Combining Embedded & Interactive Python in The LLDB Debugger

Caroline Tice

EuroPython 2011

Florence, Italy

Outline

- What is LLDB?
- Python in LLDB
- Particular Problems (& Solutions)
- Example Using Python to Debug Problem
- Questions

Outline

- What is LLDB?
- Python in LLDB
- Particular Problems (& Solutions)
- Example Using Python to Debug Problem
- Questions

What is LLDB?

What is LLDB?

- Part of LLVM Project (www.llvm.org)
  - Open source compiler & tool technologies

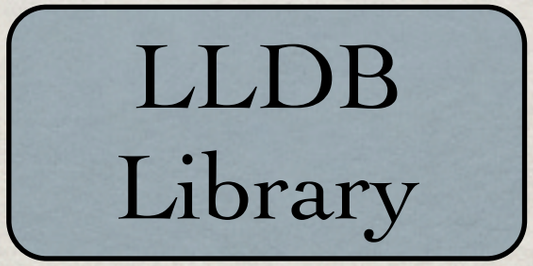
What is LLDB?

- Part of LLVM Project (www.llvm.org)
  - Open source compiler & tool technologies
- LLDB = LLVM's Debugger

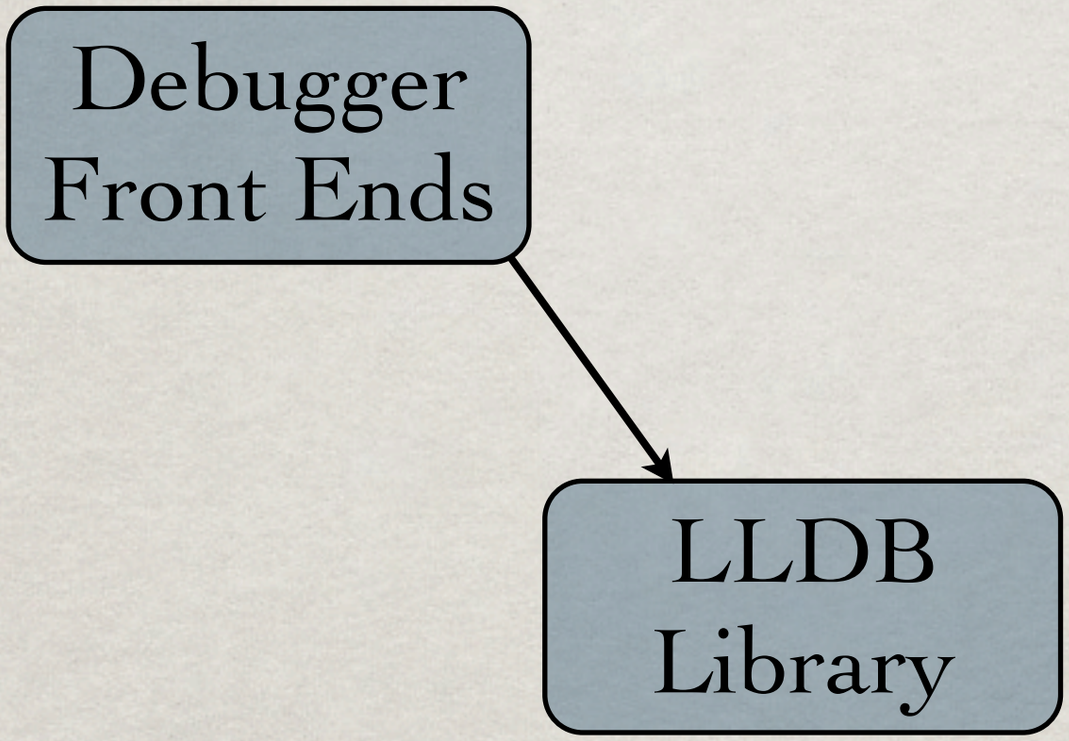
What is LLDB?

- Part of LLVM Project (www.llvm.org)
  - Open source compiler & tool technologies
- LLDB = LLVM's Debugger
- LLDB: A debugger tool library
  - Provides debugger-related services

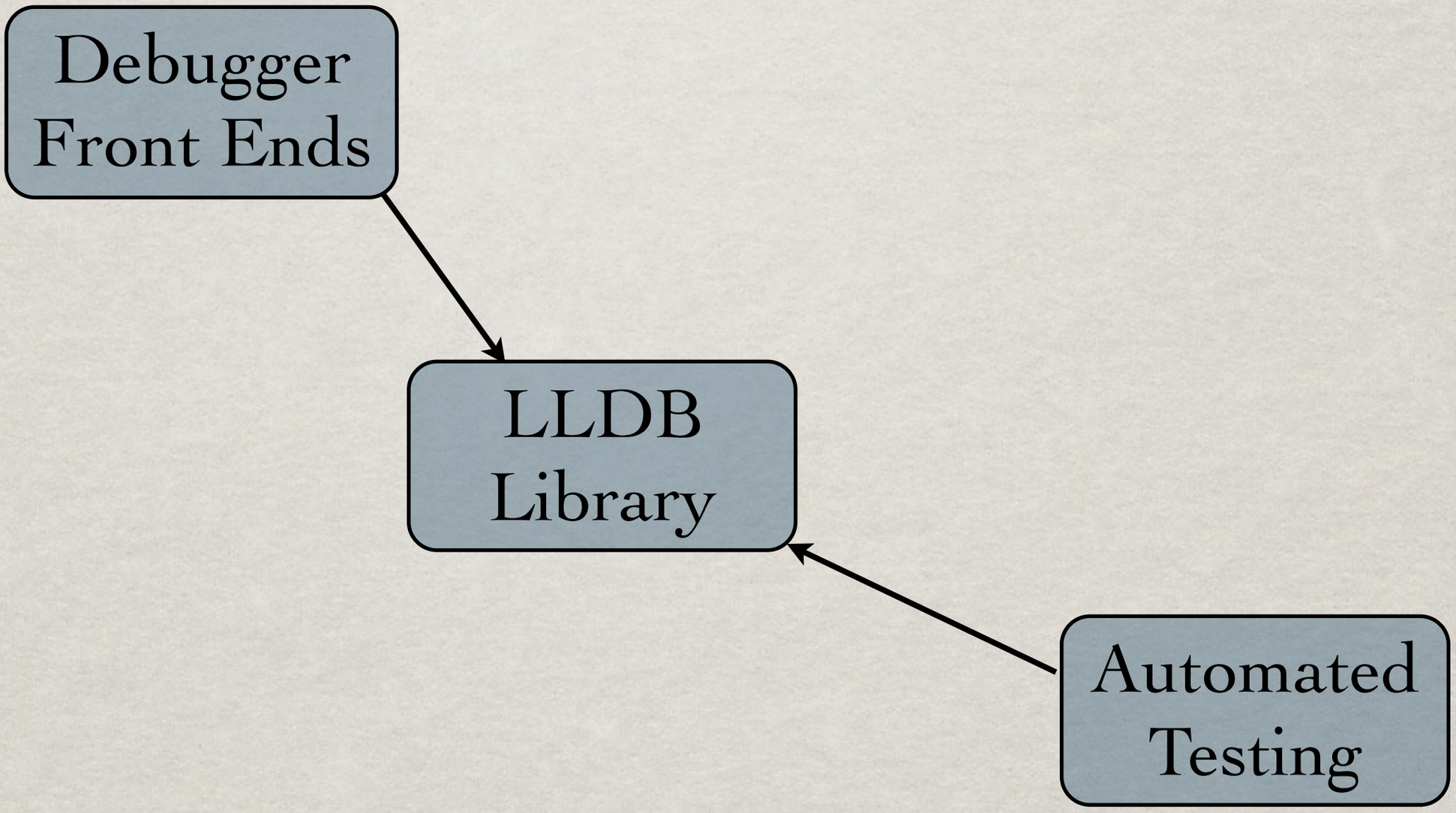
Some Uses Of the LLDB Library



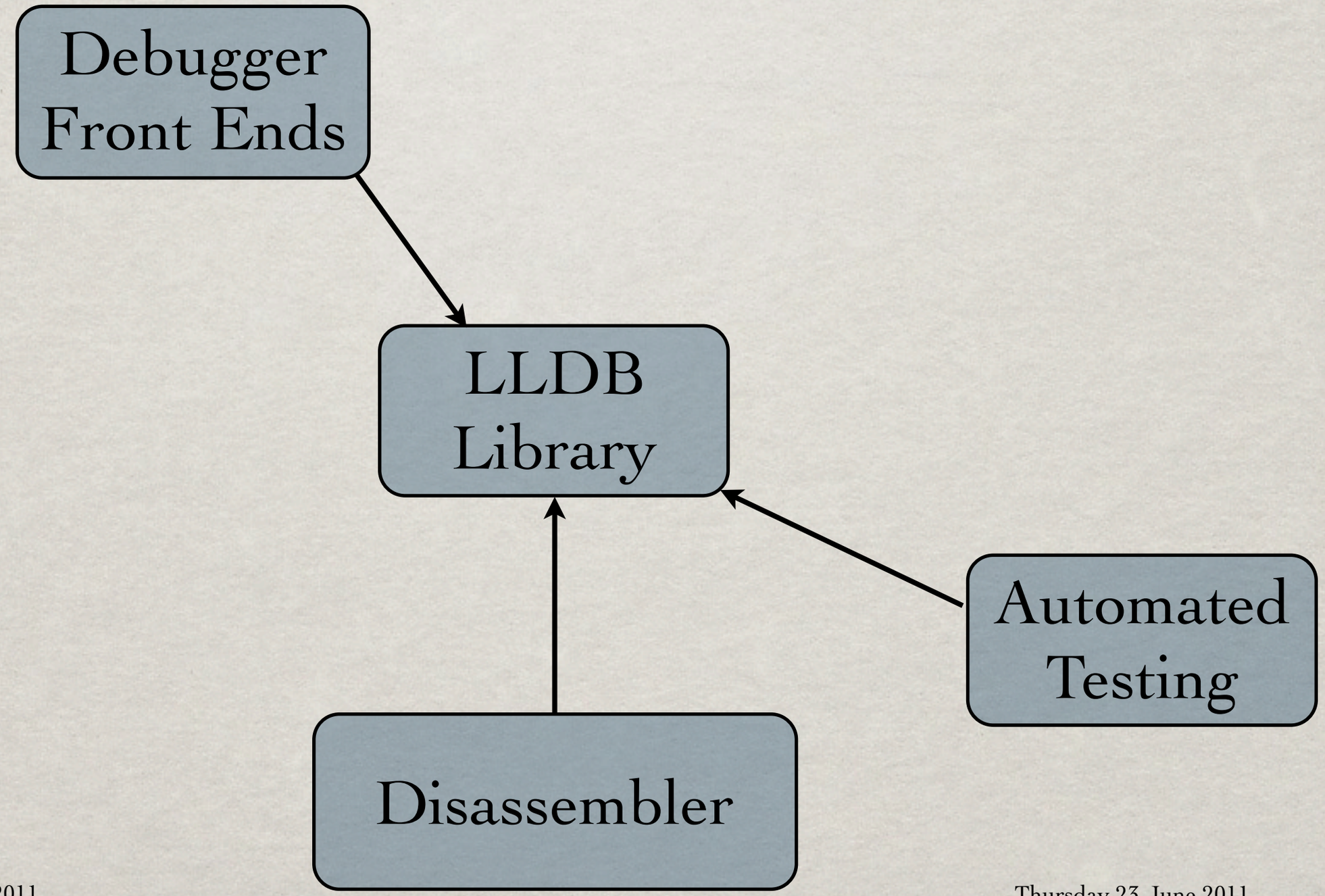
Some Uses Of the LLDB Library



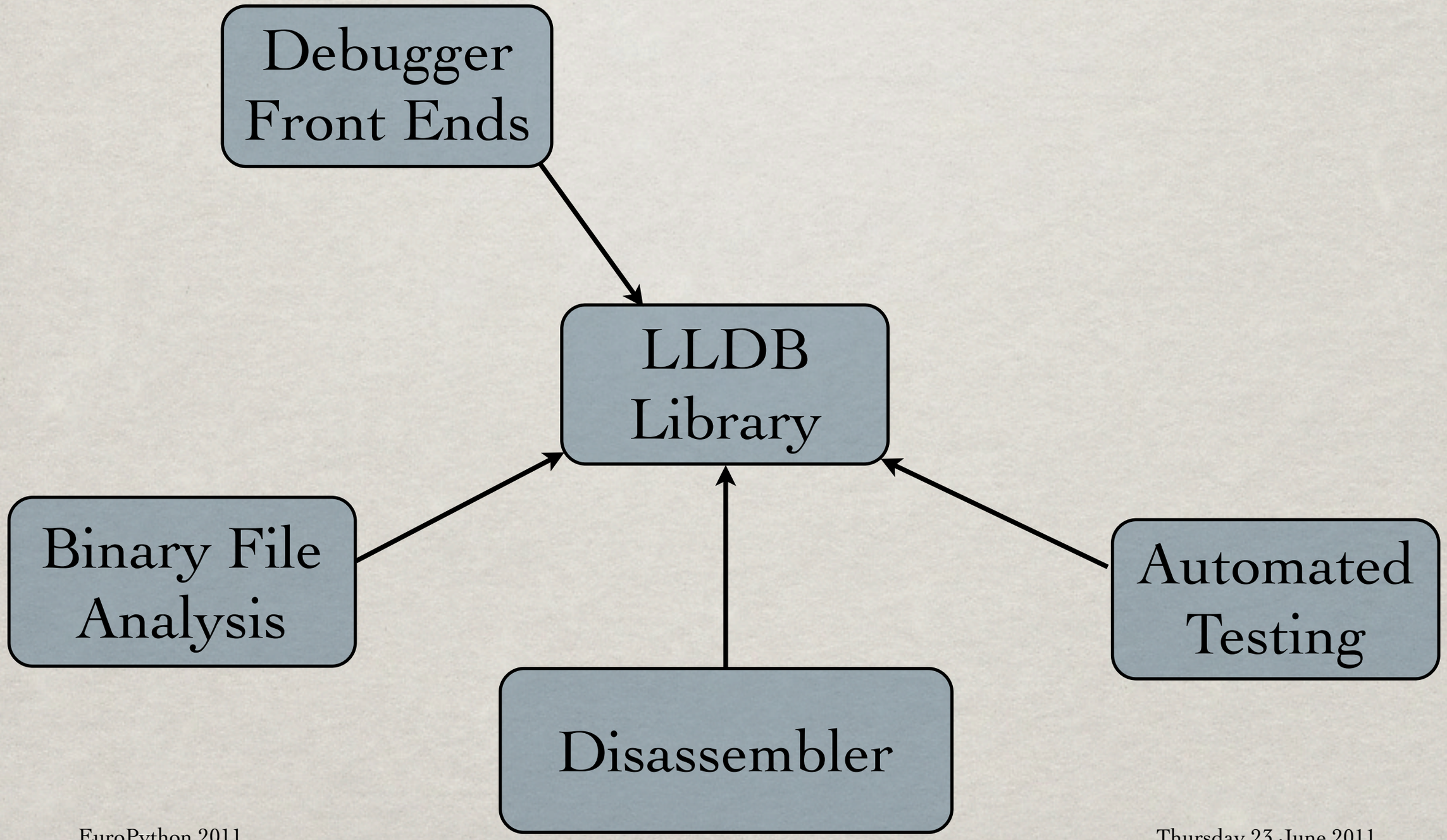
Some Uses Of the LLDB Library



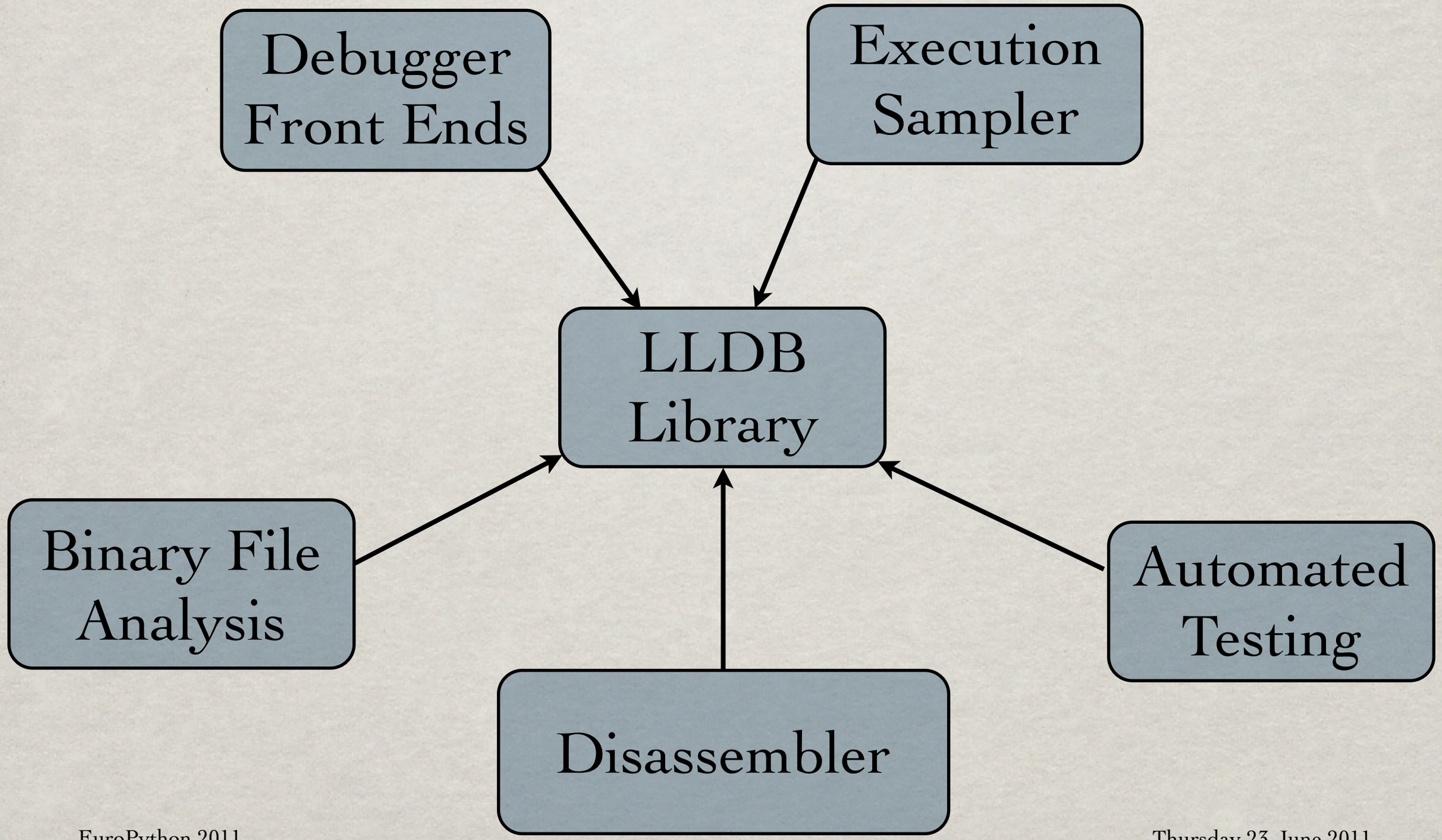
Some Uses Of the LLDB Library



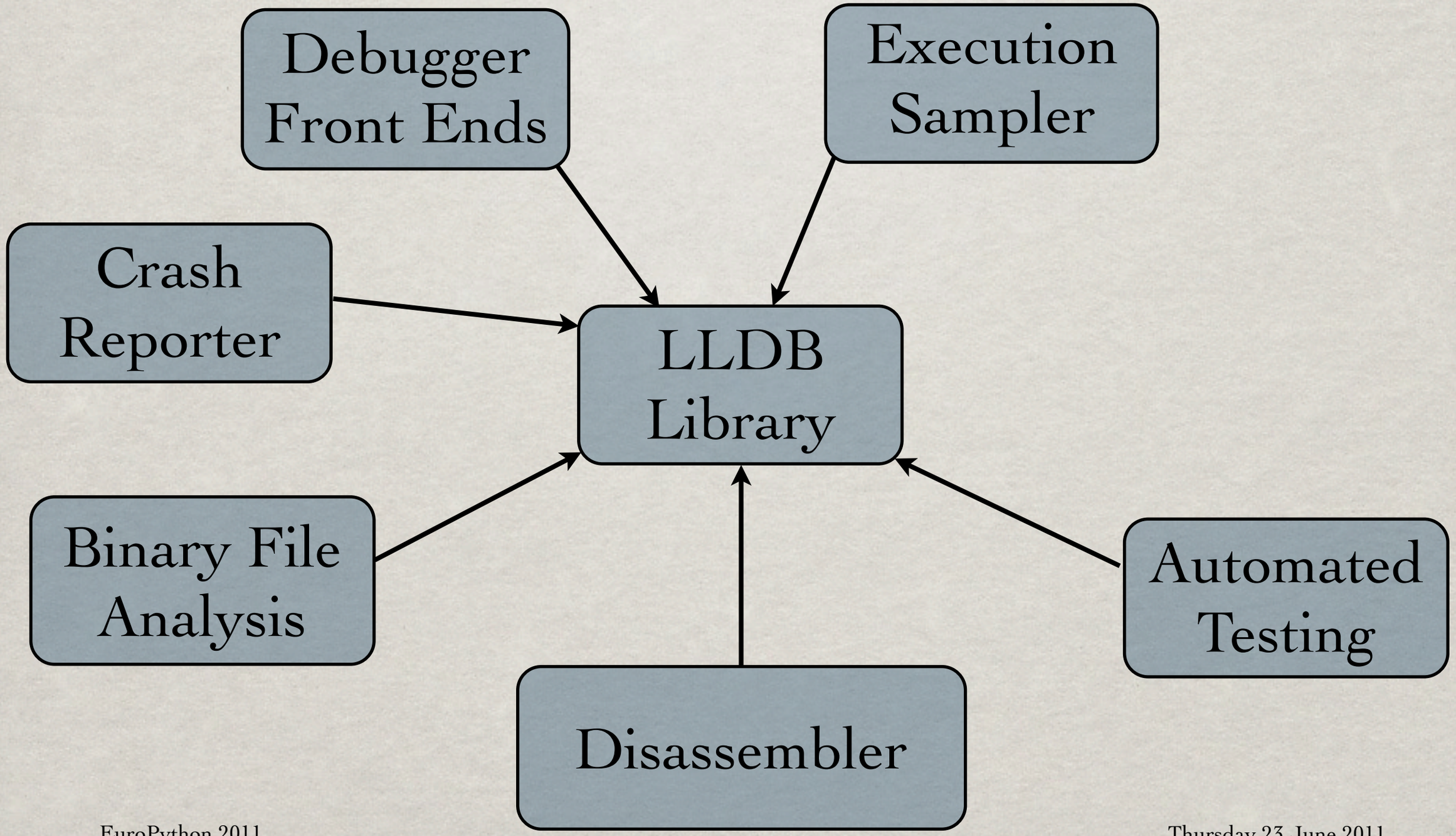
Some Uses Of the LLDB Library



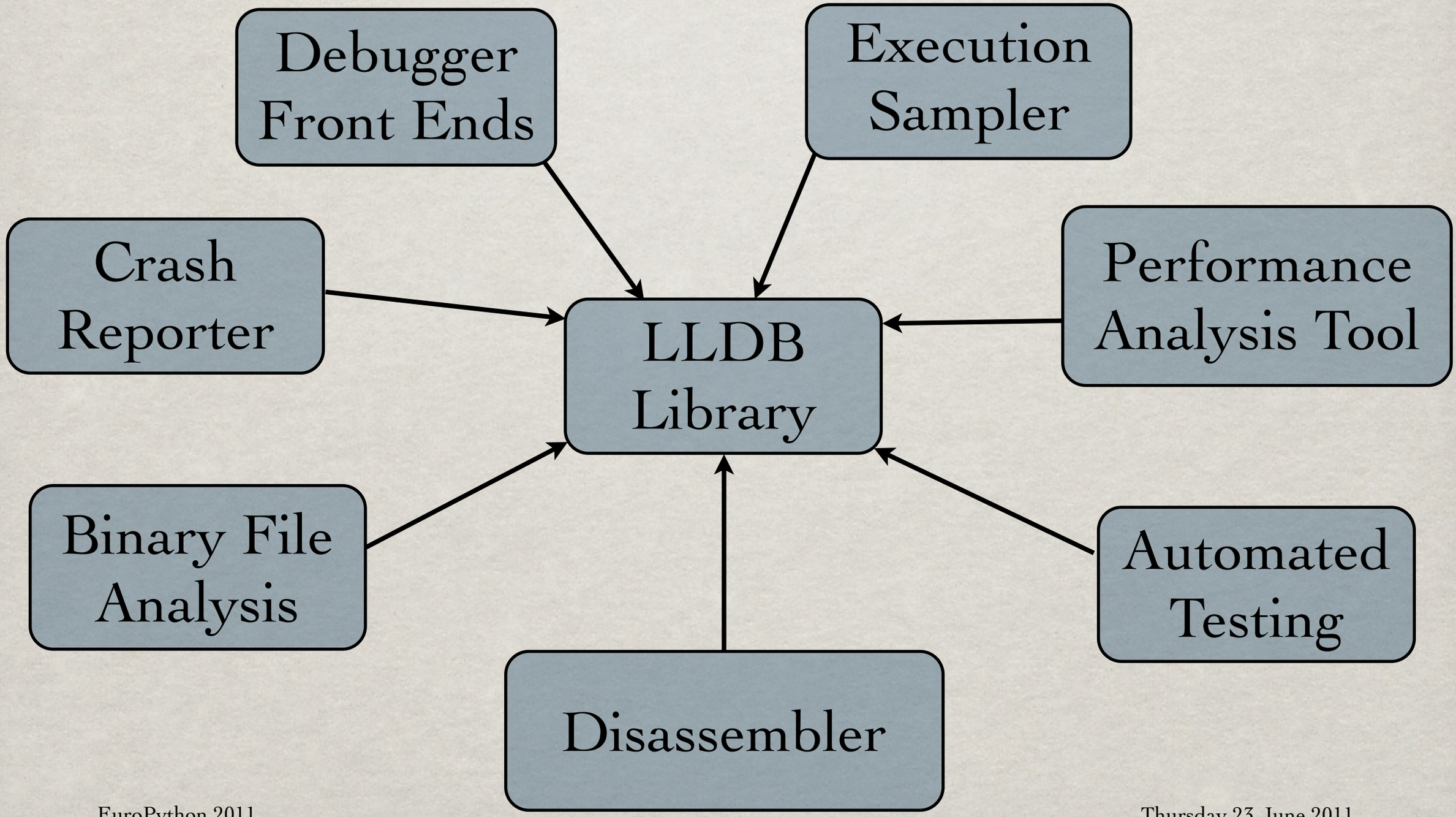
Some Uses Of the LLDB Library



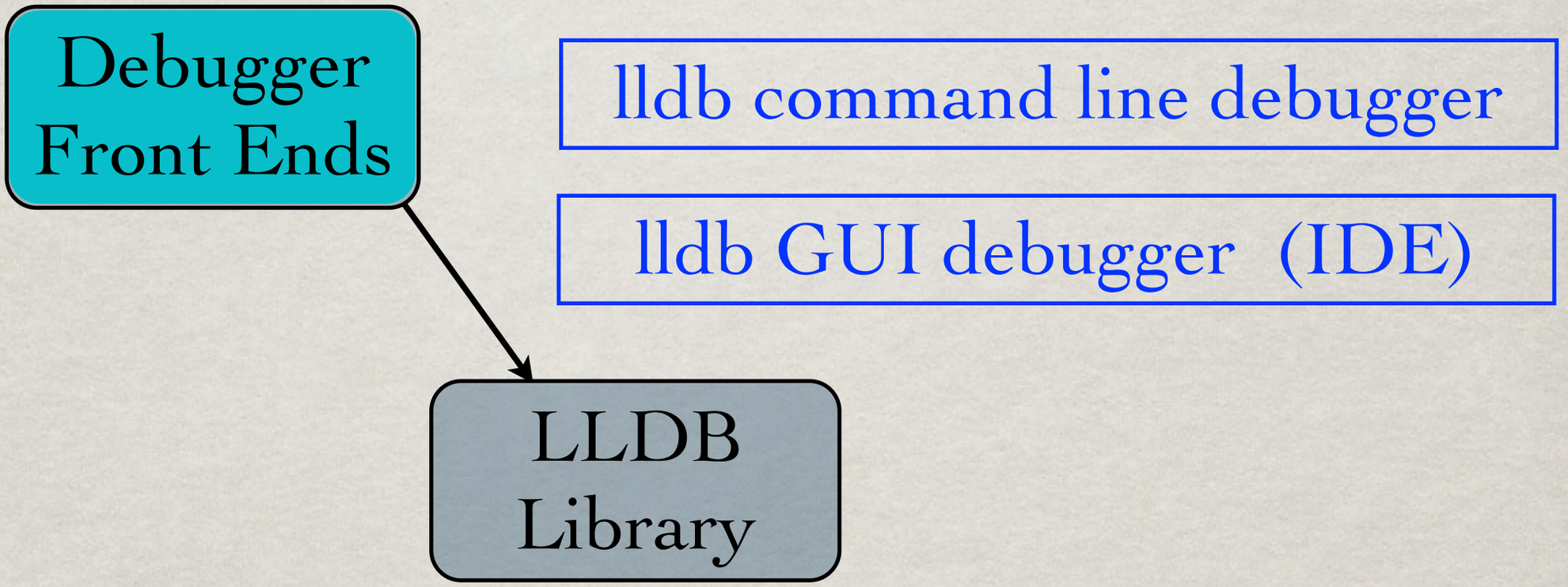
Some Uses Of the LLDB Library



Some Uses Of the LLDB Library

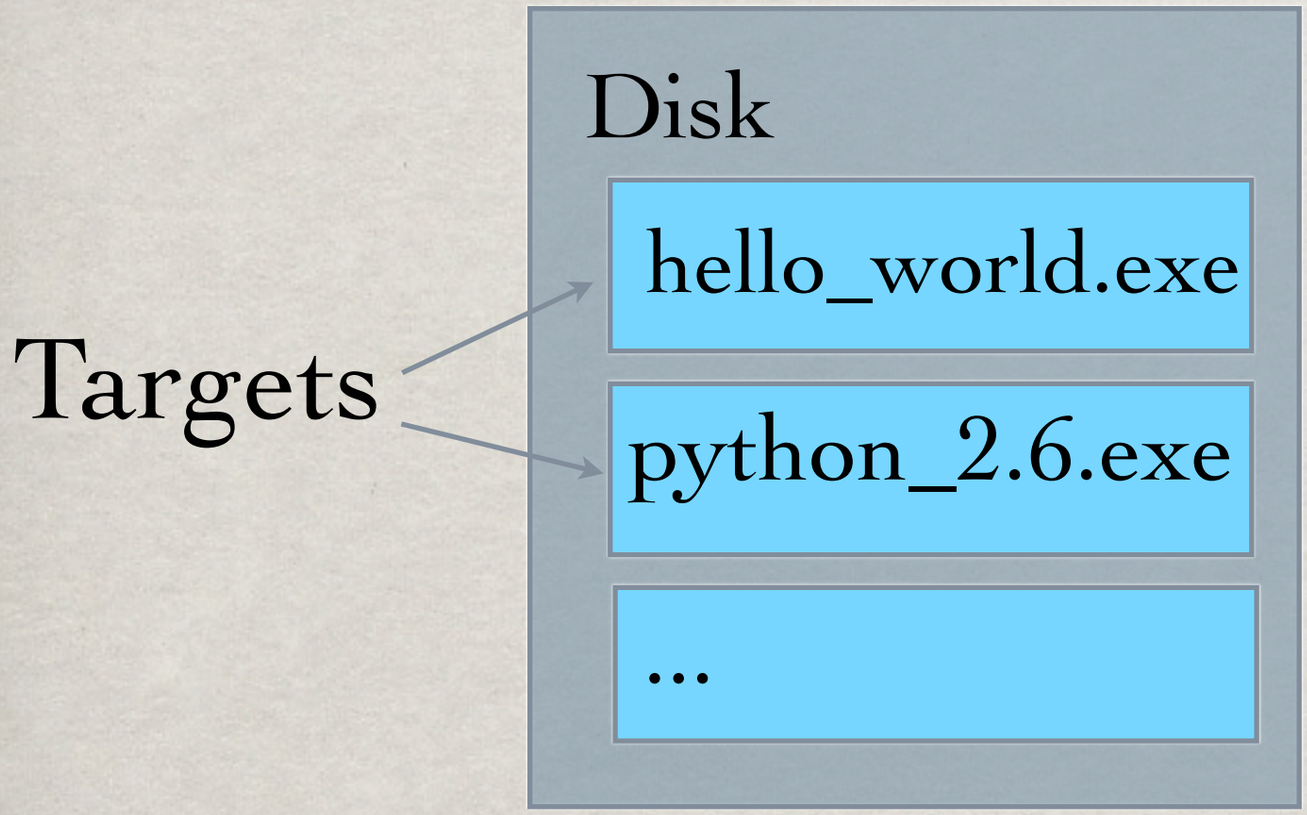


Some Uses Of the LLDB Library

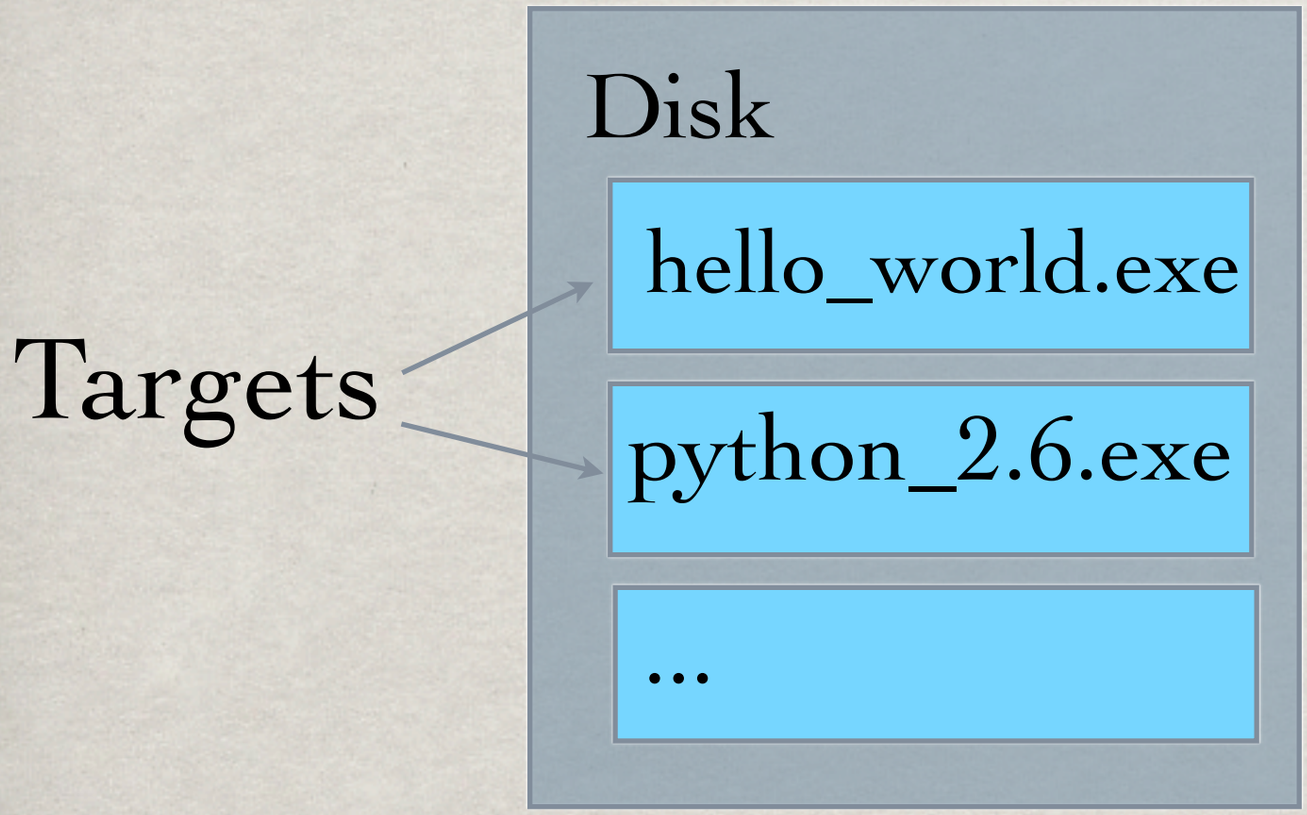


Targets & Processes

Targets & Processes

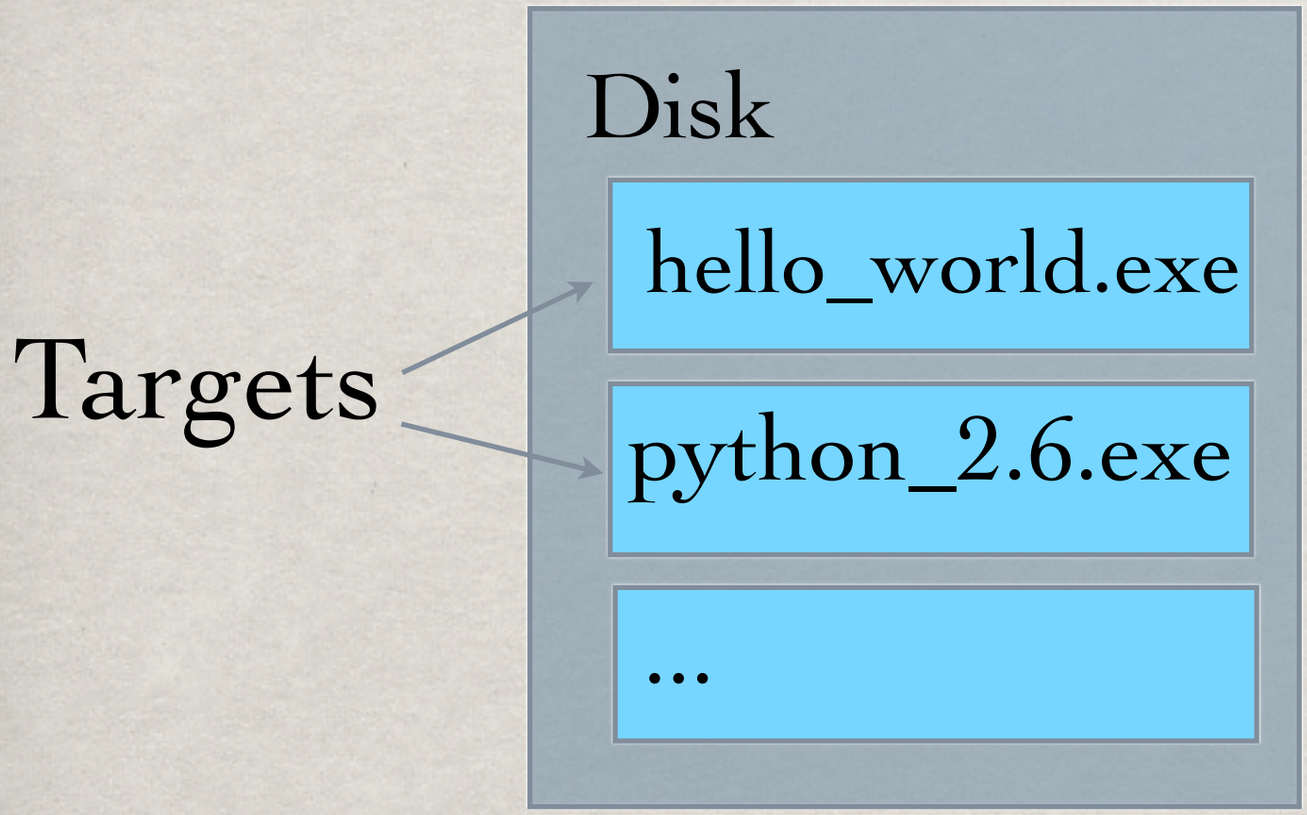


Targets & Processes



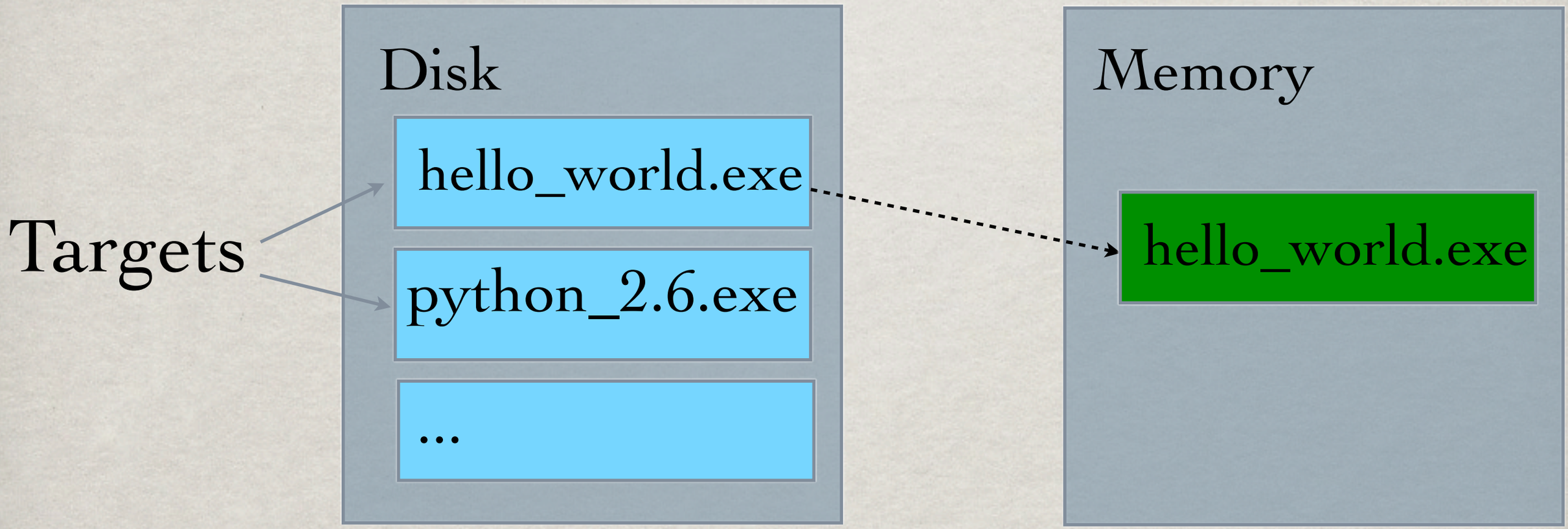
%

Targets & Processes



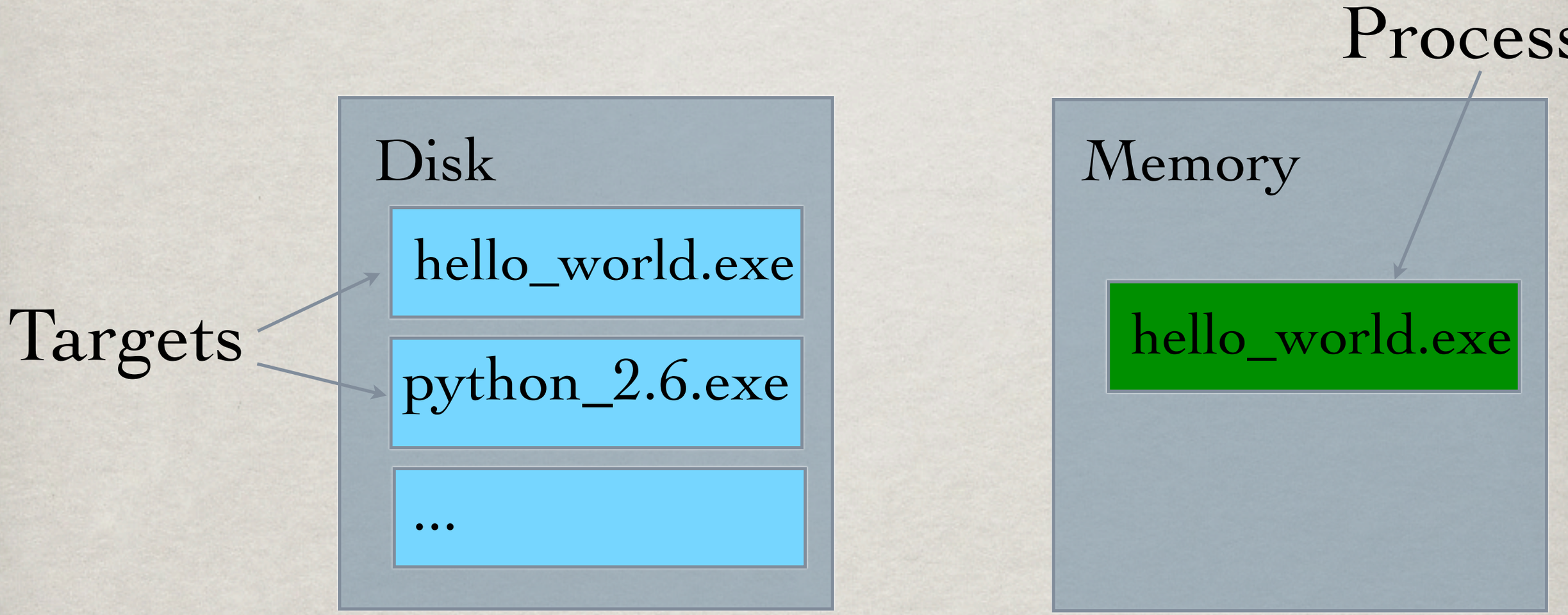
% ./hello_world.exe

Targets & Processes



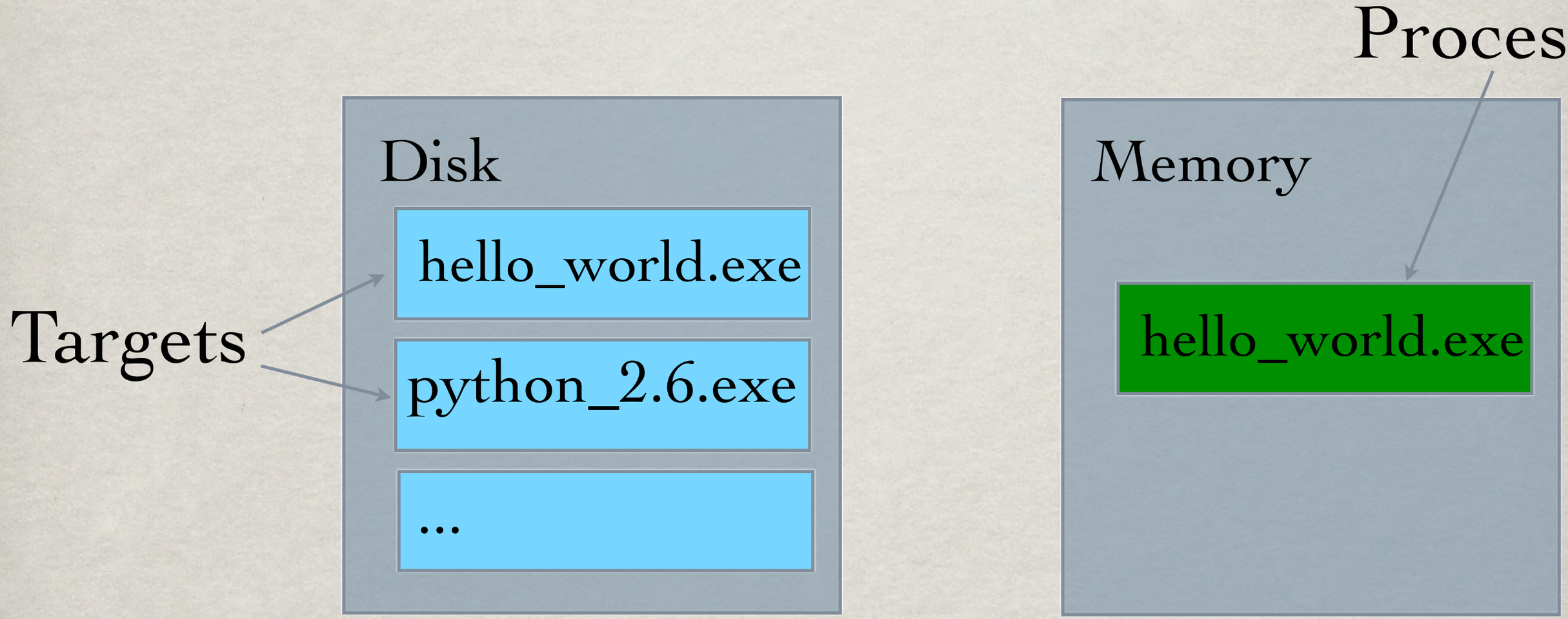
% ./hello_world.exe

Targets & Processes



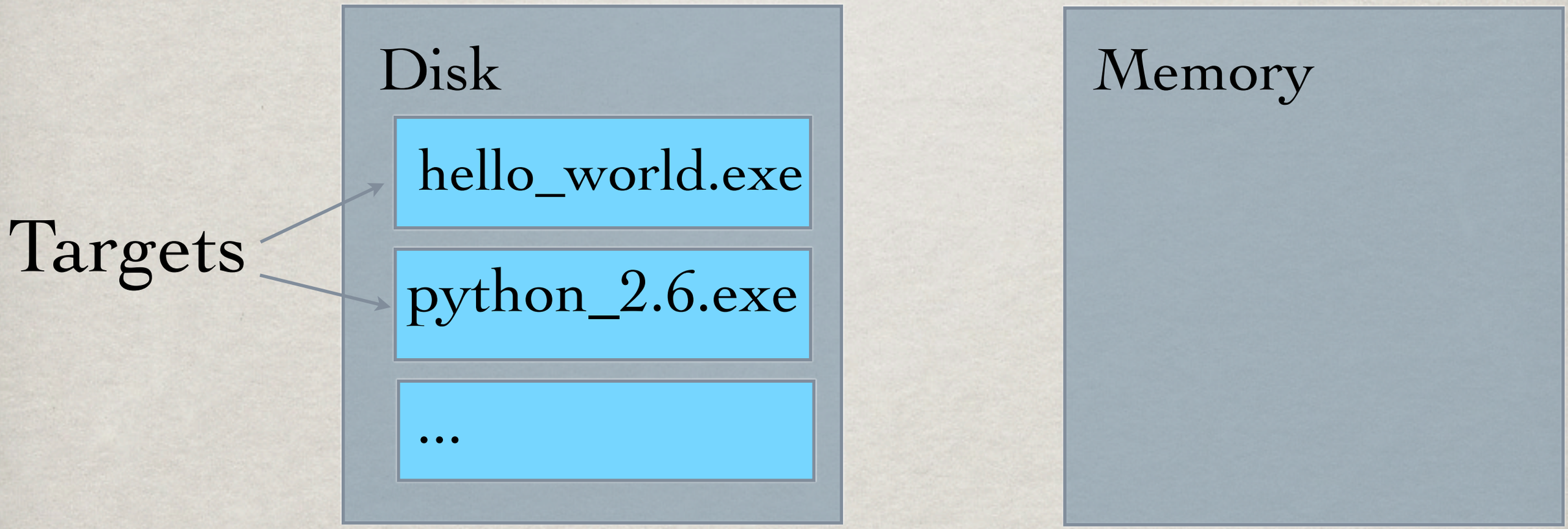
% ./hello_world.exe

Targets & Processes




```
% ./hello_world.exe
Hello!
```

Targets & Processes




```
% ./hello_world.exe
Hello!
%
```

Breakpoints

Breakpoints

- Places to pause execution

Breakpoints

- Places to pause execution
- Can be conditional

Breakpoints

- Places to pause execution
- Can be conditional
  - only stop if particular condition is met

Breakpoints

- Places to pause execution
- Can be conditional
  - only stop if particular condition is met
- Can have code attached to it

Breakpoints

- Places to pause execution
- Can be conditional
  - only stop if particular condition is met
- Can have code attached to it
  - execute attached code when breakpoint is hit

Frames & Stacks

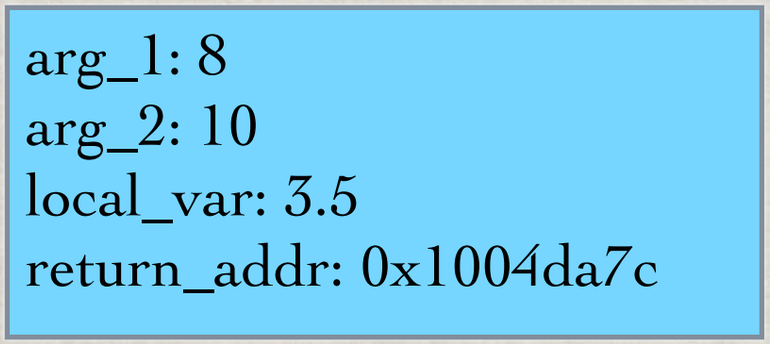
Frames & Stacks

- Every function call has a 'frame'

Frames & Stacks

- Every function call has a ‘frame’

“Foo”:

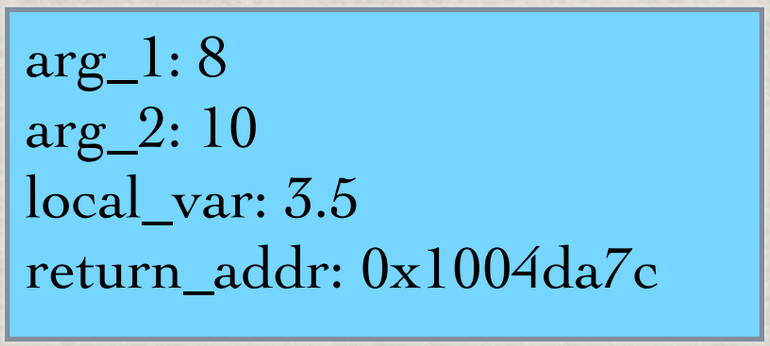


Frame

Frames & Stacks

- Every function call has a ‘frame’
- Process/Thread maintains stack of currently called functions

“Foo”:



Frame

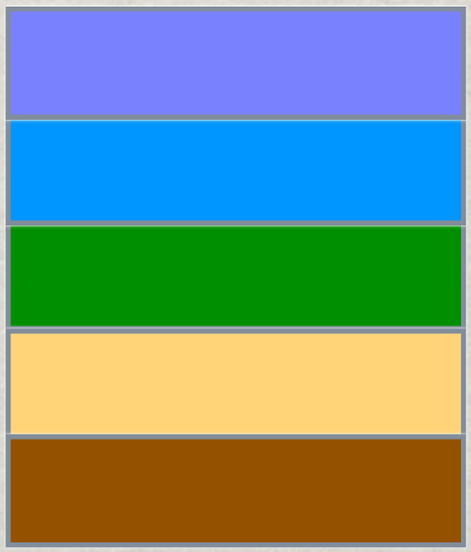
Frames & Stacks

- Every function call has a ‘frame’
- Process/Thread maintains stack of currently called functions

“Foo”:

arg_1: 8
arg_2: 10
local_var: 3.5
return_addr: 0x1004da7c

Frame



Call Stack

Frames & Stacks

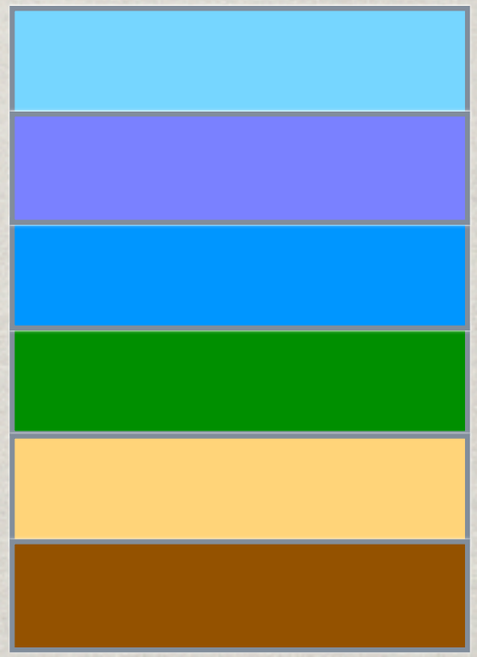
- Every function call has a ‘frame’
- Process/Thread maintains stack of currently called functions

“Foo”:

arg_1: 8
arg_2: 10
local_var: 3.5
return_addr: 0x1004da7c

Frame

“Foo”:



Call Stack

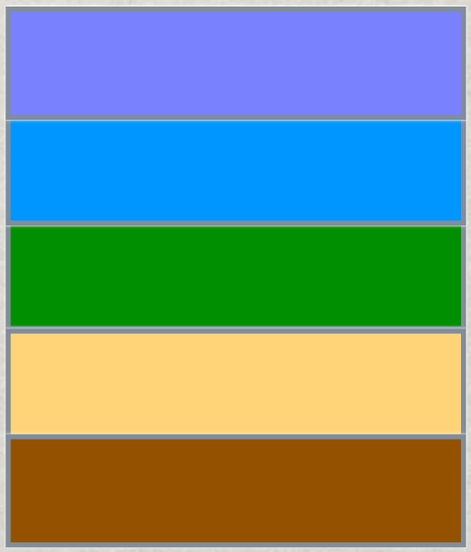
Frames & Stacks

- Every function call has a ‘frame’
- Process/Thread maintains stack of currently called functions

“Foo”:

arg_1: 8
arg_2: 10
local_var: 3.5
return_addr: 0x1004da7c

Frame



Call Stack

More About LLDB...

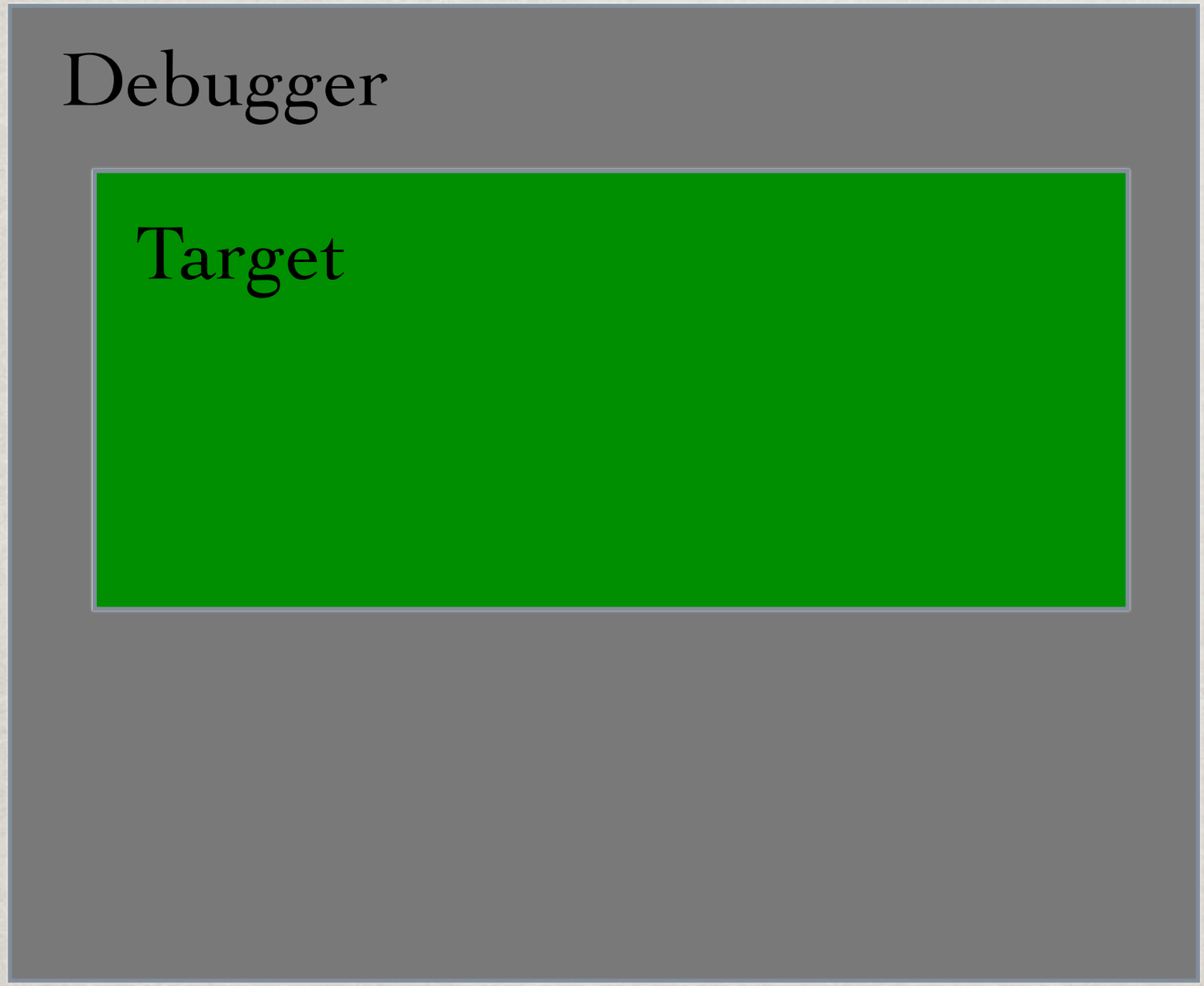
- Written in C++
- Multi-threaded
- Object-Oriented
  - Main pieces are objects
  - Objects are nested
- Currently in Beta release - still a few bumps!

Objects in LLDB

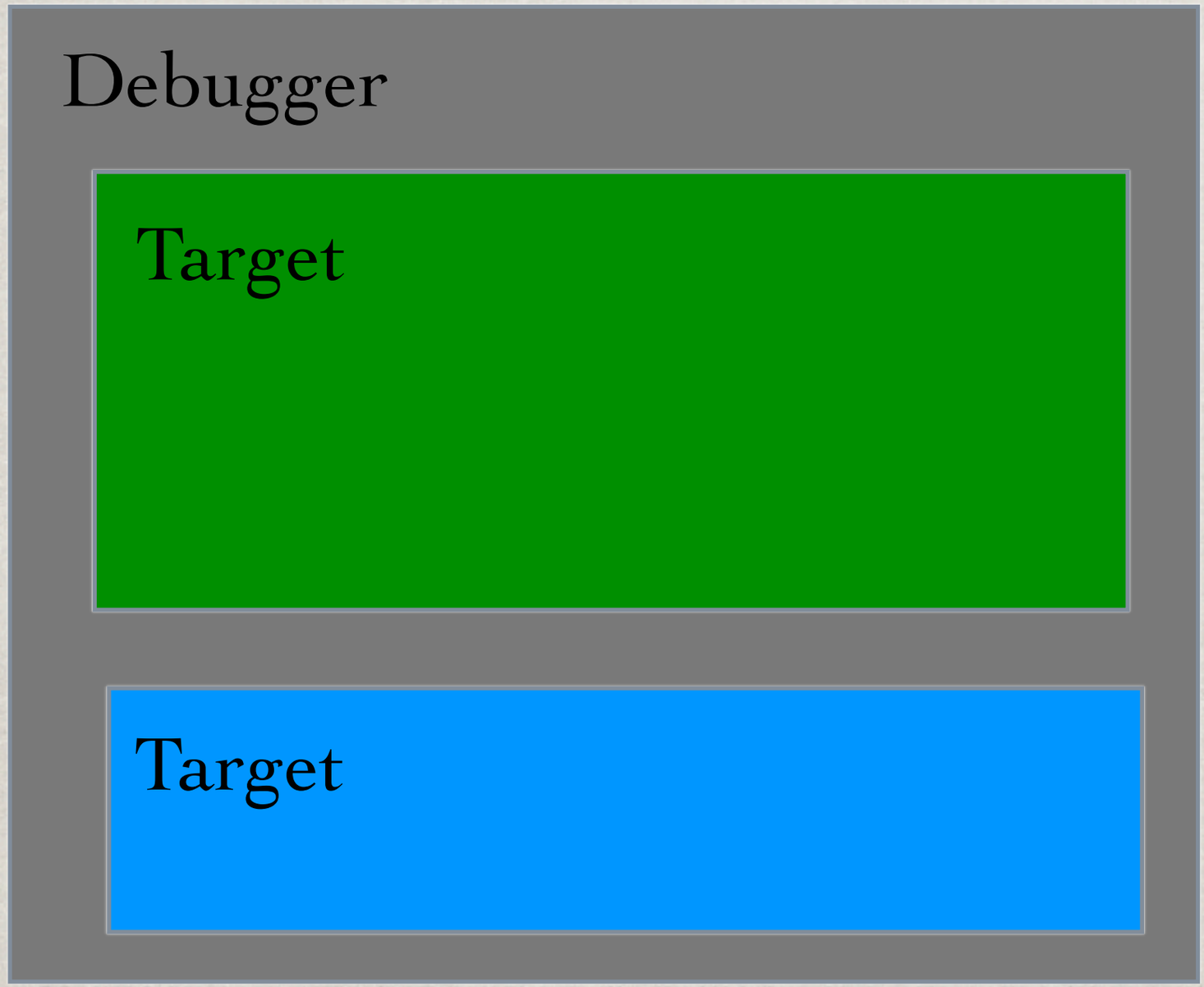
Objects in LLDB

Debugger

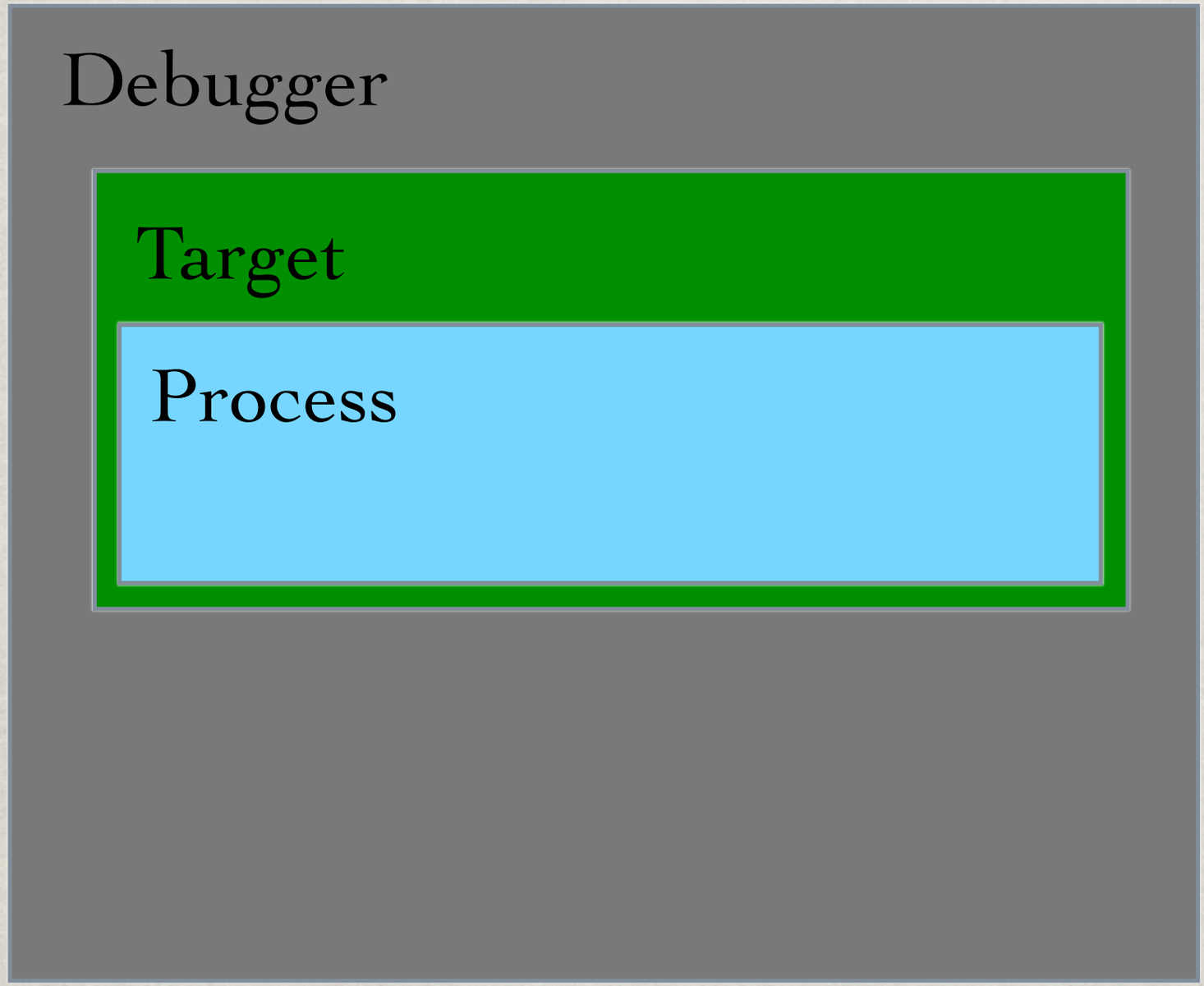
Objects in LLDB



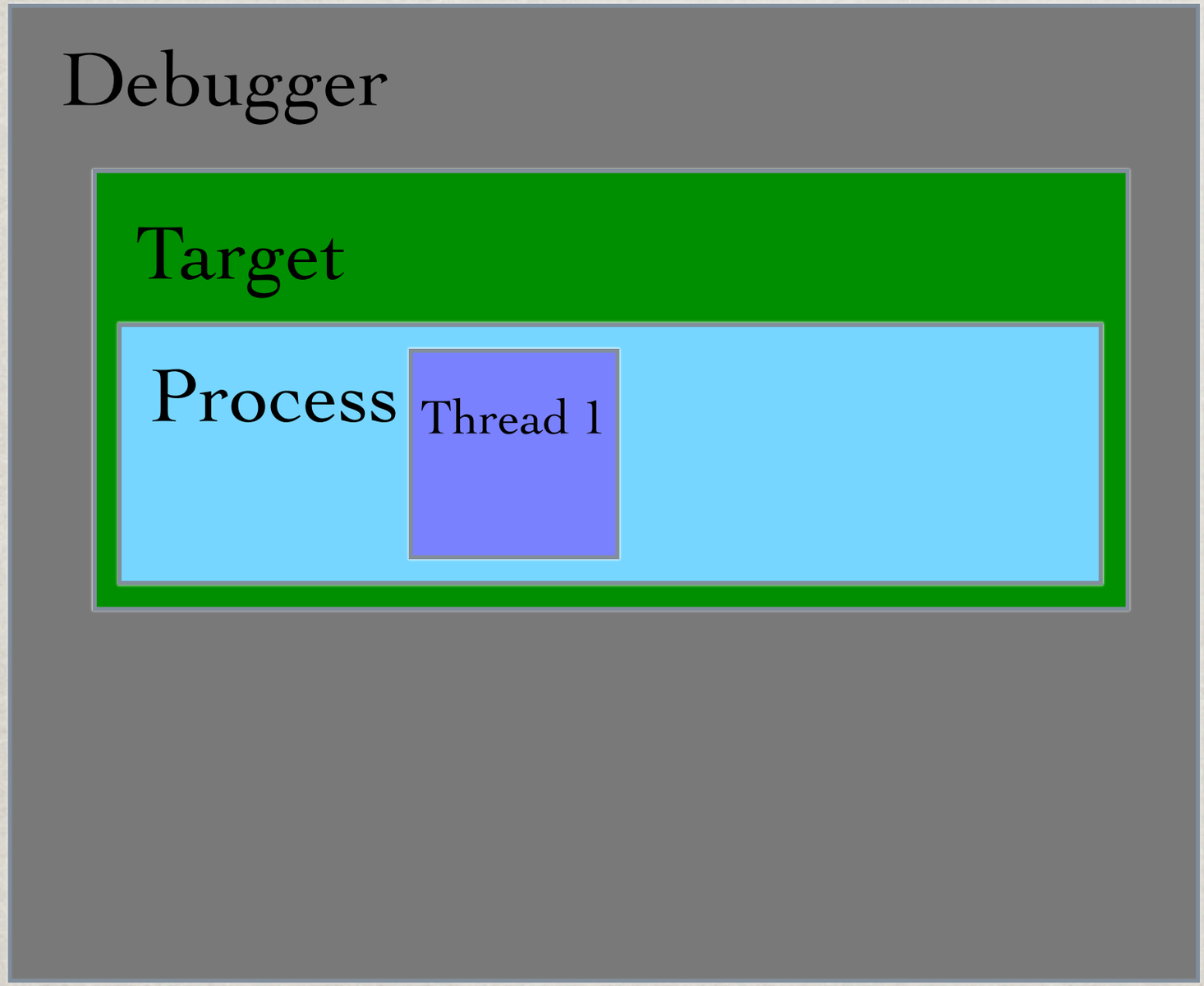
Objects in LLDB



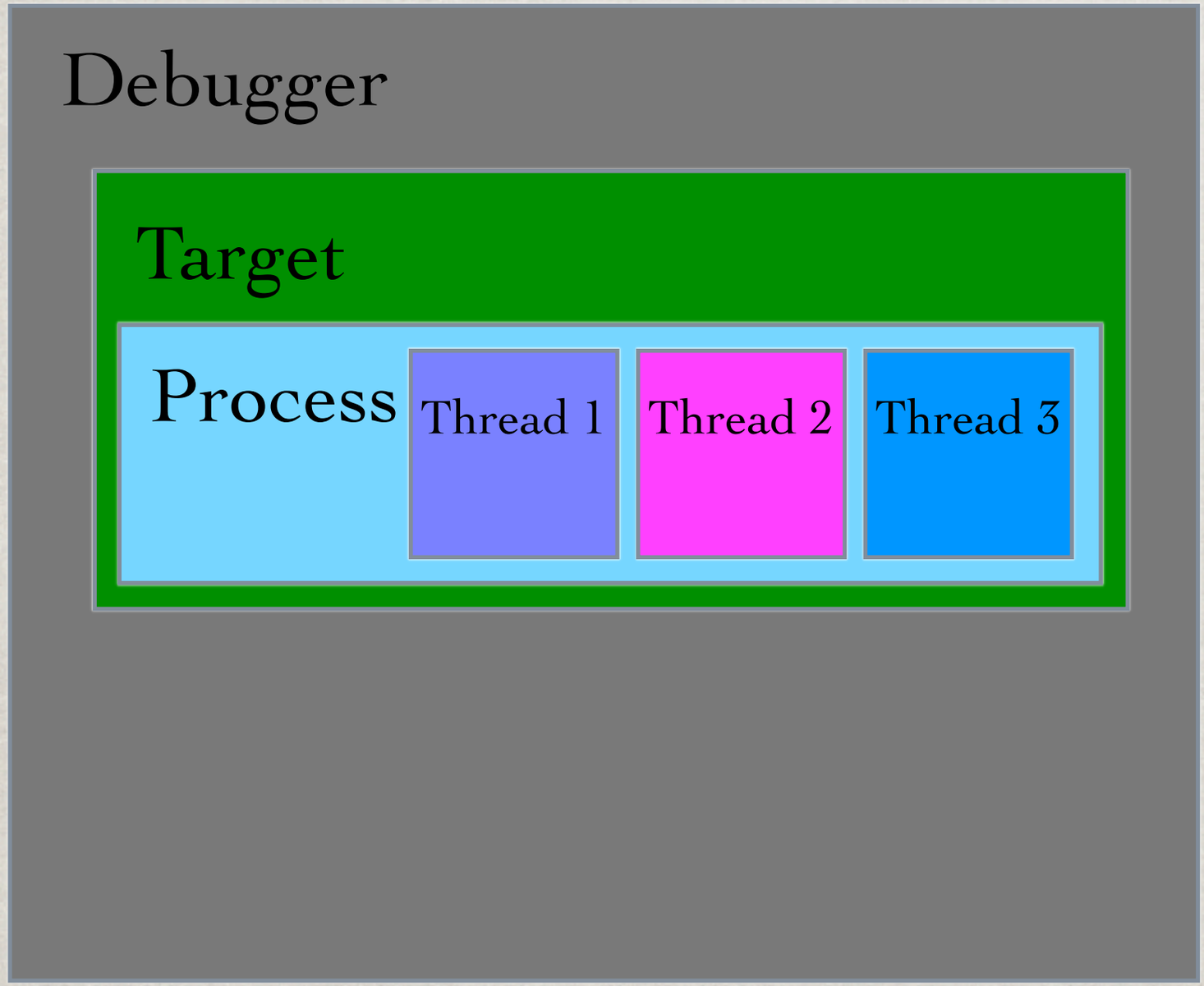
Objects in LLDB



Objects in LLDB



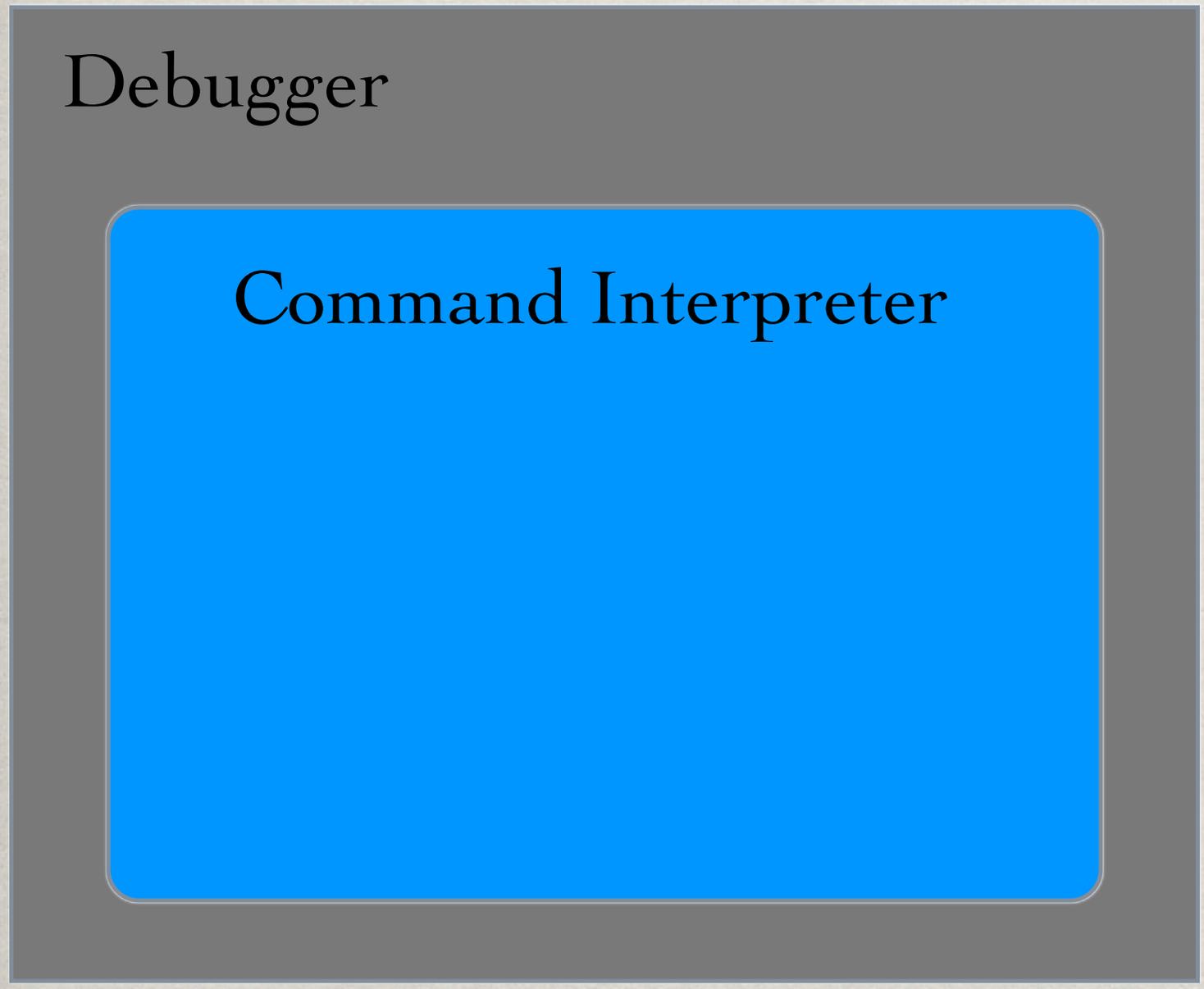
Objects in LLDB



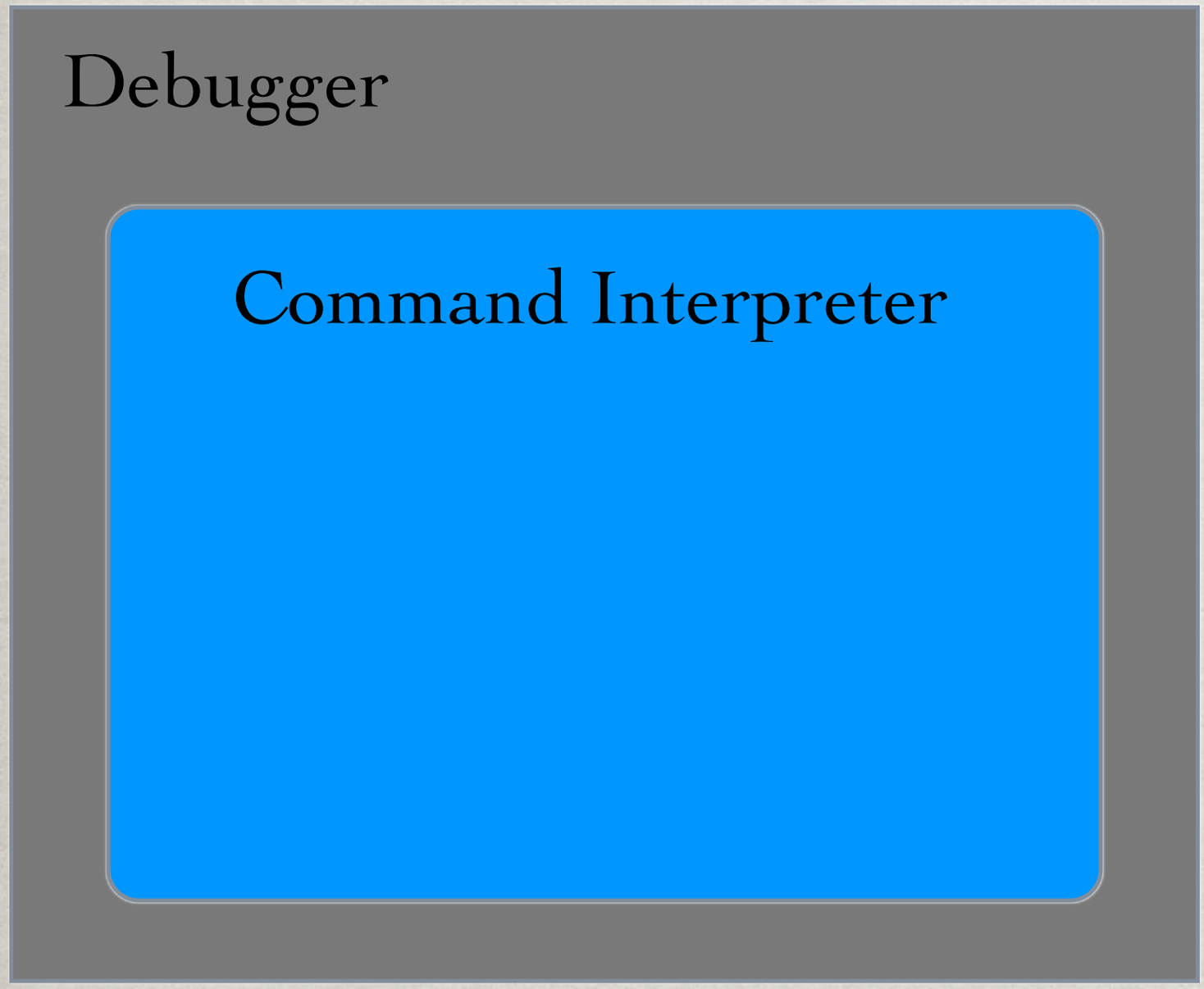
Interpreters in LLDB

Debugger

Interpreters in LLDB

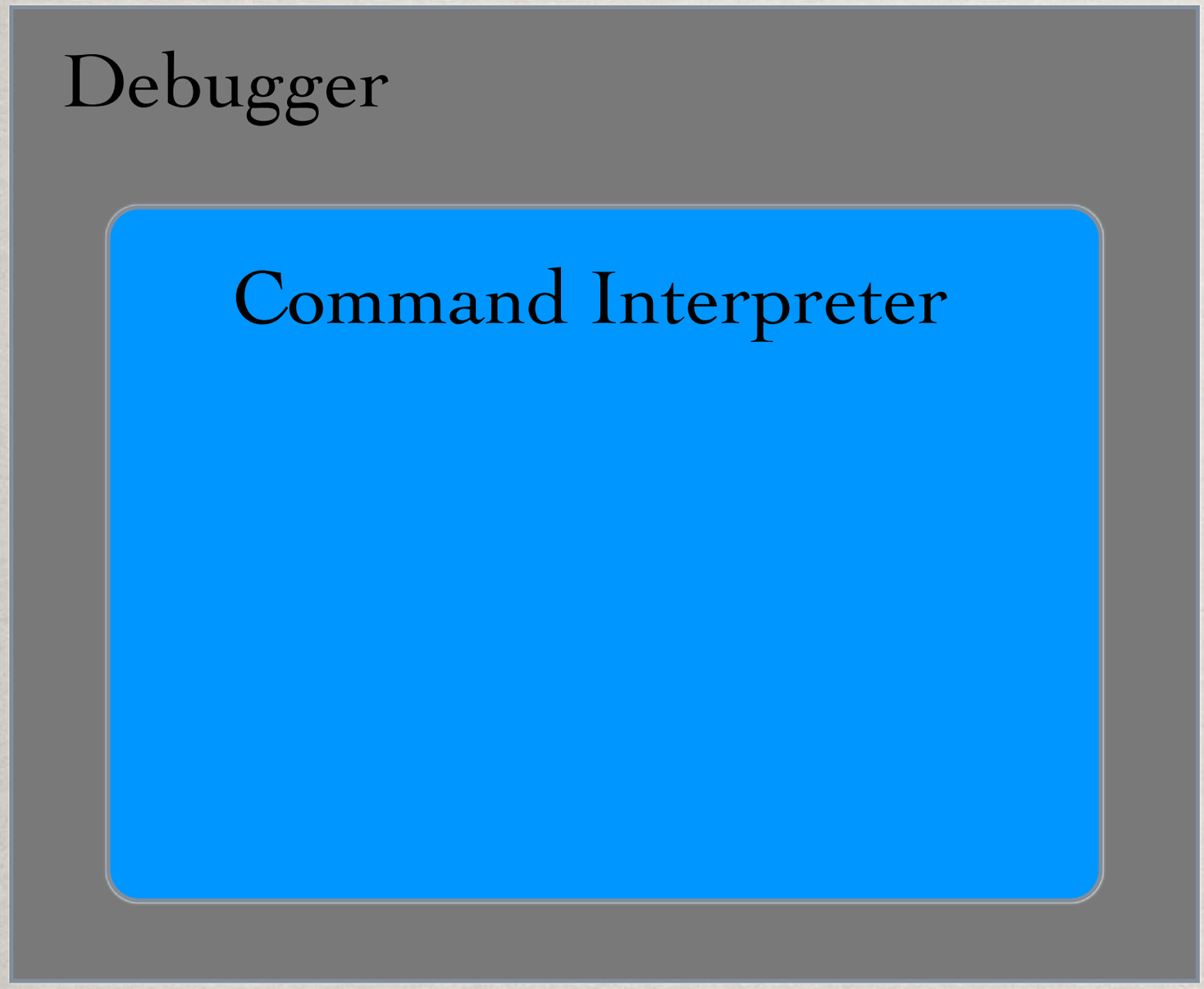


Interpreters in LLDB

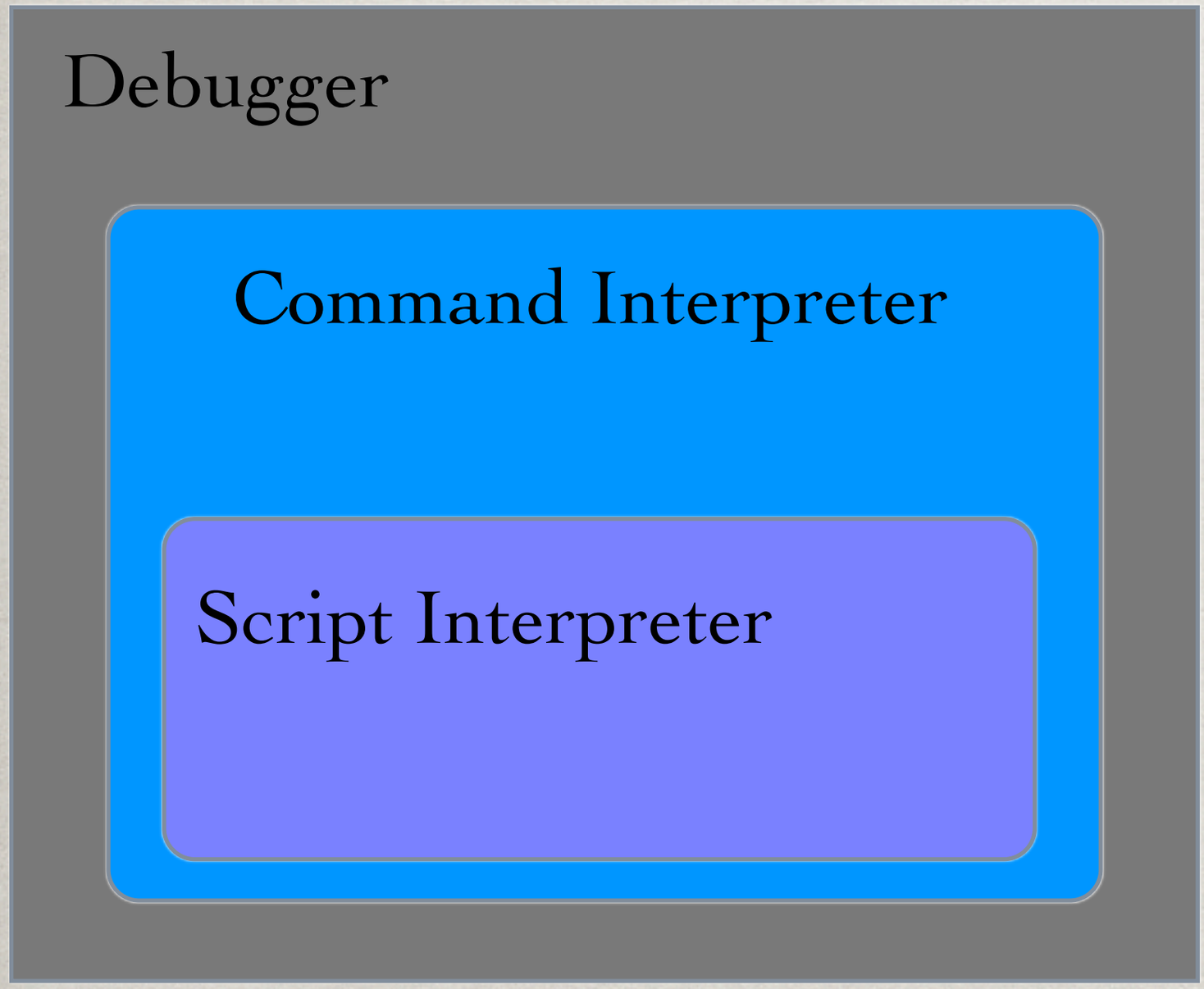



```
(lldb) file  hello-world
(lldb) break set -n main
(lldb) run
```

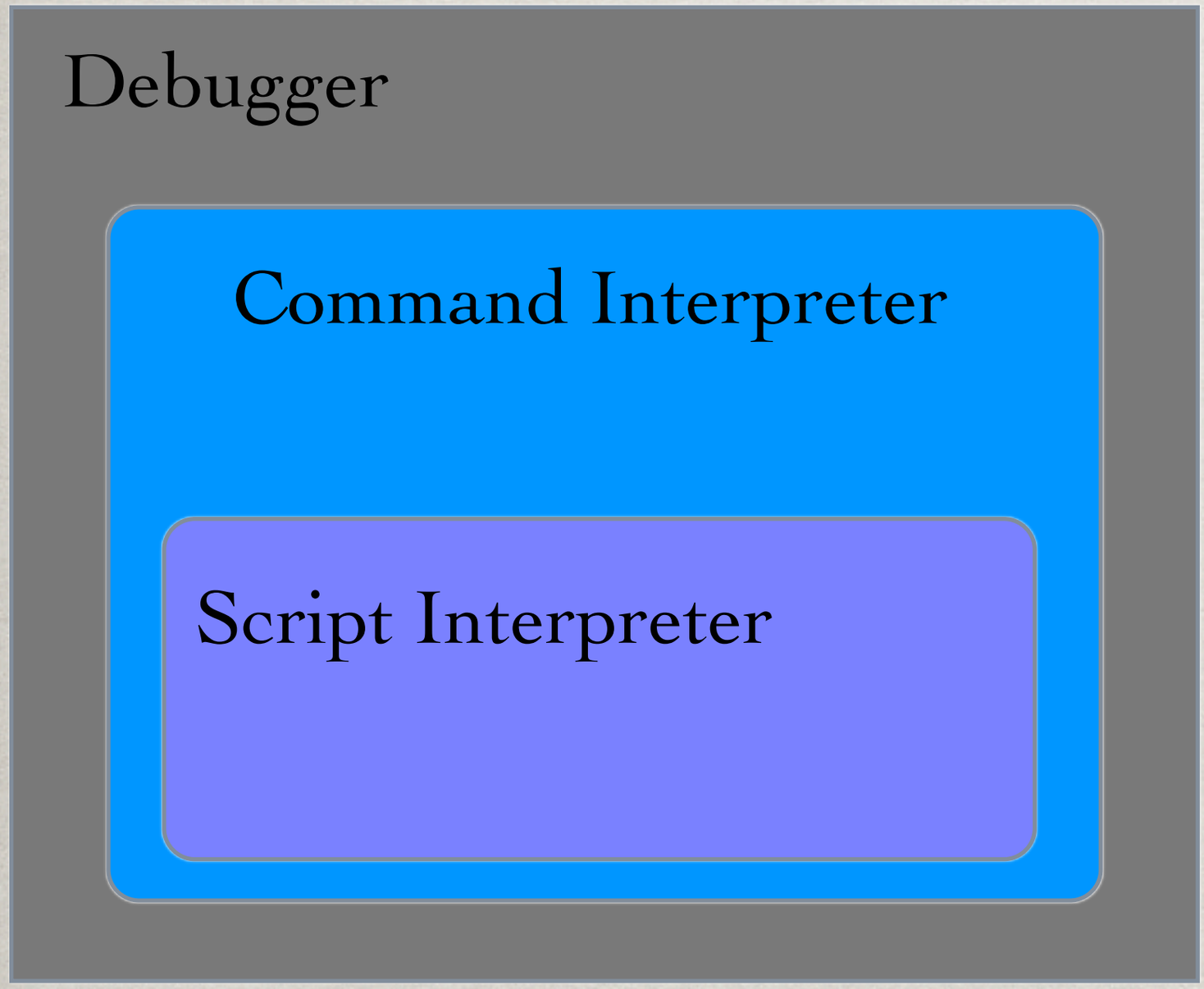
Interpreters in LLDB



Interpreters in LLDB



Interpreters in LLDB




```
>>> import sys
>>> a = 9
>>>
```

Desired GUI Debugger Behavior

Desired GUI Debugger Behavior

- From single running GUI debugger process...

Desired GUI Debugger Behavior

- From single running GUI debugger process...
- Launch multiple targets...

Desired GUI Debugger Behavior

- From single running GUI debugger process...
- Launch multiple targets...
- ...each in a separate window

Desired GUI Debugger Behavior

- From single running GUI debugger process...
- Launch multiple targets...
  - ...each in a separate window
  - ...each with a separate debugger session

GUI Debugger Python Requirements

- Multiple simultaneous debuggers
- Multiple Script Interpreters
- Ability to switch smoothly between them
- Complete isolation between sessions
- Thread safety
- No deadlocking

Debuggers in LLDB

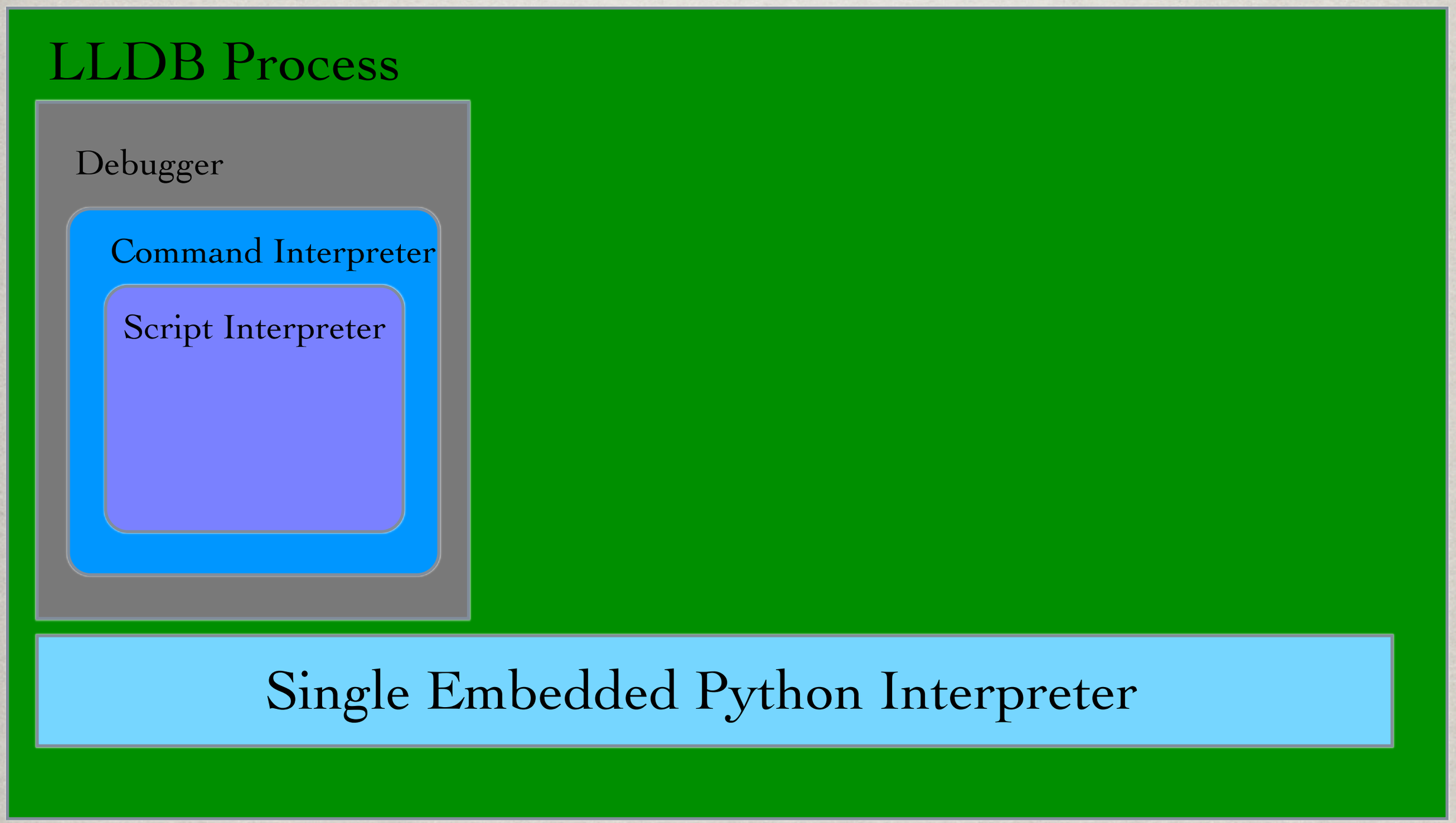
LLDB Process

Debuggers in LLDB

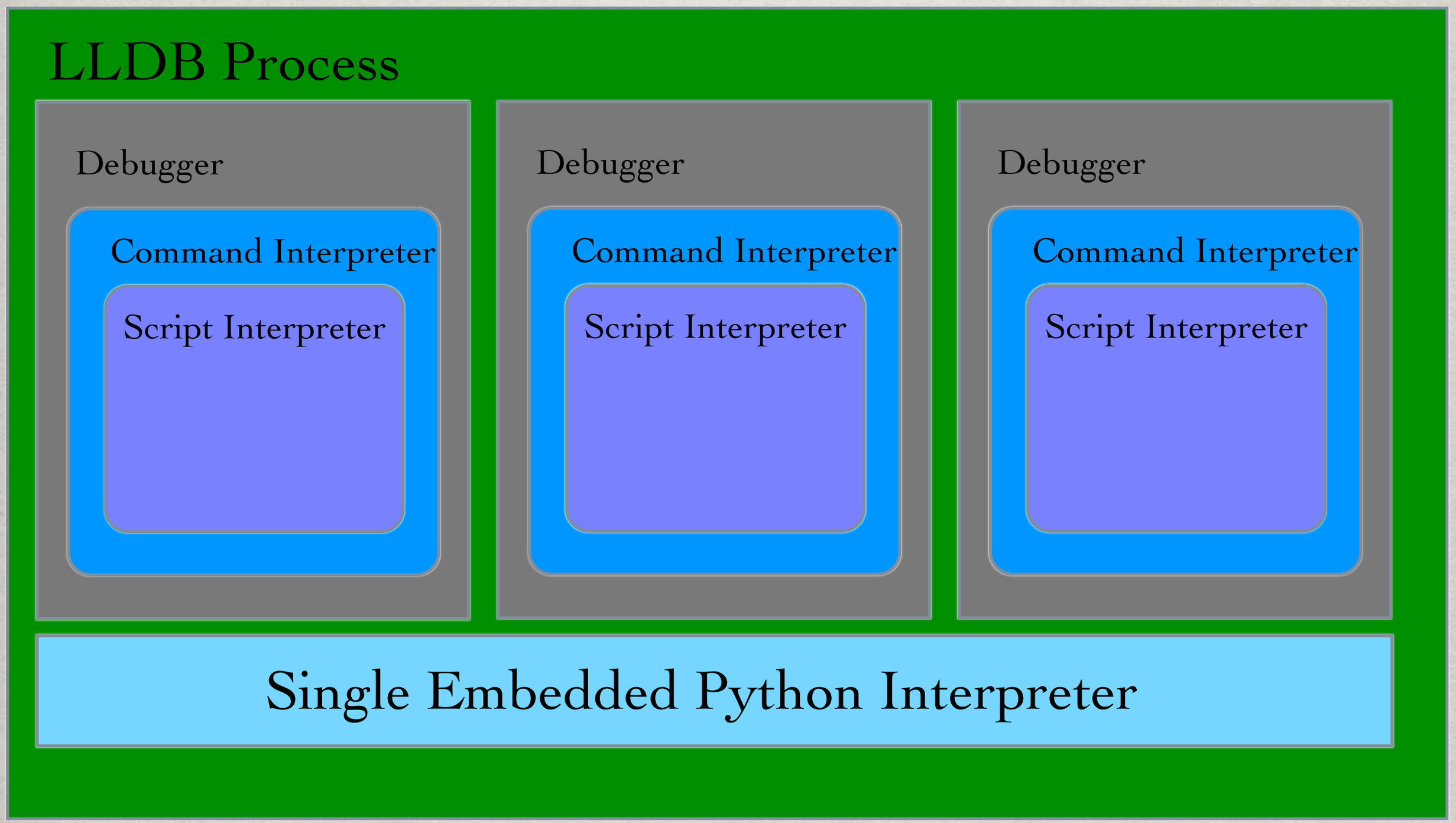
LLDB Process

Single Embedded Python Interpreter

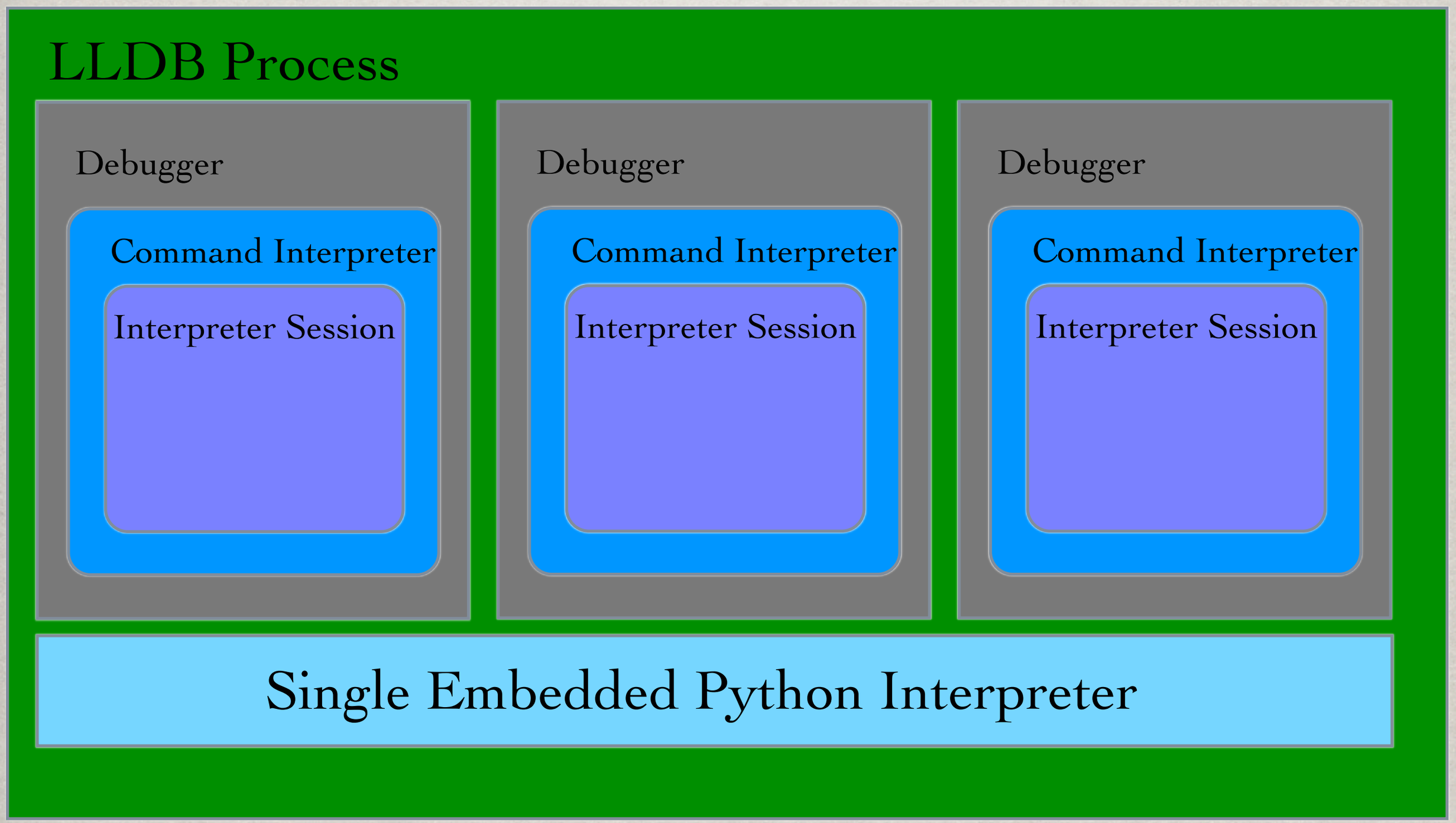
Debuggers in LLDB



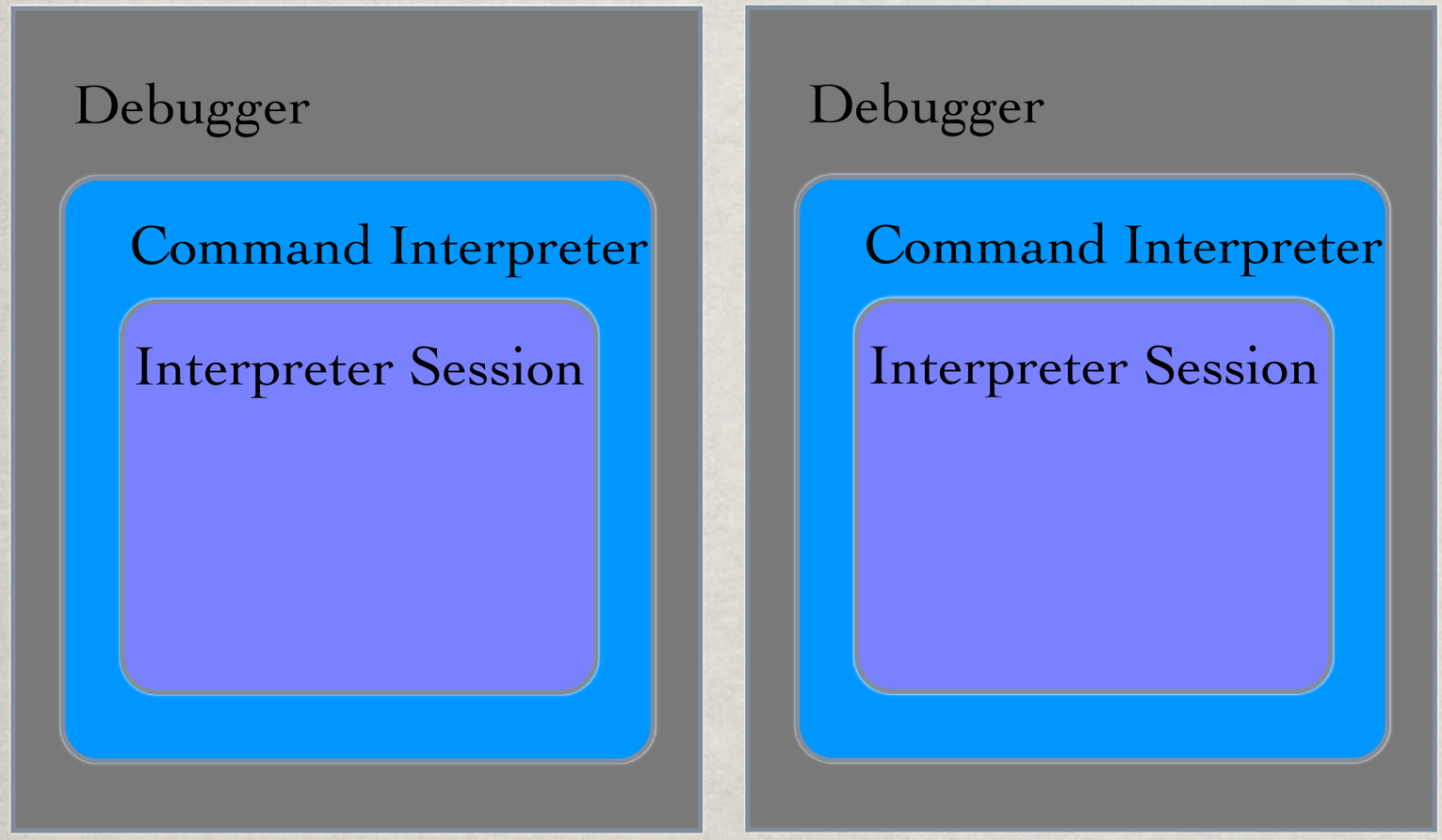
Debuggers in LLDB



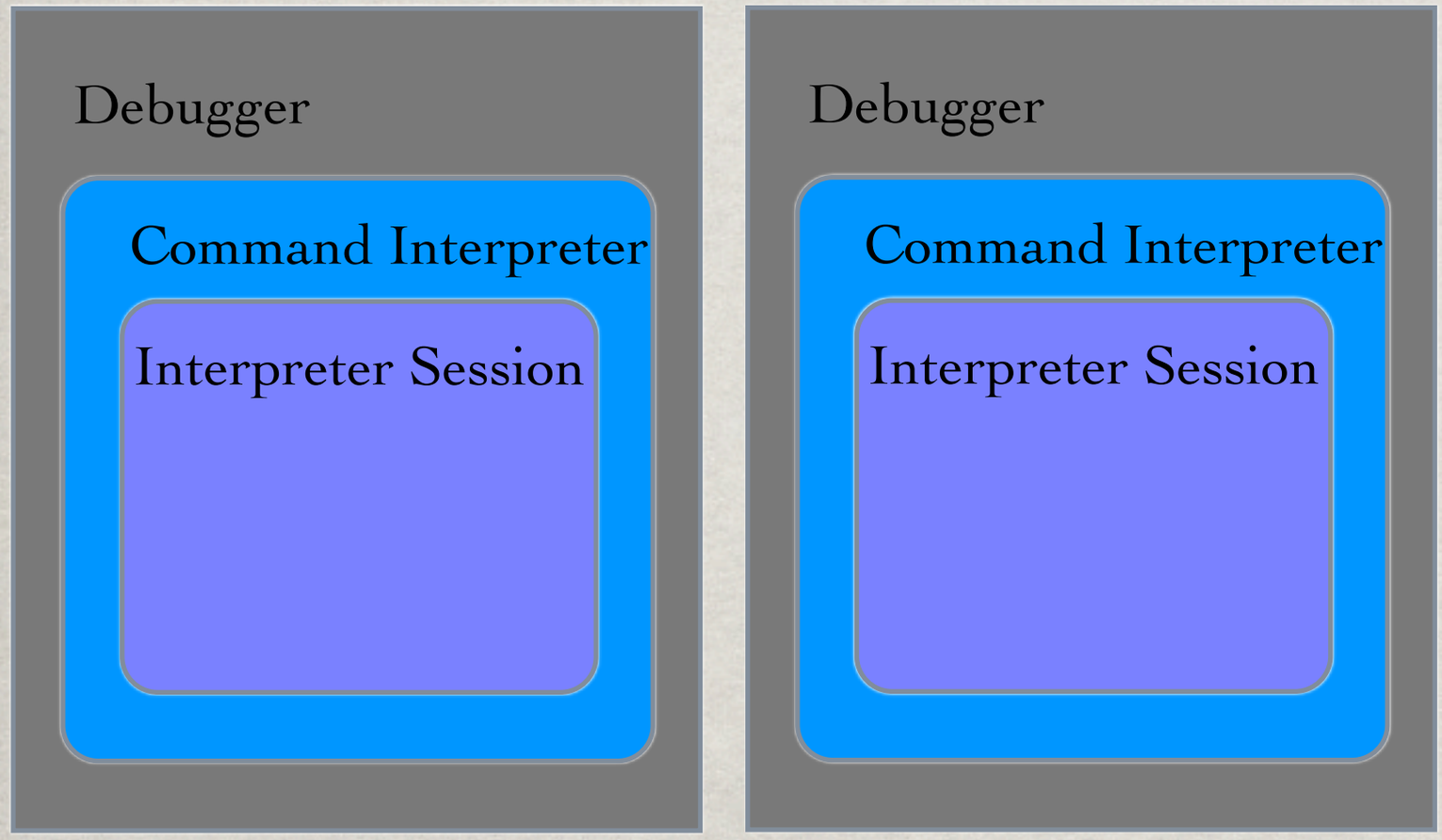
Debuggers in LLDB



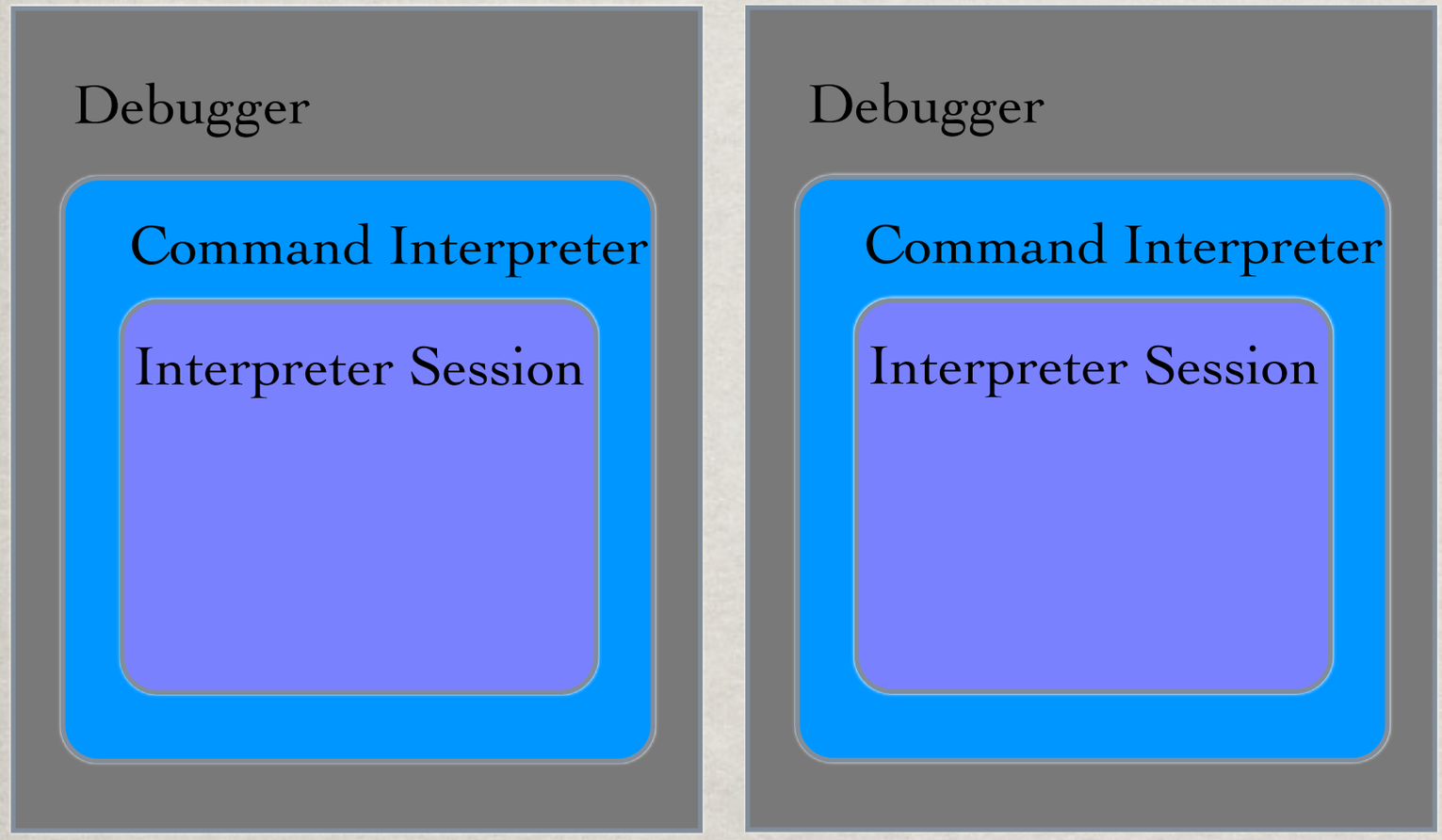
Debuggers in LLDB



Debuggers in LLDB

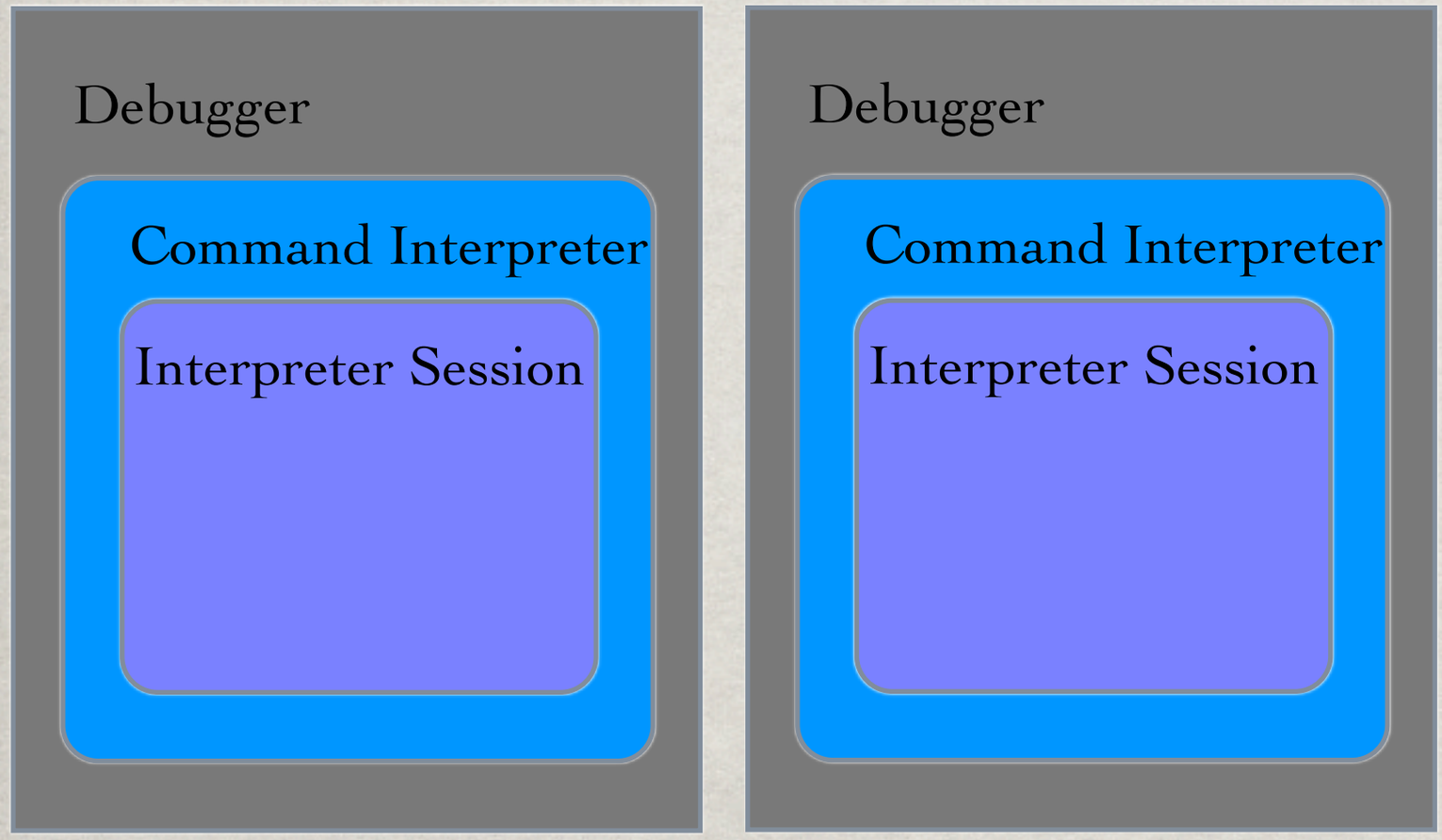


Debuggers in LLDB




```
>>> count = 24
>>> print count
```

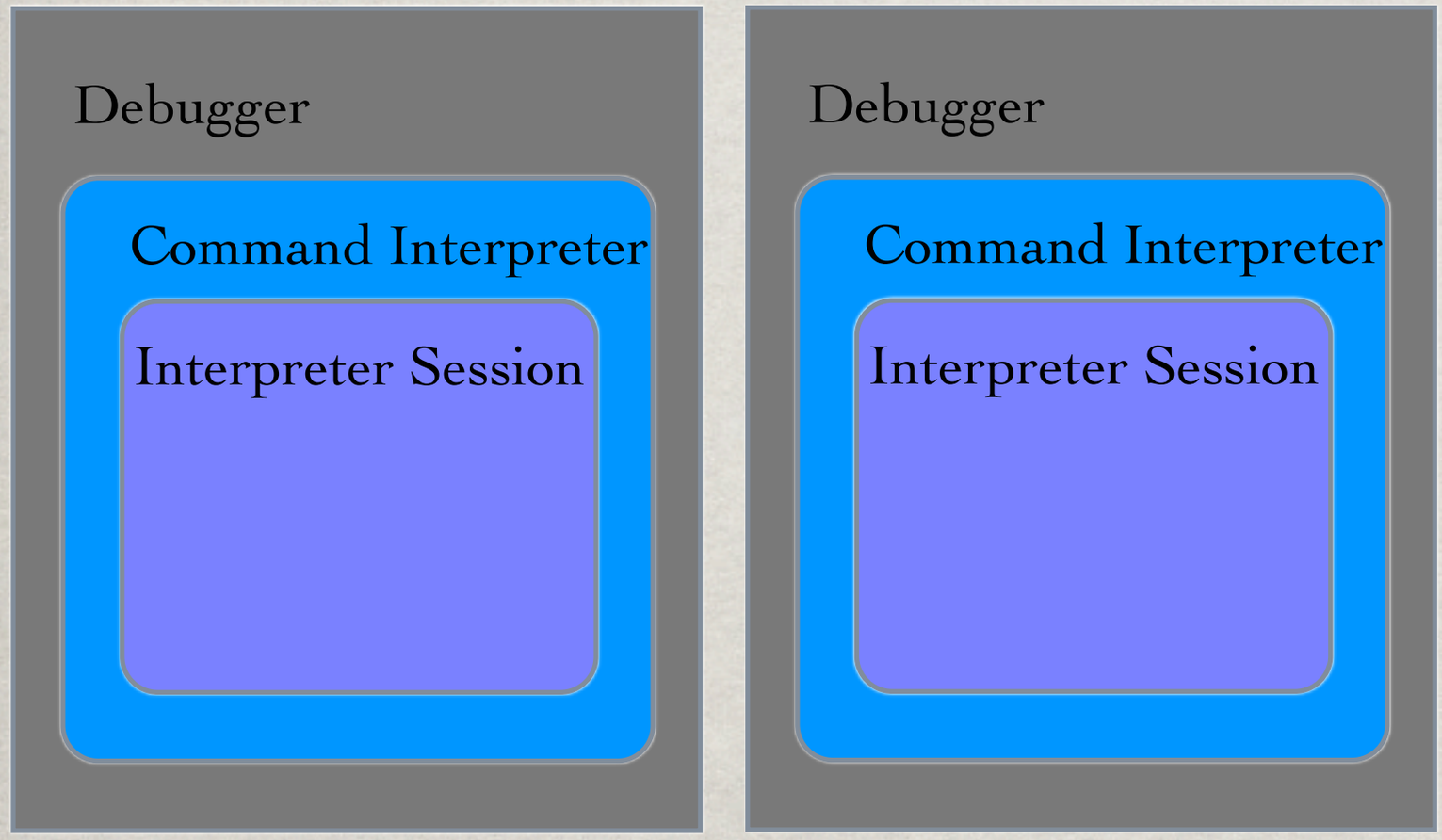
Debuggers in LLDB



>>> count = 24
>>> print count

>>> count = 9
>>> print count

Debuggers in LLDB



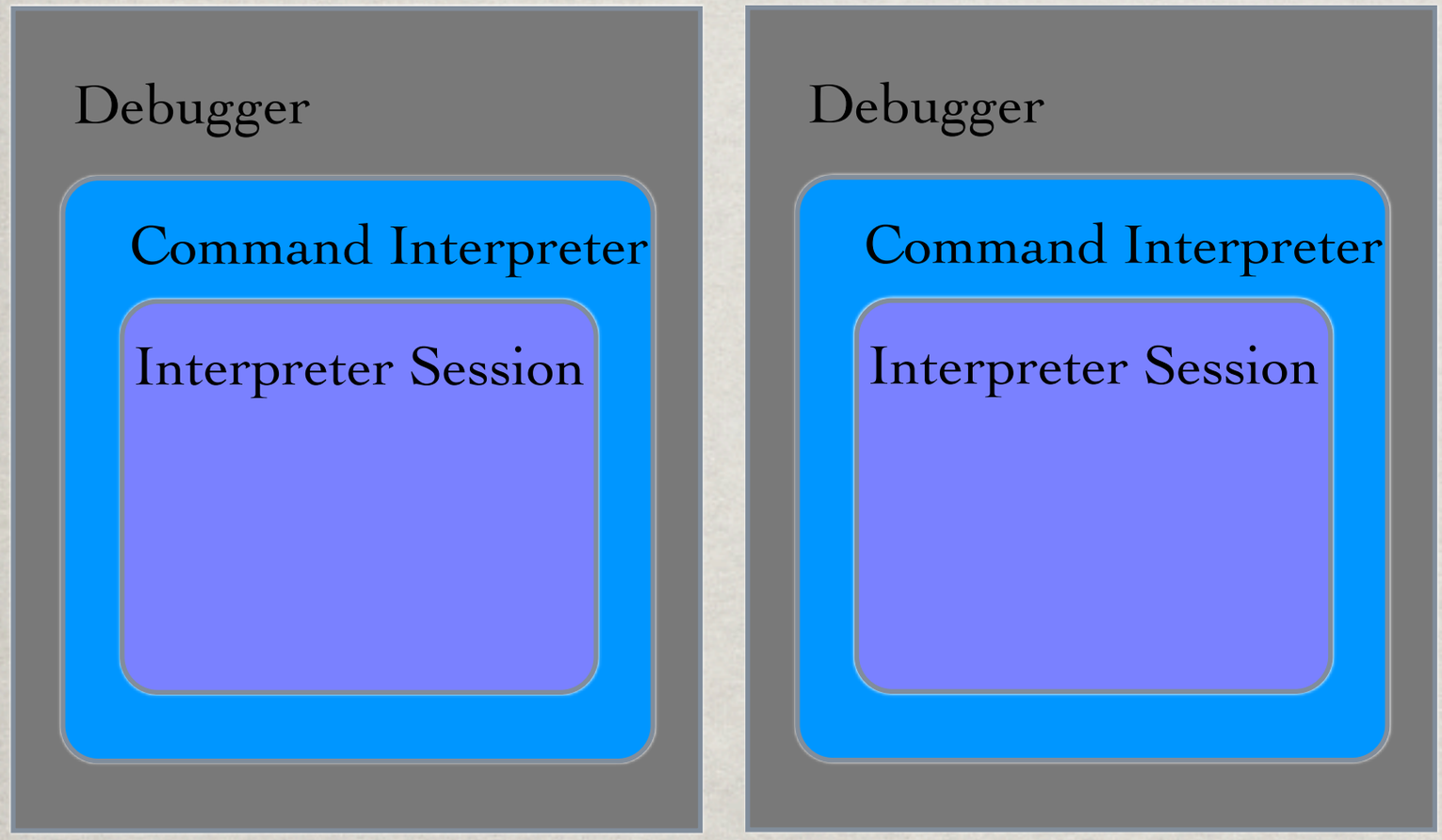

```
>>> count = 24
>>> print count
```

9

```
>>> count = 9
>>> print count
```

24

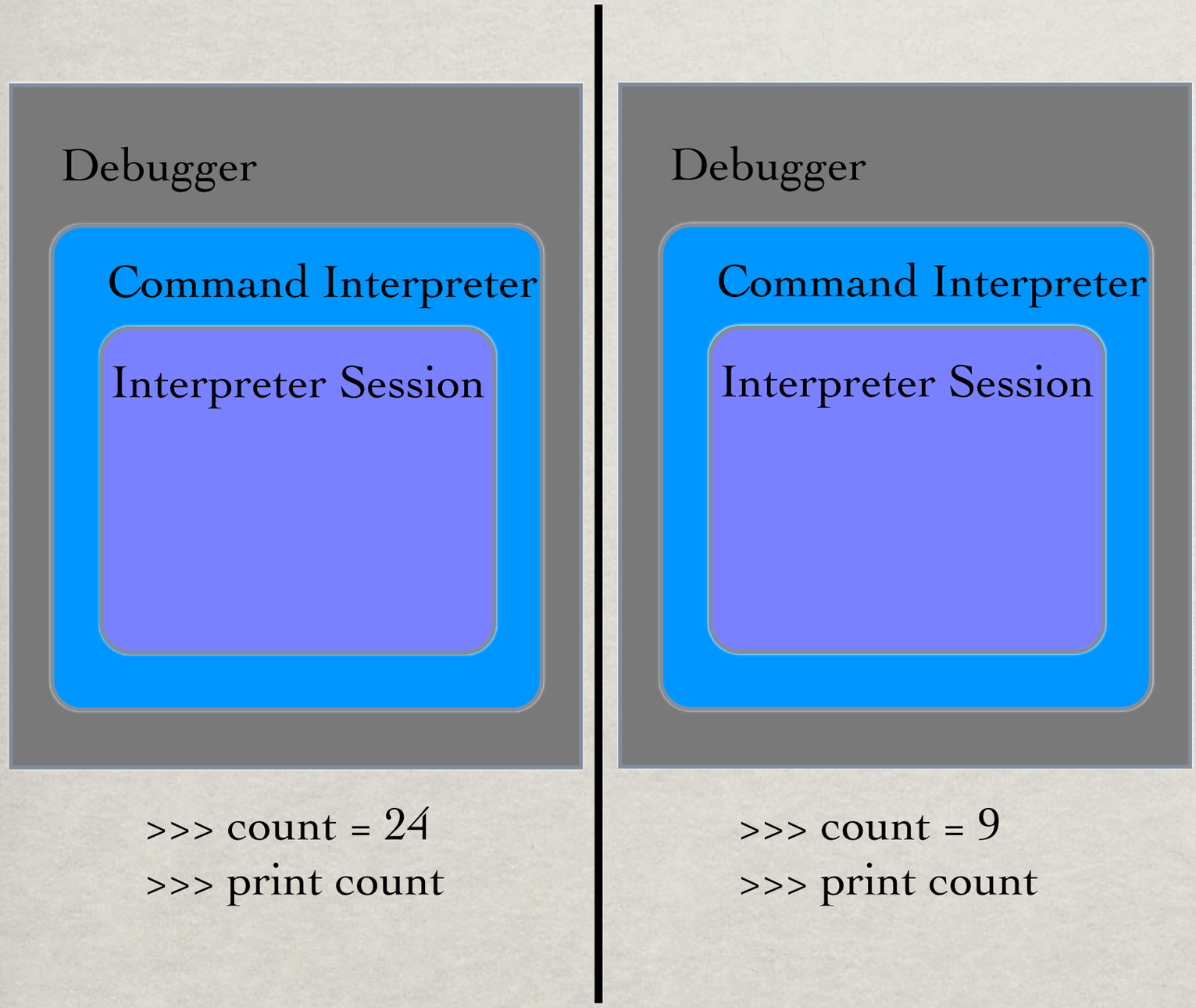
Debuggers in LLDB



>>> count = 24
>>> print count

>>> count = 9
>>> print count

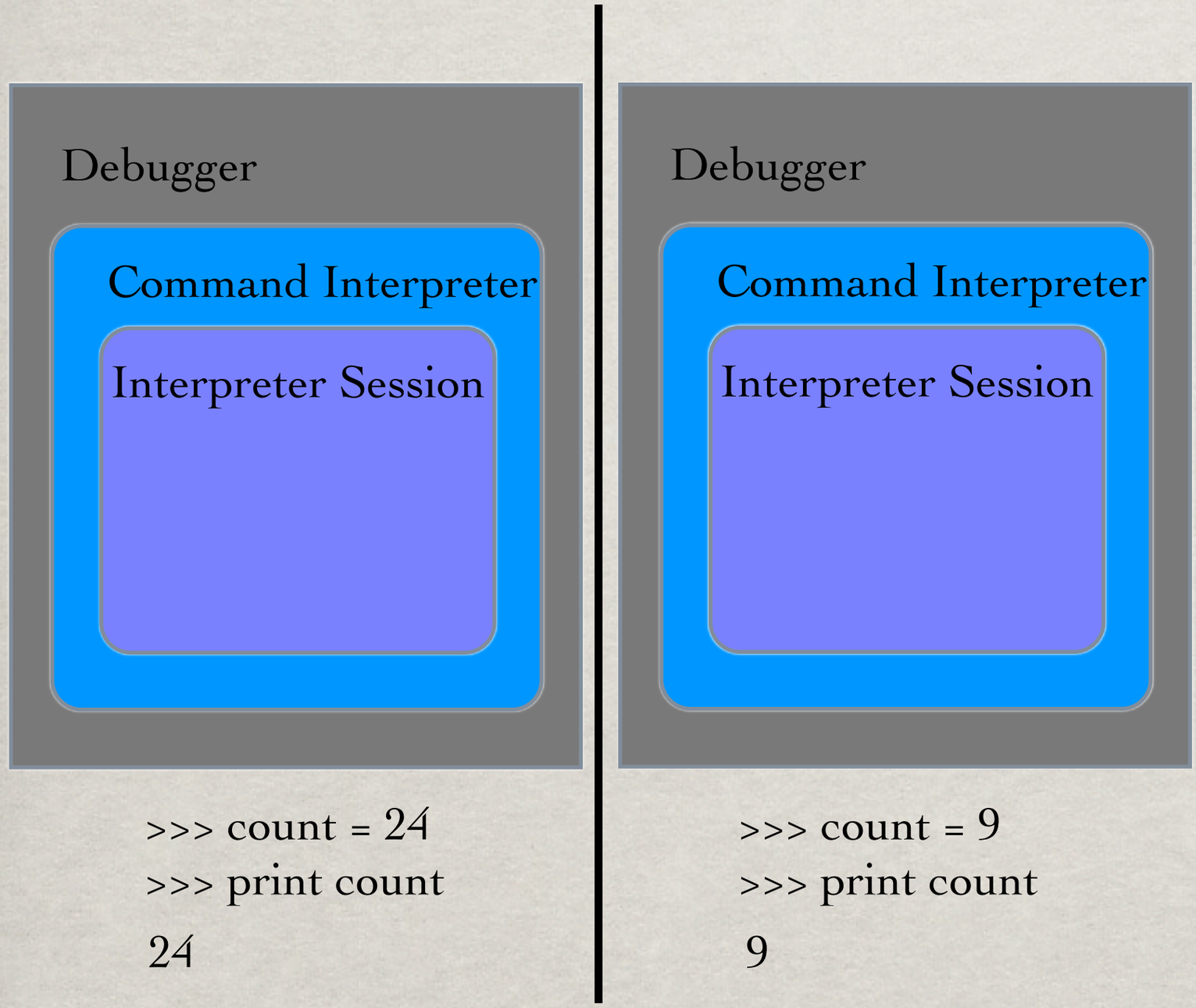
Debuggers in LLDB



>>> count = 24
>>> print count

>>> count = 9
>>> print count

Debuggers in LLDB




```
>>> count = 24
>>> print count
24
```

```
>>> count = 9
>>> print count
9
```

Demo...

Outline

- What is LLDB?
- Python in LLDB
- Particular Problems (& Solutions)
- Example Using Python to Debug Problem
- Questions

Why Scripting In A Debugger?

Why Scripting In A Debugger?

- Set REALLY useful conditional breakpoints

Why Scripting In A Debugger?

- Set REALLY useful conditional breakpoints
  - By caller's name

Why Scripting In A Debugger?

- Set REALLY useful conditional breakpoints
  - By caller's name
  - By caller's argument values

Why Scripting In A Debugger?

- Set REALLY useful conditional breakpoints
  - By caller's name
  - By caller's argument values
  - By thread

Why Scripting In A Debugger?

- Set REALLY useful conditional breakpoints
  - By caller's name
  - By caller's argument values
  - By thread

... and whether same thread hit it last time!

Why Scripting In A Debugger?

- Set REALLY useful conditional breakpoints

Why Scripting In A Debugger?

- Set REALLY useful conditional breakpoints
- Find specific data in large dynamic data structures

Why Scripting In A Debugger?

- Set REALLY useful conditional breakpoints
- Find specific data in large dynamic data structures
- Automatically record register values and program state

Why Scripting In A Debugger?

- Set REALLY useful conditional breakpoints
- Find specific data in large dynamic data structures
- Automatically record register values and program state
    - To a file...

Why Scripting In A Debugger?

- Set REALLY useful conditional breakpoints
- Find specific data in large dynamic data structures
- Automatically record register values and program state
    - To a file...
    - Each time a program point is hit...

Why Scripting In A Debugger?

- Set REALLY useful conditional breakpoints
- Find specific data in large dynamic data structures
- Automatically record register values and program state
    - To a file...
    - Each time a program point is hit...
    - Across multiple RUNS of the program

Why Scripting In A Debugger?

- Set REALLY useful conditional breakpoints
- Find specific data in large dynamic data structures
- Automatically record register values and program state

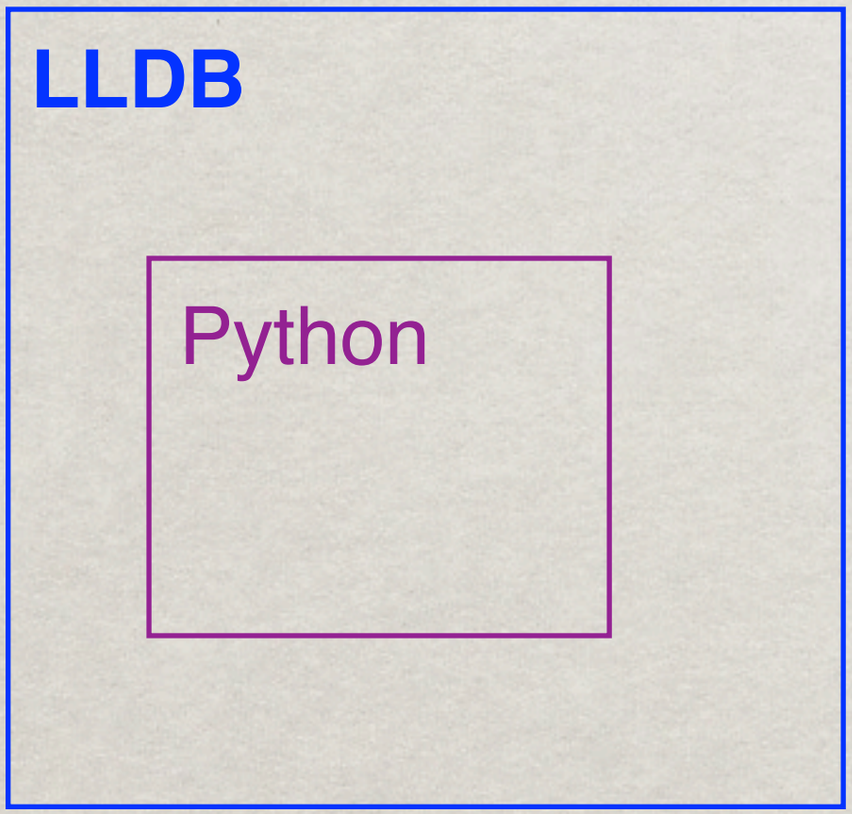
Why Scripting In A Debugger?

- Set REALLY useful conditional breakpoints
- Find specific data in large dynamic data structures
- Automatically record register values and program state

Why Scripting In A Debugger?

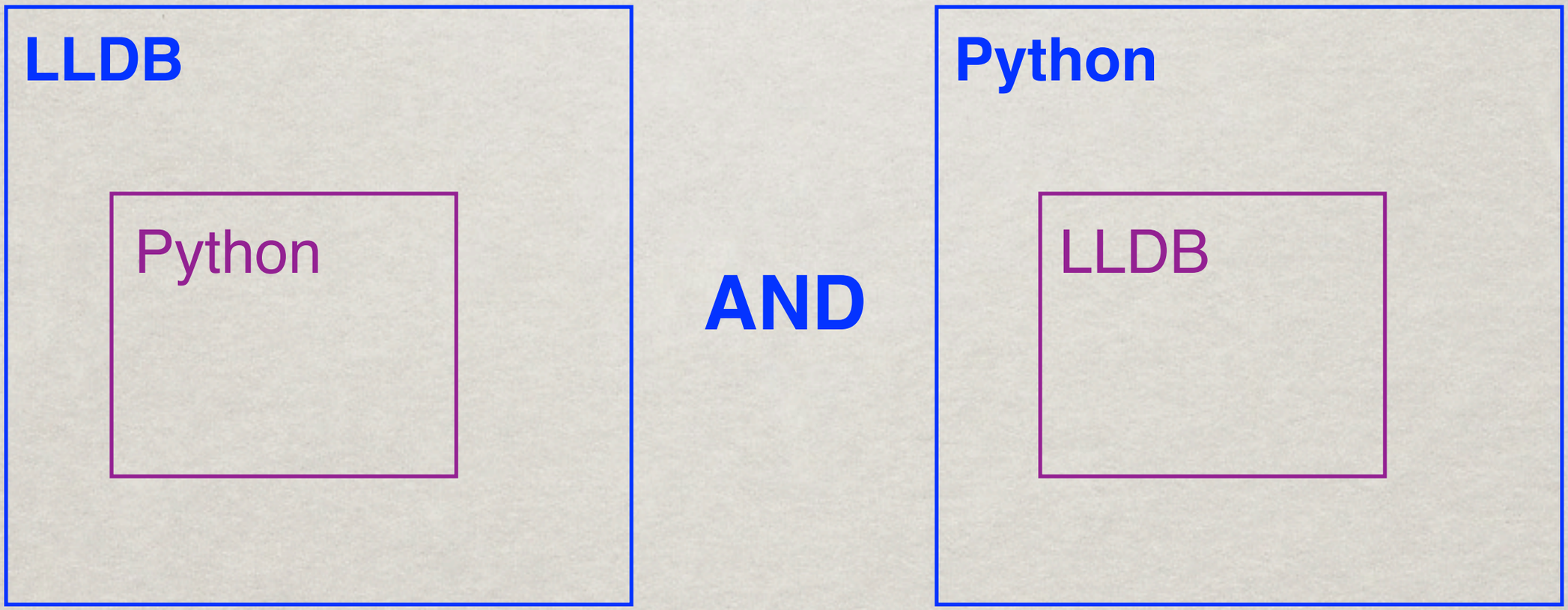
- Set REALLY useful conditional breakpoints
- Find specific data in large dynamic data structures
- Automatically record register values and program state
- Automated testing/QA

What Is Where?



Python is accessible from LLDB.

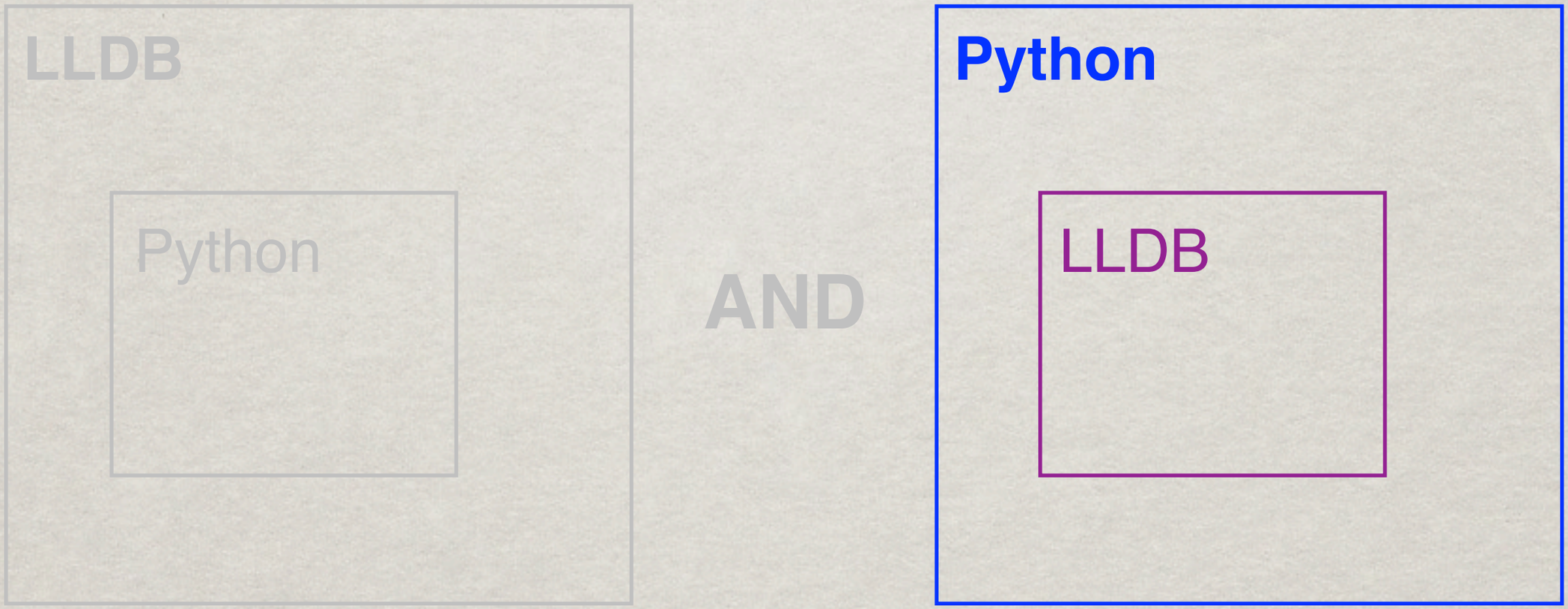
What Is Where?



Python is accessible from LLDB.

LLDB is accessible from Python.

What Is Where?



Python is accessible from LLDB.

LLDB is accessible from Python.

LLDB Creates Python Module

%

LLDB Creates Python Module

```
% python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

LLDB Creates Python Module

```
% python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import lldb
>>>
```

LLDB Creates Python Module

```
% python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import lldb
>>> dbg = lldb.SBDebugger.Create()
>>>
```

LLDB Creates Python Module

```
% python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import lldb
>>>  dbg = lldb.SBDebugger.Create()
>>>  target = dbg.CreateTarget (“/bin/ls”)
>>>
```

LLDB Creates Python Module

```
% python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import lldb
>>> dbg = lldb.SBDebugger.Create()
>>> target = dbg.CreateTarget (“/bin/ls”)
>>> target.BreakpointCreateByName (“main”)
>>>
```

LLDB Creates Python Module

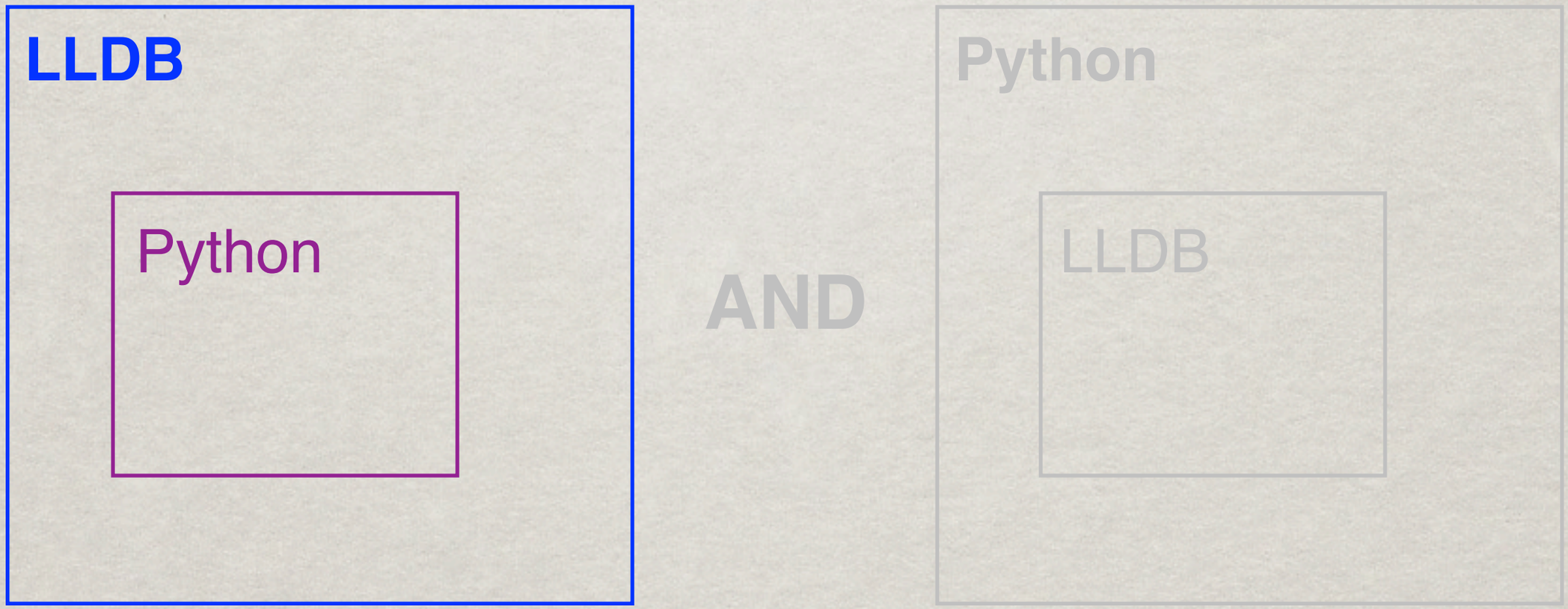
```
% python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import lldb
>>> dbg = lldb.SBDebugger.Create()
>>> target = dbg.CreateTarget (“/bin/ls”)
>>> target.BreakpointCreateByName (“main”)
>>> process = target.LaunchSimple (None, None, None)
```

LLDB Creates Python Module

```
% python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import lldb
>>> dbg = lldb.SBDebugger.Create()
>>> target = dbg.CreateTarget ("/bin/ls")
>>> target.BreakpointCreateByName ("main")
>>> process = target.LaunchSimple (None, None, None)
```

LLDB API functions

What Is Where?



Python is accessible from LLDB.

LLDB is accessible from Python.

Python In LLDB

Python In LLDB

- LLDB contains full, complete Python Interpreter

Python In LLDB

- LLDB contains full, complete Python Interpreter
- Many ways to access Python in LLDB
  - one-line script command
  - interactive interpreter
  - breakpoint commands

Python In LLDB

- LLDB contains full, complete Python Interpreter
- Many ways to access Python in LLDB
  - one-line script command
  - interactive interpreter
  - breakpoint commands

```
(lldb) script hex (123456)
'0x1e240'
(lldb)
```

Python In LLDB

- LLDB contains full, complete Python Interpreter
- Many ways to access Python in LLDB
  - one-line script command
  - interactive interpreter
  - breakpoint commands

```
(lldb) script
Python Interactive Interpreter. To exit type 'exit()', 'quit()'
or Ctrl-D.
>>>
```

Python In LLDB

- LLDB contains full, complete Python Interpreter
- Many ways to access Python in LLDB
  - one-line script command
  - interactive interpreter
  - breakpoint commands

```
(lldb) breakpoint command add -s python 1
Enter your Python commands. Type 'DONE' to end.
>
```

LLDB Extensions For Python

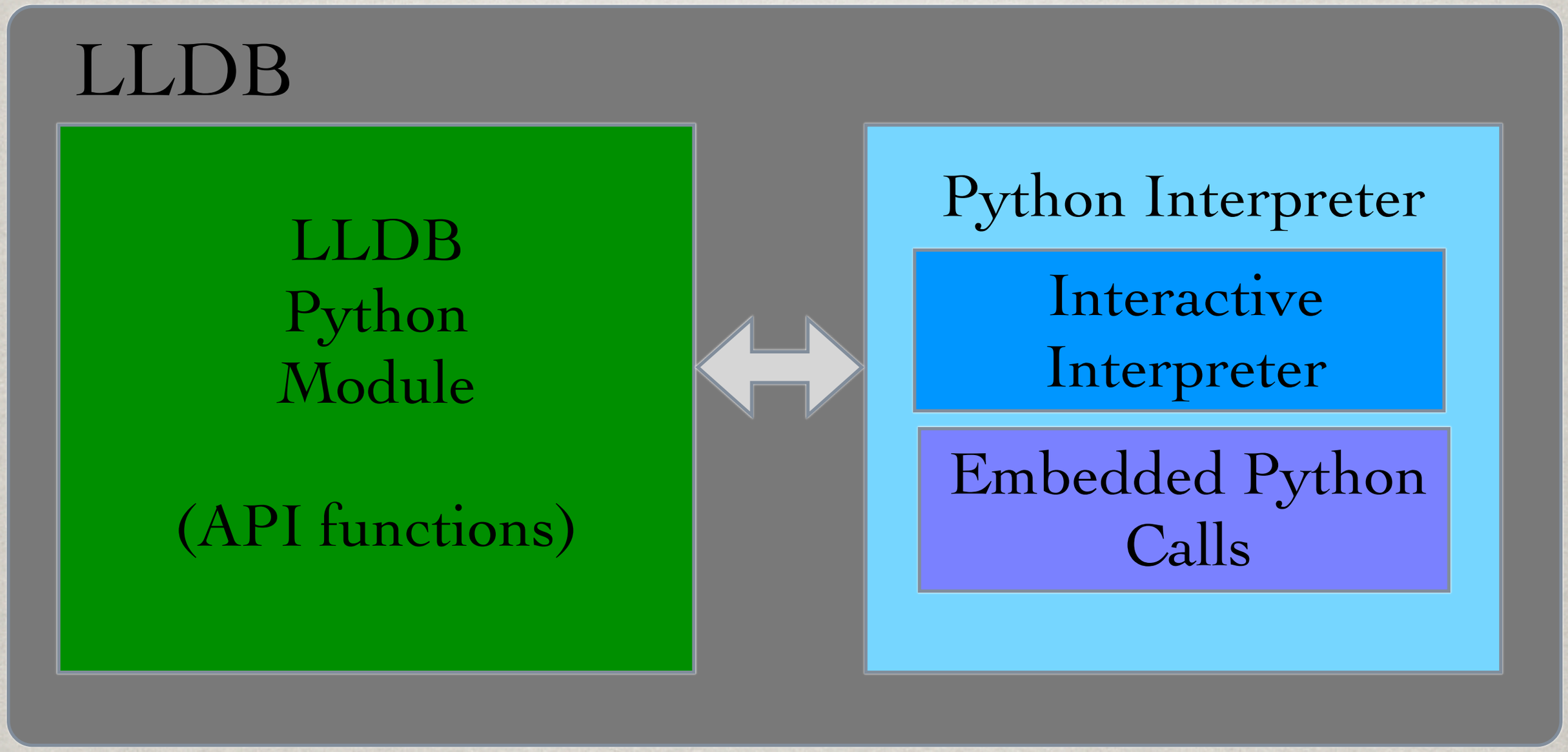
- API functions (automatically imported)
  - Create, access & manipulate debugger objects/state

- Pre-loaded LLDB objects into Python variables

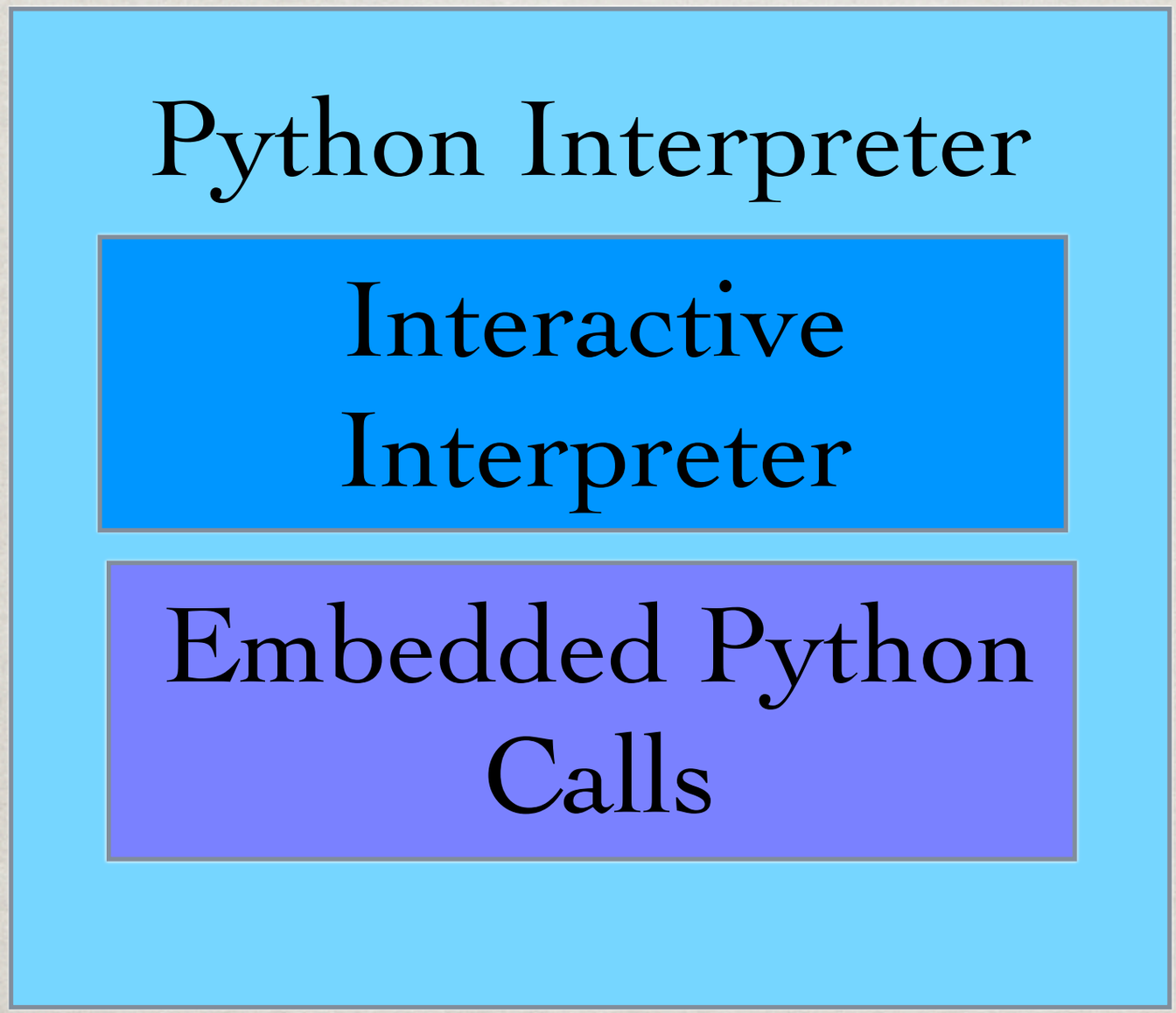
```
lldb.target, lldb.process, lldb.thread, lldb.frame
```

- Single persistent Python dictionary for entire debugger session

Python In LLDB - More Detail



Python In LLDB - More Detail



Implementing Interactive Python Interpreter

Implementing Interactive Python Interpreter

- Wrote our own interactive interpreter module (in Python)
  - Inherited InteractiveConsole class ('code' module)
  - Takes dictionary as parameter
  - Executes all code in context of dictionary

Implementing Interactive Python Interpreter

- Wrote our own interactive interpreter module (in Python)
  - Inherited InteractiveConsole class ('code' module)
  - Takes dictionary as parameter
  - Executes all code in context of dictionary
- Used Python C/API functions to:
  - Initialize Python interpreter
    - Initialize thread capabilities
    - Turn OFF Python's signal handlers
  - Import our interactive interpreter module

Implementing One-Line Script Commands

- Same mechanism as interactive interpreter...
- ...just call different method in our module

How The Dictionaries Work (1)

new Debugger();

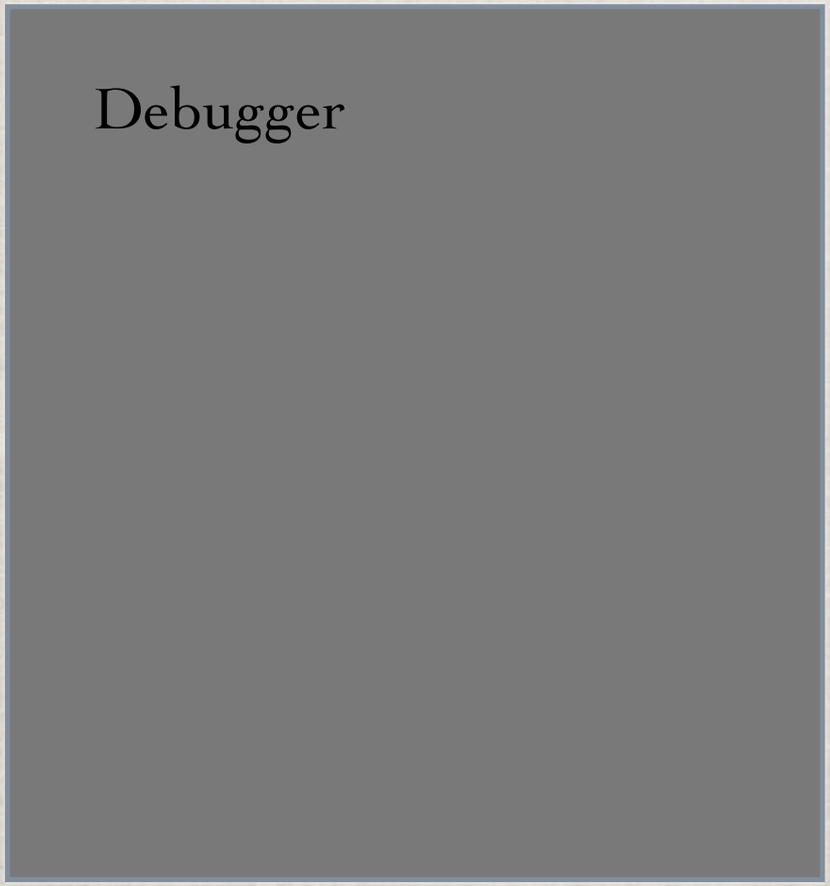
Python Interpreter

Global Dictionary

run_one_line : <code>
run_python_interpreter : <code>

How The Dictionaries Work (1)

new Debugger();



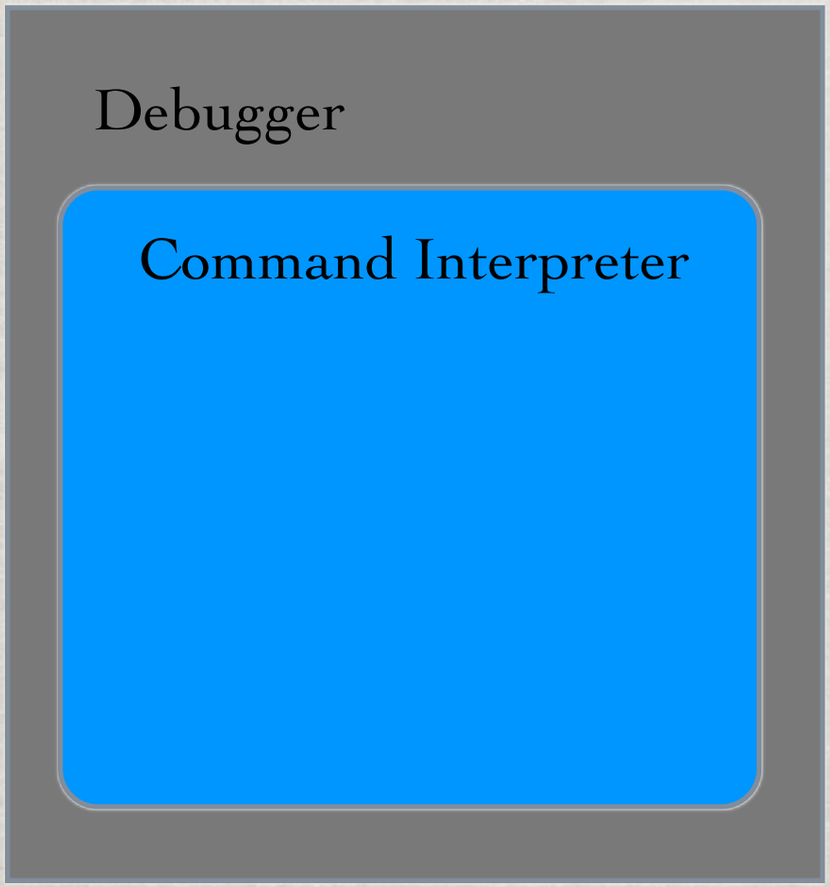
Python Interpreter

Global Dictionary

run_one_line : <code>
run_python_interpreter : <code>

How The Dictionaries Work (1)

new Debugger();



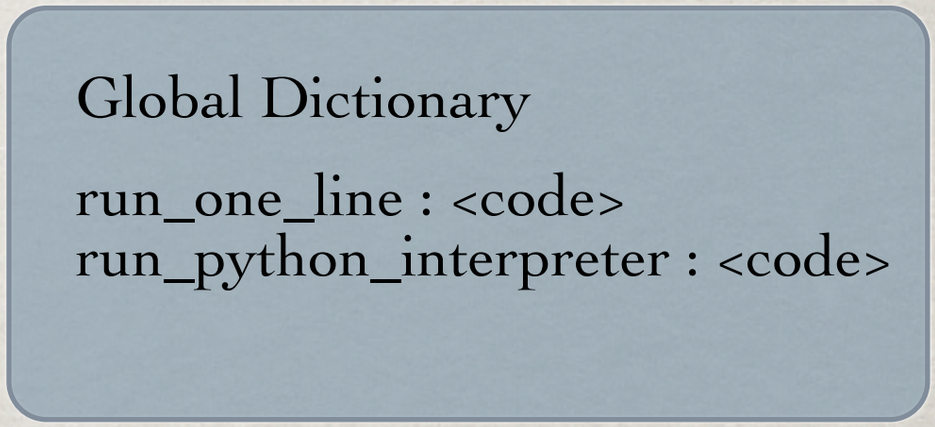
Python Interpreter

Global Dictionary

run_one_line : <code>
run_python_interpreter : <code>

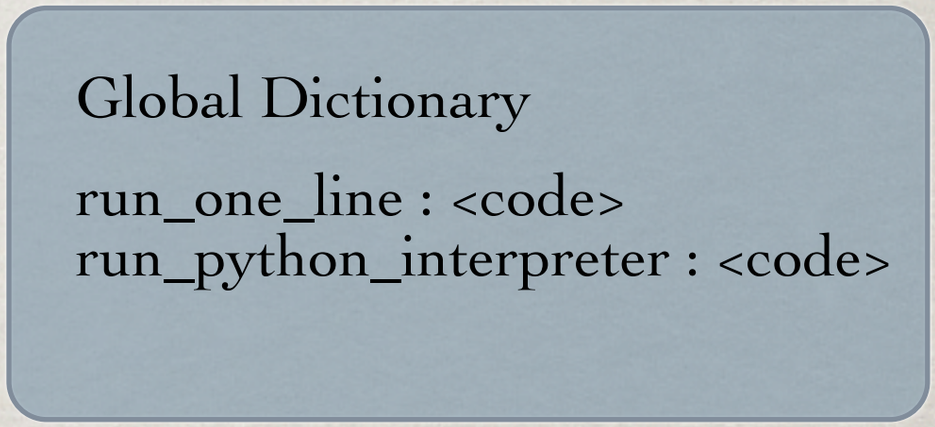
How The Dictionaries Work (1)

new Debugger();



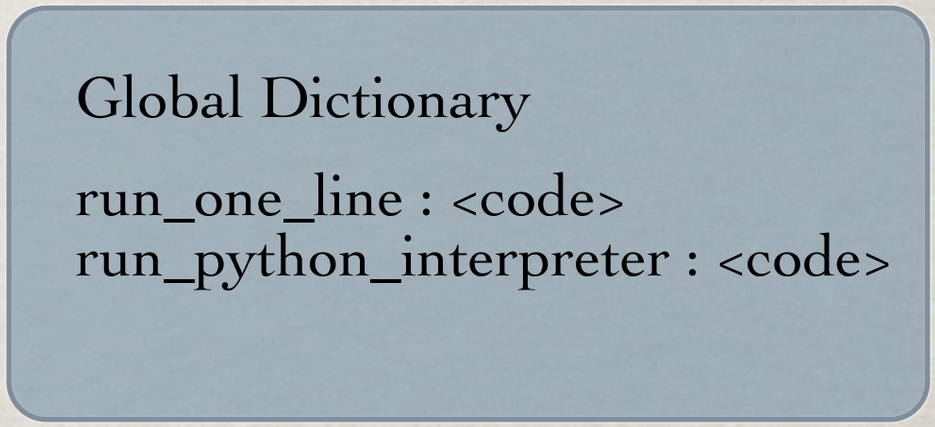
How The Dictionaries Work (1)

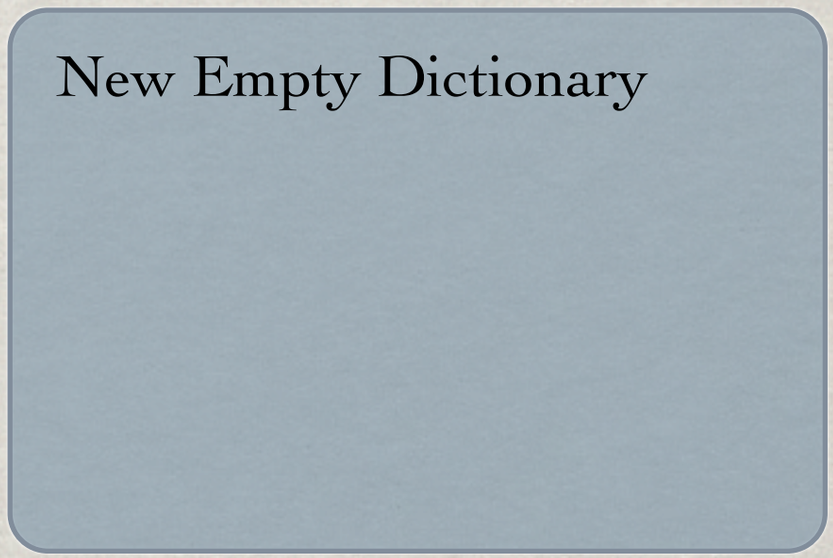
new Debugger();



How The Dictionaries Work (1)

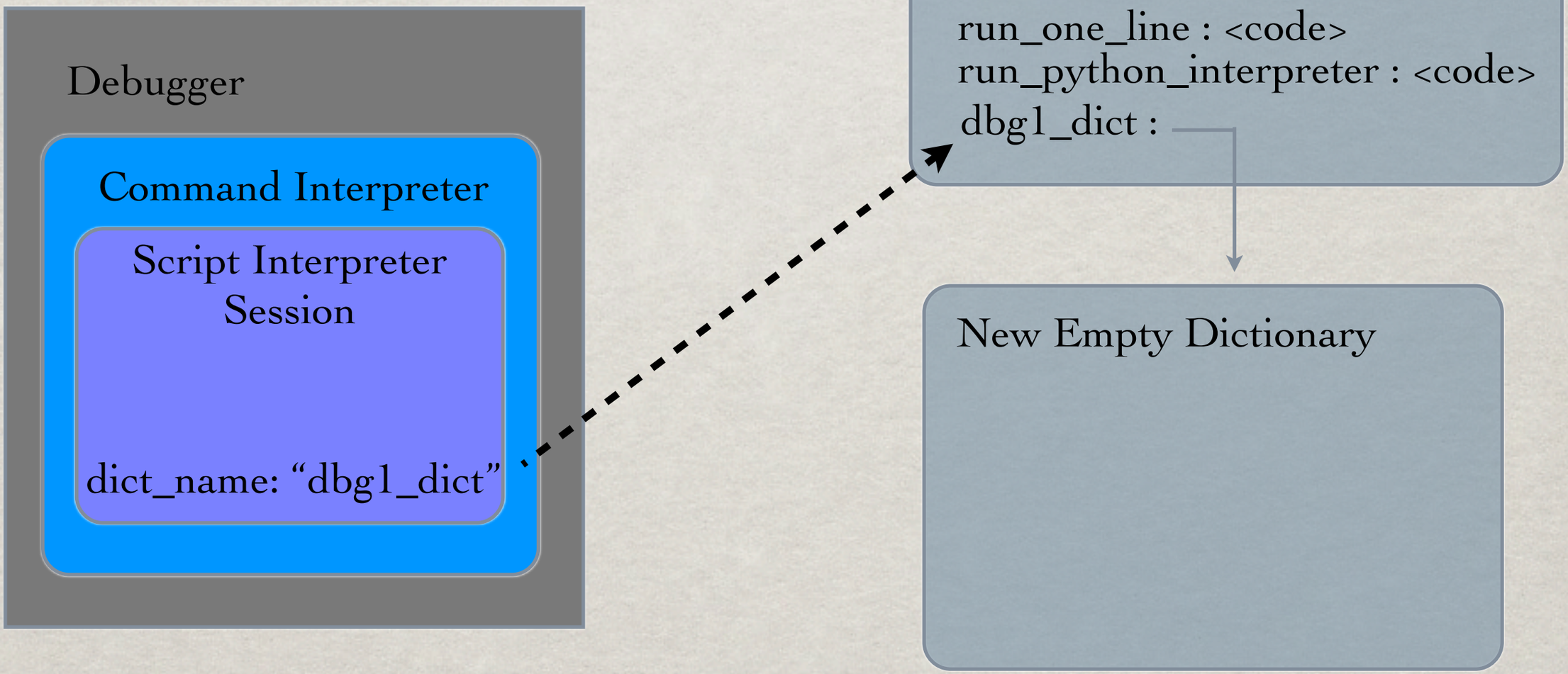
new Debugger();



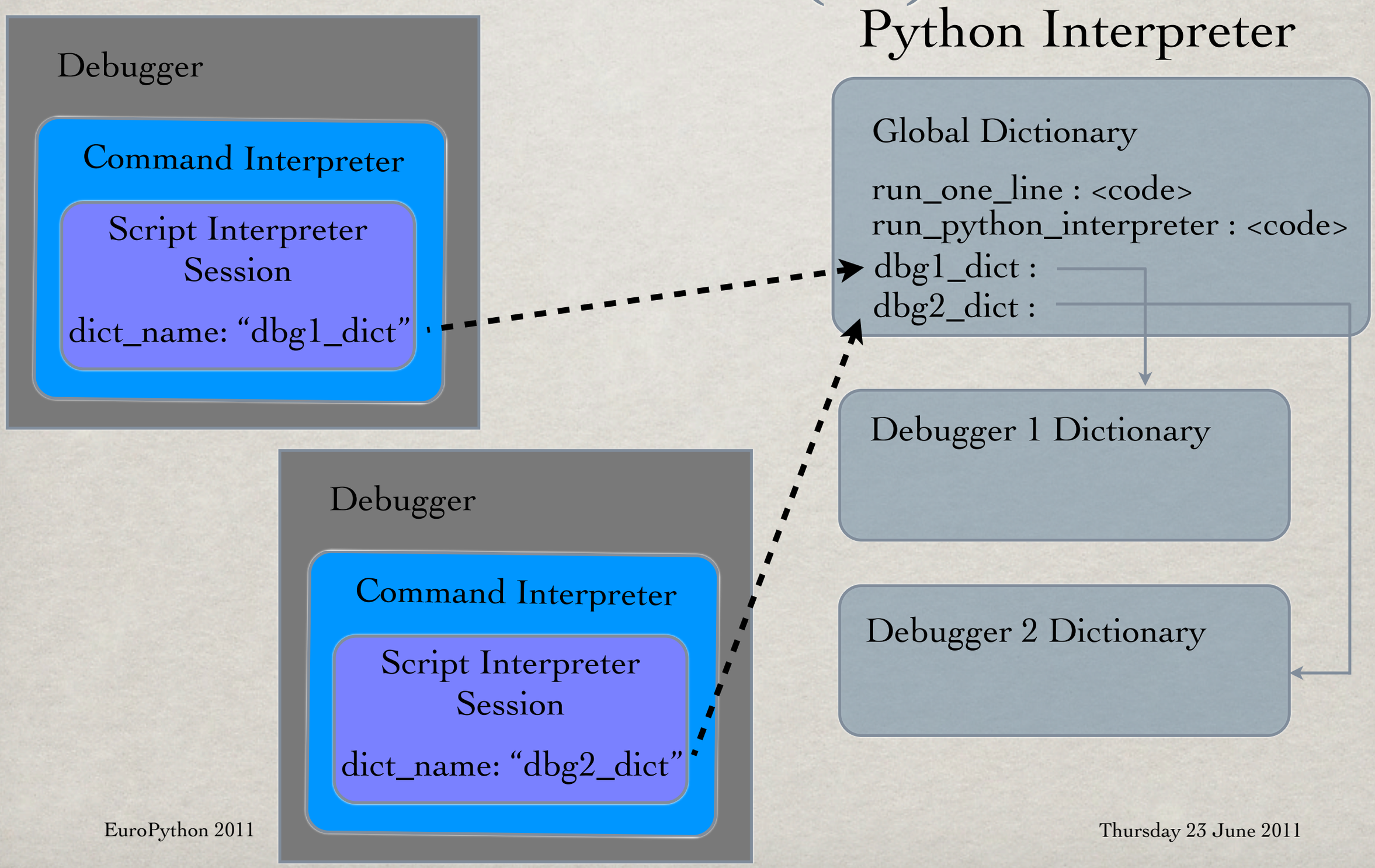


How The Dictionaries Work (1)

new Debugger();

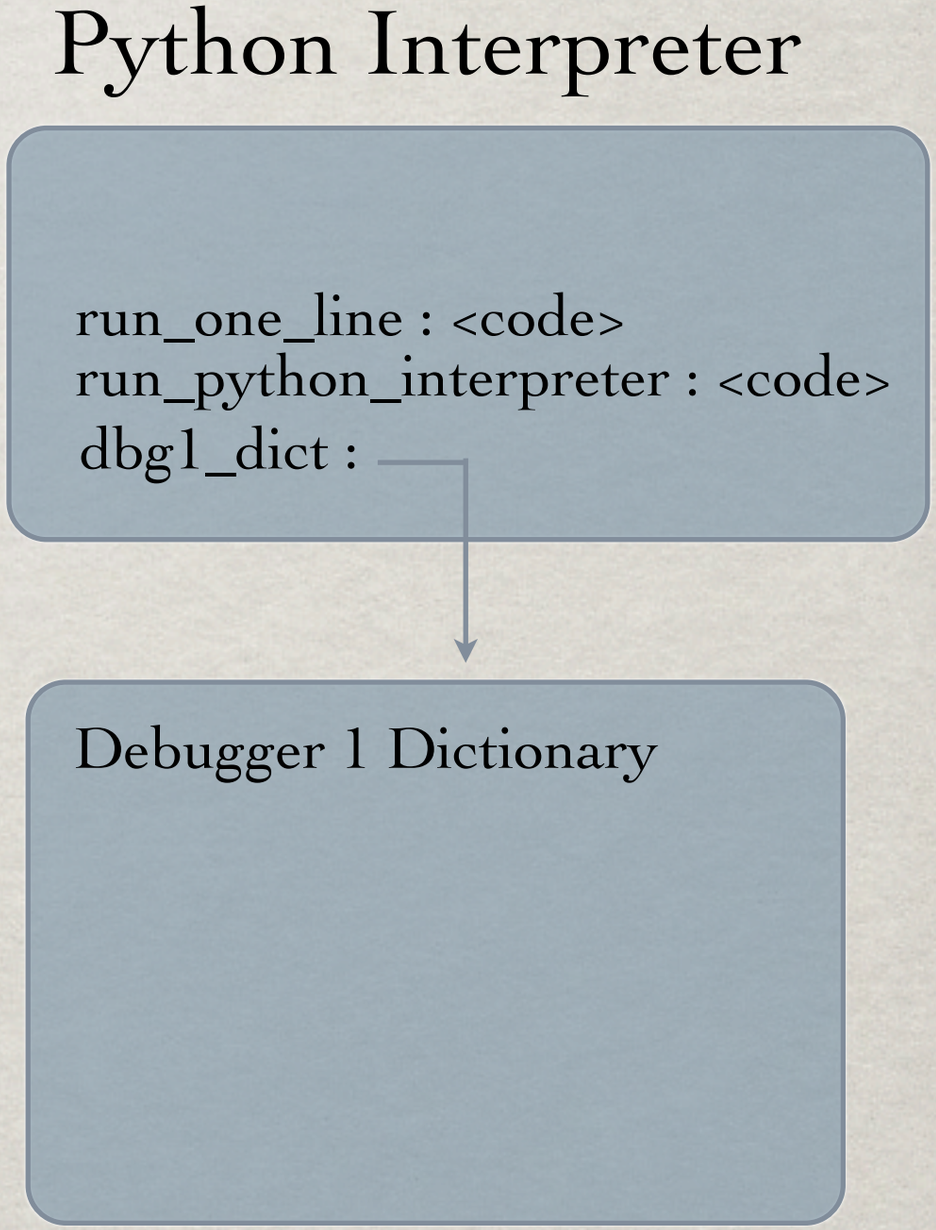


How The Dictionaries Work (2)



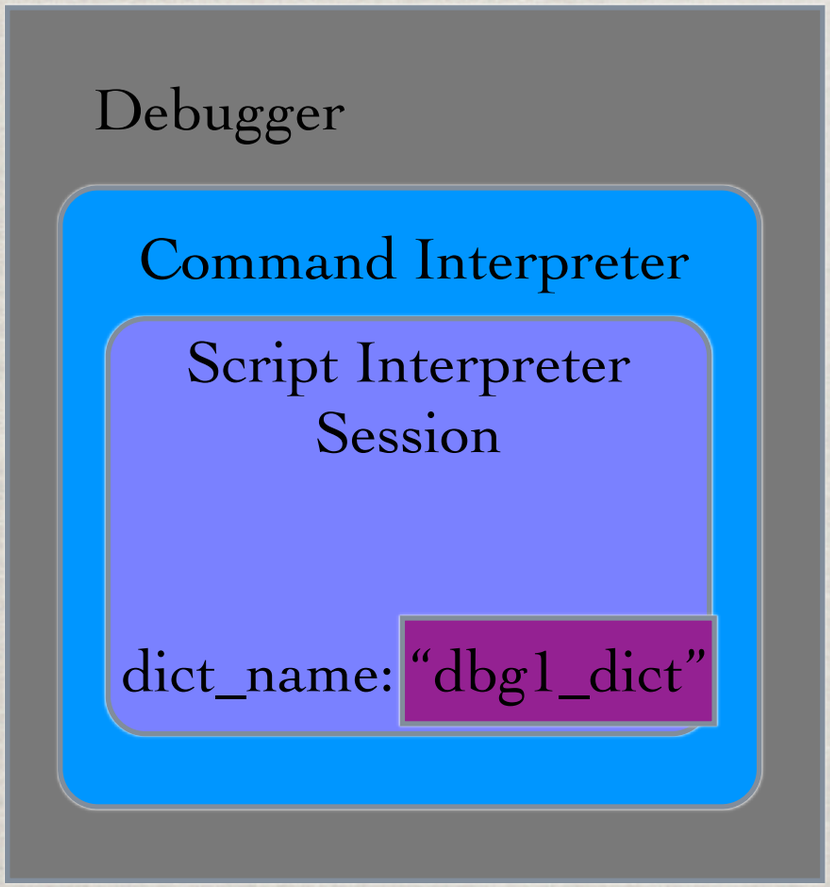
Invoking One-Line Script Commands

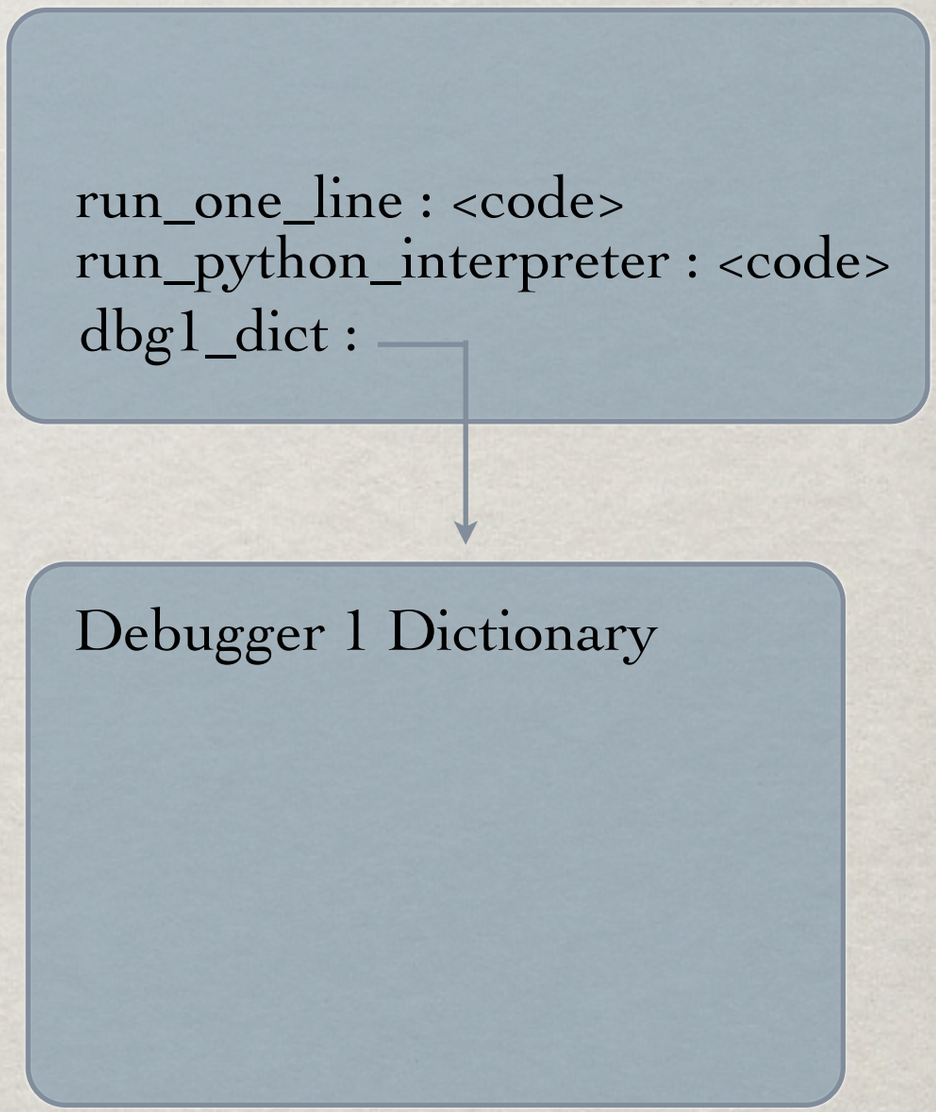
(lldb) script hex (3856)



Invoking One-Line Script Commands

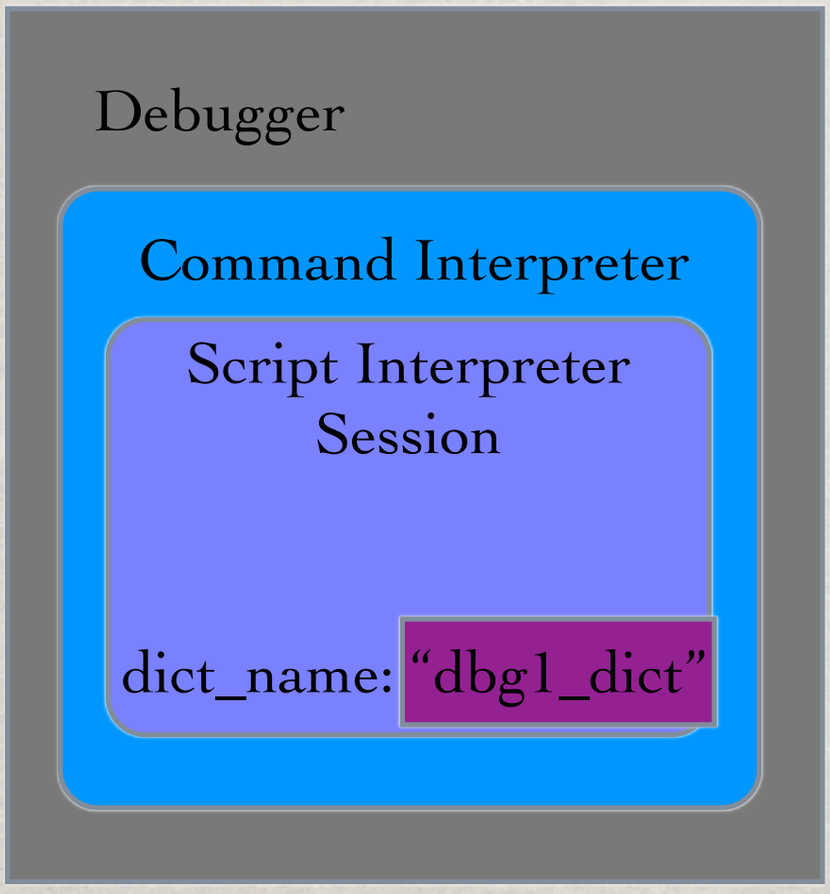
(lldb) script hex (3856)

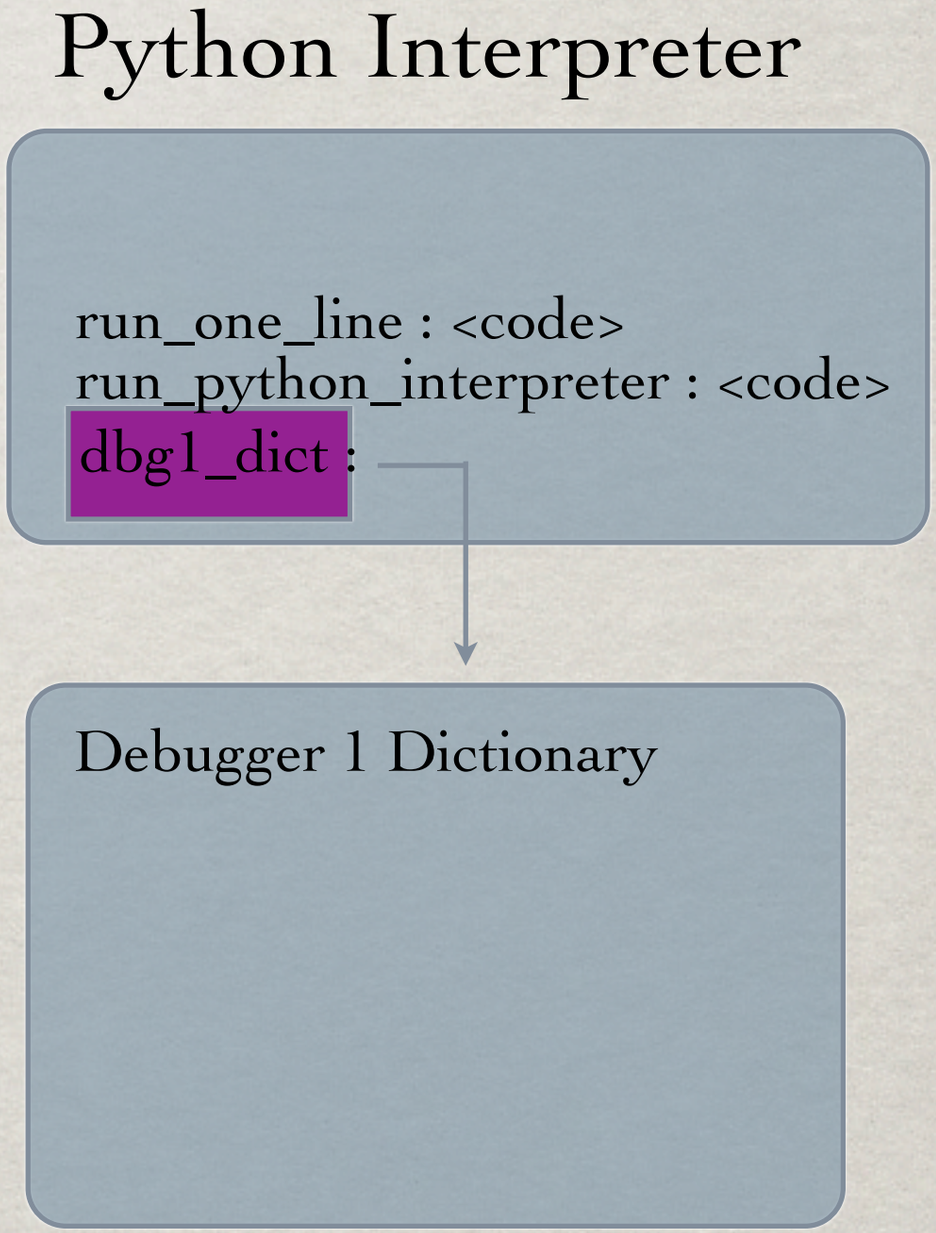




Invoking One-Line Script Commands

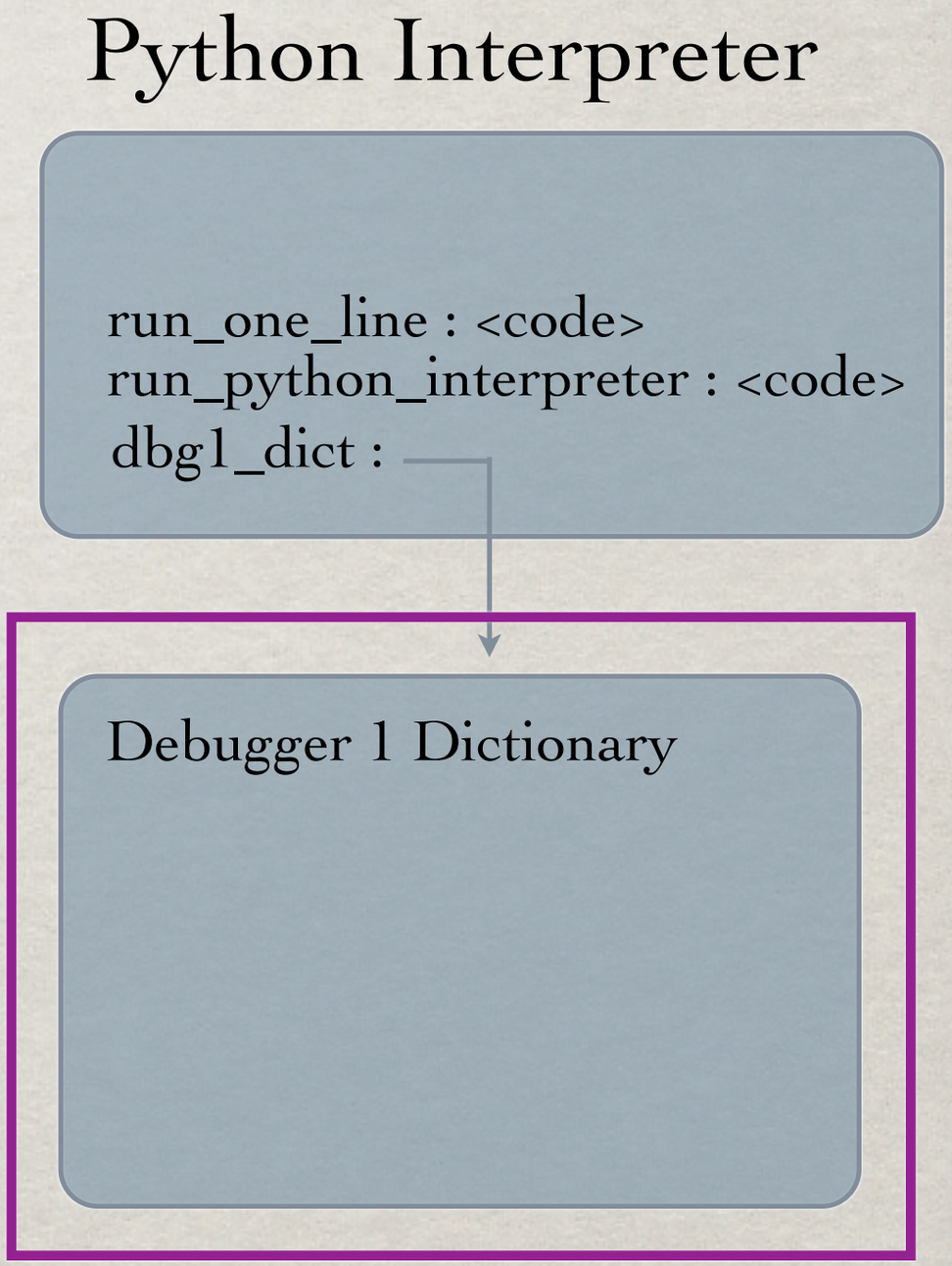
(lldb) script hex (3856)





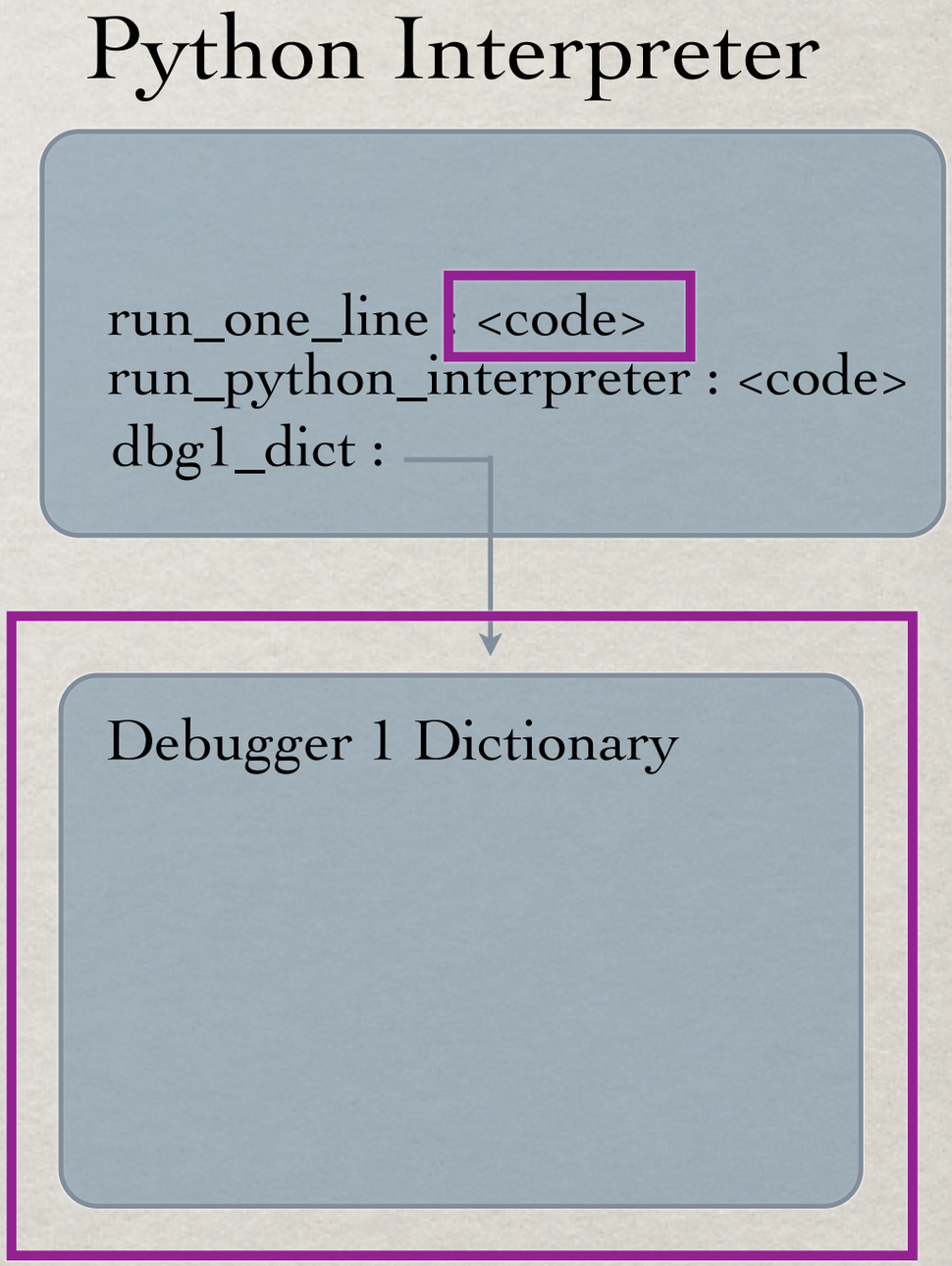
Invoking One-Line Script Commands

(lldb) script hex (3856)



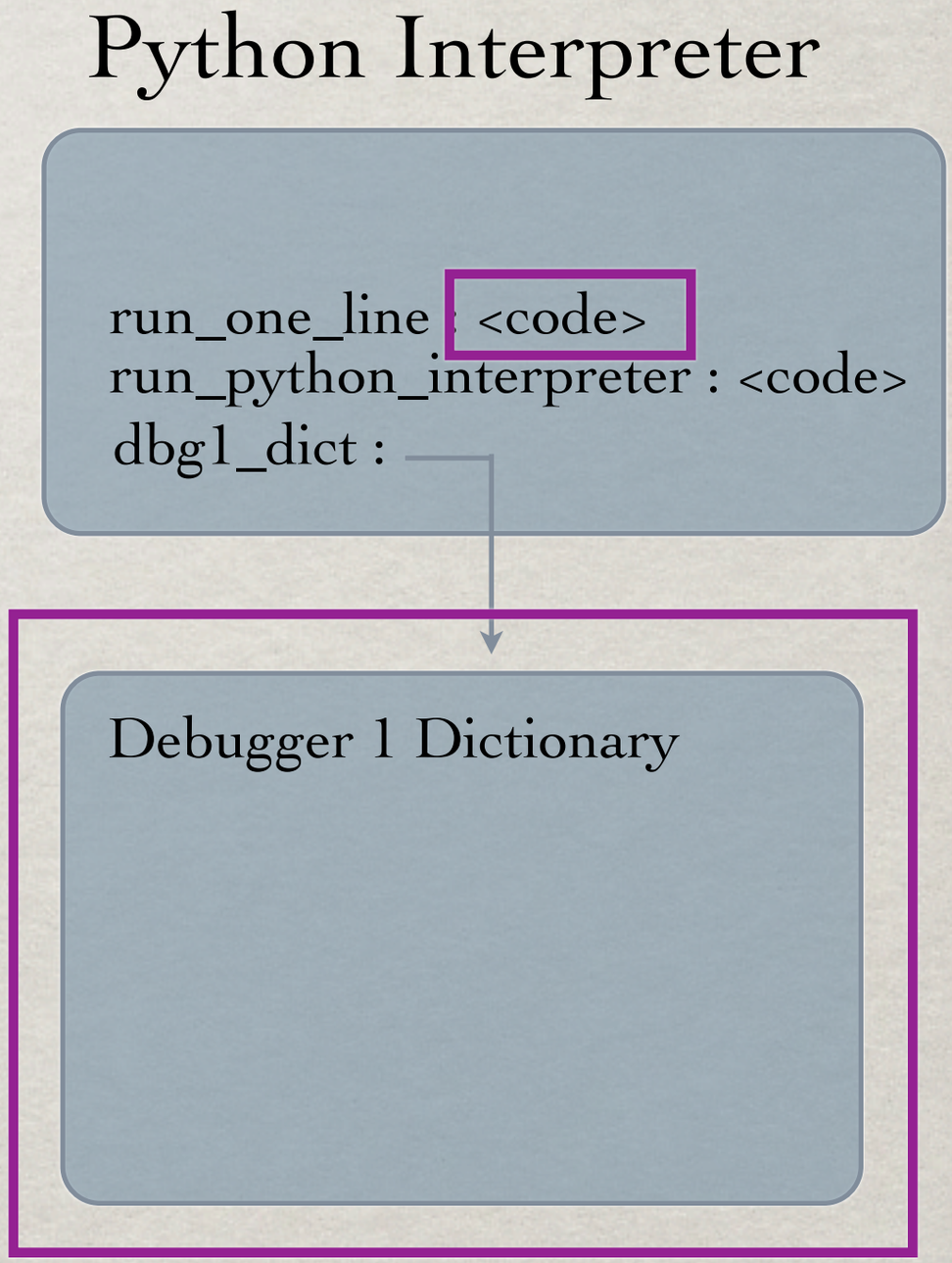
Invoking One-Line Script Commands

(lldb) script hex (3856)



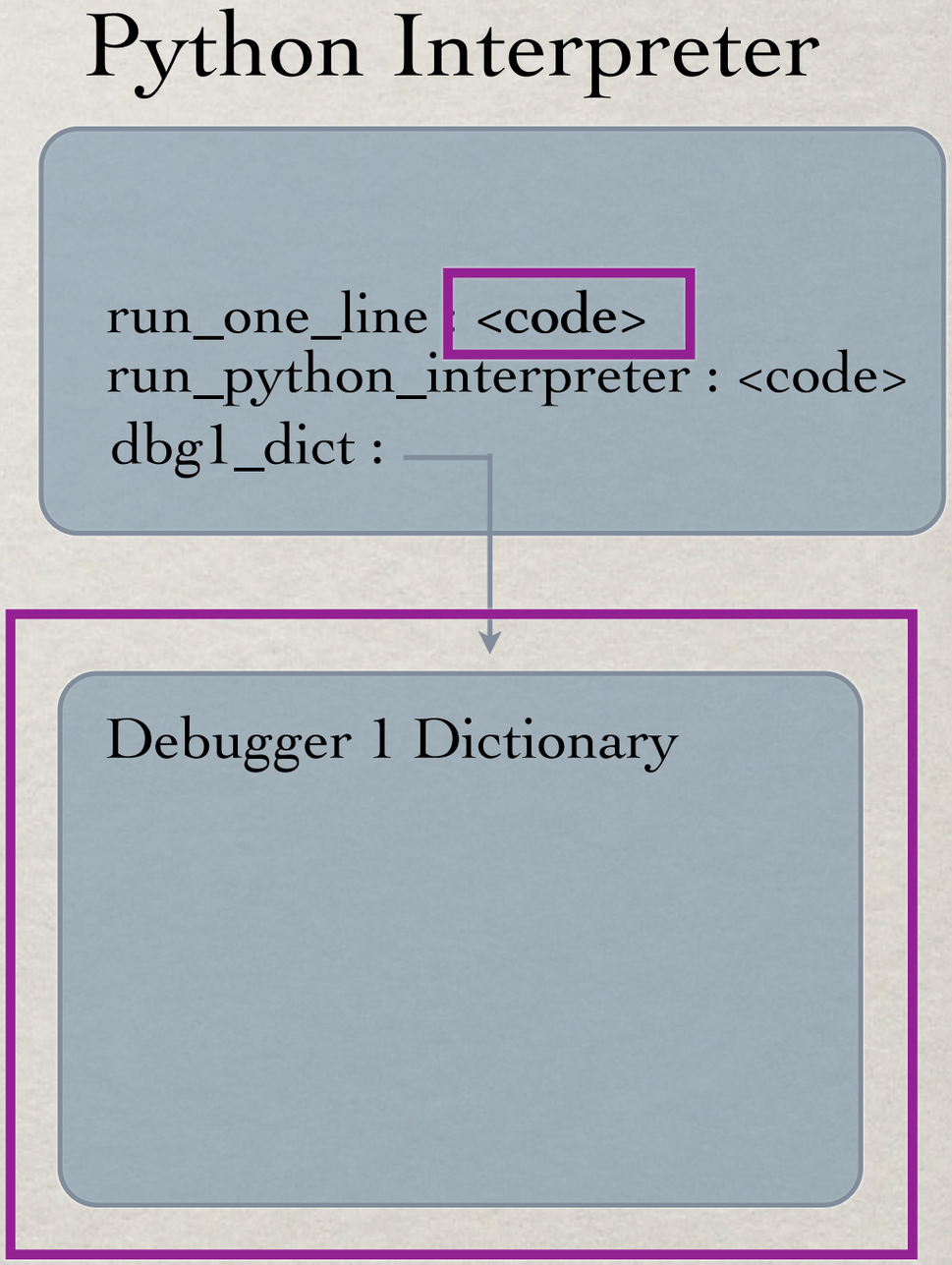
Invoking One-Line Script Commands

(lldb) script hex (3856)



Invoking One-Line Script Commands

(lldb) script hex (3856)



Invoking One-Line Script Commands

hex (3856)

<code>

Debugger 1 Dictionary

Invoking One-Line Script Commands

hex (3856)

<code>

Debugger 1 Dictionary

Invoking One-Line Script Commands

<code>

hex (3856)

Debugger 1 Dictionary

Invoking One-Line Script Commands

<code>

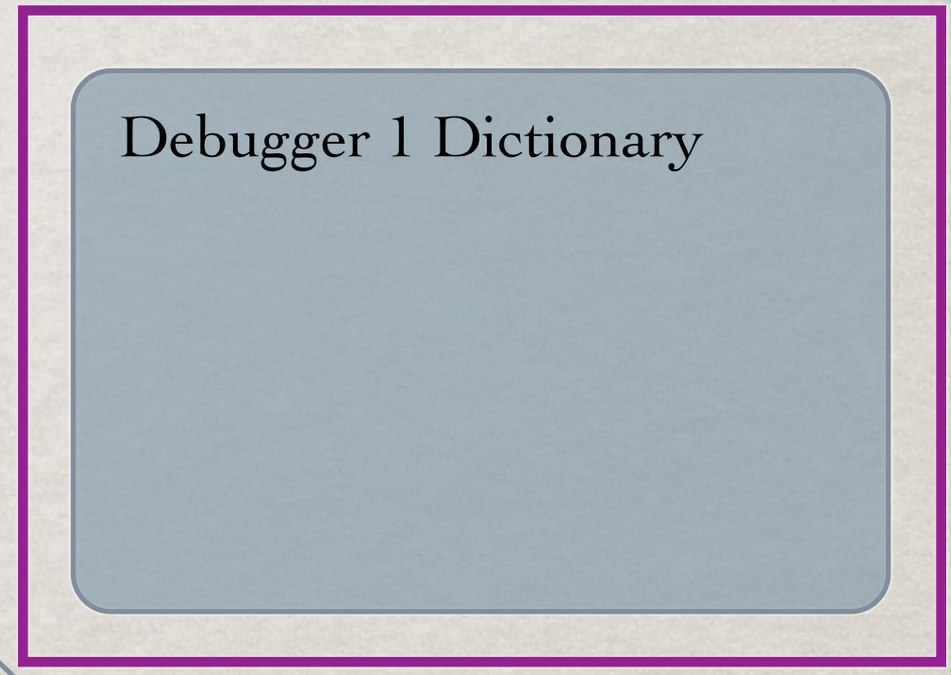
hex (3856)

Debugger 1 Dictionary

Invoking One-Line Script Commands

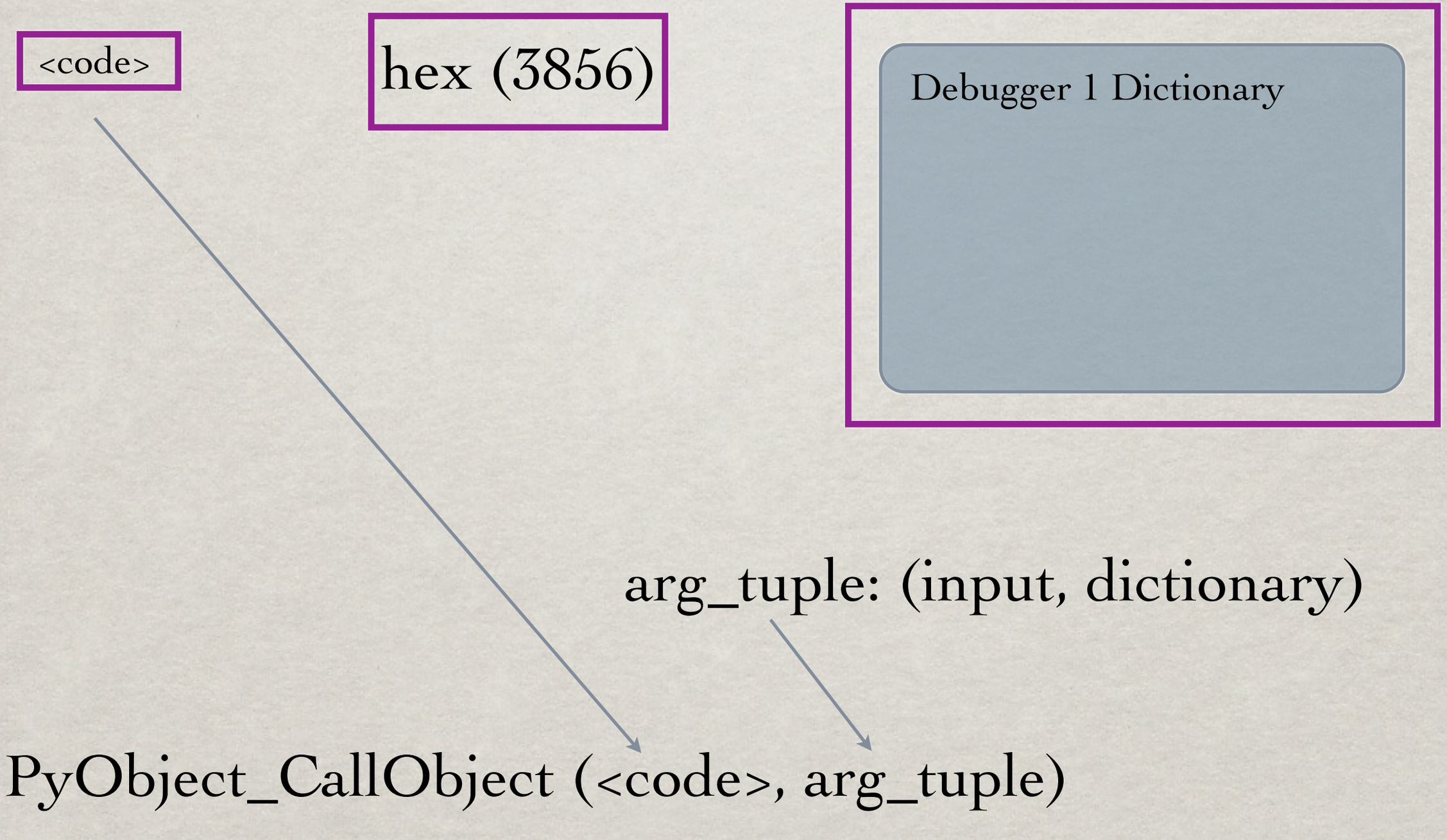
<code>

hex (3856)



arg_tuple: (input, dictionary)

Invoking One-Line Script Commands

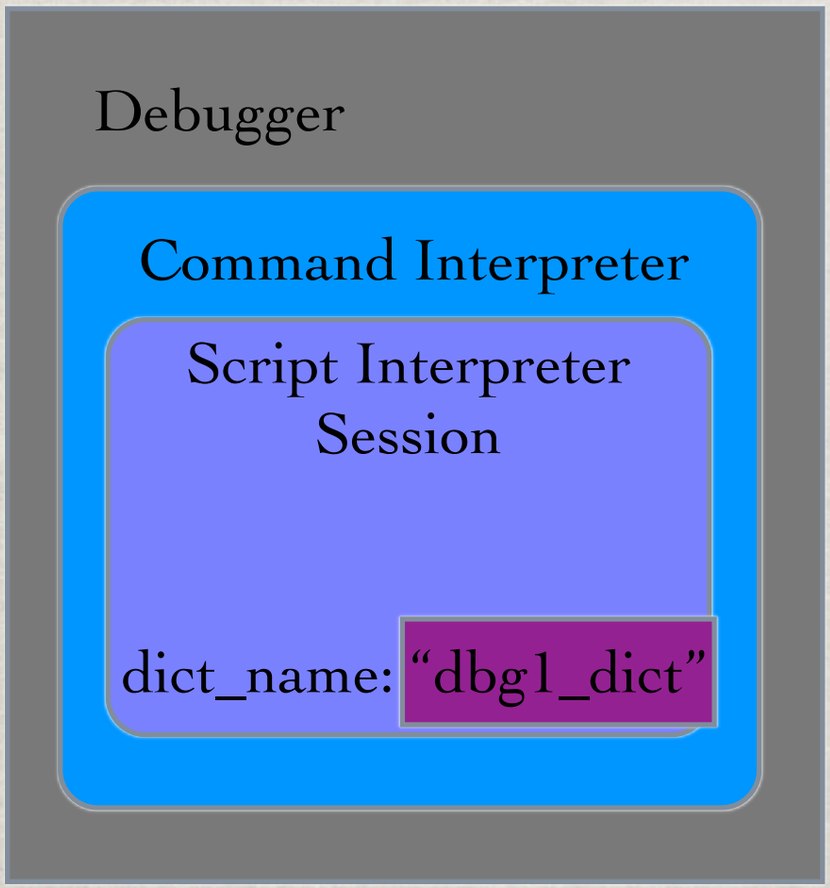


Invoking Interactive Script Interpreter

(lldb) script

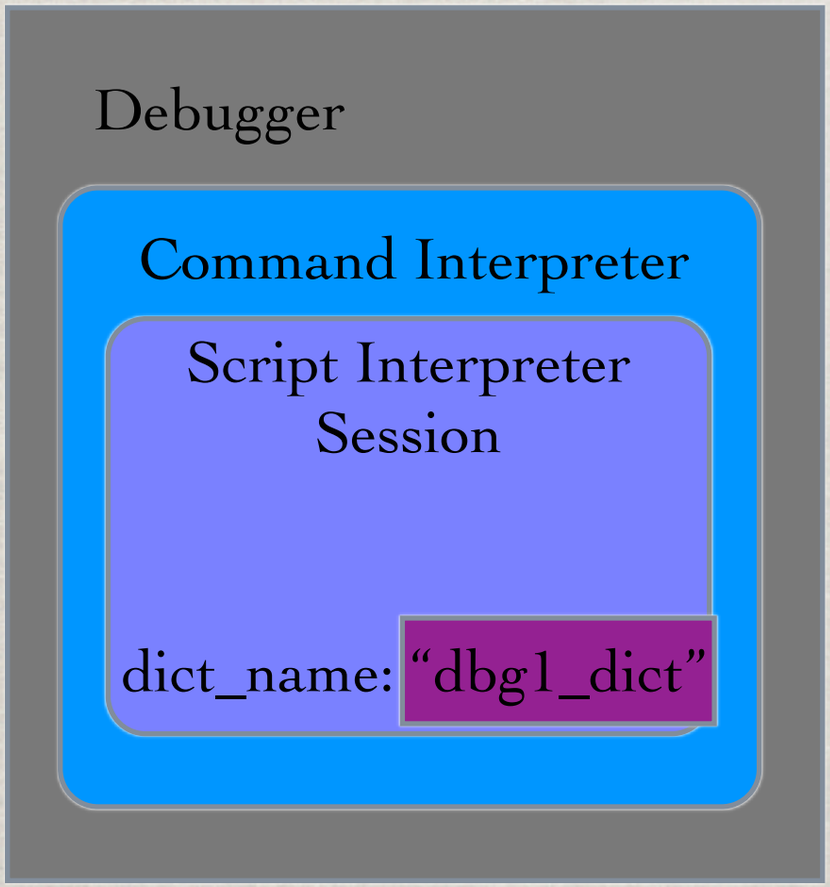
Invoking Interactive Script Interpreter

(lldb) script



Invoking Interactive Script Interpreter

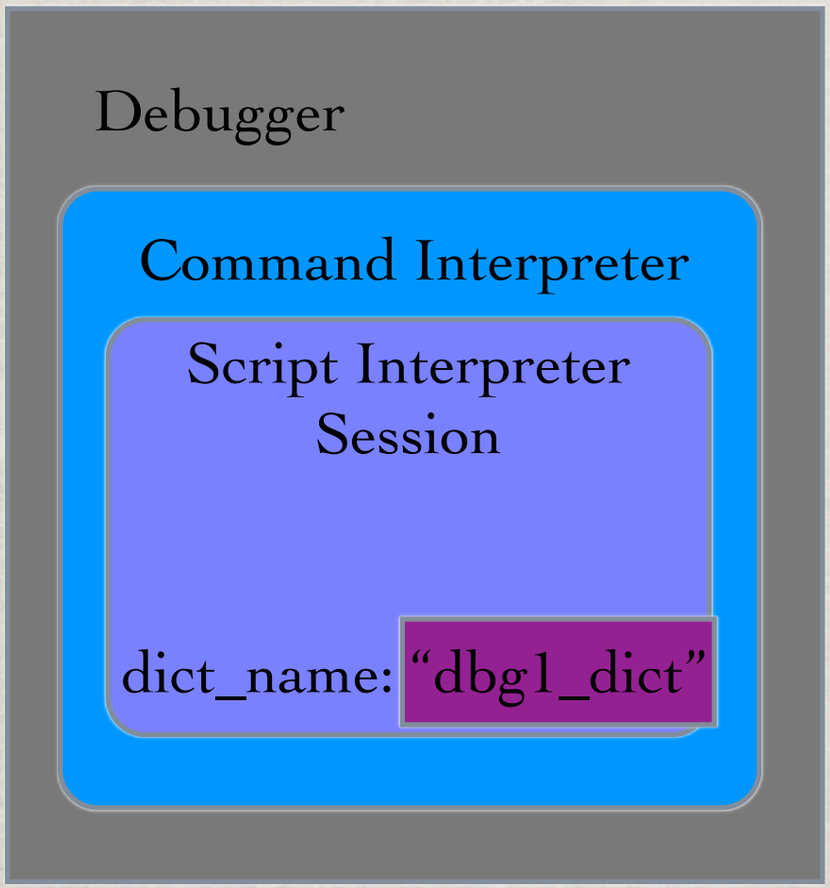
(lldb) script




```
sprintf (buffer,
         “python_run_interpreter (‘%s’)”,
          dict_name);
```

Invoking Interactive Script Interpreter

(lldb) script




```
sprintf (buffer,
         "python_run_interpreter ('%s')",
         dict_name);
```

buffer = "python_run_interpreter ('dbg1_dict')"

Invoking Interactive Script Interpreter

(lldb) script


```
sprintf (buffer,
         “python_run_interpreter (‘%s’)”,
          dict_name);
```

Invoking Interactive Script Interpreter

(lldb) script


```
sprintf (buffer,
         “python_run_interpreter (‘%s’)”,
          dict_name);
```

PyRun_SimpleString (buffer);

Invoking Interactive Script Interpreter

(lldb) script


```
sprintf (buffer,
         “python_run_interpreter (‘%s’)”,
          dict_name);
```

Set sys.stdin & sys.stdout to appropriate values
Set termios appropriately

PyRun_SimpleString (buffer);

Invoking Interactive Script Interpreter

(lldb) script


```
sprintf (buffer,
         “python_run_interpreter (‘%s’)”,
          dict_name);
```

Set sys.stdin & sys.stdout to appropriate values
Set termios appropriately

PyRun_SimpleString (buffer);

Restore sys.stdin & sys.stdout values
Restore termios

More About LLDB Breakpoint scripts...

(lldb) break command add -s python <bp_num>

More About LLDB Breakpoint scripts...

(lldb) break command add -s python <bp_num>
Enter your Python commands. Type 'DONE' to end.
>

More About LLDB Breakpoint scripts...

```
(lldb) break command add -s python <bp_num>
Enter your Python commands.  Type 'DONE' to end.
>   count = count + 1
>
```

More About LLDB Breakpoint scripts...

(lldb) break command add -s python <bp_num>
Enter your Python commands. Type 'DONE' to end.
> count = count + 1
> print "Hit this breakpoint " + repr (count) + " times!"
>

More About LLDB Breakpoint scripts...

```
(lldb) break command add -s python <bp_num>
Enter your Python commands.  Type 'DONE' to end.
>   count = count + 1
>   print "Hit this breakpoint " + repr (count) + " times!"
>   DONE
(lldb)
```

More About LLDB Breakpoint scripts...

```
count = count + 1
print “Hit this breakpoint ” + repr (count) + “ times!”
```

More About LLDB Breakpoint scripts...

```
def some_obscure_function_name (frame, bp_loc):
        count = count + 1
        print “Hit this breakpoint ” + repr (count) + “ times!”
```

More About LLDB Breakpoint scripts...

```
def some_obscure_function_name (frame, bp_loc):
    count = count + 1
    print “Hit this breakpoint ” + repr (count) + “ times!”
```

LLDB Convenience Variables

More About LLDB Breakpoint scripts...

```
def some_obscure_function_name (frame, bp_loc):
    count = count + 1
    print “Hit this breakpoint ” + repr (count) + “ times!”
```

LLDB Convenience Variables

```
some_obscure_function_name (cur_frame, cur_bp_loc)
```

More About LLDB Breakpoint scripts...

```
def some_obscure_function_name (frame, bp_loc):
    count = count + 1
    print “Hit this breakpoint “ + repr (count) + “ times!”
```

More About LLDB Breakpoint scripts...

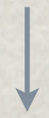
```
def some_obscure_function_name (frame, bp_loc):

    count = count + 1
    print “Hit this breakpoint “ + repr (count) + “ times!”
```

More About LLDB Breakpoint scripts...

```
def some_obscure_function_name (frame, bp_loc):
        global count
        count = count + 1
        print “Hit this breakpoint “ + repr (count) + “ times!”
```

Breakpoint Script Commands - Create

(lldb) break command add -s python <bp_num>



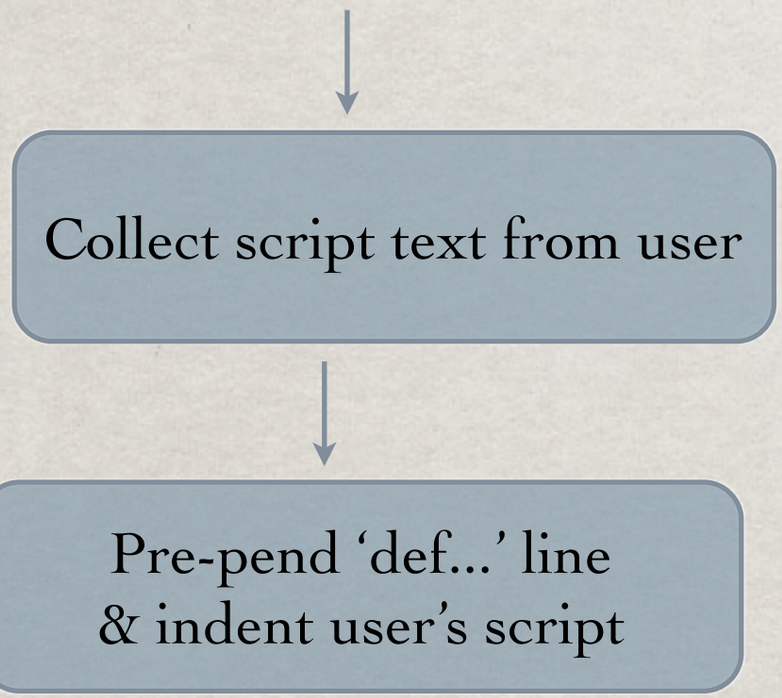
Breakpoint Script Commands - Create

(lldb) break command add -s python <bp_num>

Collect script text from user

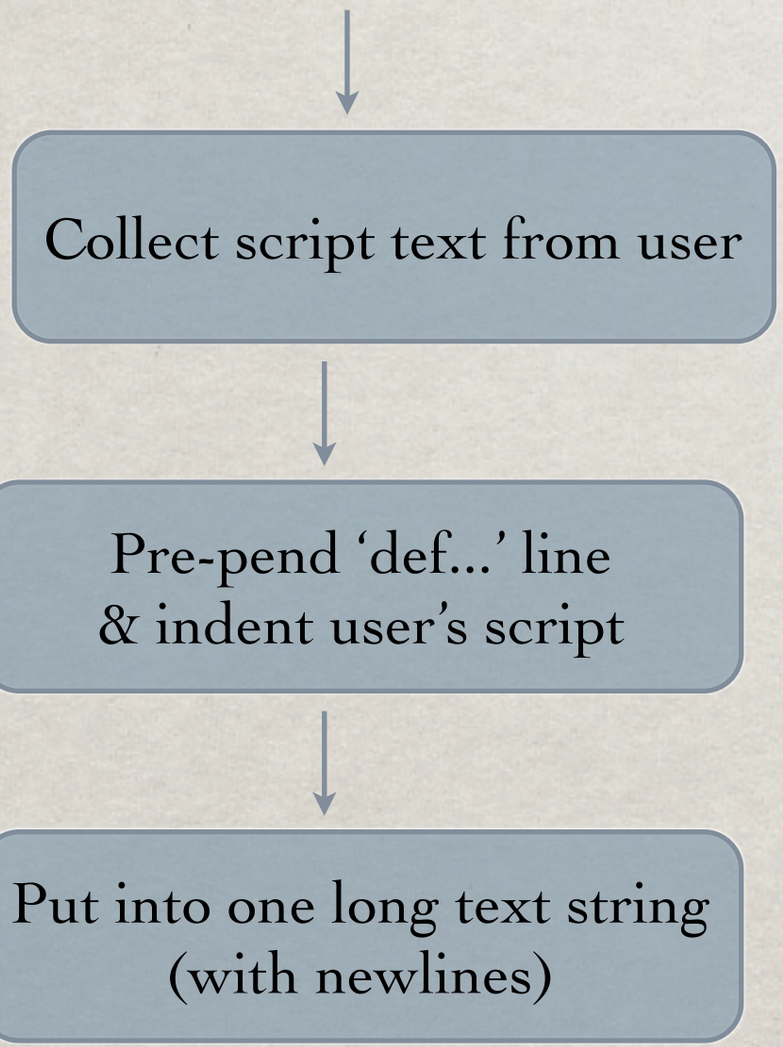
Breakpoint Script Commands - Create

(lldb) break command add -s python <bp_num>



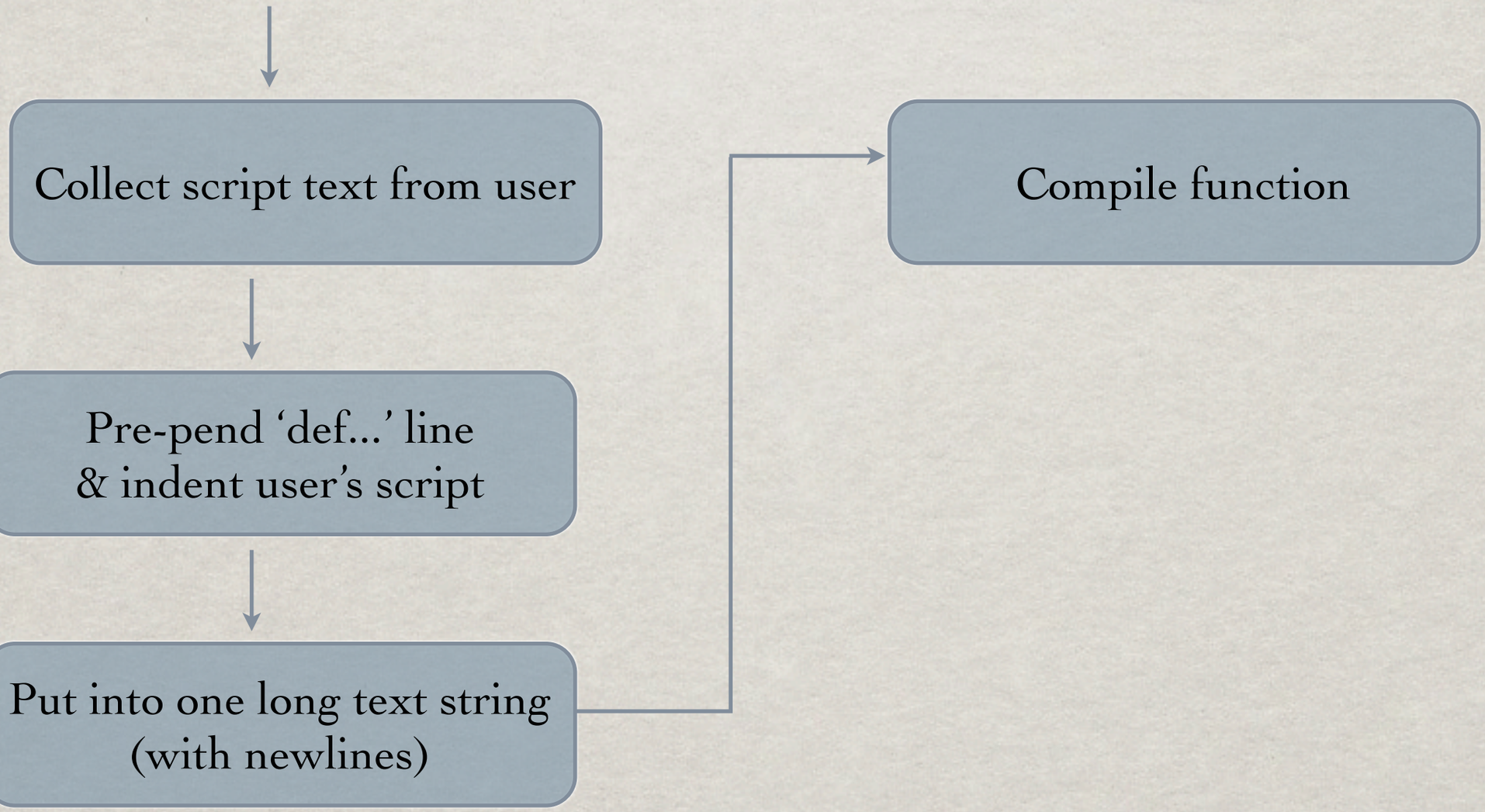
Breakpoint Script Commands - Create

(lldb) break command add -s python <bp_num>



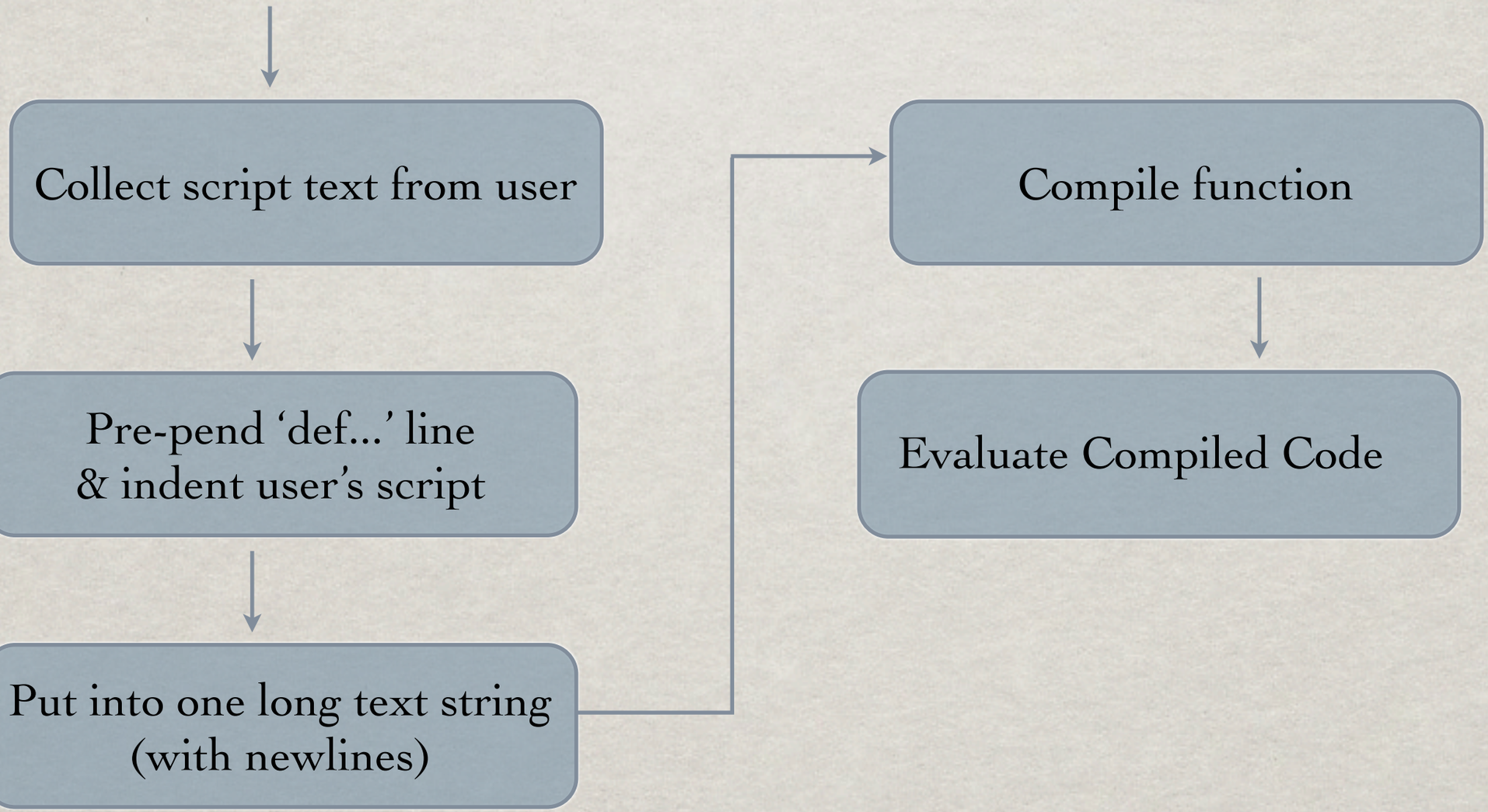
Breakpoint Script Commands - Create

(lldb) break command add -s python <bp_num>



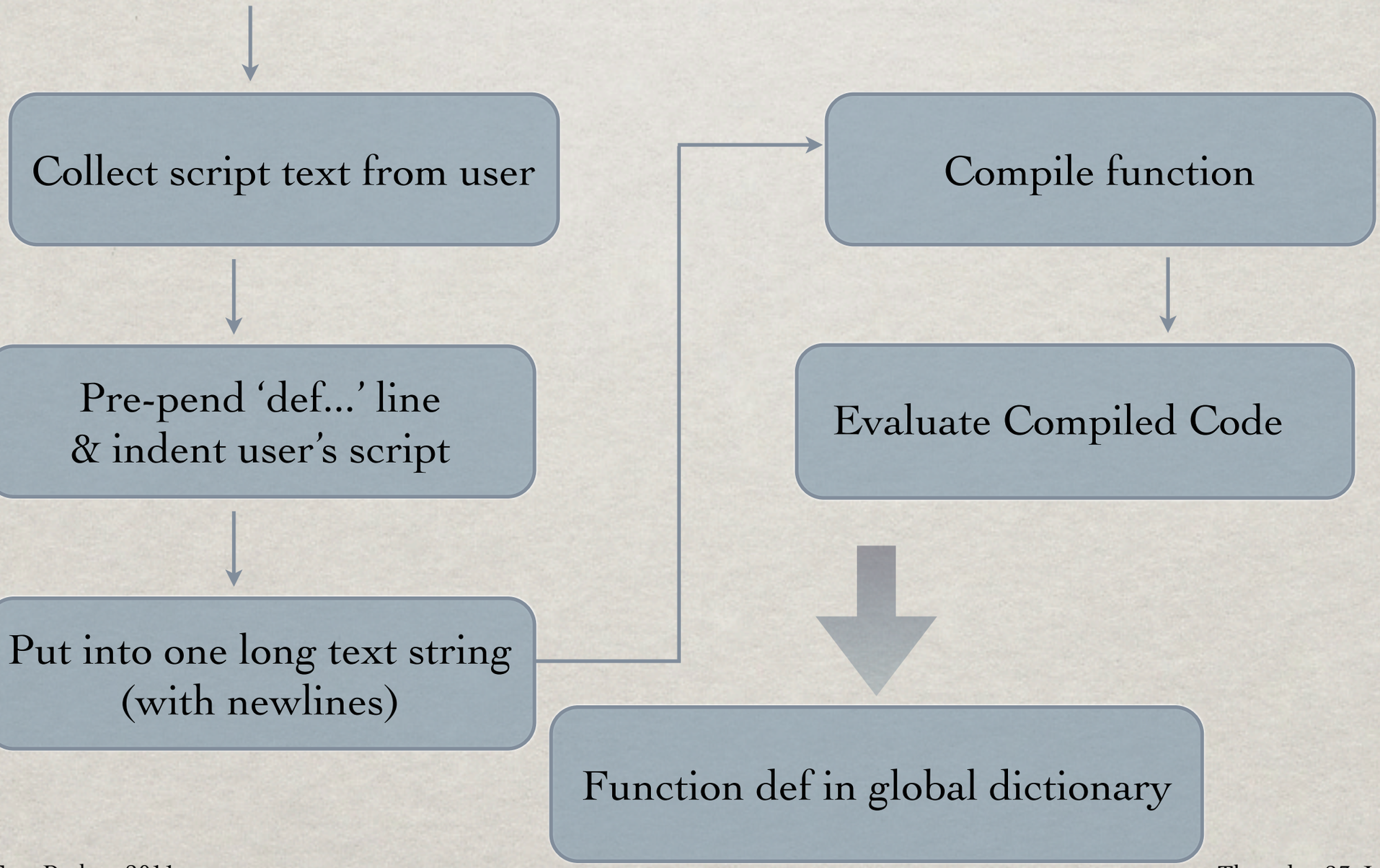
Breakpoint Script Commands - Create

(lldb) break command add -s python <bp_num>



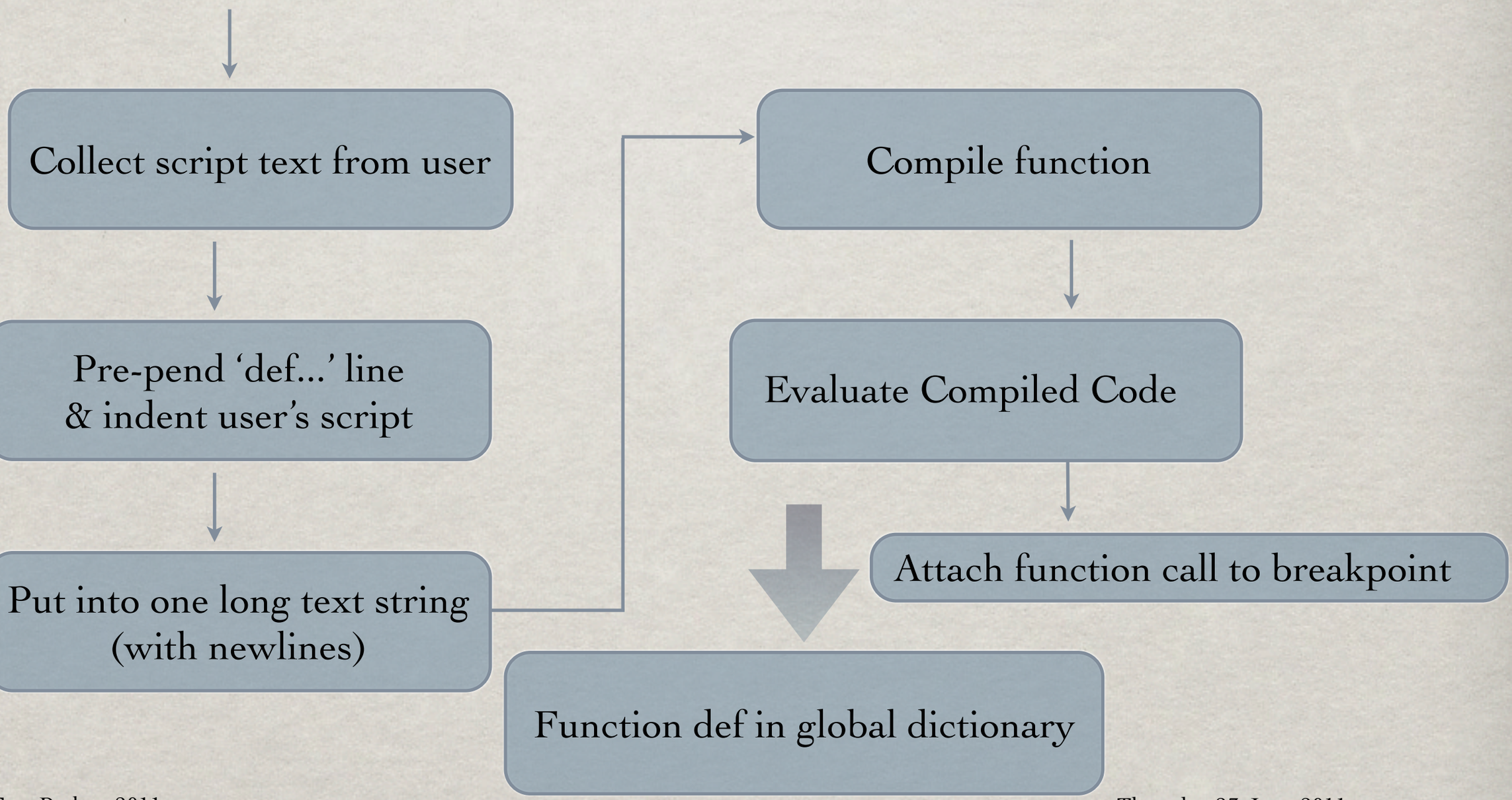
Breakpoint Script Commands - Create

(lldb) break command add -s python <bp_num>



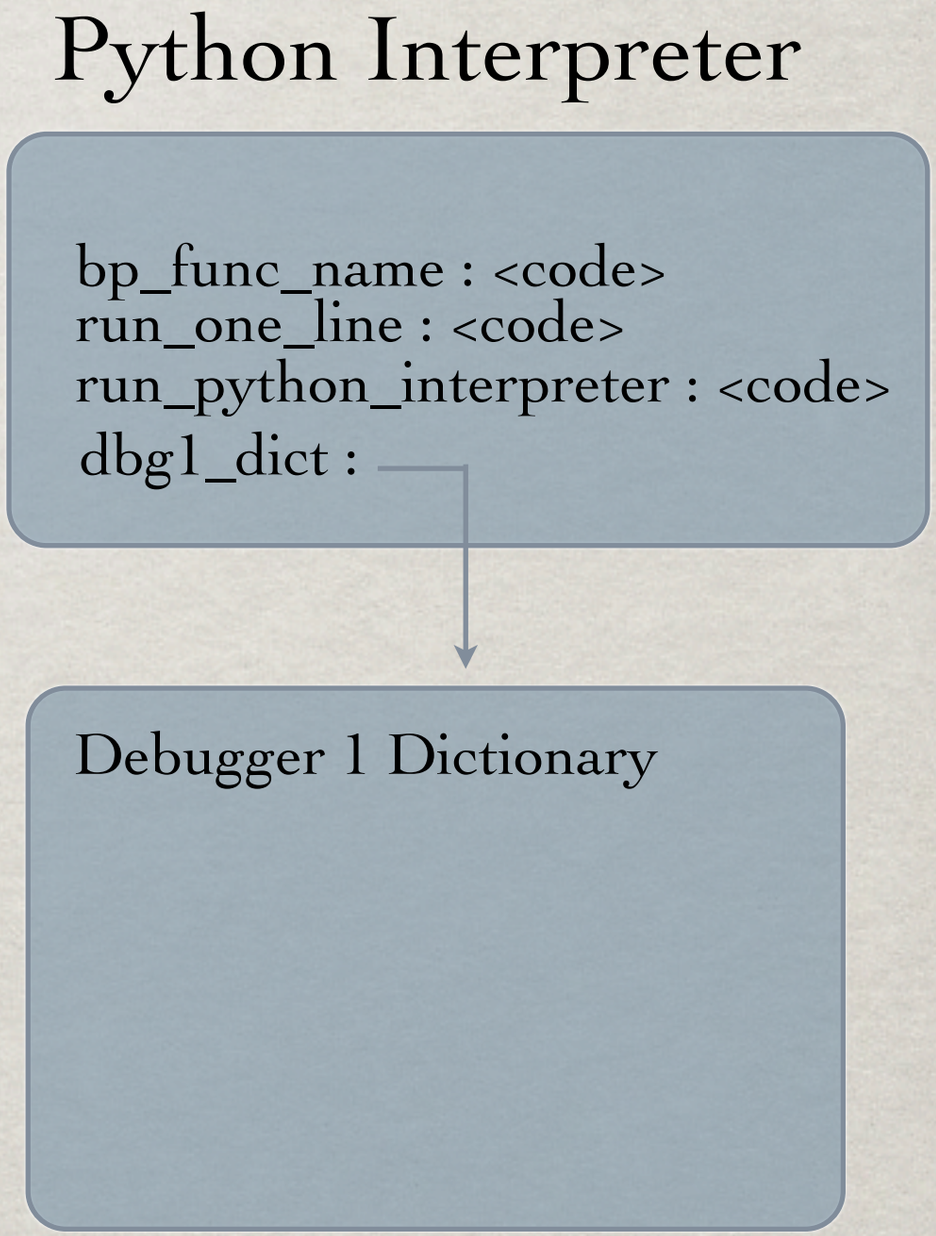
Breakpoint Script Commands - Create

(lldb) break command add -s python <bp_num>



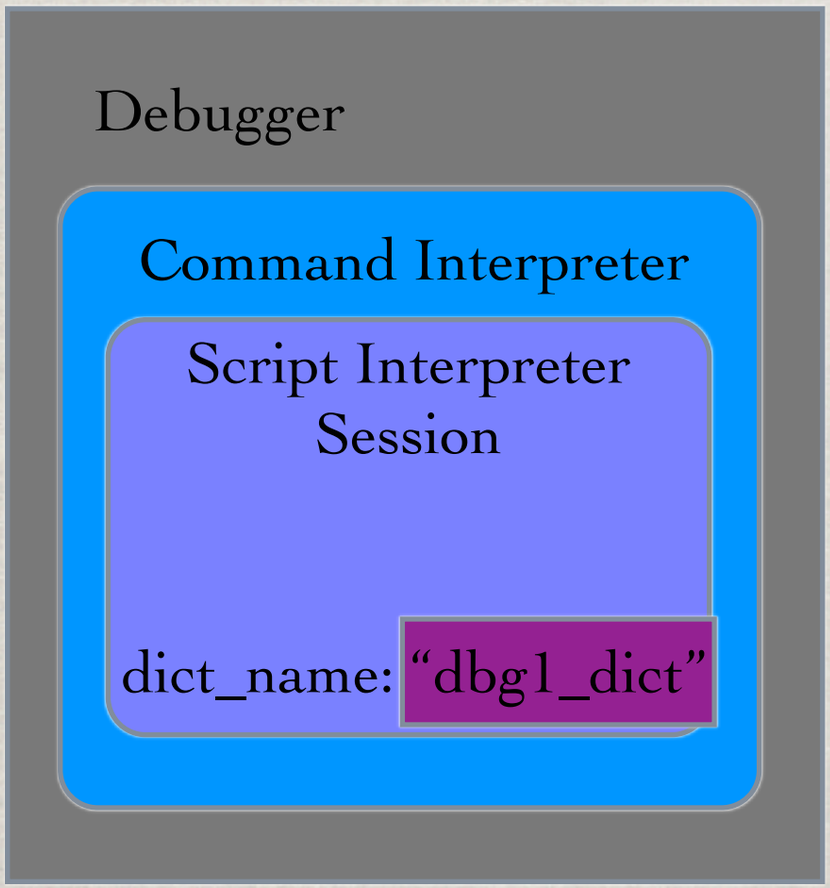
Breakpoint Script Commands - Call

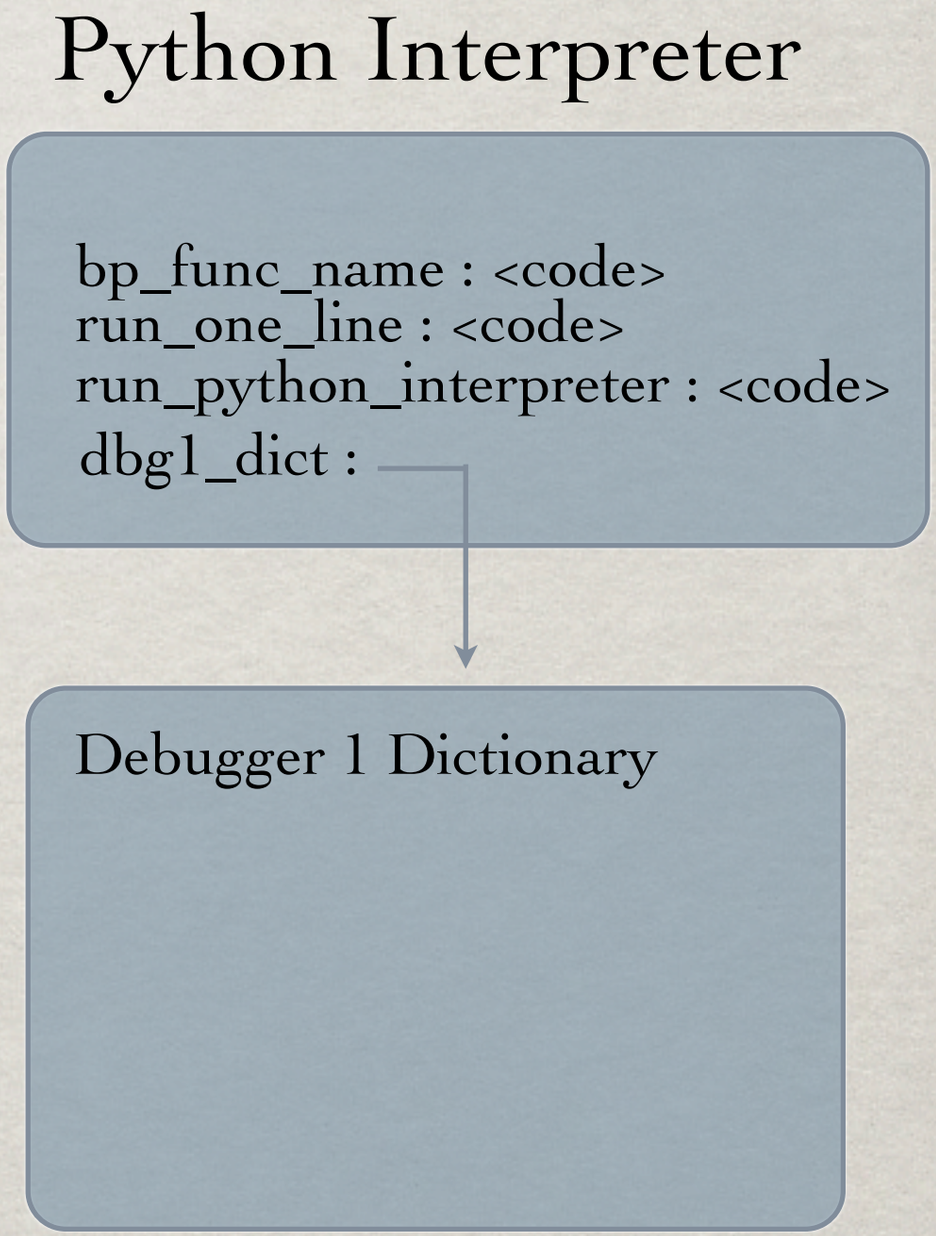
(lldb) run
Process stopped. Hit breakpoint.



Breakpoint Script Commands - Call

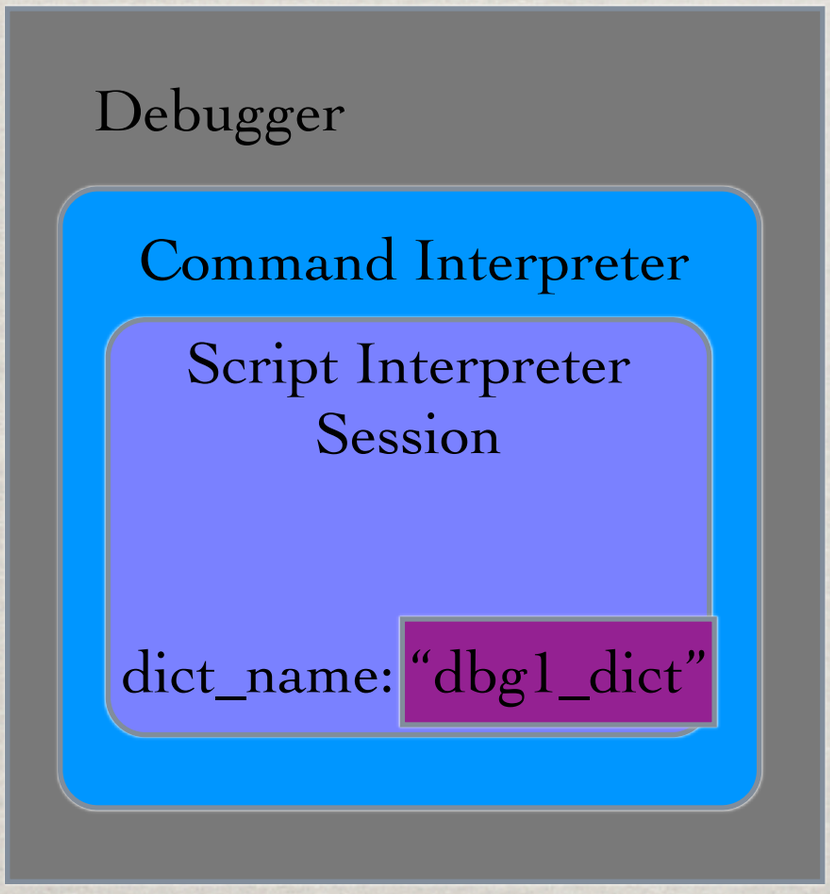
(lldb) run
Process stopped. Hit breakpoint.

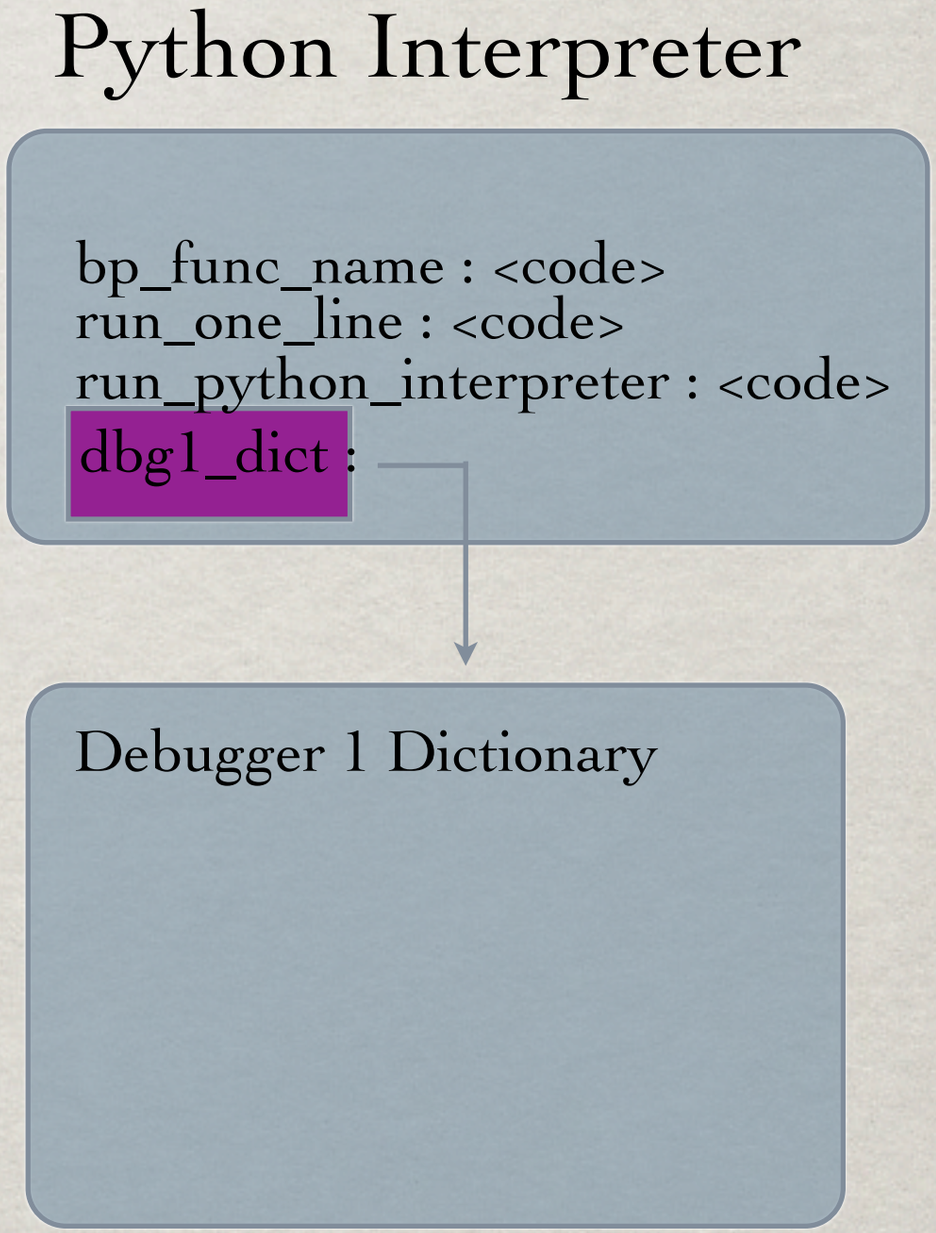




Breakpoint Script Commands - Call

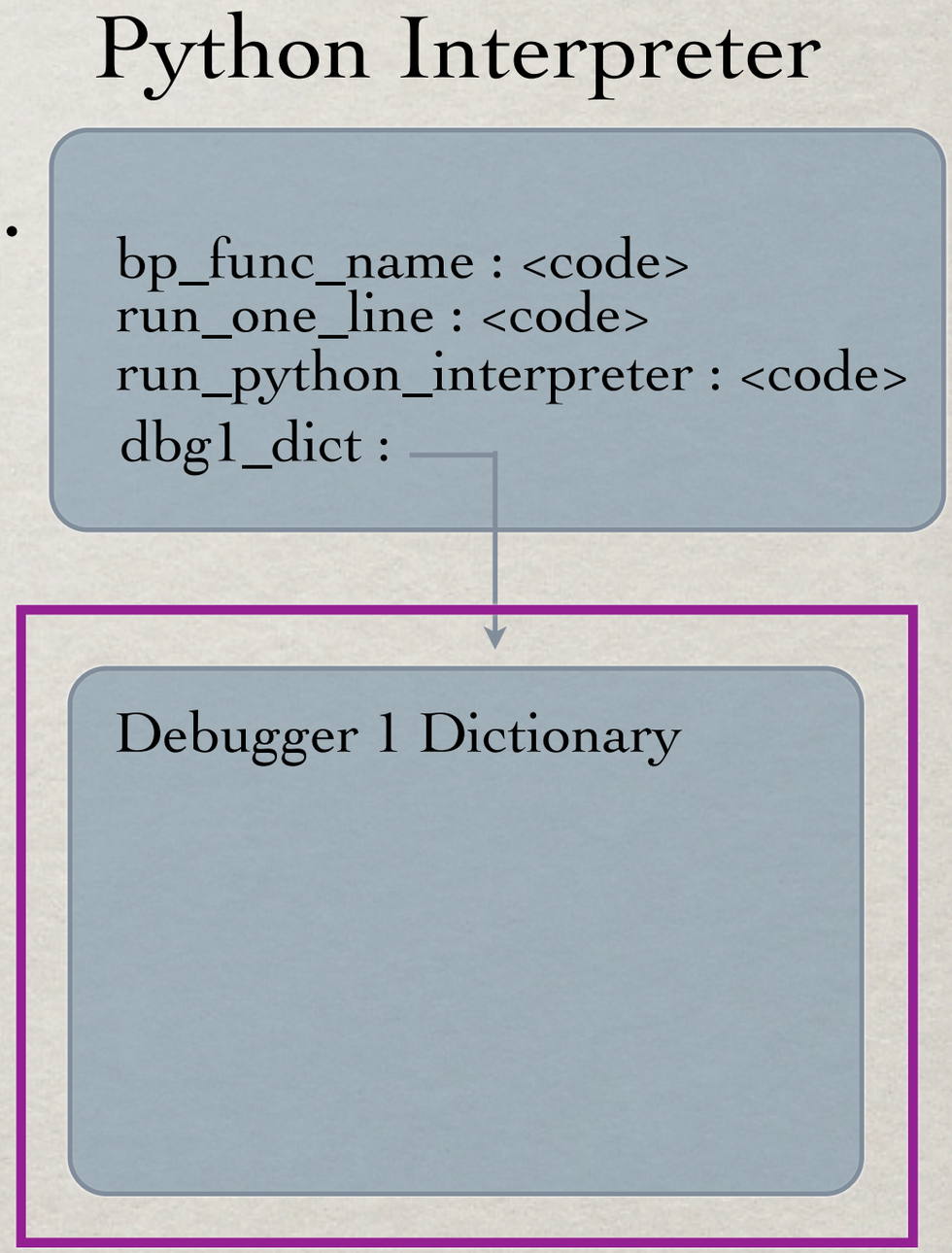
(lldb) run
Process stopped. Hit breakpoint.





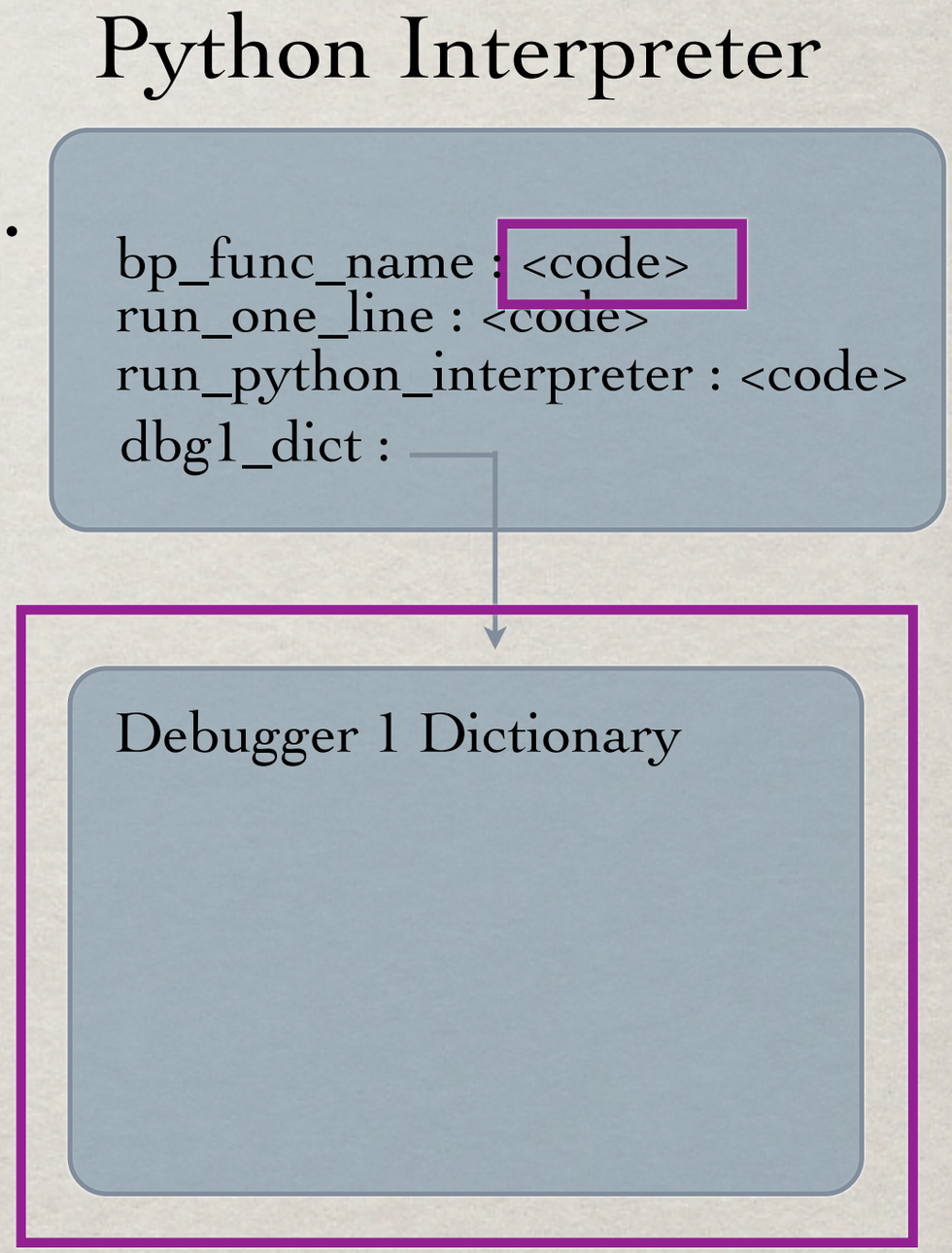
Breakpoint Script Commands - Call

(lldb) run
Process stopped. Hit breakpoint.



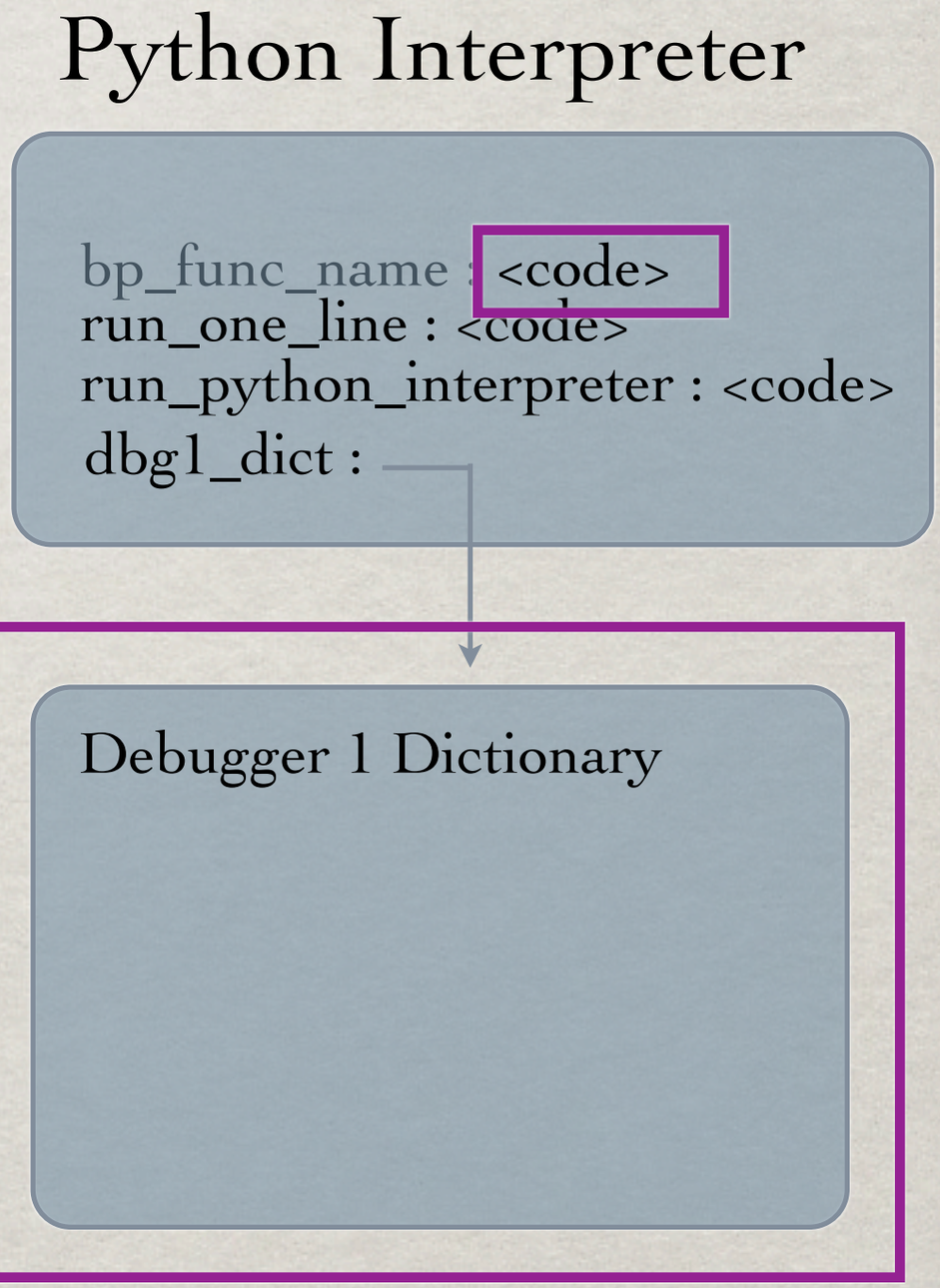
Breakpoint Script Commands - Call

(lldb) run
Process stopped. Hit breakpoint.



Breakpoint Script Commands - Call

(lldb) run
Process stopped. Hit breakpoint.



Breakpoint Script Commands - Call

(lldb) run
Process stopped. Hit breakpoint.

bp_func_name : <code>

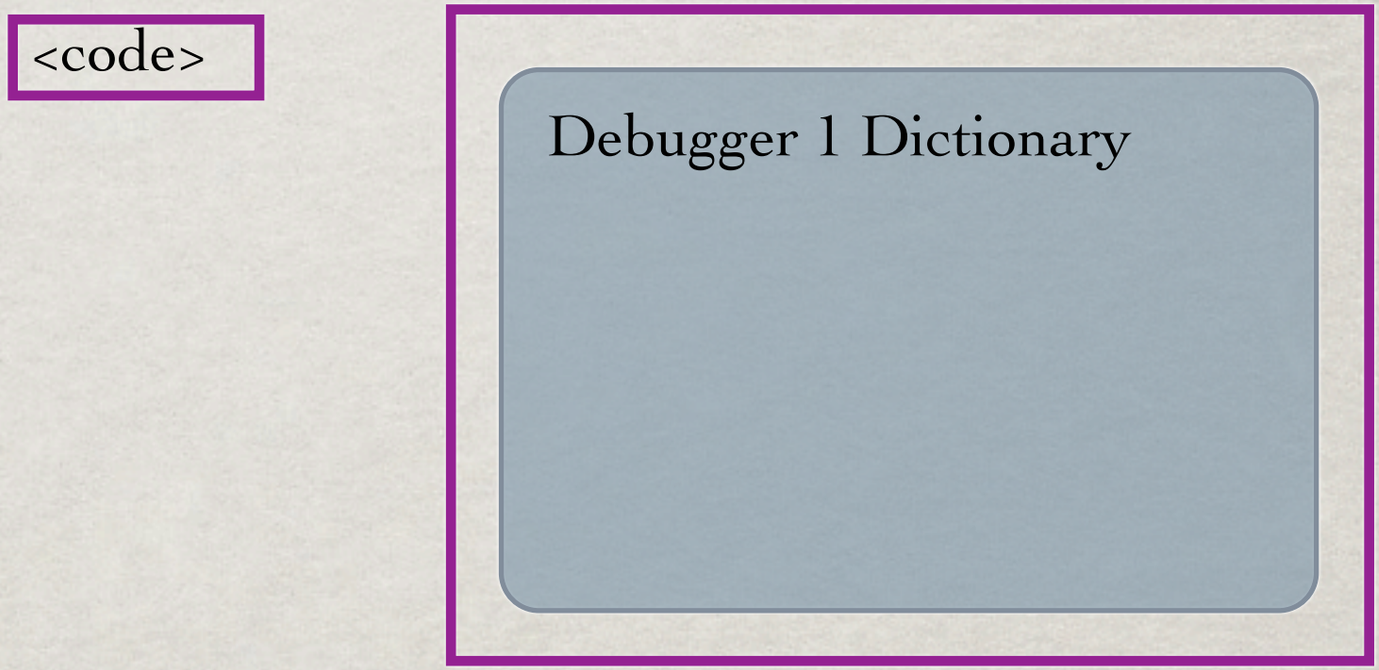
Debugger 1 Dictionary

Breakpoint Script Commands - Call

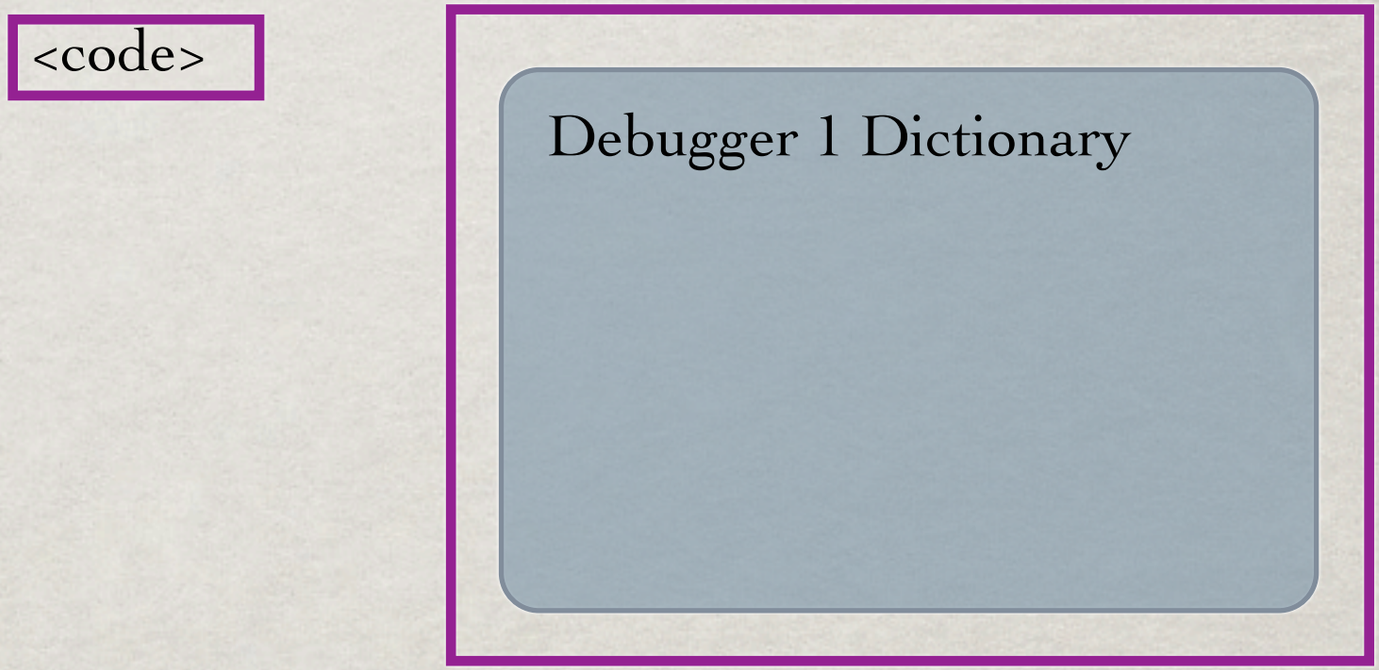
<code>

Debugger 1 Dictionary

Breakpoint Script Commands - Call



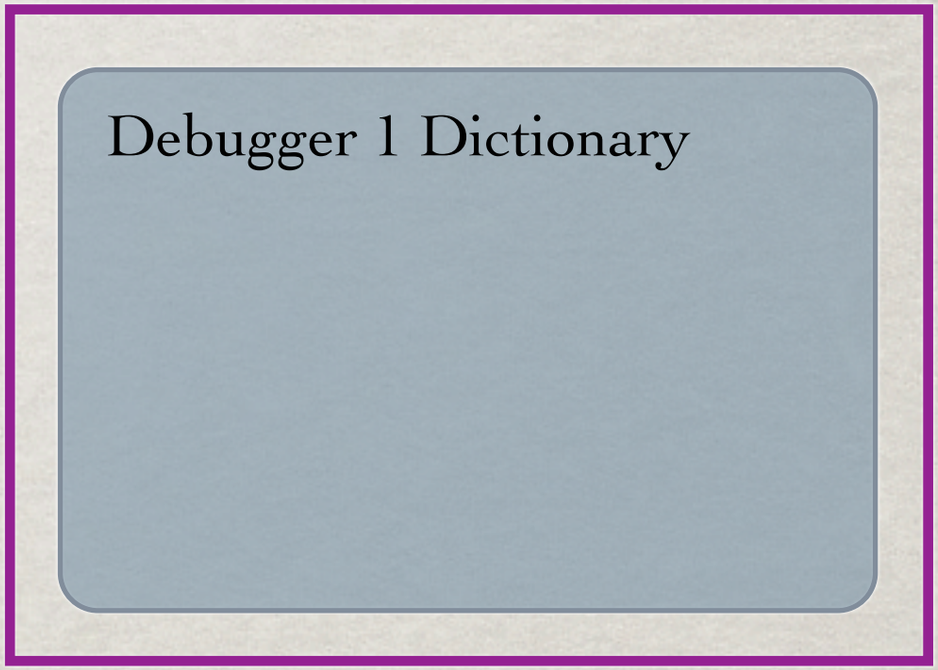
Breakpoint Script Commands - Call



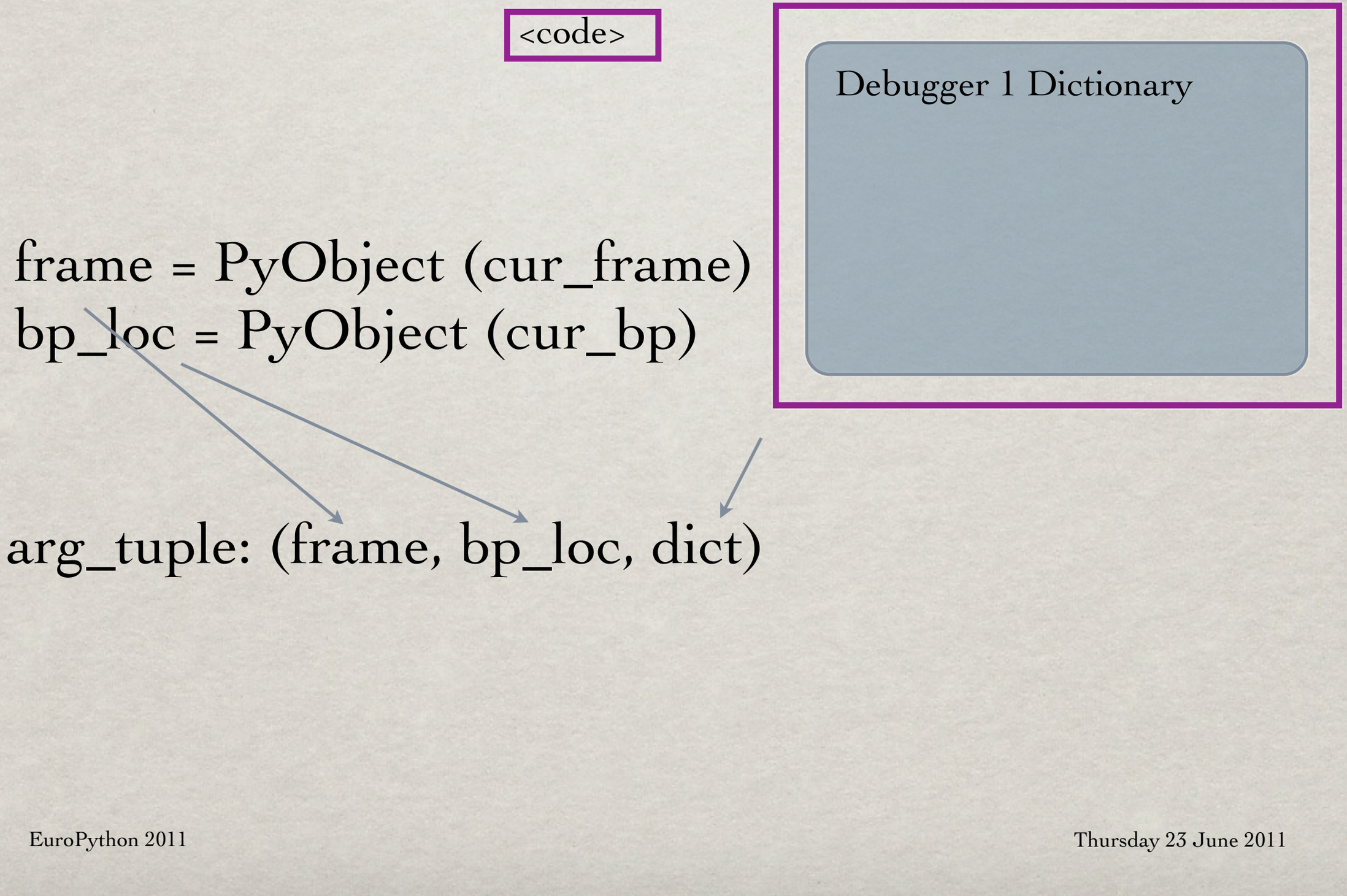
Breakpoint Script Commands - Call

<code>

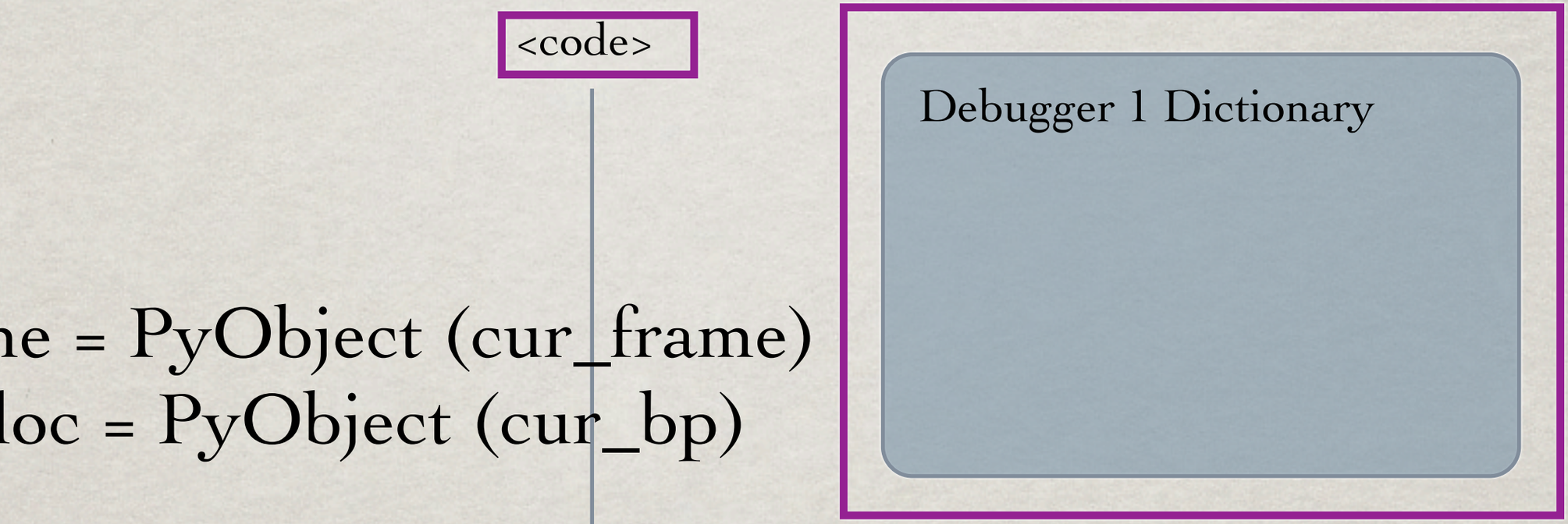
frame = PyObject (cur_frame)
bp_loc = PyObject (cur_bp)



Breakpoint Script Commands - Call



Breakpoint Script Commands - Call



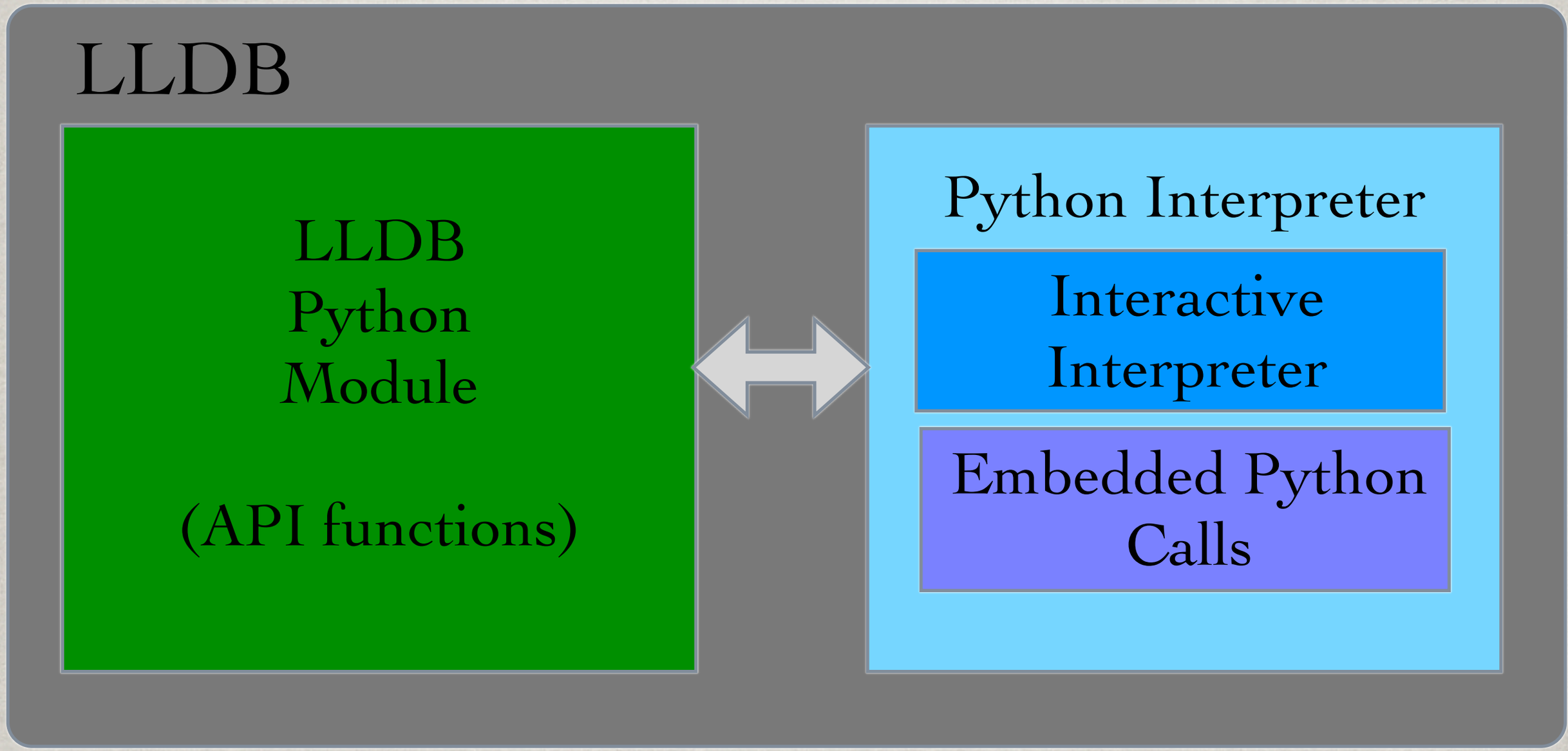
frame = PyObject (cur_frame)

bp_loc = PyObject (cur_bp)

arg_tuple: (frame, bp_loc, dict)

PyObject_CallObject (<code>, arg_tuple)

Python In LLDB - More Detail



Python In LLDB - More Detail

LLDB
Python
Module

(API functions)

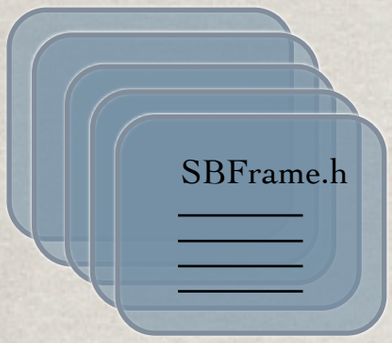
About SWIG...

- SWIG = "Simplified Wrapper and Interface Generator"

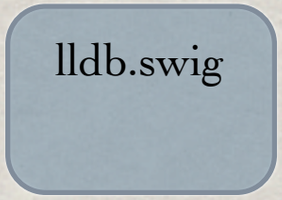
- Tool that parses C & C++ interfaces

- Generates "glue code" allowing Python to call the C/C++ code

Building The LLDB Python API Module

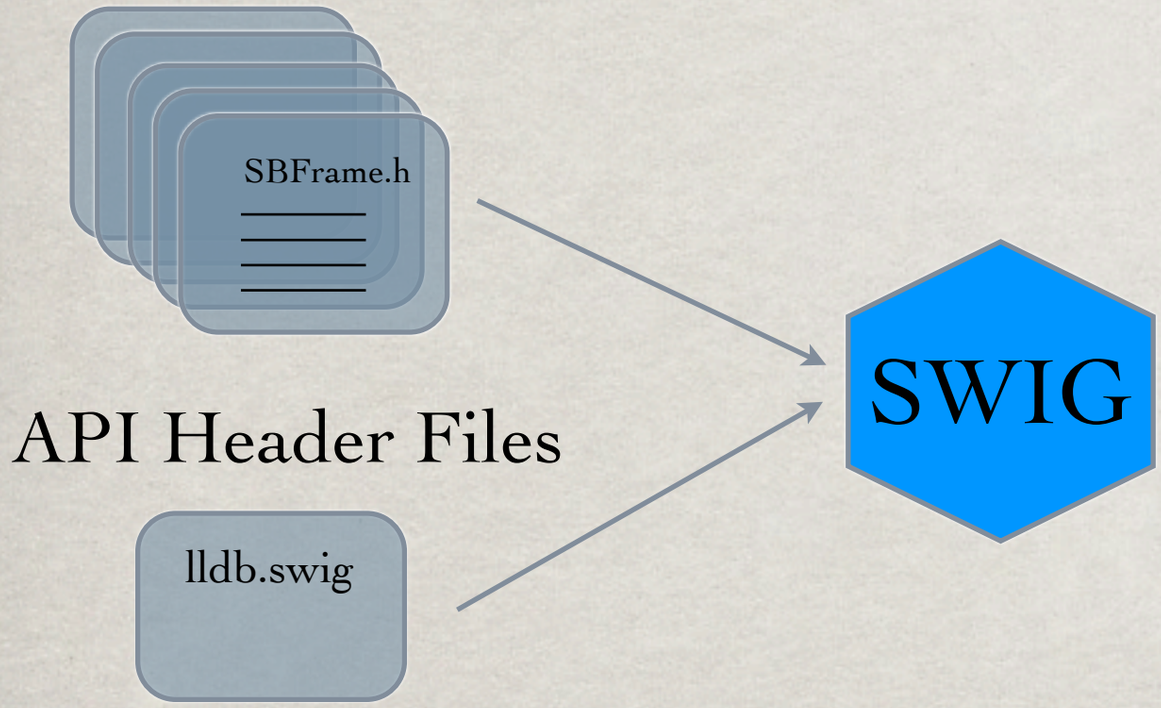


API Header Files

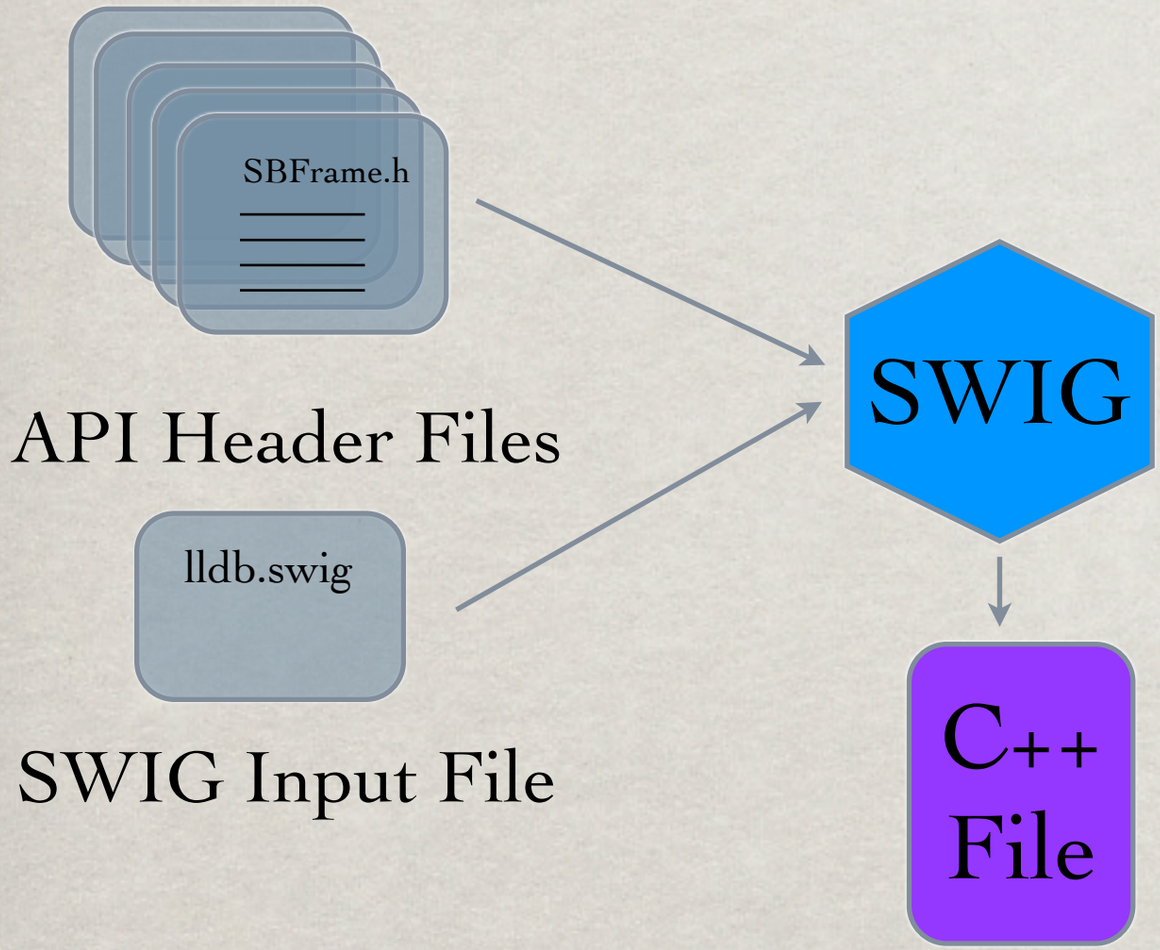


SWIG Input File

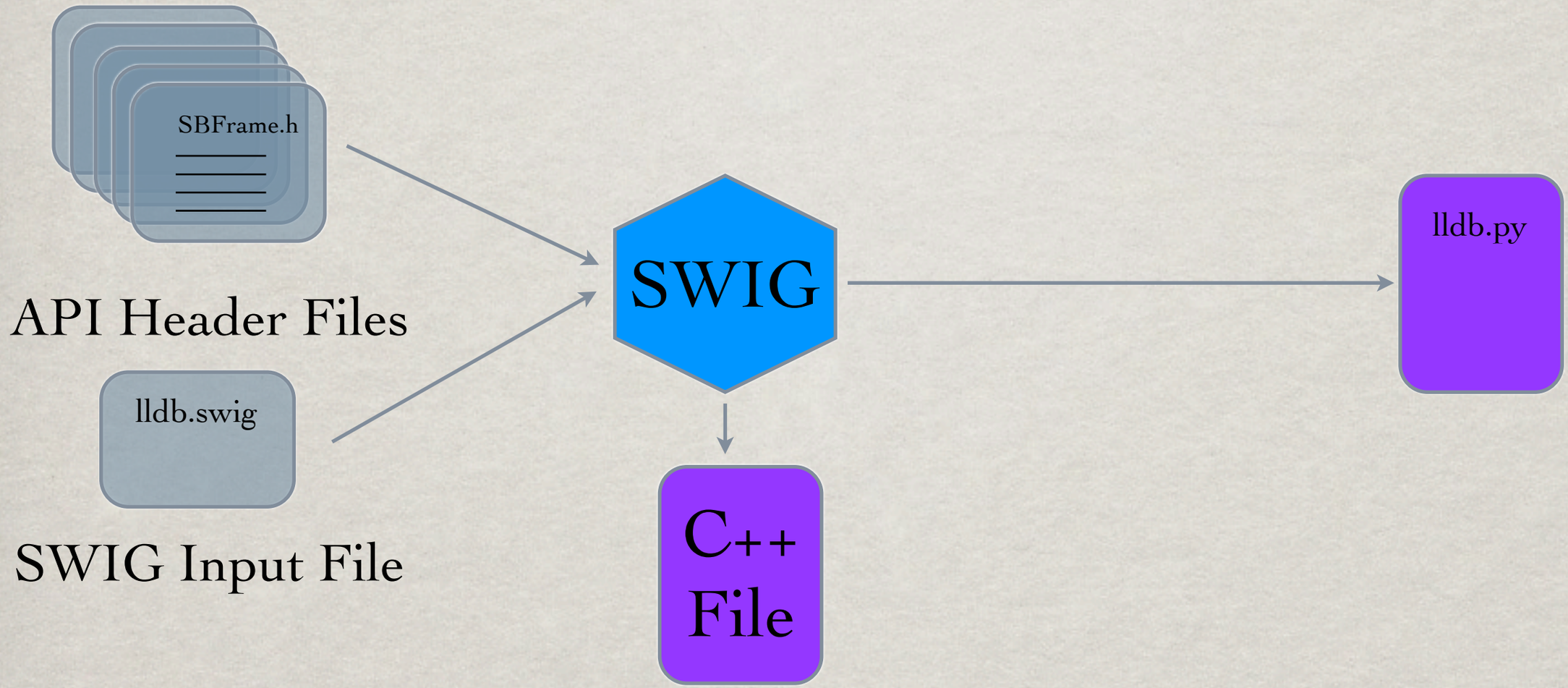
Building The LLDB Python API Module



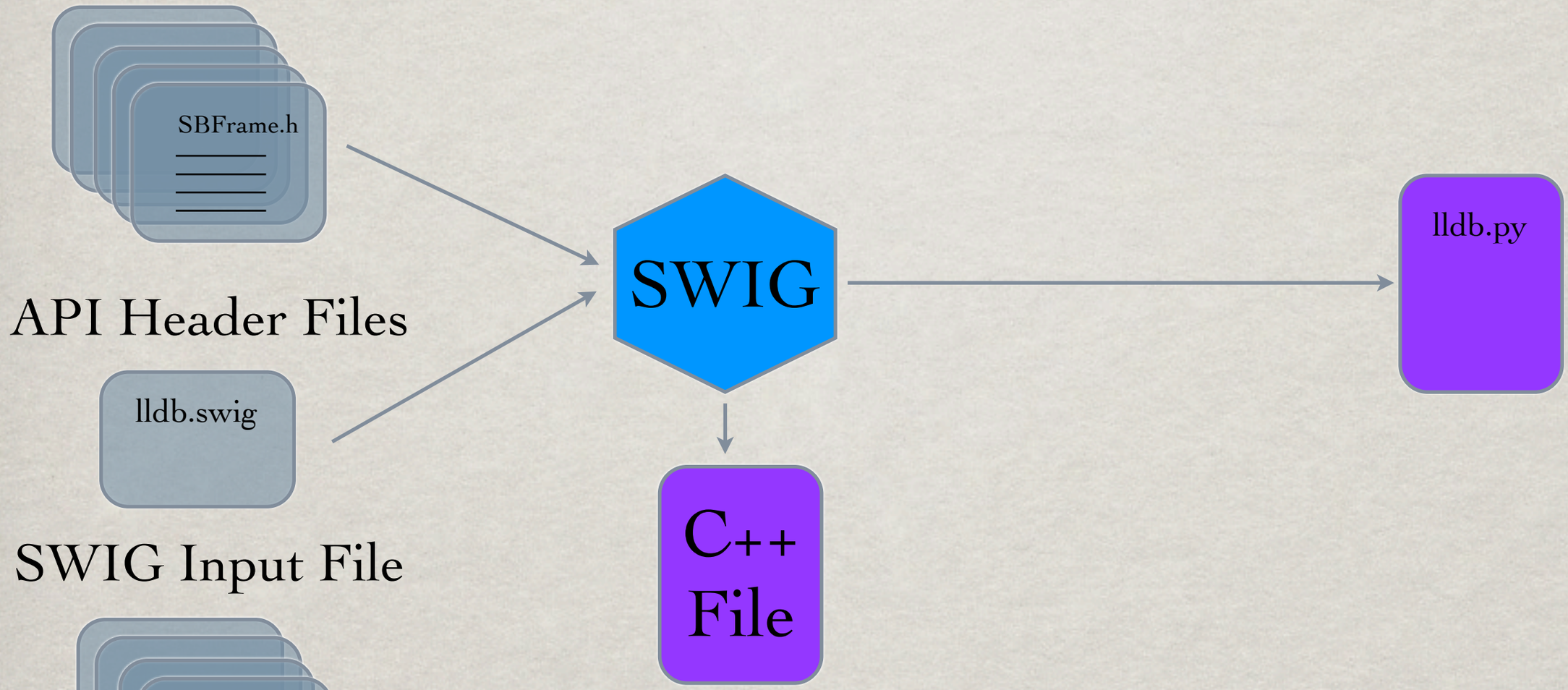
Building The LLDB Python API Module



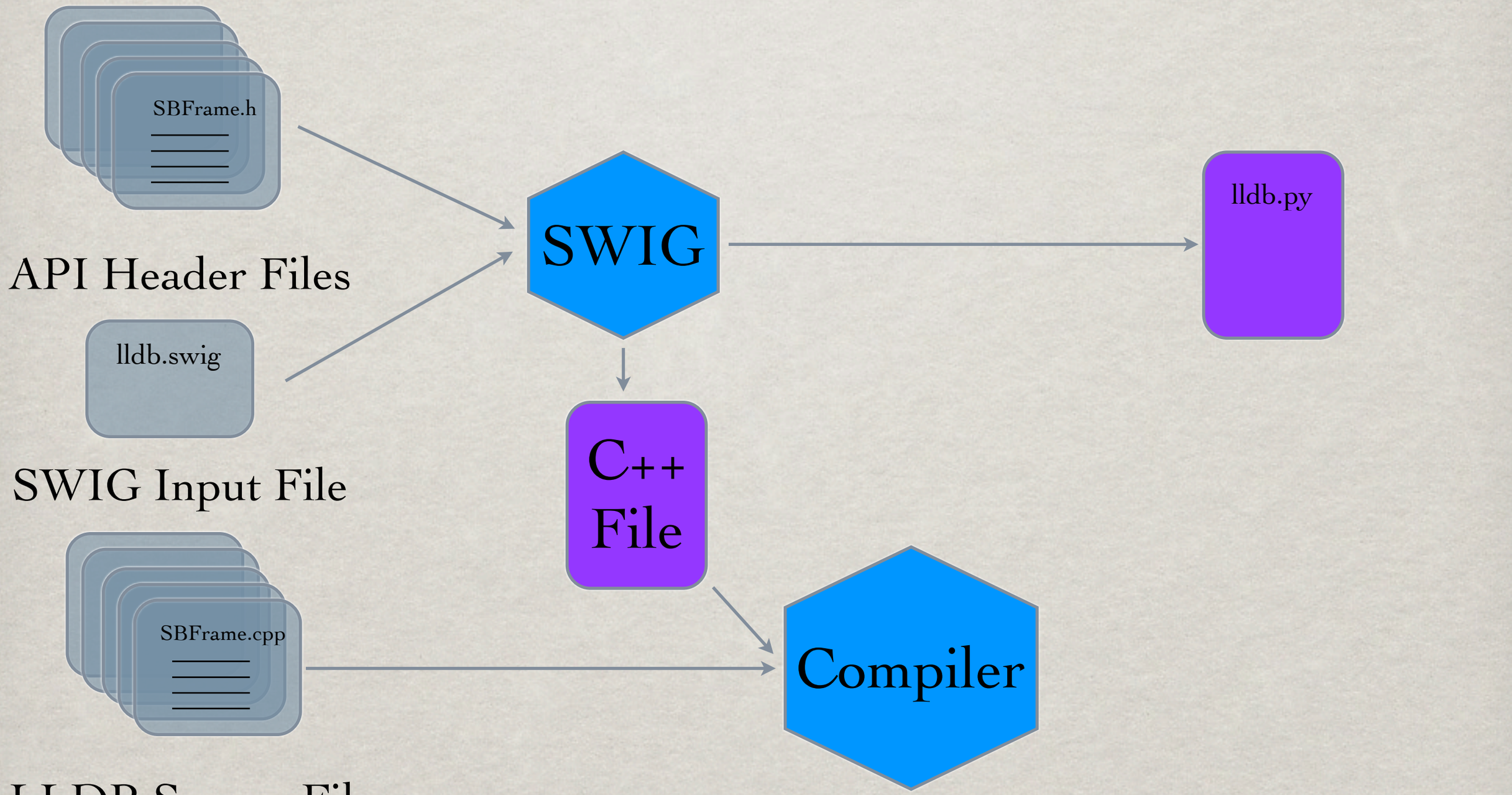
Building The LLDB Python API Module



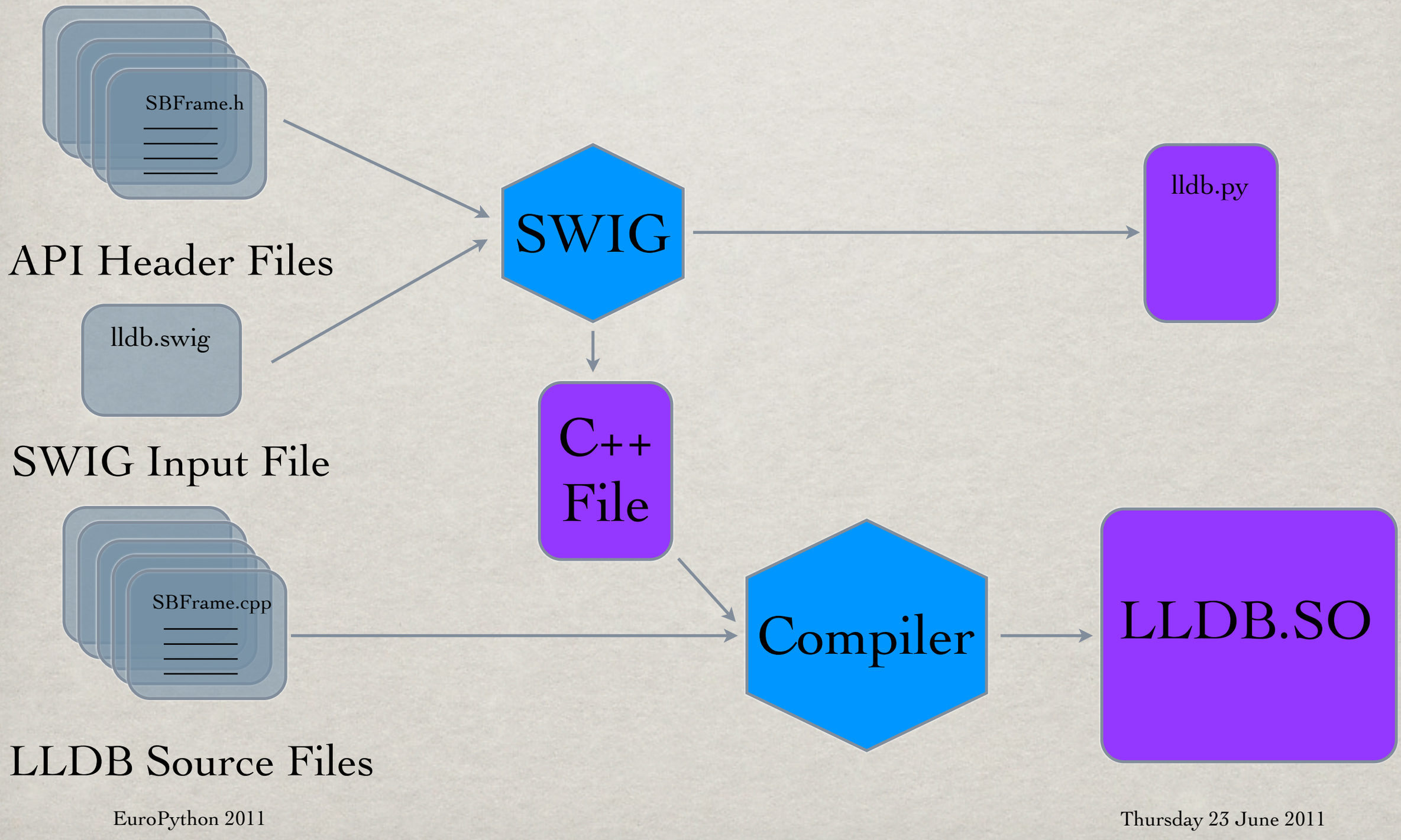
Building The LLDB Python API Module



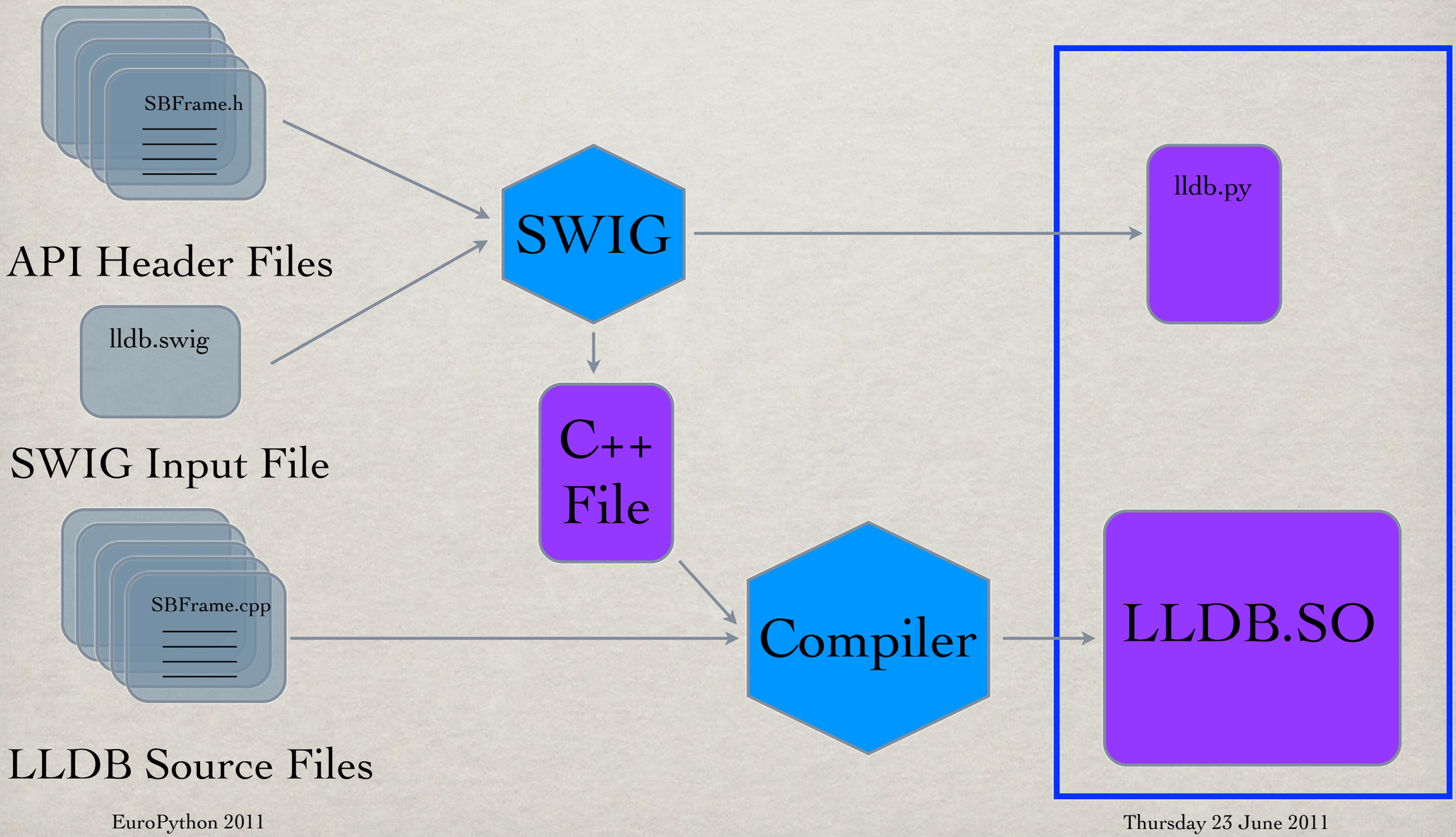
Building The LLDB Python API Module



Building The LLDB Python API Module



Building The LLDB Python API Module



Outline

- What is LLDB?
- Python in LLDB
- Particular Problems (& Solutions)
- Example Using Python to Debug Problem
- Questions

Particular Problems & Solutions

- Passing Pointers & C++ Objects to Python
- Single Dictionary Across Debugger Session
- Multiple Debuggers/Single Interpreter

Particular Problems & Solutions

- Passing Pointers & C++ Objects to Python
- Single Dictionary Across Debugger Session
- Multiple Debuggers/Single Interpreter

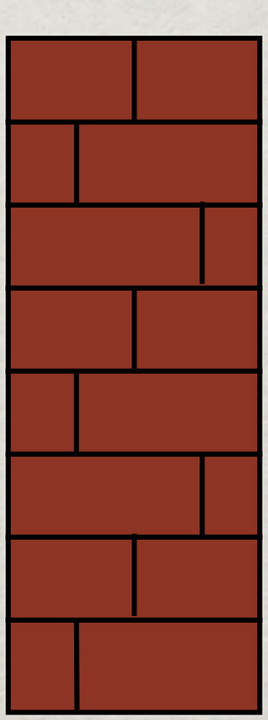
Passing Pointers & C++ Objects TO Python

- API functions operate on debugger objects
  - targets, processes, threads, frames, etc.
- LLDB stores them as objects or pointers
- Embedded Python only passes **scalar** data types back and forth (mostly)
- So...how to get LLDB objects "over" to Python?

How To Call GetNumThreads()?

(lldb) script
>>> ???

Interactive Python Interpreter



(Process *) cur_process

Process Object

Process methods:

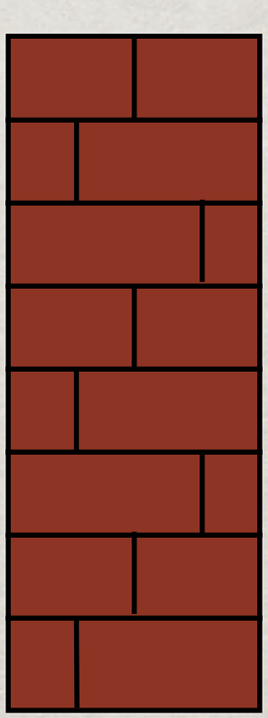
GetNumThreads()	0x1001f25ac
...	...
...	...

lldb debugger (C++ code)

How To Call GetNumThreads()?

(lldb) script
>>>

Interactive Python Interpreter



(Process *) cur_process

Process Object

Process methods:

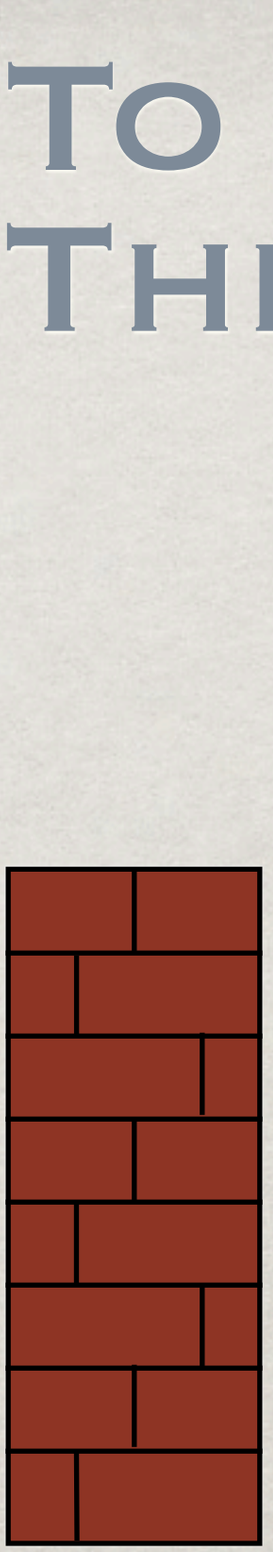
GetNumThreads()	0x1001f25ac
...	...
...	...

lldb debugger (C++ code)

How To Call GetNumThreads()?

(lldb) script
>>> GetNumThreads()
Traceback (most recent call last):
File "<console>", line 1, in <module>
NameError: name '...' is not defined

Interactive Python Interpreter



(Process *) cur_process

Process Object

Process methods:

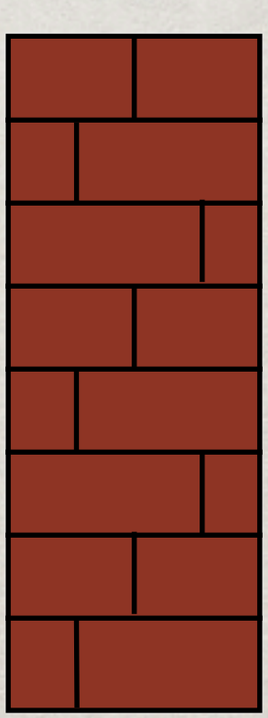
GetNumThreads()	0x1001f25ac
...	...
...	...

lldb debugger (C++ code)

How To Call GetNumThreads()?

(lldb) script
>>>

Interactive Python Interpreter



(Process *) cur_process

Process Object

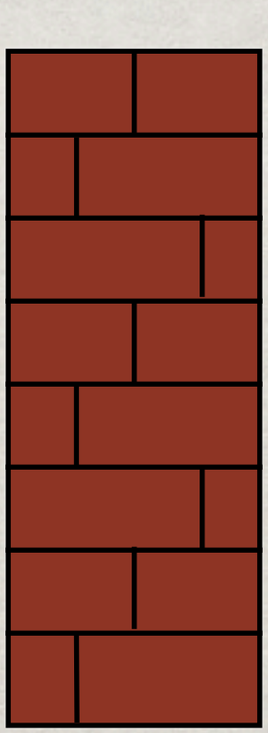
Process methods:

GetNumThreads()	0x1001f25ac
...	...
...	...

lldb debugger (C++ code)

How To Call GetNumThreads()?

(lldb) script
>>>



(Process *) cur_process

Process Object

Process methods:

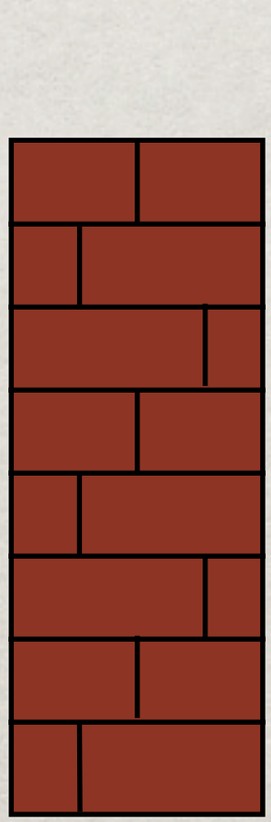
GetNumThreads()	0x1001f25ac
…	…
…	…

Interactive Python Interpreter

lldb debugger (C++ code)

How To Call GetNumThreads()?

(lldb) script
>>>lldb.SBProcess.GetNumThreads()



(Process *) cur_process

Process Object

Process methods:

GetNumThreads()	0x1001f25ac
...	...
...	...

Interactive Python Interpreter

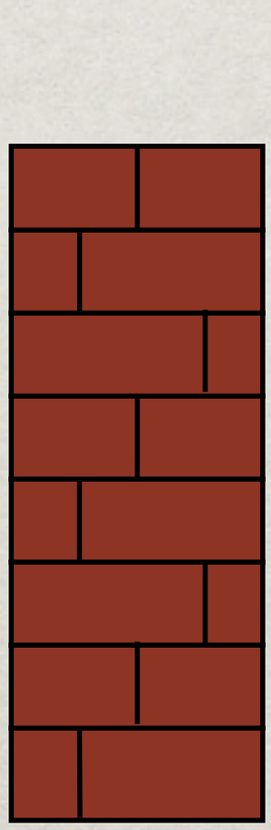
lldb debugger (C++ code)

How To Call GetNumThreads()?

(lldb) script
>>>lldb.SBProcess.GetNumThreads()

Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unbound method
GetNumThreads() must be called with
SBProcess instance as first argument
(got nothing instead)

Interactive Python Interpreter



(Process *) cur_process

Process Object

Process methods:

GetNumThreads()	0x1001f25ac
...	...
...	...

lldb debugger (C++ code)

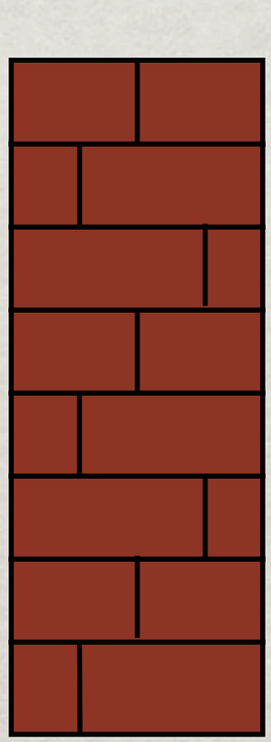
How To Call GetNumThreads()?

(lldb) script
>>>lldb.SBProcess.GetNumThreads()

Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unbound method

GetNumThreads() must be called with SBProcess instance as first argument (got nothing instead)

Interactive Python Interpreter



(Process *) cur_process

Process Object

Process methods:

GetNumThreads()	0x1001f25ac
...	...
...	...

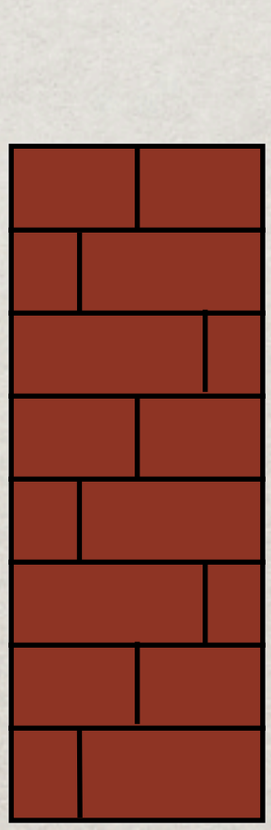
lldb debugger (C++ code)

How To Call GetNumThreads()?

(lldb) script
>>>lldb.SBProcess.GetNumThreads()

Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unbound method
GetNumThreads() must be called with
SBProcess instance as first argument
(got nothing instead)

Interactive Python Interpreter



(Process *) cur_process

Process Object

Process methods:

GetNumThreads()	0x1001f25ac
...	...
...	...

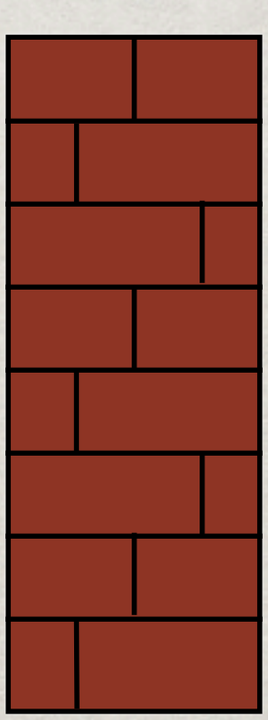
lldb debugger (C++ code)

How To Call GetNumThreads()?

(lldb) script

```
>>> process = ????
>>> process.GetNumThreads()
```

Interactive Python Interpreter



(Process *) cur_process

Process Object

Process methods:

GetNumThreads()	0x1001f25ac
...	...
...	...

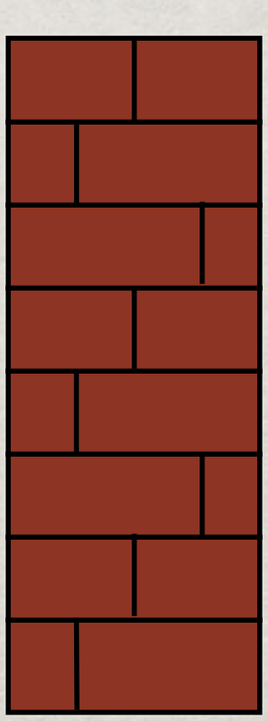
lldb debugger (C++ code)

How To Get Process Object Into Python Variable?

(lldb) script

```
>>> process = ????
>>> process.GetNumThreads()
```

Interactive Python Interpreter



(Process *) cur_process

Process Object

Process methods:

GetNumThreads()	0x1001f25ac
...	...
...	...

lldb debugger (C++ code)

Python-Provided conversions...

- PyArg_ParseTuple/Py_BuildValue convert:

Python-Provided conversions...

- PyArg_ParseTuple/Py_BuildValue convert:

  - Strings:
    standard C & unicode; null-terminated or by length

Python-Provided conversions...

- PyArg_ParseTuple/Py_BuildValue convert:
  - Strings:
    standard C & unicode; null-terminated or by length
  - Integers:
    signed & unsigned; short, long and long long

Python-Provided conversions...

- PyArg_ParseTuple/Py_BuildValue convert:
  - Strings:
    standard C & unicode; null-terminated or by length
  - Integers:
    signed & unsigned; short, long and long long
  - Other numbers:
    double, float, Py_complex

Python-Provided conversions...

- PyArg_ParseTuple/Py_BuildValue convert:
  - Strings:
    standard C & unicode; null-terminated or by length
  - Integers:
    signed & unsigned; short, long and long long
  - Other numbers:
    double, float, Py_complex
  - Other:
    Py_ssize_t, PyObject *

Python-Provided conversions...

- PyArg_ParseTuple/Py_BuildValue convert:
  - Strings:
    standard C & unicode; null-terminated or by length
  - Integers:
    signed & unsigned; short, long and long long
  - Other numbers:
    double, float, Py_complex
  - Other:
    Py_ssize_t, PyObject *
    "anything" IFF... programmer writes conversion!

Solution (Part1): API Functions & Integers

Solution (Part1): API Functions & Integers

- Key Insights:

Solution (Part 1): API Functions & Integers

- Key Insights:

1. Integers pass easily

Solution (Part1): API Functions & Integers

- Key Insights:

1. Integers pass easily

2. API functions exist to get one object from another

Solution (Part1): API Functions & Integers

- Key Insights:

1. Integers pass easily

2. API functions exist to get one object from another

3. Only need to get a single object across

Solution (Part1): API Functions & Integers

- Key Insights:

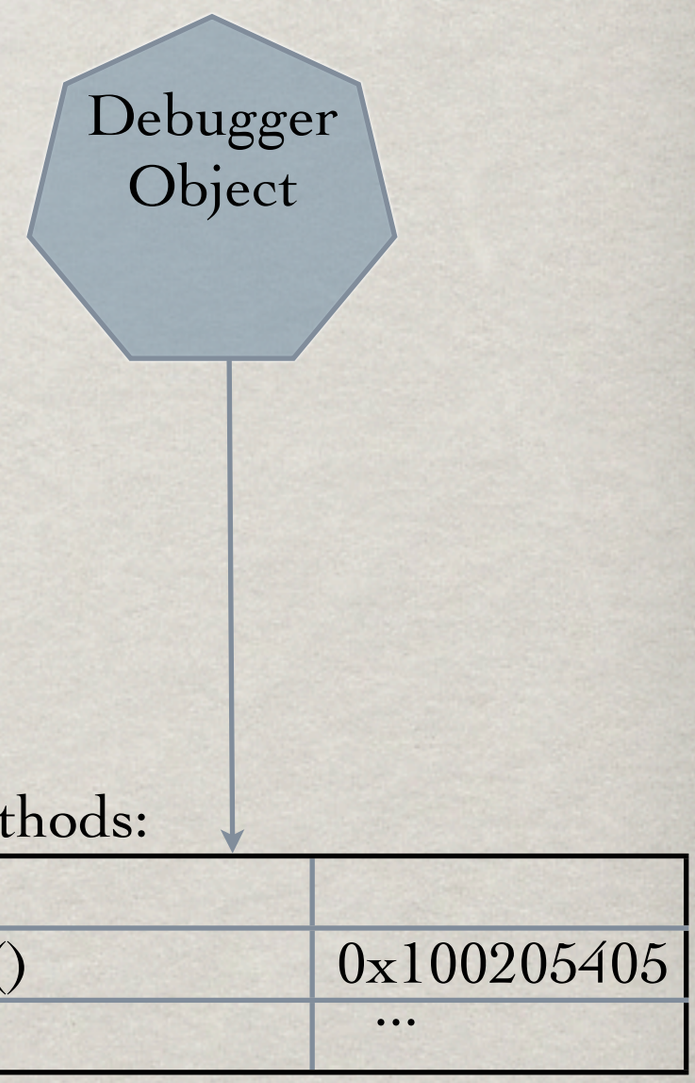
1. Integers pass easily

2. API functions exist to get one object from another

3. Only need to get a single object across

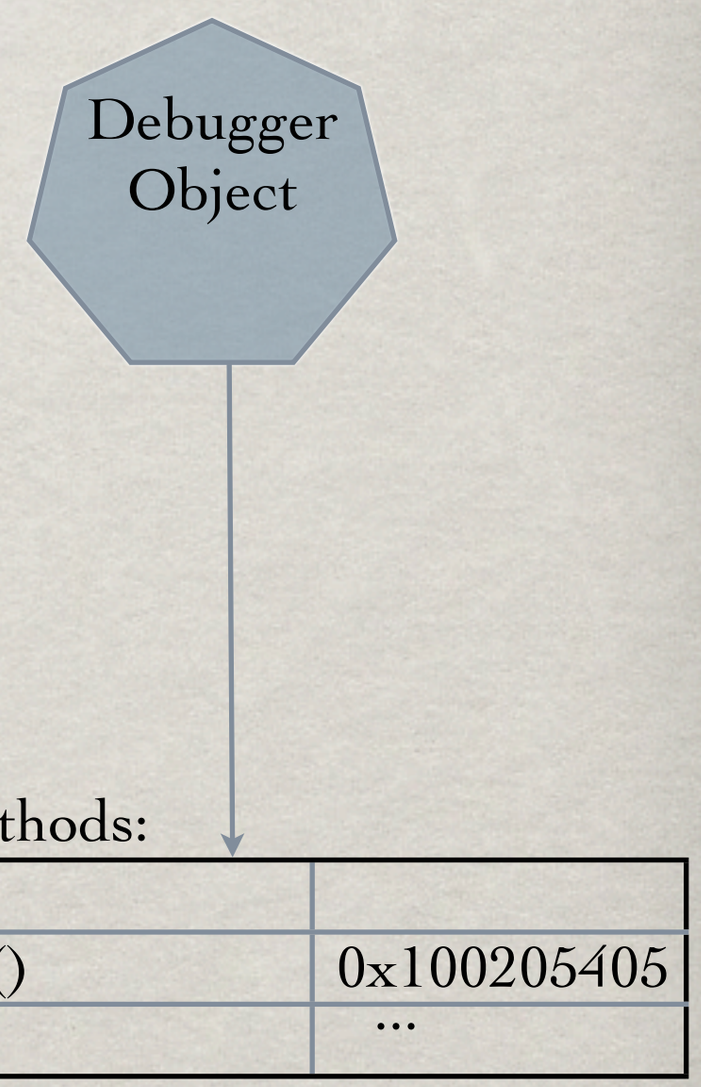
4. Use combination of API & integers

Getting a Single Object "across" - The Debugger



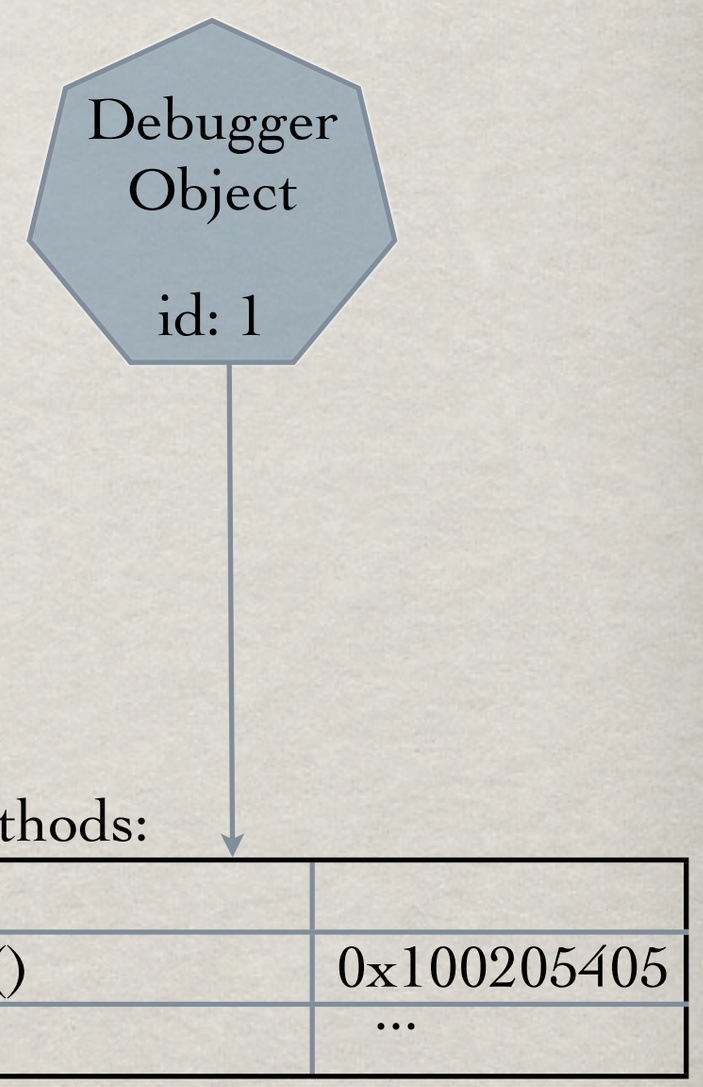
Getting a Single Object "across" - The Debugger

1. Attach unique integer
   to every debugger (id!)



Getting a Single Object "across" - The Debugger

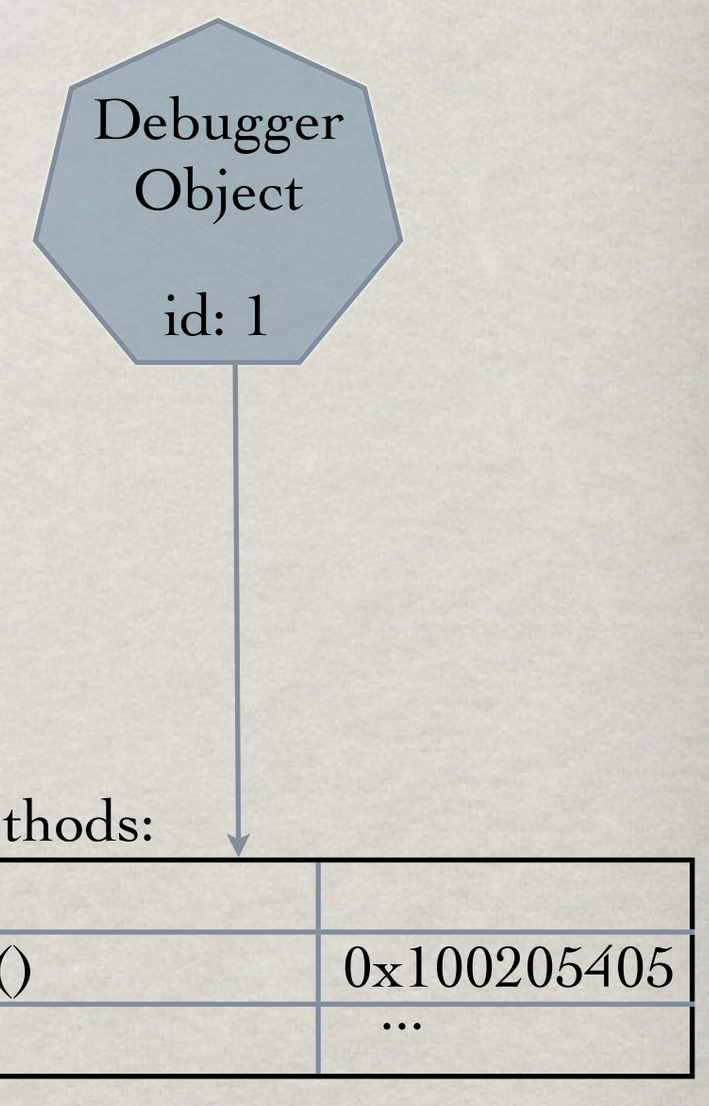
1. Attach unique integer
   to every debugger (id!)



Getting a Single Object "across" - The Debugger

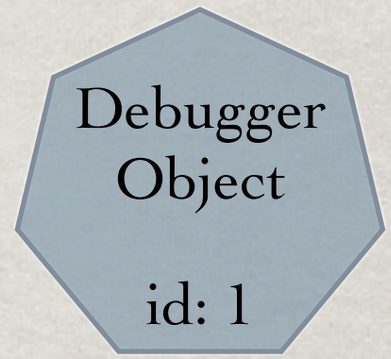
1. Attach unique integer to every debugger (id!)

2. Pass the appropriate debugger id to Python



Getting a Single Object "across" - The Debugger

1. Attach unique integer to every debugger (id!)



2. Pass the appropriate debugger id to Python

```
buffer = "lldb.debugger_unique_id = 1";
PyRun_SimpleString (buffer);
```

Debugger methods:

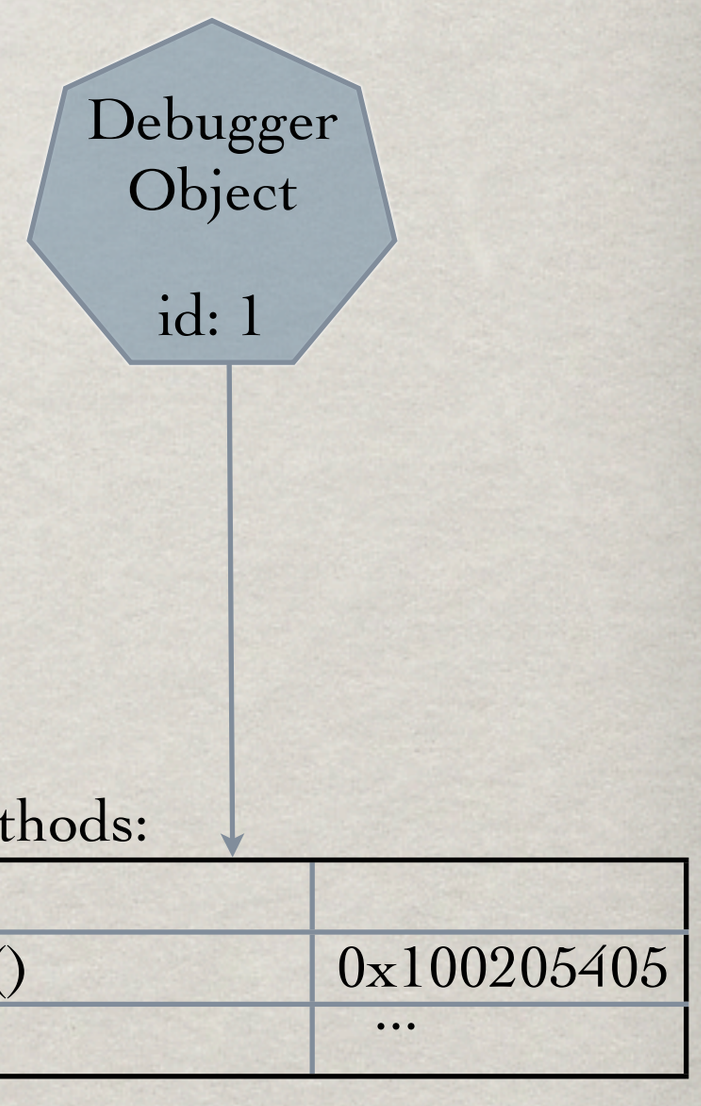
CreateTarget()	0x100205405
...	...

Getting a Single Object "across" - The Debugger

1. Attach unique integer to every debugger (id!)

2. Pass the appropriate debugger id to Python

3. Create STATIC API function to get the debugger from the id!

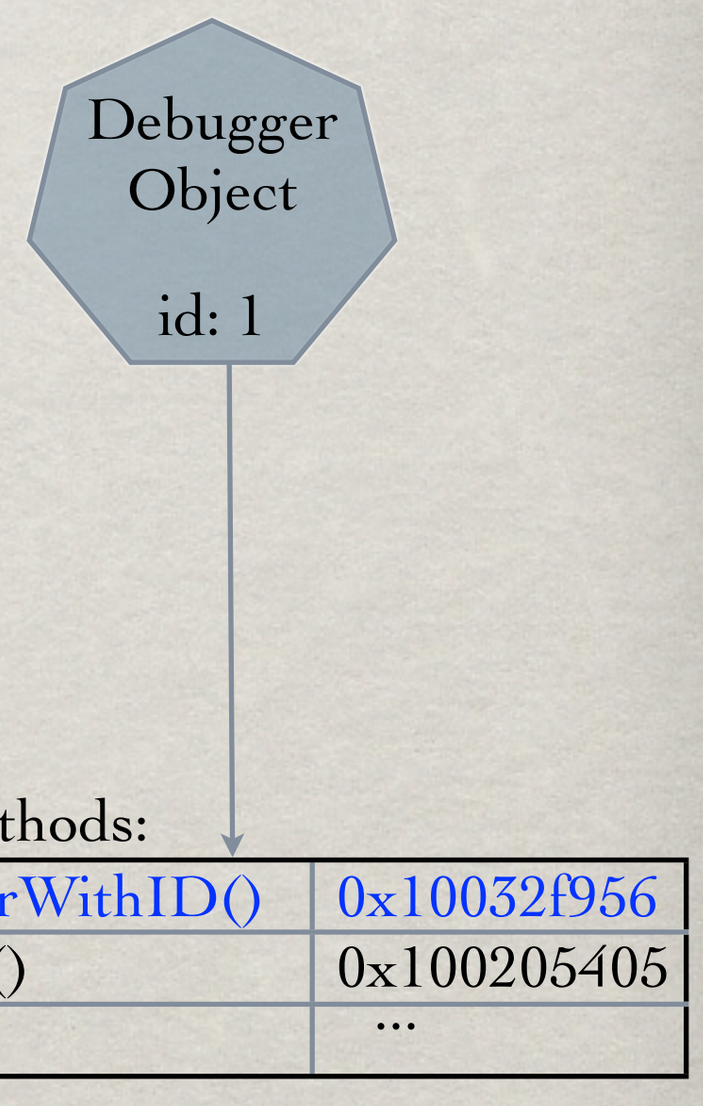


Getting a Single Object "across" - The Debugger

1. Attach unique integer to every debugger (id!)

2. Pass the appropriate debugger id to Python

3. Create STATIC API function to get the debugger from the id!

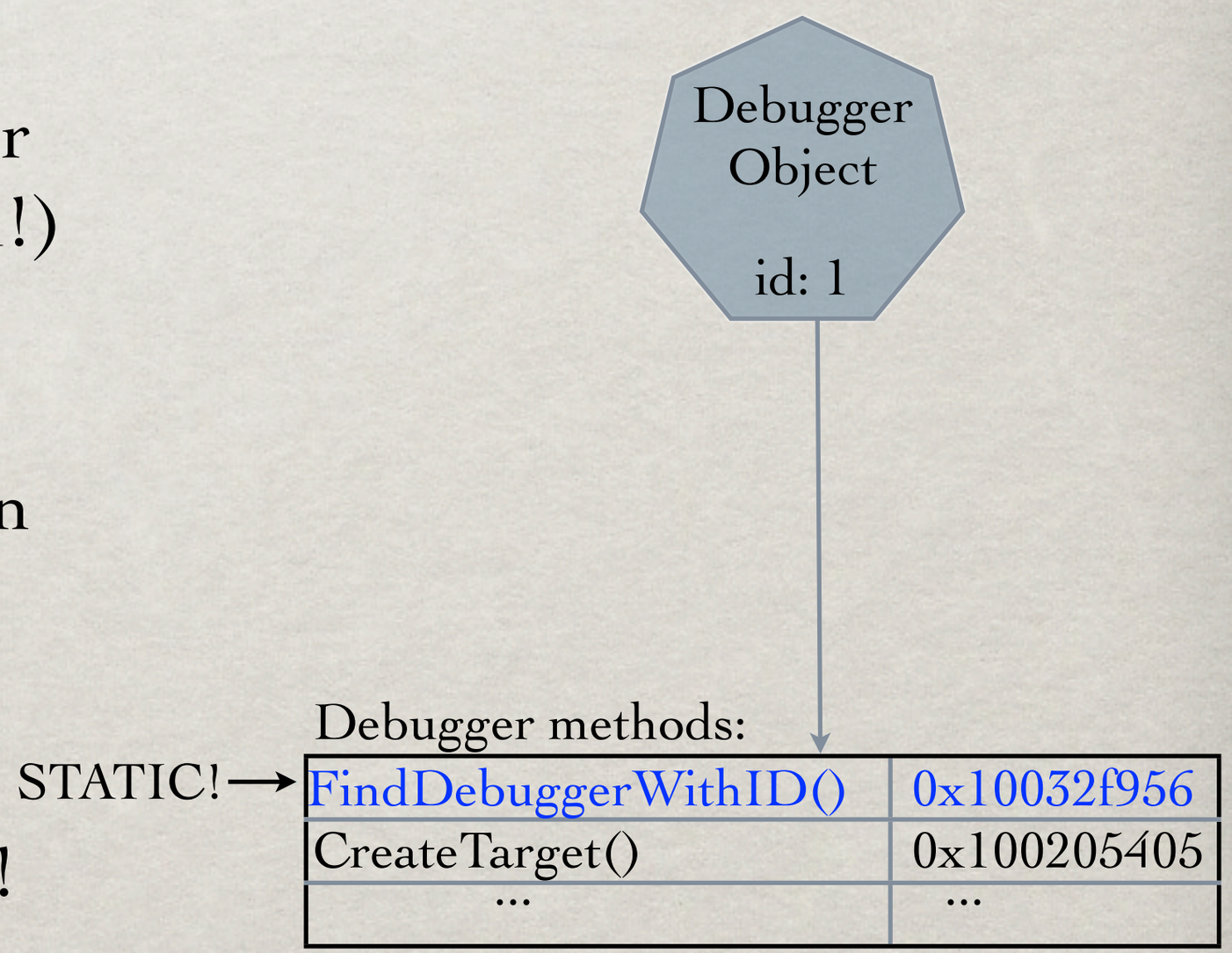


Getting a Single Object "across" - The Debugger

1. Attach unique integer to every debugger (id!)

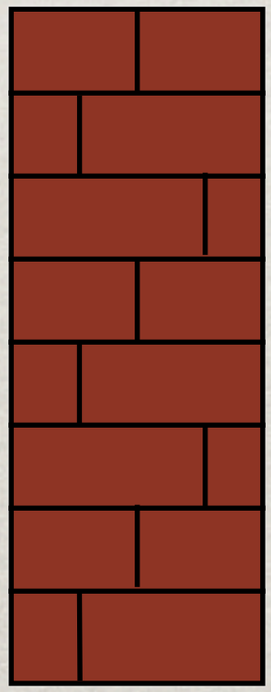
2. Pass the appropriate debugger id to Python

3. Create STATIC API function to get the debugger from the id!

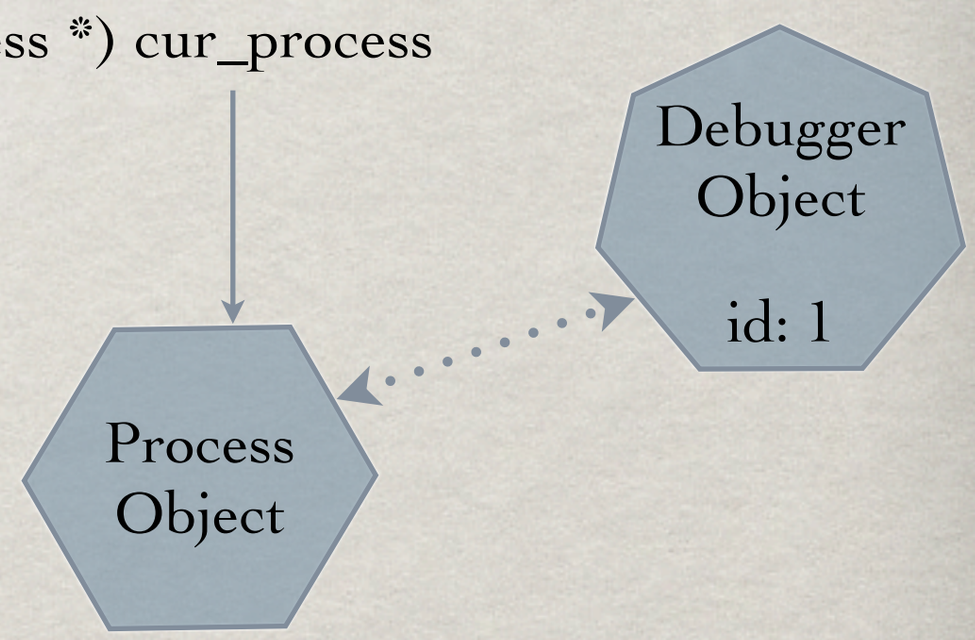


Putting It All Together...

(lldb) script



(Process *) cur_process



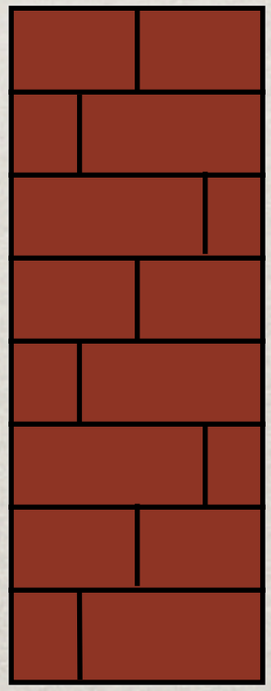
Interactive Python Interpreter

lldb debugger (C++ code)

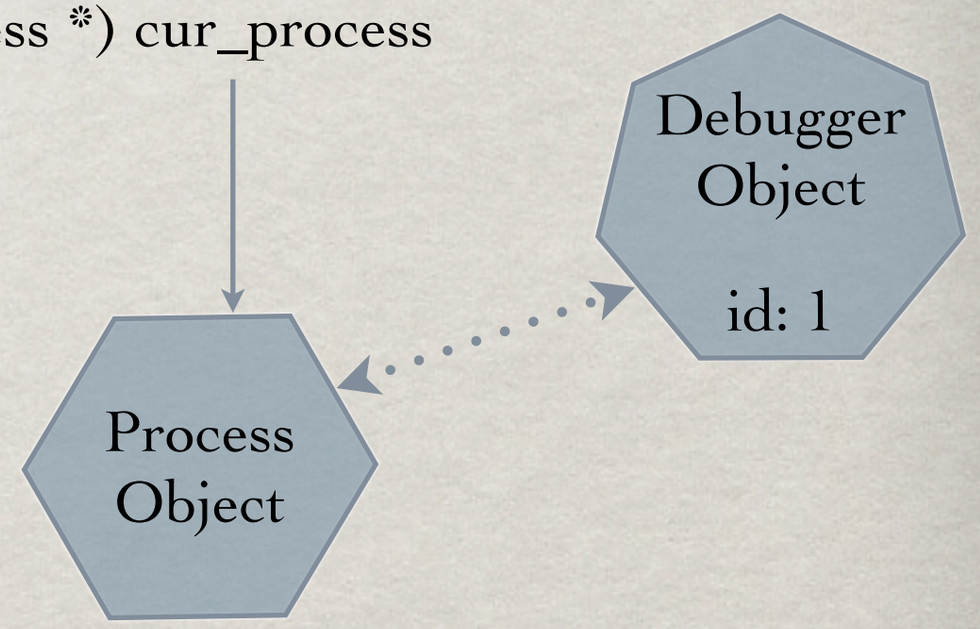
Putting It All Together...

sprintf (tmp_str,"lldb.debugger_unique_id = %d", debugger.id);
PyRun_SimpleString (tmp_str);

(lldb) script



(Process *) cur_process



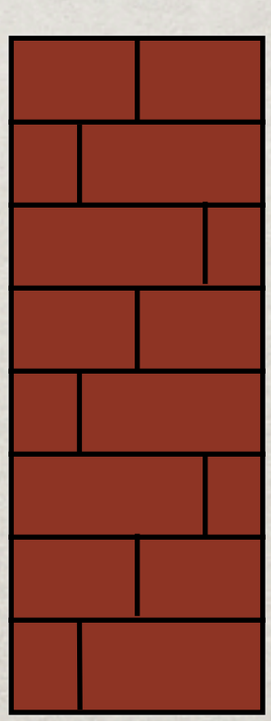
Interactive Python Interpreter

lldb debugger (C++ code)

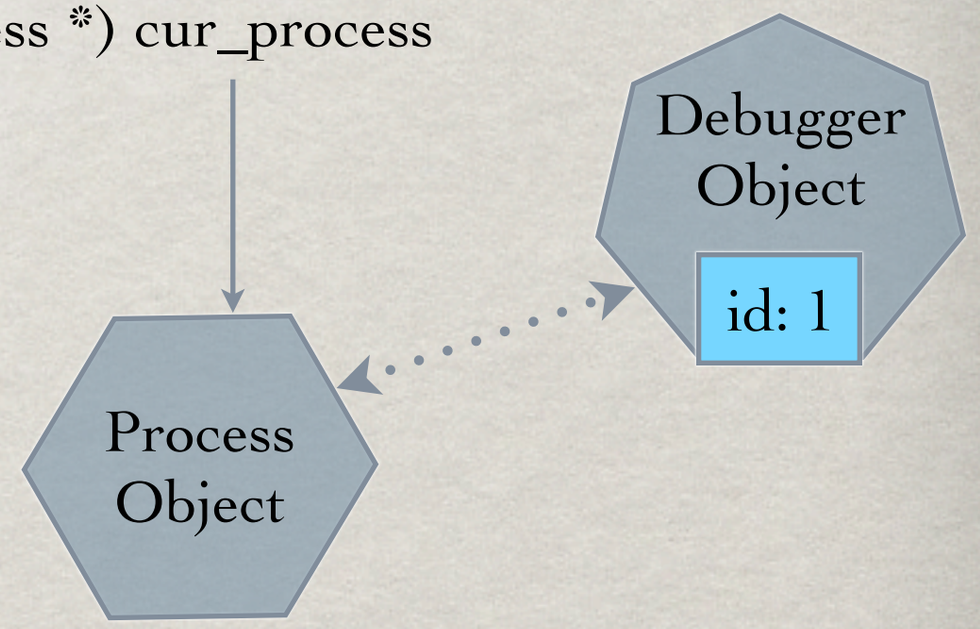
Putting It All Together...

sprintf (tmp_str,"lldb.debugger_unique_id = %d", debugger.id);
PyRun_SimpleString (tmp_str);

(lldb) script



(Process *) cur_process



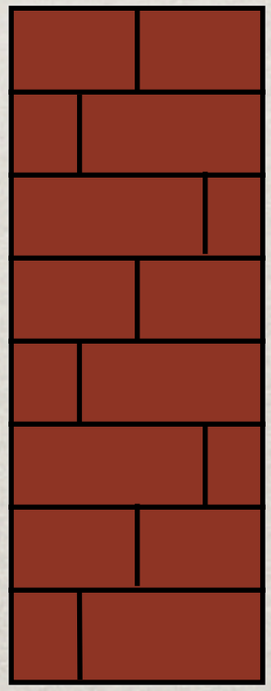
Interactive Python Interpreter

lldb debugger (C++ code)

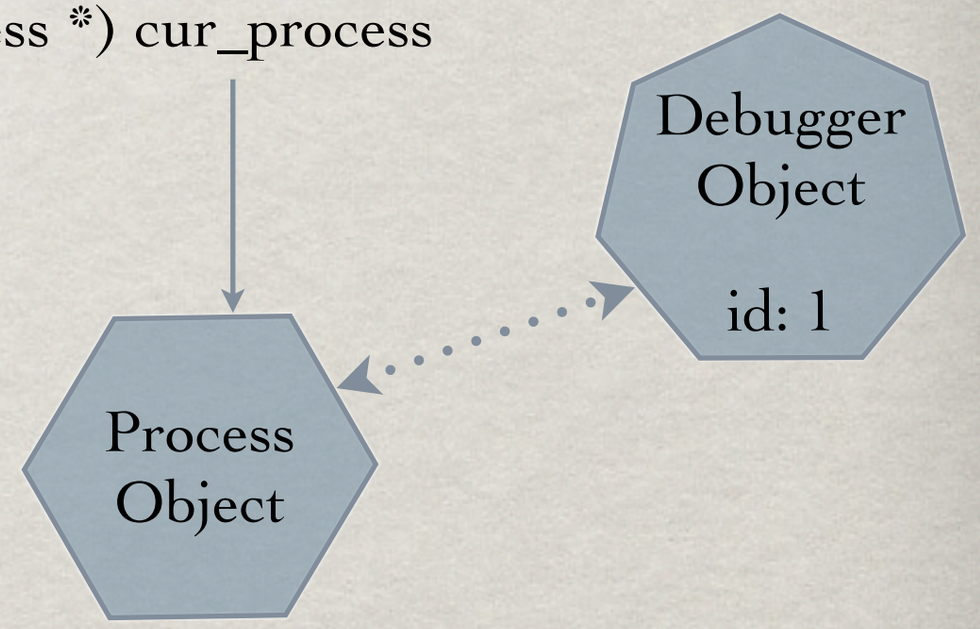
Putting It All Together...

sprintf (tmp_str,"lldb.debugger_unique_id = %d", debugger.id);
PyRun_SimpleString (tmp_str);

(lldb) script



(Process *) cur_process

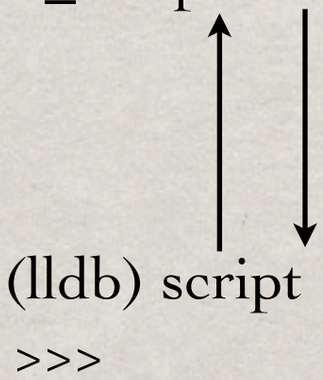


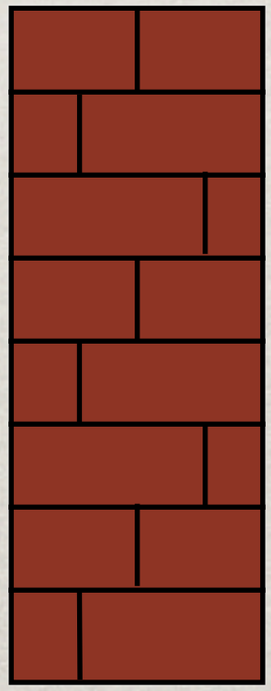
Interactive Python Interpreter

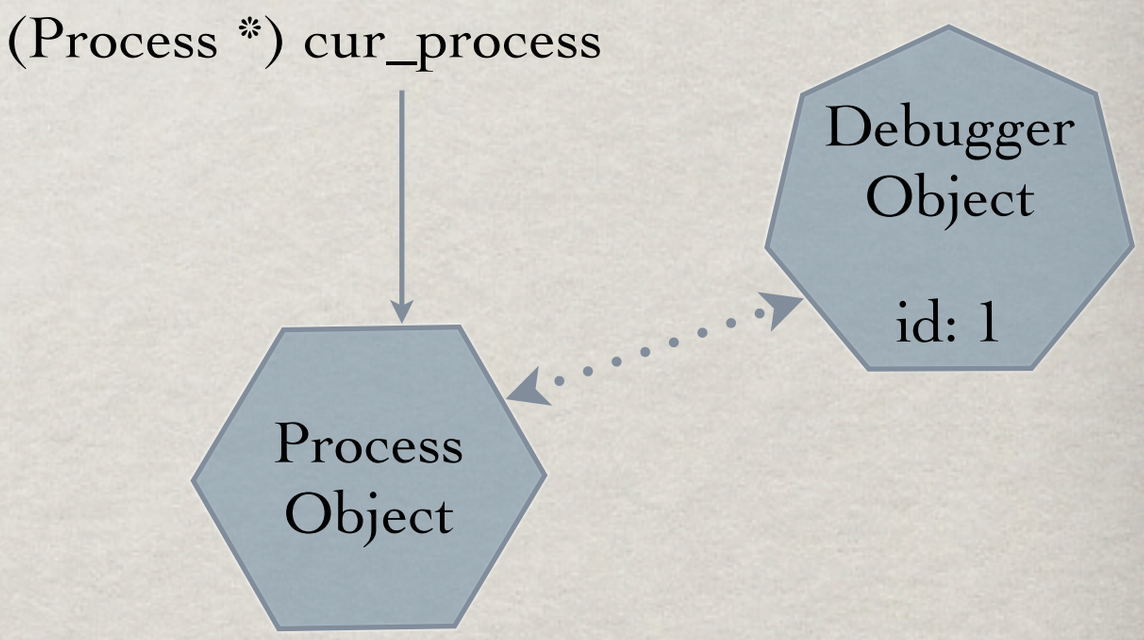
lldb debugger (C++ code)

Putting It All Together...

sprintf (tmp_str,"lldb.debugger_unique_id = %d", debugger.id);
PyRun_SimpleString (tmp_str);





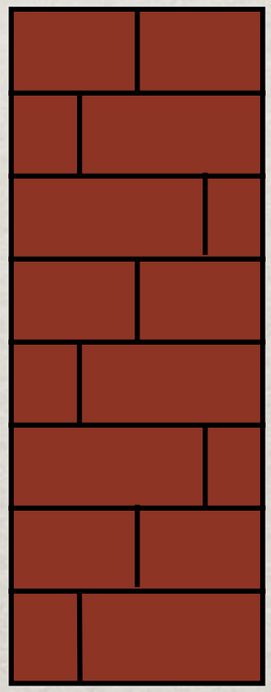


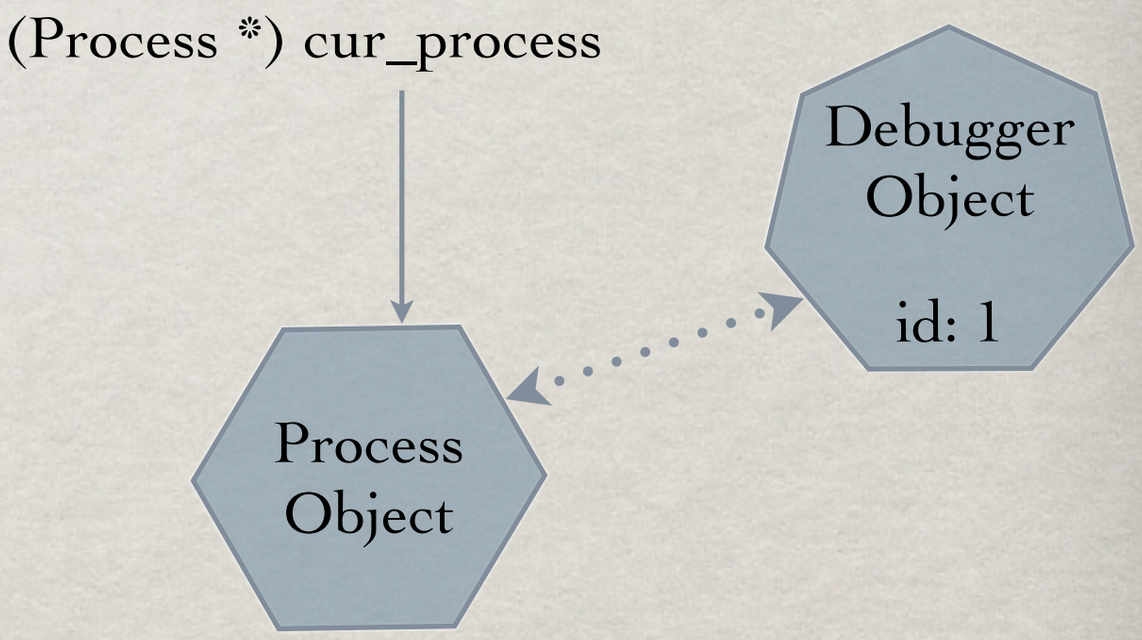
Interactive Python Interpreter

lldb debugger (C++ code)

Putting It All Together...

```
(lldb) script
>>> id = lldb.debugger_unique_id
>>>
```



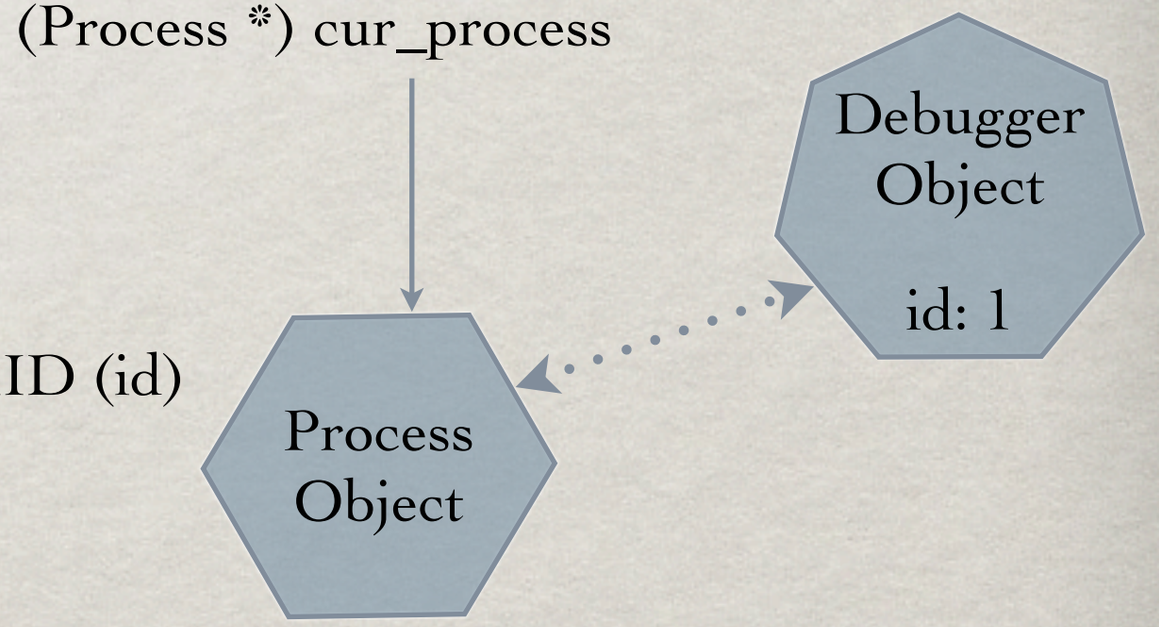


Interactive Python Interpreter

lldb debugger (C++ code)

Putting It All Together...

(lldb) script
>>> id = lldb.debugger_unique_id
>>> dbg = lldb.SBDebugger.FindDebuggerWithID (id)
>>>

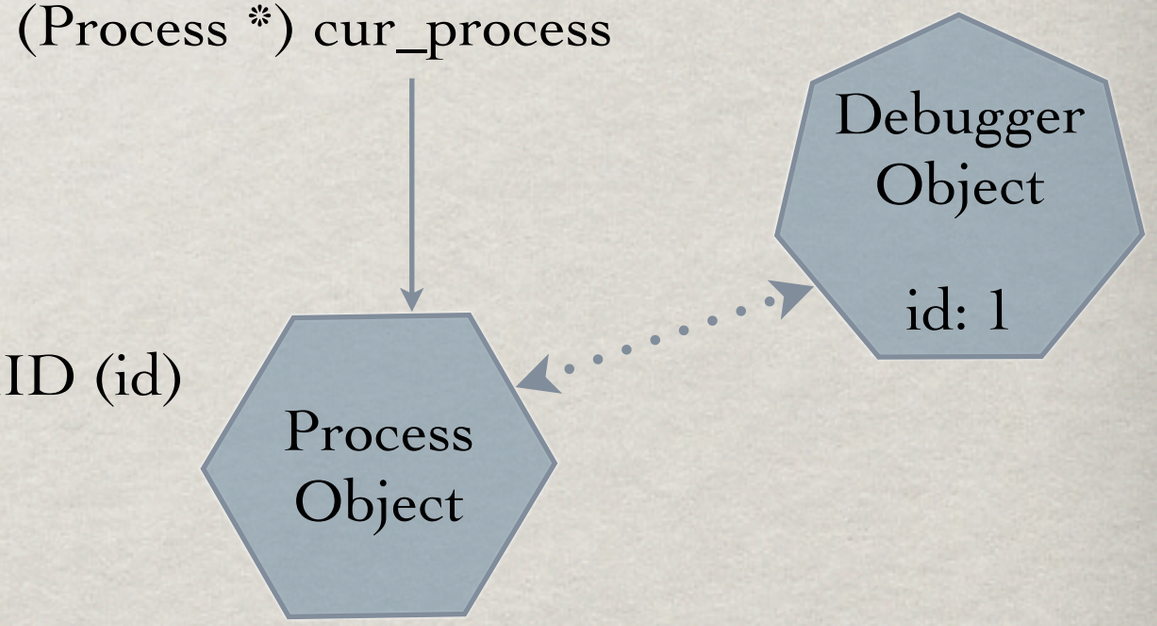


Interactive Python Interpreter

lldb debugger (C++ code)

Putting It All Together...

(lldb) script
>>> id = lldb.debugger_unique_id
>>> dbg = lldb.SBDebugger.FindDebuggerWithID (id)
>>> target = dbg.GetSelectedTarget()
>>>



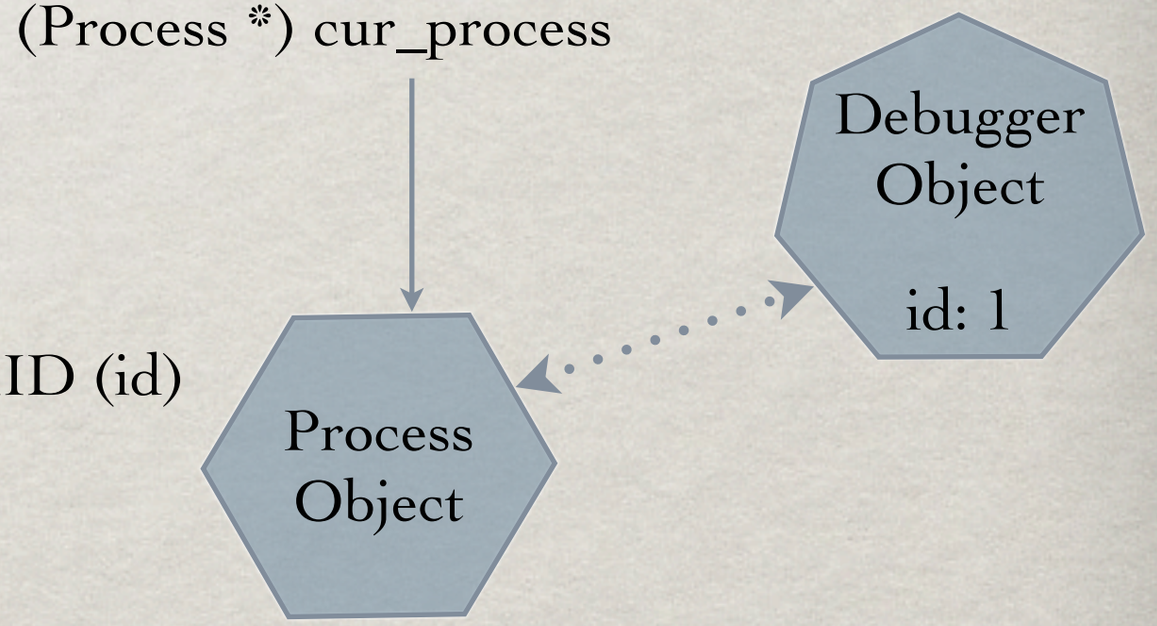
Interactive Python Interpreter

lldb debugger (C++ code)

Putting It All Together...

(lldb) script

```
>>> id = lldb.debugger_unique_id
>>> dbg = lldb.SBDebugger.FindDebuggerWithID (id)
>>> target = dbg.GetSelectedTarget()
>>> process = target.GetProcess()
>>>
```

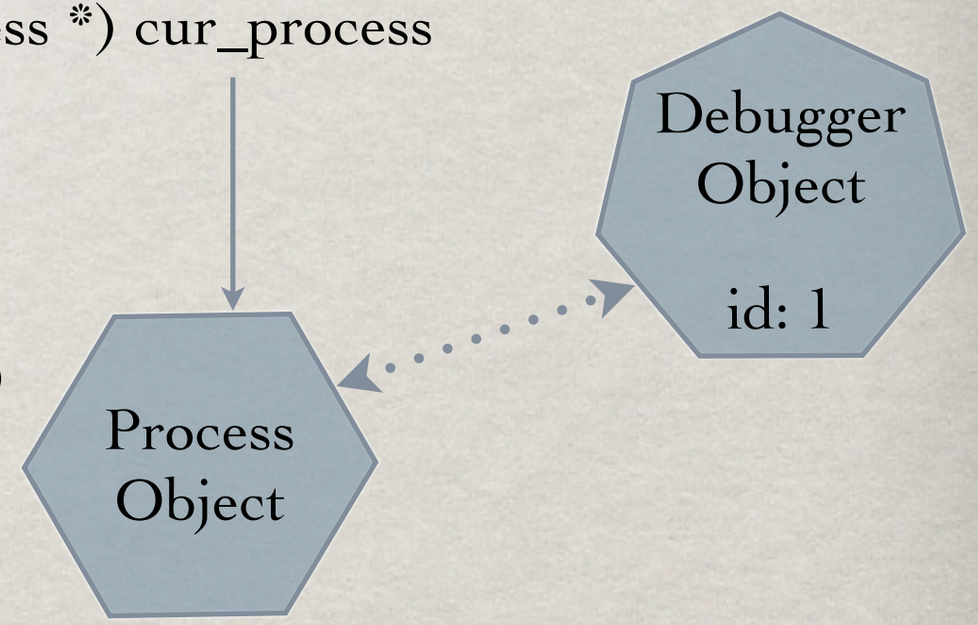


Interactive Python Interpreter

lldb debugger (C++ code)

Putting It All Together...

```
(lldb) script
>>> id = lldb.debugger_unique_id
>>> dbg = lldb.SBDebugger.FindDebuggerWithID (id)
>>> target = dbg.GetSelectedTarget()
>>> process = target.GetProcess()
>>> process.GetNumThreads()
5
>>>
```



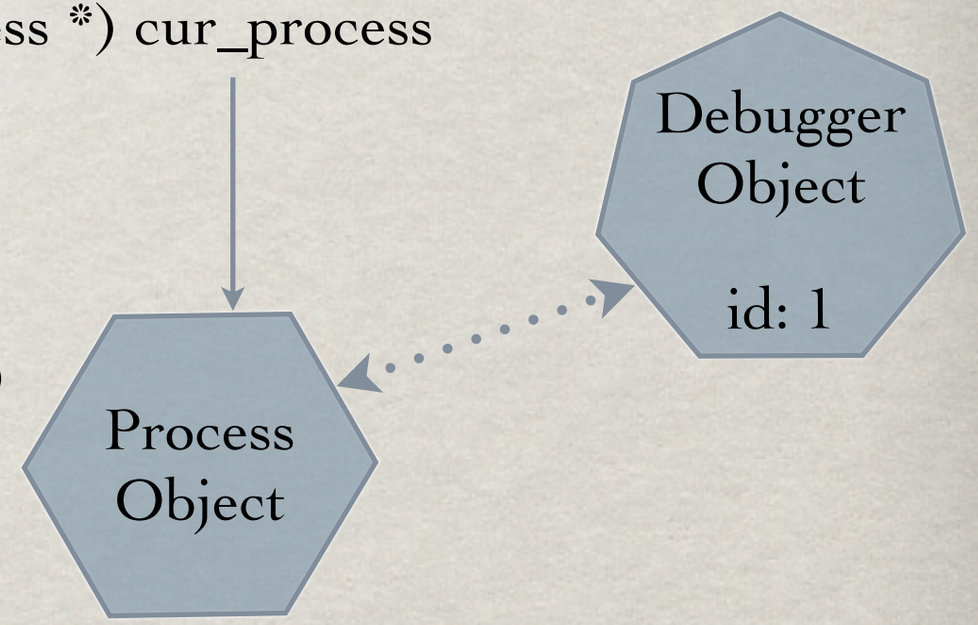
Interactive Python Interpreter

lldb debugger (C++ code)

Putting It All Together...

```
(lldb) script
>>> id = lldb.debugger_unique_id
>>> dbg = lldb.SBDebugger.FindDebuggerWithID (id)
>>> target = dbg.GetSelectedTarget()
>>> process = target.GetProcess()
>>> process.GetNumThreads()
5
>>>
```

SUCCESS!



Interactive Python Interpreter

lldb debugger (C++ code)

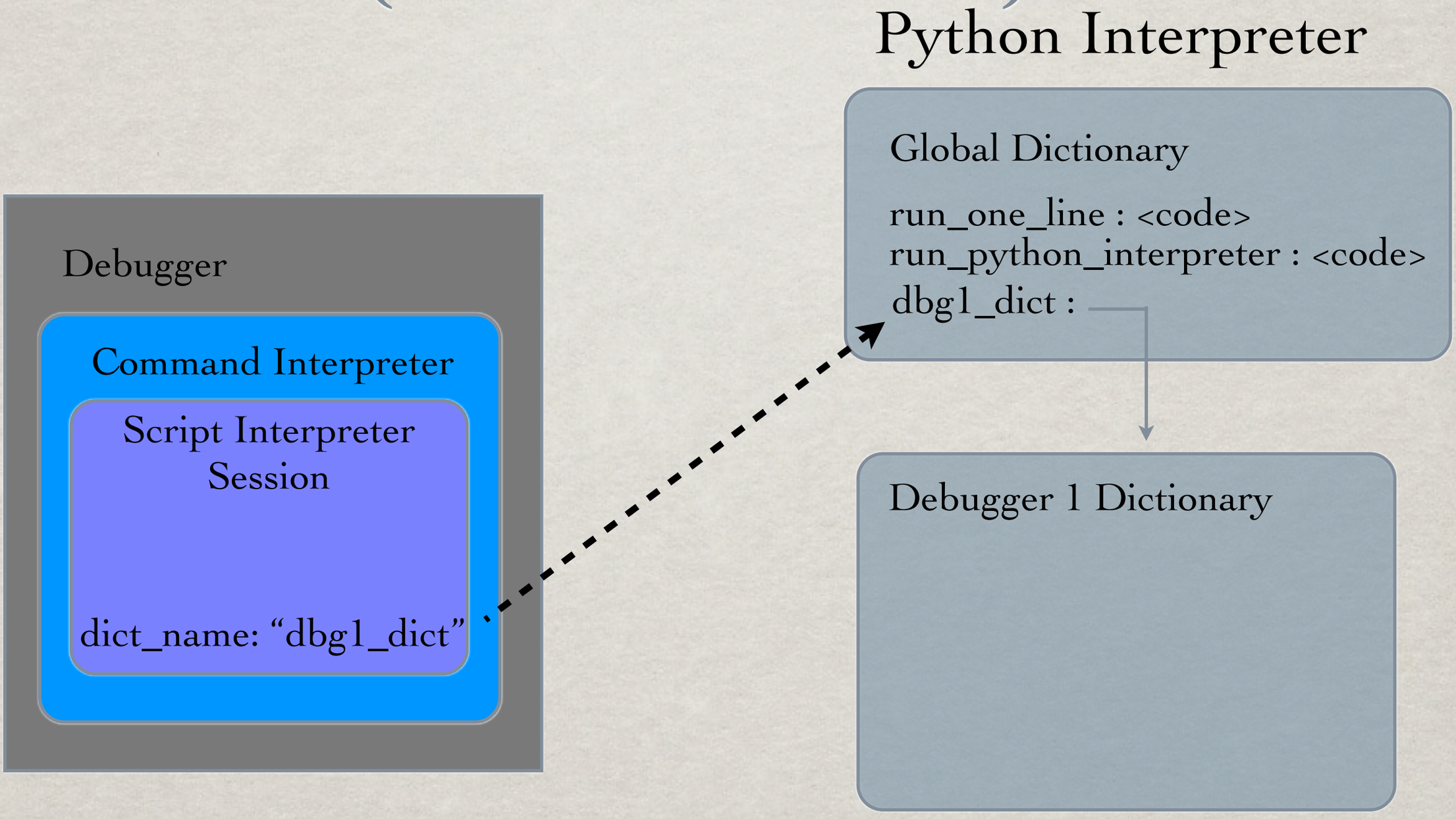
Particular Problems & Solutions

- Passing Pointers & C++ Objects to Python
- Single Dictionary Across Debugger Session
- Multiple Debuggers/Single Interpreter

Why a Debugger-Level Dictionary?

- Persistent, re-usable definitions
- Multiple debugger sessions
  - independent, non-interfering definitions

How It Works (Reminder)



Interactive Interpreter & One Line Commands

Interactive Interpreter & One Line Commands

- All function & variables defs go into session dictionary

Interactive Interpreter & One Line Commands

- All function & variables defs go into session dictionary
- All code is executed in context of session dictionary

Interactive Interpreter & One Line Commands

- All function & variables defs go into session dictionary
- All code is executed in context of session dictionary
- Dictionary persists (in globals()) => definitions persist

Breakpoint Commands: A Problem

Breakpoint Commands: A Problem

- No encapsulating run environment => no way to run using session dictionary as global dict

Breakpoint Commands: A Problem

- No encapsulating run environment => no way to run using session dictionary as global dict
- Breakpoint script function is called from global Python environment

Breakpoint Commands: A Problem

- No encapsulating run environment => no way to run using session dictionary as global dict
- Breakpoint script function is called from global Python environment
- Solution: Modify the global environment (carefully!)

When Creating Command Script...

```
global count
count = count + 1
print “Hit this breakpoint “ + repr (count) + “ times!”
```

When Creating Command Script...

```
def some_obscure_function_name (frame, bp_loc       ):
    global count
    count = count + 1
    print “Hit this breakpoint “ + repr (count) + “ times!”
```

When Creating Command Script...

```
def some_obscure_function_name (frame, bp_loc, dict):
    global count
    count = count + 1
    print "Hit this breakpoint " + repr (count) + " times!"
```

When Creating Command Script...

def some_obscure_function_name (frame, bp_loc, dict):

Dictionary set-up magic

```
global count
count = count + 1
print “Hit this breakpoint “ + repr (count) + “ times!”
```

When Creating Command Script...

```
def some_obscure_function_name (frame, bp_loc, dict):
```

Dictionary set-up magic

```
    global count
    count = count + 1
    print "Hit this breakpoint " + repr (count) + " times!"
```

Dictionary clean-up magic

Dictionary Set-Up Magic

Global Python Dictionary

Key_1: Value1
Key_2: Value2
Key_3: Value3
etc.

Debugger Session Dictionary

Key_A: Value1
Key_B: Value2
Key_C: Value3
etc.

Dictionary Set-Up Magic

Global Python Dictionary

Key_1: Value1
Key_2: Value2
Key_3: Value3
etc.

Debugger Session Dictionary

Key_A: Value1
Key_B: Value2
Key_C: Value3
etc.

[Key_1, Key_2, Key_3]

Dictionary Set-Up Magic

Global Python Dictionary

Key_1: Value1
Key_2: Value2
Key_3: Value3
etc.

Debugger Session Dictionary

Key_A: Value1
Key_B: Value2
Key_C: Value3
etc.

[Key_1, Key_2, Key_3]

[Key_A, Key_B, Key_C]

Dictionary Set-Up Magic

Global Python Dictionary

Key_1: Value1
Key_2: Value2
Key_3: Value3
etc.

Old_Keys:
New_Keys:

Debugger Session Dictionary

Key_A: Value1
Key_B: Value2
Key_C: Value3
etc.

[Key_1, Key_2, Key_3]

[Key_A, Key_B, Key_C]

Dictionary Set-Up Magic

Global Python Dictionary

Key_1: Value1
Key_2: Value2
Key_3: Value3
etc.
Old_Keys: ...
New_Keys: ...

Debugger Session Dictionary

Key_A: Value1
Key_B: Value2
Key_C: Value3
etc.

Dictionary Set-Up Magic

Global Python Dictionary

Key_1: Value1
Key_2: Value2
Key_3: Value3
etc.
Old_Keys: ...
New_Keys: ...

Debugger Session Dictionary

Key_A: Value1
Key_B: Value2
Key_C: Value3
etc.

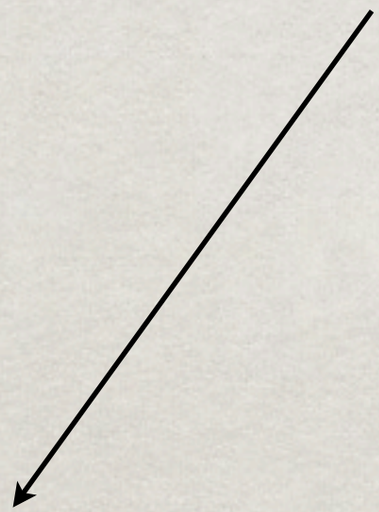
Dictionary Set-Up Magic

Global Python Dictionary

Key_1: Value1
Key_2: Value2
Key_3: Value3
etc.
Old_Keys: ...
New_Keys: ...

Debugger Session Dictionary

Key_A: Value1
Key_B: Value2
Key_C: Value3
etc.



Dictionary Set-Up Magic

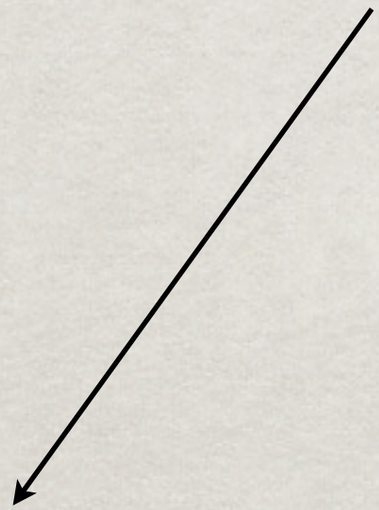
Global Python Dictionary

Key_1: Value1
Key_2: Value2
Key_3: Value3
etc.
Old_Keys: ...
New_Keys: ...

Key_A: Value1
Key_B: Value2
Key_C: Value3
etc.

Debugger Session Dictionary

Key_A: Value1
Key_B: Value2
Key_C: Value3
etc.



Dictionary Set-Up Magic

Global Python Dictionary

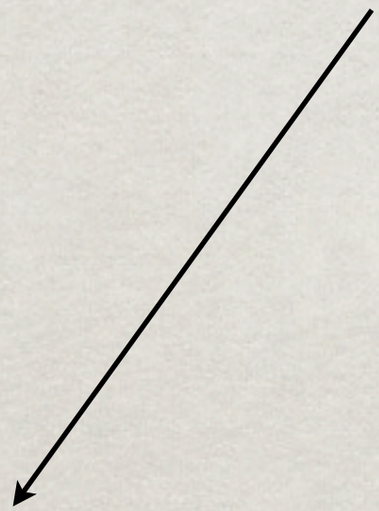
Key_1: Value1
Key_2: Value2
Key_3: Value3
etc.

Old_Keys: ...
New_Keys: ...

Key_A: Value1
Key_B: Value2
Key_C: Value3
etc.

Debugger Session Dictionary

Key_A: Value1
Key_B: Value2
Key_C: Value3
etc.



Breakpoint script executes in context of GLOBAL dictionary.

Dictionary Clean-Up Magic

Global Python Dictionary

Key_1: Value1
Key_2: Value2
Key_3: Value3
etc.

Old_Keys: ...
New_Keys: ...

Key_A: Value1
Key_B: New_1
Key_C: New_2
etc.

Debugger Session Dictionary

Key_A: Value1
Key_B: Value2
Key_C: Value3
etc.

[Key_1, Key_2, Key_3]

[Key_A, Key_B, Key_C]

Dictionary Clean-Up Magic

Global Python Dictionary

Key_1: Value1
Key_2: Value2
Key_3: Value3
etc.
Old_Keys: ...
New_Keys: ...

Key_A: Value1
Key_B: New_1
Key_C: New_2
etc.

Debugger Session Dictionary

Key_A: Value1
Key_B: New_1
Key_C: New_2
etc.

[Key_1, Key_2, Key_3]

[Key_A, Key_B, Key_C]

Dictionary Clean-Up Magic

Global Python Dictionary

Key_1: Value1
Key_2: Value2
Key_3: Value3
etc.

Old_Keys: ...
New_Keys: ...

Key_A: Value1
Key_B: New_1
Key_C: New_2
etc.

Debugger Session Dictionary

Key_A: Value1
Key_B: New_1
Key_C: New_2
etc.

[Key_1, Key_2, Key_3]

[Key_A, Key_B, Key_C]

Dictionary Clean-Up Magic

Global Python Dictionary

Key_1: Value1
Key_2: Value2
Key_3: Value3
etc.
Old_Keys: ...
New_Keys: ...

Debugger Session Dictionary

Key_A: Value1
Key_B: New_1
Key_C: New_2
etc.

[Key_1, Key_2, Key_3]

[Key_A, Key_B, Key_C]

Dictionary Clean-Up Magic

Global Python Dictionary

Key_1: Value1
Key_2: Value2
Key_3: Value3
etc.

Debugger Session Dictionary

Key_A: Value1
Key_B: New_1
Key_C: New_2
etc.

Particular Problems & Solutions

- Passing Pointers & C++ Objects to Python
- Single Dictionary Across Debugger Session
- Multiple Debuggers/Single Interpreter

Desired GUI Debugger Behavior

- From single running GUI debugger process...
- Launch multiple executables...
  - ...each in a separate window
  - ...each with a separate debugger session

GUI Debugger Python Requirements

- Multiple debugger sessions
  - Multiple Script Interpreters/Dictionaries
- Ability to switch smoothly between them
- Complete isolation between sessions
- Thread safety
- No deadlocking

Possible approaches: Py_NewInterpreter?

Possible approaches: Py_NewInterpreter?

- Py_NewInterpreter makes new sub-interpreters but...

Possible approaches: Py_NewInterpreter?

- Py_NewInterpreter makes new sub-interpreters but...
  - ☒ Don't fully load/init modules in new interpreters

Possible approaches: Py_NewInterpreter?

- Py_NewInterpreter makes new sub-interpreters but...
  - ☒ Don't fully load/init modules in new interpreters
  - ☒ Some extensions may not work properly

Possible approaches: Py_NewInterpreter?

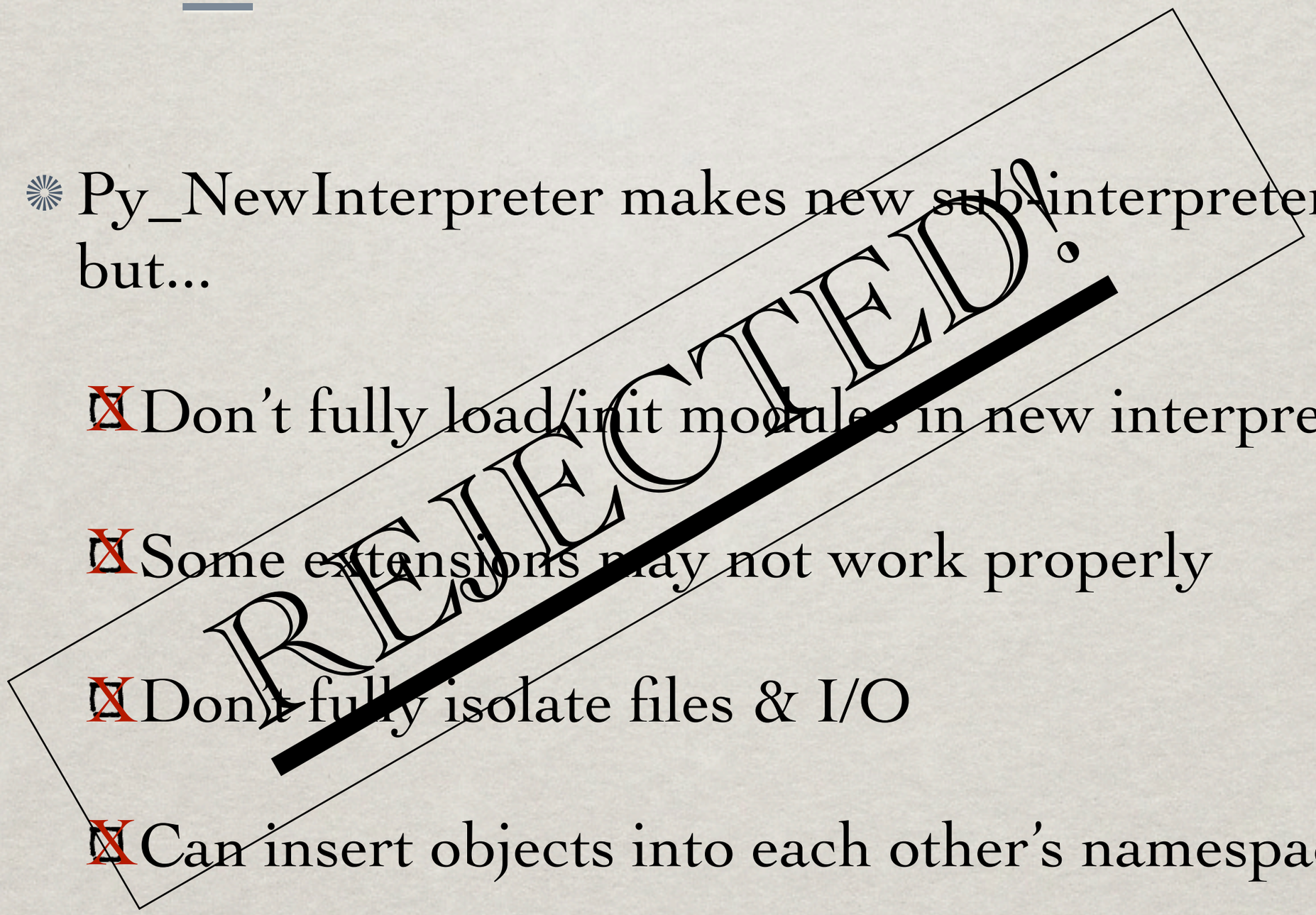
- Py_NewInterpreter makes new sub-interpreters but...
  - ☒ Don't fully load/init modules in new interpreters
  - ☒ Some extensions may not work properly
  - ☒ Don't fully isolate files & I/O

Possible approaches: Py_NewInterpreter?

- Py_NewInterpreter makes new sub-interpreters but...
  - ☒ Don't fully load/init modules in new interpreters
  - ☒ Some extensions may not work properly
  - ☒ Don't fully isolate files & I/O
  - ☒ Can insert objects into each other's namespaces

Possible approaches: Py_NewInterpreter?

- Py_NewInterpreter makes new sub-interpreters but...
  - ☒ Don't fully load/init modules in new interpreters
  - ☒ Some extensions may not work properly
  - ☒ Don't fully isolate files & I/O
  - ☒ Can insert objects into each other's namespaces



Possible Approaches: Rely On the GIL

Possible Approaches: Rely On the GIL

- Use GIL, PyGILState_Ensure, PyGILState_Release, Py_BEGIN_ALLOW_THREADS, etc.

Possible Approaches: Rely On the GIL

- Use GIL, PyGILState_Ensure, PyGILState_Release, Py_BEGIN_ALLOW_THREADS, etc.

- ☑ Can serialize calls into interpreter & prevent deadlock

Possible Approaches: Rely On the GIL

- Use GIL, PyGILState_Ensure, PyGILState_Release, Py_BEGIN_ALLOW_THREADS, etc.

- ☑ Can serialize calls into interpreter & prevent deadlock

- ☒ May release lock too soon

Possible Approaches: Rely On the GIL

- Use GIL, PyGILState_Ensure, PyGILState_Release, Py_BEGIN_ALLOW_THREADS, etc.

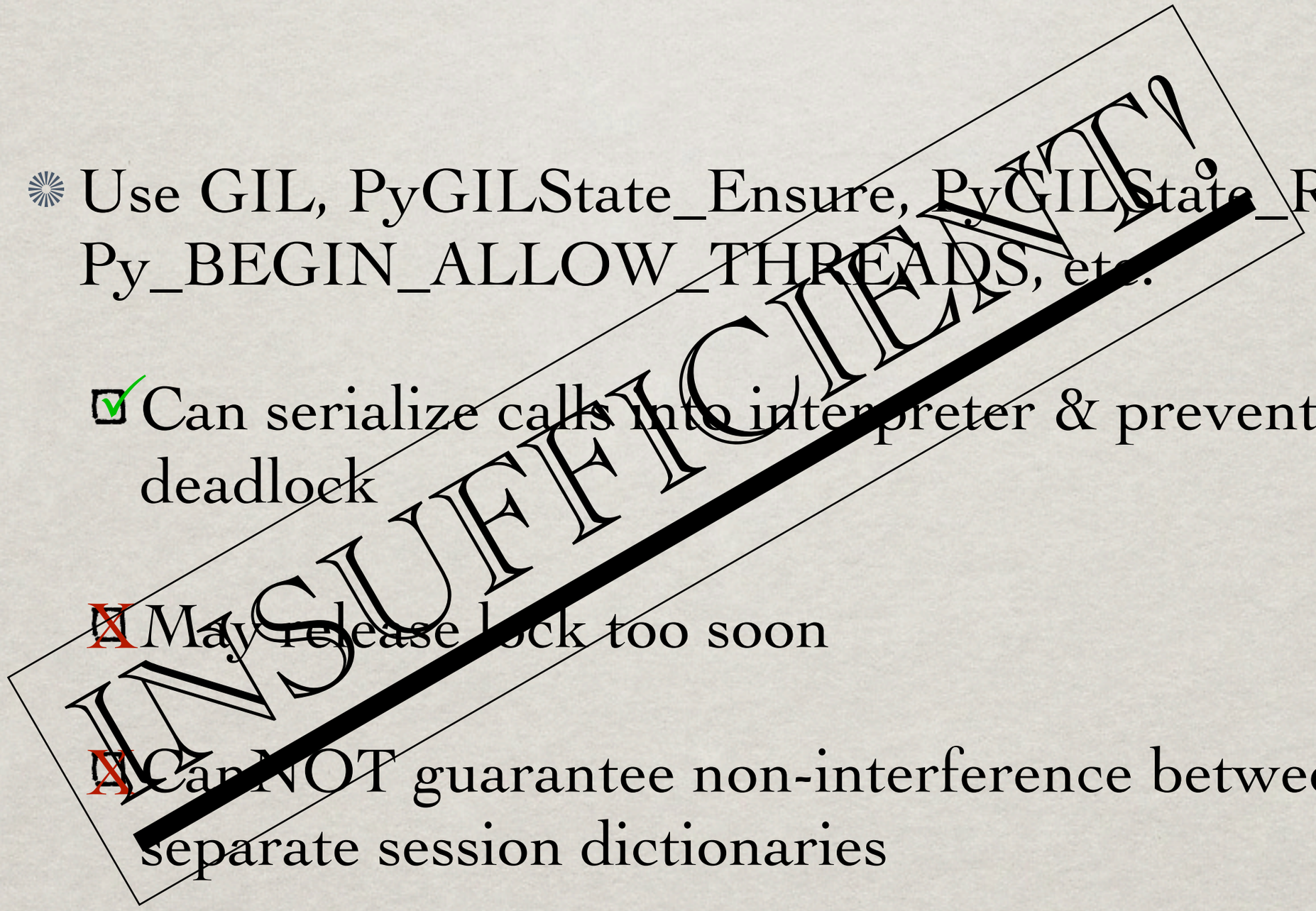
☑ Can serialize calls into interpreter & prevent deadlock

☒ May release lock too soon

☒ CanNOT guarantee non-interference between separate session dictionaries

Possible Approaches: Rely On the GIL

- Use GIL, PyGILState_Ensure, PyGILState_Release, Py_BEGIN_ALLOW_THREADS, etc.



- ☑ Can serialize calls into interpreter & prevent deadlock
- ☒ May release lock too soon
- ☒ Can NOT guarantee non-interference between separate session dictionaries

Possible Approaches: Write our Own!

Python Script Interpreter

Global Mutex-Predicate

Session 1

Input PTY
Output PTY
Session Dictionary
IsActive

Session 2

Input PTY
Output PTY
Session Dictionary
IsActive

Session 3

Input PTY
Output PTY
Session Dictionary
IsActive

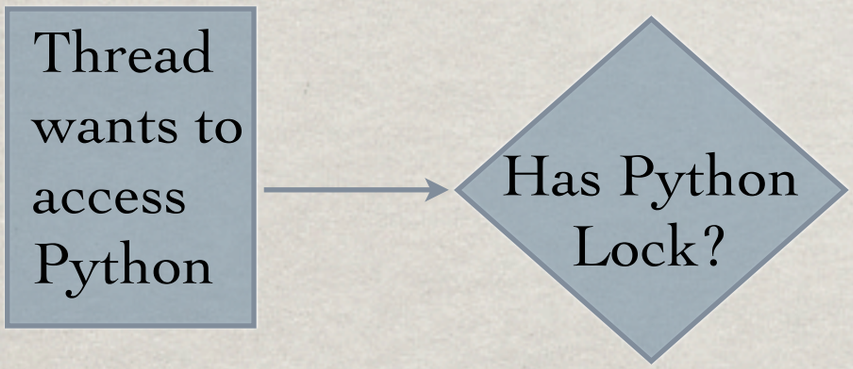
Possible Approaches: Write Our Own!

- Use a combination of pthread_mutex, pthread_cond_wait & our own Python lock:

Thread wants to access Python

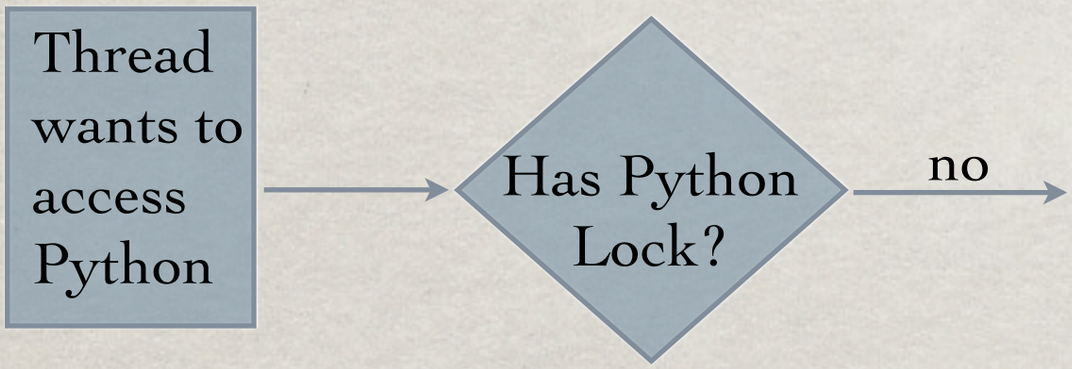
Possible Approaches: Write Our Own!

- Use a combination of pthread_mutex, pthread_cond_wait & our own Python lock:



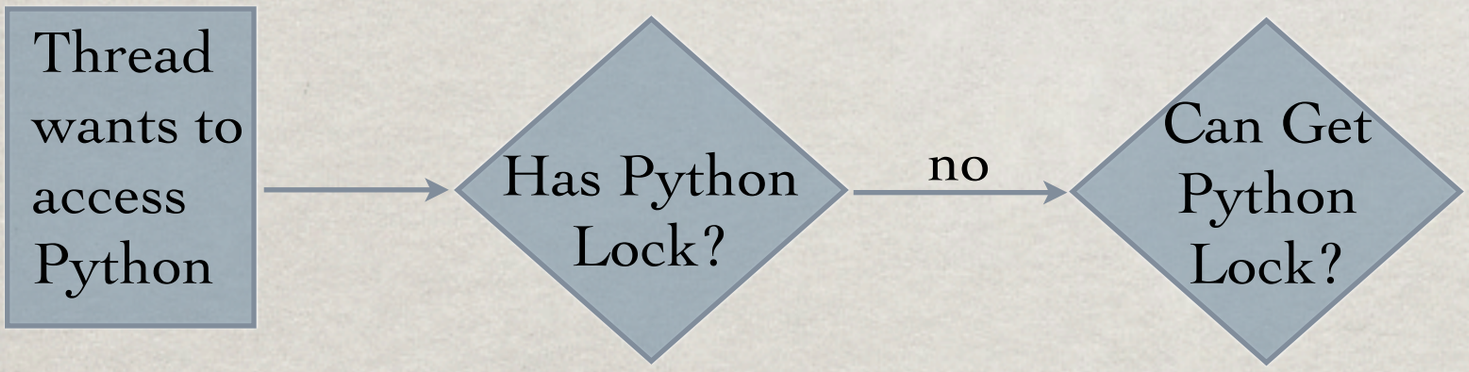
Possible Approaches: Write Our Own!

- Use a combination of pthread_mutex, pthread_cond_wait & our own Python lock:



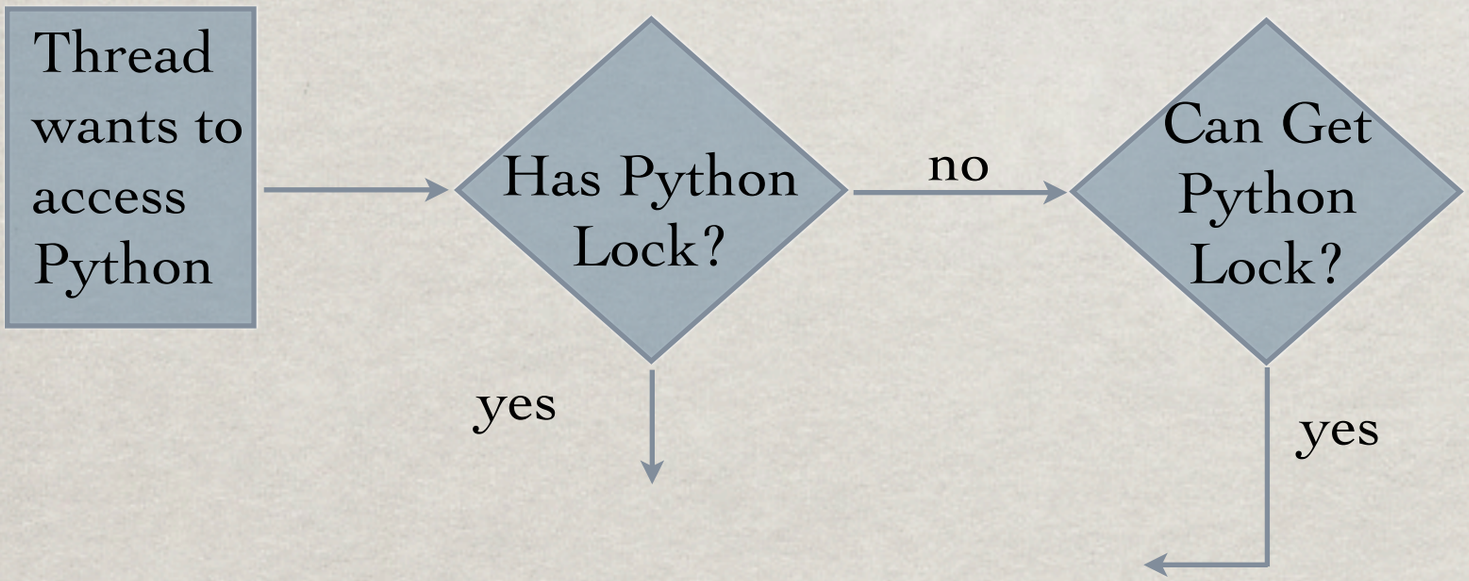
Possible Approaches: Write Our Own!

- Use a combination of pthread_mutex, pthread_cond_wait & our own Python lock:



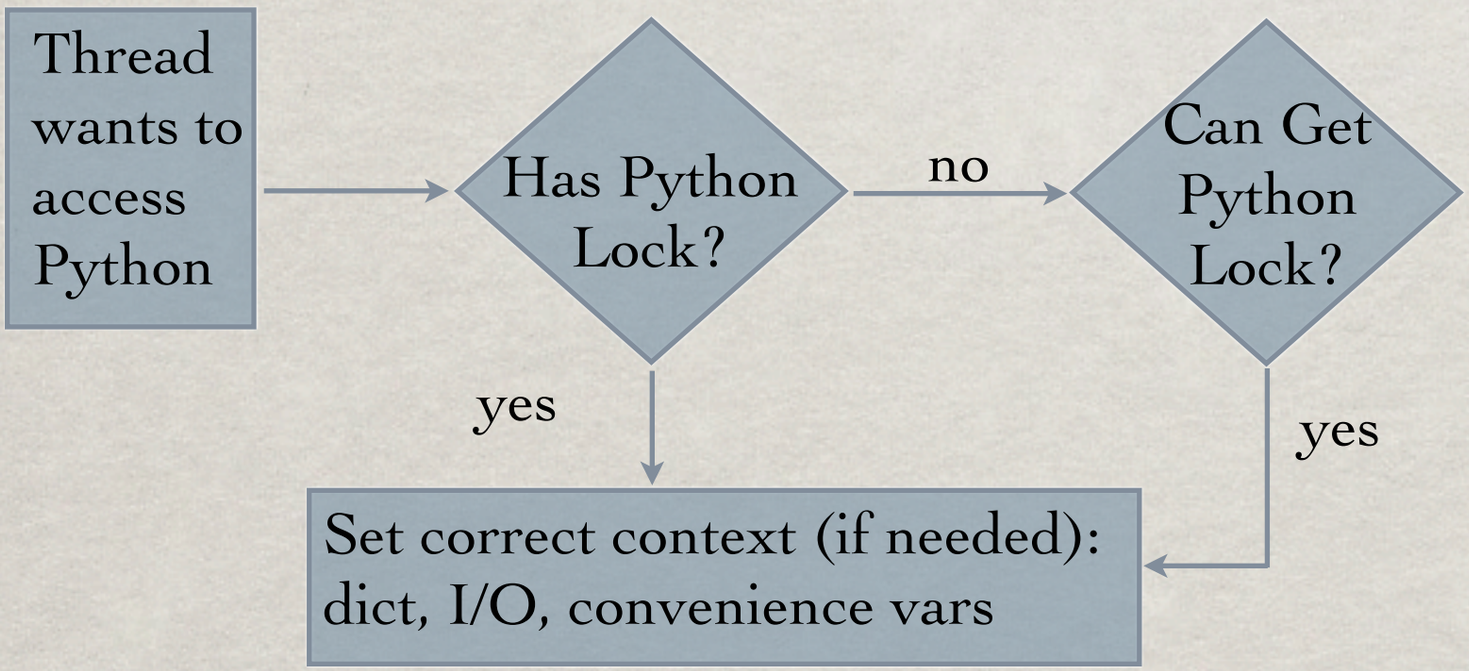
Possible Approaches: Write Our Own!

- Use a combination of pthread_mutex, pthread_cond_wait & our own Python lock:



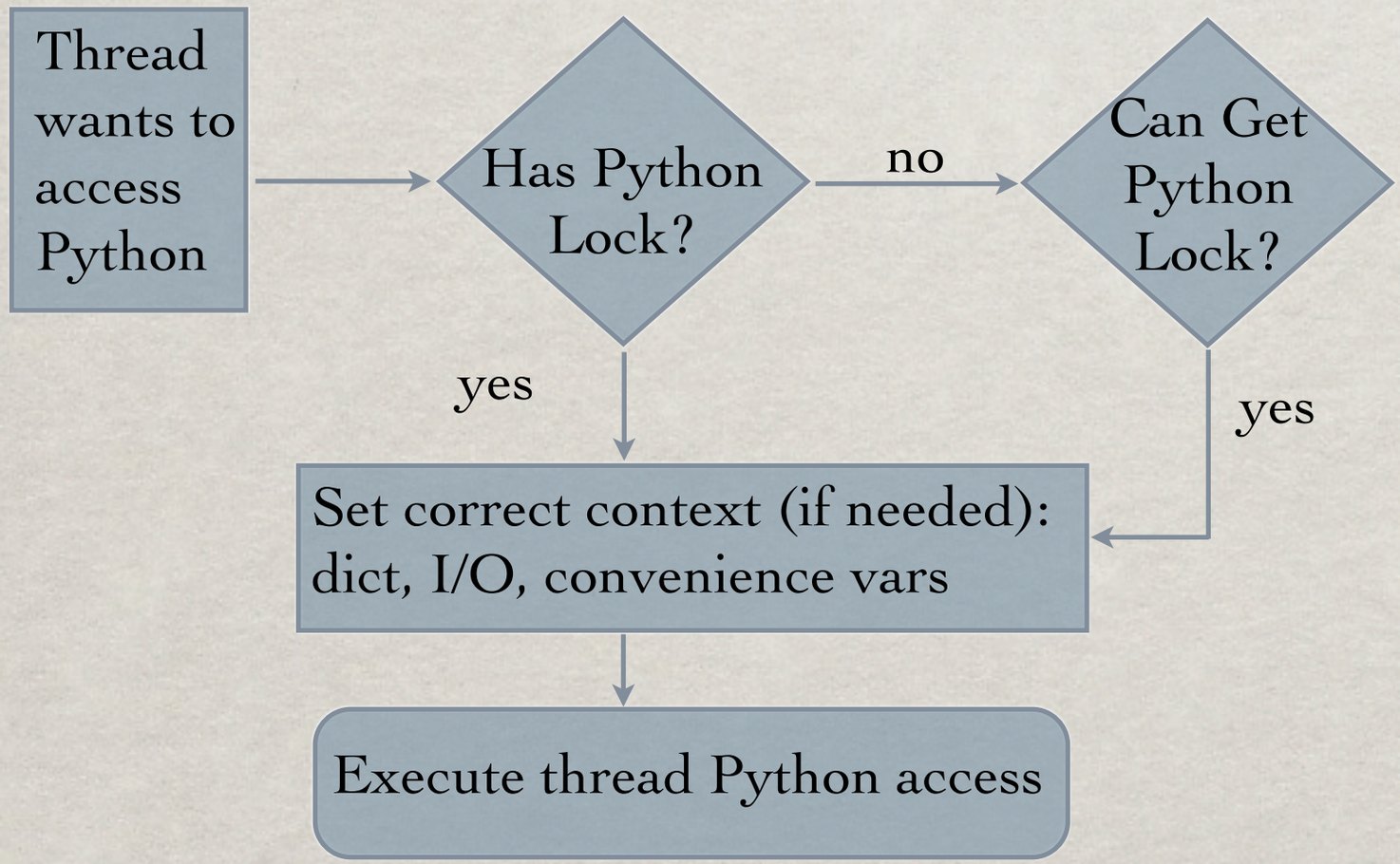
Possible Approaches: Write Our Own!

- Use a combination of pthread_mutex, pthread_cond_wait & our own Python lock:



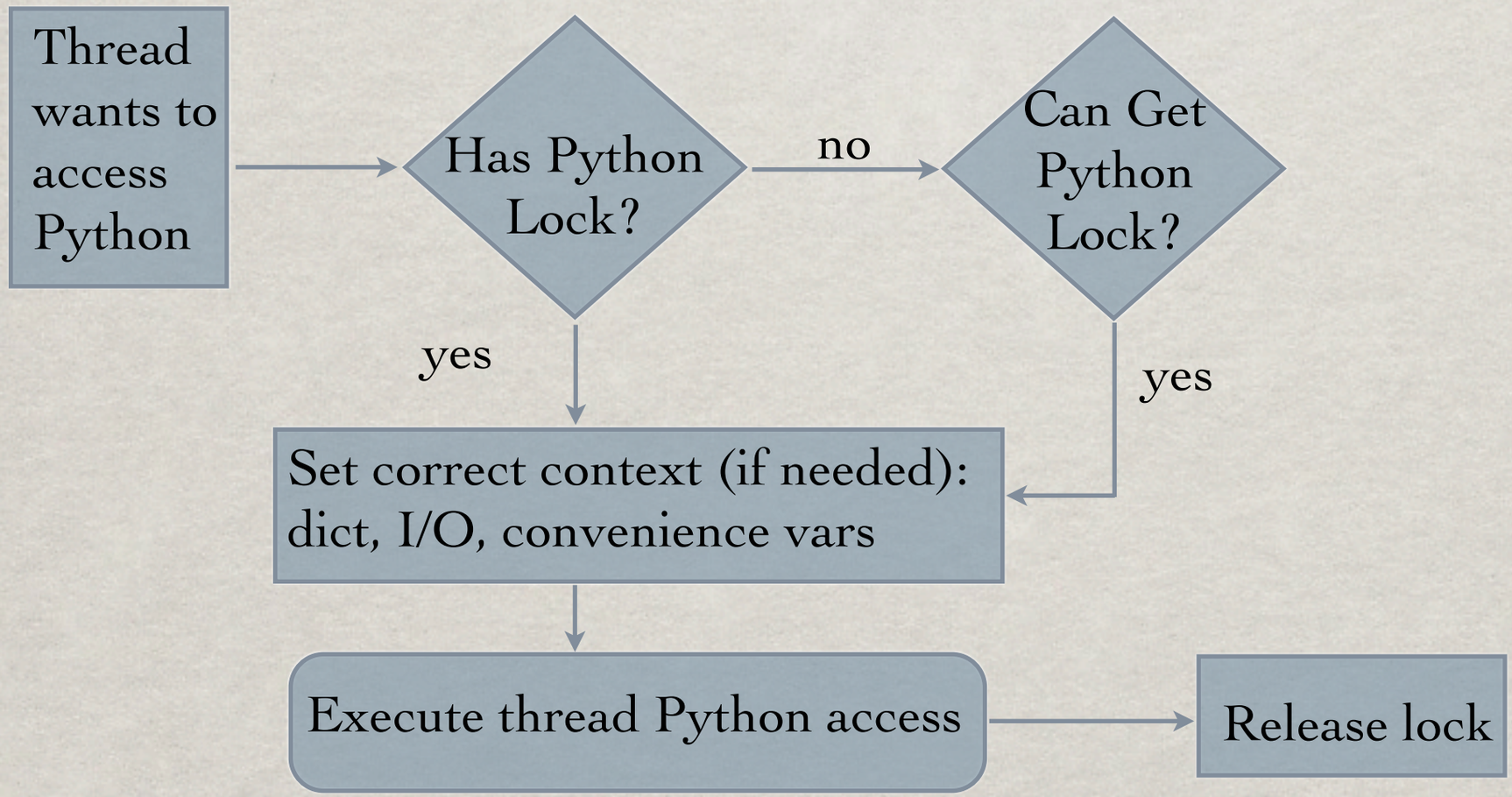
Possible Approaches: Write Our Own!

- Use a combination of pthread_mutex, pthread_cond_wait & our own Python lock:



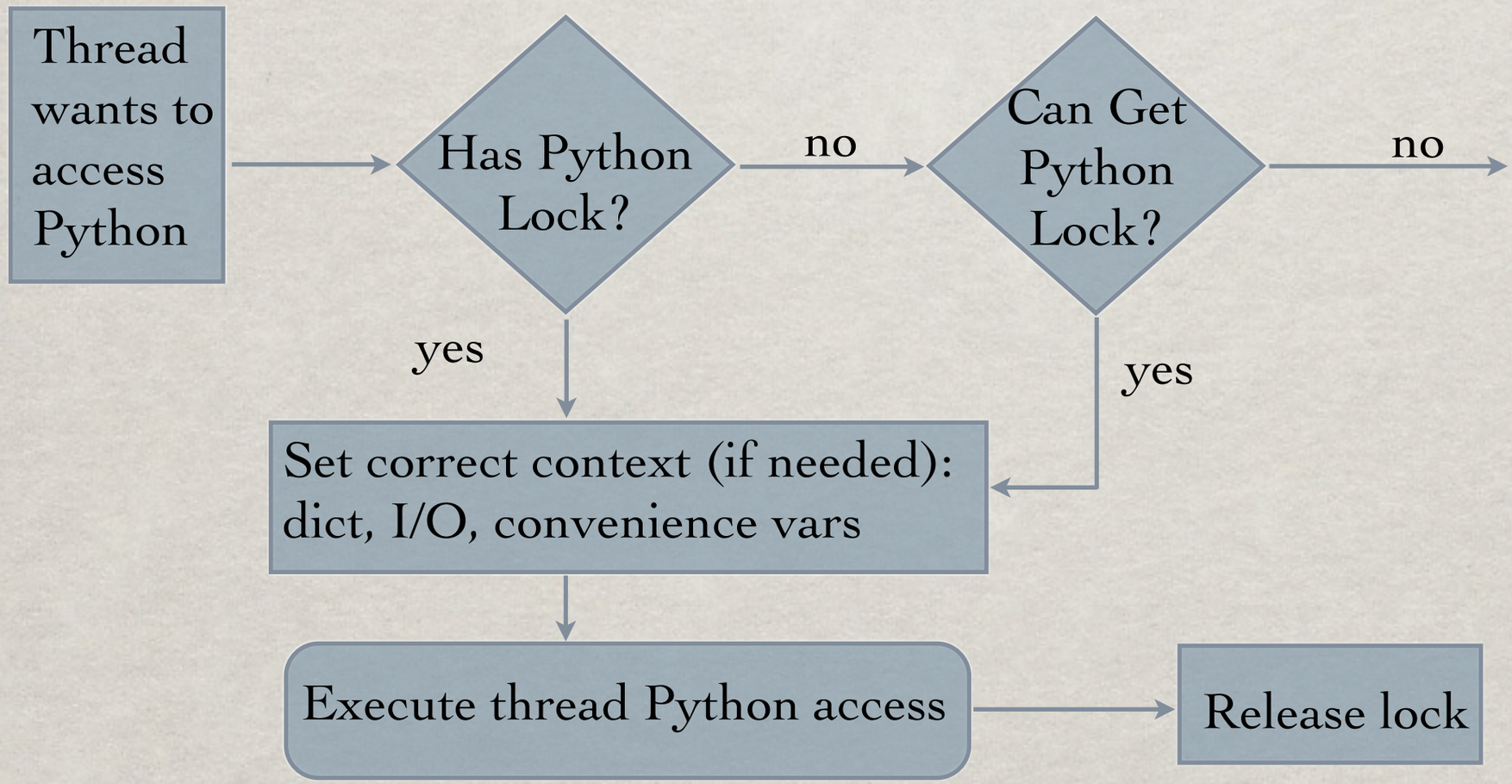
Possible Approaches: Write Our Own!

- Use a combination of pthread_mutex, pthread_cond_wait & our own Python lock:



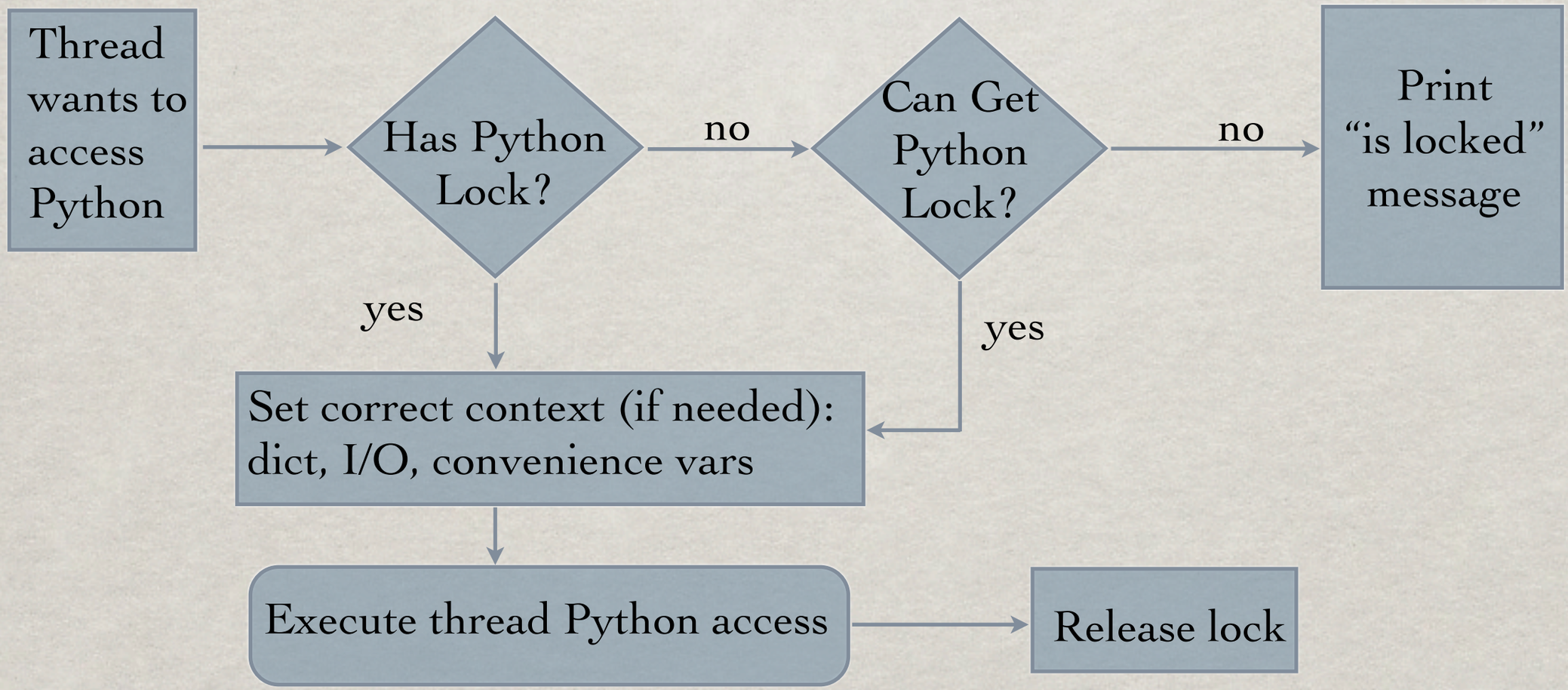
Possible Approaches: Write Our Own!

- Use a combination of pthread_mutex, pthread_cond_wait & our own Python lock:



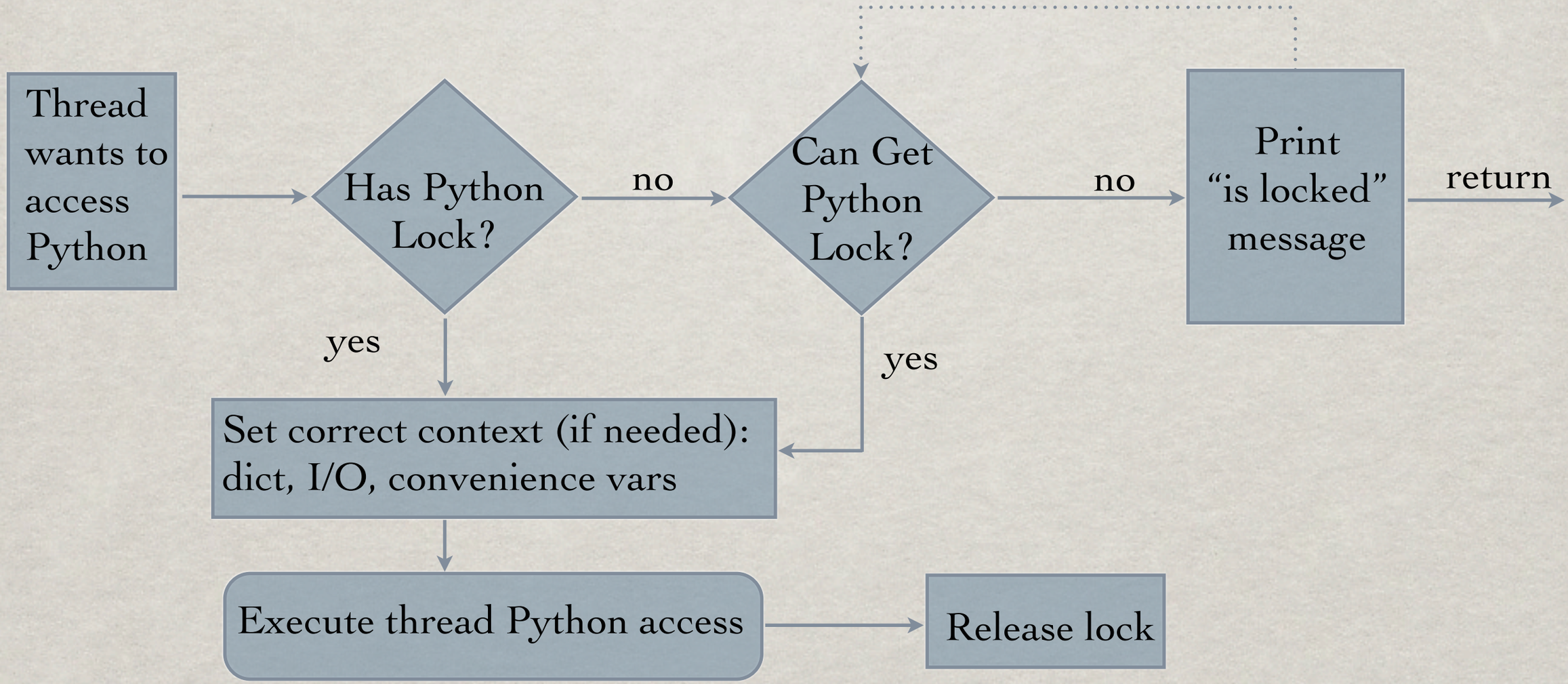
Possible Approaches: Write Our Own!

- Use a combination of pthread_mutex, pthread_cond_wait & our own Python lock:



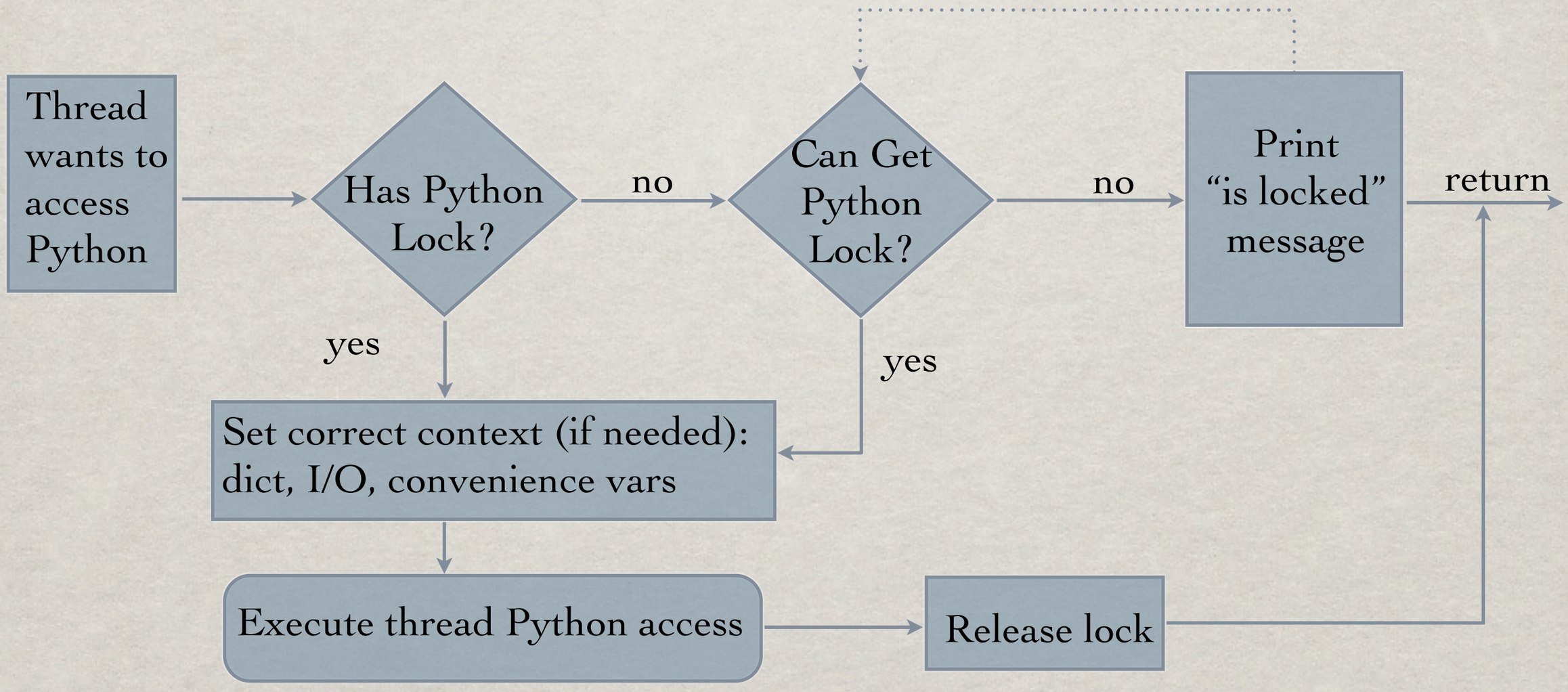
Possible Approaches: Write Our Own!

- Use a combination of pthread_mutex, pthread_cond_wait & our own Python lock:



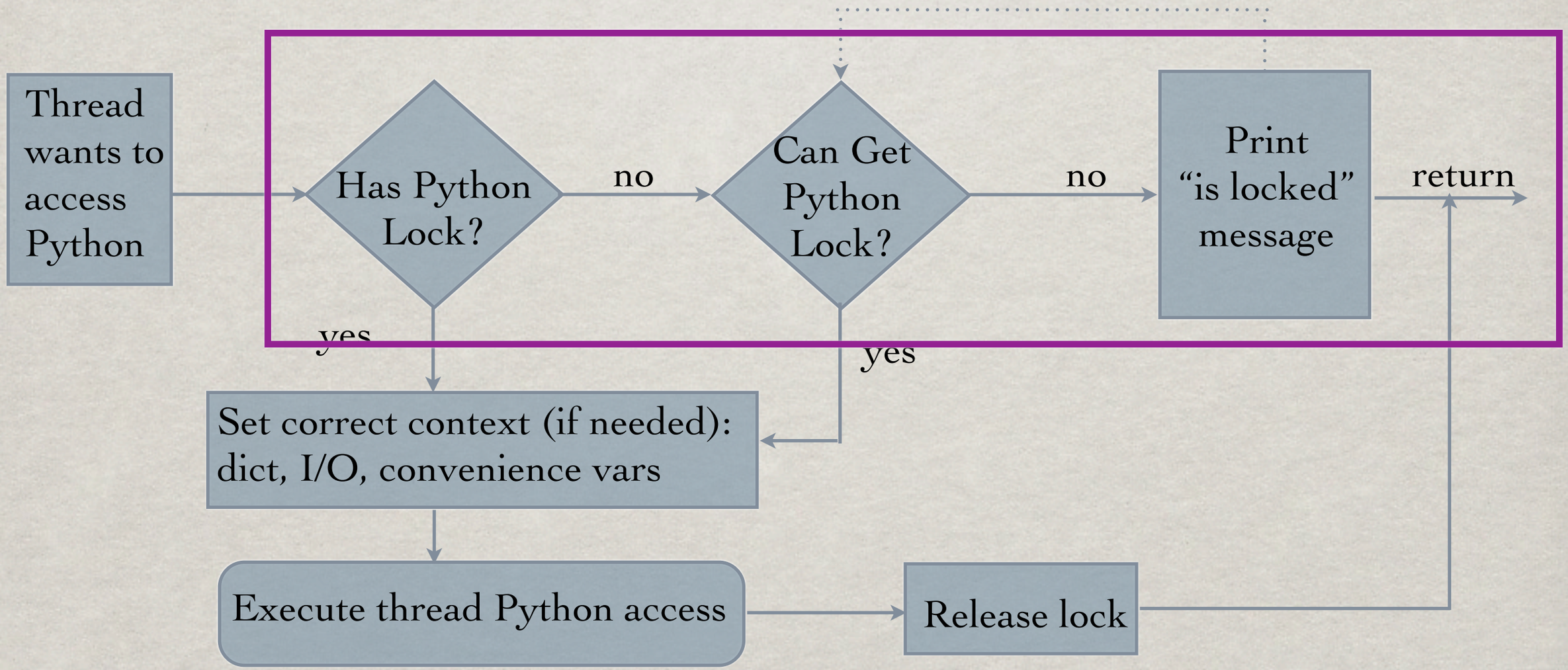
Possible Approaches: Write Our Own!

- Use a combination of pthread_mutex, pthread_cond_wait & our own Python lock:



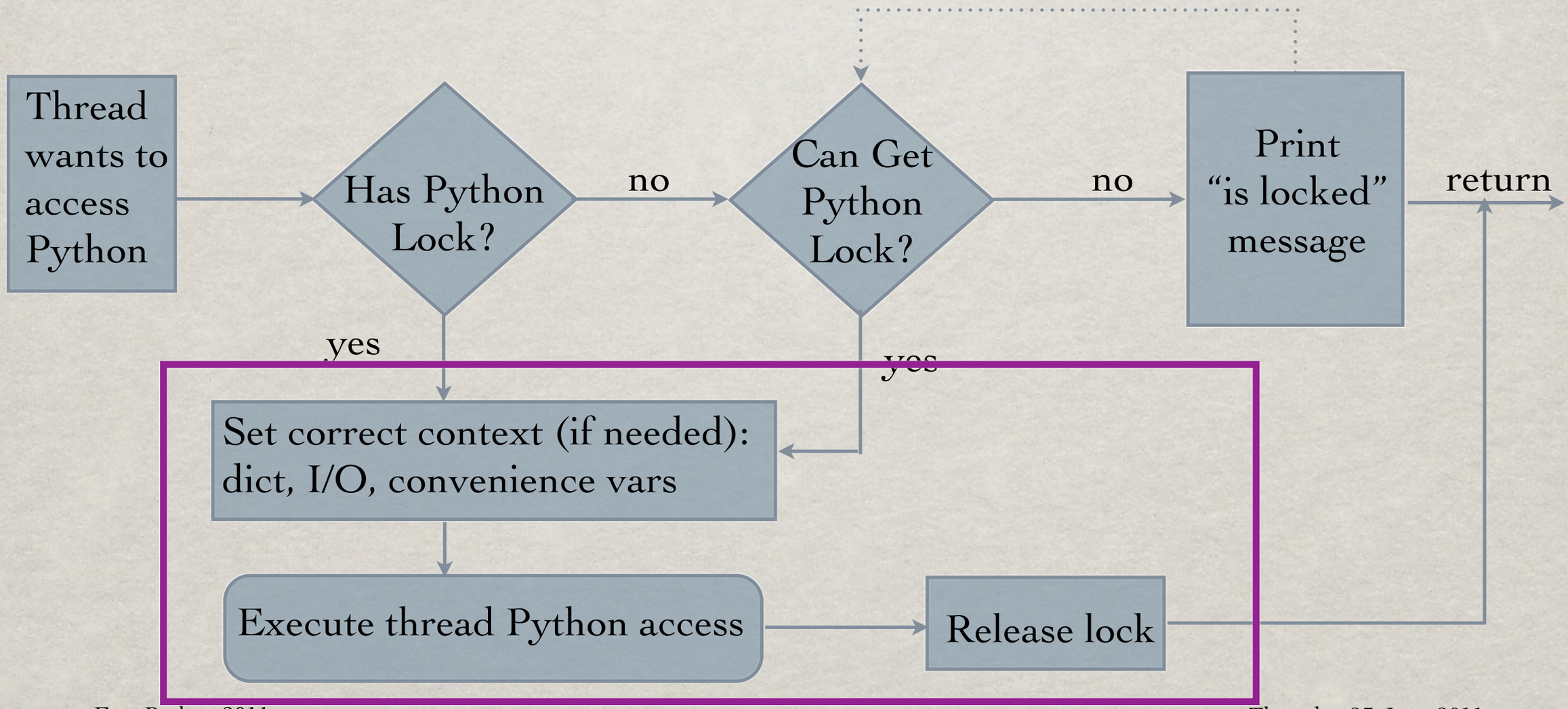
Possible Approaches: Write Our Own!

-Thread safety
-No deadlocking



Possible Approaches: Write Our Own!

-Correct dictionary
-No early release



GUI Debugger Python Requirements

- Multiple debugger sessions
  - Multiple Script Interpreters/Dictionaries
  - Ability to switch smoothly between them
  - Complete isolation between sessions
  - Thread safety
  - No deadlocking
  - True concurrency

GUI Debugger Python Requirements

- Multiple debugger sessions
  - ☑ Multiple Script Interpreters/Dictionaries
  - ☐ Ability to switch smoothly between them
  - ☐ Complete isolation between sessions
  - ☐ Thread safety
  - ☐ No deadlocking
  - ☐ True concurrency

GUI Debugger Python Requirements

- Multiple debugger sessions
  - ☑ Multiple Script Interpreters/Dictionaries
  - ☑ Ability to switch smoothly between them
  - ☐ Complete isolation between sessions
  - ☐ Thread safety
  - ☐ No deadlocking
  - ☐ True concurrency

GUI Debugger Python Requirements

- Multiple debugger sessions
  - ☑ Multiple Script Interpreters/Dictionaries
  - ☑ Ability to switch smoothly between them
  - ☑ Complete isolation between sessions
  - ☐ Thread safety
  - ☐ No deadlocking
  - ☐ True concurrency

GUI Debugger Python Requirements

- Multiple debugger sessions
  - ☑ Multiple Script Interpreters/Dictionaries
  - ☑ Ability to switch smoothly between them
  - ☑ Complete isolation between sessions
  - ☑ Thread safety
  - ☐ No deadlocking
  - ☐ True concurrency

GUI Debugger Python Requirements

- Multiple debugger sessions
  - ☑ Multiple Script Interpreters/Dictionaries
  - ☑ Ability to switch smoothly between them
  - ☑ Complete isolation between sessions
  - ☑ Thread safety
  - ☑ No deadlocking
  - ☐ True concurrency

GUI Debugger Python Requirements

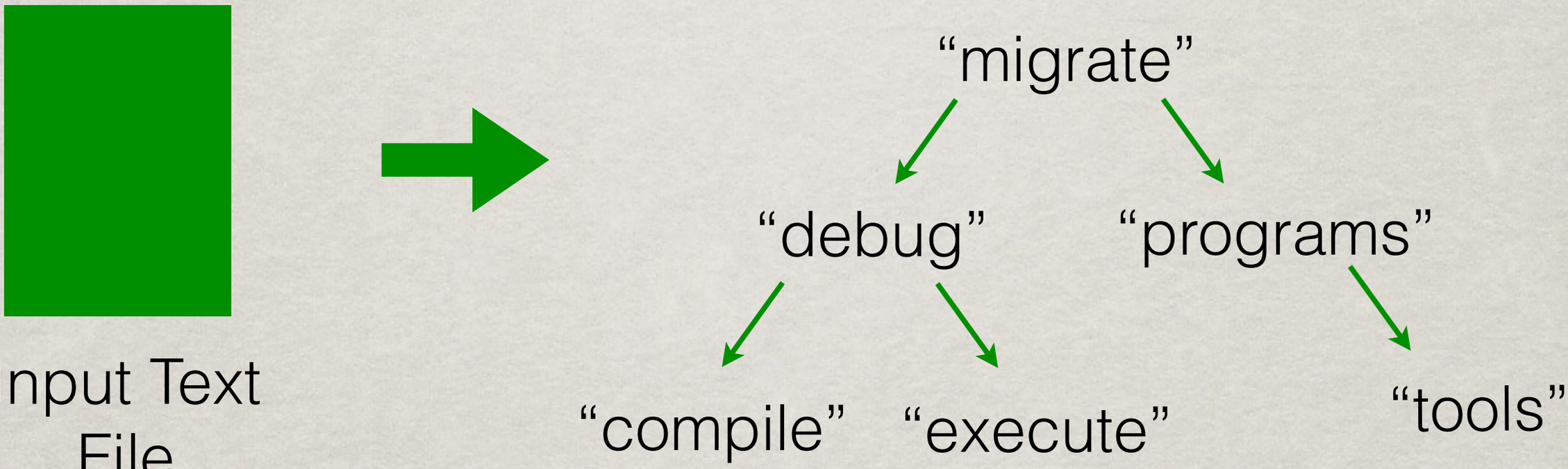
- Multiple debugger sessions
  - ☑ Multiple Script Interpreters/Dictionaries
  - ☑ Ability to switch smoothly between them
  - ☑ Complete isolation between sessions
  - ☑ Thread safety
  - ☑ No deadlocking
  - ☒ True concurrency

Outline

- What is LLDB?
- Python in LLDB
- Particular Problems (& Solutions)
- Example Using Python to Debug Problem
- Questions

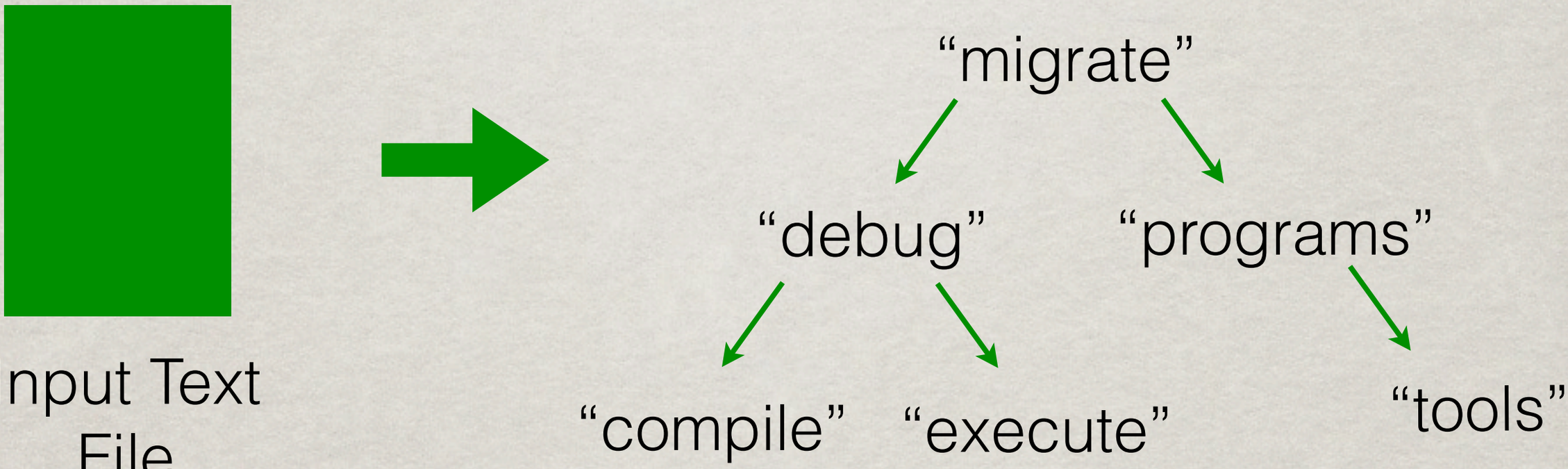
Example: Simple Dictionary Program

Store and find words in Binary Search Tree



Example: Simple Dictionary Program

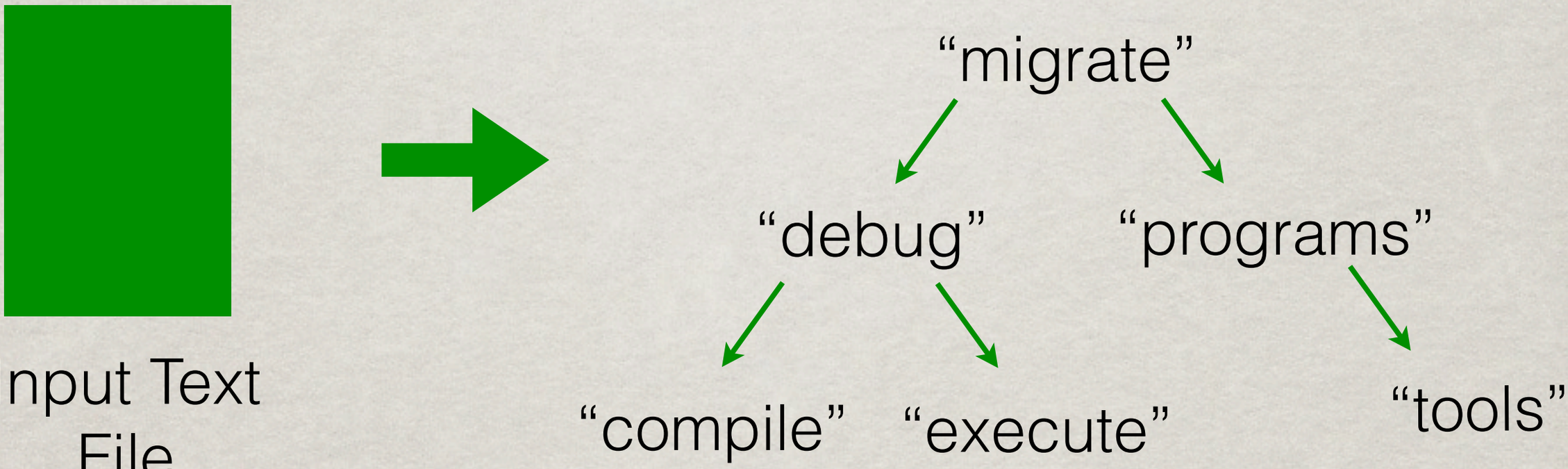
Store and find words in Binary Search Tree



Find ("tools") => Yes

Example: Simple Dictionary Program

Store and find words in Binary Search Tree



Find ("tools") => Yes
Find ("assemble") => No

Problem: Word Not Found in Dictionary

```
$  ./dictionary Romeo-and-Juliet.txt
Dictionary loaded.
Enter search word: love
Yes!
Enter search word: sun
Yes!
Enter search word: Romeo
No!
```

Problem: Word Not Found In Dictionary

- Possible causes for not finding word:
  - Word not inserted
  - Word inserted in unexpected location

- How to determine if word is in tree?
  - Traverse tree by hand?
    - Not practical: 100‘s or 1000’s of nodes!
  - Write a script!

The Plan

(Searching Tree Without Restarting Program)

- Write Depth-First Search (DFS) function in file (tree_utils.py)
- Attach to running program with LLDB
- Use interactive interpreter to call DFS on current tree
- DFS returns root-to-node path, if found

```
def DFS (root, word, cur_path):
    root_word_ptr = root.GetChildMemberWithName ("word");
    left_child_ptr = root.GetChildMemberWithName ("left");
    right_child_ptr = root.GetChildMemberWithName ("right");
    root_word = root_word_ptr.GetSummary()

    if root_word == word:
        return cur_path
    elif word < root_word:
        if left_child_ptr.GetValue() == None:
            return ""
        else:
            cur_path = cur_path + "L"
            return DFS (left_child_ptr, word, cur_path)
    else:
        if right_child_ptr.GetValue() == None:
            return ""
        else:
            cur_path = cur_path + "R"
            return DFS (right_child_ptr, word, cur_path)
```

```
def DFS (root, word, cur_path):
    root_word_ptr = root.GetChildMemberWithName ("word");
    left_child_ptr = root.GetChildMemberWithName ("left");
    right_child_ptr = root.GetChildMemberWithName ("right");
    root_word = root_word_ptr.GetSummary()

    if root_word == word:
        return cur_path
    elif word < root_word:
        if left_child_ptr.GetValue() == None:
            return ""
        else:
            cur_path = cur_path + "L"
            return DFS (left_child_ptr, word, cur_path)
    else:
        if right_child_ptr.GetValue() == None:
            return ""
        else:
            cur_path = cur_path + "R"
            return DFS (right_child_ptr, word, cur_path)
```

```
def DFS (root, word, cur_path):
    root_word_ptr = root.GetChildMemberWithName ("word");
    left_child_ptr = root.GetChildMemberWithName ("left");
    right_child_ptr = root.GetChildMemberWithName ("right");
    root_word = root_word_ptr.GetSummary()

    if root_word == word:
        return cur_path
    elif word < root_word:
        if left_child_ptr.GetValue() == None:
            return ""
        else:
            cur_path = cur_path + "L"
            return DFS (left_child_ptr, word, cur_path)
    else:
        if right_child_ptr.GetValue() == None:
            return ""
        else:
            cur_path = cur_path + "R"
            return DFS (right_child_ptr, word, cur_path)
```

```
def DFS (root, word, cur_path):
    root_word_ptr = root.GetChildMemberWithName ("word");
    left_child_ptr = root.GetChildMemberWithName ("left");
    right_child_ptr = root.GetChildMemberWithName ("right");
    root_word = root_word_ptr.GetSummary()

    if root_word == word:
        return cur_path
    elif word < root_word:
        if left_child_ptr.GetValue() == None:
            return ""
        else:
            cur_path = cur_path + "L"
            return DFS (left_child_ptr, word, cur_path)
    else:
        if right_child_ptr.GetValue() == None:
            return ""
        else:
            cur_path = cur_path + "R"
            return DFS (right_child_ptr, word, cur_path)
```

```
def DFS (root, word, cur_path):
    root_word_ptr = root.GetChildMemberWithName ("word");
    left_child_ptr = root.GetChildMemberWithName ("left");
    right_child_ptr = root.GetChildMemberWithName ("right");
    root_word = root_word_ptr.GetSummary()

    if root_word == word:
        return cur_path
    elif word < root_word:
        if left_child_ptr.GetValue() == None:
            return ""
        else:
            cur_path = cur_path + "L"
            return DFS (left_child_ptr, word, cur_path)
    else:
        if right_child_ptr.GetValue() == None:
            return ""
        else:
            cur_path = cur_path + "R"
            return DFS (right_child_ptr, word, cur_path)
```

LLDB API Functions

```
def DFS (root, word, cur_path):
    root_word_ptr = root.GetChildMemberWithName ("word");
    left_child_ptr = root.GetChildMemberWithName ("left");
    right_child_ptr = root.GetChildMemberWithName ("right");
    root_word = root_word_ptr.GetSummary()

    if root_word == word:
        return cur_path
    elif word < root_word:
        if left_child_ptr.GetValue() == None:
            return ""
        else:
            cur_path = cur_path + "L"
            return DFS (left_child_ptr, word, cur_path)
    else:
        if right_child_ptr.GetValue() == None:
            return ""
        else:
            cur_path = cur_path + "R"
            return DFS (right_child_ptr, word, cur_path)
```

Using The Interactive Interpreter

```
(lldb) process attach --name dictionary
Process 397 stopped
(lldb)
```

Using The Interactive Interpreter

```
(lldb) process attach --name dictionary
Process 397 stopped
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()'
or Ctrl-D.
>>>
```

Using The Interactive Interpreter

```
(lldb) process attach --name dictionary
Process 397 stopped
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()'
or Ctrl-D.
>>> import tree_utils
>>>
```

Using The Interactive Interpreter

```
(lldb) process attach --name dictionary
Process 397 stopped
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()'
or Ctrl-D.
>>> import tree_utils
>>> root = lldb.frame.FindVariable ("dictionary")
>>>
```

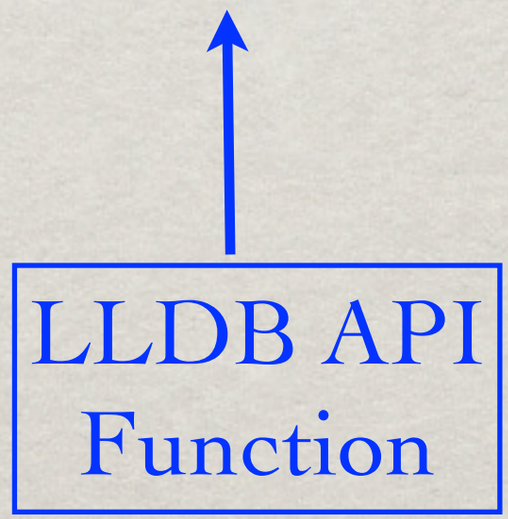
Using The Interactive Interpreter

```
(lldb) process attach --name dictionary
Process 397 stopped
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()'
or Ctrl-D.
>>> import tree_utils
>>> root = lldb.frame.FindVariable ("dictionary")
>>>
```

LLDB Convenience Variable

Using The Interactive Interpreter

```
(lldb) process attach --name dictionary
Process 397 stopped
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()'
or Ctrl-D.
>>> import tree_utils
>>> root = lldb.frame.FindVariable ("dictionary")
>>>
```



Using The Interactive Interpreter

```
(lldb) process attach --name dictionary
Process 397 stopped
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()'
or Ctrl-D.
>>> import tree_utils
>>> root = lldb.frame.FindVariable ("dictionary")
>>>
```

Using The Interactive Interpreter

```
(lldb) process attach --name dictionary
Process 397 stopped
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()'
or Ctrl-D.
>>> import tree_utils
>>> root = lldb.frame.FindVariable ("dictionary")
>>> cur_path = ""
>>>
```

Using The Interactive Interpreter

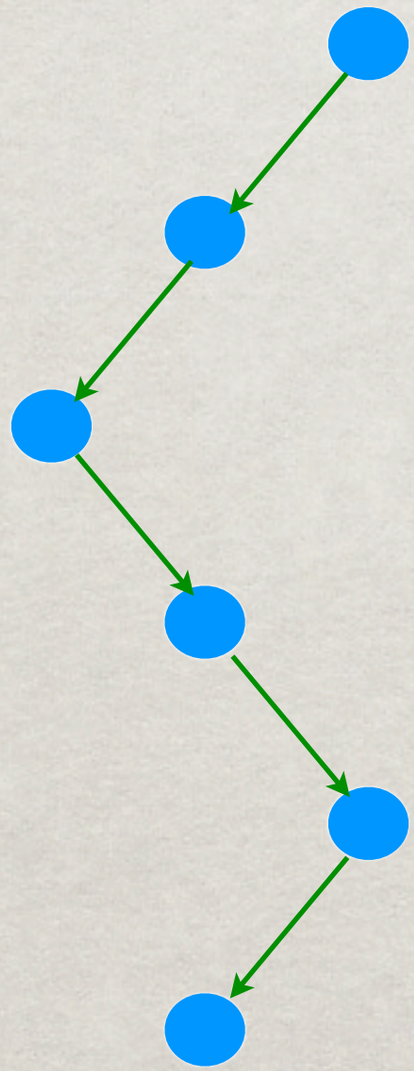
```
(lldb) process attach --name dictionary
Process 397 stopped
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()'
or Ctrl-D.
>>> import tree_utils
>>> root = lldb.frame.FindVariable ("dictionary")
>>> cur_path = ""
>>> path = tree_utils.DFS (root, "Romeo", cur_path)
>>>
```

Using The Interactive Interpreter

```
(lldb) process attach --name dictionary
Process 397 stopped
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()'
or Ctrl-D.
>>> import tree_utils
>>> root = lldb.frame.FindVariable ("dictionary")
>>> cur_path = ""
>>> path = tree_utils.DFS (root, "Romeo", cur_path)
>>> print path
LLRRL
>>> ^D
(lldb)
```

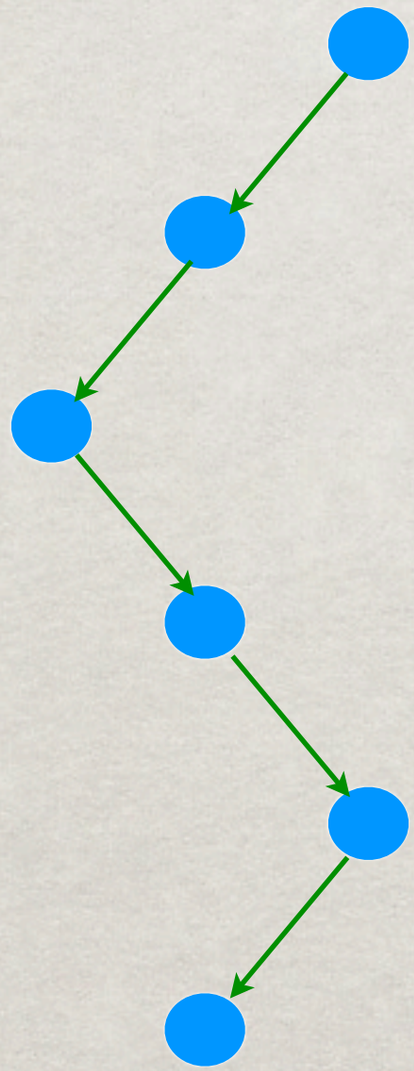
We're Halfway There!

Path: LLRRL



We're Halfway There!

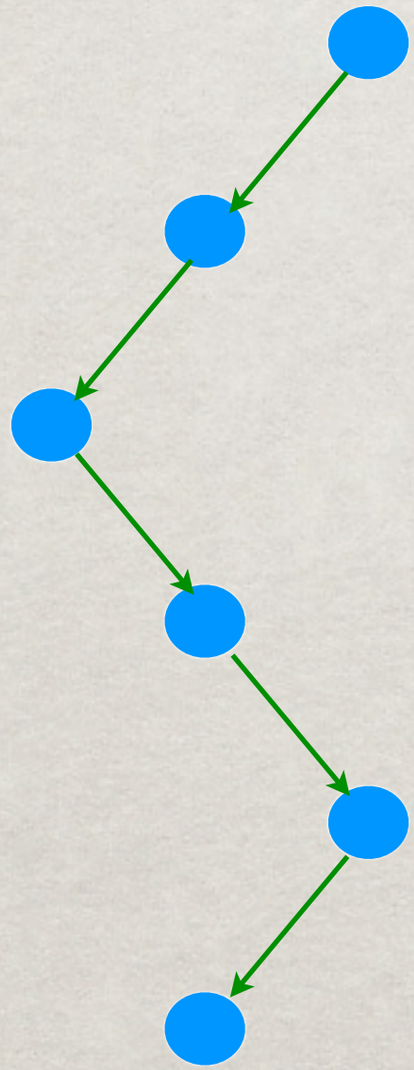
Path: LLRRL



- WE found it...why didn't the program?

We're Halfway There!

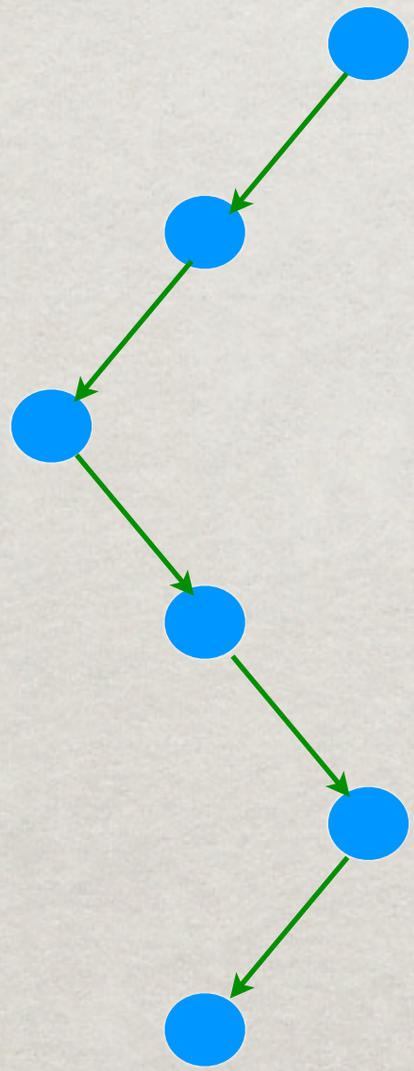
Path: LLRRL



- WE found it...why didn't the program?
- How to find the problem?

We're Halfway There!

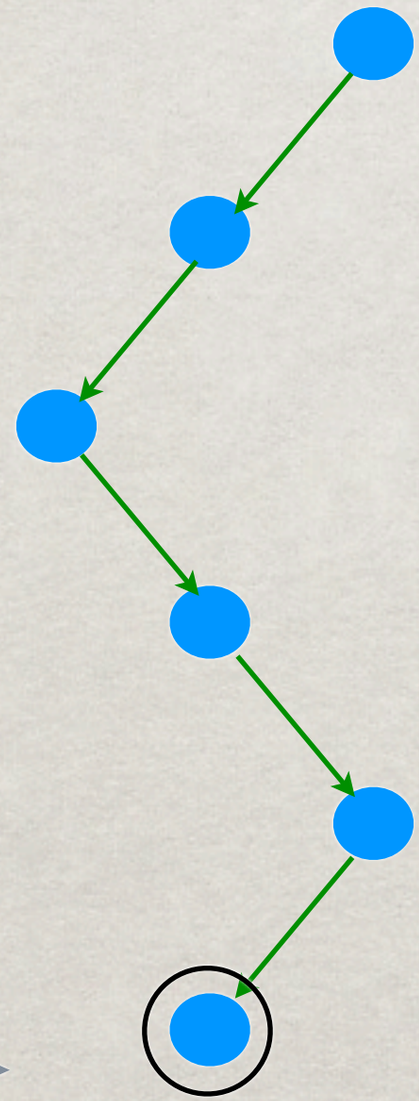
Path: LLRRL



- WE found it...why didn't the program?
- How to find the problem?
- Breakpoint command scripts!

We're Halfway There!

Path: LLRRL

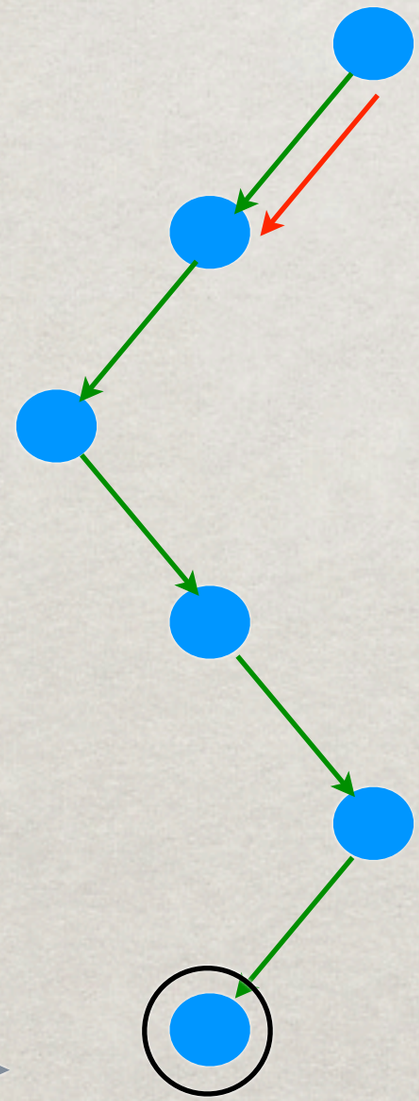


Our word

- WE found it...why didn't the program?
- How to find the problem?
- Breakpoint command scripts!

We're Halfway There!

Path: LLRRL

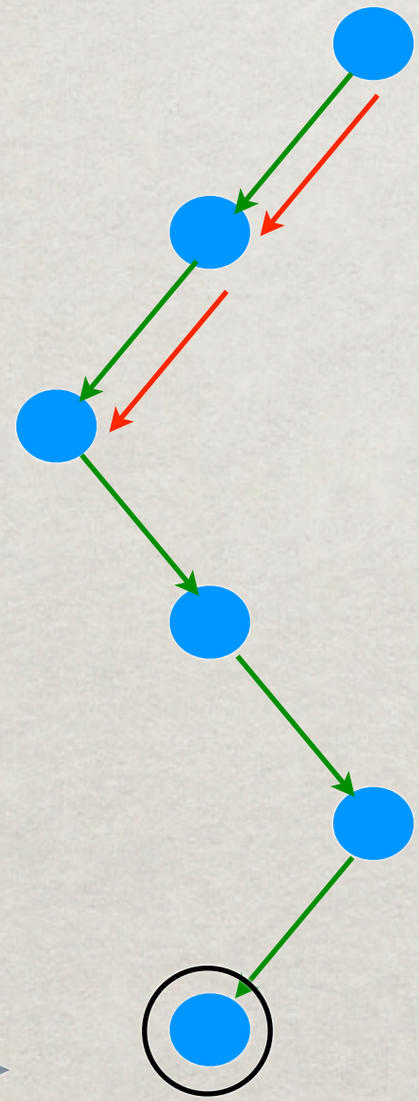


Our word

- WE found it...why didn't the program?
- How to find the problem?
- Breakpoint command scripts!

We're Halfway There!

Path: LLRRL

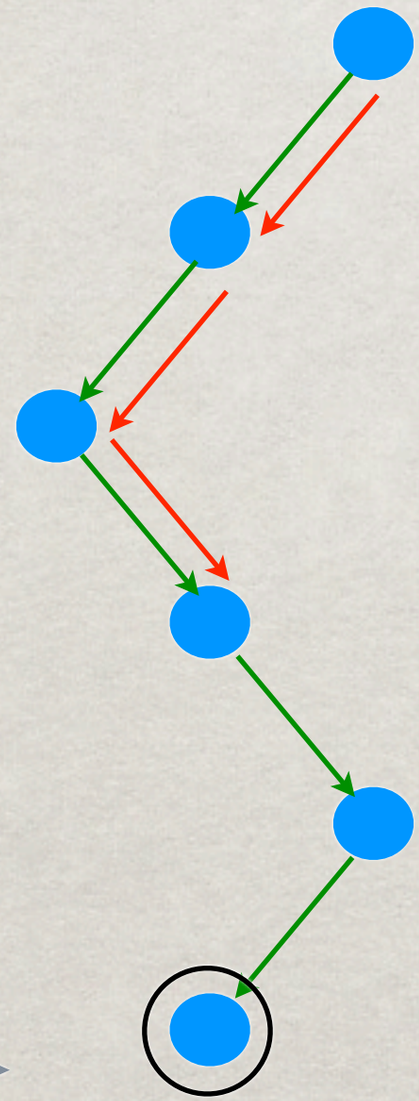


Our word

- WE found it...why didn't the program?
- How to find the problem?
- Breakpoint command scripts!

We're Halfway There!

Path: LLRRL

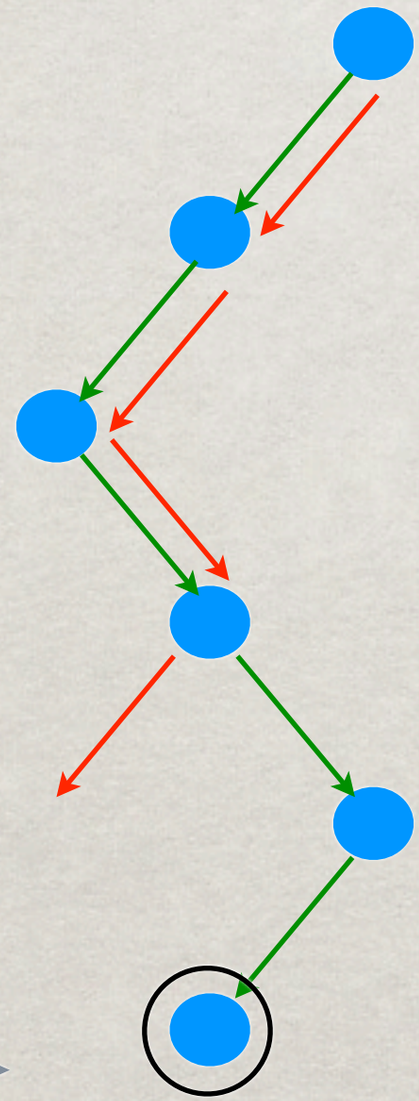


Our word

- WE found it...why didn't the program?
- How to find the problem?
- Breakpoint command scripts!

We're Halfway There!

Path: LLRRL

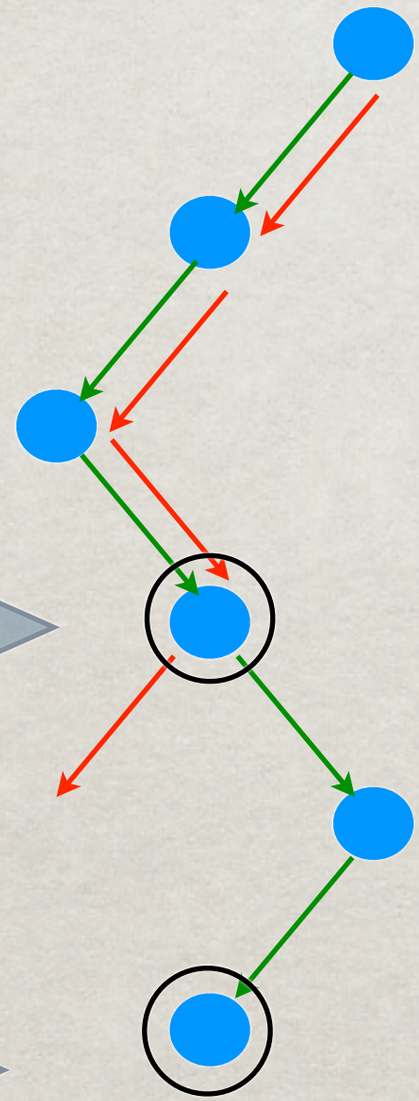


Our word

- WE found it...why didn't the program?
- How to find the problem?
- Breakpoint command scripts!

We're Halfway There!

Path: LLRRL



The problem

Our word

- WE found it...why didn't the program?
- How to find the problem?
- Breakpoint command scripts!

Python Breakpoint Command
(At Decision to Follow Right Child)

```
global path
if (path[0] == 'R'):
    path = path[1:]
    thread = frame.GetThread()
    process = thread.GetProcess()
    process.Continue()
else:
    print "Going right; should go left!"
```

Python Breakpoint Command
(At Decision to Follow Right Child)

```
def some_obscure_function_name (frame, bp_loc):
    global path
    if (path[0] == 'R'):
        path = path[1:]
        thread = frame.GetThread()
        process = thread.GetProcess()
        process.Continue()
    else:
        print "Going right; should go left!"
```

LLDB Convenience Variables

Python Breakpoint Command
(At Decision to Follow Right Child)

```
global path
if (path[0] == 'R'):
    path = path[1:]
    thread = frame.GetThread()
    process = thread.GetProcess()
    process.Continue()
else:
    print "Going right; should go left!"
```

Python Breakpoint Command
(At Decision to Follow Right Child)

```
global path
if (path[0] == 'R'):
    path = path[1:]
    thread = frame.GetThread()
    process = thread.GetProcess()
    process.Continue()
else:
    print "Going right; should go left!"
```

Python Breakpoint Command
(At Decision to Follow Right Child)

```
global path
if (path[0] == 'R'):
    path = path[1:]
    thread = frame.GetThread()
    process = thread.GetProcess()
    process.Continue()
else:
    print "Going right; should go left!"
```

LLDB Convenience Variable

Python Breakpoint Command
(At Decision to Follow Right Child)

```
global path
if (path[0] == 'R'):
    path = path[1:]
    thread = frame.GetThread()
    process = thread.GetProcess()
    process.Continue()
else:
    print "Going right; should go left!"
```

LLDB API Functions

Results!

(lldb)

Results!

```
(lldb) breakpoint command add -s python 1
...
(lldb)
```

Results!

```
(lldb) breakpoint command add -s python 1
...
(lldb) breakpoint command add -s python 2
...
(lldb)
```

Results!

```
(lldb) breakpoint command add -s python 1
...
(lldb) breakpoint command add -s python 2
...
(lldb) continue

Going right; should go left!
Process 236 stopped
...
(lldb)
```

Results!

```
(lldb) breakpoint command add -s python 1
...
(lldb) breakpoint command add -s python 2
...
(lldb) continue

Going right; should go left!
Process 236 stopped
...
(lldb) expr root->word
(const char *) $0 = “dramatis”
(lldb)
```

Results!

```
(lldb) breakpoint command add -s python 1
...
(lldb) breakpoint command add -s python 2
...
(lldb) continue

Going right; should go left!
Process 236 stopped
...
(lldb) expr root->word
(const char *) $0 = “dramatis”
(lldb) expr search_word
(char *) $1 = “romeo”
(lldb)
```

Results!

(lldb) breakpoint command add -s python 1
...
(lldb) breakpoint command add -s python 2
...
(lldb) continue

Going right; should go left!
Process 236 stopped
...
(lldb) expr root->word
(const char *) $0 = “dramatis”
(lldb) expr search_word
(char *) $1 = “romeo”
(lldb) script print path
LLRRL
(lldb)

Results!

```
(lldb) breakpoint command add -s python 1
...
(lldb) breakpoint command add -s python 2
...
(lldb) continue

Going right; should go left!
Process 236 stopped
...
(lldb) expr root->word
(const char *) $0 = “dramatis”
(lldb) expr search_word
(char *) $1 = “romeo”
(lldb) script print path
LLRRL
(lldb) expr root->left->left->right->right->left->word
(const char *) $2 = “Romeo”
(lldb)
```

Results!

```
(lldb) breakpoint command add -s python 1
...
(lldb) breakpoint command add -s python 2
...
(lldb) continue

Going right; should go left!
Process 236 stopped
...
(lldb) expr root->word
(const char *) $0 = “dramatis”
(lldb) expr search_word
(char *) $1 = “romeo”
(lldb) script print path
LLRRL
(lldb) expr root->left->left->right->right->left->word
(const char *) $2 = “Romeo”
(lldb)
```

Case conversion problem!

Summary

- Embedding Python in a debugger gives users great power
- Passing pointers/objects can be done via API tweaks & integer passing
- Maintaining separate session dictionaries allowed persistence & multi-interpreter emulation

For Further Reference

- The LLDB website: http://lldb.llvm.org
  - Full project description
  - Download code (it's open source!)
  - Scripting example with explanations
    - http://lldb.llvm.org/scripting.html
  - Developer's Mailing List

Questions/Comments!

Caroline Tice
ctice42@gmail.com