The python debugger

Michael Løiten

`mmag@fysik.dtu.dk`

Slides and programs: `github.com/loeiten/python_club`

April 29, 2015

python **deb**ugger
An elegant way to debug (find mistakes in) your python code

Short intro to the stack frame

Stack-frame

For future references: What is all this stack-frame fuzz?

```python
1   #!/usr/bin/env python
2
3   def main():
4       a = 10
5       b = 55
6       result = abs_a_minus_b(a,b)
7       print("The absolute value of {0}-{1} is {2}".format(a,b,result))
8
9
10  def abs_a_minus_b(x,y):
11      if x > y:
12          z = x - y
13      else:
14          z = y - x
15      return z
16
17
18  if __name__ == '__main__':
19      main()
```
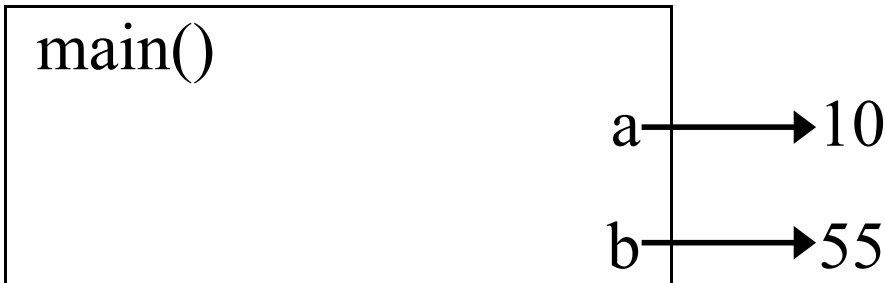
For future references: What is all this stack-frame fuzz?

```python
1   #!/usr/bin/env python
2
3   def main():
4       a = 10
5       b = 55
6       result = abs_a_minus_b(a,b)
7       print("The absolute value of {0}-{1} is {2}".format(a,b,result))
8
9
10  def abs_a_minus_b(x,y):
11      if x > y:
12          z = x - y
13      else:
14          z = y - x
15      return z
16
17
18  if __name__ == '__main__':
19      main()
```

```
main()
```

For future references: What is all this stack-frame fuzz?

```python
1   #!/usr/bin/env python
2
3   def main():
4       a = 10
5       b = 55
6       result = abs_a_minus_b(a,b)
7       print("The absolute value of {0}-{1} is {2}".format(a,b,result))
8
9
10  def abs_a_minus_b(x,y):
11      if x > y:
12          z = x - y
13      else:
14          z = y - x
15      return z
16
17
18  if __name__ == '__main__':
19      main()
```

For future references: What is all this stack-frame fuzz?
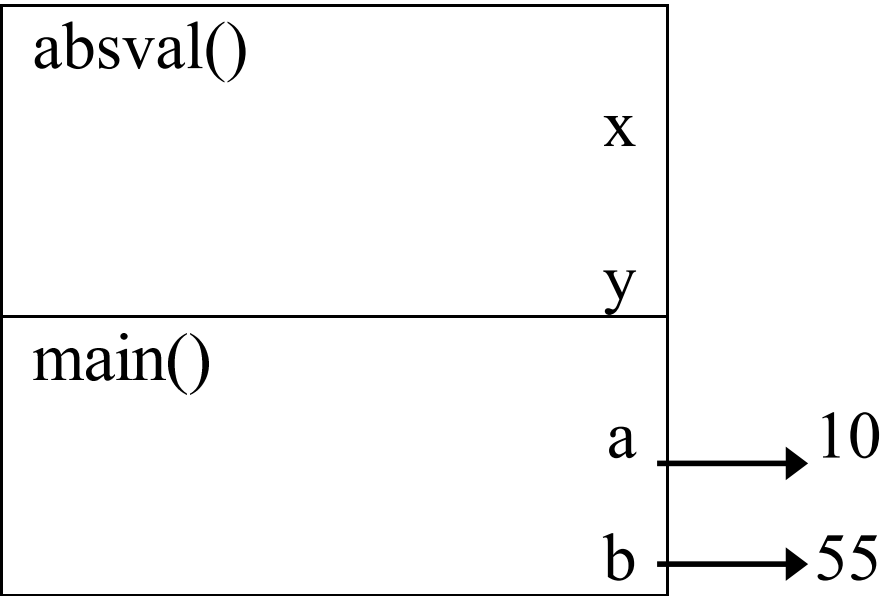
```python
1   #!/usr/bin/env python
2
3   def main():
4       a = 10
5       b = 55
6       result = abs_a_minus_b(a,b)
7       print("The absolute value of {0}-{1} is {2}".format(a,b,result))
8
9
10  def abs_a_minus_b(x,y):
11      if x > y:
12          z = x - y
13      else:
14          z = y - x
15      return z
16
17
18  if __name__ == '__main__':
19      main()
```

For future references: What is all this stack-frame fuzz?
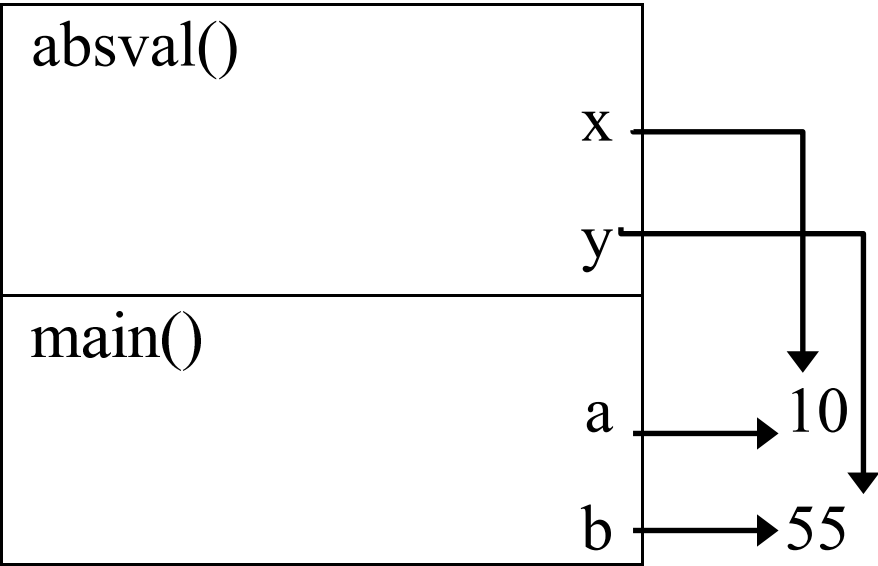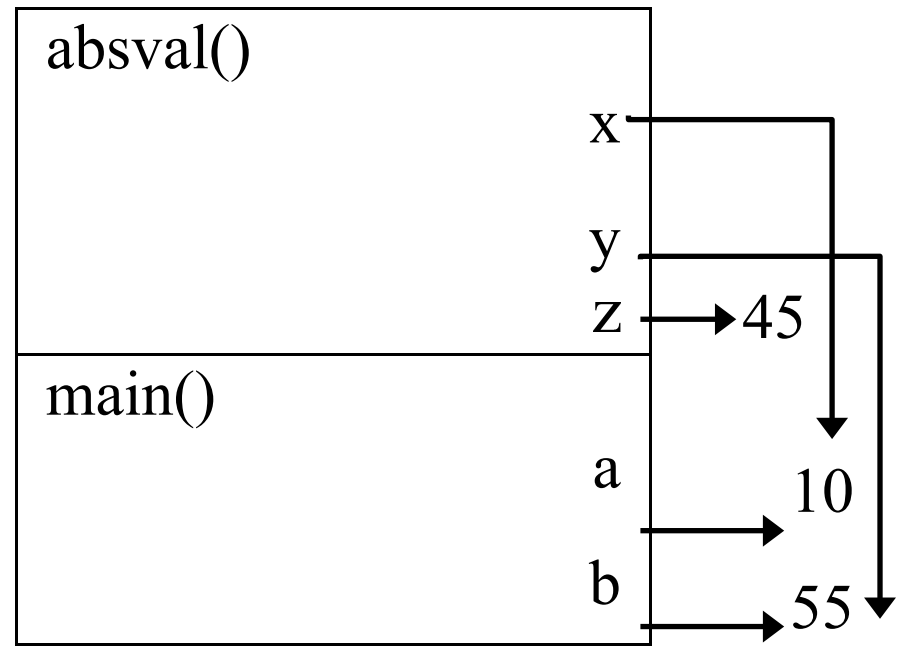
```python
1   #!/usr/bin/env python
2
3   def main():
4       a = 10
5       b = 55
6       result = abs_a_minus_b(a,b)
7       print("The absolute value of {0}-{1} is {2}".format(a,b,result))
8
9
10  def abs_a_minus_b(x,y):
11      if x > y:
12          z = x - y
13      else:
14          z = y - x
15      return z
16
17
18  if __name__ == '__main__':
19      main()
```

```
┌─────────────────────────────┐
│ absval()                    │
│                          x  │
│                             │
│                          y  │
├─────────────────────────────┤
│ main()                      │
│                          a ─┼──→ 10
│                             │
│                          b ─┼──→ 55
└─────────────────────────────┘
```

For future references: What is all this stack-frame fuzz?
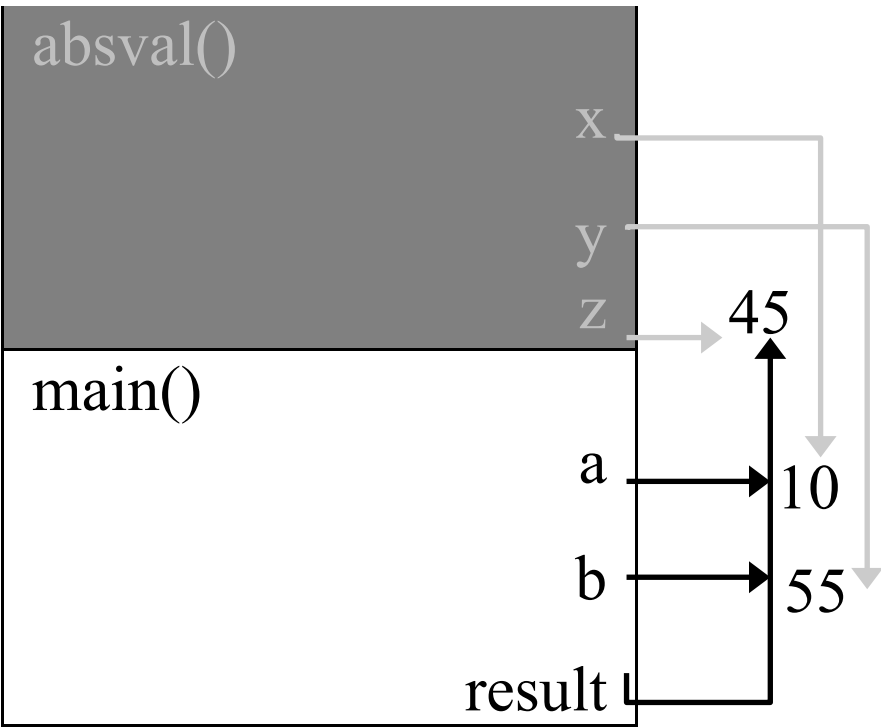
```
1   #!/usr/bin/env python
2
3   def main():
4       a = 10
5       b = 55
6       result = abs_a_minus_b(a,b)
7       print("The absolute value of {0}-{1} is {2}".format(a,b,result))
8
9
10  def abs_a_minus_b(x,y):
11      if x > y:
12          z = x - y
13      else:
14          z = y - x
15      return z
16
17
18  if __name__ == '__main__':
19      main()
```

For future references: What is all this stack-frame fuzz?

```python
1  #!/usr/bin/env python
2
3  def main():
4      a = 10
5      b = 55
6      result = abs_a_minus_b(a,b)
7      print("The absolute value of {0}-{1} is {2}".format(a,b,result))
8
9
10 def abs_a_minus_b(x,y):
11     if x > y:
12         z = x - y
13     else:
14         z = y - x
15     return z
16
17
18 if __name__ == '__main__':
19     main()
```

For future references: What is all this stack-frame fuzz?

```python
1   #!/usr/bin/env python
2
3   def main():
4       a = 10
5       b = 55
6       result = abs_a_minus_b(a,b)
7       print("The absolute value of {0}-{1} is {2}".format(a,b,result))
8
9
10  def abs_a_minus_b(x,y):
11      if x > y:
12          z = x - y
13      else:
14          z = y - x
15      return z
16
17
18  if __name__ == '__main__':
19      main()
```

For future references: What is all this stack-frame fuzz?

```python
1   #!/usr/bin/env python
2
3   def main():
4       a = 10
5       b = 55
6       result = abs_a_minus_b(a,b)
7       print("The absolute value of {0}-{1} is {2}".format(a,b,result))
8
9
10  def abs_a_minus_b(x,y):
11      if x > y:
12          z = x - y
13      else:
14          z = y - x
15      return z
16
17
18  if __name__ == '__main__':
19      main()
```

The **traceback** prints the stack trace

Executing pdb

```python
# From command line
python -m pdb my_debug_example.py

# Running script
import pdb
pdb.set_trace()

# In interpreter
import pdb
import my_debug_example
pdb.run('my_debug_example.main()')



# Post-mortem
>>> import pdb
>>> import my_debug_example_fail
>>> my_debug_example_fail.main()
...
>>> pdb.pm()
```

Source

```python
1  import pdb, traceback, sys
2
3  def bombs():
4      a = []
5      print a[0]
6
7  if __name__ == '__main__':
8      try:
9          bombs()
10     except:
11         type, value, tb = sys.exc_info()
12         traceback.print_exc()
13         pdb.post_mortem(tb)
```

```python
# Execute the statement
pdb.run(statement, globals=None, locals=None)


# As above, and returns the value of the expression
pdb.runeval(expression, globals=None, locals=None)


# Call a function
pdb.runcall(function, *args, **kwds)


# Hardcode a breakpoint
pdb.set_trace()


# Enter debugger at post mortem given the traceback
#if none is given it takes the current exception
pdb.post_mortem(traceback=None)


# Enters pdb of last found traceback
pdb.pm()


# Can also do all of this manually by
pdb.Pdb(completekey='tab', stdin=None, stdout=None, skip=None, nosigint=False)
```

pdb commands

```
# Step into first possible occasion (can step into functions)
s(tep)

# Go to next line of expression
n(ext)

# Execute until lineno OR end of frame OR to a greater line than the current
# Convenient in for loops
unt(il) [lineno]

# Continue until function returns
r(eturn)

# Continue, only stop if bp is hit
c(ont(inue))
```

```
# Set next line to be executed (at the bottom most frame)
# Jump back and execute code again or skip part of code
# (Not always allowed)
j(ump) lineno

# List the source code around current line
l(ist) [first[, last]]

# List source for current function or frame
ll | longlist

# Print arguments of current function
a(rgs)

# Evaluate the expression, and print its value
# Similar to print(), but print() is a python function
p expression

# The same as above, but with prettyprinted function
pp expression

# Empty line: Previous command is repeated
#
```

```
# Set next line to be executed (at the bottom most frame)
# Jump back and execute code again or skip part of code
# (Not always allowed)
j(ump) lineno

# List the source code around current line
l(ist) [first[, last]]

# List source for current function or frame
ll | longlist

# Print arguments of current function
a(rgs)

# Evaluate the expression, and print its value
# Similar to print(), but print() is a python function
p expression

# The same as above, but with prettyprinted function
pp expression

# Empty line: Previous command is repeated
#
```

Demonstration

Play around with `programs/my_debug_example.py` file if you are at home, not attending the presentation

```python
1   #!/usr/bin/env python
2
3   """Example file to demonstrate pdb to debug a poorly written program"""
4   from a_random_function import foo
5
6   def main():
7       """Main function to be executed"""
8
9       a_string = "I'm a string"
10      a_dict   = {'a_key':[1,2,3], 'another_key':'Hey, Macarena!'}
11      a_number = 10
12
```

More commands

```
# Help and documentation
h(elp) [command]

# Print the stac trace (recent frame in bottom)
w(here)

# Move frames
d(own) [count]
u(p) [count]

# W/o arguments: List the breakpoints
# Make a breakpoint (where the debugger will stop)
# Honor the breakpoint if condition evaluates true
b(reak) [([filename:]lineno | function) [, condition]]

# Temporary breakpoint (removed when hit)
tbreak [([filename:]lineno | function) [, condition]]

# Clear breakpoints
cl(ear) [filename:lineno | bpnumber [bpnumber ...]]

# Disable breakpoints (can be re-enabled)
disable [bpnumber [bpnumber ...]]
```

```
# Enable bp number
enable [bpnumber [bpnumber ...]]

# Ignore bp count times (if cout>0)
ignore bpnumber [count]

# Set new condition for the bp
condition bpnumber [condition]

# Specify commands for bpnumber (or the last)
# end ends the command
# silent disables info about bp reached
commands [bpnumber]

# Print type of expression
whatis expression

# Try to get the source for the given object
source expression

# Stop and display the value of the expression if changed
# Somewhat similar to watchpoints in gdb
display [expression]

# Stop displaying in current frame
undisplay [expression]
```

Last functions

```
# Start interactive interpreter and load the globals and locals in the current
scope

# Stop with CTRL-D
interact

# Create an alias
alias [name [command]]

# Delete alias
unalias name

# Execute the (one-line) statement in the context of the current stack frame
# Exclamation point can be omitted unless the first word of the statement
# resembles a debugger command
! statement

# Execute from following
run [args ...]

# Restart
restart [args ...]

# Quit the debugger
q(uit)
```

If you debug alot

```python
# Enable bp number
enable [bpnumber [bpnumber ...]]

# Ignore bp count times (if cout>0)
ignore bpnumber [count]

# Set new condition for the bp
condition bpnumber [condition]

# Specify commands for bpnumber (or the last)
# end ends the command
# silent disables info about bp reached
commands [bpnumber]

# Print type of expression
whatis expression

# Try to get the source for the given object
source expression

# Stop and display the value of the expression if changed
# Somewhat similar to watchpoints in gdb
display [expression]

# Stop displaying in current frame
undisplay [expression]
```

For defeault commands when starting the pdb:

`~/.pdbrc` is read first, setting preferences for all debugging sessions.

```
# Show python help
alias ph !help(%1)
# Overridden alias
alias redefined p 'home definition'
```

`~/.pdbrc` is read from the current working directory, setting local preferences.

```
# Breakpoints
break 10
# Overridden alias
alias redefined p 'local definition'
```

```
$ python -m pdb pdb_function_arguments.py

Breakpoint 1 at .../pdb_function_arguments.py:10
> .../pdb_function_arguments.py(7)<module>()
-> import pdb

(Pdb) alias
ph = !help(%1)
redefined = p 'local definition'

(Pdb) break
Num Type           Disp Enb   Where
1   breakpoint     keep yes   at .../pdb_function_arguments.py:10
```

More info

Documentation
`https://docs.python.org/3.4/library/pdb.html`

Basic about debugging
`https://pythonconquerstheuniverse.wordpress.com/2009/09/`

More indepth about debugging `http://pymotw.com/2/pdb/`

Thank you for your attention!