

# Debugging & Profiling Scientific Code



Karl Kosack  
CEA Paris-Saclay

*ESCAPE School, June 2022*

cea

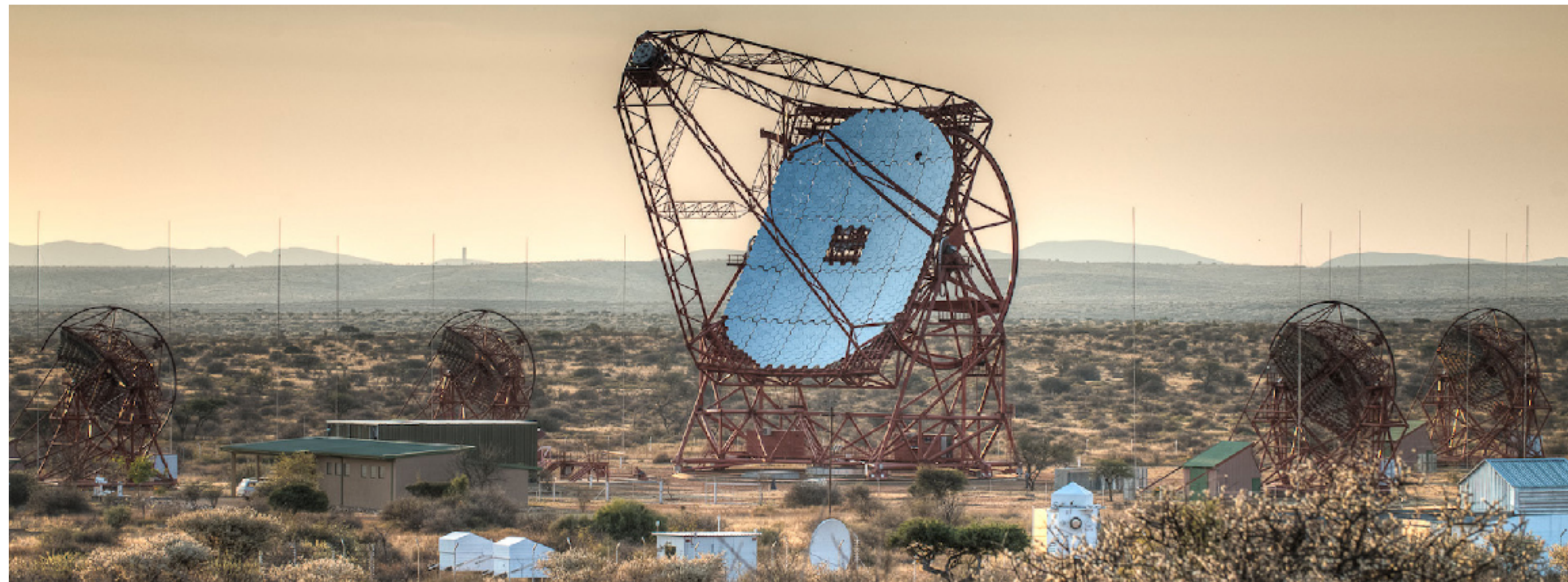




# A bit about me...



<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!**



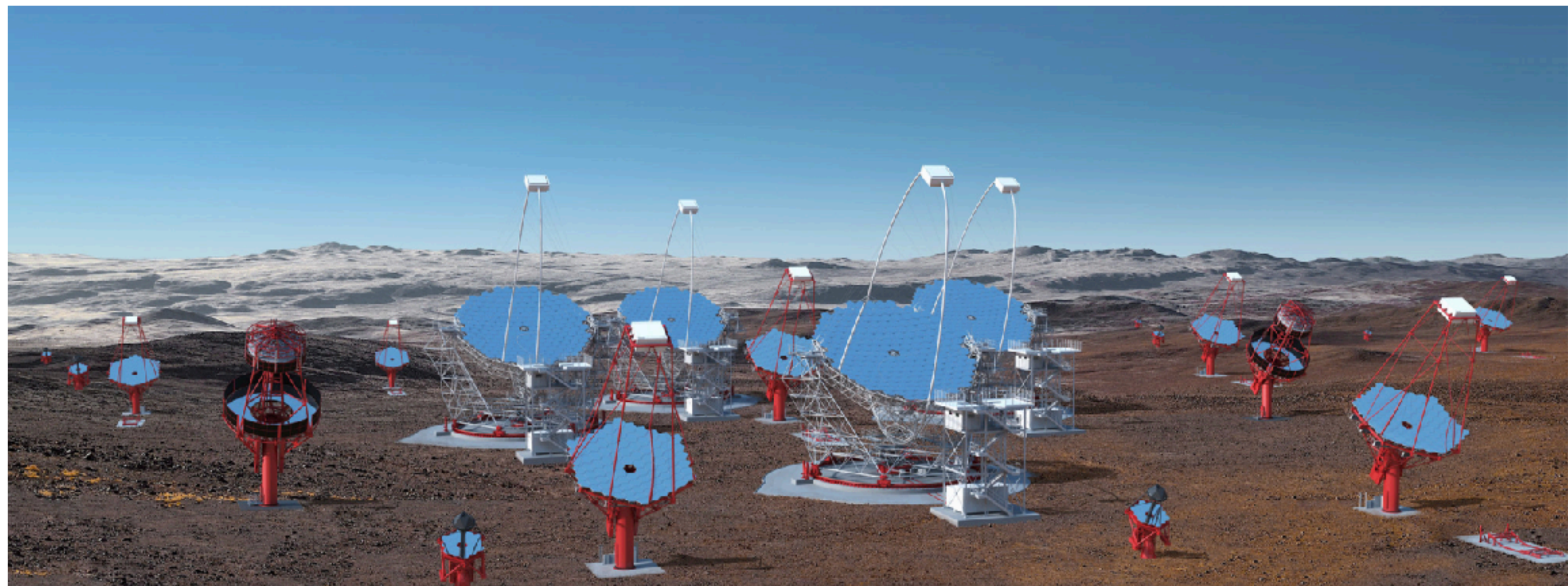
# A bit about me...



<https://www.mpi-hd.mpg.de/hfm/HESS/>

<https://www.cta-observatory.org/>

<https://github.com/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!**




The background of the slide is a dark blue gradient. On the right side, there are several glowing blue particle tracks, which are circular patterns of small dots connected by thin lines, resembling a molecular structure or a data visualization. One large, prominent track is in the upper right, and another, more detailed wireframe-like track is in the lower right. The text is overlaid on this background.

**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**)

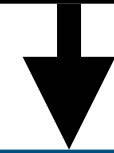
Write *tests* to  
ensure it does!



Some good  
advice for  
writing  
**Scientific  
Code**



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



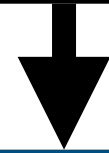
**Profile** to find bottlenecks

Write *tests* to  
ensure it does!

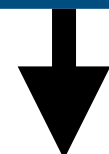
**Some good  
advice for  
writing  
Scientific  
Code**



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



Profile to find bottlenecks



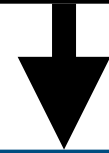
Optimize *only* what needs to be!

Write *tests* to ensure it does!

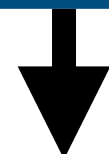
Some good advice for writing **Scientific Code**



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**)

Write *tests* to  
ensure it does!

Profile to find bottlenecks

run *tests* to  
check

Optimize *only* what needs to be!

Refactor if necessary  
(redesign/rewrite)

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

run *tests* to  
check

Optimize *only* what needs to be!

run *tests* to  
check

Refactor if necessary  
(redesign/rewrite)

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

run *tests* to  
check

Optimize *only* what needs to be!

run *tests* to  
check

Refactor if necessary  
(redesign/rewrite)

Some good  
advice for  
writing  
**Scientific  
Code**



## 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



**Topics we  
will cover  
in this  
lecture**





**ESCAPE**

European Science Cluster of Astronomy &  
Particle physics ESFRI research Infrastructures

# Debugging

ESCAPE School, June 2022





# 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"

**Do you:** (show of hands)

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)

- 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

```
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

```
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



## Global Memory

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

## Local Memory

```
n = 0
```



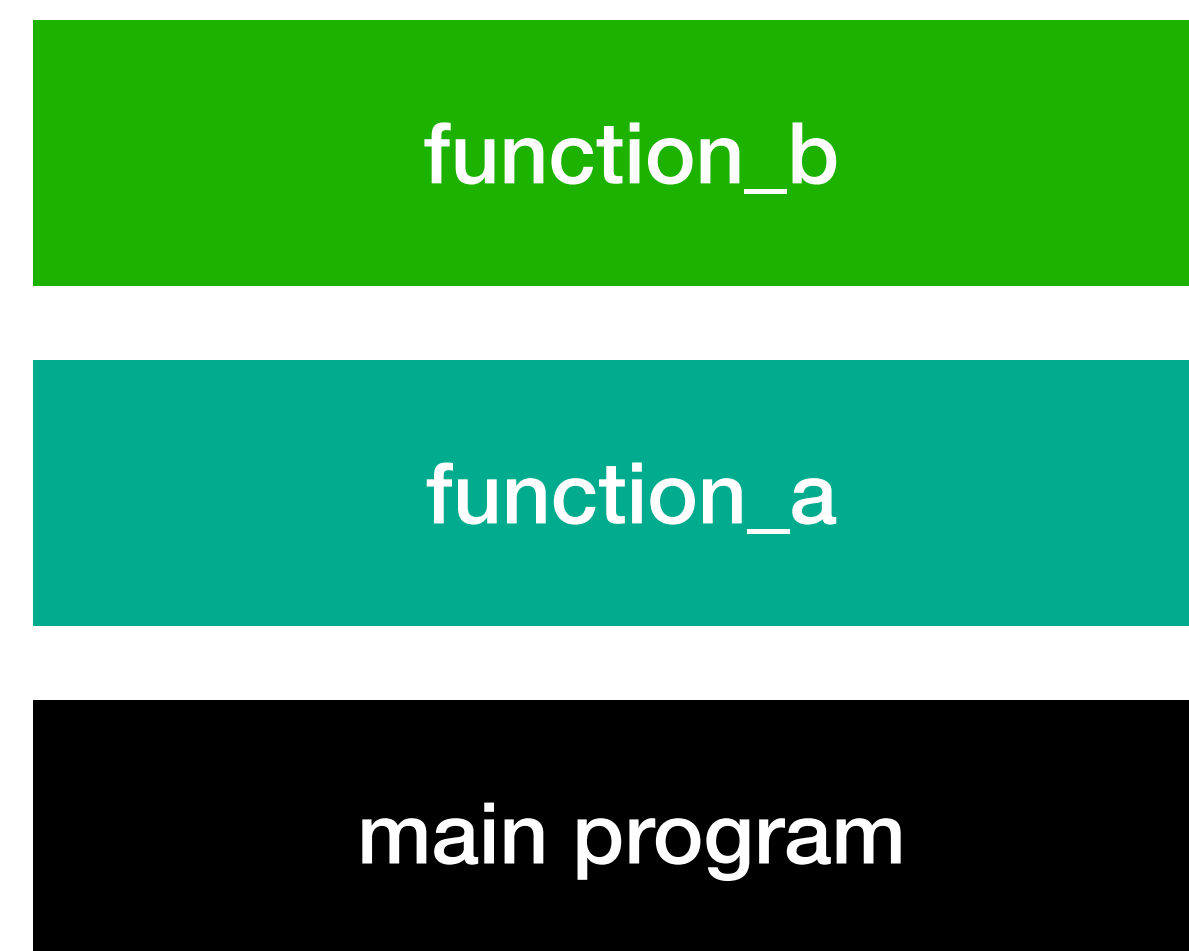
# First: how do programs run?

## 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



## Global Memory

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

## Local Memory

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



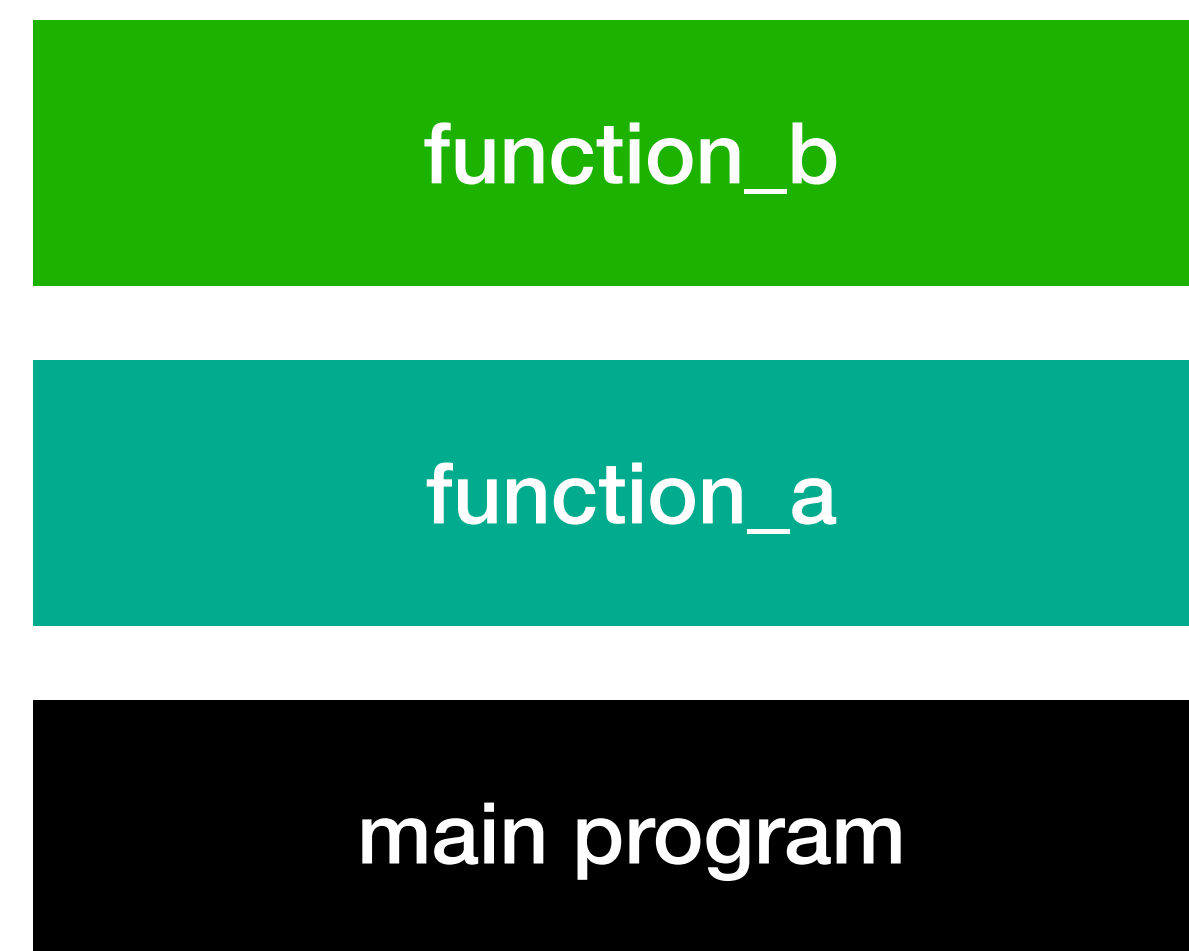
# First: how do programs run?

## 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



## Global Memory

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

## Local Memory

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



# 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



## 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

```
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

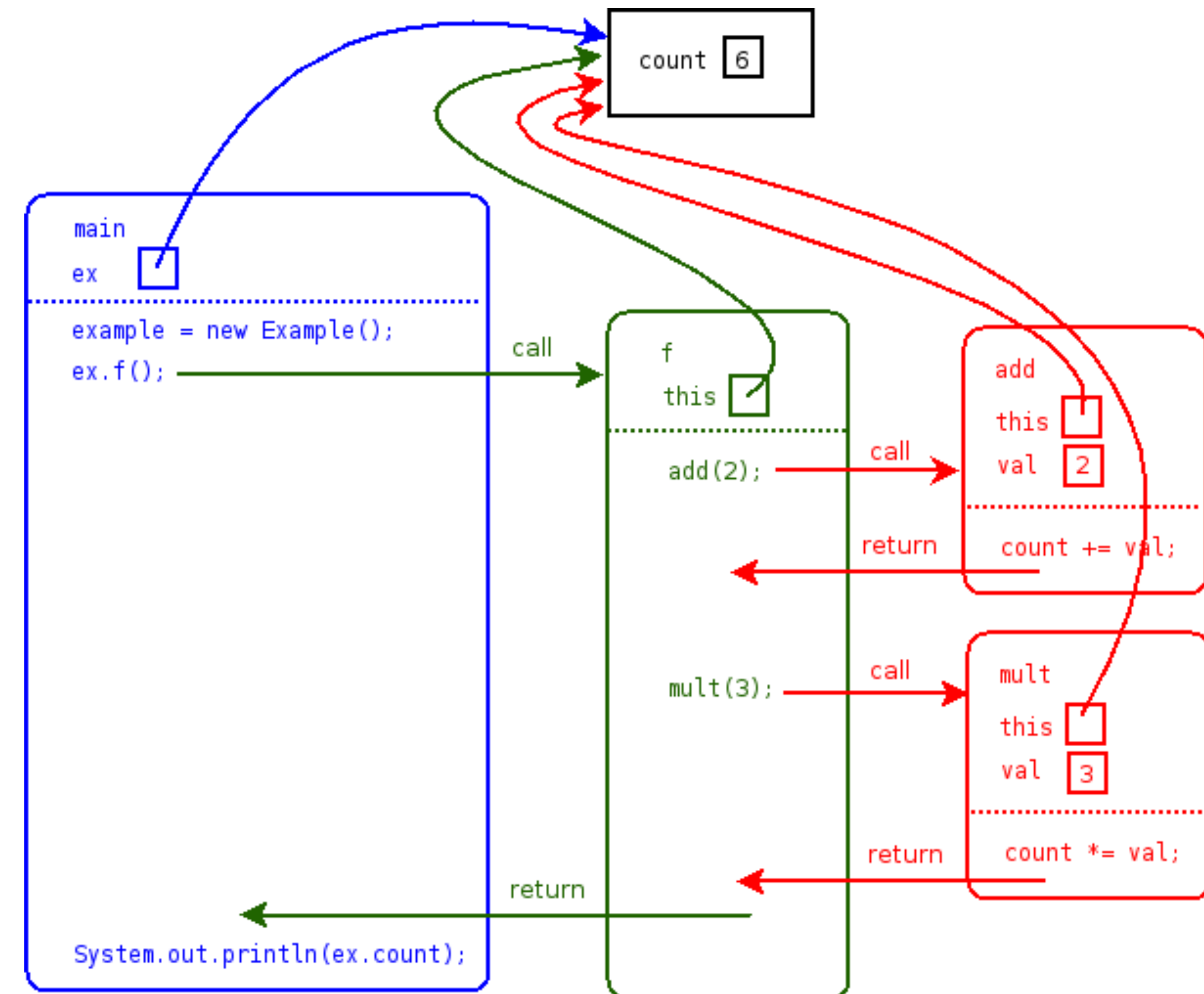


diagram from: <http://faculty.ycp.edu/~dhovemey/spring2007/cs201/info/exceptionsFileIO.html>



# 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

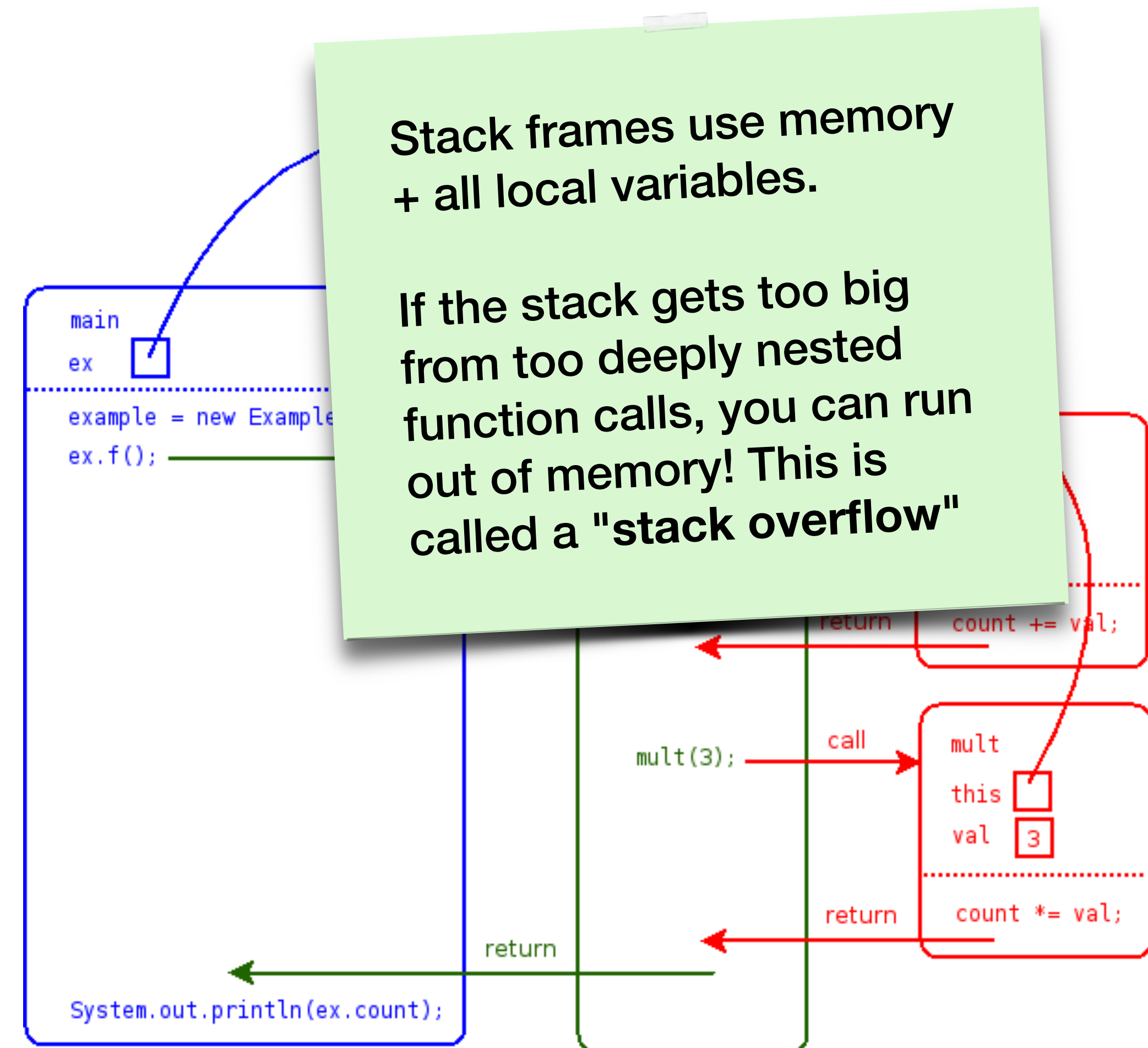


diagram from: <http://faculty.ycp.edu/~dhovemey/spring2007/cs201/info/exceptionsFileIO.html>



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

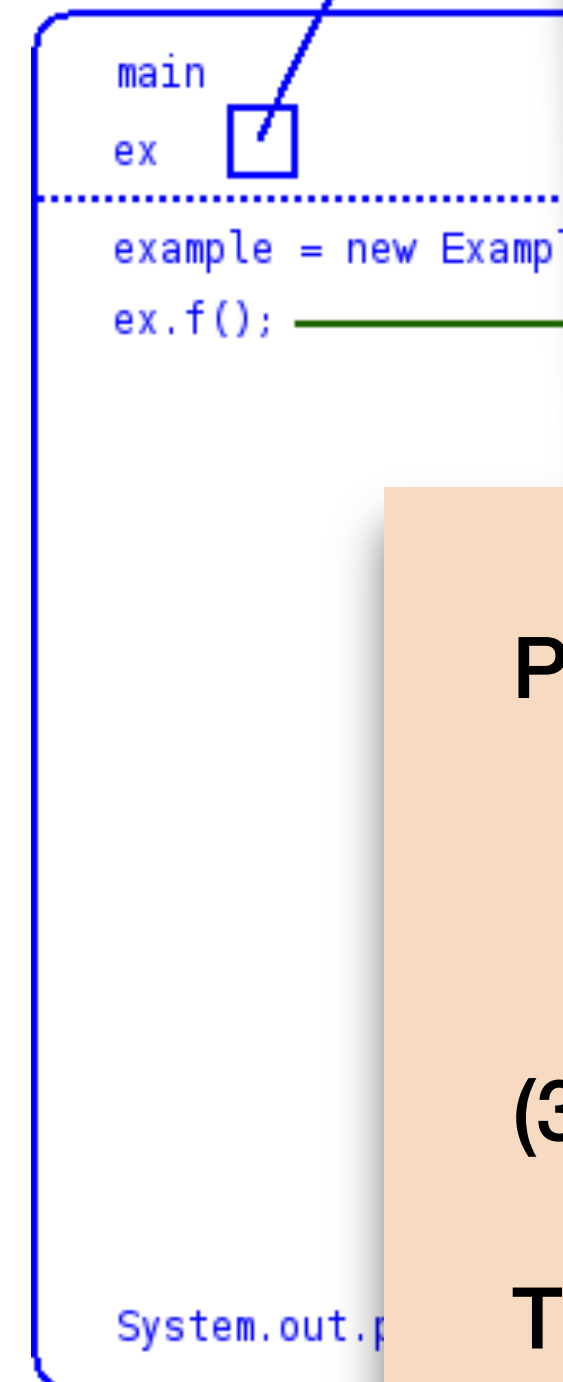
## Heap:

- 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"

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

diagram from: <http://facult>



# 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*

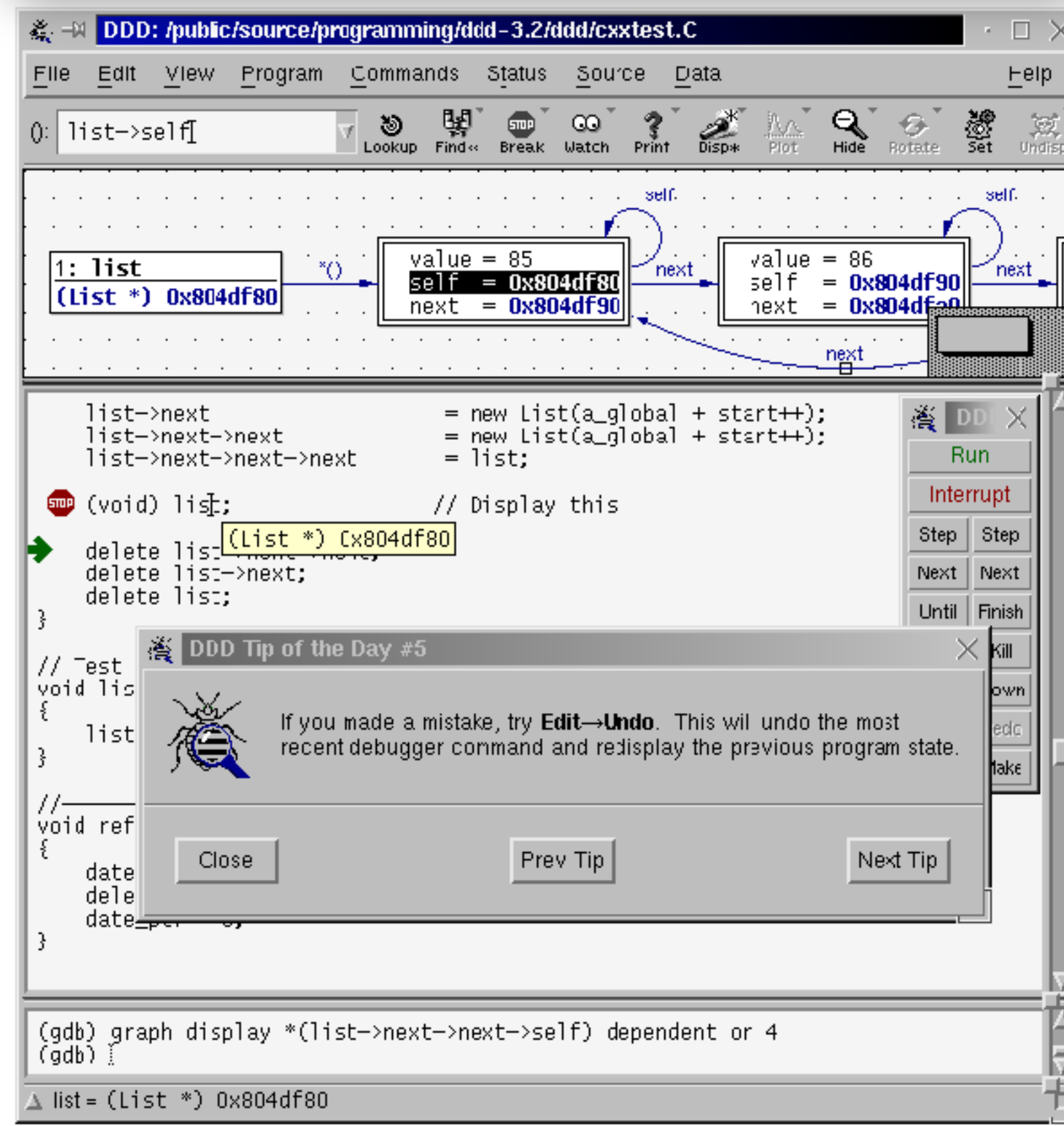
```
[ 0.86932713, 0.74726936, 0.77972359, 0.88279606, 0.76825295,
 0.39924089, 0.26050213, 0.82032474, 0.18800458, 0.43211861]],
'adc_sums': array([ 0.80428043, 0.8199334 , 0.16511381, 0.93497246, 0.81474172,
 0.32322294, 0.51430672, 0.24404024, 0.95566716, 0.52979194,
 0.656204 , 0.13846386, 0.38674983, 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, 0.06686986, 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, 0.85668434, 0.55420725, 0.88771667,
 0.55564573, 0.8569015 , 0.24182574, 0.35381984, 0.00141777]),
'num_samples': 10)
(Pdb) bt
/Users/kosack/anaconda/lib/python3.6/dbdb.py(431)run()
-> exec(cmd, globals, locals)
<string>(1)<module>()
/Users/kosack/Projects/CTA/Working/ctape/ctape/lo/tests/test_hdf5.py(77)<module>()
-> test_write_container("test_h5")
/Users/kosack/Projects/CTA/Working/ctape/ctape/lo/tests/test_hdf5.py(23)test_write_container()
-> r0tel.meta['test_attribute'] = 3.14159
(Pdb) 4=
```

- 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 version
```

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

- **More work, but more flexible:** run

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

- the script will not run, but rather for you to type commands
- use *next*, *step*, *cont* to step through
- 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!**

*Open the "executable" part of the script and click the "debug" icon in the toolbar  
(may have to first create a debug config to tell what file to run)*

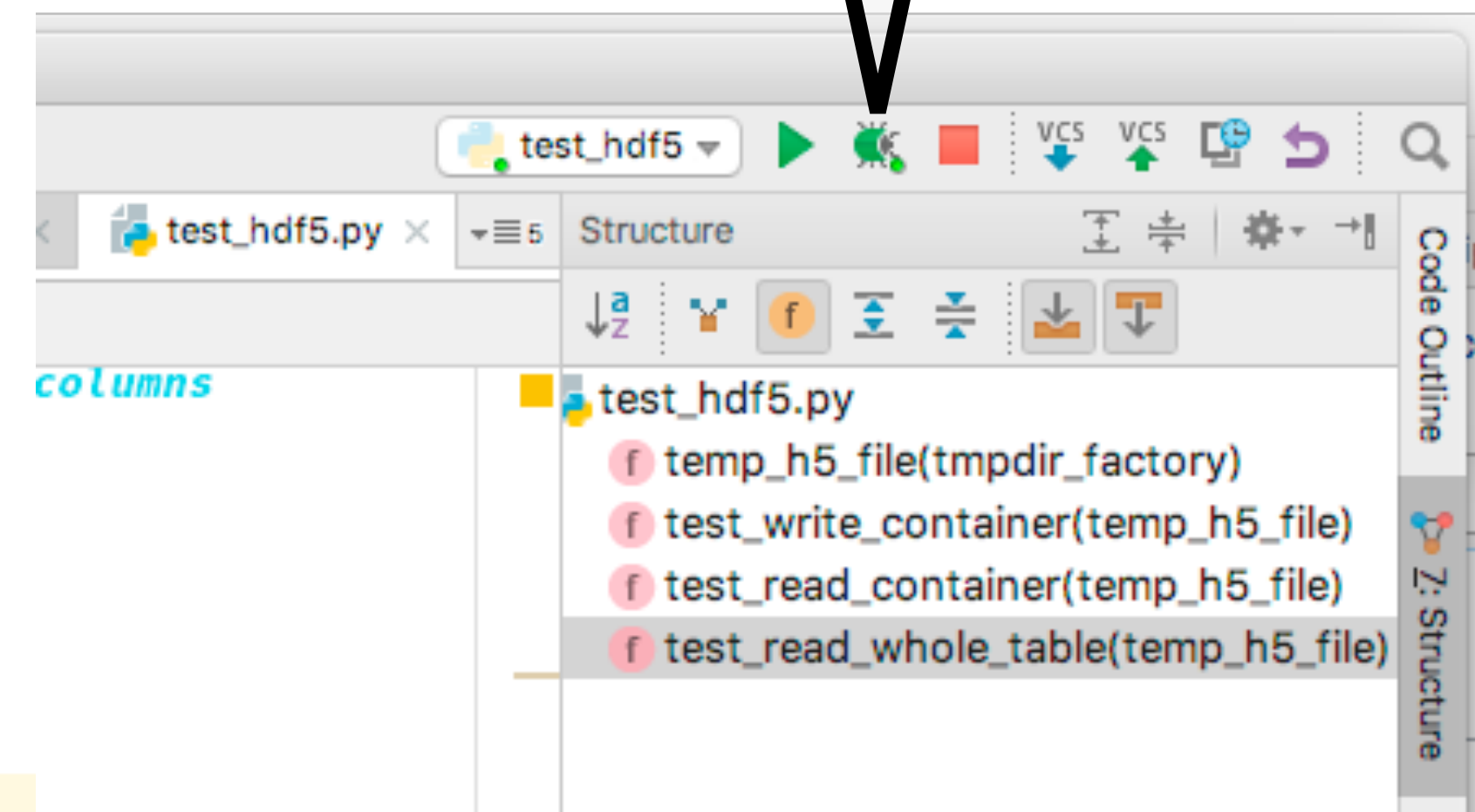
*Click in margin to set a breakpoint*

```
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 indicates a breakpoint on the line `reader = SimpleHDF5TableReader(str(temp_h5_file))`. A call stack window at the bottom shows the current stack frame with variables `mc` and `temp_h5_file`. Annotations in blue text boxes provide instructions on how to navigate and view variables.

```
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,
    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

The image shows a Python IDE with a debugger. The main window displays a code editor with Python code. A debugger window is open, showing the current stack frame and variables. A callout box highlights a specific variable, 'core\_y', which is a float with a value of 0.0. Another callout box points to the 'attributes' dictionary of a Monte-Carlo event container, listing various parameters like 'energy', 'az', 'core\_x', 'core\_y', 'h\_first\_int', and 'tel'. A third callout box points to the 'Frames' and 'Variables' panels at the bottom of the debugger, indicating that these panels allow navigation through the stack and lines of code.

```
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:",
          print("t0:",
          print("t1:",
          print("-----")

def test_read_whole_t...
    mc = MCEventContai...
    reader = SimpleHDF...

    for cont in reader...
        print(...)

if __name__ == '__main__':
    logging.basicConfig...

test_write_containe...
test_read_containe...
test_read_whole_t...
```

*Drill deep down into any data structure!*

*Attributes also appear in the code! (on mouse-over)*

*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 indicates a breakpoint on the line `reader = SimpleHDF5TableReader(str(temp_h5_file))`. A call stack window at the bottom shows the current stack frame with variables `mc` and `temp_h5_file`. Annotations in blue text boxes provide instructions on how to navigate and view variables.

```
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

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

```
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")
```

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, \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': {}}
- temp\_h5\_file = {str} 'test.h5'

Data View

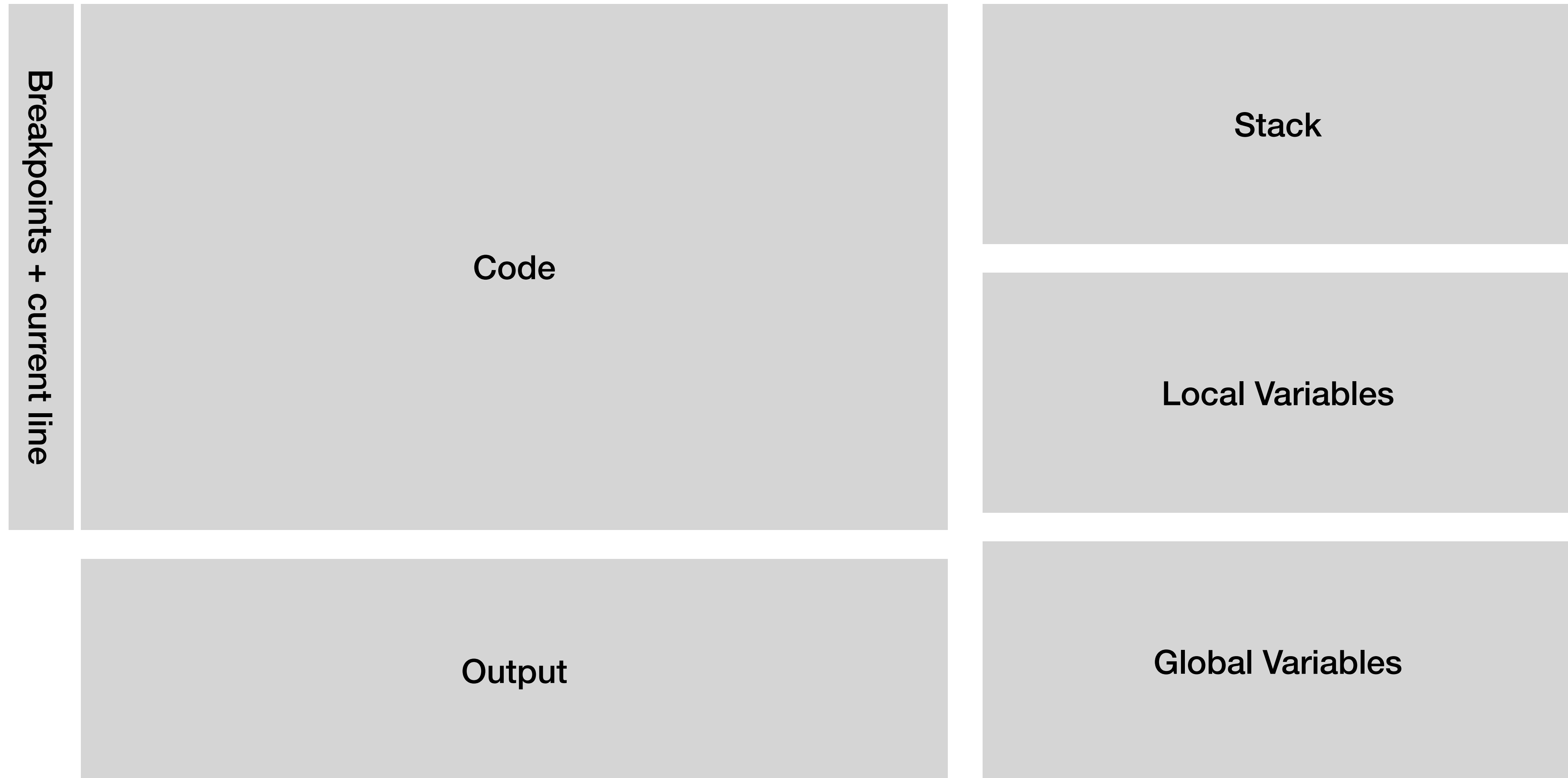
r0tel.adc\_samples x +

0	1	2	3	4
0.95997	0.98010	0.74854	0.60060	0.5954
0.68207	0.45175	0.83775	0.78088	0.6887
0.85410	0.58842	0.51579	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	0.94031	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



# GUI Debuggers: what they usually look like



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



Sometimes also a "view" of data structures

The screenshot shows the GNU Data Display Debugger (DDD) interface. At the top, there is a menu bar with options: File, Edit, View, Program, Commands, Status, Source, Data, and Help. Below the menu is a toolbar with icons for various debugging actions like Lookup, Find, Break, Watch, Print, Display, Plot, Hide, Rotate, Set, and Undisp. A command line at the top left shows '(): dev'. Below the toolbar is a row of buttons: Run, Interrupt, Step, Stepi, Next, Nexti, Until, Finish, Cont, Kill, Up, Down, Undo, Redo, Edit, and Make.

The main window displays a graph of data structures. It consists of several boxes connected by arrows. Box 3: device\_name contains '0x27920 "fd1"'. Box 4: dev contains '0x27900'. A box labeled 'disk' contains 'disk = 0x278d0' and 'net = 0x0'. A box for 'fd1' contains fields: name = 0x278b0 "fd1", dev = 0x11860, total\_sectors = 2880, has\_partitions = 0, id = 1, partition = 0x0, read\_hook = 0, and data = 0x27880. An arrow labeled 'dev' points from the 'fd1' box to a box for 'biosdisk' which contains: 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>, and next = 0x0.

A 'Backtrace' window is open, showing a list of stack frames. The current frame is #0: grub\_ls\_list\_files () at ls.c:145. Other frames include #1: grub\_cmd\_ls () at ls.c, #2: grub\_script\_execute\_cm, #3: grub\_script\_execute\_cm, #4: grub\_script\_execute\_cm, #5: grub\_script\_execute\_cm, #6: grub\_script\_execute (), and #7: grub\_command\_execute ().

The main window also shows a snippet of C code with a green arrow pointing to the line 'if (! dev) goto fail;'. The code is as follows:

```
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)
```

At the bottom of the window, there is a command prompt with the following commands and output:

```
(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)
```

The status bar at the bottom left shows: 'Display 4: dev (enabled, scope grub\_ls\_list\_files, address 0x67cc4)'. The bottom right corner of the window has a small icon.

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



# VSCode Debugger (ptvsd)

Start debugging

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

File Edit Selection View Go Run Terminal Help

app.js - myExpressApp - Visual Studio Code

```
1 var createError = require('http-errors');
2 var express = require('express');
3 var path = require('path');
4 var cookieParser = require('cookie-parser');
5 var logger = require('morgan');
6
7 var indexRouter = require('./routes/index');
8 var usersRouter = require('./routes/users');
9
10 var app = express();
11
12 // view engine setup
13 app.set('views', path.join(__dirname, 'views'));
14 app.set('view engine', 'pug');
```

DEBUG CONSOLE

Filter (e.g. text, lexclude)

C:\Program Files\nodejs\node.exe .\bin\www

Debug console panel

Debug side bar

## Emacs (M-x dap-debug)

note: need to install the debugger server first  
mamba install -c conda-forge ptvsd

```
tel_id=np.int16(tel_id),
)
pnt = event.pointing.tel[tel_id]
current_pointing = (pnt.azimuth, pnt.altitude)
if current_pointing != self._last_pointing_tel[tel_id]:
    pnt.prefix = ""
    writer.write(
        f'dll/monitoring/telescope/pointing/tel-{tel_id:03d}',
        [event.trigger.tel[tel_id], pnt],
    )
    self._last_pointing_tel[tel_id] = current_pointing
table_name = self.table_name(tel_id, str(telescopes))
writer.write(
    "dll/event/telescope/trigger", [tel_index, event.trigger.tel[tel_id],
    )
    Cannot access member "tel" for type "Field"
has_sim_camera = self._is_simulation and (
    tel_id in event.simulation.tel
    and event.simulation.tel[tel_id].true_image is not None
)
if self.write_parameters:
    writer.write(
        table_name=f"dll/event/telescope/parameters/{table_name}",
        containers=[tel_index, *dll_camera.parameters.values()],
    )
if has_sim_camera:
    writer.write(
        f"simulation/event/telescope/parameters/{table_name}",
        [
            tel_index,
            *event.simulation.tel[tel_id].true_parameters.values(),
        ],
    )
if self.write_images:
    # note that we always write the image, even if the image quality
    # criteria are not met (those are only to determine if the parameter
    # can be computed).
    self.log.debug("WRITING IMAGES")
    writer.write(
        table_name=f"dll/event/telescope/images/{table_name}",
        containers=[tel_index, dll_camera],
    )
if self._is_simulation and has_sim_camera:
    writer.write(
        f"simulation/event/telescope/images/{table_name}",
        [tel_index, event.simulation.tel[tel_id]],
    )
def _write_dll_telescope_events(
    self, writer: TableWriter, event: ArrayEventContainer
```

Locals

- current\_pointing: (<Quantity 0. rad>, <Quanti...
- dll\_camera: ctapipe.containers.DLLCamer...
- container\_prefix: 'dllcamera'
- fields: {'image': Numpy array of co...
- image: array([-0.08995 , -3.94485...
- image\_mask: array([[False, False, False...
- meta: {}
- parameters: ctapipe.containers.ImagePar...
- peak\_time: array([ 69.43996 , 77.4022...
- prefix: ''
- event: ctapipe.containers.ArrayEve...
- has\_sim\_camera: False
- pnt: ctapipe.containers.Telescop...
- self: <ctapipe.io.datawriter.Data...
- table\_name: 'tel\_021'

Python: Current File

Python: Current File<1>

MainThread Breakpoint

Python 3.8.8

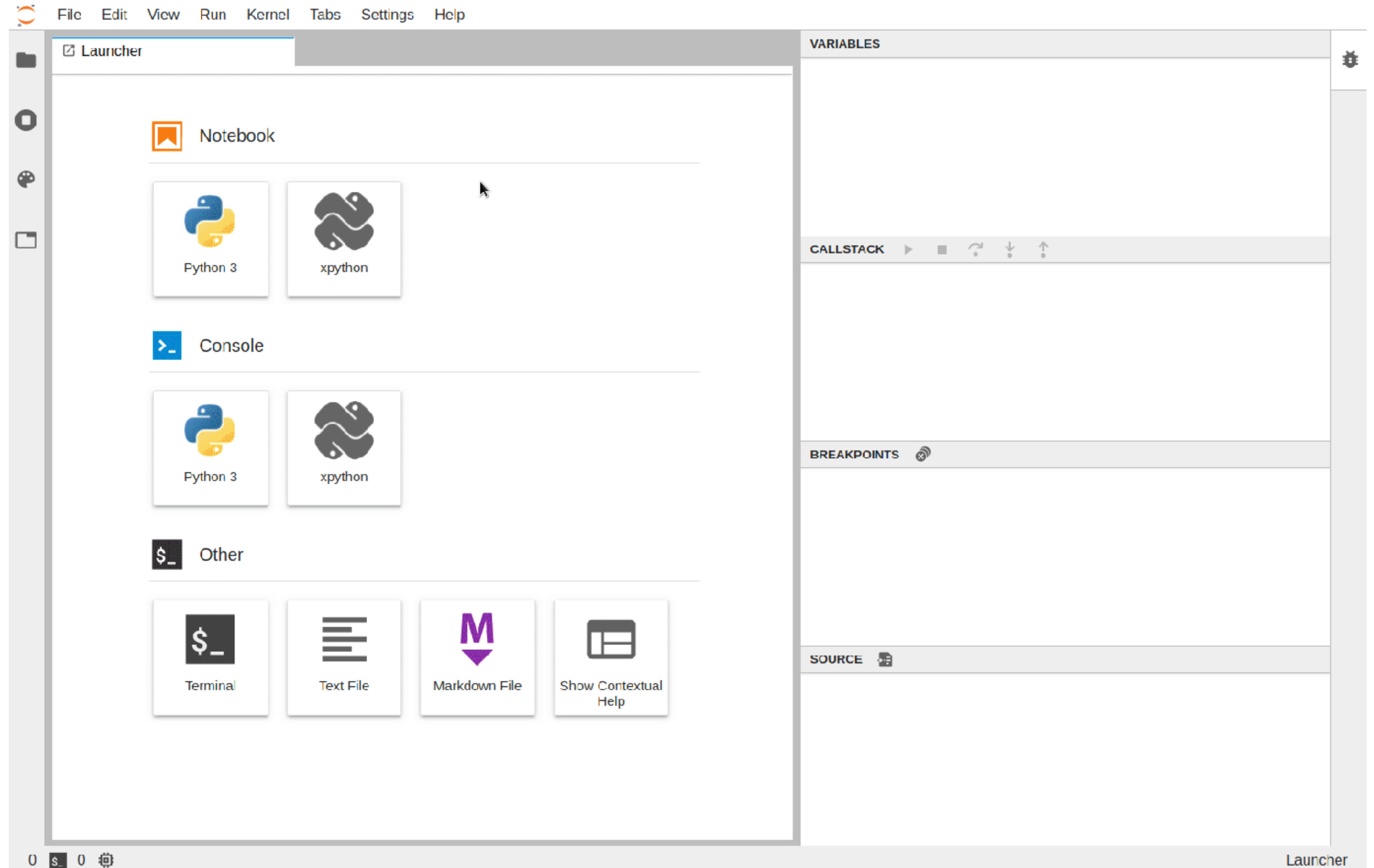


# Newish option: Jupyter-lab debugger extension

<https://github.com/jupyterlab/debugger>

## Caveat:

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



The screenshot displays the JupyterLab Launcher interface. At the top, a menu bar includes 'File', 'Edit', 'View', 'Run', 'Kernel', 'Tabs', 'Settings', and 'Help'. The main area is titled 'Launcher' and is divided into three sections: 'Notebook', 'Console', and 'Other'. Each section contains two kernel options: 'Python 3' (represented by the Python logo) and 'xpython' (represented by a black Python logo). The 'Other' section includes 'Terminal', 'Text File', 'Markdown File', and 'Show Contextual Help'. On the right side, there are three panels: 'VARIABLES', 'CALLSTACK', and 'BREAKPOINTS', all of which are currently empty. The 'SOURCE' panel is also visible at the bottom right. The bottom status bar shows '0 s 0' and a gear icon, with the word 'Launcher' in the bottom right corner.



# Newish option: Jupyter-lab debugger extension

<https://github.com/jupyterlab/debugger>

## Caveat:

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

The screenshot displays the JupyterLab Launcher interface. At the top, a menu bar includes 'File', 'Edit', 'View', 'Run', 'Kernel', 'Tabs', 'Settings', and 'Help'. The main area is titled 'Launcher' and is divided into three sections: 'Notebook', 'Console', and 'Other'. Each section contains two kernel options: 'Python 3' (represented by the Python logo) and 'xpython' (represented by a black Python logo). The 'Other' section includes 'Terminal', 'Text File', 'Markdown File', and 'Show Contextual Help'. On the right side, there are three panels: 'VARIABLES', 'CALLSTACK', and 'BREAKPOINTS', all of which are currently empty. The 'SOURCE' panel is also empty. At the bottom right, the word 'Launcher' is visible.



# demo

**Debugging with notebooks/ipython**

**Debugging with pdb**

**Debugging with a GUI (PyCharm)**





**ESCAPE**

European Science Cluster of Astronomy &  
Particle physics ESFRI research Infrastructures

# 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  $\neq$  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](http://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  $\neq$  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](http://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!



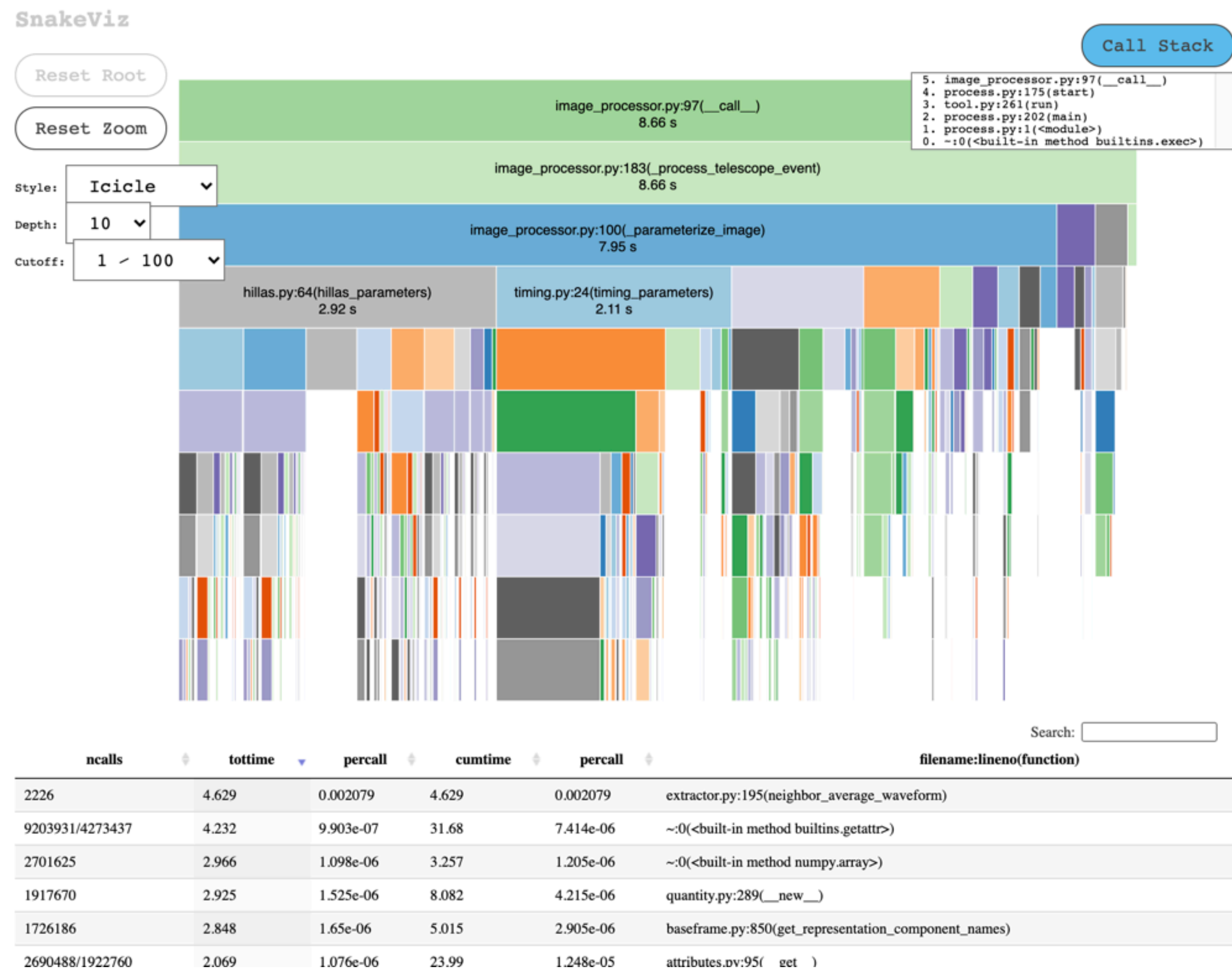
# Tip: use a gui to view stats output

## 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!





# Another stats viewer

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

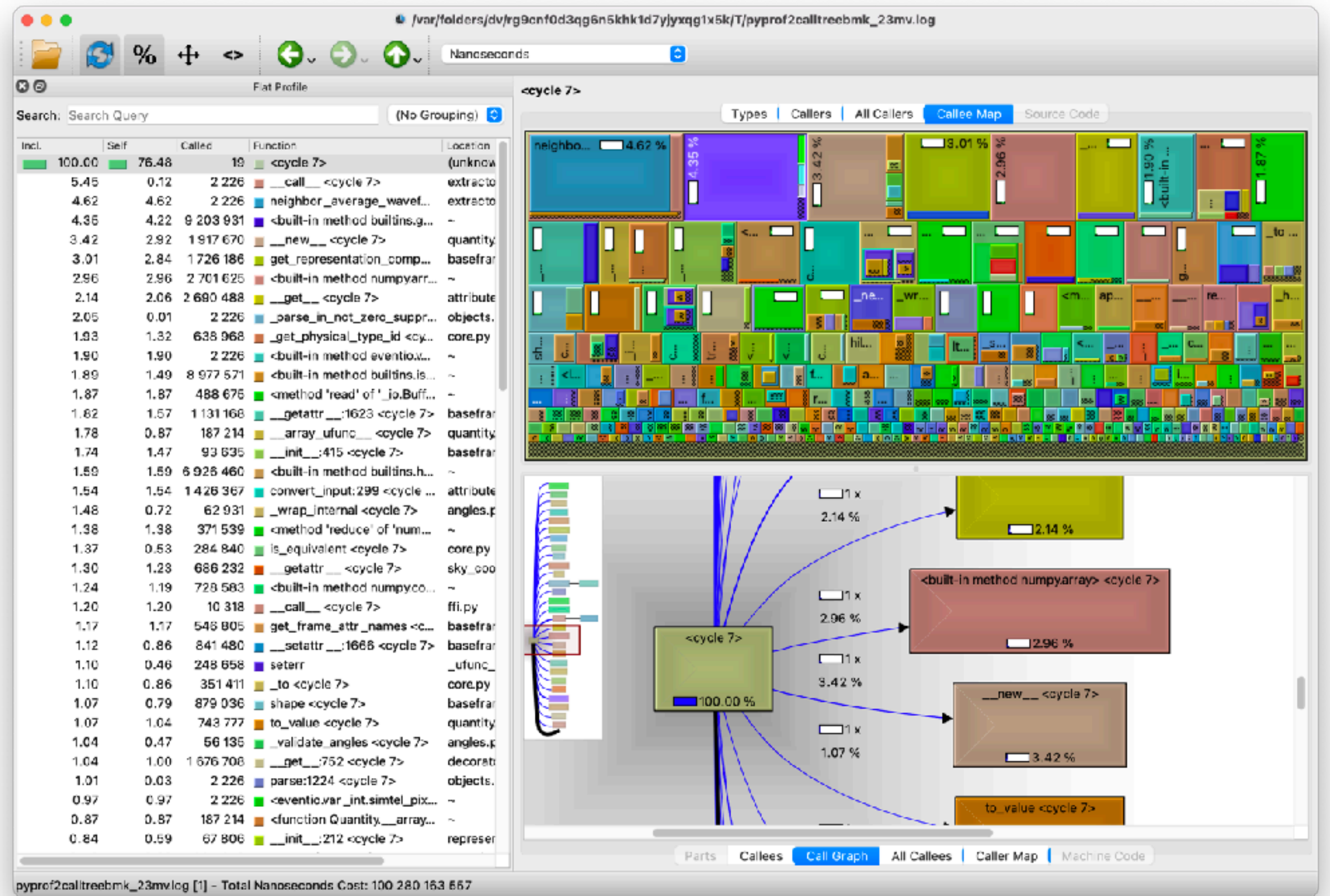
```
| % pip install pyprof2calltree  
| % pyprof2calltree -i output.pstats -k
```

This will open qCacheGrind GUI automatically

you need to first install qCacheGrind using your package manager (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

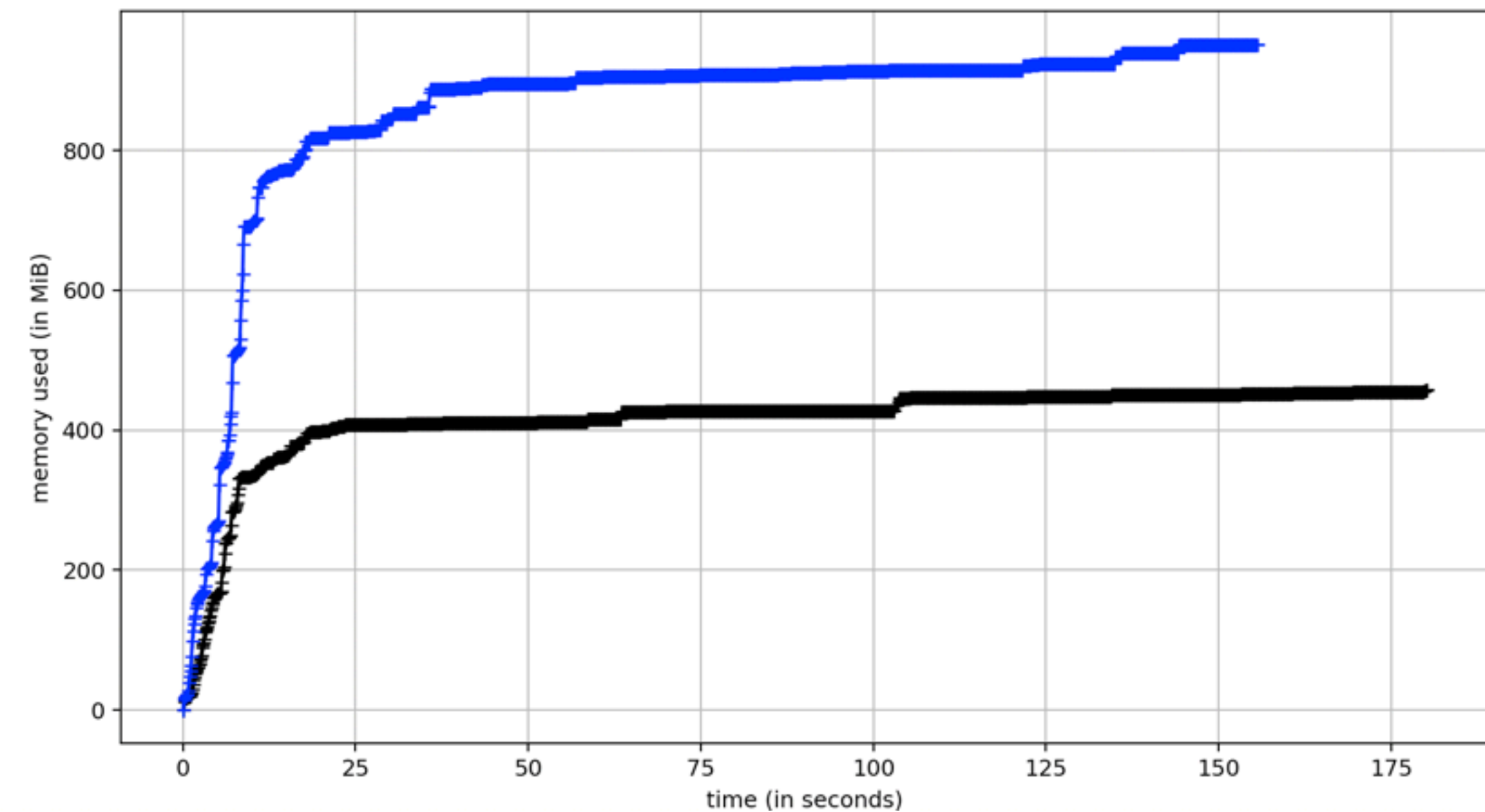
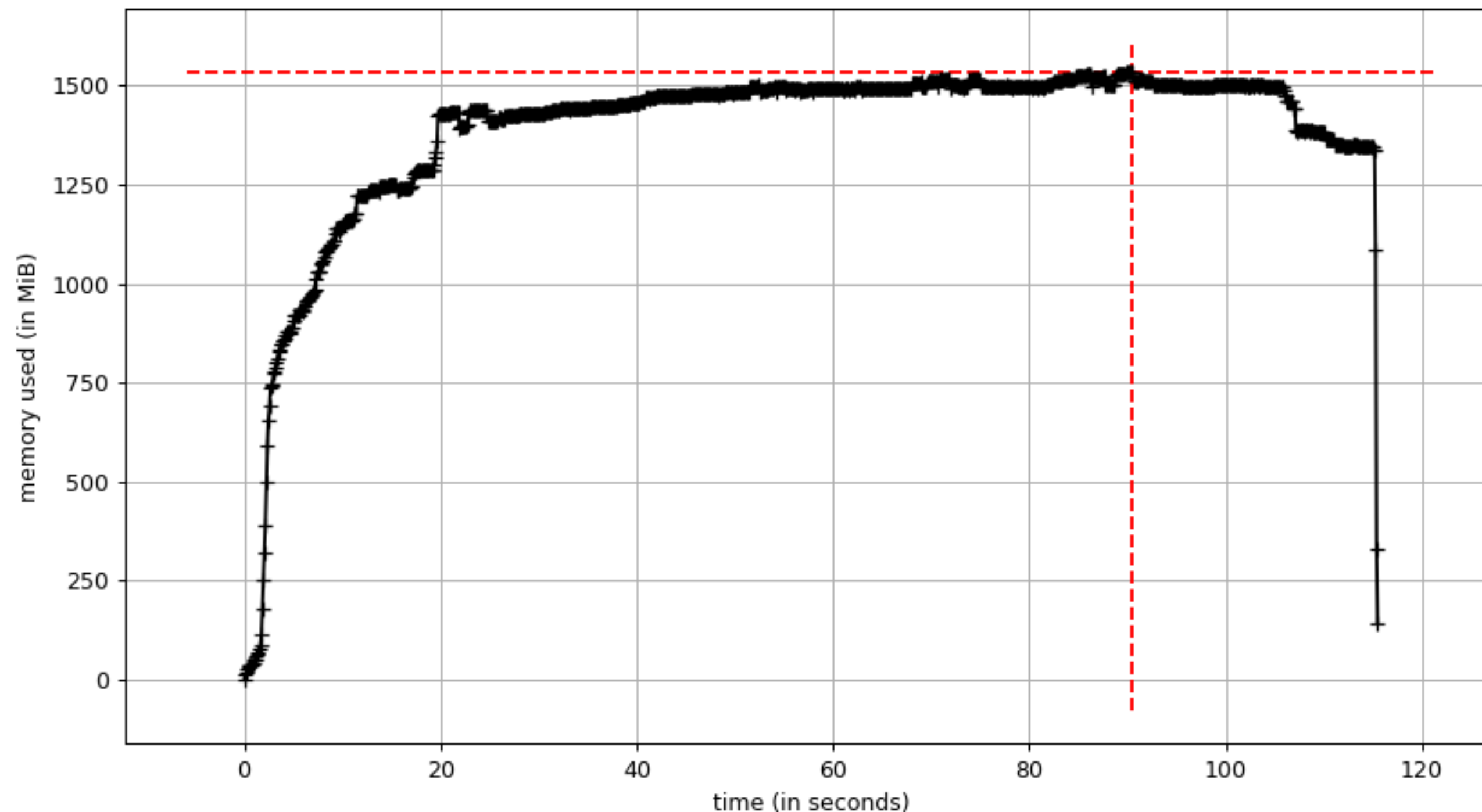


# Memory Profiling

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 < This is already in your eschool2022 environment  
| % mprof run python <script>  
| % mprof plot
```

python simple\_pipeline.py /Users/kosack/Data/CTA/Prod3/gamma.simtel.gz



# 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`

*Decorate what we want to measure (no import needed)*

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

Line #	Hits	Time	Per Hit	% Time	Line Comments
17					<b>@profile</b>
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.simt...
24	1	<b>3572651.0</b>	<b>3572651.0</b>	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





# 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**