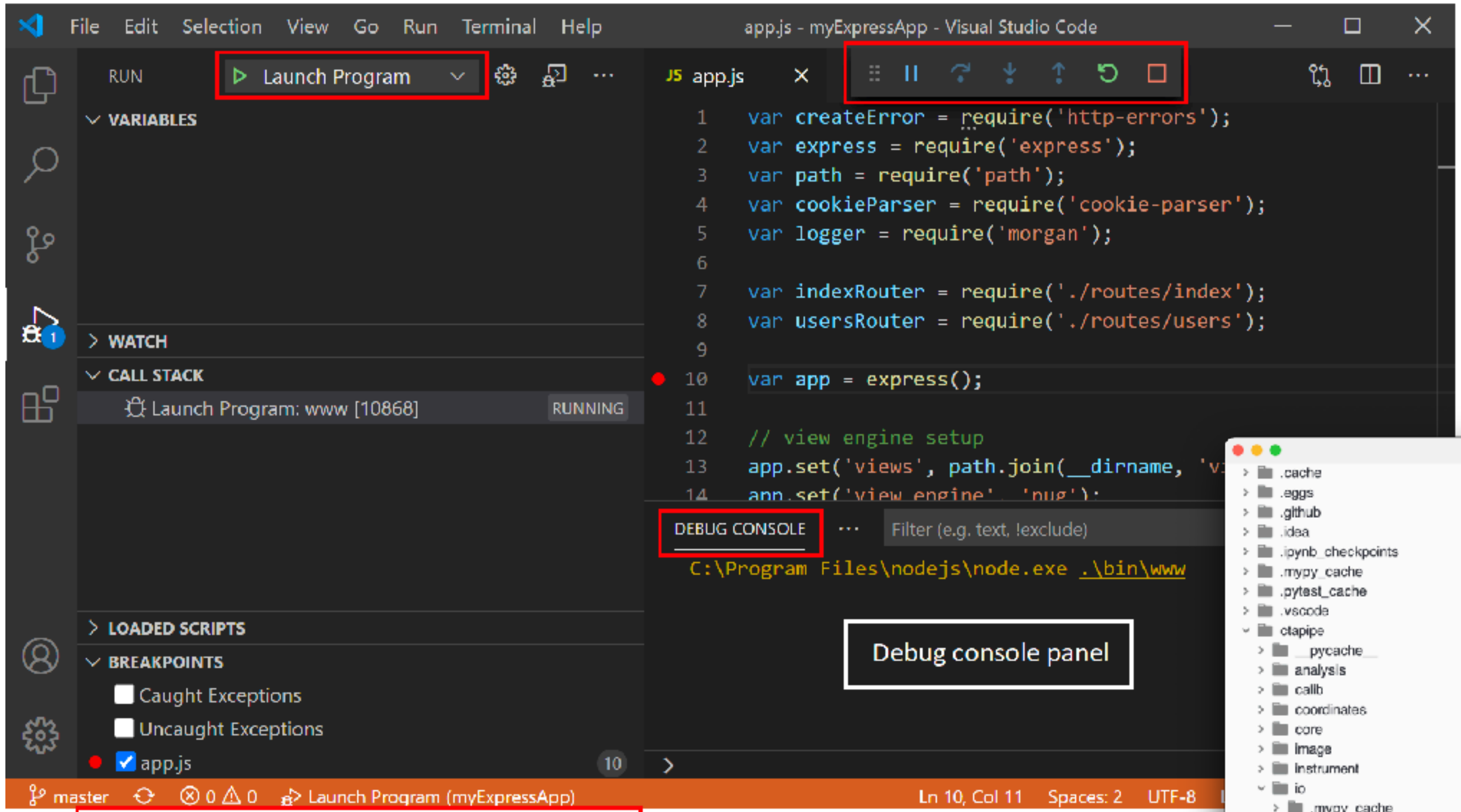


GNU Data Display Debugger (DDD)
(a C/C++ debugger)

VSCode Debugger (ptvsd)

Start debugging

Pause, step over, step in/out, restart, stop



Debug side bar

Emacs (M-x dap-debug)

note: need to install the debugger server first

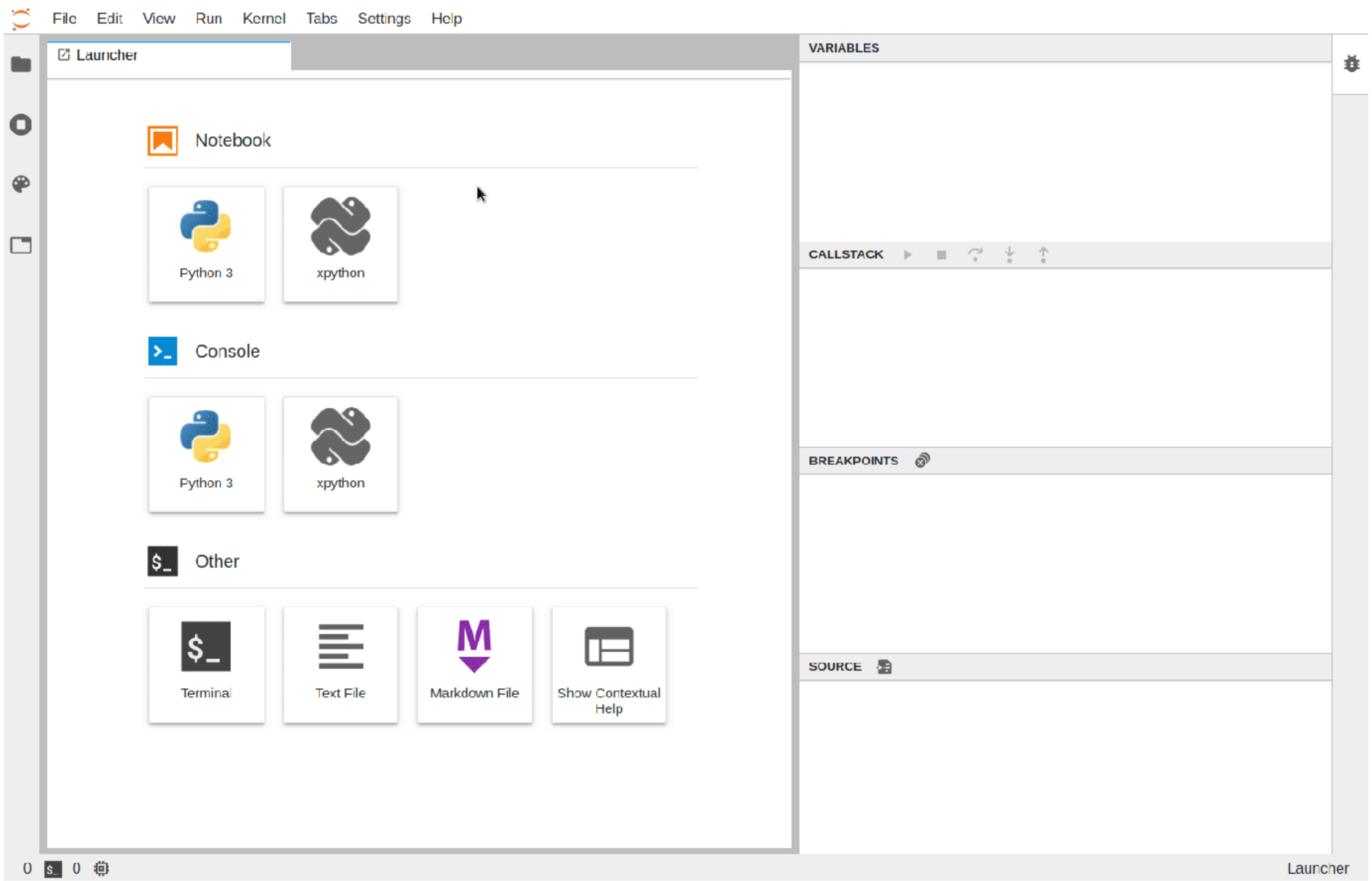
```
mamba install -c conda-forge ptvsd
```

Newish option: Jupyter-lab debugger extension

https://github.com/jupyterlab/debugger

Caveat:

requires ***xeus-python*** kernel and doesn't work with ipython kernel

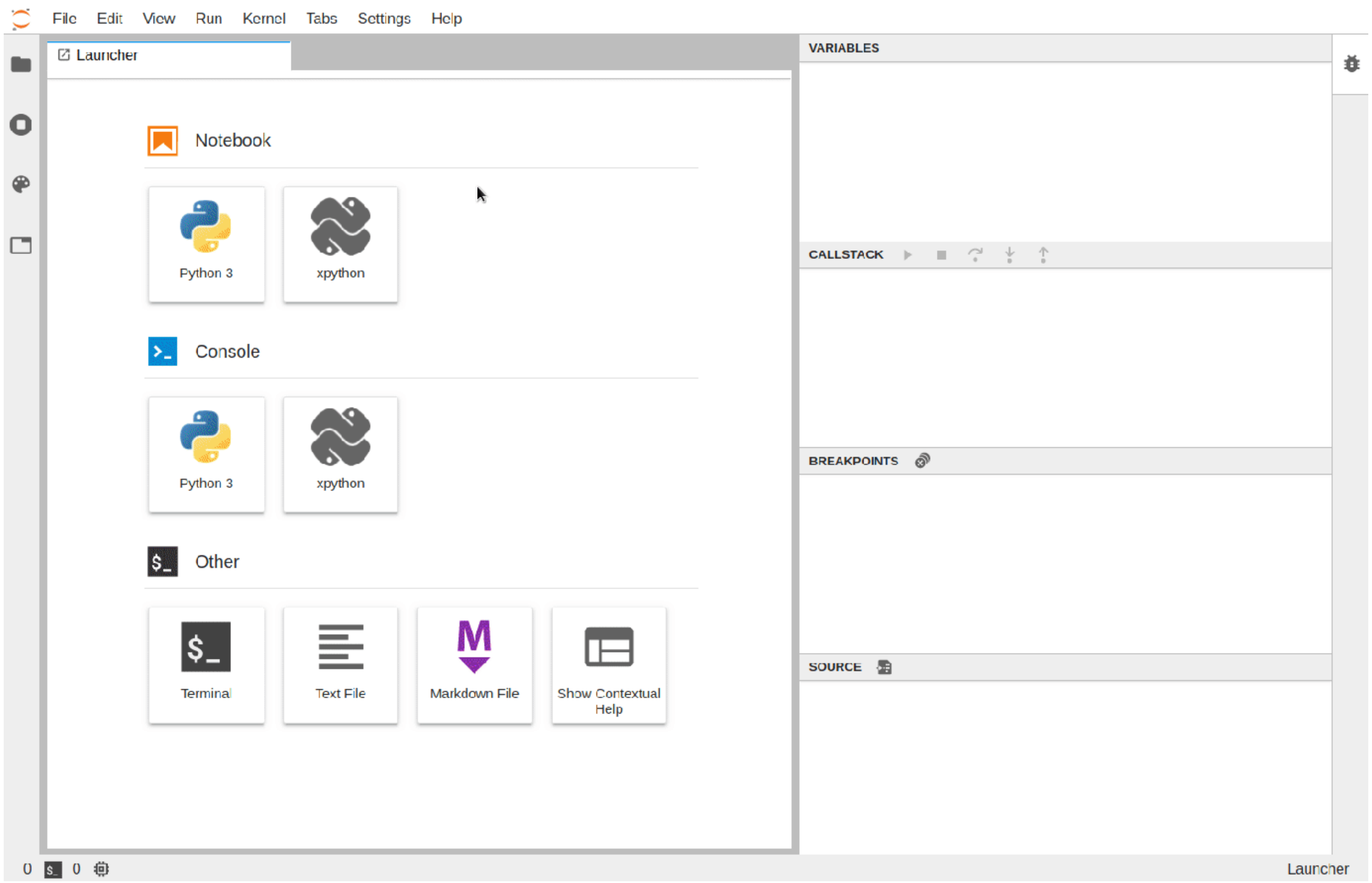


Newish option: Jupyter-lab debugger extension

https://github.com/jupyterlab/debugger

Caveat:

requires ***xeus-python*** kernel and doesn't work with ipython kernel



demo

Debugging with notebooks/ipython
Debugging with pdb
Debugging with a GUI (PyCharm)

Profiling and Optimization

ESCAPE School, June 2022

Your code works!

Your code works!

Your code works!

But it's slow.

Your code works!

But it's slow.

Now what?

“We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**

“We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**

-*Donald Knuth?*
or Sir Tony Hoare?

"We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**

-Donald Knuth?
or Sir Tony Hoare?

From a 1974 article on why GOTO statements are good

Why optimize?

* though some compilation happens

Why optimize?

You want your code to work first, but you do want it to be efficient!

- balance between
  - ➤ usability/**readability**/correctness
  - ➤ and speed/memory efficiency
- not always achievable → err on the side of *usability/readability!*

* though some compilation happens

Why optimize?

You want your code to work first, but you do want it to be efficient!

- balance between
  - usability/**readability**/correctness
  - and speed/memory efficiency
- not always achievable → err on the side of *usability/readability!*

Some things:

- Python is **interpreted*** → can be *slow*
- **For-loops in particular** → *100 - 1000x slower* than C loops…
- Mostly **one CPU Core** (GIL - Global Interpreter Lock)
- but there are ways to get around these...
  (See Tamas's Numpy/Numba lecture)

* though some compilation happens

Slowness of Python

Not an inherent problem with the *language*

- python ≠ CPython!
  - ➤ but CPython does generally get faster each release
- other python implementations exist that are trying to solve the general speed problem:
  - ➤ **pypy** - pypy.org fully JIT-compiled python
  - ➤ **pyston** - optimized CPython from Facebook
  - ➤ other efforts to remove bottlenecks from CPython (no GIL, etc)

Slowness of Python

Not an inherent problem with the *language*

- python ≠ CPython!
  - ➤ but CPython does generally get faster each release
- other python implementations exist that are trying to solve the general speed problem:
  - ➤ **pypy** - pypy.org fully JIT-compiled python
  - ➤ **pyston** - optimized CPython from Facebook
  - ➤ other efforts to remove bottlenecks from CPython (no GIL, etc)

So one option to optimization is:

Do nothing!

Wait for a faster implementation, or a new version of CPython to be released, or swap in a completely different implementation!

Steps to optimization

1) **Make sure code *works correctly* first**

- DO NOT optimize code you are writing or debugging!

2) **Identify use cases for optimization:**

- how often is a function called? Is it useful to optimize it?
- If it is not called often and finishes with reasonable time/memory, stop!

3) ***Profile*** **the code to identify bottlenecks in a more scientific way**

- Profile time spent in each function, line, etc
- Profile memory use

4) **try to re-write as little as possible to achieve improvement**

5) **refactor if it is still problematic…**

- some times the *design* is what is making the code slow... can it be improved? (e.g.: ***flat better than nested***!)

What is profiling?

A way to identify where resources are used by a program:

- CPU resources (computation time)
- Memory resources

Identify problems in your code like hangs and *memory leaks*

Identify "hotspots" in your code that may be useful to optimize!

- ➤ always ask your question: *will it make a real difference*?
- ➤ If it's good enough, STOP

Speed profiling 1: in a *notebook*

What I often see...

```
from time import time

start = time.time()

[code]

stop = time.time()
print(stop - start)
```

*this **measures only wall-clock time**!*

*You want **CPU time**!*
(not dependent on other stuff you are running)

*You want **many trials**, for statistics!*

Better method: *%timeit*

- *interactive **%timeit** "magic" jupyter/ipython function*
- *Automatically runs a function many times and measures CPU time and standard deviation*
- *Usage:*

```
%timeit <python statement>
```

Notes:

- *to time an entire cell, use **%%time***
- *you can also import the `**timeit**` **module***
- *if you really only want one trial, use **%%time***

Speed profiling 1: in a *notebook*

What I often see...

```
from time import time

start = time.time()

[code]

stop = time.time()
print(stop - start)
```

*this **measures only wall-clock time**!*

*You want **CPU time**!*
(not dependent on other stuff you are running)

*You want **many trials**, for statistics!*

Better method: *%timeit*

- *interactive **%timeit** "magic" jupyter/ipython function*
- *Automatically runs a function many times and measures CPU time and standard deviation*
- *Usage:*

```
%timeit <python statement>
```

***Notes**:*

- *to time an entire cell, use **%%time***
- *you can also import the `**timeit**` **module***
- *if you really only want one trial, use **%%time***

Speed profiling 2: profiler!

A profiler is better than a simple %timeit, in that it checks the time in *all* functions and sub-functions at once and generates a report.

Python provides several profilers, but the most common is *cProfile* (note: gprof for c++)

Profile an entire script:

- Run your script with the additional options:

```
python -m cProfile -o output.pstats  <script>
```

- this generates a **binary data file (*output.pstats*)** that contains statistics on **how often** and **for how long** each function was called
- There is a built-in **pstats** module that displays it using a command-line UI, but it's a bit difficult to use... but there are GUIs!

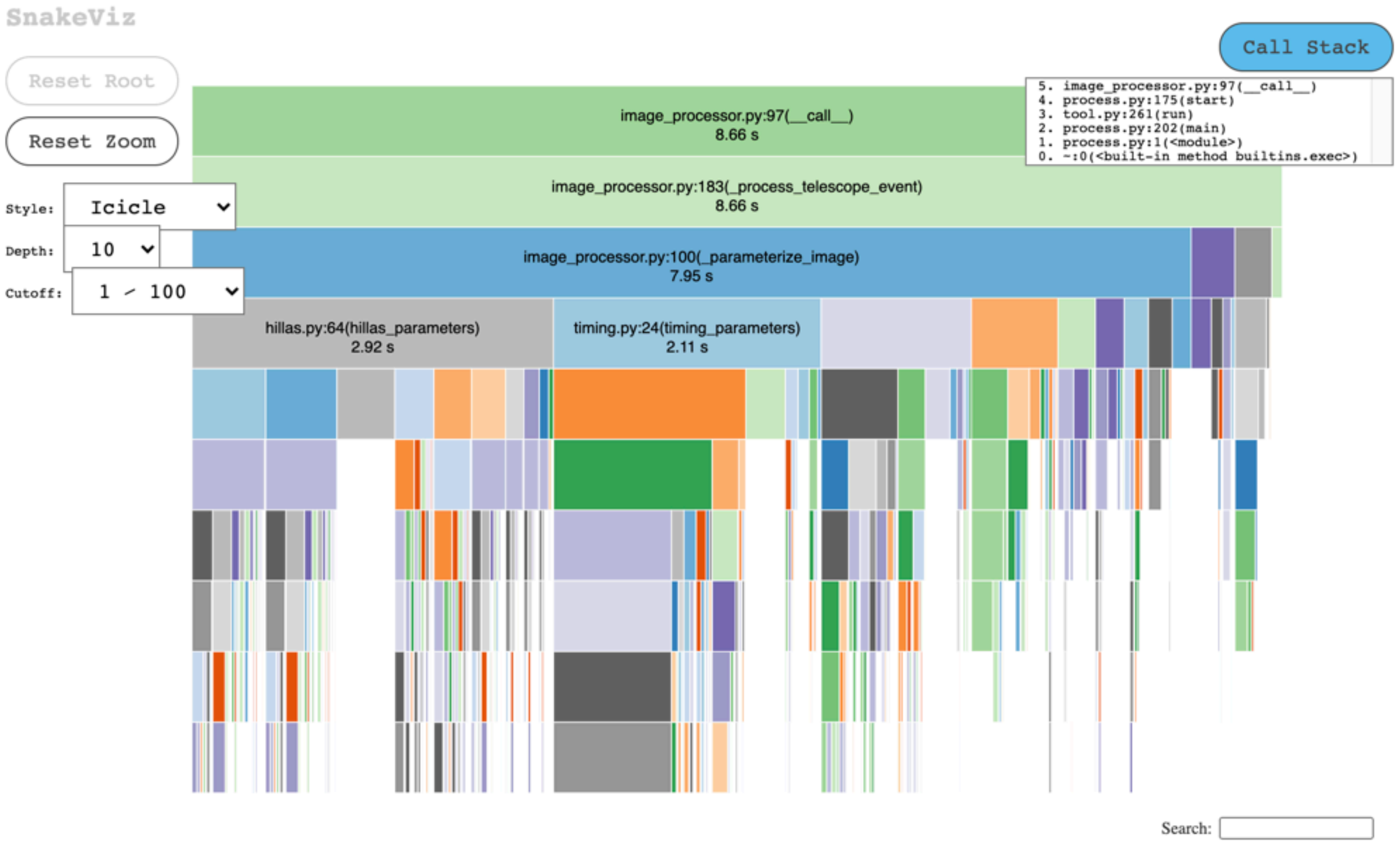
Tip: use a gui to view stats output

Viewing with *SnakeViz*

```
% conda install snakeviz
% snakeviz output.pstats
```

- interactive call statistics viewer
- this is not the only one, but it's nice and simple and runs in your browser.
- Click and zoom to see the results

Real-world demo!

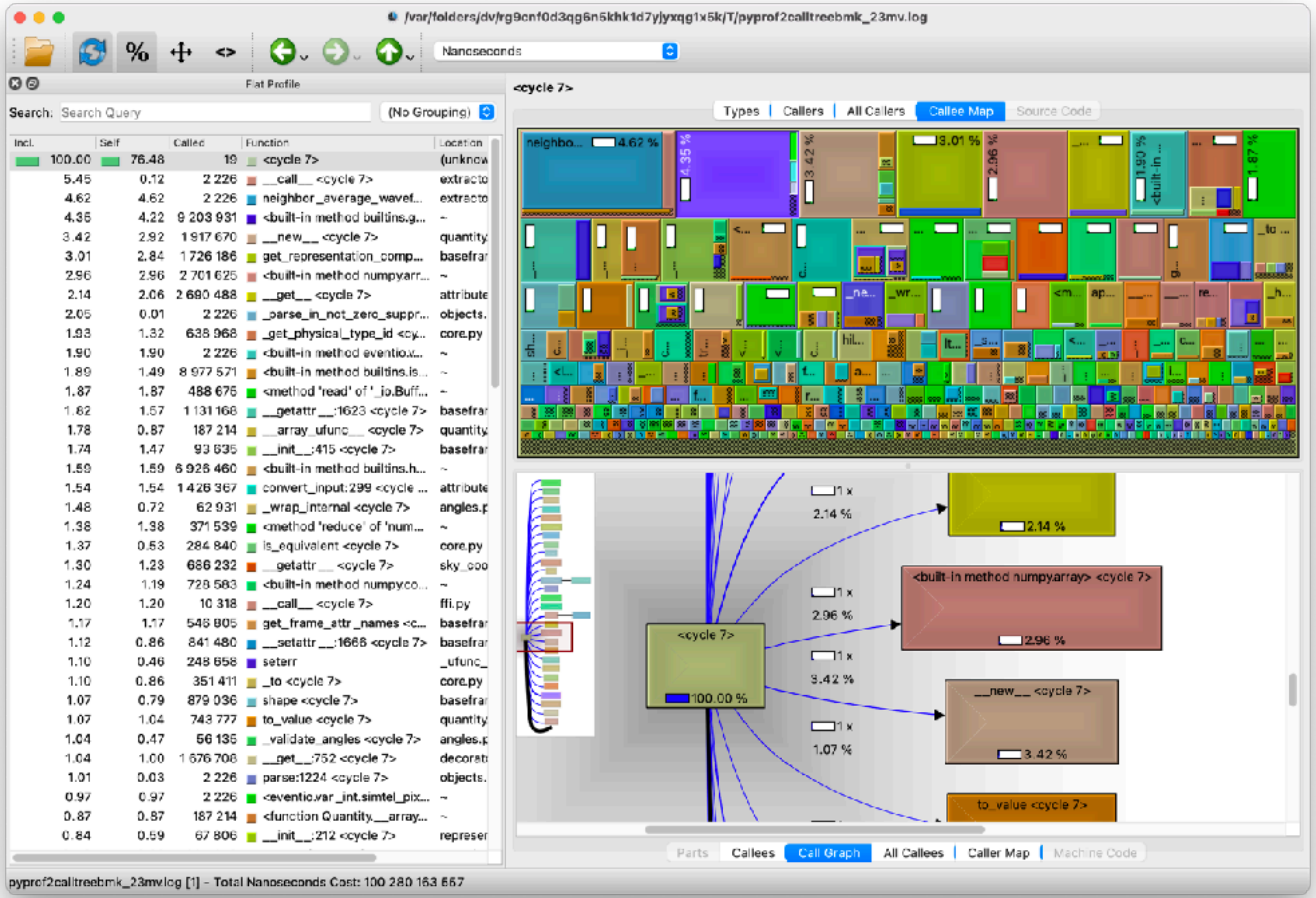


ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2226	4.629	0.002079	4.629	0.002079	extractor.py:195(neighbor_average_waveform)
9203931/4273437	4.232	9.903e-07	31.68	7.414e-06	~:0(<built-in method builtins.getattr>)
2701625	2.966	1.098e-06	3.257	1.205e-06	~:0(<built-in method numpy.array>)
1917670	2.925	1.525e-06	8.082	4.215e-06	quantity.py:289(__new__)
1726186	2.848	1.65e-06	5.015	2.905e-06	baseframe.py:850(get_representation_component_names)
2690488/1922760	2.069	1.076e-06	23.99	1.248e-05	attributes.py:95(__get__)

Another stats viewer

You can also view pstats output with the *qcachegrind* GUI application, (also for C++ C++ profiling output):

```
% pip install pyprof2calltree
% pyprof2calltree -i output.pstats -k
```



This will open qCacheGrind GUI automatically

you need to first install qCacheGrind using your package manage (it's not in Conda), e.g.

brew install qcachegrind (macOS with HomeBrew installed)

apt install qcachegrind (linux with Apt)

...

Profiling in a Notebook

You can also run the profiler directly on a statement in a notebook.

- use the magic %prun function

```
%prun <python statement>
```

- Pops up a sub-window with the results (the same as if you ran cProfile and then pstats (though you don't get an interactive viewer)

```
In [27]: %prun create_array_loop(1000,1000)
```

```
         3001004 function calls in 0.845 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.477    0.477    0.835    0.835 <ipython-input-12-6d84b414c957>:1(create_array_loop)
  1000000    0.136    0.000    0.136    0.000 {built-in method math.cos}
  1000000    0.133    0.000    0.133    0.000 {built-in method math.sin}
  1001000    0.089    0.000    0.089    0.000 {method 'append' of 'list' objects}
        1    0.010    0.010    0.845    0.845 <string>:1(<module>)
        1    0.000    0.000    0.845    0.845 {built-in method builtins.exec}
```

Line Profiling

What about time spent in each line of code?

The line_profiler module can help:

```
% conda install line_profiler
```

- mark code with @profile:

```
from line_profiler import profile

@profile
def slow_function(a, b, c):
    ...
```

- Then run:
  - % **kernprof** -l script_to_profile.py
- which generates a .lprof file that can be viewed with:
  - % **python -m line_profiler** script_to_profile.py.lprof

```
File: pystone.py
Function: Proc2 at line 149
Total time: 0.606656 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   149                                           @profile
   150                                           def Proc2(IntParIO):
   151     50000        82003      1.6     13.5      IntLoc = IntParIO + 10
   152     50000        63162      1.3     10.4      while 1:
   153     50000        69065      1.4     11.4          if Char1Glob == 'A':
   154     50000        66354      1.3     10.9              IntLoc = IntLoc - 1
   155     50000        67263      1.3     11.1              IntParIO = IntLoc - IntGlob
   156     50000        65494      1.3     10.8              EnumLoc = Ident1
   157     50000        68001      1.4     11.2          if EnumLoc == Ident1:
   158     50000        63739      1.3     10.5              break
   159     50000        61575      1.2     10.1      return IntParIO
```

Line-profiling in a Notebook

As with *cProfile* and *timeit*, you can do line profiling in a notebook:

- unlike %timeit, need to load an extension first:

```
%load_ext line_profiler
```

- Then, if you have a function defined, you must "mark" it to be profiled by adding "-f <func>"

```
%lprun -f <function name> <python statement that uses function>
```

for example:

```
%lprun -f myfunc myfunc(100,100)
```

Note you can mark more than one func

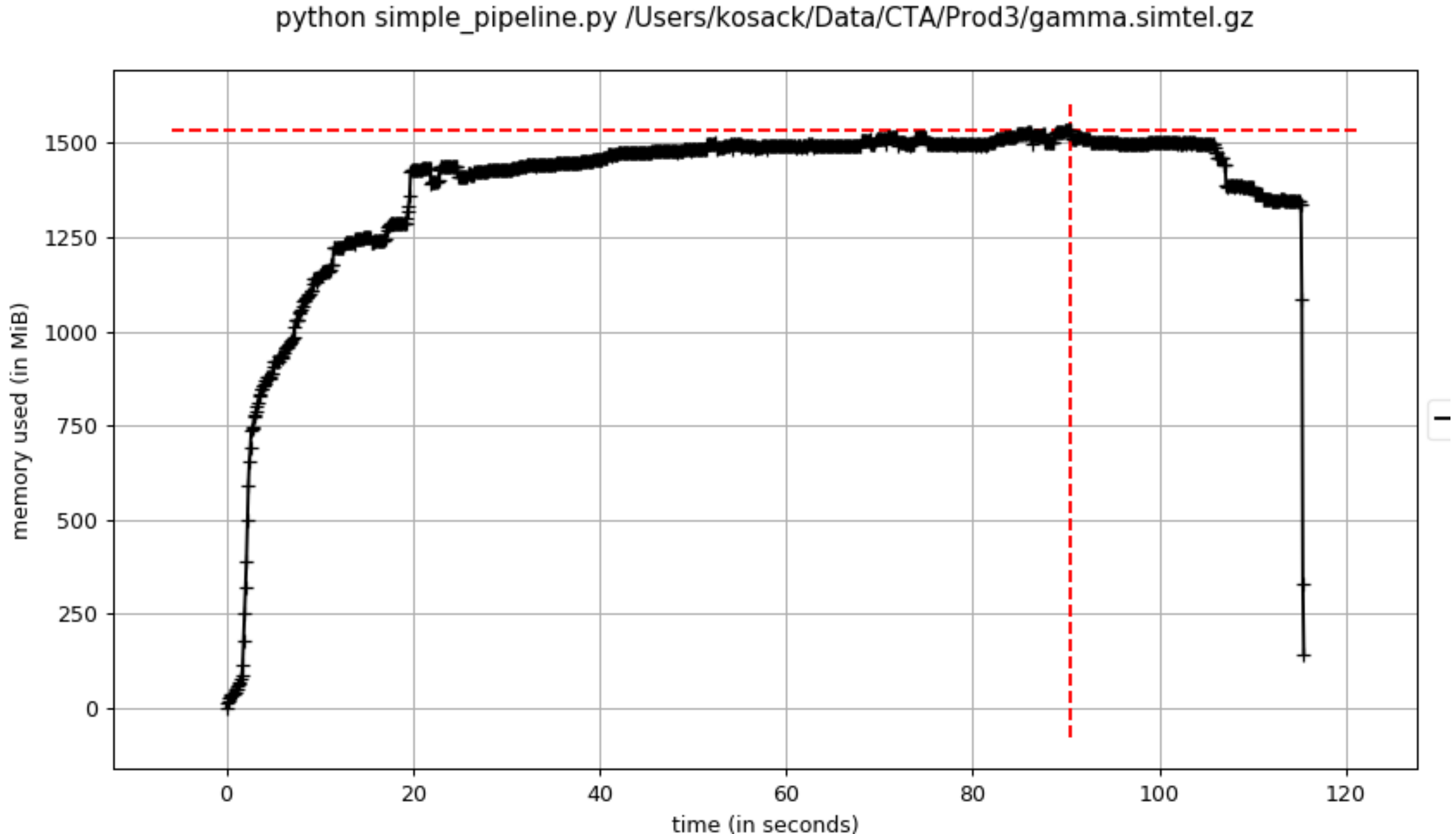
```
In [51]: %lprun -f create_array_loop create_array_loop(1000,1000)

Timer unit: 1e-06 s

Total time: 1.31799 s
File: <ipython-input-12-6d84b414c957>
Function: create_array_loop at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def create_array_loop(N,M):
     2         1            2      2.0      0.0      arr = []
     3      1001          477      0.5      0.0      for y in range(M):
     4      1000         5244      5.2      0.4          row = []
     5   1001000       463343      0.5     35.2          for x in range(N):
     6   1000000       848316      0.8     64.4              row.append(sin(x)*cos(0.1*y))
     7      1000          606      0.6      0.0          arr.append(row)
     8         1            1      1.0      0.0      return arr
```

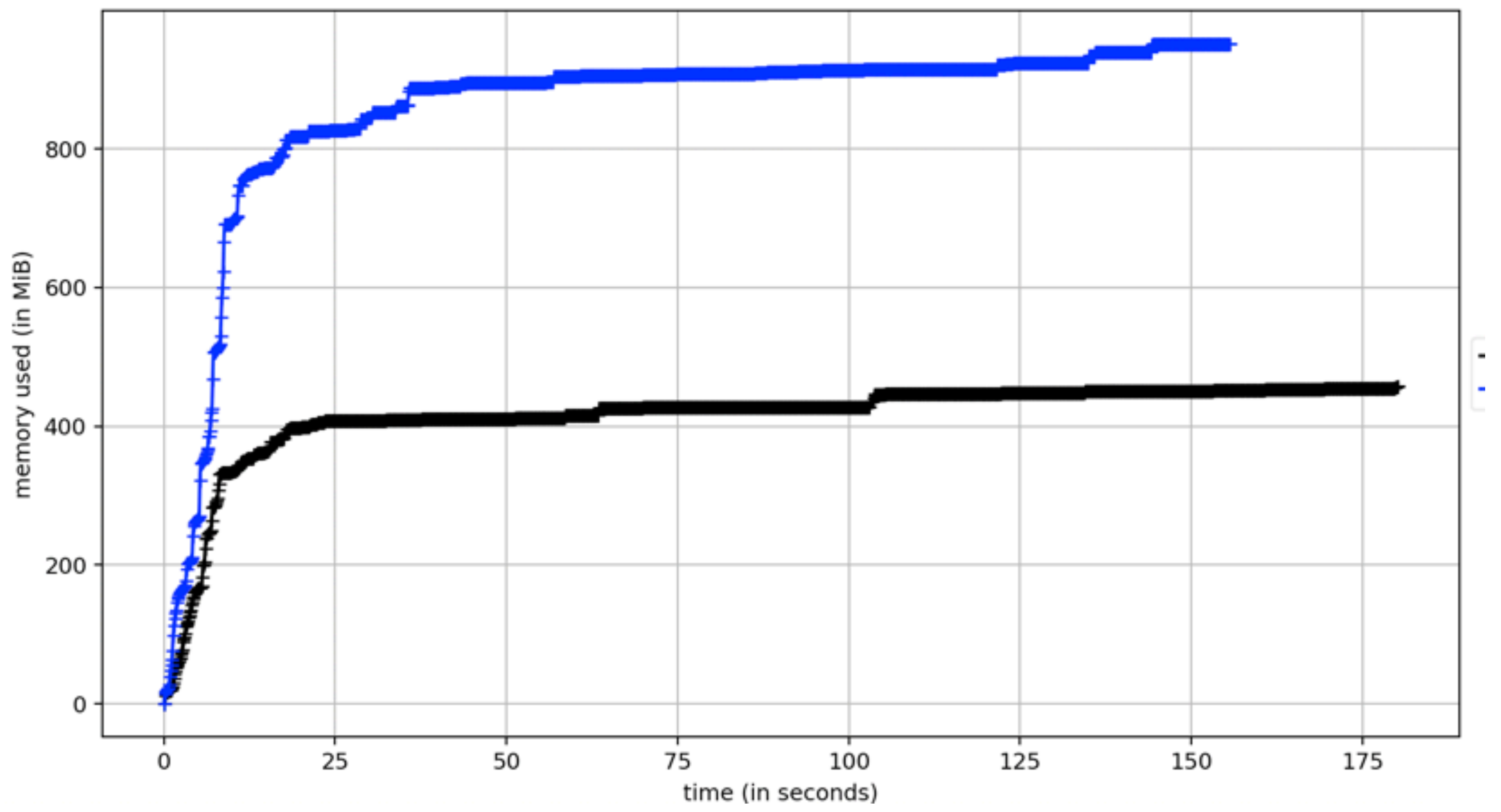
Memory Profiling

Use of CPU is not the only thing to worry about… what about RAM? Let's first check for memory leaks…

```
% conda install memory_profiler
% mprof run python <script>
% mprof plot
```

< This is already in your **eschool2022** environment





Memory Profiling in detail

Cumulative is nice, but we want to see the memory for a particular function or class…

- decorate the function you want to profile (line-wise) with memory_profiler.profile

```
% python -m memory_profiler <script>
```

Decorate what we want to measure (no import needed)

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    17                                           @profile
    18                                           def main():
    19         1          3.0      3.0      0.0      if len(sys.argv) ≥ 2:
    20                                                   filename = sys.argv[1]
    21                                               else:
    22         1        485.0    485.0      0.0          filename = get_dataset_path("gamma_test_large.simte
    24         1    3572651.0 3572651.0     9.8      with EventSource(filename, max_events=500) as source:
    26         1     438843.0 438843.0      1.2          calib = CameraCalibrator(subarray=source.subarray)
    27         2     249622.0 124811.0      0.7          process_images = ImageProcessor(
    28         1          2.0      2.0      0.0              subarray=source.subarray, is_simulation=source.
    29                                                   )
    30         1       1363.0   1363.0      0.0          process_shower = ShowerProcessor(subarray=source.su
    31         2     276938.0 138469.0      0.8          write = DataWriter(
    32         1          0.0      0.0      0.0              event_source=source, output_path="events.DL1.h5
    33                                                   )
    35       111   11506526.0 103662.4     31.5          for event in tqdm(source):
    36       110    1313386.0  11939.9      3.6              calib(event)
    37       110    2353948.0  21399.5      6.4              process_images(event)
    38       110   14044245.0 127675.0     38.4              process_shower(event)
    39       110    2814913.0  25590.1      7.7              write(event)
```

Output shows the time spent in the line or block (e.g. if , for)

Memory Profiling in a Notebook

Again, you can do memory profiling using magic commands in an iPython (Jupyter) notebook

- Enable the memory profiling notebook extension:

```
%load_ext memory_profiler
```

- Now you have access to several magic functions:

Like %timeit, but for memory usage:

```
%memit <python statement>
```

or a more full-featured report:

```
%mprun -f <function name> <statement>
```

```
In [40]: %memit range(100000)
peak memory: 89.61 MiB, increment: 0.00 MiB

In [41]: %memit np.arange(100000)
peak memory: 90.12 MiB, increment: 0.52 MiB
```

Caveats:

- the peak memory usage shown in the notebook may not relate to the function you are testing! It is the sum of all memory already allocated that has not yet been garbage collected. (so look at the "increment" instead).
- %mprun only works if your functions are **defined in a file** (not a notebook) and imported into the notebook

A real-world example from a few days ago

(a Pull-Request for code written by Max Nöthe)

kosack approved these changes 4 days ago

View changes

kosack left a comment

Member

Very nice! I did a quick check to make sure the memory is properly handled, and it looks great:

```
]: %%memit
   for chunk in tqdm(loader.read_subarray_events_chunked(10_000)):
       pass
```

100% 392/392 [00:02<00:00, 190.00it/s]

```
peak memory: 228.96 MiB, increment: 33.36 MiB
```

```
]: %%memit
   loader.read_subarray_events()
```

```
peak memory: 1110.58 MiB, increment: 881.62 MiB
```

Memory Profiling: jump to debugger

Automatic Debugger breakpoints:

- you can automatically start the debugging if the code tries to go above a memory limit, to see where the allocation is happening:

```
% python -m memory_profiler --pdb-mmem=100  <script>
```

will break and enter debugger after 100 MB is allocated, on the line where the last allocation occurred

Print out memory usage during program execution:

```
from memory_profiler import memory_usage
mem_usage = memory_usage(-1, interval=.2, timeout=1)
print(mem_usage)
    [7.296875, 7.296875, 7.296875, 7.296875, 7.296875]
```

- see the docs. you can also write it to a log periodically, etc.

demo