

REVERSING WITH IDA PRO FROM SCRATCH

PART 13

Before continuing with IDA exercises, we will see this part 13 in a relaxed way. I will talk about the use of a very comfortable plugin to handle Python better. Shaddy suggested to use the included interesting little bar and I thank him a lot.

The plugin is **IpyIDA** and you can install it just copying and pasting this line in the Python bar.

```
import urllib2; exec urllib2.urlopen('https://github.com/eset/ipyida/raw/stable/install_from_ida.py').read()
```

That is a one-line command that can be copied and pasted from here. If it fails or something, the link is here.

<https://github.com/eset/ipyida>

Install

IPyIDA has been tested with IDA 6.6 and up on Windows, OS X and Linux.

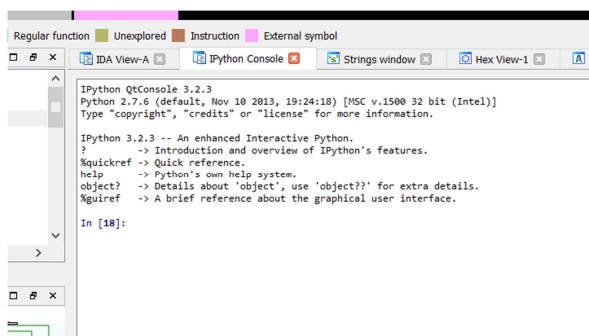
Fast and easy install

A script is provided to install IPyIDA and its dependencies automatically from the IDA console. Simply copy the following line to the IDA console.

```
import urllib2; exec urllib2.urlopen('https://github.com/eset/ipyida/raw/stable/install_from_ida.py').read()
```

It will be installed automatically in a pair of minutes. Don't pay attention to warnings about Python version. It will work anyways. At the end of this tutorial, I included a list of errors and solutions when installing this.

After installing and running it from **EDIT-PLUGINS-IpyIDA**, a little window appeared at the bottom without enough space to write, but dragging it, I could set it as a tab to work comfortably.



It is obviously more powerful than the little Python bar. If we press `?` to see the quick reference.

IPython -- An enhanced Interactive Python

IPython offers a combination of convenient shell features, special commands and a history mechanism for both input (command history) and output (results caching, similar to Mathematica). It is intended to be a fully compatible replacement for the standard Python interpreter, while offering vastly improved functionality and flexibility.

At your system command line, type `'ipython -h'` to see the command line options available. This document only describes interactive features.

MAIN FEATURES

- * Access to the standard Python help. As of Python 2.1, a help system is available with access to object docstrings and the Python manuals. Simply type `'help'` (no quotes) to access it.
- * Magic commands: type `%magic` for information on the magic subsystem.
- * System command aliases, via the `%alias` command or the configuration file(s).
- * Dynamic object information:

Typing `?word` or `word?` prints detailed information about an object. If certain strings in the object are too long (docstrings, code, etc.) they get snipped in the center for brevity.

Typing `??word` or `word??` gives access to the full information without

snipping long strings. Long strings are sent to the screen through the less pager if longer than the screen, printed otherwise.

The `???` system gives access to the full source code for any object (if available), shows function prototypes and other useful information.

If you just want to see an object's docstring, type `'%pdoc object'` (without quotes, and without `%` if you have automagic on).

- * Completion in the local namespace, by typing TAB at the prompt.

At any time, hitting tab will complete any available python commands or variable names, and show you a list of the possible completions if there's no unambiguous one. It will also complete filenames in the current directory.

This feature requires the readline and rlcomplete modules, so it won't work if your Python lacks readline support (such as under Windows).

- * Search previous command history in two ways (also requires readline):

- Start typing, and then use Ctrl-p (previous,up) and Ctrl-n (next,down) to search through only the history items that match what you've typed so far. If you use Ctrl-p/Ctrl-n at a blank prompt, they just behave like normal arrow keys.

- Hit Ctrl-r: opens a search prompt. Begin typing and the system searches your history for lines that match what you've typed so far, completing as much as it can.

- `%hist`: search history by index (this does **not** require readline).

- * Persistent command history across sessions.

- * Logging of input with the ability to save and restore a working session.

- * System escape with `!`. Typing `!ls` will run `'ls'` in the current directory.

- * The reload command does a 'deep' reload of a module: changes made to the module since you imported will actually be available without having to exit.

- * Verbose and colored exception traceback printouts. See the magic `xmode` and `xcolor` functions for details (just type `%magic`).

* Input caching system:

IPython offers numbered prompts (In/Out) with input and output caching. All input is saved and can be retrieved as variables (besides the usual arrow key recall).

The following GLOBAL variables always exist (so don't overwrite them!):

`_i`: stores previous input.

`_ii`: next previous.

`_iii`: next-next previous.

`_ih` : a list of all input `_ih[n]` is the input from line `n`.

Additionally, global variables named `_i<n>` are dynamically created (`<n>` being the prompt counter), such that `_i<n> == _ih[<n>]`

For example, what you typed at prompt 14 is available as `_i14` and `_ih[14]`.

You can create macros which contain multiple input lines from this history, for later re-execution, with the `%macro` function.

The history function `%hist` allows you to see any part of your input history by printing a range of the `_i` variables. Note that inputs which contain magic functions (%) appear in the history with a prepended comment. This is because they aren't really valid Python code, so you can't exec them.

* Output caching system:

For output that is returned from actions, a system similar to the input cache exists but using `_` instead of `_i`. Only actions that produce a result (NOT assignments, for example) are cached. If you are familiar with Mathematica, IPython's `_` variables behave exactly like Mathematica's `%` variables.

The following GLOBAL variables always exist (so don't overwrite them!):

`_` (one underscore): previous output.

`__` (two underscores): next previous.

`___` (three underscores): next-next previous.

Global variables named `_<n>` are dynamically created (`<n>` being the prompt counter), such that the result of output `<n>` is always available as `_<n>`.

Finally, a global dictionary named `_oh` exists with entries for all lines

which generated output.

* Directory history:

Your history of visited directories is kept in the global list `_dh`, and the magic `%cd` command can be used to go to any entry in that list.

* Auto-parentheses and auto-quotes (adapted from Nathan Gray's LazyPython)

1. Auto-parentheses

Callable objects (i.e. functions, methods, etc) can be invoked like this (notice the commas between the arguments)::

```
In [1]: callable_ob arg1, arg2, arg3
```

and the input will be translated to this::

```
callable_ob(arg1, arg2, arg3)
```

This feature is off by default (in rare cases it can produce undesirable side-effects), but you can activate it at the command-line by starting IPython with `--autocall 1`, set it permanently in your configuration file, or turn on at runtime with `%autocall 1`.

You can force auto-parentheses by using `'/'` as the first character of a line. For example::

```
In [1]: /globals # becomes 'globals()'
```

Note that the `'/'` MUST be the first character on the line! This won't work::

```
In [2]: print /globals # syntax error
```

In most cases the automatic algorithm should work, so you should rarely need to explicitly invoke `/`. One notable exception is if you are trying to call a function with a list of tuples as arguments (the parenthesis will confuse IPython)::

```
In [1]: zip (1,2,3),(4,5,6) # won't work
```

but this will work::

```
In [2]: /zip (1,2,3),(4,5,6)
```

```
-----> zip ((1,2,3),(4,5,6))
```

```
Out[2]= [(1, 4), (2, 5), (3, 6)]
```

IPython tells you that it has altered your command line by displaying the new command line preceded by `-->`. e.g.::

```
In [18]: callable list
```

```
-----> callable (list)
```

2. Auto-Quoting

You can force auto-quoting of a function's arguments by using `'` as the first character of a line. For example::

In [1]: `,my_function /home/me` # becomes `my_function("/home/me")`

If you use `'` instead, the whole argument is quoted as a single string (while `,` splits on whitespace)::

In [2]: `,my_function a b c` # becomes `my_function("a","b","c")`

In [3]: `;my_function a b c` # becomes `my_function("a b c")`

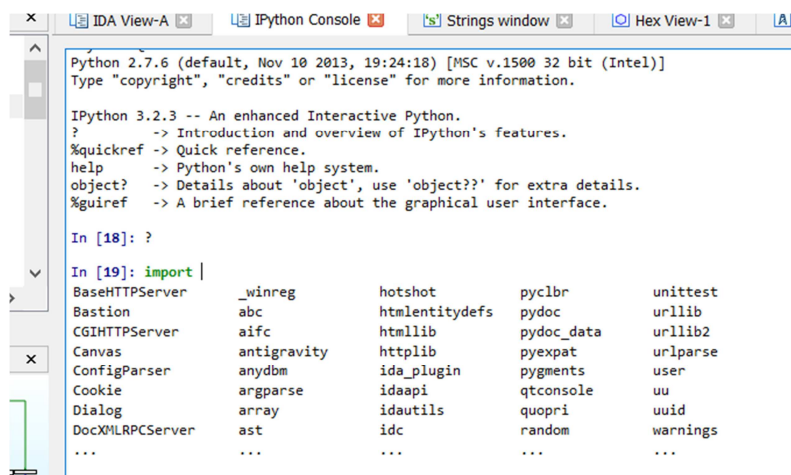
Note that the `'` MUST be the first character on the line! This won't work::

In [4]: `x = ,my_function /home/me` # syntax error


It has a lot of possibilities. I clear the log pressing **ESC**.

The autocompleting option pressing the **TAB** key is really nice.

If I type **imp** and I press **TAB**, it autocompletes the word **import** and I if press **TAB** again...



I see the import possibilities where I can navigate up and down with the direction arrows and I quit pressing **ESC**.



```
In [21]: idaapi?
Type:      module
String form: <module 'idaapi' from 'C:\Program Files (x86)\IDA 6.8\python\idaapi.py'>
File:      c:\program files (x86)\ida 6.8\python\idaapi.py
Docstring:  IDA Plugin SDK API wrapper

In [22]:
```

If I type `?`, it gives a quick info and if I type `??`, it shows the code. It lasts a little more.

```
Type:      module
String form: <module 'idaapi' from 'C:\Program Files (x86)\IDA 6.8\python\idaapi.py'>
File:      c:\program files (x86)\ida 6.8\python\idaapi.py
Source:
# This file was automatically generated by SWIG (http://www.swig.org).
# Version 2.0.12
#
# Do not make changes to this file unless you know what you are doing--modify
# the SWIG interface file instead.

"""
IDA Plugin SDK API wrapper
"""

from sys import version_info
if version_info >= (2,6,0):
    def swig_import_helper():
        from os.path import dirname
        import imp
        fp = None
```

If I quit with **ESC**, I come back where I was.

With the up and down arrows, I can go to the commands I used before.

%hist shows the command history I used.

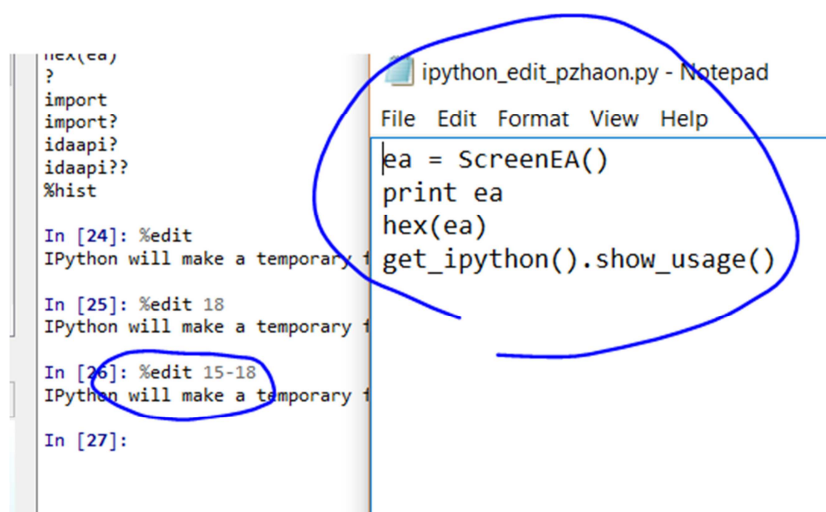
```

In [23]: %hist
%quickref
import idaapi
idaapi
idaapi.
a
a
idaapi
idaapi help
?
idaapi?
idaapi??
idaapi??
dir (idaapi)
ord ("p")
ea = ScreenEA()
print ea
hex(ea)
?

```

%edit opens notepad.

And **%edit x-y** opens notepad with that range lines.



%history -n adds the line numbers to know well if we need to open a range to make a script with edit.


```
In [31]: %history -n
1: %quickref
2: import idaapi
3: idaapi
4: idaapi.
5: a
6: a
7: idaapi
8: idaapi help
9: ?
10: idaapi?
11: idaapi??
12: idaapi??
13: dir (idaapi)
14: ord ("p")
15: ea = ScreenEA()
16: print ea
17: hex(ea)
18: ?
19: import
20: import?
21: idaapi?
22: idaapi??
23: %hist
```

IPython is powerful and it has a lot of commands which can be found here.

<http://ipython.org/ipython-doc/3/index.html>

We will make some simple examples with the **IDAPython** included APIs using this new plugin.

```
In [32]: ea = ScreenEA()

In [33]: hex(ea)
Out[33]: '0x401207L'

In [34]: |
```

The current cursor direction.

```
IPython 3.2.3 -- An enhanced IPython shell
?          -> Introduction and overview
%quickref  -> Quick reference.
help       -> Python's own help system
object?    -> Details about 'object()' and other built-in objects
%gui?      -> A brief reference to the GUI

In [32]: ea = ScreenEA()

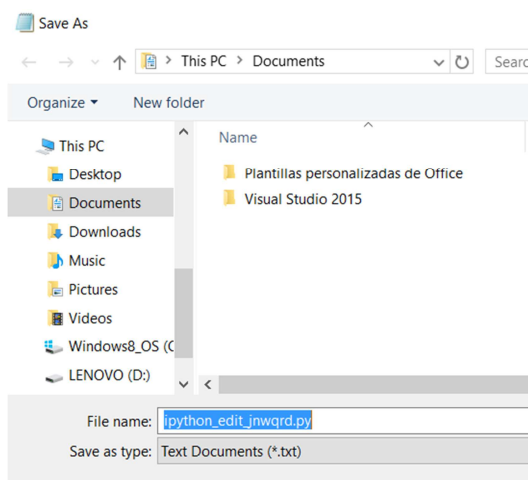
In [33]: hex(ea)
Out[33]: '0x401207L'

In [34]: print(hex(ea))
0x401207L

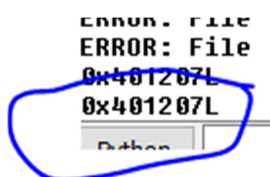
In [35]: %edit 32-34
IPython will make a temporary file...

In [36]:
```

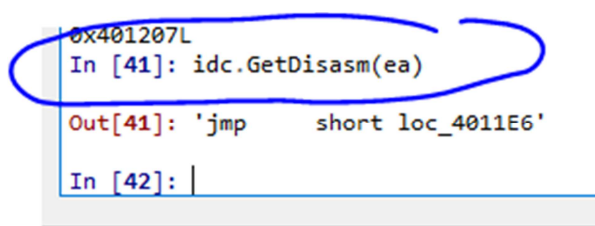
If I write a script and save it.



If we run the script from IDA menu: **FILE-SCRIPT FILE**, it will work.



The **idc.GetDisasm(ea)** command will also give us the instruction where the cursor is.



If I change the cursor to other instruction, it will have to find **ea** again.

```
In [49]: ea = ScreenEA()

In [50]: idc.GetOpnd(ea,1)
Out[50]: '66h'

In [51]: idc.GetOpnd(ea,0)
Out[51]: '[ebp+wParam]'

In [52]:
```

With **idc.GetOpnd**, I can find the first or second instruction operand.

```
In [52]: ea = ScreenEA()
...: func = idaapi.get_func(ea)
...: funcname = GetFunctionName(func.startEA)
...:

In [53]: print funcname
WndProc
```

The current function name.

```
In [54]: for funcea in Functions(SegStart(ea), SegEnd(ea)):
...:     name = GetFunctionName(funcea)
...:     print name
...:

start
WndProc
sub_401253
DialogFunc
CARTEL_BUENO
CARTEL_ERROR
sub_40137E
sub_4013C2
sub_4013D2
sub_4013D8
LoadCursorA
MessageBeep
LoadIconA
SetFocus
MessageBoxA
PostQuitMessage
InvalidateRect
TranslateMessage
ShowWindow
UpdateWindow
```

All segment functions names.

```

In [56]: E = list(FuncItems(ea))
...: for e in E:
...:     print "%X"%e, GetDisasm(e)
...:
401128 enter    0, 0
40112C push     esi
40112D push     edi
40112E push     ebx
40112F cmp      [ebp+Msg], 2
401133 jz       short loc_401193
401135 cmp      [ebp+Msg], 204h
40113C jz       short loc_4011A3
40113E nop
40113F nop
401140 nop
401141 nop
401142 cmp      [ebp+Msg], 5
401146 iz       short loc_4011A5

```

The function instructions.

```

00401128 ; expected entry point
00401128
00401128 ; Attributes: bp-based frame
00401128
00401128 ; int __stdcall WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LI
00401128 public WndProc
00401128 WndProc      proc near
00401128
00401128 hWnd         = dword ptr 8
00401128 Msg          = dword ptr 0Ch
00401128 wParam       = dword ptr 10h
00401128 lParam       = dword ptr 14h
00401128
00401128             enter    0, 0
0040112C             push     esi
0040112D             push     edi
0040112E             push     ebx
0040112F             cmp      [ebp+Msg], 2
00401133             jz       short loc_401193

```

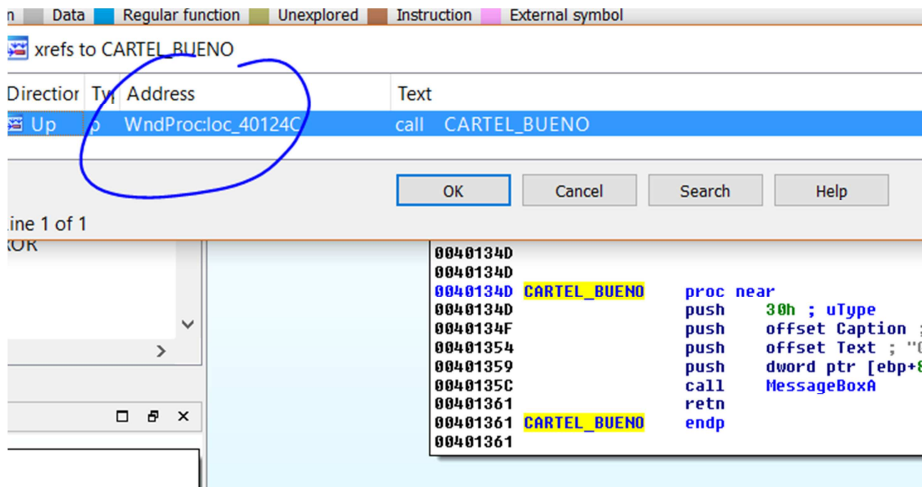
The function references. If we set the cursor at the beginning of a function that has referenced functions and we find **ea** again...

```

0040134D
0040134D
0040134D
0040134D CARTEL_BUENO  proc near
0040134D             push     30h ; uType
0040134F             push     offset Caption ; "Good work!"
00401354             push     offset Text ; "Great work, mate!\rNow try the next Cra"...
00401359             push     dword ptr [ebp+8] ; hWnd
0040135C             call     MessageBoxA
00401361             retn
00401361 CARTEL_BUENO  endp
00401361

```

I see the reference.



```
In [72]: ea = ScreenEA()
...: func = idaapi.get_func(ea)
...: funcname = GetFunctionName(func.startEA)
...:

In [73]: print funcname
CARTEL_BUENO

In [74]: for ref in CodeRefsTo(ea, 1):
...:     print "called from %s (0x%x)" % (GetFunctionName(ref), ref)
...:
called from WndProc (0x40124c)

In [75]:
```

The plugin gives us comfort and IDAPython has a lot of instructions to set breakpoints, log, running the debugger, etc.

INSTALLING PROBLEMS

Some problems occur when we have installed **pip** in Python. You can verify that in IDA before installing it.

In the Python bar, type:

import pip

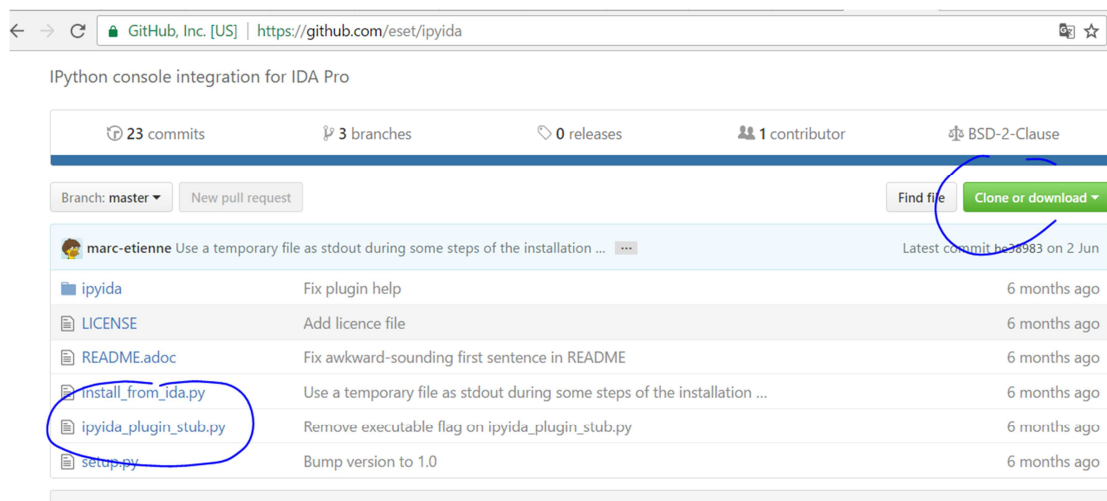
If it doesn't return any error, it means you have pip installed and it will fail. So, open a Windows console and type:

python -m pip uninstall pip setuptools

Restart IDA. Then, you should install the plugin correctly.

If after restarting IDA the plugin doesn't run, download **ipyida_plugin_stub.py** and copy it in the IDA plugin folder.

<https://github.com/eset/ipyida>



Ricardo Narvaja

Translated by: @IvinsonCLS