CH 15 Debugging Techniques using pdb

When you write software, there is always a change having bugs in it: it might not work as expected or raises un-expected error. You can always use a print statement, which can put in between the python statement to print and debug, but it is not always a good idea. Python provide a better approach for python debugging, named python **d**e**b**ugger or pdb.

This this section, we will understand how pdb help us to debug a python program, there are two way to debugging a python program:

* Putting  hard-code breakpoint at a given point in a program using **pdb.set\_trace()** and
* Run a Script using **python -m pdb option**

**15.1 Getting started — pdb.set\_trace()**

Let’s start a simple program for debugging

def sqr(a):

print 'squareing..',a

return a \* a

def addSqr(x,y):

x = sqr(x)

y = sqr(y)

print 'adding...'

return x +y

a = 10

b = 20

c = addSqr(a,b)

print c

In this program, we define two function sqr(x) and addSqr(), used to calculate the square of a number and adding the squares of two number. In main, I am trying to find out the square sum of 10 and 20.

If you run the program the output will be :

squareing.. 10

squareing.. 20

adding...

500

To Like to start debugging of the program just above declaring a =10. To do that we need to add <pdb.set\_trace()> just above declearing a. The modified code is like this:

import pdb

def sqr(a):

print 'squareing..',a

return a \* a

def addSqr(x,y):

x = sqr(x)

y = sqr(y)

print 'adding...'

return x +y

pdb.set\_trace()

a = 10

b = 20

c = addSqr(a,b)

print c

Now run the program again by typing < **python SumOfSqr.py>.** When your program encounters the line with pdb.set\_trace() it will start tracing. That is, it will stop, and display the “current statement” (that is, the line that will execute next, here a = 10 ) and wait for your input. You will see the pdb prompt, which looks like this:

**> c:\users\dipankar.dutta\desktop\test.py(11)<module>()**

**-> a = 10**

**(Pdb)**

This means we are pdb command shell and we can debug and control the flow of the program.

Pdb provide following command to debug our code, given in the following table. We will discuss some of these while debugging this program.

|  |  |
| --- | --- |
| **h(elp) [command]** | **Without argument, print the list of available commands.** |
| **w(here)** | **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.** |
| **d(own)** | **Move the current frame one level down in the stack trace (to an newer frame).** |
| **u(p)** | **Move the current frame one level up in the stack trace (to a older frame).** |
| **b(reak) [[filename:]lineno|function[, condition]]** | **With a lineno argument, set a break there in the current file.** |
| **tbreak [[filename:]lineno|function[, condition]]** | **Temporary breakpoint, which is removed automatically when it is first hit.** |
| **cl(ear) [bpnumber [bpnumber ...]]** | **With a space separated list of breakpoint numbers, clear those breakpoints.** |
| **disable [bpnumber [bpnumber ...]]** | **Disables the breakpoints given as a space separated list of breakpoint numbers. Disabling a breakpoint means it cannot cause the program to stop execution, but unlike clearing a breakpoint, it remains in the list of breakpoints and can be (re** |
| **)enabled.** |  |
| **enable [bpnumber [bpnumber ...]]** | **Enables the breakpoints specified.** |
| **ignore bpnumber [count]** | **Sets the ignore count for the given breakpoint number.** |
| **condition bpnumber [condition]** | **Condition is an expression which must evaluate to true before the breakpoint is honored.** |
| **s(tep)** | **Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).** |
| **n(ext)** | **Continue execution until the next line in the current function is reached or it returns. (The difference between "next" and "step" is that "step" stops inside a called function, while "next" executes called functions at (nearly) full speed, only stopping at the next line in the current function.)** |
| **r(eturn)** | **Continue execution until the current function returns.** |
| **c(ont(inue))** | **Continue execution, only stop when a breakpoint is encountered.** |
| **j(ump) lineno** | **Set the next line that will be executed. Only available in the bottom** |
| **most frame. This lets you jump back and execute code again, or jump forward to skip code that you don't want to run.** |  |
| **l(ist) [first[, last]]** | **List source code for the current file.** |
| **a(rgs)** | **Print the argument list of the current function.** |
| **p expression** | **Evaluate the expression in the current context and print its value.** |
| **pp expression** | **Like the "p" command, except the value of the exception is pretty printed using the pprint module.** |
|  |  |
| **[!]statement** | **The exclamation point can be used if the first word of the statement resembles a debugger command.** |
| **q(uit)** | **Quit from the debugger. The program being executed is aborted.** |

As you can see, we are in pdb prompt and going to execute **a = 10** statement

**> c:\users\dipankar.dutta\desktop\test.py(11)<module>()**

**-> a = 10**

**(Pdb)**

Let’s go through the following steps and see what can we do using pdb?

**STEP1: [Getting Help]** Press <h> go get help of pdb here.

**(Pdb) h**

**Documented commands (type help <topic>):**

**========================================**

**EOF bt cont enable jump pp run unt**

**a c continue exit l q s until**

**alias cl d h list quit step up**

**args clear debug help n r tbreak w**

**b commands disable ignore next restart u whatis**

**break condition down j p return unalias where**

**Miscellaneous help topics:**

**==========================**

**exec pdb**

**Undocumented commands:**

**======================**

**retval rv**

**STEP2: [See the source code]** Press <l> to see the source code segment and an arrow on current statements:

**(Pdb) l**

**6 x = sqr(x)**

**7 y = sqr(y)**

**8 print 'adding...'**

**9 return x +y**

**10 pdb.set\_trace()**

**11 -> a = 10**

**12 b = 20**

**13 c = addSqr(a,b)**

**14 print c**

**15**

**[EOF]**

**STEP3: [print the local data]** Press <n> to execute current statement and go to next line. Also type <p a> to see whether “a” get its value or not ! When you press <p b>, you will get an error as ‘b” is not yet defined.

(**Pdb) n**

**> c:\users\dipankar.dutta\desktop\test.py(12)<module>()**

**-> b = 20**

**(Pdb) p a**

**10**

**(Pdb) p b**

**\*\*\* NameError: NameError("name 'b' is not defined",)**

**STEP4: [Go to next steps**] Press <n> to execute current statement and go to next line. Also type <p a ,b > to see value of a and b.

**(Pdb) n**

**> c:\users\dipankar.dutta\desktop\test.py(13)<module>()**

**-> c = addSqr(a,b)**

**(Pdb) p a ,b**

**(10, 20)**

**STEP5: [See where I am]** Just like <l > we van have another command “where” Press <w> to see the current stack.

**(Pdb) w**

**<string>(1)<module>()**

**c:\python27\lib\idlelib\run.py(116)main()**

**-> ret = method(\*args, \*\*kwargs)**

**c:\python27\lib\idlelib\run.py(324)runcode()**

**-> exec code in self.locals**

**> c:\users\dipankar.dutta\desktop\test.py(13)<module>()**

**-> c = addSqr(a,b)**

**STEP6: [Let’s Step into the function]** Now you can see we are going to execute addSqr(). If you press <n> it will execute and go to the next line. But I want to go into the function, and see what’s happening inside the function. To do that you can press **<s>** to step into the function. Press <l>, to see the current code block. Press <a> to see the passing argument in this function.

(**Pdb) s**

**--Call--**

**> c:\users\dipankar.dutta\desktop\test.py(5)addSqr()**

**-> def addSqr(x,y):**

**(Pdb) l**

**1 import pdb**

**2 def sqr(a):**

**3 print 'squareing..',a**

**4 return a \* a**

**5 -> def addSqr(x,y):**

**6 x = sqr(x)**

**7 y = sqr(y)**

**8 print 'adding...'**

**9 return x +y**

**10 pdb.set\_trace()**

**11 a = 10**

**(Pdb) a**

**x = 10**

**y = 20**

**STEP7: [Pressing ENTER make your life simpler]** Press <s> <ENTER> to go into the sqr() function . <ENTER> basically repeat the previous executed command.

(**pdb) s**

**(Pdb) <ENTER>**

**--Call--**

**> c:\users\dipankar.dutta\desktop\test.py(2)sqr()**

**-> def sqr(a):**

**(Pdb) l**

**1 import pdb**

**2 -> def sqr(a):**

**3 print 'squareing..',a**

**4 return a \* a**

**5 def addSqr(x,y):**

**6 x = sqr(x)**

**7 y = sqr(y)**

**8 print 'adding...'**

**9 return x +y**

**10 pdb.set\_trace()**

**11 a = 10**

**(Pdb) n**

**> c:\users\dipankar.dutta\desktop\test.py(3)sqr()**

**-> print 'squareing..',a**

**(Pdb) arg**

**a = 10**

**STEP8: [ Back-trace the date and move up and down in calling stack ]** We can see the full backtrack by typing <bt> . It will display full backtrack of the calling stack. We can also use <u> to go and debug up stack and <d> to come down to debug down stacks .

**(Pdb) l**

**1 import pdb**

**2 def sqr(a):**

**3 -> print 'squareing..',a**

**4 return a \* a**

**5 def addSqr(x,y):**

**6 x = sqr(x)**

**7 y = sqr(y)**

**8 print 'adding...'**

**9 return x +y**

**10 pdb.set\_trace()**

**11 a = 10**

**(Pdb) bt**

**<string>(1)<module>()**

**c:\python27\lib\idlelib\run.py(116)main()**

**-> ret = method(\*args, \*\*kwargs)**

**c:\python27\lib\idlelib\run.py(324)runcode()**

**-> exec code in self.locals**

**c:\users\dipankar.dutta\desktop\test.py(13)<module>()**

**-> c = addSqr(a,b)**

**c:\users\dipankar.dutta\desktop\test.py(6)addSqr()**

**-> x = sqr(x)**

**> c:\users\dipankar.dutta\desktop\test.py(3)sqr()**

**-> print 'squareing..',a**

**(Pdb) u**

**> c:\users\dipankar.dutta\desktop\test.py(6)addSqr()**

**-> x = sqr(x)**

**(Pdb) p x**

**10**

**(Pdb) d**

**> c:\users\dipankar.dutta\desktop\test.py(3)sqr()**

**-> print 'squareing..',a**

**(Pdb) p a**

**10**

**STEP9 : [ Returning back from function ]** If you want to return from a function you can press <r> to do that. It will let you go back from current stack (SQR() ) to caller function.

(**Pdb) r**

**--Retur**

**n--**

**> c:\users\dipankar.dutta\desktop\test.py(4)sqr()->100**

**-> return a \* a**

**(Pdb) l**

**1 import pdb**

**2 def sqr(a):**

**3 print 'squareing..',a**

**4 -> return a \* a**

**5 def addSqr(x,y):**

**6 x = sqr(x)**

**7 y = sqr(y)**

**8 print 'adding...'**

**9 return x +y**

**10 pdb.set\_trace()**

**11 a = 10**

Press another <r> to return from addSQR() to main function.

**(Pdb) r**

**> c:\users\dipankar.dutta\desktop\test.py(9)addSqr()**

**-> return x +y**

**(Pdb) p x,y**

**(100, 400)**

**STEP10: [Modify as you wish]:** As you can see, I am currently back to main function and going to execute x +y statement. Yes! We can do execution of expression or change some value at runtime using pdb. The value of x is 100, we can set the value as 150 by following command.

**(Pdb) x= 150**

**(pdb)x**

**150**

We can also create a new variable at run time or do some expression at run time as well. Let’s define another variable z having value 10000

(**Pdb) p z**

**\*\*\* NameError: NameError("name 'z' is not defined",)**

**(Pdb) p z =10000**

**\*\*\* SyntaxError: SyntaxError('invalid syntax', ('<string>', 1, 3, 'z =10000'))**

**(Pdb) z =1000**

**(Pdb) z**

**1000**

**(Pdb) print z**

**1000**

**STEP11: [Continue…]** At-last you completed your debug and want to continue, press <c> to do that. It will complete the rest of the program and print the output.

**(Pdb) c**

**550**

**>>>**

Note that the answer id 550, this is because we change the value of a by 150 at runtime. ☺

**15.2 Using python -m pdb option.**

If you don’t want to hard-code **pdb.set\_trace()** in your source code, there is another way to debug your python program. You can run the following command to get into python shell, set the break print and continue by typing <c>.

C:\Users\dipankar.dutta\Desktop>**python -m pdb test**

> c:\users\dipankar.dutta\desktop\test(1)<module>()

-> def sqr(a):

**(Pdb) b 10**

Breakpoint 1 at c:\users\dipankar.dutta\desktop\test:10

**(Pdb) c**

> c:\users\dipankar.dutta\desktop\test(10)<module>()

-> a = 10

**(Pdb) n**

> c:\users\dipankar.dutta\desktop\test(11)<module>()

-> b = 20

(Pdb)

As you can see, we are in pdb shell and we are ready for debugging.