Using The Debugger In Python

“In industry there is a term for programmers who do not make extensive use of the debugger: unemployed.” – The Wizard of Oz
Background

Junior CSE
Member of DSE (Data Sci & Eng) Research Lab
Topics We’ll Cover

• Python 3’s built-in PDB Debugging module.
• Debugging, why we need it, modes of thought, using it for more than fixing.
The Online Reference

http://atbe.me

Click the Using The Debugger In Python - Workshop Reference link on the front page.
What Does a Program Do?

Whatever you tell it to.
What Happens When We Run Our code?

On a single core processor, only 1 instruction at any one moment in time can execute.
What’s An Instruction?

We don’t care atm. Just think it terms of lines of code. One line means doing 1 thing.

*Caveat: Functions* are special. Ask for one thing, they do many. If you’re confused *it’s okay*, it’ll start to clear up by the time we start debugging.
What’s A Debugger?

A debugger is a tool that can help you pause a program.

When a program is paused, it’s state is preserved and you can tinker around and change things as you’d like.
Why Do We Need Debuggers?

A lot of people might argue that we don’t need debuggers or that they can code just fine without them.

But in reality, debuggers are some of the most helpful tools when working on systems that have thousands or even hundreds of thousands of lines of code.

“The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.” – Brian Kernighan
Why Do We Need Debuggers? (continued)

Exception: We’re talking about scripting/systems languages. You can’t debug Markup code like html alone.
When To Use The Debugger?

Just a few of many examples:

When your code is broken (didn’t do what you expected it to).

When you want to test a rare condition for an if statement.

When you’re reading someone else’s code and want to understand the flow of execution. (I do this a lot)

When you want to see the values of your variables at a given moment; maybe they’re not what you expected them to be.
How Do We debug, correctly?

Debugging is a four step process:

• **Discovering the Bug.** Is there one? It is not always obvious that you have a bug. (Why?)

• **Isolating the Bug.** Where is it? Locate the part(s) of the code that is causing the bug.

• **Finding the Bug.** What exactly is wrong with the buggy code fragment?

• **Fixing the Bug.** How should the buggy fragment be rewritten?

Source: https://courses.engr.illinois.edu/ece390/books/labmanual/debugging.html
Debugging Python Code

We’re going to be using PDB

Yes, there are alternatives, I’ll briefly review those at the end.
Get a Linux Docker

We’re going to be using an online IDE provided by tutorialspoint.

http://atbe.me

Click the Using The Debugger In Python - Workshop Reference link on the front page.
Warning!!!!!

Uploading an old project will erase your exiting project at compileonlin, still you want to continue,

select tar.gz file which you had downloaded from compileonline using Download Project option

Project File: tutpoint_project.tar.gz

Upload  Cancel
Notice

Anywhere where you see ``` `, that is code to be run in the terminal.

For example

```  
ls  
```  

Means type `ls` in your terminal, and press enter.
Terminal

sh-4.3$
sh-4.3$
sh-4.3$ bash
bash-4.3$ echo "Hello, Hackers!"
Hello, Hackers!
bash-4.3$
Debugging – Starting the Debugger

Here is one way to start a program in PDB:

Run this to start our first debugging session:

```python
python3 -m pdb hello_world.py
```
Debugging – Starting the Debugger

bash-4.3$ python3 -m pdb hello_world.py
> /home/cg/root/hello_world.py(1)<module>()
--> print("Hello, World!")
(Pdb)
Debugging – Starting the Debugger

Starting the debugger at a specific point in your program only requires one line of code.
Debugging – Starting the Debugger

Starting the debugger at a specific point in your program only requires one line of code.

Add the following code to any Python file:

```python
import pdb; pdb.set_trace()
```
Debugging – All the commands – Use `help` Command

(Pdb) help

Documented commands (type help <topic>):

```none
EOF   c    d    h    list    q    rv    undisplay
a    cl   debug help ll    quit s    unt
alias clear disable ignore longlist r    source until
args commands display interact n    restart step up
b    condition down j    next return tbreak w
break cont enable jump p    retval u    whatis
bt   continue exit l    pp    run    unalias where
```
BREAKPOINTS
Breakpoints are key to efficient debugging
Breakpoints are key to efficient debugging

When the debugger sees a breakpoint, it will stop and prompt you for a command.
Let's try and set a breakpoint in the `time_activity_mapper.py` program.
b(break)
The `b(break)` command is used to see all the set breakpoints, and to set a breakpoint.
PDB Command → b

The `b`\text{(break) command} is used to see all the set breakpoints, and to set a breakpoint.

\textbf{b} \rightarrow \text{will list all the breakpoints}
PDB Command → b

The **b** command is used to see all the set breakpoints, and to set a breakpoint.

- **b** -> will list all the breakpoints
- **b #** -> will set a breakpoint at line #
cl(clear)
The **cl(clear)** command allows you to remove a breakpoint.
PDB Command → cl

The cl(clear) command allows you to remove a breakpoint.

cl # → Removes the breakpoint with number #.
l(list)
PDB Command \( \rightarrow \) I

The `I (list)` command is used to list the code surrounding our current line.
The `l (list)` command is used to list the code surrounding our current line.

Typing `l` more than once will print the next 11 lines.
The `l(list)` command is used to list the code surrounding our current line.

Typing `l` more than once will print the next 11 lines.

`l` -> prints the 11 surrounding lines
p(print/evaluate)
The `p(print)` command allows you to print the value of any of the variables in your program.
The *p(print)* command allows you to print the value of any expression.

\[ p \{\text{expression}\} \rightarrow \text{Executes and prints the value of some expression.} \]
NameError: name 'activity_time' is not defined
n(next)
The \texttt{n(\textit{next})} command tells the debugger to evaluate the current line, and move to the next.
c(continue)
The `c(continue)` command tells the debugger “continue executing until you encounter a breakpoint.”
The `c(continue)` command tells the debugger “continue executing until you encounter a breakpoint.”

Handy when you need to defer execution until you reach the breakpoint you set (example: loops).
STEPPING
Stepping is important when you’re moving around within some code in the debugger. (\textit{recursion}...cough cough)
Stepping is important when you’re moving around within some code in the debugger.

Before we learn to step, we need to be able to visualize what a step is.
Program starts

get_person_name is called

get_person_name running

get_person_name returns

get_person_age is called

get_person_age running

get_person_age returns

say_hello is called

say_hello running

year_to_sec is called

year_to_sec running

year_to_sec returns

say_hello returns

say_hello running

Program ends
Stepping is important when you’re moving around within some code in the debugger.

Step **over**.
Step **into**.
Step **out of**.
```python
python3 -m pdb function_parts.py
```
until
PDB Command → until

The `until` command tells the debugger “run the program until you hit this line number.”
The `until` command tells the debugger “run the program until you hit this line number.”

`until {line #}` → Runs code until line # is reached, and waits.
s(step)
PDB Command \( \rightarrow \) s(step)

The \texttt{s(step)} command will step into a function called on the current line.
The `s(step)` command will step into a function called on the current line.

If there isn’t a function on the current line, `s(step)` acts like `n(next)` and simply executes the current line and moves to the next line.
PDB Command → s(step)

The s(step) command will step into a function called on the current line.

If there isn’t a function on the current line, s(step) acts like n(next) and simply executes the current line and moves to the next line.

You can think of s(step) as a Step Into command.
r(return)
PDB Command → r(return)

The \texttt{r(return)} command continues the execution of the current function until the function returns.
PDB Command \( \rightarrow \text{r(return)} \)

The \texttt{r(return)} command continues the execution of the current function until the function returns.

You can think of \texttt{r(return)} as a \textit{Step Out Of} command.
n(next)...remember n?
PDB Command → n(next)

The *n(next)* command can be thought of as **Step Over**.
Commands:

- **b**(reak)  Set breakpoint
- **cl**(ear)  Delete breakpoint
- **l**(ist)  list 11 surrounding lines
- **p**(rint)  Evaluate and print code on current line
- **n**(ext)  “Step Over”
- **c**(ontinue)  Continue execution, stop at breakpoints
- **until**  Execute until given line number
STACK FRAMES
Think of Stack Frames as the list of steps taken as a program was running.
Think of Stack Frames as the list of steps taken as a program was running.

Moving through Stack Frames is very useful when working with recursive functions.
Think of Stack Frames as the list of steps taken as a program was running.

Moving through Stack Frames is very useful when working with recursive functions.

Python Note: All Python programs are started with the exec function. So exec always at the top of the Stack.
Think of Stack Frames as the steps taken while a program is running.

Stack frames tell us:

- What **function(s)** got us to where we are.
- Which **statement** from each function executed the current function.
- Where the code for each function is contained.
Stack Frames

Think of Stack Frames as the steps taken while a program is running.
Stack Frames

What does the Stack Frame here look like?
print a stack trace, with the most recent frame at the bottom. An arrow indicates the "current frame", which determines the context of most commands. 'bt' is an alias for this command.

/home/cg/root/function_parts.py(26)<module>()
-> say_hello(person_name, person_age)
   /home/cg/root/function_parts.py(20) say_hello()
-> print("Hello", name, "You are", year_to_sec(year_age), "seconds old.")
> /home/cg/root/function_parts.py(14) year_to_sec()
Stack Frames

Let's see what the Stack Frames look like for a recursive Factorial(n) function for $n = 4$
Stack Frames

```python3 –m pdb factorial_recursive.py
```
w(where)/bt
PDB Command → w(where)/bt

The \texttt{w(where)/bt} command will print the current Stack Frame of the program.
The \texttt{w(where)}/\texttt{bt} command will print the current Stack Frame of the program.

This is what Python's PDB module prints out when we use the \texttt{w} or \texttt{bt} command

(\textit{yes, they both do the same thing})
Python PDB Stack Trace – Factorial(4)

(Pdb) w
/usr/lib64/python3.4/bdb.py(431)run()
  -> exec(cmd, globals, locals)
  <string>(1)<module>()
/home/cg/root/factorial_recursive.py 8 <module>()
  print(Factorial(4))
/home/cg/root/factorial_recursive.py 6 Factorial()
  return n * Factorial(n - 1)
/home/cg/root/factorial_recursive.py 6 Factorial()
  -> return n * Factorial(n - 1)
/home/cg/root/factorial_recursive.py 6 Factorial()
  -> return n * Factorial(n - 1)
/home/cg/root/factorial_recursive.py 6 Factorial()
  -> return n * Factorial(n - 1)
/home/cg/root/factorial_recursive.py 6 Factorial()
  -> return n * Factorial(n - 1)
/home/cg/root/factorial_recursive.py 6 Factorial()
  -> return n * Factorial(n - 1)
/home/cg/root/factorial_recursive.py 1 Factorial()
  -> def Factorial(n):
Stack Frames

Let's see what the Stack Frames look like for a recursive Factorial function for $n = 4$
exec()

Factorial(4)

Factorial(3)

Factorial(2)

Factorial(1)

Factorial(0)
exec()

Factorial(4)

Factorial(3)

Factorial(2)

Factorial(1)

Factorial(0)

exec() returns 24
What if Each **Spike** Represented as a Block Of Memory Being Used (The Stacks)
Stack Frames

You can move up and down in a stack frame.

This means moving up to a function that called you, or moving into a function you called.
u(up)
PDB Command → u(up)

The **u(up)** command will move your program up in the stack frame.
The \texttt{u(up)} command will move your program \texttt{up} in the stack frame.

\texttt{u(up)} could also be used as a \textbf{Step Out Of}.
The \texttt{u(up)} command will move your program \texttt{up} in the stack frame.

\texttt{u(up)} functions as a \texttt{Step Out Of} as well.

\texttt{u(up) \# Move up the stack frame \# frames.}
d(down)
The \texttt{d(down)} command will move your program \texttt{down} in the stack frame.
The **d(down)** command will move your program **down** in the stack frame.

`d(down)#` → Move down the stack frame # frames.
Stack Frames

Look into Stack Frames some more if you can.

There are some more things to be learned about Stack Frames. Check out the recommended readings.
Other useful PDB Commands

- **source obj** – prints the source code of obj.
- **interact** – starts a python REPL who can access any variables the current line of code could have.
- **restart** – restart the current debugged program and debugging session.
- **whatis arg** – print the type of the arg.
- **longlist** – print the whole source code for the current function or frame.
Most of the concepts we’ve learned up till now can be used with debuggers for other languages.
Most of the concepts we’ve learned up till now can be used with debuggers for other languages.

lldb/gdb – C/C++

jdb – Java
debugger (browser) – Javascript

pudb – Python

...
Pudb – A visual Python Debugger
# This program will be a complex execution path with many functions

```python
# Debug in Spyder – Starting The Debugger

def get_person_name():
    print("get_person_name called")
    return input("What's your name?: ")

def get_person_age():
    print("get_person_age called")
    return input("How old are you?: ")

def year_to_sec(age_years):
    print("year_to_sec called")
    # 356 days in year, 24 hours in day, 60 min in hour, 60 sec in min
    return int(age_years.strip()) * 365 * 24 * 60 * 60

def say_hello(name, year_age):
    print("say_hello called")
```
Debugging in Spyder – Starting Options
Debugging in Spyder – Sample Output
Debugging in Spyder – Setting Breakpoints

```python
# This program will be used to demonstrate a debug run with many functions executing one after another.

def get_person_name():
    print("get_person_name called")
    return input("What's your name?: ")

def get_person_age():
    print("get_person_age called")
    return input("How old are you?: ")

def year_to_sec(age_years):
    print("year_to_sec called")
    # 365 days in year, 24 hours in day, 60 min in hour, 60 sec in min
    return int(age_years.strip()) * 355 * 24 * 60 * 60

def say_hello(name, year_age):
    print("say_hello called")
    print("Hello", name, "You are", year_to_sec(year_age), "seconds old.")

person_name = get_person_name()
pass
person_age = get_person_age()
pass
say_hello(person_name, person_age)
```
That’s All For Tonight, Folks

atbe
tht_roony
atbe.me

Questions