Migrating from GDB to LLDB

Introduction to the LLDB command line

Session 321

Jim Ingham
Senior Debugger Engineer

Talk Outline

- Introduction to the LLDB command line:
    - Basic syntax
    - Command objects
    - Command aliases
- Power user features:
    - Making use of the expression parser
        - Programmatic data introspection
    - Making use of LLDB’s Python bindings
        - Automate complex debugging tasks

What is the LLDB Project?

- A modern replacement for GDB
- A part of the LLVM project
  - Open source
  - So far most of the work was done by Apple
  - http://lldb.llvm.org
- Makes use of the clang parser for type system and expression evaluation
- Very efficient handling of debug info (incremental DWARF parser)
  - Faster startup times, lower memory usage
- Threads are first class citizens
- Powerful scripting component (using Python)

What is LLDB?

- A system “debugger library”
  - For use in Xcode
  - For use in other tools
    - Python bindings make it a do-it-yourself debugger app builder
- A command-line debugger
  - Available as Terminal tool or in Xcode Console Window
  - Quicker access to particular pieces of information
  - The console log provides a history trace

Console LLDB

```
localhost>  ./lldb Sketch.app
Current executable set to '/tmp/Sketch.app/' (x86_64).
(lldb)  b alignLeftEdges:
breakpoint set --name 'alignLeftEdges:'
Breakpoint created: 1: name = 'alignLeftEdges:', locations = 1
(lldb)  run
Process 16704 launched: '/tmp/Sketch.app/Contents/MacOS/
Sketch' (x86_64)
...
```

Console LLDB

```
Process 16704 stopped
* thread #1: SKTGraphicView.m:1405, stop reason = breakpoint 1.1
  frame #0: 0x0000000100017b77 SKTGraphicView.m:1405
   1402
   1403
   1404 - (IBAction)alignLeftEdges:(id)sender {
-> 1405     NSArray *selection = [self selectedGraphics];
   1406     NSUInteger i, c = [selection count];
   1407     if (c > 1) {
   1408         NSRect firstBounds = [[selection objectAtIndex:0] bounds];
(lldb)  po self
(SKTGraphicView *) $1 = 0x0000000102115580 <SKTGraphicView: 0x102115580>
(lldb)  n
Process 16704 stopped
```

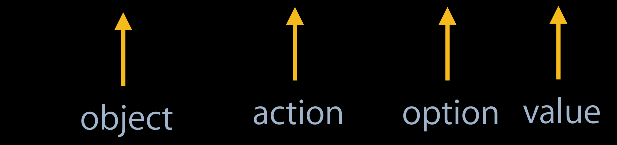
LLDB Command Syntax

- “GDB-like” commands which are very concise, but irregular
    - Fast to type for day to day use
    - If that was all, it would be hard to learn
- An underlying command language that is more explicit
    - Basic commands are regular and well structured
        - Easy to learn and discover new features
        - More consistency across commands
    - Powerful alias facility to create the “GDB-like” commands
- This talk will focus more on LLDB: for GDB -> LLDB:
    - http://lldb.llvm.org/tutorial.html

Basic Syntax

- Commands are in the form:
  - object action [options] [arguments]

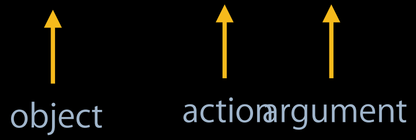
```
breakpoint set --name main
```



Basic Syntax

- Commands are in the form:
    - object action [options] [arguments]

```
breakpoint set --name main
breakpoint delete 5
```



object

action

argument

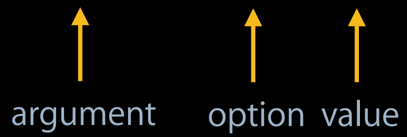
Basic Syntax

- Commands are in the form:
  - object action [options] [arguments]

```
breakpoint set --name main
breakpoint delete 5
```

  - Options have short and long form, can appear anywhere

```
target create MyApp.app -a i386
```



argument option value

Basic Syntax

- Commands are in the form:
  - object action [options] [arguments]

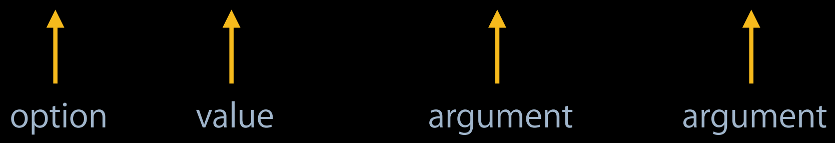
```
breakpoint set --name main
breakpoint delete 5
```

  - Options have short and long form, can appear anywhere

```
target create MyApp.app -a i386
```

  - “--” ends options (useful if arguments start with “-”)

```
process launch --working-dir /tmp -- -run-arg-1 -run-arg-2
```



Basic Syntax

- Commands are in the form:
  - object action [options] [arguments]

    ```
    breakpoint set --name main
    breakpoint delete 5
    ```

  - Options have short and long form, can appear anywhere

    ```
    target create MyApp.app -a i386
    ```

  - "`--`" ends options (useful if arguments start with "-")

    ```
    process launch --working-dir /tmp -- -run-arg-1 -run-arg-2
    ```

  - Words are white-space separated
    - Use quotes to protect spaces, "\" to protect quotes.
  - Some commands are "unparsed" after the end of options:
    - "`expression`" and "`script`"

Basic Syntax

- We favor option/value over arguments
  - Easier to document
  - Reduce dependency on “argument order”
  - More powerful auto-completion (e.g. scoped by other options):

    ```
    breakpoint set --shlibs MyApp --name ma<TAB>
    ```

    - Looks for completions only in MyApp of symbols by name
- And of course we do shortest unique match, so you can also type:

  ```
  br s -s MyApp -n ma<TAB>
  ```

Help

- “help” command for detailed explanation of command/subcommand

```
(lldb) help breakpoint delete
   Delete the specified breakpoint(s).  If no breakpoints are specified,
delete them all.

Syntax: breakpoint delete [<breakpt-id | breakpt-id-list>]
```

- Also give help on argument types:

```
(lldb) help breakpt-id
<breakpt-id> -- Breakpoint ID's consist major and minor numbers...
```

- “apropos” does help search:

```
(lldb) apropos delete
The following commands may relate to 'delete':
breakpoint command delete -- Delete the set of commands from a breakpoint.
```

- Command completion works in help...

LLDB Command Objects

- Represented by top level commands

  `target, thread, breakpoint...`

- Sometimes two words

  `target modules`

  `breakpoint commands`

LLDB Command Objects

- In some cases, many objects exist of the same sort
  - One process has many threads...
  - “`list`” will always list the instances available, e.g.

    ```
    thread list
    ```

  - “`select`” will focus on one instance

    ```
    thread select 1
    ```

  - Auto-selected when that makes sense
    - e.g., if you stop at a breakpoint, process, thread and frame are set
  - Some object are contained in others (frame in thread)
    - Selecting a thread sets the context for selecting a frame…

LLDB Command Objects

- The object/action form makes it easy to find commands
- For example, how do you do a backtrace?
  - Break it into an object and an action
  - First figure out which object would be responsible
  - For backtrace, threads have stack frames, so try “`thread`”
  - Then use the <TAB> completion to find the action:

```
(lldb) thread <TAB>
      Available completions:
      backtrace
      continue
      ...
```

  - Finally, “`help`” will give you the full syntax

Brief Tour of Objects—Target

- Specifies a particular debuggable program

```
target create MyApp.app --arch x86_64
```

- More than one target is allowed, "`target select`" to switch
- Breakpoints are specific to the target
- The target holds the shared modules loaded into your program
    - "`target modules` " is the object

      `target modules list` - lists the shared libraries loaded in the program

      `target modules lookup --symbol printf` - looks up symbols

Brief Tour of Objects—Process

- Specifies a running instance of a target

```
process launch
process attach
```

- Only one process per target (so no “`select`” or “`list`”)
- Gives you control over the life-cycle of the process:

  `process continue` - continues the whole process

  `process status` - why did your program stop (or is it running…)

  `process detach` - detach from the process you were debugging

  `process kill` - kill it

Brief Tour of Objects—Thread

- Show the threads in your process:

  `thread list`

- Control execution for a thread:

  `thread {step-in/step-over/step-out...}`

  `thread step-in --run-mode this-thread` - run only this thread

- The thread does backtrace:

  `thread backtrace`

  `thread backtrace -c 10 all` - show 10 frames for all threads

Brief Tour of Objects—Frame

- Access the frames in the selected thread
  - Select the current frame with

    ```
    frame select 1
    ```

  - Show locals and statics for the current frame

    ```
    (lldb) frame variable
    (int) argc = 1
    (char **) argv = 0x00007fff5fbff5d0
    ```

  - The selected frame sets the context for
    - Registers
    - Expressions

Brief Tour of Objects—Register

- Register—access the registers in the selected frame
- Native register names
  - rax, rbx…
- Convenience names
  - pc, sp…
  - arg1, arg2…
    - Only valid for “word sized” types
    - Only at the beginning of the function
    - Only as many as your ABI passes in registers

Brief Tour of Objects—Register

- Register values annotated with string or function

```
(lldb) register read
General Purpose Registers:
rax = 0x000000010211c540
rbx = 0x0000000102208970
   ...
rsi = 0x00007fff8eb18c00  "autorelease"
   ...
rip = 0x0000000100017b99  Sketch`-[SKTGraphicView alignLeftEdges:] + 57
```

Look up strings

Look up functions

Aliases

- Having a regular command set makes it easy to learn and find things
- But there must be accelerators for common commands
- By default, LLDB ships with a “GDB-like” set of aliases
  - Listed in “`help`“ after the built-in commands
- But you may find you have some other combination you use often
- Two kinds of short-cuts are possible:
  - Positional aliases
  - Regular expression aliases (power-user!)

Positional Aliases

- Very easy to write
- Created by the command:

```
command alias <alias-name> <substitute command line>
```

- In simplest case, just a straight substitution

```
command alias step thread step-in
```

then:

```
step →
    thread step-in
```

- Additional arguments are appended after substitution

```
step --avoid-no-debug false →
    thread step-in --avoid-no-debug false
```

Positional Aliases

- Can also route arguments to positions in the command
  - Useful when you want to fill in more than one option value
  - `%<num>` in the command line will be filled with argument `<num>`

```
command alias daddr disassemble --count %1 --start-address %2
```

  - Then

```
daddr 20 0x123456
     disassemble --start-address 0x123456 --count 20
```

  - And additional arguments are appended:

```
daddr 20 0x123456 --mixed
     disassemble --start-address 0x123456 --count 20 --mixed
```

- All arguments are required

Alias for More Than One Behavior

- disassemble has two forms, start address or function name

```
disassemble --start-address <ADDRESS> --count <NUM_LINES>
disassemble --name <SYMBOL> --count <NUM_LINES>
```

- But in C addresses are not hard to tell from names (0x vs. [a-zA-Z_])
- Can we do:
  - If there is one argument, beginning with 0x, that's a start address
  - Otherwise if there is one argument it is the function name
  - If none, disassemble at the current pc
  - In each case providing 20 instructions of disassembly…
  - If we don't recognize it, route it to the full "`disassemble`" command

Regexp Aliases—Syntax

- Trickier to write, have to know the regular expression language
- Consist of a list of substitution patterns:

```
s/<match string>/<substitution string>/
```

- The first match string matching the user-typed command wins
- The command name is stripped before matching
- Matched substrings -> `%<NUM>` in the substitution string
- Can also provide help and usage
- Syntax:

```
command regex <NAME> --help “” --syntax “” s/M1/S1/ s/M2/S2/...
```

- Multi-line entry for easier use with many patterns

Regexp Aliases—Patterns

- Remember—substring matches are denoted by “()” in regexps
- The address match would be:

```
s/^(0x[0-9a-fA-F]+)$/disassemble -s %1 -c 20/
```

- The name match:

```
s/^([^0][^x]?[^ ]*)$/disassemble -n %1 -c 20/
```

- No arguments:

```
s/^$/disassemble --pc -c 20/
```

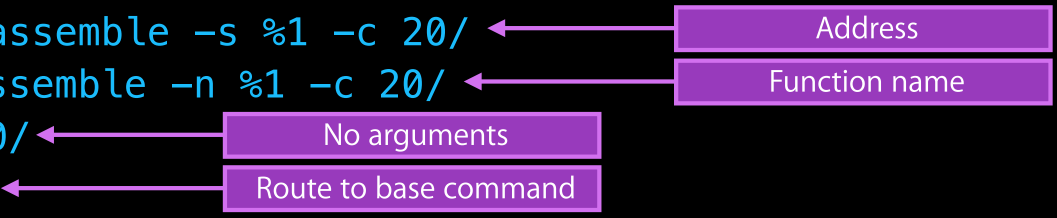
- Passthrough:

```
s/^(.*)$/disassemble %1/
```

Regexp Aliases—Final Result

- Altogether:

```
(lldb) command regex dfancy --help “disassemble by hex address or name”
Enter regular expressions in the form 's/<regex>/<subst>/'
and terminate with an empty line:
s/^(0x[0-9a-fA-F]+)$/disassemble -s %1 -c 20/
s/^([^0][^x][^ ]*)$/disassemble -n %1 -c 20/
s/^$/disassemble -p -c 20/
s/^(.*)$/disassemble %1/
```



```
(lldb)  help dfancy
   disassemble by hex address or name
(lldb)  dfancy 0x7fff8a85fa85
   disassemble -s 0x7fff8a85fa85 -c 20
      0x7fff8a85fa85:  pushq  %rbp ...
```

Summary

- To get started with lldb, you need:
  - “`help` “, a knowledge of how the lldb objects are laid out, and <TAB>
- There are already many shortcut aliases to make you more productive
- It is easy to construct simple shortcuts yourself
- With the “regexp” alias you can make much more powerful ones

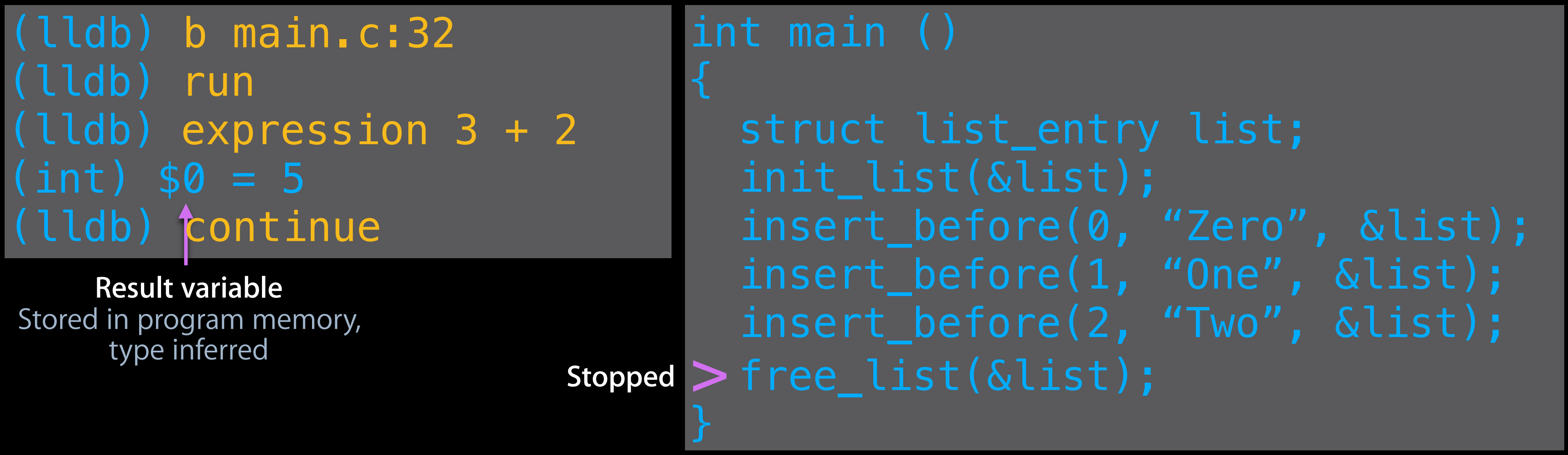
Running Code Inside Your Program

Introducing the Expression Parser

Sean Callanan
AST Wrangler

The Basics

Programming in the current context



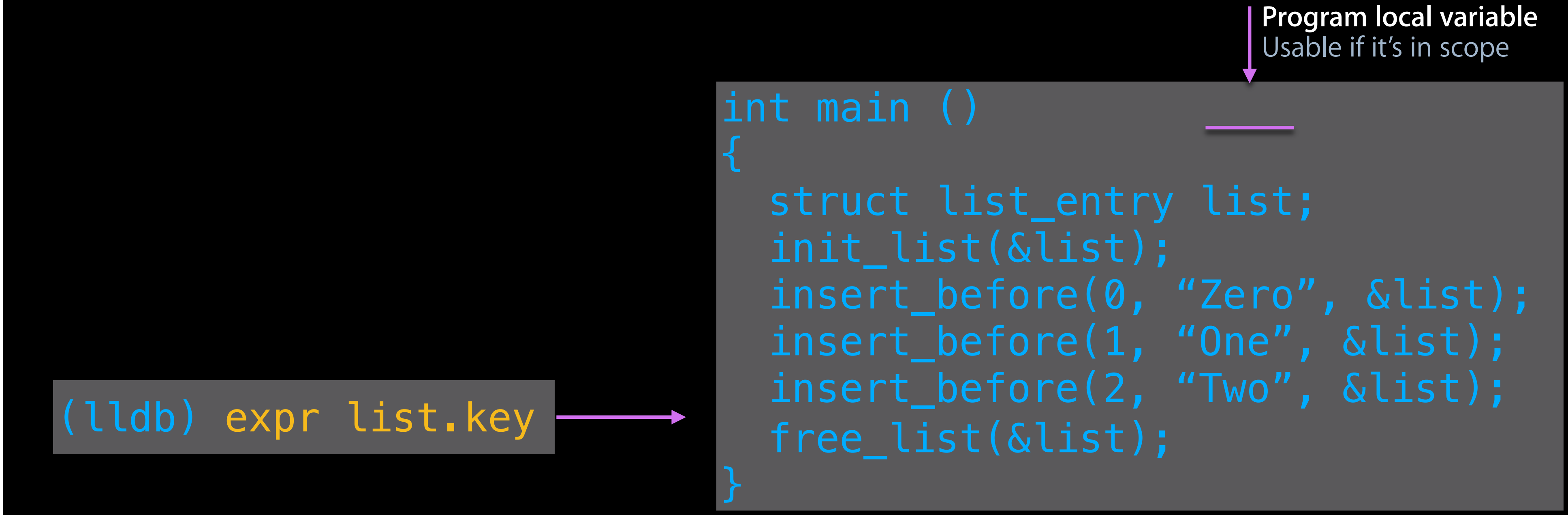
```
(lldb) b main.c:32
(lldb) run
(lldb) expression 3 + 2
(int) $0 = 5
(lldb) continue
```

Result variable
Stored in program memory,
type inferred

```
int main ()
{
  struct list_entry list;
  init_list(&list);
  insert_before(0, “Zero”, &list);
  insert_before(1, “One”, &list);
  insert_before(2, “Two”, &list);
> free_list(&list);
}
```

The Basics

Programming in the current context



Program local variable
Usable if it’s in scope

```
(lldb) expr list.key
```

```
int main ()
{
  struct list_entry list;
  init_list(&list);
  insert_before(0, “Zero”, &list);
  insert_before(1, “One”, &list);
  insert_before(2, “Two”, &list);
  free_list(&list);
}
```

The Basics

Programming in the current context

Multi-line expression
Press Enter after expr;
blank line terminates

```
(lldb) expr
  int i = 3;
  i + 2;
```

```
int main ()
{
  struct list_entry list;
  init_list(&list);
  insert_before(0, “Zero”, &list);
  insert_before(1, “One”, &list);
  insert_before(2, “Two”, &list);
    i + 2;
  }
```

Expression local variable
Usable inside the expression,
disappears afterward

The Basics

Programming in the current context

User variable
Stored in program memory,
available everywhere

```
int $i;
int main ()
{
  struct list_entry list;
  init_list(&list);
  insert_before(0, “Zero”, &list);
  insert_before(1, “One”, &list);
  {
    $i = 3;
    $i + 2;
  }
```

```
(lldb) expr
  int $i = 3;
  $i + 2;
```

The Basics

Programming in the current context

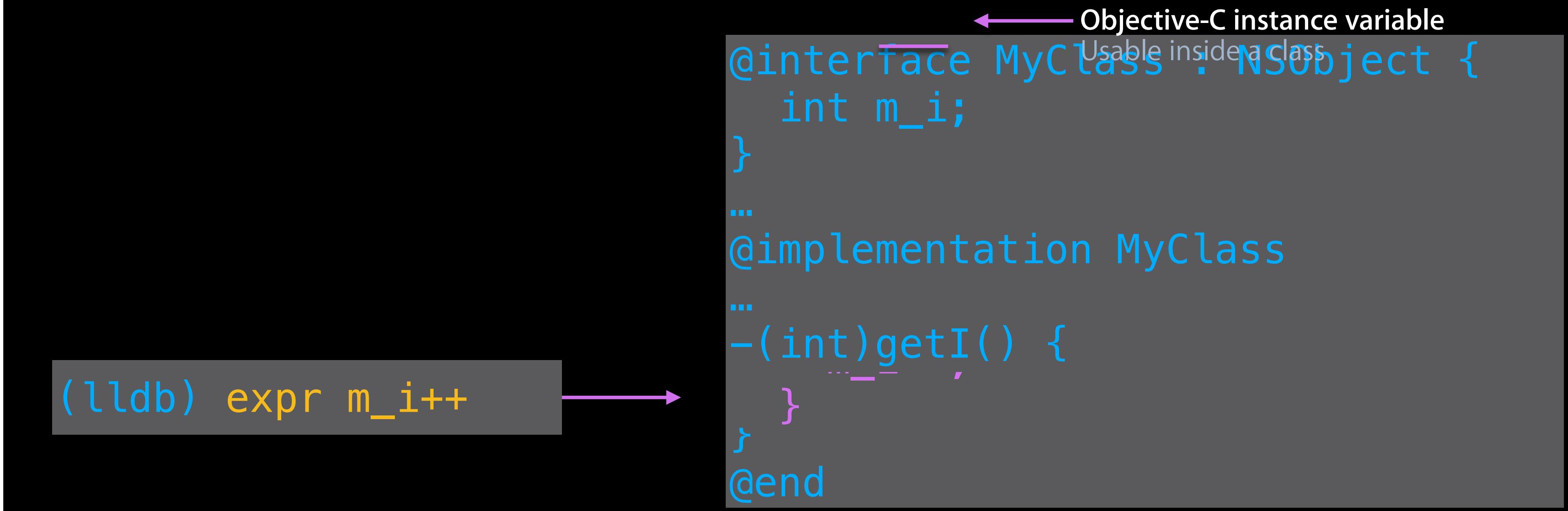
```
(lldb) expr m_i++
```

```
class MyClass {
public:
    {
      m_i++;
    }
  }
private:
  int m_i;
}
```

C++ member variable
Usable inside a class

The Basics

Programming in the current context



```
(lldb) expr m_i++
```

Objective-C instance variable

Usable inside a class

```
@interface MyClass : NSObject {
  int m_i;
}
…
@implementation MyClass
…
-(int)getI() {
  }
}
@end
```

The Basics

Summary—What you can access

- In-scope variables: `expr m_i`
- Globals and functions with debug info: `expr myfunc()`
- Global symbols without debug info (casts required)
    - Functions: `expr (int)strlen(“Hello world!”)`
    - Variables: `expr (char**)environ`
- Expression-local variables: `expr int i = 2; i + 3`
- User variables
    - Create once: `expr int $i`
    - Use repeatedly: `expr $i++`

Example

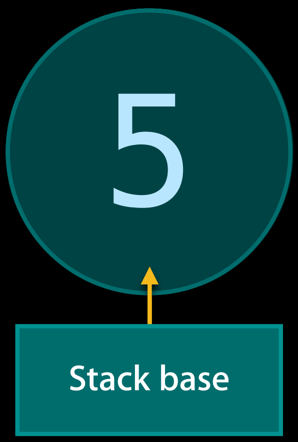
Debugging an RPN calculator

```
> 7
> 5
> +
12
```

Example

Debugging an RPN calculator

```
> 7
> 5
> +
12
```



5

Stack base

Example

Debugging an RPN calculator

```
> 7
> +
Segmentation
fault
```

Example

Inspect the stack, read variables

```
$ lldb rpn
Current executable set to 'rpn' (x86_64).
(lldb) run
Process 3088 launched: 'rpn' (x86_64)
> 7
> +
Process 3088 stopped
…
(lldb) bt
* thread #1: … stop reason = EXC_BAD_ACCESS …
  frame #0: 0x0000000100000e11 rpn`add + 33
  frame #1: 0x0000000100000ce7 rpn`main + 343
  frame #2: 0x0000000100000b84 rpn`start + 52
```

No debug information!
At add+33, args could
be anywhere.

Example

Plan B: Read arguments from registers

```
(lldb) b add
(lldb) run
There is a running process, kill it and
restart?: [Y/n] yes
> 7
> +
Process 3088 stopped
…
(lldb) bt
* thread #1 … stop reason = breakpoint 1.1
  frame #0: 0x0000000100000df0 rpn`add
  frame #1: 0x0000000100000ce7 rpn`main + 343
  frame #2: 0x0000000100000b84 rpn`start + 52
```

At the entry point
Now, arguments are
available in registers.

Example

Plan B: Read arguments from registers

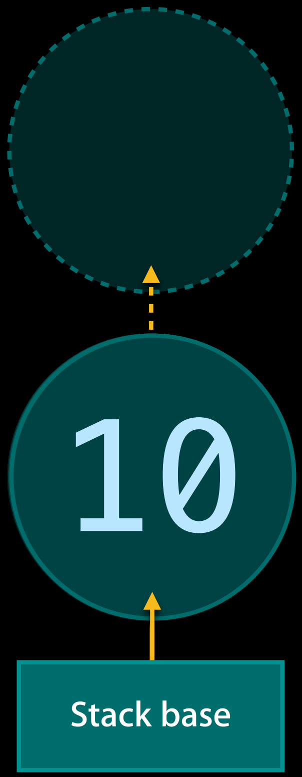
```
(lldb) expr --format x -- $arg1
(unsigned long) $0 = 0x00007fff5fbffb18
(lldb) expr
struct stack_entry {
  struct stack_entry *next;
  long long int value;
};
struct stack_entry **$stack =
  (struct stack_entry**)$arg1

Expression did not return a result
```

Argument register

Example

Fix the problem

```
(lldb) expr (*$stack)->value
(long long) $2 = 7
(lldb) expr (*$stack)->next
(struct stack_entry *) $3 =
0x0000000000000000
(lldb) expr (void)push($stack, 3)
(lldb) expr (*$stack)->next
(struct stack_entry *) $4 =
0x00000001001006f0
(lldb) continue
10
>
```



Example

Compute the depth of the stack

```
> 3
> 5
> +
Process 3088 stopped
(lldb) expr
struct s { struct s *next; long long value; };
int depth = 0;
for (struct s *current = *((struct s**)$arg1);
     current != 0;
     current = current->next)
  depth++;
depth;
(int) $5 = 2
```

Type definitions are scoped

If you create new variables, redeclare the type.

Summary

- Use the expression parser to interact directly with your code
    - Use registers, variables, and functions available where LLDB is stopped
    - Create your own user variables (`$stack`)
    - Reconstruct program state even without debug information
    - Use full Objective-C++ in expressions
- `(lldb) help expr`
    - Provides more information about arguments to the `expr` command, especially how to format output

Migrating from GDB to LLDB

Scripting and Python in LLDB

Caroline Tice
Debugger Engineer

What Can You Do with Scripting in LLDB?

- Set REALLY useful conditional breakpoints
    - By caller’s name
    - By caller’s argument values
    - By thread
        - …and whether same thread hit it last time!

What Can You Do with Scripting in LLDB?

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

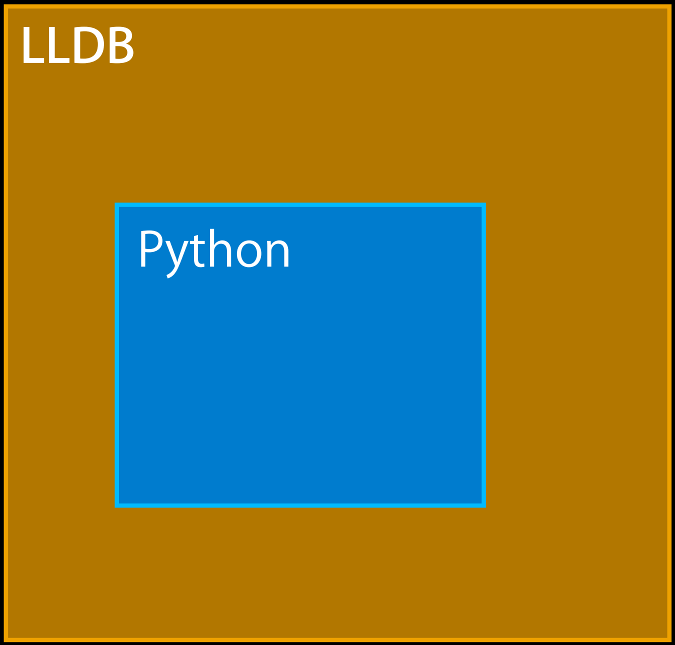
What Can You Do with Scripting in LLDB?

- 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…

What Can You Do with Scripting in LLDB?

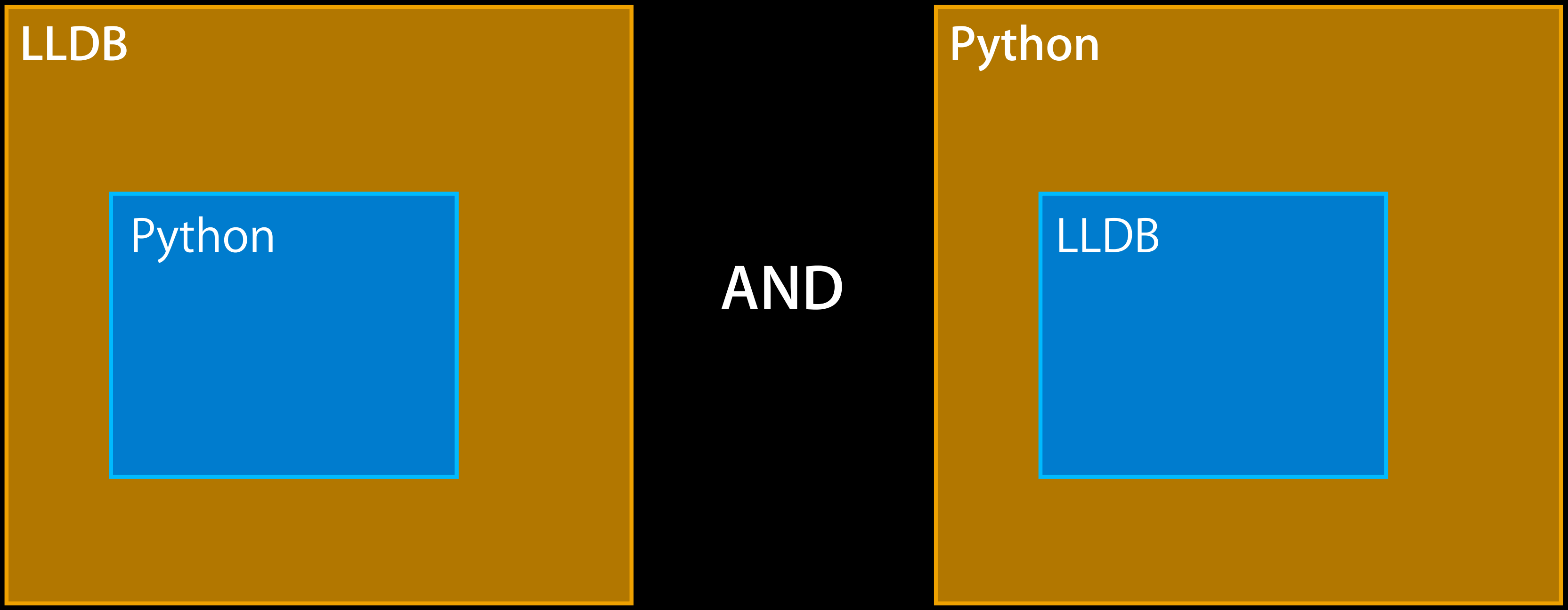
- Set REALLY useful conditional breakpoints
- Find specific data in large dynamic data structures
- Automatically record register values and program state
- Testing/QA (especially intermittent bugs)

What is Where?



Python is accessible from LLDB

What is Where?



Python is accessible from LLDB

LLDB is accessible from Python

LLDB in Python (Directly)

```
% setenv PYTHONPATH \
         /Developer/Library/PrivateFrameworks/LLDB.framework/Resources/Python
% 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 in Python (Directly)

```
% setenv PYTHONPATH \
         /Developer/Library/PrivateFrameworks/LLDB.framework/Resources/Python
% 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
function calls

LLDB in Python (Directly)

```
% setenv PYTHONPATH \
        /Developer/Library/PrivateFrameworks/LLDB.framework/Resources/Python
% 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)
```

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 'quit()', 'exit()' 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 --script-type python 1
Enter your Python command(s). Type 'DONE' to end.
>
```

LLDB Scripting/Python Enhancements

- API functions
  - Create, access and manipulate debugger objects and state

- Execution context objects
  - pre-loaded into Python “convenience variables”

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

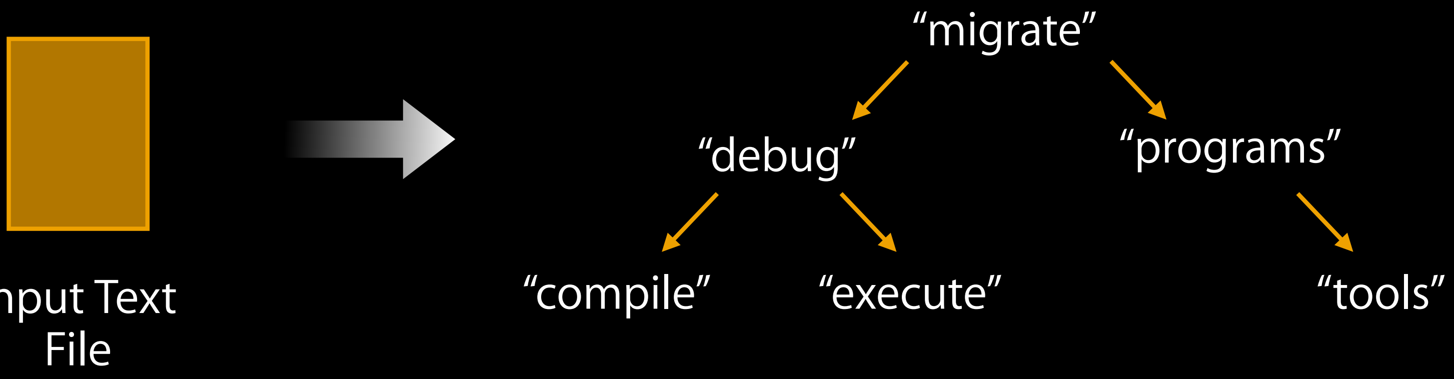
- Single Python interpreter for entire debugger session

Part 2—Scripting in Action

Using scripting in LLDB to find a bug…

Example: Simple Dictionary Program

Store and find words in Binary Search Tree


Input Text File

“migrate”

“debug”

“programs”

“compile”

“execute”

“tools”

Find (“tools”) → Yes

Find (“assemble”) → No

Problem: Word is 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 is Not Found in Dictionary

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

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

    - Write a script to do it for you!

The Plan

(Searching tree without restarting program)

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

The Plan

(Searching tree without restarting program)

- Write Depth-First Search (DFS) function in file (tree_utils.py)
  - “define DFS (root, word, cur_path): …”
- Attach to running program with LLDB
- Use interactive interpreter to call DFS on existing tree
- DFS function returns root-to-node path, if found

User-created file

The Plan

(Searching tree without restarting program)

- Write Depth-First Search (DFS) function in file (tree_utils.py)
  - “define DFS (root, word, cur_path): …”
- Attach to running program with LLDB
- Use interactive interpreter to call DFS on existing tree
- DFS function returns root-to-node path, if found

The Plan

(Searching tree without restarting program)

- Write Depth-First Search (DFS) function in file (tree_utils.py)
    - “define DFS (root, word, cur_path): …”
- Attach to running program with LLDB
- Use interactive interpreter to call DFS on existing tree
- DFS function returns root-to-node path, if found

The Plan

(Searching tree without restarting program)

- Write Depth-First Search (DFS) function in file (tree_utils.py)
  - “define DFS (root, word, cur_path): …”
- Attach to running program with LLDB
- Use interactive interpreter to call DFS on existing tree
- DFS function returns root-to-node path, if found

The Plan

(Searching tree without restarting program)

- Write Depth-First Search (DFS) function in file (tree_utils.py)
    - “define DFS (root, word, cur_path): …”
- Attach to running program with LLDB
- Use interactive interpreter to call DFS on existing tree
- DFS function returns root-to-node path, if found

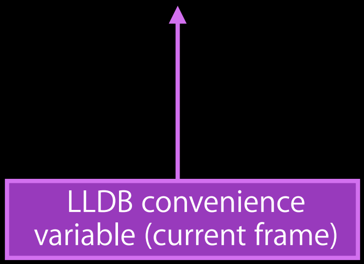
Using the Interactive Interpreter

```
(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”)
>>>
```

User-created file (module)

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 (current frame)

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 API function call

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”)
>>> current_path = “”
>>> path = tree_utils.DFS (root, “Romeo”, current_path)
>>>
```

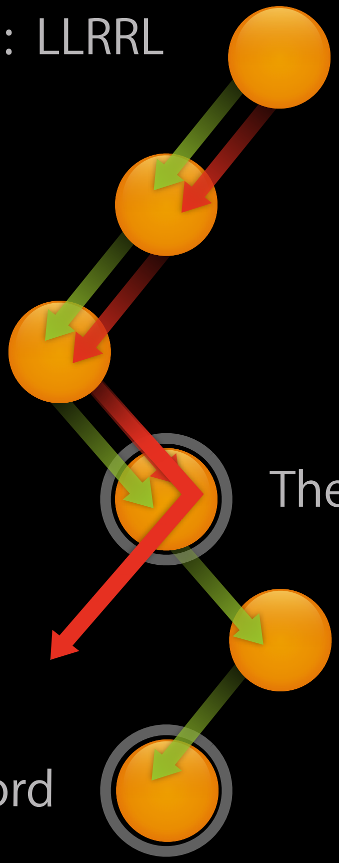
Using the Interactive Interpreter

```
(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”)
>>>  current_path = “”
>>>  path = tree_utils.DFS (root, “Romeo”, current_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”)
>>> current_path = “”
>>> path = tree_utils.DFS (root, “Romeo”, current_path)
>>> print path
LLRRL
>>> ^D
```

We’re Halfway There...



Path: LLRRL

The problem

Our word

- WE found the word…
  why didn’t the program?
- How do we find the problem?
  - Scripted breakpoint commands!

Python Breakpoint Command

(At decision to follow right child)

```
def obscure_func_name (frame, bp_loc):
 Enter your Python command(s). Type ‘DONE’ to end.
 > global path
 > if path[0] == 'R':
 >     path = path[1:]
 >     thread = frame.GetThread()
 >     process = thread.GetProcess()
 >     process.Continue()
 > else:
 >     print "Going right, should go left!"
 > DONE

obscure_func_name (cur_frame, cur_bp_loc)
```

Python Breakpoint Command

(At decision to follow right child)

```
def obscure_func_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!"


obscure_func_name (cur_frame, cur_bp_loc)
```

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!"
```

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
function calls

Results…

```
(lldb) breakpoint command add --script-type python 1
...
(lldb) breakpoint command add --script-type 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

- LLDB makes scripting easy, useful and powerful
- Convenience variables and API function calls are your friends!
- Load LLDB directly into Python
    - Great way to do automated testing and QA
    - Lots of good examples in LLDB test suite
- LOTS more you can do…

LLDB in Review

- LLDB Command Line
  - object-action syntax
  - “`help`” and “`apropos`” and <TAB>
  - Aliases
- Expression Parser
  - Executing code inside your program
  - Debugging without debug info
- Scripting and Python in LLDB
  - Easy to access; easy to use
  - LLDB convenience variables + API functions = COOL STUFF!

For Further Reference

- Information on the LLDB website
  - General info about LLDB (http://lldb.llvm.org)
  - Tutorial for GDB->LLDB transition (http://lldb.llvm.org/tutorial.html)
  - Today’s Python scripting examples (http://lldb.llvm.org/scripting.html)

- Information in the LLDB source tree (download the sources)
  - API functions: API header files (lldb/include/lldb/API)
  - Running LLDB directly from Python: LLDB test suite (lldb/test)

- Information about Python
  - http://www.python.org

More Information

Michael Jurewitz
Developer Tools and Performance Evangelist
jurewitz@apple.com

Apple Developer Forums
http://devforums.apple.com

Related Sessions

Effective Debugging with Xcode 4	Pacific Heights Friday 9:00AM

WWDC2011