Introduction to IDAPython

Ero Carrera
ero.carrera@{gmail.com,f-secure.com}

Abstract

IDAPython is an extension for *IDA*, the *Interactive Disassembler*. It brings the power and convenience of *Python* scripting to aid in the analysis of binaries. This article will cover some basic usage and provide examples to get interested individuals started. We will walk through practical examples ranging from iterating through functions, segments and instructions to data mining the binaries, collecting references and analyzing their structure.

Python and IDA

Python is a powerful scripting language which has features greatly appreciated by its followers. Versatility, speed of development and readability are among the top ones.

IDA ,(Datarescue 2005), provides the advanced user with *IDC*, a C-like scripting language to automate some of the tasks of analysis. Yet, compared to *Python*, *IDC* feels clumsy and slow. Many times has the author (and others) wished for something more versatile.

IDAPython, (Erdélyi 2005), was first introduced in an earlier joint paper, (Carrera and Erdélyi 2004), where a general overview was given together with minimal examples comparing *IDC* and equivalent *Python* scripts.

Python goes well beyond the possibilities of *IDC* by providing networking support, avanced I/O and a host of other features not available in *IDC* at all.

In this article, a series of examples will be introduced in order to get acquainted with *IDAPython* and its possibilities.

The examples presented in this paper are known to work with *IDA* 4.8 and *IDAPython* 0.7.0. running under Linux.

IDAPython keeps the same global dictionary regardless of the input method. Whether *Python* code is run from external files or typed in its notepad, the data is persistent. This is extremely convenient as one might want to run a script that will gather and parse certain data but does not yet know, or want to, do anything further with it. Having such data always accessible sets a wonderful environment for poking and tinkering around.

IDAPython provides the full API available to those writing plugins and also the well known *IDC* functions. It's possible to access nearly anything within *IDA*'s database.

Walking the functions

As an introductory script, the first example will loop through all the functions *IDA* has found and any others the user has defined, and will print their effective addresses and names. (The script is nearly identical to one of the examples in (Carrera and Erdélyi 2004))

Walk the functions

```
# Get the segment's starting address
ea = ScreenEA()

# Loop through all the functions
for function_ea in Functions(SegStart(ea), SegEnd(ea)):

    # Print the address and the function name.
    print hex(function_ea), GetFunctionName(function_ea)
```

Functions such as *ScreenEA* and *GetFunctionName* exist also in *IDC* and documentation for them can be found at .

The functions *Functions()*, is provided by *IDAPython*'s *idautils* module, which is automatically imported on load.

Walking the segments

This example will loop though all segments and fetch their data, byte by byte, storing it in a *Python* string.

Going through the segments

```
segments = dict()

# For each segment
for seg_ea in Segments():

    data = []

    # For each byte in the address range of the segment
    for ea in range(seg_ea, SegEnd(seg_ea)):

        # Fetch byte
        data.append(chr(Byte(ea)))

    # Put the data together
    segments[SegName(seg_ea)] = ''.join(data)

# Loop through the dictionary and print the segment's names
# and their sizes
for seg_name, seg_data in segments.items():
    print seg_name, len(seg_data)
```

The function *Segments()* is again provided by *idautils*. *Byte()*, *SegEnd()* and *SegName()* exist in *IDC* and their functionality is quite self-evident.

Function connectivity

The third example is a bit more elaborate. It will go through all the functions and will find all the calls performed to and from each of them. The references will be stored in two dictionaries and, in the end, a list of functions with their indegree and outdegree will be shown.

Indegree and outgedree of functions

```
from sets import Set

# Get the segment's starting address
ea = ScreenEA()

callers = dict()
callees = dict()

# Loop through all the functions
for function_ea in Functions(SegStart(ea), SegEnd(ea)):

    f_name = GetFunctionName(function_ea)

    # Create a set with all the names of the functions calling (referring to)
    # the current one.
    callers[f_name] = Set(map(GetFunctionName, CodeRefsTo(function_ea, 0)))

    # For each of the incoming references
    for ref_ea in CodeRefsTo(function_ea, 0):

        # Get the name of the referring function
        caller_name = GetFunctionName(ref_ea)

        # Add the current function to the list of functions
        # called by the referring function
        callees[caller_name] = callees.get(caller_name, Set())
        callees[caller_name].add(f_name)

# Get the list of all functions
functions = Set(callees.keys()+callers.keys())

# For each of the functions, print the number of functions calling it and
# number of functions being called. In short, indegree and outdegree
for f in functions:
    print '%d:%s:%d' % (len(callers.get(f, [])), f, len(callees.get(f, [])))
```

Walking the instructions

The fourth example will take us to the instruction level. For each segment, we will walk through all the defined elements, by means of *Heads(start address, end address)* and check whether the element defined at each address is an instruction; if so, the mnemonic will be fetched and its occurrence count will be updates in the *mnemonics* dictionary.

Finally, the mnemonics and their number of occurrences are shown.

Nmemonics histogram

```
mnemonics = dict()

# For each of the segments
for seg_ea in Segments():

    # For each of the defined elements
    for head in Heads(seg_ea, SegEnd(seg_ea)):

        # If it's an instruction
        if isCode(GetFlags(head)):

            # Get the mnemonic and increment the mnemonic
            # count
            mnem = GetMnem(head)
            mnemonics[mnem] = mnemonics.get(mnem, 0)+1

# Sort the mnemonics by number of occurrences
sorted = map(lambda x:(x[1], x[0]), mnemonics.items())
sorted.sort()

# Print the sorted list
for mnemonic, count in sorted:
    print mnemonic, count
```

Cyclomatic complexity

The next example goes a bit further. It will go through all the functions and for each of them it will compute the *Cyclomatic Complexity*. The *Cyclomatic Complexity* measures the complexity of the code by looking at the nodes and edges (basic blocks and branches) of the graph of a function. It is usually defined as:

$$CC = Edges - Nodes + 2$$

The function *cyclomatic_complexity()* will compute its value, given the function's start address as input.

The example can be run in two different modes. The first one is invoked as usual, through *IDAPython*, by locating the *Python* script and running it. A second way is to launch IDA and make it run the script in batch mode; that will be explored in the next section.

In this example function chunks are not considered. IDA added in recent versions, support for function chunks, which are a result of some compiler's optimization process. It is possible to walk the chunks by using the function API function *func_tail_iterator_t()* . The following code shows how to iterate through the chunks.

Collecting function chunks

```
function_chunks = []

#Get the tail iterator
func_iter = func_tail_iterator_t(get_func(ea))

# While the iterator's status is valid
status = func_iter.main()
while status:
    # Get the chunk
    chunk = func_iter.chunk()

    # Store its start and ending address as a tuple
    function_chunks.append((chunk.startEA, chunk.endEA))

    # Get the last status
    status = func_iter.next()
```

Cyclomatic Complexity

```
import os
from sets import Set

def cyclomatic_complexity(function_ea):
    """Calculate the cyclomatic complexity measure for a function.

    Given the starting address of a function, it will find all the basic block's boundaries and edges
    between them and will return the cyclomatic complexity, defined as:
        CC = Edges - Nodes + 2
    """

    f_start = function_ea
    f_end = FindFuncEnd(function_ea)

    edges = Set()
    boundaries = Set((f_start,))

    # For each defined element in the function.
    for head in Heads(f_start, f_end):

        # If the element is an instruction
        if isCode(GetFlags(head)):

            # Get the references made from the current instruction and keep only the ones local to
            # the function.
            refs = CodeRefsFrom(head, 0)
            refs = Set(filter(lambda x: x>=f_start and x<=f_end, refs))

            if refs:
                # If the flow continues also to the next (address-wise) instruction, we add a reference to it.
                # For instance, a conditional jump will not branch if the condition is not met, so we save that
                # reference as well.
                next_head = NextHead(head, f_end)

                if isFlow(GetFlags(next_head)):
                    refs.add(next_head)

                # Update the boundaries found so far.
                boundaries.union_update(refs)

                # For each of the references found, and edge is created.
                for r in refs:
                    # If the flow could also come from the address previous to the destination of the branching
                    # an edge is created.
                    if isFlow(GetFlags(r)):
                        edges.add((PrevHead(r, f_start), r))

                    edges.add((head, r))

    return len(edges) - len(boundaries) + 2
```

Cyclomatic Complexity

```
def do_functions():
    cc_dict = dict()

    # For each of the segments
    for seg_ea in Segments():
        # For each of the functions
        for function_ea in Functions(seg_ea, SegEnd(seg_ea)):
            cc_dict[GetFunctionName(function_ea)] = cyclomatic_complexity(function_ea)

    return cc_dict

# Wait until IDA has done all the analysis tasks.
# If loaded in batch mode, the script will be run before everything is finished, so the script will explicitly
# wait until the autoanalysis is done.
autoWait()

# Collect data
cc_dict = do_functions()

# Get the list of functions and sort it.
functions = cc_dict.keys()
functions.sort()
ccs = cc_dict.values()

# If the environment variable IDAPYTHON exists and its value is 'auto'  the results will be appended
# to a data file and the script will quit IDA. Otherwise it will just output the results
if os.getenv('IDAPYTHON') == 'auto':

    results = file('example5.dat', 'a+')

    results.write('%3.4f,%03d,%03d %s\n' % (
        sum(ccs)/float(len(ccs)), max(ccs), min(ccs), GetInputFile()))

    results.close()

    Exit(0)

else:
    # Print the cyclomatic complexity for each of the functions.

    for f in functions:
        print f, cc_dict[f]

    # Print the maximum, minimum and average cyclomatic complexity.
    print 'Max: %d, Min: %d, Avg: %f' % (max(ccs), min(ccs), sum(ccs)/float(len(ccs)))
```

Automating *IDA* through *IDAPython*

As mentioned in the last section, the previous example has a a second way of operating. *IDAPython* now supports to run *Python* scripts on start up, from the command line. Such functionality comes handy, to say the least, when analyzing a set of binaries in batch mode.

The switch *-OIDAPython:/path/to/python/script.py* can be used to tell *IDAPython* which script to run. Another switch which might come handy is *-A* which will instruct *IDA* to run in batch mode, not asking anything, just performing the auto-analysis.

With those two options combined it is possible to auto-analyze a binary and run a *Python* script to perform some mining. A function which will be usually required is *autoWait()* which will instruct the *Python* script to wait until *IDA* is done performing the analysis. It is a good idea to call it in the beginning of any script.

To analyze a bunch of files a command like the following could be entered (if working in Bash on Linux).

for virus in virus/.idb; do IDAPYTHON='auto' idal -A -OIDAPython:example5.py $virus; done*

It will go through all the *.idb* files in the *virus/* directory and will invoke *idal* which each of them, running the script *example5.py* on load.

The script is the one in the last example. If it finds the environment variable *IDAPYTHON*, it will just collect the data and append it to a file instead of showing it in IDA's messages window. Subsequently it will call *Exit()* to close the database and quit.

It would be equally easy to batch mode analyze a set of executables. If *IDB* files are given, *IDA* will just load them and no auto-analysis will be performed; otherwise, if a binary file is provided the analysis will be done and the script run once finished.

All this allows for a good degree of automation in analysis of a set of binaries. For instance, the next table is the output of running the previous script on a bunch of malware *IDBs*. A nice feature is to see the clear clustering of the families by their cyclomatic complexity features.

Output of running the example in batch mode on a set of malware binaries.

Sample	Cyclomatic complexity Avg, Max, Min, Filename
Klez	7.4197,148,001 klez_a.ex 7.4975,148,001 klez_b.ex 7.5972,148,001 klez_c.ex 7.5972,148,001 klez_d.ex 7.0349,148,001 klez_e.ex 7.0502,148,001 klez_f.ex 7.0502,148,001 klez_g.ex 7.0573,148,001 klez_h.ex 7.0573,148,001 klez_i.ex 7.0502,148,001 klez-j.ex
Mimail	3.2190,052,001 mimailA.ex_.1.unp 3.2353,052,001 mimailB.ex_ 3.2313,052,001 mimailC.ex_.1.unp 3.4148,052,001 mimailD.ex_ 2.8110,052,001 mimailE.ex_.1.unp 2.7953,052,001 mimailF.ex_.1.unp 2.7638,052,001 mimailG.ex_.1.unp 2.7874,052,001 mimailH.ex_.1.unp 2.8376,052,001 mimailI.ex_.1.unp 2.8632,052,001 mimailJ.ex_ 2.8984,052,001 mimailL.ex_.1.unp 2.8231,052,001 mimail-m_u.ex 3.4375,052,001 outlook_.dmp 3.1138,052,001 mimail-s_u.ex
Sasser	6.5301,039,001 sasser.avpe 6.5422,039,001 sasser-b.avpe 6.6098,039,001 sasser-c.avpe 6.5955,041,001 sasser-d.ex_unp.exe 6.5444,041,001 sasser-e.unp 6.8452,041,001 sasser-f.unp 8.0000,041,001 sasser-g.unp
Netsky	7.3505,041,001 netskyaa.unp 7.4947,041,001 netsky_unk.unp 7.1667,041,001 netsky_ac.ex_unp 5.9694,051,001 Netsky.AD.unp 7.3125,041,001 virus.ex_.1.unp 7.2478,041,001 your_details.doc.exe.2.unp 8.0407,123,001 userconfig9x.dl.1.unp 7.9068,041,001 netsky-q-dll.unp 7.9068,041,001 netsky-q-dll.unp 7.5702,041,001 netsky-r-dll_unp_.exe 7.5657,041,001 list0_unp_.pif 7.5743,041,001 private.unp.pi_ 7.5268,041,001 netsky_v_unp_.exe 7.8824,041,001 netsky-w.unp 6.8165,041,001 netsky.pif.2.unp

Visualizing binaries

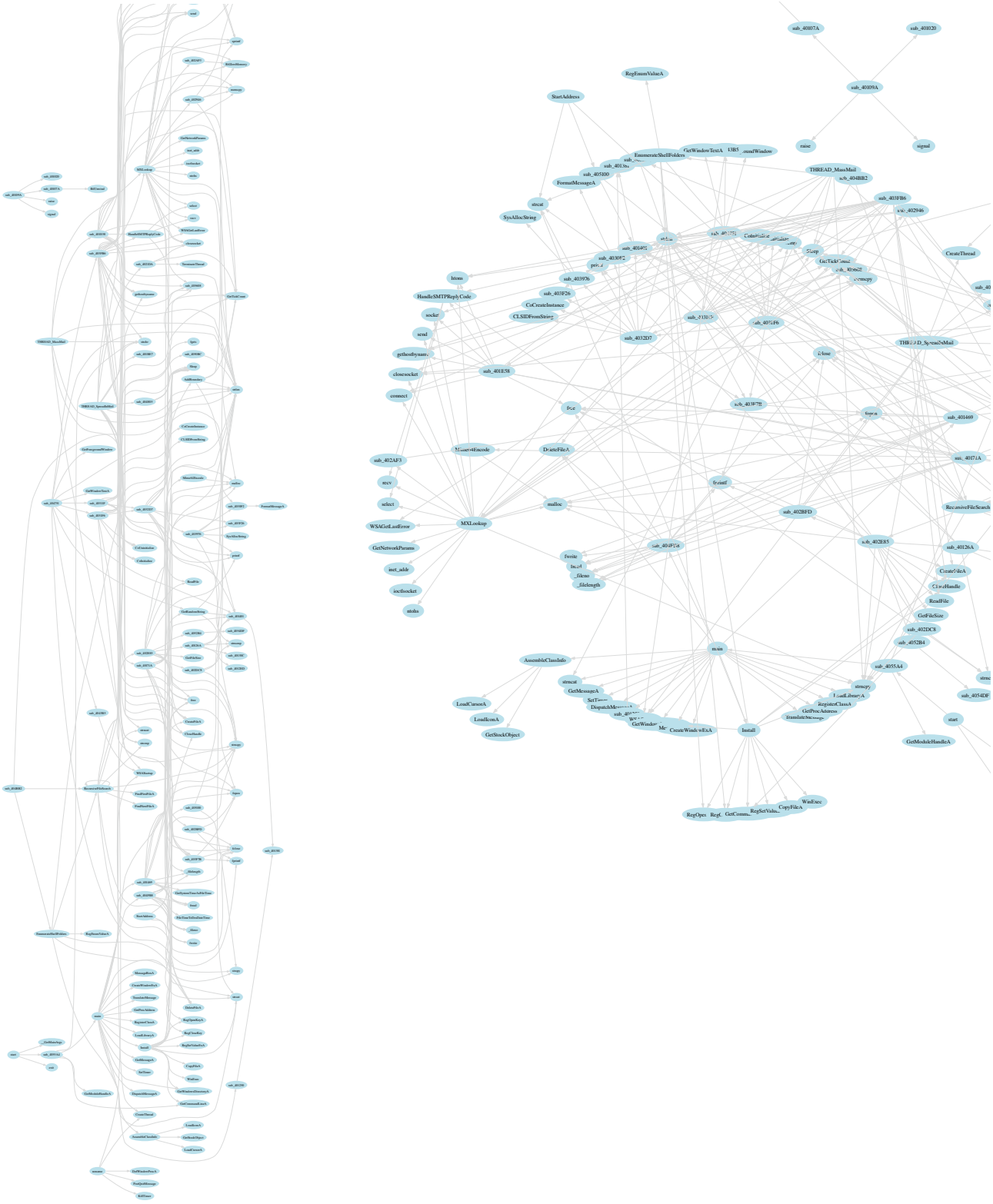
This example is based on the one collecting the indegrees and outdegree of all functions. This time, we will use that information to generate a graph of the call-tree and plot it using *pydot*, (Carrera 2005a); a package to interface *Graphviz*, (Ellson et al. 2005).

The code follows, the only changes from the example it is based on, are the lines creating the graph, setting some defaults and then adding the edges.

Visualizing binaries

```
from sets import Set
import pydot

# Get the segment's starting address
ea = ScreenEA()

callees = dict()

# Loop through all the functions
for function_ea in Functions(SegStart(ea), SegEnd(ea)):

    f_name = GetFunctionName(function_ea)

    # For each of the incoming references
    for ref_ea in CodeRefsTo(function_ea, 0):

        # Get the name of the referring function
        caller_name = GetFunctionName(ref_ea)

        # Add the current function to the list of functions called by the referring function
        callees[caller_name] = callees.get(caller_name, Set())
        callees[caller_name].add(f_name)

# Create graph
g = pydot.Dot(type='digraph')

# Set some defaults
g.set_rankdir('LR')
g.set_size('11,11')
g.add_node(pydot.Node('node', shape='ellipse', color='lightblue', style='filled'))
g.add_node(pydot.Node('edge', color='lightgrey'))

# Get the list of all functions
functions = callees.keys()

# For each of the functions and each of the called ones, add the corresponding edges.
for f in functions:
    if callees.has_key(f):
        for f2 in callees[f]:
            g.add_edge(pydot.Edge(f, f2))

# Write the output to a Postscript file
g.write_ps('example6.ps')
```

Some examples output is shown next, the different plots are obtained by using the different plotting utilities provided by *Graphviz*.



Projects using *IDAPython*

It might be also useful to check some already existing projects based solely on *IDAPython*. Some of them are:

- idb2reml, (Carrera 2005); will export *IDB* information to a XML format, *REML* (ReverseEngineering ML)
- pyreml, (Carrera 2005a); loads the *REML* produced by idb2reml and provides a set of functions to perform advanced analysis.

Function Reference

AddEntryPoint()

```
Add entry point

      ordinal  - entry point number
                 if entry point doesn't have an ordinal
                 number, 'ordinal' should be equal to 'ea'
      ea       - address of the entry point
      name     - name of the entry point. If null string,
                 the entry point won't be renamed.
      makecode - if 1 then this entry point is a start
                 of a function. Otherwise it denotes data bytes.

      returns:         0 - entry point with the specifed ordinal already
                 exists
                 1 - ok
```

AddHotkey()

```
Add hotkey for IDC function

      Arguments:

      hotkey  - hotkey name ('a', "Alt-A", etc)
      idcfunc - IDC function name
            GUI version doesn't support hotkeys

      Returns: -
```

AltOp()

```
Get manually entered operand string

      Arguments:
      ea - linear address
      n  - number of operand:
           0 - the first operand
           1 - the second operand

      Returns: string or None if it fails
```

AnalyseArea()

```
Perform full analysis of the area

      Arguments:

      sEA - starting linear address
      eEA - ending linear address (excluded)

      Returns: 1-ok, 0-Ctrl-Break was pressed.
```

AskAddr()

```
Ask the user to enter an address

      Arguments:

      defval - the default address value. This value
               will appear in the dialog box.
      prompt - the prompt to display in the dialog box

      Returns: the entered address or BADADDR.
```

AskFile()

```
Ask the user to choose a file

      Arguments:

      forsave - 0: "Open" dialog box, 1: "Save" dialog box
      mask    - the input file mask as "*.*" or the default file name.
      prompt  - the prompt to display in the dialog box

      Returns: the selected file or 0.
```

AskIdent()

```
Ask the user to enter an identifier

      Arguments:

      defval - the default identifier. This value will appear in
               the dialog box.
      prompt - the prompt to display in the dialog box

      Returns: the entered identifier or 0.
```

AskLong()

```
Ask the user to enter a number
```

```
      Arguments:

      defval - the default value. This value
               will appear in the dialog box.
      prompt - the prompt to display in the dialog box

      Returns: the entered number or -1.
```

AskSeg()

```
Ask the user to enter a segment value

      Arguments:

      defval - the default value. This value
               will appear in the dialog box.
      prompt - the prompt to display in the dialog box

      Returns: the entered segment selector or BADSEL.
```

AskSelector()

```
Get a selector value

      sel - the selector number (16bit value)

      returns:        selector value if found
                  otherwise the input value (sel)

      note:           selector values are always in paragraphs
```

AskStr()

```
Ask the user to enter a string

      Arguments:
      defval - the default string value. This value will appear
               in the dialog box.
      prompt - the prompt to display in the dialog box

      Returns: the entered string or 0.

      FIXME: Doublecheck the history type
```

AskYN()

```
Ask the user a question and let him answer Yes/No/Cancel

      Arguments:
```

```
      efval - the default answer. This answer will be selected if the user
              presses Enter. -1:cancel,0-no,1-ok
      prompt - the prompt to display in the dialog box

      Returns: -1:cancel,0-no,1-ok
```

AutoMark()

```
Plan to analyse an address
```

AutoMark2()

```
Plan to perform an action in the future.
      This function will put your request to a special autoanalysis queue.
      Later IDA will retrieve the request from the queue and process
      it. There are several autoanalysis queue types. IDA will process all
      queries from the first queue and then switch to the second queue, etc.
```

AutoUnmark()

```
Remove range of addresses from a queue.
```

Batch()

```
Enable/disable batch mode of operation

      Arguments:
      batch - Batch mode
              0 - ida will display dialog boxes and wait for the user input
              1 - ida will not display dialog boxes, warnings, etc.

      Returns: old balue of batch flag
```

Byte()

```
Get value of program byte

      ea - linear address

      returns: value of byte. If byte has no value then returns 0xFF
            If the current byte size is different from 8 bits, then the re-
turned value
            might have more 1's.
            To check if a byte has a value, use functions hasValue(Get-
Flags(ea))
```

Choose()

```
Choose - class for choose() with callbacks
```

ChooseFunction()

```
Ask the user to select a function

      Arguments:

      title - title of the dialog box

      Returns: -1 - user refused to select a function
               otherwise returns the selected function start address
```

CodeRefsFrom()

```
Get a list of code references from 'ea'

      ea   - Target address
      flow - 0 - don't follow normal code flow
           - 1 - follow code flow

      Return: list of references (may be empty list)
```

CodeRefsTo()

```
Get a list of code references to 'ea'

      ea   - Target address
      flow - 0 - don't follow normal code flow
           - 1 - follow code flow

      Return: list of references (may be empty list)
```

Comment()

```
Get regular indented comment

      Arguments:
      ea - linear address

      Returns: string or None if it fails
```

DataRefsFrom()

```
Get a list of data references from 'ea'

      ea   - Target address

      Return: list of references (may be empty list)
```

DataRefsTo()

```
Get a list of data references to 'ea'

      ea   - Target address

      Return: list of references (may be empty list)
```

DelCodeXref()

```
Unmark exec flow 'from' 'to'

      undef - make 'To' undefined if no more references to it

      returns 1 - planned to be made undefined
```

DelExtLnA()

```
Delete an additional anterior line

      Arguments:

      ea   - linear address
      n    - number of anterior additioal line (0..500)

      Returns: -
```

DelExtLnB()

```
Delete an additional posterior line

      Arguments:

      ea   - linear address
      n    - number of posterior additioal line (0..500)

      Returns: -
```

DelFixup()

```
Delete fixup information
```

```
      ea - address to delete fixup information about

      returns:        none
```

DelFunction()

```
Delete a function

      ea - any address belonging to the function

      returns:        !=0 - ok
```

DelHiddenArea()

```
Delete a hidden area

      Arguments:

      ea - any address belonging to the hidden area

      Returns:  !=0 - ok
```

DelHotkey()

```
Delete IDC function hotkey

      Arguments:

      hotkey - hotkey code to delete
```

DelSelector()

```
Delete a selector

      sel - the selector number to delete

      returns:        nothing

      note:           if the selector is found, it will be deleted
```

Demangle()

```
Demangle a name

      name - name to demangle
      disable_mask - a mask that tells how to demangle the name
                  it is a good idea to get this mask using
                  GetLongPrm(INF_SHORT_DN) or GetLongPrm(INF_LONG_DN)
```

```
      Returns: a demangled name
            If the input name cannot be demangled, returns 0
```

Dfirst()

```
Get first referred address
```

DfirstB()

```
Get first referee address
```

Dword()

```
Get value of program double word (4 bytes)

      ea - linear address

      returns: the value of the double word. If double word has no value
            then returns 0xFFFFFFFF.
```

Exec()

```
Execute an OS command.

      Arguments:
      command - command line to execute

      Returns: error code from OS

      Note:
      IDA will wait for the started program to finish.
      In order to start the command in parallel, use OS methods.
      For example, you may start another program in parallel using "start"
command.
```

Exit()

```
Stop execution of IDC program, close the database and exit to OS

      Arguments:
      code - code to exit with.

      Returns: -
```

ExtLinA()

```
Specify an additional line to display before the generated ones.

      Arguments:

      ea   - linear address
      n    - number of anterior additioal line (0..MAX_ITEM_LINES)
      line - the line to display

      Returns: -

      Notes:

      IDA displays additional lines from number 0 up to the first unexisting
      additional line. So, if you specify additional line #150 and there is
no
      additional line #149, your line will not be displayed.  MAX_ITEM_LINES
is
      defined in IDA.CFG
```

ExtLinB()

```
Specify an additional line to display after the generated ones.

      Arguments:

      ea   - linear address
      n    - number of posterior additioal line (0..MAX_ITEM_LINES)
      line - the line to display

      Returns: -

      IDA displays additional lines from number 0 up to the first unexisting
      additional line. So, if you specify additional line #150 and there is
no
      additional line #149, your line will not be displayed.  MAX_ITEM_LINES
is
      defined in IDA.CFG
```

Fatal()

```
Display a fatal message in a message box and quit IDA

      format - message to print
```

FindFuncEnd()

```
# ********************************************
# ** Determine a new function boundaries
# **
#         arguments:      ea  - starting address of a new function
```

```
#         returns:        if a function already exists, then return
#                         its end address.
#                         if a function end cannot be determined,
#                         the return BADADDR
#                         otherwise return the end address of the new func-
tion
```

FindSelector()

```
Find a selector which has the specifed value

      val - value to search for

      returns:        16bit selector number if found
                  otherwise the input value (val  0xFFFF)

      note:           selector values are always in paragraphs
```

FirstSeg()

```
Get first segment

      returns: linear address of the start of the first segment
            BADADDR - no segments are defined
```

Functions()

```
Get a list of functions

      In:

      start - start address
      end   - end address

      Return:

      list of heads between start and end

      Note:

      The last function that starts before 'end' is included even
      if it extends beyond 'end'.
```

GetBmaskCmt()

```
Get bitmask comment (only for bitfields)

      Arguments:
```

```
      enum_id - id of enum
      bmask   - bitmask of the constant
      repeatable - type of comment, 0-regular, 1-repeatable

      Returns: comment attached to bitmask if it exists.
               otherwise returns 0.

      FIXME: Check the return value
```

GetBmaskName()

```
Get bitmask name (only for bitfields)

      Arguments:

      enum_id - id of enum
      bmask   - bitmask of the constant

      Returns: name of bitmask if it exists. otherwise returns 0.

      FIXME: Check the return value
```

GetConstBmask()

```
Get bit mask of symbolic constant

      Arguments:
      const_id - id of symbolic constant

      Returns: bitmask of constant or 0
               ordinary enums have bitmask = -1
```

GetConstByName()

```
Get member of enum - a symbolic constant ID

      Arguments:
      name - name of symbolic constant

      Returns: ID of constant or -1

      FIXME: Need to check the return type!
```

GetConstCmt()

```
Get comment of a constant

      Arguments:
```

```
      const_id - id of const
      repeatable - 0:get regular comment
                   1:get repeatable comment

      Returns: comment string
```

GetConstEnum()

```
Get id of enum by id of constant

      Arguments:
      const_id - id of symbolic constant

      Returns: id of enum the constant belongs to.
             -1 if const_id is bad.
```

GetConstEx()

```
Get id of constant

      Arguments:
      enum_id - id of enum
      value   - value of constant
      serial  - serial number of the constant in the
                enumeration. See OpEnumEx() for details.
      bmask   - bitmask of the constant
                ordinary enums accept only -1 as a bitmask

      Returns:  id of constant or -1 if error
```

GetConstName()

```
Get name of a constant

      Arguments:

      const_id - id of const

      Returns: name of constant
```

GetConstValue()

```
Get value of symbolic constant

      Arguments:
      const_id - id of symbolic constant

      Returns: value of constant or 0
```

GetDouble()

```
Get value of a floating point number (8 bytes)

      Arguments:
      ea - linear address

      Returns: double
```

GetEntryOrdinal()

```
Retrieve entry point ordinal number

      index - 0..GetEntryPointQty()-1

      returns:        0 if entry point doesn't exist
                  otherwise entry point ordinal
```

GetEntryPoint()

```
Retrieve entry point address

      ordinal - entry point number
            it is returned by GetEntryPointOrdinal()

      returns:        -1 if entry point doesn't exist
                  otherwise entry point address.
                  If entry point address is equal to its ordinal
                  number, then the entry point has no ordinal.
```

GetEntryPointQty()

```
retrieve number of entry points

      returns:        number of entry points
```

GetEnum()

```
Get enum ID by the name of enum

      Arguments:
      name - name of enum

      returns:        ID of enum or -1 if no such enum exists
```

GetEnumCmt()

```
Get comment of enum

      Arguments:
      enum_id    - ID of enum
      repeatable - 0:get regular comment
                   1:get repeatable comment

      Returns:        comment of enum
```

GetEnumFlag()

```
Get flag of enum

      Arguments:
      enum_id - ID of enum

      Returns: flags of enum. These flags determine representation
               of numeric constants (binary,octal,decimal,hex)
              in the enum definition. See start of this file for
              more information about flags.
              Returns 0 if enum_id is bad.
```

GetEnumIdx()

```
Get serial number of enum by its ID

      Arguments:
      enum_id - ID of enum

      Returns:        (0..GetEnumQty()-1) or -1 if error
```

GetEnumName()

```
Get name of enum

      Arguments:
      enum_id - ID of enum

      Returns:        name of enum or empty string
```

GetEnumQty()

```
Get number of enum types

      Arguments:      none

      Returns:        number of enumerations
```

GetEnumSize()

```
Get size of enum

      Arguments:
      enum_id - ID of enum

      Returns:  number of constants in the enum
                Returns 0 if enum_id is bad.
```

GetFirstBmask()

```
Get first bitmask in the enum (bitfield)

      Arguments:
      enum_id - id of enum (bitfield)

      Returns: the smallest bitmask of constant or -1
               no bitmasks are defined yet
               All bitmasks are sorted by their values
               as unsigned longs.
```

GetFirstConst()

```
Get first constant in the enum

      Arguments:

      enum_id - id of enum
      bmask   - bitmask of the constant
                ordinary enums accept only -1 as a bitmask

      returns: value of constant or -1 no constants are defined
               All constants are sorted by their values as unsigned longs.
```

GetFirstStrucIdx()

```
Get index of first structure type

      In: none

      returns:        -1 if no structure type is defined
                   index of first structure type.
                   Each structure type has an index and ID.
                   INDEX determines position of structure definition
                   in the list of structure definitions. Index 1
                   is listed first, after index 2 and so on.
                   The index of a structure type can be changed any
                   time, leading to movement of the structure definition
```

```
                    in the list of structure definitions.
                    ID uniquely denotes a structure type. A structure
                    gets a unique ID at the creation time and this ID
                    can't be changed. Even when the structure type gets
                    deleted, its ID won't be resued in the future.
```

GetFixupTgtDispl()

```
Get fixup target displacement

      ea - address to get information about

      returns:    -1 - no fixup at the specified address
                       otherwise returns fixup target displacement
```

GetFixupTgtOff()

```
Get fixup target offset

      ea - address to get information about

      returns:   -1 - no fixup at the specified address
                       otherwise returns fixup target offset
```

GetFixupTgtSel()

```
Get fixup target selector

      ea - address to get information about

      returns:        -1 - no fixup at the specified address
                             otherwise returns fixup target selector
```

GetFixupTgtType()

```
Get fixup target type

      ea - address to get information about

      returns:    -1 - no fixup at the specified address
                       otherwise returns fixup target type:
```

GetFlags()

```
Get internal flags

      ea - linear address
```

```
      returns: 32-bit value of internal flags. See start of IDC.IDC file
            for explanations.
```

GetFloat()

```
Get value of a floating point number (4 bytes)

      Arguments:
      ea - linear address

      Returns: float
```

GetFunctionCmt()

```
Retrieve function comment

      ea - any address belonging to the function
      repeatable - 1: get repeatable comment
                   0: get regular comment

      returns:        function comment string
```

GetFunctionFlags()

```
Retrieve function flags

      arguments:      ea - any address belonging to the function

      returns:        -1 - function doesn't exist otherwise returns the flags
```

GetFunctionName()

```
Retrieve function name

      ea - any address belonging to the function

      returns:        null string - function doesn't exist
                  otherwise returns function name
```

GetIdaDirectory()

```
Get IDA directory

      This function returns the directory where IDA.EXE resides
```

GetIdbPath()

```
Get IDB full path

      This function returns full path of the current IDB database
```

GetInputFile()

```
Get input file name

      This function returns name of the file being disassembled
```

GetInputFilePath()

```
Get input file path

      This function returns the full path of the file being disassembled
```

GetLastBmask()

```
Get last bitmask in the enum (bitfield)

      Arguments:

      enum_id - id of enum

      Returns: the biggest bitmask or -1 no bitmasks are defined yet
               All bitmasks are sorted by their values as unsigned longs.
```

GetLastConst()

```
Get last constant in the enum

      Arguments:

      enum_id - id of enum
      bmask   - bitmask of the constant
                ordinary enums accept only -1 as a bitmask

      Returns: value of constant or -1 no constants are defined
               All constants are sorted by their values
               as unsigned longs.
```

GetLastStrucIdx()

```
Get index of last structure type

      Arguments:
      none
```

```
      returns:        -1 if no structure type is defined
                    index of last structure type.
                    See GetFirstStrucIdx() for the explanation of
                    structure indices and IDs.
```

GetManualInsn()

```
Get manual representation of instruction

      ea   - linear address

      This function returns value set by SetManualInsn earlier.
```

GetMnem()

```
Get instruction mnemonics

      ea - linear address of instruction

      returns: "" - no instruction at the specified location

      note: this function may not return exactly the same mnemonics
      as you see on the screen.
```

GetNextBmask()

```
Get next bitmask in the enum (bitfield)

      Arguments:

      enum_id - id of enum
      bmask   - value of the current bitmask

      Returns:  value of a bitmask with value higher than the specified
                value. -1 if no such bitmasks exist.
                All bitmasks are sorted by their values
                as unsigned longs.
```

GetNextConst()

```
Get next constant in the enum

      Arguments:

      enum_id - id of enum
      bmask   - bitmask of the constant
                ordinary enums accept only -1 as a bitmask
      value   - value of the current constant
```

```
      Returns: value of a constant with value higher than the specified
               value. -1 no such constants exist.
               All constants are sorted by their values as unsigned longs.
```

GetNextFixupEA()

```
Find next address with fixup information

      ea - current address

      returns:    -1 - no more fixups otherwise returns the next
                       address with fixup information
```

GetNextStrucIdx()

```
Get index of next structure type

      Arguments:
      index -      current structure index

      Returns:    -1 if no (more) structure type is defined
                       index of the next structure type.
                       See GetFirstStrucIdx() for the explanation of
                       structure indices and IDs.
```

GetOpType()

```
Get type of instruction operand

      ea - linear address of instruction
      n  - number of operand:
            0 - the first operand
            1 - the second operand

      Returns:
            -1      bad operand number passed
            0       None
            1       General Register
            2       Memory Reference
            3       Base + Index
            4       Base + Index + Displacement
            5       Immediate
            6       Immediate Far Address (with a Segment Selector)
            7       Immediate Near Address
      PC:
            8       386 Trace register
            9       386 Debug register
            10      386 Control register
            11      FPP register
            12      MMX register
```

```
8051:
      8       bit
      9       /bit
      10      bit
80196:
      8       [intmem]
      9       [intmem]+
      10      offset[intmem]
      11      bit
ARM:
      8       shifted register
      9       MLA operands
      10      register list (for LDM/STM)
      11      coprocessor register list (for CDP)
      12      coprocessor register (for LDC/STC)
PPC:
      8       SPR
      9       2 FPRs
      10      SH  MB  ME
      11      CR field
      12      CR bit
TMS320C5:
      8       bit
      9       bit not
      10      condition
TMS320C6:
      8       register pair (A1:A0..B15:B14)
Z8:
      8       @intmem
      9       @Rx
Z80:
      8       condition
```

GetOperandValue()

```
Get number used in the operand

      This function returns an immediate number used in the operand

      Arguments:
      ea - linear address of instruction
      n  - number of operand:
           0 - the first operand
           1 - the second operand

      Returns: value
               If the operand doesn't contain a number, it returns -1.
```

GetOpnd()

```
Get operand of an instruction

      ea - linear address of instruction
      n  - number of operand:
             0 - the first operand
             1 - the second operand

      returns: the current text representation of operand
```

GetOriginalByte()

```
Get original value of program byte

      ea - linear address

      returns: the original value of byte before any patch applied to it
```

GetPrevBmask()

```
Get prev bitmask in the enum (bitfield)

      Arguments:

      enum_id - id of enum
      value   - value of the current bitmask

      Returns: value of a bitmask with value lower than the specified
               value. -1 no such bitmasks exist.
               All bitmasks are sorted by their values as unsigned longs.
```

GetPrevConst()

```
Get prev constant in the enum

      Arguments:

      enum_id - id of enum
      bmask   - bitmask of the constant
                ordinary enums accept only -1 as a bitmask
      value   - value of the current constant

      Returns: value of a constant with value lower than the specified
               value. -1 no such constants exist.
               All constants are sorted by their values as unsigned longs.
```

GetPrevFixupEA()

```
Find previous address with fixup information
```

```
      ea - current address

      returns:    -1 - no more fixups otherwise returns the
                       previous address with fixup information
```

GetPrevStrucIdx()

```
Get index of previous structure type

      Arguments:      current structure index

      Returns:    -1 if no (more) structure type is defined
                       index of the presiouvs structure type.
                       See GetFirstStrucIdx() for the explanation of
                       structure indices and IDs.
```

GetStringType()

```
Get string type

      ea - linear address

      Returns one of ASCSTR_... constants
```

GetStrucComment()

```
Get structure type comment

      Arguments:
      id         - structure type ID
      repeatable - 1: get repeatable comment
                       0: get regular comment

      Returns:    null string if bad structure type ID is passed
                       otherwise returns comment.
```

GetStrucIdByName()

```
Get structure ID by structure name

      Arguments:  structure type name

      Returns:    -1 if bad structure type name is passed
                       otherwise returns structure ID.
```

GetStrucIdx()

```
Get structure index by structure ID

      Arguments:      structure ID

      Returns:    -1 if bad structure ID is passed
                     otherwise returns structure index.
                     See GetFirstStrucIdx() for the explanation of
                     structure indices and IDs.
```

GetStrucName()

```
Get structure type name

      Arguments:  structure type ID

      Returns:    -1 if bad structure type ID is passed
                     otherwise returns structure type name.
```

GetStrucQty()

```
Get number of defined structure types

      In:
      none

      returns:        number of structure types
```

GetTrueName()

```
Get true name of program byte

      This function returns name of byte as is without any replacements.

      ea   - linear address

      returns: "" - byte has no name
```

GetTrueNameEx()

```
Get true name of program byte

      This function returns name of byte as is without any replacements.

      from - the referring address.
             Allows to retrieve local label addresses in functions.
             If a local name is not found, then a global name is returned.
      ea   - linear address

      returns: "" - byte has no name
```

GetnEnum()

```
Get ID of the specified enum by its serial number

      Arguments:
      idx - number of enum (0..GetEnumQty()-1)

      Returns:        ID of enum or -1 if error
```

Heads()

```
Get a list of heads (instructions or data)

      In:

      start - start address (this one is always included)
      end   - end address


      Return:

      list of heads between start and end
```

HideArea()

```
Hide an area

      Hidden areas - address ranges which can be replaced by their descrip-
tions

      arguments:

      start,end   - area boundaries
      description - description to display if the area is collapsed
      header      - header lines to display if the area is expanded
      footer      - footer lines to display if the area is expanded
      visible     - the area state

      Returns:    !=0 - ok
```

INFMAP

```
dict() -> new empty dictionary.
dict(mapping) -> new dictionary initialized from a mapping object's
    (key, value) pairs.
dict(seq) -> new dictionary initialized as if via:
    d = {}
    for k, v in seq:
        d[k] = v
```

```
dict(**kwargs) -> new dictionary initialized with the name=value pairs
    in the keyword argument list.  For example:  dict(one=1, two=2)
```

ItemEnd()

```
Get address of the end of the item (instruction or data)

      ea - linear address

      returns: address past end of the item at 'ea'
```

ItemSize()

```
Get size of instruction or data item in bytes

      ea - linear address

      returns: 1..n
```

Jump()

```
Move cursor to the specifed linear address

      ea - linear address
```

LineA()

```
Get anterior line

      Arguments:

      ea - linear address
      num - number of anterior line (0..MAX_ITEM_LINES)
            MAX_ITEM_LINES is defined in IDA.CFG

      Returns: anterior line string
```

LineB()

```
Get posterior line

      Arguments:

      ea  - linear address
      num - number of posterior line (0..MAX_ITEM_LINES)

      Returns: posterior line string
```

LocByName()

```
Get linear address of a name

      name - name of program byte

      Returns: address of the name
                  badaddr - no such name
```

LocByNameEx()

```
Get linear address of a name

      fromaddr - the referring address. Allows to retrieve local label
                 addresses in functions. If a local name is not found,
                 then address of a global name is returned.

      name - name of program byte

      Returns: address of the name (BADADDR - no such name)
```

MK_FP()

```
Return value of expression: ((seg4) + off)
```

MakeAlign()

```
Convert the current item to an alignment directive

      ea      - linear address
      count   - number of bytes to convert
      align   - 0 or 1..32
                if it is 0, the correct alignment will be calculated
                by the kernel

      returns: 1-ok, 0-failure
```

MakeArray()

```
Create an array.

      ea      - linear address
      nitems  - size of array in items

      This function will create an array of the items with the same type as
      the type of the item at 'ea'. If the byte at 'ea' is undefined, then
      this function will create an array of bytes.
```

MakeByte()

```
Convert the current item to a byte

      ea - linear address

      returns: 1-ok, 0-failure
```

MakeCode()

```
Create an instruction at the specified address

      ea - linear address

      Returns: 0 - can not create an instruction (no such opcode, the in-
struction
      would overlap with existing items, etc) otherwise returns length of the
      instruction in bytes
```

MakeComm()

```
Set an indented regular comment of an item

      ea      - linear address
      comment - comment string
```

MakeDouble()

```
Convert the current item to a double floating point (8 bytes)

      ea - linear address

      returns: 1-ok, 0-failure
```

MakeDword()

```
Convert the current item to a double word (4 bytes)

      ea - linear address

      returns: 1-ok, 0-failure
```

MakeFloat()

```
Convert the current item to a floating point (4 bytes)

      ea - linear address
```

```
      returns: 1-ok, 0-failure
```

MakeFunction()

```
Create a function

      start,end - function bounds

      If the function end address is BADADDR, then
      IDA will try to determine the function bounds
      automatically. IDA will define all necessary
      instructions to determine the function bounds.

      returns:        !=0 - ok

      Note:           an instruction should be present at the start address
```

MakeName()

```
Rename a byte

      ea   - linear address
      name - new name of address. If name == "", then delete old name

      returns: 1-ok, 0-failure
```

MakeOword()

```
Convert the current item to a octa word (16 bytes)

      ea - linear address

      returns: 1-ok, 0-failure
```

MakePackReal()

```
Convert the current item to a packed real (10 or 12 bytes)

      ea - linear address

      returns: 1-ok, 0-failure

      FIXME: the size needs to be adjusted to IDP.hpp
```

MakeQword()

```
Convert the current item to a quadro word (8 bytes)

      ea - linear address

      returns: 1-ok, 0-failure
```

MakeRptCmt()

```
Set an indented repeatable comment of an item

      ea      - linear address
      comment - comment string
```

MakeStr()

```
Create a string.

      This function creates a string (the string type is determined by the
      value of GetLongPrm(INF_STRTYPE))

      ea    - linear address
      endea - ending address of the string (excluded)
            if endea == BADADDR, then length of string will be calculated
            by the kernel

      returns: 1-ok, 0-failure

      Note: the type of an existing string is returned by GetStringType()
```

MakeTbyte()

```
Convert the current item to a tbyte (10 or 12 bytes)

      ea - linear address

      returns: 1-ok, 0-failure

      FIXME: the size needs to be adjusted to IDP.hpp
```

MakeUnkn()

```
Convert the current item to an explored item

      ea     - linear address
      expand - 0: just undefine the current item
                  1: undefine other instructions if the removal of the
                       current instruction removes all references to them.
```

Note: functions will not be undefined even if they have no references to them

MakeVar()

Mark the location as "variable"

Arguments:

ea - address to mark

Returns: -

Note: All that IDA does is to mark the location as "variable".
Nothing else, no additional analysis is performed.
This function may disappear in the future.

MakeWord()

Convert the current item to a word (2 bytes)

ea - linear address

returns: 1-ok, 0-failure

Message()

Display a message in the messages window

msg - message to print (formatting is done in Python)

This function can be used to debug IDC scripts

Message()

Display a message in the messages window

msg - message to print (formatting is done in Python)

This function can be used to debug IDC scripts

Name()

Get visible name of program byte

This function returns name of byte as it is displayed on the screen.
If a name contains illegal characters, IDA replaces them by the substitution

```
      character during displaying. See IDA.CFG for the definition of the
      substitution character.

      ea   - linear address

      Returns: "" - byte has no name
```

NameEx()

```
Get visible name of program byte

      This function returns name of byte as it is displayed on the screen.
      If a name contains illegal characters, IDA replaces them by the
      substitution character during displaying. See IDA.CFG for the
      definition of the substitution character.

      Arguments:

      fromaddr - the referring address.
                 Allows to retrieve local label addresses in functions.
                 If a local name is not found, then a global name is
                 returned.
      ea       - linear address

      Returns: "" - byte has no name
```

NextAddr()

```
Get next addresss in the program

      ea - linear address

      returns: BADADDR - the specified address in the last used address
```

NextFunction()

```
Find next function

      ea - any address belonging to the function

      returns:        -1 - no more functions
                  otherwise returns the next function start address
```

NextHead()

```
Get next defined item (instruction or data) in the program

      ea    - linear address to start search from
      maxea - the search will stop at the address
```

```
            maxea is not included in the search range

      returns: BADADDR - no (more) defined items
```

NextNotTail()

```
Get next not-tail address in the program
      This function searches for the next displayable address in the program.
      The tail bytes of instructions and data are not displayable.

      ea - linear address

      returns: BADADDR - no (more) not-tail addresses
```

NextSeg()

```
Get next segment

      ea - linear address

      returns: start of the next segment
            BADADDR - no next segment
```

OpAlt()

```
Specify operand represenation manually.
      (for the explanations of 'ea' and 'n' please see OpBinary())

      str - a string represenation of the operand

      Note:

      IDA will not check the specified operand, it will simply display
      it instead of the orginal representation of the operand.
```

OpBinary()

```
Convert an operand of the item (instruction or data) to a binary number

      ea - linear address

      n  - number of operand
             0 - the first operand
             1 - the second, third and all other operands
            -1 - all operands

      Returns: 1-ok, 0-failure

      Note: the data items use only the type of the first operand
```

OpChr()

```
See explanation to Opbinary functions.
```

OpDecimal()

```
See explanation to Opbinary functions.
```

OpEnumEx()

```
Convert operand to a symbolic constant

      (for the explanations of 'ea' and 'n' please see OpBinary())

      enumid - id of enumeration type
      serial - serial number of the constant in the enumeration
               The serial numbers are used if there are more than
               one symbolic constant with the same value in the
               enumeration. In this case the first defined constant
               get the serial number 0, then second 1, etc.
               There could be 256 symbolic constants with the same
               value in the enumeration.
```

OpHex()

```
See explanation to Opbinary functions.
```

OpNot()

```
Toggle the bitwise not operator for the operand
      (for the explanations of 'ea' and 'n' please see OpBinary())
```

OpNumber()

```
Convert operand to a number (with default number base, radix)

      (for the explanations of 'ea' and 'n' please see OpBinary())
```

OpOctal()

```
Convert an operand of the item (instruction or data) to an octal number
      (see explanation to Opbinary functions)
```

OpOff()

```
Convert operand to an offset

      Arguments:

      (for the explanations of 'ea' and 'n' please see OpBinary())

    base - base of the offset as a linear address
             If base == BADADDR then the current operand becomes non-offset

      Example:
         seg000:2000 dw      1234h
         and there is a segment at paragraph 0x1000 and there is a data item
         within the segment at 0x1234:
         seg000:1234 MyString        db 'Hello, world!',0
         Then you need to specify a linear address of the segment base to
         create a proper offset:
           OpOffset(["seg000",0x2000],0,0x10000);
         and you will have:
           seg000:2000 dw      offset MyString

      Motorola 680x0 processor have a concept of "outer offsets".
      If you want to create an outer offset, you need to combine number
      of the operand with the following bit:

      Please note that the outer offsets are meaningful only for
      Motorola 680x0.
```

OpOffEx()

```
Convert operand to a complex offset expression
      This is a more powerful version of OpOff() function.
      It allows to explicitly specify the reference type (off8,off16, etc)
      and the expression target with a possible target delta.
      The complex expressions are represented by IDA in the following form:

      target + tdelta - base

      If the target is not present, then it will be calculated using

      target = operand_value - tdelta + base

      The target must be present for LOW.. and HIGH.. reference types


      Arguments:

      ea      - linear address of the instruction/data
      n       - number of operand to convert (the same as in OpOff)
      reftype - one of REF_... constants
      target  - an explicitly specified expression target. if you don't
                want to specify it, use -1. Please note that LOW... and
                HIGH... reference type requre the target.
```

```
      base    - the offset base (a linear address)
      tdelta  - a displacement from the target which will be displayed
                in the expression.

      Returns: success (boolean)
```

OpSeg()

```
Convert operand to a segment expression

      (for the explanations of 'ea' and 'n' please see OpBinary())
```

OpSign()

```
Change sign of the operand.
      (for the explanations of 'ea' and 'n' please see OpBinary())
```

OpStkvar()

```
Convert operand to a stack variable

      (for the explanations of 'ea' and 'n' please see OpBinary())
```

PatchByte()

```
Change value of a program byte

      ea    - linear address
      value - new value of the byte
```

PatchDword()

```
Change value of a double word

      ea    - linear address
      value - new value of the double word
```

PatchWord()

```
Change value of a program word (2 bytes)

      ea    - linear address
      value - new value of the word
```

PrevAddr()

```
Get previous addresss in the program

      ea - linear address

      returns: BADADDR - the specified address in the first address
```

PrevFunction()

```
Find previous function

      ea - any address belonging to the function

      returns:        -1 - no more functions
                  otherwise returns the previous function start address
```

PrevHead()

```
Get previous defined item (instruction or data) in the program

      ea    - linear address to start search from
      minea - the search will stop at the address
              minea is included in the search range

      returns: BADADDR - no (more) defined items
```

PrevNotTail()

```
Get previous not-tail address in the program
      This function searches for the previous displayable address in the pro-
gram.
      The tail bytes of instructions and data are not displayable.

      ea - linear address

      returns: BADADDR - no (more) not-tail addresses
```

RenameEntryPoint()

```
Rename entry point

      ordinal - entry point number
      name    - new name

      returns:        !=0 - ok
```

Rfirst()

```
Get first xref from 'From'
```

Rfirst0()

```
Get first xref from 'From'
```

RfirstB()

```
Get first xref to 'To'
```

RfirstB0()

```
Get first xref to 'To'
```

Rnext()

```
Get next xref from
```

Rnext0()

```
Get next xref from
```

RnextB()

```
Get next xref to 'To'
```

RnextB0()

```
Get next xref to 'To'
```

RptCmt()

```
Get repeatable indented comment

      Arguments:
      ea - linear address

      Returns: string or None if it fails
```

RunPlugin()

```
Load and run a plugin

      Arguments:
      name  - The plugin name is a short plugin name without an extension
```

```
      arg   - integer argument

      Returns: 0 if could not load the plugin, 1 if ok
```

ScreenEA()

```
Get linear address of cursor
```

SegAddrng()

```
Change segment addressing

      Arguments:

      ea      - any address in the segment
      use32   - 0: 16bit, 1: 32bit, 2: 64bit

      Returns: success (boolean)
```

SegAlign()

```
Change alignment of the segment

      Arguments:

      ea      - any address in the segment
      align   - new alignment of the segment

      Returns: success (boolean)
```

SegBounds()

```
Change segment boundaries

      Arguments:

      ea      - any address in the segment
      startea - new start address of the segment
      endea   - new end address of the segment
      disable - discard bytes that go out of the segment

      Returns: boolean success
```

SegByName()

```
Get segment by name

      segname - name of segment
```

```
      Returns: segment base address or BADADDR
```

SegClass()

```
Change class of the segment

      Arguments:

      ea      - any address in the segment
      class   - new class of the segment

      Returns: success (boolean)
```

SegComb()

```
Change combination of the segment

      Arguments:

      ea      - any address in the segment
      comb    - new combination of the segment

      Returns: success (boolean)
```

SegCreate()

```
Create a new segment

      Arguments:

      startea  - linear address of the start of the segment
      endea    - linear address of the end of the segment
                 this address will not belong to the segment
                 'endea' should be higher than 'startea'
      base     - base paragraph or selector of the segment.
                 a paragraph is 16byte memory chunk.
                 If a selector value is specified, the selector should be
                 already defined.
      use32    - 0: 16bit segment, 1: 32bit segment, 2: 64bit segment
      align    - segment alignment. see below for alignment values
      comb     - segment combination. see below for combination values.

      Returns: 0-failed, 1-ok
```

SegDelete()

```
Delete a segment
```

```
      Arguments:

      ea      - any address in the segment
      disable - 1: discard all bytes of the segment from the disassembled
text
                        0: retain byte values

      Returns: boolean success
```

SegEnd()

```
Get end address of a segment

      ea - any address in the segment

      returns: end of segment (an address past end of the segment)
            BADADDR - the specified address doesn't belong to any segment
```

SegName()

```
Get name of a segment

      ea - any address in the segment

      returns: "" - no segment at the specified address
```

SegRename()

```
Change name of the segment

      Arguments:

      ea      - any address in the segment
      name    - new name of the segment

      Returns: success (boolean)
```

SegStart()

```
Get start address of a segment

      ea - any address in the segment

      returns: start of segment
            BADADDR - the specified address doesn't belong to any segment
```

Segments()

```
Get list of segments (sections) in the binary image

      In: -

      Return:

      List of segment start addresses.
```

SelEnd()

```
Get end address of the selected area

      returns BADADDR - the user has not selected an area
```

SelStart()

```
Get start address of the selected area
      returns BADADDR - the user has not selected an area
```

SetBmaskCmt()

```
set bitmask comment (only for bitfields)

      Arguments:

      enum_id - id of enum
      bmask   - bitmask of the constant
      cmt     - comment
      repeatable - type of comment, 0-regular, 1-repeatable

      Returns: 1-ok, 0-failed
```

SetBmaskName()

```
Set bitmask name (only for bitfields)

      Arguments:

      enum_id - id of enum
      bmask   - bitmask of the constant
      name    - name of bitmask

      Returns: 1-ok, 0-failed
```

SetFixup()

```
Set fixup information
```

```
      ea        - address to set fixup information about
      type      - fixup type. see GetFixupTgtType()
                        for possible fixup types.
      targetsel - target selector
      targetoff - target offset
      displ     - displacement

      Returns:        none
```

SetFunctionCmt()

```
Set function comment

      ea - any address belonging to the function
      cmt - a function comment line
      repeatable - 1: get repeatable comment
                   0: get regular comment
```

SetFunctionEnd()

```
Change function end address

      ea - any address belonging to the function
      end - new function end address

      returns:        !=0 - ok
```

SetFunctionFlags()

```
Change function flags

      ea    - any address belonging to the function
      flags - see GetFunctionFlags() for explanations

      returns:        !=0 - ok
```

SetHiddenArea()

```
Set hidden area state

      Arguments:

      ea      - any address belonging to the hidden area
      visible - new state of the area

      Returns: !=0 - ok
```

SetManualInsn()

```
Specify instruction represenation manually.

      ea   - linear address
      insn - a string represenation of the operand

      IDA will not check the specified instruction, it will simply display
      it instead of the orginal representation.
```

SetSegmentType()

```
Set segment type

      Arguments:

      segea - any address within segment
      type  - new segment type:

      Returns:        !=0 - ok
```

SetSelector()

```
Set a selector value

      sel - 16bit selector number (should be less than 0xFFFF)
      val - value of selector

      returns:        nothing

      note:           ida supports up to 4096 selectors.
                  if 'sel' == 'val' then the selector is destroyed because
                  it has no significance
```

SetStatus()

```
Change IDA indicator.

      Arguments:

      status - new status

      Returns: the previous status.
```

Warning()

```
Display a message in a message box

      msg - message to print (formatting is done in Python)

      This function can be used to debug IDC scripts
```

```
      The user will be able to hide messages if they appear twice in a row on
      the screen
```

Word()

```
Get value of program word (2 bytes)

      ea - linear address

      returns: the value of the word. If word has no value then returns
0xFFFF
            If the current byte size is different from 8 bits, then the re-
turned value
            might have more 1's.
```

add_dref()

```
Create Data Ref
```

add_dref()

```
Create Data Ref
```

del_dref()

```
Unmark Data Ref
```

del_dref()

```
Unmark Data Ref
```

refs()

```
Generic reference collector.

      Note:

      This function is for internal use only.
```

Resources

[2005a] Carrera, Ero. pyreml. http://dkbza.org/pyreml.html (2005).

[2005] Carrera, Ero. idb2reml. http://dkbza.org/idb2reml.html (2005).

[2005a] Carrera, Ero. pydot. http://dkbza.org/pydot.html (2005).

[2004] Carrera, Ero and Erdélyi, Gergely. “Digital Genome Mapping - Advanced Binary Malware Analysis” Virus Bulletin Proceedings (2004) 187-197.

[2005] Datarescue. IDA. http://www.datarescue.com/idabase/ (2005).

[2005] Ellson, John and Gansner, Emden and Koren, Yehuda and Koutsofios, Eleftherios and Mocenigo, John and North, Stephen and Woodhull, Gordon. Graphviz. http://www.graphviz.org/ (2005).

[2005] Erdélyi, Gergely. IDAPython. http://d-dome.net/idapython (2005).