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

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

```python
#!/usr/bin/env python

def main():
    a = 10
    b = 55
    result = abs_a_minus_b(a,b)
    print("The absolute value of {0}-{1} is {2}".format(a,b,result))


def abs_a_minus_b(x,y):
    if x > y:
        z = x - y
    else:
        z = y - x
    return z


if __name__ == '__main__':
    main()
```

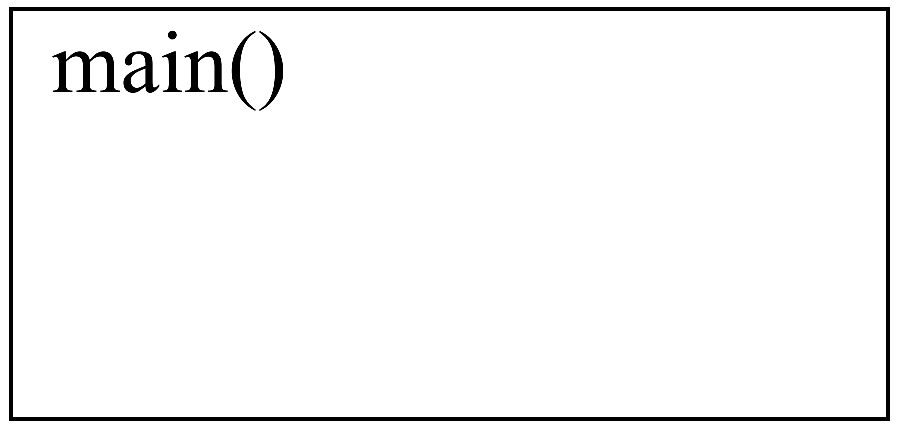
For future references: What is all this stack-frame fuzz?

```python
#!/usr/bin/env python

def main():
    a = 10
    b = 55
    result = abs_a_minus_b(a,b)
    print("The absolute value of {0}-{1} is {2}".format(a,b,result))


def abs_a_minus_b(x,y):
    if x > y:
        z = x - y
    else:
        z = y - x
    return z


if __name__ == '__main__':
    main()
```



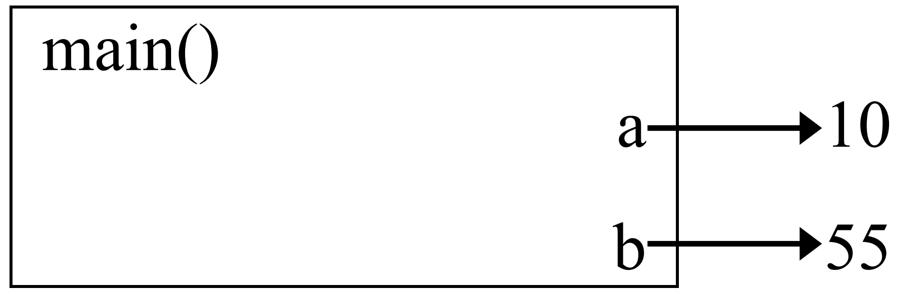
For future references: What is all this stack-frame fuzz?

```python
#!/usr/bin/env python

def main():
    a = 10
    b = 55
    result = abs_a_minus_b(a,b)
    print("The absolute value of {0}-{1} is {2}".format(a,b,result))


def abs_a_minus_b(x,y):
    if x > y:
        z = x - y
    else:
        z = y - x
    return z


if __name__ == '__main__':
    main()
```



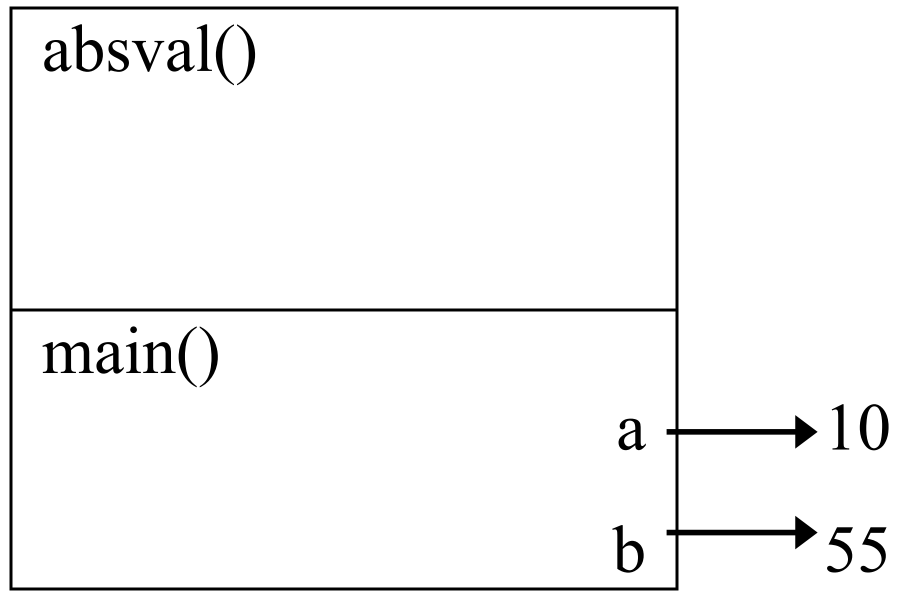
For future references: What is all this stack-frame fuzz?

```python
#!/usr/bin/env python

def main():
    a = 10
    b = 55
    result = abs_a_minus_b(a,b)
    print("The absolute value of {0}-{1} is {2}".format(a,b,result))


def abs_a_minus_b(x,y):
    if x > y:
        z = x - y
    else:
        z = y - x
    return z


if __name__ == '__main__':
    main()
```



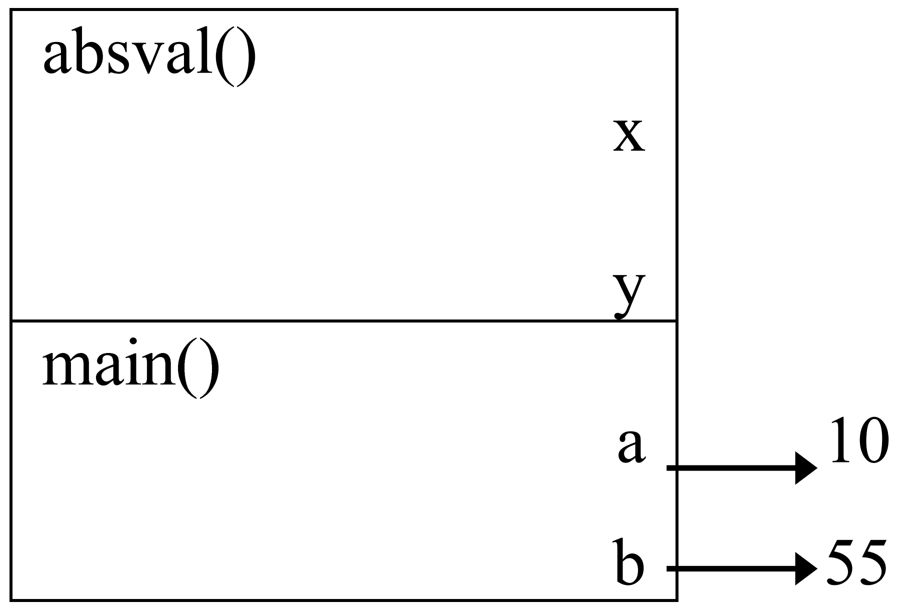
For future references: What is all this stack-frame fuzz?

```python
#!/usr/bin/env python

def main():
    a = 10
    b = 55
    result = abs_a_minus_b(a,b)
    print("The absolute value of {0}-{1} is {2}".format(a,b,result))


def abs_a_minus_b(x,y):
    if x > y:
        z = x - y
    else:
        z = y - x
    return z


if __name__ == '__main__':
    main()
```



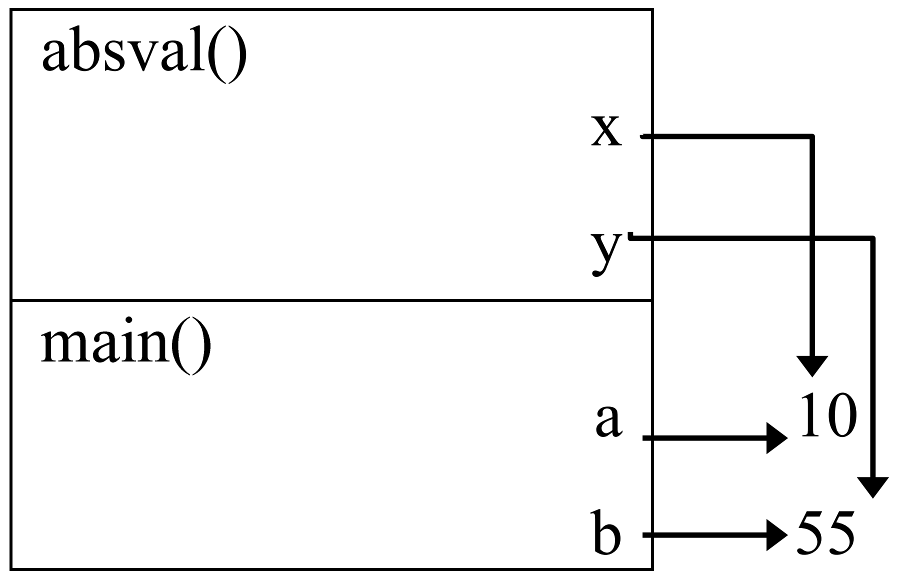
For future references: What is all this stack-frame fuzz?

```python
#!/usr/bin/env python

def main():
    a = 10
    b = 55
    result = abs_a_minus_b(a,b)
    print("The absolute value of {0}-{1} is {2}".format(a,b,result))


def abs_a_minus_b(x,y):
    if x > y:
        z = x - y
    else:
        z = y - x
    return z


if __name__ == '__main__':
    main()
```



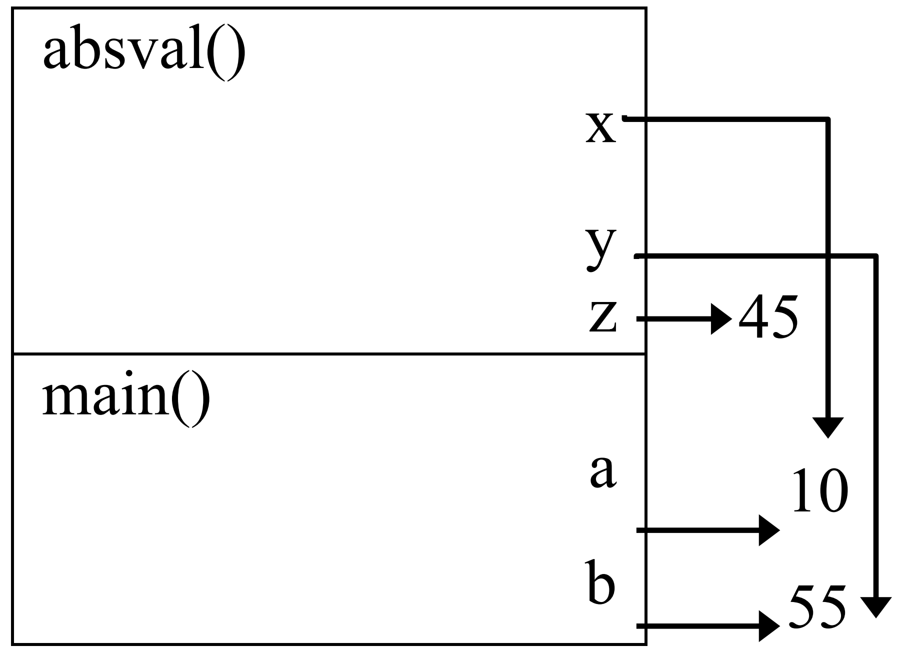
For future references: What is all this stack-frame fuzz?

```python
#!/usr/bin/env python

def main():
    a = 10
    b = 55
    result = abs_a_minus_b(a,b)
    print("The absolute value of {0}-{1} is {2}".format(a,b,result))


def abs_a_minus_b(x,y):
    if x > y:
        z = x - y
    else:
        z = y - x
    return z


if __name__ == '__main__':
    main()
```



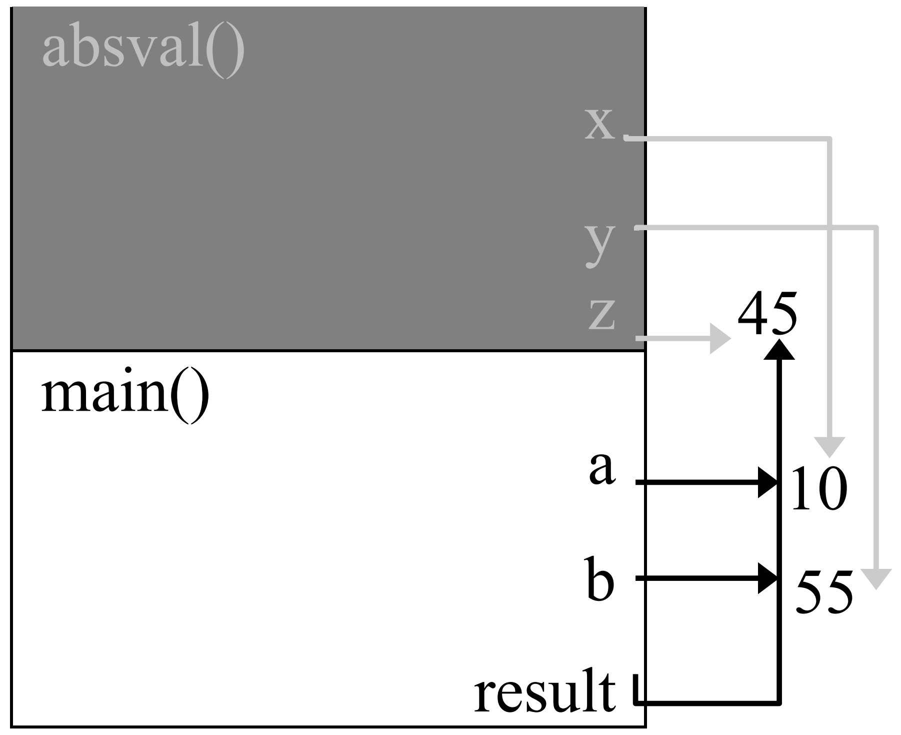
For future references: What is all this stack-frame fuzz?

```python
#!/usr/bin/env python

def main():
    a = 10
    b = 55
    result = abs_a_minus_b(a,b)
    print("The absolute value of {0}-{1} is {2}".format(a,b,result))


def abs_a_minus_b(x,y):
    if x > y:
        z = x - y
    else:
        z = y - x
    return z


if __name__ == '__main__':
    main()
```



Stack-frame

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

```python
#!/usr/bin/env python

def main():
    a = 10
    b = 55
    result = abs_a_minus_b(a,b)
    print("The absolute value of {0}-{1} is {2}".format(a,b,result))


def abs_a_minus_b(x,y):
    if x > y:
        z = x - y
    else:
        z = y - x
    return z


if __name__ == '__main__':
    main()
```

The **traceback** prints the stack trace

Executing pdb

Starting the debugger

```
# 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()
```

Post mortem running as script

Source

```
import pdb, traceback, sys

def bombs():
    a = []
    print a[0]

if __name__ == '__main__':
    try:
        bombs()
    except:
        type, value, tb = sys.exc_info()
        traceback.print_exc()
        pdb.post_mortem(tb)
```

Executing commands

```
# 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

Often used commands 1

```
# 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))
```

Often used commands 2

```
# 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
#
```

Often used commands 2

```
# 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
#!/usr/bin/env python

"""Example file to demonstrate pdb to debug a poorly written program"""
from a_random_function import foo

def main():
    """Main function to be executed"""

    a_string = "I'm a string"
    a_dict   = {'a_key':[1,2,3], 'another_key':'Hey, Macarena!'}
    a_number = 10

```

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 ...]]
```

Even more functions

```
# 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)
```

Thank you for your attention!