IDAPython Modules Reference

This document contains the reference for the IDAPython modules. It is generated from the IDAPython source code mainly.

It contains the following sections:

- IDC compatibility module
- idautils
- idaapi

IDC compatilibity module

This module contains functions that are compatible with the IDC language. For all that matters, we can use this module for most of the stuff we need to achieve in IDAPython.

For example, to read a byte, simply:

```
import idc

b = idc.byte(0x401000)
```

Here is the whole `idc.py` file for reference:

```
"""
IDC compatibility module

This file contains IDA built-in function declarations and internal bit
definitions.  Each byte of the program has 32-bit flags (low 8 bits keep
the byte value). These 32 bits are used in get_full_flags/get_flags functions.

This file is subject to change without any notice.
Future versions of IDA may use other definitions.
"""
from __future__ import print_function
# FIXME: Perhaps those should be loaded on-demand
import ida_idaapi
import ida_auto
import ida_dbg
import ida_diskio
import ida_entry
import ida_enum
import ida_expr
import ida_fixup
import ida_frame
import ida_funcs
import ida_gdl
import ida_ida
```

```
import ida_idc
import ida_bytes
import ida_idd
import ida_idp
import ida_kernwin
import ida_lines
import ida_loader
import ida_moves
import ida_nalt
import ida_name
import ida_netnode
import ida_offset
import ida_pro
import ida_search
import ida_segment
import ida_segregs
import ida_struct
import ida_typeinf
import ida_ua
import ida_xref

import _ida_idaapi

import os
import re
import struct
import time
import types
import sys

__EA64__ = ida_idaapi.BADADDR == 0xFFFFFFFFFFFFFFFF
WORDMASK = 0xFFFFFFFFFFFFFFFF if __EA64__ else 0xFFFFFFFF # just there for bw-compat purpos
class DeprecatedIDCError(Exception):
    """
    Exception for deprecated function calls
    """
    pass

__warned_deprecated_proto_confusion = {}
def __warn_once_deprecated_proto_confusion(what, alternative):
    if what not in __warned_deprecated_proto_confusion:
        print("NOTE: idc.%s is deprecated due to signature confusion with %s. Please use %s
            what,
            alternative,
            alternative))
        __warned_deprecated_proto_confusion[what] = True
```

```
def _IDC_GetAttr(obj, attrmap, attroffs):
    """
    Internal function to generically get object attributes
    Do not use unless you know what you are doing
    """
    if attroffs in attrmap and hasattr(obj, attrmap[attroffs][1]):
        return getattr(obj, attrmap[attroffs][1])
    else:
        errormsg = "attribute with offset %d not found, check the offset and report the prob
        raise KeyError(errormsg)


def _IDC_SetAttr(obj, attrmap, attroffs, value):
    """
    Internal function to generically set object attributes
    Do not use unless you know what you are doing
    """
    # check for read-only atributes
    if attroffs in attrmap:
        if attrmap[attroffs][0]:
            raise KeyError("attribute with offset %d is read-only" % attroffs)
        elif hasattr(obj, attrmap[attroffs][1]):
            return setattr(obj, attrmap[attroffs][1], value)
    errormsg = "attribute with offset %d not found, check the offset and report the problem'
    raise KeyError(errormsg)


BADADDR         = ida_idaapi.BADADDR # Not allowed address value
BADSEL          = ida_idaapi.BADSEL  # Not allowed selector value/number
SIZE_MAX        = _ida_idaapi.SIZE_MAX
ida_ida.__set_module_dynattrs(
    __name__,
    {
        "MAXADDR" : (lambda: ida_ida.inf_get_privrange_start_ea(), None),
    })

#
#      Flag bit definitions (for get_full_flags())
#
MS_VAL  = ida_bytes.MS_VAL              # Mask for byte value
FF_IVL  = ida_bytes.FF_IVL              # Byte has value ?

# Do flags contain byte value? (i.e. has the byte a value?)
# if not, the byte is uninitialized.
```

```
def has_value(F):     return ((F & FF_IVL) != 0)     # any defined value?

def byte_value(F):
    """
    Get byte value from flags
    Get value of byte provided that the byte is initialized.
    This macro works ok only for 8-bit byte machines.
    """
    return (F & MS_VAL)


def is_loaded(ea):
    """Is the byte initialized?"""
    return has_value(get_full_flags(ea))  # any defined value?

MS_CLS   = ida_bytes.MS_CLS   # Mask for typing
FF_CODE  = ida_bytes.FF_CODE  # Code ?
FF_DATA  = ida_bytes.FF_DATA  # Data ?
FF_TAIL  = ida_bytes.FF_TAIL  # Tail ?
FF_UNK   = ida_bytes.FF_UNK   # Unknown ?

def is_code(F):       return ((F & MS_CLS) == FF_CODE) # is code byte?
def is_data(F):       return ((F & MS_CLS) == FF_DATA) # is data byte?
def is_tail(F):       return ((F & MS_CLS) == FF_TAIL) # is tail byte?
def is_unknown(F):    return ((F & MS_CLS) == FF_UNK)  # is unexplored byte?
def is_head(F):       return ((F & FF_DATA) != 0)      # is start of code/data?

#
#      Common bits
#
MS_COMM  = ida_bytes.MS_COMM  # Mask of common bits
FF_COMM  = ida_bytes.FF_COMM  # Has comment?
FF_REF   = ida_bytes.FF_REF   # has references?
FF_LINE  = ida_bytes.FF_LINE  # Has next or prev cmt lines ?
FF_NAME  = ida_bytes.FF_NAME  # Has user-defined name ?
FF_LABL  = ida_bytes.FF_LABL  # Has dummy name?
FF_FLOW  = ida_bytes.FF_FLOW  # Exec flow from prev instruction?
FF_ANYNAME = FF_LABL | FF_NAME

def is_flow(F):       return ((F & FF_FLOW) != 0)
def isExtra(F):      return ((F & FF_LINE) != 0)
def isRef(F):        return ((F & FF_REF)  != 0)
def hasName(F):      return ((F & FF_NAME) != 0)
def hasUserName(F):  return ((F & FF_ANYNAME) == FF_NAME)

MS_0TYPE  = ida_bytes.MS_0TYPE  # Mask for 1st arg typing
```

```
FF_0VOID  = ida_bytes.FF_0VOID  # Void (unknown)?
FF_0NUMH  = ida_bytes.FF_0NUMH  # Hexadecimal number?
FF_0NUMD  = ida_bytes.FF_0NUMD  # Decimal number?
FF_0CHAR  = ida_bytes.FF_0CHAR  # Char ('x')?
FF_0SEG   = ida_bytes.FF_0SEG   # Segment?
FF_0OFF   = ida_bytes.FF_0OFF   # Offset?
FF_0NUMB  = ida_bytes.FF_0NUMB  # Binary number?
FF_0NUMO  = ida_bytes.FF_0NUMO  # Octal number?
FF_0ENUM  = ida_bytes.FF_0ENUM  # Enumeration?
FF_0FOP   = ida_bytes.FF_0FOP   # Forced operand?
FF_0STRO  = ida_bytes.FF_0STRO  # Struct offset?
FF_0STK   = ida_bytes.FF_0STK   # Stack variable?

MS_1TYPE  = ida_bytes.MS_1TYPE  # Mask for 2nd arg typing
FF_1VOID  = ida_bytes.FF_1VOID  # Void (unknown)?
FF_1NUMH  = ida_bytes.FF_1NUMH  # Hexadecimal number?
FF_1NUMD  = ida_bytes.FF_1NUMD  # Decimal number?
FF_1CHAR  = ida_bytes.FF_1CHAR  # Char ('x')?
FF_1SEG   = ida_bytes.FF_1SEG   # Segment?
FF_1OFF   = ida_bytes.FF_1OFF   # Offset?
FF_1NUMB  = ida_bytes.FF_1NUMB  # Binary number?
FF_1NUMO  = ida_bytes.FF_1NUMO  # Octal number?
FF_1ENUM  = ida_bytes.FF_1ENUM  # Enumeration?
FF_1FOP   = ida_bytes.FF_1FOP   # Forced operand?
FF_1STRO  = ida_bytes.FF_1STRO  # Struct offset?
FF_1STK   = ida_bytes.FF_1STK   # Stack variable?

# The following macros answer questions like
#   'is the 1st (or 2nd) operand of instruction or data of the given type'?
# Please note that data items use only the 1st operand type (is...0)

def is_defarg0(F):    return ((F & MS_0TYPE) != FF_0VOID)
def is_defarg1(F):    return ((F & MS_1TYPE) != FF_1VOID)
def isDec0(F):       return ((F & MS_0TYPE) == FF_0NUMD)
def isDec1(F):       return ((F & MS_1TYPE) == FF_1NUMD)
def isHex0(F):       return ((F & MS_0TYPE) == FF_0NUMH)
def isHex1(F):       return ((F & MS_1TYPE) == FF_1NUMH)
def isOct0(F):       return ((F & MS_0TYPE) == FF_0NUMO)
def isOct1(F):       return ((F & MS_1TYPE) == FF_1NUMO)
def isBin0(F):       return ((F & MS_0TYPE) == FF_0NUMB)
def isBin1(F):       return ((F & MS_1TYPE) == FF_1NUMB)
def is_off0(F):       return ((F & MS_0TYPE) == FF_0OFF)
def is_off1(F):       return ((F & MS_1TYPE) == FF_1OFF)
def is_char0(F):      return ((F & MS_0TYPE) == FF_0CHAR)
def is_char1(F):      return ((F & MS_1TYPE) == FF_1CHAR)
def is_seg0(F):       return ((F & MS_0TYPE) == FF_0SEG)
```

```
def is_seg1(F):       return ((F & MS_1TYPE) == FF_1SEG)
def is_enum0(F):      return ((F & MS_0TYPE) == FF_0ENUM)
def is_enum1(F):      return ((F & MS_1TYPE) == FF_1ENUM)
def is_manual0(F):       return ((F & MS_0TYPE) == FF_0FOP)
def is_manual1(F):       return ((F & MS_1TYPE) == FF_1FOP)
def is_stroff0(F):    return ((F & MS_0TYPE) == FF_0STRO)
def is_stroff1(F):    return ((F & MS_1TYPE) == FF_1STRO)
def is_stkvar0(F):    return ((F & MS_0TYPE) == FF_0STK)
def is_stkvar1(F):    return ((F & MS_1TYPE) == FF_1STK)

#
#      Bits for DATA bytes
#
DT_TYPE  = ida_bytes.DT_TYPE & 0xFFFFFFFF  # Mask for DATA typing

FF_BYTE      = ida_bytes.FF_BYTE & 0xFFFFFFFF      # byte
FF_WORD      = ida_bytes.FF_WORD & 0xFFFFFFFF      # word
FF_DWORD      = ida_bytes.FF_DWORD & 0xFFFFFFFF      # dword
FF_QWORD      = ida_bytes.FF_QWORD & 0xFFFFFFFF      # qword
FF_TBYTE      = ida_bytes.FF_TBYTE & 0xFFFFFFFF      # tbyte
FF_STRLIT      = ida_bytes.FF_STRLIT & 0xFFFFFFFF      # ASCII ?
FF_STRUCT      = ida_bytes.FF_STRUCT & 0xFFFFFFFF      # Struct ?
FF_OWORD      = ida_bytes.FF_OWORD & 0xFFFFFFFF      # octaword (16 bytes)
FF_FLOAT     = ida_bytes.FF_FLOAT & 0xFFFFFFFF     # float
FF_DOUBLE    = ida_bytes.FF_DOUBLE & 0xFFFFFFFF    # double
FF_PACKREAL  = ida_bytes.FF_PACKREAL & 0xFFFFFFFF  # packed decimal real
FF_ALIGN     = ida_bytes.FF_ALIGN & 0xFFFFFFFF     # alignment directive

def is_byte(F):     return (is_data(F) and (F & DT_TYPE) == FF_BYTE)
def is_word(F):     return (is_data(F) and (F & DT_TYPE) == FF_WORD)
def is_dword(F):     return (is_data(F) and (F & DT_TYPE) == FF_DWORD)
def is_qword(F):     return (is_data(F) and (F & DT_TYPE) == FF_QWORD)
def is_oword(F):     return (is_data(F) and (F & DT_TYPE) == FF_OWORD)
def is_tbyte(F):     return (is_data(F) and (F & DT_TYPE) == FF_TBYTE)
def is_float(F):    return (is_data(F) and (F & DT_TYPE) == FF_FLOAT)
def is_double(F):   return (is_data(F) and (F & DT_TYPE) == FF_DOUBLE)
def is_pack_real(F): return (is_data(F) and (F & DT_TYPE) == FF_PACKREAL)
def is_strlit(F):    return (is_data(F) and (F & DT_TYPE) == FF_STRLIT)
def is_struct(F):   return (is_data(F) and (F & DT_TYPE) == FF_STRUCT)
def is_align(F):    return (is_data(F) and (F & DT_TYPE) == FF_ALIGN)

#
#      Bits for CODE bytes
#
MS_CODE  = ida_bytes.MS_CODE & 0xFFFFFFFF
FF_FUNC  = ida_bytes.FF_FUNC & 0xFFFFFFFF  # function start?
```

```
FF_IMMD  = ida_bytes.FF_IMMD & 0xFFFFFFFF  # Has Immediate value ?
FF_JUMP  = ida_bytes.FF_JUMP & 0xFFFFFFFF  # Has jump table

#
#      Loader flags
#
if ida_idaapi.uses_swig_builtins:
    _scope = ida_loader.loader_t
else:
    _scope = ida_loader
NEF_SEGS   = _scope.NEF_SEGS   # Create segments
NEF_RSCS   = _scope.NEF_RSCS   # Load resources
NEF_NAME   = _scope.NEF_NAME   # Rename entries
NEF_MAN    = _scope.NEF_MAN    # Manual load
NEF_FILL   = _scope.NEF_FILL   # Fill segment gaps
NEF_IMPS   = _scope.NEF_IMPS   # Create imports section
NEF_FIRST  = _scope.NEF_FIRST  # This is the first file loaded
NEF_CODE   = _scope.NEF_CODE   # for load_binary_file:
NEF_RELOAD = _scope.NEF_RELOAD # reload the file at the same place:
NEF_FLAT   = _scope.NEF_FLAT   # Autocreated FLAT group (PE)

#         List of built-in functions
#         --------------------------
#
# The following conventions are used in this list:
#   'ea' is a linear address
#   'success' is 0 if a function failed, 1 otherwise
#   'void' means that function returns no meaningful value (always 0)
#
#  All function parameter conversions are made automatically.
#
# ------------------------------------------------------------------------
#                      M I S C E L L A N E O U S
# ------------------------------------------------------------------------
def value_is_string(var): raise NotImplementedError("this function is not needed in Python")
def value_is_long(var):   raise NotImplementedError("this function is not needed in Python")
def value_is_float(var):  raise NotImplementedError("this function is not needed in Python")
def value_is_func(var):   raise NotImplementedError("this function is not needed in Python")
def value_is_pvoid(var):  raise NotImplementedError("this function is not needed in Python")
def value_is_int64(var):  raise NotImplementedError("this function is not needed in Python")

def to_ea(seg, off):
    """
    Return value of expression: ((seg<<4) + off)
    """
    return (seg << 4) + off
```

```
def form(format, *args):
    raise DeprecatedIDCError("form() is deprecated. Use python string operations instead.")

def substr(s, x1, x2):
    raise DeprecatedIDCError("substr() is deprecated. Use python string operations instead.'

def strstr(s1, s2):
    raise DeprecatedIDCError("strstr() is deprecated. Use python string operations instead.'

def strlen(s):
    raise DeprecatedIDCError("strlen() is deprecated. Use python string operations instead.'

def xtol(s):
    raise DeprecatedIDCError("xtol() is deprecated. Use python long() instead.")

def atoa(ea):
    """
    Convert address value to a string
    Return address in the form 'seg000:1234'
    (the same as in line prefixes)

    @param ea: address to format
    """
    return ida_kernwin.ea2str(ea)

def ltoa(n, radix):
    raise DeprecatedIDCError("ltoa() is deprecated. Use python string operations instead.")

def atol(s):
    raise DeprecatedIDCError("atol() is deprecated. Use python long() instead.")


def rotate_left(value, count, nbits, offset):
    """
    Rotate a value to the left (or right)

    @param value: value to rotate
    @param count: number of times to rotate. negative counter means
                  rotate to the right
    @param nbits: number of bits to rotate
    @param offset: offset of the first bit to rotate

    @return: the value with the specified field rotated
             all other bits are not modified
    """
```

```
    assert offset >= 0, "offset must be >= 0"
    assert nbits > 0, "nbits must be > 0"
    count %= nbits # no need to spin the wheel more than 1 rotation

    mask = 2**(offset+nbits) - 2**offset
    tmp = value & mask

    if count > 0:
        for x in range(count):
            if (tmp >> (offset+nbits-1)) & 1:
                tmp = (tmp << 1) | (1 << offset)
            else:
                tmp = (tmp << 1)
    else:
        for x in range(-count):
            if (tmp >> offset) & 1:
                tmp = (tmp >> 1) | (1 << (offset+nbits-1))
            else:
                tmp = (tmp >> 1)

    value = (value-(value&mask)) | (tmp & mask)

    return value


def rotate_dword(x, count): return rotate_left(x, count, 32, 0)
def rotate_word(x, count):  return rotate_left(x, count, 16, 0)
def rotate_byte(x, count):  return rotate_left(x, count, 8, 0)


# add_idc_hotkey return codes
IDCHK_OK        =  0    # ok
IDCHK_ARG       = -1    # bad argument(s)
IDCHK_KEY       = -2    # bad hotkey name
IDCHK_MAX       = -3    # too many IDC hotkeys

add_idc_hotkey = ida_kernwin.add_idc_hotkey
del_idc_hotkey = ida_kernwin.del_idc_hotkey
jumpto = ida_kernwin.jumpto
auto_wait = ida_auto.auto_wait


def eval_idc(expr):
    """
    Evaluate an IDC expression
```

```
    @param expr: an expression

    @return: the expression value. If there are problems, the returned value will be "IDC_FA
             where xxx is the error description

    @note: Python implementation evaluates IDC only, while IDC can call other registered lan
    """
    rv = ida_expr.idc_value_t()

    err = ida_expr.eval_idc_expr(rv, BADADDR, expr)
    if err:
        return "IDC_FAILURE: "+err
    else:
        if rv.vtype == '\x02': # long
            return rv.num
        elif rv.vtype == '\x07': # VT_STR
            return rv.c_str()
        else:
            raise NotImplementedError("eval_idc() supports only expressions returning string


def EVAL_FAILURE(code):
    """
    Check the result of eval_idc() for evaluation failures

    @param code: result of eval_idc()

    @return: True if there was an evaluation error
    """
    return type(code) == bytes and code.startswith("IDC_FAILURE: ")


def save_database(idbname, flags=0):
    """
    Save current database to the specified idb file

    @param idbname: name of the idb file. if empty, the current idb
                    file will be used.
    @param flags: combination of ida_loader.DBFL_... bits or 0
    """
    if len(idbname) == 0:
        idbname = get_idb_path()
    mask = ida_loader.DBFL_KILL | ida_loader.DBFL_COMP | ida_loader.DBFL_BAK
    return ida_loader.save_database(idbname, flags & mask)

DBFL_BAK = ida_loader.DBFL_BAK # for compatiblity with older versions, eventually delete th
```

```
def validate_idb_names(do_repair = 0):
    """
    check consistency of IDB name records
    @param do_repair: try to repair netnode header it TRUE
    @return: number of inconsistent name records
    """
    return ida_nalt.validate_idb_names(do_repair)

qexit = ida_pro.qexit


def call_system(command):
    """
    Execute an OS command.

    @param command: command line to execute

    @return: 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.
    """
    return os.system(command)


def qsleep(milliseconds):
    """
    qsleep the specified number of milliseconds
    This function suspends IDA for the specified amount of time

    @param milliseconds: time to sleep
    """
    time.sleep(float(milliseconds)/1000)


load_and_run_plugin = ida_loader.load_and_run_plugin
plan_to_apply_idasgn = ida_funcs.plan_to_apply_idasgn


#--------------------------------------------------------------------------
#      C H A N G E   P R O G R A M   R E P R E S E N T A T I O N
#--------------------------------------------------------------------------
```

```
def delete_all_segments():
    """
    Delete all segments, instructions, comments, i.e. everything
    except values of bytes.
    """
    ea = ida_ida.cvar.inf.min_ea

    # Brute-force nuke all info from all the heads
    while ea != BADADDR and ea <= ida_ida.cvar.inf.max_ea:
        ida_name.del_local_name(ea)
        ida_name.del_global_name(ea)
        func = ida_funcs.get_func(ea)
        if func:
            ida_funcs.set_func_cmt(func, "", False)
            ida_funcs.set_func_cmt(func, "", True)
            ida_funcs.del_func(ea)
        ida_bytes.del_hidden_range(ea)
        seg = ida_segment.getseg(ea)
        if seg:
            ida_segment.set_segment_cmt(seg, "", False)
            ida_segment.set_segment_cmt(seg, "", True)
            ida_segment.del_segm(ea, ida_segment.SEGMOD_KEEP | ida_segment.SEGMOD_SILENT)

        ea = ida_bytes.next_head(ea, ida_ida.cvar.inf.max_ea)


create_insn = ida_ua.create_insn


def plan_and_wait(sEA, eEA, final_pass=True):
    """
    Perform full analysis of the range

    @param sEA: starting linear address
    @param eEA: ending linear address (excluded)
    @param final_pass: make the final pass over the specified range

    @return: 1-ok, 0-Ctrl-Break was pressed.
    """
    return ida_auto.plan_and_wait(sEA, eEA, final_pass)


def set_name(ea, name, flags=ida_name.SN_CHECK):
    """
```

```
    Rename an address

    @param ea: linear address
    @param name: new name of address. If name == "", then delete old name
    @param flags: combination of SN_... constants

    @return: 1-ok, 0-failure
    """
    return ida_name.set_name(ea, name, flags)

SN_CHECK      = ida_name.SN_CHECK    # Fail if the name contains invalid characters.
SN_NOCHECK    = ida_name.SN_NOCHECK  # Don't fail if the name contains invalid characters.
                                     # If this bit is set, all invalid chars
                                     # (not in NameChars or MangleChars) will be replaced
                                     # by '_'.
                                     # List of valid characters is defined in ida.cfg
SN_PUBLIC     = ida_name.SN_PUBLIC   # if set, make name public
SN_NON_PUBLIC = ida_name.SN_NON_PUBLIC # if set, make name non-public
SN_WEAK       = ida_name.SN_WEAK     # if set, make name weak
SN_NON_WEAK   = ida_name.SN_NON_WEAK # if set, make name non-weak
SN_AUTO       = ida_name.SN_AUTO     # if set, make name autogenerated
SN_NON_AUTO   = ida_name.SN_NON_AUTO # if set, make name non-autogenerated
SN_NOLIST     = ida_name.SN_NOLIST   # if set, exclude name from the list
                                     # if not set, then include the name into
                                     # the list (however, if other bits are set,
                                     # the name might be immediately excluded
                                     # from the list)
SN_NOWARN     = ida_name.SN_NOWARN   # don't display a warning if failed
SN_LOCAL      = ida_name.SN_LOCAL    # create local name. a function should exist.
                                     # local names can't be public or weak.
                                     # also they are not included into the list
                                     # of names they can't have dummy prefixes

set_cmt = ida_bytes.set_cmt


def make_array(ea, nitems):
    """
    Create an array.

    @param ea: linear address
    @param nitems: size of array in items

    @note: 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.
```

```
    """
    flags = ida_bytes.get_flags(ea)

    if ida_bytes.is_code(flags) or ida_bytes.is_tail(flags) or ida_bytes.is_align(flags):
        return False

    if ida_bytes.is_unknown(flags):
        flags = ida_bytes.FF_BYTE

    if ida_bytes.is_struct(flags):
        ti = ida_nalt.opinfo_t()
        assert ida_bytes.get_opinfo(ti, ea, 0, flags), "get_opinfo() failed"
        itemsize = ida_bytes.get_data_elsize(ea, flags, ti)
        tid = ti.tid
    else:
        itemsize = ida_bytes.get_item_size(ea)
        tid = BADADDR

    return ida_bytes.create_data(ea, flags, itemsize*nitems, tid)


def create_strlit(ea, endea):
    """
    Create a string.

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

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

    @return: 1-ok, 0-failure

    @note: The type of an existing string is returned by get_str_type()
    """
    return ida_bytes.create_strlit(ea, 0 if endea == BADADDR else endea - ea, get_inf_attr(I


create_data = ida_bytes.create_data


def create_byte(ea):
    """
    Convert the current item to a byte
```

```
    @param ea: linear address

    @return: 1-ok, 0-failure
    """
    return ida_bytes.create_byte(ea, 1)


def create_word(ea):
    """
    Convert the current item to a word (2 bytes)

    @param ea: linear address

    @return: 1-ok, 0-failure
    """
    return ida_bytes.create_word(ea, 2)


def create_dword(ea):
    """
    Convert the current item to a double word (4 bytes)

    @param ea: linear address

    @return: 1-ok, 0-failure
    """
    return ida_bytes.create_dword(ea, 4)


def create_qword(ea):
    """
    Convert the current item to a quadro word (8 bytes)

    @param ea: linear address

    @return: 1-ok, 0-failure
    """
    return ida_bytes.create_qword(ea, 8)


def create_oword(ea):
    """
    Convert the current item to an octa word (16 bytes/128 bits)

    @param ea: linear address
```

```
    @return: 1-ok, 0-failure
    """
    return ida_bytes.create_oword(ea, 16)


def create_yword(ea):
    """
    Convert the current item to a ymm word (32 bytes/256 bits)

    @param ea: linear address

    @return: 1-ok, 0-failure
    """
    return ida_bytes.create_yword(ea, 32)


def create_float(ea):
    """
    Convert the current item to a floating point (4 bytes)

    @param ea: linear address

    @return: 1-ok, 0-failure
    """
    return ida_bytes.create_float(ea, 4)


def create_double(ea):
    """
    Convert the current item to a double floating point (8 bytes)

    @param ea: linear address

    @return: 1-ok, 0-failure
    """
    return ida_bytes.create_double(ea, 8)


def create_pack_real(ea):
    """
    Convert the current item to a packed real (10 or 12 bytes)

    @param ea: linear address

    @return: 1-ok, 0-failure
```

```
    """
    return ida_bytes.create_packed_real(ea, ida_idp.ph_get_tbyte_size())


def create_tbyte(ea):
    """
    Convert the current item to a tbyte (10 or 12 bytes)

    @param ea: linear address

    @return: 1-ok, 0-failure
    """
    return ida_bytes.create_tbyte(ea, ida_idp.ph_get_tbyte_size())


def create_struct(ea, size, strname):
    """
    Convert the current item to a structure instance

    @param ea: linear address
    @param size: structure size in bytes. -1 means that the size
        will be calculated automatically
    @param strname: name of a structure type

    @return: 1-ok, 0-failure
    """
    strid = ida_struct.get_struc_id(strname)

    if size == -1:
        size = ida_struct.get_struc_size(strid)

    return ida_bytes.create_struct(ea, size, strid)


create_custom_data = ida_bytes.create_custdata
create_align = ida_bytes.create_align


def define_local_var(start, end, location, name):
    """
    Create a local variable

    @param start: start of address range for the local variable
    @param end: end of address range for the local variable
    @param location: the variable location in the "[bp+xx]" form where xx is
                     a number. The location can also be specified as a
```

```
                    register name.
    @param name: name of the local variable

    @return: 1-ok, 0-failure

    @note: For the stack variables the end address is ignored.
           If there is no function at 'start' then this function.
           will fail.
    """
    func = ida_funcs.get_func(start)

    if not func:
        return 0

    # Find out if location is in the [bp+xx] form
    r = re.compile("\[([a-z]+)([-+][0-9a-fx]+)", re.IGNORECASE)
    m = r.match(location)

    if m:
        # Location in the form of [bp+xx]
        register = ida_idp.str2reg(m.group(1))
        if register == -1:
            return 0

        offset = int(m.group(2), 0)
        return 1 if ida_frame.define_stkvar(func, name, offset, ida_bytes.byte_flag(), None,
    else:
        # Location as simple register name
        return ida_frame.add_regvar(func, start, end, location, name, None)


del_items = ida_bytes.del_items

DELIT_SIMPLE   = ida_bytes.DELIT_SIMPLE   # simply undefine the specified item
DELIT_EXPAND   = ida_bytes.DELIT_EXPAND   # propogate undefined items, for example
                                          # if removing an instruction removes all
                                          # references to the next instruction, then
                                          # plan to convert to unexplored the next
                                          # instruction too.
DELIT_DELNAMES = ida_bytes.DELIT_DELNAMES # delete any names at the specified address(es)


def set_array_params(ea, flags, litems, align):
    """
    Set array representation format
```

```
    @param ea: linear address
    @param flags: combination of AP_... constants or 0
    @param litems: number of items per line. 0 means auto
    @param align: element alignment
                  - -1: do not align
                  - 0:  automatic alignment
                  - other values: element width

    @return: 1-ok, 0-failure
    """
    return eval_idc("set_array_params(0x%X, 0x%X, %d, %d)"%(ea, flags, litems, align))

AP_ALLOWDUPS    = 0x00000001     # use 'dup' construct
AP_SIGNED       = 0x00000002     # treats numbers as signed
AP_INDEX        = 0x00000004     # display array element indexes as comments
AP_ARRAY        = 0x00000008     # reserved (this flag is not stored in database)
AP_IDXBASEMASK  = 0x000000F0     # mask for number base of the indexes
AP_IDXDEC       = 0x00000000     # display indexes in decimal
AP_IDXHEX       = 0x00000010     # display indexes in hex
AP_IDXOCT       = 0x00000020     # display indexes in octal
AP_IDXBIN       = 0x00000030     # display indexes in binary

op_bin = ida_bytes.op_bin
op_oct = ida_bytes.op_oct
op_dec = ida_bytes.op_dec
op_hex = ida_bytes.op_hex
op_chr = ida_bytes.op_chr


def op_plain_offset(ea, n, base):
    """
    Convert operand to an offset
    (for the explanations of 'ea' and 'n' please see op_bin())

    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:
```

```
        op_plain_offset(["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.

    @param ea: linear address
    @param n: number of operand
        - 0 - the first operand
        - 1 - the second, third and all other operands
        - -1 - all operands
    @param base: base of the offset as a linear address
        If base == BADADDR then the current operand becomes non-offset
    """
    if base == BADADDR:
        return ida_bytes.clr_op_type(ea, n)
    else:
        return ida_offset.op_plain_offset(ea, n, base)


OPND_OUTER = ida_bytes.OPND_OUTER # outer offset base


op_offset = ida_offset.op_offset


REF_OFF8    = ida_nalt.REF_OFF8    # 8bit full offset
REF_OFF16   = ida_nalt.REF_OFF16   # 16bit full offset
REF_OFF32   = ida_nalt.REF_OFF32   # 32bit full offset
REF_LOW8    = ida_nalt.REF_LOW8    # low 8bits of 16bit offset
REF_LOW16   = ida_nalt.REF_LOW16   # low 16bits of 32bit offset
REF_HIGH8   = ida_nalt.REF_HIGH8   # high 8bits of 16bit offset
REF_HIGH16  = ida_nalt.REF_HIGH16  # high 16bits of 32bit offset
REF_OFF64   = ida_nalt.REF_OFF64   # 64bit full offset
REFINFO_RVA     = 0x10 # based reference (rva)
REFINFO_PASTEND = 0x20 # reference past an item it may point to an nonexistitng
                       # do not destroy alignment dirs
REFINFO_NOBASE  = 0x80 # offset base is a number
```

```
                        # that base have be any value
                        # nb: base xrefs are created only if base
                        # points to the middle of a segment
REFINFO_SUBTRACT = 0x0100 # the reference value is subtracted from
                          # the base value instead of (as usual)
                          # being added to it
REFINFO_SIGNEDOP = 0x0200 # the operand value is sign-extended (only
                          # supported for REF_OFF8/16/32/64)

op_seg = ida_bytes.op_seg
op_num = ida_bytes.op_num
op_flt = ida_bytes.op_flt
op_man = ida_bytes.set_forced_operand
toggle_sign = ida_bytes.toggle_sign


def toggle_bnot(ea, n):
    """
    Toggle the bitwise not operator for the operand

    @param ea: linear address
    @param n: number of operand
        - 0 - the first operand
        - 1 - the second, third and all other operands
        - -1 - all operands
    """
    ida_bytes.toggle_bnot(ea, n)
    return True


op_enum = ida_bytes.op_enum


def op_stroff(ea, n, strid, delta):
    """
    Convert operand to an offset in a structure

    @param ea: linear address
    @param n: number of operand
        - 0 - the first operand
        - 1 - the second, third and all other operands
        - -1 - all operands
    @param strid: id of a structure type
    @param delta: struct offset delta. usually 0. denotes the difference
                    between the structure base and the pointer into the structure.
```

```
    """
    path = ida_pro.tid_array(1)
    path[0] = strid
    if isinstance(ea, ida_ua.insn_t):
        insn = ea
    else:
        insn = ida_ua.insn_t()
        ida_ua.decode_insn(insn, ea)
    return ida_bytes.op_stroff(insn, n, path.cast(), 1, delta)


op_stkvar = ida_bytes.op_stkvar


def op_offset_high16(ea, n, target):
    """
    Convert operand to a high offset
    High offset is the upper 16bits of an offset.
    This type is used by TMS320C6 processors (and probably by other
    RISC processors too)

    @param ea: linear address
    @param n: number of operand
        - 0 - the first operand
        - 1 - the second, third and all other operands
        - -1 - all operands
    @param target: the full value (all 32bits) of the offset
    """
    return ida_offset.op_offset(ea, n, ida_nalt.REF_HIGH16, target)


def MakeVar(ea):
    pass

# Every anterior/posterior line has its number.
# Anterior  lines have numbers from E_PREV
# Posterior lines have numbers from E_NEXT
E_PREV = ida_lines.E_PREV
E_NEXT = ida_lines.E_NEXT

get_extra_cmt = ida_lines.get_extra_cmt
update_extra_cmt = ida_lines.update_extra_cmt
del_extra_cmt = ida_lines.del_extra_cmt
set_manual_insn = ida_bytes.set_manual_insn
get_manual_insn = ida_bytes.get_manual_insn
patch_dbg_byte = ida_dbg.put_dbg_byte
```

```
patch_byte = ida_bytes.patch_byte
patch_word = ida_bytes.patch_word
patch_dword = ida_bytes.patch_dword
patch_qword = ida_bytes.patch_qword


SR_inherit      = 1 # value is inherited from the previous range
SR_user         = 2 # value is specified by the user
SR_auto         = 3 # value is determined by IDA
SR_autostart    = 4 # as SR_auto for segment starting address

def split_sreg_range(ea, reg, value, tag=SR_user):
    """
    Set value of a segment register.

    @param ea: linear address
    @param reg: name of a register, like "cs", "ds", "es", etc.
    @param value: new value of the segment register.
    @param tag: of SR_... constants

    @note: IDA keeps tracks of all the points where segment register change their
           values. This function allows you to specify the correct value of a segment
           register if IDA is not able to find the correct value.
    """
    rnames = [r.casefold() for r in ida_idp.ph_get_regnames()]
    for regno in range(ida_idp.ph_get_reg_first_sreg(), ida_idp.ph_get_reg_last_sreg()+1):
        if rnames[regno]==reg.casefold():
            return ida_segregs.split_sreg_range(ea, regno, value, tag)
    return False


auto_mark_range = ida_auto.auto_mark_range
auto_unmark = ida_auto.auto_unmark


def AutoMark(ea,qtype):
    """
    Plan to analyze an address
    """
    return auto_mark_range(ea,ea+1,qtype)

AU_UNK   = ida_auto.AU_UNK   # make unknown
AU_CODE  = ida_auto.AU_CODE  # convert to instruction
AU_PROC  = ida_auto.AU_PROC  # make function
AU_USED  = ida_auto.AU_USED  # reanalyze
AU_LIBF  = ida_auto.AU_LIBF  # apply a flirt signature (the current signature!)
```

```
AU_FINAL = ida_auto.AU_FINAL # coagulate unexplored items


#---------------------------------------------------------------------------
#              P R O D U C E   O U T P U T   F I L E S
#---------------------------------------------------------------------------

def gen_file(filetype, path, ea1, ea2, flags):
    """
    Generate an output file

    @param filetype:  type of output file. One of OFILE_... symbols. See below.
    @param path:  the output file path (will be overwritten!)
    @param ea1:   start address. For some file types this argument is ignored
    @param ea2:   end address. For some file types this argument is ignored
    @param flags: bit combination of GENFLG_...

    @returns: number of the generated lines.
              -1 if an error occurred
              OFILE_EXE: 0-can't generate exe file, 1-ok
    """
    f = ida_diskio.fopenWB(path)

    if f:
        retval = ida_loader.gen_file(filetype, f, ea1, ea2, flags)
        ida_diskio.eclose(f)
        return retval
    else:
        return -1


# output file types:
OFILE_MAP  = ida_loader.OFILE_MAP
OFILE_EXE  = ida_loader.OFILE_EXE
OFILE_IDC  = ida_loader.OFILE_IDC
OFILE_LST  = ida_loader.OFILE_LST
OFILE_ASM  = ida_loader.OFILE_ASM
OFILE_DIF  = ida_loader.OFILE_DIF

# output control flags:
GENFLG_MAPSEG  = ida_loader.GENFLG_MAPSEG  # map: generate map of segments
GENFLG_MAPNAME = ida_loader.GENFLG_MAPNAME # map: include dummy names
GENFLG_MAPDMNG = ida_loader.GENFLG_MAPDMNG # map: demangle names
GENFLG_MAPLOC  = ida_loader.GENFLG_MAPLOC  # map: include local names
GENFLG_IDCTYPE = ida_loader.GENFLG_IDCTYPE # idc: gen only information about types
GENFLG_ASMTYPE = ida_loader.GENFLG_ASMTYPE # asm&lst: gen information about types too
```

```
GENFLG_GENHTML = ida_loader.GENFLG_GENHTML # asm&lst: generate html (gui version only)
GENFLG_ASMINC  = ida_loader.GENFLG_ASMINC  # asm&lst: gen information only about types

def gen_flow_graph(outfile, title, ea1, ea2, flags):
    """
    Generate a flow chart GDL file

    @param outfile: output file name. GDL extension will be used
    @param title: graph title
    @param ea1: beginning of the range to flow chart
    @param ea2: end of the range to flow chart.
    @param flags: combination of CHART_... constants

    @note: If ea2 == BADADDR then ea1 is treated as an address within a function.
           That function will be flow charted.
    """
    return ida_gdl.gen_flow_graph(outfile, title, None, ea1, ea2, flags)


CHART_PRINT_NAMES = 0x1000 # print labels for each block?
CHART_GEN_GDL     = 0x4000 # generate .gdl file (file extension is forced to .gdl)
CHART_WINGRAPH    = 0x8000 # call wingraph32 to display the graph
CHART_NOLIBFUNCS  = 0x0400 # don't include library functions in the graph


def gen_simple_call_chart(outfile, title, flags):
    """
    Generate a function call graph GDL file

    @param outfile: output file name. GDL extension will be used
    @param title:   graph title
    @param flags:   combination of CHART_GEN_GDL, CHART_WINGRAPH, CHART_NOLIBFUNCS
    """
    return ida_gdl.gen_simple_call_chart(outfile, "Generating chart", title, flags)


#--------------------------------------------------------------------------
#                C O M M O N   I N F O R M A T I O N
#--------------------------------------------------------------------------
def idadir():
    """
    Get IDA directory

    This function returns the directory where IDA.EXE resides
    """
    return ida_diskio.idadir("")
```

```
get_root_filename = ida_nalt.get_root_filename
get_input_file_path = ida_nalt.get_input_file_path
set_root_filename = ida_nalt.set_root_filename


def get_idb_path():
    """
    Get IDB full path

    This function returns full path of the current IDB database
    """
    return ida_loader.get_path(ida_loader.PATH_TYPE_IDB)


retrieve_input_file_md5 = ida_nalt.retrieve_input_file_md5
get_full_flags = ida_bytes.get_full_flags
get_db_byte = ida_bytes.get_db_byte


def get_bytes(ea, size, use_dbg = False):
    """
    Return the specified number of bytes of the program

    @param ea: linear address

    @param size: size of buffer in normal 8-bit bytes

    @param use_dbg: if True, use debugger memory, otherwise just the database

    @return: None on failure
             otherwise a string containing the read bytes
    """
    if use_dbg:
        return ida_idd.dbg_read_memory(ea, size)
    else:
        return ida_bytes.get_bytes(ea, size)


get_wide_byte = ida_bytes.get_wide_byte


def __DbgValue(ea, len):
    if len not in ida_idaapi.__struct_unpack_table:
        return None
```

```
    r = ida_idd.dbg_read_memory(ea, len)
    return None if r is None else struct.unpack((">" if ida_ida.cvar.inf.is_be() else "<") +


def read_dbg_byte(ea):
    """
    Get value of program byte using the debugger memory

    @param ea: linear address
    @return: The value or None on failure.
    """
    return __DbgValue(ea, 1)


def read_dbg_word(ea):
    """
    Get value of program word using the debugger memory

    @param ea: linear address
    @return: The value or None on failure.
    """
    return __DbgValue(ea, 2)


def read_dbg_dword(ea):
    """
    Get value of program double-word using the debugger memory

    @param ea: linear address
    @return: The value or None on failure.
    """
    return __DbgValue(ea, 4)


def read_dbg_qword(ea):
    """
    Get value of program quadro-word using the debugger memory

    @param ea: linear address
    @return: The value or None on failure.
    """
    return __DbgValue(ea, 8)


read_dbg_memory = ida_idd.dbg_read_memory
```

```
def write_dbg_memory(ea, data):
    """
    Write to debugger memory.

    @param ea: linear address
    @param data: string to write
    @return: number of written bytes (-1 - network/debugger error)

    Thread-safe function (may be called only from the main thread and debthread)
    """
    __warn_once_deprecated_proto_confusion("write_dbg_memory", "ida_dbg.write_dbg_memory")
    if not ida_dbg.dbg_can_query():
        return -1
    elif len(data) > 0:
        return ida_idd.dbg_write_memory(ea, data)


get_original_byte = ida_bytes.get_original_byte
get_wide_word = ida_bytes.get_wide_word
get_wide_dword = ida_bytes.get_wide_dword
get_qword = ida_bytes.get_qword


def GetFloat(ea):
    """
    Get value of a floating point number (4 bytes)
    This function assumes number stored using IEEE format
    and in the same endianness as integers.

    @param ea: linear address

    @return: float
    """
    tmp = struct.pack("I", get_wide_dword(ea))
    return struct.unpack("f", tmp)[0]


def GetDouble(ea):
    """
    Get value of a floating point number (8 bytes)
    This function assumes number stored using IEEE format
    and in the same endianness as integers.

    @param ea: linear address
```

```
    @return: double
    """
    tmp = struct.pack("Q", get_qword(ea))
    return struct.unpack("d", tmp)[0]


def get_name_ea_simple(name):
    """
    Get linear address of a name

    @param name: name of program byte

    @return: address of the name
             BADADDR - No such name
    """
    return ida_name.get_name_ea(BADADDR, name)


get_name_ea = ida_name.get_name_ea


def get_segm_by_sel(base):
    """
    Get segment by segment base

    @param base: segment base paragraph or selector

    @return: linear address of the start of the segment or BADADDR
             if no such segment
    """
    sel = ida_segment.find_selector(base)
    seg = ida_segment.get_segm_by_sel(sel)

    if seg:
        return seg.start_ea
    else:
        return BADADDR


get_screen_ea = ida_kernwin.get_screen_ea


def get_curline():
    """
    Get the disassembly line at the cursor
```

```
    @return: string
    """
    return ida_lines.tag_remove(ida_kernwin.get_curline())


def read_selection_start():
    """
    Get start address of the selected range
    returns BADADDR - the user has not selected an range
    """
    selection, startaddr, endaddr = ida_kernwin.read_range_selection(None)

    if selection == 1:
        return startaddr
    else:
        return BADADDR


def read_selection_end():
    """
    Get end address of the selected range

    @return: BADADDR - the user has not selected an range
    """
    selection, startaddr, endaddr = ida_kernwin.read_range_selection(None)

    if selection == 1:
        return endaddr
    else:
        return BADADDR


def get_sreg(ea, reg):
    """
    Get value of segment register at the specified address

    @param ea: linear address
    @param reg: name of segment register

    @return: the value of the segment register or -1 on error

    @note: The segment registers in 32bit program usually contain selectors,
           so to get paragraph pointed to by the segment register you need to
           call sel2para() function.
    """
    reg = ida_idp.str2reg(reg);
```

```
    if reg >= 0:
        return ida_segregs.get_sreg(ea, reg)
    else:
        return -1

next_addr = ida_bytes.next_addr
prev_addr = ida_bytes.prev_addr


def next_head(ea, maxea=BADADDR):
    """
    Get next defined item (instruction or data) in the program

    @param ea: linear address to start search from
    @param maxea: the search will stop at the address
        maxea is not included in the search range

    @return: BADADDR - no (more) defined items
    """
    return ida_bytes.next_head(ea, maxea)


def prev_head(ea, minea=0):
    """
    Get previous defined item (instruction or data) in the program

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

    @return: BADADDR - no (more) defined items
    """
    return ida_bytes.prev_head(ea, minea)


next_not_tail = ida_bytes.next_not_tail
prev_not_tail = ida_bytes.prev_not_tail
get_item_head = ida_bytes.get_item_head
get_item_end = ida_bytes.get_item_end


def get_item_size(ea):
    """
    Get size of instruction or data item in bytes

    @param ea: linear address
```

```
    @return: 1..n
    """
    return ida_bytes.get_item_end(ea) - ea


def func_contains(func_ea, ea):
    """
    Does the given function contain the given address?

    @param func_ea: any address belonging to the function
    @param ea: linear address

    @return:  success
    """
    func = ida_funcs.get_func(func_ea)

    if func:
        return ida_funcs.func_contains(func, ea)
    return False


GN_VISIBLE = ida_name.GN_VISIBLE     # replace forbidden characters by SUBSTCHAR
GN_COLORED = ida_name.GN_COLORED     # return colored name
GN_DEMANGLED = ida_name.GN_DEMANGLED # return demangled name
GN_STRICT = ida_name.GN_STRICT       # fail if cannot demangle
GN_SHORT = ida_name.GN_SHORT         # use short form of demangled name
GN_LONG = ida_name.GN_LONG           # use long form of demangled name
GN_LOCAL = ida_name.GN_LOCAL         # try to get local name first; if failed, get global
GN_ISRET = ida_name.GN_ISRET         # for dummy names: use retloc
GN_NOT_ISRET = ida_name.GN_NOT_ISRET # for dummy names: do not use retloc


calc_gtn_flags = ida_name.calc_gtn_flags


def get_name(ea, gtn_flags=0):
    """
    Get name at the specified address

    @param ea: linear address
    @param gtn_flags: how exactly the name should be retrieved.
                      combination of GN_ bits

    @return: "" - byte has no name
    """
```

```
    return ida_name.get_ea_name(ea, gtn_flags)


def demangle_name(name, disable_mask):
    """
    demangle_name a name

    @param name: name to demangle
    @param disable_mask: a mask that tells how to demangle the name
            it is a good idea to get this mask using
            get_inf_attr(INF_SHORT_DN) or get_inf_attr(INF_LONG_DN)

    @return: a demangled name
        If the input name cannot be demangled, returns None
    """
    return ida_name.demangle_name(name, disable_mask, ida_name.DQT_FULL)


def generate_disasm_line(ea, flags):
    """
    Get disassembly line

    @param ea: linear address of instruction

    @param flags: combination of the GENDSM_ flags, or 0

    @return: "" - could not decode instruction at the specified location

    @note: this function may not return exactly the same mnemonics
           as you see on the screen.
    """
    text = ida_lines.generate_disasm_line(ea, flags)
    if text:
        return ida_lines.tag_remove(text)
    else:
        return ""

# flags for generate_disasm_line
# generate a disassembly line as if
# there is an instruction at 'ea'
GENDSM_FORCE_CODE = ida_lines.GENDSM_FORCE_CODE

# if the instruction consists of several lines,
# produce all of them (useful for parallel instructions)
GENDSM_MULTI_LINE = ida_lines.GENDSM_MULTI_LINE
```

```
def GetDisasm(ea):
    """
    Get disassembly line

    @param ea: linear address of instruction

    @return: "" - could not decode instruction at the specified location

    @note: this function may not return exactly the same mnemonics
           as you see on the screen.
    """
    return generate_disasm_line(ea, 0)

def print_insn_mnem(ea):
    """
    Get instruction mnemonics

    @param ea: linear address of instruction

    @return: "" - no instruction at the specified location

    @note: this function may not return exactly the same mnemonics
    as you see on the screen.
    """
    res = ida_ua.ua_mnem(ea)

    if not res:
        return ""
    else:
        return res


def print_operand(ea, n):
    """
    Get operand of an instruction or data

    @param ea: linear address of the item
    @param n: number of operand:
        0 - the first operand
        1 - the second operand

    @return: the current text representation of operand or ""
    """

    res = ida_ua.print_operand(ea, n)
```

```
    if not res:
        return ""
    else:
        return ida_lines.tag_remove(res)


def get_operand_type(ea, n):
    """
    Get type of instruction operand

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

    @return: any of o_* constants or -1 on error
    """
    insn = ida_ua.insn_t()
    inslen = ida_ua.decode_insn(insn, ea)
    return -1 if inslen == 0 else insn.ops[n].type


o_void     = ida_ua.o_void      # No Operand                           ----------
o_reg      = ida_ua.o_reg       # General Register (al,ax,es,ds...)    reg
o_mem      = ida_ua.o_mem       # Direct Memory Reference  (DATA)      addr
o_phrase   = ida_ua.o_phrase    # Memory Ref [Base Reg + Index Reg]    phrase
o_displ    = ida_ua.o_displ     # Memory Reg [Base Reg + Index Reg + Displacement] phrase+ad
o_imm      = ida_ua.o_imm       # Immediate Value                      value
o_far      = ida_ua.o_far       # Immediate Far Address  (CODE)        addr
o_near     = ida_ua.o_near      # Immediate Near Address (CODE)        addr
o_idpspec0 = ida_ua.o_idpspec0  # Processor specific type
o_idpspec1 = ida_ua.o_idpspec1  # Processor specific type
o_idpspec2 = ida_ua.o_idpspec2  # Processor specific type
o_idpspec3 = ida_ua.o_idpspec3  # Processor specific type
o_idpspec4 = ida_ua.o_idpspec4  # Processor specific type
o_idpspec5 = ida_ua.o_idpspec5  # Processor specific type
                                # There can be more processor specific types

# x86
o_trreg  =       ida_ua.o_idpspec0      # trace register
o_dbreg  =       ida_ua.o_idpspec1      # debug register
o_crreg  =       ida_ua.o_idpspec2      # control register
o_fpreg  =       ida_ua.o_idpspec3      # floating point register
o_mmxreg  =      ida_ua.o_idpspec4      # mmx register
o_xmmreg  =      ida_ua.o_idpspec5      # xmm register
```

```
# arm
o_reglist  =     ida_ua.o_idpspec1      # Register list (for LDM/STM)
o_creglist  =    ida_ua.o_idpspec2      # Coprocessor register list (for CDP)
o_creg  =        ida_ua.o_idpspec3      # Coprocessor register (for LDC/STC)
o_fpreglist  =   ida_ua.o_idpspec4      # Floating point register list
o_text  =        ida_ua.o_idpspec5      # Arbitrary text stored in the operand
o_cond  =        (ida_ua.o_idpspec5+1)  # ARM condition as an operand

# ppc
o_spr  =         ida_ua.o_idpspec0      # Special purpose register
o_twofpr  =      ida_ua.o_idpspec1      # Two FPRs
o_shmbme  =      ida_ua.o_idpspec2      # SH & MB & ME
o_crf  =         ida_ua.o_idpspec3      # crfield      x.reg
o_crb  =         ida_ua.o_idpspec4      # crbit        x.reg
o_dcr  =         ida_ua.o_idpspec5      # Device control register

def get_operand_value(ea, n):
    """
    Get number used in the operand

    This function returns an immediate number used in the operand

    @param ea: linear address of instruction
    @param n: the operand number

    @return: value
        operand is an immediate value  => immediate value
        operand has a displacement     => displacement
        operand is a direct memory ref => memory address
        operand is a register          => register number
        operand is a register phrase   => phrase number
        otherwise                      => -1
    """
    insn = ida_ua.insn_t()
    inslen = ida_ua.decode_insn(insn, ea)
    if inslen == 0:
        return -1
    op = insn.ops[n]
    if not op:
        return -1

    if op.type in [ ida_ua.o_mem, ida_ua.o_far, ida_ua.o_near, ida_ua.o_displ ]:
        value = op.addr
    elif op.type == ida_ua.o_reg:
        value = op.reg
    elif op.type == ida_ua.o_imm:
```

```
        value = op.value
    elif op.type == ida_ua.o_phrase:
        value = op.phrase
    else:
        value = -1
    return value


GetCommentEx = ida_bytes.get_cmt
get_cmt = GetCommentEx
get_forced_operand = ida_bytes.get_forced_operand

BPU_1B = ida_nalt.BPU_1B
BPU_2B = ida_nalt.BPU_2B
BPU_4B = ida_nalt.BPU_4B

STRWIDTH_1B   = ida_nalt.STRWIDTH_1B
STRWIDTH_2B   = ida_nalt.STRWIDTH_2B
STRWIDTH_4B   = ida_nalt.STRWIDTH_4B
STRWIDTH_MASK = ida_nalt.STRWIDTH_MASK

STRLYT_TERMCHR = ida_nalt.STRLYT_TERMCHR
STRLYT_PASCAL1 = ida_nalt.STRLYT_PASCAL1
STRLYT_PASCAL2 = ida_nalt.STRLYT_PASCAL2
STRLYT_PASCAL4 = ida_nalt.STRLYT_PASCAL4
STRLYT_MASK    = ida_nalt.STRLYT_MASK
STRLYT_SHIFT   = ida_nalt.STRLYT_SHIFT

# Character-terminated string. The termination characters
# are kept in the next bytes of string type.
STRTYPE_TERMCHR   = ida_nalt.STRTYPE_TERMCHR
# C-style string.
STRTYPE_C         = ida_nalt.STRTYPE_C
# Zero-terminated 16bit chars
STRTYPE_C_16      = ida_nalt.STRTYPE_C_16
# Zero-terminated 32bit chars
STRTYPE_C_32      = ida_nalt.STRTYPE_C_32
# Pascal-style, one-byte length prefix
STRTYPE_PASCAL    = ida_nalt.STRTYPE_PASCAL
# Pascal-style, 16bit chars, one-byte length prefix
STRTYPE_PASCAL_16 = ida_nalt.STRTYPE_PASCAL_16
# Pascal-style, two-byte length prefix
STRTYPE_LEN2      = ida_nalt.STRTYPE_LEN2
# Pascal-style, 16bit chars, two-byte length prefix
STRTYPE_LEN2_16   = ida_nalt.STRTYPE_LEN2_16
# Pascal-style, four-byte length prefix
```

```
STRTYPE_LEN4      = ida_nalt.STRTYPE_LEN4
# Pascal-style, 16bit chars, four-byte length prefix
STRTYPE_LEN4_16   = ida_nalt.STRTYPE_LEN4_16

# alias
STRTYPE_C16       = STRTYPE_C_16

def get_strlit_contents(ea, length = -1, strtype = STRTYPE_C):
    """
    Get string contents
    @param ea: linear address
    @param length: string length. -1 means to calculate the max string length
    @param strtype: the string type (one of STRTYPE_... constants)

    @return: string contents or empty string
    """
    if length == -1:
        length = ida_bytes.get_max_strlit_length(ea, strtype, ida_bytes.ALOPT_IGNHEADS)

    return ida_bytes.get_strlit_contents(ea, length, strtype)


def get_str_type(ea):
    """
    Get string type

    @param ea: linear address

    @return: One of STRTYPE_... constants
    """
    flags = ida_bytes.get_flags(ea)
    if ida_bytes.is_strlit(flags):
        oi = ida_nalt.opinfo_t()
        if ida_bytes.get_opinfo(oi, ea, 0, flags):
            return oi.strtype

#      The following functions search for the specified byte
#          ea - address to start from
#          flag is combination of the following bits

#      returns BADADDR - not found
find_suspop  = ida_search.find_suspop
find_code    = ida_search.find_code
find_data    = ida_search.find_data
find_unknown = ida_search.find_unknown
find_defined = ida_search.find_defined
```

```
find_imm     = ida_search.find_imm

SEARCH_UP       = ida_search.SEARCH_UP       # search backward
SEARCH_DOWN     = ida_search.SEARCH_DOWN     # search forward
SEARCH_NEXT     = ida_search.SEARCH_NEXT     # start the search at the next/prev item
                                             # useful only for find_text() and find_binary()
SEARCH_CASE     = ida_search.SEARCH_CASE     # search case-sensitive
                                             # (only for bin&txt search)
SEARCH_REGEX    = ida_search.SEARCH_REGEX    # enable regular expressions (only for text)
SEARCH_NOBRK    = ida_search.SEARCH_NOBRK    # don't test ctrl-break
SEARCH_NOSHOW   = ida_search.SEARCH_NOSHOW   # don't display the search progress


def find_text(ea, flag, y, x, searchstr):
    __warn_once_deprecated_proto_confusion("find_text", "ida_search.find_text")
    return ida_search.find_text(ea, y, x, searchstr, flag)


def find_binary(ea, flag, searchstr, radix=16):
    __warn_once_deprecated_proto_confusion("find_binary", "ida_search.find_binary")
    endea = flag & 1 and ida_ida.cvar.inf.max_ea or ida_ida.cvar.inf.min_ea
    return ida_search.find_binary(ea, endea, searchstr, radix, flag)


#----------------------------------------------------------------------------
#       G L O B A L   S E T T I N G S   M A N I P U L A T I O N
#----------------------------------------------------------------------------
def process_config_line(directive):
    """
    Obsolete. Please use ida_idp.process_config_directive().
    """
    return eval_idc('process_config_directive("%s")' % ida_kernwin.str2user(directive))


# The following functions allow you to set/get common parameters.
# Please note that not all parameters can be set directly.



INF_VERSION     = 0             # short;   Version of database
INF_PROCNAME    = 1             # char[8]; Name of current processor
INF_GENFLAGS    = 2             # ushort;  General flags:
INF_LFLAGS      = 3             # uint32;  IDP-dependent flags

INF_DATABASE_CHANGE_COUNT= 4  # uint32; database change counter; keeps track of byte and seg
INF_CHANGE_COUNTER=INF_DATABASE_CHANGE_COUNT
```

```
INF_FILETYPE   = 5            # short;   type of input file (see ida.hpp)
FT_EXE_OLD     = 0            #              MS DOS EXE File (obsolete)
FT_COM_OLD     = 1            #              MS DOS COM File (obsolete)
FT_BIN         = 2            #              Binary File
FT_DRV         = 3            #              MS DOS Driver
FT_WIN         = 4            #              New Executable (NE)
FT_HEX         = 5            #              Intel Hex Object File
FT_MEX         = 6            #              MOS Technology Hex Object File
FT_LX          = 7            #              Linear Executable (LX)
FT_LE          = 8            #              Linear Executable (LE)
FT_NLM         = 9            #              Netware Loadable Module (NLM)
FT_COFF        = 10           #              Common Object File Format (COFF)
FT_PE          = 11           #              Portable Executable (PE)
FT_OMF         = 12           #              Object Module Format
FT_SREC        = 13           #              R-records
FT_ZIP         = 14           #              ZIP file (this file is never loaded to IDA dat
FT_OMFLIB      = 15           #              Library of OMF Modules
FT_AR          = 16           #              ar library
FT_LOADER      = 17           #              file is loaded using LOADER DLL
FT_ELF         = 18           #              Executable and Linkable Format (ELF)
FT_W32RUN      = 19           #              Watcom DOS32 Extender (W32RUN)
FT_AOUT        = 20           #              Linux a.out (AOUT)
FT_PRC         = 21           #              PalmPilot program file
FT_EXE         = 22           #              MS DOS EXE File
FT_COM         = 23           #              MS DOS COM File
FT_AIXAR       = 24           #              AIX ar library
FT_MACHO       = 25           #              Mac OS X Mach-O file
INF_OSTYPE     = 6            # short;   FLIRT: OS type the program is for
OSTYPE_MSDOS= 0x0001
OSTYPE_WIN  = 0x0002
OSTYPE_OS2  = 0x0004
OSTYPE_NETW = 0x0008
INF_APPTYPE    = 7            # short;   FLIRT: Application type
APPT_CONSOLE= 0x0001          #              console
APPT_GRAPHIC= 0x0002          #              graphics
APPT_PROGRAM= 0x0004          #              EXE
APPT_LIBRARY= 0x0008          #              DLL
APPT_DRIVER = 0x0010          #              DRIVER
APPT_1THREAD= 0x0020          #              Singlethread
APPT_MTHREAD= 0x0040          #              Multithread
APPT_16BIT  = 0x0080          #              16 bit application
APPT_32BIT  = 0x0100          #              32 bit application
INF_ASMTYPE    = 8            # char;    target assembler number (0..n)
INF_SPECSEGS   = 9
```

```
INF_AF         = 10           # uint32;   Analysis flags:

def _import_module_flag_sets(module, prefixes):
    if isinstance(prefixes, str):
        prefixes = [prefixes]
    for prefix in prefixes:
        for key in dir(module):
            if key.startswith(prefix):
                value = getattr(module, key)
                if isinstance(value, ida_idaapi.integer_types):
                    globals()[key] = value
_import_module_flag_sets(
    ida_ida,
    [
        "INFFL_",
        "LFLG_",
        "IDB_",
        "AF_",
        "AF2_",
        "SW_",
        "NM_",
        "DEMNAM_",
        "LN_",
        "OFLG_",
        "SCF_",
        "LMT_",
        "PREF_",
        "STRF_",
        "ABI_",
    ])

INF_AF2        = 11           # uint32;  Analysis flags 2

INF_BASEADDR   = 12           # uval_t;  base paragraph of the program
INF_START_SS   = 13           # int32;   value of SS at the start
INF_START_CS   = 14           # int32;   value of CS at the start
INF_START_IP   = 15           # ea_t;    IP register value at the start of
                              #          program execution
INF_START_EA   = 16           # ea_t;    Linear address of program entry point
INF_START_SP   = 17           # ea_t;    SP register value at the start of
                              #          program execution
INF_MAIN       = 18           # ea_t;    address of main()
INF_MIN_EA     = 19           # ea_t;    The lowest address used
                              #          in the program
INF_MAX_EA     = 20           # ea_t;    The highest address used
                              #          in the program - 1
```

```
INF_OMIN_EA    = 21
INF_OMAX_EA    = 22
INF_LOWOFF     = 23            # ea_t;    low limit of voids
INF_LOW_OFF=INF_LOWOFF
INF_HIGHOFF    = 24            # ea_t;    high limit of voids
INF_HIGH_OFF=INF_HIGHOFF
INF_MAXREF     = 25            # uval_t;  max xref depth
INF_PRIVRANGE_START_EA = 27    # uval_t; Range of addresses reserved for internal use.
INF_START_PRIVRANGE=INF_PRIVRANGE_START_EA
INF_PRIVRANGE_END_EA = 28      # uval_t; Initially (MAXADDR, MAXADDR+0x100000)
INF_END_PRIVRANGE=INF_PRIVRANGE_END_EA

INF_NETDELTA   = 29            # sval_t; Delta value to be added to all adresses for mapping
                               # Initially 0.
# CROSS REFERENCES
INF_XREFNUM    = 30            # char;    Number of references to generate
                               #          0 - xrefs won't be generated at all
INF_TYPE_XREFNUM = 31         # char;    Number of references to generate
                               #          in the struct & enum windows
                               #          0 - xrefs won't be generated at all
INF_TYPE_XREFS=INF_TYPE_XREFNUM
INF_REFCMTNUM  = 32           # uchar; number of comment lines to
                               #        generate for refs to ASCII
                               #        string or demangled name
                               #        0 - such comments won't be
                               #        generated at all
INF_REFCMTS=INF_REFCMTNUM
INF_XREFFLAG   = 33           # char;    xrefs representation:
INF_XREFS=INF_XREFFLAG

# NAMES
INF_MAX_AUTONAME_LEN = 34      # ushort;  max name length (without zero byte)
INF_NAMETYPE   = 35            # char;    dummy names represenation type
INF_SHORT_DEMNAMES = 36       # int32;   short form of demangled names
INF_SHORT_DN=INF_SHORT_DEMNAMES
INF_LONG_DEMNAMES = 37        # int32;   long form of demangled names
                               #          see demangle.h for definitions
INF_LONG_DN=INF_LONG_DEMNAMES
INF_DEMNAMES   = 38            # char;    display demangled names as:
INF_LISTNAMES  = 39            # uchar;   What names should be included in the list?

# DISASSEMBLY LISTING DETAILS
INF_INDENT     = 40            # char;    Indention for instructions
INF_CMT_INDENT = 41            # char;    Indention for comments
INF_COMMENT    = 41            # for compatibility
INF_MARGIN     = 42            # ushort;  max length of data lines
```

```
INF_LENXREF    = 43           # ushort;  max length of line with xrefs
INF_OUTFLAGS   = 44           # uint32;  output flags
INF_CMTFLG     = 45           # char;    comments:
INF_CMTFLAG=INF_CMTFLG
INF_LIMITER    = 46           # char;    Generate borders?
INF_BORDER=INF_LIMITER
INF_BIN_PREFIX_SIZE = 47     # short;   # of instruction bytes to show
                              #          in line prefix
INF_BINPREF=INF_BIN_PREFIX_SIZE
INF_PREFFLAG   = 48           # char;    line prefix type:

# STRING LITERALS
INF_STRLIT_FLAGS= 49          # uchar;   string literal flags
INF_STRLIT_BREAK= 50         # char;    string literal line break symbol
INF_STRLIT_ZEROES= 51        # char;    leading zeroes
INF_STRTYPE    = 52          # int32;   current ascii string type
                              #          is considered as several bytes:
                              #      low byte:

INF_STRLIT_PREF  = 53         # char[16];ASCII names prefix
INF_STRLIT_SERNUM= 54         # uint32;  serial number

# DATA ITEMS
INF_DATATYPES    = 55         # int32;   data types allowed in data carousel

# COMPILER
INF_CC_ID        = 57         # uchar;   compiler
COMP_MASK    = 0x0F           #              mask to apply to get the pure compiler id
COMP_UNK     = 0x00           # Unknown
COMP_MS      = 0x01           # Visual C++
COMP_BC      = 0x02           # Borland C++
COMP_WATCOM  = 0x03           # Watcom C++
COMP_GNU     = 0x06           # GNU C++
COMP_VISAGE  = 0x07           # Visual Age C++
COMP_BP      = 0x08           # Delphi
INF_CC_CM        = 58         # uchar;  memory model & calling convention
INF_CC_SIZE_I    = 59         # uchar;  sizeof(int)
INF_CC_SIZE_B    = 60         # uchar;  sizeof(bool)
INF_CC_SIZE_E    = 61         # uchar;  sizeof(enum)
INF_CC_DEFALIGN  = 62         # uchar;  default alignment
INF_CC_SIZE_S    = 63
INF_CC_SIZE_L    = 64
INF_CC_SIZE_LL   = 65
INF_CC_SIZE_LDBL = 66         # uchar;  sizeof(long double)
INF_COMPILER    = INF_CC_ID
INF_MODEL       = INF_CC_CM
```

```
INF_SIZEOF_INT  = INF_CC_SIZE_I
INF_SIZEOF_BOOL = INF_CC_SIZE_B
INF_SIZEOF_ENUM = INF_CC_SIZE_E
INF_SIZEOF_ALGN = INF_CC_DEFALIGN
INF_SIZEOF_SHORT= INF_CC_SIZE_S
INF_SIZEOF_LONG = INF_CC_SIZE_L
INF_SIZEOF_LLONG= INF_CC_SIZE_LL
INF_SIZEOF_LDBL = INF_CC_SIZE_LDBL
INF_ABIBITS= 67               # uint32; ABI features
INF_APPCALL_OPTIONS= 68       # uint32; appcall options

_INF_attrs_accessors = {
    INF_ABIBITS               : (ida_ida.inf_get_abibits,               ida_ida.inf_set_abi
    INF_AF                    : (ida_ida.inf_get_af,                    ida_ida.inf_set_af),
    INF_AF2                   : (ida_ida.inf_get_af2,                   ida_ida.inf_set_af2)
    INF_APPCALL_OPTIONS       : (ida_ida.inf_get_appcall_options,       ida_ida.inf_set_appc
    INF_APPTYPE               : (ida_ida.inf_get_apptype,               ida_ida.inf_set_appt
    INF_ASMTYPE               : (ida_ida.inf_get_asmtype,               ida_ida.inf_set_asmt
    INF_BASEADDR              : (ida_ida.inf_get_baseaddr,              ida_ida.inf_set_base
    INF_BIN_PREFIX_SIZE       : (ida_ida.inf_get_bin_prefix_size,       ida_ida.inf_set_bin_
    INF_CC_CM                 : (ida_ida.inf_get_cc_cm,                 ida_ida.inf_set_cc_c
    INF_CC_DEFALIGN           : (ida_ida.inf_get_cc_defalign,           ida_ida.inf_set_cc_c
    INF_CC_ID                 : (ida_ida.inf_get_cc_id,                 ida_ida.inf_set_cc_i
    INF_CC_SIZE_B             : (ida_ida.inf_get_cc_size_b,             ida_ida.inf_set_cc_s
    INF_CC_SIZE_E             : (ida_ida.inf_get_cc_size_e,             ida_ida.inf_set_cc_s
    INF_CC_SIZE_I             : (ida_ida.inf_get_cc_size_i,             ida_ida.inf_set_cc_s
    INF_CC_SIZE_L             : (ida_ida.inf_get_cc_size_l,             ida_ida.inf_set_cc_s
    INF_CC_SIZE_LDBL          : (ida_ida.inf_get_cc_size_ldbl,          ida_ida.inf_set_cc_s
    INF_CC_SIZE_LL            : (ida_ida.inf_get_cc_size_ll,            ida_ida.inf_set_cc_s
    INF_CC_SIZE_S             : (ida_ida.inf_get_cc_size_s,             ida_ida.inf_set_cc_s
    INF_CMTFLAG               : (ida_ida.inf_get_cmtflg,                ida_ida.inf_set_cmtf
    INF_CMT_INDENT            : (ida_ida.inf_get_cmt_indent,            ida_ida.inf_set_cmt_
    INF_DATABASE_CHANGE_COUNT : (ida_ida.inf_get_database_change_count, ida_ida.inf_set_data
    INF_DATATYPES             : (ida_ida.inf_get_datatypes,             ida_ida.inf_set_data
    INF_DEMNAMES              : (ida_ida.inf_get_demnames,              ida_ida.inf_set_demn
    INF_END_PRIVRANGE         : (ida_ida.inf_get_privrange_end_ea,      ida_ida.inf_set_priv
    INF_FILETYPE              : (ida_ida.inf_get_filetype,              ida_ida.inf_set_file
    INF_GENFLAGS              : (ida_ida.inf_get_genflags,              ida_ida.inf_set_genf
    INF_HIGHOFF               : (ida_ida.inf_get_highoff,               ida_ida.inf_set_high
    INF_INDENT                : (ida_ida.inf_get_indent,                ida_ida.inf_set_inde
    INF_LENXREF               : (ida_ida.inf_get_lenxref,               ida_ida.inf_set_lenx
    INF_LFLAGS                : (ida_ida.inf_get_lflags,                ida_ida.inf_set_lfla
    INF_LIMITER               : (ida_ida.inf_get_limiter,               ida_ida.inf_set_limi
    INF_LISTNAMES             : (ida_ida.inf_get_listnames,             ida_ida.inf_set_list
    INF_LONG_DEMNAMES         : (ida_ida.inf_get_long_demnames,         ida_ida.inf_set_long
    INF_LOWOFF                : (ida_ida.inf_get_lowoff,                ida_ida.inf_set_lowo
```

```
    INF_MAIN                  : (ida_ida.inf_get_main,                  ida_ida.inf_set_main
    INF_MARGIN                : (ida_ida.inf_get_margin,                ida_ida.inf_set_marg
    INF_MAXREF                : (ida_ida.inf_get_maxref,                ida_ida.inf_set_maxr
    INF_MAX_AUTONAME_LEN      : (ida_ida.inf_get_max_autoname_len,      ida_ida.inf_set_max_
    INF_MAX_EA                : (ida_ida.inf_get_max_ea,                ida_ida.inf_set_max_
    INF_MIN_EA                : (ida_ida.inf_get_min_ea,                ida_ida.inf_set_min_
    INF_MODEL                 : (ida_ida.inf_get_cc_cm,                 ida_ida.inf_set_cc_c
    INF_NAMETYPE              : (ida_ida.inf_get_nametype,              ida_ida.inf_set_name
    INF_NETDELTA              : (ida_ida.inf_get_netdelta,              ida_ida.inf_set_netd
    INF_OMAX_EA               : (ida_ida.inf_get_omax_ea,               ida_ida.inf_set_omax
    INF_OMIN_EA               : (ida_ida.inf_get_omin_ea,               ida_ida.inf_set_omin
    INF_OSTYPE                : (ida_ida.inf_get_ostype,                ida_ida.inf_set_osty
    INF_OUTFLAGS              : (ida_ida.inf_get_outflags,              ida_ida.inf_set_outf
    INF_PREFFLAG              : (ida_ida.inf_get_prefflag,              ida_ida.inf_set_pref
    INF_PRIVRANGE_END_EA      : (ida_ida.inf_get_privrange_end_ea,      ida_ida.inf_set_priv
    INF_PRIVRANGE_START_EA    : (ida_ida.inf_get_privrange_start_ea,    ida_ida.inf_set_priv
    INF_PROCNAME              : (ida_ida.inf_get_procname,              ida_ida.inf_set_proc
    INF_REFCMTNUM             : (ida_ida.inf_get_refcmtnum,             ida_ida.inf_set_refc
    INF_SHORT_DEMNAMES        : (ida_ida.inf_get_short_demnames,        ida_ida.inf_set_shor
    INF_SPECSEGS              : (ida_ida.inf_get_specsegs,              ida_ida.inf_set_spec
    INF_START_CS              : (ida_ida.inf_get_start_cs,              ida_ida.inf_set_star
    INF_START_EA              : (ida_ida.inf_get_start_ea,              ida_ida.inf_set_star
    INF_START_IP              : (ida_ida.inf_get_start_ip,              ida_ida.inf_set_star
    INF_START_PRIVRANGE       : (ida_ida.inf_get_privrange_start_ea,    ida_ida.inf_set_priv
    INF_START_SP              : (ida_ida.inf_get_start_sp,              ida_ida.inf_set_star
    INF_START_SS              : (ida_ida.inf_get_start_ss,              ida_ida.inf_set_star
    INF_STRLIT_BREAK          : (ida_ida.inf_get_strlit_break,          ida_ida.inf_set_strl
    INF_STRLIT_FLAGS          : (ida_ida.inf_get_strlit_flags,          ida_ida.inf_set_strl
    INF_STRLIT_PREF           : (ida_ida.inf_get_strlit_pref,           ida_ida.inf_set_strl
    INF_STRLIT_SERNUM         : (ida_ida.inf_get_strlit_sernum,         ida_ida.inf_set_strl
    INF_STRLIT_ZEROES         : (ida_ida.inf_get_strlit_zeroes,         ida_ida.inf_set_strl
    INF_STRTYPE               : (ida_ida.inf_get_strtype,               ida_ida.inf_set_strt
    INF_TYPE_XREFNUM          : (ida_ida.inf_get_type_xrefnum,          ida_ida.inf_set_type
    INF_VERSION               : (ida_ida.inf_get_version,               ida_ida.inf_set_vers
    INF_XREFFLAG              : (ida_ida.inf_get_xrefflag,              ida_ida.inf_set_xref
    INF_XREFNUM               : (ida_ida.inf_get_xrefnum,               ida_ida.inf_set_xref
}

def get_inf_attr(attr):
    """
    Deprecated. Please ida_ida.inf_get_* instead.
    """
    return _INF_attrs_accessors[attr][0]()

def set_inf_attr(attr, value):
    """
```

```
    Deprecated. Please ida_ida.inf_set_* instead.
    """
    _INF_attrs_accessors[attr][1](value)
    return 1

set_processor_type  = ida_idp.set_processor_type

SETPROC_IDB              = ida_idp.SETPROC_IDB
SETPROC_LOADER           = ida_idp.SETPROC_LOADER
SETPROC_LOADER_NON_FATAL = ida_idp.SETPROC_LOADER_NON_FATAL
SETPROC_USER             = ida_idp.SETPROC_USER

def SetPrcsr(processor): return set_processor_type(processor, SETPROC_USER)

def get_processor_name():
    """
    Get name of the current processor
    @return: processor name
    """
    return ida_ida.inf_get_procname()

set_target_assembler = ida_idp.set_target_assembler


def batch(batch):
    """
    Enable/disable batch mode of operation

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

    @return: old balue of batch flag
    """
    batch_prev = ida_kernwin.cvar.batch
    ida_kernwin.cvar.batch = batch
    return batch_prev


#----------------------------------------------------------------------------
#          I N T E R A C T I O N   W I T H   T H E   U S E R
#----------------------------------------------------------------------------
def process_ui_action(name, flags=0):
    """
    Invokes an IDA UI action by name
```

```
    @param name: Command name
    @param flags: Reserved. Must be zero
    @return: Boolean
    """
    return ida_kernwin.process_ui_action(name, flags)

ask_seg = ida_kernwin.ask_seg
ask_yn = ida_kernwin.ask_yn
msg = ida_kernwin.msg
warning = ida_kernwin.warning
error = ida_kernwin.error
set_ida_state = ida_auto.set_ida_state


IDA_STATUS_READY    = 0 # READY     IDA is idle
IDA_STATUS_THINKING = 1 # THINKING  Analyzing but the user may press keys
IDA_STATUS_WAITING  = 2 # WAITING   Waiting for the user input
IDA_STATUS_WORK     = 3 # BUSY      IDA is busy


refresh_idaview_anyway = ida_kernwin.refresh_idaview_anyway
refresh_lists = ida_kernwin.refresh_choosers


#--------------------------------------------------------------------------
#                       S E G M E N T A T I O N
#--------------------------------------------------------------------------
def sel2para(sel):
    """
    Get a selector value

    @param sel: the selector number

    @return: selector value if found
             otherwise the input value (sel)

    @note: selector values are always in paragraphs
    """
    s = ida_pro.sel_pointer()
    base = ida_pro.ea_pointer()
    res,tmp = ida_segment.getn_selector(sel, s.cast(), base.cast())

    if not res:
        return sel
    else:
        return base.value()
```

```
def find_selector(val):
    """
    Find a selector which has the specifed value

    @param val: value to search for

    @return: the selector number if found,
             otherwise the input value (val & 0xFFFF)

    @note: selector values are always in paragraphs
    """
    return ida_segment.find_selector(val) & 0xFFFF


set_selector = ida_segment.set_selector
del_selector = ida_segment.del_selector


def get_first_seg():
    """
    Get first segment

    @return: address of the start of the first segment
        BADADDR - no segments are defined
    """
    seg = ida_segment.get_first_seg()
    if not seg:
        return BADADDR
    else:
        return seg.start_ea


def get_next_seg(ea):
    """
    Get next segment

    @param ea: linear address

    @return: start of the next segment
             BADADDR - no next segment
    """
    nextseg = ida_segment.get_next_seg(ea)
    if not nextseg:
        return BADADDR
```

```
    else:
        return nextseg.start_ea


def get_segm_start(ea):
    """
    Get start address of a segment

    @param ea: any address in the segment

    @return: start of segment
             BADADDR - the specified address doesn't belong to any segment
    """
    seg = ida_segment.getseg(ea)

    if not seg:
        return BADADDR
    else:
        return seg.start_ea


def get_segm_end(ea):
    """
    Get end address of a segment

    @param ea: any address in the segment

    @return: end of segment (an address past end of the segment)
             BADADDR - the specified address doesn't belong to any segment
    """
    seg = ida_segment.getseg(ea)

    if not seg:
        return BADADDR
    else:
        return seg.end_ea


def get_segm_name(ea):
    """
    Get name of a segment

    @param ea: any address in the segment

    @return: "" - no segment at the specified address
    """
```

```
    seg = ida_segment.getseg(ea)

    if not seg:
        return ""
    else:
        name = ida_segment.get_segm_name(seg)

        if not name:
            return ""
        else:
            return name


def add_segm_ex(startea, endea, base, use32, align, comb, flags):
    """
    Create a new segment

    @param startea: linear address of the start of the segment
    @param endea: linear address of the end of the segment
                this address will not belong to the segment
                'endea' should be higher than 'startea'
    @param 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.
    @param use32: 0: 16bit segment, 1: 32bit segment, 2: 64bit segment
    @param align: segment alignment. see below for alignment values
    @param comb: segment combination. see below for combination values.
    @param flags: combination of ADDSEG_... bits

    @return: 0-failed, 1-ok
    """
    s = ida_segment.segment_t()
    s.start_ea = startea
    s.end_ea   = endea
    s.sel      = ida_segment.setup_selector(base)
    s.bitness  = use32
    s.align    = align
    s.comb     = comb
    return ida_segment.add_segm_ex(s, "", "", flags)

ADDSEG_NOSREG  = ida_segment.ADDSEG_NOSREG  # set all default segment register values
                                            # to BADSELs
                                            # (undefine all default segment registers)
ADDSEG_OR_DIE  = ida_segment. ADDSEG_OR_DIE # qexit() if can't add a segment
ADDSEG_NOTRUNC = ida_segment.ADDSEG_NOTRUNC # don't truncate the new segment at the beginni
```

```
                                        # of the next segment if they overlap.
                                        # destroy/truncate old segments instead.
ADDSEG_QUIET   = ida_segment.ADDSEG_QUIET   # silent mode, no "Adding segment..." in the mes
ADDSEG_FILLGAP = ida_segment.ADDSEG_FILLGAP # If there is a gap between the new segment
                                        # and the previous one, and this gap is less
                                        # than 64K, then fill the gap by extending the
                                        # previous segment and adding .align directive
                                        # to it. This way we avoid gaps between segments
                                        # Too many gaps lead to a virtual array failure
                                        # It cannot hold more than ~1000 gaps.
ADDSEG_SPARSE  = ida_segment.ADDSEG_SPARSE  # Use sparse storage method for the new segment

def AddSeg(startea, endea, base, use32, align, comb):
    return add_segm_ex(startea, endea, base, use32, align, comb, ADDSEG_NOSREG)

del_segm = ida_segment.del_segm

SEGMOD_KILL   = ida_segment.SEGMOD_KILL   # disable addresses if segment gets
                                    # shrinked or deleted
SEGMOD_KEEP   = ida_segment.SEGMOD_KEEP   # keep information (code & data, etc)
SEGMOD_SILENT = ida_segment.SEGMOD_SILENT # be silent


def set_segment_bounds(ea, startea, endea, flags):
    """
    Change segment boundaries

    @param ea: any address in the segment
    @param startea: new start address of the segment
    @param endea: new end address of the segment
    @param flags: combination of SEGMOD_... flags

    @return: boolean success
    """
    return ida_segment.set_segm_start(ea, startea, flags) & \
           ida_segment.set_segm_end(ea, endea, flags)


def set_segm_name(ea, name):
    """
    Change name of the segment

    @param ea: any address in the segment
    @param name: new name of the segment

    @return: success (boolean)
```

```
    """
    seg = ida_segment.getseg(ea)

    if not seg:
        return False

    return ida_segment.set_segm_name(seg, name)


def set_segm_class(ea, segclass):
    """
    Change class of the segment

    @param ea: any address in the segment
    @param segclass: new class of the segment

    @return: success (boolean)
    """
    seg = ida_segment.getseg(ea)

    if not seg:
        return False

    return ida_segment.set_segm_class(seg, segclass)


def set_segm_alignment(ea, alignment):
    """
    Change alignment of the segment

    @param ea: any address in the segment
    @param alignment: new alignment of the segment (one of the sa... constants)

    @return: success (boolean)
    """
    return set_segm_attr(ea, SEGATTR_ALIGN, alignment)


if ida_idaapi.uses_swig_builtins:
    _scope = ida_segment.segment_t
else:
    _scope = ida_segment
saAbs        = _scope.saAbs        # Absolute segment.
saRelByte    = _scope.saRelByte    # Relocatable, byte aligned.
saRelWord    = _scope.saRelWord    # Relocatable, word (2-byte, 16-bit) aligned.
saRelPara    = _scope.saRelPara    # Relocatable, paragraph (16-byte) aligned.
```

```
saRelPage    = _scope.saRelPage    # Relocatable, aligned on 256-byte boundary
                                   # (a "page" in the original Intel specification).
saRelDble    = _scope.saRelDble    # Relocatable, aligned on a double word
                                   # (4-byte) boundary. This value is used by
                                   # the PharLap OMF for the same alignment.
saRel4K      = _scope.saRel4K      # This value is used by the PharLap OMF for
                                   # page (4K) alignment. It is not supported
                                   # by LINK.
saGroup      = _scope.saGroup      # Segment group
saRel32Bytes = _scope.saRel32Bytes # 32 bytes
saRel64Bytes = _scope.saRel64Bytes # 64 bytes
saRelQword   = _scope.saRelQword   # 8 bytes


def set_segm_combination(segea, comb):
    """
    Change combination of the segment

    @param segea: any address in the segment
    @param comb: new combination of the segment (one of the sc... constants)

    @return: success (boolean)
    """
    return set_segm_attr(segea, SEGATTR_COMB, comb)


scPriv   = _scope.scPriv   # Private. Do not combine with any other program
                           # segment.
scPub    = _scope.scPub    # Public. Combine by appending at an offset that
                           # meets the alignment requirement.
scPub2   = _scope.scPub2   # As defined by Microsoft, same as C=2 (public).
scStack  = _scope.scStack  # Stack. Combine as for C=2. This combine type
                           # forces byte alignment.
scCommon = _scope.scCommon # Common. Combine by overlay using maximum size.
scPub3   = _scope.scPub3   # As defined by Microsoft, same as C=2 (public).


def set_segm_addressing(ea, bitness):
    """
    Change segment addressing

    @param ea: any address in the segment
    @param bitness: 0: 16bit, 1: 32bit, 2: 64bit

    @return: success (boolean)
    """
```

```
    seg = ida_segment.getseg(ea)

    if not seg:
        return False

    seg.bitness = bitness

    return True


def selector_by_name(segname):
    """
    Get segment selector by name

    @param segname: name of segment

    @return: segment selector or BADADDR
    """
    seg = ida_segment.get_segm_by_name(segname)

    if not seg:
        return BADADDR

    return seg.sel


def set_default_sreg_value(ea, reg, value):
    """
    Set default segment register value for a segment

    @param ea: any address in the segment
               if no segment is present at the specified address
               then all segments will be affected
    @param reg: name of segment register
    @param value: default value of the segment register. -1-undefined.
    """
    seg = ida_segment.getseg(ea)

    reg = ida_idp.str2reg(reg);
    if seg and reg >= 0:
        return ida_segregs.set_default_sreg_value(seg, reg, value)
    else:
        return False


def set_segm_type(segea, segtype):
```

```
    """
    Set segment type

    @param segea: any address within segment
    @param segtype: new segment type:

    @return: !=0 - ok
    """
    seg = ida_segment.getseg(segea)

    if not seg:
        return False

    seg.type = segtype
    return seg.update()


SEG_NORM   = _scope.SEG_NORM
SEG_XTRN   = _scope.SEG_XTRN   # * segment with 'extern' definitions
                               #   no instructions are allowed
SEG_CODE   = _scope.SEG_CODE   # pure code segment
SEG_DATA   = _scope.SEG_DATA   # pure data segment
SEG_IMP    = _scope.SEG_IMP    # implementation segment
SEG_GRP    = _scope.SEG_GRP    # * group of segments
                               #   no instructions are allowed
SEG_NULL   = _scope.SEG_NULL   # zero-length segment
SEG_UNDF   = _scope.SEG_UNDF   # undefined segment type
SEG_BSS    = _scope.SEG_BSS    # uninitialized segment
SEG_ABSSYM = _scope.SEG_ABSSYM # * segment with definitions of absolute symbols
                               #   no instructions are allowed
SEG_COMM   = _scope.SEG_COMM   # * segment with communal definitions
                               #   no instructions are allowed
SEG_IMEM   = _scope.SEG_IMEM   # internal processor memory & sfr (8051)


def get_segm_attr(segea, attr):
    """
    Get segment attribute

    @param segea: any address within segment
    @param attr: one of SEGATTR_... constants
    """
    seg = ida_segment.getseg(segea)
    assert seg, "could not find segment at 0x%x" % segea
    if attr in [ SEGATTR_ES, SEGATTR_CS, SEGATTR_SS, SEGATTR_DS, SEGATTR_FS, SEGATTR_GS ]:
        return ida_segment.get_defsr(seg, _SEGATTRMAP[attr][1])
```

```
    else:
        return _IDC_GetAttr(seg, _SEGATTRMAP, attr)


def set_segm_attr(segea, attr, value):
    """
    Set segment attribute

    @param segea: any address within segment
    @param attr: one of SEGATTR_... constants

    @note: Please note that not all segment attributes are modifiable.
           Also some of them should be modified using special functions
           like set_segm_addressing, etc.
    """
    seg = ida_segment.getseg(segea)
    assert seg, "could not find segment at 0x%x" % segea
    if attr in [ SEGATTR_ES, SEGATTR_CS, SEGATTR_SS, SEGATTR_DS, SEGATTR_FS, SEGATTR_GS ]:
        ida_segment.set_defsr(seg, _SEGATTRMAP[attr][1], value)
    else:
        _IDC_SetAttr(seg, _SEGATTRMAP, attr, value)
    return seg.update()


SEGATTR_START   =  0      # starting address
SEGATTR_END     =  4      # ending address
SEGATTR_ORGBASE = 16
SEGATTR_ALIGN   = 20      # alignment
SEGATTR_COMB    = 21      # combination
SEGATTR_PERM    = 22      # permissions
SEGATTR_BITNESS = 23      # bitness (0: 16, 1: 32, 2: 64 bit segment)
                          # Note: modifying the attribute directly does
                          #       not lead to the reanalysis of the segment.
                          #       Using set_segm_addressing() is more correct.
SEGATTR_FLAGS   = 24      # segment flags
SEGATTR_SEL     = 28      # segment selector
SEGATTR_ES      = 32      # default ES value
SEGATTR_CS      = 36      # default CS value
SEGATTR_SS      = 40      # default SS value
SEGATTR_DS      = 44      # default DS value
SEGATTR_FS      = 48      # default FS value
SEGATTR_GS      = 52      # default GS value
SEGATTR_TYPE    = 96      # segment type
SEGATTR_COLOR   = 100     # segment color

# Redefining these for 64-bit
```

```
if __EA64__:
    SEGATTR_START   = 0
    SEGATTR_END     = 8
    SEGATTR_ORGBASE = 32
    SEGATTR_ALIGN   = 40
    SEGATTR_COMB    = 41
    SEGATTR_PERM    = 42
    SEGATTR_BITNESS = 43
    SEGATTR_FLAGS   = 44
    SEGATTR_SEL     = 48
    SEGATTR_ES      = 56
    SEGATTR_CS      = 64
    SEGATTR_SS      = 72
    SEGATTR_DS      = 80
    SEGATTR_FS      = 88
    SEGATTR_GS      = 96
    SEGATTR_TYPE    = 184
    SEGATTR_COLOR   = 188

_SEGATTRMAP = {
    SEGATTR_START   : (True, 'start_ea'),
    SEGATTR_END     : (True, 'end_ea'),
    SEGATTR_ORGBASE : (False, 'orgbase'),
    SEGATTR_ALIGN   : (False, 'align'),
    SEGATTR_COMB    : (False, 'comb'),
    SEGATTR_PERM    : (False, 'perm'),
    SEGATTR_BITNESS : (False, 'bitness'),
    SEGATTR_FLAGS   : (False, 'flags'),
    SEGATTR_SEL     : (False, 'sel'),
    SEGATTR_ES      : (False, 0),
    SEGATTR_CS      : (False, 1),
    SEGATTR_SS      : (False, 2),
    SEGATTR_DS      : (False, 3),
    SEGATTR_FS      : (False, 4),
    SEGATTR_GS      : (False, 5),
    SEGATTR_TYPE    : (False, 'type'),
    SEGATTR_COLOR   : (False, 'color'),
}

# Valid segment flags
SFL_COMORG   = 0x01       # IDP dependent field (IBM PC: if set, ORG directive is not comme
SFL_OBOK     = 0x02       # orgbase is present? (IDP dependent field)
SFL_HIDDEN   = 0x04       # is the segment hidden?
SFL_DEBUG    = 0x08       # is the segment created for the debugger?
SFL_LOADER   = 0x10       # is the segment created by the loader?
SFL_HIDETYPE = 0x20       # hide segment type (do not print it in the listing)
```

```
def move_segm(ea, to, flags):
    """
    Move a segment to a new address
    This function moves all information to the new address
    It fixes up address sensitive information in the kernel
    The total effect is equal to reloading the segment to the target address

    @param ea: any address within the segment to move
    @param to: new segment start address
    @param flags: combination MFS_... constants

    @returns: MOVE_SEGM_... error code
    """
    seg = ida_segment.getseg(ea)
    if not seg:
        return MOVE_SEGM_PARAM
    return ida_segment.move_segm(seg, to, flags)


MSF_SILENT    = 0x0001    # don't display a "please wait" box on the screen
MSF_NOFIX     = 0x0002    # don't call the loader to fix relocations
MSF_LDKEEP    = 0x0004    # keep the loader in the memory (optimization)
MSF_FIXONCE   = 0x0008    # valid for rebase_program(): call loader only once

MOVE_SEGM_OK          =  0     # all ok
MOVE_SEGM_PARAM       = -1     # The specified segment does not exist
MOVE_SEGM_ROOM        = -2     # Not enough free room at the target address
MOVE_SEGM_IDP         = -3     # IDP module forbids moving the segment
MOVE_SEGM_CHUNK       = -4     # Too many chunks are defined, can't move
MOVE_SEGM_LOADER      = -5     # The segment has been moved but the loader complained
MOVE_SEGM_ODD         = -6     # Can't move segments by an odd number of bytes
MOVE_SEGM_ORPHAN      = -7,    # Orphan bytes hinder segment movement
MOVE_SEGM_DEBUG       = -8,    # Debugger segments cannot be moved
MOVE_SEGM_SOURCEFILES = -9,    # Source files ranges of addresses hinder segment movement
MOVE_SEGM_MAPPING     = -10,   # Memory mapping ranges of addresses hinder segment movement
MOVE_SEGM_INVAL       = -11,   # Invalid argument (delta/target does not fit the address sp


rebase_program = ida_segment.rebase_program
set_storage_type = ida_bytes.change_storage_type


STT_VA = 0  # regular storage: virtual arrays, an explicit flag for each byte
STT_MM = 1  # memory map: sparse storage. useful for huge objects
```

```
#---------------------------------------------------------------------------
#                   C R O S S   R E F E R E N C E S
#---------------------------------------------------------------------------
#      Flow types (combine with XREF_USER!):
fl_CF   = 16              # Call Far
fl_CN   = 17              # Call Near
fl_JF   = 18              # jumpto Far
fl_JN   = 19              # jumpto Near
fl_F    = 21              # Ordinary flow

XREF_USER = 32            # All user-specified xref types
                          # must be combined with this bit


# Mark exec flow 'from' 'to'
add_cref = ida_xref.add_cref
del_cref = ida_xref.del_cref


# The following functions include the ordinary flows:
# (the ordinary flow references are returned first)
get_first_cref_from = ida_xref.get_first_cref_from
get_next_cref_from = ida_xref.get_next_cref_from
get_first_cref_to = ida_xref.get_first_cref_to
get_next_cref_to = ida_xref.get_next_cref_to


# The following functions don't take into account the ordinary flows:
get_first_fcref_from = ida_xref.get_first_fcref_from
get_next_fcref_from = ida_xref.get_next_fcref_from
get_first_fcref_to = ida_xref.get_first_fcref_to
get_next_fcref_to = ida_xref.get_next_fcref_to


# Data reference types (combine with XREF_USER!):
dr_O    = ida_xref.dr_O  # Offset
dr_W    = ida_xref.dr_W  # Write
dr_R    = ida_xref.dr_R  # Read
dr_T    = ida_xref.dr_T  # Text (names in manual operands)
dr_I    = ida_xref.dr_I  # Informational

add_dref = ida_xref.add_dref
del_dref = ida_xref.del_dref
get_first_dref_from = ida_xref.get_first_dref_from
```

```
get_next_dref_from = ida_xref.get_next_dref_from
get_first_dref_to = ida_xref.get_first_dref_to
get_next_dref_to = ida_xref.get_next_dref_to


def get_xref_type():
    """
    Return type of the last xref obtained by
    [RD]first/next[BO] functions.

    @return: constants fl_* or dr_*
    """
    raise DeprecatedIDCError("use XrefsFrom() XrefsTo() from idautils instead.")


#--------------------------------------------------------------------------
#                           F I L E   I / O
#--------------------------------------------------------------------------
def fopen(f, mode):
    raise DeprecatedIDCError("fopen() deprecated. Use Python file objects instead.")

def fclose(handle):
    raise DeprecatedIDCError("fclose() deprecated. Use Python file objects instead.")

def filelength(handle):
    raise DeprecatedIDCError("filelength() deprecated. Use Python file objects instead.")

def fseek(handle, offset, origin):
    raise DeprecatedIDCError("fseek() deprecated. Use Python file objects instead.")

def ftell(handle):
    raise DeprecatedIDCError("ftell() deprecated. Use Python file objects instead.")


def LoadFile(filepath, pos, ea, size):
    """
    Load file into IDA database

    @param filepath: path to input file
    @param pos: position in the file
    @param ea: linear address to load
    @param size: number of bytes to load

    @return: 0 - error, 1 - ok
    """
    li = ida_diskio.open_linput(filepath, False)
```

```
    if li:
        retval = ida_loader.file2base(li, pos, ea, ea+size, False)
        ida_diskio.close_linput(li)
        return retval
    else:
        return 0

def loadfile(filepath, pos, ea, size): return LoadFile(filepath, pos, ea, size)


def SaveFile(filepath, pos, ea, size):
    """
    Save from IDA database to file

    @param filepath: path to output file
    @param pos: position in the file
    @param ea: linear address to save from
    @param size: number of bytes to save

    @return: 0 - error, 1 - ok
    """
    if ( os.path.isfile(filepath) ):
        of = ida_diskio.fopenM(filepath)
    else:
        of = ida_diskio.fopenWB(filepath)


    if of:
        retval = ida_loader.base2file(of, pos, ea, ea+size)
        ida_diskio.eclose(of)
        return retval
    else:
        return 0

def savefile(filepath, pos, ea, size): return SaveFile(filepath, pos, ea, size)


def fgetc(handle):
    raise DeprecatedIDCError("fgetc() deprecated. Use Python file objects instead.")

def fputc(byte, handle):
    raise DeprecatedIDCError("fputc() deprecated. Use Python file objects instead.")

def fprintf(handle, format, *args):
    raise DeprecatedIDCError("fprintf() deprecated. Use Python file objects instead.")
```

```
def readshort(handle, mostfirst):
    raise DeprecatedIDCError("readshort() deprecated. Use Python file objects instead.")

def readlong(handle, mostfirst):
    raise DeprecatedIDCError("readlong() deprecated. Use Python file objects instead.")

def writeshort(handle, word, mostfirst):
    raise DeprecatedIDCError("writeshort() deprecated. Use Python file objects instead.")

def writelong(handle, dword, mostfirst):
    raise DeprecatedIDCError("writelong() deprecated. Use Python file objects instead.")

def readstr(handle):
    raise DeprecatedIDCError("readstr() deprecated. Use Python file objects instead.")

def writestr(handle, s):
    raise DeprecatedIDCError("writestr() deprecated. Use Python file objects instead.")

# ----------------------------------------------------------------------------
#                           F U N C T I O N S
# ----------------------------------------------------------------------------

add_func = ida_funcs.add_func
del_func = ida_funcs.del_func
set_func_end = ida_funcs.set_func_end


def get_next_func(ea):
    """
    Find next function

    @param ea: any address belonging to the function

    @return:        BADADDR - no more functions
            otherwise returns the next function start address
    """
    func = ida_funcs.get_next_func(ea)

    if not func:
        return BADADDR
    else:
        return func.start_ea


def get_prev_func(ea):
```

```
    """
    Find previous function

    @param ea: any address belonging to the function

    @return: BADADDR - no more functions
             otherwise returns the previous function start address
    """
    func = ida_funcs.get_prev_func(ea)

    if not func:
        return BADADDR
    else:
        return func.start_ea


def get_func_attr(ea, attr):
    """
    Get a function attribute

    @param ea: any address belonging to the function
    @param attr: one of FUNCATTR_... constants

    @return: BADADDR - error otherwise returns the attribute value
    """
    func = ida_funcs.get_func(ea)

    return _IDC_GetAttr(func, _FUNCATTRMAP, attr) if func else BADADDR


def set_func_attr(ea, attr, value):
    """
    Set a function attribute

    @param ea: any address belonging to the function
    @param attr: one of FUNCATTR_... constants
    @param value: new value of the attribute

    @return: 1-ok, 0-failed
    """
    func = ida_funcs.get_func(ea)

    if func:
        _IDC_SetAttr(func, _FUNCATTRMAP, attr, value)
        return ida_funcs.update_func(func)
    return 0
```

```
FUNCATTR_START   =  0     # readonly: function start address
FUNCATTR_END     =  4     # readonly: function end address
FUNCATTR_FLAGS   =  8     # function flags
FUNCATTR_FRAME   = 16     # readonly: function frame id
FUNCATTR_FRSIZE  = 20     # readonly: size of local variables
FUNCATTR_FRREGS  = 24     # readonly: size of saved registers area
FUNCATTR_ARGSIZE = 28     # readonly: number of bytes purged from the stack
FUNCATTR_FPD     = 32     # frame pointer delta
FUNCATTR_COLOR   = 36     # function color code
FUNCATTR_OWNER   = 16     # readonly: chunk owner (valid only for tail chunks)
FUNCATTR_REFQTY  = 20     # readonly: number of chunk parents (valid only for tail chunks)

# Redefining the constants for ea64
if __EA64__:
    FUNCATTR_START   =  0
    FUNCATTR_END     =  8
    FUNCATTR_FLAGS   = 16
    FUNCATTR_FRAME   = 24
    FUNCATTR_FRSIZE  = 32
    FUNCATTR_FRREGS  = 40
    FUNCATTR_ARGSIZE = 48
    FUNCATTR_FPD     = 56
    FUNCATTR_COLOR   = 64
    FUNCATTR_OWNER   = 24
    FUNCATTR_REFQTY  = 32

_FUNCATTRMAP = {
    FUNCATTR_START   : (True, 'start_ea'),
    FUNCATTR_END     : (True, 'end_ea'),
    FUNCATTR_FLAGS   : (False, 'flags'),
    FUNCATTR_FRAME   : (True, 'frame'),
    FUNCATTR_FRSIZE  : (True, 'frsize'),
    FUNCATTR_FRREGS  : (True, 'frregs'),
    FUNCATTR_ARGSIZE : (True, 'argsize'),
    FUNCATTR_FPD     : (False, 'fpd'),
    FUNCATTR_COLOR   : (False, 'color'),
    FUNCATTR_OWNER   : (True, 'owner'),
    FUNCATTR_REFQTY  : (True, 'refqty')
}


def get_func_flags(ea):
    """
    Retrieve function flags
```

```
    @param ea: any address belonging to the function

    @return: -1 - function doesn't exist otherwise returns the flags
    """
    func = ida_funcs.get_func(ea)

    if not func:
        return -1
    else:
        return func.flags


if ida_idaapi.uses_swig_builtins:
    _scope = ida_funcs.func_t
else:
    _scope = ida_funcs

FUNC_NORET         = _scope.FUNC_NORET         # function doesn't return
FUNC_FAR           = _scope.FUNC_FAR           # far function
FUNC_LIB           = _scope.FUNC_LIB           # library function
FUNC_STATIC        = _scope.FUNC_STATICDEF     # static function
FUNC_FRAME         = _scope.FUNC_FRAME         # function uses frame pointer (BP)
FUNC_USERFAR       = _scope.FUNC_USERFAR       # user has specified far-ness
                                               # of the function
FUNC_HIDDEN        = _scope.FUNC_HIDDEN        # a hidden function
FUNC_THUNK         = _scope.FUNC_THUNK         # thunk (jump) function
FUNC_BOTTOMBP      = _scope.FUNC_BOTTOMBP      # BP points to the bottom of the stack frame
FUNC_NORET_PENDING = _scope.FUNC_NORET_PENDING # Function 'non-return' analysis
                                               # must be performed. This flag is
                                               # verified upon func_does_return()
FUNC_SP_READY      = _scope.FUNC_SP_READY      # SP-analysis has been performed
                                               # If this flag is on, the stack
                                               # change points should not be not
                                               # modified anymore. Currently this
                                               # analysis is performed only for PC
FUNC_PURGED_OK     = _scope.FUNC_PURGED_OK     # 'argsize' field has been validated.
                                               # If this bit is clear and 'argsize'
                                               # is 0, then we do not known the real
                                               # number of bytes removed from
                                               # the stack. This bit is handled
                                               # by the processor module.
FUNC_TAIL          = _scope.FUNC_TAIL          # This is a function tail.
                                               # Other bits must be clear
                                               # (except FUNC_HIDDEN)
FUNC_LUMINA        = _scope.FUNC_LUMINA        # Function info is provided by Lumina.
```

```
FUNC_OUTLINE       = _scope.FUNC_OUTLINE       # Outlined code, not a real function.


def set_func_flags(ea, flags):
    """
    Change function flags

    @param ea: any address belonging to the function
    @param flags: see get_func_flags() for explanations

    @return: !=0 - ok
    """
    func = ida_funcs.get_func(ea)

    if not func:
        return 0
    else:
        func.flags = flags
        ida_funcs.update_func(func)
        return 1


def get_func_name(ea):
    """
    Retrieve function name

    @param ea: any address belonging to the function

    @return: null string - function doesn't exist
            otherwise returns function name
    """
    name = ida_funcs.get_func_name(ea)

    if not name:
        return ""
    else:
        return name


def get_func_cmt(ea, repeatable):
    """
    Retrieve function comment

    @param ea: any address belonging to the function
    @param repeatable: 1: get repeatable comment
            0: get regular comment
```

```
    @return: function comment string
    """
    func = ida_funcs.get_func(ea)

    if not func:
        return ""
    else:
        comment = ida_funcs.get_func_cmt(func, repeatable)

        if not comment:
            return ""
        else:
            return comment


def set_func_cmt(ea, cmt, repeatable):
    """
    Set function comment

    @param ea: any address belonging to the function
    @param cmt: a function comment line
    @param repeatable: 1: get repeatable comment
            0: get regular comment
    """
    func = ida_funcs.get_func(ea)

    if not func:
        return None
    else:
        return ida_funcs.set_func_cmt(func, cmt, repeatable)


def choose_func(title):
    """
    Ask the user to select a function

    Arguments:

    @param title: title of the dialog box

    @return: -1 - user refused to select a function
             otherwise returns the selected function start address
    """
    f = ida_kernwin.choose_func(title, ida_idaapi.BADADDR)
    return BADADDR if f is None else f.start_ea
```

```
def get_func_off_str(ea):
    """
    Convert address to 'funcname+offset' string

    @param ea: address to convert

    @return: if the address belongs to a function then return a string
             formed as 'name+offset' where 'name' is a function name
             'offset' is offset within the function else return null string
    """

    flags = ida_name.GNCN_NOCOLOR | ida_name.GNCN_REQFUNC
    return ida_name.get_nice_colored_name(ea, flags)


def find_func_end(ea):
    """
    Determine a new function boundaries

    @param ea: starting address of a new function

    @return: 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 function
    """
    func = ida_funcs.func_t(ea)

    res = ida_funcs.find_func_bounds(func, ida_funcs.FIND_FUNC_DEFINE)

    if res == ida_funcs.FIND_FUNC_UNDEF:
        return BADADDR
    else:
        return func.end_ea


def get_frame_id(ea):
    """
    Get ID of function frame structure

    @param ea: any address belonging to the function

    @return: ID of function frame or None In order to access stack variables
             you need to use structure member manipulaion functions with the
             obtained ID.
```

```
    """
    frame = ida_frame.get_frame(ea)

    if frame:
        return frame.id
    else:
        return None


def get_frame_lvar_size(ea):
    """
    Get size of local variables in function frame

    @param ea: any address belonging to the function

    @return: Size of local variables in bytes.
             If the function doesn't have a frame, return 0
             If the function does't exist, return None
    """
    return get_func_attr(ea, FUNCATTR_FRSIZE)


def get_frame_regs_size(ea):
    """
    Get size of saved registers in function frame

    @param ea: any address belonging to the function

    @return: Size of saved registers in bytes.
             If the function doesn't have a frame, return 0
             This value is used as offset for BP (if FUNC_FRAME is set)
             If the function does't exist, return None
    """
    return get_func_attr(ea, FUNCATTR_FRREGS)


def get_frame_args_size(ea):
    """
    Get size of arguments in function frame which are purged upon return

    @param ea: any address belonging to the function

    @return: Size of function arguments in bytes.
             If the function doesn't have a frame, return 0
             If the function does't exist, return -1
    """
```

```
    return get_func_attr(ea, FUNCATTR_ARGSIZE)


def get_frame_size(ea):
    """
    Get full size of function frame

    @param ea: any address belonging to the function
    @returns: Size of function frame in bytes.
                This function takes into account size of local
                variables + size of saved registers + size of
                return address + size of function arguments
                If the function doesn't have a frame, return size of
                function return address in the stack.
                If the function does't exist, return 0
    """
    func = ida_funcs.get_func(ea)

    if not func:
        return 0
    else:
        return ida_frame.get_frame_size(func)


def set_frame_size(ea, lvsize, frregs, argsize):
    """
    Make function frame

    @param ea: any address belonging to the function
    @param lvsize: size of function local variables
    @param frregs: size of saved registers
    @param argsize: size of function arguments

    @return: ID of function frame or -1
             If the function did not have a frame, the frame
             will be created. Otherwise the frame will be modified
    """
    func = ida_funcs.get_func(ea)
    if func is None:
        return -1

    frameid = ida_frame.add_frame(func, lvsize, frregs, argsize)

    if not frameid:
        if not ida_frame.set_frame_size(func, lvsize, frregs, argsize):
            return -1
```

```
    return func.frame


def get_spd(ea):
    """
    Get current delta for the stack pointer

    @param ea: end address of the instruction
               i.e.the last address of the instruction+1

    @return: The difference between the original SP upon
             entering the function and SP for the specified address
    """
    func = ida_funcs.get_func(ea)

    if not func:
        return None

    return ida_frame.get_spd(func, ea)


def get_sp_delta(ea):
    """
    Get modification of SP made by the instruction

    @param ea: end address of the instruction
               i.e.the last address of the instruction+1

    @return: Get modification of SP made at the specified location
             If the specified location doesn't contain a SP change point, return 0
             Otherwise return delta of SP modification
    """
    func = ida_funcs.get_func(ea)

    if not func:
        return None

    return ida_frame.get_sp_delta(func, ea)


# ----------------------------------------------------------------------------
#                               S T A C K
# ----------------------------------------------------------------------------

def add_auto_stkpnt(func_ea, ea, delta):
```

```
    """
    Add automatical SP register change point
    @param func_ea: function start
    @param ea: linear address where SP changes
               usually this is the end of the instruction which
               modifies the stack pointer (insn.ea+insn.size)
    @param delta: difference between old and new values of SP
    @return: 1-ok, 0-failed
    """
    pfn = ida_funcs.get_func(func_ea)
    if not pfn:
        return 0
    return ida_frame.add_auto_stkpnt(pfn, ea, delta)

add_user_stkpnt = ida_frame.add_user_stkpnt

def del_stkpnt(func_ea, ea):
    """
    Delete SP register change point

    @param func_ea: function start
    @param ea: linear address
    @return: 1-ok, 0-failed
    """
    pfn = ida_funcs.get_func(func_ea)
    if not pfn:
        return 0
    return ida_frame.del_stkpnt(pfn, ea)

def get_min_spd_ea(func_ea):
    """
    Return the address with the minimal spd (stack pointer delta)
    If there are no SP change points, then return BADADDR.

    @param func_ea: function start
    @return: BADDADDR - no such function
    """
    pfn = ida_funcs.get_func(func_ea)
    if not pfn:
        return BADADDR
    return ida_frame.get_min_spd_ea(pfn)

recalc_spd = ida_frame.recalc_spd
```

```
# ----------------------------------------------------------------------------
#                       E N T R Y   P O I N T S
# ----------------------------------------------------------------------------
get_entry_qty = ida_entry.get_entry_qty
add_entry = ida_entry.add_entry
get_entry_ordinal = ida_entry.get_entry_ordinal
get_entry = ida_entry.get_entry
get_entry_name = ida_entry.get_entry_name
rename_entry = ida_entry.rename_entry


# ----------------------------------------------------------------------------
#                               F I X U P S
# ----------------------------------------------------------------------------
get_next_fixup_ea = ida_fixup.get_next_fixup_ea
get_prev_fixup_ea = ida_fixup.get_prev_fixup_ea


def get_fixup_target_type(ea):
    """
    Get fixup target type

    @param ea: address to get information about

    @return: 0 - no fixup at the specified address
                 otherwise returns fixup type
    """
    fd = ida_fixup.fixup_data_t()

    if not fd.get(ea):
        return 0

    return fd.get_type()


FIXUP_OFF8      = 13      # 8-bit offset.
FIXUP_OFF16     = 1       # 16-bit offset.
FIXUP_SEG16     = 2       # 16-bit base--logical segment base (selector).
FIXUP_PTR32     = 3       # 32-bit long pointer (16-bit base:16-bit
                          # offset).
FIXUP_OFF32     = 4       # 32-bit offset.
FIXUP_PTR48     = 5       # 48-bit pointer (16-bit base:32-bit offset).
FIXUP_HI8       = 6       # high  8 bits of 16bit offset
FIXUP_HI16      = 7       # high 16 bits of 32bit offset
```

```
FIXUP_LOW8      = 8       # low   8 bits of 16bit offset
FIXUP_LOW16     = 9       # low  16 bits of 32bit offset
FIXUP_OFF64     = 12      # 64-bit offset
FIXUP_CUSTOM    = 0x8000  # fixups with this bit are processed by
                          # processor module/plugin

def get_fixup_target_flags(ea):
    """
    Get fixup target flags

    @param ea: address to get information about

    @return: 0 - no fixup at the specified address
                 otherwise returns fixup target flags
    """
    fd = ida_fixup.fixup_data_t()

    if not fd.get(ea):
        return 0

    return fd.get_flags()


FIXUPF_REL       = 0x1  # fixup is relative to the linear address
FIXUPF_EXTDEF    = 0x2  # target is a location (otherwise - segment)
FIXUPF_UNUSED    = 0x4  # fixup is ignored by IDA
FIXUPF_CREATED   = 0x8  # fixup was not present in the input file


def get_fixup_target_sel(ea):
    """
    Get fixup target selector

    @param ea: address to get information about

    @return: BADSEL - no fixup at the specified address
                      otherwise returns fixup target selector
    """
    fd = ida_fixup.fixup_data_t()

    if not fd.get(ea):
        return BADSEL

    return fd.sel
```

```
def get_fixup_target_off(ea):
    """
    Get fixup target offset

    @param ea: address to get information about

    @return: BADADDR - no fixup at the specified address
                       otherwise returns fixup target offset
    """
    fd = ida_fixup.fixup_data_t()

    if not fd.get(ea):
        return BADADDR

    return fd.off


def get_fixup_target_dis(ea):
    """
    Get fixup target displacement

    @param ea: address to get information about

    @return: 0 - no fixup at the specified address
                 otherwise returns fixup target displacement
    """
    fd = ida_fixup.fixup_data_t()

    if not fd.get(ea):
        return 0

    return fd.displacement


def set_fixup(ea, fixuptype, fixupflags, targetsel, targetoff, displ):
    """
    Set fixup information

    @param ea: address to set fixup information about
    @param fixuptype:  fixup type. see get_fixup_target_type()
                       for possible fixup types.
    @param fixupflags: fixup flags. see get_fixup_target_flags()
                       for possible fixup types.
    @param targetsel:  target selector
    @param targetoff:  target offset
    @param displ:      displacement
```

```
    @return:        none
    """
    fd = ida_fixup.fixup_data_t(fixuptype, fixupflags)
    fd.sel = targetsel
    fd.off = targetoff
    fd.displacement = displ

    fd.set(ea)


del_fixup = ida_fixup.del_fixup


#--------------------------------------------------------------------------
#                   M A R K E D   P O S I T I O N S
#--------------------------------------------------------------------------

put_bookmark = ida_idc.mark_position
get_bookmark = ida_idc.get_marked_pos
get_bookmark_desc = ida_idc.get_mark_comment


# ----------------------------------------------------------------------------
#                           S T R U C T U R E S
# ----------------------------------------------------------------------------

get_struc_qty = ida_struct.get_struc_qty
get_first_struc_idx = ida_struct.get_first_struc_idx
get_last_struc_idx = ida_struct.get_last_struc_idx
get_next_struc_idx = ida_struct.get_next_struc_idx
get_prev_struc_idx = ida_struct.get_prev_struc_idx
get_struc_idx = ida_struct.get_struc_idx
get_struc_by_idx = ida_struct.get_struc_by_idx
get_struc_id = ida_struct.get_struc_id
get_struc_name = ida_struct.get_struc_name
get_struc_cmt = ida_struct.get_struc_cmt
get_struc_size = ida_struct.get_struc_size


def get_member_qty(sid):
    """
    Get number of members of a structure

    @param sid: structure type ID
```

```
    @return: -1 if bad structure type ID is passed otherwise
             returns number of members.

    @note: Union members are, in IDA's internals, located
           at subsequent byte offsets: member 0 -> offset 0x0,
           member 1 -> offset 0x1, etc...
    """
    s = ida_struct.get_struc(sid)
    return -1 if not s else s.memqty


def get_member_id(sid, member_offset):
    """
    @param sid: structure type ID
    @param member_offset:. The offset can be
    any offset in the member. For example,
    is a member is 4 bytes long and starts
    at offset 2, then 2,3,4,5 denote
    the same structure member.

    @return: -1 if bad structure type ID is passed or there is
    no member at the specified offset.
    otherwise returns the member id.
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return -1

    m = ida_struct.get_member(s, member_offset)
    if not m:
        return -1

    return m.id


def get_prev_offset(sid, offset):
    """
    Get previous offset in a structure

    @param sid: structure type ID
    @param offset: current offset

    @return: -1 if bad structure type ID is passed,
             ida_idaapi.BADADDR if no (more) offsets in the structure,
             otherwise returns previous offset in a structure.
```

```
    @note: IDA allows 'holes' between members of a
           structure. It treats these 'holes'
           as unnamed arrays of bytes.
           This function returns a member offset or a hole offset.
           It will return size of the structure if input
           'offset' is bigger than the structure size.

    @note: Union members are, in IDA's internals, located
           at subsequent byte offsets: member 0 -> offset 0x0,
           member 1 -> offset 0x1, etc...
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return -1

    return ida_struct.get_struc_prev_offset(s, offset)


def get_next_offset(sid, offset):
    """
    Get next offset in a structure

    @param sid:     structure type ID
    @param offset: current offset

    @return: -1 if bad structure type ID is passed,
             ida_idaapi.BADADDR if no (more) offsets in the structure,
             otherwise returns next offset in a structure.

    @note: IDA allows 'holes' between members of a
           structure. It treats these 'holes'
           as unnamed arrays of bytes.
           This function returns a member offset or a hole offset.
           It will return size of the structure if input
           'offset' belongs to the last member of the structure.

    @note: Union members are, in IDA's internals, located
           at subsequent byte offsets: member 0 -> offset 0x0,
           member 1 -> offset 0x1, etc...
    """
    s = ida_struct.get_struc(sid)
    return -1 if not s else ida_struct.get_struc_next_offset(s, offset)


def get_first_member(sid):
    """
```

```
    Get offset of the first member of a structure

    @param sid: structure type ID

    @return: -1 if bad structure type ID is passed,
             ida_idaapi.BADADDR if structure has no members,
             otherwise returns offset of the first member.

    @note: IDA allows 'holes' between members of a
           structure. It treats these 'holes'
           as unnamed arrays of bytes.

    @note: Union members are, in IDA's internals, located
           at subsequent byte offsets: member 0 -> offset 0x0,
           member 1 -> offset 0x1, etc...
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return -1

    return ida_struct.get_struc_first_offset(s)


def get_last_member(sid):
    """
    Get offset of the last member of a structure

    @param sid: structure type ID

    @return: -1 if bad structure type ID is passed,
             ida_idaapi.BADADDR if structure has no members,
             otherwise returns offset of the last member.

    @note: IDA allows 'holes' between members of a
          structure. It treats these 'holes'
          as unnamed arrays of bytes.

    @note: Union members are, in IDA's internals, located
           at subsequent byte offsets: member 0 -> offset 0x0,
           member 1 -> offset 0x1, etc...
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return -1

    return ida_struct.get_struc_last_offset(s)
```

```
def get_member_offset(sid, member_name):
    """
    Get offset of a member of a structure by the member name

    @param sid: structure type ID
    @param member_name: name of structure member

    @return: -1 if bad structure type ID is passed
             or no such member in the structure
             otherwise returns offset of the specified member.

    @note: Union members are, in IDA's internals, located
           at subsequent byte offsets: member 0 -> offset 0x0,
           member 1 -> offset 0x1, etc...
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return -1

    m = ida_struct.get_member_by_name(s, member_name)
    if not m:
        return -1

    return m.get_soff()


def get_member_name(sid, member_offset):
    """
    Get name of a member of a structure

    @param sid: structure type ID
    @param member_offset: member offset. The offset can be
                          any offset in the member. For example,
                          is a member is 4 bytes long and starts
                          at offset 2, then 2,3,4,5 denote
                          the same structure member.

    @return: None if bad structure type ID is passed
             or no such member in the structure
             otherwise returns name of the specified member.
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return None
```

```
    m = ida_struct.get_member(s, member_offset)
    if not m:
        return None

    return ida_struct.get_member_name(m.id)


def get_member_cmt(sid, member_offset, repeatable):
    """
    Get comment of a member

    @param sid: structure type ID
    @param member_offset: member offset. The offset can be
                          any offset in the member. For example,
                          is a member is 4 bytes long and starts
                          at offset 2, then 2,3,4,5 denote
                          the same structure member.
    @param repeatable: 1: get repeatable comment
                       0: get regular comment

    @return: None if bad structure type ID is passed
             or no such member in the structure
             otherwise returns comment of the specified member.
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return None

    m = ida_struct.get_member(s, member_offset)
    if not m:
        return None

    return ida_struct.get_member_cmt(m.id, repeatable)


def get_member_size(sid, member_offset):
    """
    Get size of a member

    @param sid: structure type ID
    @param member_offset: member offset. The offset can be
                          any offset in the member. For example,
                          is a member is 4 bytes long and starts
                          at offset 2, then 2,3,4,5 denote
                          the same structure member.
```

```
    @return: None if bad structure type ID is passed,
             or no such member in the structure
             otherwise returns size of the specified
             member in bytes.
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return None

    m = ida_struct.get_member(s, member_offset)
    if not m:
        return None

    return ida_struct.get_member_size(m)


def get_member_flag(sid, member_offset):
    """
    Get type of a member

    @param sid: structure type ID
    @param member_offset: member offset. The offset can be
                          any offset in the member. For example,
                          is a member is 4 bytes long and starts
                          at offset 2, then 2,3,4,5 denote
                          the same structure member.

    @return: -1 if bad structure type ID is passed
             or no such member in the structure
             otherwise returns type of the member, see bit
             definitions above. If the member type is a structure
             then function GetMemberStrid() should be used to
             get the structure type id.
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return -1

    m = ida_struct.get_member(s, member_offset)
    return -1 if not m else m.flag


def get_member_strid(sid, member_offset):
    """
    Get structure id of a member
```

```
    @param sid: structure type ID
    @param member_offset: member offset. The offset can be
                          any offset in the member. For example,
                          is a member is 4 bytes long and starts
                          at offset 2, then 2,3,4,5 denote
                          the same structure member.
    @return: -1 if bad structure type ID is passed
             or no such member in the structure
             otherwise returns structure id of the member.
             If the current member is not a structure, returns -1.
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return -1

    m = ida_struct.get_member(s, member_offset)
    if not m:
        return -1

    cs = ida_struct.get_sptr(m)
    if cs:
        return cs.id
    else:
        return -1


def is_union(sid):
    """
    Is a structure a union?

    @param sid: structure type ID

    @return: 1: yes, this is a union id
             0: no

    @note: Unions are a special kind of structures
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return 0

    return s.is_union()


def add_struc(index, name, is_union):
```

```
    """
    Define a new structure type

    @param index: index of new structure type
                  If another structure has the specified index,
                  then index of that structure and all other
                  structures will be incremented, freeing the specifed
                  index. If index is == -1, then the biggest index
                  number will be used.
                  See get_first_struc_idx() for the explanation of
                  structure indices and IDs.
    @param name: name of the new structure type.
    @param is_union: 0: structure
                     1: union

    @return: -1 if can't define structure type because of
             bad structure name: the name is ill-formed or is
             already used in the program.
             otherwise returns ID of the new structure type
    """
    if index == -1:
        index = BADADDR

    return ida_struct.add_struc(index, name, is_union)


def del_struc(sid):
    """
    Delete a structure type

    @param sid: structure type ID

    @return: 0 if bad structure type ID is passed
             1 otherwise the structure type is deleted. All data
             and other structure types referencing to the
             deleted structure type will be displayed as array
             of bytes.
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return 0

    return ida_struct.del_struc(s)


def set_struc_idx(sid, index):
```

```
    """
    Change structure index

    @param sid: structure type ID
    @param index: new index of the structure

    @return: != 0 - ok

    @note: See get_first_struc_idx() for the explanation of
           structure indices and IDs.
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return 0

    return ida_struct.set_struc_idx(s, index)


set_struc_name = ida_struct.set_struc_name
set_struc_cmt = ida_struct.set_struc_cmt


def add_struc_member(sid, name, offset, flag, typeid, nbytes, target=-1, tdelta=0, reftype=
    """
    Add structure member

    @param sid: structure type ID
    @param name: name of the new member
    @param offset: offset of the new member
                   -1 means to add at the end of the structure
    @param flag: type of the new member. Should be one of
                 FF_BYTE..FF_PACKREAL (see above) combined with FF_DATA
    @param typeid: if is_struct(flag) then typeid specifies the structure id for the member
                   if is_off0(flag) then typeid specifies the offset base.
                   if is_strlit(flag) then typeid specifies the string type (STRTYPE_...).
                   if is_stroff(flag) then typeid specifies the structure id
                   if is_enum(flag) then typeid specifies the enum id
                   if is_custom(flags) then typeid specifies the dtid and fid: dtid|(fid<<1
                   Otherwise typeid should be -1.
    @param nbytes: number of bytes in the new member

    @param target: target address of the offset expr. You may specify it as
                   -1, ida will calculate it itself
    @param tdelta: offset target delta. usually 0
    @param reftype: see REF_... definitions
```

```
    @note: The remaining arguments are allowed only if is_off0(flag) and you want
           to specify a complex offset expression

    @return: 0 - ok, otherwise error code (one of STRUC_ERROR_*)

    """
    if is_off0(flag):
        return eval_idc('add_struc_member(%d, "%s", %d, %d, %d, %d, %d, %d, %d);' % (sid, io
                                                                                     target, tdelt
    else:
        return eval_idc('add_struc_member(%d, "%s", %d, %d, %d, %d);' % (sid, ida_kernwin.st


STRUC_ERROR_MEMBER_NAME    = -1 # already has member with this name (bad name)
STRUC_ERROR_MEMBER_OFFSET  = -2 # already has member at this offset
STRUC_ERROR_MEMBER_SIZE    = -3 # bad number of bytes or bad sizeof(type)
STRUC_ERROR_MEMBER_TINFO   = -4 # bad typeid parameter
STRUC_ERROR_MEMBER_STRUCT  = -5 # bad struct id (the 1st argument)
STRUC_ERROR_MEMBER_UNIVAR  = -6 # unions can't have variable sized members
STRUC_ERROR_MEMBER_VARLAST = -7 # variable sized member should be the last member in the st


def del_struc_member(sid, member_offset):
    """
    Delete structure member

    @param sid: structure type ID
    @param member_offset: offset of the member

    @return: != 0 - ok.

    @note: IDA allows 'holes' between members of a
           structure. It treats these 'holes'
           as unnamed arrays of bytes.
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return 0

    return ida_struct.del_struc_member(s, member_offset)


def set_member_name(sid, member_offset, name):
    """
    Change structure member name
```

```
    @param sid: structure type ID
    @param member_offset: offset of the member
    @param name: new name of the member

    @return: != 0 - ok.
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return 0

    return ida_struct.set_member_name(s, member_offset, name)


def set_member_type(sid, member_offset, flag, typeid, nitems, target=-1, tdelta=0, reftype=R
    """
    Change structure member type

    @param sid: structure type ID
    @param member_offset: offset of the member
    @param flag: new type of the member. Should be one of
                 FF_BYTE..FF_PACKREAL (see above) combined with FF_DATA
    @param typeid: if is_struct(flag) then typeid specifies the structure id for the member
                   if is_off0(flag) then typeid specifies the offset base.
                   if is_strlit(flag) then typeid specifies the string type (STRTYPE_...).
                   if is_stroff(flag) then typeid specifies the structure id
                   if is_enum(flag) then typeid specifies the enum id
                   if is_custom(flags) then typeid specifies the dtid and fid: dtid|(fid<<16
                   Otherwise typeid should be -1.
    @param nitems: number of items in the member

    @param target: target address of the offset expr. You may specify it as
                   -1, ida will calculate it itself
    @param tdelta: offset target delta. usually 0
    @param reftype: see REF_... definitions

    @note: The remaining arguments are allowed only if is_off0(flag) and you want
           to specify a complex offset expression

    @return: !=0 - ok.
    """
    if is_off0(flag):
        return eval_idc('set_member_type(%d, %d, %d, %d, %d, %d, %d, %d);' % (sid, member_of
                                                                              target, tdelta
    else:
        return eval_idc('set_member_type(%d, %d, %d, %d, %d);' % (sid, member_offset, flag,
```

```
def set_member_cmt(sid, member_offset, comment, repeatable):
    """
    Change structure member comment

    @param sid: structure type ID
    @param member_offset: offset of the member
    @param comment: new comment of the structure member
    @param repeatable: 1: change repeatable comment
                       0: change regular comment

    @return: != 0 - ok
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return 0

    m = ida_struct.get_member(s, member_offset)
    if not m:
        return 0

    return ida_struct.set_member_cmt(m, comment, repeatable)


def expand_struc(sid, offset, delta, recalc):
    """
    Expand or shrink a structure type
    @param id: structure type ID
    @param offset: offset in the structure
    @param delta: how many bytes to add or remove
    @param recalc: recalculate the locations where the structure
                               type is used
    @return: != 0 - ok
    """
    s = ida_struct.get_struc(sid)
    if not s:
        return 0

    return ida_struct.expand_struc(s, offset, delta, recalc)


def get_fchunk_attr(ea, attr):
    """
    Get a function chunk attribute

    @param ea: any address in the chunk
```

```
    @param attr: one of: FUNCATTR_START, FUNCATTR_END, FUNCATTR_OWNER, FUNCATTR_REFQTY

    @return: desired attribute or -1
    """
    func = ida_funcs.get_fchunk(ea)
    return _IDC_GetAttr(func, _FUNCATTRMAP, attr) if func else BADADDR


def set_fchunk_attr(ea, attr, value):
    """
    Set a function chunk attribute

    @param ea: any address in the chunk
    @param attr: only FUNCATTR_START, FUNCATTR_END, FUNCATTR_OWNER
    @param value: desired value

    @return: 0 if failed, 1 if success
    """
    if attr in [ FUNCATTR_START, FUNCATTR_END, FUNCATTR_OWNER ]:
        chunk = ida_funcs.get_fchunk(ea)
        if chunk:
            _IDC_SetAttr(chunk, _FUNCATTRMAP, attr, value)
            return ida_funcs.update_func(chunk)
    return 0


get_fchunk_referer = ida_funcs.get_fchunk_referer


def get_next_fchunk(ea):
    """
    Get next function chunk

    @param ea: any address

    @return:  the starting address of the next function chunk or BADADDR

    @note: This function enumerates all chunks of all functions in the database
    """
    func = ida_funcs.get_next_fchunk(ea)

    if func:
        return func.start_ea
    else:
        return BADADDR
```

```
def get_prev_fchunk(ea):
    """
    Get previous function chunk

    @param ea: any address

    @return: the starting address of the function chunk or BADADDR

    @note: This function enumerates all chunks of all functions in the database
    """
    func = ida_funcs.get_prev_fchunk(ea)

    if func:
        return func.start_ea
    else:
        return BADADDR


def append_func_tail(funcea, ea1, ea2):
    """
    Append a function chunk to the function

    @param funcea: any address in the function
    @param ea1: start of function tail
    @param ea2: end of function tail
    @return: 0 if failed, 1 if success

    @note: If a chunk exists at the specified addresses, it must have exactly
           the specified boundaries
    """
    func = ida_funcs.get_func(funcea)

    if not func:
        return 0
    else:
        return ida_funcs.append_func_tail(func, ea1, ea2)


def remove_fchunk(funcea, tailea):
    """
    Remove a function chunk from the function

    @param funcea: any address in the function
    @param tailea: any address in the function chunk to remove
```

```
    @return: 0 if failed, 1 if success
    """
    func = ida_funcs.get_func(funcea)

    if not func:
        return 0
    else:
        return ida_funcs.remove_func_tail(func, tailea)


def set_tail_owner(tailea, funcea):
    """
    Change the function chunk owner

    @param tailea: any address in the function chunk
    @param funcea: the starting address of the new owner

    @return: False if failed, True if success

    @note: The new owner must already have the chunk appended before the call
    """
    tail = ida_funcs.get_fchunk(tailea)

    if not tail:
        return False
    else:
        return ida_funcs.set_tail_owner(tail, funcea)


def first_func_chunk(funcea):
    """
    Get the first function chunk of the specified function

    @param funcea: any address in the function

    @return: the function entry point or BADADDR

    @note: This function returns the first (main) chunk of the specified function
    """
    func = ida_funcs.get_func(funcea)
    fci = ida_funcs.func_tail_iterator_t(func, funcea)
    if fci.main():
        return fci.chunk().start_ea
    else:
        return BADADDR
```

```
def next_func_chunk(funcea, tailea):
    """
    Get the next function chunk of the specified function

    @param funcea: any address in the function
    @param tailea: any address in the current chunk

    @return: the starting address of the next function chunk or BADADDR

    @note: This function returns the next chunk of the specified function
    """
    func = ida_funcs.get_func(funcea)
    fci = ida_funcs.func_tail_iterator_t(func, funcea)
    if not fci.main():
        return BADADDR

    # Iterate and try to find the current chunk
    found = False
    while True:
        if fci.chunk().start_ea <= tailea and \
           fci.chunk().end_ea > tailea:
            found = True
            break
        if not next(fci):
            break

    # Return the next chunk, if there is one
    if found and next(fci):
        return fci.chunk().start_ea
    else:
        return BADADDR


# ----------------------------------------------------------------------------
#                           E N U M S
# ----------------------------------------------------------------------------
get_enum_qty = ida_enum.get_enum_qty
getn_enum = ida_enum.getn_enum
get_enum_idx = ida_enum.get_enum_idx
get_enum = ida_enum.get_enum
get_enum_name = ida_enum.get_enum_name
get_enum_cmt = ida_enum.get_enum_cmt
get_enum_size = ida_enum.get_enum_size
get_enum_width = ida_enum.get_enum_width
get_enum_flag = ida_enum.get_enum_flag
```

```
get_enum_member_by_name = ida_enum.get_enum_member_by_name
get_enum_member_value = ida_enum.get_enum_member_value
get_enum_member_bmask = ida_enum.get_enum_member_bmask
get_enum_member_enum = ida_enum.get_enum_member_enum


def get_enum_member(enum_id, value, serial, bmask):
    """
    Get id of constant

    @param enum_id: id of enum
    @param value: value of constant
    @param serial: serial number of the constant in the
              enumeration. See op_enum() for details.
    @param bmask: bitmask of the constant
              ordinary enums accept only ida_enum.DEFMASK as a bitmask

    @return: id of constant or -1 if error
    """
    if bmask < 0:
        bmask &= BADADDR
    return ida_enum.get_enum_member(enum_id, value, serial, bmask)


get_first_bmask = ida_enum.get_first_bmask
get_last_bmask = ida_enum.get_last_bmask
get_next_bmask = ida_enum.get_next_bmask
get_prev_bmask = ida_enum.get_prev_bmask


def get_bmask_name(enum_id, bmask):
    """
    Get bitmask name (only for bitfields)

    @param enum_id: id of enum
    @param bmask: bitmask of the constant

    @return: name of bitmask or None
    """
    if bmask < 0:
        bmask &= BADADDR
    return ida_enum.get_bmask_name(enum_id, bmask)


def get_bmask_cmt(enum_id, bmask, repeatable):
    """
```

```
    Get bitmask comment (only for bitfields)

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

    @return: comment attached to bitmask or None
    """
    if bmask < 0:
        bmask &= BADADDR
    return ida_enum.get_bmask_cmt(enum_id, bmask, repeatable)


def set_bmask_name(enum_id, bmask, name):
    """
    Set bitmask name (only for bitfields)

    @param enum_id: id of enum
    @param bmask: bitmask of the constant
    @param name: name of bitmask

    @return: 1-ok, 0-failed
    """
    if bmask < 0:
        bmask &= BADADDR
    return ida_enum.set_bmask_name(enum_id, bmask, name)


def set_bmask_cmt(enum_id, bmask, cmt, repeatable):
    """
    Set bitmask comment (only for bitfields)

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

    @return: 1-ok, 0-failed
    """
    if bmask < 0:
        bmask &= BADADDR
    return ida_enum.set_bmask_cmt(enum_id, bmask, cmt, repeatable)


def get_first_enum_member(enum_id, bmask):
    """
```

```
    Get first constant in the enum

    @param enum_id: id of enum
    @param bmask: bitmask of the constant (ordinary enums accept only ida_enum.DEFMASK as a

    @return: value of constant or idaapi.BADNODE no constants are defined
             All constants are sorted by their values as unsigned longs.
    """
    if bmask < 0:
        bmask &= BADADDR
    return ida_enum.get_first_enum_member(enum_id, bmask)


def get_last_enum_member(enum_id, bmask):
    """
    Get last constant in the enum

    @param enum_id: id of enum
    @param bmask: bitmask of the constant (ordinary enums accept only ida_enum.DEFMASK as a

    @return: value of constant or idaapi.BADNODE no constants are defined
             All constants are sorted by their values
             as unsigned longs.
    """
    if bmask < 0:
        bmask &= BADADDR
    return ida_enum.get_last_enum_member(enum_id, bmask)


def get_next_enum_member(enum_id, value, bmask):
    """
    Get next constant in the enum

    @param enum_id: id of enum
    @param bmask: bitmask of the constant ordinary enums accept only ida_enum.DEFMASK as a l
    @param value: value of the current constant

    @return: value of a constant with value higher than the specified
             value. idaapi.BADNODE no such constants exist.
             All constants are sorted by their values as unsigned longs.
    """
    if bmask < 0:
        bmask &= BADADDR
    return ida_enum.get_next_enum_member(enum_id, value, bmask)
```

```
def get_prev_enum_member(enum_id, value, bmask):
    """
    Get prev constant in the enum

    @param enum_id: id of enum
    @param bmask  : bitmask of the constant
              ordinary enums accept only ida_enum.DEFMASK as a bitmask
    @param value: value of the current constant

    @return: value of a constant with value lower than the specified
        value. idaapi.BADNODE no such constants exist.
        All constants are sorted by their values as unsigned longs.
    """
    if bmask < 0:
        bmask &= BADADDR
    return ida_enum.get_prev_enum_member(enum_id, value, bmask)


def get_enum_member_name(const_id):
    """
    Get name of a constant

    @param const_id: id of const

    Returns: name of constant
    """
    name = ida_enum.get_enum_member_name(const_id)

    if not name:
        return ""
    else:
        return name


def get_enum_member_cmt(const_id, repeatable):
    """
    Get comment of a constant

    @param const_id: id of const
    @param repeatable: 0:get regular comment, 1:get repeatable comment

    @return: comment string
    """
    cmt = ida_enum.get_enum_member_cmt(const_id, repeatable)

    if not cmt:
```

```
        return ""
    else:
        return cmt


def add_enum(idx, name, flag):
    """
    Add a new enum type

    @param idx: serial number of the new enum.
            If another enum with the same serial number
            exists, then all enums with serial
            numbers >= the specified idx get their
            serial numbers incremented (in other words,
            the new enum is put in the middle of the list of enums).

            If idx >= get_enum_qty() or idx == idaapi.BADNODE
            then the new enum is created at the end of
            the list of enums.

    @param name: name of the enum.
    @param flag: flags for representation of numeric constants
                 in the definition of enum.

    @return: id of new enum or BADADDR
    """
    if idx < 0:
        idx = idx & SIZE_MAX
    return ida_enum.add_enum(idx, name, flag)


del_enum = ida_enum.del_enum
set_enum_idx = ida_enum.set_enum_idx
set_enum_name = ida_enum.set_enum_name
set_enum_cmt = ida_enum.set_enum_cmt
set_enum_flag = ida_enum.set_enum_flag
set_enum_bf = ida_enum.set_enum_bf
set_enum_width = ida_enum.set_enum_width
is_bf = ida_enum.is_bf


def add_enum_member(enum_id, name, value, bmask):
    """
    Add a member of enum - a symbolic constant

    @param enum_id: id of enum
```

```
    @param name: name of symbolic constant. Must be unique in the program.
    @param value: value of symbolic constant.
    @param bmask: bitmask of the constant
        ordinary enums accept only ida_enum.DEFMASK as a bitmask
        all bits set in value should be set in bmask too

    @return: 0-ok, otherwise error code (one of ENUM_MEMBER_ERROR_*)
    """
    if bmask < 0:
        bmask &= BADADDR
    return ida_enum.add_enum_member(enum_id, name, value, bmask)


ENUM_MEMBER_ERROR_NAME  = ida_enum.ENUM_MEMBER_ERROR_NAME  # already have member with this n
ENUM_MEMBER_ERROR_VALUE = ida_enum.ENUM_MEMBER_ERROR_VALUE # already have member with this v
ENUM_MEMBER_ERROR_ENUM  = ida_enum.ENUM_MEMBER_ERROR_ENUM  # bad enum id
ENUM_MEMBER_ERROR_MASK  = ida_enum.ENUM_MEMBER_ERROR_MASK  # bad bmask
ENUM_MEMBER_ERROR_ILLV  = ida_enum.ENUM_MEMBER_ERROR_ILLV  # bad bmask and value combinatio


def del_enum_member(enum_id, value, serial, bmask):
    """
    Delete a member of enum - a symbolic constant

    @param enum_id: id of enum
    @param value: value of symbolic constant.
    @param serial: serial number of the constant in the
        enumeration. See op_enum() for for details.
    @param bmask: bitmask of the constant ordinary enums accept
        only ida_enum.DEFMASK as a bitmask

    @return: 1-ok, 0-failed
    """
    if bmask < 0:
        bmask &= BADADDR
    return ida_enum.del_enum_member(enum_id, value, serial, bmask)


set_enum_member_name = ida_enum.set_enum_member_name
set_enum_member_cmt = ida_enum.set_enum_member_cmt

#--------------------------------------------------------------------------
#                       A R R A Y S  I N  I D C
#--------------------------------------------------------------------------

_IDC_ARRAY_PREFIX = "$ idc_array "
```

```
def __l2m1(v):
    """
    Long to minus 1: If the 'v' appears to be the
    'signed long' version of -1, then return -1.
    Otherwise, return 'v'.
    """
    if v == ida_netnode.BADNODE:
        return -1
    else:
        return v


def __m1tol(v):
    """
    Long -1 to BADNODE: If the 'v' appears to be the
    'signed long' version of -1, then return BADNODE.
    Otherwise, return 'v'.
    """
    if v == -1:
        return ida_netnode.BADNODE
    else:
        return v


AR_LONG = ida_netnode.atag
"""Array of longs"""

AR_STR = ida_netnode.stag
"""Array of strings"""


class __dummy_netnode(object):
    """
    Implements, in an "always failing" fashion, the
    netnode functions that are necessary for the
    array-related functions.

    The sole purpose of this singleton class is to
    serve as a placeholder for netnode-manipulating
    functions, that don't want to each have to perform
    checks on the existence of the netnode.
    (..in other words: it avoids a bunch of if/else's).

    See __GetArrayById() for more info.
    """
    def rename(self, *args): return 0
```

```
    def kill(self, *args): pass
    def index(self, *args): return -1
    def altset(self, *args): return 0
    def supset(self, *args): return 0
    def altval(self, *args): return 0
    def supval(self, *args): return 0
    def altdel(self, *args): return 0
    def supdel(self, *args): return 0
    def altfirst(self, *args): return -1
    def supfirst(self, *args): return -1
    def altlast(self, *args): return -1
    def suplast(self, *args): return -1
    def altnext(self, *args): return -1
    def supnext(self, *args): return -1
    def altprev(self, *args): return -1
    def supprev(self, *args): return -1
    def hashset(self, *args): return 0
    def hashval(self, *args): return 0
    def hashstr(self, *args): return 0
    def hashstr_buf(self, *args): return 0
    def hashset_idx(self, *args): return 0
    def hashset_buf(self, *args): return 0
    def hashval_long(self, *args): return 0
    def hashdel(self, *args): return 0
    def hashfirst(self, *args): return 0
    def hashnext(self, *args): return 0
    def hashprev(self, *args): return 0
    def hashlast(self, *args): return 0
__dummy_netnode.instance = __dummy_netnode()



def __GetArrayById(array_id):
    """
    Get an array, by its ID.

    This (internal) wrapper around 'idaapi.netnode(array_id)'
    will ensure a certain safety around the retrieval of
    arrays (by catching quite unexpect[ed|able] exceptions,
    and making sure we don't create & use `transient' netnodes).

    @param array_id: A positive, valid array ID.
    """
    try:
        node = ida_netnode.netnode(array_id)
        nodename = node.get_name()
```

```
        if nodename is None or not nodename.startswith(_IDC_ARRAY_PREFIX):
            return __dummy_netnode.instance
        else:
            return node
    except TypeError:
        return __dummy_netnode.instance
    except NotImplementedError:
        return __dummy_netnode.instance


def create_array(name):
    """
    Create array.

    @param name: The array name.

    @return: -1 in case of failure, a valid array_id otherwise.
    """
    node = ida_netnode.netnode()
    res  = node.create(_IDC_ARRAY_PREFIX + name)
    if res == False:
        return -1
    else:
        return node.index()


def get_array_id(name):
    """
    Get array array_id, by name.

    @param name: The array name.

    @return: -1 in case of failure (i.e., no array with that
             name exists), a valid array_id otherwise.
    """
    return __l2m1(ida_netnode.netnode(_IDC_ARRAY_PREFIX + name, 0, False).index())


def rename_array(array_id, newname):
    """
    Rename array, by its ID.

    @param id: The ID of the array to rename.
    @param newname: The new name of the array.

    @return: 1 in case of success, 0 otherwise
```

```
    """
    return __GetArrayById(array_id).rename(_IDC_ARRAY_PREFIX + newname) == 1


def delete_array(array_id):
    """
    Delete array, by its ID.

    @param array_id: The ID of the array to delete.
    """
    __GetArrayById(array_id).kill()


def set_array_long(array_id, idx, value):
    """
    Sets the long value of an array element.

    @param array_id: The array ID.
    @param idx: Index of an element.
    @param value: 32bit or 64bit value to store in the array

    @return: 1 in case of success, 0 otherwise
    """
    return __GetArrayById(array_id).altset(idx, value)


def set_array_string(array_id, idx, value):
    """
    Sets the string value of an array element.

    @param array_id: The array ID.
    @param idx: Index of an element.
    @param value: String value to store in the array

    @return: 1 in case of success, 0 otherwise
    """
    return __GetArrayById(array_id).supset(idx, value)


def get_array_element(tag, array_id, idx):
    """
    Get value of array element.

    @param tag: Tag of array, specifies one of two array types: AR_LONG, AR_STR
    @param array_id: The array ID.
    @param idx: Index of an element.
```

```
    @return: Value of the specified array element. Note that
             this function may return char or long result. Unexistent
             array elements give zero as a result.
    """
    node = __GetArrayById(array_id)
    if tag == AR_LONG:
        return node.altval(idx, tag)
    elif tag == AR_STR:
        res = node.supval(idx, tag)
        return 0 if res is None else res
    else:
        return 0


def del_array_element(tag, array_id, idx):
    """
    Delete an array element.

    @param tag: Tag of array, specifies one of two array types: AR_LONG, AR_STR
    @param array_id: The array ID.
    @param idx: Index of an element.

    @return: 1 in case of success, 0 otherwise.
    """
    node = __GetArrayById(array_id)
    if tag == AR_LONG:
        return node.altdel(idx, tag)
    elif tag == AR_STR:
        return node.supdel(idx, tag)
    else:
        return 0


def get_first_index(tag, array_id):
    """
    Get index of the first existing array element.

    @param tag: Tag of array, specifies one of two array types: AR_LONG, AR_STR
    @param array_id: The array ID.

    @return: -1 if the array is empty, otherwise index of first array
             element of given type.
    """
    node = __GetArrayById(array_id)
    if tag == AR_LONG:
```

```
        return __l2m1(node.altfirst(tag))
    elif tag == AR_STR:
        return __l2m1(node.supfirst(tag))
    else:
        return -1


def get_last_index(tag, array_id):
    """
    Get index of last existing array element.

    @param tag: Tag of array, specifies one of two array types: AR_LONG, AR_STR
    @param array_id: The array ID.

    @return: -1 if the array is empty, otherwise index of first array
             element of given type.
    """
    node = __GetArrayById(array_id)
    if tag == AR_LONG:
        return __l2m1(node.altlast(tag))
    elif tag == AR_STR:
        return __l2m1(node.suplast(tag))
    else:
        return -1


def get_next_index(tag, array_id, idx):
    """
    Get index of the next existing array element.

    @param tag: Tag of array, specifies one of two array types: AR_LONG, AR_STR
    @param array_id: The array ID.
    @param idx: Index of the current element.

    @return: -1 if no more elements, otherwise returns index of the
             next array element of given type.
    """
    node = __GetArrayById(array_id)
    try:
        if tag == AR_LONG:
            return __l2m1(node.altnext(__m1tol(idx), tag))
        elif tag == AR_STR:
            return __l2m1(node.supnext(__m1tol(idx), tag))
        else:
            return -1
    except OverflowError:
```

```
        # typically: An index of -1 was passed.
        return -1


def get_prev_index(tag, array_id, idx):
    """
    Get index of the previous existing array element.

    @param tag: Tag of array, specifies one of two array types: AR_LONG, AR_STR
    @param array_id: The array ID.
    @param idx: Index of the current element.

    @return: -1 if no more elements, otherwise returns index of the
             previous array element of given type.
    """
    node = __GetArrayById(array_id)
    try:
        if tag == AR_LONG:
            return __l2m1(node.altprev(__m1tol(idx), tag))
        elif tag == AR_STR:
            return __l2m1(node.supprev(__m1tol(idx), tag))
        else:
            return -1
    except OverflowError:
        # typically: An index of -1 was passed.
        return -1


# -------------------- hashes ------------------------

def set_hash_long(hash_id, key, value):
    """
    Sets the long value of a hash element.

    @param hash_id: The hash ID.
    @param key: Key of an element.
    @param value: 32bit or 64bit value to store in the hash

    @return: 1 in case of success, 0 otherwise
    """
    return __GetArrayById(hash_id).hashset_idx(key, value)


def get_hash_long(hash_id, key):
    """
    Gets the long value of a hash element.
```

```
    @param hash_id: The hash ID.
    @param key: Key of an element.

    @return: the 32bit or 64bit value of the element, or 0 if no such
             element.
    """
    return __GetArrayById(hash_id).hashval_long(key);


def set_hash_string(hash_id, key, value):
    """
    Sets the string value of a hash element.

    @param hash_id: The hash ID.
    @param key: Key of an element.
    @param value: string value to store in the hash

    @return: 1 in case of success, 0 otherwise
    """
    return __GetArrayById(hash_id).hashset_buf(key, value)


def get_hash_string(hash_id, key):
    """
    Gets the string value of a hash element.

    @param hash_id: The hash ID.
    @param key: Key of an element.

    @return: the string value of the element, or None if no such
             element.
    """
    return __GetArrayById(hash_id).hashstr_buf(key);


def del_hash_string(hash_id, key):
    """
    Delete a hash element.

    @param hash_id: The hash ID.
    @param key: Key of an element

    @return: 1 upon success, 0 otherwise.
    """
    return __GetArrayById(hash_id).hashdel(key)
```

```
def get_first_hash_key(hash_id):
    """
    Get the first key in the hash.

    @param hash_id: The hash ID.

    @return: the key, 0 otherwise.
    """
    r = __GetArrayById(hash_id).hashfirst()
    return 0 if r is None else r


def get_last_hash_key(hash_id):
    """
    Get the last key in the hash.

    @param hash_id: The hash ID.

    @return: the key, 0 otherwise.
    """
    r = __GetArrayById(hash_id).hashlast()
    return 0 if r is None else r


def get_next_hash_key(hash_id, key):
    """
    Get the next key in the hash.

    @param hash_id: The hash ID.
    @param key: The current key.

    @return: the next key, 0 otherwise
    """
    r = __GetArrayById(hash_id).hashnext(key)
    return 0 if r is None else r


def get_prev_hash_key(hash_id, key):
    """
    Get the previous key in the hash.

    @param hash_id: The hash ID.
    @param key: The current key.
```

```
    @return: the previous key, 0 otherwise
    """
    r = __GetArrayById(hash_id).hashprev(key)
    return 0 if r is None else r



#--------------------------------------------------------------------------
#                S O U R C E   F I L E / L I N E   N U M B E R S
#--------------------------------------------------------------------------
add_sourcefile = ida_lines.add_sourcefile
get_sourcefile = ida_lines.get_sourcefile
del_sourcefile = ida_lines.del_sourcefile

set_source_linnum = ida_nalt.set_source_linnum
get_source_linnum = ida_nalt.get_source_linnum
del_source_linnum = ida_nalt.del_source_linnum


#--------------------------------------------------------------------------
#                T Y P E   L I B R A R I E S
#--------------------------------------------------------------------------

def add_default_til(name):
    """
    Load a type library

    @param name: name of type library.
    @return: 1-ok, 0-failed.
    """
    til = ida_typeinf.add_til(name, ida_typeinf.ADDTIL_DEFAULT)
    if til:
        return 1
    else:
        return 0


def import_type(idx, type_name):
    """
    Copy information from type library to database
    Copy structure, union, or enum definition from the type library
    to the IDA database.

    @param idx: the position of the new type in the list of
                types (structures or enums) -1 means at the end of the list
```

```
    @param type_name: name of type to copy

    @return: BADNODE-failed, otherwise the type id (structure id or enum id)
    """
    return ida_typeinf.import_type(None, idx, type_name)


def get_type(ea):
    """
    Get type of function/variable

    @param ea: the address of the object

    @return: type string or None if failed
    """
    return ida_typeinf.idc_get_type(ea)

def SizeOf(typestr):
    """
    Returns the size of the type. It is equivalent to IDC's sizeof().
    Use name, tp, fld = idc.parse_decl() ; SizeOf(tp) to retrieve the size
    @return: -1 if typestring is not valid otherwise the size of the type
    """
    return ida_typeinf.calc_type_size(None, typestr)

def get_tinfo(ea):
    """
    Get type information of function/variable as 'typeinfo' object

    @param ea: the address of the object
    @return: None on failure, or (type, fields) tuple.
    """
    return ida_typeinf.idc_get_type_raw(ea)

def get_local_tinfo(ordinal):
    """
    Get local type information as 'typeinfo' object

    @param ordinal:  slot number (1...NumberOfLocalTypes)
    @return: None on failure, or (type, fields) tuple.
    """
    return ida_typeinf.idc_get_local_type_raw(ordinal)

def guess_type(ea):
    """
    Guess type of function/variable
```

```
    @param ea: the address of the object, can be the structure member id too

    @return: type string or None if failed
    """
    return ida_typeinf.idc_guess_type(ea)

TINFO_GUESSED   = 0x0000 # this is a guessed type
TINFO_DEFINITE  = 0x0001 # this is a definite type
TINFO_DELAYFUNC = 0x0002 # if type is a function and no function exists at ea,
                         # schedule its creation and argument renaming to
                         # auto-analysis otherwise try to create it immediately

def apply_type(ea, py_type, flags = TINFO_DEFINITE):
    """
    Apply the specified type to the address

    @param ea: the address of the object
    @param py_type: typeinfo tuple (type, fields) as get_tinfo() returns
                 or tuple (name, type, fields) as parse_decl() returns
                 or None
                if specified as None, then the
                item associated with 'ea' will be deleted.
    @param flags: combination of TINFO_... constants or 0
    @return: Boolean
    """

    if py_type is None:
        py_type = ""
    if isinstance(py_type, ida_idaapi.string_types) and len(py_type) == 0:
        pt = (b"", b"")
    else:
        if len(py_type) == 3:
          pt = py_type[1:]      # skip name component
        else:
          pt = py_type
    return ida_typeinf.apply_type(None, pt[0], pt[1], ea, flags)


PT_SIL      = ida_typeinf.PT_SIL      # silent, no messages
PT_NDC      = ida_typeinf.PT_NDC      # don't decorate names
PT_TYP      = ida_typeinf.PT_TYP      # return declared type information
PT_VAR      = ida_typeinf.PT_VAR      # return declared object information
PT_PACKMASK = ida_typeinf.PT_PACKMASK # mask for pack alignment values
PT_HIGH     = ida_typeinf.PT_HIGH     # assume high level prototypes (with hidden args, etc
PT_LOWER    = ida_typeinf.PT_LOWER    # lower the function prototypes
```

```
PT_REPLACE  = ida_typeinf.PT_REPLACE  # replace the old type (used in idc)
PT_RAWARGS  = ida_typeinf.PT_RAWARGS  # leave argument names unchanged (do not remove unders

PT_SILENT = PT_SIL # alias
PT_PAKDEF = 0x0000 # default pack value
PT_PAK1   = 0x0010 # #pragma pack(1)
PT_PAK2   = 0x0020 # #pragma pack(2)
PT_PAK4   = 0x0030 # #pragma pack(4)
PT_PAK8   = 0x0040 # #pragma pack(8)
PT_PAK16  = 0x0050 # #pragma pack(16)

# idc.py-specific
PT_FILE = 0x00010000 # input if a file name (otherwise contains type declarations)


def SetType(ea, newtype):
    """
    Set type of function/variable

    @param ea: the address of the object
    @param newtype: the type string in C declaration form.
                Must contain the closing ';'
                if specified as an empty string, then the
                item associated with 'ea' will be deleted.

    @return: 1-ok, 0-failed.
    """
    if newtype != '':
        pt = parse_decl(newtype, PT_SIL)
        if pt is None:
          # parsing failed
          return None
    else:
        pt = None
    return apply_type(ea, pt, TINFO_DEFINITE)

def parse_decl(inputtype, flags):
    """
    Parse type declaration

    @param inputtype: file name or C declarations (depending on the flags)
    @param flags: combination of PT_... constants or 0

    @return: None on failure or (name, type, fields) tuple
    """
    if len(inputtype) != 0 and inputtype[-1] != ';':
```

```
        inputtype = inputtype + ';'
    return ida_typeinf.idc_parse_decl(None, inputtype, flags)

def parse_decls(inputtype, flags = 0):
    """
    Parse type declarations

    @param inputtype: file name or C declarations (depending on the flags)
    @param flags: combination of PT_... constants or 0

    @return: number of parsing errors (0 no errors)
    """
    return ida_typeinf.idc_parse_types(inputtype, flags)


def print_decls(ordinals, flags):
    """
    Print types in a format suitable for use in a header file

    @param ordinals: comma-separated list of type ordinals
    @param flags: combination of PDF_... constants or 0

    @return: string containing the type definitions
    """
    class def_sink(ida_typeinf.text_sink_t):

        def __init__(self):
            ida_typeinf.text_sink_t.__init__(self)
            self.text = ""

        def _print(self, defstr):
            self.text += defstr
            return 0

    sink = def_sink()
    py_ordinals = list(map(lambda l : int(l), ordinals.split(",")))
    ida_typeinf.print_decls(sink, None, py_ordinals, flags)

    return sink.text


PDF_INCL_DEPS  = 0x1  # include dependencies
PDF_DEF_FWD    = 0x2  # allow forward declarations
PDF_DEF_BASE   = 0x4  # include base types: __int8, __int16, etc..
PDF_HEADER_CMT = 0x8  # prepend output with a descriptive comment
```

```
def get_ordinal_qty():
    """
    Get number of local types + 1

    @return: value >= 1. 1 means that there are no local types.
    """
    return ida_typeinf.get_ordinal_qty(None)


def set_local_type(ordinal, input, flags):
    """
    Parse one type declaration and store it in the specified slot

    @param ordinal:  slot number (1...NumberOfLocalTypes)
                     -1 means allocate new slot or reuse the slot
                     of the existing named type
    @param input:  C declaration. Empty input empties the slot
    @param flags:  combination of PT_... constants or 0

    @return: slot number or 0 if error
    """
    return ida_typeinf.idc_set_local_type(ordinal, input, flags)


def GetLocalType(ordinal, flags):
    """
    Retrieve a local type declaration
    @param flags: any of PRTYPE_* constants
    @return: local type as a C declaration or ""
    """
    (type, fields) = get_local_tinfo(ordinal)
    if type:
      name = get_numbered_type_name(ordinal)
      return ida_typeinf.idc_print_type(type, fields, name, flags)
    return ""

PRTYPE_1LINE  = 0x0000 # print to one line
PRTYPE_MULTI  = 0x0001 # print to many lines
PRTYPE_TYPE   = 0x0002 # print type declaration (not variable declaration)
PRTYPE_PRAGMA = 0x0004 # print pragmas for alignment


def get_numbered_type_name(ordinal):
    """
    Retrieve a local type name
```

```
    @param ordinal:  slot number (1...NumberOfLocalTypes)

    returns: local type name or None
    """
    return ida_typeinf.idc_get_local_type_name(ordinal)


# ----------------------------------------------------------------------------
#                           H I D D E N  A R E A S
# ----------------------------------------------------------------------------
add_hidden_range = ida_bytes.add_hidden_range

def update_hidden_range(ea, visible):
    """
    Set hidden range state

    @param ea:      any address belonging to the hidden range
    @param visible: new state of the range

    @return: != 0 - ok
    """
    ha = ida_bytes.get_hidden_range(ea)

    if not ha:
        return 0
    else:
        ha.visible = visible
        return ida_bytes.update_hidden_range(ha)


del_hidden_range = ida_bytes.del_hidden_range


#----------------------------------------------------------------------------
#                   D E B U G G E R  I N T E R F A C E
#----------------------------------------------------------------------------
load_debugger = ida_dbg.load_debugger
start_process = ida_dbg.start_process
exit_process = ida_dbg.exit_process
suspend_process = ida_dbg.suspend_process
get_processes = ida_dbg.get_processes
attach_process = ida_dbg.attach_process
detach_process = ida_dbg.detach_process
get_thread_qty = ida_dbg.get_thread_qty
getn_thread = ida_dbg.getn_thread
```

```
get_current_thread = ida_dbg.get_current_thread
getn_thread_name = ida_dbg.getn_thread_name
select_thread = ida_dbg.select_thread
suspend_thread = ida_dbg.suspend_thread
resume_thread = ida_dbg.resume_thread


def _get_modules():
    """
    INTERNAL: Enumerate process modules
    """
    module = ida_idd.modinfo_t()
    result = ida_dbg.get_first_module(module)
    while result:
        yield module
        result = ida_dbg.get_next_module(module)


def get_first_module():
    """
    Enumerate process modules

    @return: first module's base address or None on failure
    """
    for module in _get_modules():
        return module.base
    else:
        return None


def get_next_module(base):
    """
    Enumerate process modules

    @param base: previous module's base address

    @return: next module's base address or None on failure
    """
    foundit = False
    for module in _get_modules():
        if foundit:
            return module.base
        if module.base == base:
            foundit = True
    else:
        return None
```

```
def get_module_name(base):
    """
    Get process module name

    @param base: the base address of the module

    @return: required info or None
    """
    for module in _get_modules():
        if module.base == base:
            return module.name
    else:
        return 0


def get_module_size(base):
    """
    Get process module size

    @param base: the base address of the module

    @return: required info or -1
    """
    for module in _get_modules():
        if module.base == base:
            return module.size
    else:
        return -1


step_into = ida_dbg.step_into
step_over = ida_dbg.step_over
run_to = ida_dbg.run_to
step_until_ret = ida_dbg.step_until_ret


wait_for_next_event = ida_dbg.wait_for_next_event


def resume_process():
    return wait_for_next_event(WFNE_CONT|WFNE_NOWAIT, 0)

def send_dbg_command(cmd):
    """Sends a command to the debugger module and returns the output string.
```

```
    An exception will be raised if the debugger is not running or the current debugger does
    the 'send_dbg_command' IDC command.
    """
    s = eval_idc('send_dbg_command("%s");' % ida_kernwin.str2user(cmd))
    if s.startswith("IDC_FAILURE"):
        raise Exception("Debugger command is available only when the debugger is active!")
    return s

# wfne flag is combination of the following:
WFNE_ANY    = 0x0001 # return the first event (even if it doesn't suspend the process)
                     # if the process is still running, the database
                     # does not reflect the memory state. you might want
                     # to call refresh_debugger_memory() in this case
WFNE_SUSP   = 0x0002 # wait until the process gets suspended
WFNE_SILENT = 0x0004 # 1: be slient, 0:display modal boxes if necessary
WFNE_CONT   = 0x0008 # continue from the suspended state
WFNE_NOWAIT = 0x0010 # do not wait for any event, immediately return DEC_TIMEOUT
                     # (to be used with WFNE_CONT)

# debugger event codes
NOTASK            = -2         # process does not exist
DBG_ERROR         = -1         # error (e.g. network problems)
DBG_TIMEOUT       = 0          # timeout
PROCESS_STARTED   = 0x00000001 # New process started
PROCESS_EXITED    = 0x00000002 # Process stopped
THREAD_STARTED    = 0x00000004 # New thread started
THREAD_EXITED     = 0x00000008 # Thread stopped
BREAKPOINT        = 0x00000010 # Breakpoint reached
STEP              = 0x00000020 # One instruction executed
EXCEPTION         = 0x00000040 # Exception
LIB_LOADED        = 0x00000080 # New library loaded
LIB_UNLOADED      = 0x00000100 # Library unloaded
INFORMATION       = 0x00000200 # User-defined information
PROCESS_ATTACHED  = 0x00000400 # Attached to running process
PROCESS_DETACHED  = 0x00000800 # Detached from process
PROCESS_SUSPENDED = 0x00001000 # Process has been suspended


refresh_debugger_memory = ida_dbg.refresh_debugger_memory

take_memory_snapshot = ida_segment.take_memory_snapshot

get_process_state = ida_dbg.get_process_state

DSTATE_SUSP            = -1 # process is suspended
DSTATE_NOTASK          =  0 # no process is currently debugged
```

```
DSTATE_RUN             =  1 # process is running
DSTATE_RUN_WAIT_ATTACH =  2 # deprecated
DSTATE_RUN_WAIT_END    =  3 # deprecated

"""
 Get various information about the current debug event
 These functions are valid only when the current event exists
 (the process is in the suspended state)
"""

# For all events:

def get_event_id():
    """
    Get ID of debug event

    @return: event ID
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ev.eid()


def get_event_pid():
    """
    Get process ID for debug event

    @return: process ID
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ev.pid


def get_event_tid():
    """
    Get type ID for debug event

    @return: type ID
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ev.tid


def get_event_ea():
```

```
    """
    Get ea for debug event

    @return: ea
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ev.ea


def is_event_handled():
    """
    Is the debug event handled?

    @return: boolean
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ev.handled


# For PROCESS_STARTED, PROCESS_ATTACHED, LIB_LOADED events:

def get_event_module_name():
    """
    Get module name for debug event

    @return: module name
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ida_idd.get_event_module_name(ev)


def get_event_module_base():
    """
    Get module base for debug event

    @return: module base
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ida_idd.get_event_module_base(ev)


def get_event_module_size():
```

```
    """
    Get module size for debug event

    @return: module size
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ida_idd.get_event_module_size(ev)


def get_event_exit_code():
    """
    Get exit code for debug event

    @return: exit code for PROCESS_EXITED, THREAD_EXITED events
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ev.exit_code()


def get_event_info():
    """
    Get debug event info

    @return: event info: for THREAD_STARTED (thread name)
                         for LIB_UNLOADED (unloaded library name)
                         for INFORMATION (message to display)
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ida_idd.get_event_info(ev)


def get_event_bpt_hea():
    """
    Get hardware address for BREAKPOINT event

    @return: hardware address
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ida_idd.get_event_bpt_hea(ev)


def get_event_exc_code():
```

```
    """
    Get exception code for EXCEPTION event

    @return: exception code
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ida_idd.get_event_exc_code(ev)


def get_event_exc_ea():
    """
    Get address for EXCEPTION event

    @return: adress of exception
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ida_idd.get_event_exc_ea(ev)


def can_exc_continue():
    """
    Can it continue after EXCEPTION event?

    @return: boolean
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ida_idd.can_exc_continue(ev)


def get_event_exc_info():
    """
    Get info for EXCEPTION event

    @return: info string
    """
    ev = ida_dbg.get_debug_event()
    assert ev, "Could not retrieve debug event"
    return ida_idd.get_event_exc_info(ev)


set_debugger_options = ida_dbg.set_debugger_options
```

```
DOPT_SEGM_MSGS    = 0x00000001 # print messages on debugger segments modifications
DOPT_START_BPT    = 0x00000002 # break on process start
DOPT_THREAD_MSGS  = 0x00000004 # print messages on thread start/exit
DOPT_THREAD_BPT   = 0x00000008 # break on thread start/exit
DOPT_BPT_MSGS     = 0x00000010 # print message on breakpoint
DOPT_LIB_MSGS     = 0x00000040 # print message on library load/unlad
DOPT_LIB_BPT      = 0x00000080 # break on library load/unlad
DOPT_INFO_MSGS    = 0x00000100 # print message on debugging information
DOPT_INFO_BPT     = 0x00000200 # break on debugging information
DOPT_REAL_MEMORY  = 0x00000400 # don't hide breakpoint instructions
DOPT_REDO_STACK   = 0x00000800 # reconstruct the stack
DOPT_ENTRY_BPT    = 0x00001000 # break on program entry point
DOPT_EXCDLG       = 0x00006000 # exception dialogs:

EXCDLG_NEVER      = 0x00000000 # never display exception dialogs
EXCDLG_UNKNOWN    = 0x00002000 # display for unknown exceptions
EXCDLG_ALWAYS     = 0x00006000 # always display

DOPT_LOAD_DINFO   = 0x00008000 # automatically load debug files (pdb)


get_debugger_event_cond = ida_dbg.get_debugger_event_cond
set_debugger_event_cond = ida_dbg.set_debugger_event_cond

set_remote_debugger = ida_dbg.set_remote_debugger

define_exception = ida_dbg.define_exception

EXC_BREAK  = 0x0001 # break on the exception
EXC_HANDLE = 0x0002 # should be handled by the debugger?

get_reg_value = ida_dbg.get_reg_val

def set_reg_value(value, name):
    """
    Set register value

    @param name: the register name
    @param value: new register value

    @note: The debugger should be running
           It is not necessary to use this function to set register values.
           A register name in the left side of an assignment will do too.
    """
    return ida_dbg.set_reg_val(name, value)
```

```
get_bpt_qty = ida_dbg.get_bpt_qty


def get_bpt_ea(n):
    """
    Get breakpoint address

    @param n: number of breakpoint, is in range 0..get_bpt_qty()-1

    @return: address of the breakpoint or BADADDR
    """
    bpt = ida_dbg.bpt_t()

    if ida_dbg.getn_bpt(n, bpt):
        return bpt.ea
    else:
        return BADADDR


def get_bpt_attr(ea, bptattr):
    """
    Get the characteristics of a breakpoint

    @param ea: any address in the breakpoint range
    @param bptattr: the desired attribute code, one of BPTATTR_... constants

    @return: the desired attribute value or -1
    """
    bpt = ida_dbg.bpt_t()

    if not ida_dbg.get_bpt(ea, bpt):
        return -1
    else:
        if bptattr == BPTATTR_EA:
            return bpt.ea
        if bptattr == BPTATTR_SIZE:
            return bpt.size
        if bptattr == BPTATTR_TYPE:
            return bpt.type
        if bptattr == BPTATTR_COUNT:
            return bpt.pass_count
        if bptattr == BPTATTR_FLAGS:
            return bpt.flags
        if bptattr == BPTATTR_COND:
            return bpt.condition
```

```
        if bptattr == BPTATTR_PID:
            return bpt.pid
        if bptattr == BPTATTR_TID:
            return bpt.tid
        return -1


BPTATTR_EA    =  1   # starting address of the breakpoint
BPTATTR_SIZE  =  2   # size of the breakpoint (undefined for software breakpoint)

# type of the breakpoint
BPTATTR_TYPE  =  3

# Breakpoint types:
BPT_WRITE    = 1                     # Hardware: Write access
BPT_RDWR     = 3                     # Hardware: Read/write access
BPT_SOFT     = 4                     # Software breakpoint
BPT_EXEC     = 8                     # Hardware: Execute instruction
BPT_DEFAULT  = (BPT_SOFT|BPT_EXEC);  # Choose bpt type automaticaly

BPTATTR_COUNT  =  4
BPTATTR_FLAGS  =  5
BPT_BRK        = 0x001 # the debugger stops on this breakpoint
BPT_TRACE      = 0x002 # the debugger adds trace information when this breakpoint is reache
BPT_UPDMEM     = 0x004 # refresh the memory layout and contents before evaluating bpt condi
BPT_ENABLED    = 0x008 # enabled?
BPT_LOWCND     = 0x010 # condition is calculated at low level (on the server side)
BPT_TRACEON    = 0x020 # enable tracing when the breakpoint is reached
BPT_TRACE_INSN = 0x040 #   instruction tracing
BPT_TRACE_FUNC = 0x080 #   function tracing
BPT_TRACE_BBLK = 0x100 #   basic block tracing

BPTATTR_COND  =  6   # Breakpoint condition. NOTE: the return value is a string in this cas
BPTATTR_PID   =  7   # Brekapoint process id
BPTATTR_TID   =  8   # Brekapoint thread id

# Breakpoint location type:
BPLT_ABS  =  0   # Absolute address. Attributes:
                 # - locinfo: absolute address

BPLT_REL  =  1   # Module relative address. Attributes:
                 # - locpath: the module path
                 # - locinfo: offset from the module base address

BPLT_SYM  =  2   # Symbolic name. The name will be resolved on DLL load/unload
                 # events and on naming an address. Attributes:
```

```
                # - locpath: symbol name
                # - locinfo: offset from the symbol base address


def set_bpt_attr(address, bptattr, value):
    """
        modifiable characteristics of a breakpoint

    @param address: any address in the breakpoint range
    @param bptattr: the attribute code, one of BPTATTR_* constants
                    BPTATTR_CND is not allowed, see set_bpt_cond()
    @param value: the attibute value

    @return: success
    """
    bpt = ida_dbg.bpt_t()

    if not ida_dbg.get_bpt(address, bpt):
        return False
    else:
        if bptattr not in [ BPTATTR_SIZE, BPTATTR_TYPE, BPTATTR_FLAGS, BPTATTR_COUNT, BPTATT
            return False
        if bptattr == BPTATTR_SIZE:
            bpt.size = value
        if bptattr == BPTATTR_TYPE:
            bpt.type = value
        if bptattr == BPTATTR_COUNT:
            bpt.pass_count = value
        if bptattr == BPTATTR_FLAGS:
            bpt.flags = value
        if bptattr == BPTATTR_PID:
            bpt.pid = value
        if bptattr == BPTATTR_TID:
            bpt.tid = value

        return ida_dbg.update_bpt(bpt)


def set_bpt_cond(ea, cnd, is_lowcnd=0):
    """
    Set breakpoint condition

    @param ea: any address in the breakpoint range
    @param cnd: breakpoint condition
    @param is_lowcnd: 0 - regular condition, 1 - low level condition
```

```
    @return: success
    """
    bpt = ida_dbg.bpt_t()

    if not ida_dbg.get_bpt(ea, bpt):
        return False

    bpt.condition = cnd
    if is_lowcnd:
        bpt.flags |= BPT_LOWCND
    else:
        bpt.flags &= ~BPT_LOWCND

    return ida_dbg.update_bpt(bpt)


add_bpt    = ida_dbg.add_bpt
del_bpt    = ida_dbg.del_bpt
enable_bpt = ida_dbg.enable_bpt
check_bpt  = ida_dbg.check_bpt

BPTCK_NONE = -1  # breakpoint does not exist
BPTCK_NO   =  0  # breakpoint is disabled
BPTCK_YES  =  1  # breakpoint is enabled
BPTCK_ACT  =  2  # breakpoint is active (written to the process)


def enable_tracing(trace_level, enable):
    """
    Enable step tracing

    @param trace_level:  what kind of trace to modify
    @param enable: 0: turn off, 1: turn on

    @return: success
    """
    assert trace_level in [ TRACE_STEP, TRACE_INSN, TRACE_FUNC ], \
        "trace_level must be one of TRACE_* constants"

    if trace_level == TRACE_STEP:
        return ida_dbg.enable_step_trace(enable)

    if trace_level == TRACE_INSN:
        return ida_dbg.enable_insn_trace(enable)

    if trace_level == TRACE_FUNC:
```

```
        return ida_dbg.enable_func_trace(enable)

    return False

TRACE_STEP = 0x0  # lowest level trace. trace buffers are not maintained
TRACE_INSN = 0x1  # instruction level trace
TRACE_FUNC = 0x2  # function level trace (calls & rets)


get_step_trace_options = ida_dbg.get_step_trace_options
set_step_trace_options = ida_dbg.set_step_trace_options


ST_OVER_DEBUG_SEG = 0x01 # step tracing will be disabled when IP is in a debugger segment
ST_OVER_LIB_FUNC  = 0x02 # step tracing will be disabled when IP is in a library function
ST_ALREADY_LOGGED = 0x04 # step tracing will be disabled when IP is already logged
ST_SKIP_LOOPS     = 0x08 # step tracing will try to skip loops already recorded

load_trace_file = ida_dbg.load_trace_file
save_trace_file = ida_dbg.save_trace_file
is_valid_trace_file = ida_dbg.is_valid_trace_file
diff_trace_file = ida_dbg.diff_trace_file

def clear_trace(filename):
    """
    Clear the current trace buffer
    """
    return ida_dbg.clear_trace()

get_trace_file_desc = ida_dbg.get_trace_file_desc
set_trace_file_desc = ida_dbg.set_trace_file_desc
get_tev_qty = ida_dbg.get_tev_qty
get_tev_ea = ida_dbg.get_tev_ea

TEV_NONE  = 0 # no event
TEV_INSN  = 1 # an instruction trace
TEV_CALL  = 2 # a function call trace
TEV_RET   = 3 # a function return trace
TEV_BPT   = 4 # write, read/write, execution trace
TEV_MEM   = 5 # memory layout changed
TEV_EVENT = 6 # debug event

get_tev_type = ida_dbg.get_tev_type
get_tev_tid = ida_dbg.get_tev_tid
get_tev_reg = ida_dbg.get_tev_reg_val
get_tev_mem_qty = ida_dbg.get_tev_reg_mem_qty
```

```
get_tev_mem = ida_dbg.get_tev_reg_mem
get_tev_mem_ea = ida_dbg.get_tev_reg_mem_ea
get_call_tev_callee = ida_dbg.get_call_tev_callee
get_ret_tev_return = ida_dbg.get_ret_tev_return
get_bpt_tev_ea = ida_dbg.get_bpt_tev_ea


#--------------------------------------------------------------------------
#                            C O L O R S
#--------------------------------------------------------------------------

def get_color(ea, what):
    """
    Get item color

    @param ea: address of the item
    @param what: type of the item (one of  CIC_* constants)

    @return: color code in RGB (hex 0xBBGGRR)
    """
    if what not in [ CIC_ITEM, CIC_FUNC, CIC_SEGM ]:
        raise ValueError("'what' must be one of CIC_ITEM, CIC_FUNC and CIC_SEGM")

    if what == CIC_ITEM:
        return ida_nalt.get_item_color(ea)

    if what == CIC_FUNC:
        func = ida_funcs.get_func(ea)
        if func:
            return func.color
        else:
            return DEFCOLOR

    if what == CIC_SEGM:
        seg = ida_segment.getseg(ea)
        if seg:
            return seg.color
        else:
            return DEFCOLOR

# color item codes:
CIC_ITEM = 1         # one instruction or data
CIC_FUNC = 2         # function
CIC_SEGM = 3         # segment

DEFCOLOR = 0xFFFFFFFF  # Default color
```

```
def set_color(ea, what, color):
    """
    Set item color

    @param ea: address of the item
    @param what: type of the item (one of CIC_* constants)
    @param color: new color code in RGB (hex 0xBBGGRR)

    @return: success (True or False)
    """
    if what not in [ CIC_ITEM, CIC_FUNC, CIC_SEGM ]:
        raise ValueError("'what' must be one of CIC_ITEM, CIC_FUNC and CIC_SEGM")

    if what == CIC_ITEM:
        return ida_nalt.set_item_color(ea, color)

    if what == CIC_FUNC:
        func = ida_funcs.get_func(ea)
        if func:
            func.color = color
            return bool(ida_funcs.update_func(func))
        else:
            return False

    if what == CIC_SEGM:
        seg = ida_segment.getseg(ea)
        if seg:
            seg.color = color
            return bool(seg.update())
        else:
            return False


#----------------------------------------------------------------------------
#                       A R M   S P E C I F I C
#----------------------------------------------------------------------------
def force_bl_jump(ea):
    """
    Some ARM compilers in Thumb mode use BL (branch-and-link)
    instead of B (branch) for long jumps, since BL has more range.
    By default, IDA tries to determine if BL is a jump or a call.
    You can override IDA's decision using commands in Edit/Other menu
    (Force BL call/Force BL jump) or the following two functions.
```

```
    Force BL instruction to be a jump

    @param ea: address of the BL instruction

    @return: 1-ok, 0-failed
    """
    return eval_idc("force_bl_jump(0x%x)"%ea)


def force_bl_call(ea):
    """
    Force BL instruction to be a call

    @param ea: address of the BL instruction

    @return: 1-ok, 0-failed
    """
    return eval_idc("force_bl_call(0x%x)"%ea)


#---------------------------------------------------------------------------
def set_flag(off, bit, value):
  v = get_inf_attr(off)
  if value:
    v = v | bit
  else:
    v = v & ~bit
  set_inf_attr(off, v)

# Convenience functions:
def here(): return get_screen_ea()
def is_mapped(ea): return (prev_addr(ea+1)==ea)

ARGV = []
"""The command line arguments passed to IDA via the -S switch."""

# END OF IDC COMPATIBILY CODE
```

IDAUtils module

This module contains various utility functions that are useful during IDAPython scripting. Some of its utility functions hides multiple calls to the low level `idaapi` module.

For example, to enumerate functions with `idautils`, simply:

```
import idautils
```

```
for f in idautils.Functions():
    print("Function at 0x%x" % f)
```

Here is the whole `idautils.py` file for reference:

```
"""
idautils.py - High level utility functions for IDA
"""
import ida_bytes
import ida_dbg
import ida_entry
import ida_funcs
import ida_ida
import ida_idaapi
import ida_idd
import ida_idp
import ida_kernwin
import ida_loader
import ida_nalt
import ida_name
import ida_netnode
import ida_segment
import ida_strlist
import ida_struct
import ida_ua
import ida_xref

import idc
import types
import os
import sys

def CodeRefsTo(ea, flow):
    """
    Get a list of code references to 'ea'

    @param ea:   Target address
    @param flow: Follow normal code flow or not
    @type  flow: Boolean (0/1, False/True)

    @return: list of references (may be empty list)

    Example::

        for ref in CodeRefsTo(get_screen_ea(), 1):
            print(ref)
```

```
    """
    xref = ida_xref.xrefblk_t()
    if flow == 1:
        yield from xref.crefs_to(ea)
    else:
        yield from xref.fcrefs_to(ea)


def CodeRefsFrom(ea, flow):
    """
    Get a list of code references from 'ea'

    @param ea:   Target address
    @param flow: Follow normal code flow or not
    @type  flow: Boolean (0/1, False/True)

    @return: list of references (may be empty list)

    Example::

        for ref in CodeRefsFrom(get_screen_ea(), 1):
            print(ref)
    """
    xref = ida_xref.xrefblk_t()
    if flow == 1:
        yield from xref.crefs_from(ea)
    else:
        yield from xref.fcrefs_from(ea)

def DataRefsTo(ea):
    """
    Get a list of data references to 'ea'

    @param ea:   Target address

    @return: list of references (may be empty list)

    Example::

        for ref in DataRefsTo(get_screen_ea()):
            print(ref)
    """
    xref = ida_xref.xrefblk_t()
    yield from xref.drefs_to(ea)
```

```
def DataRefsFrom(ea):
    """
    Get a list of data references from 'ea'

    @param ea:   Target address

    @return: list of references (may be empty list)

    Example::

        for ref in DataRefsFrom(get_screen_ea()):
            print(ref)
    """
    xref = ida_xref.xrefblk_t()
    yield from xref.drefs_from(ea)


# Xref type names table
_ref_types = {
    ida_xref.fl_U  : 'Data_Unknown',
    ida_xref.dr_O  : 'Data_Offset',
    ida_xref.dr_W  : 'Data_Write',
    ida_xref.dr_R  : 'Data_Read',
    ida_xref.dr_T  : 'Data_Text',
    ida_xref.dr_I  : 'Data_Informational',
    ida_xref.fl_CF : 'Code_Far_Call',
    ida_xref.fl_CN : 'Code_Near_Call',
    ida_xref.fl_JF : 'Code_Far_Jump',
    ida_xref.fl_JN : 'Code_Near_Jump',
    20 : 'Code_User',
    ida_xref.fl_F : 'Ordinary_Flow'
}

def XrefTypeName(typecode):
    """
    Convert cross-reference type codes to readable names

    @param typecode: cross-reference type code
    """
    assert typecode in _ref_types, "unknown reference type %d" % typecode
    return _ref_types[typecode]


def XrefsFrom(ea, flags=0):
    """
    Return all references from address 'ea'
```

```
    @param ea: Reference address
    @param flags: one of ida_xref.XREF_ALL (default), ida_xref.XREF_FAR, ida_xref.XREF_DATA

    Example::
           for xref in XrefsFrom(here(), 0):
               print(xref.type, XrefTypeName(xref.type), \
                         'from', hex(xref.frm), 'to', hex(xref.to))
    """
    xref = ida_xref.xrefblk_t()
    return xref.refs_from(ea, flags)


def XrefsTo(ea, flags=0):
    """
    Return all references to address 'ea'

    @param ea: Reference address
    @param flags: one of ida_xref.XREF_ALL (default), ida_xref.XREF_FAR, ida_xref.XREF_DATA

    Example::
           for xref in XrefsTo(here(), 0):
               print(xref.type, XrefTypeName(xref.type), \
                         'from', hex(xref.frm), 'to', hex(xref.to))
    """
    xref = ida_xref.xrefblk_t()
    return xref.refs_to(ea, flags)


def Threads():
    """Returns all thread IDs for the current debugee"""
    for i in range(0, idc.get_thread_qty()):
        yield idc.getn_thread(i)


def Heads(start=None, end=None):
    """
    Get a list of heads (instructions or data items)

    @param start: start address (default: inf.min_ea)
    @param end:   end address (default: inf.max_ea)

    @return: list of heads between start and end
    """
    if start is None: start = ida_ida.inf_get_min_ea()
    if end is None:   end = ida_ida.inf_get_max_ea()
```

```
    ea = start
    if not idc.is_head(ida_bytes.get_flags(ea)):
        ea = ida_bytes.next_head(ea, end)
    while ea < end and ea != ida_idaapi.BADADDR:
        yield ea
        ea = ida_bytes.next_head(ea, end)


def Functions(start=None, end=None):
    """
    Get a list of functions

    @param start: start address (default: inf.min_ea)
    @param end:   end address (default: inf.max_ea)

    @return: list of function entrypoints between start and end

    @note: The last function that starts before 'end' is included even
    if it extends beyond 'end'. Any function that has its chunks scattered
    in multiple segments will be reported multiple times, once in each segment
    as they are listed.
    """
    if start is None: start = ida_ida.inf_get_min_ea()
    if end is None:   end = ida_ida.inf_get_max_ea()

    # find first function head chunk in the range
    chunk = ida_funcs.get_fchunk(start)
    if not chunk:
        chunk = ida_funcs.get_next_fchunk(start)
    while chunk and chunk.start_ea < end and (chunk.flags & ida_funcs.FUNC_TAIL) != 0:
        chunk = ida_funcs.get_next_fchunk(chunk.start_ea)
    func = chunk

    while func and func.start_ea < end:
        startea = func.start_ea
        yield startea
        func = ida_funcs.get_next_func(startea)


def Chunks(start):
    """
    Get a list of function chunks
    See also ida_funcs.func_tail_iterator_t

    @param start: address of the function
```

```
    @return: list of function chunks (tuples of the form (start_ea, end_ea))
             belonging to the function
    """
    func_iter = ida_funcs.func_tail_iterator_t( ida_funcs.get_func( start ) )
    for chunk in func_iter:
        yield (chunk.start_ea, chunk.end_ea)


def Modules():
    """
    Returns a list of module objects with name,size,base and the rebase_to attributes
    """
    mod = ida_idd.modinfo_t()
    result = ida_dbg.get_first_module(mod)
    while result:
        # Note: can't simply return `mod` here, since callers might
        # collect all modules in a list, and they would all re-use
        # the underlying C++ object.
        yield ida_idaapi.object_t(name=mod.name, size=mod.size, base=mod.base, rebase_to=mod
        result = ida_dbg.get_next_module(mod)


def Names():
    """
    Returns a list of names

    @return: List of tuples (ea, name)
    """
    for i in range(ida_name.get_nlist_size()):
        ea   = ida_name.get_nlist_ea(i)
        name = ida_name.get_nlist_name(i)
        yield (ea, name)


def Segments():
    """
    Get list of segments (sections) in the binary image

    @return: List of segment start addresses.
    """
    for n in range(ida_segment.get_segm_qty()):
        seg = ida_segment.getnseg(n)
        if seg:
            yield seg.start_ea
```

```
def Entries():
    """
    Returns a list of entry points (exports)

    @return: List of tuples (index, ordinal, ea, name)
    """
    n = ida_entry.get_entry_qty()
    for i in range(0, n):
        ordinal = ida_entry.get_entry_ordinal(i)
        ea      = ida_entry.get_entry(ordinal)
        name    = ida_entry.get_entry_name(ordinal)
        yield (i, ordinal, ea, name)


def FuncItems(start):
    """
    Get a list of function items (instruction or data items inside function boundaries)
    See also ida_funcs.func_item_iterator_t

    @param start: address of the function

    @return: ea of each item in the function
    """
    return ida_funcs.func_item_iterator_t(ida_funcs.get_func(start))


def Structs():
    """
    Get a list of structures

    @return: List of tuples (idx, sid, name)
    """
    idx  = idc.get_first_struc_idx()
    while idx != ida_idaapi.BADADDR:
        sid = idc.get_struc_by_idx(idx)
        yield (idx, sid, idc.get_struc_name(sid))
        idx = idc.get_next_struc_idx(idx)


def StructMembers(sid):
    """
    Get a list of structure members information (or stack vars if given a frame).

    @param sid: ID of the structure.
```

```
    @return: List of tuples (offset, name, size)

    @note: If 'sid' does not refer to a valid structure,
           an exception will be raised.
    @note: This will not return 'holes' in structures/stack frames;
           it only returns defined structure members.
    """
    sptr = ida_struct.get_struc(sid)
    if sptr is None:
        raise Exception("No structure with ID: 0x%x" % sid)
    for m in sptr.members:
        name = idc.get_member_name(sid, m.soff)
        if name:
            size = ida_struct.get_member_size(m)
            yield (m.soff, name, size)


def DecodePrecedingInstruction(ea):
    """
    Decode preceding instruction in the execution flow.

    @param ea: address to decode
    @return: (None or the decode instruction, farref)
             farref will contain 'true' if followed an xref, false otherwise
    """
    insn = ida_ua.insn_t()
    prev_addr, farref  = ida_ua.decode_preceding_insn(insn, ea)
    return (insn, farref) if prev_addr != ida_idaapi.BADADDR else (None, False)


def DecodePreviousInstruction(ea):
    """
    Decodes the previous instruction and returns an insn_t like class

    @param ea: address to decode
    @return: None or a new insn_t instance
    """
    insn = ida_ua.insn_t()
    prev_addr = ida_ua.decode_prev_insn(insn, ea)
    return insn if prev_addr != ida_idaapi.BADADDR else None


def DecodeInstruction(ea):
    """
    Decodes an instruction and returns an insn_t like class
```

```
    @param ea: address to decode
    @return: None or a new insn_t instance
    """
    insn = ida_ua.insn_t()
    inslen = ida_ua.decode_insn(insn, ea)
    return insn if inslen > 0 else None


def GetDataList(ea, count, itemsize=1):
    """
    Get data list - INTERNAL USE ONLY
    """
    if itemsize == 1:
        getdata = ida_bytes.get_byte
    elif itemsize == 2:
        getdata = ida_bytes.get_word
    elif itemsize == 4:
        getdata = ida_bytes.get_dword
    elif itemsize == 8:
        getdata = ida_bytes.get_qword
    else:
        raise ValueError("Invalid data size! Must be 1, 2, 4 or 8")

    endea = ea + itemsize * count
    curea = ea
    while curea < endea:
        yield getdata(curea)
        curea += itemsize


def PutDataList(ea, datalist, itemsize=1):
    """
    Put data list - INTERNAL USE ONLY
    """
    putdata = None

    if itemsize == 1:
        putdata = ida_bytes.patch_byte
    if itemsize == 2:
        putdata = ida_bytes.patch_word
    if itemsize == 4:
        putdata = ida_bytes.patch_dword

    assert putdata, "Invalid data size! Must be 1, 2 or 4"

    for val in datalist:
```

```
        putdata(ea, val)
        ea = ea + itemsize


def MapDataList(ea, length, func, wordsize=1):
    """
    Map through a list of data words in the database

    @param ea:       start address
    @param length:   number of words to map
    @param func:     mapping function
    @param wordsize: size of words to map [default: 1 byte]

    @return: None
    """
    PutDataList(ea, map(func, GetDataList(ea, length, wordsize)), wordsize)


GetInputFileMD5 = ida_nalt.retrieve_input_file_md5


class Strings(object):
    """
    Allows iterating over the string list. The set of strings will not be
    modified, unless asked explicitly at setup()-time. This string list also
    is used by the "String window" so it may be changed when this window is
    updated.

    Example:
        s = Strings()

        for i in s:
            print("%x: len=%d type=%d -> '%s'" % (i.ea, i.length, i.strtype, str(i)))

    """
    class StringItem(object):
        """
        Class representing each string item.
        """
        def __init__(self, si):
            self.ea = si.ea
            """String ea"""
            self.strtype = si.type
            """string type (STRTYPE_xxxxx)"""
            self.length = si.length
            """string length"""
```

```
    def is_1_byte_encoding(self):
        return ida_nalt.get_strtype_bpu(self.strtype) == 1

    def _toseq(self, as_unicode):
        strbytes = ida_bytes.get_strlit_contents(self.ea, self.length, self.strtype)
        if sys.version_info.major >= 3:
            return strbytes.decode("UTF-8", "replace") if as_unicode else strbytes
        else:
            return unicode(strbytes, "UTF-8", 'replace') if as_unicode else strbytes

    def __str__(self):
        return self._toseq(False if sys.version_info.major < 3 else True)

    def __unicode__(self):
        return self._toseq(True)

def clear_cache(self):
    """Clears the string list cache"""
    ida_strlist.clear_strlist()

def __init__(self, default_setup = False):
    """
    Initializes the Strings enumeration helper class

    @param default_setup: Set to True to use default setup (C strings, min len 5, ...)
    """
    self.size = 0
    if default_setup:
        self.setup()
    else:
        # restore saved options
        ida_strlist.get_strlist_options()
    self.refresh()

    self._si = ida_strlist.string_info_t()


def refresh(self):
    """Refreshes the string list"""
    ida_strlist.build_strlist()
    self.size = ida_strlist.get_strlist_qty()


def setup(self,
          strtypes = [ida_nalt.STRTYPE_C],
```

```
                minlen = 5,
                only_7bit = True,
                ignore_instructions = False,
                display_only_existing_strings = False):

        t = ida_strlist.get_strlist_options()
        t.strtypes = strtypes
        t.minlen = minlen
        t.only_7bit = only_7bit
        t.display_only_existing_strings = display_only_existing_strings
        t.ignore_heads = ignore_instructions
        self.refresh()


    def _get_item(self, index):
        if not ida_strlist.get_strlist_item(self._si, index):
            return None
        return Strings.StringItem(self._si)


    def __iter__(self):
        return (self._get_item(index) for index in range(0, self.size))


    def __getitem__(self, index):
        """Returns a string item or None"""
        if index >= self.size:
            raise KeyError
        else:
            return self._get_item(index)

# ----------------------------------------------------------------------
def GetIdbDir():
    """
    Get IDB directory

    This function returns directory path of the current IDB database
    """
    return os.path.dirname(ida_loader.get_path(ida_loader.PATH_TYPE_IDB)) + os.sep

# ----------------------------------------------------------------------
def GetRegisterList():
    """Returns the register list"""
    return ida_idp.ph_get_regnames()

# ----------------------------------------------------------------------
```

```
def GetInstructionList():
    """Returns the instruction list of the current processor module"""
    return [i[0] for i in ida_idp.ph_get_instruc() if i[0]]

# ----------------------------------------------------------------------
def Assemble(ea, line):
    """
    Assembles one or more lines (does not display an message dialogs)
    If line is a list then this function will attempt to assemble all the lines
    This function will turn on batch mode temporarily so that no messages are displayed on

    @param ea:       start address
    @return: (False, "Error message") or (True, asm_buf) or (True, [asm_buf1, asm_buf2, asm
    """
    if type(line) in ([bytes] + list(ida_idaapi.string_types)):
        lines = [line]
    else:
        lines = line
    ret = []
    for line in lines:
        seg = ida_segment.getseg(ea)
        if not seg:
            return (False, "No segment at ea")
        ip  = ea - (ida_segment.sel2para(seg.sel) << 4)
        buf = ida_idp.AssembleLine(ea, seg.sel, ip, seg.bitness, line)
        if not buf:
            return (False, "Assembler failed: " + line)
        ea += len(buf)
        ret.append(buf)

    if len(ret) == 1:
        ret = ret[0]
    return (True, ret)

# ----------------------------------------------------------------------
_Assemble = Assemble


# ----------------------------------------------------------------------
def _copy_obj(src, dest, skip_list = None):
    """
    Copy non private/non callable attributes from a class instance to another
    @param src: Source class to copy from
    @param dest: If it is a string then it designates the new class type that will be create
                 Otherwise dest should be an instance of another class
    @return: A new instance or "dest"
```

```
    """
    if type(dest) == bytes:
        # instantiate a new destination class of the specified type name?
        dest = new.classobj(dest, (), {})
    for x in dir(src):
        # skip special and private fields
        if x.startswith("__") and x.endswith("__"):
            continue
        # skip items in the skip list
        if skip_list and x in skip_list:
            continue
        t = getattr(src, x)
        # skip callable
        if callable(t):
            continue
        setattr(dest, x, t)
    return dest

# --------------------------------------------------------------------------
class _reg_dtyp_t(object):
    """
    INTERNAL
    This class describes a register's number and dtyp.
    The equal operator is overloaded so that two instances can be tested for equality
    """
    def __init__(self, reg, dtype):
        self.reg = reg
        self.dtype = dtype

    def __eq__(self, other):
        return (self.reg == other.reg) and (self.dtype == other.dtype)

# --------------------------------------------------------------------------
class _procregs(object):
    """Utility class allowing the users to identify registers in a decoded instruction"""
    def __getattr__(self, attr):
        ri = ida_idp.reg_info_t()
        if not ida_idp.parse_reg_name(ri, attr):
            raise AttributeError()
        r = _reg_dtyp_t(ri.reg, ida_ua.get_dtype_by_size(ri.size))
        self.__dict__[attr] = r
        return r

    def __setattr__(self, attr, value):
        raise AttributeError(attr)
```

```
# ----------------------------------------------------------------------
class _cpu(object):
    "Simple wrapper around get_reg_value/set_reg_value"
    def __getattr__(self, name):
        try:
            return idc.get_reg_value(name)
        except Exception as ex:
            raise AttributeError("_cpu: \"{}\" is not a register;"
                                 " inner exception: [{}] {}"
                                 .format(name, type(ex).__name__, ex))

    def __setattr__(self, name, value):
        return idc.set_reg_value(value, name)


# ----------------------------------------------------------------------
class __process_ui_actions_helper(object):
    def __init__(self, actions, flags = 0):
        """Expect a list or a string with a list of actions"""
        if isinstance(actions, str):
            lst = actions.split(";")
        elif isinstance(actions, (list, tuple)):
            lst = actions
        else:
            raise ValueError("Must pass a string, list or a tuple")

        # Remember the action list and the flags
        self.__action_list = lst
        self.__flags = flags

        # Reset action index
        self.__idx = 0

    def __len__(self):
        return len(self.__action_list)

    def __call__(self):
        if self.__idx >= len(self.__action_list):
            return False

        # Execute one action
        ida_kernwin.process_ui_action(
                self.__action_list[self.__idx],
                self.__flags)
```

```
        # Move to next action
        self.__idx += 1

        # Reschedule
        return True


# -----------------------------------------------------------------------
def ProcessUiActions(actions, flags=0):
    """
    @param actions: A string containing a list of actions separated by semicolon, a list or
    @param flags: flags to be passed to process_ui_action()
    @return: Boolean. Returns False if the action list was empty or execute_ui_requests() fa
    """

    # Instantiate a helper
    helper = __process_ui_actions_helper(actions, flags)
    return False if len(helper) < 1 else ida_kernwin.execute_ui_requests((helper,))


# -----------------------------------------------------------------------
class peutils_t(object):
    """
    PE utility class. Retrieves PE information from the database.

    Constants from pe.h
    """
    PE_NODE = "$ PE header" # netnode name for PE header
    PE_ALT_DBG_FPOS   = ida_idaapi.BADADDR & -1 #  altval() -> translated fpos of debuginfo
    PE_ALT_IMAGEBASE  = ida_idaapi.BADADDR & -2 #  altval() -> loading address (usually pe.
    PE_ALT_PEHDR_OFF  = ida_idaapi.BADADDR & -3 #  altval() -> offset of PE header
    PE_ALT_NEFLAGS    = ida_idaapi.BADADDR & -4 #  altval() -> neflags
    PE_ALT_TDS_LOADED = ida_idaapi.BADADDR & -5 #  altval() -> tds already loaded(1) or inva
    PE_ALT_PSXDLL     = ida_idaapi.BADADDR & -6 #  altval() -> if POSIX(x86) imports from PS

    def __init__(self):
        self.__penode = ida_netnode.netnode()
        self.__penode.create(peutils_t.PE_NODE)

    imagebase = property(lambda self: self.__penode.altval(peutils_t.PE_ALT_IMAGEBASE))
    """Loading address (usually pe.imagebase)"""

    header_offset = property(lambda self: self.__penode.altval(peutils_t.PE_ALT_PEHDR_OFF))
    """Offset of PE header"""

    def __str__(self):
```

```
        return "peutils_t(imagebase=%x, header=%x)" % (self.imagebase, self.header_offset)

    header = lambda self: self.__penode.valobj()
    """Returns the complete PE header as an instance of peheader_t (defined in the SDK)."""

# ----------------------------------------------------------------------
cpu = _cpu()
"""This is a special class instance used to access the registers as if they were attributes
For example to access the EAX register:
    print("%x" % cpu.Eax)
"""

procregs = _procregs()
"""This object is used to access the processor registers. It is useful when decoding instru
For example:
    x = idautils.DecodeInstruction(here())
    if x[0] == procregs.Esp:
        print("This operand is the register ESP)
"""
```

The 'idaapi' module

This module contains all the functionality available to IDAPython as wrapped from the C/C++ SDK. The Python method names from the `idaapi` module are the same as the IDA C/C++ SDK names. The call signatures and function prototypes, however are slightly different due to the automatic SWIG wrapping.

The function descriptions and names are minimal and might contain some keywords for quick reference.

IDAPython function idaapi.Appcall_array___ quick reference

```
This class is used with Appcall.array() method
```

IDAPython function idaapi.Appcall_callable___ quick reference

```
Helper class to issue appcalls using a natural syntax:
  appcall.FunctionNameInTheDatabase(arguments, ....)
or
  appcall["Function@8"](arguments, ...)
or
  f8 = appcall["Function@8"]
  f8(arg1, arg2, ...)
or
  o = appcall.obj()
  i = byref(5)
```

```
  appcall.funcname(arg1, i, "hello", o)
```

IDAPython function idaapi.Appcall__consts___ quick reference

```
Helper class used by Appcall.Consts attribute
It is used to retrieve constants via attribute access
```

IDAPython function idaapi.AssembleLine quick reference

```
AssembleLine(ea, cs, ip, use32, nonnul_line) -> bytes
Assemble an instruction to a string (display a warning if an error is found)

@param ea: linear address of instruction
@param cs: cs of instruction
@param ip: ip of instruction
@param use32: is 32bit segment
@param nonnul_line: char const *
@return:     - None on failure
    - or a string containing the assembled instruction
```

IDAPython function idaapi.BasicBlock quick reference

```
Basic block class. It is returned by the Flowchart class
```

IDAPython function idaapi.COLSTR quick reference

```
Utility function to create a colored line
@param str: The string
@param tag: Color tag constant. One of SCOLOR_XXXX
```

IDAPython function idaapi.Choose quick reference

```
Chooser wrapper class.

Some constants are defined in this class.
Please refer to kernwin.hpp for more information.
```

IDAPython function idaapi.FlowChart quick reference

```
Flowchart class used to determine basic blocks.
Check ex_gdl_qflow_chart.py for sample usage.
```

IDAPython function idaapi.IDAPython_Completion quick reference

```
Internal utility class for auto-completion support
```

IDAPython function idaapi.IDAPython_ExecScript quick reference

```
Run the specified script.

This function is used by the low-level plugin code.
```

IDAPython function idaapi.IDAPython_ExecSystem quick reference

```
Executes a command with popen().
```

IDAPython function idaapi.IDAPython_FormatExc quick reference

```
This function is used to format an exception given the
values returned by a PyErr_Fetch()
```

IDAPython function idaapi.IDAPython_LoadProcMod quick reference

```
Load processor module.
```

IDAPython function idaapi.IDAPython_UnLoadProcMod quick reference

```
Unload processor module.
```

IDAPython function idaapi.IDAViewWrapper quick reference

```
Deprecated. Use View_Hooks instead.

Because the lifecycle of an IDAView is not trivial to track (e.g., a user
might close, then re-open the same disassembly view), this wrapper doesn't
bring anything superior to the View_Hooks: quite the contrary, as the
latter is much more generic (and better maps IDA's internal model.)
```

IDAPython function idaapi.NearestName quick reference

```
Utility class to help find the nearest name in a given ea/name dictionary
```

IDAPython function idaapi.PluginForm quick reference

```
PluginForm class.

This form can be used to host additional controls. Please check the PyQt example.
```

IDAPython function idaapi.PyIdc_cvt_int64___ quick reference

```
Helper class for explicitly representing VT_INT64 values
```

IDAPython function idaapi.PyIdc_cvt_refclass___ quick reference

```
Helper class for representing references to immutable objects
```

IDAPython function idaapi.TRUNC quick reference

```
Truncate EA for the current application bitness
```

IDAPython function idaapi.TWidget___from_ptrval___ quick reference

```
TWidget__from_ptrval__(ptrval) -> TWidget *

@param ptrval: size_t
```

IDAPython function idaapi.accepts_small_udts quick reference

```
accepts_small_udts(op) -> bool
Is the operator allowed on small structure or union?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.accepts_udts quick reference

```
accepts_udts(op) -> bool

@param op: enum ctype_t
```

IDAPython function idaapi.activate_widget quick reference

```
activate_widget(widget, take_focus)
```

```
Activate widget (only gui version) (ui_activate_widget).

@param widget(a Widget SWIG wrapper class): existing widget to display
@param take_focus (bool): give focus to given widget
```

IDAPython function idaapi.add_auto_stkpnt quick reference

```
add_auto_stkpnt(pfn, ea, delta) -> bool
Add automatic SP register change point.

@param pfn (idaapi.func_t): pointer to function. may be nullptr.
@param ea (integer): linear address where SP changes. usually this is the end of the
           instruction which modifies the stack pointer ( insn_t::ea+
           insn_t::size)
@param delta (integer): difference between old and new values of SP
@return: success
```

IDAPython function idaapi.add_bpt quick reference

```
add_bpt(ea, size=0, type=BPT_DEFAULT) -> bool
Add a new breakpoint in the debugged process. \sq{Type, Synchronous function -
available as request, Notification, none (synchronous function)}

@param bpt (idaapi.bpt_t): Breakpoint to add. It describes the break condition, type, fl
            location (module relative, source breakpoint or absolute) and other
            attributes.
@param size: asize_t
@param type: bpttype_t

add_bpt(bpt) -> bool

@param bpt: bpt_t const &
```

IDAPython function idaapi.add_byte quick reference

```
add_byte(ea, value)
Add a value to one byte of the program. This function works for wide byte
processors too.

@param ea (integer): linear address
@param value (integer): byte value
```

IDAPython function idaapi.add_cref quick reference

```
add_cref(frm, to, type) -> bool
Create a code cross-reference.

@param from (integer): linear address of referencing instruction
@param to (integer): linear address of referenced instruction
@param type (one of the idaapi.fl_xxxx flags): cross-reference type
@return: success
```

IDAPython function idaapi.add_dref quick reference

```
add_dref(frm, to, type) -> bool
Create a data cross-reference.

@param from (integer): linear address of referencing instruction or data
@param to (integer): linear address of referenced data
@param type (one of the idaapi.dr_xxxx flags): cross-reference type
@return: success (may fail if user-defined xref exists from->to)
```

IDAPython function idaapi.add_dword quick reference

```
add_dword(ea, value)
Add a value to one dword of the program. This function works for wide byte
processors too. This function takes into account order of bytes specified in
idainfo::is_be()
@note: this function works incorrectly if processor_t::nbits > 16

@param ea (integer): linear address
@param value (integer): byte value
```

IDAPython function idaapi.add_encoding quick reference

```
add_encoding(encname) -> int
Add a new encoding (e.g. "UTF-8"). If it's already in the list, return its
index.

@param encname (string): the encoding name
@return: its index (1-based); -1 means error
```

IDAPython function idaapi.add_entry quick reference

```
add_entry(ord, ea, name, makecode, flags=0) -> bool
```

```
Add an entry point to the list of entry points.

@param ord (integer): ordinal number if ordinal number is equal to 'ea' then ordinal is
           not used
@param ea (integer): linear address
@param name (string): name of entry point. If the specified location already has a name,
            the old name will be appended to the regular comment. If name ==
            nullptr, then the old name will be retained.
@param makecode (bool): should the kernel convert bytes at the entry point to
                instruction(s)
@param flags (integer): See AEF_*
@return: success (currently always true)
```

IDAPython function idaapi.add_enum quick reference

```
add_enum(idx, name, flag) -> enum_t
Add new enum type.
* if idx==BADADDR then add as the last idx
* if name==nullptr then generate a unique name "enum_%d"

@param idx (integer):
@param name (string): char const *
@param flag (integer):
```

IDAPython function idaapi.add_enum_member quick reference

```
add_enum_member(id, name, value, bmask=(bmask_t(-1))) -> int
Add member to enum type.

@param id (integer):
@param name (string): char const *
@param value (integer):
@param bmask (integer):
@return: 0 if ok, otherwise one of Add enum member result codes
```

IDAPython function idaapi.add_extra_cmt quick reference

```
add_extra_cmt(ea, isprev, format) -> bool
Add anterior/posterior comment line(s).

@param ea (integer): linear address
@param isprev (bool): do we add anterior lines? (0-no, posterior)
@param format (string): printf() style format string. may contain \n to denote new lines
               The resulting string should not contain comment characters (;),
```

```
                the kernel will add them automatically.
@return: true if success
```

IDAPython function idaapi.add_extra_line quick reference

```
add_extra_line(ea, isprev, format) -> bool
Add anterior/posterior non-comment line(s).

@param ea (integer): linear address
@param isprev (bool): do we add anterior lines? (0-no, posterior)
@param format (string): printf() style format string. may contain \n to denote new lines
@return: true if success
```

IDAPython function idaapi.add_frame quick reference

```
add_frame(pfn, frsize, frregs, argsize) -> bool
Add function frame.

@param pfn (idaapi.func_t): pointer to function structure
@param frsize (integer): size of function local variables
@param frregs: (C++: ushort) size of saved registers
@param argsize (integer): size of function arguments range which will be purged upon
                return. this parameter is used for __stdcall and __pascal
                calling conventions. for other calling conventions please pass
                0.
@retval 1: ok
@retval 0: failed (no function, frame already exists)
```

IDAPython function idaapi.add_func quick reference

```
add_func(ea1, ea2=BADADDR) -> bool
Add a new function. If the function end address is BADADDR, then IDA will try to
determine the function bounds by calling find_func_bounds(...,
FIND_FUNC_DEFINE).

@param ea1 (integer): start address
@param ea2 (integer): end address
@return: success
```

IDAPython function idaapi.add_func_ex quick reference

```
add_func_ex(pfn) -> bool
Add a new function. If the fn->end_ea is BADADDR, then IDA will try to determine
```

```
the function bounds by calling find_func_bounds(..., FIND_FUNC_DEFINE).

@param pfn (idaapi.func_t): ptr to filled function structure
@return: success
```

IDAPython function idaapi.add_hidden_range quick reference

```
add_hidden_range(ea1, ea2, description, header, footer, color=bgcolor_t(-1)) -> bool
Mark a range of addresses as hidden. The range will be created in the invisible
state with the default color

@param ea1 (integer): linear address of start of the address range
@param ea2 (integer): linear address of end of the address range
@param description (string): ,header,footer: range parameters
@param header (string): char const *
@param footer (string): char const *
@param color (integer): the range color
@return: success
```

IDAPython function idaapi.add_hotkey quick reference

```
add_hotkey(hotkey, pyfunc) -> PyCapsule
Associates a function call with a hotkey.
Callable pyfunc will be called each time the hotkey is pressed

@param hotkey: The hotkey
@param pyfunc: Callable

@return: Context object on success or None on failure.
```

IDAPython function idaapi.add_idc_class quick reference

```
add_idc_class(name, super=None) -> idc_class_t *
Create a new IDC class.

@param name (string): name of the new class
@param super: (C++: const idc_class_t *) the base class for the new class. if the new cl
              any other class, pass nullptr
@return: pointer to the created class. If such a class already exists, a pointer
         to it will be returned. Pointers to other existing classes may be
         invalidated by this call.
```

IDAPython function idaapi.add_idc_func quick reference

```
Extends the IDC language by exposing a new IDC function that is backed up by a Python fu

Add an IDC function. This function does not modify the predefined kernel
functions. Example:
static error_t idaapi myfunc5(idc_value_t *argv, idc_value_t *res)
{
msg("myfunc is called with arg0=%a and arg1=%s\n", argv[0].num, argv[1].str);
res->num = 5;     // let's return 5
return eOk;
}
static const char myfunc5_args[] = { VT_LONG, VT_STR, 0 };
static const ext_idcfunc_t myfunc_desc = { "MyFunc5", myfunc5, myfunc5_args,
nullptr, 0, EXTFUN_BASE };
// after this:
add_idc_func(myfunc_desc);
// there is a new IDC function which can be called like this:
MyFunc5(0x123, "test");

@note: If the function already exists, it will be replaced by the new function
@return: success
```

IDAPython function idaapi.add_idc_gvar quick reference

```
add_idc_gvar(name) -> idc_value_t
Add global IDC variable.

@param name (string): name of the global variable
@return: pointer to the created variable or existing variable. NB: the returned
         pointer is valid until a new global var is added.
```

IDAPython function idaapi.add_idc_hotkey quick reference

```
add_idc_hotkey(hotkey, idcfunc) -> int
Add hotkey for IDC function (ui_add_idckey).

@param hotkey (string): hotkey name
@param idcfunc (string): IDC function name
@return: IDC hotkey error codes
```

IDAPython function idaapi.add_mapping quick reference

```
add_mapping(_from, to, size) -> bool
```

```
IDA supports memory mapping. References to the addresses from the mapped range
use data and meta-data from the mapping range.
@note: You should set flag PR2_MAPPING in ph.flag2 to use memory mapping Add
       memory mapping range.

@param from (integer): start of the mapped range (nonexistent address)
@param to (integer): start of the mapping range (existent address)
@param size (integer): size of the range
@return: success
```

IDAPython function idaapi.add_path_mapping quick reference

```
add_path_mapping(src, dst)

@param src: char const *
@param dst: char const *
```

IDAPython function idaapi.add_pgm_cmt quick reference

```
add_pgm_cmt(format) -> bool
Add anterior comment line(s) at the start of program.

@param format (string): printf() style format string. may contain \n to denote new lines
               The resulting string should not contain comment characters (;),
               the kernel will add them automatically.
@return: true if success
```

IDAPython function idaapi.add_qword quick reference

```
add_qword(ea, value)
Add a value to one qword of the program. This function does not work for wide
byte processors. This function takes into account order of bytes specified in
idainfo::is_be()

@param ea (integer): linear address
@param value (integer): byte value
```

IDAPython function idaapi.add_refinfo_dref quick reference

```
add_refinfo_dref(insn, _from, ri, opval, type, opoff) -> ea_t
Add xrefs for a reference from the given instruction ( insn_t::ea). This
function creates a cross references to the target and the base.
insn_t::add_off_drefs() calls this function to create xrefs for 'offset'
```

```
operand.

@param insn: (C++: const insn_t &) the referencing instruction
@param from (integer): the referencing instruction/data address
@param ri: (C++: const refinfo_t &) reference info block from the database
@param opval (integer): operand value (usually op_t::value or op_t::addr)
@param type (one of the idaapi.dr_xxxx flags): type of xref
@param opoff (integer): offset of the operand from the start of instruction
@return: the target address of the reference
```

IDAPython function idaapi.add_regarg quick reference

```
add_regarg(pfn, reg, tif, name)

@param pfn: func_t *
@param reg: int
@param tif: tinfo_t const &
@param name: char const *
```

IDAPython function idaapi.add_regvar quick reference

```
add_regvar(pfn, ea1, ea2, canon, user, cmt) -> int
Define a register variable.

@param pfn (idaapi.func_t): function in which the definition will be created
@param ea1 (integer): ,ea2: range of addresses within the function where the definition
                be used
@param canon (string): name of a general register
@param canon (string): name of a general register
@param user (string): user-defined name for the register
@param cmt (string): comment for the definition
@return: Register variable error codes
```

IDAPython function idaapi.add_segm quick reference

```
add_segm(para, start, end, name, sclass, flags=0) -> bool
Add a new segment, second form. Segment alignment is set to saRelByte. Segment
combination is "public" or "stack" (if segment class is "STACK"). Addressing
mode of segment is taken as default (16bit or 32bit). Default segment registers
are set to BADSEL. If a segment already exists at the specified range of
addresses, this segment will be truncated. Instructions and data in the old
segment will be deleted if the new segment has another addressing mode or
another segment base address.
```

```
@param para (integer): segment base paragraph. if paragraph can't fit in 16bit, then a r
             selector is allocated and mapped to the paragraph.
@param start (integer): start address of the segment. if start==BADADDR then start <-
              to_ea(para,0).
@param end (integer): end address of the segment. end address should be higher than star
            address. For emulate empty segments, use SEG_NULL segment type. If
            the end address is lower than start address, then fail. If
            end==BADADDR, then a segment up to the next segment will be created
            (if the next segment doesn't exist, then 1 byte segment will be
            created). If 'end' is too high and the new segment would overlap the
            next segment, 'end' is adjusted properly.
@param name (string): name of new segment. may be nullptr
@param sclass (string): class of the segment. may be nullptr. type of the new segment is
               modified if class is one of predefined names:
* "CODE" -> SEG_CODE
* "DATA" -> SEG_DATA
* "CONST" -> SEG_DATA
* "STACK" -> SEG_BSS
* "BSS" -> SEG_BSS
* "XTRN" -> SEG_XTRN
* "COMM" -> SEG_COMM
* "ABS" -> SEG_ABSSYM
@param flags (integer): Add segment flags
@retval 1: ok
@retval 0: failed, a warning message is displayed
```

IDAPython function idaapi.add_segm_ex quick reference

```
add_segm_ex(NONNULL_s, name, sclass, flags) -> bool
Add a new segment. If a segment already exists at the specified range of
addresses, this segment will be truncated. Instructions and data in the old
segment will be deleted if the new segment has another addressing mode or
another segment base address.

@param NONNULL_s: (C++: segment_t *)
@param name (string): name of new segment. may be nullptr. if specified, the segment is
             immediately renamed
@param sclass (string): class of the segment. may be nullptr. if specified, the segment
               class is immediately changed
@param flags (integer): Add segment flags
@retval 1: ok
@retval 0: failed, a warning message is displayed
```

IDAPython function idaapi.add_segment_translation quick reference

```
add_segment_translation(segstart, mappedseg) -> bool
Add segment translation.

@param segstart (integer): start address of the segment to add translation to
@param mappedseg (integer): start address of the overlayed segment
@retval 1: ok
@retval 0: too many translations or bad segstart
```

IDAPython function idaapi.add_sourcefile quick reference

```
add_sourcefile(ea1, ea2, filename) -> bool
Mark a range of address as belonging to a source file. An address range may
belong only to one source file. A source file may be represented by several
address ranges.

@param ea1 (integer): linear address of start of the address range
@param ea2 (integer): linear address of end of the address range (excluded)
@param filename (string): name of source file.
@return: success
```

IDAPython function idaapi.add_spaces quick reference

```
add_spaces(s, len) -> str
Add space characters to the colored string so that its length will be at least
'len' characters. Don't trim the string if it is longer than 'len'.

@param str: (C++: char *) pointer to colored string to modify (may not be nullptr)
@param len: (C++: ssize_t) the desired length of the string
@return: pointer to the end of input string
```

IDAPython function idaapi.add_struc quick reference

```
add_struc(idx, name, is_union=False) -> tid_t
Create a structure type. if idx==BADADDR then add as the last idx. if
name==nullptr then a name will be generated "struct_%d".

@param idx (integer):
@param name (string): char const *
@param is_union (bool):
```

IDAPython function idaapi.add_struc_member quick reference

```
add_struc_member(sptr, fieldname, offset, flag, mt, nbytes) -> struc_error_t
Add member to existing structure.

@param sptr: (C++: struc_t *) structure to modify
@param fieldname (string): if nullptr, then "anonymous_#" name will be generated
@param offset (integer): BADADDR means add to the end of structure
@param flag (integer): type + representation bits
@param mt: (C++: const opinfo_t *) additional info about member type. must be present fo
           offsets, enums, strings, struct offsets.
@param nbytes (integer): if == 0 then the structure will be a varstruct. in this case th
               member should be the last member in the structure
```

IDAPython function idaapi.add_til quick reference

```
add_til(name, flags) -> int
Load a til file and add it the database type libraries list. IDA will also apply
function prototypes for matching function names.

@param name (string): til name
@param flags (integer): combination of Load TIL flags
@return: one of Load TIL result codes
```

IDAPython function idaapi.add_tryblk quick reference

```
add_tryblk(tb) -> int
Add one try block information.

@param tb: (C++: const tryblk_t &) try block to add.
@return: error code; 0 means good
```

IDAPython function idaapi.add_user_stkpnt quick reference

```
add_user_stkpnt(ea, delta) -> bool
Add user-defined SP register change point.

@param ea (integer): linear address where SP changes
@param delta (integer): difference between old and new values of SP
@return: success
```

IDAPython function idaapi.add_virt_module quick reference

```
add_virt_module(mod) -> bool

@param mod: modinfo_t const *
```

IDAPython function idaapi.add_word quick reference

```
add_word(ea, value)
Add a value to one word of the program. This function works for wide byte
processors too. This function takes into account order of bytes specified in
idainfo::is_be()

@param ea (integer): linear address
@param value (integer): byte value
```

IDAPython function idaapi.addon_count quick reference

```
addon_count() -> int
Get number of installed addons.
```

IDAPython function idaapi.align_flag quick reference

```
align_flag() -> flags64_t
Get a flags64_t representing an alignment directive.
```

IDAPython function idaapi.alloc_type_ordinal quick reference

```
alloc_type_ordinal(ti) -> uint32
alloc_type_ordinals(ti, 1)

@param ti (idaapi.til_t):
```

IDAPython function idaapi.alloc_type_ordinals quick reference

```
alloc_type_ordinals(ti, qty) -> uint32
Allocate a range of ordinal numbers for new types.

@param ti (idaapi.til_t): type library
@param qty (integer): number of ordinals to allocate
@return: the first ordinal. 0 means failure.
```

IDAPython function idaapi.allocate_selector quick reference

```
allocate_selector(segbase) -> sel_t
Allocate a selector for a segment unconditionally. You must call this function
before calling add_segm_ex(). add_segm() calls this function itself, so you
don't need to allocate a selector. This function will allocate a new free
selector and setup its mapping using find_free_selector() and set_selector()
functions.

@param segbase (integer): a new segment base paragraph
@return: the allocated selector number
```

IDAPython function idaapi.analyzer_options quick reference

```
analyzer_options()
Allow the user to set analyzer options. (show a dialog box)
(ui_analyzer_options)
```

IDAPython function idaapi.appcall quick reference

```
appcall(func_ea, tid, _type_or_none, _fields, arg_list) -> PyObject *

@param func_ea: ea_t
@param tid: thid_t
@param _type_or_none: bytevec_t const &
@param _fields: bytevec_t const &
@param arg_list: PyObject *
```

IDAPython function idaapi.append_abi_opts quick reference

```
append_abi_opts(abi_opts, user_level=False) -> bool
Add/remove/check ABI option General form of full abi name: abiname-opt1-opt2-...
or -opt1-opt2-...

@param abi_opts (string): - ABI options to add/remove in form opt1-opt2-...
@param user_level (bool): - initiated by user if TRUE (==SETCOMP_BY_USER)
@return: success
```

IDAPython function idaapi.append_argloc quick reference

```
append_argloc(out, vloc) -> bool
Serialize argument location
```

```
@param out: (C++: qtype *)
@param vloc: (C++: const argloc_t &) argloc_t const &
```

IDAPython function idaapi.append_cmt quick reference

```
append_cmt(ea, str, rptble) -> bool
Append to an indented comment. Creates a new comment if none exists. Appends a
newline character and the specified string otherwise.

@param ea (integer): linear address
@param str (string): comment string to append
@param rptble (bool): append to repeatable comment?
@return: success
```

IDAPython function idaapi.append_func_tail quick reference

```
append_func_tail(pfn, ea1, ea2) -> bool
Append a new tail chunk to the function definition. If the tail already exists,
then it will simply be added to the function tail list Otherwise a new tail will
be created and its owner will be set to be our function If a new tail cannot be
created, then this function will fail.

@param pfn (idaapi.func_t): pointer to the function
@param ea1 (integer): start of the tail. If a tail already exists at the specified addre
            it must start at 'ea1'
@param ea2 (integer): end of the tail. If a tail already exists at the specified address
            it must end at 'ea2'. If specified as BADADDR, IDA will determine
            the end address itself.
```

IDAPython function idaapi.append_struct_fields quick reference

```
append_struct_fields(disp, n, path, flags, delta, appzero) -> str
Append names of struct fields to a name if the name is a struct name.

@param disp: (C++: adiff_t *) displacement from the name
@param n (integer): operand number in which the name appears
@param path: (C++: const tid_t *) path in the struct. path is an array of id's. maximal
             array is MAXSTRUCPATH. the first element of the array is the
             structure id. consecutive elements are id's of used union members
             (if any).
@param flags (integer): the input flags. they will be returned if the struct cannot be
              found.
@param delta (integer): delta to add to displacement
@param appzero (bool): should append a struct field name if the displacement is zero?
```

```
@return: flags of the innermost struct member or the input flags
```

IDAPython function idaapi.append_tinfo_covered quick reference

```
append_tinfo_covered(out, typid, offset) -> bool

@param out: rangeset_t *
@param typid: uint32
@param offset: uint64
```

IDAPython function idaapi.apply_callee_tinfo quick reference

```
apply_callee_tinfo(caller, tif) -> bool
Apply the type of the called function to the calling instruction. This function
will append parameter comments and rename the local variables of the calling
function. It also stores information about the instructions that initialize call
arguments in the database. Use get_arg_addrs() to retrieve it if necessary.
Alternatively it is possible to hook to processor_t::arg_addrs_ready event.

@param caller (integer): linear address of the calling instruction. must belong to a
               function.
@param tif (idaapi.tinfo_t): type info
@return: success
```

IDAPython function idaapi.apply_cdecl quick reference

```
apply_cdecl(til, ea, decl, flags=0) -> bool
Apply the specified type to the address. This function parses the declaration
and calls apply_tinfo()

@param til (idaapi.til_t): type library
@param ea (integer): linear address
@param decl (string): type declaration in C form
@param flags (integer): flags to pass to apply_tinfo (TINFO_DEFINITE is always passed)
@return: success
```

IDAPython function idaapi.apply_idasgn_to quick reference

```
apply_idasgn_to(signame, ea, is_startup) -> int
Apply a signature file to the specified address.

@param signame (string): short name of signature file (the file name without path)
@param ea (integer): address to apply the signature
```

```
@param is_startup (bool): if set, then the signature is treated as a startup one for
                  startup signature ida doesn't rename the first function of
                  the applied module.
@return: Library function codes
```

IDAPython function idaapi.apply_named_type quick reference

```
apply_named_type(ea, name) -> bool
Apply the specified named type to the address.

@param ea (integer): linear address
@param name (string): the type name, e.g. "FILE"
@return: success
```

IDAPython function idaapi.apply_once_tinfo_and_name quick reference

```
apply_once_tinfo_and_name(dea, tif, name) -> bool
Apply the specified type and name to the address. This function checks if the
address already has a type. If the old type
does not exist or the new type is 'better' than the old type, then the
new type will be applied. A type is considered better if it has more
information (e.g. BTMT_STRUCT is better than BT_INT).
The same logic is with the name: if the address already have a meaningful
name, it will be preserved. Only if the old name does not exist or it
is a dummy name like byte_123, it will be replaced by the new name.

@param dea (integer): linear address
@param tif (idaapi.tinfo_t): type string in the internal format
@param name (string): new name for the address
@return: success
```

IDAPython function idaapi.apply_startup_sig quick reference

```
apply_startup_sig(ea, startup) -> bool
Apply a startup signature file to the specified address.

@param ea (integer): address to apply the signature to; usually idainfo::start_ea
@param startup (string): the name of the signature file without path and extension
@return: true if successfully applied the signature
```

IDAPython function idaapi.apply_tinfo quick reference

```
apply_tinfo(ea, tif, flags) -> bool
Apply the specified type to the specified address. This function sets the type
and tries to convert the item at the specified address to conform the type.

@param ea (integer): linear address
@param tif (idaapi.tinfo_t): type string in internal format
@param flags (integer): combination of Apply tinfo flags
@return: success
```

IDAPython function idaapi.apply_tinfo_to_stkarg quick reference

```
apply_tinfo_to_stkarg(insn, x, v, tif, name) -> bool
Helper function for the processor modules. to be called from
processor_t::use_stkarg_type

@param insn: (C++: const insn_t &) an idaapi.insn_t, or an address (C++: const insn_t &)
@param x: (C++: const op_t &) op_t const &
@param v (integer):
@param tif (idaapi.tinfo_t): tinfo_t const &
@param name (string): char const *
```

IDAPython function idaapi.apply_type quick reference

```
apply_type(ti, type, fields, ea, flags) -> bool
Apply the specified type to the address

@param ti: Type info library. 'None' can be used.
@param type: type_t const *
@param fields: p_list const *
@param ea: the address of the object
@param flags: combination of TINFO_... constants or 0
@return: Boolean
```

IDAPython function idaapi.arglocs_overlap quick reference

```
arglocs_overlap(loc1, w1, loc2, w2) -> bool
Do two arglocs overlap?

@param loc1: (C++: const vdloc_t &) vdloc_t const &
@param w1 (integer):
@param loc2: (C++: const vdloc_t &) vdloc_t const &
@param w2 (integer):
```

IDAPython function idaapi.as_UTF16 quick reference

```
Convenience function to convert a string into appropriate unicode format
```

IDAPython function idaapi.as_cstr quick reference

```
Returns a C str from the passed value. The passed value can be of type refclass (returne
It scans for the first \x00 and returns the string value up to that point.
```

IDAPython function idaapi.as_int32 quick reference

```
Returns a number as a signed int32 number
```

IDAPython function idaapi.as_signed quick reference

```
Returns a number as signed. The number of bits are specified by the user.
The MSB holds the sign.
```

IDAPython function idaapi.as_uint32 quick reference

```
Returns a number as an unsigned int32 number
```

IDAPython function idaapi.as_unicode quick reference

```
Convenience function to convert a string into appropriate unicode format
```

IDAPython function idaapi.asgop quick reference

```
asgop(cop) -> ctype_t
Convert plain operator into assignment operator. For example, cot_add returns
cot_asgadd.

@param cop: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.asgop_revert quick reference

```
asgop_revert(cop) -> ctype_t
Convert assignment operator into plain operator. For example, cot_asgadd returns
cot_add
```

```
@param cop: (C++: ctype_t) enum ctype_t
@return: cot_empty is the input operator is not an assignment operator.
```

IDAPython function idaapi.ask_addr quick reference

```
Output a formatted string to the output window (msg) preprended with "**DATABASE
IS CORRUPTED: " Display a dialog box and wait for the user to input an address
(ui_ask_addr).

@retval 0: the user pressed Esc.
@retval 1: ok, the user entered an address
```

IDAPython function idaapi.ask_buttons quick reference

```
ask_buttons(Yes, No, Cancel, deflt, format) -> int
Display a dialog box and get choice from maximum three possibilities
(ui_ask_buttons).
@note: for all buttons:
* use "" or nullptr to take the default name for the button.
* prepend "HIDECANCEL\n" in 'format' to hide the Cancel button

@param Yes (string): text for the first button
@param No (string): text for the second button
@param Cancel (string): text for the third button
@param deflt (integer): default choice: one of Button IDs
@param format (string): printf-style format string for question. It may have some
               prefixes, see below.
@return: one of Button IDs specifying the selected button (Esc key returns
         Cancel/3rd button value)
```

IDAPython function idaapi.ask_file quick reference

```
ask_file(for_saving, defval, format) -> char *

@param for_saving: bool
@param defval: char const *
@param format: char const *
```

IDAPython function idaapi.ask_for_feedback quick reference

```
ask_for_feedback(format)
Show a message box asking to send the input file to \link{mailto:support@hex-
```

```
rays.com,support@hex-rays.com}.

@param format (string): the reason why the input file is bad
```

IDAPython function idaapi.ask_form quick reference

```
Display a dialog box and wait for the user. If the form contains the "BUTTON NO
<title>" keyword, then the return values are the same as in the ask_yn()
function (Button IDs)

@retval 0: no memory to display or form syntax error (a warning is displayed in
           this case). the user pressed the 'No' button (if the form has it) or
           the user cancelled the dialog otherwise. all variables retain their
           original values.
@retval 1: ok, all input fields are filled and validated.
@retval -1: the form has the 'No' button and the user cancelled the dialog
```

IDAPython function idaapi.ask_ident quick reference

```
Display a dialog box and wait for the user to input an identifier. If the user
enters a non-valid identifier, this function displays a warning and allows the
user to correct it.

@return: false if the user cancelled the dialog, otherwise returns true.
```

IDAPython function idaapi.ask_long quick reference

```
Display a dialog box and wait for the user to input an number (ui_ask_long). The
number is represented in C-style. This function allows to enter any IDC
expression and properly calculates it.

@retval 0: if the user pressed Esc.
@retval 1: ok, the user entered a valid number.
```

IDAPython function idaapi.ask_seg quick reference

```
Display a dialog box and wait for the user to input an segment name
(ui_ask_seg). This function allows to enter segment register names, segment base
paragraphs, segment names to denote a segment.

@retval 0: if the user pressed Esc.
@retval 1: ok, the user entered an segment name
```

IDAPython function idaapi.ask_str quick reference

```
ask_str(defval, hist, prompt) -> str or None
Asks for a long text

@param defval: The default value
@param hist: history id
@param prompt: The prompt value
@return: None or the entered string
```

IDAPython function idaapi.ask_text quick reference

```
ask_text(max_size, defval, prompt) -> str
Asks for a long text

@param max_size: Maximum text length, 0 for unlimited
@param defval: The default value
@param prompt: The prompt value
@return: None or the entered string
```

IDAPython function idaapi.ask_yn quick reference

```
ask_yn(deflt, format) -> int
Display a dialog box and get choice from "Yes", "No", "Cancel".

@param deflt (integer): default choice: one of Button IDs
@param format (string): The question in printf() style format
@return: the selected button (one of Button IDs). Esc key returns ASKBTN_CANCEL.
```

IDAPython function idaapi.assemble quick reference

```
assemble(ea, cs, ip, use32, line) -> bool
Assemble an instruction into the database (display a warning if an error is found)

@param ea: linear address of instruction
@param cs: cs of instruction
@param ip: ip of instruction
@param use32: is 32bit segment?
@param line: line to assemble

@return: Boolean. True on success.
```

IDAPython function idaapi.atoea quick reference

```
atoea(str) -> bool
Convert a number in C notation to an address. decimal: 1234
octal: 0123
hexadecimal: 0xabcd
binary: 0b00101010

@param str (string): the string to parse
```

IDAPython function idaapi.attach_action_to_menu quick reference

```
attach_action_to_menu(menupath, name, flags=0) -> bool
Attach a previously-registered action to the menu (ui_attach_action_to_menu).
@note: You should not change top level menu, or the Edit,Plugins submenus If you
       want to modify the debugger menu, do it at the ui_debugger_menu_change
       event (ida might destroy your menu item if you do it elsewhere).

@param menupath (string): path to the menu item after or before which the insertion will
                take place.
* Example: Debug/StartProcess
* Whitespace, punctuation are ignored.
* It is allowed to specify only the prefix of the menu item.
* Comparison is case insensitive.
* menupath may start with the following prefixes:
* [S] - modify the main menu of the structure window
* [E] - modify the main menu of the enum window
@param name (string): the action name
@param flags (integer): a combination of Set menu flags, to determine menu item position
@return: success
```

IDAPython function idaapi.attach_action_to_popup quick reference

```
attach_action_to_popup(widget, popup_handle, name, popuppath=None, flags=0) -> bool
Insert a previously-registered action into the widget's popup menu
(ui_attach_action_to_popup). This function has two "modes": 'single-shot', and
'permanent'.

@param widget(a Widget SWIG wrapper class): target widget
@param popup_handle: (C++: TPopupMenu *) target popup menu
* if non-nullptr, the action is added to this popup menu invocation (i.e.,
'single-shot')
* if nullptr, the action is added to a list of actions that should always be
present in context menus for this widget (i.e., 'permanent'.)
```

```
@param name (string): action name
@param popuppath (string): can be nullptr
@param flags (integer): a combination of SETMENU_ flags (see Set menu flags)
@return: success
```

IDAPython function idaapi.attach_action_to_toolbar quick reference

```
attach_action_to_toolbar(toolbar_name, name) -> bool
Attach an action to an existing toolbar (ui_attach_action_to_toolbar).

@param toolbar_name (string): the name of the toolbar
@param name (string): the action name
@return: success
```

IDAPython function idaapi.attach_custom_data_format quick reference

```
attach_custom_data_format(dtid, dfid) -> bool
Attach the data format to the data type.

@param dtid (integer): data type id that can use the data format. 0 means all standard
            data types. Such data formats can be applied to any data item or
            instruction operands. For instruction operands, the
            data_format_t::value_size check is not performed by the kernel.
@param dfid (integer): data format id
@retval true: ok
@retval false: no such `dtid', or no such `dfid', or the data format has already
              been attached to the data type
```

IDAPython function idaapi.attach_dynamic_action_to_popup quick reference

```
attach_dynamic_action_to_popup(unused, popup_handle, desc, popuppath=None, flags=0) -> b
Create & insert an action into the widget's popup menu
(::ui_attach_dynamic_action_to_popup).
Note: The action description in the 'desc' parameter is modified by
      this call so you should prepare a new description for each call.
For example:
    desc = idaapi.action_desc_t(None, 'Dynamic popup action', Handler())
    idaapi.attach_dynamic_action_to_popup(form, popup, desc)

@param unused: deprecated; should be None
@param popup_handle: target popup
```

```
@param desc: action description of type action_desc_t
@param popuppath: can be None
@param flags: a combination of SETMENU_ constants
@return: success
```

IDAPython function idaapi.attach_process quick reference

```
attach_process(pid=pid_t(-1), event_id=-1) -> int
Attach the debugger to a running process. \sq{Type, Asynchronous function -
available as Request, Notification, dbg_process_attach}
@note: This function shouldn't be called as a request if NO_PROCESS is used.

@param pid (integer): PID of the process to attach to. If NO_PROCESS, a dialog box will
            interactively ask the user for the process to attach to.
@retval -4: debugger was not inited
@retval -3: the attaching is not supported
@retval -2: impossible to find a compatible process
@retval -1: impossible to attach to the given process (process died, privilege
            needed, not supported by the debugger plugin, ...)
@retval 0: the user cancelled the attaching to the process
@retval 1: the debugger properly attached to the process
```

IDAPython function idaapi.auto_apply_tail quick reference

```
auto_apply_tail(tail_ea, parent_ea)
Plan to apply the tail_ea chunk to the parent

@param tail_ea (integer): linear address of start of tail
@param parent_ea (integer): linear address within parent. If BADADDR, automatically try
                 find parent via xrefs.
```

IDAPython function idaapi.auto_apply_type quick reference

```
auto_apply_type(caller, callee)
Plan to apply the callee's type to the calling point.

@param caller (integer):
@param callee (integer):
```

IDAPython function idaapi.auto_cancel quick reference

```
auto_cancel(ea1, ea2)
Remove an address range (ea1..ea2) from queues AU_CODE, AU_PROC, AU_USED. To
```

```
remove an address range from other queues use auto_unmark() function. 'ea1' may
be higher than 'ea2', the kernel will swap them in this case. 'ea2' doesn't
belong to the range.

@param ea1 (integer):
@param ea2 (integer):
```

IDAPython function idaapi.auto_get quick reference

```
auto_get(type, lowEA, highEA) -> ea_t
Retrieve an address from queues regarding their priority. Returns BADADDR if no
addresses not lower than 'lowEA' and less than 'highEA' are found in the queues.
Otherwise *type will have queue type.

@param type: (C++: atype_t *)
@param lowEA (integer):
@param highEA (integer):
```

IDAPython function idaapi.auto_is_ok quick reference

```
auto_is_ok() -> bool
Are all queues empty? (i.e. has autoanalysis finished?).
```

IDAPython function idaapi.auto_make_code quick reference

```
auto_make_code(ea)
Plan to make code.

@param ea (integer):
```

IDAPython function idaapi.auto_make_proc quick reference

```
auto_make_proc(ea)
Plan to make code&function.

@param ea (integer):
```

IDAPython function idaapi.auto_make_step quick reference

```
auto_make_step(ea1, ea2) -> bool
Analyze one address in the specified range and return true.
```

```
@param ea1 (integer):
@param ea2 (integer):
@return: if processed anything. false means that there is nothing to process in
         the specified range.
```

IDAPython function idaapi.auto_mark quick reference

```
auto_mark(ea, type)
Put single address into a queue. Queues keep addresses sorted.

@param ea (integer):
@param type (one of the idaapi.AU_xxxx flags):
```

IDAPython function idaapi.auto_mark_range quick reference

```
auto_mark_range(start, end, type)
Put range of addresses into a queue. 'start' may be higher than 'end', the
kernel will swap them in this case. 'end' doesn't belong to the range.

@param start (integer):
@param end (integer):
@param type (one of the idaapi.AU_xxxx flags):
```

IDAPython function idaapi.auto_recreate_insn quick reference

```
auto_recreate_insn(ea) -> int
Try to create instruction

@param ea (integer): linear address of callee
@return: the length of the instruction or 0
```

IDAPython function idaapi.auto_unmark quick reference

```
auto_unmark(start, end, type)
Remove range of addresses from a queue. 'start' may be higher than 'end', the
kernel will swap them in this case. 'end' doesn't belong to the range.

@param start (integer):
@param end (integer):
@param type (one of the idaapi.AU_xxxx flags):
```

IDAPython function idaapi.auto_wait quick reference

```
auto_wait() -> bool
Process everything in the queues and return true.

@return: false if the user clicked cancel. (the wait box must be displayed by
         the caller if desired)
```

IDAPython function idaapi.auto_wait_range quick reference

```
auto_wait_range(ea1, ea2) -> ssize_t
Process everything in the specified range and return true.

@param ea1 (integer):
@param ea2 (integer):
@return: number of autoanalysis steps made. -1 if the user clicked cancel. (the
         wait box must be displayed by the caller if desired)
```

IDAPython function idaapi.banner quick reference

```
banner(wait) -> bool
Show a banner dialog box (ui_banner).

@param wait (integer): time to wait before closing
@retval 1: ok
@retval 0: esc was pressed
```

IDAPython function idaapi.base2file quick reference

```
base2file(fp, pos, ea1, ea2) -> int
Unload database to a binary file. This function works for wide byte processors
too.

@param fp: (C++: FILE *) pointer to file
@param pos: (C++: qoff64_t) position in the file
@param ea1 (integer): ,ea2: range of source linear addresses
@param ea2 (integer):
@return: 1-ok(always), write error leads to immediate exit
```

IDAPython function idaapi.beep quick reference

```
beep(beep_type=beep_default)
Issue a beeping sound (ui_beep).
```

```
@param beep_type: (C++: beep_t)
```

IDAPython function idaapi.begin_type_updating quick reference

```
begin_type_updating(utp)
Mark the beginning of a large update operation on the types. Can be used with
add_enum_member(), add_struc_member, etc... Also see end_type_updating()

@param utp: (C++: update_type_t) enum update_type_t
```

IDAPython function idaapi.bin_flag quick reference

```
bin_flag() -> flags64_t
Get number flag of the base, regardless of current processor - better to use
num_flag()
```

IDAPython function idaapi.bin_search quick reference

```
bin_search(start_ea, end_ea, data, flags) -> ea_t
Search for a set of bytes in the program

@param start_ea: linear address, start of range to search
@param end_ea: linear address, end of range to search (exclusive)
@param data: the prepared data to search for (see parse_binpat_str())
@param flags: combination of BIN_SEARCH_* flags
@return: the address of a match, or idaapi.BADADDR if not found
bin_search(start_ea, end_ea, image, imask, step, flags) -> ea_t

@param start_ea: ea_t
@param end_ea: ea_t
@param image: bytevec_t const &
@param imask: bytevec_t const &
@param step: int
@param flags: int
```

IDAPython function idaapi.bin_search3 quick reference

```
bin_search3(start_ea, end_ea, data, flags) -> ea_t
Search for a patter in the program.

@param start_ea (integer): linear address, start of range to search
@param end_ea (integer): linear address, end of range to search (exclusive)
```

```
@param data: (C++: const compiled_binpat_vec_t &) the prepared data to search for (see p
@param flags (integer): combination of Search flags
@return: BADADDR (if pressed Ctrl-Break or not found) or pattern address.
```

IDAPython function idaapi.block_chains_begin quick reference

```
block_chains_begin(set) -> block_chains_iterator_t
Get iterator pointing to the beginning of block_chains_t.

@param set: (C++: const block_chains_t *) block_chains_t const *
```

IDAPython function idaapi.block_chains_clear quick reference

```
block_chains_clear(set)
Clear block_chains_t.

@param set: (C++: block_chains_t *)
```

IDAPython function idaapi.block_chains_end quick reference

```
block_chains_end(set) -> block_chains_iterator_t
Get iterator pointing to the end of block_chains_t.

@param set: (C++: const block_chains_t *) block_chains_t const *
```

IDAPython function idaapi.block_chains_erase quick reference

```
block_chains_erase(set, p)
Erase current element from block_chains_t.

@param set: (C++: block_chains_t *)
@param p: (C++: block_chains_iterator_t)
```

IDAPython function idaapi.block_chains_find quick reference

```
block_chains_find(set, val) -> block_chains_iterator_t
Find the specified key in set block_chains_t.

@param set: (C++: const block_chains_t *) block_chains_t const *
@param val: (C++: const chain_t &) chain_t const &
```

IDAPython function idaapi.block_chains_free quick reference

```
block_chains_free(set)
Delete block_chains_t instance.

@param set: (C++: block_chains_t *)
```

IDAPython function idaapi.block_chains_get quick reference

```
block_chains_get(p) -> chain_t
Get reference to the current set value.

@param p: (C++: block_chains_iterator_t)
```

IDAPython function idaapi.block_chains_insert quick reference

```
block_chains_insert(set, val) -> block_chains_iterator_t
Insert new (chain_t) into set block_chains_t.

@param set: (C++: block_chains_t *)
@param val: (C++: const chain_t &) chain_t const &
```

IDAPython function idaapi.block_chains_new quick reference

```
block_chains_new() -> block_chains_t
Create a new block_chains_t instance.
```

IDAPython function idaapi.block_chains_next quick reference

```
block_chains_next(p) -> block_chains_iterator_t
Move to the next element.

@param p: (C++: block_chains_iterator_t)
```

IDAPython function idaapi.block_chains_prev quick reference

```
block_chains_prev(p) -> block_chains_iterator_t
Move to the previous element.

@param p: (C++: block_chains_iterator_t)
```

IDAPython function idaapi.block_chains_size quick reference

```
block_chains_size(set) -> size_t
Get size of block_chains_t.

@param set: (C++: block_chains_t *)
```

IDAPython function idaapi.bookmarks_t_erase quick reference

```
bookmarks_t_erase(e, index, ud) -> bool

@param e: lochist_entry_t const &
@param index: uint32
@param ud
```

IDAPython function idaapi.bookmarks_t_find_index quick reference

```
bookmarks_t_find_index(e, ud) -> uint32

@param e: lochist_entry_t const &
@param ud
```

IDAPython function idaapi.bookmarks_t_get quick reference

```
bookmarks_t_get(out, _index, ud) -> PyObject *

@param out: lochist_entry_t *
@param _index: uint32
@param ud
```

IDAPython function idaapi.bookmarks_t_get_desc quick reference

```
bookmarks_t_get_desc(e, index, ud) -> str

@param e: lochist_entry_t const &
@param index: uint32
@param ud
```

IDAPython function idaapi.bookmarks_t_get_dirtree_id quick reference

```
bookmarks_t_get_dirtree_id(e, ud) -> dirtree_id_t
```

```
@param e: lochist_entry_t const &
@param ud
```

IDAPython function idaapi.bookmarks_t_mark quick reference

```
bookmarks_t_mark(e, index, title, desc, ud) -> uint32

@param e: lochist_entry_t const &
@param index: uint32
@param title: char const *
@param desc: char const *
@param ud
```

IDAPython function idaapi.bookmarks_t_size quick reference

```
bookmarks_t_size(e, ud) -> uint32

@param e: lochist_entry_t const &
@param ud
```

IDAPython function idaapi.boundaries_begin quick reference

```
boundaries_begin(map) -> boundaries_iterator_t
Get iterator pointing to the beginning of boundaries_t.

@param map: (C++: const boundaries_t *) boundaries_t const *
```

IDAPython function idaapi.boundaries_clear quick reference

```
boundaries_clear(map)
Clear boundaries_t.

@param map: (C++: boundaries_t *)
```

IDAPython function idaapi.boundaries_end quick reference

```
boundaries_end(map) -> boundaries_iterator_t
Get iterator pointing to the end of boundaries_t.

@param map: (C++: const boundaries_t *) boundaries_t const *
```

IDAPython function idaapi.boundaries_erase quick reference

```
boundaries_erase(map, p)
Erase current element from boundaries_t.

@param map: (C++: boundaries_t *)
@param p: (C++: boundaries_iterator_t)
```

IDAPython function idaapi.boundaries_find quick reference

```
boundaries_find(map, key) -> boundaries_iterator_t
Find the specified key in boundaries_t.

@param map: (C++: const boundaries_t *) boundaries_t const *
@param key: (C++: const cinsn_t *&) cinsn_t const *
```

IDAPython function idaapi.boundaries_first quick reference

```
boundaries_first(p) -> cinsn_t
Get reference to the current map key.

@param p: (C++: boundaries_iterator_t)
```

IDAPython function idaapi.boundaries_free quick reference

```
boundaries_free(map)
Delete boundaries_t instance.

@param map: (C++: boundaries_t *)
```

IDAPython function idaapi.boundaries_insert quick reference

```
boundaries_insert(map, key, val) -> boundaries_iterator_t
Insert new (cinsn_t *, rangeset_t) pair into boundaries_t.

@param map: (C++: boundaries_t *)
@param key: (C++: const cinsn_t *&) cinsn_t const *
@param val: (C++: const rangeset_t &) rangeset_t const &
```

IDAPython function idaapi.boundaries_new quick reference

```
boundaries_new() -> boundaries_t
Create a new boundaries_t instance.
```

IDAPython function idaapi.boundaries_next quick reference

```
boundaries_next(p) -> boundaries_iterator_t
Move to the next element.

@param p: (C++: boundaries_iterator_t)
```

IDAPython function idaapi.boundaries_prev quick reference

```
boundaries_prev(p) -> boundaries_iterator_t
Move to the previous element.

@param p: (C++: boundaries_iterator_t)
```

IDAPython function idaapi.boundaries_second quick reference

```
boundaries_second(p) -> rangeset_t
Get reference to the current map value.

@param p: (C++: boundaries_iterator_t)
```

IDAPython function idaapi.boundaries_size quick reference

```
boundaries_size(map) -> size_t
Get size of boundaries_t.

@param map: (C++: boundaries_t *)
```

IDAPython function idaapi.bring_debugger_to_front quick reference

```
bring_debugger_to_front()
```

IDAPython function idaapi.build_snapshot_tree quick reference

```
build_snapshot_tree(root) -> bool
Build the snapshot tree.

@param root: (C++: snapshot_t *) snapshot root that will contain the snapshot tree eleme
@return: success
```

IDAPython function idaapi.build_stkvar_name quick reference

```
build_stkvar_name(pfn, v) -> str
Build automatic stack variable name.

@param pfn: (C++: const func_t *) pointer to function (can't be nullptr!)
@param v (integer): value of variable offset
@return: length of stack variable name or -1
```

IDAPython function idaapi.build_stkvar_xrefs quick reference

```
build_stkvar_xrefs(out, pfn, mptr)
Fill 'out' with a list of all the xrefs made from function 'pfn', to the
argument or variable 'mptr' in 'pfn's stack frame.

@param out: (C++: xreflist_t *) the list of xrefs to fill.
@param pfn (idaapi.func_t): the function to scan.
@param mptr: (C++: const member_t *) the argument/variable in pfn's stack frame.
```

IDAPython function idaapi.build_strlist quick reference

```
build_strlist()
Rebuild the string list.
```

IDAPython function idaapi.byte_flag quick reference

```
byte_flag() -> flags64_t
Get a flags64_t representing a byte.
```

IDAPython function idaapi.bytesize quick reference

```
bytesize(ea) -> int
Get number of bytes required to store a byte at the given address.

@param ea (integer):
```

IDAPython function idaapi.calc_basevalue quick reference

```
calc_basevalue(target, base) -> ea_t
Calculate the value of the reference base.

@param target (integer):
@param base (integer):
```

IDAPython function idaapi.calc_bg_color quick reference

```
calc_bg_color(ea) -> bgcolor_t
Get background color for line at 'ea'

@param ea (integer):
@return: RGB color
```

IDAPython function idaapi.calc_c_cpp_name quick reference

```
calc_c_cpp_name(name, type, ccn_flags) -> str
Get C or C++ form of the name.

@param name (string): original (mangled or decorated) name
@param type: (C++: const tinfo_t *) name type if known, otherwise nullptr
@param ccn_flags (integer): one of C/C++ naming flags
```

IDAPython function idaapi.calc_dataseg quick reference

```
calc_dataseg(insn, n=-1, rgnum=-1) -> ea_t
Get data segment for the instruction operand. 'opnum' and 'rgnum' are meaningful
only if the processor has segment registers.

@param insn: (C++: const insn_t &) an idaapi.insn_t, or an address (C++: const insn_t &)
@param n (integer):
@param rgnum (integer):
```

IDAPython function idaapi.calc_def_align quick reference

```
calc_def_align(ea, mina, maxa) -> int
Calculate the default alignment exponent.

@param ea (integer): linear address
@param mina (integer): minimal possible alignment exponent.
@param maxa (integer): minimal possible alignment exponent.
```

IDAPython function idaapi.calc_default_idaplace_flags quick reference

```
calc_default_idaplace_flags() -> int
Get default disassembly line options.
```

IDAPython function idaapi.calc_dflags quick reference

```
calc_dflags(f, force) -> flags64_t

@param f: flags64_t
@param force: bool
```

IDAPython function idaapi.calc_dist quick reference

```
calc_dist(p, q) -> double
Calculate distance between p and q.

@param p: (C++: point_t)
@param q: (C++: point_t)
```

IDAPython function idaapi.calc_fixup_size quick reference

```
calc_fixup_size(type) -> int
Calculate size of fixup in bytes (the number of bytes the fixup patches)
@retval -1: means error

@param type: (C++: fixup_type_t)
```

IDAPython function idaapi.calc_func_size quick reference

```
calc_func_size(pfn) -> asize_t
Calculate function size. This function takes into account all fragments of the
function.

@param pfn (idaapi.func_t): ptr to function structure
```

IDAPython function idaapi.calc_gtn_flags quick reference

```
Calculate flags for get_ea_name() function

@param fromaddr: the referring address. May be BADADDR.
@param ea: linear address

@return: flags
```

IDAPython function idaapi.calc_idasgn_state quick reference

```
calc_idasgn_state(n) -> int
Get state of a signature in the list of planned signatures

@param n (integer): number of signature in the list (0..get_idasgn_qty()-1)
@return: state of signature or IDASGN_BADARG
```

IDAPython function idaapi.calc_max_align quick reference

```
calc_max_align(endea) -> int
Calculate the maximal possible alignment exponent.

@param endea (integer): end address of the alignment item.
@return: a value in the 0..32 range
```

IDAPython function idaapi.calc_max_item_end quick reference

```
calc_max_item_end(ea, how=15) -> ea_t
Calculate maximal reasonable end address of a new item. This function will limit
the item with the current segment bounds.

@param ea (integer): linear address
@param how (integer): when to stop the search. A combination of Item end search flags
@return: end of new item. If it is not possible to create an item, it will
         return 'ea'.
```

IDAPython function idaapi.calc_min_align quick reference

```
calc_min_align(length) -> int
Calculate the minimal possible alignment exponent.

@param length (integer): size of the item in bytes.
@return: a value in the 1..32 range
```

IDAPython function idaapi.calc_number_of_children quick reference

```
calc_number_of_children(loc, tif, dont_deref_ptr=False) -> int
Calculate max number of lines of a formatted c data, when expanded (PTV_EXPAND).

@param loc: (C++: const argloc_t &) location of the data (ALOC_STATIC or ALOC_CUSTOM)
@param tif (idaapi.tinfo_t): type info
```

```
@param dont_deref_ptr (bool): consider 'ea' as the ptr value
@retval 0: data is not expandable
@retval -1: error, see qerrno
@retval else: the max number of lines
```

IDAPython function idaapi.calc_offset_base quick reference

```
calc_offset_base(ea, n) -> ea_t
Try to calculate the offset base This function takes into account the fixup
information, current ds and cs values.

@param ea (integer): the referencing instruction/data address
@param n (integer): operand number
* 0: first operand
* 1: second operand
* ...
* 7: eighth operand
@return: output base address or BADADDR
```

IDAPython function idaapi.calc_prefix_color quick reference

```
calc_prefix_color(ea) -> color_t
Get prefix color for line at 'ea'

@param ea (integer):
@return: Line prefix colors
```

IDAPython function idaapi.calc_probable_base_by_value quick reference

```
calc_probable_base_by_value(ea, off) -> ea_t
Try to calculate the offset base. 2 bases are checked: current ds and cs. If
fails, return BADADDR

@param ea (integer):
@param off (integer):
```

IDAPython function idaapi.calc_reference_data quick reference

```
calc_reference_data(target, base, _from, ri, opval) -> bool
Calculate the target and base addresses of an offset expression. The calculated
target and base addresses are returned in the locations pointed by 'base' and
'target'. In case 'ri.base' is BADADDR, the function calculates the offset base
```

```
address from the referencing instruction/data address. The target address is
copied from ri.target. If ri.target is BADADDR then the target is calculated
using the base address and 'opval'. This function also checks if 'opval' matches
the full value of the reference and takes in account the memory-mapping.

@param target: (C++: ea_t *) output target address
@param base: (C++: ea_t *) output base address
@param from (integer): the referencing instruction/data address
@param ri: (C++: const refinfo_t &) reference info block from the database
@param opval (integer): operand value (usually op_t::value or op_t::addr)
@return: success
```

IDAPython function idaapi.calc_stkvar_struc_offset quick reference

```
calc_stkvar_struc_offset(pfn, insn, n) -> ea_t
Calculate offset of stack variable in the frame structure.

@param pfn (idaapi.func_t): pointer to function (can't be nullptr!)
@param insn: (C++: const insn_t &) the instruction
@param n (integer): 0..UA_MAXOP-1 operand number -1 if error, return BADADDR
@return: BADADDR if some error (issue a warning if stack frame is bad)
```

IDAPython function idaapi.calc_switch_cases quick reference

```
calc_switch_cases(ea, si) -> cases_and_targets_t
Get information about a switch's cases.

The returned information can be used as follows:

    for idx in range(len(results.cases)):
        cur_case = results.cases[idx]
        for cidx in range(len(cur_case)):
            print("case: %d" % cur_case[cidx])
        print("  goto 0x%x" % results.targets[idx])

@param ea: address of the 'indirect jump' instruction
@param si: switch information

@return: a structure with 2 members: 'cases', and 'targets'.
```

IDAPython function idaapi.calc_target quick reference

```
calc_target(_from, opval, ri) -> ea_t
Retrieves refinfo_t structure and calculates the target.
```

```
@param from (integer):
@param opval (integer):
@param ri: refinfo_t const &

calc_target(_from, ea, n, opval) -> ea_t

@param from: ea_t
@param ea: ea_t
@param n: int
@param opval: adiff_t
```

IDAPython function idaapi.calc_thunk_func_target quick reference

```
calc_thunk_func_target(pfn) -> ea_t
Calculate target of a thunk function.

@param pfn (idaapi.func_t): pointer to function (may not be nullptr)
@return: the target function or BADADDR
```

IDAPython function idaapi.calc_tinfo_gaps quick reference

```
calc_tinfo_gaps(out, typid) -> bool

@param out: rangeset_t *
@param typid: uint32
```

IDAPython function idaapi.calc_type_size quick reference

```
calc_type_size(ti, tp) -> PyObject *
Returns the size of a type

@param ti: Type info. 'None' can be passed.
@param tp: type string
@return:     - None on failure
    - The size of the type
```

IDAPython function idaapi.call_helper quick reference

```
Create a helper call.
```

IDAPython function idaapi.call_nav_colorizer quick reference

```
call_nav_colorizer(dict, ea, nbytes) -> uint32
To be used with the IDA-provided colorizer, that is
returned as result of the first call to set_nav_colorizer().

@param dict: PyObject *
@param ea: ea_t
@param nbytes: asize_t
```

IDAPython function idaapi.callregs_t_regcount quick reference

```
callregs_t_regcount(cc) -> int

@param cc: cm_t
```

IDAPython function idaapi.can_be_off32 quick reference

```
can_be_off32(ea) -> ea_t
Does the specified address contain a valid OFF32 value?. For symbols in special
segments the displacement is not taken into account. If yes, then the target
address of OFF32 will be returned. If not, then BADADDR is returned.

@param ea (integer):
```

IDAPython function idaapi.can_decode quick reference

```
can_decode(ea) -> bool
Can the bytes at address 'ea' be decoded as instruction?

@param ea (integer): linear address
@return: whether or not the contents at that address could be a valid
         instruction
```

IDAPython function idaapi.can_define_item quick reference

```
can_define_item(ea, length, flags) -> bool
Can define item (instruction/data) of the specified 'length', starting at 'ea'?
@note: if there is an item starting at 'ea', this function ignores it
@note: this function converts to unexplored all encountered data items with
       fixup information. Should be fixed in the future.

@param ea (integer): start of the range for the new item
```

```
@param length (integer): length of the new item in bytes
@param flags (integer): if not 0, then the kernel will ignore the data types specified b
              the flags and destroy them. For example:
1000 dw 5
                1002 db 5 ; undef
                1003 db 5 ; undef
                1004 dw 5
                1006 dd 5
                 can_define_item(1000, 6, 0) - false because of dw at 1004
can_define_item(1000, 6, word_flag()) - true, word at 1004 is destroyed
@return: 1-yes, 0-no
* a new item would cross segment boundaries
* a new item would overlap with existing items (except items specified by
'flags')
```

IDAPython function idaapi.can_exc_continue quick reference

```
can_exc_continue(ev) -> bool

@param ev: debug_event_t const *
```

IDAPython function idaapi.cancel_exec_request quick reference

```
cancel_exec_request(req_id) -> bool
Try to cancel an asynchronous exec request (ui_cancel_exec_request).

@param req_id (integer): request id
@retval true: successfully canceled
@retval false: request has already been processed.
```

IDAPython function idaapi.cancel_thread_exec_requests quick reference

```
cancel_thread_exec_requests(tid) -> int
Try to cancel asynchronous exec requests created by the specified thread.

@param tid: (C++: qthread_t) thread id
@return: number of the canceled requests.
```

IDAPython function idaapi.cexpr_operands quick reference

```
return a dictionary with the operands of a cexpr_t.
```

IDAPython function idaapi.cfg_get_cc_header_path quick reference

```
cfg_get_cc_header_path(compid) -> char const *

@param compid: comp_t
```

IDAPython function idaapi.cfg_get_cc_parm quick reference

```
cfg_get_cc_parm(compid, name) -> char const *

@param compid: comp_t
@param name: char const *
```

IDAPython function idaapi.cfg_get_cc_predefined_macros quick reference

```
cfg_get_cc_predefined_macros(compid) -> char const *

@param compid: comp_t
```

IDAPython function idaapi.cfunc_type quick reference

```
Get the function's return type tinfo_t object.
```

IDAPython function idaapi.change_hexrays_config quick reference

```
change_hexrays_config(directive) -> bool
Parse DIRECTIVE and update the current configuration variables. For the syntax
see hexrays.cfg

@param directive (string): char const *
```

IDAPython function idaapi.change_segment_status quick reference

```
change_segment_status(s, is_deb_segm) -> int
Convert a debugger segment to a regular segment and vice versa. When converting
debug->regular, the memory contents will be copied to the database.

@param s: (C++: segment_t *) segment to modify
@param is_deb_segm (bool): new status of the segment
@return: Change segment status result codes
```

IDAPython function idaapi.change_storage_type quick reference

```
change_storage_type(start_ea, end_ea, stt) -> error_t
Change flag storage type for address range.

@param start_ea (integer): should be lower than end_ea.
@param end_ea (integer): does not belong to the range.
@param stt: (C++: storage_type_t)
@return: error code
```

IDAPython function idaapi.char_flag quick reference

```
char_flag() -> flags64_t
see FF_opbits
```

IDAPython function idaapi.check_bpt quick reference

```
check_bpt(ea) -> int
Check the breakpoint at the specified address.

@param ea (integer):
@return: one of Breakpoint status codes
```

IDAPython function idaapi.check_process_exit quick reference

```
check_process_exit(handle, exit_code, msecs=-1) -> int
Check whether process has terminated or not.

@param handle : process handle to wait for
@param exit_code: (C++: int *) pointer to the buffer for the exit code
@param msecs: how long to wait. special values:
* 0: do not wait
* 1 or -1: wait infinitely
* other values: timeout in milliseconds
@retval 0: process has exited, and the exit code is available. if *exit_code <
           0: the process was killed with a signal -*exit_code
@retval 1: process has not exited yet
@retval -1: error happened, see error code for winerr() in *exit_code
```

IDAPython function idaapi.checkout_hexrays_license quick reference

```
checkout_hexrays_license(silent) -> bool
```

```
Check out a floating decompiler license. This function will display a dialog box
if the license is not available. For non-floating licenses this function is
effectively no-op. It is not necessary to call this function before decompiling.
If the license was not checked out, the decompiler will automatically do it.
This function can be used to check out a license in advance and ensure that a
license is available.

@param silent (bool): silently fail if the license cannot be checked out.
@return: false if failed
```

IDAPython function idaapi.choose_activate quick reference

```
choose_activate(_self)

@param self: PyObject *
```

IDAPython function idaapi.choose_choose quick reference

```
choose_choose(_self) -> PyObject *

@param self: PyObject *
```

IDAPython function idaapi.choose_close quick reference

```
choose_close(_self)

@param self: PyObject *
```

IDAPython function idaapi.choose_create_embedded_chobj quick reference

```
choose_create_embedded_chobj(_self) -> PyObject *

@param self: PyObject *
```

IDAPython function idaapi.choose_entry quick reference

```
choose_entry(title) -> ea_t
Choose an entry point (ui_choose, chtype_entry).

@param title (string): chooser title
@return: ea of selected entry point, BADADDR if none selected
```

IDAPython function idaapi.choose_enum quick reference

```
choose_enum(title, default_id) -> enum_t
Choose an enum (ui_choose, chtype_enum).

@param title (string): chooser title
@param default_id (integer): id of enum to select by default
@return: enum id of selected enum, BADNODE if none selected
```

IDAPython function idaapi.choose_enum_by_value quick reference

```
choose_enum_by_value(title, default_id, value, nbytes) -> enum_t
Choose an enum, restricted by value & size (ui_choose,
chtype_enum_by_value_and_size). If the given value cannot be found initially,
this function will ask if the user would like to import a standard enum.

@param title (string): chooser title
@param default_id (integer): id of enum to select by default
@param value (integer): value to search for
@param nbytes (integer): size of value
@return: enum id of selected (or imported) enum, BADNODE if none was found
```

IDAPython function idaapi.choose_find quick reference

```
choose_find(title) -> MyChoose or None

@param title: char const *
```

IDAPython function idaapi.choose_func quick reference

```
choose_func(title, default_ea) -> func_t *
Choose a function (ui_choose, chtype_func).

@param title (string): chooser title
@param default_ea (integer): ea of function to select by default
@return: pointer to function that was selected, nullptr if none selected
```

IDAPython function idaapi.choose_get_widget quick reference

```
choose_get_widget(_self) -> TWidget *
```

```
@param self: PyObject *
```

IDAPython function idaapi.choose_idasgn quick reference

```
choose_idasgn() -> PyObject *
Opens the signature chooser

@return: None or the selected signature name
```

IDAPython function idaapi.choose_ioport_device2 quick reference

```
choose_ioport_device2(_device, file, parse_params) -> bool

@param _device: qstring *
@param file: char const *
@param parse_params: choose_ioport_parser_t *
```

IDAPython function idaapi.choose_local_tinfo quick reference

```
choose_local_tinfo(ti, title, func=None, def_ord=0, ud=None) -> uint32
Choose a type from the local type library.

@param ti (idaapi.til_t): pointer to til
@param title (string): title of listbox to display
@param func: (C++: local_tinfo_predicate_t *) predicate to select types to display (mayb
@param def_ord (integer): ordinal to position cursor before choose
@param ud : user data
@return: == 0 means nothing is chosen, otherwise an ordinal number
```

IDAPython function idaapi.choose_local_tinfo_and_delta quick reference

```
choose_local_tinfo_and_delta(delta, ti, title, func=None, def_ord=0, ud=None) -> uint32
Choose a type from the local type library and specify the pointer shift value.

@param delta: (C++: int32 *) pointer shift value
@param ti (idaapi.til_t): pointer to til
@param title (string): title of listbox to display
@param func: (C++: local_tinfo_predicate_t *) predicate to select types to display (mayb
@param def_ord (integer): ordinal to position cursor before choose
@param ud : user data
@return: == 0 means nothing is chosen, otherwise an ordinal number
```

IDAPython function idaapi.choose_name quick reference

```
choose_name(title) -> ea_t
Choose a name (ui_choose, chtype_name).

@param title (string): chooser title
@return: ea of selected name, BADADDR if none selected
```

IDAPython function idaapi.choose_named_type quick reference

```
choose_named_type(out_sym, root_til, title, ntf_flags, predicate=None) -> bool
Choose a type from a type library.

@param out_sym: (C++: til_symbol_t *) pointer to be filled with the chosen type
@param root_til (idaapi.til_t): pointer to starting til (the function will inspect the b
                tils if allowed by flags)
@param title (string): title of listbox to display
@param ntf_flags (integer): combination of Flags for named types
@param predicate: (C++: predicate_t *) predicate to select types to display (maybe nullp
@return: false if nothing is chosen, otherwise true
```

IDAPython function idaapi.choose_refresh quick reference

```
choose_refresh(_self)

@param self: PyObject *
```

IDAPython function idaapi.choose_segm quick reference

```
choose_segm(title, default_ea) -> segment_t *
Choose a segment (ui_choose, chtype_segm).

@param title (string): chooser title
@param default_ea (integer): ea of segment to select by default
@return: pointer to segment that was selected, nullptr if none selected
```

IDAPython function idaapi.choose_srcp quick reference

```
choose_srcp(title) -> sreg_range_t *
Choose a segment register change point (ui_choose, chtype_srcp).

@param title (string): chooser title
@return: pointer to segment register range of selected change point, nullptr if
```

```
        none selected
```

IDAPython function idaapi.choose_stkvar_xref quick reference

```
choose_stkvar_xref(pfn, mptr) -> ea_t
Choose an xref to a stack variable (ui_choose, chtype_name).

@param pfn (idaapi.func_t): function
@param mptr: (C++: member_t *) variable
@return: ea of the selected xref, BADADDR if none selected
```

IDAPython function idaapi.choose_struc quick reference

```
choose_struc(title) -> struc_t *
Choose a structure (ui_choose, chtype_segm).

@param title (string): chooser title;
@return: pointer to structure that was selected, nullptr if none selected
```

IDAPython function idaapi.choose_til quick reference

```
choose_til() -> str
Choose a type library (ui_choose, chtype_idatil).

@retval true: 'buf' was filled with the name of the selected til
@retval false: otherwise
```

IDAPython function idaapi.choose_trace_file quick reference

```
choose_trace_file() -> str
Show the choose trace dialog.
```

IDAPython function idaapi.choose_xref quick reference

```
choose_xref(to) -> ea_t
Choose an xref to an address (ui_choose, chtype_xref).

@param to (integer): referenced address
@return: ea of selected xref, BADADDR if none selected
```

IDAPython function idaapi.chunk_size quick reference

```
chunk_size(ea) -> asize_t
Get size of the contiguous address block containing 'ea'.

@param ea (integer):
@return: 0 if 'ea' doesn't belong to the program.
```

IDAPython function idaapi.chunk_start quick reference

```
chunk_start(ea) -> ea_t
Get start of the contiguous address block containing 'ea'.

@param ea (integer):
@return: BADADDR if 'ea' doesn't belong to the program.
```

IDAPython function idaapi.cinsn_details quick reference

```
return the details pointer for the cinsn_t object depending on the value of its op membe
```

IDAPython function idaapi.cinsn_t_insn_is_epilog quick reference

```
cinsn_t_insn_is_epilog(insn) -> bool

@param insn: cinsn_t const *
```

IDAPython function idaapi.citem_to_specific_type quick reference

```
cast the citem_t object to its more specific type, either cexpr_t or cinsn_t.
```

IDAPython function idaapi.cleanup_appcall quick reference

```
cleanup_appcall(tid) -> error_t
Cleanup after manual appcall.

@param tid (integer): thread to use. NO_THREAD means to use the current thread The
            application state is restored as it was before calling the last
            appcall(). Nested appcalls are supported.
@return: eOk if successful, otherwise an error code
```

IDAPython function idaapi.cleanup_name quick reference

```
cleanup_name(ea, name, flags=0) -> str

@param ea: ea_t
@param name: char const *
@param flags: uint32
```

IDAPython function idaapi.clear_cached_cfuncs quick reference

```
clear_cached_cfuncs()
Flush all cached decompilation results.
```

IDAPython function idaapi.clear_refresh_request quick reference

```
clear_refresh_request(mask)

@param mask: uint64
```

IDAPython function idaapi.clear_requests_queue quick reference

```
clear_requests_queue()
Clear the queue of waiting requests. \sq{Type, Synchronous function,
Notification, none (synchronous function)}
@note: If a request is currently running, this one isn't stopped.
```

IDAPython function idaapi.clear_strlist quick reference

```
clear_strlist()
Clear the string list.
```

IDAPython function idaapi.clear_tinfo_t quick reference

```
clear_tinfo_t(_this)

@param _this: tinfo_t *
```

IDAPython function idaapi.clear_trace quick reference

```
clear_trace()
Clear all events in the trace buffer. \sq{Type, Synchronous function - available
as request, Notification, none (synchronous function)}
```

IDAPython function idaapi.cli_t quick reference

```
cli_t wrapper class.

This class allows you to implement your own command line interface handlers.
```

IDAPython function idaapi.close_chooser quick reference

```
close_chooser(title) -> bool
Close a non-modal chooser (ui_close_chooser).

@param title (string): window title of chooser to close
@return: success
```

IDAPython function idaapi.close_hexrays_waitbox quick reference

```
close_hexrays_waitbox()
Close the waitbox displayed by the decompiler. Useful if DECOMP_NO_HIDE was used
during decompilation.
```

IDAPython function idaapi.close_linput quick reference

```
close_linput(li)
Close loader input.

@param li: (C++: linput_t *)
```

IDAPython function idaapi.close_pseudocode quick reference

```
close_pseudocode(f) -> bool
Close pseudocode window.

@param f(a Widget SWIG wrapper class): pointer to window
@return: false if failed
```

IDAPython function idaapi.close_widget quick reference

```
close_widget(widget, options)
Close widget (ui_close_widget, only gui version).
```

```
@param widget(a Widget SWIG wrapper class): pointer to the widget to close
@param options (integer): Form close flags
```

IDAPython function idaapi.clr___bnot0 quick reference

```
clr__bnot0(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr___bnot1 quick reference

```
clr__bnot1(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr___invsign0 quick reference

```
clr__invsign0(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr___invsign1 quick reference

```
clr__invsign1(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_abits quick reference

```
clr_abits(ea, bits)

@param ea: ea_t
@param bits: aflags_t
```

IDAPython function idaapi.clr_align_flow quick reference

```
clr_align_flow(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_cancelled quick reference

```
clr_cancelled()
Clear "Cancelled" flag (ui_clr_cancelled)
```

IDAPython function idaapi.clr_colored_item quick reference

```
clr_colored_item(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_database_flag quick reference

```
clr_database_flag(dbfl)

@param dbfl: uint32
```

IDAPython function idaapi.clr_fixed_spd quick reference

```
clr_fixed_spd(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_has_lname quick reference

```
clr_has_lname(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_has_ti quick reference

```
clr_has_ti(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_has_ti0 quick reference

```
clr_has_ti0(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_has_ti1 quick reference

```
clr_has_ti1(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_libitem quick reference

```
clr_libitem(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_lzero quick reference

```
clr_lzero(ea, n) -> bool
Clear toggle lzero bit. This function reset the display of leading zeroes for
the specified operand to the default. If the default is not to display leading
zeroes, leading zeroes will not be displayed, as vice versa.

@param ea (integer): the item (insn/data) address
@param n (integer): the operand number (0-first operand, 1-other operands)
@return: success
```

IDAPython function idaapi.clr_lzero0 quick reference

```
clr_lzero0(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_lzero1 quick reference

```
clr_lzero1(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_node_info quick reference

```
clr_node_info(gid, node, flags)
Clear node info for the given node.

@param gid: (C++: graph_id_t) id of desired graph
@param node (integer): node number
@param flags (integer): combination of Node info flags, identifying which fields of
```

```
node_info_t will be cleared
```

IDAPython function idaapi.clr_noret quick reference

```
clr_noret(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_notcode quick reference

```
clr_notcode(ea)
Clear not-code mark.

@param ea (integer):
```

IDAPython function idaapi.clr_notproc quick reference

```
clr_notproc(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_op_type quick reference

```
clr_op_type(ea, n) -> bool
Remove operand representation information. (set operand representation to be
'undefined')

@param ea (integer): linear address
@param n (integer): 0..UA_MAXOP-1 operand number, OPND_ALL all operands
@return: success
```

IDAPython function idaapi.clr_retfp quick reference

```
clr_retfp(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_terse_struc quick reference

```
clr_terse_struc(ea)
```

```
@param ea: ea_t
```

IDAPython function idaapi.clr_tilcmt quick reference

```
clr_tilcmt(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_usemodsp quick reference

```
clr_usemodsp(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_usersp quick reference

```
clr_usersp(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_userti quick reference

```
clr_userti(ea)

@param ea: ea_t
```

IDAPython function idaapi.clr_zstroff quick reference

```
clr_zstroff(ea)

@param ea: ea_t
```

IDAPython function idaapi.code_flag quick reference

```
code_flag() -> flags64_t
FF_CODE
```

IDAPython function idaapi.collect_stack_trace quick reference

```
collect_stack_trace(tid, trace) -> bool
```

```
@param tid: thid_t
@param trace: call_stack_t *
```

IDAPython function idaapi.combine__flags quick reference

```
combine_flags(F) -> flags64_t

@param F: flags64_t
```

IDAPython function idaapi.compact__til quick reference

```
compact_til(ti) -> bool
Collect garbage in til. Must be called before storing the til.

@param ti (idaapi.til_t):
@return: true if any memory was freed
```

IDAPython function idaapi.compare__tinfo quick reference

```
compare_tinfo(t1, t2, tcflags) -> bool

@param t1: uint32
@param t2: uint32
@param tcflags: int
```

IDAPython function idaapi.compile__idc__file quick reference

```
compile_idc_file(nonnul_line) -> str

@param nonnul_line: char const *
```

IDAPython function idaapi.compile__idc__snippet quick reference

```
compile_idc_snippet(func, text, resolver=None, only_safe_funcs=False) -> str
Compile text with IDC statements.

@param func (string): name of the function to create out of the snippet
@param text (string): text to compile
@param resolver: (C++: idc_resolver_t *) callback object to get values of undefined vari
                 object will be called if IDC function contains references to
                 undefined variables. May be nullptr.
@param only_safe_funcs (bool): if true, any calls to functions without EXTFUN_SAFE flag
```

```
                        will lead to a compilation error.
@retval true: ok
@retval false: error, see errbuf
```

IDAPython function idaapi.compile_idc_text quick reference

```
compile_idc_text(nonnul_line) -> str

@param nonnul_line: char const *
```

IDAPython function idaapi.construct_macro quick reference

```
construct_macro(insn, enable, build_macro) -> bool
See ua.hpp's construct_macro().

@param insn: insn_t &
@param enable: bool
@param build_macro: PyObject *
```

IDAPython function idaapi.construct_macro2 quick reference

```
construct_macro2(_this, insn, enable) -> bool

@param _this: macro_constructor_t *
@param insn: insn_t *
@param enable: bool
```

IDAPython function idaapi.contains_fixups quick reference

```
contains_fixups(ea, size) -> bool
Does the specified address range contain any fixup information?

@param ea (integer):
@param size (integer):
```

IDAPython function idaapi.continue_process quick reference

```
continue_process() -> bool
Continue the execution of the process in the debugger. \sq{Type, Synchronous
function - available as Request, Notification, none (synchronous function)}
@note: The continue_process() function can be called from a notification handler
       to force the continuation of the process. In this case the request queue
```

```
will not be examined, IDA will simply resume execution. Usually it makes
sense to call request_continue_process() followed by run_requests(), so
that IDA will first start a queued request (if any) and then resume the
application.
```

IDAPython function idaapi.convert__pt__flags__to__hti quick reference

```
convert_pt_flags_to_hti(pt_flags) -> int
Convert Type parsing flags to Type formatting flags. Type parsing flags lesser
than 0x10 don't have stable meaning and will be ignored (more on these flags can
be seen in idc.idc)

@param pt_flags (integer):
```

IDAPython function idaapi.convert__to__user__call quick reference

```
convert_to_user_call(udc, cdg) -> merror_t
try to generate user-defined call for an instruction

@param udc: (C++: const udcall_t &) udcall_t const &
@param cdg: (C++: codegen_t &)
@return: Microcode error codes code: MERR_OK - user-defined call generated else
         - error (MERR_INSN == inacceptable udc.tif)
```

IDAPython function idaapi.copy__bits quick reference

```
Copy bits from a value
@param v: the value
@param s: starting bit (0-based)
@param e: ending bit
```

IDAPython function idaapi.copy__idcv quick reference

```
copy_idcv(dst, src) -> error_t
Copy 'src' to 'dst'. For idc objects only a reference is copied.

@param dst (idaapi.idc_value_t):
@param src: (C++: const idc_value_t &) idc_value_t const &
```

IDAPython function idaapi.copy__named__type quick reference

```
copy_named_type(dsttil, srctil, name) -> uint32
```

```
Copy a named type from one til to another. This function will copy the specified
type and all dependent types from the source type library to the destination
library.

@param dsttil (idaapi.til_t): Destination til. It must have orginal types enabled
@param srctil (idaapi.til_t): Source til.
@param name (string): name of the type to copy
@return: ordinal number of the copied type. 0 means error
```

IDAPython function idaapi.copy_sreg_ranges quick reference

```
copy_sreg_ranges(dst_rg, src_rg, map_selector=False)
Duplicate segment register ranges.

@param dst_rg (integer): number of destination segment register
@param src_rg (integer): copy ranges from
@param map_selector (bool): map selectors to linear addresses using sel2ea()
```

IDAPython function idaapi.copy_tinfo_t quick reference

```
copy_tinfo_t(_this, r)

@param _this: tinfo_t *
@param r: tinfo_t const &
```

IDAPython function idaapi.create_16bit_data quick reference

```
create_16bit_data(ea, length) -> bool
Convert to 16-bit quantity (take the byte size into account)

@param ea (integer):
@param length (integer):
```

IDAPython function idaapi.create_32bit_data quick reference

```
create_32bit_data(ea, length) -> bool
Convert to 32-bit quantity (take the byte size into account)

@param ea (integer):
@param length (integer):
```

IDAPython function idaapi.create_align quick reference

```
create_align(ea, length, alignment) -> bool
Create an alignment item.

@param ea (integer): linear address
@param length (integer): size of the item in bytes. 0 means to infer from ALIGNMENT
@param alignment (integer): alignment exponent. Example: 3 means align to 8 bytes. 0 mea
                 to infer from LENGTH It is forbidden to specify both LENGTH
                 and ALIGNMENT as 0.
@return: success
```

IDAPython function idaapi.create_byte quick reference

```
create_byte(ea, length, force=False) -> bool
Convert to byte.

@param ea (integer):
@param length (integer):
@param force (bool):
```

IDAPython function idaapi.create_bytearray_linput quick reference

```
create_bytearray_linput(s) -> linput_t *
Trivial memory linput.

@param s: qstring const &
```

IDAPython function idaapi.create_cfunc quick reference

```
create_cfunc(mba) -> cfuncptr_t
Create a new cfunc_t object.

@param mba: (C++: mba_t *) microcode object. After creating the cfunc object it takes th
            ownership of MBA.
```

IDAPython function idaapi.create_code_viewer quick reference

```
create_code_viewer(custview, flags=0, parent=None) -> TWidget *
Create a code viewer (ui_create_code_viewer). A code viewer contains on the left
side a widget representing the line numbers, and on the right side, the child
widget passed as parameter. It will inherit its title from the child widget.
```

```
@param custview(a Widget SWIG wrapper class): the custom view to be added
@param flags (integer): Code viewer flags
@param parent(a Widget SWIG wrapper class): widget to contain the new code viewer
```

IDAPython function idaapi.create_custdata quick reference

```
create_custdata(ea, length, dtid, fid, force=False) -> bool
Convert to custom data type.

@param ea (integer):
@param length (integer):
@param dtid (integer):
@param fid (integer):
@param force (bool):
```

IDAPython function idaapi.create_data quick reference

```
create_data(ea, dataflag, size, tid) -> bool
Convert to data (byte, word, dword, etc). This function may be used to create
arrays.

@param ea (integer): linear address
@param dataflag (integer): type of data. Value of function byte_flag(), word_flag(), etc
@param size (integer): size of array in bytes. should be divisible by the size of one it
             of the specified type. for variable sized items it can be specified
             as 0, and the kernel will try to calculate the size.
@param tid (integer): type id. If the specified type is a structure, then tid is structu
            id. Otherwise should be BADNODE.
@return: success
```

IDAPython function idaapi.create_disasm_graph quick reference

```
create_disasm_graph(ea) -> mutable_graph_t
Create a graph using an arbitrary set of ranges.

@param ea: ea_t

create_disasm_graph(ranges) -> mutable_graph_t

@param ranges: rangevec_t const &
```

IDAPython function idaapi.create_double quick reference

```
create_double(ea, length, force=False) -> bool
Convert to double.

@param ea (integer):
@param length (integer):
@param force (bool):
```

IDAPython function idaapi.create_dword quick reference

```
create_dword(ea, length, force=False) -> bool
Convert to dword.

@param ea (integer):
@param length (integer):
@param force (bool):
```

IDAPython function idaapi.create_empty_mba quick reference

```
create_empty_mba(mbr, hf=None) -> mba_t
Create an empty microcode object.

@param mbr: (C++: const mba_ranges_t &) mba_ranges_t const &
@param hf: (C++: hexrays_failure_t *)
```

IDAPython function idaapi.create_empty_widget quick reference

```
create_empty_widget(title, icon=-1) -> TWidget *
Create an empty widget, serving as a container for custom user widgets

@param title (string): char const *
@param icon (integer):
```

IDAPython function idaapi.create_encoding_helper quick reference

```
create_encoding_helper(encidx=-1, nr=nr_once) -> encoder_t *

@param encidx: int
@param nr: enum encoder_t::notify_recerr_t
```

IDAPython function idaapi.create_field_name quick reference

```
create_field_name(type, offset=BADADDR) -> qstring

@param type: tinfo_t const &
@param offset: uval_t
```

IDAPython function idaapi.create_float quick reference

```
create_float(ea, length, force=False) -> bool
Convert to float.

@param ea (integer):
@param length (integer):
@param force (bool):
```

IDAPython function idaapi.create_generic_linput quick reference

```
create_generic_linput(gl) -> linput_t *
Create a generic linput

@param gl: (C++: generic_linput_t *) linput description. this object will be destroyed b
           using "delete gl;"
```

IDAPython function idaapi.create_graph_viewer quick reference

```
create_graph_viewer(title, id, callback, ud, title_height, parent=None) -> graph_viewer_
Create a custom graph viewer.

@param title (string): the widget title
@param id (integer): graph id
@param callback: (C++: hook_cb_t *) callback to handle graph notifications (graph_notifi
@param ud : user data passed to callback
@param title_height (integer): node title height
@param parent(a Widget SWIG wrapper class): the parent widget of the graph viewer
@return: new viewer
```

IDAPython function idaapi.create_helper quick reference

```
Create a helper object..
```

IDAPython function idaapi.create_idcv_ref quick reference

```
create_idcv_ref(ref, v) -> bool
Create a variable reference. Currently only references to global variables can
be created.

@param ref (idaapi.idc_value_t): ptr to the result
@param v: (C++: const idc_value_t *) variable to reference
@return: success
```

IDAPython function idaapi.create_insn quick reference

```
create_insn(ea, out=None) -> int
Create an instruction at the specified address. This function checks if an
instruction is present at the specified address and will try to create one if
there is none. It will fail if there is a data item or other items hindering the
creation of the new instruction. This function will also fill the 'out'
structure.

@param ea (integer): linear address
@param out (idaapi.insn_t): the resulting instruction
@return: the length of the instruction or 0
```

IDAPython function idaapi.create_memory_linput quick reference

```
create_memory_linput(start, size) -> linput_t *
Create a linput for process memory. This linput will use read_dbg_memory() to
read data.

@param start (integer): starting address of the input
@param size (integer): size of the memory area to represent as linput if unknown, may be
             passed as 0
```

IDAPython function idaapi.create_menu quick reference

```
create_menu(name, label, menupath=None) -> bool
Create a menu with the given name, label and optional position, either in the
menubar, or as a submenu. If 'menupath' is non-nullptr, it provides information
about where the menu should be positioned. First, IDA will try and resolve the
corresponding menu by its name. If such an existing menu is found and is present
in the menubar, then the new menu will be inserted in the menubar before it.
Otherwise, IDA will try to resolve 'menupath' as it would for
attach_action_to_menu() and, if found, add the new menu like so:
// The new 'My menu' submenu will appear in the 'Comments' submenu
```

```
// before the 'Enter comment...' command
create_menu("(...)", "My menu", "Edit/Comments/Enter comment...");
or
// The new 'My menu' submenu will appear at the end of the
// 'Comments' submenu.
create_menu("(...)", "My menu", "Edit/Comments/");
If the above fails, the new menu will be appended to the menubar.

@param name (string): name of menu (must be unique)
@param label (string): label of menu
@param menupath (string): where should the menu be inserted
@return: success
```

IDAPython function idaapi.create_mutable_graph quick reference

```
create_mutable_graph(id) -> mutable_graph_t
Create a new empty graph with given id.

@param id (integer):
```

IDAPython function idaapi.create_numbered_type_name quick reference

```
create_numbered_type_name(ord) -> str
Create anonymous name for numbered type. This name can be used to reference a
numbered type by its ordinal Ordinal names have the following format: '#' +
set_de(ord) Returns: -1 if error, otherwise the name length

@param ord: (C++: int32)
```

IDAPython function idaapi.create_outctx quick reference

```
create_outctx(ea, F=0, suspop=0) -> outctx_base_t
Create a new output context. To delete it, just use "delete pctx"

@param ea (integer):
@param F (integer):
@param suspop (integer):
```

IDAPython function idaapi.create_oword quick reference

```
create_oword(ea, length, force=False) -> bool
Convert to octaword/xmm word.
```

```
@param ea (integer):
@param length (integer):
@param force (bool):
```

IDAPython function idaapi.create_packed_real quick reference

```
create_packed_real(ea, length, force=False) -> bool
Convert to packed decimal real.

@param ea (integer):
@param length (integer):
@param force (bool):
```

IDAPython function idaapi.create_qword quick reference

```
create_qword(ea, length, force=False) -> bool
Convert to quadword.

@param ea (integer):
@param length (integer):
@param force (bool):
```

IDAPython function idaapi.create_source_viewer quick reference

```
create_source_viewer(out_ccv, parent, custview, sf, lines, lnnum, colnum, flags) -> sou
Create a source code view.

@param out_ccv: (C++: TWidget **)
@param parent(a Widget SWIG wrapper class):
@param custview(a Widget SWIG wrapper class):
@param sf: (C++: source_file_ptr)
@param lines: (C++: strvec_t *)
@param lnnum (integer):
@param colnum (integer):
@param flags (integer):
```

IDAPython function idaapi.create_strlit quick reference

```
create_strlit(start, len, strtype) -> bool
Convert to string literal and give a meaningful name. 'start' may be higher than
'end', the kernel will swap them in this case
```

```
@param start (integer): starting address
@param len (integer): length of the string in bytes. if 0, then get_max_strlit_length()
            will be used to determine the length
@param strtype: (C++: int32) string type. one of String type codes
@return: success
```

IDAPython function idaapi.create_struct quick reference

```
create_struct(ea, length, tid, force=False) -> bool
Convert to struct.

@param ea (integer):
@param length (integer):
@param tid (integer):
@param force (bool):
```

IDAPython function idaapi.create_switch_table quick reference

```
create_switch_table(ea, si) -> bool
Create switch table from the switch information

@param ea: address of the 'indirect jump' instruction
@param si: switch information

@return: Boolean
```

IDAPython function idaapi.create_switch_xrefs quick reference

```
create_switch_xrefs(ea, si) -> bool
This function creates xrefs from the indirect jump.

Usually there is no need to call this function directly because the kernel
will call it for switch tables

Note: Custom switch information are not supported yet.

@param ea: address of the 'indirect jump' instruction
@param si: switch information

@return: Boolean
```

IDAPython function idaapi.create_tbyte quick reference

```
create_tbyte(ea, length, force=False) -> bool
Convert to tbyte.

@param ea (integer):
@param length (integer):
@param force (bool):
```

IDAPython function idaapi.create_tinfo quick reference

```
create_tinfo(_this, bt, bt2, ptr) -> bool

@param _this: tinfo_t *
@param bt: type_t
@param bt2: type_t
@param ptr
```

IDAPython function idaapi.create_toolbar quick reference

```
create_toolbar(name, label, before=None, flags=0) -> bool
Create a toolbar with the given name, label and optional position

@param name (string): name of toolbar (must be unique)
@param label (string): label of toolbar
@param before (string): if non-nullptr, the toolbar before which the new toolbar will be
              inserted
@param flags (integer): a combination of create toolbar flags, to determine toolbar
             position
@return: success
```

IDAPython function idaapi.create_typedef quick reference

```
create_typedef(name) -> tinfo_t
Create a reference to an ordinal type.

@param name: char const *

@return: type which refers to the specified ordinal. For example, if n is 1, the
         type info which refers to ordinal type 1 is created.
create_typedef(n) -> tinfo_t

@param n: int
```

IDAPython function idaapi.create_user_graph_place quick reference

```
create_user_graph_place(node, lnnum) -> user_graph_place_t
Get a copy of a user_graph_place_t (returns a pointer to static storage)

@param node (integer):
@param lnnum (integer):
```

IDAPython function idaapi.create_word quick reference

```
create_word(ea, length, force=False) -> bool
Convert to word.

@param ea (integer):
@param length (integer):
@param force (bool):
```

IDAPython function idaapi.create_yword quick reference

```
create_yword(ea, length, force=False) -> bool
Convert to ymm word.

@param ea (integer):
@param length (integer):
@param force (bool):
```

IDAPython function idaapi.create_zword quick reference

```
create_zword(ea, length, force=False) -> bool
Convert to zmm word.

@param ea (integer):
@param length (integer):
@param force (bool):
```

IDAPython function idaapi.cust_flag quick reference

```
cust_flag() -> flags64_t
Get a flags64_t representing custom type data.
```

IDAPython function idaapi.custfmt_flag quick reference

```
custfmt_flag() -> flags64_t
see FF_opbits
```

IDAPython function idaapi.custom_viewer_jump quick reference

```
custom_viewer_jump(v, loc, flags=0) -> bool
Append 'loc' to the viewer's history, and cause the viewer to display it.

@param v(a Widget SWIG wrapper class): (TWidget *)
@param loc: (C++: const lochist_entry_t &) (const lochist_entry_t &)
@param flags (integer): (uint32) or'ed combination of CVNF_* values
@return: success
```

IDAPython function idaapi.dbg_add_bpt_tev quick reference

```
dbg_add_bpt_tev(tid, ea, bp) -> bool
Add a new breakpoint trace element to the current trace. \sq{Type, Synchronous
function, Notification, none (synchronous function)}

@param tid (integer):
@param ea (integer):
@param bp (integer):
@return: false if the operation failed, true otherwise
```

IDAPython function idaapi.dbg_add_call_tev quick reference

```
dbg_add_call_tev(tid, caller, callee)
Add a new call trace element to the current trace. \sq{Type, Synchronous
function, Notification, none (synchronous function)}

@param tid (integer):
@param caller (integer):
@param callee (integer):
```

IDAPython function idaapi.dbg_add_debug_event quick reference

```
dbg_add_debug_event(event)
Add a new debug event to the current trace. \sq{Type, Synchronous function,
Notification, none (synchronous function)}

@param event: (C++: debug_event_t *)
```

IDAPython function idaapi.dbg_add_insn_tev quick reference

```
dbg_add_insn_tev(tid, ea, save=SAVE_DIFF) -> bool
Add a new instruction trace element to the current trace. \sq{Type, Synchronous
function, Notification, none (synchronous function)}

@param tid (integer):
@param ea (integer):
@param save: (C++: save_reg_values_t) enum save_reg_values_t
@return: false if the operation failed, true otherwise
```

IDAPython function idaapi.dbg_add_many_tevs quick reference

```
dbg_add_many_tevs(new_tevs) -> bool
Add many new trace elements to the current trace. \sq{Type, Synchronous
function, Notification, none (synchronous function)}

@param new_tevs: (C++: tevinforeg_vec_t *)
@return: false if the operation failed for any tev_info_t object
```

IDAPython function idaapi.dbg_add_ret_tev quick reference

```
dbg_add_ret_tev(tid, ret_insn, return_to)
Add a new return trace element to the current trace. \sq{Type, Synchronous
function, Notification, none (synchronous function)}

@param tid (integer):
@param ret_insn (integer):
@param return_to (integer):
```

IDAPython function idaapi.dbg_add_tev quick reference

```
dbg_add_tev(type, tid, address)
Add a new trace element to the current trace. \sq{Type, Synchronous function,
Notification, none (synchronous function)}

@param type: (C++: tev_type_t) enum tev_type_t
@param tid (integer):
@param address (integer):
```

IDAPython function idaapi.dbg_add_thread quick reference

```
dbg_add_thread(tid)
Add a thread to the current trace. \sq{Type, Synchronous function, Notification,
none (synchronous function)}

@param tid (integer):
```

IDAPython function idaapi.dbg_appcall quick reference

```
dbg_appcall(retval, func_ea, tid, ptif, argv, argnum) -> error_t
Call a function from the debugged application.

@param retval (idaapi.idc_value_t): function return value
* for APPCALL_MANUAL, r will hold the new stack point value
* for APPCALL_DEBEV, r will hold the exception information upon failure and the
return code will be eExecThrow
@param func_ea (integer): address to call
@param tid (integer): thread to use. NO_THREAD means to use the current thread
@param ptif: (C++: const tinfo_t *) pointer to type of the function to call
@param argv (idaapi.idc_value_t): array of arguments
@param argnum (integer): number of actual arguments
@return: eOk if successful, otherwise an error code
```

IDAPython function idaapi.dbg_bin_search quick reference

```
dbg_bin_search(start_ea, end_ea, data, srch_flags) -> str

@param start_ea: ea_t
@param end_ea: ea_t
@param data: compiled_binpat_vec_t const &
@param srch_flags: int
```

IDAPython function idaapi.dbg_can_query quick reference

```
dbg_can_query() -> bool
This function can be used to check if the debugger can be queried:
  - debugger is loaded
  - process is suspended
  - process is not suspended but can take requests. In this case some requests like
    memory read/write, bpt management succeed and register querying will fail.
    Check if idaapi.get_process_state() < 0 to tell if the process is suspended

@return: Boolean
```

IDAPython function idaapi.dbg_del_thread quick reference

```
dbg_del_thread(tid)
Delete a thread from the current trace. \sq{Type, Synchronous function,
Notification, none (synchronous function)}

@param tid (integer):
```

IDAPython function idaapi.dbg_get_input_path quick reference

```
dbg_get_input_path() -> str
Get debugger input file name/path (see LFLG_DBG_NOPATH)
```

IDAPython function idaapi.dbg_get_memory_info quick reference

```
dbg_get_memory_info() -> PyObject *
This function returns the memory configuration of a debugged process.

@return:     None if no debugger is active
    tuple(start_ea, end_ea, name, sclass, sbase, bitness, perm)
```

IDAPython function idaapi.dbg_get_name quick reference

```
dbg_get_name() -> PyObject *
This function returns the current debugger's name.

@return: Debugger name or None if no debugger is active
```

IDAPython function idaapi.dbg_get_registers quick reference

```
dbg_get_registers() -> PyObject *
This function returns the register definition from the currently loaded debugger.
Basically, it returns an array of structure similar to to idd.hpp / register_info_t

@return:     None if no debugger is loaded
    tuple(name, flags, class, dtype, bit_strings, default_bit_strings_mask)
    The bit_strings can be a tuple of strings or None (if the register does not have bit
```

IDAPython function idaapi.dbg_get_thread_sreg_base quick reference

```
dbg_get_thread_sreg_base(tid, sreg_value) -> PyObject *
Returns the segment register base value

@param tid: thread id
@param sreg_value: segment register (selector) value
@return:     - The base as an 'ea'
    - Or None on failure
```

IDAPython function idaapi.dbg_is_loaded quick reference

```
dbg_is_loaded() -> bool
Checks if a debugger is loaded

@return: Boolean
```

IDAPython function idaapi.dbg_read_memory quick reference

```
dbg_read_memory(ea, sz) -> PyObject *
Reads from the debugee's memory at the specified ea

@param ea: ea_t
@param sz: size_t
@return:     - The read buffer (as a string)
    - Or None on failure
```

IDAPython function idaapi.dbg_write_memory quick reference

```
dbg_write_memory(ea, buf) -> bool
Writes a buffer to the debugee's memory

@param ea: ea_t
@param buf: bytevec_t const &
@return: Boolean
```

IDAPython function idaapi.debug_hexrays_ctree quick reference

```
debug_hexrays_ctree(level, msg)

@param level: int
@param msg: char const *
```

IDAPython function idaapi.dec_flag quick reference

```
dec_flag() -> flags64_t
Get number flag of the base, regardless of current processor - better to use
num_flag()
```

IDAPython function idaapi.decode_insn quick reference

```
decode_insn(out, ea) -> int
Analyze the specified address and fill 'out'. This function does not modify the
database. It just tries to interpret the specified address as an instruction and
fills the 'out' structure.

@param out (idaapi.insn_t): the resulting instruction
@param ea (integer): linear address
@return: the length of the (possible) instruction or 0
```

IDAPython function idaapi.decode_preceding_insn quick reference

```
decode_preceding_insn(out, ea) -> (int, int)
Decodes the preceding instruction. Please check ua.hpp / decode_preceding_insn()

@param out: instruction storage
@param ea: current ea
@return: tuple(preceeding_ea or BADADDR, farref = Boolean)
```

IDAPython function idaapi.decode_prev_insn quick reference

```
decode_prev_insn(out, ea) -> ea_t
Decode previous instruction if it exists, fill 'out'.

@param out (idaapi.insn_t): the resulting instruction
@param ea (integer): the address to decode the previous instruction from
@return: the previous instruction address (BADADDR-no such insn)
```

IDAPython function idaapi.decompile quick reference

```
Decompile a function.

@param ea an address belonging to the function, or an idaapi.func_t object
@param hf extended error information (if failed)
```

```
@param flags decomp_flags bitwise combination of `DECOMP_...` bits
@return the decompilation result (a `idaapi.cfunc_t` wrapper), or None
```

IDAPython function idaapi.decompile_func quick reference

```
decompile_func(pfn, hf=None, decomp_flags=0) -> cfuncptr_t
Decompile a function. Multiple decompilations of the same function return the
same object.

@param pfn (idaapi.func_t): pointer to function to decompile
@param hf: (C++: hexrays_failure_t *) extended error information (if failed)
@param decomp_flags (integer): bitwise combination of decompile() flags... bits
@return: pointer to the decompilation result (a reference counted pointer).
         nullptr if failed.
```

IDAPython function idaapi.decompile_many quick reference

```
decompile_many(outfile, funcaddrs, flags) -> bool
Batch decompilation. Decompile all or the specified functions

@param outfile (string): name of the output file
@param funcaddrs: (C++: const eavec_t *) list of functions to decompile. If nullptr or e
                  decompile all nonlib functions
@param flags (integer): Batch decompilation bits
@return: true if no internal error occurred and the user has not cancelled
         decompilation
```

IDAPython function idaapi.deep_copy_idcv quick reference

```
deep_copy_idcv(dst, src) -> error_t
Deep copy an IDC object. This function performs deep copy of idc objects. If
'src' is not an object, copy_idcv() will be called

@param dst (idaapi.idc_value_t):
@param src: (C++: const idc_value_t &) idc_value_t const &
```

IDAPython function idaapi.default_compiler quick reference

```
default_compiler() -> comp_t
Get compiler specified by inf.cc.
```

IDAPython function idaapi.define_exception quick reference

```
define_exception(code, name, desc, flags) -> char const *
Convenience function: define new exception code.

@param code (integer): exception code (cannot be 0)
@param name (string): exception name (cannot be empty or nullptr)
@param desc (string): exception description (maybe nullptr)
@param flags (integer): combination of Exception info flags
@return: failure message or nullptr. You must call store_exceptions() if this
         function succeeds
```

IDAPython function idaapi.define_stkvar quick reference

```
define_stkvar(pfn, name, off, flags, ti, nbytes) -> bool
Define/redefine a stack variable.

@param pfn (idaapi.func_t): pointer to function
@param name (string): variable name, nullptr means autogenerate a name
@param off (integer): offset of the stack variable in the frame. negative values denote
            local variables, positive - function arguments.
@param flags (integer): variable type flags (byte_flag() for a byte variable, for exampl
@param ti: (C++: const opinfo_t *) additional type information (like offsets, structs, e
@param nbytes (integer): number of bytes occupied by the variable
@return: success
```

IDAPython function idaapi.del_absbase quick reference

```
del_absbase(ea)

@param ea: ea_t
```

IDAPython function idaapi.del_aflags quick reference

```
del_aflags(ea)

@param ea: ea_t
```

IDAPython function idaapi.del_alignment quick reference

```
del_alignment(ea)

@param ea: ea_t
```

IDAPython function idaapi.del_array_parameters quick reference

```
del_array_parameters(ea)

@param ea: ea_t
```

IDAPython function idaapi.del_bpt quick reference

```
del_bpt(ea) -> bool
Delete an existing breakpoint in the debugged process. \sq{Type, Synchronous
function - available as request, Notification, none (synchronous function)}

@param bptloc (idaapi.bpt_location_t): Breakpoint location
del_bpt(bptloc) -> bool

@param bptloc: bpt_location_t const &
```

IDAPython function idaapi.del_bptgrp quick reference

```
del_bptgrp(name) -> bool
Delete a folder, bpt that were part of this folder are moved to the root folder
\sq{Type, Synchronous function, Notification, none (synchronous function)}

@param name (string): full path to the folder to be deleted
@return: success
```

IDAPython function idaapi.del_cref quick reference

```
del_cref(frm, to, expand) -> bool
Delete a code cross-reference.

@param from (integer): linear address of referencing instruction
@param to (integer): linear address of referenced instruction
@param expand (bool): policy to delete the referenced instruction
* 1: plan to delete the referenced instruction if it has no more references.
* 0: don't delete the referenced instruction even if no more cross-references
point to it
@retval true: if the referenced instruction will be deleted
```

IDAPython function idaapi.del_custom_data_type_ids quick reference

```
del_custom_data_type_ids(ea)

@param ea: ea_t
```

IDAPython function idaapi.del_debug_names quick reference

```
del_debug_names(ea1, ea2)

@param ea1: ea_t
@param ea2: ea_t
```

IDAPython function idaapi.del_dref quick reference

```
del_dref(frm, to)
Delete a data cross-reference.

@param from (integer): linear address of referencing instruction or data
@param to (integer): linear address of referenced data
```

IDAPython function idaapi.del_encoding quick reference

```
del_encoding(idx) -> bool
Delete an encoding The encoding is not actually removed because its index may be
used in strtype. So the deletion just clears the encoding name. The default
encoding cannot be deleted.

@param idx (integer): the encoding index (1-based)
```

IDAPython function idaapi.del_enum quick reference

```
del_enum(id)
Delete an enum type.

@param id (integer):
```

IDAPython function idaapi.del_enum_member quick reference

```
del_enum_member(id, value, serial, bmask) -> bool
Delete member of enum type.
```

```
@param id (integer):
@param value (integer):
@param serial: (C++: uchar)
@param bmask (integer):
```

IDAPython function idaapi.del_extra_cmt quick reference

```
del_extra_cmt(ea, what)

@param ea: ea_t
@param what: int
```

IDAPython function idaapi.del_fixup quick reference

```
del_fixup(source)
Delete fixup information.

@param source (integer):
```

IDAPython function idaapi.del_frame quick reference

```
del_frame(pfn) -> bool
Delete a function frame.

@param pfn (idaapi.func_t): pointer to function structure
@return: success
```

IDAPython function idaapi.del_func quick reference

```
del_func(ea) -> bool
Delete a function.

@param ea (integer): any address in the function entry chunk
@return: success
```

IDAPython function idaapi.del_global_name quick reference

```
del_global_name(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.del_hidden_range quick reference

```
del_hidden_range(ea) -> bool
Delete hidden range.

@param ea (integer): any address in the hidden range
@return: success
```

IDAPython function idaapi.del_hotkey quick reference

```
del_hotkey(pyctx) -> bool
Deletes a previously registered function hotkey

@param ctx: Hotkey context previously returned by add_hotkey()

@return: Boolean.
```

IDAPython function idaapi.del_idasgn quick reference

```
del_idasgn(n) -> int
Remove signature from the list of planned signatures.

@param n (integer): number of signature in the list (0..get_idasgn_qty()-1)
@return: IDASGN_OK, IDASGN_BADARG, IDASGN_APPLIED
```

IDAPython function idaapi.del_idc_func quick reference

```
Unregisters the specified IDC function

Delete an IDC function
```

IDAPython function idaapi.del_idc_hotkey quick reference

```
del_idc_hotkey(hotkey) -> bool

@param hotkey: char const *
```

IDAPython function idaapi.del_idcv_attr quick reference

```
del_idcv_attr(obj, attr) -> error_t
Delete an object attribute.

@param obj (idaapi.idc_value_t): variable that holds an object reference
```

```
@param attr (string): attribute name
@return: error code, eOk on success
```

IDAPython function idaapi.del_ind_purged quick reference

```
del_ind_purged(ea)

@param ea: ea_t
```

IDAPython function idaapi.del_item_color quick reference

```
del_item_color(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.del_items quick reference

```
del_items(ea, flags=0, nbytes=1, may_destroy=None) -> bool
Convert item (instruction/data) to unexplored bytes. The whole item (including
the head and tail bytes) will be destroyed. It is allowed to pass any address in
the item to this function

@param ea (integer): any address within the first item to delete
@param flags (integer): combination of Unexplored byte conversion flags
@param nbytes (integer): number of bytes in the range to be undefined
@param may_destroy: (C++: may_destroy_cb_t *) optional routine invoked before deleting a
                    callback returns false then item is not to be deleted and
                    operation fails
@return: true on sucessful operation, otherwise false
```

IDAPython function idaapi.del_local_name quick reference

```
del_local_name(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.del_mapping quick reference

```
del_mapping(ea)
Delete memory mapping range.

@param ea (integer): any address in the mapped range
```

IDAPython function idaapi.del_member_tinfo quick reference

```
del_member_tinfo(sptr, mptr) -> bool
Delete tinfo for given member.

@param sptr: (C++: struc_t *)
@param mptr: (C++: member_t *)
```

IDAPython function idaapi.del_named_type quick reference

```
del_named_type(ti, name, ntf_flags) -> bool
Delete information about a symbol.

@param ti (idaapi.til_t): type library
@param name (string): name of symbol
@param ntf_flags (integer): combination of Flags for named types
@return: success
```

IDAPython function idaapi.del_node_info quick reference

```
del_node_info(gid, node)
Delete the node_info_t for the given node.

@param gid: (C++: graph_id_t)
@param node (integer):
```

IDAPython function idaapi.del_numbered_type quick reference

```
del_numbered_type(ti, ordinal) -> bool
Delete a numbered type.

@param ti (idaapi.til_t):
@param ordinal (integer):
```

IDAPython function idaapi.del_op_tinfo quick reference

```
del_op_tinfo(ea, n)

@param ea: ea_t
@param n: int
```

IDAPython function idaapi.del_refinfo quick reference

```
del_refinfo(ea, n) -> bool

@param ea: ea_t
@param n: int
```

IDAPython function idaapi.del_regvar quick reference

```
del_regvar(pfn, ea1, ea2, canon) -> int
Delete a register variable definition.

@param pfn (idaapi.func_t): function in question
@param ea1 (integer): ,ea2: range of addresses within the function where the definition
                holds
@param canon (string): name of a general register
@param canon (string): name of a general register
@return: Register variable error codes
```

IDAPython function idaapi.del_segm quick reference

```
del_segm(ea, flags) -> bool
Delete a segment.

@param ea (integer): any address belonging to the segment
@param flags (integer): Segment modification flags
@retval 1: ok
@retval 0: failed, no segment at 'ea'.
```

IDAPython function idaapi.del_segment_translations quick reference

```
del_segment_translations(segstart)
Delete the translation list

@param segstart (integer): start address of the segment to delete translation list
```

IDAPython function idaapi.del_selector quick reference

```
del_selector(selector)
Delete mapping of a selector. Be wary of deleting selectors that are being used
in the program, this can make a mess in the segments.
```

```
@param selector: (C++: sel_t) number of selector to remove from the translation table
```

IDAPython function idaapi.del_source_linnum quick reference

```
del_source_linnum(ea)

@param ea: ea_t
```

IDAPython function idaapi.del_sourcefile quick reference

```
del_sourcefile(ea) -> bool
Delete information about the source file.

@param ea (integer): linear address
@return: success
```

IDAPython function idaapi.del_sreg_range quick reference

```
del_sreg_range(ea, rg) -> bool
Delete segment register range started at ea. When a segment register range is
deleted, the previous range is extended to cover the empty space. The segment
register range at the beginning of a segment cannot be deleted.

@param ea (integer): start_ea of the deleted range
@param rg (integer): the segment register number
@return: success
```

IDAPython function idaapi.del_stkpnt quick reference

```
del_stkpnt(pfn, ea) -> bool
Delete SP register change point.

@param pfn (idaapi.func_t): pointer to function. may be nullptr.
@param ea (integer): linear address
@return: success
```

IDAPython function idaapi.del_str_type quick reference

```
del_str_type(ea)

@param ea: ea_t
```

IDAPython function idaapi.del_struc quick reference

```
del_struc(sptr) -> bool
Delete a structure type.

@param sptr: (C++: struc_t *)
```

IDAPython function idaapi.del_struc_member quick reference

```
del_struc_member(sptr, offset) -> bool
Delete member at given offset.

@param sptr: (C++: struc_t *)
@param offset (integer):
```

IDAPython function idaapi.del_struc_members quick reference

```
del_struc_members(sptr, off1, off2) -> int
Delete members which occupy range of offsets (off1..off2).

@param sptr: (C++: struc_t *)
@param off1 (integer):
@param off2 (integer):
@return: number of deleted members or -1 on error
```

IDAPython function idaapi.del_switch_info quick reference

```
del_switch_info(ea)

@param ea: ea_t
```

IDAPython function idaapi.del_switch_parent quick reference

```
del_switch_parent(ea)

@param ea: ea_t
```

IDAPython function idaapi.del_til quick reference

```
del_til(name) -> bool
Unload a til file.

@param name (string): char const *
```

IDAPython function idaapi.del_tinfo quick reference

```
del_tinfo(ea)

@param ea: ea_t
```

IDAPython function idaapi.del_tinfo_attr quick reference

```
del_tinfo_attr(tif, key, make_copy) -> bool

@param tif: tinfo_t *
@param key: qstring const &
@param make_copy: bool
```

IDAPython function idaapi.del_tryblks quick reference

```
del_tryblks(range)
Delete try block information in the specified range.

@param range: (C++: const range_t &) the range to be cleared
```

IDAPython function idaapi.del_value quick reference

```
del_value(ea)
Delete byte value from flags. The corresponding byte becomes uninitialized.

@param ea (integer):
```

IDAPython function idaapi.del_vftable_ea quick reference

```
del_vftable_ea(ordinal) -> bool
Delete the address of a vftable instance for a vftable type.

@param ordinal (integer): ordinal number of a vftable type.
@return: success
```

IDAPython function idaapi.del_virt_module quick reference

```
del_virt_module(base) -> bool
```

```
@param base: ea_t const
```

IDAPython function idaapi.delay_slot_insn quick reference

```
delay_slot_insn(ea, bexec, fexec) -> bool

@param ea: ea_t *
@param bexec: bool *
@param fexec: bool *
```

IDAPython function idaapi.delete_extra_cmts quick reference

```
delete_extra_cmts(ea, what)

@param ea: ea_t
@param what: int
```

IDAPython function idaapi.delete_imports quick reference

```
delete_imports()
Delete all imported modules information.
```

IDAPython function idaapi.delete_menu quick reference

```
delete_menu(name) -> bool
Delete an existing menu

@param name (string): name of menu
@return: success
```

IDAPython function idaapi.delete_mutable_graph quick reference

```
delete_mutable_graph(g)
Delete graph object.
@warning: use this only if you are dealing with mutable_graph_t instances that
          have not been used together with a graph_viewer_t. If you have called
          set_viewer_graph() with your graph, the graph's lifecycle will be
          managed by the viewer, and you shouldn't interfere with it

@param g: (C++: mutable_graph_t *)
```

IDAPython function idaapi.delete_switch_table quick reference

```
delete_switch_table(jump_ea, si)

@param jump_ea: ea_t
@param si: switch_info_t const &
```

IDAPython function idaapi.delete_toolbar quick reference

```
delete_toolbar(name) -> bool
Delete an existing toolbar

@param name (string): name of toolbar
@return: success
```

IDAPython function idaapi.delete_unreferenced_stkvars quick reference

```
delete_unreferenced_stkvars(pfn) -> int

@param pfn: func_t *
```

IDAPython function idaapi.delete_wrong_stkvar_ops quick reference

```
delete_wrong_stkvar_ops(pfn) -> int

@param pfn: func_t *
```

IDAPython function idaapi.delinf quick reference

```
delinf(tag) -> bool
Undefine a program specific information

@param tag: (C++: inftag_t) one of inftag_t constants
@return: success
```

IDAPython function idaapi.demangle_name quick reference

```
demangle_name(name, disable_mask, demreq=DQT_FULL) -> str
Demangle a name.

@param name (string): char const *
```

```
@param disable_mask (integer):
@param demreq: (C++: demreq_type_t) enum demreq_type_t
```

IDAPython function idaapi.deref_idcv quick reference

```
deref_idcv(v, vref_flags) -> idc_value_t
Dereference a VT_REF variable.

@param v (idaapi.idc_value_t): variable to dereference
@param vref_flags (integer): Dereference IDC variable flags
@return: pointer to the dereference result or nullptr. If returns nullptr,
         qerrno is set to eExecBadRef "Illegal variable reference"
```

IDAPython function idaapi.deref_ptr quick reference

```
deref_ptr(ptr_ea, tif, closure_obj=None) -> bool
Dereference a pointer.

@param ptr_ea: (C++: ea_t *) in/out parameter
* in: address of the pointer
* out: the pointed address
@param tif (idaapi.tinfo_t): type of the pointer
@param closure_obj: (C++: ea_t *) closure object (not used yet)
@return: success
```

IDAPython function idaapi.dereference quick reference

```
Dereference a pointer. This function dereferences a pointer expression. It
performs the following conversion: "ptr" => "*ptr" It can handle discrepancies
in the pointer type and the access size.

@return: dereferenced expression
```

IDAPython function idaapi.deserialize_tinfo quick reference

```
deserialize_tinfo(tif, til, ptype, pfields, pfldcmts) -> bool

@param tif: tinfo_t *
@param til: til_t const *
@param ptype: type_t const **
@param pfields: p_list const **
@param pfldcmts: p_list const **
```

IDAPython function idaapi.detach_action_from_menu quick reference

```
detach_action_from_menu(menupath, name) -> bool
Detach an action from the menu (ui_detach_action_from_menu).

@param menupath (string): path to the menu item
@param name (string): the action name
@return: success
```

IDAPython function idaapi.detach_action_from_popup quick reference

```
detach_action_from_popup(widget, name) -> bool
Remove a previously-registered action, from the list of 'permanent' context menu
actions for this widget (ui_detach_action_from_popup). This only makes sense if
the action has been added to 'widget's list of permanent popup actions by
calling attach_action_to_popup in 'permanent' mode.

@param widget(a Widget SWIG wrapper class): target widget
@param name (string): action name
```

IDAPython function idaapi.detach_action_from_toolbar quick reference

```
detach_action_from_toolbar(toolbar_name, name) -> bool
Detach an action from the toolbar (ui_detach_action_from_toolbar).

@param toolbar_name (string): the name of the toolbar
@param name (string): the action name
@return: success
```

IDAPython function idaapi.detach_custom_data_format quick reference

```
detach_custom_data_format(dtid, dfid) -> bool
Detach the data format from the data type. Unregistering a custom data type
detaches all attached data formats, no need to detach them explicitly. You still
need unregister them. Unregistering a custom data format detaches it from all
attached data types.

@param dtid (integer): data type id to detach data format from
@param dfid (integer): data format id to detach
@retval true: ok
```

```
@retval false: no such `dtid', or no such `dfid', or the data format was not
               attached to the data type
```

IDAPython function idaapi.detach_process quick reference

```
detach_process() -> bool
Detach the debugger from the debugged process. \sq{Type, Asynchronous function -
available as Request, Notification, dbg_process_detach}
```

IDAPython function idaapi.diff_trace_file quick reference

```
diff_trace_file(NONNULL_filename) -> bool
Show difference between the current trace and the one from 'filename'.

@param NONNULL_filename (string): char const *
```

IDAPython function idaapi.dirtree_cursor_t_root_cursor quick reference

```
dirtree_cursor_t_root_cursor() -> dirtree_cursor_t
```

IDAPython function idaapi.dirtree_t_errstr quick reference

```
dirtree_t_errstr(err) -> char const *

@param err: enum dterr_t
```

IDAPython function idaapi.disable_bblk_trace quick reference

```
disable_bblk_trace() -> bool
```

IDAPython function idaapi.disable_bpt quick reference

```
disable_bpt(ea) -> bool

@param ea: ea_t

disable_bpt(bptloc) -> bool

@param bptloc: bpt_location_t const &
```

IDAPython function idaapi.disable_flags quick reference

```
disable_flags(start_ea, end_ea) -> error_t
Deallocate flags for address range. Exit with an error message if not enough
disk space (this may occur too).

@param start_ea (integer): should be lower than end_ea.
@param end_ea (integer): does not belong to the range.
@return: 0 if ok, otherwise return error code
```

IDAPython function idaapi.disable_func_trace quick reference

```
disable_func_trace() -> bool
```

IDAPython function idaapi.disable_insn_trace quick reference

```
disable_insn_trace() -> bool
```

IDAPython function idaapi.disable_script_timeout quick reference

```
disable_script_timeout()
Disables the script timeout and hides the script wait box.
Calling L{set_script_timeout} will not have any effects until the script is compiled and

@return: None
```

IDAPython function idaapi.disable_step_trace quick reference

```
disable_step_trace() -> bool
```

IDAPython function idaapi.display_copyright_warning quick reference

```
display_copyright_warning() -> bool
Display copyright warning (ui_copywarn).

@return: yes/no
```

IDAPython function idaapi.display_gdl quick reference

```
display_gdl(fname) -> int
Display GDL file by calling wingraph32. The exact name of the grapher is taken
```

```
from the configuration file and set up by setup_graph_subsystem(). The path
should point to a temporary file: when wingraph32 succeeds showing the graph,
the input file will be deleted.

@param fname (string): char const *
@return: error code from os, 0 if ok
```

IDAPython function idaapi.display_widget quick reference

```
display_widget(widget, options, dest_ctrl=None)
Display a widget, dock it if not done before

@param widget(a Widget SWIG wrapper class): widget to display
@param options (integer): Widget open flags
@param dest_ctrl (string): where to dock: if nullptr or invalid then use the active
                  docker if there is not create a new tab relative to current
                  active tab
```

IDAPython function idaapi.double_flag quick reference

```
double_flag() -> flags64_t
Get a flags64_t representing a double.
```

IDAPython function idaapi.dstr quick reference

```
dstr(tif) -> char const *
Print the specified type info. This function can be used from a debugger by
typing "tif->dstr()"

@param tif: (C++: const tinfo_t *) tinfo_t const *
```

IDAPython function idaapi.dstr_tinfo quick reference

```
dstr_tinfo(tif) -> char const *

@param tif: tinfo_t const *
```

IDAPython function idaapi.dummy_ptrtype quick reference

```
dummy_ptrtype(ptrsize, isfp) -> tinfo_t
Generate a dummy pointer type
```

```
@param ptrsize (integer): size of pointed object
@param isfp (bool): is floating point object?
```

IDAPython function idaapi.dump__func__type__data quick reference

```
dump_func_type_data(fti, praloc_bits) -> str
Use func_type_data_t::dump()

@param fti: (C++: const func_type_data_t &) func_type_data_t const &
@param praloc_bits (integer):
```

IDAPython function idaapi.dword__flag quick reference

```
dword_flag() -> flags64_t
Get a flags64_t representing a double word.
```

IDAPython function idaapi.ea2node quick reference

```
ea2node(ea) -> nodeidx_t
Get netnode for the specified address.

@param ea (integer):
```

IDAPython function idaapi.ea2str quick reference

```
ea2str(ea) -> str
Convert linear address to UTF-8 string.

@param ea (integer):
```

IDAPython function idaapi.ea__array__frompointer quick reference

```
ea_array_frompointer(t) -> ea_array

@param t: ea_t *
```

IDAPython function idaapi.ea__pointer__frompointer quick reference

```
ea_pointer_frompointer(t) -> ea_pointer

@param t: ea_t *
```

IDAPython function idaapi.ea_viewer_history_push_and_jump quick reference

```
ea_viewer_history_push_and_jump(v, ea, x, y, lnnum) -> bool
Push current location in the history and jump to the given location
(ui_ea_viewer_history_push_and_jump). This will jump in the given ea viewer and
also in other synchronized views.

@param v(a Widget SWIG wrapper class): ea viewer
@param ea (integer): jump destination
@param x (integer): ,y: coords on screen
@param lnnum (integer): desired line number of given address
@param lnnum (integer): desired line number of given address
```

IDAPython function idaapi.eamap_begin quick reference

```
eamap_begin(map) -> eamap_iterator_t
Get iterator pointing to the beginning of eamap_t.

@param map: (C++: const eamap_t *) eamap_t const *
```

IDAPython function idaapi.eamap_clear quick reference

```
eamap_clear(map)
Clear eamap_t.

@param map: (C++: eamap_t *)
```

IDAPython function idaapi.eamap_end quick reference

```
eamap_end(map) -> eamap_iterator_t
Get iterator pointing to the end of eamap_t.

@param map: (C++: const eamap_t *) eamap_t const *
```

IDAPython function idaapi.eamap_erase quick reference

```
eamap_erase(map, p)
Erase current element from eamap_t.

@param map: (C++: eamap_t *)
@param p: (C++: eamap_iterator_t)
```

IDAPython function idaapi.eamap_find quick reference

```
eamap_find(map, key) -> eamap_iterator_t
Find the specified key in eamap_t.

@param map: (C++: const eamap_t *) eamap_t const *
@param key: (C++: const ea_t &) ea_t const &
```

IDAPython function idaapi.eamap_first quick reference

```
eamap_first(p) -> ea_t const &
Get reference to the current map key.

@param p: (C++: eamap_iterator_t)
```

IDAPython function idaapi.eamap_free quick reference

```
eamap_free(map)
Delete eamap_t instance.

@param map: (C++: eamap_t *)
```

IDAPython function idaapi.eamap_insert quick reference

```
eamap_insert(map, key, val) -> eamap_iterator_t
Insert new (ea_t, cinsnptrvec_t) pair into eamap_t.

@param map: (C++: eamap_t *)
@param key: (C++: const ea_t &) ea_t const &
@param val: (C++: const cinsnptrvec_t &) cinsnptrvec_t const &
```

IDAPython function idaapi.eamap_new quick reference

```
eamap_new() -> eamap_t
Create a new eamap_t instance.
```

IDAPython function idaapi.eamap_next quick reference

```
eamap_next(p) -> eamap_iterator_t
Move to the next element.

@param p: (C++: eamap_iterator_t)
```

IDAPython function idaapi.eamap_prev quick reference

```
eamap_prev(p) -> eamap_iterator_t
Move to the previous element.

@param p: (C++: eamap_iterator_t)
```

IDAPython function idaapi.eamap_second quick reference

```
eamap_second(p) -> cinsnptrvec_t
Get reference to the current map value.

@param p: (C++: eamap_iterator_t)
```

IDAPython function idaapi.eamap_size quick reference

```
eamap_size(map) -> size_t
Get size of eamap_t.

@param map: (C++: eamap_t *)
```

IDAPython function idaapi.ecleaz quick reference

```
ecleaz(x)

@param x: unsigned short [(6+3)]
```

IDAPython function idaapi.eclose quick reference

```
eclose(fp)

@param fp: FILE *
```

IDAPython function idaapi.edit_manual_regions quick reference

```
edit_manual_regions()
```

IDAPython function idaapi.enable_auto quick reference

```
enable_auto(enable) -> bool
Temporarily enable/disable autoanalyzer. Not user-facing, but rather because IDA
sometimes need to turn AA on/off regardless of inf.s_genflags:INFFL_AUTO
```

```
@param enable (bool):
@return: old state
```

IDAPython function idaapi.enable_bblk_trace quick reference

```
enable_bblk_trace(enable=True) -> bool

@param enable: bool
```

IDAPython function idaapi.enable_bpt quick reference

```
enable_bpt(ea, enable=True) -> bool

@param ea: ea_t
@param enable: bool

enable_bpt(bptloc, enable=True) -> bool

@param bptloc: bpt_location_t const &
@param enable: bool
```

IDAPython function idaapi.enable_bptgrp quick reference

```
enable_bptgrp(bptgrp_name, enable=True) -> int
Enable (or disable) all bpts in a folder \sq{Type, Synchronous function,
Notification, none (synchronous function)}

@param bptgrp_name (string): absolute path to the folder
@param enable (bool): by default true, enable bpts, false disable bpts
@return: -1 an error occured 0 no changes >0 nubmers of bpts udpated
```

IDAPython function idaapi.enable_chooser_item_attrs quick reference

```
enable_chooser_item_attrs(chooser_caption, enable) -> bool
Enable item-specific attributes for chooser items
(ui_enable_chooser_item_attrs). For example: color list items differently
depending on a criterium.
If enabled, the chooser will generate ui_get_chooser_item_attrs
events that can be intercepted by a plugin to modify the item attributes.
This event is generated only in the GUI version of IDA.
Specifying CH_ATTRS bit at the chooser creation time has the same effect.
```

```
@param chooser_caption (string): char const *
@param enable (bool):
@return: success
```

IDAPython function idaapi.enable_extlang_python quick reference

```
enable_extlang_python(enable)
Enables or disables Python extlang.
When enabled, all expressions will be evaluated by Python.

@param enable: Set to True to enable, False otherwise
```

IDAPython function idaapi.enable_flags quick reference

```
enable_flags(start_ea, end_ea, stt) -> error_t
Allocate flags for address range. This function does not change the storage type
of existing ranges. Exit with an error message if not enough disk space.

@param start_ea (integer): should be lower than end_ea.
@param end_ea (integer): does not belong to the range.
@param stt: (C++: storage_type_t)
@return: 0 if ok, otherwise an error code
```

IDAPython function idaapi.enable_func_trace quick reference

```
enable_func_trace(enable=True) -> bool

@param enable: bool
```

IDAPython function idaapi.enable_insn_trace quick reference

```
enable_insn_trace(enable=True) -> bool

@param enable: bool
```

IDAPython function idaapi.enable_manual_regions quick reference

```
enable_manual_regions(enable)

@param enable: bool
```

IDAPython function idaapi.enable_python_cli quick reference

```
enable_python_cli(enable)

@param enable: bool
```

IDAPython function idaapi.enable_step_trace quick reference

```
enable_step_trace(enable=1) -> bool

@param enable: int
```

IDAPython function idaapi.encoding_from_strtype quick reference

```
encoding_from_strtype(strtype) -> char const *
Get encoding name for this strtype
@retval nullptr: if STRTYPE has an incorrent encoding index
@retval empty: string if the encoding was deleted

@param strtype: (C++: int32)
```

IDAPython function idaapi.end_type_updating quick reference

```
end_type_updating(utp)
Mark the end of a large update operation on the types (see
begin_type_updating())

@param utp: (C++: update_type_t) enum update_type_t
```

IDAPython function idaapi.enum_flag quick reference

```
enum_flag() -> flags64_t
see FF_opbits
```

IDAPython function idaapi.enum_import_names quick reference

```
enum_import_names(mod_index, py_cb) -> int
Enumerate imports from a specific module.
Please refer to ex_imports.py example.

@param mod_index: The module index
@param callback: A callable object that will be invoked with an ea, name (could be None)
@return: 1-finished ok, -1 on error, otherwise callback return value (<=0)
```

IDAPython function idaapi.enum_type_data_t___set_bf quick reference

```
enum_type_data_t__set_bf(_this, bf) -> bool

@param _this: enum_type_data_t *
@param bf: bool
```

IDAPython function idaapi.enumerate_files quick reference

```
enumerate_files(path, fname, callback) -> PyObject *
Enumerate files in the specified directory while the callback returns 0.

@param path: directory to enumerate files in
@param fname: mask of file names to enumerate
@param callback: a callable object that takes the filename as
                 its first argument and it returns 0 to continue
                 enumeration or non-zero to stop enumeration.
@return:     None in case of script errors
    tuple(code, fname) : If the callback returns non-zero
```

IDAPython function idaapi.enumerate_files2 quick reference

```
enumerate_files2(answer, answer_size, path, fname, fv) -> int

@param answer: char *
@param answer_size: size_t
@param path: char const *
@param fname: char const *
@param fv: file_enumerator_t &
```

IDAPython function idaapi.equal_bytes quick reference

```
equal_bytes(ea, image, mask, len, bin_search_flags) -> bool
Compare 'len' bytes of the program starting from 'ea' with 'image'.

@param ea (integer): linear address
@param image: (C++: const uchar *) bytes to compare with
@param mask: (C++: const uchar *) array of mask bytes, it's length is 'len'. if the flag
             BIN_SEARCH_BITMASK is passsed, 'bitwise AND' is used to compare. if
             not; 1 means to perform the comparison of the corresponding byte. 0
             means not to perform. if mask == nullptr, then all bytes of 'image'
```

```
            will be compared. if mask == SKIP_FF_MASK then 0xFF bytes will be
            skipped
@param len (integer): length of block to compare in bytes.
@param bin_search_flags (integer): combination of Search flags
@retval 1: equal
@retval 0: not equal
```

IDAPython function idaapi.error quick reference

```
error(format)
Display a fatal message in a message box and quit IDA

@param format: message to print
```

IDAPython function idaapi.eval_expr quick reference

```
eval_expr(rv, where, line) -> str
Compile and calculate an expression.

@param rv (idaapi.idc_value_t): pointer to the result
@param where (integer): the current linear address in the addressing space of the progra
             being disassembled. If will be used to resolve names of local
             variables etc. if not applicable, then should be BADADDR.
@param line (string): the expression to evaluate
@retval true: ok
@retval false: error, see errbuf
```

IDAPython function idaapi.eval_idc_expr quick reference

```
eval_idc_expr(rv, where, line) -> str
Same as eval_expr(), but will always use the IDC interpreter regardless of the
currently installed extlang.

@param rv (idaapi.idc_value_t):
@param where (integer):
@param line: char const *
```

IDAPython function idaapi.exec_idc_script quick reference

```
exec_idc_script(result, path, func, args, argsnum) -> str
Compile and execute IDC function(s) from file.

@param result (idaapi.idc_value_t): ptr to idc_value_t to hold result of the function. I
```

```
                fails, this variable will contain the exception information. You
                may pass nullptr if you are not interested in the returned value.
@param path (string): text file containing text of IDC functions
@param func (string): function name to execute
@param args: (C++: const idc_value_t) array of parameters
@param argsnum (integer): number of parameters to pass to 'fname' This number should be
                 equal to number of parameters the function expects.
@retval true: ok
@retval false: error, see errbuf
```

IDAPython function idaapi.exec_system_script quick reference

```
exec_system_script(file, complain_if_no_file=True) -> bool
Compile and execute "main" function from system file.

@param file (string): file name with IDC function(s). The file will be searched using
             get_idc_filename().
@param complain_if_no_file (bool): * 1: display warning if the file is not found
* 0: don't complain if file doesn't exist
@retval 1: ok, file is compiled and executed
@retval 0: failure, compilation or execution error, warning is displayed
```

IDAPython function idaapi.execute_sync quick reference

```
execute_sync(py_callable, reqf) -> int
Executes a function in the context of the main thread.
If the current thread not the main thread, then the call is queued and
executed afterwards.

@param py_callable: A python callable object, must return an integer value
@param reqf: one of MFF_ flags
@return: -1 or the return value of the callable
```

IDAPython function idaapi.execute_ui_requests quick reference

```
execute_ui_requests(py_list) -> bool
Inserts a list of callables into the UI message processing queue.
When the UI is ready it will call one callable.
A callable can request to be called more than once if it returns True.

@param callable_list: A list of python callable objects.
@note: A callable should return True if it wants to be called more than once.
@return: Boolean. False if the list contains a non callable item
```

IDAPython function idaapi.exist quick reference

```
exist(n) -> bool

@param n: netnode const &
```

IDAPython function idaapi.exist_bpt quick reference

```
exist_bpt(ea) -> bool
Does a breakpoint exist at the given location?

@param ea (integer):
```

IDAPython function idaapi.exists_fixup quick reference

```
exists_fixup(source) -> bool
Check that a fixup exists at the given address.

@param source (integer):
```

IDAPython function idaapi.exit_process quick reference

```
exit_process() -> bool
Terminate the debugging of the current process. \sq{Type, Asynchronous function
- available as Request, Notification, dbg_process_exit}
```

IDAPython function idaapi.expand_struc quick reference

```
expand_struc(sptr, offset, delta, recalc=True) -> bool
Expand/Shrink structure type.

@param sptr: (C++: struc_t *)
@param offset (integer):
@param delta (integer):
@param recalc (bool):
```

IDAPython function idaapi.extend_sign quick reference

```
extend_sign(v, nbytes, sign_extend) -> uint64
Sign-, or zero-extend the value 'v' to occupy 64 bits. The value 'v' is
considered to be of size 'nbytes'.

@param v (integer):
```

```
@param nbytes (integer):
@param sign_extend (bool):
```

IDAPython function idaapi.extract_argloc quick reference

```
extract_argloc(vloc, ptype, forbid_stkoff) -> bool
Deserialize an argument location. Argument FORBID_STKOFF checks location type.
It can be used, for example, to check the return location of a function that
cannot return a value in the stack

@param vloc: (C++: argloc_t *)
@param ptype: (C++: const type_t **) type_t const **
@param forbid_stkoff (bool):
```

IDAPython function idaapi.extract_module_from_archive quick reference

```
extract_module_from_archive(fname, is_remote=False) -> (NoneType, NoneType), (str, str)
Extract a module for an archive file. Parse an archive file, show the list of
modules to the user, allow him to select a module, extract the selected module
to a file (if the extract module is an archive, repeat the process). This
function can handle ZIP, AR, AIXAR, OMFLIB files. The temporary file will be
automatically deleted by IDA at the end.

@param filename: (C++: char *) in: input file. out: name of the selected module.
@param is_remote (bool): is the input file remote?
@retval true: ok
@retval false: something bad happened (error message has been displayed to the
               user)
```

IDAPython function idaapi.extract_name quick reference

```
extract_name(line, x) -> str
Extract a name or address from the specified string.

@param line (string): input string
@param x (integer): x coordinate of cursor
@return: -1 if cannot extract. otherwise length of the name
```

IDAPython function idaapi.f_any quick reference

```
f_any(arg1, arg2) -> bool
Helper function to accept any address.
```

```
@param arg1: flags64_t
@param arg2
```

IDAPython function idaapi.f_has_cmt quick reference

```
f_has_cmt(f, arg2) -> bool

@param f: flags64_t
@param arg2
```

IDAPython function idaapi.f_has_dummy_name quick reference

```
f_has_dummy_name(f, arg2) -> bool
Does the current byte have dummy (auto-generated, with special prefix) name?

@param f (integer):
@param arg2
```

IDAPython function idaapi.f_has_extra_cmts quick reference

```
f_has_extra_cmts(f, arg2) -> bool

@param f: flags64_t
@param arg2
```

IDAPython function idaapi.f_has_name quick reference

```
f_has_name(f, arg2) -> bool
Does the current byte have non-trivial (non-dummy) name?

@param f (integer):
@param arg2
```

IDAPython function idaapi.f_has_user_name quick reference

```
f_has_user_name(F, arg2) -> bool
Does the current byte have user-specified name?

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_has_xref quick reference

```
f_has_xref(f, arg2) -> bool
Does the current byte have cross-references to it?

@param f (integer):
@param arg2
```

IDAPython function idaapi.f_is_align quick reference

```
f_is_align(F, arg2) -> bool
See is_align()

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_byte quick reference

```
f_is_byte(F, arg2) -> bool
See is_byte()

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_code quick reference

```
f_is_code(F, arg2) -> bool
Does flag denote start of an instruction?

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_custom quick reference

```
f_is_custom(F, arg2) -> bool
See is_custom()

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_data quick reference

```
f_is_data(F, arg2) -> bool
```

```
Does flag denote start of data?

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_double quick reference

```
f_is_double(F, arg2) -> bool
See is_double()

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_dword quick reference

```
f_is_dword(F, arg2) -> bool
See is_dword()

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_float quick reference

```
f_is_float(F, arg2) -> bool
See is_float()

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_head quick reference

```
f_is_head(F, arg2) -> bool
Does flag denote start of instruction OR data?

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_not_tail quick reference

```
f_is_not_tail(F, arg2) -> bool
Does flag denote tail byte?
```

```
@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_oword quick reference

```
f_is_oword(F, arg2) -> bool
See is_oword()

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_pack_real quick reference

```
f_is_pack_real(F, arg2) -> bool
See is_pack_real()

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_qword quick reference

```
f_is_qword(F, arg2) -> bool
See is_qword()

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_strlit quick reference

```
f_is_strlit(F, arg2) -> bool
See is_strlit()

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_struct quick reference

```
f_is_struct(F, arg2) -> bool
See is_struct()

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_tail quick reference

```
f_is_tail(F, arg2) -> bool
Does flag denote tail byte?

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_tbyte quick reference

```
f_is_tbyte(F, arg2) -> bool
See is_tbyte()

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_word quick reference

```
f_is_word(F, arg2) -> bool
See is_word()

@param F (integer):
@param arg2
```

IDAPython function idaapi.f_is_yword quick reference

```
f_is_yword(F, arg2) -> bool
See is_yword()

@param F (integer):
@param arg2
```

IDAPython function idaapi.file2base quick reference

```
file2base(li, pos, ea1, ea2, patchable) -> int
Load portion of file into the database. This function will include (ea1..ea2)
into the addressing space of the program (make it enabled).

@param li: (C++: linput_t *) pointer of input source
@param pos: (C++: qoff64_t) position in the file
@param ea1 (integer): ,ea2: range of destination linear addresses
```

```
@param patchable (integer): should the kernel remember correspondence of file offsets to
                  linear addresses.
@retval 1: ok
@retval 0: read error, a warning is displayed
@note: The storage type of the specified range will be changed to STT_VA.
@param patchable (integer): should the kernel remember correspondence of file offsets to
                  linear addresses.
@retval 1: ok
@retval 0: read error, a warning is displayed
@note: The storage type of the specified range will be changed to STT_VA.
```

IDAPython function idaapi.find_binary quick reference

```
find_binary(arg1, arg2, arg3, arg4, arg5) -> ea_t
Deprecated. Please use idaapi.bin_search() instead.

@param arg1: ea_t
@param arg2: ea_t
@param arg3: char const *
@param arg4: int
@param arg5: int
```

IDAPython function idaapi.find_bpt quick reference

```
find_bpt(bptloc, bpt) -> bool
Find a breakpoint by location. \sq{Type, Synchronous function - available as
request, Notification, none (synchronous function)}

@param bptloc (idaapi.bpt_location_t): Breakpoint location
@param bpt: (C++: bpt_t *) bpt is filled if the breakpoint was found
```

IDAPython function idaapi.find_byte quick reference

```
find_byte(sEA, size, value, bin_search_flags) -> ea_t
Find forward a byte with the specified value (only 8-bit value from the
database). example: ea=4 size=3 will inspect addresses 4, 5, and 6

@param sEA (integer): linear address
@param size (integer): number of bytes to inspect
@param value: (C++: uchar) value to find
@param bin_search_flags (integer): combination of Search flags
@return: address of byte or BADADDR
```

IDAPython function idaapi.find_byter quick reference

```
find_byter(sEA, size, value, bin_search_flags) -> ea_t
Find reverse a byte with the specified value (only 8-bit value from the
database). example: ea=4 size=3 will inspect addresses 6, 5, and 4

@param sEA (integer): the lower address of the search range
@param size (integer): number of bytes to inspect
@param value: (C++: uchar) value to find
@param bin_search_flags (integer): combination of Search flags
@return: address of byte or BADADDR
```

IDAPython function idaapi.find_code quick reference

```
find_code(ea, sflag) -> ea_t
Find next code address.

@param ea (integer):
@param sflag (integer):
```

IDAPython function idaapi.find_custom_data_format quick reference

```
find_custom_data_format(name) -> int
Get id of a custom data format.

@param name (string): name of the custom data format
@return: id or -1
```

IDAPython function idaapi.find_custom_data_type quick reference

```
find_custom_data_type(name) -> int
Get id of a custom data type.

@param name (string): name of the custom data type
@return: id or -1
```

IDAPython function idaapi.find_custom_fixup quick reference

```
find_custom_fixup(name) -> fixup_type_t
Get id of a custom fixup handler.

@param name (string): name of the custom fixup handler
```

```
@return: id with FIXUP_CUSTOM bit set or 0
```

IDAPython function idaapi.find_custom_refinfo quick reference

```
find_custom_refinfo(name) -> int
Get id of a custom refinfo type.

@param name (string): char const *
```

IDAPython function idaapi.find_data quick reference

```
find_data(ea, sflag) -> ea_t
Find next data address.

@param ea (integer):
@param sflag (integer):
```

IDAPython function idaapi.find_defined quick reference

```
find_defined(ea, sflag) -> ea_t
Find next ea that is the start of an instruction or data.

@param ea (integer):
@param sflag (integer):
```

IDAPython function idaapi.find_error quick reference

```
find_error(ea, sflag) -> ea_t
Find next error or problem.

@param ea (integer):
@param sflag (integer):
```

IDAPython function idaapi.find_free_chunk quick reference

```
find_free_chunk(start, size, alignment) -> ea_t
Search for a hole in the addressing space of the program.

@param start (integer): Address to start searching from
@param size (integer): Size of the desired empty range
@param alignment (integer): Alignment bitmask, must be a pow2-1. (for example, 0xF would
                  align the returned range to 16 bytes).
```

```
@return: Start of the found empty range or BADADDR
```

IDAPython function idaapi.find_free_selector quick reference

```
find_free_selector() -> sel_t
Find first unused selector.

@return: a number >= 1
```

IDAPython function idaapi.find_func_bounds quick reference

```
find_func_bounds(nfn, flags) -> int
Determine the boundaries of a new function. This function tries to find the
start and end addresses of a new function. It calls the module with
processor_t::func_bounds in order to fine tune the function boundaries.

@param nfn (idaapi.func_t): structure to fill with information \ nfn->start_ea points to
            start address of the new function.
@param flags (integer): Find function bounds flags
@return: Find function bounds result codes
```

IDAPython function idaapi.find_idc_class quick reference

```
find_idc_class(name) -> idc_class_t *
Find an existing IDC class by its name.

@param name (string): name of the class
@return: pointer to the class or nullptr. The returned pointer is valid until a
         new call to add_idc_class()
```

IDAPython function idaapi.find_idc_func quick reference

```
find_idc_func(prefix, n=0) -> str

@param prefix: char const *
@param n: int
```

IDAPython function idaapi.find_idc_gvar quick reference

```
find_idc_gvar(name) -> idc_value_t
Find an existing global IDC variable by its name.
```

```
@param name (string): name of the global variable
@return: pointer to the variable or nullptr. NB: the returned pointer is valid
         until a new global var is added. FIXME: it is difficult to use this
         function in a thread safe manner
```

IDAPython function idaapi.find_imm quick reference

```
find_imm(ea, sflag, search_value) -> ea_t
Find next immediate operand with the given value.

@param ea (integer):
@param sflag (integer):
@param search_value (integer):
```

IDAPython function idaapi.find_not_func quick reference

```
find_not_func(ea, sflag) -> ea_t
Find next code address that does not belong to a function.

@param ea (integer):
@param sflag (integer):
```

IDAPython function idaapi.find_notype quick reference

```
find_notype(ea, sflag) -> ea_t
Find next operand without any type info.

@param ea (integer):
@param sflag (integer):
```

IDAPython function idaapi.find_plugin quick reference

```
find_plugin(name, load_if_needed=False) -> plugin_t *
Find a user-defined plugin and optionally load it.

@param name (string): short plugin name without path and extension, or absolute path to
             the file name
@param load_if_needed (bool): if the plugin is not present in the memory, try to load
                        it
@return: pointer to plugin description block
```

IDAPython function idaapi.find_reg_access quick reference

```
find_reg_access(out, start_ea, end_ea, regname, sflag) -> ea_t
Find access to a register.

@param out: (C++: struct reg_access_t *) pointer to the output buffer. must be non-null.
            contains info about the found register. upon failed search for a
            read access out->range contains the info about the non-redefined
            parts of the register.
@param start_ea (integer): starting address
@param end_ea (integer): ending address. BADADDR means that the end limit is missing.
               otherwise, if the search direction is SEARCH_UP, END_EA must be
               lower than START_EA.
@param regname (string): the register to search for.
@param sflag (integer): combination of Search flags bits.
@note: This function does not care about the control flow and probes all
       instructions in the specified range, starting from START_EA. Only direct
       references to registers are detected. Function calls and system traps are
       ignored.
@return: the found address. BADADDR if not found or error.
```

IDAPython function idaapi.find_regvar quick reference

```
find_regvar(pfn, ea1, ea2, canon, user) -> regvar_t
Find a register variable definition.

@param pfn (idaapi.func_t): function in question
@param ea1: ea_t
@param canon (string): name of a general register
@param canon (string): name of a general register
@param user: char const *

@return: nullptr-not found, otherwise ptr to regvar_t
find_regvar(pfn, ea, canon) -> regvar_t

@param pfn: func_t *
@param ea: ea_t
@param canon: char const *
```

IDAPython function idaapi.find_selector quick reference

```
find_selector(base) -> sel_t
Find a selector that has mapping to the specified paragraph.

@param base (integer): paragraph to search in the translation table
```

```
@return: selector value or base
```

IDAPython function idaapi.find_suspop quick reference

```
find_suspop(ea, sflag) -> ea_t
Find next suspicious operand.

@param ea (integer):
@param sflag (integer):
```

IDAPython function idaapi.find_syseh quick reference

```
find_syseh(ea) -> ea_t
Find the start address of the system eh region including the argument.

@param ea (integer): search address
@return: start address of surrounding tryblk, otherwise BADADDR
```

IDAPython function idaapi.find_text quick reference

```
find_text(start_ea, y, x, ustr, sflag) -> ea_t
See search()

@param start_ea (integer):
@param y (integer):
@param x (integer):
@param ustr (string): char const *
@param sflag (integer):
```

IDAPython function idaapi.find_tinfo_udt_member quick reference

```
find_tinfo_udt_member(udm, typid, strmem_flags) -> int

@param udm: udt_member_t *
@param typid: uint32
@param strmem_flags: int
```

IDAPython function idaapi.find_unknown quick reference

```
find_unknown(ea, sflag) -> ea_t
Find next unexplored address.
```

```
@param ea (integer):
@param sflag (integer):
```

IDAPython function idaapi.find_widget quick reference

```
find_widget(caption) -> TWidget *
Find widget with the specified caption (only gui version) (ui_find_widget). NB:
this callback works only with the tabbed widgets!

@param caption (string): title of tab, or window title if widget is not tabbed
@return: pointer to the TWidget, nullptr if none is found
```

IDAPython function idaapi.first_idcv_attr quick reference

```
first_idcv_attr(obj) -> char const *

@param obj: idc_value_t const *
```

IDAPython function idaapi.first_named_type quick reference

```
first_named_type(ti, ntf_flags) -> char const *
Enumerate types.

@param ti (idaapi.til_t): type library. nullptr means the local type library for the cu
          database.
@param ntf_flags (integer): combination of Flags for named types
@return: Type or symbol names, depending of ntf_flags. Returns mangled names.
         Never returns anonymous types. To include them, enumerate types by
         ordinals.
```

IDAPython function idaapi.float_flag quick reference

```
float_flag() -> flags64_t
Get a flags64_t representing a float.
```

IDAPython function idaapi.flt_flag quick reference

```
flt_flag() -> flags64_t
see FF_opbits
```

IDAPython function idaapi.flush_buffers quick reference

```
flush_buffers() -> int
Flush buffers to the disk.
```

IDAPython function idaapi.fopenA quick reference

```
fopenA(file) -> FILE *
Open a file for append in text mode, deny none.

@param file (string): char const *
@return: nullptr if failure
```

IDAPython function idaapi.fopenM quick reference

```
fopenM(file) -> FILE *
Open a file for read/write in binary mode, deny write.

@param file (string): char const *
@return: nullptr if failure
```

IDAPython function idaapi.fopenRB quick reference

```
fopenRB(file) -> FILE *
Open a file for read in binary mode, deny none.

@param file (string): char const *
@return: nullptr if failure
```

IDAPython function idaapi.fopenRT quick reference

```
fopenRT(file) -> FILE *
Open a file for read in text mode, deny none.

@param file (string): char const *
@return: nullptr if failure
```

IDAPython function idaapi.fopenWB quick reference

```
fopenWB(file) -> FILE *
Open a new file for write in binary mode, deny read/write. If a file exists, it
will be removed.
```

```
@param file (string): char const *
@return: nullptr if failure
```

IDAPython function idaapi.fopenWT quick reference

```
fopenWT(file) -> FILE *
Open a new file for write in text mode, deny write. If a file exists, it will be
removed.

@param file (string): char const *
@return: nullptr if failure
```

IDAPython function idaapi.for__all__arglocs quick reference

```
for_all_arglocs(vv, vloc, size, off=0) -> int
Compress larger argloc types and initiate the aloc visitor.

@param vv: (C++: aloc_visitor_t &)
@param vloc: (C++: argloc_t &)
@param size (integer):
@param off (integer):
```

IDAPython function idaapi.for__all__const__arglocs quick reference

```
for_all_const_arglocs(vv, vloc, size, off=0) -> int
See for_all_arglocs()

@param vv: (C++: const_aloc_visitor_t &)
@param vloc: (C++: const argloc_t &) argloc_t const &
@param size (integer):
@param off (integer):
```

IDAPython function idaapi.for__all__enum__members quick reference

```
for_all_enum_members(id, cv) -> int
Visit all members of a given enum.

@param id (integer):
@param cv: (C++: enum_member_visitor_t &)
```

IDAPython function idaapi.force_name quick reference

```
force_name(ea, name, flags=0) -> bool

@param ea: ea_t
@param name: char const *
@param flags: int
```

IDAPython function idaapi.forget_problem quick reference

```
forget_problem(type, ea) -> bool
Remove an address from a problem list

@param type: (C++: problist_id_t) problem list type
@param ea (integer): linear address
@return: success
```

IDAPython function idaapi.format_basestring quick reference

```
format_basestring(_in) -> str

@param _in: PyObject *
```

IDAPython function idaapi.formchgcbfa_close quick reference

```
formchgcbfa_close(p_fa, close_normally)

@param p_fa: size_t
@param close_normally: int
```

IDAPython function idaapi.formchgcbfa_enable_field quick reference

```
formchgcbfa_enable_field(p_fa, fid, enable) -> bool

@param p_fa: size_t
@param fid: int
@param enable: bool
```

IDAPython function idaapi.formchgcbfa_get_field_value quick reference

```
formchgcbfa_get_field_value(p_fa, fid, ft, sz) -> PyObject *
```

```
@param p_fa: size_t
@param fid: int
@param ft: int
@param sz: size_t
```

IDAPython function idaapi.formchgcbfa_get_focused_field quick reference

```
formchgcbfa_get_focused_field(p_fa) -> int

@param p_fa: size_t
```

IDAPython function idaapi.formchgcbfa_move_field quick reference

```
formchgcbfa_move_field(p_fa, fid, x, y, w, h) -> bool

@param p_fa: size_t
@param fid: int
@param x: int
@param y: int
@param w: int
@param h: int
```

IDAPython function idaapi.formchgcbfa_refresh_field quick reference

```
formchgcbfa_refresh_field(p_fa, fid)

@param p_fa: size_t
@param fid: int
```

IDAPython function idaapi.formchgcbfa_set_field_value quick reference

```
formchgcbfa_set_field_value(p_fa, fid, ft, py_val) -> bool

@param p_fa: size_t
@param fid: int
@param ft: int
@param py_val: PyObject *
```

IDAPython function idaapi.formchgcbfa_set_focused_field quick reference

```
formchgcbfa_set_focused_field(p_fa, fid) -> bool

@param p_fa: size_t
@param fid: int
```

IDAPython function idaapi.formchgcbfa_show_field quick reference

```
formchgcbfa_show_field(p_fa, fid, show) -> bool

@param p_fa: size_t
@param fid: int
@param show: bool
```

IDAPython function idaapi.frame_off_args quick reference

```
frame_off_args(pfn) -> ea_t
Get starting address of arguments section.

@param pfn: (C++: const func_t *) func_t const *
```

IDAPython function idaapi.frame_off_lvars quick reference

```
frame_off_lvars(pfn) -> ea_t
Get start address of local variables section.

@param pfn: (C++: const func_t *) func_t const *
```

IDAPython function idaapi.frame_off_retaddr quick reference

```
frame_off_retaddr(pfn) -> ea_t
Get starting address of return address section.

@param pfn: (C++: const func_t *) func_t const *
```

IDAPython function idaapi.frame_off_savregs quick reference

```
frame_off_savregs(pfn) -> ea_t
Get starting address of saved registers section.

@param pfn: (C++: const func_t *) func_t const *
```

IDAPython function idaapi.free_chunk quick reference

```
free_chunk(bottom, size, step) -> ea_t

@param bottom: ea_t
@param size: asize_t
@param step: int32
```

IDAPython function idaapi.free_custom_icon quick reference

```
free_custom_icon(icon_id)
Frees an icon loaded with load_custom_icon()

@param icon_id: int
```

IDAPython function idaapi.free_idcv quick reference

```
free_idcv(v)
Free storage used by VT_STR/VT_OBJ IDC variables. After this call the variable
has a numeric value 0

@param v (idaapi.idc_value_t):
```

IDAPython function idaapi.free_regarg quick reference

```
free_regarg(v)

@param v: regarg_t *
```

IDAPython function idaapi.free_regvar quick reference

```
free_regvar(v)

@param v: regvar_t *
```

IDAPython function idaapi.free_til quick reference

```
free_til(ti)
Free memory allocated by til.
```

```
@param ti (idaapi.til_t):
```

IDAPython function idaapi.func_contains quick reference

```
func_contains(pfn, ea) -> bool
Does the given function contain the given address?

@param pfn (idaapi.func_t):
@param ea (integer):
```

IDAPython function idaapi.func_does_return quick reference

```
func_does_return(callee) -> bool
Does the function return?. To calculate the answer, FUNC_NORET flag and
is_noret() are consulted The latter is required for imported functions in the
.idata section. Since in .idata we have only function pointers but not
functions, we have to introduce a special flag for them.

@param callee (integer):
```

IDAPython function idaapi.func_has_stkframe_hole quick reference

```
func_has_stkframe_hole(ea, fti) -> bool
Looks for a hole at the beginning of the stack arguments. Will make use of the
IDB's func_t function at that place (if present) to help determine the presence
of such a hole.

@param ea (integer):
@param fti: (C++: const func_type_data_t &) func_type_data_t const &
```

IDAPython function idaapi.func_parent_iterator_set quick reference

```
func_parent_iterator_set(fpi, pfn) -> bool

@param fpi: func_parent_iterator_t *
@param pfn: func_t *
```

IDAPython function idaapi.func_t___from_ptrval___ quick reference

```
func_t__from_ptrval__(ptrval) -> func_t
```

```
@param ptrval: size_t
```

IDAPython function idaapi.func_tail_iterator_set quick reference

```
func_tail_iterator_set(fti, pfn, ea) -> bool

@param fti: func_tail_iterator_t *
@param pfn: func_t *
@param ea: ea_t
```

IDAPython function idaapi.func_tail_iterator_set_ea quick reference

```
func_tail_iterator_set_ea(fti, ea) -> bool

@param fti: func_tail_iterator_t *
@param ea: ea_t
```

IDAPython function idaapi.gcc_layout quick reference

```
gcc_layout() -> bool
Should use the struct/union layout as done by gcc?
```

IDAPython function idaapi.gen_complex_call_chart quick reference

```
gen_complex_call_chart(filename, wait, title, ea1, ea2, flags, recursion_depth=-1) -> bo
Build and display a complex xref graph.

@param filename (string): output file name. the file extension is not used. maybe
                nullptr.
@param wait (string): message to display during graph building
@param title (string): graph title
@param ea1 (integer): ,ea2: address range
@param flags (integer): combination of Call chart building flags and Flow graph building
              flags. if none of CHART_GEN_DOT, CHART_GEN_GDL, CHART_WINGRAPH is
              specified, the function will return false.
@param flags (integer): combination of Call chart building flags and Flow graph building
              flags. if none of CHART_GEN_DOT, CHART_GEN_GDL, CHART_WINGRAPH is
              specified, the function will return false.
@param recursion_depth: (C++: int32) optional limit of recursion
@return: success. if fails, a warning message is displayed on the screen
```

IDAPython function idaapi.gen_decorate_name quick reference

```
gen_decorate_name(name, mangle, cc, type) -> str
Generic function for decorate_name() (may be used in IDP modules)

@param name (string): char const *
@param mangle (bool):
@param cc: (C++: cm_t)
@param type: (C++: const tinfo_t *) tinfo_t const *
```

IDAPython function idaapi.gen_disasm_text quick reference

```
gen_disasm_text(text, ea1, ea2, truncate_lines)
Generate disassembly text for a range.

@param text: (C++: text_t &) result
@param ea1 (integer): start address
@param ea2 (integer): end address
@param truncate_lines (bool): (on idainfo::margin)
```

IDAPython function idaapi.gen_exe_file quick reference

```
gen_exe_file(fp) -> int
Generate an exe file (unload the database in binary form).

@param fp: (C++: FILE *)
@return: fp the output file handle. if fp == nullptr then return:
* 1: can generate an executable file
* 0: can't generate an executable file
@retval 1: ok
@retval 0: failed
```

IDAPython function idaapi.gen_file quick reference

```
gen_file(otype, fp, ea1, ea2, flags) -> int
Generate an output file.

@param otype: (C++: ofile_type_t) type of output file.
@param fp: (C++: FILE *) the output file handle
@param ea1 (integer): start address. For some file types this argument is ignored
@param ea2 (integer): end address. For some file types this argument is ignored as usual
            in ida, the end address of the range is not included
@param flags (integer): Generate file flagsOFILE_EXE:
@retval 0: can't generate exe file
```

```
@retval 1: ok
@return: number of the generated lines. -1 if an error occurred
```

IDAPython function idaapi.gen_fix_fixups quick reference

```
gen_fix_fixups(_from, to, size)
Relocate the bytes with fixup information once more (generic function). This
function may be called from loader_t::move_segm() if it suits the goal. If
loader_t::move_segm is not defined then this function will be called
automatically when moving segments or rebasing the entire program. Special
parameter values (from = BADADDR, size = 0, to = delta) are used when the
function is called from rebase_program(delta).

@param from (integer):
@param to (integer):
@param size (integer):
```

IDAPython function idaapi.gen_flow_graph quick reference

```
gen_flow_graph(filename, title, pfn, ea1, ea2, gflags) -> bool
Build and display a flow graph.

@param filename (string): output file name. the file extension is not used. maybe
                 nullptr.
@param title (string): graph title
@param pfn (idaapi.func_t): function to graph
@param ea1 (integer): ,ea2: if pfn == nullptr, then the address range
@param gflags (integer): combination of Flow graph building flags. if none of
               CHART_GEN_DOT, CHART_GEN_GDL, CHART_WINGRAPH is specified, the
               function will return false
@param gflags (integer): combination of Flow graph building flags. if none of
               CHART_GEN_DOT, CHART_GEN_GDL, CHART_WINGRAPH is specified, the
               function will return false
@return: success. if fails, a warning message is displayed on the screen
```

IDAPython function idaapi.gen_gdl quick reference

```
gen_gdl(g, fname)
Create GDL file for graph.

@param g: (C++: const gdl_graph_t *) gdl_graph_t const *
@param fname (string): char const *
```

IDAPython function idaapi.gen_idb_event quick reference

```
gen_idb_event(code)
the kernel will use this function to generate idb_events

@param code: (C++: idb_event::event_code_t) enum idb_event::event_code_t
```

IDAPython function idaapi.gen_microcode quick reference

```
gen_microcode(mbr, hf=None, retlist=None, decomp_flags=0, reqmat=MMAT_GLBOPT3) -> mba_t
Generate microcode of an arbitrary code snippet

@param mbr: (C++: const mba_ranges_t &) snippet ranges
@param hf: (C++: hexrays_failure_t *) extended error information (if failed)
@param retlist: (C++: const mlist_t *) list of registers the snippet returns
@param decomp_flags (integer): bitwise combination of decompile() flags... bits
@param reqmat: (C++: mba_maturity_t) required microcode maturity
@return: pointer to the microcode, nullptr if failed.
```

IDAPython function idaapi.gen_simple_call_chart quick reference

```
gen_simple_call_chart(filename, wait, title, gflags) -> bool
Build and display a simple function call graph.

@param filename (string): output file name. the file extension is not used. maybe
                nullptr.
@param wait (string): message to display during graph building
@param title (string): graph title
@param gflags (integer): combination of CHART_NOLIBFUNCS and Flow graph building flags.
               none of CHART_GEN_DOT, CHART_GEN_GDL, CHART_WINGRAPH is
               specified, the function will return false.
@return: success. if fails, a warning message is displayed on the screen
```

IDAPython function idaapi.gen_use_arg_tinfos quick reference

```
gen_use_arg_tinfos(caller, fti, rargs, set_optype, is_stkarg_load, has_delay_slot)

@param caller: ea_t
@param fti: func_type_data_t *
@param rargs: funcargvec_t *
@param set_optype: set_op_tinfo_t *
@param is_stkarg_load: is_stkarg_load_t *
@param has_delay_slot: has_delay_slot_t *
```

IDAPython function idaapi.gen_use_arg_tinfos2 quick reference

```
gen_use_arg_tinfos2(_this, caller, fti, rargs)
Do not call this function directly, use argtinfo_helper_t.

@param _this: (C++: struct argtinfo_helper_t *) argtinfo_helper_t *
@param caller (integer):
@param fti: (C++: func_type_data_t *)
@param rargs: (C++: funcargvec_t *)
```

IDAPython function idaapi.generate_disasm_line quick reference

```
generate_disasm_line(ea, flags=0) -> str

@param ea: ea_t
@param flags: int
```

IDAPython function idaapi.generate_disassembly quick reference

```
generate_disassembly(ea, max_lines, as_stack, notags) -> (int, [str, ...])
Generate disassembly lines (many lines) and put them into a buffer

@param ea: address to generate disassembly for
@param max_lines: how many lines max to generate
@param as_stack: Display undefined items as 2/4/8 bytes
@param notags: bool
@return:     - None on failure
    - tuple(most_important_line_number, list(lines)) : Returns a tuple containing
      the most important line number and a list of generated lines
```

IDAPython function idaapi.get_16bit quick reference

```
get_16bit(ea) -> uint32
Get 16bits of the program at 'ea'.

@param ea (integer):
@return: 1 byte (getFullByte()) if the current processor has 16-bit byte,
         otherwise return get_word()
```

IDAPython function idaapi.get_32bit quick reference

```
get_32bit(ea) -> uint32
Get not more than 32bits of the program at 'ea'.
```

```
@param ea (integer):
@return: 32 bit value, depending on processor_t::nbits:
* if ( nbits <= 8 ) return get_dword(ea);
* if ( nbits <= 16) return get_wide_word(ea);
* return get_wide_byte(ea);
```

IDAPython function idaapi.get__64bit quick reference

```
get_64bit(ea) -> uint64
Get not more than 64bits of the program at 'ea'.

@param ea (integer):
@return: 64 bit value, depending on processor_t::nbits:
* if ( nbits <= 8 ) return get_qword(ea);
* if ( nbits <= 16) return get_wide_dword(ea);
* return get_wide_byte(ea);
```

IDAPython function idaapi.get__8bit quick reference

```
get_8bit(ea, v, nbit) -> PyObject *

@param ea: ea_t
@param v: uint32
@param nbit: int
```

IDAPython function idaapi.get__abi__name quick reference

```
get_abi_name() -> str
Get ABI name.

@return: length of the name (>=0)
```

IDAPython function idaapi.get__absbase quick reference

```
get_absbase(ea) -> ea_t

@param ea: ea_t
```

IDAPython function idaapi.get__action__checkable quick reference

```
get_action_checkable(name) -> bool
```

```
Get an action's checkability (ui_get_action_attr).

@param name (string): the action name
@return: success
```

IDAPython function idaapi.get_action_checked quick reference

```
get_action_checked(name) -> bool
Get an action's checked state (ui_get_action_attr).

@param name (string): the action name
@return: success
```

IDAPython function idaapi.get_action_icon quick reference

```
get_action_icon(name) -> bool
Get an action's icon (ui_get_action_attr).

@param name (string): the action name
@return: success
```

IDAPython function idaapi.get_action_label quick reference

```
get_action_label(name) -> str
Get an action's label (ui_get_action_attr).

@param name (string): the action name
@return: success
```

IDAPython function idaapi.get_action_shortcut quick reference

```
get_action_shortcut(name) -> str
Get an action's shortcut (ui_get_action_attr).

@param name (string): the action name
@return: success
```

IDAPython function idaapi.get_action_state quick reference

```
get_action_state(name) -> bool
Get an action's state (ui_get_action_attr).
```

```
@param name (string): the action name
@return: success
```

IDAPython function idaapi.get_action_tooltip quick reference

```
get_action_tooltip(name) -> str
Get an action's tooltip (ui_get_action_attr).

@param name (string): the action name
@return: success
```

IDAPython function idaapi.get_action_visibility quick reference

```
get_action_visibility(name) -> bool
Get an action's visibility (ui_get_action_attr).

@param name (string): the action name
@return: success
```

IDAPython function idaapi.get_active_modal_widget quick reference

```
get_active_modal_widget() -> TWidget *
Get the current, active modal TWidget instance. Note that in this context, the
"wait dialog" is not considered: this function will return nullptr even if it is
currently shown.

@return: TWidget * the active modal widget, or nullptr
```

IDAPython function idaapi.get_addon_info quick reference

```
get_addon_info(id, info) -> bool
Get info about a registered addon with a given product code. info->cb must be
valid! NB: all pointers are invalidated by next call to register_addon or
get_addon_info

@param id (string): char const *
@param info: (C++: addon_info_t *)
@return: false if not found
```

IDAPython function idaapi.get_addon_info_idx quick reference

```
get_addon_info_idx(index, info) -> bool
Get info about a registered addon with specific index. info->cb must be valid!
NB: all pointers are invalidated by next call to register_addon or
get_addon_info

@param index (integer):
@param info: (C++: addon_info_t *)
@return: false if index is out of range
```

IDAPython function idaapi.get_aflags quick reference

```
get_aflags(ea) -> aflags_t

@param ea: ea_t
```

IDAPython function idaapi.get_alias_target quick reference

```
get_alias_target(ti, ordinal) -> uint32
Find the final alias destination. If the ordinal has not been aliased, return
the specified ordinal itself If failed, returns 0.

@param ti (idaapi.til_t): til_t const *
@param ordinal (integer):
```

IDAPython function idaapi.get_alignment quick reference

```
get_alignment(ea) -> uint32

@param ea: ea_t
```

IDAPython function idaapi.get_archive_path quick reference

```
get_archive_path() -> str
Get archive file path from which input file was extracted.
```

IDAPython function idaapi.get_arg_addrs quick reference

```
get_arg_addrs(caller) -> PyObject *
Retrieve addresses of argument initialization instructions

@param caller: the address of the call instruction
```

```
@return: list of instruction addresses
```

IDAPython function idaapi.get_array_parameters quick reference

```
get_array_parameters(out, ea) -> ssize_t

@param out: array_parameters_t *
@param ea: ea_t
```

IDAPython function idaapi.get_ash quick reference

```
get_ash() -> asm_t
```

IDAPython function idaapi.get_asm_inc_file quick reference

```
get_asm_inc_file() -> str
Get name of the include file.
```

IDAPython function idaapi.get_auto_display quick reference

```
get_auto_display(auto_display) -> bool
Get structure which holds the autoanalysis indicator contents.

@param auto_display: (C++: auto_display_t *)
```

IDAPython function idaapi.get_auto_state quick reference

```
get_auto_state() -> atype_t
Get current state of autoanalyzer. If auto_state == AU_NONE, IDA is currently
not running the analysis (it could be temporarily interrupted to perform the
user's requests, for example).
```

IDAPython function idaapi.get_base_type quick reference

```
get_base_type(t) -> type_t
Get get basic type bits (TYPE_BASE_MASK)

@param t: (C++: type_t)
```

IDAPython function idaapi.get_basic_file_type quick reference

```
get_basic_file_type(li) -> filetype_t
Get the input file type. This function can recognize libraries and zip files.

@param li: (C++: linput_t *)
```

IDAPython function idaapi.get_bblk_trace_options quick reference

```
get_bblk_trace_options() -> int
Get current basic block tracing options. Also see BT_LOG_INSTS \sq{Type,
Synchronous function, Notification, none (synchronous function)}
```

IDAPython function idaapi.get_best_fit_member quick reference

```
get_best_fit_member(sptr, offset) -> member_t
Get member that is most likely referenced by the specified offset. Useful for
offsets > sizeof(struct).

@param sptr: (C++: const struc_t *) struc_t const *
@param offset (integer):
```

IDAPython function idaapi.get_bmask_cmt quick reference

```
get_bmask_cmt(id, bmask, repeatable) -> str

@param id: enum_t
@param bmask: bmask_t
@param repeatable: bool
```

IDAPython function idaapi.get_bmask_name quick reference

```
get_bmask_name(id, bmask) -> str

@param id: enum_t
@param bmask: bmask_t
```

IDAPython function idaapi.get_bpt quick reference

```
get_bpt(ea, bpt) -> bool
Get the characteristics of a breakpoint. \sq{Type, Synchronous function,
Notification, none (synchronous function)}
```

```
@param ea (integer): any address in the breakpoint range
@param bpt: (C++: bpt_t *) if not nullptr, is filled with the characteristics.
@return: false if no breakpoint exists
```

IDAPython function idaapi.get_bpt_group quick reference

```
get_bpt_group(bptloc) -> str
Retrieve the absolute path to the folder of the bpt based on the bpt_location
find_bpt is called to retrieve the bpt \sq{Type, Synchronous function,
Notification, none (synchronous function)}

@param bptloc (idaapi.bpt_location_t): bptlocation of the bpt
@return: breakpoint correclty moved to the directory
success
```

IDAPython function idaapi.get_bpt_qty quick reference

```
get_bpt_qty() -> int
Get number of breakpoints. \sq{Type, Synchronous function, Notification, none
(synchronous function)}
```

IDAPython function idaapi.get_bpt_tev_ea quick reference

```
get_bpt_tev_ea(n) -> ea_t
Get the address associated to a read, read/write or execution trace event.
\sq{Type, Synchronous function, Notification, none (synchronous function)}

@param n (integer): number of trace event, is in range 0..get_tev_qty()-1. 0 represents
          the latest added trace event.
@return: BADADDR if not a read, read/write or execution trace event.
@note: Usually, a breakpoint is associated with a read, read/write or execution
       trace event. However, the returned address could be any address in the
       range of this breakpoint. If the breakpoint was deleted after the trace
       event, the address no longer corresponds to a valid breakpoint.
```

IDAPython function idaapi.get_bptloc_string quick reference

```
get_bptloc_string(i) -> char const *

@param i: int
```

IDAPython function idaapi.get_byte quick reference

```
get_byte(ea) -> uchar
Get one byte (8-bit) of the program at 'ea'. This function works only for 8bit
byte processors.

@param ea (integer):
```

IDAPython function idaapi.get_bytes quick reference

```
get_bytes(ea, size, gmb_flags=0x01) -> bytes or None
Get the specified number of bytes of the program.

@param ea: program address
@param size: number of bytes to return
@param gmb_flags: int
@return: the bytes (as a str), or None in case of failure
```

IDAPython function idaapi.get_bytes_and_mask quick reference

```
get_bytes_and_mask(ea, size, gmb_flags=0x01) -> PyObject *
Get the specified number of bytes of the program, and a bitmask
specifying what bytes are defined and what bytes are not.

@param ea: program address
@param size: number of bytes to return
@param gmb_flags: int
@return: a tuple (bytes, mask), or None in case of failure.
         Both 'bytes' and 'mask' are 'str' instances.
```

IDAPython function idaapi.get_c_header_path quick reference

```
get_c_header_path() -> str
Get the include directory path of the target compiler.
```

IDAPython function idaapi.get_c_macros quick reference

```
get_c_macros() -> str
Get predefined macros for the target compiler.
```

IDAPython function idaapi.get_call_tev_callee quick reference

```
get_call_tev_callee(n) -> ea_t
```

```
Get the called function from a function call trace event. \sq{Type, Synchronous
function, Notification, none (synchronous function)}

@param n (integer): number of trace event, is in range 0..get_tev_qty()-1. 0 represents
          the latest added trace event.
@return: BADADDR if not a function call event.
```

IDAPython function idaapi.get_chooser_data quick reference

```
get_chooser_data(chooser_caption, n) -> [str, ...]
Get the text corresponding to the index N in the chooser data. Use -1 to get the
header.

@param chooser_caption (string): char const *
@param n (integer):
```

IDAPython function idaapi.get_chooser_obj quick reference

```
get_chooser_obj(chooser_caption) -> void *
Get the underlying object of the specified chooser (ui_get_chooser_obj).

This attemps to find the choser by its title and, if found, returns the result
of calling its chooser_base_t::get_chooser_obj() method.

@note: This is object is chooser-specific.

@param chooser_caption (string): char const *
@return: the object that was used to create the chooser
```

IDAPython function idaapi.get_cmt quick reference

```
get_cmt(ea, rptble) -> str
Get an indented comment.

@param ea (integer): linear address. may point to tail byte, the function will find sta
           of the item
@param rptble (bool): get repeatable comment?
@return: size of comment or -1
```

IDAPython function idaapi.get_colored_demangled_name quick reference

```
get_colored_demangled_name(ea, inhibitor, demform, gtn_flags=0) -> qstring
```

```
@param ea: ea_t
@param inhibitor: int32
@param demform: int
@param gtn_flags: int
```

IDAPython function idaapi.get_colored_long_name quick reference

```
get_colored_long_name(ea, gtn_flags=0) -> qstring

@param ea: ea_t
@param gtn_flags: int
```

IDAPython function idaapi.get_colored_name quick reference

```
get_colored_name(ea) -> qstring

@param ea: ea_t
```

IDAPython function idaapi.get_colored_short_name quick reference

```
get_colored_short_name(ea, gtn_flags=0) -> qstring

@param ea: ea_t
@param gtn_flags: int
```

IDAPython function idaapi.get_comp quick reference

```
get_comp(comp) -> comp_t
Get compiler bits.

@param comp: (C++: comp_t)
```

IDAPython function idaapi.get_compiler_abbr quick reference

```
get_compiler_abbr(id) -> char const *
Get abbreviated compiler name.

@param id: (C++: comp_t)
```

IDAPython function idaapi.get__compiler__name quick reference

```
get_compiler_name(id) -> char const *
Get full compiler name.

@param id: (C++: comp_t)
```

IDAPython function idaapi.get__compilers quick reference

```
get_compilers(ids, names, abbrs)
Get names of all built-in compilers.

@param ids: (C++: compvec_t *)
@param names: (C++: qstrvec_t *)
@param abbrs: (C++: qstrvec_t *)
```

IDAPython function idaapi.get__config__value quick reference

```
get_config_value(key) -> bool

@param key: char const *
```

IDAPython function idaapi.get__cp__validity quick reference

```
get_cp_validity(kind, cp, endcp=wchar32_t(-1)) -> bool
Is the given codepoint (or range) acceptable in the given context? If 'endcp' is
not BADCP, it is considered to be the end of the range: [cp, endcp), and is not
included in the range

@param kind: (C++: ucdr_kind_t) enum ucdr_kind_t
@param cp: (C++: wchar32_t)
@param endcp: (C++: wchar32_t)
```

IDAPython function idaapi.get__ctype__name quick reference

```
get_ctype_name(op) -> char const *

@param op: enum ctype_t
```

IDAPython function idaapi.get__curline quick reference

```
get_curline() -> char const *
Get current line from the disassemble window (ui_get_curline).
```

```
@return: cptr current line with the color codes (use tag_remove() to remove the
         color codes)
```

IDAPython function idaapi.get_current_idasgn quick reference

```
get_current_idasgn() -> int
Get number of the the current signature.

@return: 0..n-1
```

IDAPython function idaapi.get_current_operand quick reference

```
get_current_operand(out) -> bool
Get the instruction operand under the cursor. This function determines the
operand that is under the cursor in the active disassembly listing. If the
operand refers to a register or stack variable, it returns true.

@param out: (C++: gco_info_t *) [out]: output buffer
```

IDAPython function idaapi.get_current_source_file quick reference

```
get_current_source_file() -> str
```

IDAPython function idaapi.get_current_source_line quick reference

```
get_current_source_line() -> int
```

IDAPython function idaapi.get_current_thread quick reference

```
get_current_thread() -> thid_t
Get current thread ID. \sq{Type, Synchronous function, Notification, none
(synchronous function)}
```

IDAPython function idaapi.get_current_viewer quick reference

```
get_current_viewer() -> TWidget *
Get current ida viewer (idaview or custom viewer) (ui_get_current_viewer)
```

IDAPython function idaapi.get_current_widget quick reference

```
get_current_widget() -> TWidget *
Get a pointer to the current widget (ui_get_current_widget).
```

IDAPython function idaapi.get_cursor quick reference

```
get_cursor() -> bool
Get the cursor position on the screen (ui_get_cursor).
@note: coordinates are 0-based

@retval true: pointers are filled
@retval false: no disassembly window open
```

IDAPython function idaapi.get_custom_data_format quick reference

```
get_custom_data_format(dfid) -> data_format_t
Get definition of a registered custom data format.

@param dfid (integer): data format id
@return: data format definition or nullptr
```

IDAPython function idaapi.get_custom_data_formats quick reference

```
get_custom_data_formats(out, dtid) -> int
Get list of attached custom data formats for the specified data type.

@param out: (C++: intvec_t *) buffer for the output. may be nullptr
@param dtid (integer): data type id
@return: number of returned custom data formats. if error, returns -1
```

IDAPython function idaapi.get_custom_data_type quick reference

```
get_custom_data_type(dtid) -> data_type_t
Get definition of a registered custom data type.

@param dtid (integer): data type id
@return: data type definition or nullptr
```

IDAPython function idaapi.get_custom_data_type_ids quick reference

```
get_custom_data_type_ids(cdis, ea) -> int

@param cdis: custom_data_type_ids_t *
@param ea: ea_t
```

IDAPython function idaapi.get_custom_data_types quick reference

```
get_custom_data_types(out, min_size=0, max_size=BADADDR) -> int
Get list of registered custom data type ids.

@param out: (C++: intvec_t *) buffer for the output. may be nullptr
@param min_size (integer): minimum value size
@param max_size (integer): maximum value size
@return: number of custom data types with the specified size limits
```

IDAPython function idaapi.get_custom_refinfo quick reference

```
get_custom_refinfo(crid) -> custom_refinfo_handler_t const *
Get definition of a registered custom refinfo type.

@param crid (integer):
```

IDAPython function idaapi.get_custom_viewer_curline quick reference

```
get_custom_viewer_curline(custom_viewer, mouse) -> char const *
Get current line of custom viewer (ui_get_custom_viewer_curline). The returned
line contains color codes

@param custom_viewer(a Widget SWIG wrapper class): view
@param mouse (bool): mouse position (otherwise cursor position)
@return: pointer to contents of current line
```

IDAPython function idaapi.get_custom_viewer_location quick reference

```
get_custom_viewer_location(out, custom_viewer, mouse=False) -> bool
Get the current location in a custom viewer (ui_get_custom_viewer_location).

@param out: (C++: lochist_entry_t *)
```

```
@param custom_viewer(a Widget SWIG wrapper class):
@param mouse (bool):
```

IDAPython function idaapi.get_custom_viewer_place quick reference

```
get_custom_viewer_place(custom_viewer, mouse) -> place_t
Get current place in a custom viewer (ui_get_curplace).

See also the more complete get_custom_viewer_location()

@param custom_viewer(a Widget SWIG wrapper class): view
@param mouse (bool): mouse position (otherwise cursor position)
```

IDAPython function idaapi.get_custom_viewer_place_xcoord quick reference

```
get_custom_viewer_place_xcoord(custom_viewer, pline, pitem) -> int
Get the X position of the item, in the line

@param custom_viewer(a Widget SWIG wrapper class): the widget
@param pline: (C++: const place_t *) a place corresponding to the line
@param pitem: (C++: const place_t *) a place corresponding to the item
@return: -1 if 'pitem' is not included in the line
-2 if 'pitem' points at the entire line
>= 0 for the X coordinate within the pline, where pitem points
```

IDAPython function idaapi.get_data_elsize quick reference

```
get_data_elsize(ea, F, ti=None) -> asize_t
Get size of data type specified in flags 'F'.

@param ea (integer): linear address of the item
@param F (integer): flags
@param ti: (C++: const opinfo_t *) additional information about the data type. For examp
           current item is a structure instance, then ti->tid is structure id.
           Otherwise is ignored (may be nullptr). If specified as nullptr, will
           be automatically retrieved from the database
@return: * byte : 1
* word : 2
* etc...
```

IDAPython function idaapi.get_data_value quick reference

```
get_data_value(v, ea, size) -> bool
Get the value at of the item at 'ea'. This function works with entities up to
sizeof(ea_t) (bytes, word, etc)

@param v: (C++: uval_t *) pointer to the result. may be nullptr
@param ea (integer): linear address
@param size (integer): size of data to read. If 0, then the item type at 'ea' will be us
@return: success
```

IDAPython function idaapi.get_db_byte quick reference

```
get_db_byte(ea) -> uchar
Get one byte (8-bit) of the program at 'ea' from the database. Works even if the
debugger is active. See also get_dbg_byte() to read the process memory directly.
This function works only for 8bit byte processors.

@param ea (integer):
```

IDAPython function idaapi.get_dbctx_id quick reference

```
get_dbctx_id() -> ssize_t
Get the current database context ID

@return: the database context ID, or -1 if no current database
```

IDAPython function idaapi.get_dbctx_qty quick reference

```
get_dbctx_qty() -> size_t
Get number of database contexts

@return: number of database contexts
```

IDAPython function idaapi.get_dbg quick reference

```
get_dbg() -> debugger_t
```

IDAPython function idaapi.get_dbg_byte quick reference

```
get_dbg_byte(ea) -> bool
Get one byte of the debugged process memory.
```

```
@param ea (integer): linear address
@return: true success
false address inaccessible or debugger not running
```

IDAPython function idaapi.get_dbg_memory_info quick reference

```
get_dbg_memory_info(ranges) -> int

@param ranges: meminfo_vec_t *
```

IDAPython function idaapi.get_dbg_reg_info quick reference

```
get_dbg_reg_info(regname, ri) -> bool
Get register information \sq{Type, Synchronous function, Notification, none
(synchronous function)}

@param regname (string): char const *
@param ri: (C++: register_info_t *)
```

IDAPython function idaapi.get_debug_event quick reference

```
get_debug_event() -> debug_event_t
Get the current debugger event.
```

IDAPython function idaapi.get_debug_name quick reference

```
get_debug_name(ea_ptr, how) -> str

@param ea_ptr: ea_t *
@param how: enum debug_name_how_t
```

IDAPython function idaapi.get_debug_name_ea quick reference

```
get_debug_name_ea(name) -> ea_t

@param name: char const *
```

IDAPython function idaapi.get_debug_names quick reference

```
get_debug_names(names, ea1, ea2)
```

```
@param names: ea_name_vec_t *
@param ea1: ea_t
@param ea2: ea_t

get_debug_names(ea1, ea2, return_list=False) -> dict or None

@param ea1: ea_t
@param ea2: ea_t
@param return_list: bool
```

IDAPython function idaapi.get_debugger_event_cond quick reference

```
get_debugger_event_cond() -> char const *
```

IDAPython function idaapi.get_default_encoding_idx quick reference

```
get_default_encoding_idx(bpu) -> int
Get default encoding index for a specific string type.

@param bpu (integer): the amount of bytes per unit (e.g., 1 for ASCII, CP1252, UTF-8...,
            for UTF-16, 4 for UTF-32)
@retval 0: bad BPU argument
```

IDAPython function idaapi.get_default_radix quick reference

```
get_default_radix() -> int
Get default base of number for the current processor.

@return: 2, 8, 10, 16
```

IDAPython function idaapi.get_default_reftype quick reference

```
get_default_reftype(ea) -> reftype_t
Get default reference type depending on the segment.

@param ea (integer):
@return: one of REF_OFF8, REF_OFF16, REF_OFF32, REF_OFF64
```

IDAPython function idaapi.get_defsr quick reference

```
get_defsr(s, reg) -> sel_t
Deprecated, use instead:
    value = s.defsr[reg]

@param s: segment_t *
@param reg: int
```

IDAPython function idaapi.get_demangled_name quick reference

```
get_demangled_name(ea, inhibitor, demform, gtn_flags=0) -> qstring

@param ea: ea_t
@param inhibitor: int32
@param demform: int
@param gtn_flags: int
```

IDAPython function idaapi.get_dtype_by_size quick reference

```
get_dtype_by_size(size) -> int
Get op_t::dtype from size.

@param size (integer):
```

IDAPython function idaapi.get_dtype_flag quick reference

```
get_dtype_flag(dtype) -> flags64_t
Get flags for op_t::dtype field.

@param dtype: (C++: op_dtype_t)
```

IDAPython function idaapi.get_dtype_size quick reference

```
get_dtype_size(dtype) -> size_t
Get size of opt_::dtype field.

@param dtype: (C++: op_dtype_t)
```

IDAPython function idaapi.get_dword quick reference

```
get_dword(ea) -> uint32
Get one dword (32-bit) of the program at 'ea'. This function takes into account
```

```
order of bytes specified in idainfo::is_be() This function works only for 8bit
byte processors.

@param ea (integer):
```

IDAPython function idaapi.get_ea_name quick reference

```
get_ea_name(ea, gtn_flags=0) -> qstring
Get name at the specified address.

@param ea (integer): linear address
@param gtn_flags (integer): how exactly the name should be retrieved. combination of bit
                  for get_ea_name() function. There is a convenience bits
@return: success
```

IDAPython function idaapi.get_ea_viewer_history_info quick reference

```
get_ea_viewer_history_info(nback, nfwd, v) -> bool
Get information about what's in the history (ui_ea_viewer_history_info).

@param nback: (C++: int *) number of available back steps
@param nfwd: (C++: int *) number of available forward steps
@param v(a Widget SWIG wrapper class): ea viewer
@retval false: if the given ea viewer does not exist
@retval true: otherwise
```

IDAPython function idaapi.get_effective_spd quick reference

```
get_effective_spd(pfn, ea) -> sval_t
Get effective difference between the initial and current values of ESP. This
function returns the sp-diff used by the instruction. The difference between
get_spd() and get_effective_spd() is present only for instructions like "pop
[esp+N]": they modify sp and use the modified value.

@param pfn (idaapi.func_t): pointer to function. may be nullptr.
@param ea (integer): linear address
@return: 0 or the difference, usually a negative number
```

IDAPython function idaapi.get_elapsed_secs quick reference

```
get_elapsed_secs() -> size_t
Get seconds database stayed open.
```

IDAPython function idaapi.get_elf_debug_file_directory quick reference

```
get_elf_debug_file_directory() -> char const *
Get the value of the ELF_DEBUG_FILE_DIRECTORY configuration directive.
```

IDAPython function idaapi.get_encoding_bpu quick reference

```
get_encoding_bpu(idx) -> int
Get the amount of bytes per unit (e.g., 2 for UTF-16, 4 for UTF-32) for the
encoding with the given index.

@param idx (integer): the encoding index (1-based)
@return: the number of bytes per units (1/2/4); -1 means error
```

IDAPython function idaapi.get_encoding_bpu_by_name quick reference

```
get_encoding_bpu_by_name(encname) -> int
Get the amount of bytes per unit for the given encoding

@param encname (string): the encoding name
@return: the number of bytes per units (1/2/4); -1 means error
```

IDAPython function idaapi.get_encoding_name quick reference

```
get_encoding_name(idx) -> char const *
Get encoding name for specific index (1-based).

@param idx (integer): the encoding index (1-based)
@retval nullptr: if IDX is out of bounds
@retval empty: string if the encoding was deleted
```

IDAPython function idaapi.get_encoding_qty quick reference

```
get_encoding_qty() -> int
Get total number of encodings (counted from 0)
```

IDAPython function idaapi.get_entry quick reference

```
get_entry(ord) -> ea_t
```

```
Get entry point address by its ordinal

@param ord (integer): ordinal number of entry point
@return: address or BADADDR
```

IDAPython function idaapi.get_entry_forwarder quick reference

```
get_entry_forwarder(ord) -> str
Get forwarder name for the entry point by its ordinal.

@param ord (integer): ordinal number of entry point
@return: size of entry forwarder name or -1
```

IDAPython function idaapi.get_entry_name quick reference

```
get_entry_name(ord) -> str
Get name of the entry point by its ordinal.

@param ord (integer): ordinal number of entry point
@return: size of entry name or -1
```

IDAPython function idaapi.get_entry_ordinal quick reference

```
get_entry_ordinal(idx) -> uval_t
Get ordinal number of an entry point.

@param idx (integer): internal number of entry point. Should be in the range
            0..get_entry_qty()-1
@return: ordinal number or 0.
```

IDAPython function idaapi.get_entry_qty quick reference

```
get_entry_qty() -> size_t
Get number of entry points.
```

IDAPython function idaapi.get_enum quick reference

```
get_enum(name) -> enum_t
Get enum by name.

@param name (string): char const *
```

IDAPython function idaapi.get_enum_cmt quick reference

```
get_enum_cmt(id, repeatable) -> str
Get enum comment.

@param id (integer):
@param repeatable (bool):
```

IDAPython function idaapi.get_enum_flag quick reference

```
get_enum_flag(id) -> flags64_t
Get flags determining the representation of the enum. (currently they define the
numeric base: octal, decimal, hex, bin) and signness.

@param id (integer):
```

IDAPython function idaapi.get_enum_id quick reference

```
get_enum_id(ea, n) -> enum_t
Get enum id of 'enum' operand.

@param ea (integer): linear address
@param n (integer): 0..UA_MAXOP-1 operand number, OPND_ALL one of the operands
@return: id of enum or BADNODE
```

IDAPython function idaapi.get_enum_idx quick reference

```
get_enum_idx(id) -> uval_t
Get the index in the list of enums.

@param id (integer):
```

IDAPython function idaapi.get_enum_member quick reference

```
get_enum_member(id, value, serial, mask) -> const_t
Find an enum member by enum, value and bitmask
@note: if serial -1, return a member with any serial

@param id (integer):
@param value (integer):
@param serial (integer):
@param mask (integer):
```

IDAPython function idaapi.get_enum_member_bmask quick reference

```
get_enum_member_bmask(id) -> bmask_t
Get bitmask of an enum member.

@param id (integer):
```

IDAPython function idaapi.get_enum_member_by_name quick reference

```
get_enum_member_by_name(name) -> const_t
Get a reference to an enum member by its name.

@param name (string): char const *
```

IDAPython function idaapi.get_enum_member_cmt quick reference

```
get_enum_member_cmt(id, repeatable) -> str
Get enum member's comment.

@param id (integer):
@param repeatable (bool):
```

IDAPython function idaapi.get_enum_member_enum quick reference

```
get_enum_member_enum(id) -> enum_t
Get the parent enum of an enum member.

@param id (integer):
```

IDAPython function idaapi.get_enum_member_expr quick reference

```
get_enum_member_expr(tif, serial, value) -> str
Return a C expression that can be used to represent an enum member. If the value
does not correspond to any single enum member, this function tries to find a
bitwise combination of enum members that correspond to it. If more than half of
value bits do not match any enum members, it fails.

@param tif (idaapi.tinfo_t): enumeration type
@param serial (integer): which enumeration member to use (0 means the first with the giv
```

```
                value)
@param value (integer): value to search in the enumeration type. only 32-bit number can
             handled yet
@return: success
```

IDAPython function idaapi.get_enum_member_name quick reference

```
get_enum_member_name(id) -> str
Get name of an enum member by const_t.

@param id (integer):
```

IDAPython function idaapi.get_enum_member_serial quick reference

```
get_enum_member_serial(cid) -> uchar
Get serial number of an enum member.

@param cid (integer):
```

IDAPython function idaapi.get_enum_member_value quick reference

```
get_enum_member_value(id) -> uval_t
Get value of an enum member.

@param id (integer):
```

IDAPython function idaapi.get_enum_name quick reference

```
get_enum_name(id) -> str

@param id: enum_t
```

IDAPython function idaapi.get_enum_name2 quick reference

```
get_enum_name2(id, flags=0) -> str
Get name of enum

@param id (integer): enum id
@param flags (integer): Enum name flags
```

IDAPython function idaapi.get_enum_qty quick reference

```
get_enum_qty() -> size_t
Get number of declared enum_t types.
```

IDAPython function idaapi.get_enum_size quick reference

```
get_enum_size(id) -> size_t
Get the number of the members of the enum.

@param id (integer):
```

IDAPython function idaapi.get_enum_type_ordinal quick reference

```
get_enum_type_ordinal(id) -> int32
Get corresponding type ordinal number.

@param id (integer):
```

IDAPython function idaapi.get_enum_width quick reference

```
get_enum_width(id) -> size_t
Get the width of a enum element allowed values: 0 (unspecified),1,2,4,8,16,32,64

@param id (integer):
```

IDAPython function idaapi.get_event_bpt_hea quick reference

```
get_event_bpt_hea(ev) -> ea_t

@param ev: debug_event_t const *
```

IDAPython function idaapi.get_event_exc_code quick reference

```
get_event_exc_code(ev) -> uint

@param ev: debug_event_t const *
```

IDAPython function idaapi.get_event_exc_ea quick reference

```
get_event_exc_ea(ev) -> ea_t

@param ev: debug_event_t const *
```

IDAPython function idaapi.get_event_exc_info quick reference

```
get_event_exc_info(ev) -> str

@param ev: debug_event_t const *
```

IDAPython function idaapi.get_event_info quick reference

```
get_event_info(ev) -> str

@param ev: debug_event_t const *
```

IDAPython function idaapi.get_event_module_base quick reference

```
get_event_module_base(ev) -> ea_t

@param ev: debug_event_t const *
```

IDAPython function idaapi.get_event_module_name quick reference

```
get_event_module_name(ev) -> str

@param ev: debug_event_t const *
```

IDAPython function idaapi.get_event_module_size quick reference

```
get_event_module_size(ev) -> asize_t

@param ev: debug_event_t const *
```

IDAPython function idaapi.get_extra_cmt quick reference

```
get_extra_cmt(ea, what) -> ssize_t

@param ea: ea_t
```

```
@param what: int
```

IDAPython function idaapi.get_fchunk quick reference

```
get_fchunk(ea) -> func_t
Get pointer to function chunk structure by address.

@param ea (integer): any address in a function chunk
@return: ptr to a function chunk or nullptr. This function may return a function
         entry as well as a function tail.
```

IDAPython function idaapi.get_fchunk_num quick reference

```
get_fchunk_num(ea) -> int
Get ordinal number of a function chunk in the global list of function chunks.

@param ea (integer): any address in the function chunk
@return: number of function chunk (0..get_fchunk_qty()-1). -1 means 'no function
         chunk at the specified address'.
```

IDAPython function idaapi.get_fchunk_qty quick reference

```
get_fchunk_qty() -> size_t
Get total number of function chunks in the program.
```

IDAPython function idaapi.get_fchunk_referer quick reference

```
get_fchunk_referer(ea, idx) -> ea_t

@param ea: ea_t
@param idx: size_t
```

IDAPython function idaapi.get_file_type_name quick reference

```
get_file_type_name() -> str
Get name of the current file type. The current file type is kept in
idainfo::filetype.

@return: size of answer, this function always succeeds
```

IDAPython function idaapi.get_fileregion_ea quick reference

```
get_fileregion_ea(offset) -> ea_t
Get linear address which corresponds to the specified input file offset. If
can't be found, return BADADDR

@param offset: (C++: qoff64_t)
```

IDAPython function idaapi.get_fileregion_offset quick reference

```
get_fileregion_offset(ea) -> qoff64_t
Get offset in the input file which corresponds to the given ea. If the specified
ea can't be mapped into the input file offset, return -1.

@param ea (integer):
```

IDAPython function idaapi.get_first_bmask quick reference

```
get_first_bmask(enum_id) -> bmask_t
Get first bitmask in the enum (bitfield)

@param enum_id (integer): id of enum (bitfield)
@return: the smallest bitmask for enum, or DEFMASK
```

IDAPython function idaapi.get_first_cref_from quick reference

```
get_first_cref_from(frm) -> ea_t
Get first instruction referenced from the specified instruction. If the
specified instruction passes execution to the next instruction then the next
instruction is returned. Otherwise the lowest referenced address is returned
(remember that xrefs are kept sorted!).

@param from (integer): linear address of referencing instruction
@return: first referenced address. If the specified instruction doesn't
         reference to other instructions then returns BADADDR.
```

IDAPython function idaapi.get_first_cref_to quick reference

```
get_first_cref_to(to) -> ea_t
Get first instruction referencing to the specified instruction. If the specified
instruction may be executed immediately after its previous instruction then the
previous instruction is returned. Otherwise the lowest referencing address is
returned. (remember that xrefs are kept sorted!).
```

```
@param to (integer): linear address of referenced instruction
@return: linear address of the first referencing instruction or BADADDR.
```

IDAPython function idaapi.get_first_dref_from quick reference

```
get_first_dref_from(frm) -> ea_t
Get first data referenced from the specified address.

@param from (integer): linear address of referencing instruction or data
@return: linear address of first (lowest) data referenced from the specified
         address. Return BADADDR if the specified instruction/data doesn't
         reference to anything.
```

IDAPython function idaapi.get_first_dref_to quick reference

```
get_first_dref_to(to) -> ea_t
Get address of instruction/data referencing to the specified data.

@param to (integer): linear address of referencing instruction or data
@return: BADADDR if nobody refers to the specified data.
```

IDAPython function idaapi.get_first_enum_member quick reference

```
get_first_enum_member(id, bmask=(bmask_t(-1))) -> uval_t

@param id: enum_t
@param bmask: bmask_t
```

IDAPython function idaapi.get_first_fcref_from quick reference

```
get_first_fcref_from(frm) -> ea_t

@param from: ea_t
```

IDAPython function idaapi.get_first_fcref_to quick reference

```
get_first_fcref_to(to) -> ea_t

@param to: ea_t
```

IDAPython function idaapi.get_first_fixup_ea quick reference

```
get_first_fixup_ea() -> ea_t
Get the first address with fixup information

@return: the first address with fixup information, or BADADDR
```

IDAPython function idaapi.get_first_free_extra_cmtidx quick reference

```
get_first_free_extra_cmtidx(ea, start) -> int

@param ea: ea_t
@param start: int
```

IDAPython function idaapi.get_first_hidden_range quick reference

```
get_first_hidden_range() -> hidden_range_t
Get pointer to the first hidden range.

@return: ptr to hidden range or nullptr
```

IDAPython function idaapi.get_first_module quick reference

```
get_first_module(modinfo) -> bool

@param modinfo: modinfo_t *
```

IDAPython function idaapi.get_first_seg quick reference

```
get_first_seg() -> segment_t
Get pointer to the first segment.
```

IDAPython function idaapi.get_first_serial_enum_member quick reference

```
get_first_serial_enum_member(id, value, bmask) -> const_t

@param id: enum_t
@param value: uval_t
@param bmask: bmask_t
```

IDAPython function idaapi.get_first_struc_idx quick reference

```
get_first_struc_idx() -> uval_t
Get index of first structure.

@return: BADADDR if no known structures, 0 otherwise
```

IDAPython function idaapi.get_fixup quick reference

```
get_fixup(fd, source) -> bool
Get fixup information.

@param fd: (C++: fixup_data_t *)
@param source (integer):
```

IDAPython function idaapi.get_fixup_desc quick reference

```
get_fixup_desc(source, fd) -> str
Get FIXUP description comment.

@param source (integer):
@param fd: (C++: const fixup_data_t &) fixup_data_t const &
```

IDAPython function idaapi.get_fixup_handler quick reference

```
get_fixup_handler(type) -> fixup_handler_t const *
Get handler of standard or custom fixup.

@param type: (C++: fixup_type_t)
```

IDAPython function idaapi.get_fixup_value quick reference

```
get_fixup_value(ea, type) -> uval_t
Get the operand value. This function get fixup bytes from data or an instruction
at `ea' and convert them to the operand value (maybe partially). It is opposite
in meaning to the `patch_fixup_value()`. For example, FIXUP_HI8 read a byte at
`ea' and shifts it left by 8 bits, or AArch64's custom fixup BRANCH26 get low 26
bits of the insn at `ea' and shifts it left by 2 bits. This function is mainly
used to get a relocation addend.

@param ea (integer): address to get fixup bytes from, the size of the fixup bytes depend
           on the fixup type.
@see: fixup_handler_t::size
```

```
@param type: (C++: fixup_type_t) fixup type
@retval operand: value
```

IDAPython function idaapi.get_fixups quick reference

```
get_fixups(out, ea, size) -> bool

@param out: fixups_t *
@param ea: ea_t
@param size: asize_t
```

IDAPython function idaapi.get_flags quick reference

```
get_flags(ea) -> flags64_t
get flags with FF_IVL & MS_VAL. It is much slower under remote debugging because
the kernel needs to read the process memory.

@param ea (integer):
```

IDAPython function idaapi.get_flags_by_size quick reference

```
get_flags_by_size(size) -> flags64_t
Get flags from size (in bytes). Supported sizes: 1, 2, 4, 8, 16, 32. For other
sizes returns 0

@param size (integer):
```

IDAPython function idaapi.get_flags_ex quick reference

```
get_flags_ex(ea, how) -> flags64_t
Get flags for the specified address, extended form.

@param ea (integer):
@param how (integer):
```

IDAPython function idaapi.get_float_type quick reference

```
get_float_type(width) -> tinfo_t
Get a type of a floating point value with the specified width

@param width (integer): width of the desired type
@return: type info object
```

IDAPython function idaapi.get_forced_operand quick reference

```
get_forced_operand(ea, n) -> str
Get forced operand.

@param ea (integer): linear address
@param n (integer): 0..UA_MAXOP-1 operand number
@return: size of forced operand or -1
```

IDAPython function idaapi.get_frame quick reference

```
get_frame(pfn) -> struc_t *
Get pointer to function frame.

@param pfn: func_t const *
```

IDAPython function idaapi.get_frame_part quick reference

```
get_frame_part(range, pfn, part)
Get offsets of the frame part in the frame.

@param range: (C++: range_t *) pointer to the output buffer with the frame part
              start/end(exclusive) offsets, can't be nullptr
@param pfn: (C++: const func_t *) pointer to function structure, can't be nullptr
@param part: (C++: frame_part_t) frame part
```

IDAPython function idaapi.get_frame_retsize quick reference

```
get_frame_retsize(pfn) -> int
Get size of function return address.

@param pfn: (C++: const func_t *) pointer to function structure, can't be nullptr
```

IDAPython function idaapi.get_frame_size quick reference

```
get_frame_size(pfn) -> asize_t
Get full size of a function frame. This function takes into account size of
local variables + size of saved registers + size of return address + number of
purged bytes. The purged bytes correspond to the arguments of the functions with
__stdcall and __fastcall calling conventions.
```

```
@param pfn: (C++: const func_t *) pointer to function structure, may be nullptr
@return: size of frame in bytes or zero
```

IDAPython function idaapi.get_full_data_elsize quick reference

```
get_full_data_elsize(ea, F, ti=None) -> asize_t
Get full size of data type specified in flags 'F'. takes into account processors
with wide bytes e.g. returns 2 for a byte element with 16-bit bytes

@param ea (integer):
@param F (integer):
@param ti: (C++: const opinfo_t *) opinfo_t const *
```

IDAPython function idaapi.get_full_flags quick reference

```
get_full_flags(ea) -> flags64_t
Get flags value for address 'ea'.

@param ea (integer):
@return: 0 if address is not present in the program
```

IDAPython function idaapi.get_full_type quick reference

```
get_full_type(t) -> type_t
Get basic type bits + type flags (TYPE_FULL_MASK)

@param t: (C++: type_t)
```

IDAPython function idaapi.get_func quick reference

```
get_func(ea) -> func_t
Get pointer to function structure by address.

@param ea (integer): any address in a function
@return: ptr to a function or nullptr. This function returns a function entry
         chunk.
```

IDAPython function idaapi.get_func_bitness quick reference

```
get_func_bitness(pfn) -> int
Get function bitness (which is equal to the function segment bitness).
pfn==nullptr => returns 0
```

```
@retval 0: 16
@retval 1: 32
@retval 2: 64

@param pfn: (C++: const func_t *) func_t const *
```

IDAPython function idaapi.get_func_bits quick reference

```
get_func_bits(pfn) -> int
Get number of bits in the function addressing.

@param pfn: (C++: const func_t *) func_t const *
```

IDAPython function idaapi.get_func_by_frame quick reference

```
get_func_by_frame(frame_id) -> ea_t
Get function by its frame id.
@warning: this function works only with databases created by IDA > 5.6

@param frame_id (integer): id of the function frame
@return: start address of the function or BADADDR
```

IDAPython function idaapi.get_func_bytes quick reference

```
get_func_bytes(pfn) -> int
Get number of bytes in the function addressing.

@param pfn: (C++: const func_t *) func_t const *
```

IDAPython function idaapi.get_func_chunknum quick reference

```
get_func_chunknum(pfn, ea) -> int
Get the containing tail chunk of 'ea'.
@retval -1: means 'does not contain ea'
@retval 0: means the 'pfn' itself contains ea
@retval >0: the number of the containing function tail chunk

@param pfn (idaapi.func_t):
@param ea (integer):
```

IDAPython function idaapi.get_func_cmt quick reference

```
get_func_cmt(pfn, repeatable) -> str
Get function comment.

@param pfn: (C++: const func_t *) ptr to function structure
@param repeatable (bool): get repeatable comment?
@return: size of comment or -1 In fact this function works with function chunks
         too.
```

IDAPython function idaapi.get_func_name quick reference

```
get_func_name(ea) -> str
Get function name.

@param ea (integer): any address in the function
@return: length of the function name
```

IDAPython function idaapi.get_func_num quick reference

```
get_func_num(ea) -> int
Get ordinal number of a function.

@param ea (integer): any address in the function
@return: number of function (0..get_func_qty()-1). -1 means 'no function at the
         specified address'.
```

IDAPython function idaapi.get_func_qty quick reference

```
get_func_qty() -> size_t
Get total number of functions in the program.
```

IDAPython function idaapi.get_func_ranges quick reference

```
get_func_ranges(ranges, pfn) -> ea_t
Get function ranges.

@param ranges: (C++: rangeset_t *) buffer to receive the range info
@param pfn (idaapi.func_t): ptr to function structure
@return: end address of the last function range (BADADDR-error)
```

IDAPython function idaapi.get_func_trace_options quick reference

```
get_func_trace_options() -> int
Get current function tracing options. Also see FT_LOG_RET \sq{Type, Synchronous
function, Notification, none (synchronous function)}
```

IDAPython function idaapi.get_global_var quick reference

```
get_global_var(prov, ea, name, out) -> bool

@param prov: srcinfo_provider_t *
@param ea: ea_t
@param name: char const *
@param out: source_item_ptr *
```

IDAPython function idaapi.get_gotea quick reference

```
get_gotea() -> ea_t
```

IDAPython function idaapi.get_graph_viewer quick reference

```
get_graph_viewer(parent) -> graph_viewer_t *
Get custom graph viewer for given form.

@param parent(a Widget SWIG wrapper class):
```

IDAPython function idaapi.get_group_selector quick reference

```
get_group_selector(grpsel) -> sel_t
Get common selector for a group of segments.

@param grpsel: (C++: sel_t) selector of group segment
@return: common selector of the group or 'grpsel' if no such group is found
```

IDAPython function idaapi.get_grp_bpts quick reference

```
get_grp_bpts(bpts, grp_name) -> ssize_t
Retrieve a copy the bpts stored in a folder \sq{Type, Synchronous function,
Notification, none (synchronous function)}

@param bpts: (C++: bpt_vec_t *) : pointer to a vector where the copy of bpts are stored
@param grp_name (string): absolute path to the folder
@return: number of bpts present in the vector
```

IDAPython function idaapi.get_hexdump_ea quick reference

```
get_hexdump_ea(hexdump_num) -> ea_t
Get the current address in a hex view.

@param hexdump_num (integer): number of hexview window
```

IDAPython function idaapi.get_hexrays_version quick reference

```
get_hexrays_version() -> char const *
Get decompiler version. The returned string is of the form
<major>.<minor>.<revision>.<build-date>

@return: pointer to version string. For example: "2.0.0.140605"
```

IDAPython function idaapi.get_hidden_range quick reference

```
get_hidden_range(ea) -> hidden_range_t
Get pointer to hidden range structure, in: linear address.

@param ea (integer): any address in the hidden range
```

IDAPython function idaapi.get_hidden_range_num quick reference

```
get_hidden_range_num(ea) -> int
Get number of a hidden range.

@param ea (integer): any address in the hidden range
@return: number of hidden range (0..get_hidden_range_qty()-1)
```

IDAPython function idaapi.get_hidden_range_qty quick reference

```
get_hidden_range_qty() -> int
Get number of hidden ranges.
```

IDAPython function idaapi.get_highlight quick reference

```
get_highlight(v, in_flags=0) -> (str, int) or None
Returns the currently highlighted identifier and flags
```

```
@param v: The UI widget to operate on
@param flags: Optionally specify a slot (see kernwin.hpp), current otherwise
@return: a tuple (text, flags), or None if nothing
         is highlighted or in case of error.
```

IDAPython function idaapi.get_ida_notepad_text quick reference

```
get_ida_notepad_text() -> str
Get notepad text.
```

IDAPython function idaapi.get_ida_subdirs quick reference

```
get_ida_subdirs(subdir, flags=0) -> int
Get list of directories in which to find a specific IDA resource (see IDA
subdirectories). The order of the resulting list is as follows:
- [$IDAUSR/subdir (0..N entries)]
- $IDADIR/subdir

@param subdir (string): name of the resource to list
@param flags (integer): Subdirectory modification flags bits
@return: number of directories appended to 'dirs'
```

IDAPython function idaapi.get_idainfo64_by_type quick reference

```
get_idainfo64_by_type(out_flags, tif) -> bool
Extract information from a tinfo_t.

@param out_flags: (C++: flags64_t *) description of type using flags64_t
@param tif (idaapi.tinfo_t): the type to inspect
```

IDAPython function idaapi.get_idainfo_by_type quick reference

```
get_idainfo_by_type(tif) -> bool
Extract information from a tinfo_t.

@param tif (idaapi.tinfo_t): the type to inspect
```

IDAPython function idaapi.get_idasgn_desc quick reference

```
get_idasgn_desc(n) -> (str, str)
Get information about a signature in the list.
It returns: (name of signature, names of optional libraries)
```

```
See also: get_idasgn_desc_with_matches

@param n: number of signature in the list (0..get_idasgn_qty()-1)
@return: None on failure or tuple(signame, optlibs)
```

IDAPython function idaapi.get_idasgn_desc_with_matches quick reference

```
get_idasgn_desc_with_matches(n) -> (str, str, int)
Get information about a signature in the list.
It returns: (name of signature, names of optional libraries, number of matches)

@param n: number of signature in the list (0..get_idasgn_qty()-1)
@return: None on failure or tuple(signame, optlibs, nmatches)
```

IDAPython function idaapi.get_idasgn_qty quick reference

```
get_idasgn_qty() -> int
Get number of signatures in the list of planned and applied signatures.

@return: 0..n
```

IDAPython function idaapi.get_idasgn_title quick reference

```
get_idasgn_title(name) -> str
Get full description of the signature by its short name.

@param name (string): short name of a signature
@return: size of signature description or -1
```

IDAPython function idaapi.get_idati quick reference

```
get_idati() -> til_t
Pointer to the local type library - this til is private for each IDB file
Function that accepts til_t* uses local type library instead of nullptr.
```

IDAPython function idaapi.get_idb_ctime quick reference

```
get_idb_ctime() -> time_t
Get database creation timestamp.
```

IDAPython function idaapi.get_idb_nopens quick reference

```
get_idb_nopens() -> size_t
Get number of times the database is opened.
```

IDAPython function idaapi.get_idb_notifier_addr quick reference

```
get_idb_notifier_addr(arg1) -> PyObject *

@param arg1: PyObject *
```

IDAPython function idaapi.get_idb_notifier_ud_addr quick reference

```
get_idb_notifier_ud_addr(hooks) -> PyObject *

@param hooks: IDB_Hooks *
```

IDAPython function idaapi.get_idc_filename quick reference

```
get_idc_filename(file) -> str
Get full name of IDC file name. Search for file in list of include directories,
IDCPATH directory and system directories.

@param file (string): file name without full path
@return: nullptr is file not found. otherwise returns pointer to buf
```

IDAPython function idaapi.get_idcv_attr quick reference

```
get_idcv_attr(res, obj, attr, may_use_getattr=False) -> error_t
Get an object attribute.

@param res (idaapi.idc_value_t): buffer for the attribute value
@param obj: (C++: const idc_value_t *) variable that holds an object reference. if obj i
            searches global variables, then user functions
@param attr (string): attribute name
@param may_use_getattr (bool): may call getattr functions to calculate the attribute if
                          it does not exist
@return: error code, eOk on success
```

IDAPython function idaapi.get_idcv_class_name quick reference

```
get_idcv_class_name(obj) -> str
```

```
Retrieves the IDC object class name.

@param obj: (C++: const idc_value_t *) class instance variable
@return: error code, eOk on success
```

IDAPython function idaapi.get_idcv_slice quick reference

```
get_idcv_slice(res, v, i1, i2, flags=0) -> error_t
Get slice.

@param res (idaapi.idc_value_t): output variable that will contain the slice
@param v: (C++: const idc_value_t *) input variable (string or object)
@param i1 (integer): slice start index
@param i2 (integer): slice end index (excluded)
@param flags (integer): IDC variable slice flags or 0
@return: eOk if success
```

IDAPython function idaapi.get_idp_name quick reference

```
get_idp_name() -> str
Get name of the current processor module. The name is derived from the file
name. For example, for IBM PC the module is named "pc.w32" (windows version),
then the module name is "PC" (uppercase). If no processor module is loaded, this
function will return nullptr
```

IDAPython function idaapi.get_idp_notifier_addr quick reference

```
get_idp_notifier_addr(arg1) -> PyObject *

@param arg1: PyObject *
```

IDAPython function idaapi.get_idp_notifier_ud_addr quick reference

```
get_idp_notifier_ud_addr(hooks) -> PyObject *

@param hooks: IDP_Hooks *
```

IDAPython function idaapi.get_ids_modnode quick reference

```
get_ids_modnode() -> netnode
Get ids modnode.
```

IDAPython function idaapi.get_imagebase quick reference

```
get_imagebase() -> ea_t
Get image base address.
```

IDAPython function idaapi.get_immvals quick reference

```
get_immvals(ea, n, F=0) -> [int, ...]
Get immediate values at the specified address. This function decodes instruction
at the specified address or inspects the data item. It finds immediate values
and copies them to 'out'. This function will store the original value of the
operands in 'out', unless the last bits of 'F' are "...0 11111111", in which
case the transformed values (as needed for printing) will be stored instead.

@param ea (integer): address to analyze
@param n (integer): 0..UA_MAXOP-1 operand number, OPND_ALL all the operands
@param F (integer): flags for the specified address
@return: number of immediate values (0..2*UA_MAXOP)
```

IDAPython function idaapi.get_import_module_name quick reference

```
get_import_module_name(mod_index) -> str
Returns the name of an imported module given its index

@param mod_index: int
@return: None or the module name
```

IDAPython function idaapi.get_import_module_qty quick reference

```
get_import_module_qty() -> uint
Get number of import modules.
```

IDAPython function idaapi.get_ind_purged quick reference

```
get_ind_purged(ea) -> ea_t

@param ea: ea_t
```

IDAPython function idaapi.get_inf_structure quick reference

```
get_inf_structure() -> idainfo
Returns the global variable 'inf' (an instance of idainfo structure, see ida.hpp)
```

IDAPython function idaapi.get_initial_ida_version quick reference

```
get_initial_ida_version() -> str
Get version of ida which created the database (string format like "7.5")
```

IDAPython function idaapi.get_initial_idb_version quick reference

```
get_initial_idb_version() -> ushort
Get initial version of the database (numeric format like 700)
```

IDAPython function idaapi.get_initial_version quick reference

```
get_initial_idb_version() -> ushort
Get initial version of the database (numeric format like 700)
```

IDAPython function idaapi.get_innermost_member quick reference

```
get_innermost_member(sptr, offset) -> (member_t, struc_t, int)
Get the innermost member at the given offset

@param sptr: the starting structure
@param offset: offset into the starting structure
@return:     - None on failure
    - tuple(member_t, struct_t, offset)
      where member_t: a member in SPTR (it is not a structure),
            struct_t: the innermost structure,
            offset:   remaining offset into the returned member
```

IDAPython function idaapi.get_input_file_path quick reference

```
get_input_file_path() -> str
Get full path of the input file.
```

IDAPython function idaapi.get_insn_tev_reg_mem quick reference

```
get_insn_tev_reg_mem(n, memmap) -> bool
Read the memory pointed by register values from an instruction trace event.
```

```
\sq{Type, Synchronous function, Notification, none (synchronous function)}

@param n (integer): number of trace event, is in range 0..get_tev_qty()-1. 0 represents
          the latest added trace event.
@param memmap: (C++: memreg_infos_t *) result
@return: false if not an instruction event or no memory is available
```

IDAPython function idaapi.get_insn_tev_reg_result quick reference

```
get_insn_tev_reg_result(n, regname, regval) -> bool

@param n: int
@param regname: char const *
@param regval: regval_t *
```

IDAPython function idaapi.get_insn_tev_reg_val quick reference

```
get_insn_tev_reg_val(n, regname, regval) -> bool

@param n: int
@param regname: char const *
@param regval: regval_t *
```

IDAPython function idaapi.get_insn_trace_options quick reference

```
get_insn_trace_options() -> int
Get current instruction tracing options. Also see IT_LOG_SAME_IP \sq{Type,
Synchronous function, Notification, none (synchronous function)}
```

IDAPython function idaapi.get_int_type_by_width_and_sign quick reference

```
get_int_type_by_width_and_sign(srcwidth, sign) -> tinfo_t
Create a type info by width and sign. Returns a simple type (examples: int,
short) with the given width and sign.

@param srcwidth (integer): size of the type in bytes
@param sign: (C++: type_sign_t) sign of the type
```

IDAPython function idaapi.get_ip_val quick reference

```
get_ip_val() -> bool
Get value of the IP (program counter) register for the current thread. Requires
a suspended debugger.
```

IDAPython function idaapi.get_item_color quick reference

```
get_item_color(ea) -> bgcolor_t

@param ea: ea_t
```

IDAPython function idaapi.get_item_end quick reference

```
get_item_end(ea) -> ea_t
Get the end address of the item at 'ea'. The returned address doesn't belong to
the current item. Unexplored bytes are counted as 1 byte entities.

@param ea (integer):
```

IDAPython function idaapi.get_item_flag quick reference

```
get_item_flag(_from, n, ea, appzero) -> flags64_t
Get flag of the item at 'ea' even if it is a tail byte of some array or
structure. This function is used to get flags of structure members or array
elements.

@param from (integer): linear address of the instruction which refers to 'ea'
@param n (integer): operand number which refers to 'ea' or OPND_ALL for one of the
          operands
@param ea (integer): the referenced address
@param appzero (bool): append a struct field name if the field offset is zero?
                meaningful only if the name refers to a structure.
@return: flags or 0 (if failed)
```

IDAPython function idaapi.get_item_head quick reference

```
get_item_head(ea) -> ea_t
Get the start address of the item at 'ea'. If there is no current item, then
'ea' will be returned (see definition at the end of bytes.hpp source)

@param ea (integer):
```

IDAPython function idaapi.get_item_refinfo quick reference

```
get_item_refinfo(ri, ea, n) -> bool
Get refinfo of the item at 'ea'. This function works for a regular offset
operand as well as for a tail byte of a structure variable (in this case refinfo
to corresponding structure member will be returned)

@param ri: (C++: refinfo_t *) refinfo holder
@param ea (integer): the item address
@param n (integer): operand number which refers to 'ea' or OPND_ALL for one of the
          operands
@return: success
```

IDAPython function idaapi.get_item_size quick reference

```
get_item_size(ea) -> asize_t
Get size of item (instruction/data) in bytes. Unexplored bytes have length of 1
byte. This function returns 0 only for BADADDR.

@param ea (integer):
```

IDAPython function idaapi.get_kernel_version quick reference

```
get_kernel_version() -> str
Get IDA kernel version (in a string like "5.1").
```

IDAPython function idaapi.get_key_code quick reference

```
get_key_code(keyname) -> ushort
Get keyboard key code by its name (ui_get_key_code)

@param keyname (string): char const *
```

IDAPython function idaapi.get_last_bmask quick reference

```
get_last_bmask(enum_id) -> bmask_t
Get last bitmask in the enum (bitfield)

@param enum_id (integer): id of enum
@return: the biggest bitmask for enum, or DEFMASK
```

IDAPython function idaapi.get_last_enum_member quick reference

```
get_last_enum_member(id, bmask=(bmask_t(-1))) -> uval_t

@param id: enum_t
@param bmask: bmask_t
```

IDAPython function idaapi.get_last_hidden_range quick reference

```
get_last_hidden_range() -> hidden_range_t
Get pointer to the last hidden range.

@return: ptr to hidden range or nullptr
```

IDAPython function idaapi.get_last_seg quick reference

```
get_last_seg() -> segment_t
Get pointer to the last segment.
```

IDAPython function idaapi.get_last_serial_enum_member quick reference

```
get_last_serial_enum_member(id, value, bmask) -> const_t

@param id: enum_t
@param value: uval_t
@param bmask: bmask_t
```

IDAPython function idaapi.get_last_struc_idx quick reference

```
get_last_struc_idx() -> uval_t
Get index of last structure.

@return: BADADDR if no known structures, get_struc_qty()-1 otherwise
```

IDAPython function idaapi.get_linput_type quick reference

```
get_linput_type(li) -> linput_type_t
Get linput type.

@param li: (C++: linput_t *)
```

IDAPython function idaapi.get_loader_format_name quick reference

```
get_loader_format_name() -> str
Get file format name for loader modules.
```

IDAPython function idaapi.get_local_var quick reference

```
get_local_var(prov, ea, name, out) -> bool

@param prov: srcinfo_provider_t *
@param ea: ea_t
@param name: char const *
@param out: source_item_ptr *
```

IDAPython function idaapi.get_local_vars quick reference

```
get_local_vars(prov, ea, out) -> bool

@param prov: srcinfo_provider_t *
@param ea: ea_t
@param out: source_items_t *
```

IDAPython function idaapi.get_long_name quick reference

```
get_long_name(ea, gtn_flags=0) -> qstring

@param ea: ea_t
@param gtn_flags: int
```

IDAPython function idaapi.get_lookback quick reference

```
get_lookback() -> int
Number of instructions to look back. This variable is not used by the kernel.
Its value may be specified in ida.cfg: LOOKBACK = <number>. IDP may use it as
you like it. (TMS module uses it)
```

IDAPython function idaapi.get_mangled_name_type quick reference

```
get_mangled_name_type(name) -> mangled_name_type_t

@param name: char const *
```

IDAPython function idaapi.get_manual_insn quick reference

```
get_manual_insn(ea) -> str
Retrieve the user-specified string for the manual instruction.

@param ea (integer): linear address of the instruction or data item
@return: size of manual instruction or -1
```

IDAPython function idaapi.get_manual_regions quick reference

```
get_manual_regions(ranges)
Returns the manual memory regions

@param ranges: meminfo_vec_t *

@return: list(start_ea, end_ea, name, sclass, sbase, bitness, perm)
get_manual_regions() -> [(int, int, str, str, int, int, int), ...] or None
```

IDAPython function idaapi.get_mapping quick reference

```
get_mapping(n) -> bool
Get memory mapping range by its number.

@param n (integer): number of mapping range (0..get_mappings_qty()-1)
@return: false if the specified range doesn't exist, otherwise returns `from',
         `to', `size'
```

IDAPython function idaapi.get_mappings_qty quick reference

```
get_mappings_qty() -> size_t
Get number of mappings.
```

IDAPython function idaapi.get_mark_comment quick reference

```
get_mark_comment(slot) -> PyObject *

@param slot: int32
```

IDAPython function idaapi.get_marked_pos quick reference

```
get_marked_pos(slot) -> ea_t
```

```
@param slot: int32
```

IDAPython function idaapi.get_max_offset quick reference

```
get_max_offset(sptr) -> ea_t
For unions: returns number of members, for structs: returns size of structure.

@param sptr: (C++: struc_t *)
```

IDAPython function idaapi.get_max_strlit_length quick reference

```
get_max_strlit_length(ea, strtype, options=0) -> size_t
Determine maximum length of string literal.

If the string literal has a length prefix (e.g., STRTYPE_LEN2 has a two-byte
length prefix), the length of that prefix (i.e., 2) will be part of the returned
value.

@param ea (integer): starting address
@param strtype: (C++: int32) string type. one of String type codes
@param options (integer): combination of string literal length options
@return: length of the string in octets (octet==8bit)
```

IDAPython function idaapi.get_member quick reference

```
get_member(sptr, offset) -> member_t
Get member at given offset.

@param sptr: (C++: const struc_t *) struc_t const *
@param offset (integer):
```

IDAPython function idaapi.get_member_by_fullname quick reference

```
get_member_by_fullname(fullname) -> member_t
Get a member by its fully qualified name, "struct.field".

@param fullname (string): char const *
```

IDAPython function idaapi.get_member_by_id quick reference

```
get_member_by_id(mid) -> member_t
Check if the specified member id points to a struct member. convenience
function.

@param mid (integer):
```

IDAPython function idaapi.get_member_by_name quick reference

```
get_member_by_name(sptr, membername) -> member_t
Get a member by its name, like "field44".

@param sptr: (C++: const struc_t *) struc_t const *
@param membername (string): char const *
```

IDAPython function idaapi.get_member_cmt quick reference

```
get_member_cmt(mid, repeatable) -> str
Get comment of structure member.

@param mid (integer):
@param repeatable (bool):
```

IDAPython function idaapi.get_member_fullname quick reference

```
get_member_fullname(mid) -> str
Get a member's fully qualified name, "struct.field".

@param mid (integer):
```

IDAPython function idaapi.get_member_id quick reference

```
get_member_id(sptr, offset) -> tid_t
Get member id at given offset.

@param sptr: (C++: const struc_t *) struc_t const *
@param offset (integer):
```

IDAPython function idaapi.get_member_name quick reference

```
get_member_name(mid) -> str
```

```
@param mid: tid_t
```

IDAPython function idaapi.get_member_size quick reference

```
get_member_size(NONNULL_mptr) -> asize_t
Get size of structure member. May return 0 for the last member of varstruct. For
union members, returns member_t::eoff.

@param NONNULL_mptr: (C++: const member_t *) member_t const *
```

IDAPython function idaapi.get_member_struc quick reference

```
get_member_struc(fullname) -> struc_t
Get containing structure of member by its full name "struct.field".

@param fullname (string): char const *
```

IDAPython function idaapi.get_member_tinfo quick reference

```
get_member_tinfo(tif, mptr) -> bool
Get tinfo for given member.

@param tif: (C++: tinfo_t *)
@param mptr: (C++: const member_t *) member_t const *
```

IDAPython function idaapi.get_member_type quick reference

```
get_member_type(mptr, type) -> bool
Get type of a structure field. This function performs validity checks of the
field type. Wrong types are rejected.

@param mptr: (C++: const member_t *) structure field
@param type: (C++: tinfo_t *) pointer to the variable where the type is returned. This p
             can be nullptr.
@return: false if failed
```

IDAPython function idaapi.get_merror_desc quick reference

```
get_merror_desc(code, mba) -> str
Get textual description of an error code

@param code: (C++: merror_t) Microcode error codes
```

```
@param mba: (C++: mba_t *) the microcode array
@return: the error address
```

IDAPython function idaapi.get_min_spd_ea quick reference

```
get_min_spd_ea(pfn) -> ea_t

@param pfn: func_t *
```

IDAPython function idaapi.get_module_info quick reference

```
get_module_info(ea, modinfo) -> bool

@param ea: ea_t
@param modinfo: modinfo_t *
```

IDAPython function idaapi.get_mreg_name quick reference

```
get_mreg_name(reg, width, ud=None) -> str
Get the microregister name.

@param reg: (C++: mreg_t) microregister number
@param width (integer): size of microregister in bytes. may be bigger than the real
             register size.
@param ud : reserved, must be nullptr
@return: width of the printed register. this value may be less than the WIDTH
         argument.
```

IDAPython function idaapi.get_name quick reference

```
get_name(ea) -> qstring

@param ea: ea_t
```

IDAPython function idaapi.get_name_base_ea quick reference

```
get_name_base_ea(_from, to) -> ea_t
Get address of the name used in the expression for the address

@param from (integer): address of the operand which references to the address
@param to (integer): the referenced address
@return: address of the name used to represent the operand
```

IDAPython function idaapi.get_name_color quick reference

```
get_name_color(_from, ea) -> color_t
Calculate flags for get_ea_name() function.

Get name color.

@param from (integer): linear address where the name is used. if not applicable, then
             should be BADADDR. The kernel returns a local name color if the
             reference is within a function, i.e. 'from' and 'ea' belong to the
             same function.
@param ea (integer): linear address
```

IDAPython function idaapi.get_name_ea quick reference

```
get_name_ea(_from, name) -> ea_t
Get the address of a name. This function resolves a name into an address. It can
handle regular global and local names, as well as debugger names.

@param from (integer): linear address where the name is used. If specified, the local
             labels of the function at the specified address will will be
             checked. BADADDR means that local names won't be consulted.
@param name (string): any name in the program or nullptr
@return: address of the name or BADADDR
```

IDAPython function idaapi.get_name_expr quick reference

```
get_name_expr(_from, n, ea, off, flags=0x0001) -> str
Convert address to name expression (name with a displacement). This function
takes into account fixup information and returns a colored name expression (in
the form <name> +/- <offset>). It also knows about structure members and arrays.
If the specified address doesn't have a name, a dummy name is generated.

@param from (integer): linear address of instruction operand or data referring to the
             name. This address will be used to get fixup information, so it
             should point to exact position of the operand in the instruction.
@param n (integer): number of referencing operand. for data items specify 0
@param ea (integer): address to convert to name expression
@param off (integer): the value of name expression. this parameter is used only to check
            that the name expression will have the wanted value. 'off' may be
            equal to BADADDR but this is discouraged because it prohibits
            checks.
@param flags (integer): Name expression flags
```

```
@return: < 0 if address is not valid, no segment or other failure. otherwise the
         length of the name expression in characters.
```

IDAPython function idaapi.get_name_value quick reference

```
get_name_value(_from, name) -> int
Get value of the name. This function knows about: regular names, enums, special
segments, etc.

@param from (integer): linear address where the name is used if not applicable, then
             should be BADADDR
@param name (string): any name in the program or nullptr
@return: Name value result codes
```

IDAPython function idaapi.get_named_type quick reference

```
get_named_type(til, name, ntf_flags) -> (int, bytes, bytes, NoneType, NoneType, int, int
Get a type data by its name.

@param til: the type library
@param name: the type name
@param ntf_flags: a combination of NTF_* constants
@return:     None on failure
    tuple(code, type_str, fields_str, cmt, field_cmts, sclass, value) on success
```

IDAPython function idaapi.get_named_type64 quick reference

```
get_named_type64(til, name, ntf_flags) -> (int, bytes, NoneType, NoneType, NoneType, int
See get_named_type() above.
@note: If the value in the 'ti' library is 32-bit, it will be sign-extended
       before being stored in the 'value' pointer.

@param til: til_t const *
@param name (string): char const *
@param ntf_flags (integer):
```

IDAPython function idaapi.get_navband_ea quick reference

```
get_navband_ea(pixel) -> ea_t
Translate the pixel position on the navigation band, into an address.

@param pixel (integer):
```

IDAPython function idaapi.get_navband_pixel quick reference

```
get_navband_pixel(ea) -> int
Maps an address, onto a pixel coordinate within the navigation band

@param ea: The address to map
@return: a list [pixel, is_vertical]
```

IDAPython function idaapi.get_next_bmask quick reference

```
get_next_bmask(enum_id, bmask) -> bmask_t
Get next bitmask in the enum (bitfield)

@param enum_id (integer): id of enum
@param bmask (integer): the current bitmask
@return: value of a bitmask with value higher than the specified value, or
         DEFMASK
```

IDAPython function idaapi.get_next_cref_from quick reference

```
get_next_cref_from(frm, current) -> ea_t
Get next instruction referenced from the specified instruction.

@param from (integer): linear address of referencing instruction
@param current (integer): linear address of current referenced instruction This value is
                returned by get_first_cref_from() or previous call to
                get_next_cref_from() functions.
@return: next referenced address or BADADDR.
```

IDAPython function idaapi.get_next_cref_to quick reference

```
get_next_cref_to(to, current) -> ea_t
Get next instruction referencing to the specified instruction.

@param to (integer): linear address of referenced instruction
@param current (integer): linear address of current referenced instruction This value is
                returned by get_first_cref_to() or previous call to
                get_next_cref_to() functions.
@return: linear address of the next referencing instruction or BADADDR.
```

IDAPython function idaapi.get_next_dref_from quick reference

```
get_next_dref_from(frm, current) -> ea_t
```

```
Get next data referenced from the specified address.

@param from (integer): linear address of referencing instruction or data
@param current (integer): linear address of current referenced data. This value is
                returned by get_first_dref_from() or previous call to
                get_next_dref_from() functions.
@return: linear address of next data or BADADDR.
```

IDAPython function idaapi.get_next_dref_to quick reference

```
get_next_dref_to(to, current) -> ea_t
Get address of instruction/data referencing to the specified data

@param to (integer): linear address of referencing instruction or data
@param current (integer): current linear address. This value is returned by
                get_first_dref_to() or previous call to get_next_dref_to()
                functions.
@return: BADADDR if nobody refers to the specified data.
```

IDAPython function idaapi.get_next_enum_member quick reference

```
get_next_enum_member(id, value, bmask=(bmask_t(-1))) -> uval_t

@param id: enum_t
@param value: uval_t
@param bmask: bmask_t
```

IDAPython function idaapi.get_next_fchunk quick reference

```
get_next_fchunk(ea) -> func_t
Get pointer to the next function chunk in the global list.

@param ea (integer): any address in the program
@return: ptr to function chunk or nullptr if next function chunk doesn't exist
```

IDAPython function idaapi.get_next_fcref_from quick reference

```
get_next_fcref_from(frm, current) -> ea_t

@param from: ea_t
@param current: ea_t
```

IDAPython function idaapi.get_next_fcref_to quick reference

```
get_next_fcref_to(to, current) -> ea_t

@param to: ea_t
@param current: ea_t
```

IDAPython function idaapi.get_next_fixup_ea quick reference

```
get_next_fixup_ea(ea) -> ea_t
Find next address with fixup information

@param ea (integer): current address
@return: the next address with fixup information, or BADADDR
```

IDAPython function idaapi.get_next_func quick reference

```
get_next_func(ea) -> func_t
Get pointer to the next function.

@param ea (integer): any address in the program
@return: ptr to function or nullptr if next function doesn't exist
```

IDAPython function idaapi.get_next_func_addr quick reference

```
get_next_func_addr(pfn, ea) -> ea_t

@param pfn: func_t *
@param ea: ea_t
```

IDAPython function idaapi.get_next_hidden_range quick reference

```
get_next_hidden_range(ea) -> hidden_range_t
Get pointer to next hidden range.

@param ea (integer): any address in the program
@return: ptr to hidden range or nullptr if next hidden range doesn't exist
```

IDAPython function idaapi.get_next_member_idx quick reference

```
get_next_member_idx(sptr, off) -> ssize_t
Get the next member idx, if it does not exist, return -1.
```

```
@param sptr: (C++: const struc_t *) struc_t const *
@param off (integer):
```

IDAPython function idaapi.get_next_module quick reference

```
get_next_module(modinfo) -> bool

@param modinfo: modinfo_t *
```

IDAPython function idaapi.get_next_seg quick reference

```
get_next_seg(ea) -> segment_t
Get pointer to the next segment.

@param ea (integer):
```

IDAPython function idaapi.get_next_serial_enum_member quick reference

```
get_next_serial_enum_member(in_out_serial, first_cid) -> const_t

@param in_out_serial: uchar *
@param first_cid: const_t
```

IDAPython function idaapi.get_next_struc_idx quick reference

```
get_next_struc_idx(idx) -> uval_t
Get next struct index.

@param idx (integer):
@return: BADADDR if resulting index is out of bounds, otherwise idx++
```

IDAPython function idaapi.get_nice_colored_name quick reference

```
get_nice_colored_name(ea, flags=0) -> str
Get a nice colored name at the specified address. Ex:
* segment:sub+offset
* segment:sub:local_label
* segment:label
* segment:address
* segment:address+offset
```

```
@param ea (integer): linear address
@param flags (integer): Nice colored name flags
@return: the length of the generated name in bytes.
```

IDAPython function idaapi.get_nlist_ea quick reference

```
get_nlist_ea(idx) -> ea_t
Get address from the list at 'idx'.

@param idx (integer):
```

IDAPython function idaapi.get_nlist_idx quick reference

```
get_nlist_idx(ea) -> size_t
Get index of the name in the list
@warning: returns the closest match. may return idx >= size.

@param ea (integer):
```

IDAPython function idaapi.get_nlist_name quick reference

```
get_nlist_name(idx) -> char const *
Get name using idx.

@param idx (integer):
```

IDAPython function idaapi.get_nlist_size quick reference

```
get_nlist_size() -> size_t
Get number of names in the list.
```

IDAPython function idaapi.get_node_info quick reference

```
get_node_info(out, gid, node) -> bool
Get node info.

@param out: (C++: node_info_t *) result
@param gid: (C++: graph_id_t) id of desired graph
@param node (integer): node number
@return: success
```

IDAPython function idaapi.get_numbered_type quick reference

```
get_numbered_type(til, ordinal) -> (bytes, NoneType, NoneType, NoneType, int), (bytes, b
Retrieve a type by its ordinal number.

@param til: til_t const *
@param ordinal (integer):
```

IDAPython function idaapi.get_numbered_type_name quick reference

```
get_numbered_type_name(ti, ordinal) -> char const *
Get type name (if exists) by its ordinal. If the type is anonymous, returns "".
If failed, returns nullptr

@param ti (idaapi.til_t): til_t const *
@param ordinal (integer):
```

IDAPython function idaapi.get_octet quick reference

```
get_octet(ea, v, nbit) -> (int, int, int, int)

@param ea: ea_t
@param v: uint64
@param nbit: int
```

IDAPython function idaapi.get_octet2 quick reference

```
get_octet2(ogen) -> bool

@param ogen: octet_generator_t *
```

IDAPython function idaapi.get_offbase quick reference

```
get_offbase(ea, n) -> ea_t
Get offset base value

@param ea (integer): linear address
@param n (integer): 0..UA_MAXOP-1 operand number
@return: offset base or BADADDR
```

IDAPython function idaapi.get_offset_expr quick reference

```
get_offset_expr(ea, n, ri, _from, offset, getn_flags=0) -> str
See get_offset_expression()

@param ea (integer):
@param n (integer):
@param ri: (C++: const refinfo_t &) refinfo_t const &
@param from (integer):
@param offset (integer):
@param getn_flags (integer):
```

IDAPython function idaapi.get_offset_expression quick reference

```
get_offset_expression(ea, n, _from, offset, getn_flags=0) -> str
Get offset expression (in the form "offset name+displ"). This function uses
offset translation function ( processor_t::translate) if your IDP module has
such a function. Translation function is used to map linear addresses in the
program (only for offsets).

Example: suppose we have instruction at linear address 0x00011000:
mov     ax, [bx+7422h]  and at ds:7422h:
array   dw      ...  We want to represent the second operand with an offset
expression, so then we call:
get_offset_expresion(0x001100, 1, 0x001102, 0x7422, buf);
                     |         |  |         |       |
                     |         |  |         |       +output buffer
                     |         |  |         +value of offset expression
                     |         |  +address offset value in the instruction
                     |         +the second operand
                     +address of instruction and the function will return a
colored string:
offset array

@param ea (integer): start of instruction or data with the offset expression
@param n (integer): operand number (may be ORed with OPND_OUTER)
* 0: first operand
* 1: second operand
* ...
* 7: eighth operand
@param from (integer): linear address of instruction operand or data referring to the
             name. This address will be used to get fixup information, so it
             should point to exact position of operand in the instruction.
@param offset (integer): value of operand or its part. The function will return text
              representation of this value as offset expression.
```

```
@param getn_flags (integer): combination of:
* GETN_APPZERO: meaningful only if the name refers to a structure. appends the
struct field name if the field offset is zero
* GETN_NODUMMY: do not generate dummy names for the expression but pretend they
already exist (useful to verify that the offset expression can be represented)
@retval 0: can't convert to offset expression
@retval 1: ok, a simple offset expression
@retval 2: ok, a complex offset expression
```

IDAPython function idaapi.get_op_signness quick reference

```
get_op_signness(op) -> type_sign_t
Get operator sign. Meaningful for sign-dependent operators, like cot_sdiv.

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.get_op_tinfo quick reference

```
get_op_tinfo(tif, ea, n) -> bool

@param tif: tinfo_t *
@param ea: ea_t
@param n: int
```

IDAPython function idaapi.get_operand_flag quick reference

```
get_operand_flag(typebits, n) -> flags64_t
Place operand `n`'s type flag in the right nibble of a 64-bit flags set.

@param typebits: (C++: uint8) the type bits (one of `FF_N_`)
@param n (integer): the operand number
@return: the shift to the nibble
```

IDAPython function idaapi.get_operand_type_shift quick reference

```
get_operand_type_shift(n) -> int
Get the shift in `flags64_t` for the nibble representing operand `n`'s type

Note: n must be < UA_MAXOP, and is not checked

@param n (integer): the operand number
@return: the shift to the nibble
```

IDAPython function idaapi.get_opinfo quick reference

```
get_opinfo(buf, ea, n, flags) -> opinfo_t
Get additional information about an operand representation.

@param buf: (C++: opinfo_t *) buffer to receive the result. may not be nullptr
@param ea (integer): linear address of item
@param n (integer): number of operand, 0 or 1
@param flags (integer): flags of the item
@return: nullptr if no additional representation information
```

IDAPython function idaapi.get_opnum quick reference

```
get_opnum() -> int
Get current operand number, -1 means no operand (ui_get_opnum)
```

IDAPython function idaapi.get_optype_flags0 quick reference

```
get_optype_flags0(F) -> flags64_t
Get flags for first operand.

@param F (integer):
```

IDAPython function idaapi.get_optype_flags1 quick reference

```
get_optype_flags1(F) -> flags64_t
Get flags for second operand.

@param F (integer):
```

IDAPython function idaapi.get_or_guess_member_tinfo quick reference

```
get_or_guess_member_tinfo(tif, mptr) -> bool
Try to get tinfo for given member - if failed, generate a tinfo using
information about the member id from the disassembly

@param tif: (C++: tinfo_t *)
@param mptr: (C++: const member_t *) member_t const *
```

IDAPython function idaapi.get_ordinal_from_idb_type quick reference

```
get_ordinal_from_idb_type(name, type) -> int
Get ordinal number of an idb type (struct/enum). The 'type' parameter is used
only to determine the kind of the type (struct or enum) Use this function to
find out the correspondence between idb types and til types

@param name (string): char const *
@param type: (C++: const type_t *) type_t const *
```

IDAPython function idaapi.get_ordinal_qty quick reference

```
get_ordinal_qty(ti) -> uint32
Get number of allocated ordinals.

@param ti (idaapi.til_t): til_t const *
@return: uint32(-1) if failed
```

IDAPython function idaapi.get_original_byte quick reference

```
get_original_byte(ea) -> uint64
Get original byte value (that was before patching). This function works for wide
byte processors too.

@param ea (integer):
```

IDAPython function idaapi.get_original_dword quick reference

```
get_original_dword(ea) -> uint64
Get original dword (that was before patching) This function works for wide byte
processors too. This function takes into account order of bytes specified in
idainfo::is_be()

@param ea (integer):
```

IDAPython function idaapi.get_original_qword quick reference

```
get_original_qword(ea) -> uint64
Get original qword value (that was before patching) This function DOESN'T work
for wide byte processors too. This function takes into account order of bytes
specified in idainfo::is_be()
```

```
@param ea (integer):
```

IDAPython function idaapi.get_original_word quick reference

```
get_original_word(ea) -> uint64
Get original word value (that was before patching). This function works for wide
byte processors too. This function takes into account order of bytes specified
in idainfo::is_be()

@param ea (integer):
```

IDAPython function idaapi.get_outfile_encoding_idx quick reference

```
get_outfile_encoding_idx() -> int
Get the index of the encoding used when producing files
@retval 0: the IDB's default 1 byte-per-unit encoding is used
```

IDAPython function idaapi.get_output_curline quick reference

```
get_output_curline(mouse) -> str
Get current line of output window (ui_get_output_curline).

@param mouse (bool): current for mouse pointer?
@return: false if output contains no text
```

IDAPython function idaapi.get_output_cursor quick reference

```
get_output_cursor() -> bool
Get coordinates of the output window's cursor (ui_get_output_cursor).
@note: coordinates are 0-based
@note: this function will succeed even if the output window is not visible

@retval false: the output window has been destroyed.
@retval true: pointers are filled
```

IDAPython function idaapi.get_output_selected_text quick reference

```
get_output_selected_text() -> str
Returns selected text from output window (ui_get_output_selected_text).
```

```
@return: true if there is a selection
```

IDAPython function idaapi.get_path quick reference

```
get_path(pt) -> char const *
Get the file path

@param pt: (C++: path_type_t) file path type Types of the file pathes
@return: file path, never returns nullptr
```

IDAPython function idaapi.get_ph quick reference

```
get_ph() -> _processor_t
```

IDAPython function idaapi.get_place_class quick reference

```
get_place_class(out_flags, out_sdk_version, id) -> place_t
Get information about a previously-registered place_t class. See also
register_place_class().

@param out_flags: (C++: int *) output flags (can be nullptr)
@param out_sdk_version: (C++: int *) sdk version the place was created with (can be null
@param id (integer): place class ID
@return: the place_t template, or nullptr if not found
```

IDAPython function idaapi.get_place_class_id quick reference

```
get_place_class_id(name) -> int
Get the place class ID for the place that has been registered as 'name'.

@param name (string): the class name
@return: the place class ID, or -1 if not found
```

IDAPython function idaapi.get_place_class_template quick reference

```
get_place_class_template(id) -> place_t
See get_place_class()

@param id (integer):
```

IDAPython function idaapi.get_plugin_options quick reference

```
get_plugin_options(plugin) -> char const *
Get plugin options from the command line. If the user has specified the options
in the -Oplugin_name:options format, them this function will return the
'options' part of it The 'plugin' parameter should denote the plugin name
Returns nullptr if there we no options specified

@param plugin (string): char const *
```

IDAPython function idaapi.get_predef_insn_cmt quick reference

```
get_predef_insn_cmt(ins) -> str
Get predefined comment.

@param ins: (C++: const insn_t &) current instruction information
@return: size of comment or -1
```

IDAPython function idaapi.get_prev_bmask quick reference

```
get_prev_bmask(enum_id, bmask) -> bmask_t
Get prev bitmask in the enum (bitfield)

@param enum_id (integer): id of enum
@param bmask (integer): the current bitmask
@return: value of a bitmask with value lower than the specified value, or
         DEFMASK
```

IDAPython function idaapi.get_prev_enum_member quick reference

```
get_prev_enum_member(id, value, bmask=(bmask_t(-1))) -> uval_t

@param id: enum_t
@param value: uval_t
@param bmask: bmask_t
```

IDAPython function idaapi.get_prev_fchunk quick reference

```
get_prev_fchunk(ea) -> func_t
Get pointer to the previous function chunk in the global list.

@param ea (integer): any address in the program
```

```
@return: ptr to function chunk or nullptr if previous function chunk doesn't
         exist
```

IDAPython function idaapi.get_prev_fixup_ea quick reference

```
get_prev_fixup_ea(ea) -> ea_t
Find previous address with fixup information

@param ea (integer): current address
@return: the previous address with fixup information, or BADADDR
```

IDAPython function idaapi.get_prev_func quick reference

```
get_prev_func(ea) -> func_t
Get pointer to the previous function.

@param ea (integer): any address in the program
@return: ptr to function or nullptr if previous function doesn't exist
```

IDAPython function idaapi.get_prev_func_addr quick reference

```
get_prev_func_addr(pfn, ea) -> ea_t

@param pfn: func_t *
@param ea: ea_t
```

IDAPython function idaapi.get_prev_hidden_range quick reference

```
get_prev_hidden_range(ea) -> hidden_range_t
Get pointer to previous hidden range.

@param ea (integer): any address in the program
@return: ptr to hidden range or nullptr if previous hidden range doesn't exist
```

IDAPython function idaapi.get_prev_member_idx quick reference

```
get_prev_member_idx(sptr, off) -> ssize_t
Get the prev member idx, if it does not exist, return -1.

@param sptr: (C++: const struc_t *) struc_t const *
@param off (integer):
```

IDAPython function idaapi.get_prev_seg quick reference

```
get_prev_seg(ea) -> segment_t
Get pointer to the previous segment.

@param ea (integer):
```

IDAPython function idaapi.get_prev_serial_enum_member quick reference

```
get_prev_serial_enum_member(in_out_serial, first_cid) -> const_t

@param in_out_serial: uchar *
@param first_cid: const_t
```

IDAPython function idaapi.get_prev_sreg_range quick reference

```
get_prev_sreg_range(out, ea, rg) -> bool
Get segment register range previous to one with address.
@note: more efficient then get_sreg_range(reg, ea-1)

@param out: (C++: sreg_range_t *) segment register range
@param ea (integer): any linear address in the program
@param rg (integer): the segment register number
@return: success
```

IDAPython function idaapi.get_prev_struc_idx quick reference

```
get_prev_struc_idx(idx) -> uval_t
Get previous struct index.

@param idx (integer):
@return: BADADDR if resulting index is negative, otherwise idx - 1
```

IDAPython function idaapi.get_printable_immvals quick reference

```
get_printable_immvals(ea, n, F=0) -> PyObject *
Get immediate ready-to-print values at the specified address

@param ea (integer): address to analyze
@param n (integer): 0..UA_MAXOP-1 operand number, OPND_ALL all the operands
@param F (integer): flags for the specified address
@return: number of immediate values (0..2*UA_MAXOP)
```

IDAPython function idaapi.get_problem quick reference

```
get_problem(type, lowea) -> ea_t
Get an address from the specified problem list. The address is not removed from
the list.

@param type: (C++: problist_id_t) problem list type
@param lowea (integer): the returned address will be higher or equal than the specified
              address
@return: linear address or BADADDR
```

IDAPython function idaapi.get_problem_desc quick reference

```
get_problem_desc(t, ea) -> str
Get the human-friendly description of the problem, if one was provided to
remember_problem.

@param t: (C++: problist_id_t) problem list type.
@param ea (integer): linear address.
@return: the message length or -1 if none
```

IDAPython function idaapi.get_problem_name quick reference

```
get_problem_name(type, longname=True) -> char const *
Get problem list description.

@param type: (C++: problist_id_t)
@param longname (bool):
```

IDAPython function idaapi.get_process_options quick reference

```
get_process_options()
Get process options. Any of the arguments may be nullptr
```

IDAPython function idaapi.get_process_state quick reference

```
get_process_state() -> int
Return the state of the currently debugged process. \sq{Type, Synchronous
function, Notification, none (synchronous function)}

@return: one of Debugged process states
```

IDAPython function idaapi.get_processes quick reference

```
get_processes(proclist) -> ssize_t
Take a snapshot of running processes and return their description. \sq{Type,
Synchronous function, Notification, none (synchronous function)}

@param proclist: (C++: procinfo_vec_t *) array with information about each running proce
@return: number of processes or -1 on error
```

IDAPython function idaapi.get_qword quick reference

```
get_qword(ea) -> uint64
Get one qword (64-bit) of the program at 'ea'. This function takes into account
order of bytes specified in idainfo::is_be() This function works only for 8bit
byte processors.

@param ea (integer):
```

IDAPython function idaapi.get_radix quick reference

```
get_radix(F, n) -> int
Get radix of the operand, in: flags. If the operand is not a number, returns
get_default_radix()

@param F (integer): flags
@param n (integer): number of operand (0, 1, -1)
@return: 2, 8, 10, 16
```

IDAPython function idaapi.get_refinfo quick reference

```
get_refinfo(ri, ea, n) -> bool

@param ri: refinfo_t *
@param ea: ea_t
@param n: int
```

IDAPython function idaapi.get_reftype_by_size quick reference

```
get_reftype_by_size(size) -> reftype_t
Get REF_... constant from size Supported sizes: 1,2,4,8,16 For other sizes
returns reftype_t(-1)
```

```
@param size (integer):
```

IDAPython function idaapi.get_reg_info quick reference

```
get_reg_info(regname, bitrange) -> char const *

@param regname: char const *
@param bitrange: bitrange_t *
```

IDAPython function idaapi.get_reg_name quick reference

```
get_reg_name(reg, width, reghi=-1) -> str
Get text representation of a register. For most processors this function will
just return processor_t::reg_names[reg]. If the processor module has implemented
processor_t::get_reg_name, it will be used instead

@param reg (integer): internal register number as defined in the processor module
@param width (integer): register width in bytes
@param reghi (integer): if specified, then this function will return the register pair
@return: length of register name in bytes or -1 if failure
```

IDAPython function idaapi.get_reg_val quick reference

```
get_reg_val(regname, regval) -> bool
Get register value as an unsigned 64-bit int.

@param regname (string): char const *
@param regval: regval_t *

get_reg_val(regname, ival) -> bool

@param regname: char const *
@param ival: uint64 *

get_reg_val(regname) -> bool, float, int

@param regname: char const *
```

IDAPython function idaapi.get_reg_vals quick reference

```
get_reg_vals(tid, clsmask=-1) -> regvals_t
Fetch live registers values for the thread
```

```
@param tid: The ID of the thread to read registers for
@param clsmask: An OR'ed mask of register classes to
       read values for (can be used to speed up the
       retrieval process)

@return: a regvals_t instance (empty if an error occurs)
```

IDAPython function idaapi.get_registered_actions quick reference

```
get_registered_actions() -> [str, ...]
Get a list of all currently-registered actions
```

IDAPython function idaapi.get_ret_tev_return quick reference

```
get_ret_tev_return(n) -> ea_t
Get the return address from a function return trace event. \sq{Type, Synchronous
function, Notification, none (synchronous function)}

@param n (integer): number of trace event, is in range 0..get_tev_qty()-1. 0 represents
          the latest added trace event.
@return: BADADDR if not a function return event.
```

IDAPython function idaapi.get_root_filename quick reference

```
get_root_filename() -> str
Get file name only of the input file.
```

IDAPython function idaapi.get_running_notification quick reference

```
get_running_notification() -> dbg_notification_t
Get the notification associated (if any) with the current running request.
\sq{Type, Synchronous function, Notification, none (synchronous function)}

@return: dbg_null if no running request
```

IDAPython function idaapi.get_running_request quick reference

```
get_running_request() -> ui_notification_t
Get the current running request. \sq{Type, Synchronous function, Notification,
none (synchronous function)}
```

```
@return: ui_null if no running request
```

IDAPython function idaapi.get_scalar_bt quick reference

```
get_scalar_bt(size) -> type_t

@param size: int
```

IDAPython function idaapi.get_screen_ea quick reference

```
get_screen_ea() -> ea_t
Get the address at the screen cursor (ui_screenea)
```

IDAPython function idaapi.get_segm_base quick reference

```
get_segm_base(s) -> ea_t
Get segment base linear address. Segment base linear address is used to
calculate virtual addresses. The virtual address of the first byte of the
segment will be (start address of segment - segment base linear address)

@param s: (C++: const segment_t *) pointer to segment
@return: 0 if s == nullptr, otherwise segment base linear address
```

IDAPython function idaapi.get_segm_by_name quick reference

```
get_segm_by_name(name) -> segment_t
Get pointer to segment by its name. If there are several segments with the same
name, returns the first of them.

@param name (string): segment name. may be nullptr.
@return: nullptr or pointer to segment structure
```

IDAPython function idaapi.get_segm_by_sel quick reference

```
get_segm_by_sel(selector) -> segment_t
Get pointer to segment structure. This function finds a segment by its selector.
If there are several segments with the same selectors, the last one will be
returned.

@param selector: (C++: sel_t) a segment with the specified selector will be returned
@return: pointer to segment or nullptr
```

IDAPython function idaapi.get_segm_class quick reference

```
get_segm_class(s) -> str
Get segment class. Segment class is arbitrary text (max 8 characters).

@param s: (C++: const segment_t *) pointer to segment
@return: size of segment class (-1 if s==nullptr or bufsize<=0)
```

IDAPython function idaapi.get_segm_name quick reference

```
get_segm_name(s, flags=0) -> str
Get true segment name by pointer to segment.

@param s: (C++: const segment_t *) pointer to segment
@param flags (integer): 0-return name as is; 1-substitute bad symbols with _ 1 correspor
              to GN_VISIBLE
@return: size of segment name (-1 if s==nullptr)
```

IDAPython function idaapi.get_segm_num quick reference

```
get_segm_num(ea) -> int
Get number of segment by address.

@param ea (integer): linear address belonging to the segment
@return: -1 if no segment occupies the specified address. otherwise returns
         number of the specified segment (0..get_segm_qty()-1)
```

IDAPython function idaapi.get_segm_para quick reference

```
get_segm_para(s) -> ea_t
Get segment base paragraph. Segment base paragraph may be converted to segment
base linear address using to_ea() function. In fact, to_ea(get_segm_para(s), 0)
== get_segm_base(s).

@param s: (C++: const segment_t *) pointer to segment
@return: 0 if s == nullptr, the segment base paragraph
```

IDAPython function idaapi.get_segm_qty quick reference

```
get_segm_qty() -> int
Get number of segments.
```

IDAPython function idaapi.get_segment_alignment quick reference

```
get_segment_alignment(align) -> char const *
Get text representation of segment alignment code.

@param align: (C++: uchar)
@return: text digestable by IBM PC assembler.
```

IDAPython function idaapi.get_segment_cmt quick reference

```
get_segment_cmt(s, repeatable) -> str
Get segment comment.

@param s: (C++: const segment_t *) pointer to segment structure
@param repeatable (bool): 0: get regular comment. 1: get repeatable comment.
@return: size of comment or -1
```

IDAPython function idaapi.get_segment_combination quick reference

```
get_segment_combination(comb) -> char const *
Get text representation of segment combination code.

@param comb: (C++: uchar)
@return: text digestable by IBM PC assembler.
```

IDAPython function idaapi.get_segment_translations quick reference

```
get_segment_translations(transmap, segstart) -> ssize_t
Get segment translation list.

@param transmap: (C++: eavec_t *) vector of segment start addresses for the translation
@param segstart (integer): start address of the segment to get information about
@return: -1 if no translation list or bad segstart. otherwise returns size of
         translation list.
```

IDAPython function idaapi.get_selector_qty quick reference

```
get_selector_qty() -> size_t
Get number of defined selectors.
```

IDAPython function idaapi.get_short_name quick reference

```
get_short_name(ea, gtn_flags=0) -> qstring

@param ea: ea_t
@param gtn_flags: int
```

IDAPython function idaapi.get_signed_mcode quick reference

```
get_signed_mcode(code) -> mcode_t

@param code: enum mcode_t
```

IDAPython function idaapi.get_source_linnum quick reference

```
get_source_linnum(ea) -> uval_t

@param ea: ea_t
```

IDAPython function idaapi.get_sourcefile quick reference

```
get_sourcefile(ea, bounds=None) -> char const *
Get name of source file occupying the given address.

@param ea (integer): linear address
@param bounds: (C++: range_t *) pointer to the output buffer with the address range for
               current file. May be nullptr.
@return: nullptr if source file information is not found, otherwise returns
         pointer to file name
```

IDAPython function idaapi.get_sp_delta quick reference

```
get_sp_delta(pfn, ea) -> sval_t
Get modification of SP made at the specified location

@param pfn (idaapi.func_t): pointer to function. may be nullptr.
@param ea (integer): linear address
@return: 0 if the specified location doesn't contain a SP change point.
         otherwise return delta of SP modification.
```

IDAPython function idaapi.get_sp_val quick reference

```
get_sp_val() -> bool
```

```
Get value of the SP register for the current thread. Requires a suspended
debugger.
```

IDAPython function idaapi.get_spd quick reference

```
get_spd(pfn, ea) -> sval_t
Get difference between the initial and current values of ESP.

@param pfn (idaapi.func_t): pointer to function. may be nullptr.
@param ea (integer): linear address of an instruction
@return: 0 or the difference, usually a negative number. returns the sp-diff
         before executing the instruction.
```

IDAPython function idaapi.get_special_folder quick reference

```
get_special_folder(csidl) -> str
Get a folder location by CSIDL (see Common CSIDLs). Path should be of at least
MAX_PATH size

@param csidl (integer):
```

IDAPython function idaapi.get_sptr quick reference

```
get_sptr(mptr) -> struc_t
Get child struct if member is a struct.

@param mptr: (C++: const member_t *) member_t const *
```

IDAPython function idaapi.get_srcdbg_paths quick reference

```
get_srcdbg_paths() -> str
Get source debug paths.
```

IDAPython function idaapi.get_srcdbg_undesired_paths quick reference

```
get_srcdbg_undesired_paths() -> str
Get user-closed source files.
```

IDAPython function idaapi.get_srcinfo_provider quick reference

```
get_srcinfo_provider(name) -> srcinfo_provider_t *
```

```
@param name: char const *
```

IDAPython function idaapi.get_sreg quick reference

```
get_sreg(ea, rg) -> sel_t
Get value of a segment register. This function uses segment register range and
default segment register values stored in the segment structure.

@param ea (integer): linear address in the program
@param rg (integer): number of the segment register
@return: value of the segment register, BADSEL if value is unknown.
```

IDAPython function idaapi.get_sreg_range quick reference

```
get_sreg_range(out, ea, rg) -> bool
Get segment register range by linear address.

@param out: (C++: sreg_range_t *) segment register range
@param ea (integer): any linear address in the program
@param rg (integer): the segment register number
@return: success
```

IDAPython function idaapi.get_sreg_range_num quick reference

```
get_sreg_range_num(ea, rg) -> int
Get number of segment register range by address.

@param ea (integer): any address in the range
@param rg (integer): the segment register number
@return: -1 if no range occupies the specified address. otherwise returns number
         of the specified range (0..get_srranges_qty()-1)
```

IDAPython function idaapi.get_sreg_ranges_qty quick reference

```
get_sreg_ranges_qty(rg) -> size_t
Get number of segment register ranges.

@param rg (integer): the segment register number
```

IDAPython function idaapi.get_std_dirtree quick reference

```
get_std_dirtree(id) -> dirtree_t

@param id: enum dirtree_id_t
```

IDAPython function idaapi.get_step_trace_options quick reference

```
get_step_trace_options() -> int
Get current step tracing options. \sq{Type, Synchronous function, Notification,
none (synchronous function)}

@return: Step trace options
```

IDAPython function idaapi.get_stkvar quick reference

```
get_stkvar(insn, op, v) -> (member_t, int) or None
Get pointer to stack variable

@param insn: an idaapi.insn_t, or an address (C++: const insn_t &)
@param op: reference to instruction operand
@param v: immediate value in the operand (usually op.addr)
@return:     - None on failure
    - tuple(member_t, actval)
      where actval: actual value used to fetch stack variable
```

IDAPython function idaapi.get_stock_tinfo quick reference

```
get_stock_tinfo(tif, id) -> bool

@param tif: tinfo_t *
@param id: enum stock_type_id_t
```

IDAPython function idaapi.get_str_encoding_idx quick reference

```
get_str_encoding_idx(strtype) -> uchar
Get index of the string encoding for this string.

@param strtype: (C++: int32)
```

IDAPython function idaapi.get_str_term1 quick reference

```
get_str_term1(strtype) -> char
```

```
@param strtype: int32
```

IDAPython function idaapi.get_str_term2 quick reference

```
get_str_term2(strtype) -> char

@param strtype: int32
```

IDAPython function idaapi.get_str_type quick reference

```
get_str_type(ea) -> uint32

@param ea: ea_t
```

IDAPython function idaapi.get_str_type_code quick reference

```
get_str_type_code(strtype) -> uchar

@param strtype: int32
```

IDAPython function idaapi.get_str_type_prefix_length quick reference

```
get_str_type_prefix_length(strtype) -> size_t

@param strtype: int32
```

IDAPython function idaapi.get_strid quick reference

```
get_strid(ea) -> tid_t

@param ea: ea_t
```

IDAPython function idaapi.get_strlist_item quick reference

```
get_strlist_item(si, n) -> bool
Get nth element of the string list (n=0..get_strlist_qty()-1)

@param si: (C++: string_info_t *)
@param n (integer):
```

IDAPython function idaapi.get_strlist_options quick reference

```
get_strlist_options() -> strwinsetup_t
Get the static string list options.
```

IDAPython function idaapi.get_strlist_qty quick reference

```
get_strlist_qty() -> size_t
Get number of elements in the string list. The list will be loaded from the
database (if saved) or built from scratch.
```

IDAPython function idaapi.get_strlit_contents quick reference

```
get_strlit_contents(ea, py_len, type, flags=0) -> bytes or None
Get contents of string literal, as UTF-8-encoded codepoints.
It works even if the string has not been created in the database yet.

Note that the returned value will be of type 'bytes'; if
you want auto-conversion to unicode strings (that is: real Python
strings), you should probably be using the idautils.Strings class.

@param ea: linear address of the string
@param py_len: length of the string in bytes (including terminating 0)
@param type: type of the string. Represents both the character encoding,
             <u>and</u> the 'type' of string at the given location.
@param flags: combination of STRCONV_..., to perform output conversion.
@return: a bytes-filled str object.
```

IDAPython function idaapi.get_stroff_path quick reference

```
get_stroff_path(path, delta, ea, n) -> int
Get struct path of operand.

@param path: (C++: tid_t *) buffer for structure path (strpath). see nalt.hpp for more i
@param delta: (C++: adiff_t *) struct offset delta
@param ea (integer): linear address
@param n (integer): 0..UA_MAXOP-1 operand number, OPND_ALL one of the operands
@return: length of strpath
```

IDAPython function idaapi.get_strtype_bpu quick reference

```
get_strtype_bpu(strtype) -> int
```

```
@param strtype: int32
```

IDAPython function idaapi.get_struc quick reference

```
get_struc(id) -> struc_t
Get pointer to struct type info.

@param id (integer):
```

IDAPython function idaapi.get_struc_by_idx quick reference

```
get_struc_by_idx(idx) -> tid_t
Get struct id by struct number.

@param idx (integer):
```

IDAPython function idaapi.get_struc_cmt quick reference

```
get_struc_cmt(id, repeatable) -> str
Get struct comment.

@param id (integer):
@param repeatable (bool):
```

IDAPython function idaapi.get_struc_first_offset quick reference

```
get_struc_first_offset(sptr) -> ea_t
Get offset of first member.

@param sptr: (C++: const struc_t *) struc_t const *
@return: BADADDR if memqty == 0
```

IDAPython function idaapi.get_struc_id quick reference

```
get_struc_id(name) -> tid_t
Get struct id by name.

@param name (string): char const *
```

IDAPython function idaapi.get_struc_idx quick reference

```
get_struc_idx(id) -> uval_t
```

```
Get internal number of the structure.

@param id (integer):
```

IDAPython function idaapi.get_struc_last_offset quick reference

```
get_struc_last_offset(sptr) -> ea_t
Get offset of last member.

@param sptr: (C++: const struc_t *) struc_t const *
@return: BADADDR if memqty == 0
```

IDAPython function idaapi.get_struc_name quick reference

```
get_struc_name(id, flags=0) -> str

@param id: tid_t
@param flags: int
```

IDAPython function idaapi.get_struc_next_offset quick reference

```
get_struc_next_offset(sptr, offset) -> ea_t
Get offset of member with smallest offset larger than 'offset'.

@param sptr: (C++: const struc_t *) struc_t const *
@param offset (integer):
@return: BADADDR if no next offset
```

IDAPython function idaapi.get_struc_prev_offset quick reference

```
get_struc_prev_offset(sptr, offset) -> ea_t
Get offset of member with largest offset less than 'offset'.

@param sptr: (C++: const struc_t *) struc_t const *
@param offset (integer):
@return: BADADDR if no prev offset
```

IDAPython function idaapi.get_struc_qty quick reference

```
get_struc_qty() -> size_t
Get number of known structures.
```

IDAPython function idaapi.get_struc_size quick reference

```
get_struc_size(sptr) -> asize_t
Get struct size (also see get_struc_size(const struc_t *))

@param sptr: struc_t const *

get_struc_size(id) -> asize_t

@param id: tid_t
```

IDAPython function idaapi.get_switch_parent quick reference

```
get_switch_parent(ea) -> ea_t

@param ea: ea_t
```

IDAPython function idaapi.get_synced_group quick reference

```
get_synced_group(w) -> synced_group_t
Get the group of widgets/registers this view is synchronized with

@param w: (C++: const TWidget *) the widget
@return: the group of widgets/registers, or nullptr
```

IDAPython function idaapi.get_tab_size quick reference

```
get_tab_size(path) -> int
Get the size of a tab in spaces (ui_get_tab_size).

@param path (string): the path of the source view for which the tab size is requested.
* if nullptr, the default size is returned.
```

IDAPython function idaapi.get_temp_regs quick reference

```
get_temp_regs() -> mlist_t
Get list of temporary registers. Tempregs are temporary registers that are used
during code generation. They do not map to regular processor registers. They are
used only to store temporary values during execution of one instruction.
Tempregs may not be used to pass a value from one block to another. In other
words, at the end of a block all tempregs must be dead.
```

IDAPython function idaapi.get_tev_ea quick reference

```
get_tev_ea(n) -> ea_t

@param n: int
```

IDAPython function idaapi.get_tev_event quick reference

```
get_tev_event(n, d) -> bool
Get the corresponding debug event, if any, for the specified tev object.
\sq{Type, Synchronous function, Notification, none (synchronous function)}

@param n (integer): number of trace event, is in range 0..get_tev_qty()-1. 0 represents
          the latest added trace event.
@param d: (C++: debug_event_t *) result
@return: false if the tev_t object doesn't have any associated debug event, true
         otherwise, with the debug event in "d".
```

IDAPython function idaapi.get_tev_info quick reference

```
get_tev_info(n, tev_info) -> bool
Get main information about a trace event. \sq{Type, Synchronous function,
Notification, none (synchronous function)}

@param n (integer): number of trace event, is in range 0..get_tev_qty()-1. 0 represents
          the latest added trace event.
@param tev_info: (C++: tev_info_t *) result
@return: success
```

IDAPython function idaapi.get_tev_memory_info quick reference

```
get_tev_memory_info(n, mi) -> bool
Get the memory layout, if any, for the specified tev object. \sq{Type,
Synchronous function, Notification, none (synchronous function)}

@param n (integer): number of trace event, is in range 0..get_tev_qty()-1. 0 represents
          the latest added trace event.
@param mi: (C++: meminfo_vec_t *) result
@return: false if the tev_t object is not of type tev_mem, true otherwise, with
         the new memory layout in "mi".
```

IDAPython function idaapi.get_tev_qty quick reference

```
get_tev_qty() -> int
Get number of trace events available in trace buffer. \sq{Type, Synchronous
function, Notification, none (synchronous function)}
```

IDAPython function idaapi.get_tev_tid quick reference

```
get_tev_tid(n) -> int

@param n: int
```

IDAPython function idaapi.get_tev_type quick reference

```
get_tev_type(n) -> int

@param n: int
```

IDAPython function idaapi.get_thread_qty quick reference

```
get_thread_qty() -> int
Get number of threads. \sq{Type, Synchronous function, Notification, none
(synchronous function)}
```

IDAPython function idaapi.get_tinfo quick reference

```
get_tinfo(tif, ea) -> bool

@param tif: tinfo_t *
@param ea: ea_t
```

IDAPython function idaapi.get_tinfo_attr quick reference

```
get_tinfo_attr(typid, key, bv, all_attrs) -> bool

@param typid: uint32
@param key: qstring const &
@param bv: bytevec_t *
@param all_attrs: bool
```

IDAPython function idaapi.get__tinfo__attrs quick reference

```
get_tinfo_attrs(typid, tav, include_ref_attrs) -> bool

@param typid: uint32
@param tav: type_attrs_t *
@param include_ref_attrs: bool
```

IDAPython function idaapi.get__tinfo__details quick reference

```
get_tinfo_details(typid, bt2, buf) -> bool

@param typid: uint32
@param bt2: type_t
@param buf
```

IDAPython function idaapi.get__tinfo__pdata quick reference

```
get_tinfo_pdata(outptr, typid, what) -> size_t

@param outptr
@param typid: uint32
@param what: int
```

IDAPython function idaapi.get__tinfo__property quick reference

```
get_tinfo_property(typid, gta_prop) -> size_t

@param typid: uint32
@param gta_prop: int
```

IDAPython function idaapi.get__tinfo__size quick reference

```
get_tinfo_size(p_effalign, typid, gts_code) -> size_t

@param p_effalign: uint32 *
@param typid: uint32
@param gts_code: int
```

IDAPython function idaapi.get__trace__base__address quick reference

```
get_trace_base_address() -> ea_t
Get the base address of the current trace. \sq{Type, Synchronous function,
```

```
Notification, none (synchronous function)}

@return: the base address of the currently loaded trace
```

IDAPython function idaapi.get_trace_dynamic_register_set quick reference

```
get_trace_dynamic_register_set(idaregs)
Get dynamic register set of current trace.

@param idaregs: (C++: dynamic_register_set_t *)
```

IDAPython function idaapi.get_trace_file_desc quick reference

```
get_trace_file_desc(filename) -> str
Get the file header of the specified trace file.

@param filename (string): char const *
```

IDAPython function idaapi.get_trace_platform quick reference

```
get_trace_platform() -> char const *
Get platform name of current trace.
```

IDAPython function idaapi.get_tryblks quick reference

```
get_tryblks(tbv, range) -> size_t
Retrieve try block information from the specified address range. Try blocks are
sorted by starting address and their nest levels calculated.

@param tbv: (C++: tryblks_t *) output buffer; may be nullptr
@param range: (C++: const range_t &) address range to change
@return: number of found try blocks
```

IDAPython function idaapi.get_type quick reference

```
get_type(id, tif, guess) -> bool
Get a global type. Global types are types of addressable objects and
struct/union/enum types

@param id (integer): address or id of the object
@param tif: (C++: tinfo_t *) buffer for the answer
```

```
@param guess: (C++: type_source_t) what kind of types to consider
@return: success
```

IDAPython function idaapi.get__type__flags quick reference

```
get_type_flags(t) -> type_t
Get type flags (TYPE_FLAGS_MASK)

@param t: (C++: type_t)
```

IDAPython function idaapi.get__type__ordinal quick reference

```
get_type_ordinal(ti, name) -> int32
Get type ordinal by its name.

@param ti (idaapi.til_t): til_t const *
@param name (string): char const *
```

IDAPython function idaapi.get__unk__type quick reference

```
get_unk_type(size) -> tinfo_t
Create a partial type info by width. Returns a partially defined type (examples:
_DWORD, _BYTE) with the given width.

@param size (integer): size of the type in bytes
```

IDAPython function idaapi.get__unsigned__mcode quick reference

```
get_unsigned_mcode(code) -> mcode_t

@param code: enum mcode_t
```

IDAPython function idaapi.get__user__idadir quick reference

```
get_user_idadir() -> char const *
Get user ida related directory.
- if $IDAUSR is defined:
- the first element in $IDAUSR
- else
- default user directory ($HOME/.idapro or %APPDATA%Hex-Rays/IDA Pro)
```

IDAPython function idaapi.get_user_input_event quick reference

```
get_user_input_event(out) -> bool
Get the current user input event (mouse button press, key press, ...) It is
sometimes desirable to be able to tell when a certain situation happens (e.g.,
'view_curpos' gets triggered); this function exists to provide that context (GUI
version only)

@param out: (C++: input_event_t *) the input event data
@return: false if we are not currently processing a user input event
```

IDAPython function idaapi.get_user_strlist_options quick reference

```
get_user_strlist_options(out)

@param out: strwinsetup_t *
```

IDAPython function idaapi.get_vftable_ea quick reference

```
get_vftable_ea(ordinal) -> ea_t
Get address of a virtual function table.

@param ordinal (integer): ordinal number of a vftable type.
@return: address of the corresponding virtual function table in the current
         database.
```

IDAPython function idaapi.get_vftable_ordinal quick reference

```
get_vftable_ordinal(vftable_ea) -> uint32
Get ordinal number of the virtual function table.

@param vftable_ea (integer): address of a virtual function table.
@return: ordinal number of the corresponding vftable type. 0 - failure.
```

IDAPython function idaapi.get_view_renderer_type quick reference

```
get_view_renderer_type(v) -> tcc_renderer_type_t
Get the type of renderer currently in use in the given view
(ui_get_renderer_type)

@param v(a Widget SWIG wrapper class):
```

IDAPython function idaapi.get_viewer_graph quick reference

```
get_viewer_graph(gv) -> mutable_graph_t
Get graph object for given custom graph viewer.

@param gv: (C++: graph_viewer_t *)
```

IDAPython function idaapi.get_viewer_place_type quick reference

```
get_viewer_place_type(viewer) -> tcc_place_type_t
Get the type of place_t instances a viewer uses & creates
(ui_get_viewer_place_type).

@param viewer(a Widget SWIG wrapper class):
```

IDAPython function idaapi.get_viewer_user_data quick reference

```
get_viewer_user_data(viewer) -> void *
Get the user data from a custom viewer (ui_get_viewer_user_data)

@param viewer(a Widget SWIG wrapper class):
```

IDAPython function idaapi.get_visible_name quick reference

```
get_visible_name(ea, gtn_flags=0) -> qstring

@param ea: ea_t
@param gtn_flags: int
```

IDAPython function idaapi.get_visible_segm_name quick reference

```
get_visible_segm_name(s) -> str
Get segment name by pointer to segment.

@param s: (C++: const segment_t *) pointer to segment
@return: size of segment name (-1 if s==nullptr)
```

IDAPython function idaapi.get_wide_byte quick reference

```
get_wide_byte(ea) -> uint64
Get one wide byte of the program at 'ea'. Some processors may access more than
8bit quantity at an address. These processors have 32-bit byte organization from
the IDA's point of view.
```

```
@param ea (integer):
```

IDAPython function idaapi.get_wide_dword quick reference

```
get_wide_dword(ea) -> uint64
Get two wide words (4 'bytes') of the program at 'ea'. Some processors may
access more than 8bit quantity at an address. These processors have 32-bit byte
organization from the IDA's point of view. This function takes into account
order of bytes specified in idainfo::is_be()
@note: this function works incorrectly if processor_t::nbits > 16

@param ea (integer):
```

IDAPython function idaapi.get_wide_word quick reference

```
get_wide_word(ea) -> uint64
Get one wide word (2 'byte') of the program at 'ea'. Some processors may access
more than 8bit quantity at an address. These processors have 32-bit byte
organization from the IDA's point of view. This function takes into account
order of bytes specified in idainfo::is_be()

@param ea (integer):
```

IDAPython function idaapi.get_widget_title quick reference

```
get_widget_title(widget) -> str
Get the TWidget's title (ui_get_widget_title).

@param widget(a Widget SWIG wrapper class):
```

IDAPython function idaapi.get_widget_type quick reference

```
get_widget_type(widget) -> twidget_type_t
Get the type of the TWidget * (ui_get_widget_type).

@param widget(a Widget SWIG wrapper class):
```

IDAPython function idaapi.get_widget_vdui quick reference

```
get_widget_vdui(f) -> vdui_t
Get the vdui_t instance associated to the TWidget
```

```
@param f(a Widget SWIG wrapper class): pointer to window
@return: a vdui_t *, or nullptr
```

IDAPython function idaapi.get__window__id quick reference

```
get_window_id(name=None) -> void *
Get the system-specific window ID (GUI version only)

@param name (string): name of the window (nullptr means the main IDA window)
@return: the low-level window ID
```

IDAPython function idaapi.get__word quick reference

```
get_word(ea) -> ushort
Get one word (16-bit) of the program at 'ea'. This function takes into account
order of bytes specified in idainfo::is_be() This function works only for 8bit
byte processors.

@param ea (integer):
```

IDAPython function idaapi.get__zero__ranges quick reference

```
get_zero_ranges(zranges, range) -> bool
Return set of ranges with zero initialized bytes. The returned set includes only
big zero initialized ranges (at least >1KB). Some zero initialized byte ranges
may be not included. Only zero bytes that use the sparse storage method (STT_MM)
are reported.

@param zranges: (C++: rangeset_t *) pointer to the return value. cannot be nullptr
@param range: (C++: const range_t *) the range of addresses to verify. can be nullptr -
              ranges
@return: true if the result is a non-empty set
```

IDAPython function idaapi.getb__reginsn quick reference

```
getb_reginsn(ins) -> minsn_t

@param ins: minsn_t *
```

IDAPython function idaapi.getf_reginsn quick reference

```
getf_reginsn(ins) -> minsn_t

@param ins: minsn_t *
```

IDAPython function idaapi.getinf_str quick reference

```
getinf_str(tag) -> str
Get program specific information (a non-scalar value)

@param tag: (C++: inftag_t) one of inftag_t constants
@return: number of bytes stored in the buffer (<0 - not defined)
```

IDAPython function idaapi.getn_bpt quick reference

```
getn_bpt(n, bpt) -> bool
Get the characteristics of a breakpoint. \sq{Type, Synchronous function,
Notification, none (synchronous function)}

@param n (integer): number of breakpoint, is in range 0..get_bpt_qty()-1
@param bpt: (C++: bpt_t *) filled with the characteristics.
@return: false if no breakpoint exists
```

IDAPython function idaapi.getn_enum quick reference

```
getn_enum(idx) -> enum_t
Get enum by its index in the list of enums (0..get_enum_qty()-1).

@param idx (integer):
```

IDAPython function idaapi.getn_fchunk quick reference

```
getn_fchunk(n) -> func_t
Get pointer to function chunk structure by number.

@param n (integer): number of function chunk, is in range 0..get_fchunk_qty()-1
@return: ptr to a function chunk or nullptr. This function may return a function
         entry as well as a function tail.
```

IDAPython function idaapi.getn_func quick reference

```
getn_func(n) -> func_t
```

```
Get pointer to function structure by number.

@param n (integer): number of function, is in range 0..get_func_qty()-1
@return: ptr to a function or nullptr. This function returns a function entry
         chunk.
```

IDAPython function idaapi.getn_hidden_range quick reference

```
getn_hidden_range(n) -> hidden_range_t
Get pointer to hidden range structure, in: number of hidden range.

@param n (integer): number of hidden range, is in range 0..get_hidden_range_qty()-1
```

IDAPython function idaapi.getn_selector quick reference

```
getn_selector(n) -> bool
Get description of selector (0..get_selector_qty()-1)

@param n (integer):
```

IDAPython function idaapi.getn_sreg_range quick reference

```
getn_sreg_range(out, rg, n) -> bool
Get segment register range by its number.

@param out: (C++: sreg_range_t *) segment register range
@param rg (integer): the segment register number
@param n (integer): number of range (0..qty()-1)
@return: success
```

IDAPython function idaapi.getn_thread quick reference

```
getn_thread(n) -> thid_t
Get the ID of a thread. \sq{Type, Synchronous function, Notification, none
(synchronous function)}

@param n (integer): number of thread, is in range 0..get_thread_qty()-1
@return: NO_THREAD if the thread doesn't exist.
```

IDAPython function idaapi.getn_thread_name quick reference

```
getn_thread_name(n) -> char const *
```

```
Get the NAME of a thread \sq{Type, Synchronous function, Notification, none
(synchronous function)}

@param n (integer): number of thread, is in range 0..get_thread_qty()-1 or -1 for the
          current thread
@return: thread name or nullptr if the thread doesn't exist.
```

IDAPython function idaapi.getnode quick reference

```
getnode(ea) -> netnode

@param ea: ea_t
```

IDAPython function idaapi.getnseg quick reference

```
getnseg(n) -> segment_t
Get pointer to segment by its number.
@warning: Obsoleted because it can slow down the debugger (it has to refresh the
          whole memory segmentation to calculate the correct answer)

@param n (integer): segment number in the range (0..get_segm_qty()-1)
@return: nullptr or pointer to segment structure
```

IDAPython function idaapi.getseg quick reference

```
getseg(ea) -> segment_t
Get pointer to segment by linear address.

@param ea (integer): linear address belonging to the segment
@return: nullptr or pointer to segment structure
```

IDAPython function idaapi.getsysfile quick reference

```
getsysfile(filename, subdir) -> str
Search for IDA system file. This function searches for a file in:
1. each directory specified by IDAUSR%
2. ida directory [+ subdir] and returns the first match.

@param filename (string): name of file to search
@param subdir (string): if specified, the file is looked for in the specified
               subdirectory of the ida directory first (see IDA subdirectories)
@return: nullptr if not found, otherwise a pointer to full file name.
```

IDAPython function idaapi.graph_trace quick reference

```
graph_trace() -> bool
Show the trace callgraph.
```

IDAPython function idaapi.guess_func_cc quick reference

```
guess_func_cc(fti, npurged, cc_flags) -> cm_t
Use func_type_data_t::guess_cc()

@param fti: (C++: const func_type_data_t &) func_type_data_t const &
@param npurged (integer):
@param cc_flags (integer):
```

IDAPython function idaapi.guess_tinfo quick reference

```
guess_tinfo(tif, id) -> int
Generate a type information about the id from the disassembly. id can be a
structure/union/enum id or an address.

@param tif: (C++: tinfo_t *)
@param id (integer):
@return: one of Guess tinfo codes
```

IDAPython function idaapi.handle_debug_event quick reference

```
handle_debug_event(ev, rqflags) -> int

@param ev: debug_event_t const *
@param rqflags: int
```

IDAPython function idaapi.handle_fixups_in_macro quick reference

```
handle_fixups_in_macro(ri, ea, other, macro_reft_and_flags) -> bool
Handle two fixups in a macro. We often combine two instruction that load parts
of a value into one macro instruction. For example:
ARM:   ADRP  X0, #var@PAGE
ADD   X0, X0, #var@PAGEOFF  --> ADRL X0, var
MIPS:  lui   $v0, %hi(var)
addiu $v0, $v0, %lo(var)    --> la   $v0, var
When applying the fixups that fall inside such a macro, we should convert them
to one refinfo. This function does exactly that. It should be called from the
apply() callback of a custom fixup.
```

```
@param ri: (C++: refinfo_t *)
@param ea (integer):
@param other: (C++: fixup_type_t)
@param macro_reft_and_flags (integer):
@return: success ('false' means that RI was not changed)
```

IDAPython function idaapi.has_aflag_linnum quick reference

```
has_aflag_linnum(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.has_aflag_lname quick reference

```
has_aflag_lname(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.has_aflag_ti quick reference

```
has_aflag_ti(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.has_aflag_ti0 quick reference

```
has_aflag_ti0(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.has_aflag_ti1 quick reference

```
has_aflag_ti1(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.has_any_name quick reference

```
has_any_name(F) -> bool
Does the current byte have any name?
```

```
@param F (integer):
```

IDAPython function idaapi.has_auto_name quick reference

```
has_auto_name(F) -> bool
Does the current byte have auto-generated (no special prefix) name?

@param F (integer):
```

IDAPython function idaapi.has_cached_cfunc quick reference

```
has_cached_cfunc(ea) -> bool
Do we have a cached decompilation result for 'ea'?

@param ea (integer):
```

IDAPython function idaapi.has_cf_chg quick reference

```
has_cf_chg(feature, opnum) -> bool
Does an instruction with the specified feature modify the i-th operand?

@param feature (integer):
@param opnum (integer):
```

IDAPython function idaapi.has_cf_use quick reference

```
has_cf_use(feature, opnum) -> bool
Does an instruction with the specified feature use a value of the i-th operand?

@param feature (integer):
@param opnum (integer):
```

IDAPython function idaapi.has_cmt quick reference

```
has_cmt(F) -> bool
Does the current byte have an indented comment?

@param F (integer):
```

IDAPython function idaapi.has_dummy_name quick reference

```
has_dummy_name(F) -> bool
Does the current byte have dummy (auto-generated, with special prefix) name?

@param F (integer):
```

IDAPython function idaapi.has_external_refs quick reference

```
has_external_refs(pfn, ea) -> bool
Does 'ea' have references from outside of 'pfn'?

@param pfn (idaapi.func_t):
@param ea (integer):
```

IDAPython function idaapi.has_extra_cmts quick reference

```
has_extra_cmts(F) -> bool
Does the current byte have additional anterior or posterior lines?

@param F (integer):
```

IDAPython function idaapi.has_immd quick reference

```
has_immd(F) -> bool
Has immediate value?

@param F (integer):
```

IDAPython function idaapi.has_insn_feature quick reference

```
has_insn_feature(icode, bit) -> bool
Does the specified instruction have the specified feature?

@param icode: (C++: uint16)
@param bit (integer):
```

IDAPython function idaapi.has_lname quick reference

```
has_lname(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.has_mcode_seloff quick reference

```
has_mcode_seloff(op) -> bool

@param op: enum mcode_t
```

IDAPython function idaapi.has_name quick reference

```
has_name(F) -> bool
Does the current byte have non-trivial (non-dummy) name?

@param F (integer):
```

IDAPython function idaapi.has_regvar quick reference

```
has_regvar(pfn, ea) -> bool
Is there a register variable definition?

@param pfn (idaapi.func_t): function in question
@param ea (integer): current address
```

IDAPython function idaapi.has_ti quick reference

```
has_ti(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.has_ti0 quick reference

```
has_ti0(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.has_ti1 quick reference

```
has_ti1(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.has_user_name quick reference

```
has_user_name(F) -> bool
```

```
Does the current byte have user-specified name?

@param F (integer):
```

IDAPython function idaapi.has_value quick reference

```
has_value(F) -> bool
Do flags contain byte value?

@param F (integer):
```

IDAPython function idaapi.has_xref quick reference

```
has_xref(F) -> bool
Does the current byte have cross-references to it?

@param F (integer):
```

IDAPython function idaapi.hex_flag quick reference

```
hex_flag() -> flags64_t
Get number flag of the base, regardless of current processor - better to use
num_flag()
```

IDAPython function idaapi.hexrays_alloc quick reference

```
hexrays_alloc(size) -> void *

@param size: size_t
```

IDAPython function idaapi.hexrays_free quick reference

```
hexrays_free(ptr)

@param ptr
```

IDAPython function idaapi.hide_all_bpts quick reference

```
hide_all_bpts() -> int
```

IDAPython function idaapi.hide_border quick reference

```
hide_border(ea)

@param ea: ea_t
```

IDAPython function idaapi.hide_item quick reference

```
hide_item(ea)

@param ea: ea_t
```

IDAPython function idaapi.hide_name quick reference

```
hide_name(ea)
Remove name from the list of names

@param ea (integer): address of the name
```

IDAPython function idaapi.hide_wait_box quick reference

```
hide_wait_box()
Hide the "Please wait dialog box".
```

IDAPython function idaapi.idadir quick reference

```
idadir(subdir) -> char const *
Get IDA directory (if subdir==nullptr) or the specified subdirectory (see IDA
subdirectories)

@param subdir (string): char const *
```

IDAPython function idaapi.idainfo_big_arg_align quick reference

```
inf_big_arg_align(cc) -> bool

@param cc: cm_t

inf_big_arg_align() -> bool
```

IDAPython function idaapi.idainfo__comment__get quick reference

```
inf_get_cmt_indent() -> uchar
```

IDAPython function idaapi.idainfo__comment__set quick reference

```
inf_set_cmt_indent(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.idainfo__gen__lzero quick reference

```
inf_gen_lzero() -> bool
```

IDAPython function idaapi.idainfo__gen__null quick reference

```
inf_gen_null() -> bool
```

IDAPython function idaapi.idainfo__gen__tryblks quick reference

```
inf_gen_tryblks() -> bool
```

IDAPython function idaapi.idainfo__get__demname__form quick reference

```
inf_get_demname_form() -> uchar
Get DEMNAM_MASK bits of #demnames.
```

IDAPython function idaapi.idainfo__get__pack__mode quick reference

```
inf_get_pack_mode() -> int
```

IDAPython function idaapi.idainfo__is__64bit quick reference

```
inf_is_64bit() -> bool
```

IDAPython function idaapi.idainfo__is__auto__enabled quick reference

```
inf_is_auto_enabled() -> bool
```

IDAPython function idaapi.idainfo_is_be quick reference

```
inf_is_be() -> bool
```

IDAPython function idaapi.idainfo_is_dll quick reference

```
inf_is_dll() -> bool
```

IDAPython function idaapi.idainfo_is_flat_off32 quick reference

```
inf_is_flat_off32() -> bool
```

IDAPython function idaapi.idainfo_is_graph_view quick reference

```
inf_is_graph_view() -> bool
```

IDAPython function idaapi.idainfo_is_hard_float quick reference

```
inf_is_hard_float() -> bool
```

IDAPython function idaapi.idainfo_is_kernel_mode quick reference

```
inf_is_kernel_mode() -> bool
```

IDAPython function idaapi.idainfo_is_mem_aligned4 quick reference

```
inf_is_mem_aligned4() -> bool
```

IDAPython function idaapi.idainfo_is_snapshot quick reference

```
inf_is_snapshot() -> bool
```

IDAPython function idaapi.idainfo_is_wide_high_byte_first quick reference

```
inf_is_wide_high_byte_first() -> bool
```

IDAPython function idaapi.idainfo__like__binary quick reference

```
inf_like_binary() -> bool
```

IDAPython function idaapi.idainfo__line__pref__with__seg quick reference

```
inf_line_pref_with_seg() -> bool
```

IDAPython function idaapi.idainfo__loading__idc quick reference

```
inf_loading_idc() -> bool
```

IDAPython function idaapi.idainfo__map__stkargs quick reference

```
inf_map_stkargs() -> bool
```

IDAPython function idaapi.idainfo__pack__stkargs quick reference

```
inf_pack_stkargs(cc) -> bool

@param cc: cm_t

inf_pack_stkargs() -> bool
```

IDAPython function idaapi.idainfo__readonly__idb quick reference

```
inf_readonly_idb() -> bool
```

IDAPython function idaapi.idainfo__set__64bit quick reference

```
inf_set_64bit(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.idainfo__set__auto__enabled quick reference

```
inf_set_auto_enabled(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.idainfo_set_be quick reference

```
inf_set_be(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.idainfo_set_gen_lzero quick reference

```
inf_set_gen_lzero(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.idainfo_set_gen_null quick reference

```
inf_set_gen_null(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.idainfo_set_gen_tryblks quick reference

```
inf_set_gen_tryblks(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.idainfo_set_graph_view quick reference

```
inf_set_graph_view(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.idainfo_set_line_pref_with_seg quick reference

```
inf_set_line_pref_with_seg(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.idainfo_set_pack_mode quick reference

```
inf_set_pack_mode(pack_mode) -> int

@param pack_mode: int
```

IDAPython function idaapi.idainfo_set_show_auto quick reference

```
inf_set_show_auto(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.idainfo_set_show_line_pref quick reference

```
inf_set_show_line_pref(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.idainfo_set_show_void quick reference

```
inf_set_show_void(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.idainfo_set_wide_high_byte_first quick reference

```
inf_set_wide_high_byte_first(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.idainfo_show_auto quick reference

```
inf_show_auto() -> bool
```

IDAPython function idaapi.idainfo_show_line_pref quick reference

```
inf_show_line_pref() -> bool
```

IDAPython function idaapi.idainfo_show_void quick reference

```
inf_show_void() -> bool
```

IDAPython function idaapi.idainfo__stack__ldbl quick reference

```
inf_stack_ldbl() -> bool
```

IDAPython function idaapi.idainfo__stack__varargs quick reference

```
inf_stack_varargs() -> bool
```

IDAPython function idaapi.idainfo__use__allasm quick reference

```
inf_use_allasm() -> bool
```

IDAPython function idaapi.idainfo__use__gcc__layout quick reference

```
inf_use_gcc_layout() -> bool
```

IDAPython function idaapi.idc__get__local__type quick reference

```
idc_get_local_type(ordinal, flags) -> str

@param ordinal: int
@param flags: int
```

IDAPython function idaapi.idc__get__local__type__name quick reference

```
idc_get_local_type_name(ordinal) -> str

@param ordinal: int
```

IDAPython function idaapi.idc__get__local__type__raw quick reference

```
idc_get_local_type_raw(ordinal) -> (bytes, bytes)

@param ordinal: int
```

IDAPython function idaapi.idc__get__type quick reference

```
idc_get_type(ea) -> str

@param ea: ea_t
```

IDAPython function idaapi.idc_get_type_raw quick reference

```
idc_get_type_raw(ea) -> PyObject *

@param ea: ea_t
```

IDAPython function idaapi.idc_guess_type quick reference

```
idc_guess_type(ea) -> str

@param ea: ea_t
```

IDAPython function idaapi.idc_parse_decl quick reference

```
idc_parse_decl(ti, decl, flags) -> (str, bytes, bytes) or None

@param ti: til_t *
@param decl: char const *
@param flags: int
```

IDAPython function idaapi.idc_parse_types quick reference

```
idc_parse_types(input, flags) -> int

@param input: char const *
@param flags: int
```

IDAPython function idaapi.idc_print_type quick reference

```
idc_print_type(type, fields, name, flags) -> str

@param type: type_t const *
@param fields: p_list const *
@param name: char const *
@param flags: int
```

IDAPython function idaapi.idc_set_local_type quick reference

```
idc_set_local_type(ordinal, dcl, flags) -> int

@param ordinal: int
@param dcl: char const *
@param flags: int
```

IDAPython function idaapi.idcv_float quick reference

```
idcv_float(v) -> error_t
Convert IDC variable to a floating point.

@param v (idaapi.idc_value_t):
```

IDAPython function idaapi.idcv_int64 quick reference

```
idcv_int64(v) -> error_t
Convert IDC variable to a 64bit number.

@param v (idaapi.idc_value_t):
@return: v = 0 if impossible to convert to int64
```

IDAPython function idaapi.idcv_long quick reference

```
idcv_long(v) -> error_t
Convert IDC variable to a long (32/64bit) number.

@param v (idaapi.idc_value_t):
@return: v = 0 if impossible to convert to long
```

IDAPython function idaapi.idcv_num quick reference

```
idcv_num(v) -> error_t
Convert IDC variable to a long number.

@param v (idaapi.idc_value_t):
@return: * v = 0 if IDC variable = "false" string
* v = 1 if IDC variable = "true" string
* v = number if IDC variable is number or string containing a number
* eTypeConflict if IDC variable = empty string
```

IDAPython function idaapi.idcv_object quick reference

```
idcv_object(v, icls=None) -> error_t
Create an IDC object. The original value of 'v' is discarded (freed).

@param v (idaapi.idc_value_t): variable to hold the object. any previous value will be d
@param icls: (C++: const idc_class_t *) ptr to the desired class. nullptr means "object'
```

```
            must be returned by add_idc_class() or find_idc_class()
@return: always eOk
```

IDAPython function idaapi.idcv_string quick reference

```
idcv_string(v) -> error_t
Convert IDC variable to a text string.

@param v (idaapi.idc_value_t):
```

IDAPython function idaapi.import_type quick reference

```
import_type(til, idx, name, flags=0) -> tid_t
Copy a named type from til to idb.

@param til (idaapi.til_t): type library
@param idx (integer): the position of the new type in the list of types (structures or
           enums). -1 means at the end of the list
@param name (string): the type name
@param flags (integer): combination of Import type flags
@return: BADNODE on error
```

IDAPython function idaapi.inf_abi_set_by_user quick reference

```
inf_abi_set_by_user() -> bool
```

IDAPython function idaapi.inf_allow_non_matched_ops quick reference

```
inf_allow_non_matched_ops() -> bool
```

IDAPython function idaapi.inf_allow_sigmulti quick reference

```
inf_allow_sigmulti() -> bool
```

IDAPython function idaapi.inf_append_sigcmt quick reference

```
inf_append_sigcmt() -> bool
```

IDAPython function idaapi.inf__big__arg__align quick reference

```
inf_big_arg_align() -> bool
inf_big_arg_align(cc) -> bool

@param cc: cm_t
```

IDAPython function idaapi.inf__check__manual__ops quick reference

```
inf_check_manual_ops() -> bool
```

IDAPython function idaapi.inf__check__unicode__strlits quick reference

```
inf_check_unicode_strlits() -> bool
```

IDAPython function idaapi.inf__coagulate__code quick reference

```
inf_coagulate_code() -> bool
```

IDAPython function idaapi.inf__coagulate__data quick reference

```
inf_coagulate_data() -> bool
```

IDAPython function idaapi.inf__compress__idb quick reference

```
inf_compress_idb() -> bool
```

IDAPython function idaapi.inf__create__all__xrefs quick reference

```
inf_create_all_xrefs() -> bool
```

IDAPython function idaapi.inf__create__func__from__call quick reference

```
inf_create_func_from_call() -> bool
```

IDAPython function idaapi.inf__create__func__from__ptr quick reference

```
inf_create_func_from_ptr() -> bool
```

IDAPython function idaapi.inf_create_func_tails quick reference

```
inf_create_func_tails() -> bool
```

IDAPython function idaapi.inf_create_jump_tables quick reference

```
inf_create_jump_tables() -> bool
```

IDAPython function idaapi.inf_create_off_on_dref quick reference

```
inf_create_off_on_dref() -> bool
```

IDAPython function idaapi.inf_create_off_using_fixup quick reference

```
inf_create_off_using_fixup() -> bool
```

IDAPython function idaapi.inf_create_strlit_on_xref quick reference

```
inf_create_strlit_on_xref() -> bool
```

IDAPython function idaapi.inf_data_offset quick reference

```
inf_data_offset() -> bool
```

IDAPython function idaapi.inf_dbg_no_store_path quick reference

```
inf_dbg_no_store_path() -> bool
```

IDAPython function idaapi.inf_decode_fpp quick reference

```
inf_decode_fpp() -> bool
```

IDAPython function idaapi.inf_del_no_xref_insns quick reference

```
inf_del_no_xref_insns() -> bool
```

IDAPython function idaapi.inf_final_pass quick reference

```
inf_final_pass() -> bool
```

IDAPython function idaapi.inf_full_sp_ana quick reference

```
inf_full_sp_ana() -> bool
```

IDAPython function idaapi.inf_gen_assume quick reference

```
inf_gen_assume() -> bool
```

IDAPython function idaapi.inf_gen_lzero quick reference

```
inf_gen_lzero() -> bool
```

IDAPython function idaapi.inf_gen_null quick reference

```
inf_gen_null() -> bool
```

IDAPython function idaapi.inf_gen_org quick reference

```
inf_gen_org() -> bool
```

IDAPython function idaapi.inf_gen_tryblks quick reference

```
inf_gen_tryblks() -> bool
```

IDAPython function idaapi.inf_get_abibits quick reference

```
inf_get_abibits() -> uint32
```

IDAPython function idaapi.inf_get_af quick reference

```
inf_get_af() -> uint32
```

IDAPython function idaapi.inf_get_af2 quick reference

```
inf_get_af2() -> uint32
```

IDAPython function idaapi.inf_get_af2_low quick reference

```
inf_get_af2_low() -> ushort
Get/set low 16bit half of inf.af2.
```

IDAPython function idaapi.inf_get_af_high quick reference

```
inf_get_af_high() -> ushort
```

IDAPython function idaapi.inf_get_af_low quick reference

```
inf_get_af_low() -> ushort
Get/set low/high 16bit halves of inf.af.
```

IDAPython function idaapi.inf_get_app_bitness quick reference

```
inf_get_app_bitness() -> uint
```

IDAPython function idaapi.inf_get_appcall_options quick reference

```
inf_get_appcall_options() -> uint32
```

IDAPython function idaapi.inf_get_apptype quick reference

```
inf_get_apptype() -> ushort
```

IDAPython function idaapi.inf_get_asmtype quick reference

```
inf_get_asmtype() -> uchar
```

IDAPython function idaapi.inf_get_baseaddr quick reference

```
inf_get_baseaddr() -> uval_t
```

IDAPython function idaapi.inf_get_bin_prefix_size quick reference

```
inf_get_bin_prefix_size() -> short
```

IDAPython function idaapi.inf_get_cc quick reference

```
inf_get_cc(out) -> bool

@param out: compiler_info_t *
```

IDAPython function idaapi.inf_get_cc_cm quick reference

```
inf_get_cc_cm() -> cm_t
```

IDAPython function idaapi.inf_get_cc_defalign quick reference

```
inf_get_cc_defalign() -> uchar
```

IDAPython function idaapi.inf_get_cc_id quick reference

```
inf_get_cc_id() -> comp_t
```

IDAPython function idaapi.inf_get_cc_size_b quick reference

```
inf_get_cc_size_b() -> uchar
```

IDAPython function idaapi.inf_get_cc_size_e quick reference

```
inf_get_cc_size_e() -> uchar
```

IDAPython function idaapi.inf_get_cc_size_i quick reference

```
inf_get_cc_size_i() -> uchar
```

IDAPython function idaapi.inf_get_cc_size_l quick reference

```
inf_get_cc_size_l() -> uchar
```

IDAPython function idaapi.inf_get_cc_size_ldbl quick reference

```
inf_get_cc_size_ldbl() -> uchar
```

IDAPython function idaapi.inf_get_cc_size_ll quick reference

```
inf_get_cc_size_ll() -> uchar
```

IDAPython function idaapi.inf_get_cc_size_s quick reference

```
inf_get_cc_size_s() -> uchar
```

IDAPython function idaapi.inf_get_cmt_indent quick reference

```
inf_get_cmt_indent() -> uchar
```

IDAPython function idaapi.inf_get_cmtflg quick reference

```
inf_get_cmtflg() -> uchar
```

IDAPython function idaapi.inf_get_comment quick reference

```
inf_get_cmt_indent() -> uchar
```

IDAPython function idaapi.inf_get_database_change_count quick reference

```
inf_get_database_change_count() -> uint32
```

IDAPython function idaapi.inf_get_datatypes quick reference

```
inf_get_datatypes() -> uval_t
```

IDAPython function idaapi.inf_get_demname_form quick reference

```
inf_get_demname_form() -> uchar
Get DEMNAM_MASK bits of #demnames.
```

IDAPython function idaapi.inf_get_demnames quick reference

```
inf_get_demnames() -> uchar
```

IDAPython function idaapi.inf_get_filetype quick reference

```
inf_get_filetype() -> filetype_t
```

IDAPython function idaapi.inf_get_genflags quick reference

```
inf_get_genflags() -> ushort
```

IDAPython function idaapi.inf_get_highoff quick reference

```
inf_get_highoff() -> ea_t
```

IDAPython function idaapi.inf_get_indent quick reference

```
inf_get_indent() -> uchar
```

IDAPython function idaapi.inf_get_lenxref quick reference

```
inf_get_lenxref() -> ushort
```

IDAPython function idaapi.inf_get_lflags quick reference

```
inf_get_lflags() -> uint32
```

IDAPython function idaapi.inf_get_limiter quick reference

```
inf_get_limiter() -> uchar
```

IDAPython function idaapi.inf_get_listnames quick reference

```
inf_get_listnames() -> uchar
```

IDAPython function idaapi.inf_get_long_demnames quick reference

```
inf_get_long_demnames() -> uint32
```

IDAPython function idaapi.inf_get_lowoff quick reference

```
inf_get_lowoff() -> ea_t
```

IDAPython function idaapi.inf_get_main quick reference

```
inf_get_main() -> ea_t
```

IDAPython function idaapi.inf_get_margin quick reference

```
inf_get_margin() -> ushort
```

IDAPython function idaapi.inf_get_max_autoname_len quick reference

```
inf_get_max_autoname_len() -> ushort
```

IDAPython function idaapi.inf_get_max_ea quick reference

```
inf_get_max_ea() -> ea_t
```

IDAPython function idaapi.inf_get_maxref quick reference

```
inf_get_maxref() -> uval_t
```

IDAPython function idaapi.inf_get_min_ea quick reference

```
inf_get_min_ea() -> ea_t
```

IDAPython function idaapi.inf_get_nametype quick reference

```
inf_get_nametype() -> char
```

IDAPython function idaapi.inf_get_netdelta quick reference

```
inf_get_netdelta() -> sval_t
```

IDAPython function idaapi.inf_get_omax_ea quick reference

```
inf_get_omax_ea() -> ea_t
```

IDAPython function idaapi.inf_get_omin_ea quick reference

```
inf_get_omin_ea() -> ea_t
```

IDAPython function idaapi.inf_get_ostype quick reference

```
inf_get_ostype() -> ushort
```

IDAPython function idaapi.inf_get_outflags quick reference

```
inf_get_outflags() -> uint32
```

IDAPython function idaapi.inf_get_pack_mode quick reference

```
inf_get_pack_mode() -> int
```

IDAPython function idaapi.inf_get_prefflag quick reference

```
inf_get_prefflag() -> uchar
```

IDAPython function idaapi.inf_get_privrange quick reference

```
inf_get_privrange(out) -> bool

@param out: range_t *

inf_get_privrange() -> range_t
```

IDAPython function idaapi.inf_get_privrange_end_ea quick reference

```
inf_get_privrange_end_ea() -> ea_t
```

IDAPython function idaapi.inf_get_privrange_start_ea quick reference

```
inf_get_privrange_start_ea() -> ea_t
```

IDAPython function idaapi.inf_get_procname quick reference

```
inf_get_procname() -> str
```

IDAPython function idaapi.inf_get_refcmtnum quick reference

```
inf_get_refcmtnum() -> uchar
```

IDAPython function idaapi.inf_get_short_demnames quick reference

```
inf_get_short_demnames() -> uint32
```

IDAPython function idaapi.inf_get_specsegs quick reference

```
inf_get_specsegs() -> uchar
```

IDAPython function idaapi.inf_get_start_cs quick reference

```
inf_get_start_cs() -> sel_t
```

IDAPython function idaapi.inf_get_start_ea quick reference

```
inf_get_start_ea() -> ea_t
```

IDAPython function idaapi.inf_get_start_ip quick reference

```
inf_get_start_ip() -> ea_t
```

IDAPython function idaapi.inf_get_start_sp quick reference

```
inf_get_start_sp() -> ea_t
```

IDAPython function idaapi.inf_get_start_ss quick reference

```
inf_get_start_ss() -> sel_t
```

IDAPython function idaapi.inf_get_strlit_break quick reference

```
inf_get_strlit_break() -> uchar
```

IDAPython function idaapi.inf_get_strlit_flags quick reference

```
inf_get_strlit_flags() -> uchar
```

IDAPython function idaapi.inf_get_strlit_pref quick reference

```
inf_get_strlit_pref() -> str
```

IDAPython function idaapi.inf_get_strlit_sernum quick reference

```
inf_get_strlit_sernum() -> uval_t
```

IDAPython function idaapi.inf_get_strlit_zeroes quick reference

```
inf_get_strlit_zeroes() -> char
```

IDAPython function idaapi.inf_get_strtype quick reference

```
inf_get_strtype() -> int32
```

IDAPython function idaapi.inf_get_type_xrefnum quick reference

```
inf_get_type_xrefnum() -> uchar
```

IDAPython function idaapi.inf_get_version quick reference

```
inf_get_version() -> ushort
```

IDAPython function idaapi.inf_get_xrefflag quick reference

```
inf_get_xrefflag() -> uchar
```

IDAPython function idaapi.inf_get_xrefnum quick reference

```
inf_get_xrefnum() -> uchar
```

IDAPython function idaapi.inf_guess_func_type quick reference

```
inf_guess_func_type() -> bool
```

IDAPython function idaapi.inf_handle_eh quick reference

```
inf_handle_eh() -> bool
```

IDAPython function idaapi.inf_handle_rtti quick reference

```
inf_handle_rtti() -> bool
```

IDAPython function idaapi.inf_hide_comments quick reference

```
inf_hide_comments() -> bool
```

IDAPython function idaapi.inf_hide_libfuncs quick reference

```
inf_hide_libfuncs() -> bool
```

IDAPython function idaapi.inf_huge_arg_align quick reference

```
inf_huge_arg_align() -> bool
inf_huge_arg_align(cc) -> bool

@param cc: cm_t
```

IDAPython function idaapi.inf_inc_database_change_count quick reference

```
inf_inc_database_change_count(cnt=1)

@param cnt: int
```

IDAPython function idaapi.inf_is_16bit quick reference

```
inf_is_16bit() -> bool
```

IDAPython function idaapi.inf_is_32bit_exactly quick reference

```
inf_is_32bit_exactly() -> bool
```

IDAPython function idaapi.inf_is_32bit_or_higher quick reference

```
inf_is_32bit_or_higher() -> bool
```

IDAPython function idaapi.inf_is_64bit quick reference

```
inf_is_64bit() -> bool
```

IDAPython function idaapi.inf_is_auto_enabled quick reference

```
inf_is_auto_enabled() -> bool
```

IDAPython function idaapi.inf_is_be quick reference

```
inf_is_be() -> bool
```

IDAPython function idaapi.inf_is_dll quick reference

```
inf_is_dll() -> bool
```

IDAPython function idaapi.inf_is_flat_off32 quick reference

```
inf_is_flat_off32() -> bool
```

IDAPython function idaapi.inf_is_graph_view quick reference

```
inf_is_graph_view() -> bool
```

IDAPython function idaapi.inf_is_hard_float quick reference

```
inf_is_hard_float() -> bool
```

IDAPython function idaapi.inf_is_kernel_mode quick reference

```
inf_is_kernel_mode() -> bool
```

IDAPython function idaapi.inf_is_limiter_empty quick reference

```
inf_is_limiter_empty() -> bool
```

IDAPython function idaapi.inf_is_limiter_thick quick reference

```
inf_is_limiter_thick() -> bool
```

IDAPython function idaapi.inf_is_limiter_thin quick reference

```
inf_is_limiter_thin() -> bool
```

IDAPython function idaapi.inf_is_mem_aligned4 quick reference

```
inf_is_mem_aligned4() -> bool
```

IDAPython function idaapi.inf_is_snapshot quick reference

```
inf_is_snapshot() -> bool
```

IDAPython function idaapi.inf_is_wide_high_byte_first quick reference

```
inf_is_wide_high_byte_first() -> bool
```

IDAPython function idaapi.inf_like_binary quick reference

```
inf_like_binary() -> bool
```

IDAPython function idaapi.inf_line_pref_with_seg quick reference

```
inf_line_pref_with_seg() -> bool
```

IDAPython function idaapi.inf_loading_idc quick reference

```
inf_loading_idc() -> bool
```

IDAPython function idaapi.inf_macros_enabled quick reference

```
inf_macros_enabled() -> bool
```

IDAPython function idaapi.inf_map_stkargs quick reference

```
inf_map_stkargs() -> bool
```

IDAPython function idaapi.inf_mark_code quick reference

```
inf_mark_code() -> bool
```

IDAPython function idaapi.inf_no_store_user_info quick reference

```
inf_no_store_user_info() -> bool
```

IDAPython function idaapi.inf_noflow_to_data quick reference

```
inf_noflow_to_data() -> bool
```

IDAPython function idaapi.inf_noret_ana quick reference

```
inf_noret_ana() -> bool
```

IDAPython function idaapi.inf_op_offset quick reference

```
inf_op_offset() -> bool
```

IDAPython function idaapi.inf_pack_idb quick reference

```
inf_pack_idb() -> bool
```

IDAPython function idaapi.inf_pack_stkargs quick reference

```
inf_pack_stkargs() -> bool
inf_pack_stkargs(cc) -> bool

@param cc: cm_t
```

IDAPython function idaapi.inf_postinc_strlit_sernum quick reference

```
inf_postinc_strlit_sernum(cnt=1) -> uval_t

@param cnt: uval_t
```

IDAPython function idaapi.inf_prefix_show_funcoff quick reference

```
inf_prefix_show_funcoff() -> bool
```

IDAPython function idaapi.inf_prefix_show_segaddr quick reference

```
inf_prefix_show_segaddr() -> bool
```

IDAPython function idaapi.inf_prefix_show_stack quick reference

```
inf_prefix_show_stack() -> bool
```

IDAPython function idaapi.inf_prefix_truncate_opcode_bytes quick reference

```
inf_prefix_truncate_opcode_bytes() -> bool
```

IDAPython function idaapi.inf_propagate_regargs quick reference

```
inf_propagate_regargs() -> bool
```

IDAPython function idaapi.inf_propagate_stkargs quick reference

```
inf_propagate_stkargs() -> bool
```

IDAPython function idaapi.inf_readonly_idb quick reference

```
inf_readonly_idb() -> bool
```

IDAPython function idaapi.inf_rename_jumpfunc quick reference

```
inf_rename_jumpfunc() -> bool
```

IDAPython function idaapi.inf_rename_nullsub quick reference

```
inf_rename_nullsub() -> bool
```

IDAPython function idaapi.inf_set_32bit quick reference

```
inf_set_32bit(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_64bit quick reference

```
inf_set_64bit(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_abi_set_by_user quick reference

```
inf_set_abi_set_by_user(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_abibits quick reference

```
inf_set_abibits(_v) -> bool

@param _v: uint32
```

IDAPython function idaapi.inf_set_af quick reference

```
inf_set_af(_v) -> bool

@param _v: uint32
```

IDAPython function idaapi.inf_set_af2 quick reference

```
inf_set_af2(_v) -> bool

@param _v: uint32
```

IDAPython function idaapi.inf_set_af2_low quick reference

```
inf_set_af2_low(saf)

@param saf: ushort
```

IDAPython function idaapi.inf_set_af_high quick reference

```
inf_set_af_high(saf2)

@param saf2: ushort
```

IDAPython function idaapi.inf_set_af_low quick reference

```
inf_set_af_low(saf)

@param saf: ushort
```

IDAPython function idaapi.inf_set_allow_non_matched_ops quick reference

```
inf_set_allow_non_matched_ops(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_allow_sigmulti quick reference

```
inf_set_allow_sigmulti(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_app_bitness quick reference

```
inf_set_app_bitness(bitness)

@param bitness: uint
```

IDAPython function idaapi.inf_set_appcall_options quick reference

```
inf_set_appcall_options(_v) -> bool

@param _v: uint32
```

IDAPython function idaapi.inf_set_append_sigcmt quick reference

```
inf_set_append_sigcmt(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_apptype quick reference

```
inf_set_apptype(_v) -> bool
```

```
@param _v: ushort
```

IDAPython function idaapi.inf_set_asmtype quick reference

```
inf_set_asmtype(_v) -> bool
```

```
@param _v: uchar
```

IDAPython function idaapi.inf_set_auto_enabled quick reference

```
inf_set_auto_enabled(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_baseaddr quick reference

```
inf_set_baseaddr(_v) -> bool
```

```
@param _v: uval_t
```

IDAPython function idaapi.inf_set_be quick reference

```
inf_set_be(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_big_arg_align quick reference

```
inf_set_big_arg_align(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_bin_prefix_size quick reference

```
inf_set_bin_prefix_size(_v) -> bool

@param _v: short
```

IDAPython function idaapi.inf_set_cc quick reference

```
inf_set_cc(_v) -> bool

@param _v: compiler_info_t const &
```

IDAPython function idaapi.inf_set_cc_cm quick reference

```
inf_set_cc_cm(_v) -> bool

@param _v: cm_t
```

IDAPython function idaapi.inf_set_cc_defalign quick reference

```
inf_set_cc_defalign(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_cc_id quick reference

```
inf_set_cc_id(_v) -> bool

@param _v: comp_t
```

IDAPython function idaapi.inf_set_cc_size_b quick reference

```
inf_set_cc_size_b(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_cc_size_e quick reference

```
inf_set_cc_size_e(_v) -> bool
```

```
@param _v: uchar
```

IDAPython function idaapi.inf_set_cc_size_i quick reference

```
inf_set_cc_size_i(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_cc_size_l quick reference

```
inf_set_cc_size_l(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_cc_size_ldbl quick reference

```
inf_set_cc_size_ldbl(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_cc_size_ll quick reference

```
inf_set_cc_size_ll(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_cc_size_s quick reference

```
inf_set_cc_size_s(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_check_manual_ops quick reference

```
inf_set_check_manual_ops(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_check_unicode_strlits quick reference

```
inf_set_check_unicode_strlits(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_cmt_indent quick reference

```
inf_set_cmt_indent(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_cmtflg quick reference

```
inf_set_cmtflg(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_coagulate_code quick reference

```
inf_set_coagulate_code(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_coagulate_data quick reference

```
inf_set_coagulate_data(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_comment quick reference

```
inf_set_cmt_indent(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_compress_idb quick reference

```
inf_set_compress_idb(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_create_all_xrefs quick reference

```
inf_set_create_all_xrefs(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_create_func_from_call quick reference

```
inf_set_create_func_from_call(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_create_func_from_ptr quick reference

```
inf_set_create_func_from_ptr(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_create_func_tails quick reference

```
inf_set_create_func_tails(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_create_jump_tables quick reference

```
inf_set_create_jump_tables(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_create_off_on_dref quick reference

```
inf_set_create_off_on_dref(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_create_off_using_fixup quick reference

```
inf_set_create_off_using_fixup(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_create_strlit_on_xref quick reference

```
inf_set_create_strlit_on_xref(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_data_offset quick reference

```
inf_set_data_offset(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_database_change_count quick reference

```
inf_set_database_change_count(_v) -> bool

@param _v: uint32
```

IDAPython function idaapi.inf_set_datatypes quick reference

```
inf_set_datatypes(_v) -> bool

@param _v: uval_t
```

IDAPython function idaapi.inf_set_dbg_no_store_path quick reference

```
inf_set_dbg_no_store_path(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_decode_fpp quick reference

```
inf_set_decode_fpp(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_del_no_xref_insns quick reference

```
inf_set_del_no_xref_insns(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_demnames quick reference

```
inf_set_demnames(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_dll quick reference

```
inf_set_dll(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_filetype quick reference

```
inf_set_filetype(_v) -> bool

@param _v: enum filetype_t
```

IDAPython function idaapi.inf_set_final_pass quick reference

```
inf_set_final_pass(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_flat_off32 quick reference

```
inf_set_flat_off32(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_full_sp_ana quick reference

```
inf_set_full_sp_ana(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_gen_assume quick reference

```
inf_set_gen_assume(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_gen_lzero quick reference

```
inf_set_gen_lzero(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_gen_null quick reference

```
inf_set_gen_null(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_gen_org quick reference

```
inf_set_gen_org(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_gen_tryblks quick reference

```
inf_set_gen_tryblks(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_genflags quick reference

```
inf_set_genflags(_v) -> bool
```

```
@param _v: ushort
```

IDAPython function idaapi.inf_set_graph_view quick reference

```
inf_set_graph_view(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_guess_func_type quick reference

```
inf_set_guess_func_type(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_handle_eh quick reference

```
inf_set_handle_eh(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_handle_rtti quick reference

```
inf_set_handle_rtti(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_hard_float quick reference

```
inf_set_hard_float(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_hide_comments quick reference

```
inf_set_hide_comments(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_hide_libfuncs quick reference

```
inf_set_hide_libfuncs(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_highoff quick reference

```
inf_set_highoff(_v) -> bool

@param _v: ea_t
```

IDAPython function idaapi.inf_set_huge_arg_align quick reference

```
inf_set_huge_arg_align(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_indent quick reference

```
inf_set_indent(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_kernel_mode quick reference

```
inf_set_kernel_mode(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_lenxref quick reference

```
inf_set_lenxref(_v) -> bool

@param _v: ushort
```

IDAPython function idaapi.inf_set_lflags quick reference

```
inf_set_lflags(_v) -> bool

@param _v: uint32
```

IDAPython function idaapi.inf_set_limiter quick reference

```
inf_set_limiter(_v) -> bool
```

```
@param _v: uchar
```

IDAPython function idaapi.inf_set_limiter_empty quick reference

```
inf_set_limiter_empty(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_limiter_thick quick reference

```
inf_set_limiter_thick(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_limiter_thin quick reference

```
inf_set_limiter_thin(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_line_pref_with_seg quick reference

```
inf_set_line_pref_with_seg(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_listnames quick reference

```
inf_set_listnames(_v) -> bool
```

```
@param _v: uchar
```

IDAPython function idaapi.inf_set_loading_idc quick reference

```
inf_set_loading_idc(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_long_demnames quick reference

```
inf_set_long_demnames(_v) -> bool

@param _v: uint32
```

IDAPython function idaapi.inf_set_lowoff quick reference

```
inf_set_lowoff(_v) -> bool

@param _v: ea_t
```

IDAPython function idaapi.inf_set_macros_enabled quick reference

```
inf_set_macros_enabled(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_main quick reference

```
inf_set_main(_v) -> bool

@param _v: ea_t
```

IDAPython function idaapi.inf_set_map_stkargs quick reference

```
inf_set_map_stkargs(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_margin quick reference

```
inf_set_margin(_v) -> bool

@param _v: ushort
```

IDAPython function idaapi.inf_set_mark_code quick reference

```
inf_set_mark_code(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_max_autoname_len quick reference

```
inf_set_max_autoname_len(_v) -> bool

@param _v: ushort
```

IDAPython function idaapi.inf_set_max_ea quick reference

```
inf_set_max_ea(_v) -> bool

@param _v: ea_t
```

IDAPython function idaapi.inf_set_maxref quick reference

```
inf_set_maxref(_v) -> bool

@param _v: uval_t
```

IDAPython function idaapi.inf_set_mem_aligned4 quick reference

```
inf_set_mem_aligned4(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_min_ea quick reference

```
inf_set_min_ea(_v) -> bool

@param _v: ea_t
```

IDAPython function idaapi.inf_set_nametype quick reference

```
inf_set_nametype(_v) -> bool

@param _v: char
```

IDAPython function idaapi.inf_set_netdelta quick reference

```
inf_set_netdelta(_v) -> bool

@param _v: sval_t
```

IDAPython function idaapi.inf_set_no_store_user_info quick reference

```
inf_set_no_store_user_info(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_noflow_to_data quick reference

```
inf_set_noflow_to_data(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_noret_ana quick reference

```
inf_set_noret_ana(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_omax_ea quick reference

```
inf_set_omax_ea(_v) -> bool

@param _v: ea_t
```

IDAPython function idaapi.inf_set_omin_ea quick reference

```
inf_set_omin_ea(_v) -> bool

@param _v: ea_t
```

IDAPython function idaapi.inf_set_op_offset quick reference

```
inf_set_op_offset(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_ostype quick reference

```
inf_set_ostype(_v) -> bool

@param _v: ushort
```

IDAPython function idaapi.inf_set_outflags quick reference

```
inf_set_outflags(_v) -> bool

@param _v: uint32
```

IDAPython function idaapi.inf_set_pack_idb quick reference

```
inf_set_pack_idb(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_pack_mode quick reference

```
inf_set_pack_mode(pack_mode) -> int

@param pack_mode: int
```

IDAPython function idaapi.inf_set_pack_stkargs quick reference

```
inf_set_pack_stkargs(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_prefflag quick reference

```
inf_set_prefflag(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_prefix_show_funcoff quick reference

```
inf_set_prefix_show_funcoff(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_prefix_show_segaddr quick reference

```
inf_set_prefix_show_segaddr(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_prefix_show_stack quick reference

```
inf_set_prefix_show_stack(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_prefix_truncate_opcode_bytes quick reference

```
inf_set_prefix_truncate_opcode_bytes(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_privrange quick reference

```
inf_set_privrange(_v) -> bool

@param _v: range_t const &
```

IDAPython function idaapi.inf_set_privrange_end_ea quick reference

```
inf_set_privrange_end_ea(_v) -> bool

@param _v: ea_t
```

IDAPython function idaapi.inf_set_privrange_start_ea quick reference

```
inf_set_privrange_start_ea(_v) -> bool
```

```
@param _v: ea_t
```

IDAPython function idaapi.inf_set_procname quick reference

```
inf_set_procname(_v, len=size_t(-1)) -> bool

@param _v: char const *
@param len: size_t
```

IDAPython function idaapi.inf_set_propagate_regargs quick reference

```
inf_set_propagate_regargs(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_propagate_stkargs quick reference

```
inf_set_propagate_stkargs(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_readonly_idb quick reference

```
inf_set_readonly_idb(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_refcmtnum quick reference

```
inf_set_refcmtnum(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_rename_jumpfunc quick reference

```
inf_set_rename_jumpfunc(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_rename_nullsub quick reference

```
inf_set_rename_nullsub(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_short_demnames quick reference

```
inf_set_short_demnames(_v) -> bool

@param _v: uint32
```

IDAPython function idaapi.inf_set_should_create_stkvars quick reference

```
inf_set_should_create_stkvars(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_should_trace_sp quick reference

```
inf_set_should_trace_sp(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_show_all_comments quick reference

```
inf_set_show_all_comments(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_show_auto quick reference

```
inf_set_show_auto(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_show_hidden_funcs quick reference

```
inf_set_show_hidden_funcs(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_show_hidden_insns quick reference

```
inf_set_show_hidden_insns(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_show_hidden_segms quick reference

```
inf_set_show_hidden_segms(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_show_line_pref quick reference

```
inf_set_show_line_pref(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_show_repeatables quick reference

```
inf_set_show_repeatables(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_show_src_linnum quick reference

```
inf_set_show_src_linnum(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_show_void quick reference

```
inf_set_show_void(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_show_xref_fncoff quick reference

```
inf_set_show_xref_fncoff(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_show_xref_seg quick reference

```
inf_set_show_xref_seg(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_show_xref_tmarks quick reference

```
inf_set_show_xref_tmarks(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_show_xref_val quick reference

```
inf_set_show_xref_val(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_snapshot quick reference

```
inf_set_snapshot(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_specsegs quick reference

```
inf_set_specsegs(_v) -> bool
```

```
@param _v: uchar
```

IDAPython function idaapi.inf_set_stack_ldbl quick reference

```
inf_set_stack_ldbl(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_stack_varargs quick reference

```
inf_set_stack_varargs(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_start_cs quick reference

```
inf_set_start_cs(_v) -> bool

@param _v: sel_t
```

IDAPython function idaapi.inf_set_start_ea quick reference

```
inf_set_start_ea(_v) -> bool

@param _v: ea_t
```

IDAPython function idaapi.inf_set_start_ip quick reference

```
inf_set_start_ip(_v) -> bool

@param _v: ea_t
```

IDAPython function idaapi.inf_set_start_sp quick reference

```
inf_set_start_sp(_v) -> bool

@param _v: ea_t
```

IDAPython function idaapi.inf_set_start_ss quick reference

```
inf_set_start_ss(_v) -> bool
```

```
@param _v: sel_t
```

IDAPython function idaapi.inf_set_strlit_autocmt quick reference

```
inf_set_strlit_autocmt(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_strlit_break quick reference

```
inf_set_strlit_break(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_strlit_flags quick reference

```
inf_set_strlit_flags(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_strlit_name_bit quick reference

```
inf_set_strlit_name_bit(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_strlit_names quick reference

```
inf_set_strlit_names(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_strlit_pref quick reference

```
inf_set_strlit_pref(_v, len=size_t(-1)) -> bool

@param _v: char const *
@param len: size_t
```

IDAPython function idaapi.inf_set_strlit_savecase quick reference

```
inf_set_strlit_savecase(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_strlit_serial_names quick reference

```
inf_set_strlit_serial_names(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_strlit_sernum quick reference

```
inf_set_strlit_sernum(_v) -> bool

@param _v: uval_t
```

IDAPython function idaapi.inf_set_strlit_zeroes quick reference

```
inf_set_strlit_zeroes(_v) -> bool

@param _v: char
```

IDAPython function idaapi.inf_set_strtype quick reference

```
inf_set_strtype(_v) -> bool

@param _v: int32
```

IDAPython function idaapi.inf_set_trace_flow quick reference

```
inf_set_trace_flow(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_truncate_on_del quick reference

```
inf_set_truncate_on_del(_v=True) -> bool
```

```
@param _v: bool
```

IDAPython function idaapi.inf_set_type_xrefnum quick reference

```
inf_set_type_xrefnum(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_unicode_strlits quick reference

```
inf_set_unicode_strlits(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_use_allasm quick reference

```
inf_set_use_allasm(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_use_flirt quick reference

```
inf_set_use_flirt(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_use_gcc_layout quick reference

```
inf_set_use_gcc_layout(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_version quick reference

```
inf_set_version(_v) -> bool

@param _v: ushort
```

IDAPython function idaapi.inf_set_wide_high_byte_first quick reference

```
inf_set_wide_high_byte_first(_v=True) -> bool

@param _v: bool
```

IDAPython function idaapi.inf_set_xrefflag quick reference

```
inf_set_xrefflag(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_set_xrefnum quick reference

```
inf_set_xrefnum(_v) -> bool

@param _v: uchar
```

IDAPython function idaapi.inf_should_create_stkvars quick reference

```
inf_should_create_stkvars() -> bool
```

IDAPython function idaapi.inf_should_trace_sp quick reference

```
inf_should_trace_sp() -> bool
```

IDAPython function idaapi.inf_show_all_comments quick reference

```
inf_show_all_comments() -> bool
```

IDAPython function idaapi.inf_show_auto quick reference

```
inf_show_auto() -> bool
```

IDAPython function idaapi.inf_show_hidden_funcs quick reference

```
inf_show_hidden_funcs() -> bool
```

IDAPython function idaapi.inf_show_hidden_insns quick reference

```
inf_show_hidden_insns() -> bool
```

IDAPython function idaapi.inf_show_hidden_segms quick reference

```
inf_show_hidden_segms() -> bool
```

IDAPython function idaapi.inf_show_line_pref quick reference

```
inf_show_line_pref() -> bool
```

IDAPython function idaapi.inf_show_repeatables quick reference

```
inf_show_repeatables() -> bool
```

IDAPython function idaapi.inf_show_src_linnum quick reference

```
inf_show_src_linnum() -> bool
```

IDAPython function idaapi.inf_show_void quick reference

```
inf_show_void() -> bool
```

IDAPython function idaapi.inf_show_xref_fncoff quick reference

```
inf_show_xref_fncoff() -> bool
```

IDAPython function idaapi.inf_show_xref_seg quick reference

```
inf_show_xref_seg() -> bool
```

IDAPython function idaapi.inf_show_xref_tmarks quick reference

```
inf_show_xref_tmarks() -> bool
```

IDAPython function idaapi.inf_show_xref_val quick reference

```
inf_show_xref_val() -> bool
```

IDAPython function idaapi.inf_stack_ldbl quick reference

```
inf_stack_ldbl() -> bool
```

IDAPython function idaapi.inf_stack_varargs quick reference

```
inf_stack_varargs() -> bool
```

IDAPython function idaapi.inf_strlit_autocmt quick reference

```
inf_strlit_autocmt() -> bool
```

IDAPython function idaapi.inf_strlit_name_bit quick reference

```
inf_strlit_name_bit() -> bool
```

IDAPython function idaapi.inf_strlit_names quick reference

```
inf_strlit_names() -> bool
```

IDAPython function idaapi.inf_strlit_savecase quick reference

```
inf_strlit_savecase() -> bool
```

IDAPython function idaapi.inf_strlit_serial_names quick reference

```
inf_strlit_serial_names() -> bool
```

IDAPython function idaapi.inf_test_mode quick reference

```
inf_test_mode() -> bool
```

IDAPython function idaapi.inf_trace_flow quick reference

```
inf_trace_flow() -> bool
```

IDAPython function idaapi.inf_truncate_on_del quick reference

```
inf_truncate_on_del() -> bool
```

IDAPython function idaapi.inf__unicode__strlits quick reference

```
inf_unicode_strlits() -> bool
```

IDAPython function idaapi.inf__use__allasm quick reference

```
inf_use_allasm() -> bool
```

IDAPython function idaapi.inf__use__flirt quick reference

```
inf_use_flirt() -> bool
```

IDAPython function idaapi.inf__use__gcc__layout quick reference

```
inf_use_gcc_layout() -> bool
```

IDAPython function idaapi.info quick reference

```
info(format)

@param format: char const *
```

IDAPython function idaapi.init__hexrays__plugin quick reference

```
init_hexrays_plugin(flags=0) -> bool
Check that your plugin is compatible with hex-rays decompiler. This function
must be called before calling any other decompiler function.

@param flags (integer): reserved, must be 0
@return: true if the decompiler exists and is compatible with your plugin
```

IDAPython function idaapi.insn__add__cref quick reference

```
insn_add_cref(insn, to, opoff, type)

@param insn: an idaapi.insn_t, or an address (C++: const insn_t &)
@param to: ea_t
@param opoff: int
@param type: enum cref_t
```

IDAPython function idaapi.insn__add__dref quick reference

```
insn_add_dref(insn, to, opoff, type)

@param insn: an idaapi.insn_t, or an address (C++: const insn_t &)
@param to: ea_t
@param opoff: int
@param type: enum dref_t
```

IDAPython function idaapi.insn__add__off__drefs quick reference

```
insn_add_off_drefs(insn, x, type, outf) -> ea_t

@param insn: an idaapi.insn_t, or an address (C++: const insn_t &)
@param x: op_t const &
@param type: enum dref_t
@param outf: int
```

IDAPython function idaapi.insn__create__stkvar quick reference

```
insn_create_stkvar(insn, x, v, flags) -> bool

@param insn: an idaapi.insn_t, or an address (C++: const insn_t &)
@param x: op_t const &
@param v: adiff_t
@param flags: int
```

IDAPython function idaapi.insn__t___from__ptrval___ quick reference

```
insn_t__from_ptrval__(ptrval) -> insn_t

@param ptrval: size_t
```

IDAPython function idaapi.install__command__interpreter quick reference

```
install_command_interpreter(py_obj) -> int
Install command line interpreter (ui_install_cli)

@param py_obj: PyObject *
```

IDAPython function idaapi.install_hexrays_callback quick reference

```
Deprecated. Please use Hexrays_Hooks instead
Install handler for decompiler events.

@return: false if failed
```

IDAPython function idaapi.install_microcode_filter quick reference

```
install_microcode_filter(filter, install=True) -> bool
register/unregister non-standard microcode generator

@param filter: (C++: microcode_filter_t *) - microcode generator object
@param install (bool): - TRUE - register the object, FALSE - unregister
@return: success
```

IDAPython function idaapi.install_user_defined_prefix quick reference

```
install_user_defined_prefix(prefix_len, udp, owner) -> bool
User-defined line-prefixes are displayed just after the autogenerated line
prefixes in the disassembly listing. There is no need to call this function
explicitly. Use the user_defined_prefix_t class.

@param prefix_len (integer): prefixed length. if 0, then uninstall UDP
@param udp: (C++: struct user_defined_prefix_t *) object to generate user-defined prefi
@param owner: (C++: const void *) pointer to the plugin_t that owns UDP if non-nullptr,
              object will be uninstalled and destroyed when the plugin gets
              unloaded
```

IDAPython function idaapi.int_pointer_frompointer quick reference

```
int_pointer_frompointer(t) -> int_pointer

@param t: int *
```

IDAPython function idaapi.internal_get_sreg_base quick reference

```
internal_get_sreg_base(tid, sreg_value) -> ea_t
Get the sreg base, for the given thread.

@param tid: thid_t
@param sreg_value: int
```

```
@return: The sreg base, or BADADDR on failure.
```

IDAPython function idaapi.internal_ioctl quick reference

```
internal_ioctl(fn, buf, poutbuf, poutsize) -> int

@param fn: int
@param buf: void const *
@param poutbuf*
@param poutsize: ssize_t *
```

IDAPython function idaapi.internal_register_place_class quick reference

```
internal_register_place_class(tmplate, flags, owner, sdk_version) -> int

@param tmplate: place_t const *
@param flags: int
@param owner: plugin_t const *
@param sdk_version: int
```

IDAPython function idaapi.invalidate_dbg_state quick reference

```
invalidate_dbg_state(dbginv) -> int
Invalidate cached debugger information. \sq{Type, Synchronous function,
Notification, none (synchronous function)}

@param dbginv (integer): Debugged process invalidation options
@return: current debugger state (one of Debugged process states)
```

IDAPython function idaapi.invalidate_dbgmem_config quick reference

```
invalidate_dbgmem_config()
Invalidate the debugged process memory configuration. Call this function if the
debugged process might have changed its memory layout (allocated more memory,
for example)
```

IDAPython function idaapi.invalidate_dbgmem_contents quick reference

```
invalidate_dbgmem_contents(ea, size)
```

```
Invalidate the debugged process memory contents. Call this function each time
the process has been stopped or the process memory is modified. If ea ==
BADADDR, then the whole memory contents will be invalidated

@param ea (integer):
@param size (integer):
```

IDAPython function idaapi.is___bnot0 quick reference

```
is__bnot0(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is___bnot1 quick reference

```
is__bnot1(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is___invsign0 quick reference

```
is__invsign0(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is___invsign1 quick reference

```
is__invsign1(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_action_enabled quick reference

```
is_action_enabled(s) -> bool
Check if the given action state is one of AST_ENABLE*.

@param s: (C++: action_state_t) enum action_state_t
```

IDAPython function idaapi.is_additive quick reference

```
is_additive(op) -> bool
```

```
Is additive operator?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.is_aflag___bnot0 quick reference

```
is_aflag__bnot0(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag___bnot1 quick reference

```
is_aflag__bnot1(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag___invsign0 quick reference

```
is_aflag__invsign0(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag___invsign1 quick reference

```
is_aflag__invsign1(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_align_flow quick reference

```
is_aflag_align_flow(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_colored_item quick reference

```
is_aflag_colored_item(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_data_guessed_by_hexrays quick reference

```
is_aflag_data_guessed_by_hexrays(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_fixed_spd quick reference

```
is_aflag_fixed_spd(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_func_guessed_by_hexrays quick reference

```
is_aflag_func_guessed_by_hexrays(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_hidden_border quick reference

```
is_aflag_hidden_border(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_hidden_item quick reference

```
is_aflag_hidden_item(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_libitem quick reference

```
is_aflag_libitem(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_lzero0 quick reference

```
is_aflag_lzero0(flags) -> bool
```

```
@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_lzero1 quick reference

```
is_aflag_lzero1(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_manual_insn quick reference

```
is_aflag_manual_insn(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_noret quick reference

```
is_aflag_noret(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_notcode quick reference

```
is_aflag_notcode(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_notproc quick reference

```
is_aflag_notproc(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_public_name quick reference

```
is_aflag_public_name(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_retfp quick reference

```
is_aflag_retfp(flags) -> bool
```

```
@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_terse_struc quick reference

```
is_aflag_terse_struc(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_tilcmt quick reference

```
is_aflag_tilcmt(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_type_determined_by_hexrays quick reference

```
is_aflag_type_determined_by_hexrays(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_type_guessed_by_hexrays quick reference

```
is_aflag_type_guessed_by_hexrays(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_type_guessed_by_ida quick reference

```
is_aflag_type_guessed_by_ida(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_usersp quick reference

```
is_aflag_usersp(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_userti quick reference

```
is_aflag_userti(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_weak_name quick reference

```
is_aflag_weak_name(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_aflag_zstroff quick reference

```
is_aflag_zstroff(flags) -> bool

@param flags: aflags_t
```

IDAPython function idaapi.is_align quick reference

```
is_align(F) -> bool
FF_ALIGN

@param F (integer):
```

IDAPython function idaapi.is_align_flow quick reference

```
is_align_flow(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_align_insn quick reference

```
is_align_insn(ea) -> int
If the instruction at 'ea' looks like an alignment instruction, return its
length in bytes. Otherwise return 0.

@param ea (integer):
```

IDAPython function idaapi.is_allowed_on_small_struni quick reference

```
accepts_small_udts(op) -> bool
Is the operator allowed on small structure or union?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.is_anonymous_member_name quick reference

```
is_anonymous_member_name(name) -> bool
Is member name prefixed with "anonymous"?

@param name (string): char const *
```

IDAPython function idaapi.is_assignment quick reference

```
is_assignment(op) -> bool
Is assignment operator?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.is_attached_custom_data_format quick reference

```
is_attached_custom_data_format(dtid, dfid) -> bool
Is the custom data format attached to the custom data type?

@param dtid (integer): data type id
@param dfid (integer): data format id
@return: true or false
```

IDAPython function idaapi.is_auto_enabled quick reference

```
is_auto_enabled() -> bool
Get autoanalyzer state.
```

IDAPython function idaapi.is_autosync quick reference

```
is_autosync(name, type) -> bool
Is the specified idb type automatically synchronized?
```

```
@param name (string): char const *
@param type: type_t const *

is_autosync(name, tif) -> bool

@param name: char const *
@param tif: tinfo_t const &
```

IDAPython function idaapi.is_basic_block_end quick reference

```
is_basic_block_end(insn, call_insn_stops_block) -> bool
Is the instruction the end of a basic block?

@param insn: (C++: const insn_t &) an idaapi.insn_t, or an address (C++: const insn_t &)
@param call_insn_stops_block (bool):
```

IDAPython function idaapi.is_bblk_trace_enabled quick reference

```
is_bblk_trace_enabled() -> bool
```

IDAPython function idaapi.is_bf quick reference

```
is_bf(id) -> bool
Is enum a bitfield? (otherwise - plain enum, no bitmasks except for DEFMASK are
allowed)

@param id (integer):
```

IDAPython function idaapi.is_binary quick reference

```
is_binary(op) -> bool
Is binary operator?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.is_bitop quick reference

```
is_bitop(op) -> bool
Is bit related operator?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.is_bnot quick reference

```
is_bnot(ea, F, n) -> bool
Should we negate the operand?. asm_t::a_bnot should be defined in the idp module
in order to work with this function

@param ea (integer):
@param F (integer):
@param n (integer):
```

IDAPython function idaapi.is_bool_type quick reference

```
is_bool_type(type) -> bool
Is a boolean type?

@param type (idaapi.tinfo_t): tinfo_t const &
@return: true if the type is a boolean type
```

IDAPython function idaapi.is_break_consumer quick reference

```
is_break_consumer(op) -> bool
Does a break statement influence the specified statement code?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.is_byte quick reference

```
is_byte(F) -> bool
FF_BYTE

@param F (integer):
```

IDAPython function idaapi.is_call_insn quick reference

```
is_call_insn(insn) -> bool
Is the instruction a "call"?

@param insn: (C++: const insn_t &) an idaapi.insn_t, or an address (C++: const insn_t &)
```

IDAPython function idaapi.is_char quick reference

```
is_char(F, n) -> bool
is character constant?
```

```
@param F (integer):
@param n (integer):
```

IDAPython function idaapi.is_char0 quick reference

```
is_char0(F) -> bool
Is the first operand character constant? (example: push 'a')

@param F (integer):
```

IDAPython function idaapi.is_char1 quick reference

```
is_char1(F) -> bool
Is the second operand character constant? (example: mov al, 'a')

@param F (integer):
```

IDAPython function idaapi.is_chooser_widget quick reference

```
is_chooser_widget(t) -> bool
Does the given widget type specify a chooser widget?

@param t: (C++: twidget_type_t)
```

IDAPython function idaapi.is_code quick reference

```
is_code(F) -> bool
Does flag denote start of an instruction?

@param F (integer):
```

IDAPython function idaapi.is_code_far quick reference

```
is_code_far(cm) -> bool
Does the given model specify far code?.

@param cm: (C++: cm_t)
```

IDAPython function idaapi.is_colored_item quick reference

```
is_colored_item(ea) -> bool
```

```
@param ea: ea_t
```

IDAPython function idaapi.is_commutative quick reference

```
is_commutative(op) -> bool
Is commutative operator?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.is_comp_unsure quick reference

```
is_comp_unsure(comp) -> comp_t
See COMP_UNSURE.

@param comp: (C++: comp_t)
```

IDAPython function idaapi.is_control_tty quick reference

```
is_control_tty(fd) -> enum tty_control_t
Check if the current process is the owner of the TTY specified by 'fd'
(typically an opened descriptor to /dev/tty).

@param fd (integer):
```

IDAPython function idaapi.is_custfmt quick reference

```
is_custfmt(F, n) -> bool
is custom data format?

@param F (integer):
@param n (integer):
```

IDAPython function idaapi.is_custfmt0 quick reference

```
is_custfmt0(F) -> bool
Does the first operand use a custom data representation?

@param F (integer):
```

IDAPython function idaapi.is_custfmt1 quick reference

```
is_custfmt1(F) -> bool
Does the second operand use a custom data representation?

@param F (integer):
```

IDAPython function idaapi.is_custom quick reference

```
is_custom(F) -> bool
FF_CUSTOM

@param F (integer):
```

IDAPython function idaapi.is_cvt64 quick reference

```
is_cvt64() -> bool
is IDA converting IDB into I64?
```

IDAPython function idaapi.is_data quick reference

```
is_data(F) -> bool
Does flag denote start of data?

@param F (integer):
```

IDAPython function idaapi.is_data_far quick reference

```
is_data_far(cm) -> bool
Does the given model specify far data?.

@param cm: (C++: cm_t)
```

IDAPython function idaapi.is_data_guessed_by_hexrays quick reference

```
is_data_guessed_by_hexrays(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_database_busy quick reference

```
is_database_busy() -> bool
Check if the database is busy (e.g. performing some critical operations and
cannot be safely accessed)
```

IDAPython function idaapi.is_database_flag quick reference

```
is_database_flag(dbfl) -> bool
Get the current database flag

@param dbfl (integer): flag Database flags
@return: the state of the flag (set or cleared)
```

IDAPython function idaapi.is_debugger_busy quick reference

```
is_debugger_busy() -> bool
Is the debugger busy?. Some debuggers do not accept any commands while the
debugged application is running. For such a debugger, it is unsafe to do
anything with the database (even simple queries like get_byte may lead to
undesired consequences). Returns: true if the debugged application is running
under such a debugger
```

IDAPython function idaapi.is_debugger_memory quick reference

```
is_debugger_memory(ea) -> bool
Is the address mapped to debugger memory?

@param ea (integer):
```

IDAPython function idaapi.is_debugger_on quick reference

```
is_debugger_on() -> bool
Is the debugger currently running?
```

IDAPython function idaapi.is_defarg quick reference

```
is_defarg(F, n) -> bool
is defined?

@param F (integer):
@param n (integer):
```

IDAPython function idaapi.is_defarg0 quick reference

```
is_defarg0(F) -> bool
Is the first operand defined? Initially operand has no defined representation.

@param F (integer):
```

IDAPython function idaapi.is_defarg1 quick reference

```
is_defarg1(F) -> bool
Is the second operand defined? Initially operand has no defined representation.

@param F (integer):
```

IDAPython function idaapi.is_double quick reference

```
is_double(F) -> bool
FF_DOUBLE

@param F (integer):
```

IDAPython function idaapi.is_dummy_member_name quick reference

```
is_dummy_member_name(name) -> bool
Is member name an auto-generated name?

@param name (string): char const *
```

IDAPython function idaapi.is_dword quick reference

```
is_dword(F) -> bool
FF_DWORD

@param F (integer):
```

IDAPython function idaapi.is_ea_tryblks quick reference

```
is_ea_tryblks(ea, flags) -> bool
Check if the given address ea is part of tryblks description.

@param ea (integer): address to check
@param flags (integer): combination of flags for is_ea_tryblks()
```

IDAPython function idaapi.is_enum quick reference

```
is_enum(F, n) -> bool
is enum?

@param F (integer):
@param n (integer):
```

IDAPython function idaapi.is_enum0 quick reference

```
is_enum0(F) -> bool
Is the first operand a symbolic constant (enum member)?

@param F (integer):
```

IDAPython function idaapi.is_enum1 quick reference

```
is_enum1(F) -> bool
Is the second operand a symbolic constant (enum member)?

@param F (integer):
```

IDAPython function idaapi.is_enum_fromtil quick reference

```
is_enum_fromtil(id) -> bool
Does enum come from type library?

@param id (integer):
```

IDAPython function idaapi.is_enum_hidden quick reference

```
is_enum_hidden(id) -> bool
Is enum collapsed?

@param id (integer):
```

IDAPython function idaapi.is_filetype_like_binary quick reference

```
is_filetype_like_binary(ft) -> bool
Is unstructured input file?
```

```
@param ft: (C++: filetype_t) enum filetype_t
```

IDAPython function idaapi.is_finally_visible_func quick reference

```
is_finally_visible_func(pfn) -> bool
Is the function visible (event after considering SCF_SHHID_FUNC)?

@param pfn (idaapi.func_t):
```

IDAPython function idaapi.is_finally_visible_item quick reference

```
is_finally_visible_item(ea) -> bool
Is instruction visible?

@param ea (integer):
```

IDAPython function idaapi.is_finally_visible_segm quick reference

```
is_finally_visible_segm(s) -> bool
See SFL_HIDDEN, SCF_SHHID_SEGM.

@param s: (C++: segment_t *)
```

IDAPython function idaapi.is_fixed_spd quick reference

```
is_fixed_spd(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_fixup_custom quick reference

```
is_fixup_custom(type) -> bool
Is fixup processed by processor module?

@param type: (C++: fixup_type_t)
```

IDAPython function idaapi.is_flag_for_operand quick reference

```
is_flag_for_operand(F, typebits, n) -> bool
Check that the 64-bit flags set has the expected type for operand `n`.
```

```
@param F (integer): the flags
@param typebits: (C++: uint8) the type bits (one of `FF_N_`)
@param n (integer): the operand number
@return: success
```

IDAPython function idaapi.is_float quick reference

```
is_float(F) -> bool
FF_FLOAT

@param F (integer):
```

IDAPython function idaapi.is_float0 quick reference

```
is_float0(F) -> bool
Is the first operand a floating point number?

@param F (integer):
```

IDAPython function idaapi.is_float1 quick reference

```
is_float1(F) -> bool
Is the second operand a floating point number?

@param F (integer):
```

IDAPython function idaapi.is_floating_dtype quick reference

```
is_floating_dtype(dtype) -> bool
Is a floating type operand?

@param dtype: (C++: op_dtype_t)
```

IDAPython function idaapi.is_flow quick reference

```
is_flow(F) -> bool
Does the previous instruction exist and pass execution flow to the current byte?

@param F (integer):
```

IDAPython function idaapi.is_fltnum quick reference

```
is_fltnum(F, n) -> bool
is floating point number?

@param F (integer):
@param n (integer):
```

IDAPython function idaapi.is_forced_operand quick reference

```
is_forced_operand(ea, n) -> bool
Is operand manually defined?.

@param ea (integer): linear address
@param n (integer): 0..UA_MAXOP-1 operand number
```

IDAPython function idaapi.is_func quick reference

```
is_func(F) -> bool
Is function start?

@param F (integer):
```

IDAPython function idaapi.is_func_entry quick reference

```
is_func_entry(pfn) -> bool
Does function describe a function entry chunk?

@param pfn: (C++: const func_t *) func_t const *
```

IDAPython function idaapi.is_func_guessed_by_hexrays quick reference

```
is_func_guessed_by_hexrays(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_func_locked quick reference

```
is_func_locked(pfn) -> bool
Is the function pointer locked?

@param pfn: (C++: const func_t *) func_t const *
```

IDAPython function idaapi.is_func_tail quick reference

```
is_func_tail(pfn) -> bool
Does function describe a function tail chunk?

@param pfn: (C++: const func_t *) func_t const *
```

IDAPython function idaapi.is_func_trace_enabled quick reference

```
is_func_trace_enabled() -> bool
Get current state of functions tracing. \sq{Type, Synchronous function,
Notification, none (synchronous function)}
```

IDAPython function idaapi.is_funcarg_off quick reference

```
is_funcarg_off(pfn, frameoff) -> bool

@param pfn: func_t const *
@param frameoff: uval_t
```

IDAPython function idaapi.is_gcc quick reference

```
is_gcc() -> bool
Is the target compiler COMP_GNU?
```

IDAPython function idaapi.is_gcc32 quick reference

```
is_gcc32() -> bool
Is the target compiler 32 bit gcc?
```

IDAPython function idaapi.is_gcc64 quick reference

```
is_gcc64() -> bool
Is the target compiler 64 bit gcc?
```

IDAPython function idaapi.is_ghost_enum quick reference

```
is_ghost_enum(id) -> bool
Is a ghost copy of a local type?
```

```
@param id (integer):
```

IDAPython function idaapi.is_golang_cc quick reference

```
is_golang_cc(cc) -> bool
GO language calling convention (return value in stack)?

@param cc: (C++: cm_t)
```

IDAPython function idaapi.is_head quick reference

```
is_head(F) -> bool
Does flag denote start of instruction OR data?

@param F (integer):
```

IDAPython function idaapi.is_hidden_border quick reference

```
is_hidden_border(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_hidden_item quick reference

```
is_hidden_item(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_idaq quick reference

```
is_idaq() -> bool
Returns True or False depending if IDAPython is hosted by IDAQ
```

IDAPython function idaapi.is_idaview quick reference

```
is_idaview(v) -> bool
Is the given custom view an idaview? (ui_is_idaview)

@param v(a Widget SWIG wrapper class):
```

IDAPython function idaapi.is_ident quick reference

```
is_ident(name) -> bool
Is a valid name? (including ::MangleChars)

@param name (string): char const *
```

IDAPython function idaapi.is_ident_cp quick reference

```
is_ident_cp(cp) -> bool
Can a character appear in a name? (present in ::NameChars or ::MangleChars)

@param cp: (C++: wchar32_t)
```

IDAPython function idaapi.is_in_nlist quick reference

```
is_in_nlist(ea) -> bool
Is the name included into the name list?

@param ea (integer):
```

IDAPython function idaapi.is_indirect_jump_insn quick reference

```
is_indirect_jump_insn(insn) -> bool
Is the instruction an indirect jump?

@param insn: (C++: const insn_t &) an idaapi.insn_t, or an address (C++: const insn_t &)
```

IDAPython function idaapi.is_inplace_def quick reference

```
is_inplace_def(type) -> bool
Is struct/union/enum definition (not declaration)?

@param type (idaapi.tinfo_t): tinfo_t const &
```

IDAPython function idaapi.is_insn_trace_enabled quick reference

```
is_insn_trace_enabled() -> bool
Get current state of instruction tracing. \sq{Type, Synchronous function,
Notification, none (synchronous function)}
```

IDAPython function idaapi.is_invsign quick reference

```
is_invsign(ea, F, n) -> bool
Should sign of n-th operand inverted during output?. allowed values of n:
0-first operand, 1-other operands

@param ea (integer):
@param F (integer):
@param n (integer):
```

IDAPython function idaapi.is_kreg quick reference

```
is_kreg(r) -> bool
Is a kernel register? Kernel registers are temporary registers that can be used
freely. They may be used to store values that cross instruction or basic block
boundaries. Kernel registers do not map to regular processor registers. See also
mba_t::alloc_kreg()

@param r: (C++: mreg_t)
```

IDAPython function idaapi.is_libitem quick reference

```
is_libitem(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_loaded quick reference

```
is_loaded(ea) -> bool
Does the specified address have a byte value (is initialized?)

@param ea (integer):
```

IDAPython function idaapi.is_logical quick reference

```
is_logical(op) -> bool
Is logical operator?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.is_loop quick reference

```
is_loop(op) -> bool
```

```
Is loop statement code?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.is_lvalue quick reference

```
is_lvalue(op) -> bool
Is Lvalue operator?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.is_lzero quick reference

```
is_lzero(ea, n) -> bool
Display leading zeroes? Display leading zeroes in operands. The global switch
for the leading zeroes is in idainfo::s_genflags Note: the leading zeroes
doesn't work if for the target assembler octal numbers start with 0.

@param ea (integer): the item (insn/data) address
@param n (integer): the operand number (0-first operand, 1-other operands)
@return: success
```

IDAPython function idaapi.is_lzero0 quick reference

```
is_lzero0(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_lzero1 quick reference

```
is_lzero1(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_main_thread quick reference

```
is_main_thread() -> bool
Are we running in the main thread?
```

IDAPython function idaapi.is_manual quick reference

```
is_manual(F, n) -> bool
```

```
is forced operand? (use is_forced_operand())

@param F (integer):
@param n (integer):
```

IDAPython function idaapi.is_manual_insn quick reference

```
is_manual_insn(ea) -> bool
Is the instruction overridden?

@param ea (integer): linear address of the instruction or data item
```

IDAPython function idaapi.is_mapped quick reference

```
is_mapped(ea) -> bool
Is the specified address 'ea' present in the program?

@param ea (integer):
```

IDAPython function idaapi.is_may_access quick reference

```
is_may_access(maymust) -> bool

@param maymust: maymust_t
```

IDAPython function idaapi.is_mcode_addsub quick reference

```
is_mcode_addsub(mcode) -> bool

@param mcode: enum mcode_t
```

IDAPython function idaapi.is_mcode_call quick reference

```
is_mcode_call(mcode) -> bool

@param mcode: enum mcode_t
```

IDAPython function idaapi.is_mcode_commutative quick reference

```
is_mcode_commutative(mcode) -> bool
```

```
@param mcode: enum mcode_t
```

IDAPython function idaapi.is_mcode_convertible_to_jmp quick reference

```
is_mcode_convertible_to_jmp(mcode) -> bool

@param mcode: enum mcode_t
```

IDAPython function idaapi.is_mcode_convertible_to_set quick reference

```
is_mcode_convertible_to_set(mcode) -> bool

@param mcode: enum mcode_t
```

IDAPython function idaapi.is_mcode_divmod quick reference

```
is_mcode_divmod(op) -> bool

@param op: enum mcode_t
```

IDAPython function idaapi.is_mcode_fpu quick reference

```
is_mcode_fpu(mcode) -> bool

@param mcode: enum mcode_t
```

IDAPython function idaapi.is_mcode_j1 quick reference

```
is_mcode_j1(mcode) -> bool

@param mcode: enum mcode_t
```

IDAPython function idaapi.is_mcode_jcond quick reference

```
is_mcode_jcond(mcode) -> bool

@param mcode: enum mcode_t
```

IDAPython function idaapi.is_mcode_propagatable quick reference

```
is_mcode_propagatable(mcode) -> bool
May opcode be propagated? Such opcodes can be used in sub-instructions (nested
instructions) There is a handful of non-propagatable opcodes, like jumps, ret,
nop, etc All other regular opcodes are propagatable and may appear in a nested
instruction.

@param mcode: (C++: mcode_t) enum mcode_t
```

IDAPython function idaapi.is_mcode_set quick reference

```
is_mcode_set(mcode) -> bool

@param mcode: enum mcode_t
```

IDAPython function idaapi.is_mcode_set1 quick reference

```
is_mcode_set1(mcode) -> bool

@param mcode: enum mcode_t
```

IDAPython function idaapi.is_mcode_shift quick reference

```
is_mcode_shift(mcode) -> bool

@param mcode: enum mcode_t
```

IDAPython function idaapi.is_mcode_xdsu quick reference

```
is_mcode_xdsu(mcode) -> bool

@param mcode: enum mcode_t
```

IDAPython function idaapi.is_member_id quick reference

```
is_member_id(mid) -> bool
Is a member id?

@param mid (integer):
```

IDAPython function idaapi.is_miniidb quick reference

```
is_miniidb() -> bool
Is the database a miniidb created by the debugger?.

@return: true if the database contains no segments or only debugger segments
```

IDAPython function idaapi.is_msg_inited quick reference

```
is_msg_inited() -> bool
Can we use msg() functions?
```

IDAPython function idaapi.is_multiplicative quick reference

```
is_multiplicative(op) -> bool
Is multiplicative operator?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.is_name_defined_locally quick reference

```
is_name_defined_locally(pfn, name, ignore_name_def, ea1=BADADDR, ea2=BADADDR) -> bool
Is the name defined locally in the specified function?

@param pfn (idaapi.func_t): pointer to function
@param name (string): name to check
@param ignore_name_def: (C++: ignore_name_def_t) which names to ignore when checking
@param ea1 (integer): the starting address of the range inside the function (optional)
@param ea2 (integer): the ending address of the range inside the function (optional)
@return: true if the name has been defined
```

IDAPython function idaapi.is_nonbool_type quick reference

```
is_nonbool_type(type) -> bool
Is definitely a non-boolean type?

@param type (idaapi.tinfo_t): tinfo_t const &
@return: true if the type is a non-boolean type (non bool and well defined)
```

IDAPython function idaapi.is_noret quick reference

```
is_noret(ea) -> bool
```

```
@param ea: ea_t
```

IDAPython function idaapi.is_noret_block quick reference

```
is_noret_block(btype) -> bool
Does this block never return?

@param btype: (C++: fc_block_type_t) enum fc_block_type_t
```

IDAPython function idaapi.is_not_tail quick reference

```
is_not_tail(F) -> bool
Does flag denote tail byte?

@param F (integer):
```

IDAPython function idaapi.is_notcode quick reference

```
is_notcode(ea) -> bool
Is the address marked as not-code?

@param ea (integer):
```

IDAPython function idaapi.is_notproc quick reference

```
is_notproc(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_numop quick reference

```
is_numop(F, n) -> bool
is number (bin, oct, dec, hex)?

@param F (integer):
@param n (integer):
```

IDAPython function idaapi.is_numop0 quick reference

```
is_numop0(F) -> bool
Is the first operand a number (i.e. binary, octal, decimal or hex?)
```

```
@param F (integer):
```

IDAPython function idaapi.is_numop1 quick reference

```
is_numop1(F) -> bool
Is the second operand a number (i.e. binary, octal, decimal or hex?)

@param F (integer):
```

IDAPython function idaapi.is_off quick reference

```
is_off(F, n) -> bool
is offset?

@param F (integer):
@param n (integer):
```

IDAPython function idaapi.is_off0 quick reference

```
is_off0(F) -> bool
Is the first operand offset? (example: push offset xxx)

@param F (integer):
```

IDAPython function idaapi.is_off1 quick reference

```
is_off1(F) -> bool
Is the second operand offset? (example: mov ax, offset xxx)

@param F (integer):
```

IDAPython function idaapi.is_one_bit_mask quick reference

```
is_one_bit_mask(mask) -> bool
Is bitmask one bit?

@param mask (integer):
```

IDAPython function idaapi.is_ordinal_name quick reference

```
is_ordinal_name(name, ord=None) -> bool
```

```
Check if the name is an ordinal name. Ordinal names have the following format:
'#' + set_de(ord)

@param name (string): char const *
@param ord: (C++: uint32 *)
```

IDAPython function idaapi.is_oword quick reference

```
is_oword(F) -> bool
FF_OWORD

@param F (integer):
```

IDAPython function idaapi.is_pack_real quick reference

```
is_pack_real(F) -> bool
FF_PACKREAL

@param F (integer):
```

IDAPython function idaapi.is_paf quick reference

```
is_paf(t) -> bool
Is a pointer, array, or function type?

@param t: (C++: type_t)
```

IDAPython function idaapi.is_pascal quick reference

```
is_pascal(strtype) -> bool

@param strtype: int32
```

IDAPython function idaapi.is_place_class_ea_capable quick reference

```
is_place_class_ea_capable(id) -> bool
See get_place_class()

@param id (integer):
```

IDAPython function idaapi.is_prepost quick reference

```
is_prepost(op) -> bool
Is pre/post increment/decrement operator?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.is_problem_present quick reference

```
is_problem_present(t, ea) -> bool
Check if the specified address is present in the problem list.

@param t: (C++: problist_id_t)
@param ea (integer):
```

IDAPython function idaapi.is_ptr_or_array quick reference

```
is_ptr_or_array(t) -> bool
Is a pointer or array type?

@param t: (C++: type_t)
```

IDAPython function idaapi.is_public_name quick reference

```
is_public_name(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_purging_cc quick reference

```
is_purging_cc(cm) -> bool
Does the calling convention clean the stack arguments upon return?.
@note: this function is valid only for x86 code

@param cm: (C++: cm_t)
```

IDAPython function idaapi.is_qword quick reference

```
is_qword(F) -> bool
FF_QWORD

@param F (integer):
```

IDAPython function idaapi.is_refresh_requested quick reference

```
is_refresh_requested(mask) -> bool
Get a refresh request state

@param mask (integer): Window refresh flags
@return: the state (set or cleared)
```

IDAPython function idaapi.is_reftype_target_optional quick reference

```
is_reftype_target_optional(type) -> bool
Can the target be calculated using operand value?

@param type: (C++: reftype_t)
```

IDAPython function idaapi.is_reg_custom quick reference

```
is_reg_custom(regname) -> bool
Does a register contain a value of a custom data type? \sq{Type, Synchronous
function, Notification, none (synchronous function)}

@param regname (string): char const *
```

IDAPython function idaapi.is_reg_float quick reference

```
is_reg_float(regname) -> bool
Does a register contain a floating point value? \sq{Type, Synchronous function,
Notification, none (synchronous function)}

@param regname (string): char const *
```

IDAPython function idaapi.is_reg_integer quick reference

```
is_reg_integer(regname) -> bool
Does a register contain an integer value? \sq{Type, Synchronous function,
Notification, none (synchronous function)}

@param regname (string): char const *
```

IDAPython function idaapi.is_relational quick reference

```
is_relational(op) -> bool
```

```
Is comparison operator?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.is_request_running quick reference

```
is_request_running() -> bool
Is a request currently running?
```

IDAPython function idaapi.is_restype_enum quick reference

```
is_restype_enum(til, type) -> bool

@param til: til_t const *
@param type: type_t const *
```

IDAPython function idaapi.is_restype_struct quick reference

```
is_restype_struct(til, type) -> bool

@param til: til_t const *
@param type: type_t const *
```

IDAPython function idaapi.is_restype_struni quick reference

```
is_restype_struni(til, type) -> bool

@param til: til_t const *
@param type: type_t const *
```

IDAPython function idaapi.is_restype_void quick reference

```
is_restype_void(til, type) -> bool

@param til: til_t const *
@param type: type_t const *
```

IDAPython function idaapi.is_ret_block quick reference

```
is_ret_block(btype) -> bool
Does this block return?
```

```
@param btype: (C++: fc_block_type_t) enum fc_block_type_t
```

IDAPython function idaapi.is_ret_insn quick reference

```
is_ret_insn(insn, strict=True) -> bool
Is the instruction a "return"?

@param insn: (C++: const insn_t &) an idaapi.insn_t, or an address (C++: const insn_t &)
@param strict (bool):
```

IDAPython function idaapi.is_retfp quick reference

```
is_retfp(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_same_data_type quick reference

```
is_same_data_type(F1, F2) -> bool
Do the given flags specify the same data type?

@param F1 (integer):
@param F2 (integer):
```

IDAPython function idaapi.is_same_func quick reference

```
is_same_func(ea1, ea2) -> bool
Do two addresses belong to the same function?

@param ea1 (integer):
@param ea2 (integer):
```

IDAPython function idaapi.is_sdacl_byte quick reference

```
is_sdacl_byte(t) -> bool
Identify an sdacl byte. The first sdacl byte has the following format: 11xx000x.
The sdacl bytes are appended to udt fields. They indicate the start of type
attributes (as the tah-bytes do). The sdacl bytes are used in the udt headers
instead of the tah-byte. This is done for compatibility with old databases, they
were already using sdacl bytes in udt headers and as udt field postfixes. (see
"sdacl-typeattrs" in the type bit definitions)
```

```
@param t: (C++: type_t)
```

IDAPython function idaapi.is_seg quick reference

```
is_seg(F, n) -> bool
is segment?

@param F (integer):
@param n (integer):
```

IDAPython function idaapi.is_seg0 quick reference

```
is_seg0(F) -> bool
Is the first operand segment selector? (example: push seg seg001)

@param F (integer):
```

IDAPython function idaapi.is_seg1 quick reference

```
is_seg1(F) -> bool
Is the second operand segment selector? (example: mov dx, seg dseg)

@param F (integer):
```

IDAPython function idaapi.is_segm_locked quick reference

```
is_segm_locked(segm) -> bool
Is a segment pointer locked?

@param segm: (C++: const segment_t *) segment_t const *
```

IDAPython function idaapi.is_signed_mcode quick reference

```
is_signed_mcode(code) -> bool

@param code: enum mcode_t
```

IDAPython function idaapi.is_small_struni quick reference

```
is_small_udt(tif) -> bool
```

```
Is a small structure or union?

@param tif (idaapi.tinfo_t): tinfo_t const &
@return: true if the type is a small UDT (user defined type). Small UDTs fit
         into a register (or pair or registers) as a rule.
```

IDAPython function idaapi.is_small_udt quick reference

```
is_small_udt(tif) -> bool
Is a small structure or union?

@param tif (idaapi.tinfo_t): tinfo_t const &
@return: true if the type is a small UDT (user defined type). Small UDTs fit
         into a register (or pair or registers) as a rule.
```

IDAPython function idaapi.is_spec_ea quick reference

```
is_spec_ea(ea) -> bool
Does the address belong to a segment with a special type?. (SEG_XTRN, SEG_GRP,
SEG_ABSSYM, SEG_COMM)

@param ea (integer): linear address
```

IDAPython function idaapi.is_spec_segm quick reference

```
is_spec_segm(seg_type) -> bool
Has segment a special type?. (SEG_XTRN, SEG_GRP, SEG_ABSSYM, SEG_COMM)

@param seg_type: (C++: uchar)
```

IDAPython function idaapi.is_special_member quick reference

```
is_special_member(id) -> bool
Is a special member with the name beginning with ' '?

@param id (integer):
```

IDAPython function idaapi.is_step_trace_enabled quick reference

```
is_step_trace_enabled() -> bool
Get current state of step tracing. \sq{Type, Synchronous function, Notification,
none (synchronous function)}
```

IDAPython function idaapi.is_stkvar quick reference

```
is_stkvar(F, n) -> bool
is stack variable?

@param F (integer):
@param n (integer):
```

IDAPython function idaapi.is_stkvar0 quick reference

```
is_stkvar0(F) -> bool
Is the first operand a stack variable?

@param F (integer):
```

IDAPython function idaapi.is_stkvar1 quick reference

```
is_stkvar1(F) -> bool
Is the second operand a stack variable?

@param F (integer):
```

IDAPython function idaapi.is_strlit quick reference

```
is_strlit(F) -> bool
FF_STRLIT

@param F (integer):
```

IDAPython function idaapi.is_strlit_cp quick reference

```
is_strlit_cp(cp, specific_ranges=None) -> bool
Can a character appear in a string literal (present in ::StrlitChars) If
'specific_ranges' are specified, those will be used instead of the ones
corresponding to the current culture (only if ::StrlitChars is configured to use
the current culture)

@param cp: (C++: wchar32_t)
@param specific_ranges: (C++: const rangeset_crefvec_t *) rangeset_crefvec_t const *
```

IDAPython function idaapi.is_stroff quick reference

```
is_stroff(F, n) -> bool
is struct offset?

@param F (integer):
@param n (integer):
```

IDAPython function idaapi.is_stroff0 quick reference

```
is_stroff0(F) -> bool
Is the first operand an offset within a struct?

@param F (integer):
```

IDAPython function idaapi.is_stroff1 quick reference

```
is_stroff1(F) -> bool
Is the second operand an offset within a struct?

@param F (integer):
```

IDAPython function idaapi.is_struct quick reference

```
is_struct(F) -> bool
FF_STRUCT

@param F (integer):
```

IDAPython function idaapi.is_suspop quick reference

```
is_suspop(ea, F, n) -> bool
is suspicious operand?

@param ea (integer):
@param F (integer):
@param n (integer):
```

IDAPython function idaapi.is_swift_cc quick reference

```
is_swift_cc(cc) -> bool
Swift calling convention (arguments and return values in registers)?
```

```
@param cc: (C++: cm_t)
```

IDAPython function idaapi.is_tah_byte quick reference

```
is_tah_byte(t) -> bool
The TAH byte (type attribute header byte) denotes the start of type attributes.
(see "tah-typeattrs" in the type bit definitions)

@param t: (C++: type_t)
```

IDAPython function idaapi.is_tail quick reference

```
is_tail(F) -> bool
Does flag denote tail byte?

@param F (integer):
```

IDAPython function idaapi.is_tbyte quick reference

```
is_tbyte(F) -> bool
FF_TBYTE

@param F (integer):
```

IDAPython function idaapi.is_terse_struc quick reference

```
is_terse_struc(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_tilcmt quick reference

```
is_tilcmt(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_trusted_idb quick reference

```
is_trusted_idb() -> bool
Is the database considered as trusted?
```

IDAPython function idaapi.is_type_arithmetic quick reference

```
is_type_arithmetic(t) -> bool
Is the type an arithmetic type? (floating or integral)

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_array quick reference

```
is_type_array(t) -> bool
See BT_ARRAY.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_bitfld quick reference

```
is_type_bitfld(t) -> bool
See BT_BITFIELD.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_bool quick reference

```
is_type_bool(t) -> bool
See BTF_BOOL.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_char quick reference

```
is_type_char(t) -> bool
Does the type specify a char value? (signed or unsigned, see Basic type:
integer)

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_complex quick reference

```
is_type_complex(t) -> bool
See BT_COMPLEX.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_const quick reference

```
is_type_const(t) -> bool
See BTM_CONST.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_correct quick reference

```
is_type_correct(ptr) -> bool
Verify a type string.

@param ptr: (C++: const type_t *) type_t const *
@return: true if type string is correct
```

IDAPython function idaapi.is_type_determined_by_hexrays quick reference

```
is_type_determined_by_hexrays(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_type_double quick reference

```
is_type_double(t) -> bool
See BTF_DOUBLE.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_enum quick reference

```
is_type_enum(t) -> bool
See BTF_ENUM.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_ext_arithmetic quick reference

```
is_type_ext_arithmetic(t) -> bool
Is the type an extended arithmetic type? (arithmetic or enum)

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_ext_integral quick reference

```
is_type_ext_integral(t) -> bool
Is the type an extended integral type? (integral or enum)

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_float quick reference

```
is_type_float(t) -> bool
See BTF_FLOAT.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_floating quick reference

```
is_type_floating(t) -> bool
Is the type a floating point type?

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_func quick reference

```
is_type_func(t) -> bool
See BT_FUNC.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_guessed_by_hexrays quick reference

```
is_type_guessed_by_hexrays(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_type_guessed_by_ida quick reference

```
is_type_guessed_by_ida(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_type_int quick reference

```
is_type_int(bt) -> bool
Does the type_t specify one of the basic types in Basic type: integer?

@param bt: (C++: type_t)
```

IDAPython function idaapi.is_type_int128 quick reference

```
is_type_int128(t) -> bool
Does the type specify a 128-bit value? (signed or unsigned, see Basic type:
integer)

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_int16 quick reference

```
is_type_int16(t) -> bool
Does the type specify a 16-bit value? (signed or unsigned, see Basic type:
integer)

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_int32 quick reference

```
is_type_int32(t) -> bool
Does the type specify a 32-bit value? (signed or unsigned, see Basic type:
integer)

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_int64 quick reference

```
is_type_int64(t) -> bool
Does the type specify a 64-bit value? (signed or unsigned, see Basic type:
integer)

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_integral quick reference

```
is_type_integral(t) -> bool
Is the type an integral type (char/short/int/long/bool)?
```

```
@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_ldouble quick reference

```
is_type_ldouble(t) -> bool
See BTF_LDOUBLE.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_paf quick reference

```
is_type_paf(t) -> bool
Is the type a pointer, array, or function type?

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_partial quick reference

```
is_type_partial(t) -> bool
Identifies an unknown or void type with a known size (see Basic type: unknown &
void)

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_ptr quick reference

```
is_type_ptr(t) -> bool
See BT_PTR.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_ptr_or_array quick reference

```
is_type_ptr_or_array(t) -> bool
Is the type a pointer or array type?

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_struct quick reference

```
is_type_struct(t) -> bool
```

```
See BTF_STRUCT.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_struni quick reference

```
is_type_struni(t) -> bool
Is the type a struct or union?

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_sue quick reference

```
is_type_sue(t) -> bool
Is the type a struct/union/enum?

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_tbyte quick reference

```
is_type_tbyte(t) -> bool
See BTF_FLOAT.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_typedef quick reference

```
is_type_typedef(t) -> bool
See BTF_TYPEDEF.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_uchar quick reference

```
is_type_uchar(t) -> bool
See BTF_UCHAR.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_uint quick reference

```
is_type_uint(t) -> bool
```

```
See BTF_UINT.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_uint128 quick reference

```
is_type_uint128(t) -> bool
See BTF_UINT128.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_uint16 quick reference

```
is_type_uint16(t) -> bool
See BTF_UINT16.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_uint32 quick reference

```
is_type_uint32(t) -> bool
See BTF_UINT32.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_uint64 quick reference

```
is_type_uint64(t) -> bool
See BTF_UINT64.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_union quick reference

```
is_type_union(t) -> bool
See BTF_UNION.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_unknown quick reference

```
is_type_unknown(t) -> bool
```

```
See BT_UNKNOWN.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_void quick reference

```
is_type_void(t) -> bool
See BTF_VOID.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_type_volatile quick reference

```
is_type_volatile(t) -> bool
See BTM_VOLATILE.

@param t: (C++: type_t)
```

IDAPython function idaapi.is_typeid_last quick reference

```
is_typeid_last(t) -> bool
Is the type_t the last byte of type declaration? (there are no additional bytes
after a basic type, see _BT_LAST_BASIC)

@param t: (C++: type_t)
```

IDAPython function idaapi.is_uname quick reference

```
is_uname(name) -> bool
Is valid user-specified name? (valid name & !dummy prefix).

@param name (string): name to test. may be nullptr.
@retval 1: yes
@retval 0: no
```

IDAPython function idaapi.is_unary quick reference

```
is_unary(op) -> bool
Is unary operator?

@param op: (C++: ctype_t) enum ctype_t
```

IDAPython function idaapi.is_union quick reference

```
is_union(id) -> bool
Is a union?

@param id (integer):
```

IDAPython function idaapi.is_unknown quick reference

```
is_unknown(F) -> bool
Does flag denote unexplored byte?

@param F (integer):
```

IDAPython function idaapi.is_unsigned_mcode quick reference

```
is_unsigned_mcode(code) -> bool

@param code: enum mcode_t
```

IDAPython function idaapi.is_user_cc quick reference

```
is_user_cc(cm) -> bool
Does the calling convention specify argument locations explicitly?

@param cm: (C++: cm_t)
```

IDAPython function idaapi.is_usersp quick reference

```
is_usersp(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_userti quick reference

```
is_userti(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_valid_cp quick reference

```
is_valid_cp(cp, kind, data=None) -> bool
```

```
Is the given codepoint acceptable in the given context?

@param cp: (C++: wchar32_t)
@param kind: (C++: nametype_t) enum nametype_t
@param data :
```

IDAPython function idaapi.is_valid_dstate quick reference

```
is_valid_dstate(state) -> bool

@param state: int
```

IDAPython function idaapi.is_valid_trace_file quick reference

```
is_valid_trace_file(filename) -> bool
Is the specified file a valid trace file for the current database?

@param filename (string): char const *
```

IDAPython function idaapi.is_valid_typename quick reference

```
is_valid_typename(name) -> bool
Is valid type name?

@param name (string): name to test. may be nullptr.
@retval 1: yes
@retval 0: no
```

IDAPython function idaapi.is_vararg_cc quick reference

```
is_vararg_cc(cm) -> bool
Does the calling convention use ellipsis?

@param cm: (C++: cm_t)
```

IDAPython function idaapi.is_varmember quick reference

```
is_varmember(mptr) -> bool
Is variable size member?

@param mptr: (C++: const member_t *) member_t const *
```

IDAPython function idaapi.is_varsize_item quick reference

```
is_varsize_item(ea, F, ti=None, itemsize=None) -> int
Is the item at 'ea' variable size?.

@param ea (integer): linear address of the item
@param F (integer): flags
@param ti: (C++: const opinfo_t *) additional information about the data type. For examp
           current item is a structure instance, then ti->tid is structure id.
           Otherwise is ignored (may be nullptr). If specified as nullptr, will
           be automatically retrieved from the database
@param itemsize: (C++: asize_t *) if not nullptr and the item is varsize, itemsize will
                 the calculated item size (for struct types, the minimal size is
                 returned)
@retval 1: varsize item
@retval 0: fixed item
@retval -1: error (bad data definition)
```

IDAPython function idaapi.is_varstr quick reference

```
is_varstr(id) -> bool
Is variable size structure?

@param id (integer):
```

IDAPython function idaapi.is_visible_cp quick reference

```
is_visible_cp(cp) -> bool
Can a character be displayed in a name? (present in ::NameChars)

@param cp: (C++: wchar32_t)
```

IDAPython function idaapi.is_visible_func quick reference

```
is_visible_func(pfn) -> bool
Is the function visible (not hidden)?

@param pfn (idaapi.func_t):
```

IDAPython function idaapi.is_visible_item quick reference

```
is_visible_item(ea) -> bool
Test visibility of item at given ea.
```

```
@param ea (integer):
```

IDAPython function idaapi.is_visible_segm quick reference

```
is_visible_segm(s) -> bool
See SFL_HIDDEN.

@param s: (C++: segment_t *)
```

IDAPython function idaapi.is_weak_name quick reference

```
is_weak_name(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_word quick reference

```
is_word(F) -> bool
FF_WORD

@param F (integer):
```

IDAPython function idaapi.is_yword quick reference

```
is_yword(F) -> bool
FF_YWORD

@param F (integer):
```

IDAPython function idaapi.is_zstroff quick reference

```
is_zstroff(ea) -> bool

@param ea: ea_t
```

IDAPython function idaapi.is_zword quick reference

```
is_zword(F) -> bool
FF_ZWORD
```

```
@param F (integer):
```

IDAPython function idaapi.jcnd2set quick reference

```
jcnd2set(code) -> mcode_t

@param code: enum mcode_t
```

IDAPython function idaapi.jumpto quick reference

```
jumpto(ea, opnum=-1, uijmp_flags=0x0001) -> bool
Set cursor position in custom ida viewer.

@param custom_viewer(a Widget SWIG wrapper class): view
@param place: (C++: place_t *) target position
@param uijmp_flags: int

@return: success
jumpto(custom_viewer, place, x, y) -> bool

@param custom_viewer: TWidget *
@param place: place_t *
@param x: int
```

IDAPython function idaapi.l__compare2 quick reference

```
l_compare2(t1, t2, ud) -> int

@param t1: place_t const *
@param t2: place_t const *
@param ud
```

IDAPython function idaapi.last__idcv__attr quick reference

```
last_idcv_attr(obj) -> char const *

@param obj: idc_value_t const *
```

IDAPython function idaapi.leading__zero__important quick reference

```
leading_zero_important(ea, n) -> bool
Check if leading zeroes are important.
```

```
@param ea (integer):
@param n (integer):
```

IDAPython function idaapi.lexcompare quick reference

```
lexcompare(a, b) -> int

@param a: mop_t const &
@param b: mop_t const &
```

IDAPython function idaapi.lexcompare_tinfo quick reference

```
lexcompare_tinfo(t1, t2, arg3) -> int

@param t1: uint32
@param t2: uint32
@param arg3: int
```

IDAPython function idaapi.list_bptgrps quick reference

```
list_bptgrps(bptgrps) -> size_t
Retrieve the list of absolute path of all folders of bpt dirtree \sq{Type,
Synchronous function, Notification, none (synchronous function)}

@param bptgrps: (C++: qstrvec_t *) list of absolute path in the bpt dirtree
@return: number of folders returned
list_bptgrps() -> [str, ...]
```

IDAPython function idaapi.lnot quick reference

```
Logically negate the specified expression. The specified expression will be
logically negated. For example, "x == y" is converted into "x != y" by this
function.

@return: logically negated expression.
```

IDAPython function idaapi.load_and_run_plugin quick reference

```
load_and_run_plugin(name, arg) -> bool
Load & run a plugin.
```

```
@param name (string): char const *
@param arg (integer):
```

IDAPython function idaapi.load_binary_file quick reference

```
load_binary_file(filename, li, _neflags, fileoff, basepara, binoff, nbytes) -> bool
Load a binary file into the database. This function usually is called from ui.

@param filename (string): the name of input file as is (if the input file is from
                library, then this is the name from the library)
@param li: (C++: linput_t *) loader input source
@param _neflags: (C++: ushort) Load file flags. For the first file, the flag NEF_FIRST m
                set.
@param fileoff: (C++: qoff64_t) Offset in the input file
@param basepara (integer): Load address in paragraphs
@param binoff (integer): Load offset (load_address=(basepara<<4)+binoff)
@param nbytes (integer): Number of bytes to load from the file.
* 0: up to the end of the file
@retval true: ok
@retval false: failed (couldn't open the file)
```

IDAPython function idaapi.load_custom_icon quick reference

```
Loads a custom icon and returns an identifier that can be used with other APIs

If file_name is passed then the other two arguments are ignored.

Load an icon and return its id (ui_load_custom_icon).

@return: icon id
```

IDAPython function idaapi.load_dbg_dbginfo quick reference

```
load_dbg_dbginfo(path, li=None, base=BADADDR, verbose=False) -> bool
Load debugging information from a file.

@param path (string): path to file
@param li: (C++: linput_t *) loader input. if nullptr, check DBG_NAME_KEY
@param base (integer): loading address
@param verbose (bool): dump status to message window
```

IDAPython function idaapi.load_debugger quick reference

```
load_debugger(dbgname, use_remote) -> bool

@param dbgname: char const *
@param use_remote: bool
```

IDAPython function idaapi.load_ids_module quick reference

```
load_ids_module(fname) -> int
Load and apply IDS file. This function loads the specified IDS file and applies
it to the database. If the program imports functions from a module with the same
name as the name of the ids file being loaded, then only functions from this
module will be affected. Otherwise (i.e. when the program does not import a
module with this name) any function in the program may be affected.

@param fname: (C++: char *) name of file to apply
@retval 1: ok
@retval 0: some error (a message is displayed). if the ids file does not exist,
           no message is displayed
```

IDAPython function idaapi.load_plugin quick reference

```
load_plugin(name) -> PyCapsule or None
Loads a plugin

@param name: char const *
@return:     - None if plugin could not be loaded
    - An opaque object representing the loaded plugin
```

IDAPython function idaapi.load_til quick reference

```
load_til(name, tildir=None) -> til_t
Load til from a file without adding it to the database list (see also add_til).
Failure to load base tils are reported into 'errbuf'. They do not prevent
loading of the main til.

@param name (string): filename of the til. If it's an absolute path, tildir is ignored.
* NB: the file extension is forced to .til
@param tildir (string): directory where to load the til from. nullptr means default til
              subdirectories.
@return: pointer to resulting til, nullptr if failed and error message is in
         errbuf
```

IDAPython function idaapi.load_til_header quick reference

```
load_til_header(tildir, name) -> til_t
Get human-readable til description.

@param tildir (string): char const *
@param name (string): char const *
```

IDAPython function idaapi.load_trace_file quick reference

```
load_trace_file(filename) -> str
Load a recorded trace file in the 'Tracing' window. If the call succeeds and
'buf' is not null, the description of the trace stored in the binary trace file
will be returned in 'buf'

@param filename (string): char const *
```

IDAPython function idaapi.loader_input_t quick reference

```
A helper class to work with linput_t related functions.
This class is also used by file loaders scripts.
```

IDAPython function idaapi.loader_input_t_from_capsule quick reference

```
loader_input_t_from_capsule(pycapsule) -> loader_input_t

@param pycapsule: PyObject *
```

IDAPython function idaapi.loader_input_t_from_fp quick reference

```
loader_input_t_from_fp(fp) -> loader_input_t

@param fp: FILE *
```

IDAPython function idaapi.loader_input_t_from_linput quick reference

```
loader_input_t_from_linput(linput) -> loader_input_t

@param linput: linput_t *
```

IDAPython function idaapi.locate_lvar quick reference

```
locate_lvar(out, func_ea, varname) -> bool
Find a variable by name.

@param out: (C++: lvar_locator_t *) output buffer for the variable locator
@param func_ea (integer): function start address
@param varname (string): variable name
@return: success Since VARNAME is not always enough to find the variable, it may
         decompile the function.
```

IDAPython function idaapi.lock_func_range quick reference

```
lock_func_range(pfn, lock)
Lock function pointer Locked pointers are guaranteed to remain valid until they
are unlocked. Ranges with locked pointers cannot be deleted or moved.

@param pfn: (C++: const func_t *) func_t const *
@param lock (bool):
```

IDAPython function idaapi.lock_segm quick reference

```
lock_segm(segm, lock)
Lock segment pointer Locked pointers are guaranteed to remain valid until they
are unlocked. Ranges with locked pointers cannot be deleted or moved.

@param segm: (C++: const segment_t *) segment_t const *
@param lock (bool):
```

IDAPython function idaapi.log2ceil quick reference

```
log2ceil(d64) -> int
calculate ceil(log2(d64)) or floor(log2(d64)), it returns 0 if d64 == 0

@param d64 (integer):
```

IDAPython function idaapi.log2floor quick reference

```
log2floor(d64) -> int

@param d64: uint64
```

IDAPython function idaapi.long__type quick reference

“‘int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.___int___(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by ‘+’ or ‘-’ and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. »> int(‘0b100’, base=0) 4

```
### IDAPython function idaapi.lookup_key_code quick reference

lookup_key_code(key, shift, is_qt) -> ushort
Get shortcut code previously created by ui_get_key_code.

@param key (integer): key constant
@param shift (integer): modifiers
@param is_qt (bool): are we using gui version?

### IDAPython function idaapi.lower_type quick reference

lower_type(til, tif, name=None, _helper=None) -> int
Lower type. Inspect the type and lower all function subtypes using
lower_func_type().
We call the prototypes usually encountered in source files "high level"
They may have implicit arguments, array arguments, big structure retvals, etc
We introduce explicit arguments (i.e. 'this' pointer) and call the result
"low level prototype". See FTI_HIGH.

In order to improve heuristics for recognition of big structure retvals,
it is recommended to pass a helper that will be used to make decisions.
That helper will be used only for lowering 'tif', and not for the children
types walked through by recursion.
@retval 1: removed FTI_HIGH,
@retval 2: made substantial changes
@retval -1: failure

@param til (idaapi.til_t):
@param tif: (C++: tinfo_t *)
@param name (string): char const *
@param _helper: (C++: lowertype_helper_t *)

### IDAPython function idaapi.lvar_mapping_begin quick reference

lvar_mapping_begin(map) -> lvar_mapping_iterator_t
Get iterator pointing to the beginning of lvar_mapping_t.
```

```
@param map: (C++: const lvar_mapping_t *) lvar_mapping_t const *

### IDAPython function idaapi.lvar_mapping_clear quick reference

lvar_mapping_clear(map)
Clear lvar_mapping_t.

@param map: (C++: lvar_mapping_t *)

### IDAPython function idaapi.lvar_mapping_end quick reference

lvar_mapping_end(map) -> lvar_mapping_iterator_t
Get iterator pointing to the end of lvar_mapping_t.

@param map: (C++: const lvar_mapping_t *) lvar_mapping_t const *

### IDAPython function idaapi.lvar_mapping_erase quick reference

lvar_mapping_erase(map, p)
Erase current element from lvar_mapping_t.

@param map: (C++: lvar_mapping_t *)
@param p: (C++: lvar_mapping_iterator_t)

### IDAPython function idaapi.lvar_mapping_find quick reference

lvar_mapping_find(map, key) -> lvar_mapping_iterator_t
Find the specified key in lvar_mapping_t.

@param map: (C++: const lvar_mapping_t *) lvar_mapping_t const *
@param key: (C++: const lvar_locator_t &) lvar_locator_t const &

### IDAPython function idaapi.lvar_mapping_first quick reference

lvar_mapping_first(p) -> lvar_locator_t
Get reference to the current map key.

@param p: (C++: lvar_mapping_iterator_t)

### IDAPython function idaapi.lvar_mapping_free quick reference

lvar_mapping_free(map)
Delete lvar_mapping_t instance.

@param map: (C++: lvar_mapping_t *)

### IDAPython function idaapi.lvar_mapping_insert quick reference

lvar_mapping_insert(map, key, val) -> lvar_mapping_iterator_t
Insert new (lvar_locator_t, lvar_locator_t) pair into lvar_mapping_t.

@param map: (C++: lvar_mapping_t *)
```

```
@param key: (C++: const lvar_locator_t &) lvar_locator_t const &
@param val: (C++: const lvar_locator_t &) lvar_locator_t const &

### IDAPython function idaapi.lvar_mapping_new quick reference

lvar_mapping_new() -> lvar_mapping_t
Create a new lvar_mapping_t instance.

### IDAPython function idaapi.lvar_mapping_next quick reference

lvar_mapping_next(p) -> lvar_mapping_iterator_t
Move to the next element.

@param p: (C++: lvar_mapping_iterator_t)

### IDAPython function idaapi.lvar_mapping_prev quick reference

lvar_mapping_prev(p) -> lvar_mapping_iterator_t
Move to the previous element.

@param p: (C++: lvar_mapping_iterator_t)

### IDAPython function idaapi.lvar_mapping_second quick reference

lvar_mapping_second(p) -> lvar_locator_t
Get reference to the current map value.

@param p: (C++: lvar_mapping_iterator_t)

### IDAPython function idaapi.lvar_mapping_size quick reference

lvar_mapping_size(map) -> size_t
Get size of lvar_mapping_t.

@param map: (C++: lvar_mapping_t *)

### IDAPython function idaapi.lvar_off quick reference

lvar_off(pfn, frameoff) -> sval_t

@param pfn: func_t const *
@param frameoff: uval_t

### IDAPython function idaapi.macros_enabled quick reference

inf_macros_enabled() -> bool

### IDAPython function idaapi.make_name_auto quick reference

make_name_auto(ea) -> bool

@param ea: ea_t

### IDAPython function idaapi.make_name_non_public quick reference
```

```
make_name_non_public(ea)

@param ea: ea_t

### IDAPython function idaapi.make_name_non_weak quick reference

make_name_non_weak(ea)

@param ea: ea_t

### IDAPython function idaapi.make_name_public quick reference

make_name_public(ea)

@param ea: ea_t

### IDAPython function idaapi.make_name_user quick reference

make_name_user(ea) -> bool

@param ea: ea_t

### IDAPython function idaapi.make_name_weak quick reference

make_name_weak(ea)

@param ea: ea_t

### IDAPython function idaapi.make_num quick reference

Create a number expression

### IDAPython function idaapi.make_pointer quick reference

make_pointer(type) -> tinfo_t
Create a pointer type. This function performs the following conversion: "type"
-> "type*"

@param type (idaapi.tinfo_t): object type.
@return: "type*". for example, if 'char' is passed as the argument,

### IDAPython function idaapi.make_ref quick reference

Create a reference. This function performs the following conversion: "obj" =>
"&obj". It can handle casts, annihilate "&*", and process other special cases.

### IDAPython function idaapi.make_str_type quick reference

make_str_type(type_code, encoding_idx, term1=0, term2=0) -> int32
Get string type for a string in the given encoding.

@param type_code: (C++: uchar)
@param encoding_idx (integer):
@param term1: (C++: uchar)
```

```
@param term2: (C++: uchar)

### IDAPython function idaapi.map_code_ea quick reference

map_code_ea(insn, addr, opnum) -> ea_t

@param insn: an idaapi.insn_t, or an address (C++: const insn_t &)
@param addr: ea_t
@param opnum: int

map_code_ea(insn, op) -> ea_t

@param insn: an idaapi.insn_t, or an address (C++: const insn_t &)
@param op: op_t const &

### IDAPython function idaapi.map_data_ea quick reference

map_data_ea(insn, addr, opnum=-1) -> ea_t

@param insn: an idaapi.insn_t, or an address (C++: const insn_t &)
@param addr: ea_t
@param opnum: int

map_data_ea(insn, op) -> ea_t

@param insn: an idaapi.insn_t, or an address (C++: const insn_t &)
@param op: op_t const &

### IDAPython function idaapi.map_ea quick reference

map_ea(insn, op, iscode) -> ea_t

@param insn: an idaapi.insn_t, or an address (C++: const insn_t &)
@param op: op_t const &
@param iscode: bool

map_ea(insn, addr, opnum, iscode) -> ea_t

@param insn: an idaapi.insn_t, or an address (C++: const insn_t &)
@param addr: ea_t
@param opnum: int
@param iscode: bool

### IDAPython function idaapi.mark_cfunc_dirty quick reference

mark_cfunc_dirty(ea, close_views=False) -> bool
Flush the cached decompilation results. Erases a cache entry for the specified
function.

@param ea (integer): function to erase from the cache
```

```
@param close_views (bool): close pseudocode windows that show the function
@return: if a cache entry existed.

### IDAPython function idaapi.mark_position quick reference

mark_position(ea, lnnum, x, y, slot, comment)


@param ea: ea_t
@param lnnum: int
@param x: short
@param y: short
@param slot: int32
@param comment: char const *

### IDAPython function idaapi.may_create_stkvars quick reference

may_create_stkvars() -> bool
Is it allowed to create stack variables automatically?. This function should be
used by IDP modules before creating stack vars.

### IDAPython function idaapi.may_trace_sp quick reference

may_trace_sp() -> bool
Is it allowed to trace stack pointer automatically?. This function should be
used by IDP modules before tracing sp.

### IDAPython function idaapi.mba_t_deserialize quick reference

mba_t_deserialize(bytes) -> mba_t


@param bytes: uchar const *

### IDAPython function idaapi.mcode_modifies_d quick reference

mcode_modifies_d(mcode) -> bool


@param mcode: enum mcode_t

### IDAPython function idaapi.mem2base quick reference

mem2base(py_mem, ea, fpos=-1) -> int
Load database from the memory.


@param py_mem: the buffer
@param ea: start linear addresses
@param fpos: position in the input file the data is taken from.
             if == -1, then no file position correspond to the data.
@return:     - Returns zero if the passed buffer was not a string
    - Otherwise 1 is returned

### IDAPython function idaapi.modify_user_lvar_info quick reference
```

```
modify_user_lvar_info(func_ea, mli_flags, info) -> bool
Modify saved local variable settings of one variable.

@param func_ea (integer): function start address
@param mli_flags (integer): bits that specify which attrs defined by INFO are to be set
@param info: (C++: const lvar_saved_info_t &) local variable info attrs
@return: true if modified, false if invalid MLI_FLAGS passed

### IDAPython function idaapi.modify_user_lvars quick reference

modify_user_lvars(entry_ea, mlv) -> bool
Modify saved local variable settings.

@param entry_ea (integer): function start address
@param mlv: (C++: user_lvar_modifier_t &) local variable modifier
@return: true if modified variables

### IDAPython function idaapi.move_bpt_to_grp quick reference

set_bpt_group(bpt, grp_name) -> bool
Move a bpt into a folder in the breakpoint dirtree if the folder didn't exists,
it will be created \sq{Type, Synchronous function, Notification, none
(synchronous function)}

@param bpt: (C++: bpt_t &) bpt that will be moved
@param grp_name (string): absolute path to the breakpoint dirtree folder
@return: success

### IDAPython function idaapi.move_idcv quick reference

move_idcv(dst, src) -> error_t
Move 'src' to 'dst'. This function is more effective than copy_idcv since it
never copies big amounts of data.

@param dst (idaapi.idc_value_t):
@param src (idaapi.idc_value_t):

### IDAPython function idaapi.move_privrange quick reference

move_privrange(new_privrange_start) -> bool
Move privrange to the specified address

@param new_privrange_start (integer): new start address of the privrange
@return: success

### IDAPython function idaapi.move_segm quick reference

move_segm(s, to, flags=0) -> move_segm_code_t
This function moves all information to the new address. It fixes up address
sensitive information in the kernel. The total effect is equal to reloading the
segment to the target address. For the file format dependent address sensitive
```

```
information, loader_t::move_segm is called. Also IDB notification event
idb_event::segm_moved is called.

@param s: (C++: segment_t *) segment to move
@param to (integer): new segment start address
@param flags (integer): Move segment flags
@return: Move segment result codes

### IDAPython function idaapi.move_segm_start quick reference

move_segm_start(ea, newstart, mode) -> bool
Move segment start. The main difference between this function and
set_segm_start() is that this function may expand the previous segment while
set_segm_start() never does it. So, this function allows to change bounds of two
segments simultaneously. If the previous segment and the specified segment have
the same addressing mode and segment base, then instructions and data are not
destroyed - they simply move from one segment to another. Otherwise all
instructions/data which migrate from one segment to another are destroyed.
@note: this function never disables addresses.

@param ea (integer): any address belonging to the segment
@param newstart (integer): new start address of the segment note that segment start
                address should be higher than segment base linear address.
@param mode (integer): policy for destroying defined items
* 0: if it is necessary to destroy defined items, display a dialog box and ask
confirmation
* 1: if it is necessary to destroy defined items, just destroy them without
asking the user
* -1: if it is necessary to destroy defined items, don't destroy them (i.e.
function will fail)
* -2: don't destroy defined items (function will succeed)
@retval 1: ok
@retval 0: failed, a warning message is displayed

### IDAPython function idaapi.move_segm_strerror quick reference

move_segm_strerror(code) -> char const *
Return string describing error MOVE_SEGM_... code.

@param code: (C++: move_segm_code_t) enum move_segm_code_t

### IDAPython function idaapi.mreg2reg quick reference

mreg2reg(reg, width) -> int
Map a microregister to a processor register.

@param reg: (C++: mreg_t) microregister number
@param width (integer): size of microregister in bytes
@return: processor register id or -1
```

```
### IDAPython function idaapi.msg quick reference

msg(o) -> int
Display an UTF-8 string in the message window

The result of the stringification of the arguments
will be treated as an UTF-8 string.

@param message: message to print (formatting is done in Python)

This function can be used to debug IDAPython scripts

### IDAPython function idaapi.msg_clear quick reference

msg_clear()
Clear the "Output" window.

### IDAPython function idaapi.msg_get_lines quick reference

msg_get_lines(count=-1) -> [str, ...]
Retrieve the last 'count' lines from the output window, in reverse order (from
most recent, to least recent)

@param count (integer): The number of lines to retrieve. -1 means: all

### IDAPython function idaapi.msg_save quick reference

msg_save(path) -> bool
Save the "Output" window contents into a file

@param path: (C++: qstring &) The path of the file to save the contents into. An empty path
             that the user will be prompted for the destination and, if the file
             already exists, the user will be asked to confirm before overriding
             its contents. Upon return, 'path' will contain the path that the
             user chose.
@return: success

### IDAPython function idaapi.must_mcode_close_block quick reference

must_mcode_close_block(mcode, including_calls) -> bool
Must an instruction with the given opcode be the last one in a block? Such
opcodes are called closing opcodes.

@param mcode: (C++: mcode_t) instruction opcode
@param including_calls (bool): should m_call/m_icall be considered as the closing
                        opcodes? If this function returns true, the opcode
                        cannot appear in the middle of a block. Calls are a
                        special case: unknown calls (is_unknown_call) are
                        considered as closing opcodes.

### IDAPython function idaapi.nbits quick reference
```

```
nbits(ea) -> int
Get number of bits in a byte at the given address.

@param ea (integer):
@return: processor_t::dnbits() if the address doesn't belong to a segment,
         otherwise the result depends on the segment type

### IDAPython function idaapi.negate_mcode_relation quick reference

negate_mcode_relation(code) -> mcode_t

@param code: enum mcode_t

### IDAPython function idaapi.negated_relation quick reference

negated_relation(op) -> ctype_t
Negate a comparison operator. For example, cot_sge becomes cot_slt.

@param op: (C++: ctype_t) enum ctype_t

### IDAPython function idaapi.netnode_exist quick reference

netnode_exist(_name) -> bool

@param _name: char const *

### IDAPython function idaapi.new_block quick reference

Create a new block-statement.

### IDAPython function idaapi.new_til quick reference

new_til(name, desc) -> til_t
Initialize a til.

@param name (string): char const *
@param desc (string): char const *

### IDAPython function idaapi.next_addr quick reference

next_addr(ea) -> ea_t
Get next address in the program (i.e. next address which has flags).

@param ea (integer):
@return: BADADDR if no such address exist.

### IDAPython function idaapi.next_chunk quick reference

next_chunk(ea) -> ea_t
Get the first address of next contiguous chunk in the program.

@param ea (integer):
@return: BADADDR if next chunk doesn't exist.
```

```
### IDAPython function idaapi.next_head quick reference

next_head(ea, maxea) -> ea_t
Get start of next defined item.

@param ea (integer): begin search at this address
@param maxea (integer): not included in the search range
@return: BADADDR if none exists.

### IDAPython function idaapi.next_idcv_attr quick reference

next_idcv_attr(obj, attr) -> char const *

@param obj: idc_value_t const *
@param attr: char const *

### IDAPython function idaapi.next_inited quick reference

next_inited(ea, maxea) -> ea_t
Find the next initialized address.

@param ea (integer):
@param maxea (integer):

### IDAPython function idaapi.next_named_type quick reference

next_named_type(ti, name, ntf_flags) -> char const *
Enumerate types.

@param ti (idaapi.til_t): type library. nullptr means the local type library for the current
           database.
@param name (string): the current name. the name that follows this one will be returned.
@param ntf_flags (integer): combination of Flags for named types
@return: Type or symbol names, depending of ntf_flags. Returns mangled names.
         Never returns anonymous types. To include them, enumerate types by
         ordinals.

### IDAPython function idaapi.next_not_tail quick reference

next_not_tail(ea) -> ea_t
Get address of next non-tail byte.

@param ea (integer):
@return: BADADDR if none exists.

### IDAPython function idaapi.next_that quick reference

next_that(ea, maxea, testf) -> ea_t
Find next address with a flag satisfying the function 'testf'.
@note: do not pass is_unknown() to this function to find unexplored bytes. It
       will fail under the debugger. To find unexplored bytes, use
```

```
       next_unknown().

@param ea (integer): start searching at this address + 1
@param maxea (integer): not included in the search range.
@param testf: (C++: testf_t *) test function to find next address
@return: the found address or BADADDR.

### IDAPython function idaapi.next_unknown quick reference

next_unknown(ea, maxea) -> ea_t
Similar to next_that(), but will find the next address that is unexplored.

@param ea (integer):
@param maxea (integer):

### IDAPython function idaapi.next_visea quick reference

next_visea(ea) -> ea_t
Get next visible address.

@param ea (integer):
@return: BADADDR if none exists.

### IDAPython function idaapi.node2ea quick reference

node2ea(ndx) -> ea_t

@param ndx: nodeidx_t

### IDAPython function idaapi.nomem quick reference

nomem(format)

@param format: char const *

### IDAPython function idaapi.notify_when quick reference

Register a callback that will be called when an event happens.
@param when: one of NW_XXXX constants
@param callback: This callback prototype varies depending on the 'when' parameter:
                 The general callback format:
                     def notify_when_callback(nw_code)
                 In the case of NW_OPENIDB:
                     def notify_when_callback(nw_code, is_old_database)
@return: Boolean

### IDAPython function idaapi.num_flag quick reference

num_flag() -> flags64_t
Get number of default base (bin, oct, dec, hex)

### IDAPython function idaapi.object_t quick reference
```

```
Helper class used to initialize empty objects

### IDAPython function idaapi.oct_flag quick reference

oct_flag() -> flags64_t
Get number flag of the base, regardless of current processor - better to use
num_flag()

### IDAPython function idaapi.off_flag quick reference

off_flag() -> flags64_t
see FF_opbits

### IDAPython function idaapi.op_adds_xrefs quick reference

op_adds_xrefs(F, n) -> bool
Should processor module create xrefs from the operand?. Currently 'offset' and
'structure offset' operands create xrefs

@param F (integer):
@param n (integer):

### IDAPython function idaapi.op_bin quick reference

op_bin(ea, n) -> bool
set op type to bin_flag()

@param ea (integer):
@param n (integer):

### IDAPython function idaapi.op_chr quick reference

op_chr(ea, n) -> bool
set op type to char_flag()

@param ea (integer):
@param n (integer):

### IDAPython function idaapi.op_custfmt quick reference

op_custfmt(ea, n, fid) -> bool
Set custom data format for operand (fid-custom data format id)

@param ea (integer):
@param n (integer):
@param fid (integer):

### IDAPython function idaapi.op_dec quick reference

op_dec(ea, n) -> bool
set op type to dec_flag()

@param ea (integer):
```

```
@param n (integer):

### IDAPython function idaapi.op_enum quick reference

op_enum(ea, n, id, serial) -> bool
Set operand representation to be 'enum_t'. If applied to unexplored bytes,
converts them to 16/32bit word data

@param ea (integer): linear address
@param n (integer): 0..UA_MAXOP-1 operand number, OPND_ALL all operands
@param id (integer): id of enum
@param serial: (C++: uchar) the serial number of the constant in the enumeration, usually 0.
               the serial numbers are used if the enumeration contains several
               constants with the same value
@return: success

### IDAPython function idaapi.op_flt quick reference

op_flt(ea, n) -> bool
set op type to flt_flag()

@param ea (integer):
@param n (integer):

### IDAPython function idaapi.op_hex quick reference

op_hex(ea, n) -> bool
set op type to hex_flag()

@param ea (integer):
@param n (integer):

### IDAPython function idaapi.op_num quick reference

op_num(ea, n) -> bool
set op type to num_flag()

@param ea (integer):
@param n (integer):

### IDAPython function idaapi.op_oct quick reference

op_oct(ea, n) -> bool
set op type to oct_flag()

@param ea (integer):
@param n (integer):

### IDAPython function idaapi.op_offset quick reference

op_offset(ea, n, type_and_flags, target=BADADDR, base=0, tdelta=0) -> bool
See op_offset_ex()
```

```
@param ea (integer):
@param n (integer):
@param type_and_flags (integer):
@param target (integer):
@param base (integer):
@param tdelta (integer):

### IDAPython function idaapi.op_offset_ex quick reference

op_offset_ex(ea, n, ri) -> bool
Convert operand to a reference. To delete an offset, use clr_op_type() function.

@param ea (integer): linear address. if 'ea' has unexplored bytes, try to convert them to
* no segment: fail
* 16bit segment: to 16bit word data
* 32bit segment: to dword
@param n (integer): operand number (may be ORed with OPND_OUTER)
* 0: first
* 1: second
* ...
* 7: eighth operand
* OPND_MASK: all operands
@param ri: (C++: const refinfo_t *) reference information
@return: success

### IDAPython function idaapi.op_plain_offset quick reference

op_plain_offset(ea, n, base) -> bool
Convert operand to a reference with the default reference type.

@param ea (integer):
@param n (integer):
@param base (integer):

### IDAPython function idaapi.op_seg quick reference

op_seg(ea, n) -> bool
Set operand representation to be 'segment'. If applied to unexplored bytes,
converts them to 16/32bit word data

@param ea (integer): linear address
@param n (integer): 0..UA_MAXOP-1 operand number, OPND_ALL all operands
@return: success

### IDAPython function idaapi.op_stkvar quick reference

op_stkvar(ea, n) -> bool
Set operand representation to be 'stack variable'. Should be applied to an
instruction within a function. Should be applied after creating a stack var
```

```
using insn_t::create_stkvar().

@param ea (integer): linear address
@param n (integer): 0..UA_MAXOP-1 operand number, OPND_ALL all operands
@return: success

### IDAPython function idaapi.op_stroff quick reference

op_stroff(insn, n, path, path_len, delta) -> bool
Set operand representation to be 'struct offset'. If applied to unexplored
bytes, converts them to 16/32bit word data

@param insn: (C++: const insn_t &) the instruction
@param n (integer): 0..UA_MAXOP-1 operand number, OPND_ALL all operands
@param path: (C++: const tid_t *) structure path (strpath). see nalt.hpp for more info.
@param path_len (integer): length of the structure path
@param delta (integer): struct offset delta. usually 0. denotes the difference between the
              structure base and the pointer into the structure.
@return: success

### IDAPython function idaapi.op_t__from_ptrval__ quick reference

op_t__from_ptrval__(ptrval) -> op_t

@param ptrval: size_t

### IDAPython function idaapi.op_uses_x quick reference

op_uses_x(op) -> bool
Does operator use the 'x' field of cexpr_t?

@param op: (C++: ctype_t) enum ctype_t

### IDAPython function idaapi.op_uses_y quick reference

op_uses_y(op) -> bool
Does operator use the 'y' field of cexpr_t?

@param op: (C++: ctype_t) enum ctype_t

### IDAPython function idaapi.op_uses_z quick reference

op_uses_z(op) -> bool
Does operator use the 'z' field of cexpr_t?

@param op: (C++: ctype_t) enum ctype_t

### IDAPython function idaapi.open_bookmarks_window quick reference

open_bookmarks_window(w) -> TWidget *
Open the bookmarks window (ui_open_builtin).
```

```
@param w(a Widget SWIG wrapper class): The widget for which the bookmarks will open. For exa
          an IDAView, or Enums view, etc.
@return: pointer to resulting window

### IDAPython function idaapi.open_bpts_window quick reference

open_bpts_window(ea) -> TWidget *
Open the breakpoints window (ui_open_builtin).

@param ea (integer): index of entry to select by default
@return: pointer to resulting window

### IDAPython function idaapi.open_calls_window quick reference

open_calls_window(ea) -> TWidget *
Open the function calls window (ui_open_builtin).

@param ea (integer):
@return: pointer to resulting window

### IDAPython function idaapi.open_disasm_window quick reference

open_disasm_window(window_title, ranges=None) -> TWidget *
Open a disassembly view (ui_open_builtin).

@param window_title (string): title of view to open
@param ranges: (C++: const rangevec_t *) if != nullptr, then display a flow chart with the s
               ranges
@return: pointer to resulting window

### IDAPython function idaapi.open_enums_window quick reference

open_enums_window(const_id=BADADDR) -> TWidget *
Open the enums window (ui_open_builtin).

@param const_id (integer): index of entry to select by default
@return: pointer to resulting window

### IDAPython function idaapi.open_exports_window quick reference

open_exports_window(ea) -> TWidget *
Open the exports window (ui_open_builtin).

@param ea (integer): index of entry to select by default
@return: pointer to resulting window

### IDAPython function idaapi.open_form quick reference

Display a dockable modeless dialog box and return a handle to it. The modeless
form can be closed in the following ways:
* by pressing the small 'x' in the window title
* by calling form_actions_t::close() from the form callback (form_actions_t)
```

```
@note: pressing the 'Yes/No/Cancel' buttons does not close the modeless form,
       except if the form callback explicitly calls close().

@return: handle to the form or nullptr. the handle can be used with TWidget

### IDAPython function idaapi.open_frame_window quick reference

open_frame_window(pfn, offset) -> TWidget *
Open the frame window for the given function (ui_open_builtin).

@param pfn (idaapi.func_t): function to analyze
@param offset (integer): offset where the cursor is placed
@return: pointer to resulting window if 'pfn' is a valid function and the window
         was displayed,
nullptr otherwise

### IDAPython function idaapi.open_funcs_window quick reference

open_funcs_window(ea) -> TWidget *
Open the 'Functions' window (ui_open_builtin).

@param ea (integer): index of entry to select by default
@return: pointer to resulting window

### IDAPython function idaapi.open_hexdump_window quick reference

open_hexdump_window(window_title) -> TWidget *
Open a hexdump view (ui_open_builtin).

@param window_title (string): title of view to open
@return: pointer to resulting window

### IDAPython function idaapi.open_imports_window quick reference

open_imports_window(ea) -> TWidget *
Open the exports window (ui_open_builtin).

@param ea (integer): index of entry to select by default
@return: pointer to resulting window

### IDAPython function idaapi.open_linput quick reference

open_linput(file, remote) -> linput_t *
Open loader input.

@param file (string): char const *
@param remote (bool):

### IDAPython function idaapi.open_loctypes_window quick reference

open_loctypes_window(ordinal) -> TWidget *
Open the local types window (ui_open_builtin).
```

```
@param ordinal (integer): ordinal of type to select by default
@return: pointer to resulting window

### IDAPython function idaapi.open_modules_window quick reference

open_modules_window() -> TWidget *
Open the modules window (ui_open_builtin).

@return: pointer to resulting window

### IDAPython function idaapi.open_names_window quick reference

open_names_window(ea) -> TWidget *
Open the names window (ui_open_builtin).

@param ea (integer): index of entry to select by default
@return: pointer to resulting window

### IDAPython function idaapi.open_navband_window quick reference

open_navband_window(ea, zoom) -> TWidget *
Open the navigation band window (ui_open_builtin).

@param ea (integer): sets the address of the navband arrow
@param zoom (integer): sets the navband zoom level
@return: pointer to resulting window

### IDAPython function idaapi.open_notepad_window quick reference

open_notepad_window() -> TWidget *
Open the notepad window (ui_open_builtin).

@return: pointer to resulting window

### IDAPython function idaapi.open_problems_window quick reference

open_problems_window(ea) -> TWidget *
Open the problems window (ui_open_builtin).

@param ea (integer): index of entry to select by default
@return: pointer to resulting window

### IDAPython function idaapi.open_pseudocode quick reference

open_pseudocode(ea, flags) -> vdui_t
Open pseudocode window. The specified function is decompiled and the pseudocode
window is opened.

@param ea (integer): function to decompile
@param flags (integer): a combination of OPF_ flags
@return: false if failed
```

```
### IDAPython function idaapi.open_segments_window quick reference

open_segments_window(ea) -> TWidget *
Open the segments window (ui_open_builtin).

@param ea (integer): index of entry to select by default
@return: pointer to resulting window

### IDAPython function idaapi.open_segregs_window quick reference

open_segregs_window(ea) -> TWidget *
Open the segment registers window (ui_open_builtin).

@param ea (integer): index of entry to select by default
@return: pointer to resulting window

### IDAPython function idaapi.open_selectors_window quick reference

open_selectors_window() -> TWidget *
Open the selectors window (ui_open_builtin).

@return: pointer to resulting window

### IDAPython function idaapi.open_signatures_window quick reference

open_signatures_window() -> TWidget *
Open the signatures window (ui_open_builtin).

@return: pointer to resulting window

### IDAPython function idaapi.open_stack_window quick reference

open_stack_window() -> TWidget *
Open the call stack window (ui_open_builtin).

@return: pointer to resulting window

### IDAPython function idaapi.open_strings_window quick reference

open_strings_window(ea, selstart=BADADDR, selend=BADADDR) -> TWidget *
Open the 'Strings' window (ui_open_builtin).

@param ea (integer): index of entry to select by default
@param selstart (integer): ,selend: only display strings that occur within this range
@param selend (integer):
@return: pointer to resulting window

### IDAPython function idaapi.open_structs_window quick reference

open_structs_window(id=BADADDR, offset=0) -> TWidget *
Open the structs window (ui_open_builtin).
```

```
@param id (integer): index of entry to select by default
@param offset (integer): offset where the cursor is placed
@return: pointer to resulting window

### IDAPython function idaapi.open_threads_window quick reference

open_threads_window() -> TWidget *
Open the threads window (ui_open_builtin).

@return: pointer to resulting window

### IDAPython function idaapi.open_tils_window quick reference

open_tils_window() -> TWidget *
Open the type libraries window (ui_open_builtin).

@return: pointer to resulting window

### IDAPython function idaapi.open_trace_window quick reference

open_trace_window() -> TWidget *
Open the tracing window (ui_open_builtin).

@return: pointer to resulting window

### IDAPython function idaapi.open_url quick reference

open_url(url)
Open the given url (ui_open_url)

@param url (string): char const *

### IDAPython function idaapi.open_xrefs_window quick reference

open_xrefs_window(ea) -> TWidget *
Open the cross references window (ui_open_builtin).

@param ea (integer): index of entry to select by default
@return: pointer to resulting window

### IDAPython function idaapi.optimize_argloc quick reference

optimize_argloc(vloc, size, gaps) -> bool
Verify and optimize scattered argloc into simple form. All new arglocs must be
processed by this function.
@retval true: success
@retval false: the input argloc was illegal

@param vloc: (C++: argloc_t *)
@param size (integer):
@param gaps: (C++: const rangeset_t *) rangeset_t const *
```

```
### IDAPython function idaapi.outctx_base_t__from_ptrval__ quick reference

outctx_base_t__from_ptrval__(ptrval) -> outctx_base_t

@param ptrval: size_t

### IDAPython function idaapi.outctx_t__from_ptrval__ quick reference

outctx_t__from_ptrval__(ptrval) -> outctx_t

@param ptrval: size_t

### IDAPython function idaapi.oword_flag quick reference

oword_flag() -> flags64_t
Get a flags64_t representing a octaword.

### IDAPython function idaapi.pack_idcobj_to_bv quick reference

pack_idcobj_to_bv(obj, tif, bytes, objoff, pio_flags=0) -> error_t
Write a typed idc object to the byte vector. Byte vector may be non-empty, this
function will append data to it

@param obj: (C++: const idc_value_t *) idc_value_t const *
@param tif (idaapi.tinfo_t): tinfo_t const &
@param bytes: (C++: relobj_t *)
@param objoff :
@param pio_flags (integer):

### IDAPython function idaapi.pack_idcobj_to_idb quick reference

pack_idcobj_to_idb(obj, tif, ea, pio_flags=0) -> error_t
Write a typed idc object to the database.

@param obj: (C++: const idc_value_t *) idc_value_t const *
@param tif (idaapi.tinfo_t): tinfo_t const &
@param ea (integer):
@param pio_flags (integer):

### IDAPython function idaapi.pack_object_to_bv quick reference

pack_object_to_bv(py_obj, ti, type, fields, base_ea, pio_flags=0) -> PyObject *
Packs a typed object to a string

@param py_obj: PyObject *
@param ti: Type info. 'None' can be passed.
@param type: type_t const *
@param fields: fields string (may be empty or None)
@param base_ea: base ea used to relocate the pointers in the packed object
@param pio_flags: flags used while unpacking
@return:     tuple(0, err_code) on failure
```

```
    tuple(1, packed_buf) on success

### IDAPython function idaapi.pack_object_to_idb quick reference

pack_object_to_idb(py_obj, ti, type, fields, ea, pio_flags=0) -> PyObject *
Write a typed object to the database.
Raises an exception if wrong parameters were passed or conversion fails
Returns the error_t returned by idaapi.pack_object_to_idb

@param py_obj: PyObject *
@param ti: Type info. 'None' can be passed.
@param type: type_t const *
@param fields: fields string (may be empty or None)
@param ea: ea to be used while packing
@param pio_flags: flags used while unpacking

### IDAPython function idaapi.packreal_flag quick reference

packreal_flag() -> flags64_t
Get a flags64_t representing a packed decimal real.

### IDAPython function idaapi.parse_binpat_str quick reference

parse_binpat_str(out, ea, _in, radix, strlits_encoding=0) -> str
Convert user-specified binary string to internal representation. The 'in'
parameter contains space-separated tokens:
- numbers (numeric base is determined by 'radix')
- if value of number fits a byte, it is considered as a byte
- if value of number fits a word, it is considered as 2 bytes
- if value of number fits a dword,it is considered as 4 bytes
- "..." string constants
- 'x'  single-character constants
- ?    variable bytes

Note that string constants are surrounded with double quotes.

Here are a few examples (assuming base 16):
CD 21          - bytes 0xCD, 0x21
21CD           - bytes 0xCD, 0x21 (little endian ) or 0x21, 0xCD (big-endian)
"Hello", 0     - the null terminated string "Hello"
L"Hello"       - 'H', 0, 'e', 0, 'l', 0, 'l', 0, 'o', 0
B8 ? ? ? ? 90  - byte 0xB8, 4 bytes with any value, byte 0x90

@param out: (C++: compiled_binpat_vec_t *) a vector of compiled binary patterns, for use wit
@param ea (integer): linear address to convert for (the conversion depends on the address,
           because the number of bits in a byte depend on the segment type)
@param in (string): input text string
@param radix (integer): numeric base of numbers (8,10,16)
@param strlits_encoding (integer): the target encoding into which the string literals
```

```
                          present in 'in', should be encoded. Can be any from [1,
                          get_encoding_qty()), or the special values PBSENC_*
@return: false either in case of parsing error, or if at least one requested
         target encoding couldn't encode the string literals present in "in".

### IDAPython function idaapi.parse_command_line3 quick reference

parse_command_line3(cmdline) -> PyObject *

@param cmdline: char const *

### IDAPython function idaapi.parse_dbgopts quick reference

parse_dbgopts(ido, r_switch) -> bool
Parse the -r command line switch (for instant debugging). r_switch points to the
value of the -r switch. Example: win32@localhost+

@param ido: (C++: struct instant_dbgopts_t *) instant_dbgopts_t *
@param r_switch (string): char const *
@return: true-ok, false-parse error

### IDAPython function idaapi.parse_decl quick reference

parse_decl(tif, til, decl, flags) -> str
Parse ONE declaration. If the input string contains more than one declaration,
the first complete type declaration (PT_TYP) or the last variable declaration
(PT_VAR) will be used.
@note: name & tif may be empty after the call!

@param tif: (C++: tinfo_t *) type info
@param til (idaapi.til_t): type library to use. may be nullptr
@param decl (string): C declaration to parse
@param flags (integer): combination of Type parsing flags bits
@retval true: ok
@retval false: declaration is bad, the error message is displayed if !PT_SIL

### IDAPython function idaapi.parse_decls quick reference

parse_decls(til, input, printer, hti_flags) -> int
Parse many declarations and store them in a til. If there are any errors, they
will be printed using 'printer'. This function uses default include path and
predefined macros from the database settings. It always uses the HTI_DCL bit.

@param til (idaapi.til_t): type library to store the result
@param input (string): input string or file name (see hti_flags)
@param printer: (C++: printer_t *) function to output error messages (use msg or nullptr or
                own callback)
@param hti_flags (integer): combination of Type formatting flags
@return: number of errors, 0 means ok.
```

```
### IDAPython function idaapi.parse_decls_for_srclang quick reference

parse_decls_for_srclang(lang, til, input, is_path) -> int
Parse type declarations in the specified language

@param lang: (C++: srclang_t) the source language(s) expected in the input
@param til (idaapi.til_t): type library to store the types
@param input (string): input source. can be a file path or decl string
@param is_path (bool): true if input parameter is a path to a source file, false if the
                input is an in-memory source snippet
@retval -1: no parser was found that supports the given source language(s)
@retval else: the number of errors encountered in the input source

### IDAPython function idaapi.parse_decls_with_parser quick reference

parse_decls_with_parser(parser_name, til, input, is_path) -> int
Parse type declarations using the parser with the specified name

@param parser_name (string): name of the target parser
@param til (idaapi.til_t): type library to store the types
@param input (string): input source. can be a file path or decl string
@param is_path (bool): true if input parameter is a path to a source file, false if the
                input is an in-memory source snippet
@retval -1: no parser was found with the given name
@retval else: the number of errors encountered in the input source

### IDAPython function idaapi.parse_reg_name quick reference

parse_reg_name(ri, regname) -> bool
Get register info by name.

@param ri: (C++: reg_info_t *) result
@param regname (string): name of register
@return: success

### IDAPython function idaapi.parse_user_call quick reference

parse_user_call(udc, decl, silent) -> bool
Convert function type declaration into internal structure

@param udc: (C++: udcall_t *) - pointer to output structure
@param decl (string): - function type declaration
@param silent (bool): - if TRUE: do not show warning in case of incorrect type
@return: success

### IDAPython function idaapi.partial_type_num quick reference

partial_type_num(type) -> int
Calculate number of partial subtypes.
```

```
@param type (idaapi.tinfo_t): tinfo_t const &
@return: number of partial subtypes. The bigger is this number, the uglier is
         the type.

### IDAPython function idaapi.patch_byte quick reference

patch_byte(ea, x) -> bool
Patch a byte of the program. The original value of the byte is saved and can be
obtained by get_original_byte(). This function works for wide byte processors
too.
@retval true: the database has been modified,
@retval false: the debugger is running and the process' memory has value 'x' at
               address 'ea', or the debugger is not running, and the IDB has
               value 'x' at address 'ea already.

@param ea (integer):
@param x (integer):

### IDAPython function idaapi.patch_bytes quick reference

patch_bytes(ea, buf)
Patch the specified number of bytes of the program. Original values of bytes are
saved and are available with get_original...() functions. See also put_bytes().

@param ea (integer): linear address
@param buf: (C++: const void *) buffer with new values of bytes

### IDAPython function idaapi.patch_dword quick reference

patch_dword(ea, x) -> bool
Patch a dword of the program. The original value of the dword is saved and can
be obtained by get_original_dword(). This function DOESN'T work for wide byte
processors. This function takes into account order of bytes specified in
idainfo::is_be()
@retval true: the database has been modified,
@retval false: the debugger is running and the process' memory has value 'x' at
               address 'ea', or the debugger is not running, and the IDB has
               value 'x' at address 'ea already.

@param ea (integer):
@param x (integer):

### IDAPython function idaapi.patch_fixup_value quick reference

patch_fixup_value(ea, fd) -> bool
Patch the fixup bytes. This function updates data or an instruction at `ea' to
the fixup bytes. For example, FIXUP_HI8 updates a byte at `ea' to the high byte
of `fd->off', or AArch64's custom fixup BRANCH26 updates low 26 bits of the insn
at `ea' to the value of `fd->off' shifted right by 2.
```

```
@param ea (integer): address where data are changed, the size of the changed data depends
           on the fixup type.
@see: fixup_handler_t::size
@param fd: (C++: const fixup_data_t &) fixup data
@retval false: the fixup bytes do not fit (e.g. `fd->off' is greater than
               0xFFFFFFC for BRANCH26). The database is changed even in this
               case.
```

IDAPython function idaapi.patch_qword quick reference

```
patch_qword(ea, x) -> bool
Patch a qword of the program. The original value of the qword is saved and can
be obtained by get_original_qword(). This function DOESN'T work for wide byte
processors. This function takes into account order of bytes specified in
idainfo::is_be()
@retval true: the database has been modified,
@retval false: the debugger is running and the process' memory has value 'x' at
               address 'ea', or the debugger is not running, and the IDB has
               value 'x' at address 'ea already.

@param ea (integer):
@param x (integer):
```

IDAPython function idaapi.patch_word quick reference

```
patch_word(ea, x) -> bool
Patch a word of the program. The original value of the word is saved and can be
obtained by get_original_word(). This function works for wide byte processors
too. This function takes into account order of bytes specified in
idainfo::is_be()
@retval true: the database has been modified,
@retval false: the debugger is running and the process' memory has value 'x' at
               address 'ea', or the debugger is not running, and the IDB has
               value 'x' at address 'ea already.

@param ea (integer):
@param x (integer):
```

IDAPython function idaapi.peek_auto_queue quick reference

```
peek_auto_queue(low_ea, type) -> ea_t
Peek into a queue 'type' for an address not lower than 'low_ea'. Do not remove
address from the queue.

@param low_ea (integer):
@param type (one of the idaapi.AU_xxxx flags):
@return: the address or BADADDR
```

IDAPython function idaapi.ph_calcrel quick reference

```
ph_calcrel(ea)

@param ea: ea_t

### IDAPython function idaapi.ph_find_op_value quick reference

ph_find_op_value(insn, op) -> ssize_t

@param insn: an idaapi.insn_t, or an address (C++: const insn_t &)
@param op: int

### IDAPython function idaapi.ph_find_reg_value quick reference

ph_find_reg_value(insn, reg) -> ssize_t

@param insn: an idaapi.insn_t, or an address (C++: const insn_t &)
@param reg: int

### IDAPython function idaapi.ph_get_cnbits quick reference

ph_get_cnbits() -> size_t
Returns the 'ph.cnbits'

### IDAPython function idaapi.ph_get_dnbits quick reference

ph_get_dnbits() -> size_t
Returns the 'ph.dnbits'

### IDAPython function idaapi.ph_get_flag quick reference

ph_get_flag() -> size_t
Returns the 'ph.flag'

### IDAPython function idaapi.ph_get_icode_return quick reference

ph_get_icode_return() -> size_t
Returns the 'ph.icode_return'

### IDAPython function idaapi.ph_get_id quick reference

ph_get_id() -> size_t
Returns the 'ph.id' field

### IDAPython function idaapi.ph_get_instruc quick reference

ph_get_instruc() -> [(str, int), ...]
Returns a list of tuples (instruction_name, instruction_feature) containing the
instructions list as defined in he processor module

### IDAPython function idaapi.ph_get_instruc_end quick reference

ph_get_instruc_end() -> size_t
Returns the 'ph.instruc_end'

### IDAPython function idaapi.ph_get_instruc_start quick reference
```

```
ph_get_instruc_start() -> size_t
Returns the 'ph.instruc_start'

### IDAPython function idaapi.ph_get_operand_info quick reference

ph_get_operand_info(ea, n) -> (int, int, int, int, int) or None
Returns the operand information given an ea and operand number.

@param ea: address
@param n: operand number

@return: Returns an idd_opinfo_t as a tuple: (modified, ea, reg_ival, regidx, value_size).
         Please refer to idd_opinfo_t structure in the SDK.

### IDAPython function idaapi.ph_get_reg_accesses quick reference

ph_get_reg_accesses(accvec, insn, flags) -> ssize_t

@param accvec: reg_accesses_t *
@param insn: an idaapi.insn_t, or an address (C++: const insn_t &)
@param flags: int

### IDAPython function idaapi.ph_get_reg_code_sreg quick reference

ph_get_reg_code_sreg() -> size_t
Returns the 'ph.reg_code_sreg'

### IDAPython function idaapi.ph_get_reg_data_sreg quick reference

ph_get_reg_data_sreg() -> size_t
Returns the 'ph.reg_data_sreg'

### IDAPython function idaapi.ph_get_reg_first_sreg quick reference

ph_get_reg_first_sreg() -> size_t
Returns the 'ph.reg_first_sreg'

### IDAPython function idaapi.ph_get_reg_last_sreg quick reference

ph_get_reg_last_sreg() -> size_t
Returns the 'ph.reg_last_sreg'

### IDAPython function idaapi.ph_get_regnames quick reference

ph_get_regnames() -> [str, ...]
Returns the list of register names as defined in the processor module

### IDAPython function idaapi.ph_get_segreg_size quick reference

ph_get_segreg_size() -> size_t
Returns the 'ph.segreg_size'

### IDAPython function idaapi.ph_get_tbyte_size quick reference
```

```
ph_get_tbyte_size() -> size_t
Returns the 'ph.tbyte_size' field as defined in he processor module

### IDAPython function idaapi.ph_get_version quick reference

ph_get_version() -> size_t
Returns the 'ph.version'

### IDAPython function idaapi.place_t_as_enumplace_t quick reference

place_t_as_enumplace_t(p) -> enumplace_t

@param p: place_t *

### IDAPython function idaapi.place_t_as_idaplace_t quick reference

place_t_as_idaplace_t(p) -> idaplace_t

@param p: place_t *

### IDAPython function idaapi.place_t_as_simpleline_place_t quick reference

place_t_as_simpleline_place_t(p) -> simpleline_place_t

@param p: place_t *

### IDAPython function idaapi.place_t_as_structplace_t quick reference

place_t_as_structplace_t(p) -> structplace_t

@param p: place_t *

### IDAPython function idaapi.plan_and_wait quick reference

plan_and_wait(ea1, ea2, final_pass=True) -> int
Analyze the specified range. Try to create instructions where possible. Make the
final pass over the specified range if specified. This function doesn't return
until the range is analyzed.
@retval 1: ok
@retval 0: Ctrl-Break was pressed

@param ea1 (integer):
@param ea2 (integer):
@param final_pass (bool):

### IDAPython function idaapi.plan_ea quick reference

plan_ea(ea)
Plan to perform reanalysis.

@param ea (integer):

### IDAPython function idaapi.plan_range quick reference
```

```
plan_range(sEA, eEA)
Plan to perform reanalysis.

@param sEA (integer):
@param eEA (integer):

### IDAPython function idaapi.plan_to_apply_idasgn quick reference

plan_to_apply_idasgn(fname) -> int
Add a signature file to the list of planned signature files.

@param fname (string): file name. should not contain directory part.
@return: 0 if failed, otherwise number of planned (and applied) signatures

### IDAPython function idaapi.plgform_close quick reference

plgform_close(py_link, options)

@param py_link: PyObject *
@param options: int

### IDAPython function idaapi.plgform_get_widget quick reference

plgform_get_widget(py_link) -> TWidget *

@param py_link: PyObject *

### IDAPython function idaapi.plgform_new quick reference

plgform_new() -> PyObject *

### IDAPython function idaapi.plgform_show quick reference

plgform_show(py_link, py_obj, caption, options=(0x0040 << 16)|0x00000004u) -> bool

@param py_link: PyObject *
@param py_obj: PyObject *
@param caption: char const *
@param options: int

### IDAPython function idaapi.plugin_t quick reference

Base class for all scripted plugins.

### IDAPython function idaapi.plugmod_t quick reference

Base class for all scripted multi-plugins.

### IDAPython function idaapi.prev_addr quick reference

prev_addr(ea) -> ea_t
Get previous address in the program.

@param ea (integer):
```

```
@return: BADADDR if no such address exist.

### IDAPython function idaapi.prev_chunk quick reference

prev_chunk(ea) -> ea_t
Get the last address of previous contiguous chunk in the program.

@param ea (integer):
@return: BADADDR if previous chunk doesn't exist.

### IDAPython function idaapi.prev_head quick reference

prev_head(ea, minea) -> ea_t
Get start of previous defined item.

@param ea (integer): begin search at this address
@param minea (integer): included in the search range
@return: BADADDR if none exists.

### IDAPython function idaapi.prev_idcv_attr quick reference

prev_idcv_attr(obj, attr) -> char const *

@param obj: idc_value_t const *
@param attr: char const *

### IDAPython function idaapi.prev_inited quick reference

prev_inited(ea, minea) -> ea_t
Find the previous initialized address.

@param ea (integer):
@param minea (integer):

### IDAPython function idaapi.prev_not_tail quick reference

prev_not_tail(ea) -> ea_t
Get address of previous non-tail byte.

@param ea (integer):
@return: BADADDR if none exists.

### IDAPython function idaapi.prev_that quick reference

prev_that(ea, minea, testf) -> ea_t
Find previous address with a flag satisfying the function 'testf'.
@note: do not pass is_unknown() to this function to find unexplored bytes It
       will fail under the debugger. To find unexplored bytes, use
       prev_unknown().

@param ea (integer): start searching from this address - 1.
@param minea (integer): included in the search range.
```

```
@param testf: (C++: testf_t *) test function to find previous address
@return: the found address or BADADDR.

### IDAPython function idaapi.prev_unknown quick reference

prev_unknown(ea, minea) -> ea_t
Similar to prev_that(), but will find the previous address that is unexplored.

@param ea (integer):
@param minea (integer):

### IDAPython function idaapi.prev_visea quick reference

prev_visea(ea) -> ea_t
Get previous visible address.

@param ea (integer):
@return: BADADDR if none exists.

### IDAPython function idaapi.print_argloc quick reference

print_argloc(vloc, size=0, vflags=0) -> str
Convert an argloc to human readable form.

@param vloc: (C++: const argloc_t &) argloc_t const &
@param size (integer):
@param vflags (integer):

### IDAPython function idaapi.print_decls quick reference

print_decls(printer, til, py_ordinals, flags) -> int
Print types (and possibly their dependencies) in a format suitable for using in
a header file. This is the reverse parse_decls().

@param printer: (C++: text_sink_t &) a handler for printing text
@param til (idaapi.til_t): the type library holding the ordinals
@param py_ordinals: ordinals of types to export. nullptr means: all ordinals in til
@param pdf_flags (integer): flags for the algorithm. A combination of PDF_ constants
@retval >0: the number of types exported
@retval 0: an error occurred
@retval <0: the negated number of types exported. There were minor errors and
            the resulting output might not be compilable.

### IDAPython function idaapi.print_idcv quick reference

print_idcv(v, name=None, indent=0) -> str
Get text representation of idc_value_t.

@param v: (C++: const idc_value_t &) idc_value_t const &
@param name (string): char const *
@param indent (integer):
```

```
### IDAPython function idaapi.print_insn_mnem quick reference

print_insn_mnem(ea) -> str
Print instruction mnemonics.

@param ea (integer): linear address of the instruction
@return: success

### IDAPython function idaapi.print_operand quick reference

print_operand(ea, n, getn_flags=0, newtype=None) -> str
Generate text representation for operand #n. This function will generate the
text representation of the specified operand (includes color codes.)

@param ea (integer): the item address (instruction or data)
@param n (integer): 0..UA_MAXOP-1 operand number, meaningful only for instructions
@param getn_flags (integer): Name expression flags Currently only GETN_NODUMMY is
                  accepted.
@param newtype: (C++: struct printop_t *) if specified, print the operand using the specifie
@return: success

### IDAPython function idaapi.print_strlit_type quick reference

print_strlit_type(strtype, flags=0) -> (str, str)
Get string type information: the string type name (possibly decorated with
hotkey markers), and the tooltip.

@param strtype: (C++: int32) the string type
@param flags (integer): or'ed PSTF_* constants
@return: length of generated text

### IDAPython function idaapi.print_tinfo quick reference

print_tinfo(prefix, indent, cmtindent, flags, tif, name, cmt) -> str

@param prefix: char const *
@param indent: int
@param cmtindent: int
@param flags: int
@param tif: tinfo_t const *
@param name: char const *
@param cmt: char const *

### IDAPython function idaapi.print_type quick reference

print_type(ea, prtype_flags) -> str
Get type declaration for the specified address.

@param ea (integer): address
@param prtype_flags (integer): combination of Type printing flags
```

```
@return: success

### IDAPython function idaapi.print_vdloc quick reference

print_vdloc(loc, nbytes) -> str
Print vdloc. Since vdloc does not always carry the size info, we pass it as
NBYTES..

@param loc: (C++: const vdloc_t &) vdloc_t const &
@param nbytes (integer):

### IDAPython function idaapi.process_archive quick reference

process_archive(temp_file, li, module_name, neflags, defmember, loader) -> str
Calls loader_t::process_archive() For parameters and return value description
look at loader_t::process_archive(). Additional parameter 'loader' is a pointer
to load_info_t structure.

@param temp_file: (C++: qstring *)
@param li: (C++: linput_t *)
@param module_name: (C++: qstring *)
@param neflags: (C++: ushort *)
@param defmember (string): char const *
@param loader: (C++: const load_info_t *) load_info_t const *

### IDAPython function idaapi.process_config_directive quick reference

process_config_directive(directive, priority=2)

@param directive: char const *
@param priority: int

### IDAPython function idaapi.process_ui_action quick reference

process_ui_action(name, flags=0) -> bool
Invokes an IDA UI action by name

@param name: action name
@param flags: int
@return: Boolean

### IDAPython function idaapi.put_byte quick reference

put_byte(ea, x) -> bool
Set value of one byte of the program. This function modifies the database. If
the debugger is active then the debugged process memory is patched too.
@note: The original value of the byte is completely lost and can't be recovered
       by the get_original_byte() function. See also put_dbg_byte() to write to
       the process memory directly when the debugger is active. This function
       can handle wide byte processors.
```

```
@param ea (integer): linear address
@param x (integer): byte value
@return: true if the database has been modified

### IDAPython function idaapi.put_bytes quick reference

put_bytes(ea, buf)
Modify the specified number of bytes of the program. This function does not save
the original values of bytes. See also patch_bytes().

@param ea (integer): linear address
@param buf: (C++: const void *) buffer with new values of bytes

### IDAPython function idaapi.put_dbg_byte quick reference

put_dbg_byte(ea, x) -> bool
Change one byte of the debugged process memory.

@param ea (integer): linear address
@param x (integer): byte value
@return: true if the process memory has been modified

### IDAPython function idaapi.put_dword quick reference

put_dword(ea, x)
Set value of one dword of the program. This function takes into account order of
bytes specified in idainfo::is_be() This function works for wide byte processors
too.

@param ea (integer): linear address
@param x (integer): dword value
@note: the original value of the dword is completely lost and can't be recovered
       by the get_original_dword() function.

### IDAPython function idaapi.put_qword quick reference

put_qword(ea, x)
Set value of one qword (8 bytes) of the program. This function takes into
account order of bytes specified in idainfo::is_be() This function DOESN'T works
for wide byte processors.

@param ea (integer): linear address
@param x (integer): qword value

### IDAPython function idaapi.put_word quick reference

put_word(ea, x)
Set value of one word of the program. This function takes into account order of
bytes specified in idainfo::is_be() This function works for wide byte processors
too.
@note: The original value of the word is completely lost and can't be recovered
```

```
       by the get_original_word() function. ea - linear address x - word value

@param ea (integer):
@param x (integer):

### IDAPython function idaapi.py_add_idc_func quick reference

py_add_idc_func(name, fp_ptr, args, defvals, flags) -> bool

@param name: char const *
@param fp_ptr: size_t
@param args: char const *
@param defvals: idc_values_t const &
@param flags: int

### IDAPython function idaapi.py_chooser_base_t_get_row quick reference

py_chooser_base_t_get_row(chobj, n) -> PyObject *

@param chobj: chooser_base_t const *
@param n: size_t

### IDAPython function idaapi.py_clinked_object_t quick reference

This is a utility and base class for C linked objects

### IDAPython function idaapi.py_get_ask_form quick reference

py_get_ask_form() -> size_t

### IDAPython function idaapi.py_get_call_idc_func quick reference

py_get_call_idc_func() -> size_t

### IDAPython function idaapi.py_get_open_form quick reference

py_get_open_form() -> size_t

### IDAPython function idaapi.py_load_custom_icon_data quick reference

py_load_custom_icon_data(data, format) -> int

@param data: PyObject *
@param format: char const *

### IDAPython function idaapi.py_load_custom_icon_fn quick reference

py_load_custom_icon_fn(filename) -> int

@param filename: char const *

### IDAPython function idaapi.py_register_compiled_form quick reference

py_register_compiled_form(py_form)
```

```
@param py_form: PyObject *

### IDAPython function idaapi.py_unregister_compiled_form quick reference

py_unregister_compiled_form(py_form)

@param py_form: PyObject *

### IDAPython function idaapi.pycim_get_widget quick reference

pycim_get_widget(_self) -> TWidget *

@param self: PyObject *

### IDAPython function idaapi.pycim_view_close quick reference

pycim_view_close(_self)

@param self: PyObject *

### IDAPython function idaapi.pyg_close quick reference

pyg_close(_self)

@param self: PyObject *

### IDAPython function idaapi.pyg_select_node quick reference

pyg_select_node(_self, nid)

@param self: PyObject *
@param nid: int

### IDAPython function idaapi.pyg_show quick reference

pyg_show(_self) -> bool

@param self: PyObject *

### IDAPython function idaapi.pygc_create_groups quick reference

pygc_create_groups(_self, groups_infos) -> [int, ...] or None

@param self: PyObject *
@param groups_infos: PyObject *

### IDAPython function idaapi.pygc_delete_groups quick reference

pygc_delete_groups(_self, groups, new_current) -> bool

@param self: PyObject *
@param groups: PyObject *
@param new_current: PyObject *
```

IDAPython function idaapi.pygc_refresh quick reference

pygc_refresh(_self)

@param self: PyObject *

IDAPython function idaapi.pygc_set_groups_visibility quick reference

pygc_set_groups_visibility(_self, groups, expand, new_current) -> bool

@param self: PyObject *
@param groups: PyObject *
@param expand: PyObject *
@param new_current: PyObject *

IDAPython function idaapi.pyidag_bind quick reference

pyidag_bind(_self) -> bool

@param self: PyObject *

IDAPython function idaapi.pyidag_unbind quick reference

pyidag_unbind(_self) -> bool

@param self: PyObject *

IDAPython function idaapi.pyidc_cvt_helper__ quick reference

This is a special helper object that helps detect which kind
of object is this python object wrapping and how to convert it
back and from IDC.
This object is characterized by its special attribute and its value

IDAPython function idaapi.pyidc_opaque_object_t quick reference

This is the base class for all Python<->IDC opaque objects

IDAPython function idaapi.pyscv_add_line quick reference

pyscv_add_line(py_this, py_sl) -> bool

@param py_this: PyObject *
@param py_sl: PyObject *

IDAPython function idaapi.pyscv_clear_lines quick reference

pyscv_clear_lines(py_this) -> PyObject *

@param py_this: PyObject *

IDAPython function idaapi.pyscv_close quick reference

```
pyscv_close(py_this)

@param py_this: PyObject *

### IDAPython function idaapi.pyscv_count quick reference

pyscv_count(py_this) -> size_t

@param py_this: PyObject *

### IDAPython function idaapi.pyscv_del_line quick reference

pyscv_del_line(py_this, nline) -> bool

@param py_this: PyObject *
@param nline: size_t

### IDAPython function idaapi.pyscv_edit_line quick reference

pyscv_edit_line(py_this, nline, py_sl) -> bool

@param py_this: PyObject *
@param nline: size_t
@param py_sl: PyObject *

### IDAPython function idaapi.pyscv_get_current_line quick reference

pyscv_get_current_line(py_this, mouse, notags) -> PyObject *

@param py_this: PyObject *
@param mouse: bool
@param notags: bool

### IDAPython function idaapi.pyscv_get_current_word quick reference

pyscv_get_current_word(py_this, mouse) -> PyObject *

@param py_this: PyObject *
@param mouse: bool

### IDAPython function idaapi.pyscv_get_line quick reference

pyscv_get_line(py_this, nline) -> PyObject *

@param py_this: PyObject *
@param nline: size_t

### IDAPython function idaapi.pyscv_get_pos quick reference

pyscv_get_pos(py_this, mouse) -> PyObject *

@param py_this: PyObject *
@param mouse: bool
```

```
### IDAPython function idaapi.pyscv_get_selection quick reference
pyscv_get_selection(py_this) -> PyObject *

@param py_this: PyObject *
### IDAPython function idaapi.pyscv_get_widget quick reference
pyscv_get_widget(py_this) -> TWidget *

@param py_this: PyObject *
### IDAPython function idaapi.pyscv_init quick reference
pyscv_init(py_link, title) -> PyObject *

@param py_link: PyObject *
@param title: char const *
### IDAPython function idaapi.pyscv_insert_line quick reference
pyscv_insert_line(py_this, nline, py_sl) -> bool

@param py_this: PyObject *
@param nline: size_t
@param py_sl: PyObject *
### IDAPython function idaapi.pyscv_is_focused quick reference
pyscv_is_focused(py_this) -> bool

@param py_this: PyObject *
### IDAPython function idaapi.pyscv_jumpto quick reference
pyscv_jumpto(py_this, ln, x, y) -> bool

@param py_this: PyObject *
@param ln: size_t
@param x: int
@param y: int
### IDAPython function idaapi.pyscv_patch_line quick reference
pyscv_patch_line(py_this, nline, offs, value) -> bool

@param py_this: PyObject *
@param nline: size_t
@param offs: size_t
@param value: int
### IDAPython function idaapi.pyscv_refresh quick reference
```

```
pyscv_refresh(py_this) -> bool

@param py_this: PyObject *

### IDAPython function idaapi.pyscv_show quick reference

pyscv_show(py_this) -> bool

@param py_this: PyObject *

### IDAPython function idaapi.pyw_convert_defvals quick reference

pyw_convert_defvals(out, py_seq) -> bool

@param out: idc_values_t *
@param py_seq: PyObject *

### IDAPython function idaapi.pyw_register_idc_func quick reference

pyw_register_idc_func(name, args, py_fp) -> size_t

@param name: char const *
@param args: char const *
@param py_fp: PyObject *

### IDAPython function idaapi.pyw_unregister_idc_func quick reference

pyw_unregister_idc_func(ctxptr) -> bool

@param ctxptr: size_t

### IDAPython function idaapi.qatoll quick reference

qatoll(nptr) -> int64

@param nptr: char const *

### IDAPython function idaapi.qcleanline quick reference

qcleanline(cmt_char='\0', flags=((1 << 0)|(1 << 1))|(1 << 2)) -> str
Performs some cleanup operations to a line.

@param cmt_char: (C++: char) character that denotes the start of a comment:
* the entire text is removed if the line begins with this character (ignoring
leading spaces)
* all text after (and including) this character is removed if flag CLNL_FINDCMT
is set
@param flags (integer): a combination of line cleanup flags. defaults to CLNL_TRIM
@return: length of line

### IDAPython function idaapi.qcontrol_tty quick reference
```

```
qcontrol_tty()
Make the current terminal the controlling terminal of the calling process.
@note: The current terminal is supposed to be /dev/tty

### IDAPython function idaapi.qdetach_tty quick reference

qdetach_tty()
If the current terminal is the controlling terminal of the calling process, give
up this controlling terminal.
@note: The current terminal is supposed to be /dev/tty

### IDAPython function idaapi.qexit quick reference

qexit(code)
Call qatexit functions, shut down UI and kernel, and exit.

@param code (integer): exit code

### IDAPython function idaapi.qfile_t quick reference

A helper class to work with FILE related functions.

### IDAPython function idaapi.qfile_t_from_capsule quick reference

qfile_t_from_capsule(pycapsule) -> qfile_t

@param pycapsule: PyObject *

### IDAPython function idaapi.qfile_t_from_fp quick reference

qfile_t_from_fp(fp) -> qfile_t

@param fp: FILE *

### IDAPython function idaapi.qfile_t_tmpfile quick reference

qfile_t_tmpfile() -> qfile_t

### IDAPython function idaapi.qlgetz quick reference

qlgetz(li, fpos) -> str
Read a zero-terminated string from the input. If fpos == -1 then no seek will be
performed.

@param li: (C++: linput_t *)
@param fpos: (C++: int64)

### IDAPython function idaapi.qstrvec_t_add quick reference

qstrvec_t_add(_self, s) -> bool

@param self: PyObject *
@param s: char const *
```

```
### IDAPython function idaapi.qstrvec_t_addressof quick reference
qstrvec_t_addressof(_self, idx) -> PyObject *

@param self: PyObject *
@param idx: size_t
### IDAPython function idaapi.qstrvec_t_assign quick reference
qstrvec_t_assign(_self, other) -> bool

@param self: PyObject *
@param other: PyObject *
### IDAPython function idaapi.qstrvec_t_clear quick reference
qstrvec_t_clear(_self, qclear) -> bool

@param self: PyObject *
@param qclear: bool
### IDAPython function idaapi.qstrvec_t_create quick reference
qstrvec_t_create() -> PyObject *
### IDAPython function idaapi.qstrvec_t_destroy quick reference
qstrvec_t_destroy(py_obj) -> bool

@param py_obj: PyObject *
### IDAPython function idaapi.qstrvec_t_from_list quick reference
qstrvec_t_from_list(_self, py_list) -> bool

@param self: PyObject *
@param py_list: PyObject *
### IDAPython function idaapi.qstrvec_t_get quick reference
qstrvec_t_get(_self, idx) -> PyObject *

@param self: PyObject *
@param idx: size_t
### IDAPython function idaapi.qstrvec_t_get_clink quick reference
qstrvec_t_get_clink(_self) -> qstrvec_t *

@param self: PyObject *
### IDAPython function idaapi.qstrvec_t_get_clink_ptr quick reference
```

```
qstrvec_t_get_clink_ptr(_self) -> PyObject *

@param self: PyObject *

### IDAPython function idaapi.qstrvec_t_insert quick reference

qstrvec_t_insert(_self, idx, s) -> bool

@param self: PyObject *
@param idx: size_t
@param s: char const *

### IDAPython function idaapi.qstrvec_t_remove quick reference

qstrvec_t_remove(_self, idx) -> bool

@param self: PyObject *
@param idx: size_t

### IDAPython function idaapi.qstrvec_t_set quick reference

qstrvec_t_set(_self, idx, s) -> bool

@param self: PyObject *
@param idx: size_t
@param s: char const *

### IDAPython function idaapi.qstrvec_t_size quick reference

qstrvec_t_size(_self) -> size_t

@param self: PyObject *

### IDAPython function idaapi.qswap quick reference

qswap(a, b)

@param a: cinsn_t &
@param b: cinsn_t &

### IDAPython function idaapi.qthread_equal quick reference

qthread_equal(q1, q2) -> bool
Are two threads equal?

@param q1: (C++: qthread_t)
@param q2: (C++: qthread_t)

### IDAPython function idaapi.quote_cmdline_arg quick reference

quote_cmdline_arg(arg) -> bool
Quote a command line argument if it contains escape characters. For example, *.c
will be converted into "*.c" because * may be inadvertently expanded by the
```

```
shell

@param arg: (C++: qstring *)
@return: true: modified 'arg'

### IDAPython function idaapi.qvector_reserve quick reference

qvector_reserve(vec, old, cnt, elsize) -> void *
Change capacity of given qvector.

@param vec : a pointer to a qvector
@param old : a pointer to the qvector's array
@param cnt (integer): number of elements to reserve
@param elsize (integer): size of each element
@return: a pointer to the newly allocated array

### IDAPython function idaapi.qword_flag quick reference

qword_flag() -> flags64_t
Get a flags64_t representing a quad word.

### IDAPython function idaapi.range_t_print quick reference

range_t_print(cb) -> str
Helper function. Should not be called directly!

@param cb: range_t const *

### IDAPython function idaapi.read_dbg_memory quick reference

read_dbg_memory(ea, buffer, size) -> ssize_t

@param ea: ea_t
@param buffer
@param size: size_t

### IDAPython function idaapi.read_ioports2 quick reference

read_ioports2(ports, device, file, callback=None) -> ssize_t

@param ports: ioports_t *
@param device: qstring *
@param file: char const *
@param callback: ioports_fallback_t *

### IDAPython function idaapi.read_range_selection quick reference

read_range_selection(v) -> bool
Get the address range for the selected range boundaries, this is the convenient
function for read_selection()

@param v(a Widget SWIG wrapper class): view, nullptr means the last active window containing
```

```
@retval 0: no range is selected
@retval 1: ok, start ea and end ea are filled

### IDAPython function idaapi.read_regargs quick reference

read_regargs(pfn)

@param pfn: func_t *

### IDAPython function idaapi.read_selection quick reference

read_selection(v, p1, p2) -> bool
Read the user selection, and store its information in p0 (from) and p1 (to).

This can be used as follows:


>>> p0 = idaapi.twinpos_t()
p1 = idaapi.twinpos_t()
view = idaapi.get_current_viewer()
idaapi.read_selection(view, p0, p1)


At that point, p0 and p1 hold information for the selection.
But, the 'at' property of p0 and p1 is not properly typed.
To specialize it, call #place() on it, passing it the view
they were retrieved from. Like so:


>>> place0 = p0.place(view)
place1 = p1.place(view)


This will effectively "cast" the place into a specialized type,
holding proper information, depending on the view type (e.g.,
disassembly, structures, enums, ...)

@param view: The view to retrieve the selection for.
@param p1: Storage for the "to" part of the selection.
@param p1: Storage for the "to" part of the selection.
@return: a bool value indicating success.

### IDAPython function idaapi.read_tinfo_bitfield_value quick reference

read_tinfo_bitfield_value(typid, v, bitoff) -> uint64

@param typid: uint32
@param v: uint64
@param bitoff: int
```

```
### IDAPython function idaapi.readbytes quick reference

readbytes(h, res, size, mf) -> int
Read at most 4 bytes from file.

@param h (integer): file handle
@param res: (C++: uint32 *) value read from file
@param size (integer): size of value in bytes (1,2,4)
@param mf (bool): is MSB first?
@return: 0 on success, nonzero otherwise

### IDAPython function idaapi.reanalyze_callers quick reference

reanalyze_callers(ea, noret)
Plan to reanalyze callers of the specified address. This function will add to
AU_USED queue all instructions that call (not jump to) the specified address.

@param ea (integer): linear address of callee
@param noret (bool): !=0: the callee doesn't return, mark to undefine subsequent
              instructions in the caller. 0: do nothing.

### IDAPython function idaapi.reanalyze_function quick reference

reanalyze_function(pfn, ea1=0, ea2=BADADDR, analyze_parents=False)
Reanalyze a function. This function plans to analyzes all chunks of the given
function. Optional parameters (ea1, ea2) may be used to narrow the analyzed
range.

@param pfn (idaapi.func_t): pointer to a function
@param ea1 (integer): start of the range to analyze
@param ea2 (integer): end of range to analyze
@param analyze_parents (bool): meaningful only if pfn points to a function tail. if
                        true, all tail parents will be reanalyzed. if false,
                        only the given tail will be reanalyzed.

### IDAPython function idaapi.reanalyze_noret_flag quick reference

reanalyze_noret_flag(ea) -> bool
Plan to reanalyze noret flag. This function does not remove FUNC_NORET if it is
already present. It just plans to reanalysis.

@param ea (integer):

### IDAPython function idaapi.rebase_program quick reference

rebase_program(delta, flags) -> int
Rebase the whole program by 'delta' bytes.

@param delta (integer): number of bytes to move the program
@param flags (integer): Move segment flags it is recommended to use MSF_FIXONCE so that
```

```
              the loader takes care of global variables it stored in the
              database
@return: Move segment result codes

### IDAPython function idaapi.rebuild_nlist quick reference

rebuild_nlist()
Rebuild the name list.

### IDAPython function idaapi.recalc_spd quick reference

recalc_spd(cur_ea) -> bool
Recalculate SP delta for an instruction that stops execution. The next
instruction is not reached from the current instruction. We need to recalculate
SP for the next instruction.

This function will create a new automatic SP register change point if necessary.
It should be called from the emulator (emu.cpp) when auto_state == AU_USED if
the current instruction doesn't pass the execution flow to the next instruction.

@param cur_ea (integer): linear address of the current instruction
@retval 1: new stkpnt is added
@retval 0: nothing is changed

### IDAPython function idaapi.refresh_chooser quick reference

refresh_chooser(title) -> bool
Mark a non-modal custom chooser for a refresh (ui_refresh_chooser).

@param title (string): title of chooser
@return: success

### IDAPython function idaapi.refresh_choosers quick reference

refresh_choosers()

### IDAPython function idaapi.refresh_custom_viewer quick reference

refresh_custom_viewer(custom_viewer)
Refresh custom ida viewer (ui_refresh_custom_viewer)

@param custom_viewer(a Widget SWIG wrapper class):

### IDAPython function idaapi.refresh_debugger_memory quick reference

refresh_debugger_memory() -> PyObject *
Refreshes the debugger memory

@return: Nothing

### IDAPython function idaapi.refresh_idaview quick reference
```

```
refresh_idaview()
Refresh marked windows (ui_refreshmarked)

### IDAPython function idaapi.refresh_idaview_anyway quick reference

refresh_idaview_anyway()
Refresh all disassembly views (ui_refresh), forces an immediate refresh. Please
consider request_refresh() instead

### IDAPython function idaapi.refresh_navband quick reference

refresh_navband(force)
Refresh navigation band if changed (ui_refresh_navband).

@param force (bool): refresh regardless

### IDAPython function idaapi.refresh_viewer quick reference

refresh_viewer(gv)
Redraw the graph in the given view.

@param gv: (C++: graph_viewer_t *)

### IDAPython function idaapi.reg2mreg quick reference

reg2mreg(reg) -> mreg_t
Map a processor register to a microregister.

@param reg (integer): processor register number
@return: microregister register id or mr_none

### IDAPython function idaapi.reg_data_type quick reference

reg_data_type(name, subkey=None) -> regval_type_t
Get data type of a given value.

@param name (string): value name
@param subkey (string): key name
@return: false if the [key+]value doesn't exist

### IDAPython function idaapi.reg_delete quick reference

reg_delete(name, subkey=None) -> bool
Delete a value from the registry.

@param name (string): value name
@param subkey (string): parent key
@return: success

### IDAPython function idaapi.reg_delete_subkey quick reference

reg_delete_subkey(name) -> bool
Delete a key from the registry.
```

```
@param name (string): char const *

### IDAPython function idaapi.reg_delete_tree quick reference

reg_delete_tree(name) -> bool
Delete a subtree from the registry.

@param name (string): char const *

### IDAPython function idaapi.reg_exists quick reference

reg_exists(name, subkey=None) -> bool
Is there already a value with the given name?

@param name (string): value name
@param subkey (string): parent key

### IDAPython function idaapi.reg_flush quick reference

reg_flush()

### IDAPython function idaapi.reg_load quick reference

reg_load()

### IDAPython function idaapi.reg_read_binary quick reference

reg_read_binary(name, subkey=None) -> bytes or None
Read binary data from the registry.

@param name (string): value name
@param subkey (string): key name
@return: success

### IDAPython function idaapi.reg_read_bool quick reference

reg_read_bool(name, defval, subkey=None) -> bool
Read boolean value from the registry.

@param name (string): value name
@param defval (bool): default value
@param subkey (string): key name
@return: boolean read from registry, or 'defval' if the read failed

### IDAPython function idaapi.reg_read_int quick reference

reg_read_int(name, defval, subkey=None) -> int
Read integer value from the registry.

@param name (string): value name
@param defval (integer): default value
@param subkey (string): key name
```

```
@return: the value read from the registry, or 'defval' if the read failed

### IDAPython function idaapi.reg_read_string quick reference

reg_read_string(name, subkey=None, _def=None) -> str
Read a string from the registry.

@param name (string): value name
@param subkey (string): key name
@param def: char const *
@return: success

### IDAPython function idaapi.reg_read_strlist quick reference

reg_read_strlist(subkey)
Retrieve all string values associated with the given key. Also see
reg_update_strlist().

@param subkey (string): char const *

### IDAPython function idaapi.reg_subkey_exists quick reference

reg_subkey_exists(name) -> bool
Is there already a key with the given name?

@param name (string): char const *

### IDAPython function idaapi.reg_subkey_subkeys quick reference

reg_subkey_subkeys(name) -> [str, ...]
Get all subkey names of given key.

@param name (string): char const *

### IDAPython function idaapi.reg_subkey_values quick reference

reg_subkey_values(name) -> [str, ...]
Get all value names under given key.

@param name (string): char const *

### IDAPython function idaapi.reg_update_filestrlist quick reference

reg_update_filestrlist(subkey, add, maxrecs, rem=None)
Update registry with a file list. Case sensitivity will vary depending on the
target OS.
@note: 'add' and 'rem' must be UTF-8, just like for regular string operations.

@param subkey (string): char const *
@param add (string): char const *
@param maxrecs (integer):
@param rem (string): char const *
```

```
### IDAPython function idaapi.reg_update_strlist quick reference

reg_update_strlist(subkey, add, maxrecs, rem=None, ignorecase=False)
Update list of strings associated with given key.

@param subkey (string): key name
@param add (string): string to be added to list, can be nullptr
@param maxrecs (integer): limit list to this size
@param rem (string): string to be removed from list, can be nullptr
@param ignorecase (bool): ignore case for 'add' and 'rem'

### IDAPython function idaapi.reg_write_binary quick reference

reg_write_binary(name, py_bytes, subkey=None) -> PyObject *
Write binary data to the registry.

@param name (string): value name
@param py_bytes: PyObject *
@param subkey (string): key name

### IDAPython function idaapi.reg_write_bool quick reference

reg_write_bool(name, value, subkey=None)
Write boolean value to the registry.

@param name (string): value name
@param value (integer): boolean to write (nonzero = true)
@param subkey (string): key name

### IDAPython function idaapi.reg_write_int quick reference

reg_write_int(name, value, subkey=None)
Write integer value to the registry.

@param name (string): value name
@param value (integer): value to write
@param subkey (string): key name

### IDAPython function idaapi.reg_write_string quick reference

reg_write_string(name, utf8, subkey=None)
Write a string to the registry.

@param name (string): value name
@param utf8 (string): utf8-encoded string
@param subkey (string): key name

### IDAPython function idaapi.register_action quick reference

register_action(desc) -> bool
Create a new action (ui_register_action). After an action has been created, it
```

is possible to attach it to menu items (attach_action_to_menu()), or to popup menus (attach_action_to_popup()).

Because the actions will need to call the handler's activate() and update() methods at any time, you shouldn't build your action handler on the stack.

Please see the SDK's "ht_view" plugin for an example how to register actions.

@param desc: (C++: const action_desc_t &) action to register
@return: success

IDAPython function idaapi.register_addon quick reference

register_addon(info) -> int
Register an add-on. Show its info in the About box. For plugins, should be called from init() function (repeated calls with the same product code overwrite previous entries) returns: index of the add-on in the list, or -1 on error

@param info: (C++: const addon_info_t *) addon_info_t const *

IDAPython function idaapi.register_and_attach_to_menu quick reference

register_and_attach_to_menu(menupath, name, label, shortcut, flags, handler, owner, action_d
Helper.

You are not encouraged to use this, as it mixes flags for both register_action(), and attach_action_to_menu().

The only reason for its existence is to make it simpler to port existing plugins to the new actions API.

@param menupath (string): char const *
@param name (string): char const *
@param label (string): char const *
@param shortcut (string): char const *
@param flags (integer):
@param handler: (C++: action_handler_t *)
@param owner :
@param action_desc_t_flags (integer):

IDAPython function idaapi.register_cfgopts quick reference

register_cfgopts(opts, nopts, cb=None, obj=None) -> bool

@param opts: cfgopt_t const []
@param nopts: size_t
@param cb: config_changed_cb_t *
@param obj

```
### IDAPython function idaapi.register_custom_data_format quick reference

register_custom_data_format(py_df) -> int
Registers a custom data format with a given data type.

@param py_df: an instance of data_format_t
@return:     < 0 if failed to register
    > 0 data format id

### IDAPython function idaapi.register_custom_data_type quick reference

register_custom_data_type(py_dt) -> int
Registers a custom data type.

@param py_dt: an instance of the data_type_t class
@return:     < 0 if failed to register
    > 0 data type id

### IDAPython function idaapi.register_data_types_and_formats quick reference

Registers multiple data types and formats at once.
To register one type/format at a time use register_custom_data_type/register_custom_data_for

It employs a special table of types and formats described below:

The 'formats' is a list of tuples. If a tuple has one element then it is the format to be re
If the tuple has more than one element, then tuple[0] is the data type and tuple[1:] are the
many_formats = [
  (pascal_data_type(), pascal_data_format()),
  (simplevm_data_type(), simplevm_data_format()),
  (makedword_data_format(),),
  (simplevm_data_format(),)
]
The first two tuples describe data types and their associated formats.
The last two tuples describe two data formats to be used with built-in data types.
The data format may be attached to several data types. The id of the
data format is stored in the first data_format_t object. For example:
assert many_formats[1][1] != -1
assert many_formats[2][0] != -1
assert many_formats[3][0] == -1

### IDAPython function idaapi.register_timer quick reference

register_timer(interval, py_callback) -> PyCapsule
Register a timer

@param interval: Interval in milliseconds
@param py_callback: A Python callable that takes no parameters and returns an integer.
                The callback may return:
```

```
                -1   : to unregister the timer
                >= 0 : the new or same timer interval
@return: None or a timer object

### IDAPython function idaapi.reload_file quick reference

reload_file(file, is_remote) -> bool
Reload the input file. This function reloads the byte values from the input
file. It doesn't modify the segmentation, names, comments, etc.

@param file (string): name of the input file. if file == nullptr then returns:
* 1: can reload the input file
* 0: can't reload the input file
@param is_remote (bool): is the file located on a remote computer with the debugger
                 server?
@return: success

### IDAPython function idaapi.reloc_value quick reference

reloc_value(value, size, delta, mf)

@param value
@param size: int
@param delta: adiff_t
@param mf: bool

### IDAPython function idaapi.relocate_relobj quick reference

relocate_relobj(_relobj, ea, mf) -> bool

@param _relobj: relobj_t *
@param ea: ea_t
@param mf: bool

### IDAPython function idaapi.remember_problem quick reference

remember_problem(type, ea, msg=None)
Insert an address to a list of problems. Display a message saying about the
problem (except of PR_ATTN,PR_FINAL) PR_JUMP is temporarily ignored.

@param type: (C++: problist_id_t) problem list type
@param ea (integer): linear address
@param msg (string): a user-friendly message to be displayed instead of the default more
           generic one associated with the type of problem. Defaults to
           nullptr.

### IDAPython function idaapi.remitem quick reference

remitem(e)

@param e: citem_t const *
```

IDAPython function idaapi.remove_abi_opts quick reference

remove_abi_opts(abi_opts, user_level=False) -> bool

@param abi_opts: char const *
@param user_level: bool

IDAPython function idaapi.remove_command_interpreter quick reference

remove_command_interpreter(cli_idx)
Remove command line interpreter (ui_install_cli)

@param cli_idx: int

IDAPython function idaapi.remove_func_tail quick reference

remove_func_tail(pfn, tail_ea) -> bool
Remove a function tail. If the tail belongs only to one function, it will be completely removed. Otherwise if the function was the tail owner, the first function using this tail becomes the owner of the tail.

@param pfn (idaapi.func_t): pointer to the function
@param tail_ea (integer): any address inside the tail to remove

IDAPython function idaapi.remove_hexrays_callback quick reference

Deprecated. Please use Hexrays_Hooks instead
Uninstall handler for decompiler events.

@return: number of uninstalled handlers.

IDAPython function idaapi.remove_pointer quick reference

remove_pointer(tif) -> tinfo_t

@param BT_PTR: If the current type is a pointer, return the pointed object. If the current type is not a pointer, return the current type. See also get_ptrarr_object() and get_pointed_object()

IDAPython function idaapi.remove_tinfo_pointer quick reference

remove_tinfo_pointer(tif, name, til) -> (bool, NoneType), (bool, str)
Remove pointer of a type. (i.e. convert "char *" into "char"). Optionally remove the "lp" (or similar) prefix of the input name. If the input type is not a pointer, then fail.

@param tif: (C++: tinfo_t *)
@param name: char const *
@param til (idaapi.til_t): til_t const *

IDAPython function idaapi.rename_bptgrp quick reference

```
rename_bptgrp(old_name, new_name) -> bool
Rename a folder of bpt dirtree \sq{Type, Synchronous function, Notification,
none (synchronous function)}

@param old_name (string): absolute path to the folder to be renamed
@param new_name (string): absolute path of the new folder name
@return: success

### IDAPython function idaapi.rename_encoding quick reference

rename_encoding(idx, encname) -> bool
Change name for an encoding The number of bytes per unit (BPU) of the new
encoding must match this number of the existing default encoding. Specifying the
empty name simply deletes this encoding.

@param idx (integer): the encoding index (1-based)
@param encname (string): the new encoding name

### IDAPython function idaapi.rename_entry quick reference

rename_entry(ord, name, flags=0) -> bool
Rename entry point.

@param ord (integer): ordinal number of the entry point
@param name (string): name of entry point. If the specified location already has a name,
             the old name will be appended to a repeatable comment.
@param flags (integer): See AEF_*
@return: success

### IDAPython function idaapi.rename_lvar quick reference

rename_lvar(func_ea, oldname, newname) -> bool
Rename a local variable.

@param func_ea (integer): function start address
@param oldname (string): old name of the variable
@param newname (string): new name of the variable
@return: success This is a convenience function. For bulk renaming consider
         using modify_user_lvars.

### IDAPython function idaapi.rename_regvar quick reference

rename_regvar(pfn, v, user) -> int
Rename a register variable.

@param pfn (idaapi.func_t): function in question
@param v: (C++: regvar_t *) variable to rename
@param user (string): new user-defined name for the register
@return: Register variable error codes
```

```
### IDAPython function idaapi.reorder_dummy_names quick reference

reorder_dummy_names()
Renumber dummy names.

### IDAPython function idaapi.repaint_custom_viewer quick reference

repaint_custom_viewer(custom_viewer)
Repaint the given widget immediately (ui_repaint_qwidget)

@param custom_viewer(a Widget SWIG wrapper class):

### IDAPython function idaapi.replace_ordinal_typerefs quick reference

replace_ordinal_typerefs(til, tif) -> int
Replace references to ordinal types by name references. This function 'unties'
the type from the current local type library and makes it easier to export it.

@param til (idaapi.til_t): type library to use. may be nullptr.
@param tif: (C++: tinfo_t *) type to modify (in/out)
@retval number: of replaced subtypes, -1 on failure

### IDAPython function idaapi.replace_wait_box quick reference

replace_wait_box(format)
Replace the label of "Please wait dialog box".

@param format (string): char const *

### IDAPython function idaapi.request_add_bpt quick reference

request_add_bpt(ea, size=0, type=BPT_DEFAULT) -> bool
Post an add_bpt(const bpt_t &) request.

@param ea: ea_t
@param size: asize_t
@param type: bpttype_t

request_add_bpt(bpt) -> bool

@param bpt: bpt_t const &

### IDAPython function idaapi.request_attach_process quick reference

request_attach_process(pid, event_id) -> int
Post an attach_process() request.

@param pid (integer):
@param event_id (integer):

### IDAPython function idaapi.request_clear_trace quick reference
```

```
request_clear_trace()
Post a clear_trace() request.

### IDAPython function idaapi.request_continue_process quick reference

request_continue_process() -> bool
Post a continue_process() request.
@note: This requires an explicit call to run_requests()

### IDAPython function idaapi.request_del_bpt quick reference

request_del_bpt(ea) -> bool
Post a del_bpt(const bpt_location_t &) request.

@param ea: ea_t

request_del_bpt(bptloc) -> bool

@param bptloc: bpt_location_t const &

### IDAPython function idaapi.request_detach_process quick reference

request_detach_process() -> bool
Post a detach_process() request.

### IDAPython function idaapi.request_disable_bblk_trace quick reference

request_disable_bblk_trace() -> bool

### IDAPython function idaapi.request_disable_bpt quick reference

request_disable_bpt(ea) -> bool

@param ea: ea_t

request_disable_bpt(bptloc) -> bool

@param bptloc: bpt_location_t const &

### IDAPython function idaapi.request_disable_func_trace quick reference

request_disable_func_trace() -> bool

### IDAPython function idaapi.request_disable_insn_trace quick reference

request_disable_insn_trace() -> bool

### IDAPython function idaapi.request_disable_step_trace quick reference

request_disable_step_trace() -> bool

### IDAPython function idaapi.request_enable_bblk_trace quick reference

request_enable_bblk_trace(enable=True) -> bool
```

```
@param enable: bool

### IDAPython function idaapi.request_enable_bpt quick reference

request_enable_bpt(ea, enable=True) -> bool

@param ea: ea_t
@param enable: bool

request_enable_bpt(bptloc, enable=True) -> bool

@param bptloc: bpt_location_t const &
@param enable: bool

### IDAPython function idaapi.request_enable_func_trace quick reference

request_enable_func_trace(enable=True) -> bool

@param enable: bool

### IDAPython function idaapi.request_enable_insn_trace quick reference

request_enable_insn_trace(enable=True) -> bool

@param enable: bool

### IDAPython function idaapi.request_enable_step_trace quick reference

request_enable_step_trace(enable=1) -> bool

@param enable: int

### IDAPython function idaapi.request_exit_process quick reference

request_exit_process() -> bool
Post an exit_process() request.

### IDAPython function idaapi.request_refresh quick reference

request_refresh(mask, cnd=True)
Request a refresh of a builtin window.

@param mask (integer): Window refresh flags
@param cnd (bool): set if true or clear flag otherwise

### IDAPython function idaapi.request_resume_thread quick reference

request_resume_thread(tid) -> int
Post a resume_thread() request.

@param tid (integer):

### IDAPython function idaapi.request_run_to quick reference
```

```
request_run_to(ea, pid=pid_t(-1), tid=0) -> bool
Post a run_to() request.

@param ea (integer):
@param pid (integer):

### IDAPython function idaapi.request_select_thread quick reference

request_select_thread(tid) -> bool
Post a select_thread() request.

@param tid (integer):

### IDAPython function idaapi.request_set_bblk_trace_options quick reference

request_set_bblk_trace_options(options)
Post a set_bblk_trace_options() request.

@param options (integer):

### IDAPython function idaapi.request_set_func_trace_options quick reference

request_set_func_trace_options(options)
Post a set_func_trace_options() request.

@param options (integer):

### IDAPython function idaapi.request_set_insn_trace_options quick reference

request_set_insn_trace_options(options)
Post a set_insn_trace_options() request.

@param options (integer):

### IDAPython function idaapi.request_set_reg_val quick reference

request_set_reg_val(regname, o) -> PyObject *
Post a set_reg_val() request.

@param regname (string): char const *
@param o: PyObject *

### IDAPython function idaapi.request_set_resume_mode quick reference

request_set_resume_mode(tid, mode) -> bool
Post a set_resume_mode() request.

@param tid (integer):
@param mode: (C++: resume_mode_t) enum resume_mode_t

### IDAPython function idaapi.request_set_step_trace_options quick reference
```

```
request_set_step_trace_options(options)
Post a set_step_trace_options() request.

@param options (integer):

### IDAPython function idaapi.request_start_process quick reference

request_start_process(path=None, args=None, sdir=None) -> int
Post a start_process() request.

@param path (string): char const *
@param args (string): char const *
@param sdir (string): char const *

### IDAPython function idaapi.request_step_into quick reference

request_step_into() -> bool
Post a step_into() request.

### IDAPython function idaapi.request_step_over quick reference

request_step_over() -> bool
Post a step_over() request.

### IDAPython function idaapi.request_step_until_ret quick reference

request_step_until_ret() -> bool
Post a step_until_ret() request.

### IDAPython function idaapi.request_suspend_process quick reference

request_suspend_process() -> bool
Post a suspend_process() request.

### IDAPython function idaapi.request_suspend_thread quick reference

request_suspend_thread(tid) -> int
Post a suspend_thread() request.

@param tid (integer):

### IDAPython function idaapi.require quick reference

Load, or reload a module.

When under heavy development, a user's tool might consist of multiple
modules. If those are imported using the standard 'import' mechanism,
there is no guarantee that the Python implementation will re-read
and re-evaluate the module's Python code. In fact, it usually doesn't.
What should be done instead is 'reload()'-ing that module.

This is a simple helper function that will do just that: In case the
module doesn't exist, it 'import's it, and if it does exist,
```

```
'reload()'s it.

The importing module (i.e., the module calling require()) will have
the loaded module bound to its globals(), under the name 'modulename'.
(If require() is called from the command line, the importing module
will be '__main__'.)

For more information, see: <http://www.hexblog.com/?p=749>.

### IDAPython function idaapi.requires_color_esc quick reference

Checks if the given character requires escaping
Is the given char a color escape character?

### IDAPython function idaapi.resolve_typedef quick reference

resolve_typedef(til, type) -> type_t const *

@param til: til_t const *
@param type: type_t const *

### IDAPython function idaapi.restore_database_snapshot quick reference

restore_database_snapshot(ss, pyfunc_or_none, pytuple_or_none) -> bool
Restore a database snapshot. Note: This call is asynchronous. When it is
completed, the callback will be triggered.

@param ss: (C++: const snapshot_t *) snapshot instance (see build_snapshot_tree())
@param pyfunc_or_none: PyObject *
@param pytuple_or_none: PyObject *
@return: false if restoration could not be started (snapshot file was not
         found).
If the returned value is True then check if the operation succeeded from the
callback.

### IDAPython function idaapi.restore_user_cmts quick reference

restore_user_cmts(func_ea) -> user_cmts_t
Restore user defined comments from the database.

@param func_ea (integer): the entry address of the function
@return: collection of user defined comments. The returned object must be
         deleted by the caller using delete_user_cmts()

### IDAPython function idaapi.restore_user_defined_calls quick reference

restore_user_defined_calls(udcalls, func_ea) -> bool
Restore user defined function calls from the database.

@param udcalls: (C++: udcall_map_t *) ptr to output buffer
@param func_ea (integer): entry address of the function
```

```
@return: success

### IDAPython function idaapi.restore_user_iflags quick reference

restore_user_iflags(func_ea) -> user_iflags_t
Restore user defined citem iflags from the database.

@param func_ea (integer): the entry address of the function
@return: collection of user defined iflags. The returned object must be deleted
         by the caller using delete_user_iflags()

### IDAPython function idaapi.restore_user_labels quick reference

restore_user_labels(func_ea) -> user_labels_t
Restore user defined labels from the database.

@param func_ea (integer): the entry address of the function, ignored if FUNC != nullptr
@return: collection of user defined labels. The returned object must be deleted
         by the caller using delete_user_labels()

### IDAPython function idaapi.restore_user_labels2 quick reference

restore_user_labels2(func_ea, func=None) -> user_labels_t

@param func_ea: ea_t
@param func: cfunc_t const *

### IDAPython function idaapi.restore_user_lvar_settings quick reference

restore_user_lvar_settings(lvinf, func_ea) -> bool
Restore user defined local variable settings in the database.

@param lvinf: (C++: lvar_uservec_t *) ptr to output buffer
@param func_ea (integer): entry address of the function
@return: success

### IDAPython function idaapi.restore_user_numforms quick reference

restore_user_numforms(func_ea) -> user_numforms_t
Restore user defined number formats from the database.

@param func_ea (integer): the entry address of the function
@return: collection of user defined number formats. The returned object must be
         deleted by the caller using delete_user_numforms()

### IDAPython function idaapi.restore_user_unions quick reference

restore_user_unions(func_ea) -> user_unions_t
Restore user defined union field selections from the database.

@param func_ea (integer): the entry address of the function
@return: collection of union field selections The returned object must be
```

```
        deleted by the caller using delete_user_unions()

### IDAPython function idaapi.resume_thread quick reference

resume_thread(tid) -> int
Resume thread. \sq{Type, Synchronous function - available as request,
Notification, none (synchronous function)}

@param tid (integer): thread id
@retval -1: network error
@retval 0: failed
@retval 1: ok

### IDAPython function idaapi.retrieve_exceptions quick reference

retrieve_exceptions() -> excvec_t
Retrieve the exception information. You may freely modify the returned vector
and add/edit/delete exceptions You must call store_exceptions() after any
modifications Note: exceptions with code zero, multiple exception codes or names
are prohibited

### IDAPython function idaapi.retrieve_input_file_crc32 quick reference

retrieve_input_file_crc32() -> uint32
Get input file crc32 stored in the database. it can be used to check that the
input file has not been changed.

### IDAPython function idaapi.retrieve_input_file_md5 quick reference

retrieve_input_file_md5() -> bytes
Get input file md5.

### IDAPython function idaapi.retrieve_input_file_sha256 quick reference

retrieve_input_file_sha256() -> bytes
Get input file sha256.

### IDAPython function idaapi.retrieve_input_file_size quick reference

retrieve_input_file_size() -> size_t
Get size of input file in bytes.

### IDAPython function idaapi.retrieve_member_info quick reference

retrieve_member_info(buf, mptr) -> opinfo_t
Get operand type info for member.

@param buf: (C++: opinfo_t *)
@param mptr: (C++: const member_t *) member_t const *

### IDAPython function idaapi.revert_byte quick reference

revert_byte(ea) -> bool
Revert patched byte
```

```
@retval true: byte was patched before and reverted now

@param ea (integer):

### IDAPython function idaapi.revert_ida_decisions quick reference

revert_ida_decisions(ea1, ea2)
Delete all analysis info that IDA generated for for the given range.

@param ea1 (integer):
@param ea2 (integer):

### IDAPython function idaapi.run_plugin quick reference

run_plugin(plg, arg) -> bool
Runs a plugin

@param plg: A plugin object (returned by load_plugin())
@param arg: size_t
@return: Boolean

### IDAPython function idaapi.run_requests quick reference

run_requests() -> bool
Execute requests until all requests are processed or an asynchronous function is
called. \sq{Type, Synchronous function, Notification, none (synchronous
function)}

@return: false if not all requests could be processed (indicates an asynchronous
         function was started)
@note: If called from a notification handler, the execution of requests will be
       postponed to the end of the execution of all notification handlers.

### IDAPython function idaapi.run_to quick reference

run_to(ea, pid=pid_t(-1), tid=0) -> bool
Execute the process until the given address is reached. If no process is active,
a new process is started. Technically, the debugger sets up a temporary
breakpoint at the given address, and continues (or starts) the execution of the
whole process. So, all threads continue their execution! \sq{Type, Asynchronous
function - available as Request, Notification, dbg_run_to}

@param ea (integer): target address
@param pid (integer): not used yet. please do not specify this parameter.

### IDAPython function idaapi.save_database quick reference

save_database(outfile, flags, root=None, attr=None) -> bool
Save current database using a new file name.

@param outfile (string): output database file name
```

```
@param flags (integer): Database flags
@param root: (C++: const snapshot_t *) optional: snapshot tree root.
@param attr: (C++: const snapshot_t *) optional: snapshot attributes
@note: when both root and attr are not nullptr then the snapshot attributes will
       be updated, otherwise the snapshot attributes will be inherited from the
       current database.
@return: success
```

```
### IDAPython function idaapi.save_struc quick reference
```

```
save_struc(sptr, may_update_ltypes=True)
Update struct information in the database (internal function)

@param sptr: (C++: struc_t *)
@param may_update_ltypes (bool):
```

```
### IDAPython function idaapi.save_tinfo quick reference
```

```
save_tinfo(tif, til, ord, name, ntf_flags) -> tinfo_code_t

@param tif: tinfo_t *
@param til: til_t *
@param ord: size_t
@param name: char const *
@param ntf_flags: int
```

```
### IDAPython function idaapi.save_tinfo2 quick reference
```

```
save_tinfo2(tif, til, ord, name, cmt, ntf_flags) -> tinfo_code_t

@param tif: tinfo_t *
@param til: til_t *
@param ord: size_t
@param name: char const *
@param cmt: char const *
@param ntf_flags: int
```

```
### IDAPython function idaapi.save_trace_file quick reference
```

```
save_trace_file(filename, description) -> bool
Save the current trace in the specified file.

@param filename (string): char const *
@param description (string): char const *
```

```
### IDAPython function idaapi.save_user_cmts quick reference
```

```
save_user_cmts(func_ea, user_cmts)
Save user defined comments into the database.

@param func_ea (integer): the entry address of the function
```

@param user_cmts: (C++: const user_cmts_t *) collection of user defined comments

IDAPython function idaapi.save_user_defined_calls quick reference

save_user_defined_calls(func_ea, udcalls)
Save user defined local function calls into the database.

@param func_ea (integer): entry address of the function
@param udcalls: (C++: const udcall_map_t &) user-specified info about user defined function

IDAPython function idaapi.save_user_iflags quick reference

save_user_iflags(func_ea, iflags)
Save user defined citem iflags into the database.

@param func_ea (integer): the entry address of the function
@param iflags: (C++: const user_iflags_t *) collection of user defined citem iflags

IDAPython function idaapi.save_user_labels quick reference

save_user_labels(func_ea, user_labels)
Save user defined labels into the database.

@param func_ea (integer): the entry address of the function, ignored if FUNC != nullptr
@param user_labels: (C++: const user_labels_t *) collection of user defined labels

IDAPython function idaapi.save_user_labels2 quick reference

save_user_labels2(func_ea, user_labels, func=None)

@param func_ea: ea_t
@param user_labels: user_labels_t const *
@param func: cfunc_t const *

IDAPython function idaapi.save_user_lvar_settings quick reference

save_user_lvar_settings(func_ea, lvinf)
Save user defined local variable settings into the database.

@param func_ea (integer): entry address of the function
@param lvinf: (C++: const lvar_uservec_t &) user-specified info about local variables

IDAPython function idaapi.save_user_numforms quick reference

save_user_numforms(func_ea, numforms)
Save user defined number formats into the database.

@param func_ea (integer): the entry address of the function
@param numforms: (C++: const user_numforms_t *) collection of user defined comments

IDAPython function idaapi.save_user_unions quick reference

```
save_user_unions(func_ea, unions)
Save user defined union field selections into the database.

@param func_ea (integer): the entry address of the function
@param unions: (C++: const user_unions_t *) collection of union field selections

### IDAPython function idaapi.score_tinfo quick reference

score_tinfo(tif) -> uint32

@param tif: tinfo_t const *

### IDAPython function idaapi.search_down quick reference

search_down(sflag) -> bool
Is the SEARCH_DOWN bit set?

@param sflag (integer):

### IDAPython function idaapi.seg_flag quick reference

seg_flag() -> flags64_t
see FF_opbits

### IDAPython function idaapi.segm_adjust_diff quick reference

segm_adjust_diff(s, delta) -> adiff_t
Truncate and sign extend a delta depending on the segment.

@param s: (C++: const segment_t *) segment_t const *
@param delta (integer):

### IDAPython function idaapi.segm_adjust_ea quick reference

segm_adjust_ea(s, ea) -> ea_t
Truncate an address depending on the segment.

@param s: (C++: const segment_t *) segment_t const *
@param ea (integer):

### IDAPython function idaapi.segtype quick reference

segtype(ea) -> uchar
Get segment type.

@param ea (integer): any linear address within the segment
@return: Segment types, SEG_UNDF if no segment found at 'ea'

### IDAPython function idaapi.sel2ea quick reference

sel2ea(selector) -> ea_t
Get mapping of a selector as a linear address.
```

```
@param selector: (C++: sel_t) number of selector to translate to linear address
@return: linear address the specified selector is mapped to. if there is no
         mapping, returns to_ea(selector,0);

### IDAPython function idaapi.sel2para quick reference

sel2para(selector) -> ea_t
Get mapping of a selector.

@param selector: (C++: sel_t) number of selector to translate
@return: paragraph the specified selector is mapped to. if there is no mapping,
         returns 'selector'.

### IDAPython function idaapi.sel_array_frompointer quick reference

sel_array_frompointer(t) -> sel_array

@param t: sel_t *

### IDAPython function idaapi.sel_pointer_frompointer quick reference

sel_pointer_frompointer(t) -> sel_pointer

@param t: sel_t *

### IDAPython function idaapi.select_parser_by_name quick reference

select_parser_by_name(name) -> bool
Set the parser with the given name as the current parser. Pass nullptr or an
empty string to select the default parser.

@param name (string): char const *
@return: false if no parser was found with the given name

### IDAPython function idaapi.select_parser_by_srclang quick reference

select_parser_by_srclang(lang) -> bool
Set the parser that supports the given language(s) as the current parser. The
selected parser must support all languages specified by the given srclang_t.

@param lang: (C++: srclang_t)
@return: false if no such parser was found

### IDAPython function idaapi.select_thread quick reference

select_thread(tid) -> bool
Select the given thread as the current debugged thread. All thread related
execution functions will work on this thread. The process must be suspended to
select a new thread. \sq{Type, Synchronous function - available as request,
Notification, none (synchronous function)}

@param tid (integer): ID of the thread to select
```

```
@return: false if the thread doesn't exist.

### IDAPython function idaapi.select_udt_by_offset quick reference

select_udt_by_offset(udts, ops, applicator) -> int
Select UDT

@param udts: (C++: const qvector< tinfo_t > *) list of UDT tinfo_t for the selection, if nul
             UDTs from the "Local types" will be used
@param ops: (C++: const ui_stroff_ops_t &) operands
@param applicator: (C++: ui_stroff_applicator_t &) callback will be called to apply the sele
                   operand

### IDAPython function idaapi.send_database quick reference

send_database(err, silent)
Send the database to Hex-Rays. This function sends the current database to the
Hex-Rays server. The database is sent in the compressed form over an encrypted
(SSL) connection.

@param err: (C++: const hexrays_failure_t &) failure description object. Empty hexrays_failu
            used if error information is not available.
@param silent (bool): if false, a dialog box will be displayed before sending the
                database.

### IDAPython function idaapi.send_dbg_command quick reference

Send a direct command to the debugger backend, and
retrieve the result as a string.

Note: any double-quotes in 'command' must be backslash-escaped.
Note: this only works with some debugger backends: Bochs, WinDbg, GDB.

Returns: (True, <result string>) on success, or (False, <Error message string>) on failure

### IDAPython function idaapi.serialize_tinfo quick reference

serialize_tinfo(type, fields, fldcmts, tif, sudt_flags) -> bool

@param type: qtype *
@param fields: qtype *
@param fldcmts: qtype *
@param tif: tinfo_t const *
@param sudt_flags: int

### IDAPython function idaapi.set2jcnd quick reference

set2jcnd(code) -> mcode_t

@param code: enum mcode_t
```

```
### IDAPython function idaapi.set__bnot0 quick reference

set__bnot0(ea)

@param ea: ea_t

### IDAPython function idaapi.set__bnot1 quick reference

set__bnot1(ea)

@param ea: ea_t

### IDAPython function idaapi.set__invsign0 quick reference

set__invsign0(ea)

@param ea: ea_t

### IDAPython function idaapi.set__invsign1 quick reference

set__invsign1(ea)

@param ea: ea_t

### IDAPython function idaapi.set_abi_name quick reference

set_abi_name(abiname, user_level=False) -> bool
Set abi name (see Compiler IDs)

@param abiname (string): char const *
@param user_level (bool):

### IDAPython function idaapi.set_abits quick reference

set_abits(ea, bits)

@param ea: ea_t
@param bits: aflags_t

### IDAPython function idaapi.set_absbase quick reference

set_absbase(ea, x)

@param ea: ea_t
@param x: ea_t

### IDAPython function idaapi.set_aflags quick reference

set_aflags(ea, flags)

@param ea: ea_t
@param flags: aflags_t

### IDAPython function idaapi.set_align_flow quick reference
```

```
set_align_flow(ea)

@param ea: ea_t

### IDAPython function idaapi.set_alignment quick reference

set_alignment(ea, x)

@param ea: ea_t
@param x: uint32

### IDAPython function idaapi.set_archive_path quick reference

set_archive_path(file) -> bool
Set archive file path from which input file was extracted.

@param file (string): char const *

### IDAPython function idaapi.set_array_parameters quick reference

set_array_parameters(ea, _in)

@param ea: ea_t
@param in: array_parameters_t const *

### IDAPython function idaapi.set_asm_inc_file quick reference

set_asm_inc_file(file) -> bool
Set name of the include file.

@param file (string): char const *

### IDAPython function idaapi.set_auto_state quick reference

set_auto_state(new_state) -> atype_t
Set current state of autoanalyzer.

@param new_state (one of the idaapi.AU_xxxx flags): new state of autoanalyzer
@return: previous state

### IDAPython function idaapi.set_bblk_trace_options quick reference

set_bblk_trace_options(options)
Modify basic block tracing options (see BT_LOG_INSTS)

@param options (integer):

### IDAPython function idaapi.set_bmask_cmt quick reference

set_bmask_cmt(id, bmask, cmt, repeatable) -> bool

@param id: enum_t
@param bmask: bmask_t
```

```
@param cmt: char const *
@param repeatable: bool

### IDAPython function idaapi.set_bmask_name quick reference

set_bmask_name(id, bmask, name) -> bool

@param id: enum_t
@param bmask: bmask_t
@param name: char const *

### IDAPython function idaapi.set_bpt_group quick reference

set_bpt_group(bpt, grp_name) -> bool
Move a bpt into a folder in the breakpoint dirtree if the folder didn't exists,
it will be created \sq{Type, Synchronous function, Notification, none
(synchronous function)}

@param bpt: (C++: bpt_t &) bpt that will be moved
@param grp_name (string): absolute path to the breakpoint dirtree folder
@return: success

### IDAPython function idaapi.set_bptloc_group quick reference

set_bptloc_group(bptloc, grp_name) -> bool
Move a bpt into a folder in the breakpoint dirtree based on the bpt_location
find_bpt is called to retrieve the bpt and then set_bpt_group if the folder
didn't exists, it will be created \sq{Type, Synchronous function, Notification,
none (synchronous function)}

@param bptloc (idaapi.bpt_location_t): bptlocation of the bpt that will be moved
@param grp_name (string): absolute path to the breakpoint dirtree folder
@return: success

### IDAPython function idaapi.set_bptloc_string quick reference

set_bptloc_string(s) -> int

@param s: char const *

### IDAPython function idaapi.set_c_header_path quick reference

set_c_header_path(incdir)
Set include directory path the target compiler.

@param incdir (string): char const *

### IDAPython function idaapi.set_c_macros quick reference

set_c_macros(macros)
Set predefined macros for the target compiler.
```

```
@param macros (string): char const *

### IDAPython function idaapi.set_cancelled quick reference

set_cancelled()
Set "Cancelled" flag (ui_set_cancelled)

### IDAPython function idaapi.set_cmt quick reference

set_cmt(ea, comm, rptble) -> bool
Set an indented comment.

@param ea (integer): linear address
@param comm (string): comment string
* nullptr: do nothing (return 0)
* "" : delete comment
@param rptble (bool): is repeatable?
@return: success

### IDAPython function idaapi.set_code_viewer_handler quick reference

set_code_viewer_handler(code_viewer, handler_id, handler_or_data) -> void *
Set a handler for a code viewer event (ui_set_custom_viewer_handler).

@param code_viewer(a Widget SWIG wrapper class): the code viewer
@param handler_id: (C++: custom_viewer_handler_id_t) one of CDVH_ in custom_viewer_handler_i
@param handler_or_data : can be a handler or data. see examples in Functions:
                         custom viewer handlers
@return: old value of the handler or data

### IDAPython function idaapi.set_code_viewer_is_source quick reference

set_code_viewer_is_source(code_viewer) -> bool
Specify that the given code viewer is used to display source code
(ui_set_custom_viewer_handler).

@param code_viewer(a Widget SWIG wrapper class):

### IDAPython function idaapi.set_code_viewer_line_handlers quick reference

set_code_viewer_line_handlers(code_viewer, click_handler, popup_handler, dblclick_handler, c
Set handlers for code viewer line events. Any of these handlers may be nullptr

@param code_viewer(a Widget SWIG wrapper class):
@param click_handler: (C++: code_viewer_lines_click_t *)
@param popup_handler: (C++: code_viewer_lines_click_t *)
@param dblclick_handler: (C++: code_viewer_lines_click_t *)
@param drawicon_handler: (C++: code_viewer_lines_icon_t *)
@param linenum_handler: (C++: code_viewer_lines_linenum_t *)

### IDAPython function idaapi.set_code_viewer_lines_alignment quick reference
```

```
set_code_viewer_lines_alignment(code_viewer, align) -> bool
Set alignment for lines in a code viewer (ui_set_custom_viewer_handler).

@param code_viewer(a Widget SWIG wrapper class):
@param align (integer):

### IDAPython function idaapi.set_code_viewer_lines_icon_margin quick reference

set_code_viewer_lines_icon_margin(code_viewer, margin) -> bool
Set space allowed for icons in the margin of a code viewer
(ui_set_custom_viewer_handler).

@param code_viewer(a Widget SWIG wrapper class):
@param margin (integer):

### IDAPython function idaapi.set_code_viewer_lines_radix quick reference

set_code_viewer_lines_radix(code_viewer, radix) -> bool
Set radix for values displayed in a code viewer (ui_set_custom_viewer_handler).

@param code_viewer(a Widget SWIG wrapper class):
@param radix (integer):

### IDAPython function idaapi.set_code_viewer_user_data quick reference

set_code_viewer_user_data(code_viewer, ud) -> bool
Set the user data on a code viewer (ui_set_custom_viewer_handler).

@param code_viewer(a Widget SWIG wrapper class):
@param ud :

### IDAPython function idaapi.set_colored_item quick reference

set_colored_item(ea)

@param ea: ea_t

### IDAPython function idaapi.set_compiler quick reference

set_compiler(cc, flags, abiname=None) -> bool
Change current compiler.

@param cc: (C++: const compiler_info_t &) compiler to switch to
@param flags (integer): Set compiler flags
@param abiname (string): ABI name
@return: success

### IDAPython function idaapi.set_compiler_id quick reference

set_compiler_id(id, abiname=None) -> bool
Set the compiler id (see Compiler IDs)
```

```
@param id: (C++: comp_t)
@param abiname (string): char const *

### IDAPython function idaapi.set_compiler_string quick reference

set_compiler_string(compstr, user_level) -> bool

@param compstr (string): - compiler description in form <abbr>:<abiname>
@param user_level (bool): - initiated by user if TRUE
@return: success

### IDAPython function idaapi.set_cp_validity quick reference

set_cp_validity(kind, cp, endcp=wchar32_t(-1), valid=True)
Mark the given codepoint (or range) as acceptable or unacceptable in the given
context If 'endcp' is not BADCP, it is considered to be the end of the range:
[cp, endcp), and is not included in the range

@param kind: (C++: ucdr_kind_t) enum ucdr_kind_t
@param cp: (C++: wchar32_t)
@param endcp: (C++: wchar32_t)

### IDAPython function idaapi.set_custom_data_type_ids quick reference

set_custom_data_type_ids(ea, cdis)

@param ea: ea_t
@param cdis: custom_data_type_ids_t const *

### IDAPython function idaapi.set_custom_viewer_qt_aware quick reference

set_custom_viewer_qt_aware(custom_viewer) -> bool
Allow the given viewer to interpret Qt events (ui_set_custom_viewer_handler)

@param custom_viewer(a Widget SWIG wrapper class):

### IDAPython function idaapi.set_data_guessed_by_hexrays quick reference

set_data_guessed_by_hexrays(ea)

@param ea: ea_t

### IDAPython function idaapi.set_database_flag quick reference

set_database_flag(dbfl, cnd=True)
Set or clear database flag

@param dbfl (integer): flag Database flags
@param cnd (bool): set if true or clear flag otherwise

### IDAPython function idaapi.set_debug_event_code quick reference
```

```
set_debug_event_code(ev, id)

@param ev: debug_event_t *
@param id: enum event_id_t

### IDAPython function idaapi.set_debug_name quick reference

set_debug_name(ea, name) -> bool

@param ea: ea_t
@param name: char const *

### IDAPython function idaapi.set_debugger_event_cond quick reference

set_debugger_event_cond(NONNULL_evcond)

@param NONNULL_evcond: char const *

### IDAPython function idaapi.set_debugger_options quick reference

set_debugger_options(options) -> uint
Set debugger options. Replaces debugger options with the specification
combination Debugger options

@param options (integer):
@return: the old debugger options

### IDAPython function idaapi.set_default_dataseg quick reference

set_default_dataseg(ds_sel)
Set default value of DS register for all segments.

@param ds_sel: (C++: sel_t)

### IDAPython function idaapi.set_default_encoding_idx quick reference

set_default_encoding_idx(bpu, idx) -> bool
Set default encoding for a string type

@param bpu (integer): the amount of bytes per unit
@param idx (integer): the encoding index. It cannot be 0

### IDAPython function idaapi.set_default_sreg_value quick reference

set_default_sreg_value(sg, rg, value) -> bool
Set default value of a segment register for a segment.

@param sg: (C++: segment_t *) pointer to segment structure if nullptr, then set the registe
           all segments
@param rg (integer): number of segment register
@param value: (C++: sel_t) its default value. this value will be used by get_sreg() if value
              of the register is unknown at the specified address.
```

```
@return: success

### IDAPython function idaapi.set_defsr quick reference

set_defsr(s, reg, value)
Deprecated, use instead:
    s.defsr[reg] = value

@param s: segment_t *
@param reg: int
@param value: sel_t

### IDAPython function idaapi.set_dock_pos quick reference

set_dock_pos(src_ctrl, dest_ctrl, orient, left=0, top=0, right=0, bottom=0) -> bool
Sets the dock orientation of a window relatively to another window.

Use the left, top, right, bottom parameters if DP_FLOATING is used,
or if you want to specify the width of docked windows.

@param src_ctrl: char const *
@param dest_ctrl: char const *
@param orient: One of DP_XXXX constants
@param left: int
@param top: int
@param right: int
@param bottom: int
@return: Boolean

Example:
    set_dock_pos('Structures', 'Enums', DP_RIGHT) <- docks the Structures window to the righ

### IDAPython function idaapi.set_dummy_name quick reference

set_dummy_name(_from, ea) -> bool
Give an autogenerated (dummy) name. Autogenerated names have special prefixes
(loc_...).

@param from (integer): linear address of the operand which references to the address
@param ea (integer): linear address
@retval 1: ok, dummy name is generated or the byte already had a name
@retval 0: failure, invalid address or tail byte

### IDAPython function idaapi.set_entry_forwarder quick reference

set_entry_forwarder(ord, name, flags=0) -> bool
Set forwarder name for ordinal.

@param ord (integer): ordinal number of the entry point
@param name (string): forwarder name for entry point.
```

```
@param flags (integer): See AEF_*
@return: success

### IDAPython function idaapi.set_enum_bf quick reference

set_enum_bf(id, bf) -> bool
Set 'bitfield' bit of enum (i.e. convert it to a bitfield)

@param id (integer):
@param bf (bool):

### IDAPython function idaapi.set_enum_cmt quick reference

set_enum_cmt(id, cmt, repeatable) -> bool
Set comment for enum type.

@param id (integer):
@param cmt (string): char const *
@param repeatable (bool):

### IDAPython function idaapi.set_enum_flag quick reference

set_enum_flag(id, flag) -> bool
Set data representation flags.

@param id (integer):
@param flag (integer):

### IDAPython function idaapi.set_enum_fromtil quick reference

set_enum_fromtil(id, fromtil) -> bool
Specify that enum comes from a type library.

@param id (integer):
@param fromtil (bool):

### IDAPython function idaapi.set_enum_ghost quick reference

set_enum_ghost(id, ghost) -> bool
Specify that enum is a ghost copy of a local type.

@param id (integer):
@param ghost (bool):

### IDAPython function idaapi.set_enum_hidden quick reference

set_enum_hidden(id, hidden) -> bool
Collapse enum.

@param id (integer):
@param hidden (bool):
```

```
### IDAPython function idaapi.set_enum_idx quick reference

set_enum_idx(id, idx) -> bool
Set serial number of enum. Also see get_enum_idx().

@param id (integer):
@param idx (integer):

### IDAPython function idaapi.set_enum_member_cmt quick reference

set_enum_member_cmt(id, cmt, repeatable) -> bool
Set comment for enum member.

@param id (integer):
@param cmt (string): char const *
@param repeatable (bool):

### IDAPython function idaapi.set_enum_member_name quick reference

set_enum_member_name(id, name) -> bool
Set name of enum member.

@param id (integer):
@param name (string): char const *

### IDAPython function idaapi.set_enum_name quick reference

set_enum_name(id, name) -> bool
Set name of enum type.

@param id (integer):
@param name (string): char const *

### IDAPython function idaapi.set_enum_type_ordinal quick reference

set_enum_type_ordinal(id, ord)
Set corresponding type ordinal number.

@param id (integer):
@param ord: (C++: int32)

### IDAPython function idaapi.set_enum_width quick reference

set_enum_width(id, width) -> bool
See comment for get_enum_width()

@param id (integer):
@param width (integer):

### IDAPython function idaapi.set_fixed_spd quick reference

set_fixed_spd(ea)
```

```
@param ea: ea_t

### IDAPython function idaapi.set_fixup quick reference

set_fixup(source, fd)
Set fixup information. You should fill fixup_data_t and call this function and
the kernel will remember information in the database.

@param source (integer): the fixup source address, i.e. the address modified by the fixup
@param fd: (C++: const fixup_data_t &) fixup data

### IDAPython function idaapi.set_forced_operand quick reference

set_forced_operand(ea, n, op) -> bool
Set forced operand.

@param ea (integer): linear address
@param n (integer): 0..UA_MAXOP-1 operand number
@param op (string): text of operand
* nullptr: do nothing (return 0)
* "" : delete forced operand
@return: success

### IDAPython function idaapi.set_frame_size quick reference

set_frame_size(pfn, frsize, frregs, argsize) -> bool
Set size of function frame. Note: The returned size may not include all stack
arguments. It does so only for __stdcall and __fastcall calling conventions. To
get the entire frame size for all cases use get_struc_size(get_frame(pfn)).

@param pfn (idaapi.func_t): pointer to function structure
@param frsize (integer): size of function local variables
@param frregs: (C++: ushort) size of saved registers
@param argsize (integer): size of function arguments that will be purged from the stack
                upon return
@return: success

### IDAPython function idaapi.set_func_cmt quick reference

set_func_cmt(pfn, cmt, repeatable) -> bool
Set function comment. This function works with function chunks too.

@param pfn: (C++: const func_t *) ptr to function structure
@param cmt (string): comment string, may be multiline (with '
'). Use empty str ("") to delete comment
@param repeatable (bool): set repeatable comment?

### IDAPython function idaapi.set_func_end quick reference

set_func_end(ea, newend) -> bool
```

```
Move function chunk end address.

@param ea (integer): any address in the function
@param newend (integer): new end address of the function
@return: success

### IDAPython function idaapi.set_func_guessed_by_hexrays quick reference

set_func_guessed_by_hexrays(ea)

@param ea: ea_t

### IDAPython function idaapi.set_func_name_if_jumpfunc quick reference

set_func_name_if_jumpfunc(pfn, oldname) -> int
Give a meaningful name to function if it consists of only 'jump' instruction.

@param pfn (idaapi.func_t): pointer to function (may be nullptr)
@param oldname (string): old name of function. if old name was in "j_..." form, then we
                may discard it and set a new name. if oldname is not known, you
                may pass nullptr.
@return: success

### IDAPython function idaapi.set_func_start quick reference

set_func_start(ea, newstart) -> int
Move function chunk start address.

@param ea (integer): any address in the function
@param newstart (integer): new end address of the function
@return: Function move result codes

### IDAPython function idaapi.set_func_trace_options quick reference

set_func_trace_options(options)
Modify function tracing options. \sq{Type, Synchronous function - available as
request, Notification, none (synchronous function)}

@param options (integer):

### IDAPython function idaapi.set_gotea quick reference

set_gotea(gotea)

@param gotea: ea_t

### IDAPython function idaapi.set_group_selector quick reference

set_group_selector(grp, sel) -> int
Create a new group of segments (used OMF files).

@param grp: (C++: sel_t) selector of group segment (segment type is SEG_GRP) You should
```

```
            create an 'empty' (1 byte) group segment It won't contain anything
            and will be used to redirect references to the group of segments to
            the common selector.
@param sel: (C++: sel_t) common selector of all segments belonging to the segment You should
            create all segments within the group with the same selector value.
@return: 1 ok
0 too many groups (see MAX_GROUPS)
```

```
### IDAPython function idaapi.set_has_lname quick reference
```

```
set_has_lname(ea)
```

```
@param ea: ea_t
```

```
### IDAPython function idaapi.set_has_ti quick reference
```

```
set_has_ti(ea)
```

```
@param ea: ea_t
```

```
### IDAPython function idaapi.set_has_ti0 quick reference
```

```
set_has_ti0(ea)
```

```
@param ea: ea_t
```

```
### IDAPython function idaapi.set_has_ti1 quick reference
```

```
set_has_ti1(ea)
```

```
@param ea: ea_t
```

```
### IDAPython function idaapi.set_header_path quick reference
```

```
set_header_path(path, add) -> bool
Set or append a header path. IDA looks for the include files in the appended
header paths, then in the ida executable directory.

@param path (string): list of directories to add (separated by ';') may be nullptr, in
             this case nothing is added
@param add (bool): true: append. false: remove old paths.
@retval true: success
@retval false: no memory
```

```
### IDAPython function idaapi.set_highlight quick reference
```

```
set_highlight(viewer, str, flags) -> bool
Set the highlighted identifier in the viewer (ui_set_highlight).

@param viewer(a Widget SWIG wrapper class): the viewer
@param str (string): the text to match, or nullptr to remove current
@param flags (integer): combination of HIF_... bits (see set_highlight flags)
```

```
@return: false if an error occurred

### IDAPython function idaapi.set_highlight_trace_options quick reference

set_highlight_trace_options(hilight, color, diff)
Set highlight trace parameters.

@param hilight (bool):
@param color (integer):
@param diff (integer):

### IDAPython function idaapi.set_ida_notepad_text quick reference

set_ida_notepad_text(text, size=0)
Set notepad text.

@param text (string): char const *
@param size (integer):

### IDAPython function idaapi.set_ida_state quick reference

set_ida_state(st) -> idastate_t
Change IDA status indicator value

@param st: (C++: idastate_t) - new indicator status
@return: old indicator status

### IDAPython function idaapi.set_idcv_attr quick reference

set_idcv_attr(obj, attr, value, may_use_setattr=False) -> error_t
Set an object attribute.

@param obj (idaapi.idc_value_t): variable that holds an object reference. if obj is nullptr
            tries to modify a global variable with the attribute name
@param attr (string): attribute name
@param value: (C++: const idc_value_t &) new attribute value
@param may_use_setattr (bool): may call setattr functions for the class
@return: error code, eOk on success

### IDAPython function idaapi.set_idcv_slice quick reference

set_idcv_slice(v, i1, i2, _in, flags=0) -> error_t
Set slice.

@param v (idaapi.idc_value_t): variable to modify (string or object)
@param i1 (integer): slice start index
@param i2 (integer): slice end index (excluded)
@param in: (C++: const idc_value_t &) new value for the slice
@param flags (integer): IDC variable slice flags or 0
@return: eOk on success
```

```
### IDAPython function idaapi.set_ids_modnode quick reference

set_ids_modnode(id)
Set ids modnode.

@param id: (C++: netnode)

### IDAPython function idaapi.set_imagebase quick reference

set_imagebase(base)
Set image base address.

@param base (integer):

### IDAPython function idaapi.set_immd quick reference

set_immd(ea) -> bool
Set 'has immediate operand' flag. Returns true if the FF_IMMD bit was not set
and now is set

@param ea (integer):

### IDAPython function idaapi.set_import_name quick reference

set_import_name(modnode, ea, name)
Set information about the named import entry. This function performs
'modnode.supset_ea(ea, name);'

@param modnode (integer): node with information about imported entries
@param ea (integer): linear address of the entry
@param name (string): name of the entry

### IDAPython function idaapi.set_import_ordinal quick reference

set_import_ordinal(modnode, ea, ord)
Set information about the ordinal import entry. This function performs
'modnode.altset(ord, ea2node(ea));'

@param modnode (integer): node with information about imported entries
@param ea (integer): linear address of the entry
@param ord (integer): ordinal number of the entry

### IDAPython function idaapi.set_insn_trace_options quick reference

set_insn_trace_options(options)
Modify instruction tracing options. \sq{Type, Synchronous function - available
as request, Notification, none (synchronous function)}

@param options (integer):

### IDAPython function idaapi.set_item_color quick reference
```

```
set_item_color(ea, color)

@param ea: ea_t
@param color: bgcolor_t

### IDAPython function idaapi.set_libitem quick reference

set_libitem(ea)

@param ea: ea_t

### IDAPython function idaapi.set_loader_format_name quick reference

set_loader_format_name(name)
Set file format name for loader modules.

@param name (string): char const *

### IDAPython function idaapi.set_lzero quick reference

set_lzero(ea, n) -> bool
Set toggle lzero bit. This function changes the display of leading zeroes for
the specified operand. If the default is not to display leading zeroes, this
function will display them and vice versa.

@param ea (integer): the item (insn/data) address
@param n (integer): the operand number (0-first operand, 1-other operands)
@return: success

### IDAPython function idaapi.set_lzero0 quick reference

set_lzero0(ea)

@param ea: ea_t

### IDAPython function idaapi.set_lzero1 quick reference

set_lzero1(ea)

@param ea: ea_t

### IDAPython function idaapi.set_manual_insn quick reference

set_manual_insn(ea, manual_insn)
Set manual instruction string.

@param ea (integer): linear address of the instruction or data item
@param manual_insn (string): "" - delete manual string. nullptr - do nothing

### IDAPython function idaapi.set_manual_regions quick reference

set_manual_regions(ranges)
```

```
@param ranges: meminfo_vec_t const *

### IDAPython function idaapi.set_member_cmt quick reference

set_member_cmt(mptr, cmt, repeatable) -> bool
Set member comment.

@param mptr: (C++: member_t *)
@param cmt (string): char const *
@param repeatable (bool):

### IDAPython function idaapi.set_member_name quick reference

set_member_name(sptr, offset, name) -> bool
Set name of member at given offset.

@param sptr: (C++: struc_t *)
@param offset (integer):
@param name (string): char const *

### IDAPython function idaapi.set_member_tinfo quick reference

set_member_tinfo(sptr, mptr, memoff, tif, flags) -> smt_code_t
Set tinfo for given member.

@param sptr: (C++: struc_t *) containing struct
@param mptr: (C++: member_t *) target member
@param memoff (integer): offset within member
@param tif (idaapi.tinfo_t): type info
@param flags (integer): Set member tinfo flags

### IDAPython function idaapi.set_member_type quick reference

set_member_type(sptr, offset, flag, mt, nbytes) -> bool
Set type of member at given offset (also see add_struc_member())

@param sptr: (C++: struc_t *)
@param offset (integer):
@param flag (integer):
@param mt: (C++: const opinfo_t *) opinfo_t const *
@param nbytes (integer):

### IDAPython function idaapi.set_name quick reference

set_name(ea, name, flags=0) -> bool
Set or delete name of an item at the specified address. An item can be anything:
instruction, function, data byte, word, string, structure, etc... Include name
into the list of names.

@param ea (integer): linear address. do nothing if ea is not valid (return 0). tail bytes
           can't have names.
```

```
@param name (string): new name.
* nullptr: do nothing (return 0).
* "" : delete name.
* otherwise this is a new name.
@param flags (integer): Set name flags. If a bit is not specified, then the corresponding
              action is not performed and the name will retain the same bits as
              before calling this function. For new names, default is: non-
              public, non-weak, non-auto.
@retval 1: ok, name is changed
@retval 0: failure, a warning is displayed

### IDAPython function idaapi.set_nav_colorizer quick reference

set_nav_colorizer(new_py_colorizer) -> dict or None
Set a new colorizer for the navigation band.

The 'callback' is a function of 2 arguments:
   - ea (the EA to colorize for)
   - nbytes (the number of bytes at that EA)
and must return a 'long' value.

The previous colorizer is returned, allowing
the new 'callback' to use 'call_nav_colorizer'
with it.

Note that the previous colorizer is returned
only the first time set_nav_colorizer() is called:
due to the way the colorizers API is defined in C,
it is impossible to chain more than 2 colorizers
in IDAPython: the original, IDA-provided colorizer,
and a user-provided one.

Example: colorizer inverting the color provided by the IDA colorizer:
    def my_colorizer(ea, nbytes):
        global ida_colorizer
        orig = idaapi.call_nav_colorizer(ida_colorizer, ea, nbytes)
        return long(~orig)

    ida_colorizer = idaapi.set_nav_colorizer(my_colorizer)

@param new_py_colorizer: PyObject *

### IDAPython function idaapi.set_node_info quick reference

set_node_info(gid, node, ni, flags)
Set node info.

@param gid: (C++: graph_id_t) id of desired graph
```

```
@param node (integer): node number
@param ni: (C++: const node_info_t &) node info to use
@param flags (integer): combination of Node info flags, identifying which fields of 'ni'
              will be used

### IDAPython function idaapi.set_noret quick reference

set_noret(ea)

@param ea: ea_t

### IDAPython function idaapi.set_noret_insn quick reference

set_noret_insn(insn_ea, noret) -> bool
Signal a non-returning instruction. This function can be used by the processor
module to tell the kernel about non-returning instructions (like call exit). The
kernel will perform the global function analysis and find out if the function
returns at all. This analysis will be done at the first call to
func_does_return()

@param insn_ea (integer):
@param noret (bool):
@return: true if the instruction 'noret' flag has been changed

### IDAPython function idaapi.set_notcode quick reference

set_notcode(ea)
Mark address so that it cannot be converted to instruction.

@param ea (integer):

### IDAPython function idaapi.set_notproc quick reference

set_notproc(ea)

@param ea: ea_t

### IDAPython function idaapi.set_numbered_type quick reference

set_numbered_type(ti, ordinal, ntf_flags, name, type, fields=None, cmt=None, fldcmts=None, s
Store a type in the til. 'name' may be nullptr for anonymous types. The
specified ordinal must be free (no other type is using it). For ntf_flags, only
NTF_REPLACE is consulted.

@param ti (idaapi.til_t):
@param ordinal (integer):
@param ntf_flags (integer):
@param name (string): char const *
@param type: (C++: const type_t *) type_t const *
@param fields: (C++: const p_list *) p_list const *
@param cmt (string): char const *
```

```
@param fldcmts: (C++: const p_list *) p_list const *
@param sclass: (C++: const sclass_t *) sclass_t const *

### IDAPython function idaapi.set_op_tinfo quick reference

set_op_tinfo(ea, n, tif) -> bool

@param ea: ea_t
@param n: int
@param tif: tinfo_t const *

### IDAPython function idaapi.set_op_type quick reference

set_op_type(ea, type, n) -> bool
(internal function) change representation of operand(s).

@param ea (integer): linear address
@param type (integer): new flag value (should be obtained from char_flag(), num_flag() and
            similar functions)
@param n (integer): 0..UA_MAXOP-1 operand number, OPND_ALL all operands
@retval 1: ok
@retval 0: failed (applied to a tail byte)

### IDAPython function idaapi.set_opinfo quick reference

set_opinfo(ea, n, flag, ti, suppress_events=False) -> bool
Set additional information about an operand representation. This function is a
low level one. Only the kernel should use it.

@param ea (integer): linear address of the item
@param n (integer): number of operand, 0 or 1 (see the note below)
@param flag (integer): flags of the item
@param ti: (C++: const opinfo_t *) additional representation information
@param suppress_events (bool): do not generate changing_op_type and op_type_changed
                         events
@return: success
@note: for custom formats (if is_custfmt(flag, n) is true) or for offsets (if
       is_off(flag, n) is true) N can be in range 0..UA_MAXOP-1 or equal to
       OPND_ALL. In the case of OPND_ALL the additional information about all
       operands will be set.

### IDAPython function idaapi.set_outfile_encoding_idx quick reference

set_outfile_encoding_idx(idx) -> bool
set encoding to be used when producing files

@param idx (integer): the encoding index IDX can be 0 to use the IDB's default 1-byte-per-
            unit encoding

### IDAPython function idaapi.set_parser_argv quick reference
```

```
set_parser_argv(parser_name, argv) -> int
Set the command-line args to use for invocations of the parser with the given
name

@param parser_name (string): name of the target parser
@param argv (string): argument list
@retval -1: no parser was found with the given name
@retval -2: the operation is not supported by the given parser
@retval 0: success

### IDAPython function idaapi.set_path quick reference

set_path(pt, path)
Set the file path

@param pt: (C++: path_type_t) file path type Types of the file pathes
@param path (string): new file path, use nullptr or empty string to clear the file path

### IDAPython function idaapi.set_process_options quick reference

set_process_options(path, args, sdir, host, _pass, port)
Set process options. Any of the arguments may be nullptr, which means 'do not
modify'

@param path (string): char const *
@param args (string): char const *
@param sdir (string): char const *
@param host (string): char const *
@param pass (string): char const *
@param port (integer):

### IDAPython function idaapi.set_process_state quick reference

set_process_state(newstate, p_thid, dbginv) -> int
Set new state for the debugged process. Notifies the IDA kernel about the change
of the debugged process state. For example, a debugger module could call this
function when it knows that the process is suspended for a short period of time.
Some IDA API calls can be made only when the process is suspended. The process
state is usually restored before returning control to the caller. You must know
that it is ok to change the process state, doing it at arbitrary moments may
crash the application or IDA. \sq{Type, Synchronous function, Notification, none
(synchronous function)}

@param newstate (integer): new process state (one of Debugged process states) if
                 DSTATE_NOTASK is passed then the state is not changed
@param p_thid: (C++: thid_t *) ptr to new thread id. may be nullptr or pointer to NO_THREAD.
               pointed variable will contain the old thread id upon return
@param dbginv (integer): Debugged process invalidation options
@return: old debugger state (one of Debugged process states)
```

```
### IDAPython function idaapi.set_processor_type quick reference

set_processor_type(procname, level) -> bool
Set target processor type. Once a processor module is loaded, it cannot be
replaced until we close the idb.

@param procname (string): name of processor type (one of names present in
                processor_t::psnames)
@param level: (C++: setproc_level_t) SETPROC_
@return: success

### IDAPython function idaapi.set_purged quick reference

set_purged(ea, nbytes, override_old_value) -> bool
Set the number of purged bytes for a function or data item (funcptr). This
function will update the database and plan to reanalyze items referencing the
specified address. It works only for processors with PR_PURGING bit in 16 and 32
bit modes.

@param ea (integer): address of the function of item
@param nbytes (integer): number of purged bytes
@param override_old_value (bool): may overwrite old information about purged bytes
@return: success

### IDAPython function idaapi.set_refinfo quick reference

set_refinfo(ea, n, type, target=BADADDR, base=0, tdelta=0) -> bool

@param ea: ea_t
@param n: int
@param type: reftype_t
@param target: ea_t
@param base: ea_t
@param tdelta: adiff_t

### IDAPython function idaapi.set_refinfo_ex quick reference

set_refinfo_ex(ea, n, ri) -> bool

@param ea: ea_t
@param n: int
@param ri: refinfo_t const *

### IDAPython function idaapi.set_reg_val quick reference

set_reg_val(regname, o) -> PyObject
Write a register value to the current thread.

@param regname (string): char const *
@param o: PyObject *
```

```
set_reg_val(tid, regidx, o) -> bool, int

@param tid: thid_t
@param regidx: int
@param o: PyObject *

### IDAPython function idaapi.set_registry_root quick reference

set_registry_root(name) -> bool

@param name: char const *

### IDAPython function idaapi.set_regvar_cmt quick reference

set_regvar_cmt(pfn, v, cmt) -> int
Set comment for a register variable.

@param pfn (idaapi.func_t): function in question
@param v: (C++: regvar_t *) variable to rename
@param cmt (string): new comment
@return: Register variable error codes

### IDAPython function idaapi.set_remote_debugger quick reference

set_remote_debugger(host, _pass, port=-1)
Set remote debugging options. Should be used before starting the debugger.

@param host (string): If empty, IDA will use local debugger. If nullptr, the host will
             not be set.
@param pass (string): If nullptr, the password will not be set
@param port (integer): If -1, the default port number will be used

### IDAPython function idaapi.set_resume_mode quick reference

set_resume_mode(tid, mode) -> bool
How to resume the application. Set resume mode but do not resume process.

@param tid (integer):
@param mode: (C++: resume_mode_t) enum resume_mode_t

### IDAPython function idaapi.set_retfp quick reference

set_retfp(ea)

@param ea: ea_t

### IDAPython function idaapi.set_root_filename quick reference

set_root_filename(file)
Set full path of the input file.
```

```
@param file (string): char const *

### IDAPython function idaapi.set_script_timeout quick reference

set_script_timeout(timeout) -> int
Changes the script timeout value. The script wait box dialog will be hidden and shown again
See also L{disable_script_timeout}.

@param timeout: This value is in seconds.
                If this value is set to zero then the script will never timeout.
@return: Returns the old timeout value

### IDAPython function idaapi.set_segm_addressing quick reference

set_segm_addressing(s, bitness) -> bool
Change segment addressing mode (16, 32, 64 bits). You must use this function to
change segment addressing, never change the 'bitness' field directly. This
function will delete all instructions, comments and names in the segment

@param s: (C++: segment_t *) pointer to segment
@param bitness (integer): new addressing mode of segment
* 2: 64bit segment
* 1: 32bit segment
* 0: 16bit segment
@return: success

### IDAPython function idaapi.set_segm_base quick reference

set_segm_base(s, newbase) -> bool
Internal function.

@param s: (C++: segment_t *)
@param newbase (integer):

### IDAPython function idaapi.set_segm_class quick reference

set_segm_class(s, sclass, flags=0) -> int
Set segment class.

@param s: (C++: segment_t *) pointer to segment (may be nullptr)
@param sclass (string): segment class (may be nullptr). If segment type is SEG_NORM and
                segment class is one of predefined names, then segment type is
                changed to:
* "CODE" -> SEG_CODE
* "DATA" -> SEG_DATA
* "STACK" -> SEG_BSS
* "BSS" -> SEG_BSS
* if "UNK" then segment type is reset to SEG_NORM.
@param flags (integer): Add segment flags
@retval 1: ok, name is good and segment is renamed
```

```
@retval 0: failure, name is nullptr or bad or segment is nullptr

### IDAPython function idaapi.set_segm_end quick reference

set_segm_end(ea, newend, flags) -> bool
Set segment end address. The next segment is shrinked to allow expansion of the
specified segment. The kernel might even delete the next segment if necessary.
The kernel will ask the user for a permission to destroy instructions or data
going out of segment scope if such instructions exist.

@param ea (integer): any address belonging to the segment
@param newend (integer): new end address of the segment
@param flags (integer): Segment modification flags
@retval 1: ok
@retval 0: failed, a warning message is displayed

### IDAPython function idaapi.set_segm_name quick reference

set_segm_name(s, name, flags=0) -> int
Rename segment. The new name is validated (see validate_name). A segment always
has a name. If you hadn't specified a name, the kernel will assign it "seg###"
name where ### is segment number.

@param s: (C++: segment_t *) pointer to segment (may be nullptr)
@param name (string): new segment name
@param flags (integer): ADDSEG_IDBENC or 0
@retval 1: ok, name is good and segment is renamed
@retval 0: failure, name is bad or segment is nullptr

### IDAPython function idaapi.set_segm_start quick reference

set_segm_start(ea, newstart, flags) -> bool
Set segment start address. The previous segment is trimmed to allow expansion of
the specified segment. The kernel might even delete the previous segment if
necessary. The kernel will ask the user for a permission to destroy instructions
or data going out of segment scope if such instructions exist.

@param ea (integer): any address belonging to the segment
@param newstart (integer): new start address of the segment note that segment start
                address should be higher than segment base linear address.
@param flags (integer): Segment modification flags
@retval 1: ok
@retval 0: failed, a warning message is displayed

### IDAPython function idaapi.set_segment_cmt quick reference

set_segment_cmt(s, cmt, repeatable)
Set segment comment.

@param s: (C++: const segment_t *) pointer to segment structure
```

```
@param cmt (string): comment string, may be multiline (with '
'). maximal size is 4096 bytes. Use empty str ("") to delete comment
@param repeatable (bool): 0: set regular comment. 1: set repeatable comment.

### IDAPython function idaapi.set_segment_translations quick reference

set_segment_translations(segstart, transmap) -> bool
Set new translation list.

@param segstart (integer): start address of the segment to add translation to
@param transmap: (C++: const eavec_t &) vector of segment start addresses for the translatio
                 transmap is empty, the translation list is deleted.
@retval 1: ok
@retval 0: too many translations or bad segstart

### IDAPython function idaapi.set_selector quick reference

set_selector(selector, paragraph) -> int
Set mapping of selector to a paragraph. You should call this function _before_
creating a segment which uses the selector, otherwise the creation of the
segment will fail.

@param selector: (C++: sel_t) number of selector to map
* if selector == BADSEL, then return 0 (fail)
* if the selector has had a mapping, old mapping is destroyed
* if the selector number is equal to paragraph value, then the mapping is
destroyed because we don't need to keep trivial mappings.
@param paragraph (integer): paragraph to map selector
@retval 1: ok
@retval 0: failure (bad selector or too many mappings)

### IDAPython function idaapi.set_source_linnum quick reference

set_source_linnum(ea, lnnum)

@param ea: ea_t
@param lnnum: uval_t

### IDAPython function idaapi.set_srcdbg_paths quick reference

set_srcdbg_paths(paths)
Set source debug paths.

@param paths (string): char const *

### IDAPython function idaapi.set_srcdbg_undesired_paths quick reference

set_srcdbg_undesired_paths(paths)
Set user-closed source files.

@param paths (string): char const *
```

IDAPython function idaapi.set_sreg_at_next_code quick reference

set_sreg_at_next_code(ea1, ea2, rg, value)
Set the segment register value at the next instruction. This function is designed to be called from idb_event::sgr_changed handler in order to contain the effect of changing a segment register value only until the next instruction.

It is useful, for example, in the ARM module: the modification of the T register does not affect existing instructions later in the code.

@param ea1 (integer): address to start to search for an instruction
@param ea2 (integer): the maximal address
@param rg (integer): the segment register number
@param value: (C++: sel_t) the segment register value

IDAPython function idaapi.set_step_trace_options quick reference

set_step_trace_options(options)
Modify step tracing options. \sq{Type, Synchronous function - available as request, Notification, none (synchronous function)}

@param options (integer):

IDAPython function idaapi.set_str_encoding_idx quick reference

set_str_encoding_idx(strtype, encoding_idx) -> int32
Set index of the string encoding in the string type.

@param strtype: (C++: int32)
@param encoding_idx (integer):

IDAPython function idaapi.set_str_type quick reference

set_str_type(ea, x)

@param ea: ea_t
@param x: uint32

IDAPython function idaapi.set_struc_align quick reference

set_struc_align(sptr, shift) -> bool
Set structure alignment (SF_ALIGN)

@param sptr: (C++: struc_t *)
@param shift (integer):

IDAPython function idaapi.set_struc_cmt quick reference

set_struc_cmt(id, cmt, repeatable) -> bool
Set structure comment.

```
@param id (integer):
@param cmt (string): char const *
@param repeatable (bool):

### IDAPython function idaapi.set_struc_hidden quick reference

set_struc_hidden(sptr, is_hidden)
Hide/unhide a struct type.

@param sptr: (C++: struc_t *)
@param is_hidden (bool):

### IDAPython function idaapi.set_struc_idx quick reference

set_struc_idx(sptr, idx) -> bool
Set internal number of struct. Also see get_struc_idx(), get_struc_by_idx().

@param sptr: (C++: const struc_t *) struc_t const *
@param idx (integer):

### IDAPython function idaapi.set_struc_listed quick reference

set_struc_listed(sptr, is_listed)
Add/remove a struct type from the struct list.

@param sptr: (C++: struc_t *)
@param is_listed (bool):

### IDAPython function idaapi.set_struc_name quick reference

set_struc_name(id, name) -> bool
Set structure name.

@param id (integer):
@param name (string): char const *

### IDAPython function idaapi.set_switch_info quick reference

set_switch_info(ea, _in)

@param ea: ea_t
@param in: switch_info_t const &

### IDAPython function idaapi.set_switch_parent quick reference

set_switch_parent(ea, x)

@param ea: ea_t
@param x: ea_t

### IDAPython function idaapi.set_tail_owner quick reference
```

```
set_tail_owner(fnt, new_owner) -> bool
Set a new owner of a function tail. The new owner function must be already
referring to the tail (after append_func_tail).

@param fnt (idaapi.func_t): pointer to the function tail
@param new_owner (integer): the entry point of the new owner function

### IDAPython function idaapi.set_target_assembler quick reference

set_target_assembler(asmnum) -> bool
Set target assembler.

@param asmnum (integer): number of assembler in the current processor module
@return: success

### IDAPython function idaapi.set_terse_struc quick reference

set_terse_struc(ea)

@param ea: ea_t

### IDAPython function idaapi.set_tilcmt quick reference

set_tilcmt(ea)

@param ea: ea_t

### IDAPython function idaapi.set_tinfo quick reference

set_tinfo(ea, tif) -> bool

@param ea: ea_t
@param tif: tinfo_t const *

### IDAPython function idaapi.set_tinfo_attr quick reference

set_tinfo_attr(tif, ta, may_overwrite) -> bool

@param tif: tinfo_t *
@param ta: type_attr_t const &
@param may_overwrite: bool

### IDAPython function idaapi.set_tinfo_attrs quick reference

set_tinfo_attrs(tif, ta) -> bool

@param tif: tinfo_t *
@param ta: type_attrs_t *

### IDAPython function idaapi.set_tinfo_property quick reference

set_tinfo_property(tif, sta_prop, x) -> size_t
```

```
@param tif: tinfo_t *
@param sta_prop: int
@param x: size_t

### IDAPython function idaapi.set_trace_base_address quick reference

set_trace_base_address(ea)
Set the base address of the current trace. \sq{Type, Synchronous function,
Notification, none (synchronous function)}

@param ea (integer):

### IDAPython function idaapi.set_trace_dynamic_register_set quick reference

set_trace_dynamic_register_set(idaregs)
Set dynamic register set of current trace.

@param idaregs: (C++: dynamic_register_set_t &)

### IDAPython function idaapi.set_trace_file_desc quick reference

set_trace_file_desc(filename, description) -> bool
Change the description of the specified trace file.

@param filename (string): char const *
@param description (string): char const *

### IDAPython function idaapi.set_trace_platform quick reference

set_trace_platform(platform)
Set platform name of current trace.

@param platform (string): char const *

### IDAPython function idaapi.set_trace_size quick reference

set_trace_size(size) -> bool
Specify the new size of the circular buffer. \sq{Type, Synchronous function,
Notification, none (synchronous function)}

@param size (integer): if 0, buffer isn't circular and events are never removed. If the
             new size is smaller than the existing number of trace events, a
             corresponding number of trace events are removed.
@note: If you specify 0, all available memory can be quickly used !!!

### IDAPython function idaapi.set_type quick reference

set_type(id, tif, source, force=False) -> bool
Set a global type.

@param id (integer): address or id of the object
@param tif (idaapi.tinfo_t): new type info
```

```
@param source: (C++: type_source_t) where the type comes from
@param force (bool): true means to set the type as is, false means to merge the new
             type with the possibly existing old type info.
@return: success

### IDAPython function idaapi.set_type_alias quick reference

set_type_alias(ti, src_ordinal, dst_ordinal) -> bool
Create a type alias. Redirects all references to source type to the destination
type. This is equivalent to instantaneous replacement all reference to srctype
by dsttype.

@param ti (idaapi.til_t):
@param src_ordinal (integer):
@param dst_ordinal (integer):

### IDAPython function idaapi.set_type_determined_by_hexrays quick reference

set_type_determined_by_hexrays(ea)

@param ea: ea_t

### IDAPython function idaapi.set_type_guessed_by_ida quick reference

set_type_guessed_by_ida(ea)

@param ea: ea_t

### IDAPython function idaapi.set_usemodsp quick reference

set_usemodsp(ea)

@param ea: ea_t

### IDAPython function idaapi.set_user_defined_prefix quick reference

set_user_defined_prefix(width, pycb) -> bool
Deprecated. Please use install_user_defined_prefix() instead

@param width: size_t
@param pycb: PyObject *

### IDAPython function idaapi.set_usersp quick reference

set_usersp(ea)

@param ea: ea_t

### IDAPython function idaapi.set_userti quick reference

set_userti(ea)

@param ea: ea_t
```

```
### IDAPython function idaapi.set_vftable_ea quick reference

set_vftable_ea(ordinal, vftable_ea) -> bool
Set the address of a vftable instance for a vftable type.

@param ordinal (integer): ordinal number of the corresponding vftable type.
@param vftable_ea (integer): address of a virtual function table.
@return: success

### IDAPython function idaapi.set_view_renderer_type quick reference

set_view_renderer_type(v, rt)
Set the type of renderer to use in a view (ui_set_renderer_type)

@param v(a Widget SWIG wrapper class):
@param rt: (C++: tcc_renderer_type_t) enum tcc_renderer_type_t

### IDAPython function idaapi.set_viewer_graph quick reference

set_viewer_graph(gv, g)
Set the underlying graph object for the given viewer.

@param gv: (C++: graph_viewer_t *)
@param g: (C++: mutable_graph_t *)

### IDAPython function idaapi.set_visible_func quick reference

set_visible_func(pfn, visible)
Set visibility of function.

@param pfn (idaapi.func_t):
@param visible (bool):

### IDAPython function idaapi.set_visible_item quick reference

set_visible_item(ea, visible)
Change visibility of item at given ea.

@param ea (integer):
@param visible (bool):

### IDAPython function idaapi.set_visible_segm quick reference

set_visible_segm(s, visible)
See SFL_HIDDEN.

@param s: (C++: segment_t *)
@param visible (bool):

### IDAPython function idaapi.set_zstroff quick reference

set_zstroff(ea)
```

```
@param ea: ea_t

### IDAPython function idaapi.setup_selector quick reference

setup_selector(segbase) -> sel_t
Allocate a selector for a segment if necessary. You must call this function
before calling add_segm_ex(). add_segm() calls this function itself, so you
don't need to allocate a selector. This function will allocate a selector if
'segbase' requires more than 16 bits and the current processor is IBM PC.
Otherwise it will return the segbase value.

@param segbase (integer): a new segment base paragraph
@return: the allocated selector number

### IDAPython function idaapi.should_create_stkvars quick reference

inf_should_create_stkvars() -> bool

### IDAPython function idaapi.should_trace_sp quick reference

inf_should_trace_sp() -> bool

### IDAPython function idaapi.show_addr quick reference

show_addr(ea)
Show an address on the autoanalysis indicator. The address is displayed in the
form " @:12345678".

@param ea (integer): - linear address to display

### IDAPython function idaapi.show_all_comments quick reference

inf_show_all_comments() -> bool

### IDAPython function idaapi.show_auto quick reference

show_auto(ea, type=AU_NONE)
Change autoanalysis indicator value.

@param ea (integer): linear address being analyzed
@param type (one of the idaapi.AU_xxxx flags): autoanalysis type (see Autoanalysis queues)

### IDAPython function idaapi.show_name quick reference

show_name(ea)
Insert name to the list of names.

@param ea (integer):

### IDAPython function idaapi.show_repeatables quick reference

inf_show_repeatables() -> bool
```

```
### IDAPython function idaapi.show_wait_box quick reference

show_wait_box(message)
Display a dialog box with "Please wait...". The behavior of the dialog box can
be configured with well-known
tokens, that should be placed at the start of the format string:
"NODELAY\n": the dialog will show immediately, instead of
appearing after usual grace threshold
"HIDECANCEL\n": the cancel button won't be added to the dialog box
and user_cancelled() will always return false (but
can be called to refresh UI)
Using "HIDECANCEL" implies "NODELAY"
Plugins must call hide_wait_box() to close the dialog box, otherwise
the user interface will remain disabled.
Note that, if the wait dialog is already visible, show_wait_box() will
1) push the currently-displayed text on a stack
2) display the new text
Then, when hide_wait_box() is called, if that stack isn't empty its top
label will be popped and restored in the wait dialog.
This implies that a plugin should call hide_wait_box() exactly as many
times as it called show_wait_box(), or the wait dialog might remain
visible and block the UI.
Also, in case the plugin knows the wait dialog is currently displayed,
alternatively it can call replace_wait_box(), to replace the text of the
dialog without pushing the currently-displayed text on the stack.

@param message: char const *

### IDAPython function idaapi.simplecustviewer_t quick reference

The base class for implementing simple custom viewers

### IDAPython function idaapi.sizeof_ldbl quick reference

sizeof_ldbl() -> size_t

### IDAPython function idaapi.soff_to_fpoff quick reference

soff_to_fpoff(pfn, soff) -> sval_t
Convert struct offsets into fp-relative offsets. This function converts the
offsets inside the struc_t object into the frame pointer offsets (for example,
EBP-relative).

@param pfn (idaapi.func_t):
@param soff (integer):

### IDAPython function idaapi.split_sreg_range quick reference

split_sreg_range(ea, rg, v, tag, silent=False) -> bool
Create a new segment register range. This function is used when the IDP emulator
```

```
detects that a segment register changes its value.

@param ea (integer): linear address where the segment register will have a new value. if
           ea==BADADDR, nothing to do.
@param rg (integer): the number of the segment register
@param v: (C++: sel_t) the new value of the segment register. If the value is unknown, you
          should specify BADSEL.
@param tag: (C++: uchar) the register info tag. see Segment register range tags
@param silent (bool): if false, display a warning() in the case of failure
@return: success

### IDAPython function idaapi.srcdbg_request_step_into quick reference

srcdbg_request_step_into() -> bool

### IDAPython function idaapi.srcdbg_request_step_over quick reference

srcdbg_request_step_over() -> bool

### IDAPython function idaapi.srcdbg_request_step_until_ret quick reference

srcdbg_request_step_until_ret() -> bool

### IDAPython function idaapi.srcdbg_step_into quick reference

srcdbg_step_into() -> bool

### IDAPython function idaapi.srcdbg_step_over quick reference

srcdbg_step_over() -> bool

### IDAPython function idaapi.srcdbg_step_until_ret quick reference

srcdbg_step_until_ret() -> bool

### IDAPython function idaapi.start_process quick reference

start_process(path=None, args=None, sdir=None) -> int
Start a process in the debugger. \sq{Type, Asynchronous function - available as
Request, Notification, dbg_process_start}
@note: You can also use the run_to() function to easily start the execution of a
       process until a given address is reached.
@note: For all parameters, a nullptr value indicates the debugger will take the
       value from the defined Process Options.

@param path (string): path to the executable to start
@param args (string): arguments to pass to process
@param sdir (string): starting directory for the process
@retval -1: impossible to create the process
@retval 0: the starting of the process was cancelled by the user
@retval 1: the process was properly started

### IDAPython function idaapi.std_out_segm_footer quick reference
```

```
std_out_segm_footer(ctx, seg)
Generate segment footer line as a comment line. This function may be used in IDP
modules to generate segment footer if the target assembler doesn't have 'ends'
directive.

@param ctx: (C++: struct outctx_t &) outctx_t &
@param seg: (C++: segment_t *)

### IDAPython function idaapi.step_into quick reference

step_into() -> bool
Execute one instruction in the current thread. Other threads are kept suspended.
\sq{Type, Asynchronous function - available as Request, Notification,
dbg_step_into}

### IDAPython function idaapi.step_over quick reference

step_over() -> bool
Execute one instruction in the current thread, but without entering into
functions. Others threads keep suspended. \sq{Type, Asynchronous function -
available as Request, Notification, dbg_step_over}

### IDAPython function idaapi.step_until_ret quick reference

step_until_ret() -> bool
Execute instructions in the current thread until a function return instruction
is executed (aka "step out"). Other threads are kept suspended. \sq{Type,
Asynchronous function - available as Request, Notification, dbg_step_until_ret}

### IDAPython function idaapi.stkvar_flag quick reference

stkvar_flag() -> flags64_t
see FF_opbits

### IDAPython function idaapi.store_exceptions quick reference

store_exceptions() -> bool
Update the exception information stored in the debugger module by invoking its
dbg->set_exception_info callback

### IDAPython function idaapi.store_til quick reference

store_til(ti, tildir, name) -> bool
Store til to a file. If the til contains garbage, it will be collected before
storing the til. Your plugin should call compact_til() before calling
store_til().

@param ti (idaapi.til_t): type library to store
@param tildir (string): directory where to store the til. nullptr means current
              directory.
@param name (string): filename of the til. If it's an absolute path, tildir is ignored.
* NB: the file extension is forced to .til
```

```
@return: success

### IDAPython function idaapi.str2ea quick reference

str2ea(str, screen_ea=BADADDR) -> bool
Convert string to linear address. Tries to interpret the string as:
1) "current IP" keyword if supported by assembler (e.g. "$" in x86)
2) segment:offset expression, where "segment" may be a name or a fixed segment
register (e.g. cs, ds)
3) just segment name/register (translated to segment's start address)
4) a name in the database (or debug name during debugging)
5) +delta or -delta, where numerical 'delta' is added to or subtracted from
'screen_ea'
6) if all else fails, try to evaluate 'str' as an IDC expression

@param str (string): string to parse
@param screen_ea (integer): the current address in the disassembly/pseudocode view
@return: success

### IDAPython function idaapi.str2ea_ex quick reference

str2ea_ex(str, screen_ea=BADADDR, flags=0) -> bool
Same as str2ea() but possibly with some steps skipped.

@param str (string): string to parse
@param screen_ea (integer): the current address in the disassembly/pseudocode view
@param flags (integer): see String to address conversion flags
@return: success

### IDAPython function idaapi.str2reg quick reference

str2reg(p) -> int
Get any reg number (-1 on error)

@param p (string): char const *

### IDAPython function idaapi.str2user quick reference

str2user(str) -> str or None
Insert C-style escape characters to string

@param str: char const *
@return: new string with escape characters inserted

### IDAPython function idaapi.strarray quick reference

strarray(array, array_size, code) -> char const *
Find a line with the specified code in the strarray_t array. If the last element
of the array has code==0 then it is considered as the default entry.
If no default entry exists and the code is not found, strarray() returns "".
```

```
@param array: (C++: const strarray_t *) strarray_t const *
@param array_size (integer):
@param code (integer):

### IDAPython function idaapi.strlit_flag quick reference

strlit_flag() -> flags64_t
Get a flags64_t representing a string literal.

### IDAPython function idaapi.stroff_as_size quick reference

stroff_as_size(plen, sptr, value) -> bool
Should display a structure offset expression as the structure size?

@param plen (integer):
@param sptr: (C++: const struc_t *) struc_t const *
@param value (integer):

### IDAPython function idaapi.stroff_flag quick reference

stroff_flag() -> flags64_t
see FF_opbits

### IDAPython function idaapi.stru_flag quick reference

stru_flag() -> flags64_t
Get a flags64_t representing a struct.

### IDAPython function idaapi.struct_unpack quick reference

Unpack a buffer given its length and offset using struct.unpack_from().
This function will know how to unpack the given buffer by using the lookup table '__struct_u
If the buffer is of unknown length then None is returned. Otherwise the unpacked value is re

### IDAPython function idaapi.suspend_process quick reference

suspend_process() -> bool
Suspend the process in the debugger. \sq{ Type,
* Synchronous function (if in a notification handler)
* Asynchronous function (everywhere else)
* available as Request, Notification,
* none (if in a notification handler)
* dbg_suspend_process (everywhere else) }
@note: The suspend_process() function can be called from a notification handler
       to force the stopping of the process. In this case, no notification will
       be generated. When you suspend a process, the running command is always
       aborted.

### IDAPython function idaapi.suspend_thread quick reference

suspend_thread(tid) -> int
Suspend thread. Suspending a thread may deadlock the whole application if the
suspended was owning some synchronization objects. \sq{Type, Synchronous
```

```
function - available as request, Notification, none (synchronous function)}

@param tid (integer): thread id
@retval -1: network error
@retval 0: failed
@retval 1: ok

### IDAPython function idaapi.sval_pointer_frompointer quick reference

sval_pointer_frompointer(t) -> sval_pointer

@param t: sval_t *

### IDAPython function idaapi.swap_idcvs quick reference

swap_idcvs(v1, v2)
Swap 2 variables.

@param v1 (idaapi.idc_value_t):
@param v2 (idaapi.idc_value_t):

### IDAPython function idaapi.swap_mcode_relation quick reference

swap_mcode_relation(code) -> mcode_t

@param code: enum mcode_t

### IDAPython function idaapi.swapped_relation quick reference

swapped_relation(op) -> ctype_t
Swap a comparison operator. For example, cot_sge becomes cot_sle.

@param op: (C++: ctype_t) enum ctype_t

### IDAPython function idaapi.switch_dbctx quick reference

switch_dbctx(idx) -> dbctx_t *
Switch to the database with the provided context ID

@param idx (integer): the index of the database to switch to
@return: the current dbctx_t instance or nullptr

### IDAPython function idaapi.switch_info_t__from_ptrval__ quick reference

switch_info_t__from_ptrval__(ptrval) -> switch_info_t

@param ptrval: size_t

### IDAPython function idaapi.switch_to_golang quick reference

switch_to_golang()
switch to GOLANG calling convention (to be used as default CC)
```

```
### IDAPython function idaapi.sync_sources quick reference

sync_sources(what, _with, sync) -> bool
[Un]synchronize sources

@param what: (C++: const sync_source_t &)
@param with: (C++: const sync_source_t &)
@param sync (bool):
@return: success

### IDAPython function idaapi.tag_addr quick reference

tag_addr(ea) -> PyObject *
Insert an address mark into a string.

@param ea (integer): address to include

### IDAPython function idaapi.tag_advance quick reference

tag_advance(line, cnt) -> int
Move pointer to a 'line' to 'cnt' positions right. Take into account escape
sequences.

@param line (string): pointer to string
@param cnt (integer): number of positions to move right
@return: moved pointer

### IDAPython function idaapi.tag_remove quick reference

tag_remove(nonnul_instr) -> str

@param nonnul_instr: char const *

### IDAPython function idaapi.tag_skipcode quick reference

tag_skipcode(line) -> int
Skip one color code. This function should be used if you are interested in color
codes and want to analyze all of them. Otherwise tag_skipcodes() function is
better since it will skip all colors at once. This function will skip the
current color code if there is one. If the current symbol is not a color code,
it will return the input.

@param line (string): char const *
@return: moved pointer

### IDAPython function idaapi.tag_skipcodes quick reference

tag_skipcodes(line) -> int
Move the pointer past all color codes.

@param line (string): can't be nullptr
```

```
@return: moved pointer, can't be nullptr

### IDAPython function idaapi.tag_strlen quick reference

tag_strlen(line) -> ssize_t
Calculate length of a colored string This function computes the length in
unicode codepoints of a line

@param line (string): char const *
@return: the number of codepoints in the line, or -1 on error

### IDAPython function idaapi.take_database_snapshot quick reference

take_database_snapshot(ss) -> (bool, NoneType)
Take a database snapshot (ui_take_database_snapshot).

@param ss: (C++: snapshot_t *) in/out parameter.
* in: description, flags
* out: filename, id
@return: success

### IDAPython function idaapi.take_memory_snapshot quick reference

take_memory_snapshot(type) -> bool
Take a memory snapshot of the running process.

@param type (integer): specifies which snapshot we want (see SNAP_ Snapshot types)
@return: success

### IDAPython function idaapi.tbyte_flag quick reference

tbyte_flag() -> flags64_t
Get a flags64_t representing a tbyte.

### IDAPython function idaapi.term_hexrays_plugin quick reference

term_hexrays_plugin()
Stop working with hex-rays decompiler.

### IDAPython function idaapi.textctrl_info_t quick reference

Class representing textctrl_info_t

### IDAPython function idaapi.textctrl_info_t_assign quick reference

textctrl_info_t_assign(_self, other) -> bool

@param self: PyObject *
@param other: PyObject *

### IDAPython function idaapi.textctrl_info_t_create quick reference

textctrl_info_t_create() -> PyObject *
```

```
### IDAPython function idaapi.textctrl_info_t_destroy quick reference
textctrl_info_t_destroy(py_obj) -> bool

@param py_obj: PyObject *
### IDAPython function idaapi.textctrl_info_t_get_clink quick reference
textctrl_info_t_get_clink(_self) -> textctrl_info_t *

@param self: PyObject *
### IDAPython function idaapi.textctrl_info_t_get_clink_ptr quick reference
textctrl_info_t_get_clink_ptr(_self) -> PyObject *

@param self: PyObject *
### IDAPython function idaapi.textctrl_info_t_get_flags quick reference
textctrl_info_t_get_flags(_self) -> unsigned int

@param self: PyObject *
### IDAPython function idaapi.textctrl_info_t_get_tabsize quick reference
textctrl_info_t_get_tabsize(_self) -> unsigned int

@param self: PyObject *
### IDAPython function idaapi.textctrl_info_t_get_text quick reference
textctrl_info_t_get_text(_self) -> char const *

@param self: PyObject *
### IDAPython function idaapi.textctrl_info_t_set_flags quick reference
textctrl_info_t_set_flags(_self, flags) -> bool

@param self: PyObject *
@param flags: unsigned int
### IDAPython function idaapi.textctrl_info_t_set_tabsize quick reference
textctrl_info_t_set_tabsize(_self, tabsize) -> bool

@param self: PyObject *
@param tabsize: unsigned int
### IDAPython function idaapi.textctrl_info_t_set_text quick reference
textctrl_info_t_set_text(_self, s) -> bool
```

```
@param self: PyObject *
@param s: char const *

### IDAPython function idaapi.throw_idc_exception quick reference

throw_idc_exception(r, desc) -> error_t
Create an idc execution exception object. This helper function can be used to
return an exception from C++ code to IDC. In other words this function can be
called from idc_func_t() callbacks. Sample usage: if ( !ok ) return
throw_idc_exception(r, "detailed error msg");

@param r (idaapi.idc_value_t): object to hold the exception object
@param desc (string): exception description
@return: eExecThrow

### IDAPython function idaapi.tid_array_frompointer quick reference

tid_array_frompointer(t) -> tid_array

@param t: tid_t *

### IDAPython function idaapi.tinfo_t_get_stock quick reference

tinfo_t_get_stock(id) -> tinfo_t

@param id: enum stock_type_id_t

### IDAPython function idaapi.to_ea quick reference

to_ea(reg_cs, reg_ip) -> ea_t
Convert (sel,off) value to a linear address.

@param reg_cs: (C++: sel_t)
@param reg_ip (integer):

### IDAPython function idaapi.toggle_bnot quick reference

toggle_bnot(ea, n) -> bool
Toggle binary negation of operand. also see is_bnot()

@param ea (integer):
@param n (integer):

### IDAPython function idaapi.toggle_lzero quick reference

toggle_lzero(ea, n) -> bool
Toggle lzero bit.

@param ea (integer): the item (insn/data) address
@param n (integer): the operand number (0-first operand, 1-other operands)
@return: success
```

```
### IDAPython function idaapi.toggle_sign quick reference

toggle_sign(ea, n) -> bool
Toggle sign of n-th operand. allowed values of n: 0-first operand, 1-other
operands

@param ea (integer):
@param n (integer):

### IDAPython function idaapi.try_to_add_libfunc quick reference

try_to_add_libfunc(ea) -> int
Apply the currently loaded signature file to the specified address. If a library
function is found, then create a function and name it accordingly.

@param ea (integer): any address in the program
@return: Library function codes

### IDAPython function idaapi.ua_mnem quick reference

print_insn_mnem(ea) -> str
Print instruction mnemonics.

@param ea (integer): linear address of the instruction
@return: success

### IDAPython function idaapi.uchar_array_frompointer quick reference

uchar_array_frompointer(t) -> uchar_array

@param t: uchar *

### IDAPython function idaapi.udcall_map_begin quick reference

udcall_map_begin(map) -> udcall_map_iterator_t
Get iterator pointing to the beginning of udcall_map_t.

@param map: (C++: const udcall_map_t *) udcall_map_t const *

### IDAPython function idaapi.udcall_map_clear quick reference

udcall_map_clear(map)
Clear udcall_map_t.

@param map: (C++: udcall_map_t *)

### IDAPython function idaapi.udcall_map_end quick reference

udcall_map_end(map) -> udcall_map_iterator_t
Get iterator pointing to the end of udcall_map_t.

@param map: (C++: const udcall_map_t *) udcall_map_t const *
```

```
### IDAPython function idaapi.udcall_map_erase quick reference

udcall_map_erase(map, p)
Erase current element from udcall_map_t.

@param map: (C++: udcall_map_t *)
@param p: (C++: udcall_map_iterator_t)

### IDAPython function idaapi.udcall_map_find quick reference

udcall_map_find(map, key) -> udcall_map_iterator_t
Find the specified key in udcall_map_t.

@param map: (C++: const udcall_map_t *) udcall_map_t const *
@param key: (C++: const ea_t &) ea_t const &

### IDAPython function idaapi.udcall_map_first quick reference

udcall_map_first(p) -> ea_t const &
Get reference to the current map key.

@param p: (C++: udcall_map_iterator_t)

### IDAPython function idaapi.udcall_map_free quick reference

udcall_map_free(map)
Delete udcall_map_t instance.

@param map: (C++: udcall_map_t *)

### IDAPython function idaapi.udcall_map_insert quick reference

udcall_map_insert(map, key, val) -> udcall_map_iterator_t
Insert new (ea_t, udcall_t) pair into udcall_map_t.

@param map: (C++: udcall_map_t *)
@param key: (C++: const ea_t &) ea_t const &
@param val: (C++: const udcall_t &) udcall_t const &

### IDAPython function idaapi.udcall_map_new quick reference

udcall_map_new() -> udcall_map_t *
Create a new udcall_map_t instance.

### IDAPython function idaapi.udcall_map_next quick reference

udcall_map_next(p) -> udcall_map_iterator_t
Move to the next element.

@param p: (C++: udcall_map_iterator_t)

### IDAPython function idaapi.udcall_map_prev quick reference
```

```
udcall_map_prev(p) -> udcall_map_iterator_t
Move to the previous element.

@param p: (C++: udcall_map_iterator_t)

### IDAPython function idaapi.udcall_map_second quick reference

udcall_map_second(p) -> udcall_t
Get reference to the current map value.

@param p: (C++: udcall_map_iterator_t)

### IDAPython function idaapi.udcall_map_size quick reference

udcall_map_size(map) -> size_t
Get size of udcall_map_t.

@param map: (C++: udcall_map_t *)

### IDAPython function idaapi.ui_load_new_file quick reference

ui_load_new_file(temp_file, filename, pli, neflags, ploaders) -> bool
Display a load file dialog and load file (ui_load_file).

@param temp_file: (C++: qstring *) name of the file with the extracted archive member.
@param filename: (C++: qstring *) the name of input file as is, library or archive name
@param pli: (C++: linput_t **) loader input source, may be changed to point to temp_file
@param neflags: (C++: ushort) combination of NEF_... bits (see Load file flags)
@param ploaders: (C++: load_info_t **) list of loaders which accept file, may be changed fo
                 of temp_file
@retval true: file was successfully loaded
@retval false: otherwise

### IDAPython function idaapi.ui_run_debugger quick reference

ui_run_debugger(dbgopts, exename, argc, argv) -> bool
Load a debugger plugin and run the specified program (ui_run_dbg).

@param dbgopts (string): value of the -r command line switch
@param exename (string): name of the file to run
@param argc (integer): number of arguments for the executable
@param argv: (C++: const char *const *) argument vector
@return: success

### IDAPython function idaapi.unhide_border quick reference

unhide_border(ea)

@param ea: ea_t

### IDAPython function idaapi.unhide_item quick reference
```

```
unhide_item(ea)

@param ea: ea_t

### IDAPython function idaapi.unmark_selection quick reference

unmark_selection()
Unmark selection (ui_unmarksel)

### IDAPython function idaapi.unpack_idcobj_from_bv quick reference

unpack_idcobj_from_bv(obj, tif, bytes, pio_flags=0) -> error_t
Read a typed idc object from the byte vector.

@param obj (idaapi.idc_value_t):
@param tif (idaapi.tinfo_t): tinfo_t const &
@param bytes: (C++: const bytevec_t &) bytevec_t const &
@param pio_flags (integer):

### IDAPython function idaapi.unpack_idcobj_from_idb quick reference

unpack_idcobj_from_idb(obj, tif, ea, off0, pio_flags=0) -> error_t
Collection of register objects.

Read a typed idc object from the database

@param obj (idaapi.idc_value_t):
@param tif (idaapi.tinfo_t): tinfo_t const &
@param ea (integer):
@param off0: (C++: const bytevec_t *) bytevec_t const *
@param pio_flags (integer):

### IDAPython function idaapi.unpack_object_from_bv quick reference

unpack_object_from_bv(ti, type, fields, bytes, pio_flags=0) -> PyObject *
Unpacks a buffer into an object.
Returns the error_t returned by idaapi.pack_object_to_idb

@param ti: Type info. 'None' can be passed.
@param type: type_t const *
@param fields: fields string (may be empty or None)
@param bytes: the bytes to unpack
@param pio_flags: flags used while unpacking
@return:     - tuple(0, err) on failure
    - tuple(1, obj) on success

### IDAPython function idaapi.unpack_object_from_idb quick reference

unpack_object_from_idb(ti, type, fields, ea, pio_flags=0) -> PyObject *

@param ti: til_t *
```

```
@param type: type_t const *
@param fields: p_list const *
@param ea: ea_t
@param pio_flags: int

### IDAPython function idaapi.unregister_action quick reference

unregister_action(name) -> bool
Delete a previously-registered action (ui_unregister_action).

@param name (string): name of action
@return: success

### IDAPython function idaapi.unregister_custom_data_format quick reference

unregister_custom_data_format(dfid) -> bool
Unregisters a custom data format

@param dfid: data format id
@return: Boolean

### IDAPython function idaapi.unregister_custom_data_type quick reference

unregister_custom_data_type(dtid) -> bool
Unregisters a custom data type.

@param dtid: the data type id
@return: Boolean

### IDAPython function idaapi.unregister_data_types_and_formats quick reference

As opposed to register_data_types_and_formats(), this function
unregisters multiple data types and formats at once.

### IDAPython function idaapi.unregister_timer quick reference

unregister_timer(py_timerctx) -> bool
Unregister a timer

@param timer_obj: a timer object previously returned by a register_timer()
@return: Boolean
@note: After the timer has been deleted, the timer_obj will become invalid.

### IDAPython function idaapi.upd_abits quick reference

upd_abits(ea, clr_bits, set_bits)

@param ea: ea_t
@param clr_bits: aflags_t
@param set_bits: aflags_t

### IDAPython function idaapi.update_action_checkable quick reference
```

```
update_action_checkable(name, checkable) -> bool
Update an action's checkability (ui_update_action_attr).

@param name (string): action name
@param checkable (bool): new checkability
@return: success
```

IDAPython function idaapi.update_action_checked quick reference

```
update_action_checked(name, checked) -> bool
Update an action's checked state (ui_update_action_attr).

@param name (string): action name
@param checked (bool): new checked state
@return: success
```

IDAPython function idaapi.update_action_icon quick reference

```
update_action_icon(name, icon) -> bool
Update an action's icon (ui_update_action_attr).

@param name (string): action name
@param icon (integer): new icon id
@return: success
```

IDAPython function idaapi.update_action_label quick reference

```
update_action_label(name, label) -> bool
Update an action's label (ui_update_action_attr).

@param name (string): action name
@param label (string): new label
@return: success
```

IDAPython function idaapi.update_action_shortcut quick reference

```
update_action_shortcut(name, shortcut) -> bool
Update an action's shortcut (ui_update_action_attr).

@param name (string): action name
@param shortcut (string): new shortcut
@return: success
```

IDAPython function idaapi.update_action_state quick reference

```
update_action_state(name, state) -> bool
Update an action's state (ui_update_action_attr).

@param name (string): action name
@param state: (C++: action_state_t) new state
@return: success
```

```
### IDAPython function idaapi.update_action_tooltip quick reference

update_action_tooltip(name, tooltip) -> bool
Update an action's tooltip (ui_update_action_attr).

@param name (string): action name
@param tooltip (string): new tooltip
@return: success

### IDAPython function idaapi.update_action_visibility quick reference

update_action_visibility(name, visible) -> bool
Update an action's visibility (ui_update_action_attr).

@param name (string): action name
@param visible (bool): new visibility
@return: success

### IDAPython function idaapi.update_bpt quick reference

update_bpt(bpt) -> bool
Update modifiable characteristics of an existing breakpoint. To update the
breakpoint location, use change_bptlocs() \sq{Type, Synchronous function,
Notification, none (synchronous function)}
@note: Only the following fields can be modified:
* bpt_t::cndbody
* bpt_t::pass_count
* bpt_t::flags
* bpt_t::size
* bpt_t::type
@note: Changing some properties will require removing and then re-adding the
       breakpoint to the process memory (or the debugger backend), which can
       lead to race conditions (i.e., breakpoint(s) can be missed) in case the
       process is not suspended. Here are a list of scenarios that will require
       the breakpoint to be removed & then re-added:
* bpt_t::size is modified
* bpt_t::type is modified
* bpt_t::flags's BPT_ENABLED is modified
* bpt_t::flags's BPT_LOWCND is changed
* bpt_t::flags's BPT_LOWCND remains set, but cndbody changed

@param bpt: (C++: const bpt_t *) bpt_t const *

### IDAPython function idaapi.update_extra_cmt quick reference

update_extra_cmt(ea, what, str)

@param ea: ea_t
@param what: int
```

```
@param str: char const *

### IDAPython function idaapi.update_fpd quick reference

update_fpd(pfn, fpd) -> bool
Update frame pointer delta.

@param pfn (idaapi.func_t): pointer to function structure
@param fpd (integer): new fpd value. cannot be bigger than the local variable range size.
@return: success

### IDAPython function idaapi.update_func quick reference

update_func(pfn) -> bool
Update information about a function in the database (func_t). You must not
change the function start and end addresses using this function. Use
set_func_start() and set_func_end() for it.

@param pfn (idaapi.func_t): ptr to function structure
@return: success

### IDAPython function idaapi.update_hidden_range quick reference

update_hidden_range(ha) -> bool
Update hidden range information in the database. You cannot use this function to
change the range boundaries

@param ha: (C++: const hidden_range_t *) range to update
@return: success

### IDAPython function idaapi.update_segm quick reference

update_segm(s) -> bool

@param s: segment_t *

### IDAPython function idaapi.use_golang_cc quick reference

use_golang_cc() -> bool
is GOLANG calling convention used by default?

### IDAPython function idaapi.use_mapping quick reference

use_mapping(ea) -> ea_t
Translate address according to current mappings.

@param ea (integer): address to translate
@return: translated address

### IDAPython function idaapi.user_cancelled quick reference

user_cancelled() -> bool
Test the ctrl-break flag (ui_test_cancelled).
```

```
@retval 1: Ctrl-Break is detected, a message is displayed
@retval 2: Ctrl-Break is detected again, a message is not displayed
@retval 0: Ctrl-Break is not detected

### IDAPython function idaapi.user_cmts_begin quick reference

user_cmts_begin(map) -> user_cmts_iterator_t
Get iterator pointing to the beginning of user_cmts_t.

@param map: (C++: const user_cmts_t *) user_cmts_t const *

### IDAPython function idaapi.user_cmts_clear quick reference

user_cmts_clear(map)
Clear user_cmts_t.

@param map: (C++: user_cmts_t *)

### IDAPython function idaapi.user_cmts_end quick reference

user_cmts_end(map) -> user_cmts_iterator_t
Get iterator pointing to the end of user_cmts_t.

@param map: (C++: const user_cmts_t *) user_cmts_t const *

### IDAPython function idaapi.user_cmts_erase quick reference

user_cmts_erase(map, p)
Erase current element from user_cmts_t.

@param map: (C++: user_cmts_t *)
@param p: (C++: user_cmts_iterator_t)

### IDAPython function idaapi.user_cmts_find quick reference

user_cmts_find(map, key) -> user_cmts_iterator_t
Find the specified key in user_cmts_t.

@param map: (C++: const user_cmts_t *) user_cmts_t const *
@param key: (C++: const treeloc_t &) treeloc_t const &

### IDAPython function idaapi.user_cmts_first quick reference

user_cmts_first(p) -> treeloc_t
Get reference to the current map key.

@param p: (C++: user_cmts_iterator_t)

### IDAPython function idaapi.user_cmts_free quick reference

user_cmts_free(map)
Delete user_cmts_t instance.
```

```
@param map: (C++: user_cmts_t *)

### IDAPython function idaapi.user_cmts_insert quick reference

user_cmts_insert(map, key, val) -> user_cmts_iterator_t
Insert new (treeloc_t, citem_cmt_t) pair into user_cmts_t.

@param map: (C++: user_cmts_t *)
@param key: (C++: const treeloc_t &) treeloc_t const &
@param val: (C++: const citem_cmt_t &) citem_cmt_t const &

### IDAPython function idaapi.user_cmts_new quick reference

user_cmts_new() -> user_cmts_t
Create a new user_cmts_t instance.

### IDAPython function idaapi.user_cmts_next quick reference

user_cmts_next(p) -> user_cmts_iterator_t
Move to the next element.

@param p: (C++: user_cmts_iterator_t)

### IDAPython function idaapi.user_cmts_prev quick reference

user_cmts_prev(p) -> user_cmts_iterator_t
Move to the previous element.

@param p: (C++: user_cmts_iterator_t)

### IDAPython function idaapi.user_cmts_second quick reference

user_cmts_second(p) -> citem_cmt_t
Get reference to the current map value.

@param p: (C++: user_cmts_iterator_t)

### IDAPython function idaapi.user_cmts_size quick reference

user_cmts_size(map) -> size_t
Get size of user_cmts_t.

@param map: (C++: user_cmts_t *)

### IDAPython function idaapi.user_iflags_begin quick reference

user_iflags_begin(map) -> user_iflags_iterator_t
Get iterator pointing to the beginning of user_iflags_t.

@param map: (C++: const user_iflags_t *) user_iflags_t const *

### IDAPython function idaapi.user_iflags_clear quick reference
```

```
user_iflags_clear(map)
Clear user_iflags_t.

@param map: (C++: user_iflags_t *)

### IDAPython function idaapi.user_iflags_end quick reference

user_iflags_end(map) -> user_iflags_iterator_t
Get iterator pointing to the end of user_iflags_t.

@param map: (C++: const user_iflags_t *) user_iflags_t const *

### IDAPython function idaapi.user_iflags_erase quick reference

user_iflags_erase(map, p)
Erase current element from user_iflags_t.

@param map: (C++: user_iflags_t *)
@param p: (C++: user_iflags_iterator_t)

### IDAPython function idaapi.user_iflags_find quick reference

user_iflags_find(map, key) -> user_iflags_iterator_t
Find the specified key in user_iflags_t.

@param map: (C++: const user_iflags_t *) user_iflags_t const *
@param key: (C++: const citem_locator_t &) citem_locator_t const &

### IDAPython function idaapi.user_iflags_first quick reference

user_iflags_first(p) -> citem_locator_t
Get reference to the current map key.

@param p: (C++: user_iflags_iterator_t)

### IDAPython function idaapi.user_iflags_free quick reference

user_iflags_free(map)
Delete user_iflags_t instance.

@param map: (C++: user_iflags_t *)

### IDAPython function idaapi.user_iflags_insert quick reference

user_iflags_insert(map, key, val) -> user_iflags_iterator_t
Insert new (citem_locator_t, int32) pair into user_iflags_t.

@param map: (C++: user_iflags_t *)
@param key: (C++: const citem_locator_t &) citem_locator_t const &
@param val: (C++: const int32 &) int32 const &

### IDAPython function idaapi.user_iflags_new quick reference
```

```
user_iflags_new() -> user_iflags_t
Create a new user_iflags_t instance.

### IDAPython function idaapi.user_iflags_next quick reference

user_iflags_next(p) -> user_iflags_iterator_t
Move to the next element.

@param p: (C++: user_iflags_iterator_t)

### IDAPython function idaapi.user_iflags_prev quick reference

user_iflags_prev(p) -> user_iflags_iterator_t
Move to the previous element.

@param p: (C++: user_iflags_iterator_t)

### IDAPython function idaapi.user_iflags_second quick reference

user_iflags_second(p) -> int32 const &
Get reference to the current map value.

@param p: (C++: user_iflags_iterator_t)

### IDAPython function idaapi.user_iflags_size quick reference

user_iflags_size(map) -> size_t
Get size of user_iflags_t.

@param map: (C++: user_iflags_t *)

### IDAPython function idaapi.user_labels_begin quick reference

user_labels_begin(map) -> user_labels_iterator_t
Get iterator pointing to the beginning of user_labels_t.

@param map: (C++: const user_labels_t *) user_labels_t const *

### IDAPython function idaapi.user_labels_clear quick reference

user_labels_clear(map)
Clear user_labels_t.

@param map: (C++: user_labels_t *)

### IDAPython function idaapi.user_labels_end quick reference

user_labels_end(map) -> user_labels_iterator_t
Get iterator pointing to the end of user_labels_t.

@param map: (C++: const user_labels_t *) user_labels_t const *

### IDAPython function idaapi.user_labels_erase quick reference
```

```
user_labels_erase(map, p)
Erase current element from user_labels_t.

@param map: (C++: user_labels_t *)
@param p: (C++: user_labels_iterator_t)

### IDAPython function idaapi.user_labels_find quick reference

user_labels_find(map, key) -> user_labels_iterator_t
Find the specified key in user_labels_t.

@param map: (C++: const user_labels_t *) user_labels_t const *
@param key: (C++: const int &) int const &

### IDAPython function idaapi.user_labels_first quick reference

user_labels_first(p) -> int const &
Get reference to the current map key.

@param p: (C++: user_labels_iterator_t)

### IDAPython function idaapi.user_labels_free quick reference

user_labels_free(map)
Delete user_labels_t instance.

@param map: (C++: user_labels_t *)

### IDAPython function idaapi.user_labels_insert quick reference

user_labels_insert(map, key, val) -> user_labels_iterator_t
Insert new (int, qstring) pair into user_labels_t.

@param map: (C++: user_labels_t *)
@param key: (C++: const int &) int const &
@param val: (C++: const qstring &) qstring const &

### IDAPython function idaapi.user_labels_new quick reference

user_labels_new() -> user_labels_t
Create a new user_labels_t instance.

### IDAPython function idaapi.user_labels_next quick reference

user_labels_next(p) -> user_labels_iterator_t
Move to the next element.

@param p: (C++: user_labels_iterator_t)

### IDAPython function idaapi.user_labels_prev quick reference

user_labels_prev(p) -> user_labels_iterator_t
Move to the previous element.
```

```
@param p: (C++: user_labels_iterator_t)

### IDAPython function idaapi.user_labels_second quick reference

user_labels_second(p) -> qstring &
Get reference to the current map value.

@param p: (C++: user_labels_iterator_t)

### IDAPython function idaapi.user_labels_size quick reference

user_labels_size(map) -> size_t
Get size of user_labels_t.

@param map: (C++: user_labels_t *)

### IDAPython function idaapi.user_numforms_begin quick reference

user_numforms_begin(map) -> user_numforms_iterator_t
Get iterator pointing to the beginning of user_numforms_t.

@param map: (C++: const user_numforms_t *) user_numforms_t const *

### IDAPython function idaapi.user_numforms_clear quick reference

user_numforms_clear(map)
Clear user_numforms_t.

@param map: (C++: user_numforms_t *)

### IDAPython function idaapi.user_numforms_end quick reference

user_numforms_end(map) -> user_numforms_iterator_t
Get iterator pointing to the end of user_numforms_t.

@param map: (C++: const user_numforms_t *) user_numforms_t const *

### IDAPython function idaapi.user_numforms_erase quick reference

user_numforms_erase(map, p)
Erase current element from user_numforms_t.

@param map: (C++: user_numforms_t *)
@param p: (C++: user_numforms_iterator_t)

### IDAPython function idaapi.user_numforms_find quick reference

user_numforms_find(map, key) -> user_numforms_iterator_t
Find the specified key in user_numforms_t.

@param map: (C++: const user_numforms_t *) user_numforms_t const *
@param key: (C++: const operand_locator_t &) operand_locator_t const &
```

```
### IDAPython function idaapi.user_numforms_first quick reference

user_numforms_first(p) -> operand_locator_t
Get reference to the current map key.

@param p: (C++: user_numforms_iterator_t)

### IDAPython function idaapi.user_numforms_free quick reference

user_numforms_free(map)
Delete user_numforms_t instance.

@param map: (C++: user_numforms_t *)

### IDAPython function idaapi.user_numforms_insert quick reference

user_numforms_insert(map, key, val) -> user_numforms_iterator_t
Insert new (operand_locator_t, number_format_t) pair into user_numforms_t.

@param map: (C++: user_numforms_t *)
@param key: (C++: const operand_locator_t &) operand_locator_t const &
@param val: (C++: const number_format_t &) number_format_t const &

### IDAPython function idaapi.user_numforms_new quick reference

user_numforms_new() -> user_numforms_t
Create a new user_numforms_t instance.

### IDAPython function idaapi.user_numforms_next quick reference

user_numforms_next(p) -> user_numforms_iterator_t
Move to the next element.

@param p: (C++: user_numforms_iterator_t)

### IDAPython function idaapi.user_numforms_prev quick reference

user_numforms_prev(p) -> user_numforms_iterator_t
Move to the previous element.

@param p: (C++: user_numforms_iterator_t)

### IDAPython function idaapi.user_numforms_second quick reference

user_numforms_second(p) -> number_format_t
Get reference to the current map value.

@param p: (C++: user_numforms_iterator_t)

### IDAPython function idaapi.user_numforms_size quick reference

user_numforms_size(map) -> size_t
Get size of user_numforms_t.
```

```
@param map: (C++: user_numforms_t *)

### IDAPython function idaapi.user_unions_begin quick reference

user_unions_begin(map) -> user_unions_iterator_t
Get iterator pointing to the beginning of user_unions_t.

@param map: (C++: const user_unions_t *) user_unions_t const *

### IDAPython function idaapi.user_unions_clear quick reference

user_unions_clear(map)
Clear user_unions_t.

@param map: (C++: user_unions_t *)

### IDAPython function idaapi.user_unions_end quick reference

user_unions_end(map) -> user_unions_iterator_t
Get iterator pointing to the end of user_unions_t.

@param map: (C++: const user_unions_t *) user_unions_t const *

### IDAPython function idaapi.user_unions_erase quick reference

user_unions_erase(map, p)
Erase current element from user_unions_t.

@param map: (C++: user_unions_t *)
@param p: (C++: user_unions_iterator_t)

### IDAPython function idaapi.user_unions_find quick reference

user_unions_find(map, key) -> user_unions_iterator_t
Find the specified key in user_unions_t.

@param map: (C++: const user_unions_t *) user_unions_t const *
@param key: (C++: const ea_t &) ea_t const &

### IDAPython function idaapi.user_unions_first quick reference

user_unions_first(p) -> ea_t const &
Get reference to the current map key.

@param p: (C++: user_unions_iterator_t)

### IDAPython function idaapi.user_unions_free quick reference

user_unions_free(map)
Delete user_unions_t instance.

@param map: (C++: user_unions_t *)
```

```
### IDAPython function idaapi.user_unions_insert quick reference

user_unions_insert(map, key, val) -> user_unions_iterator_t
Insert new (ea_t, intvec_t) pair into user_unions_t.

@param map: (C++: user_unions_t *)
@param key: (C++: const ea_t &) ea_t const &
@param val: (C++: const intvec_t &) intvec_t const &

### IDAPython function idaapi.user_unions_new quick reference

user_unions_new() -> user_unions_t
Create a new user_unions_t instance.

### IDAPython function idaapi.user_unions_next quick reference

user_unions_next(p) -> user_unions_iterator_t
Move to the next element.

@param p: (C++: user_unions_iterator_t)

### IDAPython function idaapi.user_unions_prev quick reference

user_unions_prev(p) -> user_unions_iterator_t
Move to the previous element.

@param p: (C++: user_unions_iterator_t)

### IDAPython function idaapi.user_unions_second quick reference

user_unions_second(p) -> intvec_t
Get reference to the current map value.

@param p: (C++: user_unions_iterator_t)

### IDAPython function idaapi.user_unions_size quick reference

user_unions_size(map) -> size_t
Get size of user_unions_t.

@param map: (C++: user_unions_t *)

### IDAPython function idaapi.uses_aflag_modsp quick reference

uses_aflag_modsp(flags) -> bool

@param flags: aflags_t

### IDAPython function idaapi.uses_modsp quick reference

uses_modsp(ea) -> bool

@param ea: ea_t
```

```
### IDAPython function idaapi.uval_array_frompointer quick reference

uval_array_frompointer(t) -> uval_array

@param t: uval_t *

### IDAPython function idaapi.validate_idb quick reference

validate_idb(vld_flags=0) -> size_t
Validate the database

@param vld_flags (integer): combination of VLD_.. constants
@return: number of corrupted/fixed records

### IDAPython function idaapi.validate_idb_names quick reference

validate_idb_names(do_repair) -> int

@param do_repair: bool

### IDAPython function idaapi.validate_name quick reference

validate_name(name, type, flags=0) -> PyObject *
Validate a name. This function replaces all invalid characters in the name with
SUBSTCHAR. However, it will return false if name is valid but not allowed to be
an identifier (is a register name).

@param name: (C++: qstring *) ptr to name. the name will be modified
@param type: (C++: nametype_t) the type of name we want to validate
@param flags (integer): see SN_* . Only SN_IDBENC is currently considered
@return: success

### IDAPython function idaapi.verify_argloc quick reference

verify_argloc(vloc, size, gaps) -> int
Verify argloc_t.

@param vloc: (C++: const argloc_t &) argloc to verify
@param size (integer): total size of the variable
@param gaps: (C++: const rangeset_t *) if not nullptr, specifies gaps in structure definitio
             should not map to any argloc, but everything else must be covered
@return: 0 if ok, otherwise an interr code.

### IDAPython function idaapi.verify_tinfo quick reference

verify_tinfo(typid) -> int

@param typid: uint32

### IDAPython function idaapi.viewer_attach_menu_item quick reference

viewer_attach_menu_item(g, name) -> bool
```

```
Attach a previously-registered action to the view's context menu. See
kernwin.hpp for how to register actions.

@param g: (C++: graph_viewer_t *) graph viewer
@param name (string): action name
@return: success

### IDAPython function idaapi.viewer_center_on quick reference

viewer_center_on(gv, node)
Center the graph view on the given node.

@param gv: (C++: graph_viewer_t *)
@param node (integer):

### IDAPython function idaapi.viewer_create_groups quick reference

viewer_create_groups(gv, out_group_nodes, gi) -> bool
This will perform an operation similar to what happens when a user manually
selects a set of nodes, right-clicks and selects "Create group". This is a
wrapper around mutable_graph_t::create_group that will, in essence:
* clone the current graph
* for each group_crinfo_t, attempt creating group in that new graph
* if all were successful, animate to that new graph.
@note: this accepts parameters that allow creating of multiple groups at once;
       which means only one graph animation will be triggered.

@param gv: (C++: graph_viewer_t *)
@param out_group_nodes: (C++: intvec_t *)
@param gi: (C++: const groups_crinfos_t &) groups_crinfos_t const &

### IDAPython function idaapi.viewer_del_node_info quick reference

viewer_del_node_info(gv, n)
Delete node info for node in given viewer (see del_node_info())

@param gv: (C++: graph_viewer_t *)
@param n (integer):

### IDAPython function idaapi.viewer_delete_groups quick reference

viewer_delete_groups(gv, groups, new_current=-1) -> bool
Wrapper around mutable_graph_t::delete_group. This function will:
* clone the current graph
* attempt deleting the groups in that new graph
* if successful, animate to that new graph.

@param gv: (C++: graph_viewer_t *)
@param groups: (C++: const intvec_t &) intvec_t const &
@param new_current (integer):
```

```
### IDAPython function idaapi.viewer_fit_window quick reference

viewer_fit_window(gv)
Fit graph viewer to its parent form.

@param gv: (C++: graph_viewer_t *)

### IDAPython function idaapi.viewer_get_curnode quick reference

viewer_get_curnode(gv) -> int
Get number of currently selected node (-1 if none)

@param gv: (C++: graph_viewer_t *)

### IDAPython function idaapi.viewer_get_gli quick reference

viewer_get_gli(out, gv, flags=0) -> bool
Get location info for given graph view If flags contains GLICTL_CENTER, then the
gli that will be retrieved, will be the one at the center of the view. Otherwise
it will be the top-left.

@param out: (C++: graph_location_info_t *)
@param gv: (C++: graph_viewer_t *)
@param flags (integer):

### IDAPython function idaapi.viewer_get_node_info quick reference

viewer_get_node_info(gv, out, n) -> bool
Get node info for node in given viewer (see get_node_info())

@param gv: (C++: graph_viewer_t *)
@param out: (C++: node_info_t *)
@param n (integer):

### IDAPython function idaapi.viewer_get_selection quick reference

viewer_get_selection(gv, sgs) -> bool
Get currently selected items for graph viewer.

@param gv: (C++: graph_viewer_t *)
@param sgs: (C++: screen_graph_selection_t *)

### IDAPython function idaapi.viewer_set_gli quick reference

viewer_set_gli(gv, gli, flags=0)
Set location info for given graph view If flags contains GLICTL_CENTER, then the
gli will be set to be the center of the view. Otherwise it will be the top-left.

@param gv: (C++: graph_viewer_t *)
@param gli: (C++: const graph_location_info_t *) graph_location_info_t const *
@param flags (integer):
```

```
### IDAPython function idaapi.viewer_set_groups_visibility quick reference

viewer_set_groups_visibility(gv, groups, expand, new_current=-1) -> bool
Wrapper around mutable_graph_t::change_visibility. This function will:
* clone the current graph
* attempt changing visibility of the groups in that new graph
* if successful, animate to that new graph.

@param gv: (C++: graph_viewer_t *)
@param groups: (C++: const intvec_t &) intvec_t const &
@param expand (bool):
@param new_current (integer):

### IDAPython function idaapi.viewer_set_node_info quick reference

viewer_set_node_info(gv, n, ni, flags)
Set node info for node in given viewer (see set_node_info())

@param gv: (C++: graph_viewer_t *)
@param n (integer):
@param ni: (C++: const node_info_t &) node_info_t const &
@param flags (integer):

### IDAPython function idaapi.viewer_set_titlebar_height quick reference

viewer_set_titlebar_height(gv, height) -> int
Set height of node title bars (grcode_set_titlebar_height)

@param gv: (C++: graph_viewer_t *)
@param height (integer):

### IDAPython function idaapi.visit_patched_bytes quick reference

visit_patched_bytes(ea1, ea2, py_callable) -> int
Enumerates patched bytes in the given range and invokes a callable

@param ea1: start address
@param ea2: end address
@param py_callable: a Python callable with the following prototype:
                callable(ea, fpos, org_val, patch_val).
                If the callable returns non-zero then that value will be
                returned to the caller and the enumeration will be
                interrupted.
@return: Zero if the enumeration was successful or the return
         value of the callback if enumeration was interrupted.

### IDAPython function idaapi.visit_stroff_fields quick reference

visit_stroff_fields(sfv, path, disp, appzero) -> flags64_t
Visit structure fields in a stroff expression or in a reference to a struct data
```

```
variable. This function can be used to enumerate all components of an expression
like 'a.b.c'.

@param sfv: (C++: struct_field_visitor_t &) visitor object
@param path: (C++: const tid_t *) struct path (path[0] contains the initial struct id)
@param disp: (C++: adiff_t *) offset into structure
@param appzero (bool): should visit field at offset zero?

### IDAPython function idaapi.visit_subtypes quick reference

visit_subtypes(visitor, out, tif, name, cmt) -> int

@param visitor: tinfo_visitor_t *
@param out: type_mods_t *
@param tif: tinfo_t const &
@param name: char const *
@param cmt: char const *

### IDAPython function idaapi.wait_for_next_event quick reference

wait_for_next_event(wfne, timeout) -> dbg_event_code_t
Wait for the next event.

This function (optionally) resumes the process execution, and waits for a
debugger event until a possible timeout occurs.

@param wfne (integer): combination of Wait for debugger event flags constants
@param timeout (integer): number of seconds to wait, -1-infinity
@return: either an event_id_t (if > 0), or a dbg_event_code_t (if <= 0)

### IDAPython function idaapi.warning quick reference

warning(format)
Display a message in a message box

@param message: message to print (formatting is done in Python)

This function can be used to debug IDAPython scripts
The user will be able to hide messages if they appear twice in a row on
the screen

### IDAPython function idaapi.was_ida_decision quick reference

was_ida_decision(ea) -> bool

@param ea: ea_t

### IDAPython function idaapi.word_flag quick reference

word_flag() -> flags64_t
Get a flags64_t representing a word.
```

```
### IDAPython function idaapi.write_dbg_memory quick reference

write_dbg_memory(ea, py_buf, size=size_t(-1)) -> ssize_t

@param ea: ea_t
@param py_buf: PyObject *
@param size: size_t

### IDAPython function idaapi.write_tinfo_bitfield_value quick reference

write_tinfo_bitfield_value(typid, dst, v, bitoff) -> uint64

@param typid: uint32
@param dst: uint64
@param v: uint64
@param bitoff: int

### IDAPython function idaapi.writebytes quick reference

writebytes(h, l, size, mf) -> int
Write at most 4 bytes to file.

@param h (integer): file handle
@param l (integer): value to write
@param size (integer): size of value in bytes (1,2,4)
@param mf (bool): is MSB first?
@return: 0 on success, nonzero otherwise

### IDAPython function idaapi.xrefchar quick reference

xrefchar(xrtype) -> char
Get character describing the xref type.

@param xrtype: (C++: char) combination of Cross-Reference type flags and a cref_t of dref_t
               value

### IDAPython function idaapi.yword_flag quick reference

yword_flag() -> flags64_t
Get a flags64_t representing a ymm word.

### IDAPython function idaapi.zword_flag quick reference

zword_flag() -> flags64_t
Get a flags64_t representing a zmm word.
‘‘‘
```