Department of Computer Science and Engineering
PES University, Bangalore, India

Lecture Notes
Python for Computational Problem Solving
UE23CS151A

Lecture #91

Python Debugger

By,
Prof. Sindhu R Pai,
Anchor, PCPS - 2023
Assistant Professor
Dept. of CSE, PESU

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Centers, Former Chairperson, CSE, PES University)
Prof. Chitra G M(Asst. Prof, Dept. of CSE, PCPS Anchor – 2022)

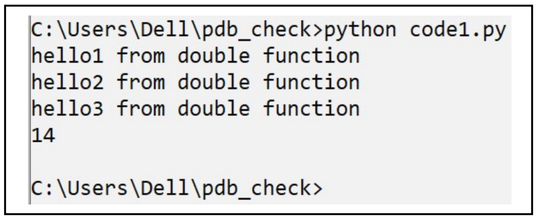
Introduction

A **debugger is a program** that can help you **find out what is going on in a computer program.** Developers can stop the execution at any prescribed line number, print out variables, continue execution, stop again, execute statements one by one, and repeat such actions until the program is tracked down find the abnormal behavior and found bugs. **Debugging in Python is facilitated by pdb module** (python debugger) which comes built-in to the Python standard library. It is actually defined as the class **Pdb which internally makes use of bdb(basic debugger functions)** and **cmd (support for line-oriented command interpreters) modules.** The module **pdb defines an interactive source code debugger for Python programs.**

- It supports setting (conditional) **breakpoints** and **single stepping** at the source line level, **inspection of stack frames**, **source code listing**, and evaluation of arbitrary Python code in the context of any stack frame.
- Also, supports **post-mortem debugging** and **can be called under program control**.

Consider this program:

```
def double(x):
   print("hello1 from double function")
   print("hello2 from double function")
   print("hello3 from double function")
   return x*2
print(double(7))
```



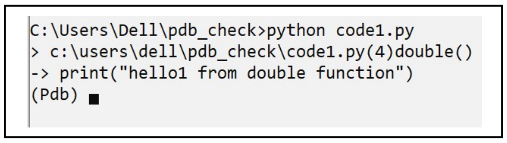
```
C:\Users\Dell\pdb_check>python code1.py
hello1 from double function
hello2 from double function
hello3 from double function
14

C:\Users\Dell\pdb_check>
```

If we want to debug this step by step using pdb, there are different methods to invoke it.

Method_1: Using **import pdb; pdb.set_trace()**

set_trace() : Enters the debugger at the calling frame. Useful to hardcode a breakpoint at a given point in a program, even if the code is not otherwise being debugged.

```
import pdb
def double(x):
   pdb.set_trace()
   print("hello1 from double function")
   print("hello2 from double function")
```



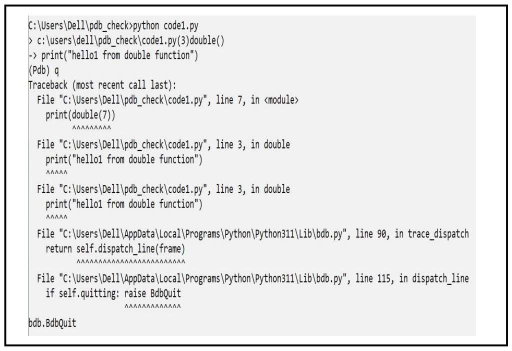
```
C:\Users\Dell\pdb_check>python code1.py
> c:\users\dell\pdb_check\code1.py(4)double()
-> print("hello1 from double function")
(Pdb) ■
```

```
    print("hello3 from double function")
    return x*2
print(double(7))
```

Note: In the output, (Pdb) is the debugger prompt. It is the indicator that you are in debug mode. Press q or quit to come out of the debugger mode.

Method_2: Using **breakpoint();**

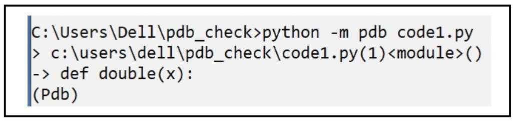
```
def double(x):
    breakpoint()  # same as import pdb; pdb.set_trace()
    print("hello1 from double function")
    print("hello2 from double function")
    print("hello3 from double function")
    return x*2
print(double(7))
```



```
C:\Users\Dell\pdb_check>python code1.py
> c:\users\dell\pdb_check\code1.py(3)double()
-> print("hello1 from double function")
(Pdb) q
Traceback (most recent call last):
  File "C:\Users\Dell\pdb_check\code1.py", line 7, in <module>
    print(double(7))
          ^^^^^^^^^
  File "C:\Users\Dell\pdb_check\code1.py", line 3, in double
    print("hello1 from double function")
    ^^^^^
  File "C:\Users\Dell\pdb_check\code1.py", line 3, in double
    print("hello1 from double function")
    ^^^^^
  File "C:\Users\Dell\AppData\Local\Programs\Python\Python311\Lib\bdb.py", line 90, in trace_dispatch
    return self.dispatch_line(frame)
           ^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Dell\AppData\Local\Programs\Python\Python311\Lib\bdb.py", line 115, in dispatch_line
    if self.quitting: raise BdbQuit
                      ^^^^^^^^^^^^^
bdb.BdbQuit
```

Method_3: Invoking it from command line using –m option

```
def double(x):
    print("hello1 from double function")
    print("hello2 from double function")
    print("hello3 from double function")
    return x*2
print(double(7))
```



```
C:\Users\Dell\pdb_check>python -m pdb code1.py
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb)
```

Note: -m means begin the debugger on the specified file. Press exit() to come out of this mode.

Typical usage of pdb is to execute a statement under the control of the DeBugger.

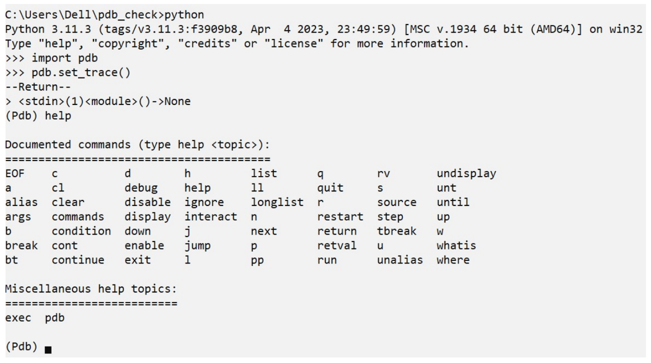
run() - > Executes the statement under debugger control. Debugger prompt appears before any statement is executed. Now, you can use any commands [Covered in next section] which pdb understands.

```
>>> def fun1():
...     print("Hello")
...     print("this is the use of debugger")
...
>>> import pdb
>>> pdb.run("fun1")
> <string>(1)<module>()
(Pdb)
```

Note: run() and set_trace() are the aliases for instantiating the pdb class

DeBugger Commands:

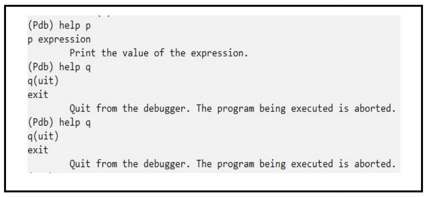
The commands recognized by the debugger are listed below. Most commands can be abbreviated to one or two letters. Example: h(elp) means that either **h** or **help** can be used to enter the help.



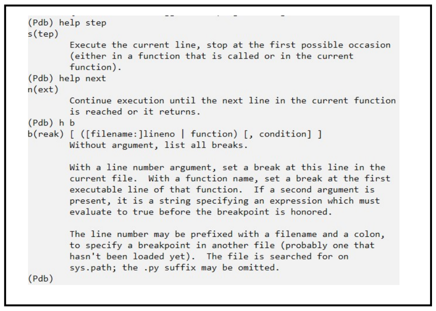
```
C:\Users\Dell\pdb_check>python
Python 3.11.3 (tags/v3.11.3:f3909b8, Apr  4 2023, 23:49:59) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import pdb
>>> pdb.set_trace()
--Return--
> <stdin>(1)<module>()->None
(Pdb) help

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

Miscellaneous help topics:
==========================
exec  pdb

(Pdb) ■
```

The arguments or options to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets ([]) in the command syntax; the square brackets must not be typed. Alternatives in the command syntax are separated by a vertical bar (|). Entering a blank line repeats the last command entered. Exception: list command. Commands that the debugger doesn't recognize are assumed to be Python statements and are executed in the context of the program being debugged. **Python statements** can also be prefixed with an **exclamation point (`!`).**

Can make use of **help <topic>** for more information on a given command.



```
(Pdb) help p
p expression
        Print the value of the expression.
(Pdb) help q
q(uit)
exit
        Quit from the debugger. The program being executed is aborted.
(Pdb) help q
q(uit)
exit
        Quit from the debugger. The program being executed is aborted.
```



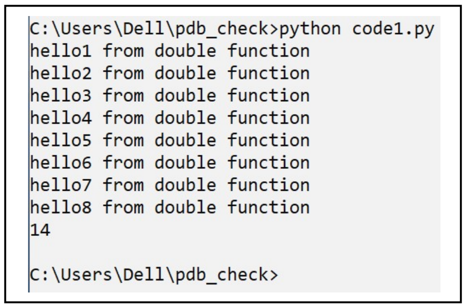
```
(Pdb) help step
s(tep)
        Execute the current line, stop at the first possible occasion
        (either in a function that is called or in the current
        function).
(Pdb) help next
n(ext)
        Continue execution until the next line in the current function
        is reached or it returns.
(Pdb) h b
b(reak) [ ([filename:]lineno | function) [, condition] ]
        Without argument, list all breaks.

        With a line number argument, set a break at this line in the
        current file.  With a function name, set a break at the first
        executable line of that function.  If a second argument is
        present, it is a string specifying an expression which must
        evaluate to true before the breakpoint is honored.

        The line number may be prefixed with a filename and a colon,
        to specify a breakpoint in another file (probably one that
        hasn't been loaded yet).  The file is searched for on
        sys.path; the .py suffix may be omitted.
(Pdb)
```

Try help on these commands!

up, down, break, enable, disable, cl, pp

Consider the below simple coding example. The output is also shown.

```
def double(x):
   a = 6
   print("hello1 from double function")
   print("hello2 from double function")
   b = 10
   print("hello3 from double function")
   c = 16
   print("hello4 from double function")
   print("hello5 from double function")
   d = 20
   print("hello6 from double function")
   print("hello7 from double function")
   print("hello8 from double function")
   e = 25
   return x*2
print(double(7))
```

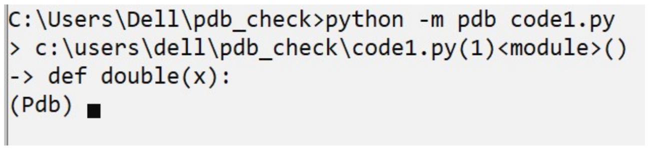


```
C:\Users\Dell\pdb_check>python code1.py
hello1 from double function
hello2 from double function
hello3 from double function
hello4 from double function
hello5 from double function
hello6 from double function
hello7 from double function
hello8 from double function
14

C:\Users\Dell\pdb_check>
```

Now, let us try to understand a few pdb associated commands in detail using the above code line by line.

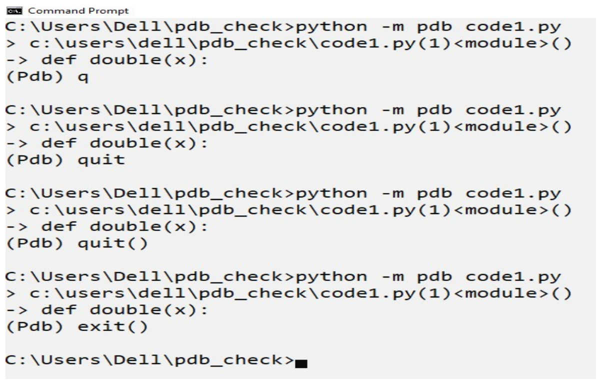
1. Using a module pdb from the command line on this file. **–m is the option with python.**

- > represents the line which will be executed next if you use next or step command.



```
C:\Users\Dell\pdb_check>python -m pdb code1.py
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb)
```

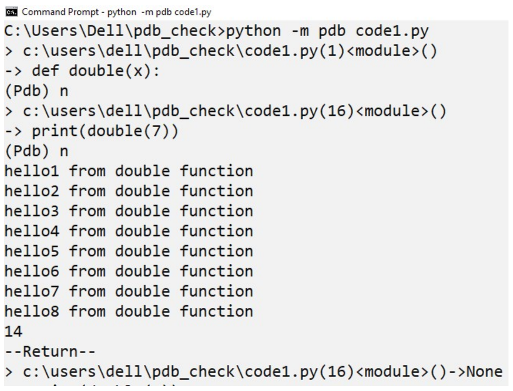
2. Different ways of exiting from the pdb mode are as below.



```
C:\Users\Dell\pdb_check>python -m pdb code1.py
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb) q

C:\Users\Dell\pdb_check>python -m pdb code1.py
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb) quit

C:\Users\Dell\pdb_check>python -m pdb code1.py
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb) quit()

C:\Users\Dell\pdb_check>python -m pdb code1.py
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb) exit()

C:\Users\Dell\pdb_check>
```

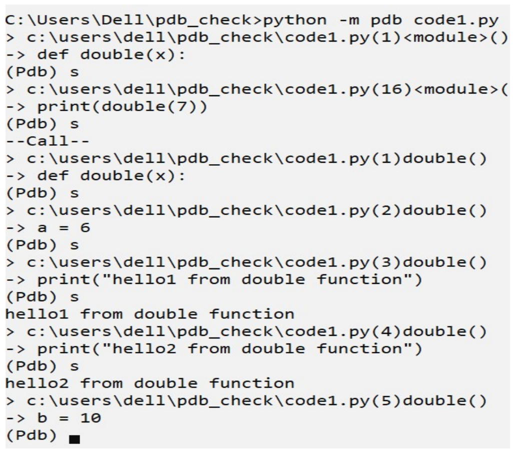
3. Let us try few commands.

next/n: Continue execution until the next line in the current function is reached or it returns



```
C:\Users\Dell\pdb_check>python -m pdb code1.py
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb) n
> c:\users\dell\pdb_check\code1.py(16)<module>()
-> print(double(7))
(Pdb) n
hello1 from double function
hello2 from double function
hello3 from double function
hello4 from double function
hello5 from double function
hello6 from double function
hello7 from double function
hello8 from double function
14
--Return--
> c:\users\dell\pdb_check\code1.py(16)<module>()->None
```

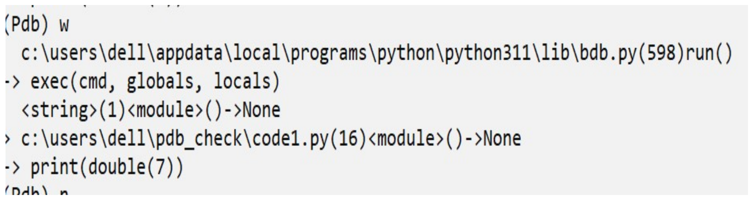
step/s: Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).



```
C:\Users\Dell\pdb_check>python -m pdb code1.py
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb) s
> c:\users\dell\pdb_check\code1.py(16)<module>(
-> print(double(7))
(Pdb) s
--Call--
> c:\users\dell\pdb_check\code1.py(1)double()
-> def double(x):
(Pdb) s
> c:\users\dell\pdb_check\code1.py(2)double()
-> a = 6
(Pdb) s
> c:\users\dell\pdb_check\code1.py(3)double()
-> print("hello1 from double function")
(Pdb) s
hello1 from double function
> c:\users\dell\pdb_check\code1.py(4)double()
-> print("hello2 from double function")
(Pdb) s
hello2 from double function
> c:\users\dell\pdb_check\code1.py(5)double()
-> b = 10
(Pdb)
```

Note: 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"

where/w: Print a stack trace, with the most recent frame at the bottom. An arrow (>) indicates the current frame.



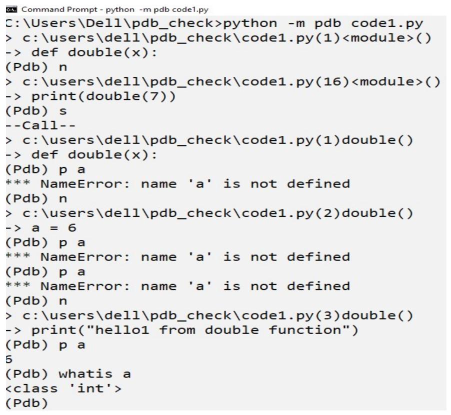
```
(Pdb) w
  c:\users\dell\appdata\local\programs\python\python311\lib\bdb.py(598)run()
-> exec(cmd, globals, locals)
  <string>(1)<module>()->None
> c:\users\dell\pdb_check\code1.py(16)<module>()->None
-> print(double(7))
```

print/p: Evaluates the expression in the current context and print its value.

Usage: p expression

whatis: Print the type of *expression*.

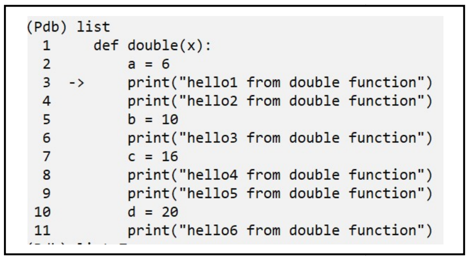
Usage: whatis expression



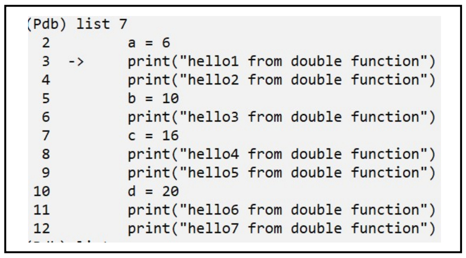
Command Prompt - python -m pdb code1.py

```
C:\Users\Dell\pdb_check>python -m pdb code1.py
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb) n
> c:\users\dell\pdb_check\code1.py(16)<module>()
-> print(double(7))
(Pdb) s
--Call--
> c:\users\dell\pdb_check\code1.py(1)double()
-> def double(x):
(Pdb) p a
*** NameError: name 'a' is not defined
(Pdb) n
> c:\users\dell\pdb_check\code1.py(2)double()
-> a = 6
(Pdb) p a
*** NameError: name 'a' is not defined
(Pdb) p a
*** NameError: name 'a' is not defined
(Pdb) n
> c:\users\dell\pdb_check\code1.py(3)double()
-> print("hello1 from double function")
(Pdb) p a
5
(Pdb) whatis a
<class 'int'>
(Pdb)
```

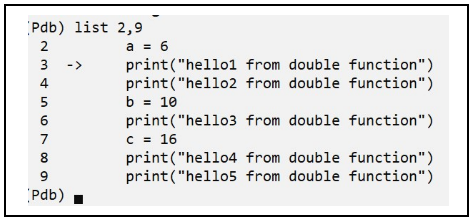
list/l : Lists source code for the current file. Without arguments, it shows 11 lines around the current line or continue the previous listing. With one argument, list 11 lines starting at that line. With two arguments, list the given range. If the second argument is less than the first, it is a count.



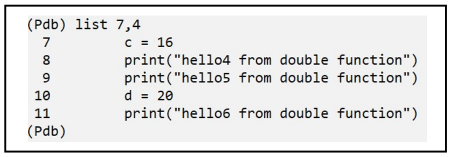
```
(Pdb) list
  1     def double(x):
  2         a = 6
  3  ->     print("hello1 from double function")
  4         print("hello2 from double function")
  5         b = 10
  6         print("hello3 from double function")
  7         c = 16
  8         print("hello4 from double function")
  9         print("hello5 from double function")
 10         d = 20
 11         print("hello6 from double function")
```



```
(Pdb) list 7
  2         a = 6
  3  ->     print("hello1 from double function")
  4         print("hello2 from double function")
  5         b = 10
  6         print("hello3 from double function")
  7         c = 16
  8         print("hello4 from double function")
  9         print("hello5 from double function")
 10         d = 20
 11         print("hello6 from double function")
 12         print("hello7 from double function")
```

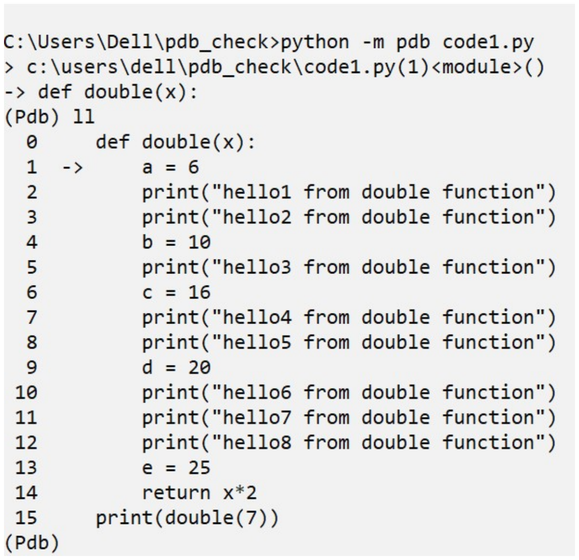


```
(Pdb) list 2,9
  2         a = 6
  3  ->     print("hello1 from double function")
  4         print("hello2 from double function")
  5         b = 10
  6         print("hello3 from double function")
  7         c = 16
  8         print("hello4 from double function")
  9         print("hello5 from double function")
(Pdb)
```



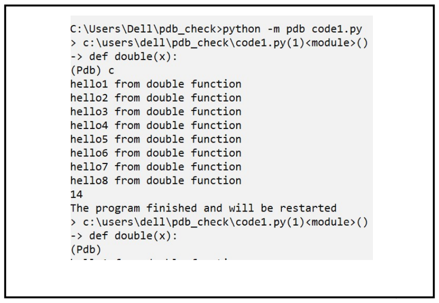
```
(Pdb) list 7,4
  7         c = 16
  8         print("hello4 from double function")
  9         print("hello5 from double function")
 10         d = 20
 11         print("hello6 from double function")
(Pdb)
```

longlist/ll: Lists source code for the current function or frame.

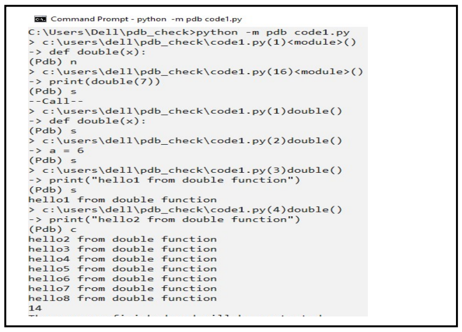


```
C:\Users\Dell\pdb_check>python -m pdb code1.py
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb) ll
  0     def double(x):
  1  ->     a = 6
  2         print("hello1 from double function")
  3         print("hello2 from double function")
  4         b = 10
  5         print("hello3 from double function")
  6         c = 16
  7         print("hello4 from double function")
  8         print("hello5 from double function")
  9         d = 20
 10         print("hello6 from double function")
 11         print("hello7 from double function")
 12         print("hello8 from double function")
 13         e = 25
 14         return x*2
 15     print(double(7))
(Pdb)
```

continue/ c: To start execution of the code, we can use this command. If the program executes successfully, you will be taken back to the (Pdb) prompt where you can restart the execution again.



```
C:\Users\Dell\pdb_check>python -m pdb code1.py
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb) c
hello1 from double function
hello2 from double function
hello3 from double function
hello4 from double function
hello5 from double function
hello6 from double function
hello7 from double function
hello8 from double function
14
The program finished and will be restarted
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb)
```



Command Prompt - python -m pdb code1.py

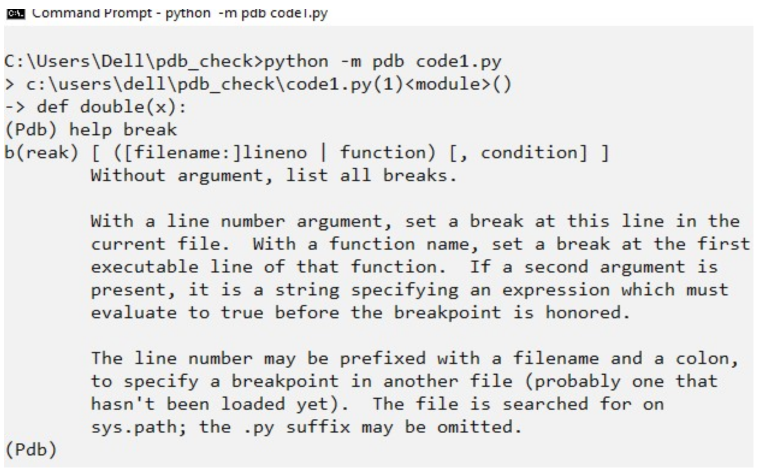
```
C:\Users\Dell\pdb_check>python -m pdb code1.py
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb) n
> c:\users\dell\pdb_check\code1.py(16)<module>()
-> print(double(7))
(Pdb) s
--Call--
> c:\users\dell\pdb_check\code1.py(1)double()
-> def double(x):
(Pdb) s
> c:\users\dell\pdb_check\code1.py(2)double()
-> a = 6
(Pdb) s
> c:\users\dell\pdb_check\code1.py(3)double()
-> print("hello1 from double function")
(Pdb) s
hello1 from double function
> c:\users\dell\pdb_check\code1.py(4)double()
-> print("hello2 from double function")
(Pdb) c
hello2 from double function
hello3 from double function
hello4 from double function
hello5 from double function
hello6 from double function
hello7 from double function
hello8 from double function
14
```

Think about these points!

- How do you switch between python interactive mode and pdb debugger mode?
  Hint: Use import code; code.interact()   OR interact<press enter>
- Is there any way to list last few lines of code using list command?
- Can you evaluate any expression using pdb like how python interpreter does it?
- If there is any Exception in the code raised by pdb while running the code line by line, will it affect the further lines? If yes, how to address this? Which function from pdb module helps us to do this?

Let us try understanding few terms related to breakpoints .

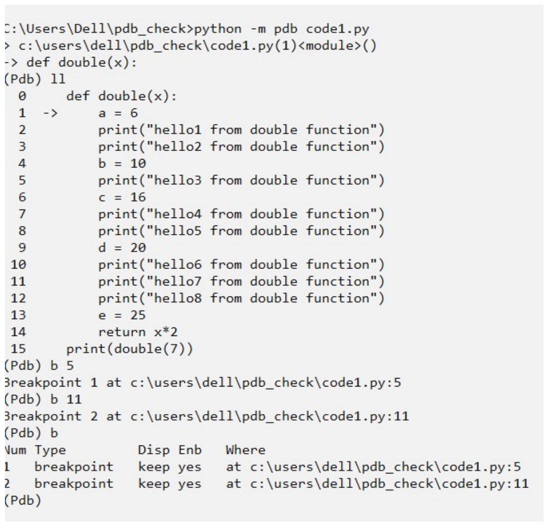
A breakpoint instructs the debugger to stop at a particular code location in the program, returning control of the debugger to them. The user may then inspect the application's state. Breakpoints are the most common type of debug event that developers use. There are many ways to implement breakpoints including inserting a trap or illegal instructions that cause the kernel to raise a signal which is then handled by the debugger. Let us try to understand how to use breakpoint effectively.



Command Prompt - python -m pdb code1.py

```
C:\Users\Dell\pdb_check>python -m pdb code1.py
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb) help break
b(reak) [ ([filename:]lineno | function) [, condition] ]
        Without argument, list all breaks.

        With a line number argument, set a break at this line in the
        current file.  With a function name, set a break at the first
        executable line of that function.  If a second argument is
        present, it is a string specifying an expression which must
        evaluate to true before the breakpoint is honored.

        The line number may be prefixed with a filename and a colon,
        to specify a breakpoint in another file (probably one that
        hasn't been loaded yet).  The file is searched for on
        sys.path; the .py suffix may be omitted.
(Pdb)
```

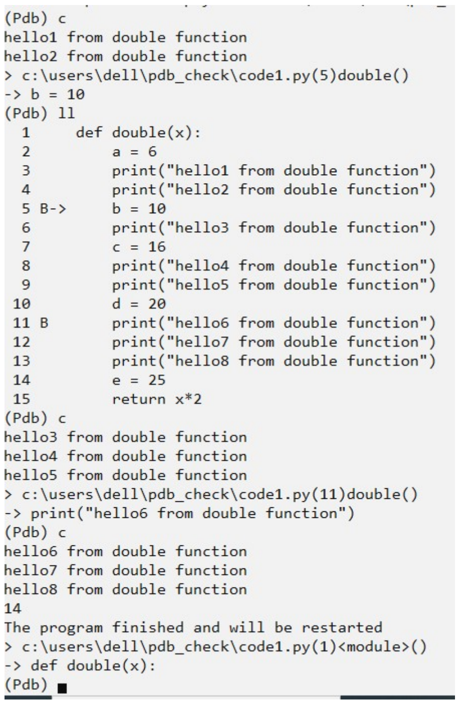
Consider the code_1.py and introduce breakpoints to this code at line #5 and #11. Note the number of the breakpoint set . 1 and 2 respectively.



```
C:\Users\Dell\pdb_check>python -m pdb code1.py
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb) ll
  0         def double(x):
  1  ->         a = 6
  2             print("hello1 from double function")
  3             print("hello2 from double function")
  4             b = 10
  5             print("hello3 from double function")
  6             c = 16
  7             print("hello4 from double function")
  8             print("hello5 from double function")
  9             d = 20
 10             print("hello6 from double function")
 11             print("hello7 from double function")
 12             print("hello8 from double function")
 13             e = 25
 14             return x*2
 15         print(double(7))
(Pdb) b 5
Breakpoint 1 at c:\users\dell\pdb_check\code1.py:5
(Pdb) b 11
Breakpoint 2 at c:\users\dell\pdb_check\code1.py:11
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at c:\users\dell\pdb_check\code1.py:5
2   breakpoint   keep yes   at c:\users\dell\pdb_check\code1.py:11
(Pdb)
```

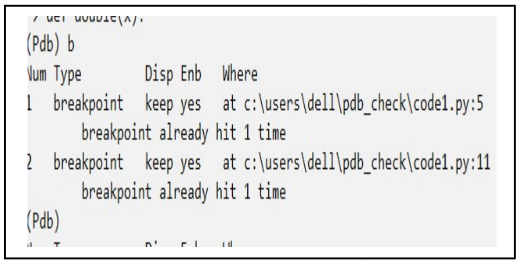
Note: To list all the breakpoints, just say b or break without any expression.

Running this using the debugger commands c/continue. Observe the output [This is neither s nor n]



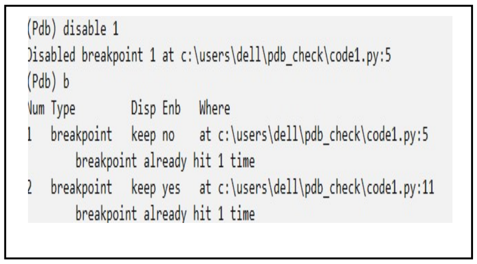
```
(Pdb) c
hello1 from double function
hello2 from double function
> c:\users\dell\pdb_check\code1.py(5)double()
-> b = 10
(Pdb) ll
  1     def double(x):
  2         a = 6
  3         print("hello1 from double function")
  4         print("hello2 from double function")
  5 B->     b = 10
  6         print("hello3 from double function")
  7         c = 16
  8         print("hello4 from double function")
  9         print("hello5 from double function")
 10         d = 20
 11 B       print("hello6 from double function")
 12         print("hello7 from double function")
 13         print("hello8 from double function")
 14         e = 25
 15         return x*2
(Pdb) c
hello3 from double function
hello4 from double function
hello5 from double function
> c:\users\dell\pdb_check\code1.py(11)double()
-> print("hello6 from double function")
(Pdb) c
hello6 from double function
hello7 from double function
hello8 from double function
14
The program finished and will be restarted
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb)
```

Can we disable these breakpoints? Use the disable command. Check the help on it.



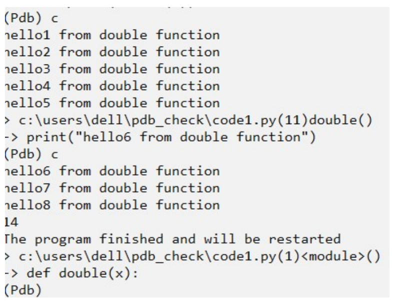
```
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at c:\users\dell\pdb_check\code1.py:5
        breakpoint already hit 1 time
2   breakpoint   keep yes   at c:\users\dell\pdb_check\code1.py:11
        breakpoint already hit 1 time
(Pdb)
```

Observe yes below Enb column



```
(Pdb) disable 1
Disabled breakpoint 1 at c:\users\dell\pdb_check\code1.py:5
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep no    at c:\users\dell\pdb_check\code1.py:5
        breakpoint already hit 1 time
2   breakpoint   keep yes   at c:\users\dell\pdb_check\code1.py:11
        breakpoint already hit 1 time
```

Observe no for breakpoint #1 below Enb column

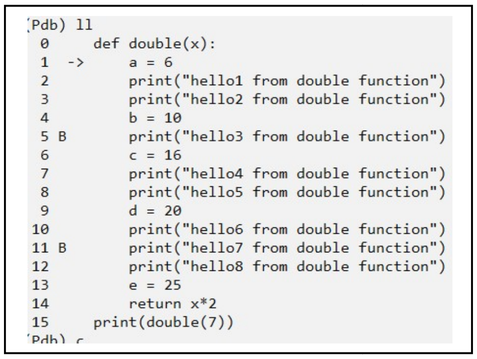


```
(Pdb) c
hello1 from double function
hello2 from double function
hello3 from double function
hello4 from double function
hello5 from double function
> c:\users\dell\pdb_check\code1.py(11)double()
-> print("hello6 from double function")
(Pdb) c
hello6 from double function
hello7 from double function
hello8 from double function
14
The program finished and will be restarted
> c:\users\dell\pdb_check\code1.py(1)<module>()
-> def double(x):
(Pdb)
```

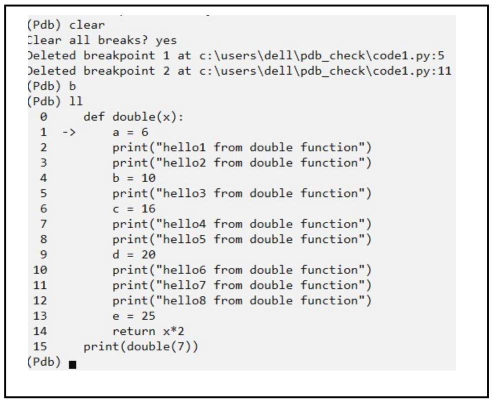
Now, apply continue command

Try it - > enable command!

Can we clear all the breakpoints? Use the **clear** or **cl** command.



```
(Pdb) ll
  0     def double(x):
  1  ->     a = 6
  2         print("hello1 from double function")
  3         print("hello2 from double function")
  4         b = 10
  5 B       print("hello3 from double function")
  6         c = 16
  7         print("hello4 from double function")
  8         print("hello5 from double function")
  9         d = 20
 10         print("hello6 from double function")
 11 B       print("hello7 from double function")
 12         print("hello8 from double function")
 13         e = 25
 14         return x*2
 15     print(double(7))
(Pdb) c
```



```
(Pdb) clear
Clear all breaks? yes
Deleted breakpoint 1 at c:\users\dell\pdb_check\code1.py:5
Deleted breakpoint 2 at c:\users\dell\pdb_check\code1.py:11
(Pdb) b
(Pdb) ll
  0     def double(x):
  1  ->     a = 6
  2         print("hello1 from double function")
  3         print("hello2 from double function")
  4         b = 10
  5         print("hello3 from double function")
  6         c = 16
  7         print("hello4 from double function")
  8         print("hello5 from double function")
  9         d = 20
 10         print("hello6 from double function")
 11         print("hello7 from double function")
 12         print("hello8 from double function")
 13         e = 25
 14         return x*2
 15     print(double(7))
(Pdb) ■
```

More on debugging!

Try these!

- **Debugging a failure after a program terminates** is called post-mortem debugging. **Pdb** supports post-mortem debugging through the **pm()**

- **Pysnooper** is a poor man's debugger and need **not to use print for debugging.**

Steps include

1. pip install PySnooper
2. import pysnooper
3. Add function @pysnooper.snoop() in your code.

If you don't want to trace an entire function, you can wrap the relevant part in with a block using with **pysnooper.snoop():**

-END–