# The Beginner's Guide to IDAPython

## by Alexander Hanel

## Table of Contents



## Introduction

Hello!

This is a book about IDAPython.

I originally wrote it as a reference for myself - I wanted a place to go to where I could find examples of functions that I commonly use (and forget) in IDAPython. Since I started this book I have used it many times as a quick reference to understand syntax or see an example of some code - if you follow my blog you may notice a few familiar faces - lots of scripts that I cover here are result of sophomoric experiments that I documented online.

Over the years I have received numerous emails asking what is the best guide for learning IDAPython. Usually I point them to to Ero Carrera's *Introduction to IDAPython* or the example scripts in the IDAPython's public repo. They are excellent sources for learning but they don't cover some common issues that I have come across. I wanted to create a book that addresses these issues. I feel this book is of value for anyone learning IDAPython or wanting a quick reference for examples and snippets. Being an e-book it will not be a static document and I plan on updating it in the future on regular basis.

If you come across any issues, typos or have questions please send me an email **alexander< dot >hanel< at >gmail< dot > com**.

## Updates

Version 1.0 - Published. Version 2.0 - Table of Contents and closing added. Version 3.0 - Grammar fixes provided by Russell V. and added example of renaming operands. Version 4.0 - Support for IDAPython 7.0 Version 4.1 - Bug fixes provided by Minh-Triet Pham Tran

## Intended Audience & Disclaimer

This book is not intended for beginner reverse engineers. It is also not to serve as an introduction to IDA. If you are new to IDA, I would recommend purchasing Chris Eagles `The IDA PRO Book`. It is an excellent book and is worth every penny.

There are a couple of prerequisites for readers of this book. You should be comfortable with reading assembly, have a background in reverse engineering and know your way around IDA. If you have hit a point where you have asked yourself "How can I automate this task using IDAPython?" then this book might be for you. If you already have a handful of programming in IDAPython under your belt then you probably know all the material. That said, it will serve as a handy reference to find examples of commonly used functions.

It should be stated that my background is in reverse engineering of malware. This book does not cover compiler concepts such as basic blocks or other academic concepts used in static analysis. The reason be, is I rarely ever use these concepts when reverse engineering malware. Occasionally I have used them for de-obfuscating code but not often enough that I feel they would be of value for a beginner. After reading this book the reader will feel comfortable with digging into the IDAPython documentation on their own. One last disclaimer, functions for IDA's debugger are not covered.

## Conventions

IDA's Output Windows (command line interface) was used for the examples and output. For the sake of brevity some examples do not contain the assignment of the current address to a variable. Usually represented as `ea = here()`. All of the code can be cut and paste into the command line or IDA's script command option `shift-F2`. Reading from beginning to end is the recommended approach for this book. There are a number of examples that are not explained line by line because it assumed the reader understands the code from previous examples. Different authors call IDAPython's in different ways. Sometimes the code is called as `idc.get_segm_name(ea)` or `get_segm_name(ea)`. This book uses the first style. I have found this convention to be easier to read and debug.

Sometimes when using this convention an error can be thrown.

```
Python>DataRefsTo(here())
<generator object refs at 0x05247828>
Python>idautils.DataRefsTo(here())
Traceback (most recent call last):
  File "<string>", line 1, in <module>
NameError: name 'idautils' is not defined
Python>import idautils   # manual importing of module
Python>idautils.DataRefsTo(here())
<generator object refs at 0x06A398C8>
```

If this happens the module needs to be manually imported as shown above.

## IDAPython Background

IDAPython was created in 2004. It was a joint effort by Gergely Erdelyi and Ero Carrera. Their goal was to combine the power of Python with the analysis automation of IDA's IDC C-like scripting language. IDAPython consists of three separate modules. The first is `idc`. It is a compatibility module for wrapping IDA's IDC functions. The second module is `idautils`. It is a high level utility functions for IDA. The third module is `idaapi`. It allows access to more low level data. With the release of 6.95, IDA started to include more modules that cover functionality that historically have been covered by `idaapi`. These newer modules have a naming convention of `ida_*`. One such module referenced later in this book is `ida_kernwin.py`. Once the reader has finished this book, I would recommend exploring these modules on your own. They are located in `IDADIR\python\ida_*.py`.

### Old vs New

In September of 2017 IDAPython 7.0 was released. This release was a substantial update for IDA because it was ported from x86 to x86_64 binaries. A side effect of this release is that older plugins will need to be recompiled. Even though some major changes happened under the hood for IDAPython (See Hex-Rays' *IDA 7.0: IDAPython backward-compatibility with 6.95 APIs* for more details); older scripts should successfully execute. The backwards-compatibility from 6.95 to 7.0 is due to a compatibility layer that exists in `IDADIR\python\idc_bc695.py`. The following code is an example of the compatibility layer code.

```
def MakeName(ea, name): return set_name(ea, name, SN_CHECK)
```

The old IDAPython function `MakeName` has been renamed to `set_name`. If we wanted to quickly print the new API name from `idc_bc695.py` using the command line we can use the module `inspect`.

```
Python>import inspect
Python>inspect.getsource(MakeName)
def MakeName(ea, name): return set_name(ea, name, SN_CHECK)
```

In future versions of IDA this compatibility layer will be turned off by default. Users of IDA can re-enable it by modifying `IDADIR\cfg\python.cfg` and making sure the `AUTOIMPORT_COMPAT_IDA695` equals `Yes`. Due to the backwards compatibility not being supported in future version of IDA, this book has been re-written using the "new" API names. As of publication date, the compatibility layer only targets APIs within in `idc.py`.

### Python-x86_64

Previously installed modules (such as Pefile) will need to be upgraded from x86 to x86_64 to be used in IDA 7+. The easiest way to update them is by executing the following command `C:\>python27-x64\python.exe -m pip install <package>`.

## Basics

Before we dig too deep we should define some keywords and go over the structure of IDA's disassembly output. We can use the following line of code as an example.

```
.text:00401570                 lea     eax, [ebp+arg_0]
```

The `.text` is the section name and the address is `00401570`. The displayed address is in a hexadecimal format with the prefix `0x` missing. The instruction `lea` is referred to as a mnemonic. After the mnemonic is the first operand `eax` and the second operand is `[ebp+arg_0]`. When working with IDAPython APIs, the most common passed variable is an address. In the IDAPython documentation the address is referenced as `ea`. An address can be accessed manually by a couple of different functions. The most commonly used functions are `idc.get_screen_ea()` or `here()`. These functions return an integer value that contain the address at which the cursor is placed at. If we want to get the minimum address that is present in an IDB we can use `idc.get_inf_attr(INF_MIN_EA)` or to get the max we can use `idc.get_inf_attr(INF_MAX_EA)`.

```
Python>ea = idc.get_screen_ea()
Python>print "0x%x %s" % (ea, ea)
0x401570 4199792
Python>ea = here()
Python>print "0x%x %s" % (ea, ea)
0x401570 419972
Python>hex(idc.get_inf_attr(INF_MIN_EA))
0x401000L
Python>hex(idc.get_inf_attr(INF_MAX_EA))
0x41d000L
```

Each described element in the disassembly output can be accessed by a function in IDAPython. Below is an example of how to access each element. Please recall that we previously stored the address in `ea`.

```
Python>idc.get_segm_name(ea) # get text
.text
```

```
Python>idc.generate_disasm_line(ea, 0) # get disassembly
lea eax, [ebp+arg_0]
Python>idc.print_insn_mnem(ea) # get mnemonic
lea
Python>idc.print_operand(ea,0) # get first operand
eax
Python>idc.print_operand(ea,1) # get second operand
[ebp+arg_0]
```

To get a string representation of the segments name we would use `idc.get_segm_name(ea)` with `ea` being an address within the segment. Printing a string of the disassembly can be done using `idc.generate_disasm_line(ea, 0)`. The arguments are the address stored in `ea` and a flag of `0`. The flag `0` returns the displayed disassembly that IDA discovered during its analysis. `ea` can be any address within the instruction offset range when the `0` flag is passed. To disassembly a particular offset and ignore IDA's analysis a flag of `1` is used. To get the mnemonic or the instruction name we would call `idc.print_insn_mnem(ea)`. To get the operands of the mnemonic we would call `idc.print_operand(ea, long n)`. The first argument is the address and the second `long n` is the operand index. The first operand is 0 and the second is 1.

In some situations it is important to verify an address exists. `idaapi.BADADDR`, `idc.BADADDR` or `BADADDR` can be used to check for valid addresses.

```
Python>idaapi.BADADDR
4294967295
Python>hex(idaapi.BADADDR)
0xffffffffL
Python>if BADADDR != here(): print "valid address"
valid address
```

Example of `BADADDR` on a 64-bit binary.

```
Python>idc.BADADDR
18446744073709551615
Python>hex(idc.BADADDR)
0xffffffffffffffff
```

## Segments

Printing a single line is not very useful. The power of IDAPython comes from iterating through all instructions, cross-references addresses and searching for code or data. The last two will be described in more details later. Iterating through all segments is a good place to start.

```
Python>for seg in idautils.Segments():
    print idc.get_segm_name(seg),  idc.get_segm_start(seg), idc.get_segm_end(seg)
.text 4198400 4272128
.idata 4272128 4272456
.rdata 4272456 4300800
```

```
.data 4300800 4308992
.gfids 4308992 4313088
```

`idautils.Segments()` returns an iterator type object. We can loop through the object by using a for loop. Each item in the list is a segment's start address. The address can be used to get the name if we pass it as an argument to `idc.get_segm_name(ea)`. The start and end of the segments can be found by calling `idc.get_segm_start(ea)` or `idc.get_segm_end(ea)`. The address or `ea` needs to be within the range of the start or end of the segment. If we didn't want to iterate through all segments but wanted to find the next segment from an offset, we could use `idc.get_next_seg(ea)`. The address passed can be any address within the segment range for which we would want to find the next segment for. If by chance we wanted to get a segment's start address by name we could use `idc.get_segm_by_sel(idc.selector_by_name(str_SectionName))`. The function `idc.selector_by_name()` returns the segment selector. The segment selector is an integer value that starts at 1 and increments for each segment (aka section) in the executable. `idc.get_segm_by_sel(int)` is passed the segment selector and returns the start address of segment.

## Functions

Now that we know how to iterate through all segments we should go over how to iterate through all known functions.

```
Python>for func in idautils.Functions():
        print hex(func), idc.get_func_name(func)
Python>
0x401000 ?DefWindowProcA@CWnd@@MAEJIIJ@Z
0x401006 ?LoadFrame@CFrameWnd@@UAEHIKPAVCWnd@@PAUCCreateContext@@@Z
0x40100c ??2@YAPAXI@Z
0x401020 save_xored
0x401030 sub_401030
....
0x45c7b9 sub_45C7B9
0x45c7c3 sub_45C7C3
0x45c7cd SEH_44A590
0x45c7e0 unknown_libname_14
0x45c7ea SEH_43EE30
```

`idautils.Functions()` returns a list of known functions. The list contains the start address of each function. `idautils.Functions()` can be passed arguments to search within a range. If we wanted to do this we would pass the start and end address `idautils.Functions(start_addr, end_addr)`. To get a functions name we use `idc.get_func_name(ea)`. `ea` can be any address within the function boundaries. IDAPython contains a large set of APIs for working with functions. Let's start with a simple function. The semantics of this function is not important but we should create a mental note of the addresses.

```
.text:0045C7C3 sub_45C7C3      proc near
```

```
.text:0045C7C3                 mov     eax, [ebp-60h]
.text:0045C7C6                 push    eax             ; void *
.text:0045C7C7                 call    w_delete
.text:0045C7CC                 retn
.text:0045C7CC sub_45C7C3      endp
```

To get the boundaries we can use `idaapi.get_func(ea)`.

```
Python>func = idaapi.get_func(ea)
Python>type(func)
<class 'idaapi.func_t'>
Python>print "Start: 0x%x, End: 0x%x" % (func.startEA, func.endEA)
Start: 0x45c7c3, End: 0x45c7cd
```

`idaapi.get_func(ea)` returns a class of `idaapi.func_t`. Sometimes it is not always obvious how to use a class returned by a function call. A useful command to explore classes in Python is the `dir(class)` function.

```
Python>dir(func)
['__class__', '__del__', '__delattr__', '__dict__', '__doc__', '__eq__',
'__format__', '__getattribute__', '__gt__', '__hash__', '__init__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__swig_destroy__',
'__weakref__', '_print', 'analyzed_sp', 'argsize', 'clear', 'color', 'compare',
'contains', 'does_return', 'empty', 'endEA', 'extend', 'flags', 'fpd', 'frame',
'frregs', 'frsize', 'intersect', 'is_far', 'llabelqty', 'llabels', 'overlaps',
'owner', 'pntqty', 'points', 'referers', 'refqty', 'regargqty', 'regargs',
'regvarqty', 'regvars', 'size', 'startEA', 'tailqty', 'tails', 'this', 'thisown']
```

From the output we can see the function `startEA` and `endEA`. These are used to access the start and end of the function. These attributes are only applicable towards the current function. If we wanted to access surrounding functions we could use `idc.get_next_func(ea)` and `idc.get_prev_func(ea)`. The value of `ea` only needs to be an address within the boundaries of the analyzed function. A caveat with enumerating functions, is that it only works if IDA has identified the block of code as a function. Until the block of code is marked as a function, it will be skipped during the function enumeration process. Code that is not marked as a function is labeled red in the legend (colored bar at the top in IDA's GUI). These can be manually fixed or automated using the function `idc.create_insn(ea)`.

IDAPython has a lot of different ways to access the same data. A common approach for accessing the boundaries within a function is using `idc.get_func_attr(ea, FUNCATTR_START)` and `idc.get_func_attr(ea, FUNCATTR_END)`.

```
Python>ea = here()
Python>start = idc.get_func_attr(ea, FUNCATTR_START)
Python>end = idc.get_func_attr(ea, FUNCATTR_END)
Python>cur_addr = start
```

```
Python>while cur_addr <= end:
    print hex(cur_addr), idc.generate_disasm_line(cur_addr, 0)
    cur_addr = idc.next_head(cur_addr, end)
Python>
0x45c7c3 mov     eax, [ebp-60h]
0x45c7c6 push    eax             ; void *
0x45c7c7 call    w_delete
0x45c7cc retn
```

`idc.get_func_attr(ea, attr)` is used to get the start and end of the function. We then print the current address and the disassembly by using `idc.generate_disasm_line(ea, 0)`. We use `idc.next_head(eax)` to get the start of the next instruction and continue until we reach the end of this function. A flaw to this approach is it relies on the instructions to be contained within the boundaries of the start and end of the function. If there was a jump to an address higher than the end of the function the loop would prematurely exit. These types of jumps are quite common in obfuscation techniques such as code transformation. Since boundaries can be unreliable it is best practice to call `idautils.FuncItems(ea)` to loop through addresses in a function. We will go into more details about this approach in the following section.

Similar to `idc.get_func_attr(ea, attr)` another useful function for gathering information about functions is `idc.get_func_attr(ea, FUNCATTR_FLAGS)`. It can be used to retrieve information about a function such as if it's library code or if the function doesn't return a value. There are nine possible flags for a function. If we wanted to enumerate all the flags for all the functions we could use the following code.

```
Python>import idautils
Python>for func in idautils.Functions():
    flags = idc.get_func_attr(func,FUNCATTR_FLAGS)
    if flags & FUNC_NORET:
        print hex(func), "FUNC_NORET"
    if flags & FUNC_FAR:
        print hex(func), "FUNC_FAR"
    if flags & FUNC_LIB:
        print hex(func), "FUNC_LIB"
    if flags & FUNC_STATIC:
        print hex(func), "FUNC_STATIC"
    if flags & FUNC_FRAME:
        print hex(func), "FUNC_FRAME"
    if flags & FUNC_USERFAR:
        print hex(func), "FUNC_USERFAR"
    if flags & FUNC_HIDDEN:
        print hex(func), "FUNC_HIDDEN"
    if flags & FUNC_THUNK:
        print hex(func), "FUNC_THUNK"
    if flags & FUNC_LIB:
        print hex(func), "FUNC_BOTTOMBP"
```

We use `idautils.Functions()` to get a list of all known functions addresses and then we use `idc.get_func_attr(ea, FUNCATTR_FLAGS)` to get the flags. We check the value by using a logical

AND (`&`) operation on the returned value. For example, to check if the function does not have a return value, we would use the following comparison `if flags & FUNC_NORET`. Now lets go over all the function flags. Some of these flags are very common while the other are rare.

#### FUNC_NORET

This flag is used to identify a function that does not execute a return instruction. It's internally represented as equal to 1. An example of a function that does not return a value can be seen below.

```
CODE:004028F8 sub_4028F8      proc near
CODE:004028F8
CODE:004028F8                 and     eax, 7Fh
CODE:004028FB                 mov     edx, [esp+0]
CODE:004028FE                 jmp     sub_4028AC
CODE:004028FE sub_4028F8      endp
```

Notice how `ret` or `leave` is not the last instruction.

#### FUNC_FAR

This flag is rarely seen unless reversing software that uses segmented memory. It is internally represented as an integer of 2.

#### FUNC_USERFAR

This flag is rarely seen and has little documentation. HexRays describes the flag as "user has specified far-ness of the function". It has an internal value of 32.

#### FUNC_LIB

This flag is used to find library code. Identifying library code is very useful because it is code that typically can be ignored when doing analysis. Its internally represented as an integer value of 4. Below is an example of its usage and functions it has identified.

```
Python>for func in idautils.Functions():
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    if flags & FUNC_LIB:
        print hex(func), "FUNC_LIB", idc.get_func_name(func)
Python>
0x1a711160 FUNC_LIB _strcpy
0x1a711170 FUNC_LIB _strcat
0x1a711260 FUNC_LIB _memcmp
0x1a711320 FUNC_LIB _memcpy
0x1a711662 FUNC_LIB __onexit
...
0x1a711915 FUNC_LIB _exit
0x1a711926 FUNC_LIB __exit
0x1a711937 FUNC_LIB __cexit
0x1a711946 FUNC_LIB __c_exit
0x1a711955 FUNC_LIB _puts
```

```
0x1a7119c0 FUNC_LIB _strcmp
```

#### FUNC_STATIC

This flag is used to identify functions that were compiled as a static function. In C functions are global by default. If the author defines a function as static it can be only accessed by other functions within that file. In a limited way this could be used to aid in understanding how the source code was structured.

#### FUNC_FRAME

This flag indicates the function uses a frame pointer `ebp`. Functions that use frame pointers typically start with the standard function prologue for setting up the stack frame.

```
.text:1A716697      push    ebp
.text:1A716698      mov     ebp, esp
.text:1A71669A      sub     esp, 5Ch
```

#### FUNC_BOTTOMBP

Similar to `FUNC_FRAM` this flag is used to track the frame pointer. It identifies functions that base pointer points to the stack pointer.

#### FUNC_HIDDEN

Functions with the `FUNC_HIDDEN` flag means they are hidden and needs to be expanded to view. If we were to go to an address of a function that is marked as hidden it would automatically be expanded.

#### FUNC_THUNK

This flag identifies functions that are thunk functions. They are simple functions that jump to another function.

```
.text:1A710606 Process32Next proc near
.text:1A710606      jmp     ds:__imp_Process32Next
.text:1A710606 Process32Next endp
```

It should be noted that a function can consist of multiple flags.

```
0x1a716697 FUNC_LIB
0x1a716697 FUNC_FRAME
0x1a716697 FUNC_HIDDEN
0x1a716697 FUNC_BOTTOMBP
```

## Instructions

Since we know how to work with functions, its now time to go over how to access instructions within a function. If we have the address of a function we can use `idautils.FuncItems(ea)` to get a list of all the addresses.

```
Python>dism_addr = list(idautils.FuncItems(here()))
Python>type(dism_addr)
<type 'list'>
Python>print dism_addr
[4573123, 4573126, 4573127, 4573132]
Python>for line in dism_addr: print hex(line), idc.generate_disasm_line(line, 0)
0x45c7c3 mov     eax, [ebp-60h]
0x45c7c6 push    eax             ; void *
0x45c7c7 call    w_delete
0x45c7cc retn
```

`idautils.FuncItems(ea)` returns an iterator type but is cast to a `list`. The list contains the start address of each instruction in consecutive order. Now that we have a good knowledge base for looping through segments, functions and instructions let show a useful example. Sometimes when reversing packed code it is useful to only know where dynamic calls happens. A dynamic call would be a call or jump to an operand that is a register such as `call eax` or `jmp edi`.

```
Python>
for func in idautils.Functions():
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
    dism_addr = list(idautils.FuncItems(func))
    for line in dism_addr:
        m = idc.print_insn_mnem(line)
        if m == 'call' or m == 'jmp':
            op = idc.get_operand_type(line, 0)
            if op == o_reg:
                print "0x%x %s" % (line, idc.generate_disasm_line(line, 0))
Python>
0x43ebde call    eax             ; VirtualProtect
```

We call `idautils.Functions()` to get a list of all known functions. For each function we retrieve the functions flags by calling `idc.get_func_attr(ea, FUNCATTR_FLAGS)`. If the function is library code or a thunk function the function is passed. Next we call `idautils.FuncItems(ea)`to get all the addresses within the function. We loop through the list using a `for` loop. Since we are only interested in `call` and `jmp` instructions we need to get the mnemonic by calling `idc.print_insn_mnem(ea)`. We then use a simple string comparison to check the mnemonic. If the mnemonic is a jump or call we get the operand type by calling `idc.get_operand_type(ea, n)`. This function returns an integer that is internally called `op_t.type`. This value can be used to determine if the operand is a register, memory reference, etc. We then check if the `op_t.type` is a register. If so, we print the line. Casting the return of `idautils.FuncItems(ea)` into a `list` is useful because iterators do not have objects such as `len()`. By casting it as a list we could easily get the number of lines or instructions in a function.

```
Python>ea = here()
Python>len(idautils.FuncItems(ea))
Traceback (most recent call last):
  File "<string>", line 1, in <module>
TypeError: object of type 'generator' has no len()
Python>len(list(idautils.FuncItems(ea)))
39
```

In the previous example we used a list that contained all addresses within a function. We looped through each entity to access the next instruction. What if we only had an address and wanted to get the next instruction? To move to the next instruction address we can use `idc.next_head(ea)` and to get the previous instruction address we use `idc.prev_head(ea)`. These functions get the start of the next instruction but not the next address. To get the next address we use `idc.next_addr(ea)` and to get the previous address we use `idc.prev_head(ea)`.

```
Python>ea = here()
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x10004f24 call    sub_10004F32
Python>next_instr = idc.next_head(ea)
Python>print hex(next_instr), idc.generate_disasm_line(next_instr, 0)
0x10004f29 mov     [esi], eax
Python>prev_instr = idc.prev_head(ea)
Python>print hex(prev_instr), idc.generate_disasm_line(prev_instr, 0)
0x10004f1e mov     [esi+98h], eax
Python>print hex(idc.next_addr(ea))
0x10004f25
Python>print hex(idc.prev_head(ea))
0x10004f23
```

In the dynamic call example, the IDAPython code relies on using a string comparison of `jmp` and `call`. Rather than using a string comparison, we can also decode the instructions using `idaapi.decode_insn(ea)`. Decoding an instruction can be advantageous because working with the integer representation of the instruction can be faster and less error prone. Unfortunately, the integer representation is specific to IDA and cannot be easily ported to other disassembly tools. Below is the same example but using `idaapi.decode_insn(ea)` and comparing the integer representation.

```
Python>JMPS = [idaapi.NN_jmp, idaapi.NN_jmpfi, idaapi.NN_jmpni]
Python>CALLS = [idaapi.NN_call, idaapi.NN_callfi, idaapi.NN_callni]
Python>for func in idautils.Functions():
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
    dism_addr = list(idautils.FuncItems(func))
    for line in dism_addr:
```

```
        idaapi.decode_insn(line)
        if idaapi.cmd.itype in CALLS or idaapi.cmd.itype in JMPS:
            if idaapi.cmd.Op1.type == o_reg:
                print "0x%x %s" % (line, idc.generate_disasm_line(line, 0))
Python>
0x43ebde call    eax             ; VirtualProtect
```

The output is the same as the previous example. The first two lines put the constants for `jmp` and `call` into two lists. Since we are not working with the string representation of the mnemonic, we need to be cognizant that a mnemonic (such as `call` or `jmp`) could have multiple values. For example, `jmp` could be represented by `idaapi.NN_jmp` for a jump, `idaapi.NN_jmpfi` for an indirect far jump or `idaapi.NN_jmpni` for an indirect near jump. X86 and X64 instruction types all start with `NN`. To explore all 1,700+ instruction types we can execute `[name for name in dir(idaapi) if "NN" in name]` in the command line or review them in IDA's SDK file `allins.hpp`. Once we have the instructions in lists, we use a combination of `idautils.Functions()` and `get_func_attr(ea, FUNCATTR_FLAGS)` to get all applicable functions while ignoring libraries and thunks. We get each instruction in a function by calling `idautils.FuncItems(ea)`. This is where are newly introduced function `idaapi.decode_insn(ea)` is called. This function takes the address of instruction we want decoded. Once it is decoded we can access different properties of the instruction by accessing it via `idaapi.cmd`.

```
Python>dir(idaapi.cmd)
['Op1', 'Op2', 'Op3', 'Op4', 'Op5', 'Op6', 'Operands', ....., 'assign', 'auxpref',
'clink', 'clink_ptr', 'copy', 'cs', 'ea', 'flags', 'get_canon_feature',
'get_canon_mnem', 'insnpref', 'ip', 'is_canon_insn', 'is_macro', 'itype', 'segpref',
'size']
```

As we can see from the `dir()` command `idaapi.cmd` has a good amount of attributes. The operand type is accessed by using `idaapi.cmd.Op1.type`. Please note that the operand index starts at 1 rather than 0 which is different than `idc.get_operand_type(ea,n)`.

## Operands

Operand types are commonly used so it is beneficial to go over all the types. As previous stated we can use `idc.get_operand_type(ea,n)` to get the operand type. `ea` is the address and `n` is the index. There are eight different type of operand types.

#### o_void

If an instruction does not have any operands it returns 0.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0xa09166 retn
Python>print idc.get_operand_type(ea,0)
0
```

**o_reg**

If an operand is a general register it returns this type. This value is internally represented as 1.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0xa09163 pop     edi
Python>print idc.get_operand_type(ea,0)
1
```

**o_mem**

If an operand is direct memory reference it returns this type. This value is internally represented as 2. This type is useful for finding references to DATA.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0xa05d86 cmp     ds:dword_A152B8, 0
Python>print idc.get_operand_type(ea,0)
2
```

**o_phrase**

This operand is returned if the operand consists of a base register and/or an index register. This value is internally represented as 3.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x1000b8c2 mov     [edi+ecx], eax
Python>print idc.get_operand_type(ea,0)
3
```

**o_displ**

This operand is returned if the operand consists of registers and a displacement value. The displacement is an integer value such 0x18. It is commonly seen when an instruction accesses values in a structure. Internally it is represented as a value of 4.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0xa05dc1 mov     eax, [edi+18h]
Python>print idc.get_operand_type(ea,1)
4
```

**o_imm**

Operands that are a value such as an integer of 0xC are of this type. Internally it is represented as 5.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
```

```
0xa05da1 add     esp, 0Ch
Python>print idc.get_operand_type(ea,1)
5
```

**o_far**

This operand is not very common when reversing x86 or x86_64. It is used to find operands that are accessing immediate far addresses. It is represented internally as 6

**o_near**

This operand is not very common when reversing x86 or x86_64. It is used to find operands that are accessing immediate near addresses. It is represented internally as 7.

## Example

Sometimes when reversing a memory dump of an executable the operands are not recognized as an offset.

```
seg000:00BC1388           push    0Ch
seg000:00BC138A           push    0BC10B8h
seg000:00BC138F           push    [esp+10h+arg_0]
seg000:00BC1393           call    ds:_strnicmp
```

The second value being pushed is a memory offset. If we were to right click on it and change it to a data type; we would see the offset to a string. This is okay to do once or twice but after that we might as well automate the process.

```
min = idc.get_inf_attr(INF_MIN_EA)
max = idc.get_inf_attr(INF_MAX_EA)
# for each known function
for func in idautils.Functions():
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    # skip library & thunk functions
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
    dism_addr = list(idautils.FuncItems(func))
    for curr_addr in dism_addr:
        if idc.get_operand_type(curr_addr, 0) == 5 and \
                (min < idc.get_operand_value(curr_addr, 0) < max):
            idc.OpOff(curr_addr, 0, 0)
        if idc.get_operand_type(curr_addr, 1) == 5 and \
                (min < idc.get_operand_value(curr_addr, 1) < max):
            idc.op_plain_offset(curr_addr, 1, 0)
```

After running the above code we would now see the string.

```
seg000:00BC1388           push    0Ch
seg000:00BC138A           push    offset aNtoskrnl_exe  ; "ntoskrnl.exe"
```

```
seg000:00BC138F             push    [esp+10h+arg_0]
seg000:00BC1393             call    ds:_strnicmp
```

At the start we get the minimum and maximum address by calling `idc.get_inf_attr(INF_MIN_EA)` and `idc.get_inf_attr(INF_MAX_EA)` We loop through all functions and instructions. For each instruction we check if the operand type is of `o_imm` and is represented internally as the number 5. `o_imm` types are values such as an integer or an offset. Once a value is found we read the value by calling `idc.get_operand_value(ea,n)`. The value is then checked to see if it is in the range of the minimum and maximum addresses. If so, we use `idc.op_plain_offset(ea, n, base)` to convert the operand to an offset. The first argument `ea` is the address, `n` is the operand index and `base` is the base address. Our example only needs to have a `base` of zero.

# Xrefs

Being able to locate cross-references (aka xrefs) to data or code is very important. Xrefs are important because they provide locations of where certain data is being used or where a function is being called from. For example, what if we wanted to locate all the address where `WriteFile` was called from. Using Xrefs all we would need to do is locate the address of `WriteFile` in the import table and then find all xrefs to it.

```
Python>wf_addr = idc.get_name_ea_simple("WriteFile")
Python>print hex(wf_addr), idc.generate_disasm_line(wf_addr, 0)
0x1000e1b8 extrn WriteFile:dword
Python>for addr in idautils.CodeRefsTo(wf_addr, 0):\
    print hex(addr), idc.generate_disasm_line(addr, 0)
0x10004932 call    ds:WriteFile
0x10005c38 call    ds:WriteFile
0x10007458 call    ds:WriteFile
```

In the first line we get the address of the API `WriteFile` by using `idc.get_name_ea_simple(str)`. This function returns the address of the API. We print out the address of `WriteFile` and it's string representation. Then loop through all code cross references by calling `idautils.CodeRefsTo(ea, flow)`. It returns an iterator that can be looped through. `ea` is the address that we would like to have cross-referenced to. The argument flow is a `bool`. It is used to specify to follow normal code flow or not. Each cross reference to the address is then displayed. A quick note about the use of `idc.get_name_ea_simple(str)`. All renamed functions and APIs in an IDB can be accessed by calling `idautils.Names()`. This function returns an iterator object which can be looped through to print or access the names. Each named item is a tuple of `(ea, str_name)`.

```
Python>[x for x in Names()]
[(268439552, 'SetEventCreateThread'), (268439615, 'StartAddress'), (268441102,
'SetSleepClose'),....]
```

If we wanted to get where code was referenced from we would use `idautisl.CodeRefsFrom(ea,flow)`. For example let's get the address of where `0x10004932` is referenced from.

```
Python>ea = 0x10004932
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x10004932 call    ds:WriteFile
Python>for addr in idautils.CodeRefsFrom(ea, 0):\
    print hex(addr), idc.generate_disasm_line(addr, 0)
Python>
0x1000e1b8 extrn WriteFile:dword
```

If we review the `idautils.CodeRefsTo(ea, flow)` example we see the address `0x10004932` is a `to` address to `WriteFile`. `idautils.CodeRefsTo(ea, flow)` and `idautils.CodeRefsFrom(ea, flow)` are used to search for cross references to and from code. A limitation of using `idautils.CodeRefsTo(ea, flow)` is that APIs that are imported dynamically and then manually renamed, do not show up as code cross-references. Say we manually rename a dword address to `"RtlCompareMemory"` using `idc.set_name(ea, name, SN_CHECK)`.

```
Python>hex(ea)
0xa26c78
Python>idc.set_name(ea, "RtlCompareMemory", SN_CHECK)
True
Python>for addr in idautils.CodeRefsTo(ea, 0):\
    print hex(addr), idc.generate_disasm_line(addr, 0)
```

IDA does not label these APIs as code cross references. A little later we will describe a generic technique to get all cross references. If we wanted to search for cross references to and from data we could use `idautils.DataRefsTo(e)` or `idautils.DataRefsFrom(ea)`.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x1000e3ec db 'vnc32',0
Python>for addr in idautils.DataRefsTo(ea): print hex(addr),
idc.generate_disasm_line(addr, 0)
0x100038ac push    offset aVnc32         ; "vnc32"
```

`idautils.DataRefsTo(ea)` takes an argument of the address and returns an iterator of all the addresses that cross reference to the data.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x100038ac push    offset aVnc32         ; "vnc32"
Python>for addr in idautils.DataRefsFrom(ea):  print hex(addr),
idc.generate_disasm_line(addr, 0)
0x1000e3ec db 'vnc32',0
```

To do the reverse and show the from address we call `idautils.DataRefsFrom(ea)`, pass the address as an argument. Which returns an iterator of all the addresses that cross reference back to the data. The different usage of code and data can be a little confusing. Let's describe a more generic technique. This approach can be used to get all cross references to an address by calling a single function. We can get all cross references *to* an address using `idautils.XrefsTo(ea, flags=0)` and get all cross references *from* an address by calling `idautils.XrefsFrom(ea, flags=0)`.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x1000eee0 unicode 0, <Path>,0
Python>for xref in idautils.XrefsTo(ea, 1):
          print xref.type, idautils.XrefTypeName(xref.type), \
              hex(xref.frm), hex(xref.to), xref.iscode
Python>
1 Data_Offset 0x1000ac0d 0x1000eee0 0
Python>print hex(xref.frm), idc.generate_disasm_line(xref.frm, 0)
0x1000ac0d push    offset KeyName        ; "Path"
```

The first line displays our address and a string named `<Path>`. We use `idautils.XrefsTo(ea, 1)` to get all cross references to the string. We then use `xref.type` to print the xrefs type value. `idautils.XrefTypeName(xref.type)` is used to print the string representation of this type. There are twelve different documented reference type values. The value can be seen on the left and it's corresponding name can be seen below.

```
0  = 'Data_Unknown'
1  = 'Data_Offset'
2  = 'Data_Write'
3  = 'Data_Read'
4  = 'Data_Text'
5  = 'Data_Informational'
16 = 'Code_Far_Call'
17 = 'Code_Near_Call'
18 = 'Code_Far_Jump'
19 = 'Code_Near_Jump'
20 = 'Code_User'
21 = 'Ordinary_Flow'
```

The `xref.frm` prints out the from address and `xref.to` prints out the two address. `xref.iscode` prints if the xref is in a code segment. In the previous example we had the flag of `idautils.XrefsTo(ea, 1)` set to the value 1. If the flag is zero any cross reference are displayed. We can use the following block of assembly to illustrate this point.

```
.text:1000AAF6           jnb     short loc_1000AB02   ; XREF
.text:1000AAF8           mov     eax, [ebx+0Ch]
.text:1000AAFB           mov     ecx, [esi]
.text:1000AAFD           sub     eax, edi
.text:1000AAFF           mov     [edi+ecx], eax
.text:1000AB02
.text:1000AB02 loc_1000AB02:                           ; ea is here()
.text:1000AB02           mov     byte ptr [ebx], 1
```

We have the cursor at `1000AB02`. This address has a cross reference from `1000AAF6` but it also has second cross reference.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x1000ab02 mov     byte ptr [ebx], 1
Python>for xref in idautils.XrefsTo(ea, 1):
          print xref.type, idautils.XrefTypeName(xref.type), \
              hex(xref.frm), hex(xref.to), xref.iscode
Python>
19 Code_Near_Jump 0x1000aaf6 0x1000ab02 1
Python>for xref in idautils.XrefsTo(ea, 0):
          print xref.type, idautils.XrefTypeName(xref.type), \
              hex(xref.frm), hex(xref.to), xref.iscode
Python>
21 Ordinary_Flow 0x1000aaff 0x1000ab02 1
19 Code_Near_Jump 0x1000aaf6 0x1000ab02 1
```

The second cross reference is from `1000AAFF` to `1000AB02`. Cross references do not have to be caused by branch instructions. They can also be caused by normal ordinary code flow. If we set the flag to 1 `Ordinary_Flow` reference types won't be added. Now back to our `RtlCompareMemory` example from earlier. We can use `idautils.XrefsTo(ea, flow)` to get all cross references.

```
Python>hex(ea)
0xa26c78
Python>idc.set_name(ea, "RtlCompareMemory", SN_CHECK)
True
Python>for xref in idautils.XrefsTo(ea, 1):
          print xref.type, idautils.XrefTypeName(xref.type), \
              hex(xref.frm), hex(xref.to), xref.iscode
Python>
3 Data_Read 0xa142a3 0xa26c78 0
3 Data_Read 0xa143e8 0xa26c78 0
3 Data_Read 0xa162da 0xa26c78 0
```

Getting all cross references can be a little verbose sometimes.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0xa21138 extrn GetProcessHeap:dword
Python>for xref in idautils.XrefsTo(ea, 1):
          print xref.type, idautils.XrefTypeName(xref.type), \
              hex(xref.frm), hex(xref.to), xref.iscode
Python>
17 Code_Near_Call 0xa143b0 0xa21138 1
17 Code_Near_Call 0xa1bb1b 0xa21138 1
3 Data_Read 0xa143b0 0xa21138 0
3 Data_Read 0xa1bb1b 0xa21138 0
Python>print idc.generate_disasm_line(0xa143b0, 0)
call    ds:GetProcessHeap
```

The verboseness comes from the `Data_Read` and the `Code_Near` both added to the xrefs. Getting all the addresses and adding them to a set can be useful to slim down on all the addresses.

```
def get_to_xrefs(ea):
    xref_set = set([])
    for xref in idautils.XrefsTo(ea, 1):
        xref_set.add(xref.frm)
    return xref_set


def get_frm_xrefs(ea):
    xref_set = set([])
    for xref in idautils.XrefsFrom(ea, 1):
        xref_set.add(xref.to)
    return xref_set
```

Example of the slim down functions on out `GetProcessHeap` example.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0xa21138 extrn GetProcessHeap:dword
Python>get_to_xrefs(ea)
set([10568624, 10599195])
Python>[hex(x) for x in get_to_xrefs(ea)]
['0xa143b0', '0xa1bb1b']
```

## Searching

We have already gone over some basic searches by iterating over all known functions or instructions. This is useful but sometimes we need to search for specific bytes such as `0x55 0x8B 0xEC`. This byte pattern is the classic function prologue `push ebp, mov ebp, esp`. To search for byte or binary patterns we can use `idc.find_binary(ea, flag, searchstr, radix=16)`. `ea` is the address that we would like to search from the `flag` is the direction or condition. There are a number of different types of flags. The names and values can be seen below.

```
SEARCH_UP = 0
SEARCH_DOWN = 1
SEARCH_NEXT = 2
SEARCH_CASE = 4
SEARCH_REGEX = 8
SEARCH_NOBRK = 16
SEARCH_NOSHOW = 32
SEARCH_UNICODE = 64  **
SEARCH_IDENT = 128 **
SEARCH_BRK = 256  **
    ** Older versions of IDAPython do not support these
```

Not all of these flags are worth going over but we can touch upon the most commonly used flags.

- `SEARCH_UP` and `SEARCH_DOWN` are used to select the direction we would like our search to follow.
- `SEARCH_NEXT` is used to get the next found object.
- `SEARCH_CASE` is used to specify case sensitivity.

- `SEARCH_NOSHOW` does not show the search progress.
- `SEARCH_UNICODE` is used to treat all search strings as Unicode.

`searchstr` is the pattern we are search for. The `radix` is used when writing processor modules. This topic is outside of the scope of this book. I would recommend reading Chapter 19 of Chris Eagle's The IDA Pro Book. For now the `radix` field can be left blank. Let's go over a quick walk through on finding the function prologue byte pattern mentioned earlier.

```
Python>pattern = '55 8B EC'
addr = idc.get_inf_attr(INF_MIN_EA)
for x in range(0, 5):
    addr = idc.find_binary(addr, SEARCH_DOWN, pattern);
    if addr != idc.BADADDR:
        print hex(addr), idc.generate_disasm_line(addr, 0)
Python>
0x401000 push    ebp
0x401000 push    ebp
0x401000 push    ebp
0x401000 push    ebp
0x401000 push    ebp
```

In the first line we define our search pattern. The search pattern can be in the format of hexadecimal starting with `0x` as in`0x55 0x8B 0xEC` or as bytes appear in IDA's hex view `55 8B EC`. The format `\x55\x8B\xEC` cannot be used unless we were using `idc.find_text(ea, flag, y, x, searchstr)`. `idc.get_inf_attr(INF_MIN_EA)` is used to get the first address in the executable. We then assign the return of `idc.find_binary(ea, flag, searchstr, radix=16)` to a variable called `addr`.

When searching it is important to verify that the search did find the pattern. This is tested by comparing `addr` with `idc.BADADDR`. We then print the address and disassembly. Notice how the address did not increment? This is because we did not pass the `SEARCH_NEXT` flag. If this flag is not passed the current address is used to search for the pattern. If the last address contained our byte pattern the search will never increment passed it. Below is the corrected version with the `SEARCH_NEXT` flag.

```
Python>pattern = '55 8B EC'
addr = idc.get_inf_attr(INF_MIN_EA)
for x in range(0,5):
    addr = idc.find_binary(addr, SEARCH_DOWN|SEARCH_NEXT, pattern);
    if addr !=  idc.BADADDR:
        print hex(addr), idc.generate_disasm_line(addr, 0)
Python>
0x401040 push    ebp
0x401070 push    ebp
0x4010e0 push    ebp
0x401150 push    ebp
0x4011b0 push    ebp
```

Searching for byte patterns is useful but sometimes we might want to search for strings such as "chrome.dll". We could convert the strings to a hex bytes using `[hex(y) for y in bytearray("chrome.dll")]` but this is a little ugly. Also, if the string is unicode we would have to account for that format. The simplest approach is using `idc.find_text(ea, flag, y, x, searchstr)`. Most of these fields should look familiar because they are the same as `idc.find_binary`. `ea` is the start address and flag is the direction and types to search for. `y` is the number of lines at `ea` to search from and `x` is the coordinate in the line. These fields are typically assigned as `0`. Now search for occurrences of the string "Accept". Any string from the strings window `shift+F12` can be used for this example.

```
Python>cur_addr = idc.get_inf_attr(INF_MIN_EA)
end = idc.get_inf_attr(INF_MAX_EA)
while cur_addr < end:
    cur_addr = idc.find_text(cur_addr, SEARCH_DOWN, 0, 0, "Accept")
    if cur_addr == idc.BADADDR:
        break
    else:
        print hex(cur_addr), idc.generate_disasm_line(cur_addr, 0)
    cur_addr = idc.next_head(cur_addr)
Python>
0x40da72 push    offset aAcceptEncoding; "Accept-Encoding:\n"
0x40face push    offset aHttp1_1Accept; " HTTP/1.1\r\nAccept: */* \r\n "
0x40fadf push    offset aAcceptLanguage; "Accept-Language: ru \r\n"
...
0x423c00 db 'Accept',0
0x423c14 db 'Accept-Language',0
0x423c24 db 'Accept-Encoding',0
0x423ca4 db 'Accept-Ranges',0
```

We use `idc.get_inf_attr(INF_MIN_EA)` to get the minimum address and assign that to a variable named `cur_addr`. This is similarly done again for the maximum address by calling `idc.get_inf_attr(INF_MAX_EA)` and assigning the return to a variable named the `end`. Since we do not know how many occurrences of the string are present, we need to check that the search continues down and is less than the maximum address. We then assign the return of `idc.find_text` to the current address. Since we are manually incrementing the address by calling `idc.next_head(ea)` we do not need the `SEARCH_NEXT` flag. The reason why we manually increment the current address to the following line is because a string can occur multiple times on a single line. This can make it tricky to get the address of the next string.

Along with pattern searching previously described there a couple of functions that can be used to find other types. The naming conventions of the find APIs makes it easy to infer it's overall functionality. Before we discuss finding the different types we firstly go over identifying types by their address. There is a subset of APIs that start with `is` that can be used to determine an address's type. The APIs return a Boolean value of `True` or `False`.

**idc.is_code(f)**

Returns `True` if IDA has marked the address as code.

**idc.is_data(f)**

Returns `True` if IDA has marked the address as data.

**idc.is_tail(f)**

Returns `True` if IDA has marked the address as tail.

**idc.is_unknown(f)**

Returns `True` if IDA has marked the address as unknown. This type is used when IDA has not identified if the address is code or data.

**idc.is_head(f)**

Returns `True` if IDA has marked the address as head.

The `f` is new to us. Rather than passing an address we first need to get the internal flags representation and then pass it to our `idc.is_*` set of functions. To get the internal flags we use `idc.get_full_flags(ea)`. Now that we have a basics on how the function can be used and the different types let's do a quick example.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x10001000 push    ebp
Python>idc.is_code(idc.get_full_flags(ea))
True
```

**idc.find_code(ea, flag)**

It is used to find the next address that is marked as code. This can be useful if we want to find the end of a block of data. If `ea` is an address that is already marked as code it returns the next address. The `flag` is used as previously described in `idc.find_text`.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x4140e8 dd offset dword_4140EC
Python>addr = idc.find_code(ea, SEARCH_DOWN|SEARCH_NEXT)
Python>print hex(addr), idc.generate_disasm_line(addr, 0)
0x41410c push    ebx
```

As we can see `ea` is the address `0x4140e8` of some data. We assign the return of `idc.find_code(ea, SEARCH_DOWN|SEARCH_NEXT)` to `addr`. Then we print `addr` and it's disassembly. By calling this single function we skipped 36 bytes of data to get the start of a section marked as code.

**idc.find_data(ea, flag)**

It is used exactly as `idc.find_code` except it returns the start of the next address that is marked as a block of data. If we reverse the previous scenario and start from the address of code and search up to find the start of the data.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x41410c push    ebx
Python>addr = idc.find_data(ea, SEARCH_UP|SEARCH_NEXT)
Python>print hex(addr), idc.generate_disasm_line(addr, 0)
0x4140ec dd 49540E0Eh, 746E6564h, 4570614Dh, 7972746Eh, 8, 1, 4010BCh
```

The only thing that is slightly different than the previous example is the direction of `SEARCH_UP|SEARCH_NEXT` and searching for data.

**idc.find_unknown(ea, flag)**

This function is used to find the address of bytes that IDA did not identify as code or data. The `unknown` type requires further manual analysis either visually or through scripting.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x406a05 jge     short loc_406A3A
Python>addr = idc.find_unknown(ea, SEARCH_DOWN)
Python>print hex(addr), idc.generate_disasm_line(addr, 0)
0x41b004 db 0DFh ; ?
```

**idc.find_defined(ea, flag)**

It is used to find an address that IDA identified as code or data.

```
0x41b900 db    ? ;
Python>addr = idc.find_defined(ea, SEARCH_UP)
Python>print hex(addr), idc.generate_disasm_line(addr, 0)
0x41b5f4 dd ?
```

This might not seem of any real value but if we were to print the cross references of `addr` we would see it is being used.

```
Python>for xref in idautils.XrefsTo(addr, 1):
          print hex(xref.frm),  idc.generate_disasm_line(xref.frm, 0)
Python>
0x4069c3 mov     eax, dword_41B5F4[ecx*4]
```

**idc.find_imm(ea, flag, value)**

Rather than searching for a type we might want to search for a specific value. say for example that we have a feeling that the code calls `rand` to generate a random number but we can't find the code. If we knew that `rand` uses the value `0x343FD` as a seed we could search for that number.

```
Python>addr = idc.find_imm(get_inf_attr(INF_MIN_EA), SEARCH_DOWN, 0x343FD )
Python>addr
[268453092, 0]
```

```
Python>print "0x%x %s %x" % (addr[0], idc.generate_disasm_line(addr[0], 0),  addr[1]
)
0x100044e4 imul    eax, 343FDh 0
```

In the first line we pass the minimum address via `get_inf_attr(INF_MIN_EA)`, search down and then search for the value `0x343FD`. Rather than returning an address as shown in the previous Find APIs `idc.find_imm` returns a tuple. The first item in the tuple is the address and second is the operand. Similar to the return of `idc.print_operand` the first operand starts at zero. When we print the address and disassembly we can see the value is the second operand. If we wanted to search for all uses of an immediate value we could do the following.

```
Python>addr = idc.get_inf_attr(INF_MIN_EA)
while True:
    addr, operand = idc.find_imm(addr, SEARCH_DOWN | SEARCH_NEXT, 0x7a)
    if addr != BADADDR:
        print hex(addr), idc.generate_disasm_line(addr, 0), "Operand ", operand
    else:
        break
Python>
0x402434 dd 9, 0FF0Bh, 0Ch, 0FF0Dh, 0Dh, 0FF13h, 13h, 0FF1Bh, 1Bh Operand  0
0x40acee cmp     eax, 7Ah Operand  1
0x40b943 push    7Ah Operand  0
0x424a91 cmp     eax, 7Ah Operand  1
0x424b3d cmp     eax, 7Ah Operand  1
0x425507 cmp     eax, 7Ah Operand  1
```

Most of the code should look familiar but since we are searching for multiple values it uses a while loop and the `SEARCH_DOWN|SEARCH_NEXT` flag.

## Selecting Data

We will not always need to search for code or data. In some instances we already know the location of the code or data but we want to select it for analysis. In situations like this we might just want to highlight the code and start working with it in IDAPython. To get the boundaries of selected data we can use `idc.read_selection_start()` to get the start and `idc.read_selection_end()` to get the end. Let's say we have the below code selected.

```
.text:00408E46                  push    ebp
.text:00408E47                  mov     ebp, esp
.text:00408E49                  mov     al, byte ptr dword_42A508
.text:00408E4E                  sub     esp, 78h
.text:00408E51                  test    al, 10h
.text:00408E53                  jz      short loc_408E78
.text:00408E55                  lea     eax, [ebp+Data]
```

We can use the following code to print out the addresses.

```
Python>start = idc.read_selection_start()
Python>hex(start)
0x408e46
Python>end = idc.read_selection_end()
Python>hex(end)
0x408e58
```

We assign the return of `idc.read_selection_start()` to `start`. This is the address of the first selected address. We then use the return of `idc.read_selection_end()` and assign it to `end`. One thing to note is that `end` is not the last selected address but the start of the next address. If we preferred to make only one API call we could use `idaapi.read_selection()`. It returns a tuple with the first value being a bool if the selection was read, the second being the start address and the last address being the end.

```
Python>Worked, start, end = idaapi.read_selection()
Python>print Worked, hex(start), hex(end)
True 0x408e46 0x408e58
```

Be cautious when working with 64 bit samples. The base address is not always correct because the selected start address can cause an integer overflow and the leading digit can be incorrect. This likely has been fixed in IDA 7.+.

## Comments & Renaming

A personal belief of mine is "If I'm not writing, I'm not reversing". Adding comments, renaming functions and interacting with the assembly is one of the best ways to understand what the code is doing. Over time some of the interaction becomes redundant. In situations like this it useful to automate the process.

Before we go over some examples we should first discuss the basics of comments and renaming. There are two types of comments. The first one is a regular comment and the second is a repeatable comment. A regular comment appears at address `0041136B` as the text `regular comment`. A repeatable comment can be seen at address `00411372`, `00411386` and `00411392`. Only the last comment is a comment that was manually entered. The other comments appear when an instruction references an address (such as a branch condition) that contains a repeatable comment.

```
00411365         mov     [ebp+var_214], eax
0041136B         cmp     [ebp+var_214], 0    ; regular comment
00411372         jnz     short loc_411392  ; repeatable comment
00411374         push    offset sub_4110E0
00411379         call    sub_40D060
0041137E         add     esp, 4
00411381         movzx   edx, al
00411384         test    edx, edx
00411386         jz      short loc_411392 ; repeatable comment
00411388         mov     dword_436B80, 1
```

```
00411392
00411392 loc_411392:
00411392
00411392            mov     dword_436B88, 1   ; repeatable comment
0041139C            push    offset sub_4112C0
```

To add comments we use `idc.set_cmt(ea, comment,0)` and for repeatable comments we use `idc.set_cmt(ea, comment, 1)`. `ea` is the address, `comment` is a string we would like added, 0 specifies the comment is not repeatable and 1 states the comment as repeatable. The below code adds a comment every time an instruction zeroes out a register or value with `XOR`.

```
for func in idautils.Functions():
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    # skip library & thunk functions
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        continue
    dism_addr = list(idautils.FuncItems(func))
    for ea in dism_addr:
        if idc.print_insn_mnem(ea) == "xor":
            if idc.print_operand(ea, 0) == idc.print_operand(ea, 1):
                comment = "%s = 0" % (idc.print_operand(ea, 0))
                idc.set_cmt(ea, comment, 0)
```

As previously described we loop through all functions by calling `idautils.Functions()` and loop through all the instructions by calling `list(idautils.FuncItems(func))`. We read the mnemonic using `idc.print_insn_mnem(ea)` and check it is equal to `xor`. If so, we verify the operands are equal with `idc.print_operand(ea, n)`. If equal, we create a string with the operand and then make add a non-repeatable comment.

```
0040B0F7            xor     al, al                  ; al = 0
0040B0F9            jmp     short loc_40B163
```

To add a repeatable comment we would replace `idc.set_cmt(ea, comment, 0)` with `set_cmt(ea, comment, 1)`. This might be a little more useful because we would see references to branches that zero out a value and likely return 0. To get a comments we simple use `idc.get_cmt(ea, repeatable)`. `ea` is the address that contains the comment and `repeatable` is a bool of True (1) or False (0). To get the above comments we would use the following code snippet.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x40b0f7 xor     al, al                  ; al = 0
Python>idc.get_cmt(ea, False)
al = 0
```

If the comment was repeatable we would replace `idc.get_cmt(ea, False)` with `idc.get_cmt(ea, True)`. Instructions are not the only field that can have comments added. Functions can also have comments added. To add a function comment we use `idc.set_func_cmt(ea, cmt, repeatable)` and to get a function comment we call `idc.get_func_cmt(ea, repeatable)`. `ea` can be any address that is within the boundaries of the start and end of the function. `cmt` is the string comment we would like to add and `repeatable` is a Boolean value marking the comment as repeatable or not. This is represented either as 0 or False for the comment not being repeatable or 1 or True for the comment to be repeatable. Having the function comment as repeatable adds a comment whenever the function is cross-referenced, called or viewed in IDA's GUI.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x401040 push    ebp
Python>idc.get_func_name(ea)
sub_401040
Python>idc.set_func_cmt(ea, "check out later", 1)
True
```

We print the address, disassembly and function name in the first couple of lines. We then use `idc.set_func_cmt(ea, comment, repeatable)` to set a repeatable comment of `"check out later"` . If we look at the start of the function we would see our comment.

```
00401040 ; check out later
00401040 ; Attributes: bp-based frame
00401040
00401040 sub_401040 proc near
00401040                                        .
00401040 var_4     = dword ptr -4
00401040 arg_0     = dword ptr  8
00401040
00401040           push    ebp
00401041           mov     ebp, esp
00401043           push    ecx
00401044           push    723EB0D5h
```

Since the comment is repeatable, it is displayed whenever the function is viewed. This is a great place to add reminders or notes about a function.

```
00401C07           push    ecx
00401C08           call    sub_401040     ; check out later
00401C0D           add     esp, 4
```

Renaming functions and addresses is a commonly automated task, especially when dealing with position independent code (PIC), packers or wrapper functions. The reason why this is common in PIC or unpacked code is because the import table might not be present in the dump. In the case of wrapper functions the full function simply calls an API.

```
10005B3E sub_10005B3E proc near
10005B3E
10005B3E dwBytes   = dword ptr  8
10005B3E
```

```
10005B3E           push    ebp
10005B3F           mov     ebp, esp
10005B41           push    [ebp+dwBytes]          ; dwBytes
10005B44           push    8                      ; dwFlags
10005B46           push    hHeap                  ; hHeap
10005B4C           call    ds:HeapAlloc
10005B52           pop     ebp
10005B53           retn
10005B53 sub_10005B3E endp
```

In the above code the function could be called `w_HeapAlloc`. The `w_` is short for wrapper. To rename an address we can use the function `idc.set_name(ea, name, SN_CHECK)`. `ea` is the address and `name` is the string name such as `"w_HeapAlloc"`. To rename a function `ea` needs to be the first address of the function. To rename the function of our HeapAlloc wrapper we would use the following code.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x10005b3e push    ebp
Python>idc.set_name(ea, "w_HeapAlloc", SN_CHECK)
True
```

`ea` is the first address in the function and `name` is `"w_HeapAlloc"`.

```
10005B3E w_HeapAlloc proc near
10005B3E
10005B3E dwBytes   = dword ptr  8
10005B3E
10005B3E           push    ebp
10005B3F           mov     ebp, esp
10005B41           push    [ebp+dwBytes]          ; dwBytes
10005B44           push    8                      ; dwFlags
10005B46           push    hHeap                  ; hHeap
10005B4C           call    ds:HeapAlloc
10005B52           pop     ebp
10005B53           retn
10005B53 w_HeapAlloc endp
```

Above we can see the function has been renamed. To confirm it has been renamed we can use `idc.get_func_name(ea)` to print the new function's name.

```
Python>idc.get_func_name(ea)
w_HeapAlloc
```

To rename an operand we first need to get the address of it. At address `004047B0` we have a dword that we would like to rename.

```
.text:004047AD           lea     ecx, [ecx+0]
```

```
.text:004047B0           mov     eax, dword_41400C
.text:004047B6           mov     ecx, [edi+4BCh]
```

To get the operand value we can use `get_operand_value(ea, n)`.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x4047b0 mov     eax, dword_41400C
Python>op = idc.get_operand_value(ea, 1)
Python>print hex(op), idc.generate_disasm_line(op, 0)
0x41400c dd 2
Python>idc.set_name(op, "BETA", SN_CHECK)
True
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0x4047b0 mov     eax, BETA[esi]
```

In the first line we print the current working address. We assign the second operand value `dword_41400C` to `op` by calling `idc.get_operand_value(ea, n)`. We pass the address of the operand to `idc.set_name(ea, name, SN_CHECK)` and then print the newly renamed operand.

Now that we have a good basis of knowledge we can use what we have learned so far to automate the naming of wrapper functions. Please see the inline comments to get an idea about the logic.

```
import idautils


def rename_wrapper(name, func_addr):
    if idc.set_name(func_addr, name, SN_NOWARN):
        print "Function at 0x%x renamed %s" % (func_addr, idc.get_func_name(func))
    else:
        print "Rename at 0x%x failed. Function %s is being used." % (func_addr,
name)
    return


def check_for_wrapper(func):
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    # skip library & thunk functions
    if flags & FUNC_LIB or flags & FUNC_THUNK:
        return
    dism_addr = list(idautils.FuncItems(func))
    # get length of the function
    func_length = len(dism_addr)
    # if over 32 lines of instruction return
    if func_length > 0x20:
        return
    func_call = 0
    instr_cmp = 0
    op = None
    op_addr = None
    op_type = None
    # for each instruction in the function
    for ea in dism_addr:
```

```
        m = idc.print_insn_mnem(ea)
        if m == 'call' or m == 'jmp':
            if m == 'jmp':
                temp = idc.get_operand_value(ea, 0)
                # ignore jump conditions within the function boundaries
                if temp in dism_addr:
                    continue
            func_call += 1
            # wrappers should not contain multiple function calls
            if func_call == 2:
                return
            op_addr = idc.get_operand_value(ea, 0)
            op_type = idc.get_operand_type(ea, 0)
        elif m == 'cmp' or m == 'test':
            # wrappers functions should not contain much logic.
            instr_cmp += 1
            if instr_cmp == 3:
                return
        else:
            continue
    # all instructions in the function have been analyzed
    if op_addr == None:
        return
    name = idc.get_name(op_addr, ida_name.GN_VISIBLE)
    # skip mangled function names
    if "[" in name or "$" in name or "?" in name or "@" in name or name == "":
        return
    name = "w_" + name
    if op_type == 7:
        if idc.get_func_attr(op_addr, FUNCATTR_FLAGS) & FUNC_THUNK:
            rename_wrapper(name, func)
            return
    if op_type == 2 or op_type == 6:
        rename_wrapper(name, func)
        return


for func in idautils.Functions():
    check_for_wrapper(func)
```

Example Output

```
Function at 0xa14040 renamed w_HeapFree
Function at 0xa14060 renamed w_HeapAlloc
Function at 0xa14300 renamed w_HeapReAlloc
Rename at 0xa14330 failed. Function w_HeapAlloc is being used.
Rename at 0xa14360 failed. Function w_HeapFree is being used.
Function at 0xa1b040 renamed w_RtlZeroMemory
```

Most of the code should be familiar. One notable difference is the use of `idc.set_name(ea, name, flag)` from `rename_wrapper`. We use this function because `idc.set_name` throws a warning dialogue if the function name is already in use. By passing a flag value of `SN_NOWARN` or 256 we avoid the dialogue box. We could apply some logic to rename the function to `w_HeapFree_1` but for brevity we will leave that out.

## Accessing Raw Data

Being able to access raw data is essential when reverse engineering. Raw data is the binary representation of the code or data. We can see the raw data or bytes of the instructions on the left side following the address.

```
00A14380 8B 0D 0C 6D A2 00          mov     ecx, hHeap
00A14386 50                         push    eax
00A14387 6A 08                      push    8
00A14389 51                         push    ecx
00A1438A FF 15 30 11 A2 00          call    ds:HeapAlloc
00A14390 C3                         retn
```

To access the data we first need to decide on the unit size. The naming convention of the APIs used to access data is the unit size. To access a byte we would call `idc.get_wide_byte(ea)` or to access a word we would call `idc.get_wide_word(ea)`, etc.

- `idc.get_wide_byte(ea)`
- `idc.get_wide_word(ea)`
- `idc.get_wide_dword(ea)`
- `idc.get_qword(ea)`
- `idc.GetFloat(ea)`
- `idc.GetDouble(ea)`

If the cursor was at `00A14380` in the assembly from above we would have the following output.

```
Python>print hex(ea), idc.generate_disasm_line(ea, 0)
0xa14380 mov     ecx, hHeap
Python>hex( idc.get_wide_byte(ea) )
0x8b
Python>hex( idc.get_wide_word(ea) )
0xd8b
Python>hex( idc.get_wide_dword(ea) )
0x6d0c0d8b
Python>hex( idc.get_qword(ea) )
0x6a5000a26d0c0d8bL
Python>idc.GetFloat(ea) # Example not a float value
2.70901711372e+27
Python>idc.GetDouble(ea)
1.25430839165e+204
```

When writing decoders it is not always useful to get a single byte or read a dword but to read a block of raw data. To read a specified size of bytes at an address we can use `idc.get_bytes(ea, size, use_dbg=False)`. The last argument is optional and is only needed if we wanted the debuggers memory.

```
Python>for byte in idc.get_bytes(ea, 6):
    print "0x%X" % ord(byte),
0x8B 0xD 0xC 0x6D 0xA2 0x0
```

It should be noted that `idc.get_bytes(ea, size)` returns the char representation of the byte(s). This is different than `idc.get_wide_word(ea)` or `idc.get_qword(ea)` which returns an integer.

## Patching

Sometimes when reversing malware the sample contains strings that are encoded. This is done to slow down the analysis process and to thwart using a strings viewer to recover indicators. In situations like this patching the IDB is useful. We could rename the address but renaming is limited. This is due to the naming convention restrictions. To patch an address with a value we can use the following functions.

- `idc.patch_byte(ea, value)`
- `idc.patch_word(ea, value)`
- `idc.patch_dword(ea, value)`

`ea` is the address and `value` is the integer value that we would like to patch the IDB with. The size of the value needs to match the size specified by the function name we choose. One example that we found the following encoded strings.

```
.data:1001ED3C aGcquEUdg_bUfuD db 'gcqu^E]~UDG_B[uFU^DC',0
.data:1001ED51                 align 8
.data:1001ED58 aGcqs_cuufuD    db 'gcqs\_CUuFU^D',0
.data:1001ED66                 align 4
.data:1001ED68 aWud@uubQU      db 'WUD@UUB^Q]U',0
.data:1001ED74                 align 8
```

During our analysis we were able to identify the decoder function.

```
100012A0            push    esi
100012A1            mov     esi, [esp+4+_size]
100012A5            xor     eax, eax
100012A7            test    esi, esi
100012A9            jle     short _ret
100012AB            mov     dl, [esp+4+_key]      ; assign key
100012AF            mov     ecx, [esp+4+_string]
100012B3            push    ebx
100012B4
100012B4 _loop:                                   ;
100012B4            mov     bl, [eax+ecx]
100012B7            xor     bl, dl                ; data ^ key
100012B9            mov     [eax+ecx], bl         ; save off byte
100012BC            inc     eax                   ; index/count
100012BD            cmp     eax, esi
100012BF            jl      short _loop
```

```
100012C1           pop     ebx
100012C2
100012C2 _ret:                                   ;
100012C2           pop     esi
100012C3           retn
```

The function is a standard XOR decoder function with arguments of size, key and a decoded buffer.

```
Python>start = idc.read_selection_start()
Python>end = idc.read_selection_end()
Python>print hex(start)
0x1001ed3c
Python>print hex(end)
0x1001ed50
Python>def xor(size, key, buff):
    for index in range(0, size):
        cur_addr = buff + index
        temp = idc.get_wide_byte(cur_addr ) ^ key
        idc.patch_byte(cur_addr, temp)
Python>
Python>xor(end - start, 0x30, start)
Python>idc.get_strlit_contents(start)
WSAEnumNetworkEvents
```

We select the highlighted data address start and end using `idc.read_selection_start()` and `idc.read_selection_end()`. Then we have a function that reads the byte by calling `idc.get_wide_byte(ea)`, XOR the byte with key passed to the function and then patch the byte by calling `idc.patch_byte(ea, value)`.

## Input and Output

Importing and exporting files into IDAPython can be useful when we do not know the file path or when we do not know where the user wants to save their data. To import or save a file by name we use `ida_kernwin.ask_file(forsave, mask, prompt)`. `forsave` can be a value of `0` if we want to open a dialog box or `1` is we want to open the save dialog box. `mask` is the file extension or pattern. If we want to open only `.dll`files we would use a mask of `"*.dll"` and `prompt` is the title of the window. A good example of input and output and selecting data is the following `IO_DATA` class.

```
import sys
import idaapi


class IO_DATA():
    def __init__(self):
        self.start = idc.read_selection_start()
        self.end = idc.read_selection_end()
        self.buffer = ''
        self.ogLen = None
        self.status = True
        self.run()
```

```
    def checkBounds(self):
        if self.start is BADADDR or self.end is BADADDR:
            self.status = False

    def getData(self):
        """get data between start and end put them into object.buffer"""
        self.ogLen = self.end - self.start
        self.buffer = ''
        try:
            for byte in idc.get_bytes(self.start, self.ogLen):
                self.buffer = self.buffer + byte
        except:
            self.status = False
        return

    def run(self):
        """basically main"""
        self.checkBounds()
        if self.status == False:
            sys.stdout.write('ERROR: Please select valid data\n')
            return
        self.getData()


def patch(self, temp=None):
    """patch idb with data in object.buffer"""
    if temp != None:
        self.buffer = temp
        for index, byte in enumerate(self.buffer):
            idc.patch_byte(self.start + index, ord(byte))


def importb(self):
    '''import file to save to buffer'''
    fileName = ida_kernwin.ask_file(0, "*.*", 'Import File')
    try:
        self.buffer = open(fileName, 'rb').read()
    except:
        sys.stdout.write('ERROR: Cannot access file')


def export(self):
    '''save the selected buffer to a file'''
    exportFile = ida_kernwin.ask_file(1, "*.*", 'Export Buffer')
    f = open(exportFile, 'wb')
    f.write(self.buffer)
    f.close()


def stats(self):
    print "start: %s" % hex(self.start)
    print "end:   %s" % hex(self.end)
    print "len:   %s" % hex(len(self.buffer))
```

With this class data can be selected saved to a buffer and then stored to a file. This is useful for encoded or encrypted data in an IDB. We can use `IO_DATA` to select the data decode the buffer in

Python and then patch the IDB. Example of how to use the IO_DATA class.

```
Python>f = IO_DATA()
Python>f.stats()
start: 0x401528
end:   0x401549
len:   0x21
```

Rather than explaining each line of the code it would be useful for the reader to go over the functions one by one and see how they work. The below bullet points explain each variable and what the functions does. `obj` is whatever variable we assign the class. `f` is the `obj` in `f = IO_DATA()`.

- obj.start
  - contains the address of the start of the selected offset
- . obj.end
  - contains the address of the end of the selected offset.
- obj.buffer
  - contains the binary data.
- obj.ogLen
  - contains the size of the buffer.
- obj.getData()
  - copies the binary data between obj.start and obj.end to obj.buffer obj.run() the selected data is copied to the buffer in a binary format
- obj.patch()
  - patch the IDB at obj.start with the data in the obj.buffer.
- obj.patch(d)
  - patch the IDB at obj.start with the argument data.
- obj.importb()
  - opens a file and saves the data in
- obj.buffer. obj.export()
  - exports the data in obj.buffer to a save as file.
- obj.stats()
  - print hex of obj.start, obj.end and obj.buffer length.

## Intel Pin Logger

Pin is a dynamic binary instrumentation framework for the IA-32 and x86-64. Combing the dynamic analysis results of PIN with the static analysis of IDA makes it a powerful mix. A hurdle for combing IDA and Pin is the initial setup and running of Pin. The below steps are the 30 second (minus downloads) guide to installing, executing a Pintool that traces an executable and adds the executed addresses to an IDB.

```
Notes about steps
    * Pre-install Visual Studio 2010 (vc10) or 2012 (vc11)
```

```
    * If executing malware do steps 1,2,6,7,8,9,10 & 11 in an analysis machine
1. Download PIN
    * https://software.intel.com/en-us/articles/pintool-downloads
    * Compiler Kit is for version of Visual Studio you are using.
2. Unzip pin to the root dir and rename the folder to "pin"
    * example path C:\pin\
    * There is a known but that Pin does not always parse the arguments correctly
if there is spacing in the file path
3. Open the following file in Visual Studio
    * C:\pin\source\tools\MyPinTool\MyPinTool.sln
        - This file contains all the needed setting for Visual Studio.
        - Useful to back up and reuse the directory when starting new pintools.
4. Open the below file, then cut and paste the code into MyPinTool.cpp (currently
opened in Visual Studio)
    * C:\pin\source\tools\ManualExamples\itrace.cpp
        - This directory along with ../SimpleExamples is very useful for example
code.
5. Build Solution (F7)
6. Copy traceme.exe to C:\pin
7. Copy compiled MyPinTool.dll to C:\pin
    * path C:\pin\source\tools\MyPinTool\Debug\MyPinTool.dll
8. Open a command line and set the working dir to C:\pin
9. Execute the following command
    * pin -t traceme.exe -- MyPinTool.dll
        - "-t" = name of file to be analyzed
        - "-- MyPinTool.dll" = specifies that pin is to use the following
pintool/dll
10. While pin is executing open traceme.exe in IDA.
11. Once pin has completed (command line will have returned) execute the following
in IDAPython
    * The pin output (itrace.out) must be in the working dir of the IDB. \
```

`itrace.cpp` is a pintool that prints the EIPs of every instruction executed to `itrace.out`. The data looks like the following output.

```
00401500
00401506
00401520
00401526
00401549
0040154F
0040155E
00401564
0040156A
```

After the pintools has executed we can run the following IDAPython code to add comments to all the executed addresses. The output file `itrace.out` needs to be in the working directory of the IDB.

```
f = open('itrace.out', 'r')
lines = f.readlines()

for y in lines:
    y = int(y, 16)
```

```
    idc.set_color(y, CIC_ITEM, 0xfffff)
    com = idc.get_cmt(y, 0)
    if com == None or 'count' not in com:
        idc.set_cmt(y, "count:1", 0)
    else:
        try:
            count = int(com.split(':')[1], 16)
        except:
            print hex(y)
        tmp = "count:0x%x" % (count + 1)
        idc.set_cmt(y, tmp, 0)
f.close()
```

We first open up `itrace.out` and read all lines into a list. We then iterate over each line in the list. Since the address in the output file was in hexadecimal string format we need to convert it into an integer.

```
.text:00401500 loc_401500:                            ; CODE XREF:
sub_4013E0+106□j
.text:00401500           cmp     ebx, 457F4C6Ah        ; count:0x16
.text:00401506           ja      short loc_401520      ; count:0x16
.text:00401508           cmp     ebx, 1857B5C5h        ; count:1
.text:0040150E           jnz     short loc_4014E0      ; count:1
.text:00401510           mov     ebx, 80012FB8h        ; count:1
.text:00401515           jmp     short loc_4014E0      ; count:1
.text:00401515 ; ---------------------------------------------------------
.text:00401517           align 10h
.text:00401520
.text:00401520 loc_401520:                            ; CODE XREF:
sub_4013E0+126□j
.text:00401520           cmp     ebx, 4CC5E06Fh        ; count:0x15
.text:00401526           ja      short loc_401549      ; count:0x15
```

## Batch File Generation

Sometimes it can be useful to create IDBs or ASMs for all the files in a directory. This can help save time when analyzing a set of samples that are part of the same family of malware. It's much easier to do batch file generation than doing it manually on a large set. To do batch analysis we need to pass the `-B` argument to the text`idaw.exe`. The below code can be copied to the directory that contains all the files we would like to generate files for.

```
import os
import subprocess
import glob

paths = glob.glob("*")
ida_path = os.path.join(os.environ['PROGRAMFILES'], "IDA", "idat.exe")

for file_path in paths:
    if file_path.endswith(".py"):
        continue
```

```
    subprocess.call([ida_path, "-B", file_path])
```

We use `glob.glob("*")` to get a list of all files in the directory. The argument can be modified if we wanted to only select a certain regular expression pattern or file type. If we wanted to only get files with a `.exe` extension we would use `glob.glob("*.exe")`. `os.path.join(os.environ['PROGRAMFILES'], "IDA", "idat.exe")` is used to the get the path to `idat.exe`. Some versions of IDA have a folder name with the version number present. If this is the case the argument `"IDA"` needs to be modified to the folder name. Also, the whole command might have to be modified if we choose to use a non-standard install location for IDA. For now let's assume the install path for IDA is `C:\Program Files\IDA`. After we found the path we loop through all the files in the directory that do not contain a `.py` extension and then pass them to IDA. For an individual file it would look like `C:\Program Files\IDA\idat.exe -B bad_file.exe`. Once ran it would generate an ASM and IDB for the file. All files will be written in the working directory. An example output can be seen below.

```
C:\injected>dir

0?/**/____  09:30 AM    <DIR>          .
0?/**/____  09:30 AM    <DIR>          ..
0?/**/____  10:48 AM           167,936 bad_file.exe
0?/**/____  09:29 AM               270 batch_analysis.py
0?/**/____  06:55 PM           104,889 injected.dll

C:\injected>python batch_analysis.py

Thank you for using IDA. Have a nice day!

C:\injected>dir

0?/**/____  09:30 AM    <DIR>          .
0?/**/____  09:30 AM    <DIR>          ..
0?/**/____  09:30 AM           506,142 bad_file.asm
0?/**/____  10:48 AM           167,936 bad_file.exe
0?/**/____  09:30 AM         1,884,601 bad_file.idb
0?/**/____  09:29 AM               270 batch_analysis.py
0?/**/____  09:30 AM           682,602 injected.asm
0?/**/____  06:55 PM           104,889 injected.dll
0?/**/____  09:30 AM         1,384,765 injected.idb
```

`bad_file.asm`, `bad_file.idb`, `injected.asm` and `injected.idb` were generated files.

## Executing Scripts

IDAPython scripts can be executed from the command line. We can use the following code to count each instruction in the IDB and then write it to a file named `instru_count.txt`.

```
import idc
import idaapi
```

```
import idautils

idaapi.autoWait()

count = 0
for func in idautils.Functions():
    # Ignore Library Code
    flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
    if flags & FUNC_LIB:
        continue
    for instru in idautils.FuncItems(func):
        count += 1

f = open("instru_count.txt", 'w')
print_me = "Instruction Count is %d" % (count)
f.write(print_me)
f.close()

idc.Exit(0)
```

From a command line perspective the two most important functions are `idaapi.autoWait()` and `idc.Exit(0)`. When IDA opens a file it is important to wait for the analysis to complete. This allows IDA to populate all functions, structures, or other values that are based on IDA's analysis engine. To wait for the analysis to complete we call `idaapi.autoWait()`. It will wait/pause until IDA is completed with its analysis. Once the analysis is completed it returns control back to the script. It is important to execute this at the beginning of the script before we call any IDAPython functions that rely on the analysis to be completed. Once our script has executed we need to call `idc.Exit(0)`. This stops execution of our script, close out the database and return to the caller of the script. If not our IDB would not be closed properly.

If we wanted to execute the IDAPython to count all lines we IDB we would execute the following command line.

```
C:\Cridix\idbs>"C:\Program Files (x86)\IDA 6.3\idat.exe" -A -Scount.py
cur-analysis.idb
```

`-A` is for Autonomous mode and `-S` signals for IDA to run a script on the IDB once it has opened. In the working directory we would see a file named `instru_count.txt` that contained a count of all instructions.

# Closing

I hope you gained some knowledge on how to use IDAPython or a trick to solve an issue you are working on. If you have any questions, comments or feedback please send me an email. I plan to keep editing the book. Please not the version number and check it out again in the future. Cheers.