Introduction to GDB Python

Pierre-Marie de Rodat

pmderodat@lse.epita.fr
PM @ {rezosup, freenode, geeknode, . . . }
`http://lse.epita.fr`

February 12, 2013

Plan

1 Introduction

GDB

- Well known source-level debugger from the GNU Project
- Can be scripted using a quite resticting specific language (one more to learn!)
- As far as I could read, it can only define macros

GDB

- Well known source-level debugger from the GNU Project
- Can be scripted using a quite resticting specific language (one more to learn!)
- As far as I could read, it can only define macros

GDB

- Well known source-level debugger from the GNU Project
- Can be scripted using a quite resticting specific language (one more to learn!)
- As far as I could read, it can only define macros

PythonGDB

- GDB 7 introduced Python scripting capabilities
- It can be accessed through a dedicated API (gdb module)
- This API enables scripts to:
    - Define pretty-printers for types defined in debugged programs (*inferiors*)
    - Define new commands (like macros)
    - Deal with symbols, stack frames, values type
    - Inspect and modify inferiors memory
    - Register callbacks for events (inferior termination, . . .)
    - Set breakpoints
    - . . .

PythonGDB

- GDB 7 introduced Python scripting capabilities
- It can be accessed through a dedicated API (`gdb` module)
- This API enables scripts to:
  - Define pretty-printers for types defined in debugged programs (*inferiors*)
  - Define new commands (like macros)
  - Deal with symbols, stack frames, values type
  - Inspect and modify inferiors memory
  - Register callbacks for events (inferior termination, . . .)
  - Set breakpoints
  - . . .

PythonGDB

- GDB 7 introduced Python scripting capabilities
- It can be accessed through a dedicated API (`gdb` module)
- This API enables scripts to:
    - Define pretty-printers for types defined in debugged programs (*inferiors*)
    - Define new commands (like macros)
    - Deal with symbols, stack frames, values type
    - Inspect and modify inferiors memory
    - Register callbacks for events (inferior termination, . . .)
    - Set breakpoints
    - . . .

PythonGDB

- GDB 7 introduced Python scripting capabilities
- It can be accessed through a dedicated API (`gdb` module)
- This API enables scripts to:
  - Define pretty-printers for types defined in debugged programs (*inferiors*)
  - Define new commands (like macros)
  - Deal with symbols, stack frames, values type
  - Inspect and modify inferiors memory
  - Register callbacks for events (inferior termination, . . .)
  - Set breakpoints
  - . . .

PythonGDB

- GDB 7 introduced Python scripting capabilities
- It can be accessed through a dedicated API (`gdb` module)
- This API enables scripts to:
    - Define pretty-printers for types defined in debugged programs (*inferiors*)
    - Define new commands (like macros)
    - Deal with symbols, stack frames, values type
    - Inspect and modify inferiors memory
    - Register callbacks for events (inferior termination, . . .)
    - Set breakpoints
    - . . .

PythonGDB

- GDB 7 introduced Python scripting capabilities
- It can be accessed through a dedicated API (`gdb` module)
- This API enables scripts to:
  - Define pretty-printers for types defined in debugged programs (*inferiors*)
  - Define new commands (like macros)
  - Deal with symbols, stack frames, values type
  - Inspect and modify inferiors memory
  - Register callbacks for events (inferior termination, . . .)
  - Set breakpoints
  - . . .

PythonGDB

- GDB 7 introduced Python scripting capabilities
- It can be accessed through a dedicated API (`gdb` module)
- This API enables scripts to:
    - Define pretty-printers for types defined in debugged programs (*inferiors*)
    - Define new commands (like macros)
    - Deal with symbols, stack frames, values type
    - Inspect and modify inferiors memory
    - Register callbacks for events (inferior termination, . . .)
    - Set breakpoints
    - . . .

PythonGDB

- GDB 7 introduced Python scripting capabilities
- It can be accessed through a dedicated API (`gdb` module)
- This API enables scripts to:
  - Define pretty-printers for types defined in debugged programs (*inferiors*)
  - Define new commands (like macros)
  - Deal with symbols, stack frames, values type
  - Inspect and modify inferiors memory
  - Register callbacks for events (inferior termination, ...)
  - Set breakpoints
  - ...

Plan

2 Usage examples

- Loading scripts
- Defining a pretty-printer
- Defining breakpoints
- Convenience functions

Loading scripts

- Most simple way (by hand):

  ```
  source my_script.py
  ```

  ```
  gdb my_program -ex"source my_script.py"
  ```

- Auto-loading (eg. for libraries):
  - Enable `auto-load python-scripts`
  - Rename your script to `*objfile*-gdb.py`
  - Fix security-related settings (`scripts-directory`, `safe-path`, ...)
- Other strange ways (`.debug_gdb_scripts`) ...

Loading scripts

- Most simple way (by hand):

  ```
  source my_script.py
  ```

  ```
  gdb my_program -ex"source my_script.py"
  ```

- Auto-loading (eg. for libraries):
  - Enable `auto-load python-scripts`
  - Rename your script to *`objfile`*`-gdb.py`
  - Fix security-related settings (`scripts-directory`, `safe-path`, . . .)
- Other strange ways (`.debug_gdb_scripts`) . . .

Loading scripts

- Most simple way (by hand):

  ```
  source my_script.py
  ```

  ```
  gdb my_program -ex"source my_script.py"
  ```

- Auto-loading (eg. for libraries):
  - Enable `auto-load python-scripts`
  - Rename your script to *`objfile`*`-gdb.py`
  - Fix security-related settings (`scripts-directory`, `safe-path`, . . .)

- Other strange ways (`.debug_gdb_scripts`) . . .

Loading scripts

- Most simple way (by hand):

  ```
  source my_script.py
  ```

  ```
  gdb my_program -ex"source my_script.py"
  ```

- Auto-loading (eg. for libraries):
  - Enable `auto-load python-scripts`
  - Rename your script to `*objfile*-gdb.py`
  - Fix security-related settings (`scripts-directory`, `safe-path`, . . .)
- Other strange ways (`.debug_gdb_scripts`) . . .

Loading scripts

- Most simple way (by hand):

  ```
  source my_script.py
  ```

  ```
  gdb my_program -ex"source my_script.py"
  ```

- Auto-loading (eg. for libraries):
  - Enable `auto-load python-scripts`
  - Rename your script to *`objfile`*`-gdb.py`
  - Fix security-related settings (`scripts-directory`, `safe-path`, . . .)
- Other strange ways (`.debug_gdb_scripts`) . . .

Why? (1/2)

- GDB itself can pretty-print process data following the (eg. C) layout (with `-g`)

```c
struct my_list
{
    int value;
    struct my_list *next;
};
...
struct my_list *my_list = ...
```

```
(gdb) print my_list
$1 = (struct my_list *) 0x7fffffffe200
(gdb) print *my_list
$2 = {value = 0, next = 0x7fffffffe210}
(gdb) print *my_list->next
$3 = {value = 1, next = 0x7fffffffe220}
```

Why? (1/2)

- GDB itself can pretty-print process data following the (eg. C) layout (with -g)

```
struct my_list
{
    int value;
    struct my_list *next;
};
...
struct my_list *my_list = ...
```

```
(gdb) print my_list
$1 = (struct my_list *) 0x7fffffffe200
(gdb) print *my_list
$2 = {value = 0, next = 0x7fffffffe210}
(gdb) print *my_list->next
$3 = {value = 1, next = 0x7fffffffe220}
```

Why? (1/2)

- GDB itself can pretty-print process data following the (eg. C) layout (with -g)

```
struct my_list
{
    int value;
    struct my_list *next;
};
...
struct my_list *my_list = ...
```

```
(gdb) print my_list
$1 = (struct my_list *) 0x7fffffffe200
(gdb) print *my_list
$2 = {value = 0, next = 0x7fffffffe210}
(gdb) print *my_list->next
$3 = {value = 1, next = 0x7fffffffe220}
```

Why? (2/2)

- Useful to debug low-level stuff, not algorithms
- Register pretty-printers for your data types and print them!
- Container libraries (like the `libstdc++`) already bundle some

Why? (2/2)

- Useful to debug low-level stuff, not algorithms
- Register pretty-printers for your data types and print them!
- Container libraries (like the `libstdc++`) already bundle some

Why? (2/2)

- Useful to debug low-level stuff, not algorithms
- Register pretty-printers for your data types and print them!
- Container libraries (like the `libstdc++`) already bundle some

Example — Definition

```
import gdb

class MyListPrinter(object):
    ’’’Print a struct my_list (*)’’’

    def __init__(self, value):
        my_list_type = \
            gdb.lookup_type(’struct my_list’)
        self.value = (value.address
            if value.type == my_list_type
            else value)

    def to_string(self):
        elts, node_p = [], self.value
        while node_p != gdb.Value(0):
            node = node_p.dereference()
            elts.append(str(node[’value’]))
            node_p = node[’next’]
        return ’[{}]’.format(’; ’.join(elts))
```

Example — Registering

- Pretty-printer matching can be very complex
- Think about generic data structure in C++...
- GDB has a list of (user provided) "value handlers" to look for a pretty-printer
- "handlers" are called with the value to print
- If one of them returns a pretty-printer, GDB uses it.

```
def my_list_lookup(value):
    my_list_type = gdb.lookup_type('struct my_list')
    my_list_p = gdb.Type.pointer(my_list_type)
    if value.type in (my_list_type, my_list_p):
        return MyListPrinter(value)
    else:
        return None

gdb.pretty_printers.append(my_list_lookup)
```

Example — Registering

- Pretty-printer matching can be very complex
- Think about generic data structure in C++...
- GDB has a list of (user provided) "value handlers" to look for a pretty-printer
- "handlers" are called with the value to print
- If one of them returns a pretty-printer, GDB uses it.

```
def my_list_lookup(value):
    my_list_type = gdb.lookup_type(’struct my_list’)
    my_list_p = gdb.Type.pointer(my_list_type)
    if value.type in (my_list_type, my_list_p):
        return MyListPrinter(value)
    else:
        return None

gdb.pretty_printers.append(my_list_lookup)
```

Example — Registering

- Pretty-printer matching can be very complex
- Think about generic data structure in C++. . .
- GDB has a list of (user provided) "value handlers" to look for a pretty-printer
- "handlers" are called with the value to print
- If one of them returns a pretty-printer, GDB uses it.

```
def my_list_lookup(value):
    my_list_type = gdb.lookup_type(’struct my_list’)
    my_list_p = gdb.Type.pointer(my_list_type)
    if value.type in (my_list_type, my_list_p):
        return MyListPrinter(value)
    else:
        return None

gdb.pretty_printers.append(my_list_lookup)
```

Example — Registering

- Pretty-printer matching can be very complex
- Think about generic data structure in C++. . .
- GDB has a list of (user provided) "value handlers" to look for a pretty-printer
- "handlers" are called with the value to print
- If one of them returns a pretty-printer, GDB uses it.

```
def my_list_lookup(value):
    my_list_type = gdb.lookup_type('struct my_list')
    my_list_p = gdb.Type.pointer(my_list_type)
    if value.type in (my_list_type, my_list_p):
        return MyListPrinter(value)
    else:
        return None

gdb.pretty_printers.append(my_list_lookup)
```

Example — Registering

- Pretty-printer matching can be very complex
- Think about generic data structure in C++. . .
- GDB has a list of (user provided) "value handlers" to look for a pretty-printer
- "handlers" are called with the value to print
- If one of them returns a pretty-printer, GDB uses it.

```
def my_list_lookup(value):
    my_list_type = gdb.lookup_type('struct my_list')
    my_list_p = gdb.Type.pointer(my_list_type)
    if value.type in (my_list_type, my_list_p):
        return MyListPrinter(value)
    else:
        return None

gdb.pretty_printers.append(my_list_lookup)
```

Example — Usage

```
(gdb) source my_script.py
(gdb) print my_list
$1 = [0; 1; 2; 3]
(gdb) print *my_list
$2 = [0; 1; 2; 3]
(gdb) print /r *my_list
$3 = {value = 0, next = 0x7fffffffe210}
```

```
(gdb) print my_struct
$4 = {
    str = 0x4005f4 "Hello, world!\n",
    code = 204,
    args = [0; 1; 2; 3]
}
```

Example — Usage

```
(gdb) source my_script.py
(gdb) print my_list
$1 = [0; 1; 2; 3]
(gdb) print *my_list
$2 = [0; 1; 2; 3]
(gdb) print /r *my_list
$3 = {value = 0, next = 0x7fffffffe210}

(gdb) print my_struct
$4 = {
    str = 0x4005f4 "Hello, world!\n",
    code = 204,
    args = [0; 1; 2; 3]
}
```

Use case

- You want to compute statistics abount function calls.
- Say you have a factorial function:

```c
#include <stdio.h>

unsigned fact(unsigned n)
{
    if (n <= 1) return 1;
    else return n * fact(n - 1);
}

void print_fact(unsigned n)
{ printf("fact(%u) =\t%u\n", n, fact(n)); }

int main(void)
{
    unsigned i, n = 0;
    for (i = 0; i<100; ++i, n=(n+7)%17)
        print_fact(n);
}
```

- You want to know how many times `fact` is called for each argument value.

Example — Breakpoint definition

```python
from collections import defaultdict
import gdb

class FactHistogram(gdb.Breakpoint):
    def __init__(self):
        super(FactHistogram, self).__init__(
            'fact', internal=True)
        # By default, a value is never used.
        self.histogram = defaultdict(lambda: 0)

    def stop(self):
        arg = int(gdb.newest_frame().read_var('n'))
        self.histogram[arg] += 1
        return False # Resume execution

fact_histo = FactHistogram()
```

Example — Helper command definition

```
class PrintHistogram(gdb.Command):
    def __init__(self):
        super(PrintHistogram, self).__init__(
            'histo', gdb.COMMAND_BREAKPOINTS,
            gdb.COMPLETE_NONE)

    def invoke(self, argument, from_tty):
        for arg, times in sorted(
            fact_histo.histogram.items(),
            key=lambda item: item[1]):
            gdb.write('fact({}):\t{} times\n'.format(
                arg, times))

PrintHistogram()
```

Example — Usage

```
(gdb) source my_script.py
(gdb) r
Starting program: /tmp/fact.c
fact(0) =       1
fact(7) =       5040
fact(14) =      1278945280
[...]
fact(16) =      2004189184
fact(6) =       720
fact(13) =      1932053504
[Inferior 1 (process 7605) exited with code 03]
(gdb) histo
fact(0):        6 times
fact(16):       6 times
fact(15):       12 times
[...]
fact(5):        71 times
fact(4):        77 times
fact(3):        82 times
fact(2):        88 times
fact(1):        94 times
```

Example — Debugged program

```c
#include <stdio.h>

typedef void (*callback_fn)(int);

static void internal_callback(int no)
{ printf("Internal callback call with no = %d\n", no); }

callback_fn get_callback(void)
{ return internal_callback; }

void internal_process_something(void)
{ get_callback()(0xcd); }

int main(void)
{
    internal_process_something();
    get_callback()(0xcc);
    return 0;
}
```

Example — Convenience function definition

```python
import gdb

class ForbiddenCaller(gdb.Function):
    def __init__(self):
        super(ForbiddenCaller, self).__init__('forbidden_caller')
    def invoke(self, prefix):
        prefix = prefix.string()
        frame = gdb.newest_frame().older()
        fr_name = frame.name()
        return not fr_name.startswith(prefix)

ForbiddenCaller()
```

Example — Usage

```
(gdb) source my_script.py
(gdb) b internal_callback if $forbidden_call("internal_")
Breakpoint 1 at 0x400507: file callback.c, line 7.
(gdb) r
Starting program: /tmp/callback
Internal callback call with no = 205

Breakpoint 1, internal_callback (no=204) at callback.c:7
7           printf("Internal callback call with no = %d\n", no);
(gdb) bt
#0  internal_callback (no=204) at callback.c:7
#1  0x000000000040054f in main () at callback.c:23
```

Plan

3 Conclusion

Conclusion

- The Python API is really easy to use
- It looks more powerful than the specific language
- It is still evolving: stay tuned!

Conclusion

- The Python API is really easy to use
- It looks more powerful than the specific language
- It is still evolving: stay tuned!

Conclusion

- The Python API is really easy to use
- It looks more powerful than the specific language
- It is still evolving: stay tuned!