

Dear Students,

Welcome to the Introduction to PDB and GDB debugger with sample example programs.

Regards,

Savitri S

Asst Professor, Dept of CSE

PES University

Mail-id: savitris@pes.edu

PDB Debugger:

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. It also supports post-mortem debugging and can be called under program control.

Let us consider the Python code sample.py as listed below in order to understand usage of pdb debugger to debug the code. Sampe.py has bug in the code, which will try to debug using pdb debugger:

sample_pgm.py

```
def add(x,y):
```

```
res=x+y
```

```
return res
```

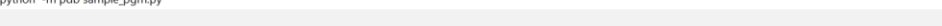
```
x=input("Enter value of x:")
```

```
y=input("Enter value of y:")
```

```
z=add(x,y)
```

```
print(z)
```

The pdb module has a very convenient command line interface. It is imported at the time of execution of Python script by using `-m` followed by name of the python file in the command prompt as shown below:



The screenshot shows a Windows Command Prompt window with the title bar "Command Prompt - python -m pdb sample_pgm.py". The command prompt displays the following sequence of commands and output:

```
C:\Users\Savitri\Desktop\debug>python -m pdb sample_pgm.py
> c:\users\savitri\desktop\debug\sample_pgm.py(1)<module>()
-> def add(x,y):
(Pdb) █
```

The cursor is positioned at the end of the line "(Pdb) █", indicating that the debugger is ready for the next command.

Debugging starts from the first line of the function and as shown in the above now we are into the pdb debugger environment with the cursor blinking ready to take command as input.

The commands recognized by the debugger are listed below. Most commands can be abbreviated to one or two letters as indicated; For example as h or help. Arguments 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.

```
(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)
```

Once we are into the pdb debugger environment, then we have to make use of commands .To see a list of all debugger commands type 'help' in front of the debugger prompt.

```
(Pdb) h
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) _
```

To know more about any command use 'help <command>' syntax. For example: help next or h n

```

(Pdb) help next
n(ext)
    Continue execution until the next line in the current function
    is reached or it returns.
(Pdb) help n
n(ext)
    Continue execution until the next line in the current function
    is reached or it returns.
(Pdb) help s
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 c
c(ontinue)
    Continue execution, only stop when a breakpoint is encountered.
(Pdb)

```

Print a stack trace, with the most recent frame at the bottom using where command as shown below:

```

(Pdb) where
c:\users\savitri\appdata\local\programs\python\python37-32\lib\bdb.py(585)
run()
-> exec(cmd, globals, locals)
    <string>(1)<module>()
> c:\users\savitri\desktop\debug\sample_pgm.py(1)<module>()
-> def add(x,y):
(Pdb) _

```

Pdb will restart the execution from the beginning.

```

-> def add(x,y):
(Pdb) n
> c:\users\savitri\desktop\debug\sample_pgm.py(5)<module>()
-> x=input("Enter value of x:")
(Pdb)
Enter value of x:11
> c:\users\savitri\desktop\debug\sample_pgm.py(6)<module>()
-> y=input("Enter value of y:")
(Pdb) c
Enter value of y:12
1112
The program finished and will be restarted
> c:\users\savitri\desktop\debug\sample_pgm.py(1)<module>()
-> def add(x,y):
(Pdb) _

```

```

The program finished and will be restarted
> c:\users\savitri\desktop\debug\sample_pgm.py(1)<module>()
-> def add(x,y):
(Pdb) q

C:\Users\Savitri\Desktop\debug>

```

Whenever we want to the value of the variable then use print or p as shown below:

```

(Pdb) print(x)
10
(Pdb) p y
'20'
(Pdb) _

```

To know the type of the value then use command shown below that is `whatis` followed by the variable name:

```
(Pdb) whatis x
<class 'str'>
(Pdb) whatis y
<class 'str'>
(Pdb)
```

In order to step into the function use `s` or `step` command and then use `next` to move to next line or `continue` till the end of the function:

```
(Pdb) step
--Call--
> c:\users\savitri\desktop\debug\sample_pgm.py(1)add()
-> def add(x,y):
(Pdb) n
> c:\users\savitri\desktop\debug\sample_pgm.py(2)add()
-> res=x+y
(Pdb) c
1020
The program finished and will be restarted
> c:\users\savitri\desktop\debug\sample_pgm.py(1)<module>()
-> def add(x,y):
(Pdb) _
```

To list the lines above and below the line that is executing using the command called `list` or `l`.

```
-> def add(x,y):
(Pdb) n
> c:\users\savitri\desktop\debug\sample_pgm.py(5)<module>()
-> x=input("Enter value of x:")
(Pdb) n
Enter value of x:10
> c:\users\savitri\desktop\debug\sample_pgm.py(6)<module>()
-> y=input("Enter value of y:")
(Pdb)
Enter value of y:20
> c:\users\savitri\desktop\debug\sample_pgm.py(7)<module>()
-> z=add(x,y)
(Pdb) l
  2             res=x+y
  3             return res
  4
  5     x=input("Enter value of x:")
  6     y=input("Enter value of y:")
  7 -> z=add(x,y)
  8     print(z)
[EOF]
(Pdb)
```

To list all the lines of the code using the command called `longlist` or `ll`.

```
(Pdb) ll
1      def add(x,y):
2          res=x+y
3          return res
4
5      x=input("Enter value of x:")
6      y=input("Enter value of y:")
7 -> z=add(x,y)
8      print(z)
(Pdb)
```

There may be necessary where we want to display only few lines or lines in between particular lines then we can use the list command along the lines numbers as shown below:

```
(Pdb) l 3,7
3           return res
4
5     x=input("Enter value of x:")
6     y=input("Enter value of y:")
7     z=add(x,y)
(Pdb)
```

```
(Pdb) help b
b(break) [( [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)
```

Will add breakpoint at line 7 as below and followed by continue in order to continue the execution and execution stops at the line 7 as breakpoint is set at line 7 as shown below:

```
(Pdb) b 7
Breakpoint 1 at c:\users\savitri\desktop\debug\sample_pgm.py:7
(Pdb) c
1020
The program finished and will be restarted
> c:\users\savitri\desktop\debug\sample_pgm.py(1)<module>()
-> def add(x,y):
(Pdb)
Enter value of x:10
Enter value of y:20
> c:\users\savitri\desktop\debug\sample_pgm.py(7)<module>()
-> z=add(x,y)
(Pdb) _
```

Another way of using pdb debugger is by importing the module as shown below code and set breakpoint using set_trace(). Although this will result the same but this is another way to introduce the debugger in python version 3.6 and below.

sample_pgm.py

```
import pdb

def add(x,y):

    res=x+y

    return res

x=input("Enter value of x:")
y=input("Enter value of y:")
pdb.set_trace()
```

```
z=add(x,y)
```

```
print(z)
```

While executing the code need not use any option as pdb is already imported in the file, hence execute the code like a regular python program and can then step through the code following this statement, and continue running without the debugger using the continue command. In order to run the debugger just type c and press enter and once all the lines are executed it exits out of the debugging environment.

```
C:\Users\Savitri\Desktop\debug>python sample_pgm.py
Enter value of x:10
Enter value of y:20
> c:\users\savitri\desktop\debug\sample_pgm.py(9)<module>()
-> z=add(x,y)
(Pdb)
(Pdb) c
1020
C:\Users\Savitri\Desktop\debug>_
```

Python version after 3.7 and above allows the built-in breakpoint(), when called with defaults, can be used instead of import pdb; pdb.set_trace()

```
C:\Users\Savitri\Desktop\debug>python sample_pgm.py
Enter value of x:12
Enter value of y:12
> c:\users\savitri\desktop\debug\sample_pgm.py(8)<module>()
-> z=add(x,y)
(Pdb) c
1212
C:\Users\Savitri\Desktop\debug>_
```

GDB Debugger:

GDB, the GNU Project debugger, allows us to see what is going on inside another program while it executes or what another program was doing at the moment it crashed.

GDB can run on most popular UNIX and Microsoft Windows variants, as well as on Mac OS X, it's available with gcc.

Let us consider a sample code to understand gdb debugger. Sample code is to calculate sum of all the factors of a given number and display the result as shown below:

sample_factors.c

```
#include<stdio.h>
```

```
int sum_factors(int n)
```

```
{
```

```
    int sum=0;
```

```
    for(int i=1;i<n;++i)
```

```
    {
```

```
        if(n%i==0)
```

```
        {
```

```
            sum+=i;
```

```
        }
```

```
    }
```

```
    return sum;
```

```
}
```

```
int main()
```

```
{
```

```
    int number;int sum;
```

```
    scanf("%d",&number);
```

```
    sum=sum_factors(number);
```

```
printf("Sum of factors of a number %d is %d\n",number,sum);

return 0;

}
```

Let us execute the code to check the expected result:

```
C:\Users\Savitri\Desktop\debug>gcc sample_factors.c -o factors_op
C:\Users\Savitri\Desktop\debug>factors_op.exe
12
Sum of factors of a number 12 is 16
C:\Users\Savitri\Desktop\debug>factors_op.exe
24
Sum of factors of a number 24 is 36
C:\Users\Savitri\Desktop\debug>_
```

Expected result of the above code is not the same as the actual result or output. There is a bug in the above code, so to find it will execute each statement with debugger as shown below, firstly execute the code to enable debugging with `-g` option, then followed by name of the file in the command line. So now we have an executable file (in this `factors_op.exe`) and we want to debug it. First you must launch the debugger. The debugger is called `gdb` and you can tell it which file to debug at the shell prompt as shown below:

```
C:\Users\Savitri\Desktop\debug>gcc -g sample_factors.c -o factors_op
C:\Users\Savitri\Desktop\debug>gdb factors_op.exe
GNU gdb (GDB) 7.6.1
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.htm
l>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "mingw32".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from C:\Users\Savitri\Desktop\debug\factors_op.exe...done.
(gdb)
```

Now we are into `gdb` environment, where we are supposed to use `gdb` commands in order to use `gdb` to debug, so first command that will be learning is `help` command, where it lists different commands and usage of them along with the options:


```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in
that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb)
```

In order to start the debugging we are supposed to execute the code using run command as shown:

```
(gdb) run
Starting program: C:\Users\Savitri\Desktop\debug\debug\factor_op.exe
[New Thread 3964.0x63c]
[New Thread 3964.0x4678]
12
Sum of factors of a number 12 is 16
[Inferior 1 (process 3964) exited normally]
(gdb) _
```

user-defined -- User-defined commands

```
Type "help" followed by a class name for a list of commands in
that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb) run
Starting program: C:\Users\Savitri\Desktop\debug\debug\factor_op.exe
[New Thread 3964.0x63c]
[New Thread 3964.0x4678]
12
Sum of factors of a number 12 is 16
[Inferior 1 (process 3964) exited normally]
(gdb) run
Starting program: C:\Users\Savitri\Desktop\debug\debug\factor_op.exe
[New Thread 18076.0xf48]
[New Thread 18076.0x368]
24
Sum of factors of a number 24 is 36
[Inferior 1 (process 18076) exited normally]
(gdb) _
```

By looking into the output we can see that the expected result is wrong, So we have to debug this code by using different commands available. Firstly there are a lot of commands to know the list of the commands type: help all. And to know any command details then have to use as below i.e, help name of the command, name of the command can be full name or short name:

```
(gdb) help l
List specified function or line.
With no argument, lists ten more lines after or around previous listing.
"list -" lists the ten lines before a previous ten-line listing.
One argument specifies a line, and ten lines are listed around that line.
Two arguments with comma between specify starting and ending lines to list.

Lines can be specified in these ways:
  LINENUM, to list around that line in current file,
  FILE:LINENUM, to list around that line in that file,
  FUNCTION, to list around beginning of that function,
  FILE:FUNCTION, to distinguish among like-named static functions.
  *ADDRESS, to list around the line containing that address.
With two args if one is empty it stands for ten lines away from the other a
rg.
(gdb) help next
Step program, proceeding through subroutine calls.
Usage: next [N]
Unlike "step", if the current source line calls a subroutine,
this command does not enter the subroutine, but instead steps over
the call, in effect treating it as a single source line.
(gdb) _
```

Let us look into different commands like list, next, breakpoint, step etc. Below is the command to list the lines of the code that we require.

```
(gdb) l 1,20
1      #include<stdio.h>
2      int sum_factors(int n)
3      {
4          int sum=0;
5          for(int i=1;i<n;++i)
6          {
7              if(n%i==0)
8              {
9                  sum+=i;
10             }
11         }
12         return sum;
13     }
14     int main()
15     {
16         int number;int sum;
17         scanf("%d",&number);
18         sum=sum_factors(number);
19         printf("Sum of factors of a number %d is %d\n",number,sum);
20         return 0;
(gdb) _
```

We can set the breakpoint where we want to stop the execution by using command as break or b followed by line number, so here we are setting breakpoint at 17th line.

```
(gdb) b 17
Breakpoint 1 at 0x401458: file sample_factors.c, line 17.
(gdb) run
Starting program: C:\Users\Savitri\Desktop\debug\factors_op.exe
[New Thread 8460.0x2f30]
[New Thread 8460.0x1a3c]

Breakpoint 1, main () at sample_factors.c:17
17         scanf("%d",&number);
(gdb) next
12
18         sum=sum_factors(number);
(gdb)
```

And then followed by next or n command in order to go to the next line and followed by continue in order to continue the execution by using command continue or c as shown below:

```
(gdb) n
19          printf("Sum of factors of a number %d is %d\n",number,sum);

(gdb) continue
Continuing.
Sum of factors of a number 12 is 16
[Inferior 1 (process 8460) exited normally]
(gdb) _
```

We can also display the value of the variable or the expression by using the command print or p followed by name or the variable. In the below, as can look into the value of the variable number initially the variable number holds the un-initialized or junk value and after the next statement gets executed and the function takes input and hence has the updated value of number. Similarly, to print the value of the sum variable as p sum, displays some number which is not the proper value of sum, even though the function has returned the result which is stored in variable sum, reason is because in order to get the variable value of the function then have to go to that function, which is performed by using step command which shown in further example.

```
Starting program: C:\Users\Savitri\Desktop\debug/factors_op.exe
[New Thread 6036.0x17fc]
[New Thread 6036.0x35ac]

Breakpoint 1, main () at sample_factors.c:17
17          scanf("%d",&number);
(gdb) p number
$1 = 2404352
(gdb) n
12
18          sum=sum_factors(number);
(gdb) p number
$2 = 12
(gdb) p sum
$3 = 4194432
(gdb) c
Continuing.
Sum of factors of a number 12 is 16
[Inferior 1 (process 6036) exited normally]
(gdb)
```

Now here added breakpoint at line 16, then to print number, we can see the value getting changed similarly instead of printing every time we can use watch in order to keep track of value every and display its value each time. In case if we want to execute it from beginning then use command run again and say 'y' to start from the beginning.


```
12
18          sum=sum_factors(number);
(gdb) step
sum_factors (n=12) at sample_factors.c:4
4          int sum=0;
(gdb) n
5          for(int i=1;i<n;++i)
(gdb) p sum_
$3 = 0
(gdb) watch i_
Hardware watchpoint 5: i
(gdb) n
Hardware watchpoint 5: i

Old value = 0
New value = 1
sum_factors (n=12) at sample_factors.c:5
5          for(int i=1;i<n;++i)
(gdb) n_
7          if(n%i==0)
(gdb) p n
$4 = 12
(gdb)
```

```
(gdb) n
9          sum+=i;
(gdb) n
5          for(int i=1;i<n;++i)
(gdb) n_
Hardware watchpoint 5: i

Old value = 1
New value = 2
sum_factors (n=12) at sample_factors.c:5
5          for(int i=1;i<n;++i)
(gdb) n
7          if(n%i==0)
(gdb) n
9          sum+=i;
(gdb) n_
5          for(int i=1;i<n;++i)
(gdb) n
Hardware watchpoint 5: i

Old value = 2
New value = 3
sum_factors (n=12) at sample_factors.c:5
```

```

5           for(int i=1;i<n;++i)
(gdb) n
7           if(n%i==0)
(gdb) n
9           sum+=i;
(gdb) n
5           for(int i=1;i<n;++i)
(gdb) n
Hardware watchpoint 5: i

Old value = 3
New value = 4
sum_factors (n=12) at sample_factors.c:5
5           for(int i=1;i<n;++i)
(gdb) n
7           if(n%i==0)
(gdb) n
9           sum+=i;
(gdb) n
5           for(int i=1;i<n;++i)
(gdb) n
Hardware watchpoint 5: i

```

```

Hardware watchpoint 5: i

Old value = 4
New value = 5
sum_factors (n=12) at sample_factors.c:5
5           for(int i=1;i<n;++i)
(gdb) n
7           if(n%i==0)
(gdb) n
5           for(int i=1;i<n;++i)
(gdb) n
Hardware watchpoint 5: i

Old value = 5
New value = 6
sum_factors (n=12) at sample_factors.c:5
5           for(int i=1;i<n;++i)
(gdb) n
7           if(n%i==0)
(gdb) n
9           sum+=i;
(gdb)

```

And keep doing till the last step, now we can observe as below it didn't enter the loop once it's equal to n value, as reason it was not calculating the sum for last value when it became equal to n value as shown below and hence end up getting wrong output.

```

(gdb) n
5           for(int i=1;i<n;++i)
(gdb) n
Hardware watchpoint 5: i

Old value = 10
New value = 11
sum_factors (n=12) at sample_factors.c:5
5           for(int i=1;i<n;++i)
(gdb) n
7           if(n%i==0)
(gdb) n
5           for(int i=1;i<n;++i)
(gdb) n
Hardware watchpoint 5: i

Old value = 11
New value = 12
sum_factors (n=12) at sample_factors.c:5
5           for(int i=1;i<n;++i)
(gdb) n
12          return sum;
(gdb) _

```

This is how we could be able to trace the code using debugger and could be able to find the bug.

And now will quit the debugger as shown below and correct the code to get the required output:

```

12          return sum;
(gdb) q
A debugging session is active.

    Inferior 1 [process 17168] will be killed.

Quit anyway? (y or n) y
error return ../../gdb-7.6.1/gdb/windows-nat.c:1275 was 5
c:\Users\Savitri\Desktop\debug>_

```

There are a lot of other commands in gdb which are required to debug the code like setting the variable, back tracing when you have multiple function calls, debugging the code which results in core dump etc., which can be referred from the gnu official site link: <https://www.gnu.org/software/gdb/>

END