Debugging and Profiling

Karl Kosack
CEA Paris-Saclay

ESCAPE School, June 2021

DE LA RECHERCHE À L'INDUSTRIE
cea
SACLAY

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***

→ Fundamental science institute (DRF/*IRFU*)
→ Astrophysics Department (*DAp* + *AIM*)
→ High-energy Astrophysics group (LEPCHE)

- 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** (50% of time)

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

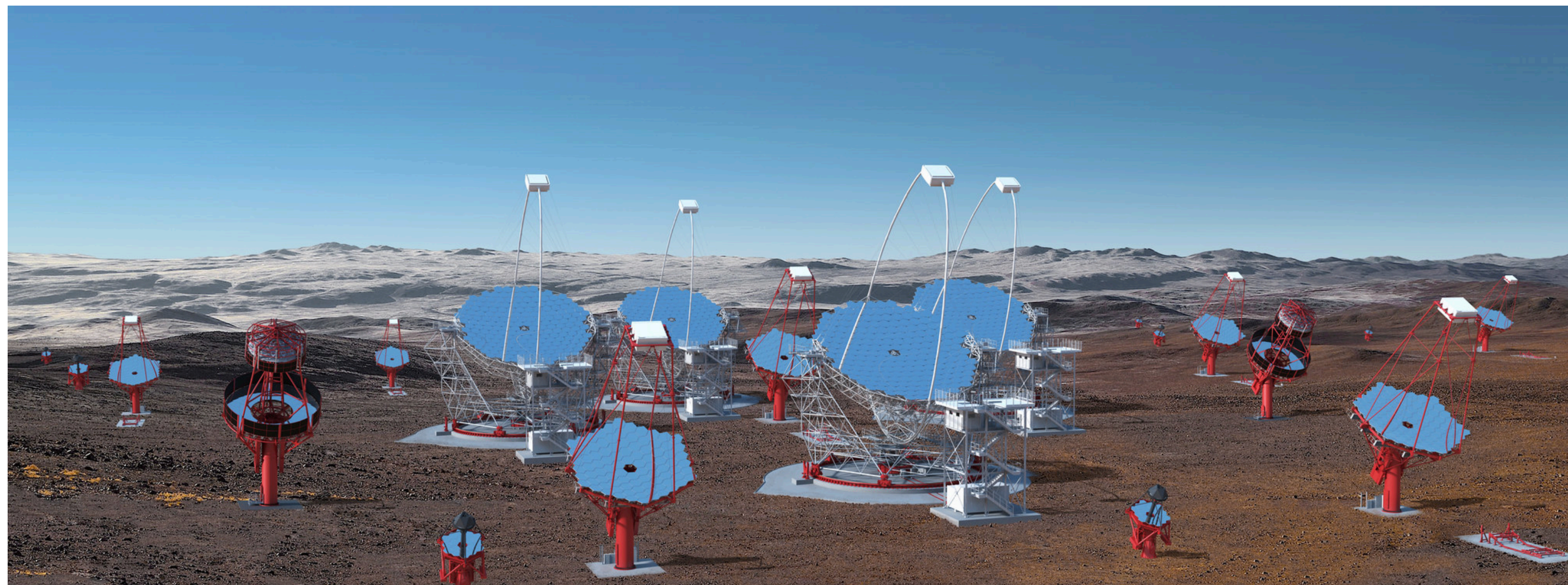
- Computational Physics
- Data analysis, processing, statistics
- Lots of scientific software development over the years...
- Was a hard-core C/C++/perl(!) user, now **essentially 100% python for 5+ years**!

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***

→ Fundamental science institute (DRF/*IRFU*)
  → Astrophysics Department (*DAp* + *AIM*)
    → High-energy Astrophysics group (LEPCHE)

- 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** (50% of time)

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 5+ years**!

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

Now that your code is debugged and you know where the slow parts are....

Optimizing your code:

- With Memoization
- With NumPy
- With Numba
- [With Cython]

Parallelizing your code:

- On a single machine with multiple cores
- On multiple machines

Debugging

ESCAPE School, June 2021

What is your current approach?

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:

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

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

```
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

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_b

function_a

main program

Global Memory

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

Local Memory

```
n = 0
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

function_b

function_a

main program

Global Memory

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

Local Memory

```
n = 0
x = 3.3
```

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

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)
```

The Call Stack

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

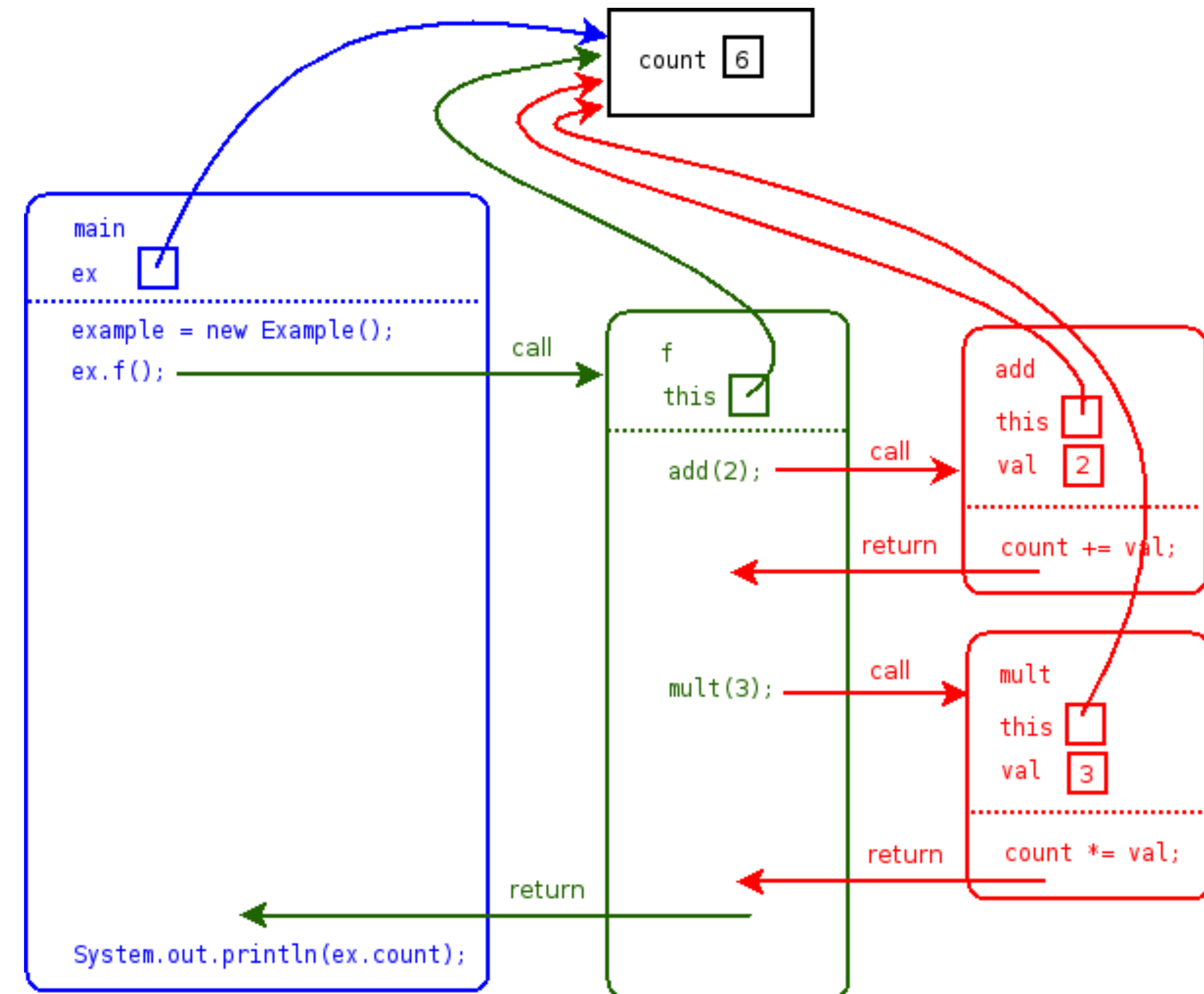


diagram from: http://faculty.ycp.edu/~dhovemey/spring2007/cs201/info/exceptionsFileIO.html

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

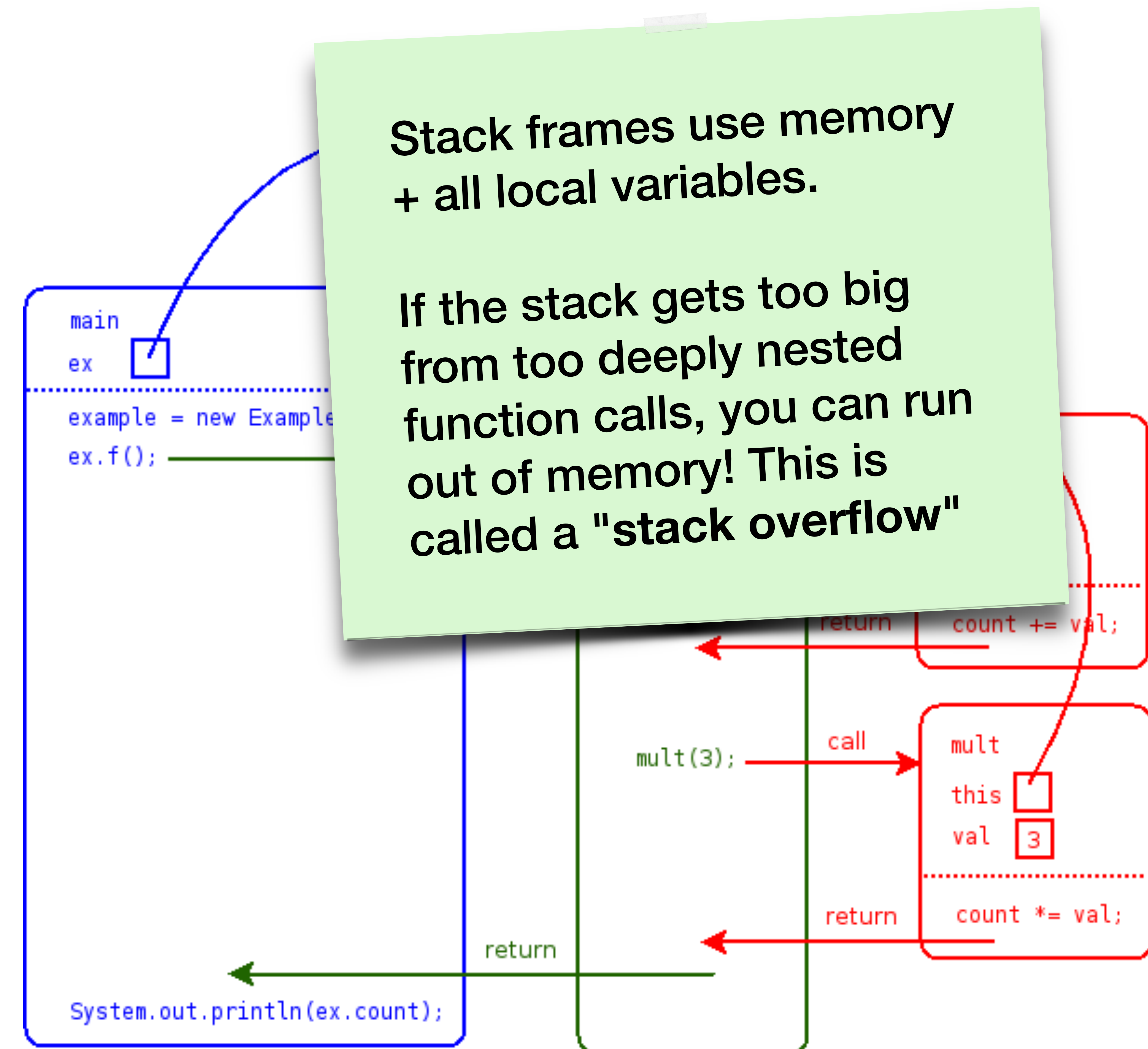


diagram from: http://faculty.ycp.edu/~dhovemey/spring2007/cs201/info/exceptionsFileIO.html

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

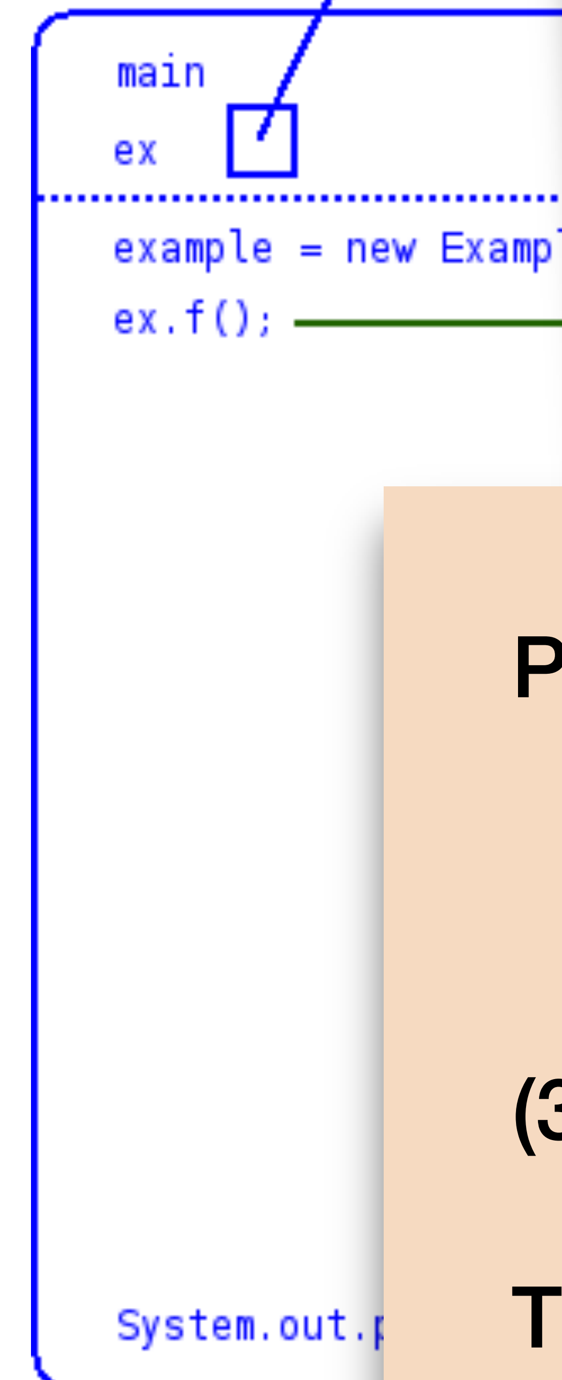


diagram from: http://facult

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:

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

pdb

GUI Debuggers:

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

GNU ddd

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
- **?** to show help!

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 -

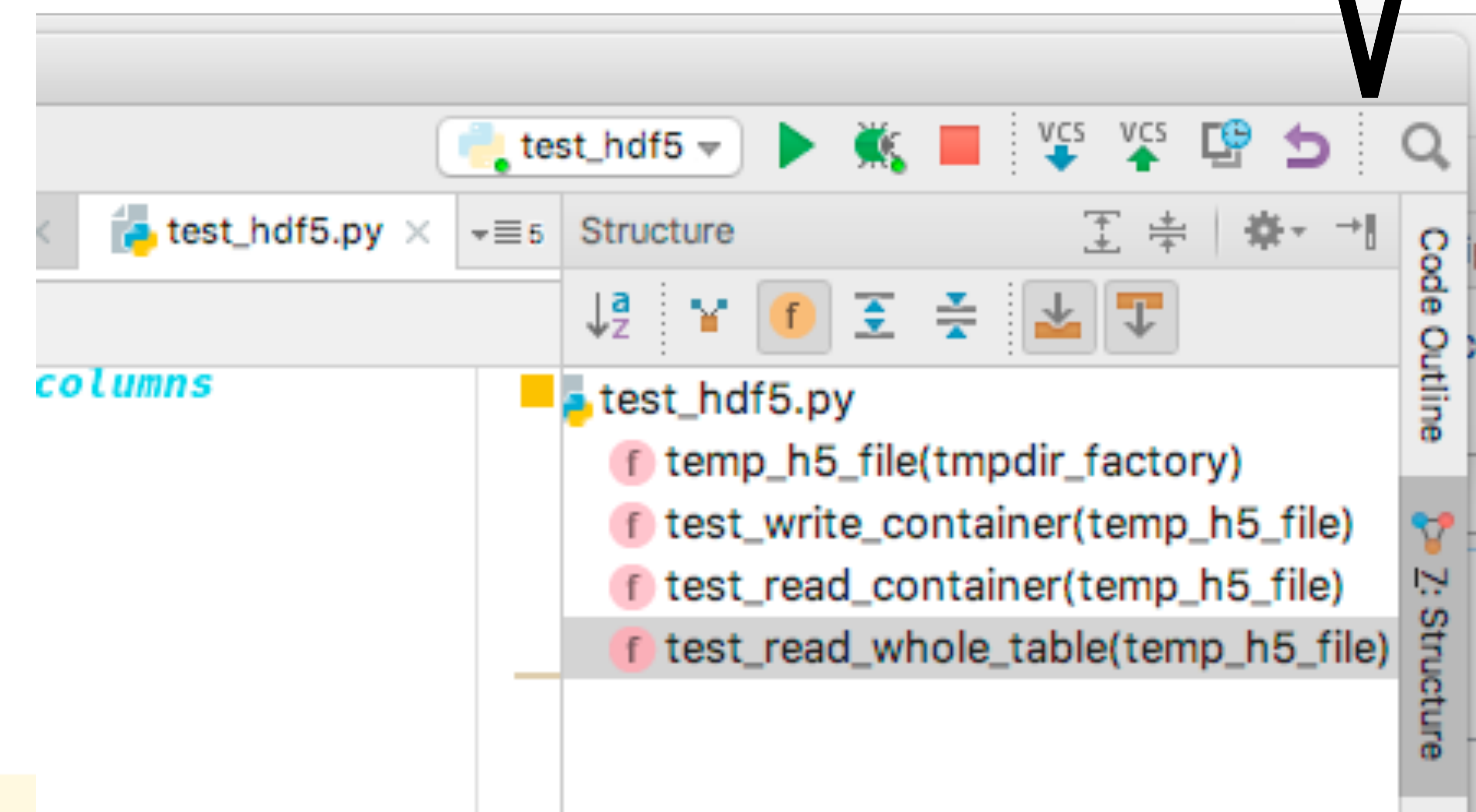
GUI Debugging

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

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)

Solution: use a GUI debugger!



Click in margin to set a breakpoint

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

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

def test_read_whole_table(temp_h5_file):

    mc = MCEventContainer()

●   reader = SimpleHDF5TableReader(str(temp_h5_file))

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

if __name__ == '__main__':

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

    test_write_container("test.h5")
```

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):   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")
```

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'

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

def test_read_whole_ta

    mc = MCEventContai

    reader = SimpleHDF

    for cont in reade
        print(

if __name__ =

    import lo
    logging.basicConf

    test_write_contai
    test_read_containe
    test_read_whole_ta
```

Drill deep down into any data structure!

```
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
energy = {float} 0.0
h_first_int = {float} 0.0
```

curre
this

...ues also appear
...t in the code!
...on mouse-over)

Debug: test_hdf5 | test_hdf5 | test_hdf5

Debugger | Console

Frames | Variables

Main...

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

```
mc = {MCEventContainer} {'alt': 0.0,\n '
temp_h5_file = {str} 'test.h5'
```

Move up and down stack or lines

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

GUI debugging

```
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__ == ...

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

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

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

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'

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):   temp_h5_file: 'test.h5'

    mc = MCEventContainer()   mc: {'alt': 0.0,\n 'az': 0.0,\n 'core_x': 0.0,\n

    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")
```

currently at this line

values also right in th (or on mo

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'

Move up and down stack or lines

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

Data View

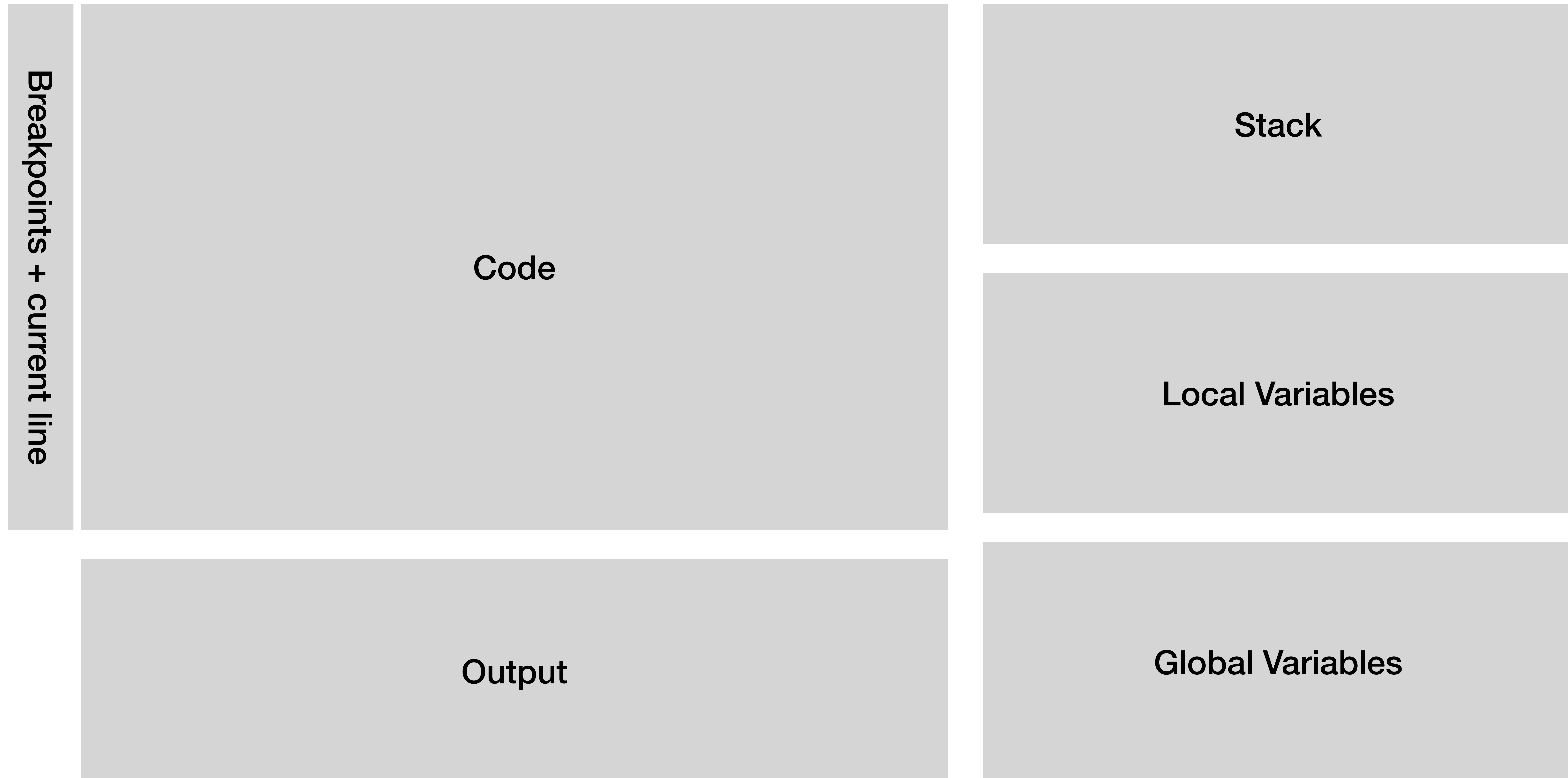
r0tel.adc_samples

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

0	1	2	3	4
0.95997	0.98010	0.74854	0.60060	0.5954
0.68207	0.45175	[illegible]	[illegible]	[illegible]
0.85410	0.58842	[illegible]	0.36246	0.2527
0.87389	0.83798	[illegible]	0.93956	0.6563
0.68928	0.53708	[illegible]	0.49141	0.6709
0.38935	0.57417	[illegible]	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

File Edit View Program Commands Status Source Data Help

(): dev

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

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

```
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
```

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

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

    return 0;
  }

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

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

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

Backtrace

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

Up Down Close Help

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

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

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

RUN Launch Program

VARIABLES

```
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');

var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'v
```

WATCH

CALL STACK

Launch Program: www [10868] RUNNING

DEBUG CONSOLE

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

Debug console panel

LOADED SCRIPTS

BREAKPOINTS

Caught Exceptions

Uncaught Exceptions

app.js 10

master Launch Program (myExpressApp) Ln 10, Col 11 Spaces: 2 UTF-8

Debug side bar

Emacs (M-x dap-debug)

note: need to install the debugger server first

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

demo

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

Profiling

ESCAPE School, June 2021

What is profiling?

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

- CPU resources (computation time)
- Memory resources

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

Identify "hotspots" in your code that may be useful to optimize (we'll talk about optimization later today!)

Speed profiling 1: in a *notebook*

Simplest method: *timeit*

- *no need to calculate start and stop times, python's standard lib has a nice module to help with that…*
- *easiest way is to use interactive* **%timeit** *magic ipython function*

DEMO NOTEBOOK

- *Usage:*

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

Why not just roll your own?

```
start = time.now()
[code]
stop = time.now()
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!*

***Note** you can also import the `timeit` module and use it similar to the magic %timeit function in non-notebook scripts*

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>
```

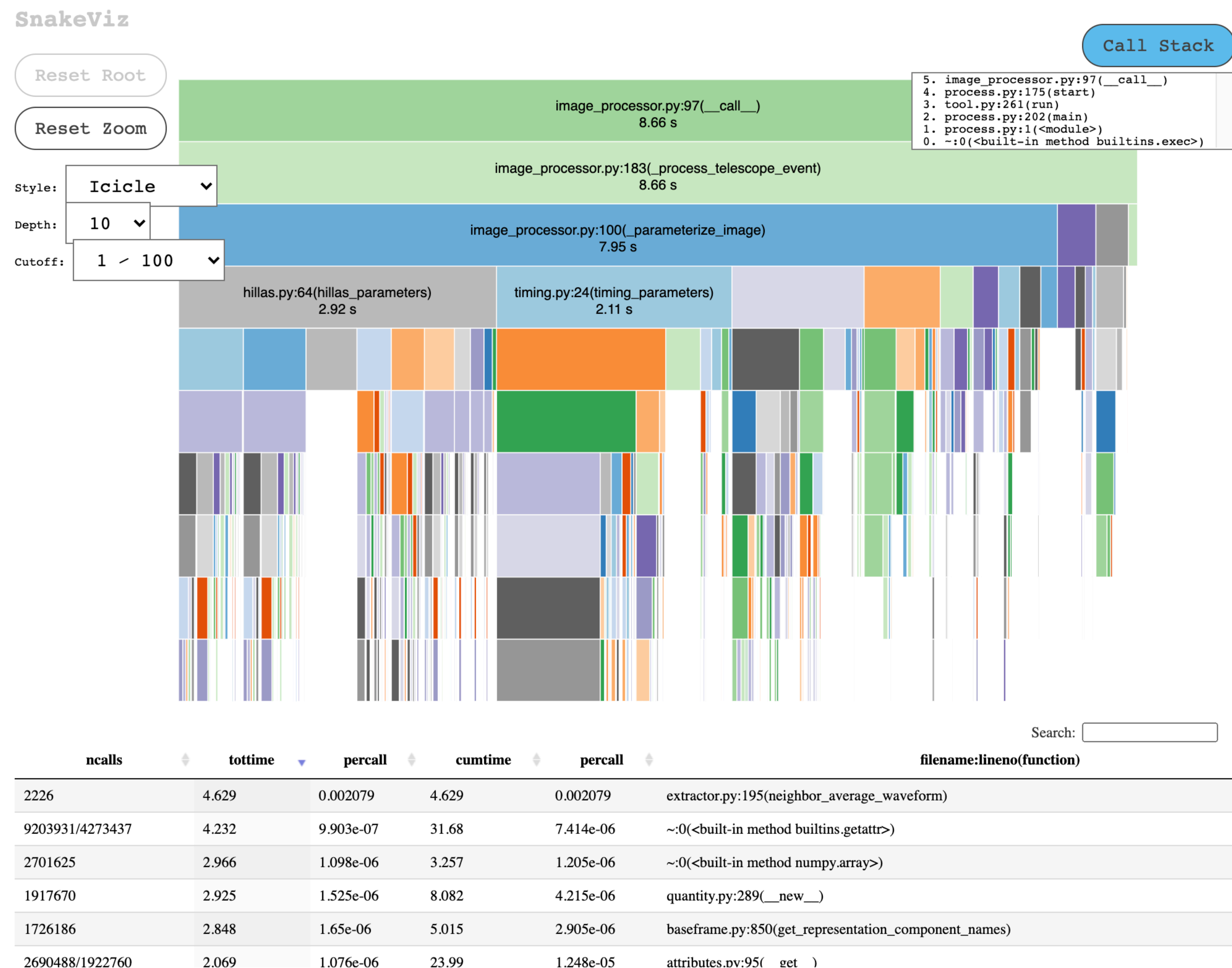
- this generates a **binary data file (*output.pstats*)** that contains statistics on **how often** and **for how long** each function was called
- There is a built-in **pstats** module that displays it using a command-line UI, but it's a bit difficult to use... but there are GUIs!

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



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__)

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
```

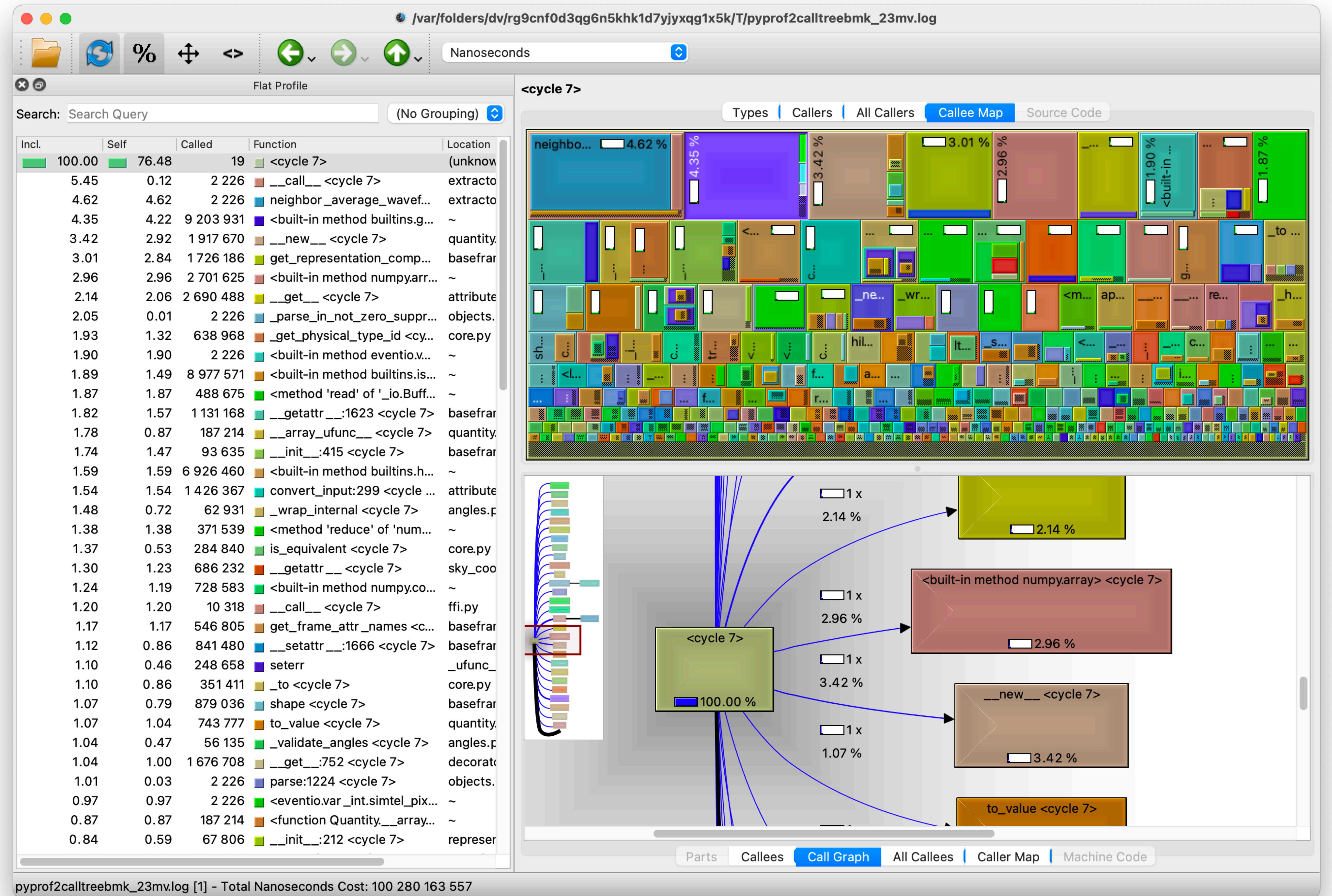
This will open qCacheGrind GUI automatically

you need to first install qCacheGrind using your package 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

Sometimes you need more detail than function-level stats…
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)
```

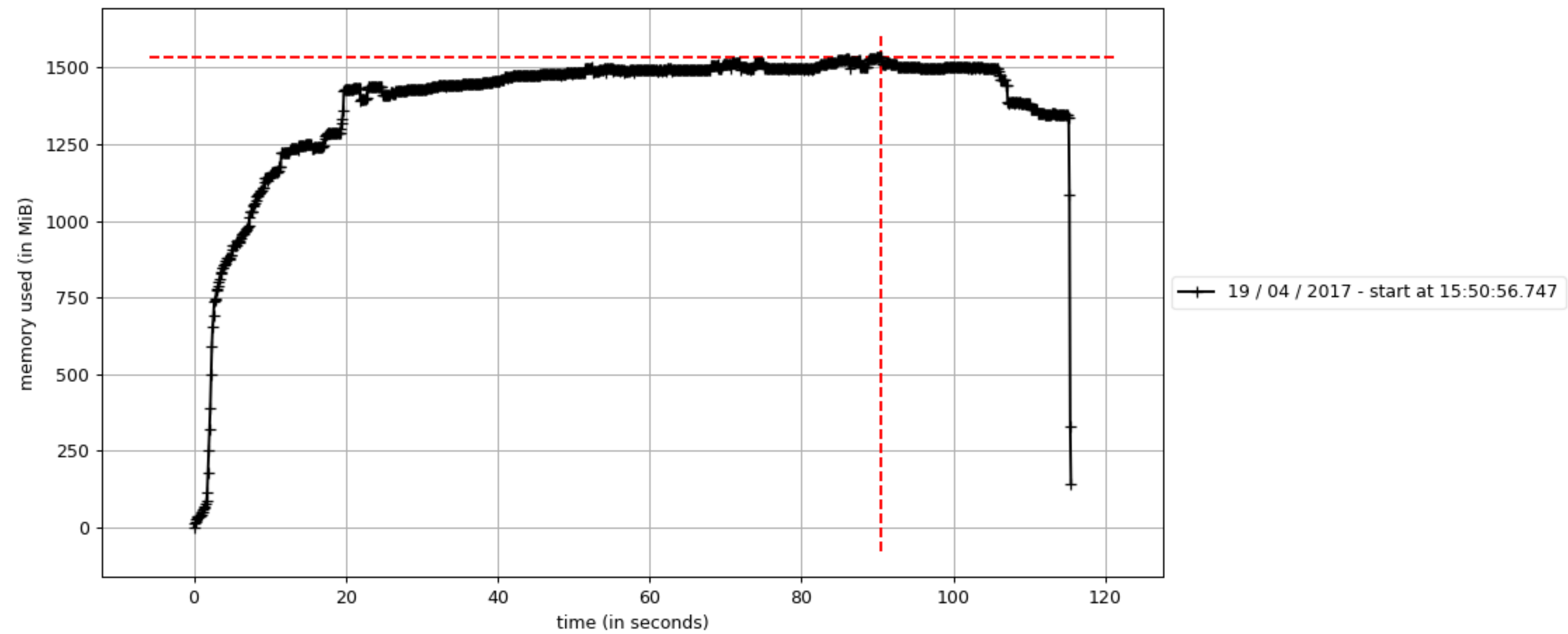
Note you can mark more than one func

```
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
```



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

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

Optimization

ESCAPE School, June 2021

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

- *Sir Tony Hoare? or Donald Knuth?*

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

- *Sir Tony Hoare?*
or Donald Knuth?

From a 1974 article on why GOTO statements are good

Why optimize?

Why optimize?

However… once code is working, you do want it to be efficient!

- want a balance between usability/readability/correctness and speed/memory efficiency
- These are not always both achievable, so err on the side of *usability*

Why optimize?

However… once code is working, you do want it to be efficient!

- want a balance between usability/readability/correctness and speed/memory efficiency
- These are not always both achievable, so err on the side of *usability*

Some things:

- Python is interpreted (though some compilation happens), and can therefore be *slow*
- For-loops in particular are 100 - 1000x slower than C loops…
- There are some nice ways to speed up code, however, and get close to low-level language speed

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!

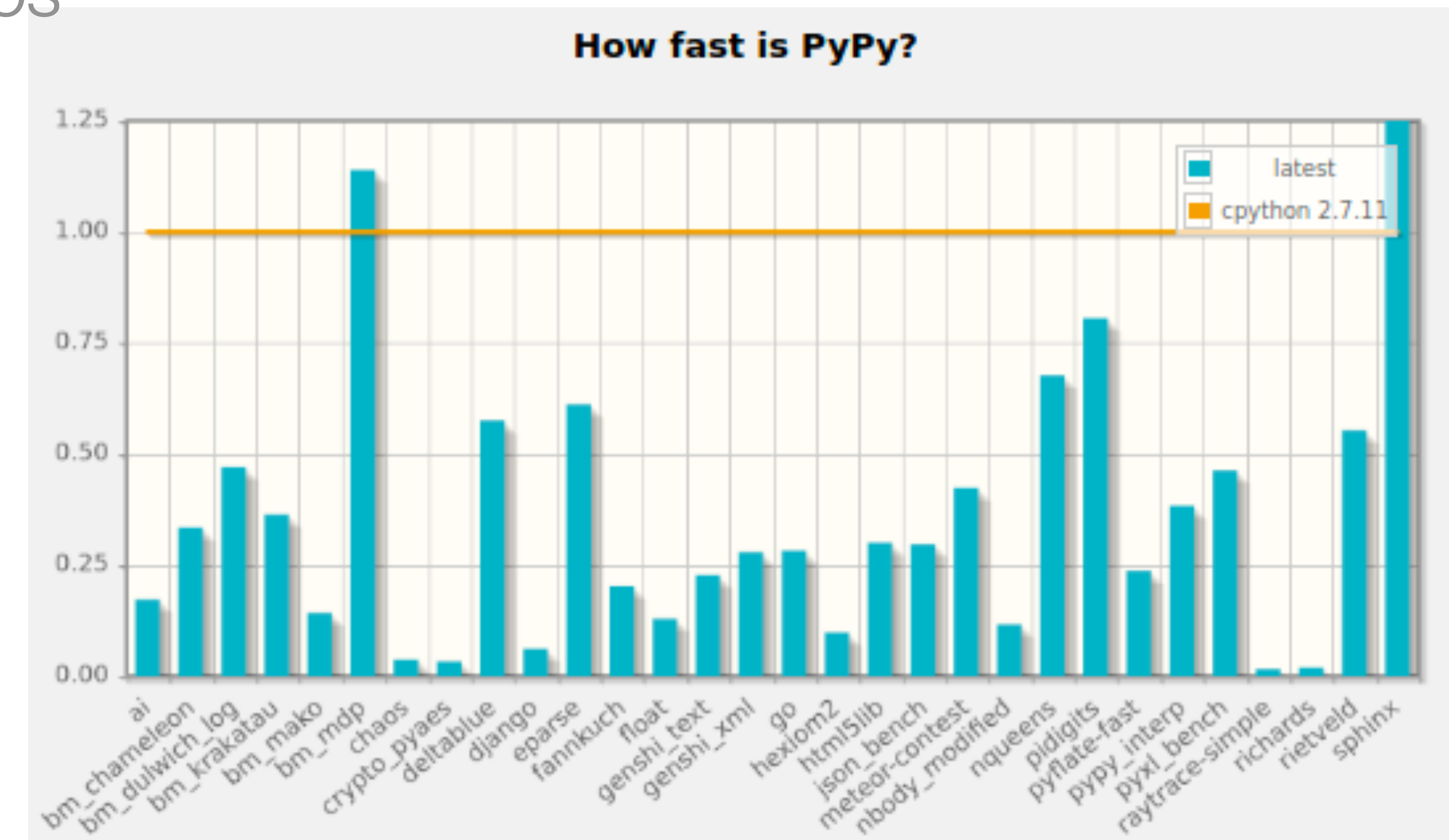
Some notes on PyPy

Advantages of PyPy:

Just In time → compiled when used, not before

A compiler framework similar to GCC, the default on macOS

- all PyPy code is **JIT**-compiled with **LLVM**
- support for most (but not all) of NumPy
- some support for C-extensions, but not all c-code can be run yet
- supports (so far) Python language up to version 3.7.9



Disadvantages:

- Works well speeding-up pure-python code, but scientific code is often a mix of Numpy/scipy/c-code: *it's often slower than CPython!*
- C-extensions not fully supported

But... there is a lot you can do to make your python code faster *now.*

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***!)

Speeding up code 1: Memoization

Basic principle: don't recompute things you computed already!

Instead, compute them once, and just return the pre-computed result when asked. (trade memory for speed)

The hard way:

- keep a dictionary keyed by the input to a function with the output as the value. If the key exists, return the value:

```
RESULTS_CACHE = {}

def memoized_compute(x):
    if x in RESULTS_CACHE:
        return RESULTS_CACHE[x]
    result = do_some_large_computation(x)
    RESULTS_CACHE[x] = result
```

It works, but is ugly and not very *pythonic*...

Also if there are many values of x, you will use a lot of memory

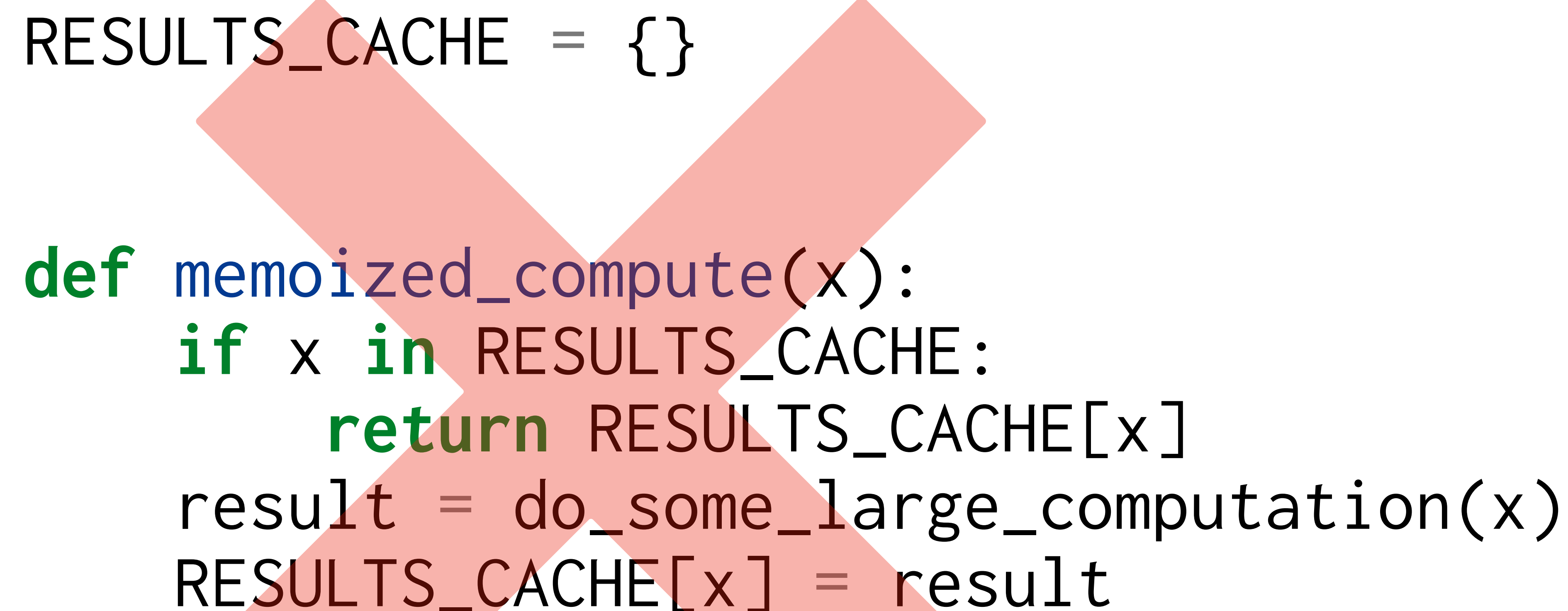
Speeding up code 1: Memoization

The better way: as usual, python already has you covered!

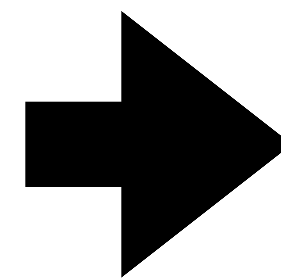
- use **functools.lru_cache**
  → built-in memoization as a decorator
- Specify (roughly) the expected maximum size of the cache
  - ➤ it will still work if you go over it, but just not be as efficient
- It uses (a hash of) all inputs to the function as the key

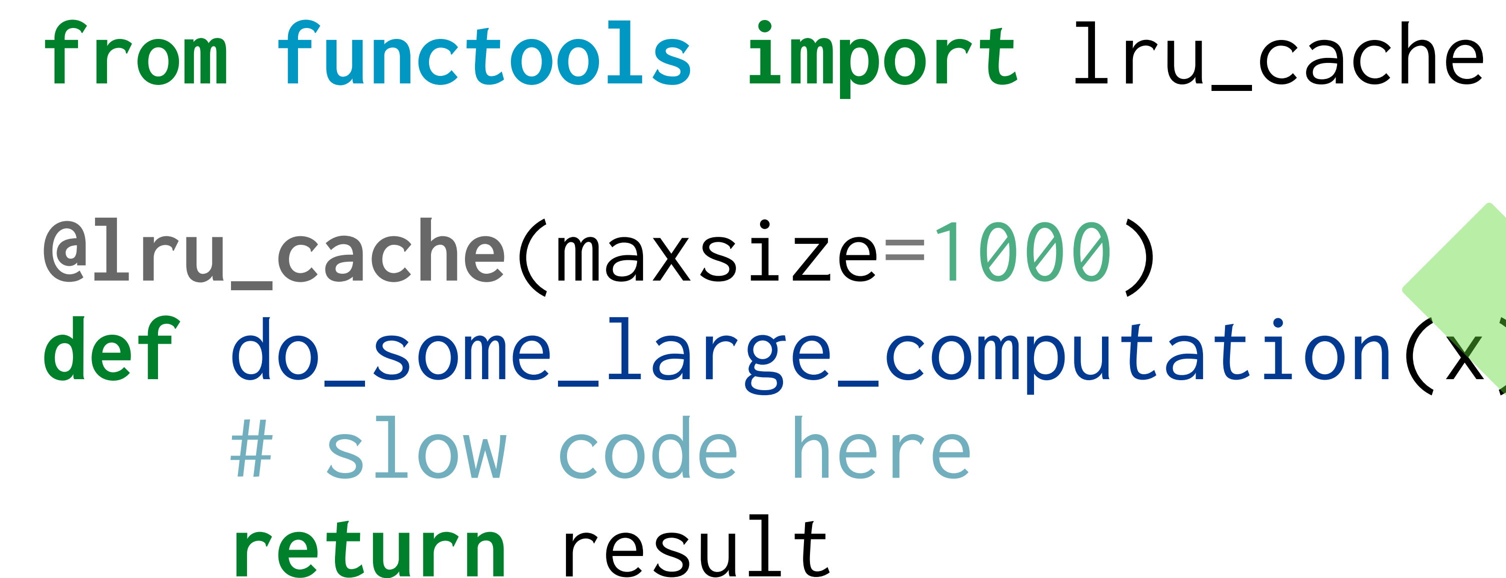
LRU: Least Recently Used:
Throw away cached items that were not accessed recently, if memory gets slim

(one method for caching, there are many others)



```
RESULTS_CACHE = {}


def memoized_compute(x):
    if x in RESULTS_CACHE:
        return RESULTS_CACHE[x]
    result = do_some_large_computation(x)
    RESULTS_CACHE[x] = result
```





```
from functools import lru_cache

@lru_cache(maxsize=1000)
def do_some_large_computation(x):
    # slow code here
    return result
```

Speeding up code 2: Numpy

For-loops are slow! (in pure python)

Use NumPy vector operations as much as possible → they are optimized already**!**

- don't call a function on many small pieces of data when you can **call it on an array all at once**
- numpy is implemented in C & Fortran *and* it uses fast numerical libraries, optimized for your CPU (e.g. Intel Math Kernel Library MKL, BLAS, LAPACK etc)
- usually just vectorizing your code to avoid some for-loops, will give you great performance.
  - bad:

```
for ii in range(100):
    x = ii*0.1
    y[ii] = f(x)
```

  - Good:

```
x = np.linspace(0,10,100)
y = f(x)
```

Speeding up code 2: Numpy

For-loops are slow! (in pure python)

Use NumPy vector operations as much as possible → they are optimized already!

- don't call a function on many small pieces of data when you can **call it on an array all at once**
- numpy is implemented in C & Fortran *and* it uses fast numerical libraries, optimized for your CPU (e.g. Intel Math Kernel Library MKL, BLAS, LAPACK etc)
- usually just vectorizing your code to avoid some for-loops, will give you great performance.
  - bad:

```
for ii in range(100):
    x = ii*0.1
    y[ii] = f(x)
```

  - Good:

```
x = np.linspace(0,10,100)
y = f(x)
```

This requires practice, and feels very strange at first if you are coming from C programming!

Take some time to look through the *NumPy* and *SciPy* **API documentation** - there are tons of interesting functions to help you!

Speeding up code 3: Numba

Takes python code and *directly* uses introspection to compile it with LLVM

- Pretty **automatic**, *but doesn't always help*! Still need code written in a way that can be optimized (for-loops are actually good here, it can't do much with numpy operations since they are already compiled code)
- Can generate **NumPy "ufuncs"** directly (function that works on scalars but is run on all elements of an array), which are too slow to write in python normally.
- Can even compile to **GPU** code for nVidia *CUDA* and AMD *ROC* GPUs!

```
from numba import jit
from numpy import arange

# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when function is called.
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result

a = arange(9).reshape(3,3)
print(sum2d(a))
```

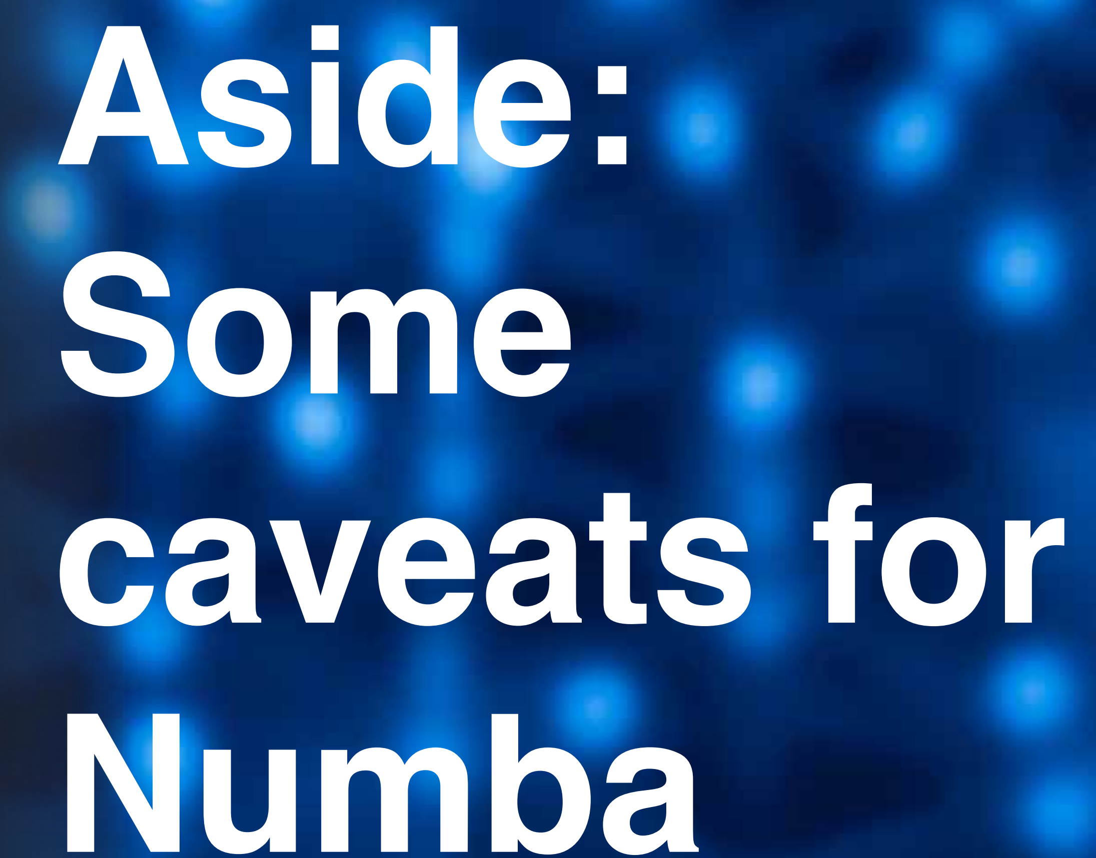
just add this decorator, and it's magic (nearly)

Numba operates in two modes:

- No-Python Mode:
  - ➤ gives large performance boost
  - ➤ but only supports basic python types and a subset of numpy/scipy operations
- Object Mode
  - ➤ fall-back if No Python mode fails
  - ➤ supports any python object
  - ➤ but gives little or not speed up in most situations

Tip:

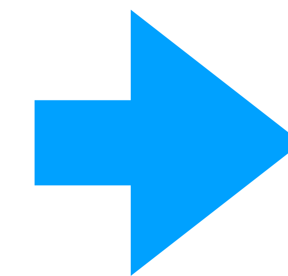
- To force it to use No-Python mode
  - ➤ set *nopython=True* in the options
  - ➤ better: use **@njit**
- **@njit will fail if the code cannot be optimized by numba, and it will tell you why!**
- **There is some discussion that @njit will become the default in the future**



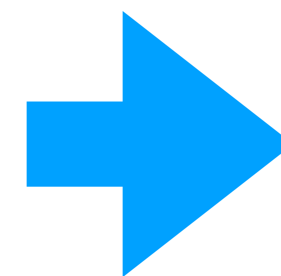
More numba caveats:

note that you need to "jit" not only the parent function, but any function that it calls that needs to be sped up.   Otherwise, only Object Mode can work!

```python
from timeit import default_timer as timer
from matplotlib.pylab import imshow, jet, show, ion
import numpy as np

from numba import jit


@jit
def mandel(x, y, max_iters):
    """
    Given the real and imaginary parts of a
    complex number,
    determine if it is a candidate for membership
    in the Mandelbrot
    set given a fixed number of iterations.
    """
    i = 0
    c = complex(x,y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 4:
            return i

    return 255
```



```python
@jit
def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

    return image
```



```python
image = np.zeros((500 * 2, 750 * 2), dtype=np.uint8)
s = timer()
create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
e = timer()
print(e - s)
imshow(image)
```

example from the Numba docs

Numba with NumPy

Numba supports a large number of NumPy functions (and even some scipy)**:**

- It does not actually call NumPy code!
- it *re-implements* it in a way that is compilable with LLVM.

  https://numba.pydata.org/numba-doc/dev/reference/numpysupported.html

So what is the point? Isn't NumPy really optimized already?

- Minimize intermediate results!
  - numpy operations often have to allocate memory for data that is not needed in the end:

```
x = np.arange(1000)
result = A * x**2 + B * x + C
```

in C, you might do this all in one loop, with no extra memory needed:

```
for (i=0; i<x.size; i++) {
    result[i] = A*x[i]*x[i] + B*x[i] + C;
}
```

- More control over parallelization (See next lecture)

Advanced Numba

Numba includes a lot of advanced features and options to *jit* that can help speed things up

- e.g. specify the input and output type mapping, rather than infer it
- Easy NumPy Ufunc generation with *vectorize* and *guvectorize (generalized)*
  - ➤ *e.g. let you write code that operates on 1D array, and broadcast it to N-dimensional arrays*
- Options like target='GPU' for producing CUDA code or similar
- Parallelization onto multiple threads with parallel=True (see next lecture)

```python
import numpy as np

from numba import guvectorize

@guvectorize(['void(float64[:], intp[:], float64[:])'], '(n),()->(n)')
def move_mean(a, window_arr, out):
    window_width = window_arr[0]
    asum = 0.0
    count = 0
    for i in range(window_width):
        asum += a[i]
        count += 1
        out[i] = asum / count
    for i in range(window_width, len(a)):
        asum += a[i] - a[i - window_width]
        out[i] = asum / count

arr = np.arange(20, dtype=np.float64).reshape(2, 10)
print(arr)
print(move_mean(arr, 3))
```

example from the Numba docs

Summary

Write good clean code first!

Identify bottlenecks in speed and memory with profiling tools

- don't worry so much about things that are not called often!
- try to narrow it down to the most critical parts of code

Use numpy, cython, numba or other technologies to improve the bottleneck

- try not to obfuscate the code to achieve speed! Readability still counts.

Demo