开篇与介绍

一、前言

《IDA脚本开发之旅》是一个新的系列文章，它适合所有对 IDA 脚本开发感兴趣，并且希望用它处理逆向场景里各种需求的所有朋友。这是一个完全值得一看的新系列，毫不夸张的说，在我本人学习 IDA 时，真希望看到这么一套教程！希望它也会得到你的喜爱。

二、背景简述

没有哪个软件能够直接满足所有使用者的所有需求，IDA 也不能。即使是高频的版本更新，往往也只能处理最主要的一部分诉求，无法兼顾所有需求。为了处理和满足用户各种各样的自定义需求，IDA 提供了一种机制，让用户可以用编程的形式对 IDA 做扩展。

回想一下日常的分析场景，其中有不少自定义的需求。比如识别加密算法、字符串批量解密、去除花指令、启发式的自动反混淆工具、对 VMP 的辅助分析等等。

其中一些需求相对轻松和表层，可能不到一百行代码就能处理，比如常规的字符串解密和花指令去除；另一些需求则相对复杂，至少上千行代码才能搞定，比如设计一个启发式的去混淆插件。

根据任务的复杂程度等因素，这种扩展可以延展为两种不同的机制——脚本和插件。

脚本：灵活方便，适合处理小型任务，比如字符串加密。

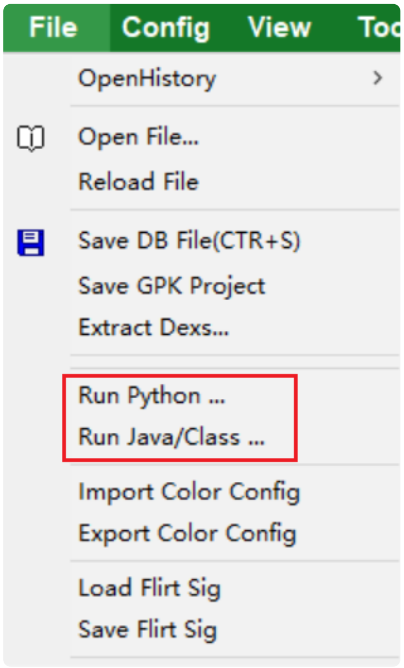
插件：复杂笨重，可以处理更大型或更多类型的任务，比如设计一个新的反编译模块。

当我们的需求比较小时，脚本是好选择，需求比较复杂而且要求持久化时，可以写一个插件。需要注意，一般来说，用脚本能做到的事，用插件一定也能做到，反过来却不一定，因为脚本引擎的核心功能来源于插件所提供的接口。

脚本和插件就像公路上的机动车专用道以及正常车道，各有用处，一方灵活简洁但功能相对弱，另一方笨重但功能强大。这么做比只有一种统一的扩展更合理和好用。

在介绍 IDA 的脚本与插件功能之前，我们先看看其它主流反编译工具怎么处理这个问题。

首先看 GDA，对这个功能强大且由国人开发的 Android 反编译工具，大家一定不陌生。GDA 支持扩展，但没有做脚本和插件的区分，因此你可以叫它脚本，也可以叫它插件，我认为它更像脚本。



GDA 脚本引擎支持 JAVA 和 Python 两种语言，两者平级且提供相同作用的 API 。下面是一个 GDA 字符串解密脚本，看起来还不错吧？具体内容可以看这篇 文章 <http://www.gda.wiki:9090/decodestr.php> 。

Python 复制代码

```python
import GdaImport
#gjden
#example of decoding strings which is located by smali code
def printStringHex(stri):
    ret=''
    for ch in stri:...
    return ret
def decodeString(gda,idx):
    rawstr=gda.GetStringById(idx)
    if rawstr==None:
        return ''
    stri=rawstr
    ret=list(stri)
    i=len(stri)-1
    xx=''
    while i>= 0:
        ret[i]=chr(ord(stri[i])^39)
        if i <= 0:
            break
        i=i-1
        ret[i]=chr(ord(stri[i])^101)
        i=i-1
    xx = ''.join(ret)
    return xx

def GDA_MAIN(gda_obj):
    gda=gda_obj
    Dex0=gda.DexList[0]
    midx=0x4fc9
    method=Dex0.MethodTable[str(midx)]
    clist=method.callorIdxList
    destr=''
    callorTable={}
    strIdxTable={}
    for idx in clist:
        if callorTable.has_key(str(idx)):
            continue
        #dump smali code of callors
        smalicode=gda.GetSmaliCodeById(idx)
        splitstr=smalicode.split('\r\n')
        i=0
        for sstr in splitstr:
            if '@4fc9' in sstr:
                line=splitstr[i-1]
                if 'string@' in line:
                    pos=line.find('ing@')+4
                    strIdx=line[pos:pos+4]
                    if strIdxTable.has_key(strIdx):
                        i=i+1
                        continue
                    strIdxTable[strIdx]=strIdx
                    dstr=decodeString(gda,int(strIdx,16))
```

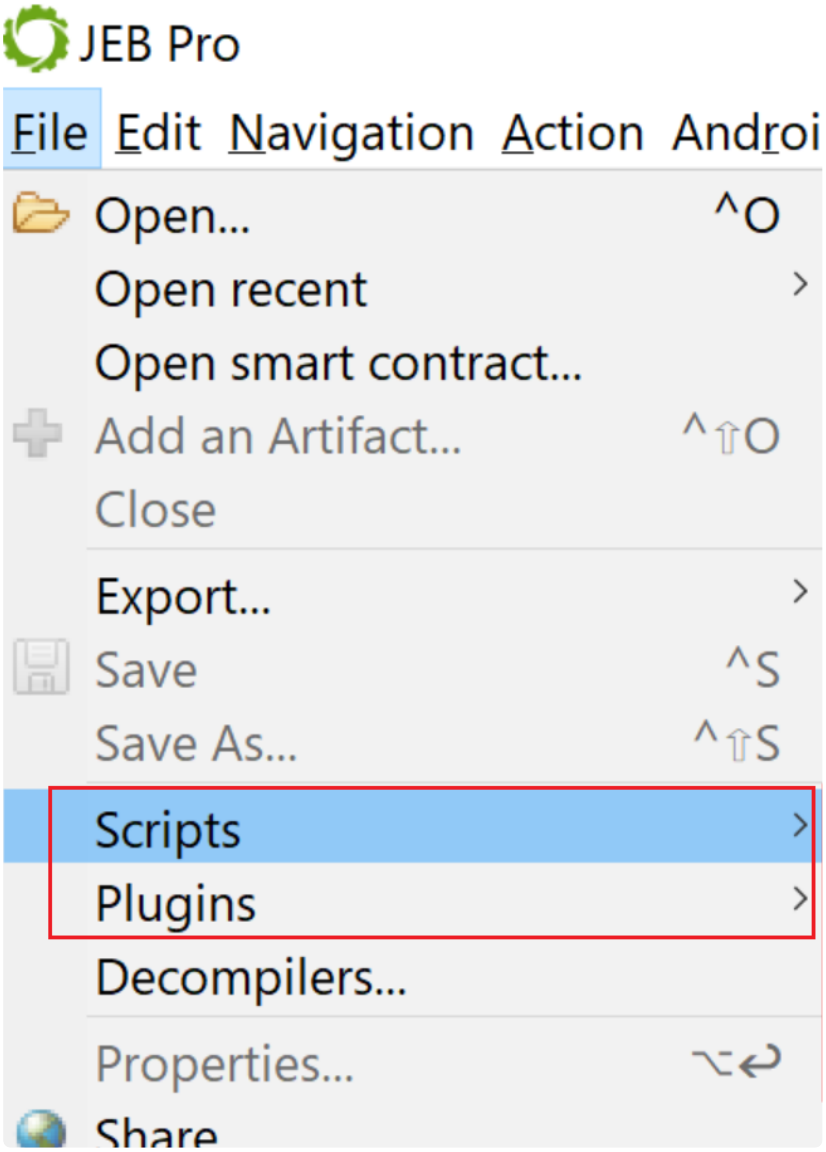
```
54                      gda.SetStringById(int(strIdx,16),dstr)
55                      destr+="[string@"
56                      destr+=strIdx
57                      destr+="] "
58                      destr+=dstr
59                      destr+='\n'
60              i=i+1
61      gda.log(destr)
62      return 0
```

为什么 GDA 提供了功能等价的两种语言支持？这不难理解，我们可以列一下原因。

- Python 简单易学，生态丰富，是首选的脚本开发语言。
- 作为一个 JAVA 反编译器，和反编译的 JAVA 代码会频繁打交道，用 JAVA 写脚本会更方便。以字符串解密举例，当脚本引擎中需要执行样本的字符串解密函数时，如果用 Python 脚本，需要先用 Python 重新实现解密函数，而如果用 JAVA 脚本，直接拷贝反编译的 JAVA 代码就行，不需要复写。这是很明显和直接的优势，Unidbg 比 AndroidNativeEmu <https://github.com/AeonLucid/AndroidNativeEmu> 好用的一个原因也在这里，前者是用 JAVA 写的，而后者是用 Python 写的。在补 JNI 环境时，直接抄 JAVA 代码肯定比用 Python 复写舒服的多。

如果读者对在 GDA 中编写脚本感兴趣，可以看 GDA 官方的 Github <https://github.com/charles2gan/GDA-android-reversing-Tool> ，里面有用 JAVA + Python 双语展示的各类脚本。

下面浅谈 JEB ，作为一个以 IDA 为目标的反编译器，它在各方面都做的十分规范和地道。JEB 完整支持脚本和插件两种性质的扩展，在语言上支持 Python 和 JAVA ，对它感兴趣的可以看看 这篇 <https://blog.flanker017.me/writing-jeb2-plugin/> 文章。



下面吐槽 JADX，它是一个反面教材。JADX 在设计上没有考虑对扩展的支持，这意味着如果你想根据自己的需求给 JADX 添加一个功能，只有两个选择。一是给 JADX 提 issue，等待作者在数个月后做更新，如果作者对实现它并不感兴趣，你只能考虑第二个方案，对 JADX 的源码做修改与重新编译。比如 pkiller 的 super-jadx <https://github.com/pkillller/super-jadx> 项目就是此种情况的产物，详情可见这篇 文章 <https://pkiller.com/android/super-jadx_v1.0.0/> 。我想，如果 JADX 支持一定的扩展，那么解决问题的方案就会优雅很多。

JADX 的作者 skylot 于今年五月表示，他计划给 JADX 增加插件功能，但这可能在比较久以后才会实现，因为需要对代码做大改动。

最后简单谈论一下 Ghidra 的扩展设计，它提供了 JAVA 和 Python 等多种语言支持的脚本与插件功能，代码编写体验也很好。Ghidra 还有非常整洁规范的文档以及官方的插件管理器，不愧是 NSA 出品的工具，十分优雅。对密码学有了解的朋友对 NSA（美国国家安全局） 一定印象深刻，它在密码学上的研究领先世界很多年，是非常可怕的机构 。

聊了这么多其他逆向工具，你可能对 IDA 所支持的扩展形式充满了兴趣，IDA 作为逆向行业的标杆，它会支持何种形式的扩展？但你可能会失望， IDA 所提供的扩展机制，相比较其他工具，并不更好或更完美。

三、IDA 的扩展体系

上文我们说到，IDA 所提供的扩展体系并不完美，在某种程度上，这和 IDA 的悠久历史有很大关联。

IDA 诞生于 90 年代初，至今已有超过 30 年的历史，而它作为一个正式的商业软件发版，也有超过 20年的历史。不得避免的，它存在许多历史遗留问题。

在最初，IDA 就提供了脚本和插件两种扩展实现。插件的开发依赖于 IDA 的 `C++ SDK` ，也就是说可以用 C++ 开发 IDA 插件，而脚本的开发依赖于内置的 IDC 脚本引擎。

用户可以用 C++写 IDA 插件， 用 IDC 写 IDA 脚本。这是最初的设计。那么这个最初的设计有什么问题呢？我们要从 IDC 聊起。IDC 是 IDA 原生支持的脚本语言，它可以方便的实现各种简单功能。比如下面的代码就是一个小的 IDC 脚本，它可以在调试时 dump 指定范围的内存。

C | 复制代码

```
auto i,fp;
fp = fopen("d:\\dump.so","wb");
for (i = start_address; i <= end_address; i++)
     fputc(Byte(i),fp);
```

但 IDC 并不是一个受欢迎的脚本语言，入行较晚的朋友很可能没用过它。第一个原因就是它存在一定的学习门槛。在最初，IDC 借鉴了 C 语言的语法，IDC 即“IDA 中的 C 语言”之意。IDA 5.6 版本以后，IDC 借用了 C++ 在面向对象和异常处理上的相关特性。又因为 IDC 的定位是脚本语言，使用上追求灵活方便，所以 IDC 存在另外一些特性，比如它的数据类型十分松散，局部变量用 auto 声明即可，像下面这样。

C | 复制代码

```
auto a,b,c;
auto d = 0;
```

IDC 就像一个缝合怪，因此即使你熟悉 C/C++/Python ，也不能直接上手它，因为你不清楚它采摘和融合了这些语言的哪些特性，这是第一个原因。

第二个问题在于 IDC 不够强大，IDC 环境中无法使用 C 的标准库，也不存在其他什么代码库可用。

除此之外， IDC 还有一些古怪的特性，使用时需要注意。

话说回来，我们都知道 Python 有大量的库可以用，而且它非常热门，几乎人人会用，可以最大程度降低了学习和开发的门槛。有人会想，为什么不选择 Python 作为内置的脚本语言？这就是前面说到的历史遗留问题。拜托，IDA 设计 IDC 脚本引擎是 1995 年前后的事，而 Python 的第一个主流版本 Python2 的发布时间是 2000 年。因此， IDA 在设计脚本引擎的那时候，Python 既没有好的生态，也没有广泛的受众，更没有人预料到在之后会这么火热。 同样的问题还发生在其他地方，比如 IDA 为什么不和 LLVM 在中间语言上保持一致，从时间线上看，MicroCode（IDA 反编译引擎所采用的中间语言）的设计也远早于 LLVM IR。换句话说，因为搞得比较早，现在广为人知的优秀语言和项目都还没出现，所以 IDA 要自己造一堆轮子。

继续聊 Python，它和 IDA 的故事还很长。2000年，Python2 发布，Python 生态逐步发展。Python 受到了越来越多的关注，2004年，Gergely Erdelyi 主导开发了 IDAPython，它在 IDA 中集成了 Python 解释器，并将 IDC 的功能做了包装。利用 IDAPython (本质上它是一个 IDA 插件)，我们可以在 IDA 中使用Python 编写脚本。IDAPython 受到了 IDA 社区极大的喜爱，它在 2009 年成为 IDA 的标准内置插件。

故事并未在此处结束，Python 在 2008 年发布了 Python3 版本，整个 Python 社区经历了漫长的割裂和阵痛期 —— 从 Python 2 迁移到 Python 3。IDA 也是旅途的一员，它经历了如下的转变。

- IDAPython 仅支持 Python 2
- IDAPython 默认使用 Python 2，可切换为 Python 3
- IDAPython 默认使用 Python 3，可切换为 Python 2
- IDAPython 仅支持 Python 3 （最新的 IDA 8.0）

这还没完，IDAPython 认为自己的 API 结构设计不够好，在 IDA 7.3 以后支持一套新的包结构以及命名规则，并放弃了对原先 API 的支持，前后对比如下。

Before	After	
idc.PatchByte	ida_bytes.patch_byte	
idc.PatchWord	ida_bytes.patch_word	
idc.PatchDword	ida_bytes.patch_dword	

详细的对照表可见 IDA 官方 文档 <https://hex-rays.com/products/ida/support/ida74_idapython_no_bc695_porting_guide.shtml> ，以上就是 IDA 在脚本发展这一方面所存在的历史遗留问题。它给新手们学习和复现互联网上各种 IDA 脚本和文章带来了不少阻碍，包括下面几种主要情况

- 将一个实用的 IDC 脚本改写成 Python 脚本
- 一个 Python 2 脚本需要改写成 Python 3 脚本

- 一个好用但缺少维护的 IDA 插件，在 IDA 7.5 或更高版本上，因为 API 未适配无法运行
- 手上只有 IDA 7.0 版本，想用的 IDA 插件只适配了IDA 7.5 甚至更高的版本

这些问题的解决方案并不复杂，使用上文提到的 文档 <https://hex-rays.com/products/ida/support/ida74_idapython_no_bc695_porting_guide.shtml> 可以做好这件事。首先说场景一：IDC 脚本转 Python 脚本，下面就是一个 IDC 脚本，我们进行举例。

C 复制代码

```
#include <idc.idc>

static main()
{
    auto i,pos,size,JMP_SIZE,FLOWER1_SIZE,FLOWER2_SIZE;
    pos=0x286C; //START
    size=0x1A000;//SIZE
    JMP_SIZE = 0x40;
    FLOWER1_SIZE = 0x1e;
    FLOWER2_SIZE = 0x8;


    for ( i=0; i < size;i++ ) {
        //PATCH JMPS
        if (
            (Byte(pos)==0x13)&&(Byte(pos+1)==0xe0)&&(Byte(pos+2)==0xbd)&&
            (Byte(pos+3)==0xe8)&& (Byte(pos+4)==0xf0)&&(Byte(pos+5)==0x47))
        {
            for(i=0;i<JMP_SIZE;i++)
            {
                PatchByte(pos+i,0x0);
            }
            HideArea(pos,pos+JMP_SIZE,atoa(pos),atoa(pos),atoa(pos+JMP_SIZE
            continue;
        }

        // PATCH FLOWER1
        //.text:00002A80 B1 B5                         PUSH            {R0,R4
        //.text:00002A82 82 B0                         SUB             SP, SP
        //.text:00002A84 12 46                         MOV             R2, R2
        //.text:00002A86 02 B0                         ADD             SP, SP
        //.text:00002A88 00 F1 01 00                   ADD.W           R0, R0
        //.text:00002A8C A0 F1 01 00                   SUB.W           R0, R0
        //.text:00002A90 1B 46                         MOV             R3, R3
        //.text:00002A92 BD E8 B1 40                   POP.W           {R0,R4
        //.text:00002A96 01 F1 01 01                   ADD.W           R1, R1
        //.text:00002A9A A1 F1 01 01                   SUB.W           R1, R1

        if (
            (Byte(pos)==0xb1)&&(Byte(pos+1)==0xb5)&&(Byte(pos+2)==0x82)&&(B
            (Byte(pos+0x1a)==0xa1)&&(Byte(pos+0x1b)==0xf1)&&(Byte(pos+0x1c)
        {
            for(i=0;i<FLOWER1_SIZE;i++)
            {
                PatchByte(pos+i,0x0);
            }
            HideArea(pos,pos+FLOWER1_SIZE,atoa(pos),atoa(pos),atoa(pos+FLOW
            continue;
        }

        //PATCH FLOWER2
        // "PUSH.W {R4-R10,LR}"
```

```
        // "POP.W {R4-R10,LR}"

        if (
            (Byte(pos)==0x2d)&&(Byte(pos+1)==0xe9)&&(Byte(pos+2)==0xf0)&&(B
            (Byte(pos+4)==0xbd)&&(Byte(pos+5)==0xe8)&&(Byte(pos+6)==0xf0)&&
        {
            for(i=0;i<FLOWER2_SIZE;i++)
            {
                PatchByte(pos +i,0x0);
            }
            HideArea(pos,pos+FLOWER2_SIZE,atoa(pos),atoa(pos),atoa(pos+FLOW
            continue;
        }
        pos++;
    }

    Message("\n" + "DE-FLOWERS FINISH BY Ericky\n");
}
```

比如其中出现的 `PatchByte` 方法，在 IDA 7.5 上与哪个 Python 方法对应？对于 IDA 7.3 以前的版本，IDAPython 提供的 IDC 模块负责以同名形式提供 IDC 中所有的函数，这意味着 IDC 中的 `PatchByte` 方法在 IDAPython 中是 `idc.PatchByte`，我们在文档中找到它，确认新版对应于 `ida_bytes.patch_byte`。

idc.DbgWord	idc.read_dbg_word
idc.DbgDword	idc.read_dbg_dword
idc.DbgQword	idc.read_dbg_qword
idc.DbgRead	idc.read_dbg_memory
idc.DbgWrite	idc.write_dbg_memory
idc.PatchDbgByte	idc.patch_dbg_byte
idc.PatchByte	ida_bytes.patch_byte
idc.PatchWord	ida_bytes.patch_word

有些特殊的 IDC 方法无法在表中找到，那需要 Google 做单独的处理。

再看第二个场景——脚本或插件是 Python 2，那么改写成 Python 3 就行，没什么名堂。第三或第四个场景通过对照那个文档，做一下 API 的向上或向下转换就行。

简而言之，IDA 的脚本开发，整体上是从 IDC 到 Python 2 到 Python 3 再到新版 Python API 这样一个路子。

如果你想用写一个 IDA 脚本，那么相较于IDAPython，IDC 没有任何优势，IDC 可以做的事 IDAPython 都可以做。

再谈一下插件部分，前文说过，插件开发依赖于 IDA C++ SDK。在 Windows 上你可以使用 Visual Studio 搭建对应的开发环境。IDA 社区认为，这样开发太笨重了。所以 IDA 提出了脚本化插件的概念，也就是用 IDC 或 IDAPython 写插件。IDAPython 的 Idaapi 模块负责访问大部分核心的 IDA API，即 `C++ SDK` 所提供的那些。

如果你想写一个 IDA 插件，那么 IDAPython 是一个好选择，可以观察到，近年来绝大多数流行的 IDA 插件都通过 IDAPython 开发。但它相对于 C++ 并不具备绝对优势。有下面这些原因

- 从语言角度看，写脚本可以通过 Python 或 IDC ，绝大多数人会选择 Python，而写插件可以用 Python 或 `C++` ，尽管采用 `C ++` 开发相对繁琐，但它的良好性能、丰富特性有目共睹。
- 从功能角度看，IDAPython 所能访问到的 API 是 IDA `C++ SDK` 的子集，或者说 IDAPython 并没有实现完全且良好的封装。
- 从新特性角度看，IDA 最新的功能和特性往往只提供 `C++ SDK` 的接口，IDAPython 则会在稍后的版本予以支持。比如从 IDA 7.1 开始，支持通过 `C++ SDK` 对反编译过程的最重要结构 —— MicroCode 进行访问和操控，IDA 7.2 上 IDAPython 才支持做同样的事，而且还略有不如。

在本系列文章中，我们只关注于用 IDAPython 开发脚本与插件。但 `C++` 的爱好者可以尝试用它写 IDA 插件，相关教程可以看 《Write IDA plugin in C++》，尽管它是较老的教程，但好在逆向相关的核心概念并没有发生改变，因此依然有很好的价值和参考性。

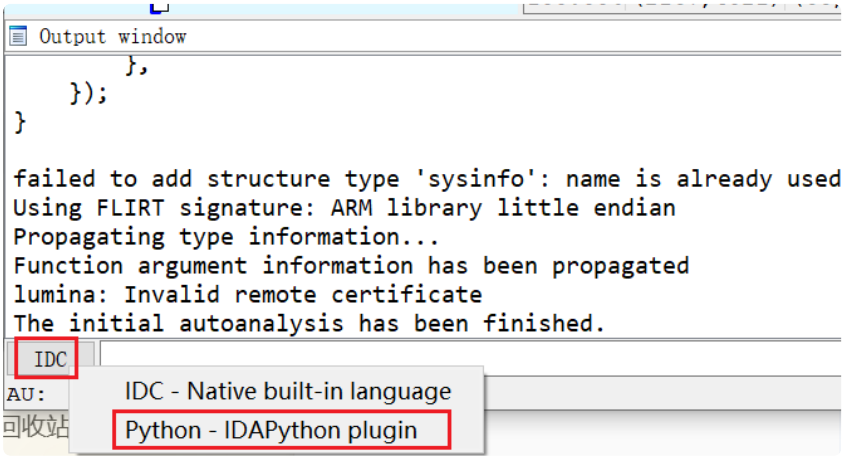
除此之外，也有很多研究人员根据自身需求以及所擅长的语言，将 IDA 的扩展封装和绑定到其他语言上。C# 的绑定有 IDACSharp <https://github.com/NewLifeX/IDACSharp> ，JavaScript 的绑定有 jside <http://sandsprite.com//blogs/index.php?uid=7&pid=361> ，JAVA 的绑定有 IDAJava，还有 Ruby 的绑定 idarub <https://github.com/spoonm/idarub> ，甚至还存在 restful API 风格的接口 idarest <https://github.com/dshikashio/idarest> 。但总体而言，它们的用处不大，受众很少，我们不会对它们做更多介绍。

四、开发环境

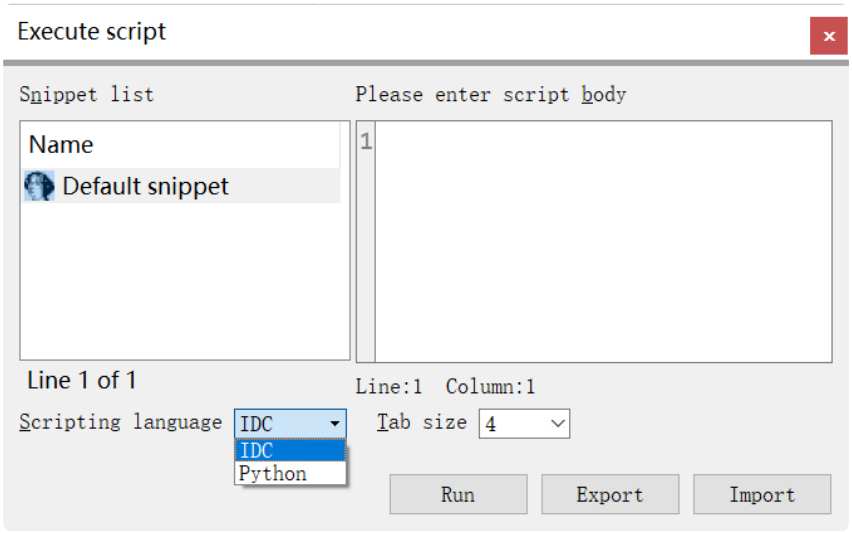
读者应当自行保证基础环境无误，包括 IDA 的环境以及 IDA 脚本开发的环境两方面。我使用的 IDA 是 Windows 下的 IDA 7.5，使用其他版本或不同系统的读者需自行做调整。

脚本开发上，我们有两方面的需求，1是对于交互式验证的需求，2是 IDE 中开发脚本或插件的需求。

首先聊一下IDA自带的三个输入方式，指IDA底部的这个交互窗口，它可以在IDC和Python之间切换，切换为Python时就像一个普通的Python命令行，但写代码的体验很差。

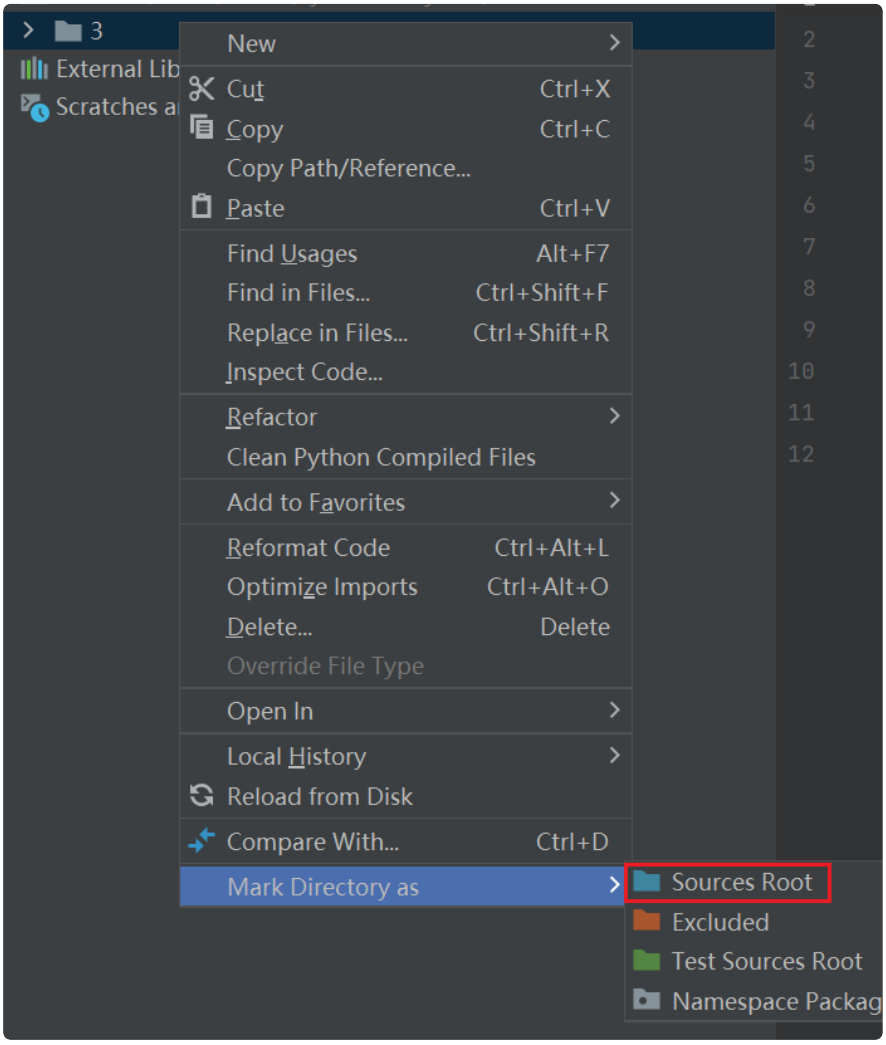


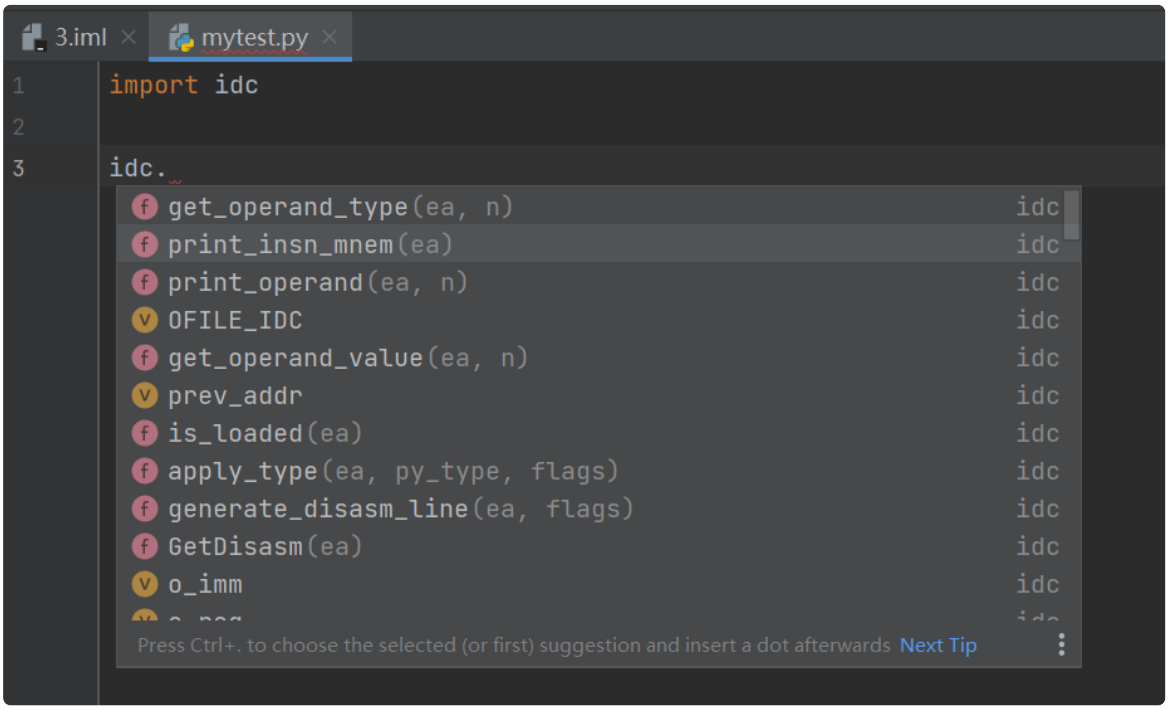
也可以通过菜单栏 File - Script command 传入代码片段，但缺少自动缩进以及代码提示。



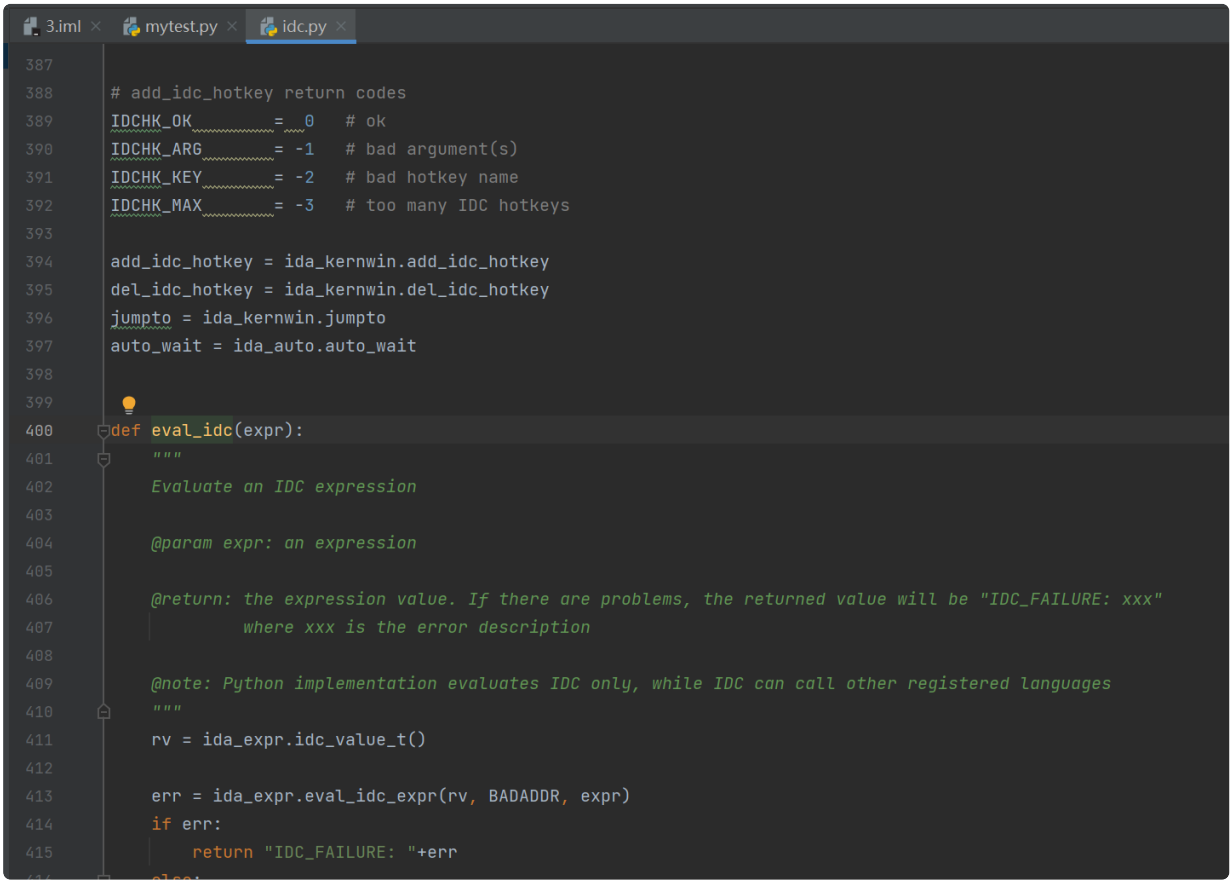
除此之外，可以通过 File - Script File 导入代码文件。但这三种显然都不是好用的代码编辑环境。

首先，为了在 IDE 中获得代码提示，可以将 yourPath/IDA/python 路径下的 3 文件夹（即 Python3）拷贝到 IDE 中，我这里使用 Pycharm，右键设置为 `Sources Root`，即可获得代码提示。





遇到 API 使用上的问题也可以直接点到方法里，看具体的详细注释。

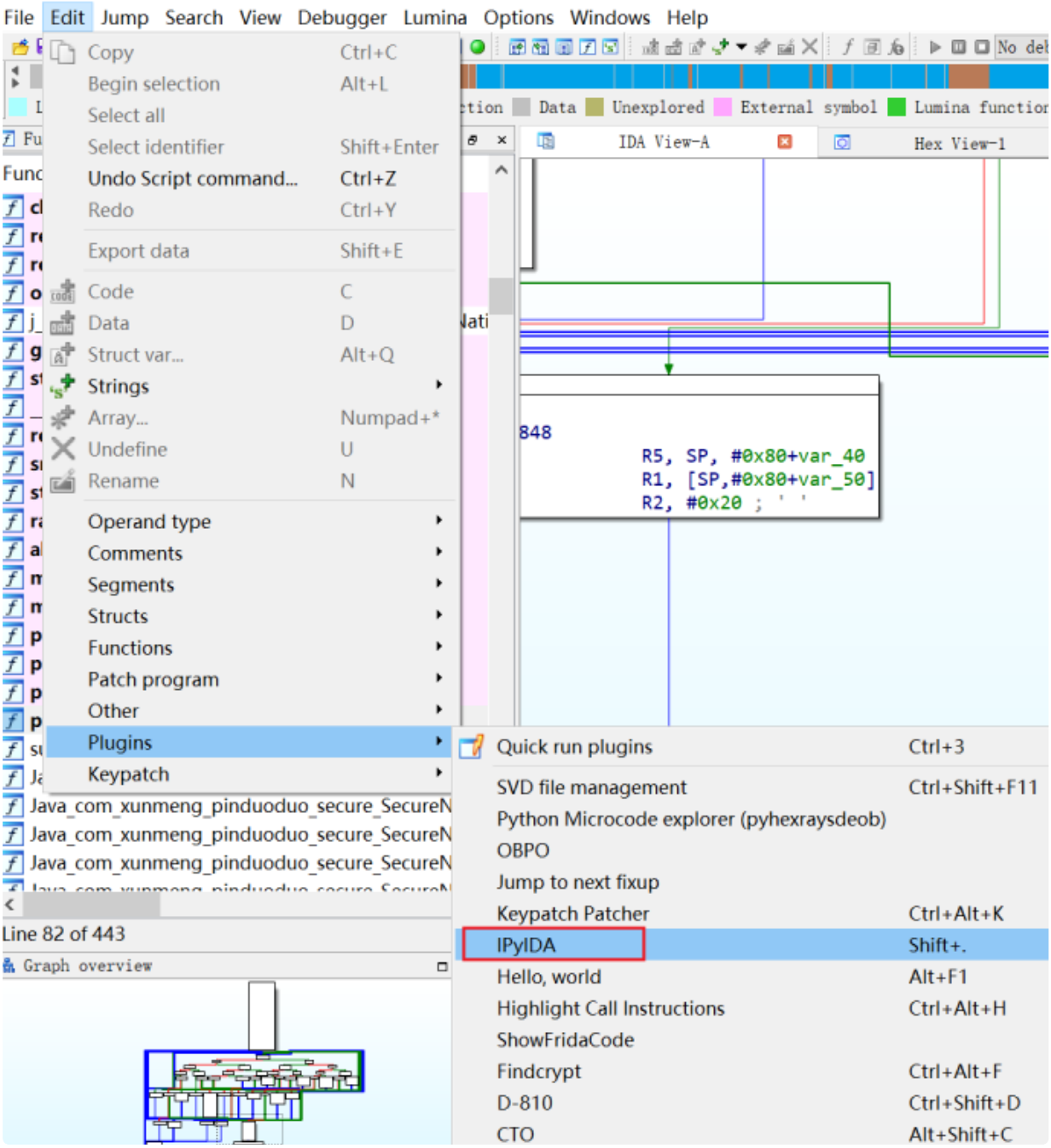


这么做之后，我们就拥有了一个具有代码补全的 IDAPython 编写环境，但需要注意，它仅仅是代码补全而已，并不能在 IDE 中直接执行 IDAPython 代码，这类似于我们在写 Frida 时的代码补全，这并不意味着我们可以在IDE 里像对待普通 JS 项目那样运行 Frida 代码。如果你习惯于 VSCODE，可以尝试 nu11 哥推荐的这个办法 <https://t.zsxq.com/05euzB6eq> 。

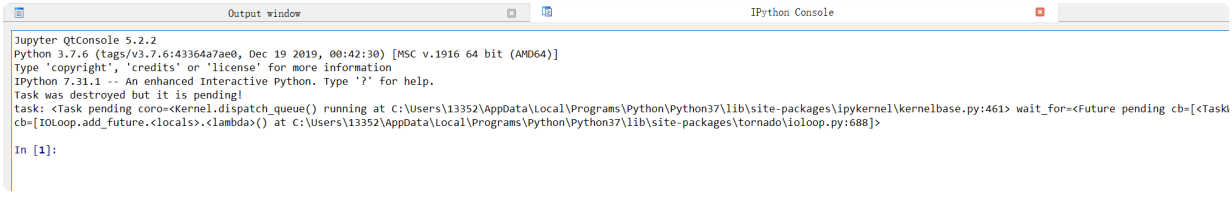
我们还需要一个快速验证代码的交互方式。有很多插件致力于提升此事的体验。

- IPyIDA <https://github.com/eset/ipyida>
- ida_ipython <https://github.com/james91b/ida_ipython>
- Python_editor <https://github.com/techbliss/Python_editor>

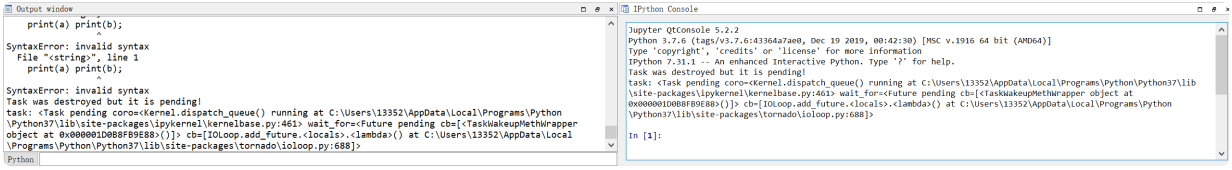
其中使用体验最好的是 IPyIDA，请读者自行安装。安装成功后开启IPyIDA插件



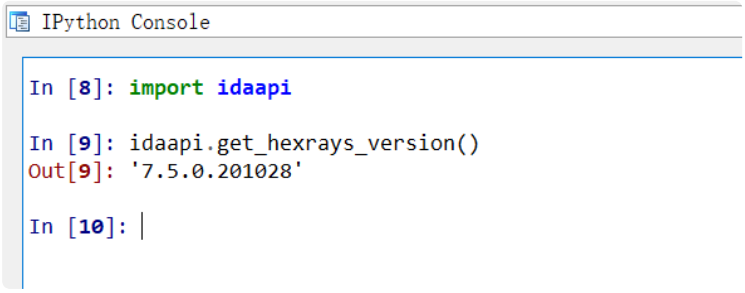
底部出现 IPython Console



不喜欢在多个窗口之间来回切换的话，可以拖动窗口，变成下面这样的布局



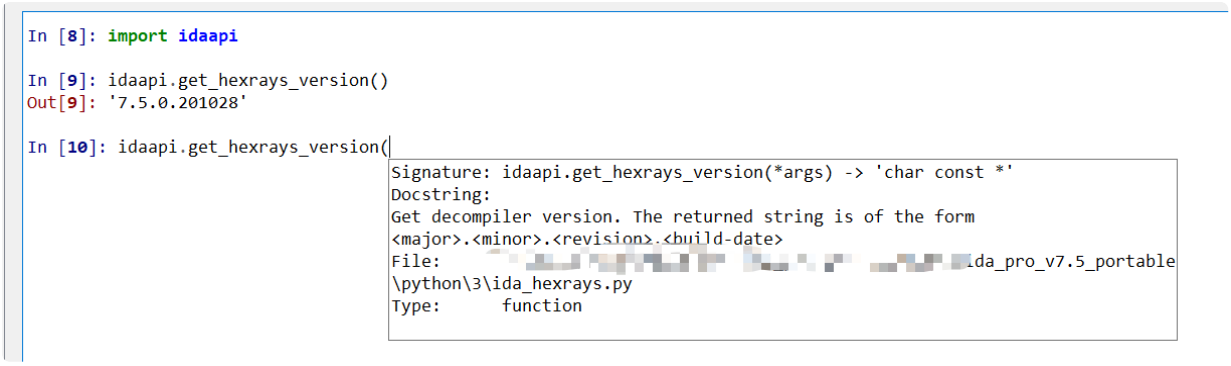
请读者自行安装IPyIDA，并测试其使用无碍



它有许多好处

1是 TAB 键代码补全

2是输入方法的左括号时浮现用法



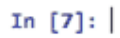
在实例代码中，我不会逐行解释代码的作用，这并无必要。

Python 复制代码

```
idc.get_func_attr(0xa5c4, 0)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_64884\2139044542.py in <module>
----> 1 idc.get_func_attr(0xa5c4, 0)

NameError: name 'idc' is not defined
```

出现报错时，请像在Python中一样，手动引入未定义的类库即可，比如这里就是import idc。不喜欢 IPyIDA 的朋友也可以尝试 ida_ipython <https://github.com/james91b/ida_ipython> ，它们都提供同类型的功能。



除了这两个需求外，还有两个次要的需求。1 是调试脚本或插件，我觉得可以代码补全+快速验证已经够用了，有需求的的可以自行了解 idacode <https://github.com/ioncodes/idacode> 项目。2 是对脚本和包进行管理。Ghidra 自带一个不赖的包管理器和一个好用的 Python Editor，IDA 并不标准内置这些玩意。在互联网上，你可以看到 IDA 使用人员所设计的各类插件管理器，我并没有实际测试过它们的可用性和易用性。读者如果对此感兴趣，也可用自己把玩下面这些项目。

- idapm <https://github.com/tkmru/idapm>
- idaenv <https://github.com/deactivated/idaenv>
- idapkg <https://github.com/Jinmo/idapkg>
- IDA-Plugin-Manager <https://github.com/tmr232/IDA-Plugin-Manager>
- Tarkus <https://github.com/tmr232/Tarkus>
- qscript <https://github.com/0xeb/ida-qscripts>

五、IDAPython 的不足

IDA 社区对 IDA 所提供的编程接口以及 IDAPython 并非毫无意见，批评主要在于 —— IDA 接口以及 IDAPython 所提供的 API 都过于 **low level**，既不易用，也不够 pythonic。在后面的具体学习中，大家也会更深刻的意识到这一点。

除了 IDA 对做好这件事不够上心这一种原因外，IDA 开发团队还给出了另一个理由 —— 他们不确定怎样组织 API 可以更 pythonic，所以他们只提供够用的低级 API，希望并鼓励用户去用一种更 pythonic 的方式处理它们。

在这样的思潮下，确实涌现出了一批更好用、强大、优雅的 API 封装，我们介绍其中流行的几个。

2015 年出现的 Sark <https://github.com/tmr232/Sark> 十分热门，它的口号是“让 IDAPython 更易用”，事实上它确实做到了。在 IDC 以及 IDAPython 中，各种粒度与层级的概念，以一个个单独的工具类，比如汇编、函数、基本块、数据等，同级的呈现在我们面前。这意味着如果你想研究和分析一块数据，其中包含函数、汇编、基本块等多个概念时，就需要从不同的柜

子里取出它们，调用其中的方法。而 Sark 遵从面向对象的理念包装了原生 API，围绕着函数等几个主要类，你可以优雅简单的通过调用对象的属性，解决相关的所有问题。

Sark 比 IDAPython 原有 API 更优雅易用，实现同样的功能，只需要原先 2/3 的代码量，而且它提供了清晰的 官方文档 <https://sark.readthedocs.io/en/latest/> ，因此许多插件基于 Sark 开发，我们后续也会更多介绍它。

2020 年出现的 bip <https://github.com/synacktiv/bip/> 是一个青出于蓝的工具，它做了更进一步的封装，并实现了一系列易用的工具方法。使用 BIP 做脚本开发的体验非常好，比 Sark 更好，除此之外，它也提供了很好的官方文档 <https://github.com/synacktiv/bip/> 。

除此之外还有 ida-minsc <https://github.com/arizvisa/ida-minsc> 、wilhelm <https://github.com/zerotypic/wilhelm> 等一系列致力于此道的工具，我们会在合适的场景对它们做介绍。

六、学习障碍

系列教程面向 IDA 脚本开发的新手，而非 IDA 使用的新手。如果读者感到学起来很费劲，欢迎私聊或群聊讨论您的问题。除此之外，我向您推荐学习 IDA 以及 IDAPython 的两本权威教材，它们都是很好的。

- 《 The Beginner's Guide to IDAPython》，目前最新是 6.0 。
- 《IDA 逆向权威指南》目前是第二版

<https://leeyuxun.github.io/IDA%E5%9F%BA%E7%A1%80%E5%8A%9F%E8%83%BD%E6%80%BB%E7%BB%93.html>