IDAPython Example Scripts

This document contains a variety of IDAPython scripts that are useful for reverse engineering tasks.

They cover various aspects of the IDAPython framework: core, UI, database, etc.

IDAPython example: analysis/dump_func_info.py

dump (some) information about the current function.

description: Dump some of the most interesting bits of information about the function we are currently looking at.

```
import binascii

import ida_kernwin
import ida_funcs


def dump_flags(fn):
    "dump some flags of the func_t object"
    print("Function flags: %08X" % fn.flags)
    if fn.is_far():
        print("  Far function")
    if not fn.does_return():
        print("  Function does not return")
    if fn.flags & ida_funcs.FUNC_FRAME:
        print("  Function uses frame pointer")
    if fn.flags & ida_funcs.FUNC_THUNK:
        print("  Thunk function")
    if fn.flags & ida_funcs.FUNC_LUMINA:
        print("  Function info is provided by Lumina")
    if fn.flags & ida_funcs.FUNC_OUTLINE:
        print("  Outlined code, not a real function")


def dump_regvars(pfn):
    "dump renamed registers information"
    assert ida_funcs.is_func_entry(pfn)
    print("Function has %d renamed registers" % pfn.regvarqty)
    for rv in pfn.regvars:
        print("%08X..%08X '%s'->'%s'" % (rv.start_ea, rv.end_ea, rv.canon, rv.user))
```

```
def dump_regargs(pfn):
    "dump register arguments information"
    assert ida_funcs.is_func_entry(pfn)
    print("Function has %d register arguments" % pfn.regargqty)
    for ra in pfn.regargs:
        print("  register #=%d, argument name=\"%s\", (serialized) type=\"%s\"" % (
            ra.reg,
            ra.name,
            binascii.hexlify(ra.type)))


def dump_tails(pfn):
    "dump function tails for entry chunk pfn"
    assert ida_funcs.is_func_entry(pfn)
    print("Function has %d tails" % pfn.tailqty)
    for i in range(pfn.tailqty):
        ft = pfn.tails[i]
        print("  tail %i: %08X..%08X" % (i, ft.start_ea, ft.end_ea))


def dump_stkpnts(pfn):
    "dump function stack points"
    print("Function has %d stack points" % pfn.pntqty)
    for i in range(pfn.pntqty):
        pnt = pfn.points[i]
        print("  stkpnt %i @%08X: %d" % (i, pnt.ea, pnt.spd))


def dump_frame(fn):
    "dump function frame info"
    assert ida_funcs.is_func_entry(fn)
    print("frame structure id: %08X" % fn.frame)
    print("local variables area size: %8X" % fn.frsize)
    print("saved registers area size: %8X" % fn.frregs)
    print("bytes purged on return   : %8X" % fn.argsize)
    print("frame pointer delta      : %8X" % fn.fpd)


def dump_parents(fn):
    "dump parents of a function tail"
    assert ida_funcs.is_func_tail(fn)
    print("owner function: %08X" % fn.owner)
    print("tail has %d referers" % fn.refqty)
    for i in range(fn.refqty):
        print("  referer %i: %08X" % (i, fn.referers[i]))
```

```
def dump_func_info(ea):
    "dump info about function chunk at address 'ea'"
    pfn = ida_funcs.get_fchunk(ea)
    if pfn is None:
        print("No function at %08X!" % ea)
        return
    print("current chunk boundaries: %08X..%08X" % (pfn.start_ea, pfn.end_ea))
    dump_flags(pfn)
    if (ida_funcs.is_func_entry(pfn)):
        print ("This is an entry chunk")
        dump_tails(pfn)
        dump_frame(pfn)
        dump_regvars(pfn)
        dump_regargs(pfn)
        dump_stkpnts(pfn)
    elif (ida_funcs.is_func_tail(pfn)):
        print ("This is a tail chunk")
        dump_parents(pfn)


ea = ida_kernwin.get_screen_ea()
dump_func_info(ea)
```

IDAPython example: core/actions.py

custom actions, with icons & tooltips

description: How to create user actions, that once created can be inserted in menus, toolbars, context menus, …

Those actions, when triggered, will be passed a 'context' that contains some of the most frequently needed bits of information.

In addition, custom actions can determine when they want to be available (through their `ida_kernwin.action_handler_t.update` callback)

keywords: actions

see_also: add_hotkey

```
import ida_kernwin

class SayHi(ida_kernwin.action_handler_t):
    def __init__(self, message):
        ida_kernwin.action_handler_t.__init__(self)
        self.message = message
```

```
    def activate(self, ctx):
        print("Hi, %s" % (self.message))
        # print("context fields: %s" % dir(ctx))
        print(" cur_ea %08X" % ctx.cur_ea)
        print(" cur_value: %08X" % ctx.cur_value)
        print(" cur_extracted_ea %08X" % ctx.cur_extracted_ea)
        return 1

    # You can implement update(), to inform IDA when:
    #  * your action is enabled
    #  * update() should queried again
    # E.g., returning 'ida_kernwin.AST_ENABLE_FOR_WIDGET' will
    # tell IDA that this action is available while the
    # user is in the current widget, and that update()
    # must be queried again once the user gives focus
    # to another widget.
    #
    # For example, the following update() implementation
    # will let IDA know that the action is available in
    # "IDA View-*" views, and that it's not even worth
    # querying update() anymore until the user has moved
    # to another view..
    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_FOR_WIDGET if ctx.widget_type == ida_kernwin.BWN_DISAS


print("Creating a custom icon from raw data!")
# Stunned panda face icon data.
icon_data = b"".join([
        b"\x89\x50\x4E\x47\x0D\x0A\x1A\x0A\x00\x00\x00\x0D\x49\x48\x44\x52\x00\x00\x00\x10\x
        b"\x53\x6D\x48\x53\x6F\x14\x3F\xBA\xB5\xB7\xA0\x8D\x20\x41\xF2\xBA\x5D\xB6\x0F\x56\x
        b"\x3E\x4A\x50\x19\xE4\xB0\xD0\x22\xCD\x44\x45\x4A\x31\x8C\x92\xA2\x3E\x65\x0A\x4D\x
        b"\x7B\x39\xF7\xEE\x19\x17\xA8\xAC\x56\xDB\x54\x82\x60\x41\xB3\x59\xBC\xFF\xAC\xF9\x
        b"\x22\x02\xD0\x40\xE4\x81\x6C\x3B\x76\x37\x56\xE3\x37\x5F\x2F\x62\xE8\x0B\xD3\x66\x
        b"\x6F\xB0\x79\x82\x61\x81\x21\xCC\xDE\x21\x54\x16\x02\xD4\x69\x26\x9E\x74\xEE\xCB\x
        b"\x74\x66\x65\xE1\x98\x6F\x00\x31\x32\x87\x9F\x59\x77\x66\x66\x61\x42\xBC\xC0\xF5\x
        b"\x9A\x63\x68\xEA\x7C\x8A\xF6\x14\x3B\x9F\xF6\xA6\xA4\x60\xEB\xE3\x3E\x9C\x5F\xD6\x
        b"\x41\x83\x4E\x83\x54\xDB\x92\x76\x20\xCA\xBF\xD0\x99\x9D\xBB\x4E\xDB\xBD\xC7\x8E\x
        b"\x86\x2D\x71\x71\x00\x52\x10\x16\x17\xE6\xC1\xE7\x1B\x61\x9A\x81\x69\x31\x30\xFC\x
        b"\x32\x94\x95\x95\xC3\xA5\xD2\x53\x00\x51\x09\xAA\x4B\x0B\xA1\xB8\xA4\x0C\x52\x53\x
        b"\x18\x90\x22\x0A\x98\x8C\x46\xF0\x54\x14\x42\x6D\x7D\x3B\xE4\x1C\x75\x41\xAD\xB7\x
        b"\x82\xA6\xCD\x5B\x0D\xB2\x12\xE6\xE4\x06\xB5\x1A\x66\xA7\x26\x41\x92\xC2\xA0\xD5\x
        b"\xE8\x1A\xFF\xE4\x63\x8A\x0E\xE6\x02\x41\xF8\x3F\x18\x82\x40\x28\x04\xFD\xDD\x75\x
        b"\x25\x76\x50\x26\x76\x6B\x61\x86\x08\xE4\x1D\xAF\x81\xBC\x13\x97\xA9\xD3\x4C\x3C\x
        b"\xF6\xC1\xED\x52\xB8\x77\xAB\x98\x3A\xCD\xC4\x73\x9D\x7C\x6F\xDE\xF9\xCF\x53\x0E\x
```

```
        b"\xF1\xA4\x93\x0F\x00\x36\xAD\x3E\x4C\x6B\xC5\xC9\x5D\x77\x6A\x2F\xB4\x31\xA3\xC4\x
        b"\x2E\xF8\x0B\x2F\x3D\xE5\xC3\x97\x06\xCF\xCF\x00\x00\x00\x00\x49\x45\x4E\x44\xAE\x
act_icon = ida_kernwin.load_custom_icon(data=icon_data, format="png")

hooks = None
act_name = "example:add_action"

if ida_kernwin.register_action(ida_kernwin.action_desc_t(
        act_name,           # Name. Acts as an ID. Must be unique.
        "Say hi!",          # Label. That's what users see.
        SayHi("developer"), # Handler. Called when activated, and for updating
        "Ctrl+F12",         # Shortcut (optional)
        "Greets the user",  # Tooltip (optional)
        act_icon)):         # Icon ID (optional)
    print("Action registered. Attaching to menu.")

    # Insert the action in the menu
    if ida_kernwin.attach_action_to_menu("Edit/Export data", act_name, ida_kernwin.SETMENU_A
        print("Attached to menu.")
    else:
        print("Failed attaching to menu.")

    # Insert the action in a toolbar
    if ida_kernwin.attach_action_to_toolbar("AnalysisToolBar", act_name):
        print("Attached to toolbar.")
    else:
        print("Failed attaching to toolbar.")

    # We will also want our action to be available in the context menu
    # for the "IDA View-A" widget.
    #
    # To do that, we could in theory retrieve a reference to "IDA View-A", and
    # then request to "permanently" attach the action to it, using something
    # like this:
    #   ida_kernwin.attach_action_to_popup(ida_view_a, None, act_name, None)
    #
    # but alas, that won't do: widgets in IDA are very "volatile", and
    # can be deleted & re-created on some occasions (e.g., starting a
    # debugging session), and our efforts to permanently register our
    # action on "IDA View-A" would be annihilated as soon as "IDA View-A"
    # is deleted.
    #
    # Instead, we can opt for a different method: attach our action on-the-fly,
    # when the popup for "IDA View-A" is being populated, right before
    # it is displayed.
    class Hooks(ida_kernwin.UI_Hooks):
```

```
        def finish_populating_widget_popup(self, widget, popup):
            # We'll add our action to all "IDA View-*"s.
            # If we wanted to add it only to "IDA View-A", we could
            # also discriminate on the widget's title:
            #
            #  if ida_kernwin.get_widget_title(widget) == "IDA View-A":
            #      ...
            #
            if ida_kernwin.get_widget_type(widget) == ida_kernwin.BWN_DISASM:
                ida_kernwin.attach_action_to_popup(widget, popup, act_name, None)

    hooks = Hooks()
    hooks.hook()
else:
    print("Action found; unregistering.")
    # No need to call detach_action_from_menu(); it'll be
    # done automatically on destruction of the action.
    if ida_kernwin.unregister_action(act_name):
        print("Unregistered.")
    else:
        print("Failed to unregister action.")

    if hooks is not None:
        hooks.unhook()
        hooks = None
```

IDAPython example: core/add_hotkey.py

triggering bits of code by pressing a shortcut

description: `ida_kernwin.add_hotkey` is a simpler, but much less flexible alternative to `ida_kernwin.register_action` (though it does use the same mechanism under the hood.)

It's particularly useful during prototyping, but note that the actions that are created cannot be inserted in menus, toolbars or cannot provide a custom `ida_kernwin.action_handler_t.update` callback.

keywords: actions

see_also: actions

```
import ida_kernwin

def hotkey_pressed():
    print("hotkey pressed!")
```

```
try:
    hotkey_ctx
    if ida_kernwin.del_hotkey(hotkey_ctx):
        print("Hotkey unregistered!")
        del hotkey_ctx
    else:
        print("Failed to delete hotkey!")
except:
    hotkey_ctx = ida_kernwin.add_hotkey("Shift-A", hotkey_pressed)
    if hotkey_ctx is None:
        print("Failed to register hotkey!")
        del hotkey_ctx
    else:
        print("Hotkey registered!")
```

IDAPython example: core/add_idc_hotkey.py

triggering bits of code by pressing a shortcut (older version)

description: This is a somewhat ancient way of registering actions & binding shortcuts. It's still here for reference, but "fresher" alternatives should be preferred.

keywords: actions

see_also: actions, add_hotkey

```
import ida_expr
import ida_kernwin

def say_hi():
    print("Hotkey activated!")

# IDA binds hotkeys to IDC functions so a trampoline IDC function must be created
ida_expr.compile_idc_text('static key_2() { RunPythonStatement("say_hi()"); }')

# Add the hotkey
ida_kernwin.add_idc_hotkey("2", 'key_2')

# Press 2 to activate foo()

# The hotkey can be removed with
# ida_kernwin.del_idc_hotkey('2')
```

IDAPython example: core/auto_instantiate_widget_plugin.py

better integrating custom widgets in the desktop layout

description: This is an example demonstrating how one can create widgets from a plugin, and have them re-created automatically at IDA startup-time or at desktop load-time.

This example should be placed in the 'plugins' directory of the IDA installation, for it to work.

There are 2 ways to use this example: 1) reloading an IDB, where the widget was opened - open the widget ('View > Open subview > ...') - save this IDB, and close IDA - restart IDA with this IDB => the widget will be visible

2) reloading a desktop, where the widget was opened
   - open the widget ('View > Open subview > ...')
   - save the desktop ('Windows > Save desktop...') under, say, the name 'with_auto'
   - start another IDA instance with some IDB, and load that desktop => the widget will be visible

keywords: desktop

```
import ida_idaapi
import ida_kernwin

title = "Auto-instantiable at IDA startup"

# --------------------------------------------------------------------------
class auto_inst_t(ida_kernwin.simplecustviewer_t):
    def __init__(self):
        ida_kernwin.simplecustviewer_t.__init__(self)

    def Create(self):
        if not ida_kernwin.simplecustviewer_t.Create(self, title):
            return False

        text = __doc__
        for l in text.split("\n"):
            self.AddLine(l)
        return True

# --------------------------------------------------------------------------
auto_inst = None

# --------------------------------------------------------------------------
```

```
def register_open_action():
    """
    Provide the action that will create the widget
    when the user asks for it.
    """
    class create_widget_t(ida_kernwin.action_handler_t):
        def activate(self, ctx):
            if ida_kernwin.find_widget(title) is None:
                global auto_inst
                auto_inst = auto_inst_t()
                assert(auto_inst.Create())
                assert(auto_inst.Show())

        def update(self, ctx):
            return ida_kernwin.AST_ENABLE_ALWAYS

    action_name = "autoinst:create"
    ida_kernwin.register_action(
        ida_kernwin.action_desc_t(
            action_name,
            title,
            create_widget_t()))
    ida_kernwin.attach_action_to_menu(
        "View/Open subviews/Strings",
        action_name,
        ida_kernwin.SETMENU_APP)

# -----------------------------------------------------------------------
auto_inst_hooks = None
def register_autoinst_hooks():
    """
    Register hooks that will create the widget when IDA
    requires it because of the IDB/desktop
    """
    class auto_inst_hooks_t(ida_kernwin.UI_Hooks):
        def create_desktop_widget(self, ttl, cfg):
            if ttl == title:
                global auto_inst
                auto_inst = auto_inst_t()
                assert(auto_inst.Create())
                return auto_inst.GetWidget()

    global auto_inst_hooks
    auto_inst_hooks = auto_inst_hooks_t()
    auto_inst_hooks.hook()
```

```
# -----------------------------------------------------------------------
class auto_inst_plugin_t(ida_idaapi.plugin_t):
    flags = 0
    comment = "This plugin creates a widget that will be recreated automatically if needed,
    help = "No help, really"
    wanted_name = "autoinst"
    wanted_hotkey = ""

    def init(self):
        register_open_action()
        register_autoinst_hooks()

    def run(self, arg):
        pass

    def term(self):
        pass


def PLUGIN_ENTRY():
    return auto_inst_plugin_t()
```

IDAPython example: core/bin_search.py

showcasing `ida_bytes.bin_search`

description: IDAPython's ida_bytes.bin_search function is pretty powerful, but can be tough to figure out at first. This example introduces

- `ida_bytes.bin_search`, and
- `ida_bytes.parse_binpat_str`

in order to implement a simple replacement for the 'Search > Sequence of bytes...' dialog, that lets users search for sequences of bytes that compose string literals in the binary file (either in the default 1-byte-per-char encoding, or as UTF-16.)

```
import ida_kernwin
import ida_bytes
import ida_ida
import ida_idaapi
import ida_nalt

class search_strlit_form_t(ida_kernwin.Form):
    def __init__(self):
        ida_kernwin.Form.__init__(
            self,
```

```
            r"""Please enter string literal

<Text: {Text}>
<#UTF16-BE if file is big-endian, UTF16-LE otherwise#As UTF-16: {UTF16}>{Encoding}>
""",
            {
                "Text" : ida_kernwin.Form.StringInput(),
                "Encoding" : ida_kernwin.Form.ChkGroupControl(("UTF16",)),
            })

class search_strlit_ah_t(ida_kernwin.action_handler_t):
    def __init__(self):
        ida_kernwin.action_handler_t.__init__(self)

    def activate(self, ctx):
        f = search_strlit_form_t()
        f, args = f.Compile()
        ok = f.Execute()
        if ok:
            current_ea = ida_kernwin.get_screen_ea()
            patterns = ida_bytes.compiled_binpat_vec_t()
            encoding = ida_nalt.get_default_encoding_idx(
                ida_nalt.BPU_2B if f.Encoding.value else ida_nalt.BPU_1B)
            # string literals must be quoted. That's how parse_binpat_str
            # recognizes them (we want to be careful though: the user
            # might type in something like 'L"hello"', which should
            # decode to the IDB-specific wide-char set of bytes)
            text = f.Text.value
            if text.find('"') < 0:
                text = '"%s"' % text
            err = ida_bytes.parse_binpat_str(
                patterns,
                current_ea,
                text,
                10, # radix (not that it matters though, since we're all about string litera
                encoding)
            if not err:
                ea = ida_bytes.bin_search(
                    current_ea,
                    ida_ida.inf_get_max_ea(),
                    patterns,
                    ida_bytes.BIN_SEARCH_FORWARD
                  | ida_bytes.BIN_SEARCH_NOBREAK
                  | ida_bytes.BIN_SEARCH_NOSHOW)
                ok = ea != ida_idaapi.BADADDR
                if ok:
```

```
                    ida_kernwin.jumpto(ea)
                else:
                    print("Failed parsing binary pattern: \"%s\"" % err)
            return ok

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_FOR_WIDGET \
            if ctx.widget_type == ida_kernwin.BWN_DISASM \
            else ida_kernwin.AST_DISABLE_FOR_WIDGET


ACTION_NAME = "bin_search:search"
ACTION_SHORTCUT = "Ctrl+Shift+S"

if ida_kernwin.register_action(
        ida_kernwin.action_desc_t(
            ACTION_NAME,
            "Search for string literal",
            search_strlit_ah_t(),
            ACTION_SHORTCUT)):
    print("Please use \"%s\" to search for string literals" % ACTION_SHORTCUT)
```

IDAPython example: core/colorize_disassembly.py

change background colours

description: This illustrates the setting/retrieval of background colours using the IDC wrappers

In order to do so, we'll be assigning colors to specific ranges (item, function, or segment). Those will be persisted in the database.

category: disassembly

keywords: coloring, idc

see_also: colorize_disassembly_on_the_fly

```
BG_BLUE  = 0xc02020
BG_GREEN = 0x208020
BG_RED   = 0x2020c0

import idc

ea = idc.here()
idc.set_color(ea, idc.CIC_SEGM, BG_BLUE)
idc.set_color(ea, idc.CIC_FUNC, BG_GREEN)
```

```
idc.set_color(ea, idc.CIC_ITEM, BG_RED)
print("Segment:  %x" % idc.get_color(ea, idc.CIC_SEGM))
print("Function: %x" % idc.get_color(ea, idc.CIC_FUNC))
print("Item:     %x" % idc.get_color(ea, idc.CIC_ITEM))
```

IDAPython example: core/colorize_disassembly_on_the_fly.py

an easy-to-use way to colorize lines

description: This builds upon the `ida_kernwin.UI_Hooks.get_lines_rendering_info` feature, to provide a quick & easy way to colorize disassembly lines.

Contrary to @colorize_disassembly, the coloring is not persisted in the database, and will therefore be lost after the session.

By triggering the action multiple times, the user can "carousel" across 4 predefined colors (and return to the "no color" state.)

keywords: coloring

see_also: colorize_disassembly

```
import ida_kernwin
import ida_moves

class on_the_fly_coloring_hooks_t(ida_kernwin.UI_Hooks):

    # We'll offer the users the ability to carousel around the
    # following colors. Well, note that these are in fact not
    # colors, but rather color "keys": each theme might have its
    # own values for those.
    AVAILABLE_COLORS = [
        ida_kernwin.CK_EXTRA5,
        ida_kernwin.CK_EXTRA6,
        ida_kernwin.CK_EXTRA7,
        ida_kernwin.CK_EXTRA8,
    ]

    def __init__(self):
        ida_kernwin.UI_Hooks.__init__(self)

        # Each view can have on-the-fly coloring.
        # We'll store the custom colors keyed on the widget's title
        self.by_widget = {}

    def get_lines_rendering_info(self, out, widget, rin):
        """
```

```
        Called by IDA, at rendering-time.

        We'll look in our set of marked lines, and for those that are
        found, will produce additional rendering information for IDA
        to use.
        """
        title = ida_kernwin.get_widget_title(widget)
        assigned = self.by_widget.get(title, None)
        if assigned is not None:
            for section_lines in rin.sections_lines:
                for line in section_lines:
                    for loc, color in assigned:
                        if self._same_lines(widget, line.at, loc.place()):
                            e = ida_kernwin.line_rendering_output_entry_t(line)
                            e.bg_color = color
                            out.entries.push_back(e)

    def _same_lines(self, viewer, p0, p1):
        return ida_kernwin.get_custom_viewer_place_xcoord(viewer, p0, p1) != -1

    def _find_loc_index(self, viewer, assigned, loc):
        for idx, tpl in enumerate(assigned):
            _loc = tpl[0]
            if self._same_lines(viewer, loc.place(), _loc.place()):
                return idx
        return -1

    def carousel_color(self, viewer, title):
        """
        This performs the work of iterating across the available
        colors (and the 'no-color' state.)
        """
        loc = ida_moves.lochist_entry_t()
        if ida_kernwin.get_custom_viewer_location(loc, viewer):
            assigned = self.by_widget.get(title, [])
            new_color = None

            idx = self._find_loc_index(viewer, assigned, loc)
            if idx > -1:
                prev_color = assigned[idx][1]
                prev_color_idx = self.AVAILABLE_COLORS.index(prev_color)
                new_color = None \
                            if prev_color_idx >= (len(self.AVAILABLE_COLORS) - 1) \
                            else self.AVAILABLE_COLORS[prev_color_idx + 1]
            else:
                new_color = self.AVAILABLE_COLORS[0]
```

```
            if idx > -1:
                del assigned[idx]
            if new_color is not None:
                assigned.append((loc, new_color))

            if assigned:
                self.by_widget[title] = assigned
            else:
                if title in self.by_widget:
                    del self.by_widget[title]


class carousel_color_ah_t(ida_kernwin.action_handler_t):
    """
    The action that will be invoked by IDA when the user
    activates its shortcut.
    """
    def __init__(self, hooks):
        ida_kernwin.action_handler_t.__init__(self)
        self.hooks = hooks

    def activate(self, ctx):
        v = ida_kernwin.get_current_viewer()
        if v:
            self.hooks.carousel_color(v, ctx.widget_title)
            return 1 # will cause the widget to redraw

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_FOR_WIDGET \
            if ida_kernwin.get_current_viewer() \
            else ida_kernwin.AST_DISABLE_FOR_WIDGET


ACTION_NAME = "example:colorize_disassembly_on_the_fly"
ACTION_LABEL = "Pick line color"
ACTION_SHORTCUT = "!"
ACTION_HELP = "Press %s to carousel around available colors (or remove a previously-set colo

otf_coloring = on_the_fly_coloring_hooks_t()
if ida_kernwin.register_action(ida_kernwin.action_desc_t(
        ACTION_NAME,
        ACTION_LABEL,
        carousel_color_ah_t(otf_coloring),
        ACTION_SHORTCUT)):
    print("Registered action \"%s\". %s" % (ACTION_LABEL, ACTION_HELP))
```

```
    otf_coloring.hook()
```

IDAPython example: core/create_structure_programmatically.py

programmatically create & populate a structure

description: Usage of the API to create & populate a structure with members of different types.

author: Gergely Erdelyi (gergely.erdelyi@d-dome.net)

```
#---------------------------------------------------------------------
# Structure test
#
# This script demonstrates how to create structures and populate them
# with members of different types.
#---------------------------------------------------------------------

import ida_struct
import ida_idaapi
import ida_bytes
import ida_nalt

import idc

sid = ida_struct.get_struc_id("mystr1")
if sid != -1:
    idc.del_struc(sid)
sid = ida_struct.add_struc(ida_idaapi.BADADDR, "mystr1", 0)
print("%x" % sid)

# Test simple data types
simple_types_data = [
    (ida_bytes.FF_BYTE, 1),
    (ida_bytes.FF_WORD, 2),
    (ida_bytes.FF_DWORD, 4),
    (ida_bytes.FF_QWORD, 8),
    (ida_bytes.FF_TBYTE, 10),
    (ida_bytes.FF_OWORD, 16),
    (ida_bytes.FF_FLOAT, 4),
    (ida_bytes.FF_DOUBLE, 8),
    (ida_bytes.FF_PACKREAL, 10),
]
for i, tpl in enumerate(simple_types_data):
    t, nsize = tpl
```

```
    print("t%x:"% ((t|ida_bytes.FF_DATA) & 0xFFFFFFFF),
          idc.add_struc_member(sid, "t%02d"%i, ida_idaapi.BADADDR, (t|ida_bytes.FF_DATA )&0x

# Test ASCII type
print("ASCII:", idc.add_struc_member(sid, "tascii", -1, ida_bytes.FF_STRLIT|ida_bytes.FF_DAT

# Test struc member type
msid = ida_struct.get_struc_id("mystr2")
if msid != -1:
    idc.del_struc(msid)
msid = idc.add_struc(-1, "mystr2", 0)
print(idc.add_struc_member(msid, "member1", -1, (ida_bytes.FF_DWORD|ida_bytes.FF_DATA )&0xFF
print(idc.add_struc_member(msid, "member2", -1, (ida_bytes.FF_DWORD|ida_bytes.FF_DATA )&0xFF

msize = ida_struct.get_struc_size(msid)
print("Struct:", idc.add_struc_member(sid, "tstruct", -1, ida_bytes.FF_STRUCT|ida_bytes.FF_D
print("Stroff:", idc.add_struc_member(sid, "tstroff", -1, ida_bytes.stroff_flag()|ida_bytes.

# Test offset types
print("Offset:", idc.add_struc_member(sid, "toffset", -1, ida_bytes.off_flag()|ida_bytes.FF_
print("Offset:", idc.set_member_type(sid, 0, ida_bytes.off_flag()|ida_bytes.FF_DATA|ida_byte

print("Done")
```

IDAPython example: core/custom_cli.py

a custom command-line interpreter

description: Illustrates how one can add command-line interpreters to IDA

This custom interpreter doesn't actually run any code; it's there as a 'getting started'. It provides an example tab completion support.

```
# --------------------------------------------------------------------------
# This is an example illustrating how to implement a CLI
#
# A trivial example is also provided for tab completion. To try it,
# type "bon" in the input field, and then press <Tab> multiple times.
#
# (c) Hex-Rays

import ida_kernwin
import ida_idaapi

class mycli_t(ida_kernwin.cli_t):
```

```
        flags = 0
        sname = "pycli"
        lname = "Python CLI"
        hint  = "pycli hint"

        def OnExecuteLine(self, line):
            print("OnExecute:", line)
            return True

        def OnKeydown(self, line, x, sellen, vkey, shift):
            print("Onkeydown: line=%s x=%d sellen=%d vkey=%d shift=%d" % (line, x, sellen, vkey,
            return None

        completions = [
            "bonnie & clyde",
            "bonfire of the vanities",
            "bongiorno",
        ]

        def OnCompleteLine(self, prefix, n, line, prefix_start):
            print("OnCompleteLine: prefix=%s n=%d line=%s prefix_start=%d" % (prefix, n, line, p
            if prefix == "bon":
                if n < len(self.completions):
                    return self.completions[n]
            return None

    # --------------------------------------------------------------------------
    def nw_handler(code, old=0):
        if code == ida_idaapi.NW_OPENIDB:
            print("nw_handler(): installing CLI")
            mycli.register()
        elif code == ida_idaapi.NW_CLOSEIDB:
            print("nw_handler(): removing CLI")
            mycli.unregister()
        elif code == ida_idaapi.NW_TERMIDA:
            print("nw_handler(): uninstalled nw handler")
            when = ida_idaapi.NW_TERMIDA | ida_idaapi.NW_OPENIDB | ida_idaapi.NW_CLOSEIDB | ida_
            ida_idaapi.notify_when(when, nw_handler)

    # --------------------------------------------------------------------------

    # Already installed?
    try:
        mycli
        # remove previous CLI
        mycli.unregister()
```

```
    del mycli
    # remove previous handler
    nw_handler(ida_idaapi.NW_TERMIDA)
except:
    pass
finally:
    mycli = mycli_t()

# register CLI
if mycli.register():
    print("CLI installed")
    # install new handler
    when = ida_idaapi.NW_TERMIDA | ida_idaapi.NW_OPENIDB | ida_idaapi.NW_CLOSEIDB
    ida_idaapi.notify_when(when, nw_handler)
else:
    del mycli
    print("Failed to install CLI")
```

IDAPython example: core/custom_data_types_and_formats.py

using custom data types & printers

description: IDA can be extended to support certain data types that it does not know about out-of-the-box.

A 'custom data type' provide information about the type & size of a piece of data, while a 'custom data format' is in charge of formatting that data (there can be more than one format for a specific 'custom data type'.)

```
import ida_bytes
import ida_idaapi
import ida_lines
import ida_struct
import ida_netnode
import ida_nalt

import sys
import struct
import ctypes
import platform

# --------------------------------------------------------------------------
class pascal_data_type(ida_bytes.data_type_t):
    def __init__(self):
```

```
            ida_bytes.data_type_t.__init__(
                self,
                "py_pascal_string",
                2,
                "Pascal string",
                None,
                "pstr")

        def calc_item_size(self, ea, maxsize):
            # Custom data types may be used in structure definitions. If this case
            # ea is a member id. Check for this situation and return 1
            if ida_struct.is_member_id(ea):
                return 1

            # get the length byte
            n = ida_bytes.get_byte(ea)

            # string too big?
            if n > maxsize:
                return 0
            # ok, accept the string
            return n + 1

    class pascal_data_format(ida_bytes.data_format_t):
        FORMAT_NAME = "py_pascal_string_pstr"
        def __init__(self):
            ida_bytes.data_format_t.__init__(
                self,
                pascal_data_format.FORMAT_NAME)

        def printf(self, value, current_ea, operand_num, dtid):
            # Take the length byte
            n = ord(value[0]) if sys.version_info.major < 3 else value[0]
            o = ['"']
            for ch in value[1:]:
                b = ord(ch) if sys.version_info.major < 3 else ch
                if b < 0x20 or b > 128:
                    o.append(r'\x%02x' % b)
                else:
                    o.append(ch)
            o.append('"')
            return "".join(o)

    # --------------------------------------------------------------------------
    class simplevm_data_type(ida_bytes.data_type_t):
        ASM_KEYWORD = "svm_emit"
```

```
    def __init__(
            self,
            name="py_simple_vm",
            value_size=1,
            menu_name="SimpleVM",
            asm_keyword=ASM_KEYWORD):
        ida_bytes.data_type_t.__init__(
            self,
            name,
            value_size,
            menu_name,
            None,
            asm_keyword)

    def calc_item_size(self, ea, maxsize):
        if ida_struct.is_member_id(ea):
            return 1
        # get the opcode and see if it has an imm
        n = 5 if (ida_bytes.get_byte(ea) & 3) == 0 else 1
        # string too big?
        if n > maxsize:
            return 0
        # ok, accept
        return n

class simplevm_data_format(ida_bytes.data_format_t):
    def __init__(
            self,
            name="py_simple_vm_format",
            menu_name="SimpleVM"):
        ida_bytes.data_format_t.__init__(
            self,
            name,
            0,
            menu_name)

    # Some tables for the disassembler
    INST = {1: 'add', 2: 'mul', 3: 'sub', 4: 'xor', 5: 'mov'}
    REGS = {1: 'r1', 2: 'r2', 3: 'r3'}
    def disasm(self, inst):
        """A simple local disassembler. In reality one can use a full-blown disassembler to
        opbyte = ord(inst[0]) if sys.version_info.major < 3 else inst[0]
        op     = opbyte >> 4
        if not (1<=op<=5):
            return None
        r1     = (opbyte & 0xf) >> 2
```

```
        r2     = opbyte & 3
        sz     = 0
        if r2 == 0:
            if len(inst) != 5:
                return None
            imm = struct.unpack_from('L', inst, 1)[0]
            sz  = 5
        else:
            imm = None
            sz  = 1
        text = "%s %s, %s" % (
            ida_lines.COLSTR(simplevm_data_format.INST[op], ida_lines.SCOLOR_INSN),
            ida_lines.COLSTR(simplevm_data_format.REGS[r1], ida_lines.SCOLOR_REG),
            ida_lines.COLSTR("0x%08X" % imm, ida_lines.SCOLOR_NUMBER) if imm is not None els
        return (sz, text)

    def printf(self, value, current_ea, operand_num, dtid):
        r = self.disasm(value)
        if not r:
            return None
        if dtid == 0:
            return "%s(%s)" % (simplevm_data_type.ASM_KEYWORD, r[1])
        return r[1]

# --------------------------------------------------------------------------
# This format will display DWORD values as MAKE_DWORD(0xHI, 0xLO)
class makedword_data_format(ida_bytes.data_format_t):
    def __init__(self):
        ida_bytes.data_format_t.__init__(
            self,
            "py_makedword",
            4,
            "Make DWORD")

    def printf(self, value, current_ea, operand_num, dtid):
        if len(value) != 4: return None
        w1 = struct.unpack_from("H", value, 0)[0]
        w2 = struct.unpack_from("H", value, 2)[0]
        return "MAKE_DWORD(0x%04X, 0x%04X)" % (w2, w1)

# --------------------------------------------------------------------------
# This format will try to load a resource string given a number
# So instead of displaying:
#    push 66h
#    call message_box_from_rsrc_string
# It can be rendered as;
```

```
#    push RSRC("The message")
#    call message_box_from_rsrc_string
#
# The get_rsrc_string() is not optimal since it loads/unloads the
# DLL each time for a new string. It can be improved in many ways.
class rsrc_string_format(ida_bytes.data_format_t):
    def __init__(self):
        ida_bytes.data_format_t.__init__(
            self,
            "py_w32rsrcstring",
            1,
            "Resource string")
        self.cache_node = ida_netnode.netnode("$ py_w32rsrcstring", 0, 1)

    def get_rsrc_string(self, fn, id):
        """
        Simple method that loads the input file as a DLL with LOAD_LIBRARY_AS_DATAFILE flag
        It then tries to LoadString()
        """
        k32 = ctypes.windll.kernel32
        u32 = ctypes.windll.user32

        hinst = k32.LoadLibraryExA(fn, 0, 0x2)
        if hinst == 0:
            return ""
        buf = ctypes.create_string_buffer(1024)
        r   = u32.LoadStringA(hinst, id, buf, 1024-1)
        k32.FreeLibrary(hinst)
        return buf.value if r else ""

    def printf(self, value, current_ea, operand_num, dtid):
        # Is it already cached?
        val = self.cache_node.supval(current_ea)

        # Not cached?
        if val == None:
            # Retrieve it
            num = ida_idaapi.struct_unpack(value)
            val = self.get_rsrc_string(ida_nalt.get_input_file_path(), num)
            # Cache it
            self.cache_node.supset(current_ea, val)

        # Failed to retrieve?
        if val == "" or val == "\x00":
            return None
        # Return the format
```

```
        return "RSRC_STR(\"%s\")" % ida_lines.COLSTR(val, ida_lines.SCOLOR_IMPNAME)

# --------------------------------------------------------------------------
# Table of formats and types to be registered/unregistered
# If a tuple has one element then it is the format to be registered with dtid=0
# If the tuple has more than one element, the tuple[0] is the data type and tuple[1:] are th
new_formats = [
  (pascal_data_type(), pascal_data_format()),
  (simplevm_data_type(), simplevm_data_format()),
  (makedword_data_format(),),
  (simplevm_data_format(),),
]

try:
    if platform.system() == 'Windows':
        new_formats.append((rsrc_string_format(),))
except:
    pass

# --------------------------------------------------------------------------
def nw_handler(code, old=0):
    # delete notifications
    if code == ida_idaapi.NW_OPENIDB:
        if not ida_bytes.register_data_types_and_formats(new_formats):
            print("Failed to register types!")
    elif code == ida_idaapi.NW_CLOSEIDB:
        ida_bytes.unregister_data_types_and_formats(new_formats)
    elif code == ida_idaapi.NW_TERMIDA:
        f = ida_idaapi.NW_TERMIDA | ida_idaapi.NW_OPENIDB | ida_idaapi.NW_CLOSEIDB | ida_ida
        ida_idaapi.notify_when(f, nw_handler)

# --------------------------------------------------------------------------
# Check if already installed
if ida_bytes.find_custom_data_type(pascal_data_format.FORMAT_NAME) == -1:
    if not ida_bytes.register_data_types_and_formats(new_formats):
        print("Failed to register types!")
    else:
        f = ida_idaapi.NW_TERMIDA | ida_idaapi.NW_OPENIDB | ida_idaapi.NW_CLOSEIDB
        ida_idaapi.notify_when(f, nw_handler)
        print("Formats installed!")
else:
    print("Formats already installed!")
```

IDAPython example: core/dump_extra_comments.py

retrieve extra comments

description: Use the `ida_lines.get_extra_cmt` API to retrieve anterior and posterior extra comments.

This script registers two actions, that can be used to dump the previous and next extra comments.

```
import ida_lines
import ida_kernwin

# -----------------------------------------------------------------------
class dump_at_point_handler_t(ida_kernwin.action_handler_t):
    def __init__(self, anchor):
        ida_kernwin.action_handler_t.__init__(self)
        self.anchor = anchor

    def activate(self, ctx):
        ea = ida_kernwin.get_screen_ea()
        index = self.anchor
        while True:
            cmt = ida_lines.get_extra_cmt(ea, index)
            if cmt is None:
                break
            print("Got: '%s'" % cmt)
            index += 1

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_FOR_WIDGET \
            if ctx.widget_type == ida_kernwin.BWN_DISASM \
            else ida_kernwin.AST_DISABLE_FOR_WIDGET

    @staticmethod
    def compose_action_name(v):
        return "dump_extra_comments:%s" % v


# -----------------------------------------------------------
# action variants

class action_previous_handler_t(dump_at_point_handler_t):
    ACTION_LABEL    = "previous"
    ACTION_SHORTCUT = "Ctrl+Shift+Y"

    def __init__(self):
        super(action_previous_handler_t, self).__init__(ida_lines.E_PREV)
```

```
class action_next_handler_t(dump_at_point_handler_t):
    ACTION_LABEL    = "next"
    ACTION_SHORTCUT = "Ctrl+Shift+Z"

    def __init__(self):
        super(action_next_handler_t, self).__init__(ida_lines.E_NEXT)


# -----------------------------------------------------------------------
# create actions (and attach them to IDA View-A's context menu if possible)
widget_title = "IDA View-A"
ida_view = ida_kernwin.find_widget(widget_title)

action_variants = [
    action_previous_handler_t,
    action_next_handler_t,
]
for variant in action_variants:
    actname = dump_at_point_handler_t.compose_action_name(variant.ACTION_LABEL)
    if ida_kernwin.unregister_action(actname):
        print("Unregistered previously-registered action \"%s\"" % actname)

    desc = ida_kernwin.action_desc_t(
        actname,
        "Dump %s extra comments" % variant.ACTION_LABEL,
        variant(),
        variant.ACTION_SHORTCUT)
    if ida_kernwin.register_action(desc):
        print("Registered action \"%s\"" % actname)

    if ida_view and ida_kernwin.attach_action_to_popup(ida_view, None, actname):
        print("Permanently attached action \"%s\" to \"%s\"" % (actname, widget_title))
```

IDAPython example: core/dump_flowchart.py

dump function flowchart

description: Dumps the current function's flowchart, using 2 methods:

```
* the low-level `ida_gdl.qflow_chart_t` type
* the somewhat higher-level, and slightly more pythonic
  `ida_gdl.FlowChart` type.
```

```
# -*- coding: utf-8 -*-


import ida_gdl
```

```
import ida_funcs
import ida_kernwin

def out(p, msg):
    if p:
        print(msg)

def out_succ(p, start_ea, end_ea):
    out(p, "  SUCC:  %x - %x" % (start_ea, end_ea))

def out_pred(p, start_ea, end_ea):
    out(p, "  PRED:  %x - %x" % (start_ea, end_ea))


# -----------------------------------------------------------------------
# Using ida_gdl.qflow_chart_t
def using_qflow_chart_t(ea, p=True):
    f = ida_funcs.get_func(ea)
    if not f:
        return

    q = ida_gdl.qflow_chart_t("The title", f, 0, 0, 0)
    for n in range(q.size()):
        b = q[n]
        out(p, "%x - %x [%d]:" % (b.start_ea, b.end_ea, n))
        for ns in range(q.nsucc(n)):
            b2 = q[q.succ(n, ns)]
            out_succ(p, b2.start_ea, b2.end_ea)

        for ns in range(q.npred(n)):
            b2 = q[q.pred(n, ns)]
            out_pred(p, b2.start_ea, b2.end_ea)

# -----------------------------------------------------------------------
# Using ida_gdl.FlowChart
def using_FlowChart(ea, p=True):
    f = ida_gdl.FlowChart(ida_funcs.get_func(ea))

    for block in f:
        out(p, "%x - %x [%d]:" % (block.start_ea, block.end_ea, block.id))
        for succ_block in block.succs():
            out_succ(p, succ_block.start_ea, succ_block.end_ea)

        for pred_block in block.preds():
            out_pred(p, pred_block.start_ea, pred_block.end_ea)
```

```
ea = ida_kernwin.get_screen_ea()

print(">>> Dumping flow chart using ida_gdl.qflow_chart_t")
using_qflow_chart_t(ea)

print(">>> Dumping flow chart using the higher-level ida_gdl.FlowChart")
using_FlowChart(ea)
```

IDAPython example: core/dump_selection.py

retrieve & dump current selection

description: Shows how to retrieve the selection from a listing widget ("IDA View-A", "Hex View-1", "Pseudocode-A", ...) as two "cursors", and from there retrieve (in fact, generate) the corresponding text.

After running this script:

```
* select some text in one of the listing widgets (i.e.,
  "IDA View-*", "Enums", "Structures", "Pseudocode-*")
* press Ctrl+Shift+S to dump the selection
```

```
import ida_kernwin
import ida_lines

class dump_selection_handler_t(ida_kernwin.action_handler_t):
    def activate(self, ctx):
        if ctx.has_flag(ida_kernwin.ACF_HAS_SELECTION):
            tp0, tp1 = ctx.cur_sel._from, ctx.cur_sel.to
            ud = ida_kernwin.get_viewer_user_data(ctx.widget)
            lnar = ida_kernwin.linearray_t(ud)
            lnar.set_place(tp0.at)
            lines = []
            while True:
                cur_place = lnar.get_place()
                first_line_ref = ida_kernwin.l_compare2(cur_place, tp0.at, ud)
                last_line_ref = ida_kernwin.l_compare2(cur_place, tp1.at, ud)
                if last_line_ref > 0: # beyond last line
                    break
                line = ida_lines.tag_remove(lnar.down())
                if last_line_ref == 0: # at last line
                    line = line[0:tp1.x]
                elif first_line_ref == 0: # at first line
                    line = ' ' * tp0.x + line[tp0.x:]
                lines.append(line)
```

```
            for line in lines:
                print(line)
        return 1

    def update(self, ctx):
        ok_widgets = [
            ida_kernwin.BWN_DISASM,
            ida_kernwin.BWN_STRUCTS,
            ida_kernwin.BWN_ENUMS,
            ida_kernwin.BWN_PSEUDOCODE,
        ]
        return ida_kernwin.AST_ENABLE_FOR_WIDGET \
            if ctx.widget_type in ok_widgets \
            else ida_kernwin.AST_DISABLE_FOR_WIDGET


# --------------------------------------------------------------------------
# create actions (and attach them to IDA View-A's context menu if possible)
ACTION_NAME = "dump_selection"
ACTION_SHORTCUT = "Ctrl+Shift+S"

if ida_kernwin.unregister_action(ACTION_NAME):
    print("Unregistered previously-registered action \"%s\"" % ACTION_NAME)

if ida_kernwin.register_action(
        ida_kernwin.action_desc_t(
            ACTION_NAME,
            "Dump selection",
            dump_selection_handler_t(),
            ACTION_SHORTCUT)):
    print("Registered action \"%s\"" % ACTION_NAME)
```

IDAPython example: core/extend_idc.py

add functions to the IDC runtime from IDAPython

description: You can add IDC functions to IDA, whose "body" consists of IDAPython statements!

We'll register a 'pow' function, available to all IDC code, that when invoked will call back into IDAPython, and execute the provided function body.

After running this script, try switching to the IDC interpreter (using the button on the lower-left corner of IDA) and executing `pow(3, 7)`

```
import ida_expr
```

```
if ida_expr.add_idc_func(
        "pow",
        lambda n, e: n ** e,
        (ida_expr.VT_LONG, ida_expr.VT_LONG)):
    print("The pow() function is now available in IDC")
else:
    print("Failed to register pow() IDC function")
```

IDAPython example: core/idapythonrc.py

code to be run right after IDAPython initialization

description: The `idapythonrc.py` file:

```
* %APPDATA%\Hex-Rays\IDA Pro\idapythonrc.py (on Windows)
* ~/.idapro/idapythonrc.py (on Linux & Mac)
```

can contain any IDAPython code that will be run as soon as IDAPython is done successfully initializing.

```
# Add your favourite script to ScriptBox for easy access
# scriptbox.addscript("/here/is/my/favourite/script.py")

# Uncomment if you want to set Python as default interpreter in IDA
# import ida_idaapi
# ida_idaapi.enable_extlang_python(True)

# Disable the Python from interactive command-line
# import ida_idaapi
# ida_idaapi.enable_python_cli(False)

# Set the timeout for the script execution cancel dialog
# import ida_idaapi
# ida_idaapi.set_script_timeout(10)
```

IDAPython example: core/install_user_defined_prefix.py

inserting information into disassembly prefixes

description: By default, disassembly line prefixes contain segment + address information (e.g., '.text:08047718'), but it is possible to "inject" other bits of information in there, thanks to the `ida_lines.user_defined_prefix_t` helper type.

```
import ida_lines
import ida_idaapi

PREFIX = ida_lines.SCOLOR_INV + ' ' + ida_lines.SCOLOR_INV

class my_user_prefix_t(ida_lines.user_defined_prefix_t):
    def get_user_defined_prefix(self, ea, insn, lnnum, indent, line):
        if (ea % 2 == 0) and indent == -1:
            return PREFIX
        else:
            return ""


class prefix_plugin_t(ida_idaapi.plugin_t):
    flags = 0
    comment = "This is a user defined prefix sample plugin"
    help = "This is help"
    wanted_name = "user defined prefix"
    wanted_hotkey = ""

    def __init__(self):
        self.prefix = None

    def init(self):
        self.prefix = my_user_prefix_t(8)
        print("prefix installed")
        return ida_idaapi.PLUGIN_KEEP

    def run(self, arg):
        pass

    def term(self):
        self.prefix = None
        print("prefix uninstalled!")


def PLUGIN_ENTRY():
    return prefix_plugin_t()
```

IDAPython example: core/list_bookmarks.py

list bookmarks associated to a listing

description: This sample shows how to programmatically access the list of bookmarks placed in a listing widget (e.g., “IDA View-A”, “Pseudocode-”, â€¦) using

the low-level `ida_moves.bookmarks_t` type.

keywords: bookmarks

```
import ida_kernwin
import ida_moves

class list_bookmarks_ah_t(ida_kernwin.action_handler_t):
    def activate(self, ctx):
        v = ida_kernwin.get_current_viewer()
        if v:
            print("### Bookmarks for %s" % ida_kernwin.get_widget_title(v))
            ud = ida_kernwin.get_viewer_user_data(v)
            for loc, desc in ida_moves.bookmarks_t(v):
                print("\t'%s': %s" % (loc.place()._print(ud), desc))

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_FOR_WIDGET \
            if ida_kernwin.get_current_viewer() \
            else ida_kernwin.AST_DISABLE_FOR_WIDGET

ACTION_NAME = "example:list_bookmarks"
ACTION_LABEL = "List bookmarks"
ACTION_SHORTCUT = "Ctrl+!"
ACTION_HELP = "Press %s to list bookmarks" % ACTION_SHORTCUT

if ida_kernwin.register_action(ida_kernwin.action_desc_t(
        ACTION_NAME,
        ACTION_LABEL,
        list_bookmarks_ah_t(),
        ACTION_SHORTCUT)):
    print("Registered action \"%s\". %s" % (ACTION_LABEL, ACTION_HELP))
```

IDAPython example: core/list_function_items.py

showcases (a few of) the iterators available on a function

description: This demonstrates how to use some of the iterators available on the func_t type.

This example will focus on:

```
* `func_t[.__iter__]`: the default iterator; iterates on instructions
* `func_t.data_items`: iterate on data items contained within a function
* `func_t.head_items`: iterate on 'heads' (i.e., addresses containing
                       the start of an instruction, or a data item.
```

```
* `func_t.addresses`: iterate on all addresses within function (code
                      and data, beginning of an item or not)
```

Type `help(ida_funcs.func_t)` for a full list of iterators.

In addition, one can use:

```
* `func_tail_iterator_t`: iterate on all the chunks (including
                          the main one) of the function
* `func_parent_iterator_t`: iterate on all the parent functions,
                            that include this chunk
```

keywords: funcs iterator

```
import ida_bytes
import ida_kernwin
import ida_funcs
import ida_ua

class logger_t(object):

    class section_t(object):
        def __init__(self, logger, header):
            self.logger = logger
            self.logger.log(header)
        def __enter__(self):
            self.logger.indent += 2
            return self
        def __exit__(self, tp, value, traceback):
            self.logger.indent -= 2
            if value:
                return False # Re-raise

    def __init__(self):
        self.indent = 0

    def log(self, *args):
        print("  " * self.indent + "".join(args))

    def log_ea(self, ea):
        F = ida_bytes.get_flags(ea)
        parts = ["0x%08x" % ea, ": "]
        if ida_bytes.is_code(F):
            parts.append("instruction (%s)" % ida_ua.print_insn_mnem(ea))
        if ida_bytes.is_data(F):
            parts.append("data")
```

```
        if ida_bytes.is_tail(F):
            parts.append("tail")
        if ida_bytes.is_unknown(F):
            parts.append("unknown")
        if ida_funcs.get_func(ea) != ida_funcs.get_fchunk(ea):
            parts.append(" (in function chunk)")
        self.log(*parts)

def main():
    # Get current ea
    ea = ida_kernwin.get_screen_ea()

    pfn = ida_funcs.get_func(ea)

    if pfn is None:
        print("No function defined at 0x%x" % ea)
        return

    func_name = ida_funcs.get_func_name(pfn.start_ea)
    logger = logger_t()
    logger.log("Function %s at 0x%x" % (func_name, ea))

    with logger_t.section_t(logger, "Code items:"):
        for item in pfn:
            logger.log_ea(item)

    with logger_t.section_t(logger, "'head' items:"):
        for item in pfn.head_items():
            logger.log_ea(item)

    with logger_t.section_t(logger, "Addresses:"):
        for item in pfn.addresses():
            logger.log_ea(item)

    with logger_t.section_t(logger, "Function chunks:"):
        for chunk in ida_funcs.func_tail_iterator_t(pfn):
            logger.log("%s chunk: 0x%08x..0x%08x" % (
                "Main" if chunk.start_ea == pfn.start_ea else "Tail",
                chunk.start_ea,
                chunk.end_ea))

if __name__ == '__main__':
    main()
```

IDAPython example: core/list_imports.py

enumerate file imports

description: Using the API to enumerate file imports.

```
import ida_nalt

nimps = ida_nalt.get_import_module_qty()

print("Found %d import(s)..." % nimps)

for i in range(nimps):
    name = ida_nalt.get_import_module_name(i)
    if not name:
        print("Failed to get import module name for #%d" % i)
        name = "<unnamed>"

    print("Walking imports for module %s" % name)
    def imp_cb(ea, name, ordinal):
        if not name:
            print("%08x: ordinal #%d" % (ea, ordinal))
        else:
            print("%08x: %s (ordinal #%d)" % (ea, name, ordinal))
        # True -> Continue enumeration
        # False -> Stop enumeration
        return True
    ida_nalt.enum_import_names(i, imp_cb)

print("All done...")
```

IDAPython example: core/list_patched_bytes.py

enumerate patched bytes

description: Using the API to iterate over all the places in the file, that were patched using IDA.

```
import ida_bytes
import ida_idaapi

# --------------------------------------------------------------------------
class patched_bytes_visitor(object):
    def __init__(self):
```

```
        self.skip = 0
        self.patch = 0

    def __call__(self, ea, fpos, o, v, cnt=()):
        if fpos == -1:
            self.skip += 1
            print("  ea: %x o: %x v: %x...skipped" % (ea, o, v))
        else:
            self.patch += 1
            print("  ea: %x fpos: %x o: %x v: %x" % (ea, fpos, o, v))
        return 0


# --------------------------------------------------------------------------
def main():
    print("Visiting all patched bytes:")
    v = patched_bytes_visitor()
    r = ida_bytes.visit_patched_bytes(0, ida_idaapi.BADADDR, v)
    if r != 0:
        print("visit_patched_bytes() returned %d" % r)
    else:
        print("Patched: %d Skipped: %d" % (v.patch, v.skip))


# --------------------------------------------------------------------------
if __name__ == '__main__':
    main()
```

IDAPython example: core/list_problems.py

enumerate problems

description: Using the API to list all problem[atic situation]s that IDA encountered during analysis.

```
import ida_ida
import ida_idaapi
import ida_problems

for ptype in [
        ida_problems.PR_NOBASE,
        ida_problems.PR_NONAME,
        ida_problems.PR_NOFOP,
        ida_problems.PR_NOCMT,
        ida_problems.PR_NOXREFS,
```

```
        ida_problems.PR_JUMP,
        ida_problems.PR_DISASM,
        ida_problems.PR_HEAD,
        ida_problems.PR_ILLADDR,
        ida_problems.PR_MANYLINES,
        ida_problems.PR_BADSTACK,
        ida_problems.PR_ATTN,
        ida_problems.PR_FINAL,
        ida_problems.PR_ROLLED,
        ida_problems.PR_COLLISION,
        ida_problems.PR_DECIMP,
]:
    plistdesc = ida_problems.get_problem_name(ptype)
    ea = ida_ida.inf_get_min_ea()
    while True:
        ea = ida_problems.get_problem(ptype, ea+1)
        if ea == ida_idaapi.BADADDR:
            break
        print("0x%08x: %s" % (ea, plistdesc))
```

IDAPython example: core/list_segment_functions.py

list all functions (and xrefs) in segment

description: List all the functions in the current segment, as well as all the cross-references to them.

keywords: xrefs

see_also: list_segment_functions_using_idautils

```
#
# Reference Lister
#
# List all functions and all references to them in the current section.
#
# Implemented using direct IDA Plugin API calls
#

import ida_kernwin
import ida_segment
import ida_funcs
import ida_xref
import ida_idaapi

def main():
```

```
    # Get current ea
    ea = ida_kernwin.get_screen_ea()

    # Get segment class
    seg = ida_segment.getseg(ea)

    # Loop from segment start to end
    func_ea = seg.start_ea

    # Get a function at the start of the segment (if any)
    func = ida_funcs.get_func(func_ea)
    if func is None:
        # No function there, try to get the next one
        func = ida_funcs.get_next_func(func_ea)

    seg_end = seg.end_ea
    while func is not None and func.start_ea < seg_end:
        funcea = func.start_ea
        print("Function %s at 0x%x" % (ida_funcs.get_func_name(funcea), funcea))

        xb = ida_xref.xrefblk_t()
        for ref in xb.crefs_to(funcea):
            print("  called from %s(0x%x)" % (ida_funcs.get_func_name(ref), ref))

        func = ida_funcs.get_next_func(funcea)


main()
```

IDAPython example: core/list_segment_functions_using_idautils.py

list all functions (and xrefs) in segment

description: List all the functions in the current segment, as well as all the cross-references to them.

Contrary to @list_segment_functions, this uses the somewhat higher-level `idautils` module.

keywords: xrefs

see_also: list_segment_functions

```
#
# Reference Lister
#
```

```
# List all functions and all references to them in the current section.
#
# Implemented with the idautils module
#

import ida_kernwin
import ida_idaapi
import ida_segment
import ida_funcs

import idautils

def main():
    # Get current ea
    ea = ida_kernwin.get_screen_ea()
    if ea == ida_idaapi.BADADDR:
        print("Could not get get_screen_ea()")
        return

    seg = ida_segment.getseg(ea)
    if seg:
        # Loop from start to end in the current segment
        for funcea in idautils.Functions(seg.start_ea, seg.end_ea):
            print("Function %s at 0x%x" % (ida_funcs.get_func_name(funcea), funcea))

            # Find all code references to funcea
            for ref in idautils.CodeRefsTo(funcea, 1):
                print("  called from %s(0x%x)" % (ida_funcs.get_func_name(ref), ref))
    else:
        print("Please position the cursor within a segment")

if __name__=='__main__':
    main()
```

IDAPython example: core/list_stkvar_xrefs.py

list all xrefs to a function stack variable

description: Contrary to (in-memory) data & code xrefs, retrieving stack variables xrefs requires a bit more work than just using ida_xref's first_to(), next_to() (or higher level utilities such as idautils.XrefsTo)

keywords: xrefs

```
ACTION_NAME = "list_stkvar_xrefs:list"
```

```
ACTION_SHORTCUT = "Ctrl+Shift+F7"

import ida_bytes
import ida_frame
import ida_funcs
import ida_ida
import ida_kernwin
import ida_struct
import ida_ua

class list_stkvar_xrefs_ah_t(ida_kernwin.action_handler_t):
    def activate(self, ctx):
        cur_ea = ida_kernwin.get_screen_ea()
        pfn = ida_funcs.get_func(cur_ea)
        if pfn:
            v = ida_kernwin.get_current_viewer()
            result = ida_kernwin.get_highlight(v)
            if result:
                stkvar_name, _ = result
                frame = ida_frame.get_frame(cur_ea)
                sptr = ida_struct.get_struc(frame.id)
                mptr = ida_struct.get_member_by_name(sptr, stkvar_name)
                if mptr:
                    for ea in pfn:
                        F = ida_bytes.get_flags(ea)
                        for n in range(ida_ida.UA_MAXOP):
                            if not ida_bytes.is_stkvar(F, n):
                                continue
                            insn = ida_ua.insn_t()
                            if not ida_ua.decode_insn(insn, ea):
                                continue
                            v = ida_frame.calc_stkvar_struc_offset(pfn, insn, n)
                            if v >= mptr.soff and v < mptr.eoff:
                                print("Found xref at 0x%08x, operand #%d" % (ea, n))
                else:
                    print("No stack variable named \"%s\"" % stkvar_name)
        else:
            print("Please position the cursor within a function")

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_FOR_WIDGET \
            if ctx.widget_type == ida_kernwin.BWN_DISASM \
               else ida_kernwin.AST_DISABLE_FOR_WIDGET

adesc = ida_kernwin.action_desc_t(
    ACTION_NAME,
```

```
    "List stack variable xrefs",
    list_stkvar_xrefs_ah_t(),
    ACTION_SHORTCUT)

if ida_kernwin.register_action(adesc):
    print("Action registered. Please press \"%s\" to use" % ACTION_SHORTCUT)
```

IDAPython example: core/list_strings.py

retrieve the strings that are present in the IDB

description: This uses `idautils.Strings` to iterate over the string literals that are present in the IDB. Contrary to @show_selected_strings, this will not require that the "Strings" window is opened & available.

see_also: show_selected_strings

```
import ida_nalt
import idautils

s = idautils.Strings(False)
s.setup(strtypes=[ida_nalt.STRTYPE_C, ida_nalt.STRTYPE_C_16])
for i, v in enumerate(s):
    if v is None:
        print("Failed to retrieve string index %d" % i)
    else:
        print("%x: len=%d type=0x%x index=%d-> '%s'" % (v.ea, v.length, v.strtype, i, str(v)
```

IDAPython example: core/produce_c_file.py

decompile entire file

description: automate IDA to perform auto-analysis on a file and, once that is done, produce a .c file containing the decompilation of all the functions in that file.

Run like so:

```
    ida -A "-S...path/to/produce_c_file.py" <binary-file>
```

where:

```
* -A instructs IDA to run in non-interactive mode
* -S holds a path to the script to run (note this is a single token;
     there is no space between '-S' and its path.)
```

```
import ida_pro
import ida_auto
import ida_loader
import ida_hexrays

# derive output file name
idb_path = ida_loader.get_path(ida_loader.PATH_TYPE_IDB)
c_path = "%s.c" % idb_path

ida_auto.auto_wait() # wait for end of auto-analysis
ida_hexrays.decompile_many( # generate .c file
    c_path,
    None,
    ida_hexrays.VDRUN_NEWFILE
   |ida_hexrays.VDRUN_SILENT
   |ida_hexrays.VDRUN_MAYSTOP)

ida_pro.qexit(0)
```

IDAPython example: core/produce_lst_file.py

produce listing

description: automate IDA to perform auto-analysis on a file and, once that is done, produce a .lst file with the disassembly.

Run like so:

```
    ida -A "-S...path/to/produce_lst_file.py" <binary-file>
```

where:

```
* -A instructs IDA to run in non-interactive mode
* -S holds a path to the script to run (note this is a single token;
     there is no space between '-S' and its path.)
```

```
import ida_auto
import ida_fpro
import ida_ida
import ida_loader
import ida_pro

# derive output file name
idb_path = ida_loader.get_path(ida_loader.PATH_TYPE_IDB)
lst_path = "%s.lst" % idb_path

ida_auto.auto_wait() # wait for end of auto-analysis
```

```
fptr = ida_fpro.qfile_t() # FILE * wrapper
if fptr.open(lst_path, "wt"):
    try:
        ida_loader.gen_file( # generate .lst file
            ida_loader.OFILE_LST,
            fptr.get_fp(),
            ida_ida.inf_get_min_ea(),
            ida_ida.inf_get_max_ea(),
            0)
    finally:
        fptr.close()

ida_pro.qexit(0)
```

IDAPython example: core/register_timer.py

using timers for delayed execution

description: Register (possibly repeating) timers.

```
import ida_kernwin

# --------------------------------------------------------------------------
class timercallback_t(object):
    def __init__(self):
        self.interval = 1000
        self.obj = ida_kernwin.register_timer(self.interval, self)
        if self.obj is None:
            raise RuntimeError("Failed to register timer")
        self.times = 5

    def __call__(self):
        print("Timer invoked. %d time(s) left" % self.times)
        self.times -= 1
        # Unregister the timer when the counter reaches zero
        return -1 if self.times == 0 else self.interval

    def __del__(self):
        print("Timer object disposed %s" % self)


# --------------------------------------------------------------------------
def main():
    try:
        t = timercallback_t()
```

```
        # No need to unregister the timer.
        # It will unregister itself in the callback when it returns -1
    except Exception as e:
        print("Error: %s" % e)


# -----------------------------------------------------------------------
if __name__ == '__main__':
    main()
```

IDAPython example: core/trigger_actions_programmatically.py

execute existing actions programmatically

description: It's possible to invoke any action programmatically, by using either of those two:

```
* ida_kernwin.execute_ui_requests()
* ida_kernwin.process_ui_action()
```

Ideally, this script should be run through the "File > Script file..." menu, so as to keep focus on "IDA View-A" and have the 'ProcessUiActions' part work as intended.

keywords: actions

```
import ida_kernwin

# -----------------------------------------------------------------------
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
        aname = self.__action_list[self.__idx]
        print("executing: %s (flags=0x%x)" % (aname, self.__flags))
        print("=> %s" % ida_kernwin.process_ui_action(aname, self.__flags))

        # Move to next action
        self.__idx += 1
        print("index=%d" % self.__idx)

        # 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 print_req_t(object):
    def __init__(self, s):
        self.s = s
    def __call__(self):
        ida_kernwin.msg("%s" % self.s)
        return False # Don't reschedule


if ida_kernwin.ask_yn(
        1, ("HIDECANCEL\nDo you want to run execute_ui_requests() example?\n"
            "Press NO to execute ProcessUiActions() example\n")):
    ida_kernwin.execute_ui_requests(
        (print_req_t("Hello"),
```

```
        print_req_t(" world\n")) )
else:
    ProcessUiActions("JumpQ;Breakpoints")
```

IDAPython example: cvt64/py_cvt64_sample.py

This file contains the CVT64 examples.

description: For more infortmation see SDK/plugins/cvt64_sample example

```
import idaapi
import ida_idaapi
import ida_netnode

SAMPLE_NETNODE_NAME = "$ cvt64 py_sample netnode"
DEVICE_INDEX   = idaapi.BADADDR     # -1
IDPFLAGS_INDEX = idaapi.BADADDR     # -1
HASH_COMMENT   = "Comment"
HASH_ADDRESS   = "Address"

#--------------------------------------------------------------------------
class idp_listener_t(idaapi.IDP_Hooks):
    def __init__(self):
        idaapi.IDP_Hooks.__init__(self)

    def ev_cvt64_hashval(self, node, tag, name, data):
        helper = idaapi.netnode(SAMPLE_NETNODE_NAME)
        if helper == node and tag == ida_netnode.htag:
            if name == HASH_COMMENT:
                comment = helper.hashstr(name)
                helper.hashset_buf(name, comment)
                return 1
            if name == HASH_ADDRESS:
                address = helper.hashval_long(name)
                if address == ida_idaapi.BADADDR32:
                    address = ida_idaapi.BADADDR
                    helper.hashset_idx(name, address)
                return 1
        return 0

    def ev_cvt64_supval(self, node, tag, idx, data):
        helper = idaapi.netnode(SAMPLE_NETNODE_NAME)
        if helper == node:
            if tag == ida_netnode.stag and idx == ida_idaapi.BADADDR32:
                helper.supset(DEVICE_INDEX, data)
```

```
                return 1
            if tag == ida_netnode.atag and len(data):
                if idx == ida_idaapi.BADADDR32:
                    idx = IDPFLAGS_INDEX
                val = int.from_bytes(data, 'little')
                if val == ida_idaapi.BADADDR32:
                    val = ida_idaapi.BADADDR
                helper.altset(idx, val)
                return 1
        return 0

#--------------------------------------------------------------------------
# This class is instantiated once per each opened database.
class cvt64_ctx_t(idaapi.plugmod_t):

    def __init__(self):
        self.prochook = idp_listener_t()
        self.prochook.hook()

    def __del__(self):
        self.prochook.unhook()

    def run(self, arg):
        pass

#--------------------------------------------------------------------------
# This class is instantiated when IDA loads the plugin.
class cvt64_sample_t(idaapi.plugin_t):
    flags = idaapi.PLUGIN_MULTI | idaapi.PLUGIN_MOD
    comment = "IDAPython: An example how to implement CVT64 functionality"
    wanted_name = "IDAPython: CVT64 sample"
    wanted_hotkey = ""
    help = ""

    def init(self):
        return cvt64_ctx_t()

#--------------------------------------------------------------------------
def PLUGIN_ENTRY():
    return cvt64_sample_t()
```

IDAPython example: debugging/show_debug_names.py

retrieving & dumping debuggee symbols

description: Queries the debugger (possibly remotely) for the list of symbols

that the process being debugged, provides.

```
import ida_dbg
import ida_ida
import ida_name

def main():
    if not ida_dbg.is_debugger_on():
        print("Please run the process first!")
        return
    if ida_dbg.get_process_state() != ida_dbg.DSTATE_SUSP:
        print("Please suspend the debugger first!")
        return

    dn = ida_name.get_debug_names(
        ida_ida.inf_get_min_ea(),
        ida_ida.inf_get_max_ea())
    for i in dn:
        print("%08x: %s" % (i, dn[i]))

main()
```

IDAPython example: debugging/appcall/simple_appcall_linux.py

executing code into the application being debugged (on Linux)

description: Using the `ida_idd.Appcall` utility to execute code in the process being debugged.

This example will run the test program and stop wherever the cursor currently is, and then perform an appcall to execute the `ref4` and `ref8` functions.

To use this example:

```
* run `ida64` on test program `simple_appcall_linux64`, or
  `ida` on test program `simple_appcall_linux32`, and wait for
  auto-analysis to finish
* select the 'linux debugger' (either local, or remote)
* run this script
```

Note: the real body of code is in `simple_appcall_common.py`.

```
import os
import sys
sys.path.append(os.path.dirname(__file__))
```

```
import simple_appcall_common
appcall_hooks = simple_appcall_common.appcall_hooks_t()
appcall_hooks.hook()
appcall_hooks.run()
```

IDAPython example: debugging/appcall/simple_appcall_win.py

executing code into the application being debugged (on Windows)

description: Using the `ida_idd.Appcall` utility to execute code in the process being debugged.

This example will run the test program and stop wherever the cursor currently is, and then perform an appcall to execute the `ref4` and `ref8` functions.

To use this example:

```
* run `ida64` on test program `simple_appcall_win64.exe`, or
  `ida` on test program `simple_appcall_win32.exe`, and wait for
  auto-analysis to finish
* select the 'windows debugger' (either local, or remote)
* run this script
```

Note: the real body of code is in `simple_appcall_common.py`.

```
import os
import sys
sys.path.append(os.path.dirname(__file__))

# Windows binaries don't have any symbols, thus we'll have
# to assign names to addresses of interest before we can
# appcall them by name.
import ida_ida
if ida_ida.inf_is_64bit():
    ref4_ea = 0x140001000
    ref8_ea = 0x140001060
else:
    ref4_ea = 0x401000
    ref8_ea = 0x401050

import simple_appcall_common
appcall_hooks = simple_appcall_common.appcall_hooks_t(
    name_funcs=[
        (ref4_ea, "ref4"),
        (ref8_ea, "ref8"),
    ])
```

```
appcall_hooks.hook()
appcall_hooks.run()
```

IDAPython example: debugging/dbghooks/automatic_steps.py

programmatically drive a debugging session

description: Start a debugging session, step through the first five instructions.
Each instruction is disassembled after execution.

```
import ida_dbg
import ida_ida
import ida_lines

class MyDbgHook(ida_dbg.DBG_Hooks):
    """ Own debug hook class that implementd the callback functions """

    def __init__(self):
        ida_dbg.DBG_Hooks.__init__(self) # important
        self.steps = 0

    def log(self, msg):
        print(">>> %s" % msg)

    def dbg_process_start(self, pid, tid, ea, name, base, size):
        self.log("Process started, pid=%d tid=%d name=%s" % (pid, tid, name))

    def dbg_process_exit(self, pid, tid, ea, code):
        self.log("Process exited pid=%d tid=%d ea=0x%x code=%d" % (pid, tid, ea, code))

    def dbg_library_unload(self, pid, tid, ea, info):
        self.log("Library unloaded: pid=%d tid=%d ea=0x%x info=%s" % (pid, tid, ea, info))

    def dbg_process_attach(self, pid, tid, ea, name, base, size):
        self.log("Process attach pid=%d tid=%d ea=0x%x name=%s base=%x size=%x" % (pid, tid,

    def dbg_process_detach(self, pid, tid, ea):
        self.log("Process detached, pid=%d tid=%d ea=0x%x" % (pid, tid, ea))

    def dbg_library_load(self, pid, tid, ea, name, base, size):
        self.log("Library loaded: pid=%d tid=%d name=%s base=%x" % (pid, tid, name, base))

    def dbg_bpt(self, tid, ea):
        self.log("Break point at 0x%x pid=%d" % (ea, tid))
```

```
        # return values:
        #   -1 - to display a breakpoint warning dialog
        #        if the process is suspended.
        #    0 - to never display a breakpoint warning dialog.
        #    1 - to always display a breakpoint warning dialog.
        return 0

    def dbg_suspend_process(self):
        self.log("Process suspended")

    def dbg_exception(self, pid, tid, ea, exc_code, exc_can_cont, exc_ea, exc_info):
        self.log("Exception: pid=%d tid=%d ea=0x%x exc_code=0x%x can_continue=%d exc_ea=0x%x
            pid, tid, ea, exc_code & ida_idaapi.BADADDR, exc_can_cont, exc_ea, exc_info))
        # return values:
        #   -1 - to display an exception warning dialog
        #        if the process is suspended.
        #   0  - to never display an exception warning dialog.
        #   1  - to always display an exception warning dialog.
        return 0

    def dbg_trace(self, tid, ea):
        self.log("Trace tid=%d ea=0x%x" % (tid, ea))
        # return values:
        #   1  - do not log this trace event;
        #   0  - log it
        return 0

    def dbg_step_into(self):
        self.log("Step into")
        self.dbg_step_over()

    def dbg_run_to(self, pid, tid=0, ea=0):
        self.log("Runto: tid=%d, ea=%x" % (tid, ea))
        ida_dbg.request_step_over()

    def dbg_step_over(self):
        eip = ida_dbg.get_reg_val("EIP")
        disasm = ida_lines.tag_remove(
            ida_lines.generate_disasm_line(
                eip))
        self.log("Step over: EIP=0x%x, disassembly=%s" % (eip, disasm))

        self.steps += 1
        if self.steps >= 5:
            ida_dbg.request_exit_process()
        else:
```

```
            ida_dbg.request_step_over()


# Remove an existing debug hook
try:
    if debughook:
        print("Removing previous hook ...")
        debughook.unhook()
except:
    pass

# Install the debug hook
debughook = MyDbgHook()
debughook.hook()

ep = ida_ida.inf_get_start_ip()
if ida_dbg.request_run_to(ep): # Request stop at entry point
    ida_dbg.run_requests()     # Launch process
else:
    print("Impossible to prepare debugger requests. Is a debugger selected?")
```

IDAPython example: debugging/dbghooks/dbg_trace.py

using the low-level tracing hook

description: This script demonstrates using the low-level tracing hook (ida_dbg.DBG_Hooks.dbg_trace). It can be run like so:

```
ida[t].exe -B -Sdbg_trace.py -Ltrace.log file.exe
```

```
import time

import ida_dbg
import ida_ida
import ida_pro
import ida_ua
from ida_allins import NN_callni, NN_call, NN_callfi
from ida_lines import generate_disasm_line, GENDSM_FORCE_CODE, GENDSM_REMOVE_TAGS

# Note: this try/except block below is just there to
# let us (at Hex-Rays) test this script in various
# situations.
try:
    import idc
    print(idc.ARGV[1])
```

```
    under_test = bool(idc.ARGV[1])
except:
    under_test = False

class TraceHook(ida_dbg.DBG_Hooks):
    def __init__(self):
        ida_dbg.DBG_Hooks.__init__(self)
        self.traces = 0
        self.epReached = False

    def _log(self, msg):
        print(">>> %s" % msg)

    def dbg_trace(self, tid, ea):
        # Log all traced addresses
        if ea < ida_ida.inf_get_min_ea() or ea > ida_ida.inf_get_max_ea():
            raise Exception(
                "Received a trace callback for an address outside this database!"
            )

        self._log("trace %08X" % ea)
        self.traces += 1
        insn = ida_ua.insn_t()
        insnlen = ida_ua.decode_insn(insn, ea)
        # log disassembly and ESP for call instructions
        if insnlen > 0 and insn.itype in [NN_callni, NN_call, NN_callfi]:
            self._log(
                "call insn: %s"
                % generate_disasm_line(ea, GENDSM_FORCE_CODE | GENDSM_REMOVE_TAGS)
            )
            self._log("ESP=%08X" % ida_dbg.get_reg_val("ESP"))

        return 1

    def dbg_run_to(self, pid, tid=0, ea=0):
        # this hook is called once execution reaches temporary breakpoint set by run_to(ep)
        if not self.epReached:
            ida_dbg.refresh_debugger_memory()
            self._log("reached entry point at 0x%X" % ida_dbg.get_reg_val("EIP"))
            self._log("current step trace options: %x" % ida_dbg.get_step_trace_options())
            self.epReached = True

        # enable step tracing (single-step the program and generate dbg_trace events)
        ida_dbg.request_enable_step_trace(1)
        # change options to only "over debugger segments" (i.e. library functions will be t
        ida_dbg.request_set_step_trace_options(ida_dbg.ST_OVER_DEBUG_SEG)
```

```
            ida_dbg.request_continue_process()
            ida_dbg.run_requests()

        def dbg_process_exit(self, pid, tid, ea, code):
            self._log("process exited with %d" % code)
            self._log("traced %d instructions" % self.traces)
            return 0


    def do_trace(then_quit_ida=True):
        debugHook = TraceHook()
        debugHook.hook()

        # Start tracing when entry point is hit
        ep = ida_ida.inf_get_start_ip()
        ida_dbg.enable_step_trace(1)
        ida_dbg.set_step_trace_options(ida_dbg.ST_OVER_DEBUG_SEG | ida_dbg.ST_OVER_LIB_FUNC)
        print("Running to %x" % ep)
        ida_dbg.run_to(ep)

        while ida_dbg.get_process_state() != 0:
            ida_dbg.wait_for_next_event(1, 0)

        if not debugHook.epReached:
            raise Exception("Entry point wasn't reached!")

        if not debugHook.unhook():
            raise Exception("Error uninstalling hooks!")

        del debugHook

        if then_quit_ida:
            # we're done; exit IDA
            ida_pro.qexit(0)


    # load the debugger module depending on the file type
    if ida_ida.inf_get_filetype() == ida_ida.f_PE:
        ida_dbg.load_debugger("win32", 0)
    elif ida_ida.inf_get_filetype() == ida_ida.f_ELF:
        ida_dbg.load_debugger("linux", 0)
    elif ida_ida.inf_get_filetype() == ida_ida.f_MACHO:
        ida_dbg.load_debugger("mac", 0)
    if not under_test:
        do_trace()
```

IDAPython example: debugging/misc/print_call_stack.py

print call stack (on Linux)

description: print the return addresses from the call stack at a breakpoint. (and print also the module and the debug name from debugger)

To use this example:

```
* run `ida64` on test program `simple_appcall_linux64`, or
  `ida` on test program `simple_appcall_linux32`, and wait for
  auto-analysis to finish
* put a breakpoint where you want to see the call stack
* select the 'linux debugger' (either local, or remote)
* start debugging
* Press Shift+C at the breakpoint
```

```
import os
import ida_idaapi
import ida_idd
import ida_dbg
import ida_kernwin
import ida_name

def log(msg):
    print(">>> %s" % msg)

class print_call_stack_ah_t():
    def activate(self, ctx):
        log("=== start of call stack impression ===")
        tid = ida_dbg.get_current_thread()
        trace = ida_idd.call_stack_t()
        if ida_dbg.collect_stack_trace(tid, trace):
            for frame in trace:
                mi = ida_idd.modinfo_t()
                if ida_dbg.get_module_info(frame.callea, mi):
                    module = os.path.basename(mi.name)
                    name = ida_name.get_nice_colored_name(
                    frame.callea,
                    ida_name.GNCN_NOCOLOR|ida_name.GNCN_NOLABEL|ida_name.GNCN_NOSEG|ida_name
                    log("Return address: " + hex(frame.callea) + " from: " + module + " with
                else:
                    log("Return address: " + hex(frame.callea))
        log("===  end of call stack impression  ===")

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_ALWAYS
```

```
ACTION_NAME = "example:print_call_stack"
ACTION_LABEL = "Print call stack"
ACTION_SHORTCUT = "Shift+C"
ACTION_HELP = "Press %s to dump the call stack" % ACTION_SHORTCUT

if ida_kernwin.register_action(ida_kernwin.action_desc_t(
        ACTION_NAME,
        ACTION_LABEL,
        print_call_stack_ah_t(),
        ACTION_SHORTCUT)):
    print("Registered action \"%s\". %s" % (ACTION_LABEL, ACTION_HELP))
```

IDAPython example: debugging/misc/print_registers.py

print all registers, for all threads

description: iterate over the list of threads in the program being debugged, and dump all registers contents

To use this example:

```
* run `ida64` on test program `simple_appcall_linux64`, or
  `ida` on test program `simple_appcall_linux32`, and wait for
  auto-analysis to finish
* put a breakpoint somewhere in the code
* select the 'linux debugger' (either local, or remote)
* start debugging
* Press Alt+Shift+C at the breakpoint


import ida_idd
import ida_dbg

def log(msg):
    print(">>> %s" % msg)

class print_registers_ah_t():
    def activate(self, ctx):
        log("=== registers ===")
        dbg = ida_idd.get_dbg()
        for tidx in range(ida_dbg.get_thread_qty()):
            tid = ida_dbg.getn_thread(tidx)
            log("  Thread #%d" % tid)
            regvals = ida_dbg.get_reg_vals(tid)
            for ridx, rv in enumerate(regvals):
                rinfo = dbg.regs(ridx)
```

```
                rval = rv.pyval(rinfo.dtype)
                if isinstance(rval, int):
                    rval = "0x%x" % rval
                log("    %s: %s" % (rinfo.name, rval))
        log("=== end of registers  ===")

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_ALWAYS

ACTION_NAME = "example:print_registers"
ACTION_LABEL = "Print registers"
ACTION_SHORTCUT = "Alt+Shift+C"
ACTION_HELP = "Press %s to print the registers" % ACTION_SHORTCUT

if ida_kernwin.register_action(ida_kernwin.action_desc_t(
        ACTION_NAME,
        ACTION_LABEL,
        print_registers_ah_t(),
        ACTION_SHORTCUT)):
    print("Registered action \"%s\". %s" % (ACTION_LABEL, ACTION_HELP))
```

IDAPython example: debugging/misc/registers_context_menu.py

adding actions to the “registers” widget(s)

description: It’s possible to add actions to the context menu of pretty much all widgets in IDA.

This example shows how to do just that for registers-displaying widgets (e.g., “General registers”)

```
import ida_dbg
import ida_idd
import ida_kernwin
import ida_ua

ACTION_NAME = "registers_context_menu:dump_reg"

class dump_reg_ah_t(ida_kernwin.action_handler_t):
    def activate(self, ctx):
        name = ctx.regname
        value = ida_dbg.get_reg_val(name)
        rtype = "integer"
        rinfo = ida_idd.register_info_t()
```

```
            if ida_dbg.get_dbg_reg_info(name, rinfo):
                if rinfo.dtype == ida_ua.dt_byte:
                    value = "0x%02x" % value
                elif rinfo.dtype == ida_ua.dt_word:
                    value = "0x%04x" % value
                elif rinfo.dtype == ida_ua.dt_dword:
                    value = "0x%08x" % value
                elif rinfo.dtype == ida_ua.dt_qword:
                    value = "0x%016x" % value
                else:
                    rtype = "float"
            print("> Register %s (of type %s): %s" % (name, rtype, value))

        def update(self, ctx):
            return ida_kernwin.AST_ENABLE_FOR_WIDGET \
                if ctx.widget_type == ida_kernwin.BWN_CPUREGS \
                else ida_kernwin.AST_DISABLE_FOR_WIDGET


    if ida_kernwin.register_action(
            ida_kernwin.action_desc_t(
                ACTION_NAME,
                "Dump register info",
                dump_reg_ah_t())):

        class registers_hooks_t(ida_kernwin.UI_Hooks):
            def finish_populating_widget_popup(self, form, popup):
                if ida_kernwin.get_widget_type(form) == ida_kernwin.BWN_CPUREGS:
                    ida_kernwin.attach_action_to_popup(form, popup, ACTION_NAME)

        hooks = registers_hooks_t()
        hooks.hook()
    else:
        print("Failed to register action")
```

IDAPython example: hexrays/colorize_pseudocode_lines.py

interactively color certain pseudocode lines

description: Provides an action that can be used to dynamically alter the lines background rendering for pseudocode listings (as opposed to using `ida_hexrays.cfunc_t.pseudocode[N].bgcolor`)

After running this script, pressing 'M' on a line in a "Pseudocode-?" widget, will cause that line to be rendered with a special background color.

keywords: colors

```
import ida_kernwin
import ida_hexrays
import ida_moves
import ida_idaapi

class pseudo_line_t(object):
    def __init__(self, func_ea, line_nr):
        self.func_ea = func_ea
        self.line_nr = line_nr

    def __hash__(self):
        return hash((self.func_ea, self.line_nr))

    def __eq__(self, r):
        return self.func_ea == r.func_ea \
            and self.line_nr == r.line_nr


def _place_to_line_number(p):
    return ida_kernwin.place_t.as_simpleline_place_t(p).n


class pseudocode_lines_rendering_hooks_t(ida_kernwin.UI_Hooks):
    def __init__(self):
        ida_kernwin.UI_Hooks.__init__(self)
        self.marked_lines = {}

    def get_lines_rendering_info(self, out, widget, rin):
        vu = ida_hexrays.get_widget_vdui(widget)
        if vu:
            entry_ea = vu.cfunc.entry_ea
            for section_lines in rin.sections_lines:
                for line in section_lines:
                    coord = pseudo_line_t(
                        entry_ea,
                        _place_to_line_number(line.at))
                    color = self.marked_lines.get(coord, None)
                    if color is not None:
                        e = ida_kernwin.line_rendering_output_entry_t(line)
                        e.bg_color = color
                        out.entries.push_back(e)


class toggle_line_marked_ah_t(ida_kernwin.action_handler_t):
```

```
        """
        We could very well use an ARGB value, but instead let's go
        go with a color 'key': those can be altered by the user/theme,
        and therefore have a better chance of being appropriate (or at
        least expected.)
        """
        COLOR_KEY = ida_kernwin.CK_EXTRA11

        def __init__(self, hooks):
            ida_kernwin.action_handler_t.__init__(self)
            self.hooks = hooks

        def activate(self, ctx):
            vu = ida_hexrays.get_widget_vdui(ctx.widget)
            if vu:
                loc = ida_moves.lochist_entry_t()
                if ida_kernwin.get_custom_viewer_location(loc, ctx.widget):
                    coord = pseudo_line_t(
                        vu.cfunc.entry_ea,
                        _place_to_line_number(loc.place()))
                    if coord in self.hooks.marked_lines.keys():
                        del self.hooks.marked_lines[coord]
                    else:
                        self.hooks.marked_lines[coord] = self.COLOR_KEY
                    ida_kernwin.refresh_custom_viewer(ctx.widget)

        def update(self, ctx):
            return ida_kernwin.AST_ENABLE_FOR_WIDGET \
                if ctx.widget_type == ida_kernwin.BWN_PSEUDOCODE \
                else ida_kernwin.AST_DISABLE_FOR_WIDGET


    hooks = pseudocode_lines_rendering_hooks_t()
    act_name = "example:colorize_pseudocode_line"
    act_shortcut = "M"
    if ida_kernwin.register_action(ida_kernwin.action_desc_t(
            act_name,
            "Mark pseudocode line",
            toggle_line_marked_ah_t(hooks),
            act_shortcut)):
        hooks.hook()
        print("Action registered. Please press '%s' in a pseudocode window to mark a line" % act
```

IDAPython example: hexrays/curpos_details.py

a focus on the 'curpos' hook, printing additional details about user input

description: Shows how user input information can be retrieved during processing of a notification triggered by that input

see_also: vds_hooks

```
import ida_hexrays
import ida_kernwin

class curpos_details_t(ida_hexrays.Hexrays_Hooks):
    def curpos(self, v):
        parts = ["cpos={lnnum=%d, x=%d, y=%d}" % (v.cpos.lnnum, v.cpos.x, v.cpos.y)]
        uie = ida_kernwin.input_event_t()
        if ida_kernwin.get_user_input_event(uie):
            kind_str = {
                ida_kernwin.iek_shortcut : "shortcut",
                ida_kernwin.iek_key_press : "key_press",
                ida_kernwin.iek_key_release : "key_release",
                ida_kernwin.iek_mouse_button_press : "mouse_button_press",
                ida_kernwin.iek_mouse_button_release : "mouse_button_release",
                ida_kernwin.iek_mouse_wheel : "mouse_wheel",
            }[uie.kind]

            #
            # Retrieve input kind-specific information
            #
            if uie.kind == ida_kernwin.iek_shortcut:
                payload_str = "shortcut={action_name=%s}" % uie.shortcut.action_name
            elif uie.kind in [
                    ida_kernwin.iek_key_press,
                    ida_kernwin.iek_key_release]:
                payload_str = "keyboard={key=%d, text=%s}" % (uie.keyboard.key, uie.keyboard
            else:
                payload_str = "mouse={x=%d, y=%d, button=%d}" % (
                    uie.mouse.x,
                    uie.mouse.y,
                    uie.mouse.button)

            #
            # And while at it, retrieve a few extra bits from the
            # source QEvent as well, why not
            #
```

```
                qevent = uie.get_source_QEvent()
                qevent_str = str(qevent)
                from PyQt5 import QtCore
                if qevent.type() in [
                        QtCore.QEvent.KeyPress,
                        QtCore.QEvent.KeyRelease]:
                    qevent_str="{count=%d}" % qevent.count()
                elif qevent.type() in [
                        QtCore.QEvent.MouseButtonPress,
                        QtCore.QEvent.MouseButtonRelease]:
                    qevent_str="{globalX=%d, globalY=%d, flags=%s}" % (
                        qevent.globalX(),
                        qevent.globalY(),
                        qevent.flags())
                elif qevent.type() == QtCore.QEvent.Wheel:
                    qevent_str="{angleDelta={x=%s, y=%s}, phase=%s}" % (
                        qevent.angleDelta().x(),
                        qevent.angleDelta().y(),
                        qevent.phase())

                #
                # If the target QWidget is a scroll area's viewport,
                # pick up the parent
                #
                from PyQt5 import QtWidgets
                qwidget = uie.get_target_QWidget()
                if qwidget:
                    parent = qwidget.parentWidget()
                    if parent and isinstance(parent, QtWidgets.QAbstractScrollArea):
                        qwidget = parent

                parts.append("user_input_event={kind=%s, modifiers=0x%x, target={metaObject={cla
                    kind_str,
                    uie.modifiers,
                    qwidget.metaObject().className(),
                    qwidget.windowTitle(),
                    uie.source,
                    payload_str,
                    qevent_str))
            print("### curpos: %s" % ", ".join(parts))
            return 0

curpos_details = curpos_details_t()
curpos_details.hook()
```

IDAPython example: hexrays/decompile_entry_points.py

automatic decompilation of functions

description: Attempts to load a decompiler plugin corresponding to the current architecture (and address size) right after auto-analysis is performed, and then tries to decompile the function at the first entrypoint.

It is particularly suited for use with the '-S' flag, for example: idat -Ldecompile.log -Sdecompile_entry_points.py -c file

```
import ida_ida
import ida_auto
import ida_loader
import ida_hexrays
import ida_idp
import ida_entry
import ida_kernwin

# because the -S script runs very early, we need to load the decompiler
# manually if we want to use it
def init_hexrays():
    ALL_DECOMPILERS = {
        ida_idp.PLFM_386: "hexrays",
        ida_idp.PLFM_ARM: "hexarm",
        ida_idp.PLFM_PPC: "hexppc",
        ida_idp.PLFM_MIPS: "hexmips",
    }
    cpu = ida_idp.ph.id
    decompiler = ALL_DECOMPILERS.get(cpu, None)
    if not decompiler:
        print("No known decompilers for architecture with ID: %d" % ida_idp.ph.id)
        return False
    if ida_ida.inf_is_64bit():
        if cpu == ida_idp.PLFM_386:
            decompiler = "hexx64"
        else:
            decompiler += "64"
    if ida_loader.load_plugin(decompiler) and ida_hexrays.init_hexrays_plugin():
        return True
    else:
        print('Couldn\'t load or initialize decompiler: "%s"' % decompiler)
        return False
```

```
def decompile_func(ea, outfile):
    ida_kernwin.msg("Decompiling at: %X..." % ea)
    cf = ida_hexrays.decompile(ea)
    if cf:
        ida_kernwin.msg("OK\n")
        outfile.write(str(cf) + "\n")
    else:
        ida_kernwin.msg("failed!\n")
        outfile.write("decompilation failure at %X!\n" % ea)


def main():
    print("Waiting for autoanalysis...")
    ida_auto.auto_wait()
    if init_hexrays():
        eqty = ida_entry.get_entry_qty()
        if eqty:
            idbpath = idc.get_idb_path()
            cpath = idbpath[:-4] + ".c"
            with open(cpath, "w") as outfile:
                print("writing results to '%s'..." % cpath)
                for i in range(eqty):
                    ea = ida_entry.get_entry(ida_entry.get_entry_ordinal(i))
                    decompile_func(ea, outfile)
        else:
            print("No known entrypoint. Cannot decompile.")
    if ida_kernwin.cvar.batch:
        print("All done, exiting.")
        ida_pro.qexit(0)

main()
```

IDAPython example: hexrays/vds1.py

decompile & print current function.

```
import ida_hexrays
import ida_lines
import ida_funcs
import ida_kernwin

def main():
    if not ida_hexrays.init_hexrays_plugin():
        return False
```

```
    print("Hex-rays version %s has been detected" % ida_hexrays.get_hexrays_version())

    f = ida_funcs.get_func(ida_kernwin.get_screen_ea());
    if f is None:
        print("Please position the cursor within a function")
        return True

    cfunc = ida_hexrays.decompile(f);
    if cfunc is None:
        print("Failed to decompile!")
        return True

    sv = cfunc.get_pseudocode();
    for sline in sv:
        print(ida_lines.tag_remove(sline.line));

    return True

main()
```

IDAPython example: hexrays/vds10.py

a custom microcode instruction optimization rule

description: Installs a custom microcode instruction optimization rule, to transform:

```
  call   !DbgRaiseAssertionFailure <fast:>.0
```

into

```
  call   !DbgRaiseAssertionFailure <fast:"char *" "assertion text">.0
```

To see this plugin in action please use arm64_brk.i64

```
import ida_bytes
import ida_range
import ida_kernwin
import ida_hexrays
import ida_typeinf
import ida_idaapi

class nt_assert_optimizer_t(ida_hexrays.optinsn_t):
    def func(self, blk, ins, optflags):
        if self.handle_nt_assert(ins):
            return 1
```

```
        return 0

    def handle_nt_assert(self, ins):
        # recognize call   !DbgRaiseAssertionFailure <fast:>.0
        if not ins.is_helper("DbgRaiseAssertionFailure"):
            return False

        # did we already add an argument?
        fi = ins.d.f;
        if not fi.args.empty():
            return False

        # use a comment from the disassembly listing as the call argument
        cmt = ida_bytes.get_cmt(ins.ea, False)
        if not cmt:
            return False

        # remove "NT_ASSERT("...")" to make the listing nicer
        if cmt.startswith("NT_ASSERT(\""):
            cmt = cmt[11:]
            if cmt.endswith("\")"):
                cmt = cmt[:-2]

        # all ok, transform the instruction by adding one more call argument
        fa = fi.args.push_back()
        fa.t = ida_hexrays.mop_str;
        fa.cstr = cmt
        fa.type = ida_typeinf.tinfo_t.get_stock(ida_typeinf.STI_PCCHAR) # const char *
        fa.size = fa.type.get_size()
        return True

# --------------------------------------------------------------------------
# a plugin interface, boilerplate code
class my_plugin_t(ida_idaapi.plugin_t):
    flags = ida_idaapi.PLUGIN_HIDE
    wanted_name = "Optimize DbgRaiseAssertionFailure (IDAPython)"
    wanted_hotkey = ""
    comment = "Sample plugin10 for Hex-Rays decompiler"
    help = ""
    def init(self):
        if ida_hexrays.init_hexrays_plugin():
            self.optimizer = nt_assert_optimizer_t()
            self.optimizer.install()
        return ida_idaapi.PLUGIN_KEEP # keep us in the memory
    def term(self):
        self.optimizer.remove()
```

```
    def run(self, arg):
        if arg == 1:
            return self.optimizer.remove()
        elif arg == 2:
            return self.optimizer.install()

def PLUGIN_ENTRY():
    return my_plugin_t()
```

IDAPython example: hexrays/vds11.py

a custom microcode block optimization rule (resolve `goto` chains)

description: Installs a custom microcode block optimization rule, to transform:

```
    goto L1
    ...
  L1:
    goto L2
```

into

```
    goto L2
```

In other words we fix a goto target if it points to a chain of gotos. This improves the decompiler output in some cases.

```
import ida_bytes
import ida_range
import ida_kernwin
import ida_hexrays
import ida_typeinf
import ida_idaapi

class goto_optimizer_t(ida_hexrays.optblock_t):
    def func(self, blk):
        if self.handle_goto_chain(blk):
            return 1
        return 0

    def handle_goto_chain(self, blk):
        mgoto = blk.tail
        if not mgoto or mgoto.opcode != ida_hexrays.m_goto:
            return False

        visited = []
```

```
        t0 = mgoto.l.b
        i = t0
        mba = blk.mba

        # follow the goto chain
        while True:
            if i in visited:
                return False
            visited.append(i)
            b = mba.get_mblock(i)
            m2 = ida_hexrays.getf_reginsn(b.head)
            if not m2 or m2.opcode != ida_hexrays.m_goto:
                break
            i = m2.l.b

        if i == t0:
            return False # not a chain

        # all ok, found a goto chain
        mgoto.l.b = i # jump directly to the end of the chain

        # fix the successor/predecessor lists
        blk.succset[0] = i
        mba.get_mblock(i).predset.add(blk.serial)
        mba.get_mblock(t0).predset._del(blk.serial)

        # since we changed the control flow graph, invalidate the use/def chains.
        # stricly speaking it is not really necessary in our plugin because
        # we did not move around any microcode operands.
        mba.mark_chains_dirty()

        # it is a good idea to verify microcode after each change
        # however, it may be time consuming, so comment it out eventually
        mba.verify(True);
        return True

# --------------------------------------------------------------------------
# a plugin interface, boilerplate code
class my_plugin_t(ida_idaapi.plugin_t):
    flags = ida_idaapi.PLUGIN_HIDE
    wanted_name = "Optimize goto chains (IDAPython)"
    wanted_hotkey = ""
    comment = "Sample plugin11 for Hex-Rays decompiler"
    help = ""
    def init(self):
        if ida_hexrays.init_hexrays_plugin():
```

```
            self.optimizer = goto_optimizer_t()
            self.optimizer.install()
            return ida_idaapi.PLUGIN_KEEP # keep us in the memory
    def term(self):
        self.optimizer.remove()
    def run(self, arg):
        if arg == 1:
            return self.optimizer.remove()
        elif arg == 2:
            return self.optimizer.install()

def PLUGIN_ENTRY():
    return my_plugin_t()
```

IDAPython example: hexrays/vds12.py

list instruction registers

description: Shows a list of direct references to a register from the current instruction.

```
import ida_pro
import ida_hexrays
import ida_kernwin
import ida_funcs
import ida_bytes
import ida_lines

def collect_block_xrefs(out, mlist, blk, ins, find_uses):
    p = ins
    while p and not mlist.empty():
        use = blk.build_use_list(p, ida_hexrays.MUST_ACCESS); # things used by the insn
        _def = blk.build_def_list(p, ida_hexrays.MUST_ACCESS); # things defined by the insn
        plst = use if find_uses else _def
        if mlist.has_common(plst):
            if not p.ea in out:
                out.append(p.ea) # this microinstruction seems to use our operand
        mlist.sub(_def)
        p = p.next if find_uses else p.prev


def collect_xrefs(out, ctx, mop, mlist, du, find_uses):
    # first collect the references in the current block
    start = ctx.topins.next if find_uses else ctx.topins.prev;
```

```
    collect_block_xrefs(out, mlist, ctx.blk, start, find_uses)

    # then find references in other blocks
    serial = ctx.blk.serial; # block number of the operand
    bc = du[serial]          # chains of that block
    voff = ida_hexrays.voff_t(mop)
    ch = bc.get_chain(voff)   # chain of the operand
    if not ch:
        return # odd
    for bn in ch:
        b = ctx.mba.get_mblock(bn)
        ins = b.head if find_uses else b.tail
        tmp = ida_hexrays.mlist_t()
        tmp.add(mlist)
        collect_block_xrefs(out, tmp, b, ins, find_uses)


class xref_chooser_t(ida_kernwin.Choose):
    def __init__(self, xrefs, t, n, ea, gco):
        ida_kernwin.Choose.__init__(
            self,
            t,
            [["Type", 3], ["Address", 16], ["Instruction", 60]])

        self.xrefs = xrefs
        self.ndefs = n
        self.curr_ea = ea
        self.gco = gco
        self.items = [ self._make_item(idx) for idx in range(len(xrefs)) ]

    def OnGetSize(self):
        return len(self.items)

    def OnGetLine(self, n):
        return self.items[n]

    def _make_item(self, idx):
        ea = self.xrefs[idx]
        both_mask = ida_hexrays.GCO_USE|ida_hexrays.GCO_DEF
        both = (self.gco.flags & both_mask) == both_mask
        if ea == self.curr_ea and both:
            type_str = "use/def"
        elif idx < self.ndefs:
            type_str = "def"
        else:
            type_str = "use"
```

```
        insn = ida_lines.generate_disasm_line(ea, ida_lines.GENDSM_REMOVE_TAGS)
        return [type_str, "%08x" % ea, insn]


def show_xrefs(ea, gco, xrefs, ndefs):
    title = "xrefs to %s at %08x" % (gco.name, ea)
    xc = xref_chooser_t(xrefs, title, ndefs, ea, gco)
    i = xc.Show(True)
    if i >= 0:
        ida_kernwin.jumpto(xrefs[i])


if ida_hexrays.init_hexrays_plugin():
    ea = ida_kernwin.get_screen_ea()
    pfn = ida_funcs.get_func(ea)
    w = ida_kernwin.warning
    if pfn:
        F = ida_bytes.get_flags(ea)
        if ida_bytes.is_code(F):
            gco = ida_hexrays.gco_info_t()
            if ida_hexrays.get_current_operand(gco):
                # generate microcode
                hf = ida_hexrays.hexrays_failure_t()
                mbr = ida_hexrays.mba_ranges_t(pfn)
                mba = ida_hexrays.gen_microcode(
                    mbr,
                    hf,
                    None,
                    ida_hexrays.DECOMP_WARNINGS | ida_hexrays.DECOMP_NO_CACHE,
                    ida_hexrays.MMAT_PREOPTIMIZED)
                if mba:
                    merr = mba.build_graph()
                    if merr == ida_hexrays.MERR_OK:
                        ncalls = mba.analyze_calls(ida_hexrays.ACFL_GUESS)
                        if ncalls < 0:
                            print("%08x: failed to determine some calling conventions", pfn
                        mlist = ida_hexrays.mlist_t()
                        if gco.append_to_list(mlist, mba):
                            ctx = ida_hexrays.op_parent_info_t()
                            mop = mba.find_mop(ctx, ea, gco.is_def(), mlist)
                            if mop:
                                xrefs = ida_pro.eavec_t()
                                ndefs = 0
                                graph = mba.get_graph()
                                ud = graph.get_ud(ida_hexrays.GC_REGS_AND_STKVARS)
                                du = graph.get_du(ida_hexrays.GC_REGS_AND_STKVARS)
```

```
                                    if gco.is_use():
                                        collect_xrefs(xrefs, ctx, mop, mlist, ud, False)
                                        ndefs = xrefs.size()
                                        if ea not in xrefs:
                                            xrefs.append(ea)
                                    if gco.is_def():
                                        if ea not in xrefs:
                                            xrefs.append(ea)
                                            ndefs = len(xrefs)
                                        collect_xrefs(xrefs, ctx, mop, mlist, du, True)
                                    show_xrefs(ea, gco, xrefs, ndefs)
                                else:
                                    w("Could not find the operand in the microcode, sorry")
                            else:
                                w("Failed to represent %s as microcode list" % gco.name)
                        else:
                            w("%08x: %s" % (errea, ida_hexrays.get_merror_desc(merr, mba)))
                    else:
                        w("%08x: %s" % (hf.errea, hf.str))
                else:
                    w("Could not find a register or stkvar in the current operand")
            else:
                w("Please position the cursor on an instruction")
        else:
            w("Please position the cursor within a function")
    else:
        print('vds12: Hex-rays is not available.')
```

IDAPython example: hexrays/vds13.py

generates microcode for selection

description: Generates microcode for selection and dumps it to the output window.

```
import ida_bytes
import ida_range
import ida_kernwin
import ida_hexrays

if ida_hexrays.init_hexrays_plugin():
    sel, sea, eea = ida_kernwin.read_range_selection(None)
    w = ida_kernwin.warning
    if sel:
```

```
        F = ida_bytes.get_flags(sea)
        if ida_bytes.is_code(F):
            hf = ida_hexrays.hexrays_failure_t()
            mbr = ida_hexrays.mba_ranges_t()
            mbr.ranges.push_back(ida_range.range_t(sea, eea))
            mba = ida_hexrays.gen_microcode(mbr, hf, None, ida_hexrays.DECOMP_WARNINGS)
            if mba:
                print("Successfully generated microcode for 0x%08x..0x%08x\n" % (sea, eea))
                vp = ida_hexrays.vd_printer_t()
                mba._print(vp)
            else:
                w("0x%08x: %s" % (hf.errea, hf.str))
        else:
            w("The selected range must start with an instruction")
    else:
        w("Please select a range of addresses to analyze")
else:
    print('vds13: Hex-rays is not available.')
```

IDAPython example: hexrays/vds17.py

using the "Select offsets" widget

description: Registers an action opens the "Select offsets" widget (select_udt_by_offset() call).

This effectively repeats the functionality already available through Alt+Y.

Place cursor on the union field and press Shift+T

```
import ida_idaapi
import ida_hexrays
import ida_lines
import ida_typeinf
import ida_kernwin

# --------------------------------------------------------------------------
class func_stroff_ah_t(ida_kernwin.action_handler_t):
    def __init__(self):
        ida_kernwin.action_handler_t.__init__(self)

    def activate(self, ctx):
        # get the current item
        vu = ida_hexrays.get_widget_vdui(ctx.widget)
        vu.get_current_item(ida_hexrays.USE_KEYBOARD)
```

```
# REGION1, will be referenced later
# check that the current item is a union field
if not vu.item.is_citem():
    ida_kernwin.warning("Please position the cursor on a union member")
    return 0
e = vu.item.e
while True:
    op = e.op
    if op != ida_hexrays.cot_memptr and op != ida_hexrays.cot_memref:
        ida_kernwin.warning("Please position the cursor on a union member")
        return 0
    e = e.x
    if op == ida_hexrays.cot_memptr:
        if e.type.is_union():
            break
    else:
        if ida_typeinf.remove_pointer(e.type).is_union():
            break
    if not e.type.is_udt():
        ida_kernwin.warning("Please position the cursor on a union member")
        return 0
# END REGION1

# REGION2
# calculate the member offset
off = 0
e = vu.item.e
while True:
    e2 = e.x
    tif = ida_typeinf.remove_pointer(e2.type)
    if not tif.is_union():
        off += e.m
    e = e2
    if e2.op != ida_hexrays.cot_memptr and e2.op != ida_hexrays.cot_memref:
        break
    if not e2.type.is_udt():
        break
# END REGION2

# REGION3
# go up and collect more member references (in order to calculate the final offset)
p = vu.item.e
while True:
    p2 = vu.cfunc.body.find_parent_of(p)
    if p2.op == ida_hexrays.cot_memptr:
```

```
            break
        if p2.op == ida_hexrays.cot_memref:
            e2 = p2.cexpr
            tif = ida_typeinf.remove_pointer(e2.x.type)
            if not tif.is_union():
                off += e2.m
            p = p2
            continue
        if p2.op == ida_hexrays.cot_ref:
            # handle &a.b + N (this expression may appear if the user previously selecte
            #                  a wrong field)
            delta = 0
            add = vu.cfunc.body.find_parent_of(p2)
            if add.op == ida_hexrays.cot_cast:
                add = vu.cfunc.body.find_parent_of(add)
            if add.op == ida_hexrays.cot_add and add.y.op == ida_hexrays.cot_num:
                delta = add.y.numval()
                objsize = add.type.get_ptrarr_objsize()
                nbytes = delta * objsize
                off += nbytes
        # we could use helpers like WORD/BYTE/... to calculate a more precise offset
        # if ( p2->op == cot_call && (e2->exflags & EXFL_LVALUE) != 0 )
        break
# END REGION3

# REGION4
ea = vu.item.e.ea
# the item itself may be unaddressable.
# TODO: find its addressable parent
if ea == ida_idaapi.BADADDR:
    ida_kernwin.warning("Sorry, the current item is not addressable")
    return 0
# END REGION4

# REGION5
# prepare the text representation for the item,
# use the neighborhoods of cursor
line = ida_lines.tag_remove(ida_kernwin.get_custom_viewer_curline(vu.ct, False))
line_len = len(line)
x = max(0, vu.cpos.x - 10)
l = min(10, line_len - vu.cpos.x) + 10
line = line[x:x+l]
# END REGION5

# REGION6
ops = ida_hexrays.ui_stroff_ops_t()
```

```
            op = ops.push_back()
            op.offset = off
            op.text = line
            # END REGION6

            # REGION7
            class set_union_sel_t(ida_hexrays.ui_stroff_applicator_t):
                def __init__(self, ea):
                    ida_hexrays.ui_stroff_applicator_t.__init__(self)
                    self.ea = ea

                def apply(self, opnum, path, top_tif, spath):
                    typename = ida_typeinf.print_tinfo('', 0, 0, ida_typeinf.PRTYPE_1LINE, top_t
                    ida_kernwin.msg("User selected %s of type %s\n" % (spath, typename))
                    if path.empty():
                        return False
                    vu.cfunc.set_user_union_selection(self.ea, path)
                    vu.cfunc.save_user_unions()
                    return True
            # END REGION7

            # REGION8
            su = set_union_sel_t(ea)
            res = ida_hexrays.select_udt_by_offset(None, ops, su)
            if res != 0:
                # regenerate ctree
                vu.refresh_view(True)
            # END REGION8

            return 1

        def update(self, ctx):
            return ida_kernwin.AST_ENABLE_FOR_WIDGET if \
                ctx.widget_type == ida_kernwin.BWN_PSEUDOCODE else \
                ida_kernwin.AST_DISABLE_FOR_WIDGET


    # --------------------------------------------------------------------------
    # a plugin interface, boilerplate code
    class my_plugin_t(ida_idaapi.plugin_t):
        flags = ida_idaapi.PLUGIN_HIDE
        wanted_name = "Structure offsets (IDAPython)"
        wanted_hotkey = ""
        comment = "Sample plugin17 for Hex-Rays decompiler"
        help = ""
        def init(self):
```

```
        if ida_hexrays.init_hexrays_plugin():
            print("Hex-rays version %s has been detected, Structure offsets ready to use" %
            ida_kernwin.register_action(
                ida_kernwin.action_desc_t(
                    "vds17:strchoose",
                    "Structure offsets",
                    func_stroff_ah_t(),
                    "Shift+T"))
            return ida_idaapi.PLUGIN_KEEP # keep us in the memory
    def term(self):
        pass
    def run(self, arg):
        pass

def PLUGIN_ENTRY():
    return my_plugin_t()

"""
# A few notes about the VDS17 sample

You can find two VDS17 samples in the IDA Pro install directory:

    python/examples/hexrays/vds17.py
    plugins/hexrays_sdk/plugins/vds17

The former is an IDAPython plugin and the latter is a C++ IDA Pro plugin.
Actually they have the same functionality.
Just to be more concrete the vds17.py plugin will be used below.


## How the user interface works

Let us suppose that we have the following local types:

    1 C     struct {int c0;int c1;}
    2 U     union {int u0;__int16 u1;}
    3 D     struct {int d0;C d1;U d2;}
    4 E     struct {int e0;D e1;}
    5 res_t struct {int r0;int r1;__int16 r2;}

and the decompiler generates the following pseudocode:

    void __cdecl f(res_t *r)
    {
      r->r0 = e->e1.d1.c0;
      r->r1 = e->e1.d2.u0;
```

```
  r->r2 = e->e1.d2.u1;
}
```

As we see, it looks good, the decompiler did a really good job.

But let us imagine that we need reference to the ```e1.d2.u0``` union member on the last li

At first we need to load the VDS17 plugin. For that,
place the cursor at the ```u1``` union member on the last line
and use ```Shift-T```.

The "Structure offsets" dialog appears on the screen.
The left pane of the dialog contains the available local types
and the right pane has the following view:

```
[checked] e->e1.d2.u1; | 10h | |
```

Use the left pane to select ```E```, expand it, and select (Double-click) on the ```u0``` m

The right pane will change to:

```
[checked] e->e1.d2.u1; | 10h | [checke] E.e1.d2.u0 |
```

Since this is what we want, press the ```OK``` button and the pseudocode gets changed:

```
void __cdecl f(res_t *r)
{
  r->r0 = e->e1.d1.c0;
  r->r1 = e->e1.d2.u0;
  r->r2 = e->e1.d2.u0;
}
```

What do we need this plugin for?

All union members have the same offset in the parent structure.
The decompiler selects the first suitable union member.
Sometimes we need to change this selection and use another union member.

API details

Let us look into ```python/examples/hexrays/vds17.py```.

* REGION1: check that the cursor points to the union member
* REGION2: calculate the member offset

```
* REGION3: sometimes there is a need to adjust the offset
* REGION4: the pointed item must be addressable

Then the magic begins:

* REGION5: we need something to show in the ```Operand``` column of the right pane.
    The text around the cursor looks like a good compromise.
* REGION6: we are ready to fill the right pane: we have calculated the offset and
    have prepared the descriptive text for the operand.
* REGION7: it is a callback that informs us about the user selection (will be described late
* REGION8: activate the "Structure offsets" dialog, let the user select,
    the callback specified above will update the union member, refresh pseudocode


### The devil is in the details

The main part of callback is the ```apply``` method (REGION7).
It receives two arguments:

1. ```opnum``` is the number of the selected operand (line) in the right pane of the dialog
     We need something to map the line number to our operand.
     In the case of VDS17 ```ea``` performs this role.
2. ```path```  the path that describes the union selection.
3. ```top_tif``` typeinfo of the top-level UDT which user selected
4. ```spath``` the field names path to the selected member

The union selection path is denotes a concrete member inside a UDT.

For structure types there is no need in the union selection path because the member
offset uniquely denotes the desired member.

For unions, on the other hand, the member offset is not enough because all union
members start at the same offset zero. For them, we remember the ordinal number of the
union member in the path. E.g., for the local types given above, the following holds:

* e1.d1.c0 is denoted by an empty path because it does not have any unions
* e1.d2.u0 is denoted by path consisting of 0: we use the first member of U
* e1.d2.u1 is denoted by path consisting of 1: we use the second member of U

You can retrieve the union selection path using API call ```get_user_union_selection```
or apply it using ```set_user_union_selection```.
"""
```

IDAPython example: hexrays/vds19.py

a custom microcode instruction optimization rule (`x | ~x => -1`)

description: Installs a custom microcode instruction optimization rule, to transform:

```
x | ~x
```

into

```
-1
```

To see this plugin in action please use be_ornot_be.idb

```
import ida_hexrays
import ida_idaapi

# recognize "x | ~x" and replace by -1
class subinsn_optimizer_t(ida_hexrays.minsn_visitor_t):
    cnt = 0
    def visit_minsn(self):      # for each instruction...
        ins = self.curins       # take a reference to the current instruction

        # THE CORE OF THE PLUGIN IS HERE:
        # check the pattern "x | ~x"
        if ins.opcode == ida_hexrays.m_or and ins.r.is_insn(ida_hexrays.m_bnot) and ins.l ==
            if not ins.l.has_side_effects(): # avoid destroying side effects
                # pattern matched, convert to "mov -1, ..."
                ins.opcode = ida_hexrays.m_mov
                ins.l.make_number(-1, ins.r.size)
                ins.r = ida_hexrays.mop_t()
                self.cnt = self.cnt + 1 # number of changes we made
        return 0 # continue traversal

# a custom instruction optimizer, boilerplate code
class sample_optimizer_t(ida_hexrays.optinsn_t):
    def func(self, blk, ins, optflags):
        opt = subinsn_optimizer_t()
        ins.for_all_insns(opt)
        if opt.cnt != 0:                # if we modified microcode,
            blk.mba.verify(True)        # run the verifier
        return opt.cnt                  # report the number of changes

# a plugin interface, boilerplate code
class my_plugin_t(ida_idaapi.plugin_t):
    flags = ida_idaapi.PLUGIN_HIDE
    wanted_name = "optimize x|~x"
    wanted_hotkey = ""
    comment = ""
```

```
    help = ""
    def init(self):
        if ida_hexrays.init_hexrays_plugin():
            self.optimizer = sample_optimizer_t()
            self.optimizer.install()
            print("Installed sample optimizer for 'x | ~x'")
            return ida_idaapi.PLUGIN_KEEP # keep us in the memory
    def term(self):
        self.optimizer.remove()
    def run(self, arg):
        if arg == 1:
            return self.optimizer.remove()
        elif arg == 2:
            return self.optimizer.install()

def PLUGIN_ENTRY():
    return my_plugin_t()
```

IDAPython example: hexrays/vds21.py

dynamically provide a custom call type

description: This plugin can greatly improve decompilation of indirect calls:

```
  call    [eax+4]
```

For them, the decompiler has to guess the prototype of the called function. This has to be done at a very early phase of decompilation because the function prototype influences the data flow analysis. On the other hand, we do not have global data flow analysis results yet because we haven't analyzed all calls in the function. It is a chicked-and-egg problem.

The decompiler uses various techniques to guess the called function prototype. While it works very well, it may fail in some cases.

To fix, the user can specify the call prototype manually, using "Edit, Operand types, Set operand type" at the call instruction.

This plugin illustrates another approach to the problem: if you happen to be able to calculate the call prototypes dynamically, this is how to inform the decompiler about them.

```
import ida_idaapi
import ida_nalt
import ida_kernwin
import ida_typeinf
import ida_hexrays
```

```
testing = False # only for testing purposes

class callinfo_provider_t(ida_hexrays.Hexrays_Hooks):

    # this callback will be called for all call instructions
    # our plugin may provide the function prototype or even a complete new callinfo
    # object. The callinfo object may be useful if the prototype is not enough
    # to express all details of the call.
    def build_callinfo(self, blk, type):
        # it is a good idea to skip direct calls.
        # note that some indirect calls may be resolved and become direct calls,
        # and will be filtered out here:
        ida_kernwin.msg("%x: got called for: %s\n" % (blk.tail.ea, blk.tail.dstr()))
        tail = blk.tail
        if tail.opcode == ida_hexrays.m_call:
            return

        # also, if the type was specified by the user, do not interfere
        call_ea = tail.ea
        tif = ida_typeinf.tinfo_t()
        if ida_nalt.get_op_tinfo(tif, call_ea, 0):
            return

        global testing
        if not testing:
            # ok, the decompiler really has to guess the type.
            # just for the sake of an example, return a predefined prototype.
            # in real life you will provide the prototype you discovered yourself,
            # using your magic of yours :)
            my_proto = "int f();"
            ida_kernwin.msg("%x: providing prototype %s\n" % (call_ea, my_proto))
            ida_typeinf.parse_decl(type, None, my_proto, 0)
        else:
            # as an alternative to filling 'type', you can
            # choose to return a mcallinfo_t instance.
            mi = ida_hexrays.mcallinfo_t()
            mi.cc = ida_typeinf.CM_CC_STDCALL | ida_typeinf.CM_N32_F48 # let's use stdcall
            mi.return_type = ida_typeinf.tinfo_t(ida_typeinf.BT_INT)
            mi.return_argloc.set_reg1(0) # eax
            return mi

# a plugin interface, boilerplate code
class my_plugin_t(ida_idaapi.plugin_t):
    flags = ida_idaapi.PLUGIN_HIDE
    wanted_name = "Hex-Rays custom prototype provider (IDAPython)"
```

```
    wanted_hotkey = ""
    comment = "Sample plugin21 for Hex-Rays decompiler"
    help = ""
    def init(self):
        if ida_hexrays.init_hexrays_plugin():
            self.hooks = callinfo_provider_t()
            self.hooks.hook()
            ida_kernwin.warning(
                "Installed callinfo provider sample (vds21.py)\n" +\
                "Please note that it is just an example\n" +\
                "and will spoil your decompilations!")
            return ida_idaapi.PLUGIN_KEEP # keep us in the memory
    def term(self):
        self.hooks.unhook()
    def run(self, arg):
        pass

def PLUGIN_ENTRY():
    return my_plugin_t()
```

IDAPython example: hexrays/vds3.py

invert if/else blocks

description: Registers an action that can be used to invert the `if` and `else` blocks of a `ida_hexrays.cif_t`.

For example, a statement like

```
if ( cond )
{
  statements1;
}
else
{
  statements2;
}
```

will be displayed as

```
if ( !cond )
{
  statements2;
}
else
{
  statements1;
```

```
}
```

The modifications are persistent: the user can quit & restart IDA, and the changes will be present.

author: EiNSTeiN_ (einstein@g3nius.org)

```
import idautils

import ida_kernwin
import ida_hexrays
import ida_netnode
import ida_idaapi
import ida_idp

import traceback

NETNODE_NAME = '$ hexrays-inverted-if'

inverter_actname = "vds3:invert"

class invert_action_handler_t(ida_kernwin.action_handler_t):
    def __init__(self, inverter):
        ida_kernwin.action_handler_t.__init__(self)
        self.inverter = inverter

    def activate(self, ctx):
        vdui = ida_hexrays.get_widget_vdui(ctx.widget)
        self.inverter.invert_if_event(vdui)
        return 1

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_FOR_WIDGET if \
            ctx.widget_type == ida_kernwin.BWN_PSEUDOCODE else \
            ida_kernwin.AST_DISABLE_FOR_WIDGET


class hexrays_callback_info(object):

    def __init__(self):
        self.vu = None

        self.node = ida_netnode.netnode()
        if not self.node.create(NETNODE_NAME):
            # node exists
```

```
            self.load()
        else:
            self.stored = []

        return

    def load(self):

        self.stored = []

        try:
            data = self.node.getblob(0, 'I')
            if data:
                self.stored = eval(data.decode("UTF-8"))
                print('Invert-if: Loaded %s' % (repr(self.stored), ))
        except:
            print('Failed to load invert-if locations')
            traceback.print_exc()
            return

        return

    def save(self):

        try:
            self.node.setblob(repr(self.stored).encode("UTF-8"), 0, 'I')
        except:
            print('Failed to save invert-if locations')
            traceback.print_exc()
            return

        return

    def invert_if(self, insn):

        if insn.opname != 'if':
            return False

        cif = insn.details

        if not cif.ithen or not cif.ielse:
            return False

        ida_hexrays.qswap(cif.ithen, cif.ielse)
        # Make a copy of 'cif.expr': 'lnot' might destroy its toplevel
        # cexpr_t and return a pointer to its direct child (but we'll want to
```

```
        # 'swap' it later, the 'cif.expr' cexpr_t object must remain valid.)
        cond = ida_hexrays.cexpr_t(cif.expr)
        notcond = ida_hexrays.lnot(cond)

        cif.expr.swap(notcond)

        return True

    def add_location(self, ea):
        if ea in self.stored:
            self.stored.remove(ea)
        else:
            self.stored.append(ea)
        self.save()
        return

    def find_if_statement(self, vu):

        vu.get_current_item(ida_hexrays.USE_KEYBOARD)
        item = vu.item

        if item.is_citem() and item.it.op == ida_hexrays.cit_if and item.it.to_specific_type
            return item.it.to_specific_type

        if vu.tail.citype == ida_hexrays.VDI_TAIL and vu.tail.loc.itp == ida_hexrays.ITP_ELS
            # for tail marks, we know only the corresponding ea,
            # not the pointer to if-statement
            # find it by walking the whole ctree
            class if_finder_t(ida_hexrays.ctree_visitor_t):
                def __init__(self, ea):
                    ida_hexrays.ctree_visitor_t.__init__(self, ida_hexrays.CV_FAST | ida_hex

                    self.ea = ea
                    self.found = None
                    return

                def visit_insn(self, i):
                    if i.op == ida_hexrays.cit_if and i.ea == self.ea:
                        self.found = i
                        return 1 # stop enumeration
                    return 0

            iff = if_finder_t(vu.tail.loc.ea)
            if iff.apply_to(vu.cfunc.body, None):
                return iff.found
```

```
            return

        def invert_if_event(self, vu):

            i = self.find_if_statement(vu)
            if not i:
                return False

            if self.invert_if(i):
                vu.refresh_ctext()
                self.add_location(i.ea)

            return True

        def restore(self, cfunc):

            class visitor(ida_hexrays.ctree_visitor_t):

                def __init__(self, inverter, cfunc):
                    ida_hexrays.ctree_visitor_t.__init__(self, ida_hexrays.CV_FAST | ida_hexrays
                    self.inverter = inverter
                    self.cfunc = cfunc
                    return

                def visit_insn(self, i):
                    try:
                        if i.op == ida_hexrays.cit_if and i.ea in self.inverter.stored:
                            self.inverter.invert_if(i)
                    except:
                        traceback.print_exc()
                    return 0 # continue enumeration

            visitor(self, cfunc).apply_to(cfunc.body, None)

            return


    class vds3_hooks_t(ida_hexrays.Hexrays_Hooks):
        def __init__(self, i):
            ida_hexrays.Hexrays_Hooks.__init__(self)
            self.i = i

        def populating_popup(self, widget, phandle, vu):
            ida_kernwin.attach_action_to_popup(vu.ct, None, inverter_actname)
            return 0
```

```
    def maturity(self, cfunc, maturity):
        if maturity == ida_hexrays.CMAT_FINAL:
            self.i.restore(cfunc)
        return 0

class idp_hooks_t(ida_idp.IDP_Hooks):
    def __init__(self, i):
        ida_idp.IDP_Hooks.__init__(self)
        self.i = i

    # 'node' refers to index of the named node, this index became invalid after
    # privrange moving, so we recreate the node here to update nodeidx
    def ev_privrange_changed(self, old_privrange, delta):
        i.node.create(NETNODE_NAME)


# a plugin interface, boilerplate code
class my_plugin_t(ida_idaapi.plugin_t):
    flags = ida_idaapi.PLUGIN_HIDE
    wanted_name = "Hex-Rays if-inverter (IDAPython)"
    wanted_hotkey = ""
    comment = "Sample plugin3 for Hex-Rays decompiler"
    help = ""
    def init(self):
        if ida_hexrays.init_hexrays_plugin():
            i = hexrays_callback_info()
            ida_kernwin.register_action(
                ida_kernwin.action_desc_t(
                    inverter_actname,
                    "Invert then/else",
                    invert_action_handler_t(i),
                    "I"))
            self.vds3_hooks = vds3_hooks_t(i)
            self.vds3_hooks.hook()
            # we need this hook to react to privrange moving event
            self.idp_hooks = idp_hooks_t(i)
            self.idp_hooks.hook()
            return ida_idaapi.PLUGIN_KEEP # keep us in the memory
    def term(self):
        self.vds3_hooks.unhook()
    def run(self, arg):
        pass

def PLUGIN_ENTRY():
    return my_plugin_t()
```

IDAPython example: hexrays/vds4.py

dump user-defined information

description: Prints user-defined information to the "Output" window. Namely:

```
* user defined label names
* user defined indented comments
* user defined number formats
* user defined local variable names, types, comments
```

This script loads information from the database without decompiling anything.

author: EiNSTeiN_ (einstein@g3nius.org)

```
import ida_kernwin
import ida_hexrays
import ida_bytes

def run():
    f = ida_funcs.get_func(ida_kernwin.get_screen_ea());
    if f is None:
        print("Please position the cursor within a function")
        return True
    entry_ea = f.start_ea
    print("Dump of user-defined information for function at %x" % entry_ea)

    # Display user defined labels.
    labels = ida_hexrays.restore_user_labels(entry_ea);
    if labels is not None:
        print("------- %u user defined labels" % len(labels))
        for org_label, name in labels.items():
            print("Label %d: %s" % (org_label, str(name)))
        ida_hexrays.user_labels_free(labels)

    # Display user defined comments
    cmts = ida_hexrays.restore_user_cmts(entry_ea);
    if cmts is not None:
        print("------- %u user defined comments" % (len(cmts), ))
        for tl, cmt in cmts.items():
            print("Comment at %x, preciser %x:\n%s\n" % (tl.ea, tl.itp, str(cmt)))
        ida_hexrays.user_cmts_free(cmts)

    # Display user defined citem iflags
    iflags = ida_hexrays.restore_user_iflags(entry_ea)
    if iflags is not None:
```

```
    print("------- %u user defined citem iflags" % (len(iflags), ))
    for cl, f in iflags.items():
        print("%x(%d): %08X%s" % (cl.ea, cl.op, f, " CIT_COLLAPSED" if f & ida_hexrays.(
    ida_hexrays.user_iflags_free(iflags)

# Display user defined number formats
numforms = ida_hexrays.restore_user_numforms(entry_ea)
if numforms is not None:
    print("------- %u user defined number formats" % (len(numforms), ))
    for ol, nf in numforms.items():
        print("Number format at %a, operand %d: %s" % \
              (ol.ea,
               ol.opnum,
               "negated " if (ord(nf.props) & ida_hexrays.NF_NEGATE) != 0 else ""))

        if nf.is_enum():
            print("enum %s (serial %d)" % (str(nf.type_name), nf.serial))

        elif nf.is_char():
            print("char")

        elif nf.is_stroff():
            print("struct offset %s" % (str(nf.type_name), ))

        else:
            print("number base=%d" % (ida_bytes.get_radix(nf.flags, ol.opnum), ))

    ida_hexrays.user_numforms_free(numforms)

# Display user-defined local variable information
lvinf = ida_hexrays.lvar_uservec_t()
if ida_hexrays.restore_user_lvar_settings(lvinf, entry_ea):
    print("------- User defined local variable information\n")
    for lv in lvinf.lvvec:
        print("Lvar defined at %x" % (lv.ll.defea, ))

        if len(str(lv.name)):
            print("  Name: %s" % (str(lv.name), ))

        if len(str(lv.type)):
            #~ print_type_to_one_line(buf, sizeof(buf), idati, .c_str());
            print("  Type: %s" % (str(lv.type), ))

        if len(str(lv.cmt)):
            print("  Comment: %s" % (str(lv.cmt), ))
```

```
    return


if ida_hexrays.init_hexrays_plugin():
    run()
else:
    print('dump user info: hexrays is not available.')
```

IDAPython example: hexrays/vds5.py

show ctree graph

description: Registers an action that can be used to show the graph of the ctree. The current item will be highlighted in the graph.

The command shortcut is `Ctrl+Shift+G`, and is also added to the context menu.

To display the graph, we produce a .gdl file, and request that ida displays that using `ida_gdl.display_gdl`.

```
import ida_pro
import ida_hexrays
import ida_kernwin
import ida_gdl
import ida_lines
import ida_idaapi

ACTION_NAME = "vds5.py:displaygraph"
ACTION_SHORTCUT = "Ctrl+Shift+G"

CL_WHITE            = ((255)+  (255<<8)+  (255<<16)) #   0
CL_BLUE             = ((0  )+  (0  <<8)+  (255<<16)) #   1
CL_RED              = ((255)+  (0  <<8)+  (0  <<16)) #   2
CL_GREEN            = ((0  )+  (255<<8)+  (0  <<16)) #   3
CL_YELLOW           = ((255)+  (255<<8)+  (0  <<16)) #   4
CL_MAGENTA          = ((255)+  (0  <<8)+  (255<<16)) #   5
CL_CYAN             = ((0  )+  (255<<8)+  (255<<16)) #   6
CL_DARKGREY         = ((85 )+  (85 <<8)+  (85 <<16)) #   7
CL_DARKBLUE         = ((0  )+  (0  <<8)+  (128<<16)) #   8
CL_DARKRED          = ((128)+  (0  <<8)+  (0  <<16)) #   9
CL_DARKGREEN        = ((0  )+  (128<<8)+  (0  <<16)) #  10
CL_DARKYELLOW       = ((128)+  (128<<8)+  (0  <<16)) #  11
CL_DARKMAGENTA      = ((128)+  (0  <<8)+  (128<<16)) #  12
CL_DARKCYAN         = ((0  )+  (128<<8)+  (128<<16)) #  13
CL_GOLD             = ((255)+  (215<<8)+  (0  <<16)) #  14
```

```
CL_LIGHTGREY        = ((170)+  (170<<8)+  (170<<16)) #  15
CL_LIGHTBLUE        = ((128)+  (128<<8)+  (255<<16)) #  16
CL_LIGHTRED         = ((255)+  (128<<8)+  (128<<16)) #  17
CL_LIGHTGREEN       = ((128)+  (255<<8)+  (128<<16)) #  18
CL_LIGHTYELLOW      = ((255)+  (255<<8)+  (128<<16)) #  19
CL_LIGHTMAGENTA     = ((255)+  (128<<8)+  (255<<16)) #  20
CL_LIGHTCYAN        = ((128)+  (255<<8)+  (255<<16)) #  21
CL_LILAC            = ((238)+  (130<<8)+  (238<<16)) #  22
CL_TURQUOISE        = ((64 )+  (224<<8)+  (208<<16)) #  23
CL_AQUAMARINE       = ((127)+  (255<<8)+  (212<<16)) #  24
CL_KHAKI            = ((240)+  (230<<8)+  (140<<16)) #  25
CL_PURPLE           = ((160)+  (32 <<8)+  (240<<16)) #  26
CL_YELLOWGREEN      = ((154)+  (205<<8)+  (50 <<16)) #  27
CL_PINK             = ((255)+  (192<<8)+  (203<<16)) #  28
CL_ORANGE           = ((255)+  (165<<8)+  (0  <<16)) #  29
CL_ORCHID           = ((218)+  (112<<8)+  (214<<16)) #  30
CL_BLACK            = ((0  )+  (0  <<8)+  (0  <<16)) #  31

COLORS_LUT = {
    CL_WHITE            : "white",
    CL_BLUE             : "blue",
    CL_RED              : "red",
    CL_GREEN            : "green",
    CL_YELLOW           : "yellow",
    CL_MAGENTA          : "magenta",
    CL_CYAN             : "cyan",
    CL_DARKGREY         : "darkgrey",
    CL_DARKBLUE         : "darkblue",
    CL_DARKRED          : "darkred",
    CL_DARKGREEN        : "darkgreen",
    CL_DARKYELLOW       : "darkyellow",
    CL_DARKMAGENTA      : "darkmagenta",
    CL_DARKCYAN         : "darkcyan",
    CL_GOLD             : "gold",
    CL_LIGHTGREY        : "lightgrey",
    CL_LIGHTBLUE        : "lightblue",
    CL_LIGHTRED         : "lightred",
    CL_LIGHTGREEN       : "lightgreen",
    CL_LIGHTYELLOW      : "lightyellow",
    CL_LIGHTMAGENTA     : "lightmagenta",
    CL_LIGHTCYAN        : "lightcyan",
    CL_LILAC            : "lilac",
    CL_TURQUOISE        : "turquoise",
    CL_AQUAMARINE       : "aquamarine",
    CL_KHAKI            : "khaki",
    CL_PURPLE           : "purple",
```

```
    CL_YELLOWGREEN       : "yellowgreen",
    CL_PINK              : "pink",
    CL_ORANGE            : "orange",
    CL_ORCHID            : "orchid",
    CL_BLACK             : "black",
}

def get_color_name(c):
    return COLORS_LUT[c] if c in COLORS_LUT.keys() else "?"

class cfunc_graph_t: # alas we can't inherit gdl_graph_t
    def __init__(self, highlight):
        self.items = [] # list of citem_t
        self.highlight = highlight
        self.succs = [] # list of lists of next nodes
        self.preds = [] # list of lists of previous nodes

    def nsucc(self, n):
        return len(self.succs[n]) if self.size() else 0

    def npred(self, n):
        return len(self.preds[n]) if self.size() else 0

    def succ(self, n, i):
        return self.succs[n][i]

    def pred(self, n, i):
        return self.preds[n][i]

    def size(self):
        return len(self.preds)

    def add_node(self):
        n = self.size()

        def resize(array, new_size):
            if new_size > len(array):
                while len(array) < new_size:
                    array.append([])
            else:
                array = array[:new_size]
            return array

        self.preds = resize(self.preds, n+1)
        self.succs = resize(self.succs, n+1)
        return n
```

```
def add_edge(self, x, y):
    self.preds[y].append(x)
    self.succs[x].append(y)

def get_expr_name(self, expr):
    name = expr.print1(None)
    name = ida_lines.tag_remove(name)
    name = ida_pro.str2user(name)
    return name

def get_node_label(self, n):
    item = self.items[n]
    op = item.op
    insn = item.cinsn
    expr = item.cexpr
    parts = [ida_hexrays.get_ctype_name(op)]
    if op == ida_hexrays.cot_ptr:
        parts.append(".%d" % expr.ptrsize)
    elif op == ida_hexrays.cot_memptr:
        parts.append(".%d (m=%d)" % (expr.ptrsize, expr.m))
    elif op == ida_hexrays.cot_memref:
        parts.append(" (m=%d)" % (expr.m,))
    elif op in [
            ida_hexrays.cot_obj,
            ida_hexrays.cot_var]:
        name = self.get_expr_name(expr)
        parts.append(".%d %s" % (expr.refwidth, name))
    elif op in [
            ida_hexrays.cot_num,
            ida_hexrays.cot_helper,
            ida_hexrays.cot_str]:
        name = self.get_expr_name(expr)
        parts.append(" %s" % (name,))
    elif op == ida_hexrays.cit_goto:
        parts.append(" LABEL_%d" % insn.cgoto.label_num)
    elif op == ida_hexrays.cit_asm:
        parts.append("<asm statements; unsupported ATM>")
        # parts.append(" %a.%d" % ())
    parts.append("\\n")
    parts.append("ea: %08X" % item.ea)
    if item.is_expr() and not expr.type.empty():
        parts.append("\\n")
        tstr = expr.type._print()
        parts.append(tstr if tstr else "?")
    return "".join(parts)
```

```
def get_node_color(self, n):
    item = self.items[n]
    if self.highlight is not None and item.obj_id == self.highlight.obj_id:
        return CL_GREEN
    return None

def gen_gdl(self, fname):
    with open(fname, "wb") as out:
        def write_out(s):
            out.write(s.encode("UTF-8"))
        write_out("graph: {\n")

        write_out("// *** nodes\n")
        for n in range(len(self.items)):
            item = self.items[n]
            node_label = self.get_node_label(n)
            node_props = [""]
            if n == 0:
                node_props.append("vertical_order: 0")
            color = self.get_node_color(n)
            if color is not None:
                node_props.append("color: %s" % get_color_name(color))
            write_out("""node: { title: "%d" label: "%d: %s" %s}\n""" % (
                n,
                n,
                node_label,
                " ".join(node_props)))

        write_out("// *** edges\n")
        for n in range(len(self.items)):
            item = self.items[n]
            write_out("// edges %d -> ?\n" % n)
            for i in range(self.nsucc(n)):
                t = self.succ(n, i)
                label = ""
                if item.is_expr():
                    target = self.items[t]
                    if item.x is not None and item.x == target:
                        label = "x"
                    elif item.y is not None and item.y == target:
                        label = "y"
                    elif item.z is not None and item.z == target:
                        label = "z"
                    if label:
                        label = """ label: "%s" """ % label
```

```
                    write_out("""edge: { sourcename: "%s" targetname: "%s"%s}\n""" % (
                        str(n), str(t), label))

            write_out("}\n")

    def dump(self):
        print("%d items:" % len(self.items))
        for i in self.items:
            print("\t%s (%08x)" % (i, i.ea))

        print("succs:")
        for s in self.succs:
            print("\t%s" % s)

        print("preds:")
        for p in self.preds:
            print("\t%s" % p)


class graph_builder_t(ida_hexrays.ctree_parentee_t):

    def __init__(self, cg):
        ida_hexrays.ctree_parentee_t.__init__(self)
        self.cg = cg
        self.reverse = [] # (citem_t, node#) tuples

    def add_node(self, i):
        for k, _ in self.reverse:
            if i.obj_id == k.obj_id:
                ida_kernwin.warning("bad ctree - duplicate nodes! (i.ea=%x)" % i.ea)
                self.cg.dump()
                return -1

        n = self.cg.add_node()
        if n <= len(self.cg.items):
            self.cg.items.append(i)
        self.cg.items[n] = i
        self.reverse.append((i, n))
        return n

    def process(self, i):
        n = self.add_node(i)
        if n < 0:
            return n
        if len(self.parents) > 1:
            lp = self.parents.back().obj_id
```

```
            for k, v in self.reverse:
                if k.obj_id == lp:
                    p = v
                    break
            self.cg.add_edge(p, n)
        return 0

    def visit_insn(self, i):
        return self.process(i)

    def visit_expr(self, e):
        return self.process(e)


class display_graph_ah_t(ida_kernwin.action_handler_t):
    def __init__(self):
        ida_kernwin.action_handler_t.__init__(self)

    def activate(self, ctx):
        vu = ida_hexrays.get_widget_vdui(ctx.widget)
        vu.get_current_item(ida_hexrays.USE_KEYBOARD)
        highlight = vu.item.e if vu.item.is_citem() else None

        cg = cfunc_graph_t(highlight)
        gb = graph_builder_t(cg)
        gb.apply_to(vu.cfunc.body, None)

        import tempfile
        fname = tempfile.mktemp(suffix=".gdl")
        cg.gen_gdl(fname)
        ida_gdl.display_gdl(fname)
        return 1

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_FOR_WIDGET if \
            ctx.widget_type == ida_kernwin.BWN_PSEUDOCODE else \
            ida_kernwin.AST_DISABLE_FOR_WIDGET


class vds5_hooks_t(ida_hexrays.Hexrays_Hooks):
    def populating_popup(self, widget, handle, vu):
        ida_kernwin.attach_action_to_popup(vu.ct, None, ACTION_NAME)
        return 0

# a plugin interface, boilerplate code
class my_plugin_t(ida_idaapi.plugin_t):
```

```
    flags = ida_idaapi.PLUGIN_HIDE
    wanted_name = "Hex-Rays show C graph (IDAPython)"
    wanted_hotkey = ""
    comment = "Sample plugin5 for Hex-Rays decompiler"
    help = ""
    def init(self):
        if ida_hexrays.init_hexrays_plugin():
            ida_kernwin.register_action(
                ida_kernwin.action_desc_t(
                    ACTION_NAME,
                    "Hex-Rays show C graph (IDAPython)",
                    display_graph_ah_t(),
                    ACTION_SHORTCUT))
            self.vds5_hooks = vds5_hooks_t()
            self.vds5_hooks.hook()
            return ida_idaapi.PLUGIN_KEEP # keep us in the memory
    def term(self):
        self.vds5_hooks.unhook()
    def run(self, arg):
        pass

def PLUGIN_ENTRY():
    return my_plugin_t()
```

IDAPython example: hexrays/vds6.py

superficially modify the decompilation output

description: modifies the decompilation output in a superficial manner, by removing some white spaces

Note: this is rather crude, not quite “pythonic” code.

```
import idautils
import idc
import ida_idaapi
import ida_hexrays
import ida_lines

def dbg(msg):
    #print(msg)
    pass

def is_cident_char(c):
    return c.isalnum() or c == '_'
```

```
def my_tag_skipcodes(l, storage):
    n = ida_lines.tag_skipcodes(l)
    dbg("Skipping %d chars ('%s')" % (n, l[0:n]))
    storage.append(l[0:n])
    return l[n:]

def remove_spaces(sl):
    dbg("*" * 80)
    l = sl.line
    out = []

    def push(c):
        dbg("Appending '%s'" % c)
        out.append(c)

    # skip initial spaces, do not compress them
    while True:
        l = my_tag_skipcodes(l, out)
        if not l:
            break
        c = l[0]
        if not c.isspace():
            break
        push(c)
        l = l[1:]

    # remove all spaces except in string and char constants
    delim = None # if not None, then we are skipping until 'delim'
    last = None # last seen character
    while True:
        # go until comments
        if l.startswith("//"):
            push(l)
            break
        dbg("-" * 60)
        nchars = ida_lines.tag_advance(l, 1)
        push(l[0:nchars])
        l = l[nchars:]
        l = my_tag_skipcodes(l, out)
        if not l:
            break
        c = l[0]
        dbg("c: '%s', last: '%s', l: '%s'" % (c, last, l))
        if delim:
            # we're inside a literal.
```

```
            if c == delim:
                delim = None # literal ended
        elif c == '"' or c == "'":
            delim = c # string/char literal started
        elif c.isspace():
            end = l.lstrip()
            nptr = my_tag_skipcodes(end, out)
            dbg("end: '%s', nptr: '%s'" % (end, nptr))
            # do not concatenate idents
            if not is_cident_char(last) or not is_cident_char(nptr[0]):
                l = end
                c = l[0] if l else ''
                dbg("new l: '%s'" % l)
        last = l[0] if l else ''

    sl.line = "".join(out)


class vds6_hooks_t(ida_hexrays.Hexrays_Hooks):
    def func_printed(self, cfunc):
        for sl in cfunc.get_pseudocode():
            remove_spaces(sl);
        return 0

# a plugin interface, boilerplate code
class my_plugin_t(ida_idaapi.plugin_t):
    flags = ida_idaapi.PLUGIN_HIDE
    wanted_name = "Hex-Rays space remover (IDAPython)"
    wanted_hotkey = ""
    comment = "Sample plugin6 for Hex-Rays decompiler"
    help = ""
    def init(self):
        if ida_hexrays.init_hexrays_plugin():
            self.vds6_hooks = vds6_hooks_t()
            self.vds6_hooks.hook()
            return ida_idaapi.PLUGIN_KEEP # keep us in the memory
    def term(self):
        self.vds6_hooks.unhook()
    def run(self, arg):
        pass

def PLUGIN_ENTRY():
    return my_plugin_t()
```

IDAPython example: hexrays/vds7.py

iterate a cblock_t object

description: Using a `ida_hexrays.ctree_visitor_t`, search for `ida_hexrays.cit_block` instances and dump them.

author: EiNSTeiN_ (einstein@g3nius.org)

```
import ida_hexrays

class cblock_visitor_t(ida_hexrays.ctree_visitor_t):

    def __init__(self):
        ida_hexrays.ctree_visitor_t.__init__(self, ida_hexrays.CV_FAST)

    def visit_insn(self, ins):
        if ins.op == ida_hexrays.cit_block:
            self.dump_block(ins.ea, ins.cblock)
        return 0

    def dump_block(self, ea, b):
        # iterate over all block instructions
        print("dumping block %x" % (ea, ))
        for ins in b:
            print("  %x: insn %s" % (ins.ea, ins.opname))


class vds7_hooks_t(ida_hexrays.Hexrays_Hooks):
    def maturity(self, cfunc, maturity):
        if maturity == ida_hexrays.CMAT_BUILT:
            cbv = cblock_visitor_t()
            cbv.apply_to(cfunc.body, None)
        return 0


if ida_hexrays.init_hexrays_plugin():
    vds7_hooks = vds7_hooks_t()
    vds7_hooks.hook()
else:
    print('cblock visitor: hexrays is not available.')
```

IDAPython example: hexrays/vds8.py

using `ida_hexrays.udc_filter_t`

description: Registers an action that uses a `ida_hexrays.udc_filter_t` to decompile `svc 0x900001` and `svc 0x9000F8` as function calls to `svc_exit()` and `svc_exit_group()` respectively.

You will need to have an ARM + Linux IDB for this script to be usable

In addition to having a shortcut, the action will be present in the context menu.

```
import ida_idaapi
import ida_hexrays
import ida_kernwin
import ida_allins

ACTION_NAME = "vds8.py:udcall"
ACTION_SHORTCUT = "Ctrl+Shift+U"

# --------------------------------------------------------------------------
class udc_exit_t(ida_hexrays.udc_filter_t):
    def __init__(self, code, name):
        ida_hexrays.udc_filter_t.__init__(self)
        if not self.init("int __usercall %s@<R0>(int status@<R1>);" % name):
            raise Exception("Couldn't initialize udc_exit_t instance")
        self.code = code
        self.installed = False

    def match(self, cdg):
        return cdg.insn.itype == ida_allins.ARM_svc and cdg.insn.Op1.value == self.code

    def install(self):
        ida_hexrays.install_microcode_filter(self, True);
        self.installed = True

    def uninstall(self):
        ida_hexrays.install_microcode_filter(self, False);
        self.installed = False

    def toggle_install(self):
        if self.installed:
            self.uninstall()
        else:
            self.install()


# --------------------------------------------------------------------------
class toggle_udc_ah_t(ida_kernwin.action_handler_t):
```

```
    def __init__(self):
        ida_kernwin.action_handler_t.__init__(self)

    def activate(self, ctx):
        udc_exit.toggle_install();
        udc_exit_group.toggle_install();
        vu = ida_hexrays.get_widget_vdui(ctx.widget)
        vu.refresh_view(True)
        return 1

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_FOR_WIDGET if \
            ctx.widget_type == ida_kernwin.BWN_PSEUDOCODE else \
            ida_kernwin.AST_DISABLE_FOR_WIDGET


# --------------------------------------------------------------------------
class my_hooks_t(ida_kernwin.UI_Hooks):
    def populating_widget_popup(self, widget, popup):
        if ida_kernwin.get_widget_type(widget) == ida_kernwin.BWN_PSEUDOCODE:
            ida_kernwin.attach_action_to_popup(widget, popup, ACTION_NAME)
my_hooks = my_hooks_t()
my_hooks.hook()


# --------------------------------------------------------------------------
SVC_EXIT       = 0x900001
SVC_EXIT_GROUP = 0x9000f8

if ida_hexrays.init_hexrays_plugin():
    udc_exit = udc_exit_t(SVC_EXIT, "svc_exit")
    udc_exit.toggle_install()

    udc_exit_group = udc_exit_t(SVC_EXIT_GROUP, "svc_exit_group")
    udc_exit_group.toggle_install()

    ida_kernwin.register_action(
        ida_kernwin.action_desc_t(
            ACTION_NAME,
            "vds8.py:Toggle UDC",
            toggle_udc_ah_t(),
            ACTION_SHORTCUT))
```

IDAPython example: hexrays/vds_create_hint.py

decompiler hints

description: Handle `ida_hexrays.hxe_create_hint` notification using hooks, to return our own.

If the object under the cursor is:

- a function call, prefix the original decompiler hint with `==>`
- a local variable declaration, replace the hint with our own in the form of `!{varname}` (where `{varname}` is replaced with the variable name)
- an `if` statement, replace the hint with our own, saying "condition"

```
import ida_idaapi
import ida_hexrays

class hint_hooks_t(ida_hexrays.Hexrays_Hooks):
    def create_hint(self, vu):
        if vu.get_current_item(ida_hexrays.USE_MOUSE):
            cit = vu.item.citype
            if cit == ida_hexrays.VDI_LVAR:
                return 1, "!%s" % vu.item.l.name, 1
            elif cit == ida_hexrays.VDI_EXPR:
                ce = vu.item.e
                if ce.op == ida_hexrays.cot_call:
                    return 0, "==> ", 1
                if ce.op == ida_hexrays.cit_if:
                    return 1, "condition", 1
        return 0

vds_hooks = hint_hooks_t()
vds_hooks.hook()
```

IDAPython example: hexrays/vds_hooks.py

various decompiler hooks

description: Shows how to hook to many notifications sent by the decompiler.

This plugin doesn't really accomplish anything: it just prints the parameters.

The list of notifications handled below should be exhaustive, and is there to hint at what is possible to accomplish by subclassing `ida_hexrays.Hexrays_Hooks`

see_also: curpos_details

```
import inspect

import ida_idaapi
import ida_typeinf
import ida_hexrays

class vds_hooks_t(ida_hexrays.Hexrays_Hooks):
    def __init__(self):
        ida_hexrays.Hexrays_Hooks.__init__(self)
        self.display_shortened_cfuncs = False
        self.display_vdui_curpos = False
        self.inhibit_log = 0;

    def _format_lvar(self, v):
        parts = []
        if v:
            if v.name:
                parts.append("name=%s" % v.name)
            if v.cmt:
                parts.append("cmt=%s" % v.cmt)
            parts.append("width=%s" % v.width)
            parts.append("defblk=%s" % v.defblk)
            parts.append("divisor=%s" % v.divisor)
        return "{%s}" % ", ".join(parts)

    def _format_vdui_curpos(self, v):
        return "cpos={lnnum=%d, x=%d, y=%d}" % (v.cpos.lnnum, v.cpos.x, v.cpos.y)

    def _format_value(self, v):
        if isinstance(v, ida_hexrays.lvar_t):
            v = self._format_lvar(v)
        elif isinstance(v, ida_hexrays.cfunc_t):
            if self.display_shortened_cfuncs:
                self.inhibit_log += 1
                v = str(v)
                if len(v) > 20:
                    v = v[0:20] + "[...snipped...]"
                self.inhibit_log -= 1
            else:
                v = "<cfunc>" # cannot print contents: we'll end up being called recursivel
        elif isinstance(v, ida_hexrays.vdui_t) and self.display_vdui_curpos:
            v = str(v) + " " + self._format_vdui_curpos(v)
        return str(v)

    def _log(self):
        if self.inhibit_log <= 0:
```

```
            stack = inspect.stack()
            frame, _, _, _, _, _ = stack[1]
            args, _, _, values = inspect.getargvalues(frame)
            method_name = inspect.getframeinfo(frame)[2]
            argstrs = []
            for arg in args[1:]:
                argstrs.append("%s=%s" % (arg, self._format_value(values[arg])))
            print("### %s: %s" % (method_name, ", ".join(argstrs)))
        return 0

    def flowchart(self, fc):
        return self._log()

    def stkpnts(self, mba, stkpnts):
        return self._log()

    def prolog(self, mba, fc, reachable_blocks, decomp_flags):
        return self._log()

    def microcode(self, mba):
        return self._log()

    def preoptimized(self, mba):
        return self._log()

    def locopt(self, mba):
        return self._log()

    def prealloc(self, mba):
        return self._log()

    def glbopt(self, mba):
        return self._log()

    def structural(self, ctrl_graph):
        return self._log()

    def maturity(self, cfunc, maturity):
        return self._log()

    def interr(self, code):
        return self._log()

    def combine(self, blk, insn):
        return self._log()
```

```
def print_func(self, cfunc, printer):
    return self._log()

def func_printed(self, cfunc):
    return self._log()

def resolve_stkaddrs(self, mba):
    return self._log()

def open_pseudocode(self, vu):
    return self._log()

def switch_pseudocode(self, vu):
    return self._log()

def refresh_pseudocode(self, vu):
    return self._log()

def close_pseudocode(self, vu):
    return self._log()

def keyboard(self, vu, key_code, shift_state):
    return self._log()

def right_click(self, vu):
    return self._log()

def double_click(self, vu, shift_state):
    return self._log()

def curpos(self, vu):
    return self._log()

def create_hint(self, vu):
    return self._log()

def text_ready(self, vu):
    return self._log()

def populating_popup(self, widget, popup, vu):
    return self._log()

def lvar_name_changed(self, vu, v, name, is_user_name):
    return self._log()

def lvar_type_changed(self, vu, v, tif):
```

```
        return self._log()

    def lvar_cmt_changed(self, vu, v, cmt):
        return self._log()

    def lvar_mapping_changed(self, vu, _from, to):
        return self._log()

    def cmt_changed(self, cfunc, loc, cmt):
        return self._log()

    def build_callinfo(self, *args):
        return self._log()

vds_hooks = vds_hooks_t()
vds_hooks.hook()
```

IDAPython example: hexrays/vds_modify_user_lvars.py

modifying local variables

description: Use a `ida_hexrays.user_lvar_modifier_t` to modify names, comments and/or types of local variables.

```
import ida_hexrays
import ida_typeinf

import idc

class my_modifier_t(ida_hexrays.user_lvar_modifier_t):
    def __init__(self, name_prefix="", cmt_prefix="", new_types={}):
        ida_hexrays.user_lvar_modifier_t.__init__(self)
        self.name_prefix = name_prefix
        self.cmt_prefix = cmt_prefix
        self.new_types = new_types

    def modify_lvars(self, lvars):
        def log(msg):
            print("modify_lvars: %s" % msg)
        log("len(lvars.lvvec) = %d" % len(lvars.lvvec))
        log("lvars.lmaps.size() = %d" % lvars.lmaps.size())
        log("lvars.stkoff_delta = %d" % lvars.stkoff_delta)
        log("lvars.ulv_flags = %x" % lvars.ulv_flags)

        for idx, one in enumerate(lvars.lvvec):
```

```
            def varlog(msg):
                log("var #%d: %s" % (idx, msg))
            varlog("name = '%s'" % one.name)
            varlog("type = '%s'" % one.type._print())
            varlog("cmt = '%s'" % one.cmt)
            varlog("size = %d" % one.size)
            varlog("flags = %x" % one.flags)
            new_type = self.new_types.get(one.name)
            if new_type:
                tif = ida_typeinf.tinfo_t()
                ida_typeinf.parse_decl(tif, None, new_type, 0)
                one.type = tif
            one.name = self.name_prefix + one.name
            one.cmt = self.cmt_prefix + one.cmt

        return True


def modify_function_lvars(name_prefix="patched_", cmt_prefix="(patched) ", new_types={}):
    ea = idc.here()
    my_mod = my_modifier_t(
        name_prefix=name_prefix,
        cmt_prefix=cmt_prefix,
        new_types=new_types)
    ida_hexrays.modify_user_lvars(ea, my_mod)
```

IDAPython example: hexrays/vds_xrefs.py

show decompiler xrefs

description: Show decompiler-style Xref when the `Ctrl+X` key is pressed in the Decompiler window.

- supports any global name: functions, strings, integers, ...
- supports structure member.

author: EiNSTeiN_ (einstein@g3nius.org)

```
import ida_kernwin
import ida_hexrays
import ida_typeinf
import ida_idaapi
import ida_struct
import ida_funcs

import idautils
```

```
import traceback

from PyQt5 import QtCore, QtWidgets

XREF_EA = 0
XREF_STRUC_MEMBER = 1

class XrefsForm(ida_kernwin.PluginForm):

    def __init__(self, target):

        ida_kernwin.PluginForm.__init__(self)

        self.target = target

        if type(self.target) == ida_hexrays.cfunc_t:

            self.__type = XREF_EA
            self.__ea = self.target.entry_ea
            self.__name = 'Xrefs of %x' % (self.__ea, )

        elif type(self.target) == ida_hexrays.cexpr_t and self.target.opname == 'obj':

            self.__type = XREF_EA
            self.__ea = self.target.obj_ea
            self.__name = 'Xrefs of %x' % (self.__ea, )

        elif type(self.target) == ida_hexrays.cexpr_t and self.target.opname in ('memptr',

            self.__type = XREF_STRUC_MEMBER
            name = self.get_struc_name()
            self.__name = 'Xrefs of %s' % (name, )

        else:
            raise ValueError('cannot show xrefs for this kind of target')

        return

    def get_struc_name(self):

        x = self.target.operands['x']
        m = self.target.operands['m']

        xtype = x.type
        xtype.remove_ptr_or_array()
```

```
        typename = ida_typeinf.print_tinfo('', 0, 0, ida_typeinf.PRTYPE_1LINE, xtype, '', ''

        sid = ida_struct.get_struc_id(typename)
        sptr = ida_struct.get_struc(sid)
        member = ida_struct.get_member(sptr, m)

        return '%s::%s' % (typename, member)

    def OnCreate(self, widget):

        # Get parent widget
        self.parent = self.FormToPyQtWidget(widget)

        self.populate_form()

        return

    def Show(self):
        ida_kernwin.PluginForm.Show(self, self.__name)
        return

    def populate_form(self):
        # Create layout
        layout = QtWidgets.QVBoxLayout()

        layout.addWidget(QtWidgets.QLabel(self.__name))
        self.table = QtWidgets.QTableWidget()
        layout.addWidget(self.table)

        self.table.setColumnCount(3)
        self.table.setHorizontalHeaderItem(0, QtWidgets.QTableWidgetItem("Address"))
        self.table.setHorizontalHeaderItem(1, QtWidgets.QTableWidgetItem("Function"))
        self.table.setHorizontalHeaderItem(2, QtWidgets.QTableWidgetItem("Line"))

        self.table.setColumnWidth(0, 80)
        self.table.setColumnWidth(1, 150)
        self.table.setColumnWidth(2, 450)

        self.table.cellDoubleClicked.connect(self.double_clicked)

        #~ self.table.setSelectionMode(QtWidgets.QAbstractItemView.NoSelection)
        self.table.setSelectionBehavior(QtWidgets.QAbstractItemView.SelectRows )
        self.parent.setLayout(layout)

        self.populate_table()
```

```
        return

    def double_clicked(self, row, column):

        ea = self.functions[row]
        ida_hexrays.open_pseudocode(ea, True)

        return

    def get_decompiled_line(self, cfunc, ea):

        print(repr(ea))
        if ea not in cfunc.eamap:
            print('strange, %x is not in %x eamap' % (ea, cfunc.entry_ea))
            return

        insnvec = cfunc.eamap[ea]

        lines = []
        for stmt in insnvec:

            qp = ida_hexrays.qstring_printer_t(cfunc, False)

            stmt._print(0, qp)
            s = qp.s.split('\n')[0]

            #~ s = ida_lines.tag_remove(s)
            lines.append(s)

        return '\n'.join(lines)

    def get_items_for_ea(self, ea):

        frm = [x.frm for x in idautils.XrefsTo(self.__ea)]

        items = []
        for ea in frm:
            cfunc = ida_hexrays.decompile(ea)
            if not cfunc:
                print('Decompilation of %x failed' % (ea, ))
                continue

            self.functions.append(cfunc.entry_ea)
            self.items.append((ea, ida_funcs.get_func_name(cfunc.entry_ea) or "", self.get_c
    def get_items_for_type(self):
```

```
x = self.target.operands['x']
m = self.target.operands['m']

xtype = x.type
xtype.remove_ptr_or_array()
typename = ida_typeinf.print_tinfo('', 0, 0, ida_typeinf.PRTYPE_1LINE, xtype, '', ''

addresses = []
for ea in idautils.Functions():

    cfunc = ida_hexrays.decompile(ea, flags=ida_hexrays.DECOMP_GXREFS_FORCE)
    if not cfunc:
        print('Decompilation of %x failed' % (ea, ))
        continue

    for citem in cfunc.treeitems:
        citem = citem.to_specific_type
        if not (type(citem) == ida_hexrays.cexpr_t and citem.opname in ('memptr', 'm
            continue

        _x = citem.operands['x']
        _m = citem.operands['m']
        _xtype = _x.type
        _xtype.remove_ptr_or_array()
        _typename = ida_typeinf.print_tinfo('', 0, 0, ida_typeinf.PRTYPE_1LINE, _xty

        if not (_typename == typename and _m == m):
            continue

        parent = citem
        while parent:
            if type(parent.to_specific_type) == ida_hexrays.cinsn_t:
                break
            parent = cfunc.body.find_parent_of(parent)

        if not parent:
            print('cannot find parent statement (?!)')
            continue

        if parent.ea in addresses:
            continue

        if parent.ea == ida_idaapi.BADADDR:
            print('parent.ea is BADADDR')
            continue
```

```
                addresses.append(parent.ea)

                self.functions.append(cfunc.entry_ea)
                self.items.append((
                        parent.ea,
                        ida_funcs.get_func_name(cfunc.entry_ea) or "",
                        self.get_decompiled_line(cfunc, parent.ea)))


        return []

    def populate_table(self):

        self.functions = []
        self.items = []

        if self.__type == XREF_EA:
            self.get_items_for_ea(self.__ea)
        else:
            self.get_items_for_type()

        self.table.setRowCount(len(self.items))

        i = 0
        for item in self.items:
            address, func, line = item
            item = QtWidgets.QTableWidgetItem('0x%x' % (address, ))
            item.setFlags(item.flags() ^ QtCore.Qt.ItemIsEditable)
            self.table.setItem(i, 0, item)
            item = QtWidgets.QTableWidgetItem(func)
            item.setFlags(item.flags() ^ QtCore.Qt.ItemIsEditable)
            self.table.setItem(i, 1, item)
            item = QtWidgets.QTableWidgetItem(line)
            item.setFlags(item.flags() ^ QtCore.Qt.ItemIsEditable)
            self.table.setItem(i, 2, item)

            i += 1

        self.table.resizeRowsToContents()

        return

    def OnClose(self, widget):
        pass
```

```
class show_xrefs_ah_t(ida_kernwin.action_handler_t):
    def __init__(self):
        ida_kernwin.action_handler_t.__init__(self)
        self.sel = None

    def activate(self, ctx):
        vu = ida_hexrays.get_widget_vdui(ctx.widget)
        if not vu or not self.sel:
            print("No vdui? Strange, since this action should be enabled only for pseudocode
            return 0

        form = XrefsForm(self.sel)
        form.Show()
        return 1

    def update(self, ctx):
        if ctx.widget_type != ida_kernwin.BWN_PSEUDOCODE:
            return ida_kernwin.AST_DISABLE_FOR_WIDGET
        vu = ida_hexrays.get_widget_vdui(ctx.widget)
        vu.get_current_item(ida_hexrays.USE_KEYBOARD)
        item = vu.item
        self.sel = None
        if item.citype == ida_hexrays.VDI_EXPR and item.it.to_specific_type.opname in ('obj'
            # if an expression is selected. verify that it's either a cot_obj, cot_memref o
            self.sel = item.it.to_specific_type

        elif item.citype == ida_hexrays.VDI_FUNC:
            # if the function itself is selected, show xrefs to it.
            self.sel = item.f

        return ida_kernwin.AST_ENABLE if self.sel else ida_kernwin.AST_DISABLE


class vds_xrefs_hooks_t(ida_hexrays.Hexrays_Hooks):
    def populating_popup(self, widget, phandle, vu):
        ida_kernwin.attach_action_to_popup(widget, phandle, "vdsxrefs:show", None)
        return 0


if ida_hexrays.init_hexrays_plugin():
    adesc = ida_kernwin.action_desc_t('vdsxrefs:show', 'Show xrefs', show_xrefs_ah_t(), "Ctr
    if ida_kernwin.register_action(adesc):
        vds_xrefs_hooks = vds_xrefs_hooks_t()
        vds_xrefs_hooks.hook()
    else:
        print("Couldn't register action.")
```

```
else:
    print('hexrays is not available.')
```

IDAPython example: idbhooks/log_idb_events.py

logging IDB events

description: these hooks will be notified about IDB events, and dump their information to the "Output" window

```
import inspect

import ida_idp

class idb_logger_hooks_t(ida_idp.IDB_Hooks):

    def __init__(self):
        ida_idp.IDB_Hooks.__init__(self)
        self.inhibit_log = 0;

    def _format_value(self, v):
        return str(v)

    def _log(self, msg=None):
        if self.inhibit_log <= 0:
            if msg:
                print(">>> idb_logger_hooks_t: %s" % msg)
            else:
                stack = inspect.stack()
                frame, _, _, _, _, _ = stack[1]
                args, _, _, values = inspect.getargvalues(frame)
                method_name = inspect.getframeinfo(frame)[2]
                argstrs = []
                for arg in args[1:]:
                    argstrs.append("%s=%s" % (arg, self._format_value(values[arg])))
                print(">>> idb_logger_hooks_t.%s: %s" % (method_name, ", ".join(argstrs)))
        return 0

    def adding_segm(self, segment):
        return self._log()

    def allsegs_moved(self, info):
        return self._log()

    def auto_empty(self):
```

```
        return self._log()

    def auto_empty_finally(self):
        return self._log()

    def bookmark_changed(self, index, pos, desc, op):
        return self._log()

    def byte_patched(self, ea, old_value):
        return self._log()

    def callee_addr_changed(self, ea, callee):
        return self._log()

    def changing_cmt(self, ea, is_repeatable, new_comment):
        return self._log()

    def changing_enum_bf(self, tid, new_bf):
        return self._log()

    def changing_enum_cmt(self, tid, is_repeatable, new_comment):
        return self._log()

    def changing_op_ti(self, ea, n, new_type, new_fnames):
        return self._log()

    def changing_op_type(self, ea, n, opinfo):
        return self._log()

    def changing_range_cmt(self, kind, _range, comment, is_repeatable):
        return self._log()

    def changing_segm_class(self, segment):
        return self._log()

    def changing_segm_end(self, segment, new_end, flags):
        return self._log()

    def changing_segm_name(self, segment, old_name):
        return self._log()

    def changing_segm_start(self, segment, new_start, flags):
        return self._log()

    def changing_struc_align(self, sptr):
        return self._log()
```

```
def changing_struc_cmt(self, tid, is_repeatable, comment):
    return self._log()

def changing_struc_member(self, sptr, mptr, flags, ti, nbytes):
    return self._log()

def changing_ti(self, ea, new_type, new_fnames):
    return self._log()

def closebase(self):
    return self._log()

def cmt_changed(self, ea, is_repeatable):
    return self._log()

def compiler_changed(self, may_adjust_inf_fields):
    return self._log()

def deleting_enum(self, tid):
    return self._log()

def deleting_enum_member(self, tid, cid):
    return self._log()

def deleting_func(self, pfn):
    return self._log()

def deleting_func_tail(self, pfn, tail):
    return self._log()

def deleting_segm(self, start_ea):
    return self._log()

def deleting_struc(self, sptr):
    return self._log()

def deleting_struc_member(self, sptr, mptr):
    return self._log()

def deleting_tryblks(self, _range):
    return self._log()

def destroyed_items(self, ea1, ea2, will_disable_range):
    return self._log()
```

```
def determined_main(self, main):
    return self._log()

def dirtree_link(self, dt, path, is_link):
    return self._log()

def dirtree_mkdir(self, dt, path):
    return self._log()

def dirtree_move(self, dt, _from, to):
    return self._log()

def dirtree_rank(self, dt, path, rank):
    return self._log()

def dirtree_rmdir(self, dt, path):
    return self._log()

def dirtree_rminode(self, dt, inode):
    return self._log()

def dirtree_segm_moved(self, dt):
    return self._log()

def enum_bf_changed(self, tid):
    return self._log()

def enum_cmt_changed(self, tid, is_repeatable):
    return self._log()

def enum_created(self, tid):
    return self._log()

def enum_deleted(self, tid):
    return self._log()

def enum_member_created(self, tid, cid):
    return self._log()

def enum_member_deleted(self, tid, cid):
    return self._log()

def enum_renamed(self, tid):
    return self._log()

def expanding_struc(self, sptr, offset, delta):
```

```
        return self._log()

    def extlang_changed(self, kind, el, idx):
        return self._log()

    def extra_cmt_changed(self, ea, line_idx, comment):
        return self._log()

    def flow_chart_created(self, fc):
        return self._log()

    def frame_deleted(self, pfn):
        return self._log()

    def func_added(self, pfn):
        return self._log()

    def func_deleted(self, func_ea):
        return self._log()

    def func_noret_changed(self, pfn):
        return self._log()

    def func_tail_appended(self, pfn, tail):
        return self._log()

    def func_tail_deleted(self, pfn, tail_ea):
        return self._log()

    def func_updated(self, pfn):
        return self._log()

    def idasgn_loaded(self, sig_name):
        return self._log()

    def item_color_changed(self, ea, color):
        return self._log()

    def kernel_config_loaded(self, pass_number):
        return self._log()

    def loader_finished(self, li, neflags, filetypename):
        return self._log()

    def local_types_changed(self):
        return self._log()
```

```
def make_code(self, insn):
    return self._log()

def make_data(self, ea, flags, tid, _len):
    return self._log()

def op_ti_changed(self, ea, n, _type, fnames):
    return self._log()

def op_type_changed(self, ea, n):
    return self._log()

def range_cmt_changed(self, kind, _range, comment, is_repeatable):
    return self._log()

def renamed(self, ea, new_name, is_local_name, old_name):
    return self._log()

def renaming_enum(self, tid, is_enum, new_name):
    return self._log()

def renaming_struc(self, tid, old_name, new_name):
    return self._log()

def renaming_struc_member(self, sptr, mptr, new_name):
    return self._log()

def savebase(self):
    return self._log()

def segm_added(self, segment):
    return self._log()

def segm_attrs_updated(self, segment):
    return self._log()

def segm_class_changed(self, segment, sclass):
    return self._log()

def segm_deleted(self, start_ea, end_ea, flags):
    return self._log()

def segm_end_changed(self, segment, old_end):
    return self._log()
```

```
def segm_moved(self, _from, to, size, changed_netmap):
    return self._log()

def segm_name_changed(self, segment, name):
    return self._log()

def segm_start_changed(self, segment, old_start):
    return self._log()

def set_func_end(self, pfn, new_end):
    return self._log()

def set_func_start(self, pfn, new_start):
    return self._log()

def sgr_changed(self, start_ea, end_ea, regnum, value, old_value, tag):
    return self._log()

def sgr_deleted(self, start_ea, end_ea, regnum):
    return self._log()

def stkpnts_changed(self, pfn):
    return self._log()

def struc_align_changed(self, sptr):
    return self._log()

def struc_cmt_changed(self, tid, is_repeatable):
    return self._log()

def struc_created(self, tid):
    return self._log()

def struc_deleted(self, tid):
    return self._log()

def struc_expanded(self, sptr):
    return self._log()

def struc_member_changed(self, sptr, mptr):
    return self._log()

def struc_member_created(self, sptr, mptr):
    return self._log()

def struc_member_deleted(self, sptr, mid, offset):
```

```
        return self._log()

    def struc_member_renamed(self, sptr, mptr):
        return self._log()

    def struc_renamed(self, sptr, success):
        return self._log()

    def tail_owner_changed(self, tail, owner_func, old_owner):
        return self._log()

    def thunk_func_created(self, pfn):
        return self._log()

    def ti_changed(self, ea, _type, fnames):
        return self._log()

    def tryblks_updated(self, tbv):
        return self._log()

    def updating_tryblks(self, tbv):
        return self._log()

    def upgraded(self, _from):
        return self._log()

    def enum_width_changed(self, tid, width):
        return self._log()

    def enum_flag_changed(self, tid, flags):
        return self._log()

    def enum_ordinal_changed(self, tid, ordinal):
        return self._log()


idb_hooks = idb_logger_hooks_t()
idb_hooks.hook()
```

IDAPython example: idbhooks/operand_changed.py

notify the user when an instruction operand changes

description: Show notifications whenever the user changes an instruction's operand, or a data item.

```
import binascii

import ida_idp
import ida_bytes
import ida_nalt
import ida_struct
import ida_enum

class operand_changed_t(ida_idp.IDB_Hooks):
    def log(self, msg):
        print(">>> %s" % msg)

    def op_type_changed(self, ea, n):
        flags = ida_bytes.get_flags(ea)
        self.log("op_type_changed(ea=0x%08X, n=%d). Flags now: 0x%08X" % (ea, n, flags))

        buf = ida_nalt.opinfo_t()
        opi = ida_bytes.get_opinfo(buf, ea, n, flags)
        if opi:
            if ida_bytes.is_struct(flags):
                self.log("New struct: 0x%08X (name=%s)" % (
                    opi.tid,
                    ida_struct.get_struc_name(opi.tid)))
            elif ida_bytes.is_strlit(flags):
                encidx = ida_nalt.get_str_encoding_idx(opi.strtype)
                if encidx == ida_nalt.STRENC_DEFAULT:
                    encidx = ida_nalt.get_default_encoding_idx(ida_nalt.get_strtype_bpu(opi
                encname = ida_nalt.get_encoding_name(encidx)
                strlen = ida_bytes.get_max_strlit_length(
                    ea,
                    opi.strtype,
                    ida_bytes.ALOPT_IGNHEADS | ida_bytes.ALOPT_IGNCLT)
                raw = ida_bytes.get_strlit_contents(ea, strlen, opi.strtype) or b""
                self.log("New strlit: 0x%08X, raw hex=%s (encoding=%s)" % (
                    opi.strtype,
                    binascii.hexlify(raw),
                    encname))
            elif ida_bytes.is_off(flags, n):
                self.log("New offset: refinfo={target=0x%08X, base=0x%08X, tdelta=0x%08X, fl
                    opi.ri.target,
                    opi.ri.base,
                    opi.ri.tdelta,
                    opi.ri.flags))
            elif ida_bytes.is_enum(flags, n):
                self.log("New enum: 0x%08X (enum=%s), serial=%d" % (
```

```
                    opi.ec.tid,
                    ida_enum.get_enum_name(opi.ec.tid),
                    opi.ec.serial))
                pass
            elif ida_bytes.is_stroff(flags, n):
                parts = []
                for i in range(opi.path.len):
                    tid = opi.path.ids[i]
                    parts.append("0x%08X (name=%s)" % (tid, ida_struct.get_struc_name(tid)))
                self.log("New stroff: path=[%s] (len=%d, delta=0x%08X)" % (
                    ", ".join(parts),
                    opi.path.len,
                    opi.path.delta))
            elif ida_bytes.is_custom(flags) or ida_bytes.is_custfmt(flags, n):
                self.log("New custom data type") # unimplemented
        else:
            print("Cannot retrieve opinfo_t")
```

IDAPython example: idbhooks/replay_prototypes_changes.py

Record and replay changes in function prototypes

description: This is a sample script, that will record (in memory) all changes in functions prototypes, in order to re-apply them later.

```
To use this script:
 - open an IDB (say, "test.idb")
 - modify some functions prototypes (e.g., by triggering the 'Y'
   shortcut when the cursor is placed on the first address of a
   function)
 - reload that IDB, *without saving it first*
 - call rpc.replay(), to re-apply the modifications.

Note: 'ti_changed' is also called for changes to the function
frames, but we'll only record function prototypes changes.


import ida_idp
import ida_funcs
import ida_typeinf

class replay_prototypes_changes_t(ida_idp.IDB_Hooks):
    def __init__(self):
        ida_idp.IDB_Hooks.__init__(self)
        # we'll store tuples (ea, typ, fields). We cannot store
        # tinfo_t instances in there, because tinfo_t's are only
        # valid while the IDB is opened.
```

```
        # Since the very purpose of this example is to re-apply
        # types after the IDB has been closed & re-opened, we
        # must therefore keep the serialized version only.
        self.memo = []
        self.replaying = False

    def _deser(self, typ, fields):
        tif = ida_typeinf.tinfo_t()
        if not tif.deserialize(ida_typeinf.get_idati(), typ, fields):
            tif = None
        return tif

    def ti_changed(self, ea, typ, fields):
        if not self.replaying:
            pfn = ida_funcs.get_func(ea)
            if pfn and pfn.start_ea == ea:
                self.memo.append((ea, typ, fields))

                # de-serialize, just for the sake of printing
                tif = self._deser(typ, fields)
                if tif:
                    print("%x: type changed: %s" % (
                        ea,
                        tif._print(None, ida_typeinf.PRTYPE_1LINE)))

    def replay(self):
        self.replaying = True
        try:
            for ea, typ, fields in self.memo:
                tif = self._deser(typ, fields)
                if tif:
                    print("%x: applying type: %s" % (
                        ea,
                        tif._print(None, ida_typeinf.PRTYPE_1LINE)))
                    # Since that type information was remembered from a change
                    # the user made, we'll re-apply it as a definite type (i.e.,
                    # can't be overriden by IDA's auto-analysis/heuristics.)
                    apply_flags = ida_typeinf.TINFO_DEFINITE
                    if not ida_typeinf.apply_tinfo(ea, tif, apply_flags):
                        print("FAILED")
        finally:
            self.replaying = False

rpc = replay_prototypes_changes_t()
if rpc.hook():
    print("""
```

```
Please modify some functions prototypes (press 'Y' when the
cursor is on the function name, or first address), and when
you are done reload this IDB, *WITHOUT* saving it first,
and type 'rpc.replay()'
""")
else:
    print("Couldn't create hooks")
```

IDAPython example: idphooks/ana_emu_out.py

override some parts of the processor module

description: Implements disassembly of BUG_INSTR used in Linux kernel BUG() macro, which is architecturally undefined and is not disassembled by IDA's ARM module

See Linux/arch/arm/include/asm/bug.h for more info

```
import ida_idp
import ida_bytes
import ida_segregs

ITYPE_BUGINSN = ida_idp.CUSTOM_INSN_ITYPE + 10
MNEM_WIDTH = 16

class MyHooks(ida_idp.IDP_Hooks):

    def __init__(self):
        ida_idp.IDP_Hooks.__init__(self)
        self.reported = []

    def ev_ana_insn(self, insn):
        t_reg = ida_idp.str2reg("T")
        t = ida_segregs.get_sreg(insn.ea, t_reg)
        if t==0 and ida_bytes.get_wide_dword(insn.ea) == 0xE7F001F2:
            insn.itype = ITYPE_BUGINSN
            insn.size = 4
        elif t!=0 and ida_bytes.get_wide_word(insn.ea) == 0xde02:
            insn.itype = ITYPE_BUGINSN
            insn.size = 2
        return insn.size

    def ev_emu_insn(self, insn):
        if insn.itype == ITYPE_BUGINSN:
            return 1 # do not add any xrefs (stop code flow)
```

```
        # use default processing for all other functions
        return 0

    def ev_out_mnem(self, outctx):
        if outctx.insn.itype == ITYPE_BUGINSN:
            outctx.out_custom_mnem("BUG_INSTR", MNEM_WIDTH)
            return 1
        return 0

if ida_idp.ph.id == ida_idp.PLFM_ARM:
    bahooks = MyHooks()
    bahooks.hook()
    print("BUG_INSTR processor extension installed")
else:
    warning("This script only supports ARM files")
```

IDAPython example: idphooks/assemble.py

an `ida_idp.IDP_Hooks.assembly` implementation

description: We add support for assembling the following pseudo instructions:

- "zero eax" -> xor eax, eax
- "nothing" -> nop

```
import ida_idp
import idautils

#--------------------------------------------------------------------------
class assemble_idp_hook_t(ida_idp.IDP_Hooks):
    def ev_assemble(self, ea, cs, ip, use32, line):
        line = line.strip()
        if line == "zero eax":
            return b"\x33\xC0"
        elif line == "nothing":
            # Decode current instruction to figure out its size
            cmd = idautils.DecodeInstruction(ea)
            if cmd:
                # NOP all the instruction bytes
                return b"\x90" * cmd.size
        return None


#--------------------------------------------------------------------------
# Remove an existing hook on second run
```

```
try:
    idp_hook_stat = "un"
    print("IDP hook: checking for hook...")
    idphook
    print("IDP hook: unhooking....")
    idphook.unhook()
    del idphook
except:
    print("IDP hook: not installed, installing now....")
    idp_hook_stat = ""
    idphook = assemble_idp_hook_t()
    idphook.hook()

print("IDP hook %sinstalled. Run the script again to %sinstall" % (idp_hook_stat, idp_hook_s
```

IDAPython example: pyqt/inject_command.py

injecting commands in the "Output" window

description: This example illustrates how one can execute commands in the "Output" window, from their own widgets.

A few notes:

- the original, underlying `cli:Execute` action, that has to be triggered for the code present in the input field to execute and be placed in the history, requires that the input field has focus (otherwise it simply won't do anything.)
- this, in turn, forces us to do "delayed" execution of that action, hence the need for a `QTimer`
- the IDA/SWiG 'TWidget' type that we retrieve through `ida_kernwin.find_widget`, is not the same type as a `QtWidgets.QWidget`. We therefore need to convert it using `ida_kernwin.PluginForm.TWidgetToPyQtWidget`

```
from PyQt5 import QtCore
from PyQt5 import QtGui
from PyQt5 import QtWidgets

import ida_kernwin
import ida_segment

import idc

delayed_exec_timer = QtCore.QTimer()

def show_dialog():
```

```
dialog = QtWidgets.QDialog()
dialog.setWindowTitle("Inject command")
dialog.setMinimumSize(600, 480)

run_text = "Run"
buttons_box = QtWidgets.QDialogButtonBox()
button = buttons_box.addButton(run_text, QtWidgets.QDialogButtonBox.AcceptRole)
button.setDefault(True)
button.clicked.connect(dialog.accept)

text_edit = QtWidgets.QPlainTextEdit()
text_edit.setPlaceholderText(
    "Type an expression, and press '%s' to execute through the regular input" % run_text

layout = QtWidgets.QVBoxLayout()
layout.addWidget(text_edit)
layout.addWidget(buttons_box)

dialog.setLayout(layout)

# disable script timeout, otherwise a "Please wait ..." dialog
# might briefly show after the dialog was accepted/rejected
with ida_kernwin.disabled_script_timeout_t():
    if dialog.exec_() == QtWidgets.QDialog.Accepted:

        # We'll now have to schedule a call to the standard
        # 'execute' action. We can't call it right away, because
        # the "Output" window doesn't have focus, and thus
        # the action will fail to execute since it requires
        # the "Output" window as context.
        text = text_edit.toPlainText()

        def delayed_exec(*args):
            output_window_title = "Output"
            tw = ida_kernwin.find_widget(output_window_title)
            if not tw:
                raise Exception("Couldn't find widget '%s'" % output_window_title)

            # convert from a SWiG 'TWidget*' facade,
            # into an object that PyQt will understand
            w = ida_kernwin.PluginForm.TWidgetToPyQtWidget(tw)

            line_edit = w.findChild(QtWidgets.QLineEdit)
            if not line_edit:
                raise Exception("Couldn't find input")
            line_edit.setFocus() # ensure it has focus
```

```
                QtWidgets.QApplication.instance().processEvents() # and that it received the

                # inject text into widget
                line_edit.setText(text)

                # and execute the standard 'execute' action
                ida_kernwin.process_ui_action("cli:Execute")

            delayed_exec_timer.singleShot(0, delayed_exec)


show_dialog()
```

IDAPython example: pyqt/paint_over_graph.py

custom painting on top of graph view edges

description: This sample registers an action enabling painting of a recognizable string of text over horizontal nodes edge sections beyond a satisfying size threshold.

In a disassembly view, open the context menu and select "Paint on edges". This should work for both graph disassembly, and proximity browser.

Using an "event filter", we will intercept paint events targeted at the disassembly view, let it paint itself, and then add our own markers along.

```
from PyQt5 import QtCore
from PyQt5 import QtGui
from PyQt5 import QtWidgets

import ida_graph
import ida_kernwin
import ida_moves

edge_segment_threshold = 50
text_color = QtGui.QColor(0, 0, 0)
text_antialiasing = True
verbose = False

class painter_t(QtCore.QObject):
    def __init__(self, w, verbose=False):
        QtCore.QObject.__init__(self)
        self.idaview = w
        self.idaview_pyqt = ida_kernwin.PluginForm.FormToPyQtWidget(w)
        self.target = self.idaview_pyqt.viewport()
```

```
        self.target.installEventFilter(self)
        self.painting = False

    def eventFilter(self, receiver, event):
        if not self.painting and \
           receiver == self.target and \
           event.type() == QtCore.QEvent.Paint:
            # Send a paint event that we won't intercept
            self.painting = True
            try:
                pev = QtGui.QPaintEvent(self.target.rect())
                QtWidgets.QApplication.instance().sendEvent(self.target, pev)
            finally:
                self.painting = False

            # now we can paint our items
            viewer = ida_graph.get_graph_viewer(self.idaview)
            graph = ida_graph.get_viewer_graph(viewer)
            if graph:
                painter = QtGui.QPainter(receiver)
                if text_antialiasing:
                    painter.setRenderHints(QtGui.QPainter.TextAntialiasing)
                else:
                    # this is primarily used for testing
                    font = painter.font()
                    font.setStyleStrategy(font.NoAntialias)
                    painter.setFont(font)
                painter.setPen(text_color)

                # The edge layout info we retrieve will be in "graph
                # coordinates". In order to transform those points to
                # view coordinates we will need the graph location info
                gli = ida_moves.graph_location_info_t()
                ida_graph.viewer_get_gli(gli, viewer);
                def to_view_coords(pt):
                    x = int((pt.x - gli.orgx) * gli.zoom)
                    y = int((pt.y - gli.orgy) * gli.zoom)
                    return ida_graph.point_t(x, y)

                # Let `src_node` be each visible node in the graph...
                for src_node in range(graph.size()):
                    if not graph.is_visible_node(src_node):
                        continue

                    # ...and `dst_node` be each visible node
                    # to which `src_node` is connected
```

```
                    for dst_node_idx in range(graph.nsucc(src_node)):
                        dst_node = graph.succ(src_node, dst_node_idx)
                        if not graph.is_visible_node(dst_node):
                            continue

                        edge_info = graph.get_edge(ida_graph.edge_t(src_node, dst_node))
                        if edge_info:

                            # For all horizontal edge segments satisfying the length requir
                            for idx in range(len(edge_info.layout)-1):
                                src = to_view_coords(edge_info.layout[idx])
                                dst = to_view_coords(edge_info.layout[idx+1])
                                if src.y == dst.y and abs(src.x - dst.x) > edge_segment_thre
                                    off = 6
                                    text = "%s -> %s (#%d)" % (src_node, dst_node, idx)
                                    if verbose:
                                        print("Painting \"%s\"" % text)
                                    painter.drawText(min(src.x, dst.x) + off, src.y - off, t

                painter.end()

                # ...and prevent the widget form painting itself again
                return True
        return QtCore.QObject.eventFilter(self, receiver, event)

painter = None

class paint_on_edges_t(ida_kernwin.action_handler_t):
    def activate(self, ctx):
        if self.get_idaview(ctx):
            global painter
            painter = painter_t(ctx.widget)
            return 1
        return 0

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_FOR_WIDGET \
            if self.get_idaview(ctx) \
               else ida_kernwin.AST_DISABLE_FOR_WIDGET

    def get_idaview(self, ctx):
        return ctx.widget if ctx.widget_type == ida_kernwin.BWN_DISASM else None

action_name = "paint_over_graph:enable"
ida_kernwin.register_action(
    ida_kernwin.action_desc_t(
```

```
        action_name,
        "Paint on edges",
        paint_on_edges_t()))

#
# Make sure our action is available for all disassembly views
#
class context_menu_hooks_t(ida_kernwin.UI_Hooks):
    def finish_populating_widget_popup(self, widget, popup):
        if ida_kernwin.get_widget_type(widget) == ida_kernwin.BWN_DISASM:
            ida_kernwin.attach_action_to_popup(widget, popup, action_name, None)

hooks = context_menu_hooks_t()
hooks.hook()
```

IDAPython example: pyqt/paint_over_navbar.py

custom painting on top of the navigation band

description: Using an "event filter", we will intercept paint events targeted at the navigation band widget, let it paint itself, and then add our own markers on top.

```
import random

from PyQt5 import QtCore
from PyQt5 import QtGui
from PyQt5 import QtWidgets

import ida_kernwin
import ida_segment

import idc

class painter_t(QtCore.QObject):
    def __init__(self):
        QtCore.QObject.__init__(self)
        self.target = ida_kernwin.PluginForm.FormToPyQtWidget(ida_kernwin.open_navband_windo
        self.target.installEventFilter(self)
        self.items = []
        self.painting = False

    def add_item(self, ea, radius, color):
        self.items.append((ea, radius, color))
        self.target.update()
```

```
    def add_random_item(self):
        R = random.random
        s = ida_segment.getnseg(int(ida_segment.get_segm_qty() * R()))
        ea = s.start_ea + int((s.end_ea - s.start_ea) * R())
        radius = 4 + int(R() * 8)
        color = QtGui.QColor(int(255 * R()), int(255 * R()), int(255 * R()))
        self.add_item(ea, radius, color)

    def eventFilter(self, receiver, event):
        if not self.painting and \
           self.target == receiver and \
           event.type() == QtCore.QEvent.Paint:

            # Send a paint event that we won't intercept
            self.painting = True
            try:
                pev = QtGui.QPaintEvent(self.target.rect())
                QtWidgets.QApplication.instance().sendEvent(self.target, pev)
            finally:
                self.painting = False

            # now we can paint our items
            for ea, radius, color in self.items:
                painter = QtGui.QPainter(receiver)
                painter.setRenderHints(QtGui.QPainter.Antialiasing)
                pxl, is_vertical = ida_kernwin.get_navband_pixel(ea)
                if pxl >= 0:
                    x = (self.target.width() // 2) if is_vertical else pxl
                    y = pxl if is_vertical else (self.target.height() // 2)
                    painter.setPen(color)
                    painter.setBrush(color)
                    painter.drawEllipse(QtCore.QPoint(x, y), radius, radius)
                painter.end()

            # ...and prevent the widget form painting itself again
            return True
        return QtCore.QObject.eventFilter(self, receiver, event)

painter = painter_t()

# Try the following:
# for i in range(100): painter.add_random_item()
```

IDAPython example: pyqt/populate_pluginform_with_pyqt_widgets.py

adding PyQt5 widgets into an `ida_kernwin.PluginForm`

description: Using `ida_kernwin.PluginForm.FormToPyQtWidget`, this script converts IDA's own dockable widget into a type that is recognized by PyQt5, which then enables populating it with regular Qt widgets.

```
from PyQt5 import QtCore, QtGui, QtWidgets

class MyPluginFormClass(ida_kernwin.PluginForm):
    def OnCreate(self, form):
        """
        Called when the widget is created
        """

        # Get parent widget
        self.parent = self.FormToPyQtWidget(form)
        self.PopulateForm()


    def PopulateForm(self):
        # Create layout
        layout = QtWidgets.QVBoxLayout()

        layout.addWidget(
            QtWidgets.QLabel("Hello from <font color=red>PyQt</font>"))
        layout.addWidget(
            QtWidgets.QLabel("Hello from <font color=blue>IDAPython</font>"))

        self.parent.setLayout(layout)


    def OnClose(self, form):
        """
        Called when the widget is closed
        """
        pass

plg = MyPluginFormClass()
plg.Show("PyQt hello world")
```

IDAPython example: uihooks/func_chooser_coloring.py

using `ida_kernwin.UI_Hooks.get_chooser_item_attrs` to override some defaults

description: color the function in the Function window according to its size. The larger the function, the darker the color.

```
import ida_kernwin
import ida_funcs
import math

class func_chooser_coloring_hooks_t(ida_kernwin.UI_Hooks):
    def __init__(self):
        ida_kernwin.UI_Hooks.__init__(self)
        self.colors = [0x808080 + (32-i) * 0x400 for i in range(32-5)]

    def get_chooser_item_attrs(self, chobj, n, attrs):
        if attrs.color != 0xFFFFFFFF:
            return # the color is already set
        ea = chobj.get_ea(n)
        fn = ida_funcs.get_func(ea)
        size = fn.size()
        if size < 32:
            return # do not color small functions
        attrs.color = self.colors[int(math.log2(size))]

fcch = func_chooser_coloring_hooks_t()
fcch.hook()
ida_kernwin.enable_chooser_item_attrs("Functions", True)
```

IDAPython example: uihooks/lines_rendering.py

dynamically colorize lines backgrounds (or parts of them)

description: shows how one can dynamically alter the lines background rendering (as opposed to, say, using ida_nalt.set_item_color()), and also shows how that rendering can be limited to just a few glyphs, not the whole line.

```
import ida_kernwin
import ida_bytes

class lines_rendering_hooks_t(ida_kernwin.UI_Hooks):
```

```
def __init__(self):
    ida_kernwin.UI_Hooks.__init__(self)

    # We'll color all lines starting with the current
    # one, with all available highlights...
    self.instantiated_at = ida_kernwin.get_screen_ea()
    self.color_info = []

    data = [
        ida_kernwin.CK_EXTRA1,
        ida_kernwin.CK_EXTRA2,
        ida_kernwin.CK_EXTRA3,
        ida_kernwin.CK_EXTRA4,
        ida_kernwin.CK_EXTRA5,
        ida_kernwin.CK_EXTRA6,
        ida_kernwin.CK_EXTRA7,
        ida_kernwin.CK_EXTRA8,
        ida_kernwin.CK_EXTRA9,
        ida_kernwin.CK_EXTRA10,
        ida_kernwin.CK_EXTRA11,
        ida_kernwin.CK_EXTRA12,
        ida_kernwin.CK_EXTRA13,
        ida_kernwin.CK_EXTRA14,
        ida_kernwin.CK_EXTRA15,
        ida_kernwin.CK_EXTRA16,
        # let's also try these colors keys, because why not
        ida_kernwin.CK_TRACE,
        ida_kernwin.CK_TRACE_OVL,
        [
            ida_kernwin.CK_TRACE,
            ida_kernwin.CK_TRACE_OVL,
        ],
    ]
    ea = self.instantiated_at
    for one in data:
        self.color_info.append((ea, one))
        ea = ida_bytes.next_head(ea, ida_idaapi.BADADDR)

    # ...and then we'll a few more things, such as
    # overriding parts of a previously-specified overlay,
    # and restricting the override to a few glyphs
    self.color_info.append(
        (
            self.color_info[6][0],
            [
                (ida_kernwin.CK_EXTRA2, 7, 3),
```

```
                (ida_kernwin.CK_EXTRA4, 2, 1),
                (ida_kernwin.CK_EXTRA10, 2, 0),
                (ida_kernwin.CK_EXTRA10, 20, 10),
            ]
        ))
    self.color_info.append(
        (
            self.color_info[7][0],
            [
                (ida_kernwin.CK_EXTRA1, 1, 1),
                (ida_kernwin.CK_EXTRA2, 3, 1),
                (ida_kernwin.CK_EXTRA3, 5, 1),
                (ida_kernwin.CK_EXTRA4, 7, 1),
                (ida_kernwin.CK_EXTRA5, 9, 1),
                (ida_kernwin.CK_EXTRA6, 11, 1),
                (ida_kernwin.CK_EXTRA7, 13, 1),
                (ida_kernwin.CK_EXTRA8, 15, 1),
                (ida_kernwin.CK_EXTRA9, 17, 1),
                (ida_kernwin.CK_EXTRA10, 19, 1),
                (ida_kernwin.CK_EXTRA11, 21, 1),
                (ida_kernwin.CK_EXTRA12, 23, 1),
                (ida_kernwin.CK_EXTRA13, 25, 1),
                (ida_kernwin.CK_EXTRA14, 27, 1),
                (ida_kernwin.CK_EXTRA15, 29, 1),
                (ida_kernwin.CK_EXTRA16, 31, 1),
            ]
        ))
    self.color_info.append(
        (
            self.color_info[8][0],
            [
                (ida_kernwin.CK_EXTRA1, 16, 45),
                (ida_kernwin.CK_EXTRA2, 19, 45),
                (ida_kernwin.CK_EXTRA3, 22, 45),
                (ida_kernwin.CK_EXTRA4, 25, 45),
            ]
        ))


def get_lines_rendering_info(self, out, widget, rin):
    for section_lines in rin.sections_lines:
        for line in section_lines:
            line_ea = line.at.toea()
            for ea, directives in self.color_info:
                if ea == line_ea:
                    if not isinstance(directives, list):
```

```
                directives = [directives]
            for directive in directives:
                e = ida_kernwin.line_rendering_output_entry_t(line)
                if isinstance(directive, tuple):
                    color, cpx, nchars = directive
                    e.bg_color = color
                    e.cpx = cpx
                    e.nchars = nchars
                    e.flags |= ida_kernwin.LROEF_CPS_RANGE
                else:
                    e.bg_color = directive
                out.entries.push_back(e)


lrh = lines_rendering_hooks_t()
lrh.hook()

# Force a refresh of IDA View-A
ida_kernwin.refresh_idaview_anyway()
```

IDAPython example: uihooks/log_misc_events.py

being notified, and logging a few UI events

description: hooks to be notified about certain UI events, and dump their information to the "Output" window

```
import inspect

import ida_kernwin

class MyUiHook(ida_kernwin.UI_Hooks):
    def __init__(self):
        ida_kernwin.UI_Hooks.__init__(self)
        self.cmdname = "<no command>"
        self.inhibit_log = 0;

    def _format_value(self, v):
        return str(v)

    def _log(self, msg=None):
        if self.inhibit_log <= 0:
            if msg:
                print(">>> MyUiHook: %s" % msg)
            else:
```

```
                stack = inspect.stack()
                frame, _, _, _, _, _ = stack[1]
                args, _, _, values = inspect.getargvalues(frame)
                method_name = inspect.getframeinfo(frame)[2]
                argstrs = []
                for arg in args[1:]:
                    argstrs.append("%s=%s" % (arg, self._format_value(values[arg])))
                print(">>> MyUiHook.%s: %s" % (method_name, ", ".join(argstrs)))
        return 0

    def preprocess_action(self, name):
        self._log("IDA preprocessing command: %s" % name)
        self.cmdname = name
        return 0

    def postprocess_action(self):
        self._log("IDA finished processing command: %s" % self.cmdname)
        return 0

    def saving(self):
        """
        The kernel is saving the database.

        @return: Ignored
        """
        self._log("Saving....")

    def saved(self):
        """
        The kernel has saved the database.

        @return: Ignored
        """
        self._log("Saved")

    def term(self):
        """
        IDA is terminated and the database is already closed.
        The UI may close its windows in this callback.

        This callback is best used within the context of a plugin_t with PLUGIN_FIX flags
        """
        self._log("IDA terminated")

    def get_ea_hint(self, ea):
        """
```

```
        The UI wants to display a simple hint for an address in the navigation band

        @param ea: The address
        @return: String with the hint or None
        """
        self._log("get_ea_hint(%x)" % ea)

    def populating_widget_popup(self, widget, popup, ctx):
        """
        The UI is currently populating the widget popup. Now is a good time to
        attach actions.
        """
        self._log("populating_widget_popup; title: %s" % (ctx.widget_title,))

    def finish_populating_widget_popup(self, widget, popup, ctx):
        """
        The UI is done populating the widget popup. Now is the last chance to
        attach actions.
        """
        self._log("finish_populating_widget_popup; title: %s" % (ctx.widget_title,))

    def range(self):
        self._log()

    def idcstart(self):
        self._log()

    def idcstop(self):
        self._log()

    def suspend(self):
        self._log()

    def resume(self):
        self._log()

    def debugger_menu_change(self, enable):
        self._log()

    def widget_visible(self, widget):
        self._log()

    def widget_closing(self, widget):
        self._log()

    def widget_invisible(self, widget):
```

```
        self._log()

    def get_item_hint(self, ea, max_lines):
        self._log()

    def get_custom_viewer_hint(self, viewer, place):
        self._log()

    def database_inited(self, is_new_database, idc_script):
        self._log()

    def ready_to_run(self):
        self._log()

    def get_chooser_item_attrs(self, chooser, n, attrs):
        self._log()

    def updating_actions(self, ctx):
        self._log()

    def updated_actions(self):
        self._log()

    def plugin_loaded(self, plugin_info):
        self._log()

    def plugin_unloading(self, plugin_info):
        self._log()

    def current_widget_changed(self, widget, prev_widget):
        self._log()

    def screen_ea_changed(self, ea, prev_ea):
        self._log()

    def create_desktop_widget(self, title, cfg):
        self._log()

    def get_lines_rendering_info(self, out, widget, info):
        self._log()

    def get_widget_config(self, widget, cfg):
        self._log()

    def set_widget_config(self, widget, cfg):
        self._log()
```

```
    def initing_database(self):
        self._log()

    def destroying_procmod(self, procmod):
        self._log()

    def destroying_plugmod(self, plugmod, entry):
        self._log()

    def desktop_applied(self, name, from_idb, type):
        self._log()


#--------------------------------------------------------------------------
# Remove an existing hook on second run
try:
    ui_hook_stat = "un"
    print("UI hook: checking for hook...")
    uihook
    print("UI hook: unhooking....")
    ui_hook_stat2 = ""
    uihook.unhook()
    del uihook
except:
    print("UI hook: not installed, installing now....")
    ui_hook_stat = ""
    ui_hook_stat2 = "un"
    uihook = MyUiHook()
    uihook.hook()

print("UI hook %sinstalled. Run the script again to %sinstall" % (ui_hook_stat, ui_hook_stat
```

IDAPython example: uihooks/prevent_jump.py

taking precedence over actions

description: Using `ida_kernwin.UI_Hooks.preprocess_action`, it is possible to respond to a command instead of the action that would otherwise do it.

```
import ida_kernwin

class prevent_jump_t(ida_kernwin.UI_Hooks):
    def preprocess_action(self, action_name):
        if action_name == "JumpEnter":
```

```
            print("Inhibiting 'jump'!")
            return 1
        return 0

phh = prevent_jump_t()
if phh.hook():
    print("From now on, pressing <Enter> will prevent IDA from jumping. "\
          +"Please type 'phh.unhook()' to revert to the normal behavior.")
```

IDAPython example: widgets/forms/askusingform.py

Non-trivial uses of the `ida_kernwin.Form` helper class

description: How to query for complex user input, using IDA's built-in forms.

Note: while this example produces full-fledged forms for complex input, simpler types of inputs might can be retrieved by using `ida_kernwin.ask_str` and similar functions.

keywords: forms

```
# --------------------------------------------------------------------------
# This is an example illustrating how to use the Form class
# (c) Hex-Rays
#
import ida_kernwin

# --------------------------------------------------------------------------
class busy_form_t(ida_kernwin.Form):

    class test_chooser_t(ida_kernwin.Choose):
        """
        A simple chooser to be used as an embedded chooser
        """
        def __init__(self, title, nb=5, flags=ida_kernwin.Choose.CH_MULTI):
            ida_kernwin.Choose.__init__(
                self,
                title,
                [
                    ["Address", 10],
                    ["Name", 30]
                ],
                flags=flags,
                embedded=True,
                width=30,
                height=6)
```

```
            self.items = [ [str(x), "func_%04d" % x] for x in range(nb + 1) ]
            self.icon = 5

        def OnGetLine(self, n):
            print("getline %d" % n)
            return self.items[n]

        def OnGetSize(self):
            n = len(self.items)
            print("getsize -> %d" % n)
            return n

    def __init__(self):
        self.invert = False
        F = ida_kernwin.Form
        F.__init__(
            self,
            r"""STARTITEM {id:rNormal}
BUTTON YES* Yeah
BUTTON NO Nope
BUTTON CANCEL Nevermind
Form Test

{FormChangeCb}
This is a string:  |+{cStr1}+
This is an address:|+{cAddr1}+
This is some HTML: |+{cHtml1}+
This is a number:  |+{cVal1}+

<#Hint1#Enter text  :{iStr1}>
<#Hint2#Select color:{iColor1}>
Browse test
<#Select a file to open#Browse to open:{iFileOpen}>
<#Select a file to save#Browse to save:{iFileSave}>
<#Select dir#Browse for dir:{iDir}>
Misc
<##Enter a selector value:{iSegment}>
<##Enter a raw hex       :{iRawHex}>
<##Enter a character     :{iChar}>
<##Enter an address      :{iAddr}>
<##Write a type name     :{iType}>
Button test: <##Button1:{iButton1}> <##Button2:{iButton2}>

<##Check boxes##Error output:{rError}> | <##Radio boxes##Green:{rGreen}>
<Normal output:{rNormal}>              | <Red:{rRed}>
<Warnings:{rWarnings}>{cGroup1}>       | <Blue:{rBlue}>{cGroup2}>
```

```
<Embedded chooser:{cEChooser}>
The end!
""", {
            'cStr1': F.StringLabel("Hello"),
            'cHtml1': F.StringLabel("<span style='color: red'>Is this red?<span>", tp=F.FT_H
            'cAddr1': F.NumericLabel(0x401000, F.FT_ADDR),
            'cVal1' : F.NumericLabel(99, F.FT_HEX),
            'iStr1': F.StringInput(),
            'iColor1': F.ColorInput(),
            'iFileOpen': F.FileInput(open=True),
            'iFileSave': F.FileInput(save=True),
            'iDir': F.DirInput(),
            'iType': F.StringInput(tp=F.FT_TYPE),
            'iSegment': F.NumericInput(tp=F.FT_SEG),
            'iRawHex': F.NumericInput(tp=F.FT_RAWHEX),
            'iAddr': F.NumericInput(tp=F.FT_ADDR),
            'iChar': F.NumericInput(tp=F.FT_CHAR),
            'iButton1': F.ButtonInput(self.OnButton1),
            'iButton2': F.ButtonInput(self.OnButton2),
            'cGroup1': F.ChkGroupControl(("rNormal", "rError", "rWarnings")),
            'cGroup2': F.RadGroupControl(("rRed", "rGreen", "rBlue")),
            'FormChangeCb': F.FormChangeCb(self.OnFormChange),
            'cEChooser' : F.EmbeddedChooserControl(busy_form_t.test_chooser_t("E1"))
        })

    def OnButton1(self, code=0):
        print("Button1 pressed")

    def OnButton2(self, code=0):
        print("Button2 pressed")

    def OnFormChange(self, fid):
        if fid == self.iButton1.id:
            print("Button1 fchg;inv=%s" % self.invert)
            self.SetFocusedField(self.rNormal)
            self.EnableField(self.rError, self.invert)
            self.invert = not self.invert
        elif fid == self.iButton2.id:
            g1 = self.GetControlValue(self.cGroup1)
            g2 = self.GetControlValue(self.cGroup2)
            d = self.GetControlValue(self.iDir)
            f = self.GetControlValue(self.iFileOpen)
            print("cGroup2:%x;Dir=%s;fopen=%s;cGroup1:%x" % (g1, d, f, g2))
        elif fid == self.cEChooser.id:
            l = self.GetControlValue(self.cEChooser)
```

```
            print("Chooser: %s" % l)
        elif fid in [self.rGreen.id, self.rRed.id, self.rBlue.id]:
            color = {
                self.rGreen.id : 0x00FF00,
                self.rRed.id   : 0x0000FF,
                self.rBlue.id  : 0xFF0000,
            }
            self.SetControlValue(self.iColor1, color[fid])
        elif fid == self.iColor1.id:
            print("Color changed: %06x" % self.GetControlValue(self.iColor1))
        else:
            print(">>fid:%d" % fid)
        return 1

    @staticmethod
    def compile_and_fiddle_with_fields():
        f = busy_form_t()
        f, args = f.Compile()
        print(args[0])
        print(args[1:])
        f.rNormal.checked = True
        f.rWarnings.checked = True
        print(hex(f.cGroup1.value))

        f.rGreen.selected = True
        print(f.cGroup2.value)
        print("Title: '%s'" % f.title)

        f.Free()

    @staticmethod
    def test():
        f = busy_form_t()

        # Compile (in order to populate the controls)
        f.Compile()

        f.iColor1.value = 0x5bffff
        f.iDir.value = os.getcwd()
        f.iChar.value = ord("a")
        f.rNormal.checked = True
        f.rWarnings.checked = True
        f.rGreen.selected = True
        f.iStr1.value = "Hello"
        f.iFileSave.value = "*.*"
        f.iFileOpen.value = "*.*"
```

```
        # Execute the form
        ok = f.Execute()
        print("r=%d" % ok)
        if ok == 1:
            print("f.str1=%s" % f.iStr1.value)
            print("f.color1=%x" % f.iColor1.value)
            print("f.openfile=%s" % f.iFileOpen.value)
            print("f.savefile=%s" % f.iFileSave.value)
            print("f.dir=%s" % f.iDir.value)
            print("f.type=%s" % f.iType.value)
            print("f.seg=%s" % f.iSegment.value)
            print("f.rawhex=%x" % f.iRawHex.value)
            print("f.char=%x" % f.iChar.value)
            print("f.addr=%x" % f.iAddr.value)
            print("f.cGroup1=%x" % f.cGroup1.value)
            print("f.cGroup2=%x" % f.cGroup2.value)
            sel = f.cEChooser.selection
            if sel is None:
                print("No selection")
            else:
                print("Selection: %s" % sel)

        # Dispose the form
        f.Free()

# ----------------------------------------------------------------------
class multiline_text_t(ida_kernwin.Form):
    """Simple Form to test multilinetext"""
    def __init__(self):
        F = ida_kernwin.Form
        F.__init__(self, r"""STARTITEM 0
BUTTON YES* Yeah
BUTTON NO Nope
BUTTON CANCEL NONE
Form Test

{FormChangeCb}
<Multilinetext:{txtMultiLineText}>
""", {
            'txtMultiLineText': F.MultiLineTextControl(text="Hello"),
            'FormChangeCb': F.FormChangeCb(self.OnFormChange),
        })

    def OnFormChange(self, fid):
        if fid == self.txtMultiLineText.id:
```

```
            pass
        elif fid == -2:
            ti = self.GetControlValue(self.txtMultiLineText)
            print("ti.text = %s" % ti.text)
        else:
            print(">>fid:%d" % fid)
        return 1

    @staticmethod
    def test(execute=True):
        f = multiline_text_t()
        f, args = f.Compile()
        if execute:
            ok = f.Execute()
        else:
            print(args[0])
            print(args[1:])
            ok = 0
        if ok == 1:
            assert f.txtMultiLineText.text == f.txtMultiLineText.value
            print(f.txtMultiLineText.text)
        f.Free()


# --------------------------------------------------------------------------
class multiline_text_and_dropdowns_t(ida_kernwin.Form):
    """Simple Form to test multilinetext and combo box controls"""
    def __init__(self):
        self.__n = 0
        F = ida_kernwin.Form
        F.__init__(self,
r"""BUTTON YES* Yeah
BUTTON NO Nope
BUTTON CANCEL NONE
Dropdown list test

{FormChangeCb}
<Dropdown list (readonly):{cbReadonly}> <Add element:{iButtonAddelement}> <Set index:{iButto
<Dropdown list (editable):{cbEditable}> <Set string:{iButtonSetString}>
""", {
            'FormChangeCb': F.FormChangeCb(self.OnFormChange),
            'cbReadonly': F.DropdownListControl(
                        items=["red", "green", "blue"],
                        readonly=True,
                        selval=1),
            'cbEditable': F.DropdownListControl(
```

```
                    items=["1MB", "2MB", "3MB", "4MB"],
                    readonly=False,
                    selval="4MB"),
        'iButtonAddelement': F.ButtonInput(self.OnButtonNop),
        'iButtonSetIndex': F.ButtonInput(self.OnButtonNop),
        'iButtonSetString': F.ButtonInput(self.OnButtonNop),
    })


def OnButtonNop(self, code=0):
    """Do nothing, we will handle events in the form callback"""
    pass

def OnFormChange(self, fid):
    if fid == self.iButtonSetString.id:
        s = ida_kernwin.ask_str("none", 0, "Enter value")
        if s:
            self.SetControlValue(self.cbEditable, s)
    elif fid == self.iButtonSetIndex.id:
        s = ida_kernwin.ask_str("1", 0, "Enter index value:")
        if s:
            try:
                i = int(s)
            except:
                i = 0
            self.SetControlValue(self.cbReadonly, i)
    elif fid == self.iButtonAddelement.id:
        # add a value to the string list
        self.__n += 1
        self.cbReadonly.add("some text #%d" % self.__n)
        # Refresh the control
        self.RefreshField(self.cbReadonly)
    elif fid == -2:
        s = self.GetControlValue(self.cbEditable)
        print("user entered: %s" % s)
        sel_idx = self.GetControlValue(self.cbReadonly)
    return 1

@staticmethod
def test(execute=True):
    f = multiline_text_and_dropdowns_t()
    f, args = f.Compile()
    if execute:
        ok = f.Execute()
    else:
        print(args[0])
```

```
            print(args[1:])
            ok = 0
        if ok == 1:
            print("Editable: %s" % f.cbEditable.value)
            print("Readonly: %s" % f.cbReadonly.value)
        f.Free()

    NON_MODAL_INSTANCE = None

    @staticmethod
    def test_non_modal():
        if multiline_text_and_dropdowns_t.NON_MODAL_INSTANCE is None:
            f = multiline_text_and_dropdowns_t()
            f.modal = False
            f.openform_flags = ida_kernwin.PluginForm.FORM_TAB
            f, _ = f.Compile()
            multiline_text_and_dropdowns_t.NON_MODAL_INSTANCE = f
        multiline_text_and_dropdowns_t.NON_MODAL_INSTANCE.Open()

# -----------------------------------------------------------------------
busy_form_t.test()
```

IDAPython example: widgets/graphs/custom_graph_with_actions.py

drawing custom graphs

description: Showing custom graphs, using `ida_graph.GraphViewer`. In addition, show how to write actions that can be performed on those.

keywords: graph, actions

```
# -----------------------------------------------------------------------
# This is an example illustrating how to use the user graphing functionality
# in Python
# (c) Hex-Rays
#

import ida_kernwin
import ida_graph
import ida_ua
import ida_idp
import ida_funcs
import ida_xref

import idautils
```

```
class _base_graph_action_handler_t(ida_kernwin.action_handler_t):
    def __init__(self, graph):
        ida_kernwin.action_handler_t.__init__(self)
        self.graph = graph

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_ALWAYS


class GraphCloser(_base_graph_action_handler_t):
    def activate(self, ctx):
        self.graph.Close()


class ColorChanger(_base_graph_action_handler_t):
    def activate(self, ctx):
        self.graph.color = self.graph.color ^ 0xffffff
        self.graph.Refresh()
        return 1


class SelectionPrinter(_base_graph_action_handler_t):
    def activate(self, ctx):
        try:
            sel = ctx.graph_selection
        except:
            # IDA < 7.4 doesn't provide graph selection as part of
            # the action_activation_ctx_t; it needs to be queried.
            sel = ida_graph.screen_graph_selection_t()
            gv = ida_graph.get_graph_viewer(self.graph.GetWidget())
            ida_graph.viewer_get_selection(gv, sel)
        if sel:
            for s in sel:
                if s.is_node:
                    print("Selected node %d" % s.node)
                else:
                    print("Selected edge %d -> %d" % (s.elp.e.src, s.elp.e.dst))
        return 1


class MyGraph(ida_graph.GraphViewer):
    def __init__(self, funcname, result):
        self.title = "call graph of " + funcname
        ida_graph.GraphViewer.__init__(self, self.title)
        self.funcname = funcname
```

```
        self.result = result
        self.color = 0xff00ff

        #
        # for the sake of this example, here's how one can use
        # 'ida_kernwin.View_Hooks' (which can be used with all
        # "listing-like" and "graph" widgets) to be notified
        # of cursor movement, current node changes, etc...
        # in this graph.
        #
        class my_view_hooks_t(ida_kernwin.View_Hooks):
            def __init__(self, v):
                ida_kernwin.View_Hooks.__init__(self)
                self.hook()
                # let's use weakrefs, so as soon as the last ref to
                # the 'MyGraph' instance is dropped, the 'my_view_hooks_t'
                # instance hooks can be automatically un-hooked, and deleted.
                # (in other words: avoid circular reference.)
                import weakref
                self.v = weakref.ref(v)

            def view_loc_changed(self, w, now, was):
                now_node = now.renderer_info().pos.node
                was_node = was.renderer_info().pos.node
                if now_node != was_node:
                    if self.v().GetWidget() == w:
                        print("Current node now: #%d (was #%d)" % (now_node, was_node))

        self.my_view_hooks = my_view_hooks_t(self)


    def OnRefresh(self):
        self.Clear()
        id = self.AddNode((self.funcname, self.color))
        for x in self.result:
            callee = self.AddNode((x, self.color))
            self.AddEdge(id, callee)

        return True

    def OnGetText(self, node_id):
        return self[node_id]

    def OnPopup(self, widget, popup_handle):
        # graph closer
        actname = "graph_closer:%s" % self.title
```

```
        desc = ida_kernwin.action_desc_t(actname, "Close: %s" % self.title, GraphCloser(self
        ida_kernwin.attach_dynamic_action_to_popup(None, popup_handle, desc)

        # color changer
        actname = "color_changer:%s" % self.title
        desc = ida_kernwin.action_desc_t(actname, "Change colors: %s" % self.title, ColorCha
        ida_kernwin.attach_dynamic_action_to_popup(None, popup_handle, desc)

        # selection printer
        actname = "selection_printer:%s" % self.title
        desc = ida_kernwin.action_desc_t(actname, "Print selection: %s" % self.title, Select
        ida_kernwin.attach_dynamic_action_to_popup(None, popup_handle, desc)


def show_graph():
    f = ida_funcs.get_func(ida_kernwin.get_screen_ea())
    if not f:
        print("Must be in a function")
        return
    # Iterate through all function instructions and take only call instructions
    result = []
    tmp = ida_ua.insn_t()
    for x in [x for x in f if (ida_ua.decode_insn(tmp, x) and ida_idp.is_call_insn(tmp))]:
        xb = ida_xref.xrefblk_t()
        for xref in xb.refs_from(x, ida_xref.XREF_FAR):
            if not xref.iscode: continue
            t = ida_funcs.get_func_name(xref.to)
            if not t:
                t = hex(xref.to)
            result.append(t)
    g = MyGraph(ida_funcs.get_func_name(f.start_ea), result)
    if g.Show():
        return g
    else:
        return None

g = show_graph()
if g:
    print("Graph created and displayed!")
```

IDAPython example: widgets/graphs/sync_two_graphs.py

follow the movements of a disassembly graph, in another.

description: Since it is possible to be notified of movements that happen take place in a widget, it's possible to "replay" those movements in another.

In this case, “IDA View-B” (will be opened if necessary) will show the same contents as “IDA View-A”, slightly zoomed out.

keywords: graph, idaview

see_also: wrap_idaview

```
import ida_kernwin
import ida_moves
import ida_graph

#
# Cleanup (in case the script is run more than once)
#
try:
    wrap_a.Unbind()
except:
    pass


wrap_a = None

#
# The IDA View-A "monitor": changes will be reported into IDA View-B
#
class IDAViewA_monitor_t(ida_kernwin.IDAViewWrapper):
    def __init__(self):
        ida_kernwin.IDAViewWrapper.__init__(self, "IDA View-A")

    def OnViewLocationChanged(self, now, was):
        self.update_widget_b()

    def update_widget_b(self):

        # Make sure we are in the same function
        place_a, _, _ = ida_kernwin.get_custom_viewer_place(widget_a, False)
        ida_kernwin.jumpto(widget_b, place_a, -1, -1)

        # and that we show the right place (slightly zoomed out)
        widget_a_center_gli = ida_moves.graph_location_info_t()
        if ida_graph.viewer_get_gli(widget_a_center_gli, widget_a, ida_graph.GLICTL_CENTER):
            widget_b_center_gli = ida_moves.graph_location_info_t()
            widget_b_center_gli.orgx = widget_a_center_gli.orgx
            widget_b_center_gli.orgy = widget_a_center_gli.orgy
            widget_b_center_gli.zoom = widget_a_center_gli.zoom * 0.5
```

```
            ida_graph.viewer_set_gli(widget_b, widget_b_center_gli, ida_graph.GLICTL_CENTER)

#
# Make sure both views are opened...
#
for label in ["A", "B"]:
    title = "IDA View-%s" % label
    if not ida_kernwin.find_widget(title):
        print("View %s not available. Opening." % title)
        ida_kernwin.open_disasm_window(label)

#
# ...and that they are both in graph mode
#
widget_a = ida_kernwin.find_widget("IDA View-A")
ida_kernwin.set_view_renderer_type(widget_a, ida_kernwin.TCCRT_GRAPH)

widget_b = ida_kernwin.find_widget("IDA View-B")
ida_kernwin.set_view_renderer_type(widget_b, ida_kernwin.TCCRT_GRAPH)

#
# Put view B to the right of view A
#
ida_kernwin.set_dock_pos("IDA View-B", "IDA View-A", ida_kernwin.DP_RIGHT)

#
# Start monitoring IDA View-A
#
wrap_a = IDAViewA_monitor_t()
wrap_a.Bind()

#
# This is to get a properly initialized set of views to begin with.
# At this point, all UI resize/move events resulting of 'set_dock_pos()'
# haven't yet been processed, and thus the views don't know their final
# geometries. We'll give them a bit of time to process those events, and
# then we'll request that "IDA View-A" shows the whole graph (and
# "IDA View-B" will obviously follow.)
#
def fit_widget_a():
    def do_fit_widget_a():
        ida_graph.viewer_fit_window(widget_a)
    ida_kernwin.execute_sync(do_fit_widget_a, ida_kernwin.MFF_FAST)
import threading
threading.Timer(0.25, fit_widget_a).start()
```

IDAPython example: widgets/idaview/wrap_idaview.py

manipulate IDAView and graph

description: This is an example illustrating how to manipulate an existing IDA-provided view (and thus possibly its graph), in Python.

keywords: idaview, graph

see_also: custom_graph_with_actions, sync_two_graphs

```
# -----------------------------------------------------------------------
# (c) Hex-Rays

from time import sleep
import threading

import ida_kernwin
import ida_graph

class Worker(threading.Thread):
    def __init__(self, w):
        threading.Thread.__init__(self)
        self.w = w

    def log(self, msg):
        print(">>> thread: %s" % msg)

    def req_SetCurrentRendererType(self, switch_to):
        w = self.w
        def f():
            self.log("Switching to %s" % switch_to)
            w.SetCurrentRendererType(switch_to)
        ida_kernwin.execute_sync(f, ida_kernwin.MFF_FAST)

    def req_SetNodeInfo(self, node, info, flags):
        w = self.w
        def f():
            self.log("Setting node info..")
            w.SetNodeInfo(node, info, flags)
        ida_kernwin.execute_sync(f, ida_kernwin.MFF_FAST)

    def req_DelNodesInfos(self, *nodes):
        w = self.w
        def f():
            self.log("Deleting nodes infos..")
```

```
            w.DelNodesInfos(*nodes)
        ida_kernwin.execute_sync(f, ida_kernwin.MFF_FAST)

    def run(self):
        # Note, in order to leave the UI available
        # to the user, we'll perform UI operations
        # in this thread.
        #
        # But.
        #
        # Qt expects that all UI operations be performed from
        # the main thread. Therefore, we'll have to use
        # 'ida_kernwin.execute_sync' to send requests to the main thread.

        # Switch back & forth to & from graph view
        for i in range(3):
            self.req_SetCurrentRendererType(ida_kernwin.TCCRT_FLAT)
            sleep(1)
            self.req_SetCurrentRendererType(ida_kernwin.TCCRT_GRAPH)
            sleep(1)

        # Go to graph view, and set the first node's color
        self.req_SetCurrentRendererType(ida_kernwin.TCCRT_GRAPH)
        ni = ida_graph.node_info_t()
        ni.bg_color    = 0x00ff00ff
        ni.frame_color = 0x0000ff00
        self.req_SetNodeInfo(0, ni, ida_graph.NIF_BG_COLOR|ida_graph.NIF_FRAME_COLOR)
        sleep(3)

        # This was fun. But let's revert it.
        self.req_DelNodesInfos(0)
        sleep(3)

        self.log("Done.")


class MyIDAViewWrapper(ida_kernwin.IDAViewWrapper):
    # A wrapper around the standard IDA view wrapper.
    # We'll react to some events and print the parameters
    # that were sent to us, that's all.
    def __init__(self, viewName):
        ida_kernwin.IDAViewWrapper.__init__(self, viewName)

    # Helper function, to be called by "On*" event handlers.
    # This will print all the arguments that were passed!
    def printPrevFrame(self):
```

```
        import inspect
        stack = inspect.stack()
        frame, _, _, _, _, _ = stack[1]
        args, _, _, values = inspect.getargvalues(frame)
        print("EVENT: %s: args=%s" % (
            inspect.getframeinfo(frame)[2],
            [(i, values[i]) for i in args[1:]]))

    def OnViewKeydown(self, key, state):
        self.printPrevFrame()

    def OnViewClick(self, x, y, state):
        self.printPrevFrame()

    def OnViewDblclick(self, x, y, state):
        self.printPrevFrame()

    def OnViewSwitched(self, rt):
        self.printPrevFrame()

    def OnViewMouseOver(self, x, y, state, over_type, over_data):
        self.printPrevFrame()



viewName = "IDA View-A"
w = MyIDAViewWrapper(viewName)
if w.Bind():
    print("Succesfully bound to %s" % viewName)

    # We'll launch the sequence of operations in another thread,
    # so that sleep() calls don't freeze the UI
    worker = Worker(w)
    worker.start()

else:
    print("Couldn't bind to view %s. Is it available?" % viewName)
```

IDAPython example: widgets/listings/custom_viewer.py

create custom listings in IDA

description: How to create simple listings, that will share many of the features as the built-in IDA widgets (highlighting, copy & paste, notifications, …)

In addition, creates actions that will be bound to the freshly-created widget (using `ida_kernwin.attach_action_to_popup`.)

keywords: listing, actions

```
import ida_kernwin
import ida_lines

# -----------------------------------------------------------------------
class say_something_handler_t(ida_kernwin.action_handler_t):
    def __init__(self, thing):
        ida_kernwin.action_handler_t.__init__(self)
        self.thing = thing

    def activate(self, ctx):
        print(self.thing)

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_ALWAYS

    @staticmethod
    def compose_action_name(v):
        return "custview:say_%s" % v


# create actions
actions_variants = ["Hello", "World"]
for av in actions_variants:
    actname = say_something_handler_t.compose_action_name(av)
    if ida_kernwin.unregister_action(actname):
        print("Unregistered previously-registered action \"%s\"" % actname)

    desc = ida_kernwin.action_desc_t(actname, "Say %s" % av, say_something_handler_t(av))
    if ida_kernwin.register_action(desc):
        print("Registered action \"%s\"" % actname)



# -----------------------------------------------------------------------
class mycv_t(ida_kernwin.simplecustviewer_t):
    def Create(self, sn=None, use_colors=True):
        # Form the title
        title = "Simple custom view test"
        if sn:
            title += " %d" % sn
        self.use_colors = use_colors
```

```
        # Create the customviewer
        if not ida_kernwin.simplecustviewer_t.Create(self, title):
            return False

        for i in range(0, 100):
            prefix, bg = ida_lines.COLOR_DEFAULT, None
            # make every 10th line a bit special
            if i % 10 == 0:
                prefix = ida_lines.COLOR_DNAME   # i.e., dark yellow...
                bg = 0xFFFF00                    # ...on cyan
            pfx = ida_lines.COLSTR("%3d" % i, ida_lines.SCOLOR_PREFIX)
            if self.use_colors:
                self.AddLine("%s: Line %d" % (pfx, i), fgcolor=prefix, bgcolor=bg)
            else:
                self.AddLine("%s: Line %d" % (pfx, i))

        return True

    def OnClick(self, shift):
        """
        User clicked in the view
        @param shift: Shift flag
        @return: Boolean. True if you handled the event
        """
        print("OnClick, shift=%d" % shift)
        return True

    def OnDblClick(self, shift):
        """
        User dbl-clicked in the view
        @param shift: Shift flag
        @return: Boolean. True if you handled the event
        """
        word = self.GetCurrentWord()
        if not word: word = "<None>"
        print("OnDblClick, shift=%d, current word=%s" % (shift, word))
        return True

    def OnCursorPosChanged(self):
        """
        Cursor position changed.
        @return: Nothing
        """
        print("OnCurposChanged")

    def OnClose(self):
```

```
        """
        The view is closing. Use this event to cleanup.
        @return: Nothing
        """
        print("OnClose " + self.title)

    def OnKeydown(self, vkey, shift):
        """
        User pressed a key
        @param vkey: Virtual key code
        @param shift: Shift flag
        @return: Boolean. True if you handled the event
        """
        print("OnKeydown, vk=%d shift=%d" % (vkey, shift))
        # ESCAPE?
        if vkey == 27:
            self.Close()
        # VK_DELETE
        elif vkey == 46:
            n = self.GetLineNo()
            if n is not None:
                self.DelLine(n)
                self.Refresh()
                print("Deleted line %d" % n)
        # Goto?
        elif vkey == ord('G'):
            n = self.GetLineNo()
            if n is not None:
                v = ida_kernwin.ask_long(self.GetLineNo(), "Where to go?")
                if v:
                    self.Jump(v, 0, 5)
        elif vkey == ord('R'):
            print("refreshing....")
            self.Refresh()
        elif vkey == ord('C'):
            print("refreshing current line...")
            self.RefreshCurrent()
        elif vkey == ord('A'):
            s = ida_kernwin.ask_str("NewLine%d" % self.Count(), 0, "Append new line")
            self.AddLine(s)
            self.Refresh()
        elif vkey == ord('X'):
            print("Clearing all lines")
            self.ClearLines()
            self.Refresh()
        elif vkey == ord('I'):
```

```
            n = self.GetLineNo()
            s = ida_kernwin.ask_str("InsertedLine%d" % n, 0, "Insert new line")
            self.InsertLine(n, s)
            self.Refresh()
        elif vkey == ord('E'):
            l = self.GetCurrentLine(notags=1)
            if not l:
                return False
            n = self.GetLineNo()
            print("curline=<%s>" % l)
            l = l + ida_lines.COLSTR("*", ida_lines.SCOLOR_VOIDOP)
            self.EditLine(n, l)
            self.RefreshCurrent()
            print("Edited line %d" % n)
        else:
            return False
        return True

    def OnHint(self, lineno):
        """
        Hint requested for the given line number.
        @param lineno: The line number (zero based)
        @return:
            - tuple(number of important lines, hint string)
            - None: if no hint available
        """
        return (1, "OnHint, line=%d" % lineno)

    def Show(self, *args):
        ok = ida_kernwin.simplecustviewer_t.Show(self, *args)
        if ok:
            # permanently attach actions to this viewer's popup menu
            for av in actions_variants:
                actname = say_something_handler_t.compose_action_name(av)
                ida_kernwin.attach_action_to_popup(self.GetWidget(), None, actname)
        return ok

# -----------------------------------------------------------------------
try:
    # created already?
    mycv
    print("Already created, will close it...")
    mycv.Close()
    del mycv
except:
    pass
```

```
def show_win():
    x = mycv_t()
    if not x.Create():
        print("Failed to create!")
        return None
    x.Show()
    tcc = x.GetWidget()
    return x

mycv = show_win()
if not mycv:
    del mycv

def make_many(n):
    L = []
    for i in range(1, n+1):
        v = mycv_t()
        if not v.Create(i):
            break
        v.Show()
        L.append(v)
    return L
```

IDAPython example: widgets/listings/jump_next_comment.py

implement a "jump to next comment" action within IDA's disassembly view.

description: We want our action not only to find the next line containing a comment, but to also place the cursor at the right horizontal position.

To find that position, we will have to inspect the text that IDA generates, looking for the start of a comment. However, we won't be looking for a comment "prefix" (e.g., ";"), as that would be too fragile.

Instead, we will look for special "tags" that IDA injects into textual lines, and that bear semantic information.

Those tags are primarily used for rendering (i.e., switching colors), but can also be very handy for spotting tokens of interest (registers, addresses, comments, prefixes, instruction mnemonics, …)

keywords: idaview, actions

see_also: save_and_restore_listing_pos

```
import ida_idaapi
```

```
import ida_kernwin
import ida_bytes
import ida_moves
import ida_lines

def find_comment_visual_position_in_tagged_line(line):
    """
    We'll look for tags for all types of comments, and if
    found return the visual position of the tag in the line
    (using 'ida_lines.tag_strlen')
    """
    for cmt_type in [
            ida_lines.SCOLOR_REGCMT,
            ida_lines.SCOLOR_RPTCMT,
            ida_lines.SCOLOR_AUTOCMT]:
        cmt_idx = line.find(ida_lines.SCOLOR_ON + cmt_type)
        if cmt_idx > -1:
            return ida_lines.tag_strlen(line[:cmt_idx])
    return -1


def jump_next_comment(v):
    """
    Starting at the current line, keep generating lines until
    a comment is found. When this happens, position the viewer
    at the right coordinates.
    """
    loc = ida_moves.lochist_entry_t()
    if ida_kernwin.get_custom_viewer_location(loc, v):
        place = loc.place()
        idaplace = ida_kernwin.place_t_as_idaplace_t(place)
        ea = idaplace.ea
        while ea != ida_idaapi.BADADDR:
            _, disass = ida_lines.generate_disassembly(
                ea,
                1000,  # maximum number of lines
                False, # as_stack=False
                False) # notags=False - we want tags, in order to spot comments

            found = None

            # If this is the start item, start at the next line
            start_lnnum = (idaplace.lnnum + 1) if ea == idaplace.ea else 0

            for rel_lnnum, line in enumerate(disass[start_lnnum:]):
                vis_cx = find_comment_visual_position_in_tagged_line(line)
```

```
                if vis_cx > -1:
                    found = (ea, rel_lnnum, vis_cx)
                    break

            if found is not None:
                idaplace.ea = found[0]
                idaplace.lnnum = start_lnnum + found[1]
                loc.set_place(idaplace)
                loc.renderer_info().pos.cx = found[2]
                ida_kernwin.custom_viewer_jump(v, loc, ida_kernwin.CVNF_LAZY)
                break

            ea = ida_bytes.next_head(ea, ida_idaapi.BADADDR)


class jump_next_comment_ah_t(ida_kernwin.action_handler_t):
    def activate(self, ctx):
        jump_next_comment(ctx.widget)

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_FOR_WIDGET \
            if ctx.widget_type == ida_kernwin.BWN_DISASM \
            else ida_kernwin.AST_DISABLE_FOR_WIDGET


ACTION_NAME = "jump_next_comment:jump"
ACTION_LABEL = "Jump to the next comment"
ACTION_SHORTCUT = "Ctrl+Alt+C"
ACTION_HELP = "Press %s to jump to the next comment" % ACTION_SHORTCUT

if ida_kernwin.unregister_action(ACTION_NAME):
    print("Unregistered previously-registered action \"%s\"" % ACTION_LABEL)

if ida_kernwin.register_action(
        ida_kernwin.action_desc_t(
            ACTION_NAME,
            ACTION_LABEL,
            jump_next_comment_ah_t(),
            ACTION_SHORTCUT)):
    print("Registered action \"%s\". %s" % (ACTION_LABEL, ACTION_HELP))
```

IDAPython example: widgets/listings/save_and_restore_listing_pos.py

save, and then restore, positions in a listing

description: Shows how it is possible re-implement IDA's bookmark capability, using 2 custom actions: one action saves the current location, and the other restores it.

Note that, contrary to actual bookmarks, this example:

```
* remembers only 1 saved position
* doesn't save that position in the IDB (and therefore cannot
  be restored if IDA is closed & reopened.)
```

keywords: listing, actions

see_also: jump_next_comment

```
import ida_kernwin
import ida_moves

class listing_action_handler_t(ida_kernwin.action_handler_t):
    def update(self, ctx):
        is_listing = ctx.widget_type in [
            ida_kernwin.BWN_ENUMS,
            ida_kernwin.BWN_STRUCTS,
            ida_kernwin.BWN_DISASM,
            ida_kernwin.BWN_CUSTVIEW,
            ida_kernwin.BWN_PSEUDOCODE,
        ]
        return ida_kernwin.AST_ENABLE_FOR_WIDGET if is_listing else ida_kernwin.AST_DISABLE_

class last_pos_t(object):
    def __init__(self, widget_title, lochist_entry):
        self.widget_title = widget_title
        self.lochist_entry = lochist_entry

last_pos = None

class save_position_ah_t(listing_action_handler_t):

    ACTION_NAME = "save_and_restore_listing_pos:save_position"
    ACTION_LABEL = "Save position"
    ACTION_SHORTCUT = "Ctrl+Shift+S"
    HELP_TEXT = "Press %s in a 'listing' widget such as 'IDA View-A', 'Enums', 'Structures',

    def activate(self, ctx):
        global last_pos
        e = ida_moves.lochist_entry_t()
        if ida_kernwin.get_custom_viewer_location(e, ctx.widget):
```

```
            last_pos = last_pos_t(ctx.widget_title, e)
        else:
            print("Failed to retrieve position")


class restore_position_ah_t(listing_action_handler_t):

    ACTION_NAME = "save_and_restore_listing_pos:restore_position"
    ACTION_LABEL = "Restore position"
    ACTION_SHORTCUT = "Ctrl+Shift+O"
    HELP_TEXT = "Press %s in a 'listing' widget such as 'IDA View-A', 'Enums', 'Structures',

    def activate(self, ctx):
        global last_pos
        if last_pos:
            w = ida_kernwin.find_widget(last_pos.widget_title)
            if w:
                ida_kernwin.custom_viewer_jump(w, last_pos.lochist_entry)
            else:
                print("Widget \"%s\" not found" % last_pos.widget_title)
        else:
            print("No last position to restore")


klasses = [
    save_position_ah_t,
    restore_position_ah_t,
]

for klass in klasses:
    if ida_kernwin.unregister_action(klass.ACTION_NAME):
        print("Unregistered previously-registered action \"%s\"" % klass.ACTION_LABEL)

    if ida_kernwin.register_action(
            ida_kernwin.action_desc_t(
                klass.ACTION_NAME,
                klass.ACTION_LABEL,
                klass(),
                klass.ACTION_SHORTCUT)):
        print("Registered action \"%s\". %s" % (klass.ACTION_LABEL, klass.HELP_TEXT))
```

IDAPython example: widgets/misc/add_menus.py

adding custom menus to IDA

description: It is possible to add custom menus to IDA, either at the toplevel (i.e., into the menubar), or as submenus of existing menus.

Notes:

```
* the same action can be present in more than 1 menu
* this example does not deal with context menus
```

keywords: actions

```
import ida_kernwin

# Create custom menus
ida_kernwin.create_menu("MyToplevelMenu", "&Custom menu", "View")
ida_kernwin.create_menu("MySubMenu", "Custom s&ubmenu", "View/Print internal flags")

# Create some actions
class greeter_t(ida_kernwin.action_handler_t):
    def __init__(self, greetings):
        ida_kernwin.action_handler_t.__init__(self)
        self.greetings = greetings

    def activate(self, ctx):
        print(self.greetings)

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_ALWAYS

ACTION_NAME_0 = "my_action_0"
ACTION_NAME_1 = "my_action_1"
for action_name, greetings in [
        (ACTION_NAME_0, "Hello, world"),
        (ACTION_NAME_1, "Hi there"),
]:
    desc = ida_kernwin.action_desc_t(
        action_name, "Say \"%s\"" % greetings, greeter_t(greetings))
    if ida_kernwin.register_action(desc):
        print("Registered action \"%s\"" % action_name)


# Then, let's attach some actions to them - both core actions
# and custom ones is allowed (also, any action can be attached
# to multiple menus.)
for action_name, path in [
        (ACTION_NAME_0, "Custom menu"),
        (ACTION_NAME_0, "View/Custom submenu/"),
```

```
        (ACTION_NAME_1, "Custom menu"),
        (ACTION_NAME_1, "View/Custom submenu/"),
        ("About", "Custom menu"),
        ("About", "View/Custom submenu/"),
]:
    ida_kernwin.attach_action_to_menu(
        path,
        action_name,
        ida_kernwin.SETMENU_INS)
```

IDAPython example: widgets/tabular_views/custom/choose.py

A widget showing data in a tabular fashion

description: Shows how to subclass the ida_kernwin.Choose class to show data organized in a simple table. In addition, registers a couple actions that can be applied to it.

keywords: chooser, actions

see_also: choose_multi, chooser_with_folders

```
import ida_kernwin
from ida_kernwin import Choose

# -----------------------------------------------------------------------
class chooser_handler_t(ida_kernwin.action_handler_t):
    def __init__(self, thing):
        ida_kernwin.action_handler_t.__init__(self)
        self.thing = thing

    def activate(self, ctx):
        sel = []
        for idx in ctx.chooser_selection:
            sel.append(str(idx))
        print("command %s selected @ %s" % (self.thing, ", ".join(sel)))

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_FOR_WIDGET \
            if ida_kernwin.is_chooser_widget(ctx.widget_type) \
          else ida_kernwin.AST_DISABLE_FOR_WIDGET

    @staticmethod
    def compose_action_name(v):
        return "choose:act%s" % v
```

```
# create actions
actions_variants = ["A", "B"]
for av in actions_variants:
    actname = chooser_handler_t.compose_action_name(av)
    if ida_kernwin.unregister_action(actname):
        print("Unregistered previously-registered action \"%s\"" % actname)

    desc = ida_kernwin.action_desc_t(actname, "command %s" % av, chooser_handler_t(av))
    if ida_kernwin.register_action(desc):
        print("Registered action \"%s\"" % actname)


# -----------------------------------------------------------------------
class MyChoose(Choose):

    def __init__(self, title, nb = 5, flags = 0,
                 modal = False,
                 embedded = False, width = None, height = None):
        Choose.__init__(
            self,
            title,
            [ ["Address", 10], ["Name", 30] ],
            flags = flags | Choose.CH_RESTORE
                          | (Choose.CH_CAN_INS
                           | Choose.CH_CAN_DEL
                           | Choose.CH_CAN_EDIT
                           | Choose.CH_CAN_REFRESH),
            embedded = embedded,
            width = width,
            height = height)
        self.n = 0
        self.items = [ self.make_item() for x in range(nb) ]
        self.icon = 5
        self.selcount = 0
        self.modal = modal
        self.popup_names = ["Inzert", "Del leet", "Ehdeet", "Ree frech"]

        print("created %s" % str(self))

    def OnInit(self):
        print("inited", str(self))
        return True

    def OnGetSize(self):
        n = len(self.items)
```

```
        print("getsize -> %d" % n)
        return n

    def OnGetLine(self, n):
        print("getline %d" % n)
        return self.items[n]

    def OnGetIcon(self, n):
        r = self.items[n]
        t = self.icon + r[1].count("*")
        print("geticon", n, t)
        return t

    def OnGetLineAttr(self, n):
        print("getlineattr %d" % n)
        if n == 1:
            return [0xFF0000, 0]

    def OnInsertLine(self, n):
        # we ignore current selection
        n = self.n # position at the just added item
        self.items.append(self.make_item())
        print("insert line")
        return (Choose.ALL_CHANGED, n)

    def OnDeleteLine(self, n):
        print("del %d " % n)
        del self.items[n]
        return [Choose.ALL_CHANGED] + self.adjust_last_item(n)

    def OnEditLine(self, n):
        self.items[n][1] = self.items[n][1] + "*"
        print("editing %d" % n)
        return (Choose.ALL_CHANGED, n)

    def OnRefresh(self, n):
        print("refresh %d" % n)
        return None # call standard refresh

    def OnSelectLine(self, n):
        self.selcount += 1
        warning("[%02d] selectline '%d'" % (self.selcount, n))
        return (Choose.NOTHING_CHANGED, )

    def OnClose(self):
        print("closed", str(self))
```

```
    def show(self):
        ok = self.Show(self.modal) >= 0
        if ok:
            # permanently attach actions to this chooser's popup menu
            for av in actions_variants:
                actname = chooser_handler_t.compose_action_name(av)
                ida_kernwin.attach_action_to_popup(self.GetWidget(), None, actname)
        return ok

    def make_item(self):
        r = [str(self.n), "func_%04d" % self.n]
        self.n += 1
        return r


# -----------------------------------------------------------------------
def test_choose(modal = False, nb = 10):
    global c
    c = MyChoose("Choose - sample 1", nb = nb, modal = modal)
    c.show()

# -----------------------------------------------------------------------
def test_choose_embedded():
    global c
    c = MyChoose("Choose - embedded", nb=12, embedded = True, width=123, height=222)
    r = c.Embedded()
    if r == 0:
        try:
            if test_embedded:
                o, sel = _idaapi.choose_get_embedded(c)
                print("o=%s, type(o)=%s" % (str(o), type(o)))
                test_embedded(o)
        finally:
            c.Close()

# -----------------------------------------------------------------------
if __name__ == '__main__':
    #test_choose_embedded()
    test_choose(False)
```

IDAPython example: widgets/tabular_views/custom/chooser_with_folders.py

A widget that can show tabular data either as a simple table, or with a tree-like structure.

description: By adding the necessary bits to a ida_kernwin.Choose subclass, IDA can show the otherwise tabular data, in a tree-like fashion.

The important bits to enable this are:

```
* ida_dirtree.dirspec_t (and my_dirspec_t)
* ida_kernwin.CH_HAS_DIRTREE
* ida_kernwin.Choose.OnGetDirTree
* ida_kernwin.Choose.OnIndexToInode
```

keywords: chooser, folders, actions

see_also: choose, choose_multi

```
import inspect

import ida_kernwin
import ida_dirtree
import ida_netnode

class my_dirspec_t(ida_dirtree.dirspec_t):

    def __init__(self, chooser):
        ida_dirtree.dirspec_t.__init__(self)
        self.chooser = chooser

    def log_frame(self):
        if self.chooser.dirspec_log:
            stack = inspect.stack()
            frame, _, _, _, _, _ = stack[1]
            args, _, _, values = inspect.getargvalues(frame)
            print(">>> %s: args=%s" % (inspect.getframeinfo(frame)[2], [(i, values[i]) for i

    def get_name(self, inode, flags):
        self.log_frame()
        def find_inode(index, ordinal, _inode):
            if inode == _inode:
                return "inode #%d" % inode
        return self.chooser._for_each_item(find_inode)

    def get_inode(self, dirpath, name):
        self.log_frame()
        if not name.startswith("inode #"):
            return ida_dirtree.direntry_t.BADIDX
        return int(name[7:])
```

```
    def get_size(self, inode):
        self.log_frame()
        return 1

    def get_attrs(self, inode):
        self.log_frame()

    def rename_inode(self, inode, newname):
        self.log_frame()
        def set_column0_contents(index, ordinal, _inode):
            if inode == _inode:
                ordinal = self.chooser._get_ordinal_at(index)
                self.chooser.netnode.supset(index, newname, SUPVAL_COL0_DATA_TAG)
                return True
        return self.chooser._for_each_item(set_column0_contents)

    def unlink_inode(self, inode):
        self.log_frame()

ALTVAL_NEW_ORDINAL_TAG = 'L'
ALTVAL_ORDINAL_TAG = 'O'
ALTVAL_INODE_TAG = 'I'
SUPVAL_COL0_DATA_TAG = '0'
SUPVAL_COL1_DATA_TAG = '1'
SUPVAL_COL2_DATA_TAG = '2'


class base_idapython_tree_view_t(ida_kernwin.Choose):

    def __init__(self, title, nitems=100, dirspec_log=True, flags=0):
        flags |= ida_kernwin.CH_MULTI
        flags |= ida_kernwin.CH_HAS_DIRTREE
        ida_kernwin.Choose.__init__(self,
                        title,
                        [
                            ["First",
                             10
                           | ida_kernwin.Choose.CHCOL_PLAIN
                           | ida_kernwin.Choose.CHCOL_DRAGHINT
                           | ida_kernwin.Choose.CHCOL_INODENAME
                            ],
                            ["Second", 10 | ida_kernwin.Choose.CHCOL_PLAIN],
                            ["Third", 10 | ida_kernwin.Choose.CHCOL_PLAIN],
                        ],
                        flags=flags)
        self.debug_items = False
```

```
        self.dirspec_log = dirspec_log
        self.dirtree = None
        self.dirspec = None
        self.netnode = ida_netnode.netnode()
        self.netnode.create("$ idapython_tree_view %s" % title)
        for i in range(nitems):
            self._new_item()

    def _get_new_ordinal(self):
        return self.netnode.altval(0, ALTVAL_NEW_ORDINAL_TAG)

    def _set_new_ordinal(self, ordinal):
        self.netnode.altset(0, ordinal, ALTVAL_NEW_ORDINAL_TAG)

    def _allocate_ordinal(self):
        ordinal = self._get_new_ordinal()
        self._set_new_ordinal(ordinal + 1)
        return ordinal

    def _move_items(self, src, dst, sz):
        self.netnode.altshift(src, dst, sz, ALTVAL_ORDINAL_TAG)
        self.netnode.altshift(src, dst, sz, ALTVAL_INODE_TAG)
        self.netnode.supshift(src, dst, sz, SUPVAL_COL0_DATA_TAG)
        self.netnode.supshift(src, dst, sz, SUPVAL_COL1_DATA_TAG)
        self.netnode.supshift(src, dst, sz, SUPVAL_COL2_DATA_TAG)

    def _new_item(self, index=None):
        new_ord = self._allocate_ordinal()
        new_inode = new_ord + 1000
        nitems = self._get_items_count()
        if index is None:
            index = nitems
        else:
            assert(index < nitems)
        if index < nitems:
            self._move_items(index, index + 1, nitems - index)
        self.netnode.altset(index, new_ord, ALTVAL_ORDINAL_TAG)
        self.netnode.altset(index, new_inode, ALTVAL_INODE_TAG)
        return index, new_ord, new_inode

    def _dump_items(self):
        if self.debug_items:
            data = []
            def collect(index, ordinal, inode):
                data.append([inode] + self._make_item_contents_from_index(index))
            self._for_each_item(collect)
```

```
            import pprint
            print(pprint.pformat(data))

    def _get_ordinal_at(self, index):
        assert(index <= self.netnode.altlast(ALTVAL_ORDINAL_TAG))
        return self.netnode.altval(index, ALTVAL_ORDINAL_TAG)

    def _get_inode_at(self, index):
        assert(index <= self.netnode.altlast(ALTVAL_INODE_TAG))
        return self.netnode.altval(index, ALTVAL_INODE_TAG)

    def _for_each_item(self, cb):
        for i in range(self._get_items_count()):
            rc = cb(i, self._get_ordinal_at(i), self._get_inode_at(i))
            if rc is not None:
                return rc

    def _get_items_count(self):
        l = self.netnode.altlast(ALTVAL_ORDINAL_TAG)
        return 0 if l == ida_netnode.BADNODE else l + 1

    def _make_item_contents_from_index(self, index):
        ordinal = self._get_ordinal_at(index)
        c0 = self.netnode.supstr(index, SUPVAL_COL0_DATA_TAG) or "a%d" % ordinal
        c1 = self.netnode.supstr(index, SUPVAL_COL1_DATA_TAG) or "b%d" % ordinal
        c2 = self.netnode.supstr(index, SUPVAL_COL2_DATA_TAG) or "c%d" % ordinal
        return [c0, c1, c2]

    def OnGetLine(self, n):
        return self._make_item_contents_from_index(n)

    def OnGetSize(self):
        return self._get_items_count()

    def OnGetDirTree(self):
        self.dirspec = my_dirspec_t(self)
        self.dirtree = ida_dirtree.dirtree_t(self.dirspec)
        def do_link(index, ordinal, inode):
            de = ida_dirtree.direntry_t(inode, False)
            self.dirtree.link("/%s" % self.dirtree.get_entry_name(de))
        self._for_each_item(do_link)
        return (self.dirspec, self.dirtree)

    def OnIndexToInode(self, n):
        return self._get_inode_at(n)
```

```
    # Helper function, to be called by "On*" event handlers.
    # This will print all the arguments that were passed
    def _print_prev_frame(self):
        import inspect
        stack = inspect.stack()
        frame, _, _, _, _, _ = stack[1]
        args, _, _, values = inspect.getargvalues(frame)
        print("EVENT: %s: args=%s" % (
            inspect.getframeinfo(frame)[2],
            [(i, values[i]) for i in args[1:]]))

    def OnSelectionChange(self, sel):
        self._print_prev_frame()

    def OnSelectLine(self, sel):
        self._print_prev_frame()


class idapython_tree_view_t(base_idapython_tree_view_t):

    def __init__(self, title, nitems=100, dirspec_log=True, flags=0):
        flags |= ida_kernwin.CH_CAN_INS
        flags |= ida_kernwin.CH_CAN_DEL
        flags |= ida_kernwin.CH_CAN_EDIT
        base_idapython_tree_view_t.__init__(self, title, nitems, dirspec_log, flags)

    def OnInsertLine(self, sel):
        self._print_prev_frame()

        # Add item into storage
        index = sel[0] if sel else None
        prev_inode = self._get_inode_at(index) if index is not None else None
        final_index, new_ordinal, new_inode = self._new_item(sel[0] if sel else None)

        # Link in the tree (unless an absolute path is provided,
        # 'link()' will use the current directory, which is set
        # by the 'OnInsertLine' caller.)
        dt = self.dirtree
        cwd = dt.getcwd()
        parent_de = dt.resolve_path(cwd)
        wanted_rank = -1
        if prev_inode is not None:
            wanted_rank = dt.get_rank(parent_de.idx, ida_dirtree.direntry_t(prev_inode, Fals
        de = ida_dirtree.direntry_t(new_inode, False)
        name = dt.get_entry_name(de)
        code = dt.link(name)
```

```
            assert(code == ida_dirtree.DTE_OK)
            if wanted_rank >= 0:
                assert(ida_dirtree.dirtree_t.isdir(parent_de))
                cur_rank = dt.get_rank(parent_de.idx, de)
                dt.change_rank(cwd + "/" + name, wanted_rank - cur_rank)
            self._dump_items()
            return [ida_kernwin.Choose.ALL_CHANGED] + [final_index]

        def OnDeleteLine(self, sel):
            self._print_prev_frame()
            dt = self.dirtree
            for index in reversed(sorted(sel)):
                # Note: when it comes to deletion of items, the dirtree_t is
                # designed in such a way folders contents will be re-computed
                # on-demand after the deletion of an inode. Consequently,
                # there is no need to perform an unlink() operation here, only
                # notify the dirtree that something changed
                nitems = self._get_items_count()
                assert(index < nitems)
                inode = self._get_inode_at(index)
                self.netnode.altdel(index, ALTVAL_ORDINAL_TAG)
                self.netnode.altdel(index, ALTVAL_INODE_TAG)
                self._move_items(index + 1, index, nitems - index + 1)
                dt.notify_dirtree(False, inode)
            self._dump_items()
            return [ida_kernwin.Choose.ALL_CHANGED]

        def OnEditLine(self, sel):
            self._print_prev_frame()
            for idx in sel:
                repl = ida_kernwin.ask_str("", 0, "Please enter replacement for index %d" % idx)
                if repl:
                    self.netnode.supset(idx, repl, SUPVAL_COLO_DATA_TAG)
            self._dump_items()
            return [ida_kernwin.Choose.ALL_CHANGED] + sel

    # --------------------------------------------------------------------------
    if __name__ == '__main__':
        form = idapython_tree_view_t("idapython_tree_view_t test", 100)
        form.Show()
```

IDAPython example: widgets/tabular_views/custom/choose_multi.py

A widget showing data in a tabular fashion, providing multiple selection

description: Similar to @{choose}, but with multiple selection

keywords: chooser, actions

see_also: choose, chooser_with_folders

```
from ida_kernwin import Choose

class MyChoose(Choose):

    def __init__(self, title, nb = 5):
        Choose.__init__(
            self,
            title,
            [ ["Bit", Choose.CHCOL_HEX | 10] ],
            flags = Choose.CH_MULTI)
        self.items = [ str(1 << x) for x in range(nb) ]

    def OnGetSize(self):
        return len(self.items)

    def OnGetLine(self, n):
        return [self.items[n]]

    def OnSelectLine(self, n):
        self.deflt = n  # save current selection
        return (Choose.NOTHING_CHANGED, )

    def OnDeleteLine(self, indices):
        new_items = []
        for idx, item in enumerate(self.items):
            if idx not in indices:
                new_items.append(item)
        self.items = new_items
        return [Choose.ALL_CHANGED] + indices

    def show(self, num):
        self.deflt = [x
                      for x in range(len(self.items))
                      if (num & (1 << x)) != 0]
        if self.Show(True) < 0:
            return 0
        return sum([(1 << x) for x in self.deflt])


# -----------------------------------------------------------------------
def test_choose(num):
```

```
    c = MyChoose("Choose - sample 2", nb = 5)
    return c.show(num)

# --------------------------------------------------------------------------
if __name__ == '__main__':
    print(test_choose(11))
```

IDAPython example: widgets/tabular_views/custom/func_chooser.py

An alternative view over the list of functions

description: Partially re-implements the "Functions" widget present in IDA, with a custom widget.

keywords: chooser, functions

see_also: choose, choose_multi, chooser_with_folders

```
import idautils
import idc
import ida_funcs
import ida_kernwin

class my_funcs_t(ida_kernwin.Choose):

    def __init__(self, title):
        ida_kernwin.Choose.__init__(
            self,
            title,
            [ ["Address", 10 | ida_kernwin.Choose.CHCOL_HEX],
              ["Name",    30 | ida_kernwin.Choose.CHCOL_PLAIN | ida_kernwin.Choose.CHCOL_FNA
        self.items = []
        self.icon = 41

    def OnInit(self):
        self.items = [ [hex(x), ida_funcs.get_func_name(x), x]
                       for x in idautils.Functions() ]
        return True

    def OnGetSize(self):
        return len(self.items)

    def OnGetLine(self, n):
        return self.items[n]

    def OnDeleteLine(self, n):
```

```
        ea = self.items[n][2]
        idc.del_func(ea)
        return (ida_kernwin.Choose.ALL_CHANGED, n)

    def OnGetEA(self, n):
        return self.items[n][2]

    def OnRefresh(self, n):
        self.OnInit()
        # try to preserve the cursor
        return [ida_kernwin.Choose.ALL_CHANGED] + self.adjust_last_item(n)

    def OnClose(self):
        print("closed ", self.title)


def show_my_funcs_t(modal=False):
    c = my_funcs_t("My functions list")
    c.Show(modal=modal)

if __name__ == "__main__":
    show_my_funcs_t()
```

IDAPython example: widgets/tabular_views/string_window/show_selected_str

retrieve the strings that are selected in the “Strings” window.

description: In IDA it’s possible to write actions that can be applied even to core (i.e., “standard”) widgets. The actions in this example use the action “context” to know what the current selection is.

This example shows how you can either retrieve string literals data directly from the chooser (`ida_kernwin.get_chooser_data`), or by querying the IDB (`ida_bytes.get_strlit_contents`)

keywords: actions

see_also: list_strings

```
import ida_kernwin
import ida_strlist
import ida_bytes

class show_strings_base_ah_t(ida_kernwin.action_handler_t):
```

```
    def __init__(self, use_get_chooser_data):
        ida_kernwin.action_handler_t.__init__(self)
        self.use_get_chooser_data = use_get_chooser_data

    def activate(self, ctx):
        for idx in ctx.chooser_selection:
            if self.use_get_chooser_data:
                _, _, _, s = ida_kernwin.get_chooser_data(ctx.widget_title, idx)
            else:
                si = ida_strlist.string_info_t()
                if ida_strlist.get_strlist_item(si, idx):
                    s = ida_bytes.get_strlit_contents(si.ea, si.length, si.type)
            print("Selected string (retrieved using %s) at index %d: \"%s\"" % (
                "get_chooser_data()" if self.use_get_chooser_data else "get_strlist_item()",
                idx,
                s))
        return 0

    def update(self, ctx):
        return ida_kernwin.AST_ENABLE_FOR_WIDGET \
            if ctx.widget_type == ida_kernwin.BWN_STRINGS \
            else ida_kernwin.AST_DISABLE_FOR_WIDGET


class show_strings_using_get_chooser_data_ah_t(show_strings_base_ah_t):
    ACTION_NAME = "test:show_string_using_get_chooser_data"
    ACTION_LABEL = "Show current string(s) using get_chooser_data()"
    ACTION_SHORTCUT = "Ctrl+Shift+S"

    def __init__(self):
        show_strings_base_ah_t.__init__(self, True)


class show_strings_using_get_strlist_item_ah_t(show_strings_base_ah_t):
    ACTION_NAME = "test:show_string_using_get_strlist_item"
    ACTION_LABEL = "Show current string(s) using get_strlist_item() + get_strlit_contents()"
    ACTION_SHORTCUT = "Ctrl+Shift+K"

    def __init__(self):
        show_strings_base_ah_t.__init__(self, False)


klasses = [
    show_strings_using_get_chooser_data_ah_t,
    show_strings_using_get_strlist_item_ah_t,
]
```

```
sw = ida_kernwin.find_widget("Strings")
if not sw:
    sw = ida_kernwin.open_strings_window(ida_idaapi.BADADDR)

for klass in klasses:
    if ida_kernwin.unregister_action(klass.ACTION_NAME):
        print("Unregistered previously-registered action \"%s\"" % klass.ACTION_LABEL)

    if ida_kernwin.register_action(
            ida_kernwin.action_desc_t(
                klass.ACTION_NAME,
                klass.ACTION_LABEL,
                klass(),
                klass.ACTION_SHORTCUT)):
        print("Registered action \"%s\"" % (klass.ACTION_LABEL,))
        if sw:
            ida_kernwin.attach_action_to_popup(sw, None, klass.ACTION_NAME)
            print("Permanently added action to \"String window\"'s popup")
```

IDAPython example: widgets/waitbox/show_and_hide_waitbox.py

showing, updating & hiding the progress dialog

description: Using the progress dialog (aka 'wait box') primitives.

keywords: actions

```
import time
import random

import ida_kernwin
import ida_hexrays
import ida_funcs

import idautils

perform_decompilation=False

# Note: this try/except block below is just there to
# let us (at Hex-Rays) test this script in various
# situations.
try:
    perform_decompilation = under_test__perform_decompilation
except:
```

```
    pass


step_sleep = 0.5
ida_kernwin.show_wait_box("Processing")
try:
    all_eas = list(idautils.Functions())
    neas = len(all_eas)
    for i, ea in enumerate(all_eas):
        if ida_kernwin.user_cancelled():
            break
        ida_kernwin.replace_wait_box("Processing; step %d/%d" % (i+1, neas))

        if perform_decompilation:
            if not ida_hexrays.decompile(ea):
                print("Decompilation failure: %x" % ea)

        time.sleep(step_sleep * random.random())
finally:
    ida_kernwin.hide_wait_box()
```