08 – Encontrando errores en un código de Python 3

Diego Andrés Alvarez Marín
Profesor Asociado
Universidad Nacional de Colombia
Sede Manizales

pdb

- Existen dos formas de invocar este depurador. Simplemente escriba esta línea en el lugar donde quiere poner un breakpoint:
  - `import pdb; pdb.set_trace()`
  - `breakpoint()` → a partir de Python 3.7
- El uso de `breakpoint()` es preferible, ya que nos da acceso a la variable de entorno `PYTHONBREAKPOINT`. Si dicha variable tiene el valor de `0`, la depuración no se llevará a cabo.
- Desde la consola, llame a su programa con:
  - `python -m pdb example1.py`

Ejemplo de sesión

Al haber ejecutado un `python -m pdb example1.py`, el depurador comienza su acción en la primera línea de código, que en este caso es la 3. Si el programa se hubiera ejecutado con `python example1.py`, la depuración inmediatamente después del `breakpoint()` (línea 5).

```
daalvarez@eredron ~ $ cat example1.py
#!/usr/bin/env python3

filename = __file__
breakpoint()
print(f'path = {filename}')
daalvarez@eredron ~ $
daalvarez@eredron ~ $ python -m pdb example1.py
> /home/daalvarez/example1.py(3)<module>()
-> filename = __file__
(Pdb) p filename
*** NameError: name 'filename' is not defined
(Pdb) n
> /home/daalvarez/example1.py(4)<module>()
-> breakpoint()
(Pdb) p filename
'example1.py'
(Pdb) filename
'example1.py'
(Pdb) q
daalvarez@eredron ~ $
```

Nombre archivo (línea actual) <ámbito actual>
→ línea actual (no ejecutada aun)

Imprima variable `filename`

Ejecute línea actual

Imprima variable `filename`

Salga del depurador

Comandos

```
(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) h h
h(elp)
        Without argument, print the list of available commands.
        With a command name as argument, print help about that command.
        "help pdb" shows the full pdb documentation.
        "help exec" gives help on the ! command.
      _
```

Invocando la ayuda

```
(Pdb) help list
l(ist) [first [,last] | .]

        List source code for the current file.  Without arguments,
        list 11 lines around the current line or continue the previous
        listing.  With . as argument, list 11 lines around the current
        line.  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.

        The current line in the current frame is indicated by "->".
        If an exception is being debugged, the line where the
        exception was originally raised or propagated is indicated by
        ">>", if it differs from the current line.
(Pdb) help ll
longlist | ll
        List the whole source code for the current function or frame.
(Pdb) help n
n(ext)
        Continue execution until the next line in the current function
        is reached or it returns.
(Pdb) help p
p expression
        Print the value of the expression.
(Pdb)
```

`p`	Print the value of an expression.
`pp`	Pretty-print the value of an expression.
`n`	Continue execution until the next line in the current function is reached or it returns.
`s`	Execute the current line and stop at the first possible occasion (either in a function that is called or in the current function).
`c`	Continue execution and only stop when a breakpoint is encountered.
`unt`	Continue execution until the line with a number greater than the current one is reached. With a line number argument, continue execution until a line with a number greater or equal to that is reached.
`l`	List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing.
`ll`	List the whole source code for the current function or frame.

`b`	With no arguments, list all breaks. With a line number argument, set a breakpoint at this line in the current file.
`w`	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.
`u`	Move the current frame count (default one) levels up in the stack trace (to an older frame).
`d`	Move the current frame count (default one) levels down in the stack trace (to a newer frame).
`h`	See a list of available commands.
`h <topic>`	Show help for a command or topic.
`h pdb`	Show the full pdb documentation.
`q`	Quit the debugger and exit.

Tips

- Si se presiona ENTER, se ejecuta el último comando entrado.

Imprimir expresiones (p)

- Use p (print) o pp (pretty print). Se puede imprimir cualquier expresión válida de python. Ejemplos:

- Imprima la variable mivar:
  - `p mivariable`
- Imprima las variables mivar1 y mivar2
  - `p mivar1, mivar2`
- Imprima la concatenación de las cadenas cad1 y cad2
  - `p cad1+cad2`

Navegando por el código

- n (next)→ es como un step over. No entra a la función.
- s (step) → es como un step into. Entra a la función
- Ambos comandos paran la ejecución cuando se alcanza el final de la función.

```
(Pdb) h r
r(eturn)
        Continue execution until the current function returns.
```

Listando el código

- `ll (longlist)` → Muestra el código fuente de la función actual
- `list, l` → Muestra 11 líneas de código y si se presiona ENTER continua con el listado anterior.

Los breakpoints

```
(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.
```

Ejemplos:

Tenemos previamente un archivo que llamamos con import util

b util:5 → pare en la línea 5 del archivo util.py

b util.get_path → pare cuando se invoque get_path() del archivo util.py

Enter `b` with no arguments to see a list of all breakpoints:

Shell

```
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /code/util.py:1
(Pdb)
```

You can disable and re-enable breakpoints using the command `disable bpnumber` and `enable bpnumber`. `bpnumber` is the breakpoint number from the breakpoints list's 1st column `Num`. Notice the `Enb` column's value change:

Shell

```
(Pdb) disable 1
Disabled breakpoint 1 at /code/util.py:1
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep no    at /code/util.py:1
(Pdb) enable 1
Enabled breakpoint 1 at /code/util.py:1
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /code/util.py:1
(Pdb)
```

To delete a breakpoint, use the command `cl` (clear):

Shell

```
cl(ear) filename:lineno
cl(ear) [bpnumber [bpnumber...]]
```

Now let's use a Python expression to set a breakpoint. Imagine a situation where you wanted to break only if your troubled function received a certain input.

In this example scenario, the `get_path()` function is failing when it receives a relative path, i.e. the file's path doesn't start with `/`. I'll create an expression that evaluates to true in this case and pass it to `b` as the 2nd argument:

Shell

```
$ ./example4.py
> /code/example4.py(7)<module>()
-> filename_path = util.get_path(filename)
(Pdb) b util.get_path, not filename.startswith('/')
Breakpoint 1 at /code/util.py:1
(Pdb) c
> /code/util.py(3)get_path()
-> import os
(Pdb) a
filename = './example4.py'
(Pdb)
```

```
(Pdb) h a
a(rgs)
        Print the argument list of the current function.
(Pdb) h c
c(ont(inue))
        Continue execution, only stop when a breakpoint is encountered.
(Pdb) h cl
cl(ear) filename:lineno
cl(ear) [bpnumber [bpnumber...]]
        With a space separated list of breakpoint numbers, clear
        those breakpoints.  Without argument, clear all breaks (but
        first ask confirmation).  With a filename:lineno argument,
        clear all breaks at that line in that file.
(Pdb) h tbreak
tbreak [ ([filename:]lineno | function) [, condition] ]
        Same arguments as break, but sets a temporary breakpoint: it
        is automatically deleted when first hit.
```

Continuando la ejecución

```
(Pdb) h c
c(ont(inue))
        Continue execution, only stop when a breakpoint is encountered.
(Pdb) h unt
unt(il) [lineno]
        Without argument, continue execution until the line with a
        number greater than the current one is reached.  With a line
        number, continue execution until a line with a number greater
        or equal to that is reached.  In both cases, also stop when
        the current frame returns.
(Pdb)
```

Use unt when you want to continue execution and stop farther down in the current source file. You can treat it like a hybrid of n (next) and b (break), depending on whether you pass a line number argument or not. unt es muy útil haciendo loops.

Mostrando expresiones

```
(Pdb) h display
display [expression]

        Display the value of the expression if it changed, each time execution
        stops in the current frame.

        Without expression, list all display expressions for the current frame.
(Pdb) h undisplay
undisplay [expression]

        Do not display the expression any more in the current frame.

        Without expression, clear all display expressions for the current frame.
```

```
(Pdb) display char
display char: 'e'
(Pdb) c
> /code/example4display.py(11)get_path()
-> pass  # Check filename char
display char: 'x'  [old: 'e']
(Pdb)
> /code/example4display.py(11)get_path()
-> pass  # Check filename char
display char: 'a'  [old: 'x']
(Pdb)
> /code/example4display.py(11)get_path()
-> pass  # Check filename char
display char: 'm'  [old: 'a']
```

You can enter display multiple times to build a watch list of expressions. This can be easier to use than p. After adding all of the expressions you're interested in, simply enter displayto see the current values:

```
(Pdb) display char
display char: 'e'
(Pdb) display fname
display fname: './example4display.py'
(Pdb) display head
display head: '.'
(Pdb) display tail
display tail: 'example4display.py'
(Pdb) c
> /code/example4display.py(11)get_path()
-> pass  # Check filename char
display char: 'x'  [old: 'e']
(Pdb) display
Currently displaying:
char: 'x'
fname: './example4display.py'
head: '.'
tail: 'example4display.py'
```

jump

```
(Pdb) h jump
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.

        It should be noted that not all jumps are allowed -- for
        instance it is not possible to jump into the middle of a
        for loop or out of a finally clause.
```

Por ejemplo `j 12` salta la ejecución a la línea 12. Incluso podemos volver al inicio de la función o del programa de este modo.

Navegando en la pila de llamadas

```
(Pdb) h w    w, where o bt
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.  'bt' is an alias for this command.
(Pdb) h u
u(p) [count]
        Move the current frame count (default one) levels up in the
        stack trace (to an older frame).
(Pdb) h d
d(own) [count]
        Move the current frame count (default one) levels down in the
        stack trace (to a newer frame).
```

Setting the values of variables

```
(Pdb) h exec
(!) statement
        Execute the (one-line) statement in the context of the current
        stack frame.  The exclamation point can be omitted unless the
        first word of the statement resembles a debugger command.  To
        assign to a global variable you must always prefix the command
        with a 'global' command, e.g.:
        (Pdb) global list_options; list_options = ['-l']
        (Pdb)
```

So how can we assign a new value to **b**? The trick is to start the command with an exclamation point (!).

```
(Pdb)!b = "BBB"
```

An exclamation point tells pdb that what follows is a Python statement, not a pdb command.

Y ese cambio es válido solo para el ámbito actual.

Use w, u y d si quiere cambiar variables en otro ámbitos

Modo interactivo (interact)

```
(Pdb) h interact
interact

        Start an interactive interpreter whose global namespace
        contains all the (global and local) names found in the current scope.
(Pdb) !a = 100
(Pdb) !b = 200
(Pdb) interact
*interactive*
>>> a
100
>>> b
200
>>> print(f"a = {a} y b = {b}")
a = 100 y b = 200
>>> # Me salgo con Ctrl+D en Linux o Ctrl+C en Windows
>>> 
```

Saliendose del debugger

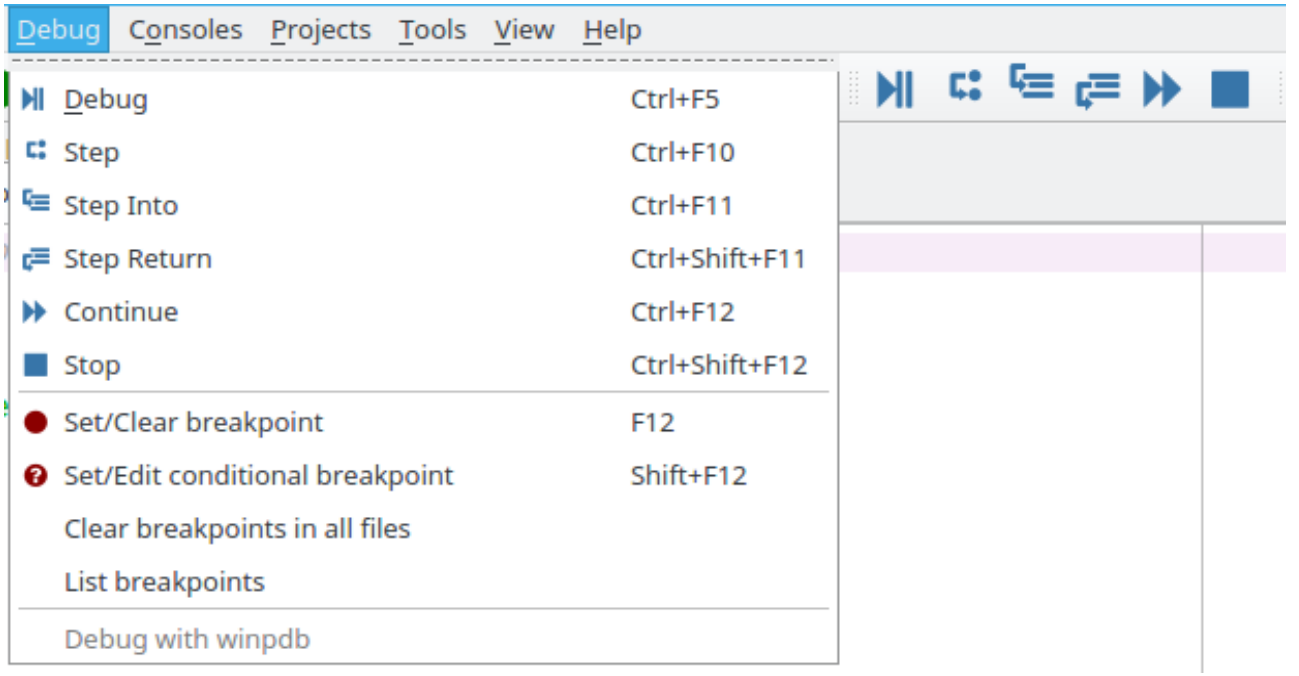
- Se sale con `exit`, `quit` o `q`

```
(Pdb) h exit
q(uit)
exit
        _ Quit from the debugger. The program being executed is aborted.
```

- `restart` o `run` reinicia el programa manteniendo los puntos de interrupción

```
(Pdb) h restart
run [args...]
        Restart the debugged python program. If a string is supplied
        it is split with "shlex", and the result is used as the new
        sys.argv.  History, breakpoints, actions and debugger options
        are preserved.  "restart" is an alias for "run".
        _
```

Y en spyder ...



Variable explorer

Name	Type	Size	Value
C	int	1	1
F	int	1	0
capsula_alargamiento	dict	2	{'comida':False, 'pos':(3, 50)}
copia_mapa	str32	(42, 66)	ndarray object of numpy module
dir_mov0	int	1	2
dir_mov1	int	1	1
i	int	1	1
mapa	str32	(42, 66)	ndarray object of numpy module
proxima_dir	list	2	[2, 1]

You left out one of the best ad-hock troubleshooting ways to figure out what caused an exception post mortem.

Run:

```
python -i your script.py
```

After the exception you will be left at the python interpreter prompt staring at your exception.

Type:

```
import pdb
pdb.pm
```

And you will be put into the context of the stack of the last exception. This means you can print/examine the local variables at the point of failure after the failure has occurred without having to change a line of code or import pdb in advance.

Another method that requires a little bit of preparation is to put the import pdb and pdb.set_trace() in a signal handler trap. That way you can do a kill -SIGNAL PID or if the signal you trap is INT you can just Ctrl-C to get dropped into the debugger at any point. The signal handler technique is good for testing release candidates and released code because the signal handler doesn't create any runtime overhead.

Signal handler example. Debugger starts with Ctrl-C:

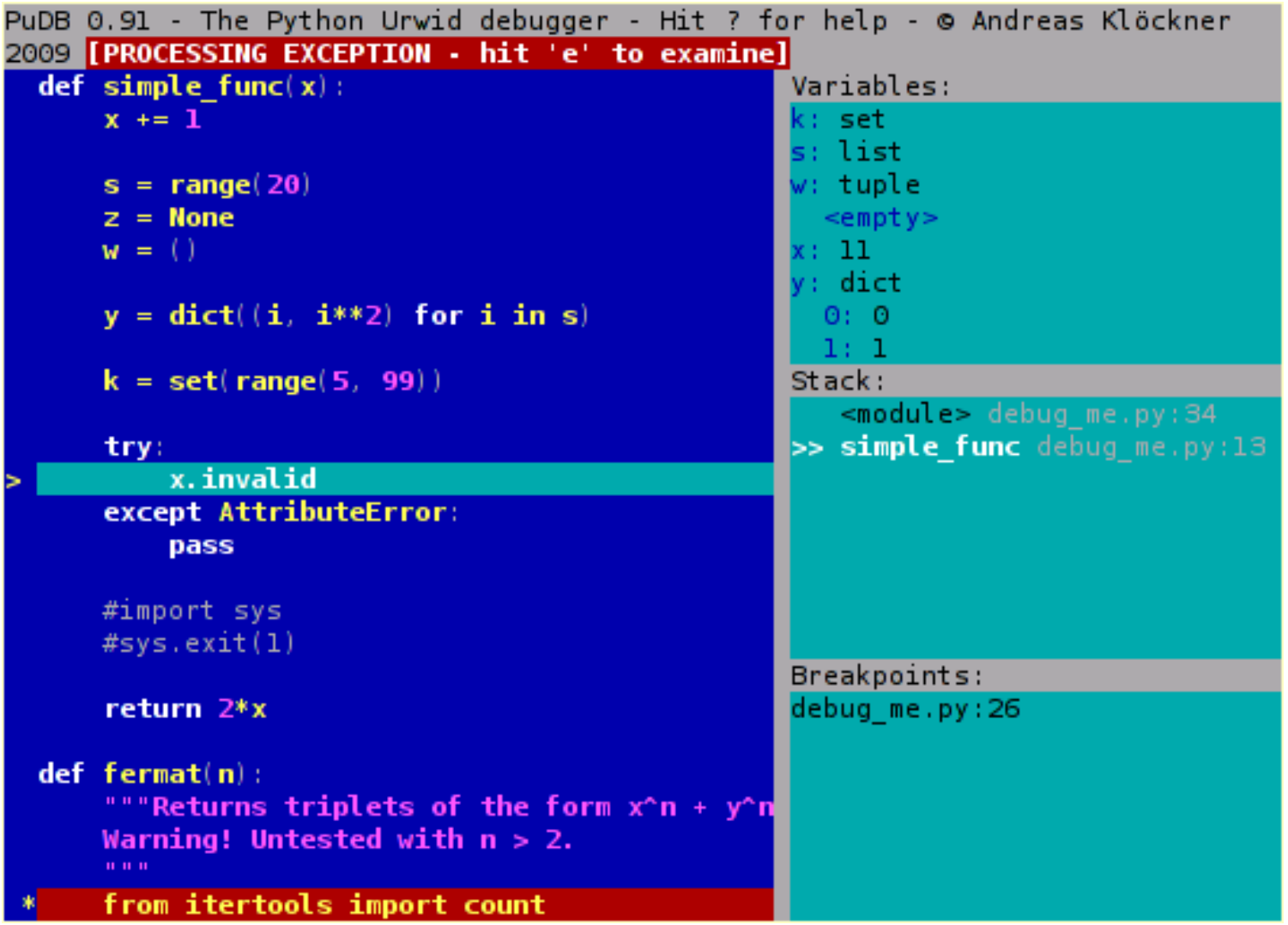
```
1  import signal
2  def  int_handler(signal, frame):
3          import pdb
4          pdb.set_trace(frame)
5  signal.signal(signal.SIGINT, int_handler)
```

Put that at the top of your script and you can start debugging your script at any point by type Ctrl-C. Resume programe execution by typing exit at the Pdb prompt.

ipdb

- Adicionalmente tiene syntax highlighting
- Completion

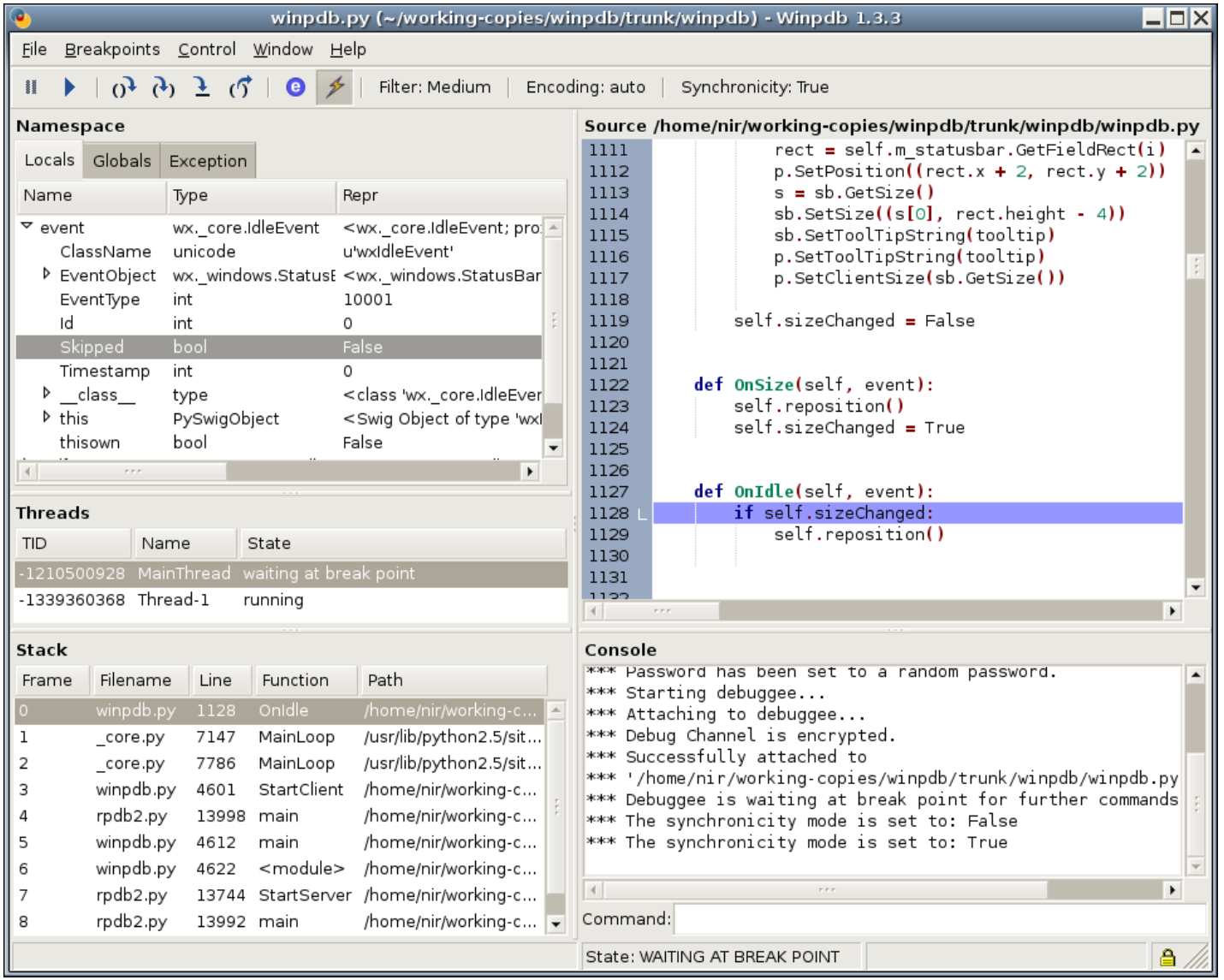
pudb



Se parece mucho al depurador del Turbo Pascal

winpdb

(a pesar de su nombre, corre también en Linux)



%debug in jupyter

Referencias

- https://realpython.com/python-debugging-pdb/
- https://docs.python.org/3/library/pdb.html