

Python 在 ASIC 中的应用——目录

最近写的 Python 的文章越来越多，从历史文章菜单找不太好找了，还是做个目录更方便些。并大概列了下后面的计划，会随实际情况有所调整。

如果各位童鞋有好的想法，欢迎在后面留言！



基础篇

Why Python

初始 Python 语言

写 Python 前的准备

我的第一个 Python 程序

Python 的数据类型（一）：介绍

Python 的数据类型（二）：数字

Python 的数据类型（三）：字符串

Python 的数据类型（四）：列表 List 与元组 Tuple

Python 的数据类型（五）：字典 Dict

Python 的条件与循环

Python 的函数（一）：基本概念

Python 的函数（二）：作用域

Python 的函数（三）：参数传递

Python 的函数（四）：递归函数与匿名函数

中级篇

Python 的模块

如何安装自己写的模块

Python 读写文件

Python 的正则

高级篇

Python 的类（一）：基础

Python 的类（二）：继承派生

Python 的类（三）：重载

Python 的类（四）：其它高级 Topic

Python 的多进程

Python 的异常处理

实战篇

常用的模块介绍（一）：sys、os、shutil

常用的模块介绍（二）：xml、excel、database

常用的模块介绍（三）：网络

常用的模块介绍（四）：GUI

常用的模块介绍（五）：大数据、人工智能相关的模块介绍

利用 Python 来实现更灵活的仿真脚本

利用 Python 来生成 UVM 寄存器模型

利用 Python 来跟踪 Bug

利用 Python 来分析门级网表

利用 Python 的模板来做通用 Flow

实例太多，各位想学哪部分可以后面留言哈



[更多教程](#)

《Perl 在 ASIC 中的应用》，里面有很多实例，欢迎讨论！

为什么选择 python

https://mp.weixin.qq.com/s?_biz=MzlyMjYxNzA4NQ==&mid=2247483897&idx=1&sn=e93c540587d4eacfe75e5226c859ea8b&chksm=e82b8d6bdf5c047ddf9340f50cea66cdf56b7921fb59af4e53701c81703c443256ea18ab9e4f&mpshare=1&sc

ene=1&srcid=0108T0Cn4RRQnjwBlBsaahGM&pass_ticket=nrVAwXRpsjSVVTPR5
HaEJjk%2FKy4I55IxtAP8PvZ5%2FRhIFR9jyMdd4KygBxSJCIR#rd

原创2018-01-08ExASICExASICExASIC

有个后端朋友问我，在芯片设计领域为什么选择 python，而不是 perl 或 tcl? 我才突然意识到，写了这么多天的《Python 在 ASIC 中的应用》系列文章，居然忘了对比介绍 Python、Perl、Tcl 的优缺点。因此我周末抽空写了这篇文章。

本文包括以下内容。共计 1900 多字，阅读时间大概 10 分钟。

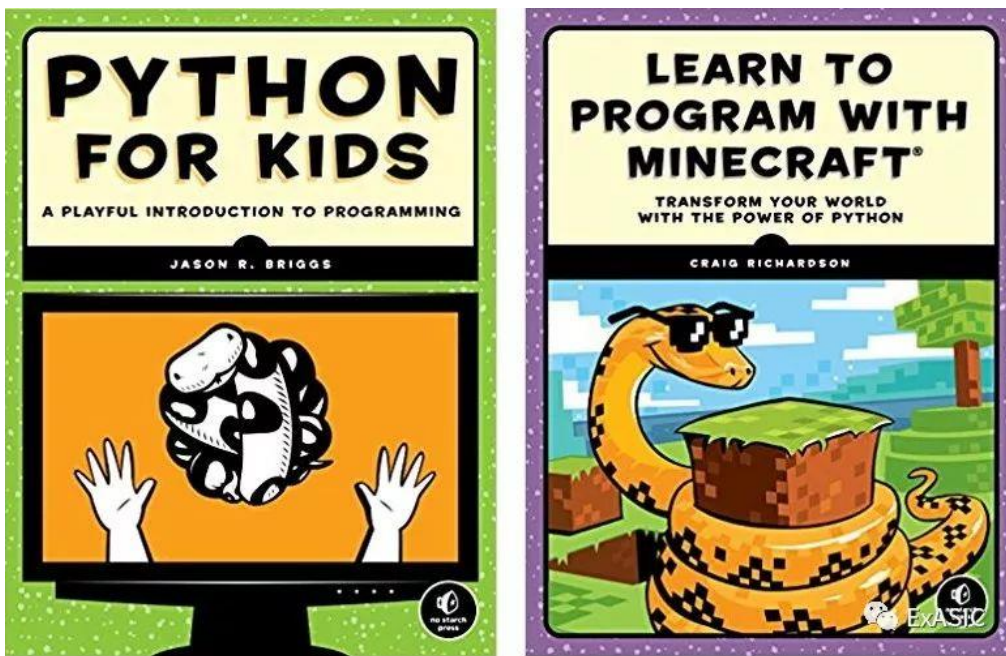
- 了解 Python 发展的大背景
- Perl 语言的优缺点
- Tcl/Tk 语言的优缺点
- Python 在 ASIC 领域的应用
- 总结

了解 Python 发展的大背景

我们都知道现在 python 很火。小到 python 在学校受到前所未有的重视（如，被编入小学教材，纳入浙江省高考范畴，成为全国计算机二级考试科目）。大到 Python 是人工智能 AI 的主要编程语言之一（如，google 的 TensorFlow，facebook 的 PyTorch）。

为什么火？就是因为 Python 简单易学、完善的面向对象（封装、继承、多态）编程。

简单易学体现在 Python 的语法格式上，强制利用缩进来规范代码风格，不能随意换行等。简单易学还体现在 Python 的设计思想“只用最好的方法做一件事”。正因为简单易学，所以小白容易入门，小学生有了 python 的入门课程，社会上有了 python 兴趣班，出现了《python for kids》、《learn to program with minecraft》这样的儿童书籍。



面向对象的特征使得 python 具有了像 C++ 那样的建模能力。在软件开发初期时，常常被用来快速建模，只需要把其中对性能有要求的功能模块改用 C++ 实现即可。

由于上面的优势，各行各业都开始使用 Python。Web 开发、游戏脚本、网络攻防、大数据分析、计算机视觉、机器人开发、人工智能 AI 等。连微软的 office 编程也在计划支持 python 语言。

Python 也有缺点。我们知道 Python 有两个版本，Python2 和 Python3。Python3 设计时没有做向下兼容，这使得 2 和 3 的语法有较大的不同，给初学者带来很大的困惑（关于版本的选择，我的建议是直接学习 Python3）。但换个角度看，正是这种破釜沉舟的精神使得 Python 保持了简洁。

Perl 语言的优缺点

Perl 语言是由 C、Sed/Awk、Shell、Basic、Pascal 等语言发展而来，因此语法类似 C 语言，相对容易学。Perl 作为一种粘合语言，功能强大、实用，而不是优美、简洁，就像它的标志骆驼（据说是 O'Reilly 出版社首先使用），“有点难看，气味不好，但干起

活来好使唤👍”。



正因为 Perl 结合了多种语言的优势，功能变得非常强大和灵活。perl 语言最初被设计用来就处理 log 等文本文件，因此内嵌了功能强大的正则表达式，这也是 perl 语言的灵魂所在。

perl 语言的强大离不开 CPAN 库。CPAN 也是广大开发者学习交流的平台。据官网介绍，从 1995 年至今，有 13,343 位作者提交了 194,785 个 perl 模块（这一数字每天都在增长，我验证过这一点，连续看了三天），并镜像在全球 256 个服务器上。

Perl 语言正是因为它的灵活多变，每个人写出来的代码风格可能完全不一样，甚至同一个人每次写出来代码也不尽相同。有时候把自己一两个月之前写的代码拿来看看，都不一定能一下看明白。不少初学者就被 \$ @ % \$_ @_ \$! \$' \$& \$' \$! \$| \$/ @\$ %\$... 这些特



殊符号给吓住了。因此，入门和精通都相对不容易，吓走了很多初学者。（可以



点击阅读我写的《Perl 在 ASIC 中的应用》系列文章哈)

Tcl/Tk 语言的优缺点

Tcl 语言语法规则简单，任何语句都看起来像一条 Linux 命令。因此 Tcl 语言简单易学，入门零门槛。

Tcl 语言，正如它的名字 Tool Command Language，常常 EDA 软件结合比较紧密，像前端的 vcs、modelsim、dc，后端的 icc 都支持 tcl 语言。因此，我们常常在用 EDA 时，不经意之间就学会了一些基本的 Tcl 语法。

Tcl 语言把一切都看作字符串，有丰富的字符串处理函数（比 Python 还要多出不少）。Tcl 语言有正则表达式，适合做复杂的文本处理，如自动分析 log 文件。另外 Tcl 语言常常用来做综合、PR、LEC、STA 的 flow。

由于 Tcl 原生支持 tk 库，因此被很多后端工程师用来做带图形界面的小工具，以辅助后端设计。事实上，现在 tk 库也被其他动态语言，如 Perl、Python 支持。

Tcl 类似 Linux 命令的语法也有缺点。每个命令的参数很多，不容易记住。Tcl 程序语言上不如 Perl 和 Python 灵活，写大程序时相对臃肿很多。

Tcl 语言本身不支持面向对象，需要利用 C++ 或 Java 来扩展。这也是一个很大的缺陷。

Python 在 ASIC 领域的应用

最后再回到 Python 语言上来。在 ASIC 领域，Python 的应用已经无处不在。我简单罗列了一下我所知道的应用：

1. 算法开发
2. Git database 创建和维护、自动生成 Makefile 等
3. 寄存器数据库管理、读写 Excel、XML、Json 等格式的配置文件
4. RTL 模板、部分 RTL 自动生成、顶层 RTL 自动连线等
5. 测试激励生成、验证脚本（前仿、后仿、回归测试）
6. 自动产生及配置 UVM 环境
7. 直接或间接维护前后端 Flow（包括利用模板实现后端通用 Flow）
8. 利用脚本修改数字网表、模拟 Spice 网表等
9. 前后端的报告分析、及可视化（图表分析）
10. 前后端 GUI 图形界面（支持 Tk, QT, wxPython 等库）
11. 辅助 FPGA 验证、测试
12. CP 测试、良率分析，及可视化
13. 辅助芯片应用、测试

14.项目管理、日报周报及绩效管理系统、BUG 跟踪管理等等。

因此，我们几乎可以说 Python 能做一切你想做的事情。（如果你知道 Python 在 ASIC



领域的其它应用，请在文章后面留言，一起来补充)

总结

Python 从语言本身到应用大环境都有绝对的优势，所以 ICer 们不要再犹豫，赶紧学起来用起来。

但是不是说我们只学 Python 就够了呢？不是的。我们实际做项目时，需要从方便、简洁、快速等角度，根据各语言的优缺点，择优而用，甚至多种语言混用。

初识 Python 语言

原创2017-08-14ExASICExASICExASIC

ExASIC

为什么选择 Python

由于 ASIC 验证的脚本规模越来越大，让脚本更易阅读、维护、扩展变得非常重要。

Python 语言的哲学是“优雅”、“明确”、“简单”。“用一种方法，最好是只有一种方法来做一件事”。而 Perl 语言做一件事有 N 种实现方法，不同人写出来的程序风格迥异，不利于 ASIC 验证领域所需要的水平和垂直复用，不利于新进验证工程师快速掌握验证脚本。

Python 正是由于这种思想使得不同人写出来的 python 程序看起来总是相似的。Python 不使用花俏的语法，而选择明确的没有歧义的语法。因此，Python 程序通常比 Perl 具备更好的可读性。

Python（与 C++ 一样）是面向对象的语言，支持继承、重载、派生、多继承，有益于增强源代码的复用性，能够支撑大规模的程序开发。

Python 能做些什么

在 ASIC 领域，Python 可以做快速算法原型开发、开发 Flow、验证脚本、报告分析、数据分析、UVM 验证模板、数据库管理、GUI 界面，以及各种辅助芯片研发的小工具，如项目管理、BUG 跟踪管理等等。

因此，我们几乎可以说 Python 能做一切你想做的事情。

如何学习 Python

跟着 [ExASIC](#) 的《Python 在 ASIC 中的应用》系列文章，从零开始学 Python。我们由浅入深、循序渐进，带领零基础的童鞋入门。

如果你已经对 Python 比较熟悉，这里有进阶书籍推荐

<https://www.zhihu.com/question/51261338>

写 Python 程序前的准备

原创2017-08-15ExASICExASICExASIC

上一次我们了解了 Python 是什么，今天我们来学习 Python 的安装、配置，以及学习 Python 运行脚本的过程，为写程序做准备。

如何安装 Python

学习 Python 的第一步当然是要安装 Python 软件了。

我们从官网 <https://www.python.org/downloads/> 下载安装程序。如下图，根据你的操作系统是 windows、Linux 还是 Mac 选择对应的版本。另外，我们看到有 Python 3.6.2 和 Python 2.7.13 两种可下载。

Download the latest version for Windows

[Download Python 3.6.2](#) [Download Python 2.7.13](#)

Wondering which version to use? [Here's more about the difference between Python 2 and 3.](#)

Looking for Python with a different OS? Python for [Windows](#), [Linux/UNIX](#), [Mac OS X](#), [Other](#)

Want to help test development versions of Python? [Pre-releases](#)  ExASIC

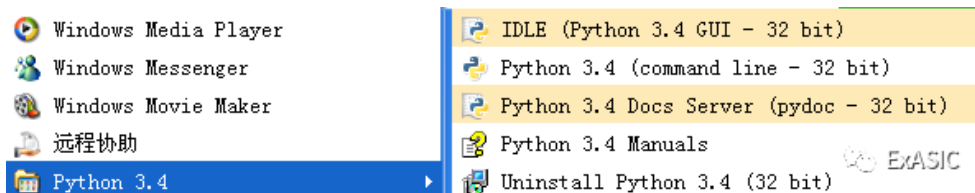
我们当然新版本 Python3.6.2。选择 Python3 的理由是新版本肯定有很大的改进，支持 Python2 不具备的特性。细节上两者的区别可参考官网的 <https://wiki.python.org/moin/Python2orPython3>。选择 Python2 的唯一情况是：你之前的项目是基于 Python2 的，需要维护老项目的脚本。

当然如果官网下载速度很慢，还是可以百度对应 Python 版本，然后你会看到很多网站都



可以下载。

Windows 平台安装 Python 很简单，就是双击一下安装文件。安装好后，开始菜单就有了 IDE 和 command line。如下图（由于 windows xp 最高只支持到 Python3.4，下图是 3.4 下的截图）。



Linux 平台的下载，我们选择 XZ compressed source tarball，原因是 XZ 压缩格式文件小很多。

- [Python 3.6.2 - 2017-07-17](#)
 - [Download XZ compressed source tarball](#)
 - [Download Gzipped source tarball](#) VC9 EtASiC

安装命令如下：

```
xz -d Python-3.6.2.tar.xz      #解压 xz 文件
tar -xf Python-3.6.2.tar      #解压 tar 文件
cd Python-3.6.2               #进入源码目录
./configure --prefix=/home/abc #配置，指定安装目录
make                           #编译
make install                   #安装
```

如何配置环境变量

Python 的环境变量设置比较简单，只需要把 Python 安装目录（即 python.exe 或 python 所在目录）加到 PATH 环境变量。Windows 平台，右击“我的电脑”->“属性”->“高级”->“环境变量”->“用户环境变量”，修改 path，在最前面加入 Python 的安装目录。如下图。



Linux 平台，根据 shell 类型，编辑对应的配置文件，把 python 安装路径增加到 PATH 变量。

csh, tcsh: 编辑/home/abc/.cshrc 或/home/abc/.tcshrc, 增加

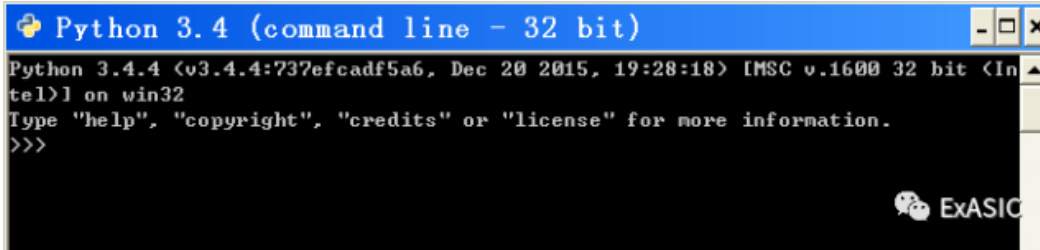
```
set path (/home/abc/bin $path)
```

bash: 编辑/home/abc/.bashrc, 增加

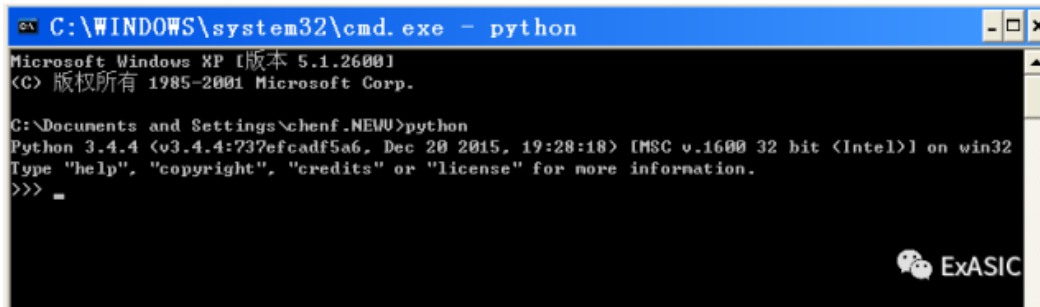
```
export PATH = /home/abc/bin:$PATH
```

打开 Python 程序，确认是否安装成功。

Windows 平台，点击开始菜单里的 Python 3.4 (command line - 32 bit)或者按 Win+R 运行 cmd.exe 再输入 python 命令。



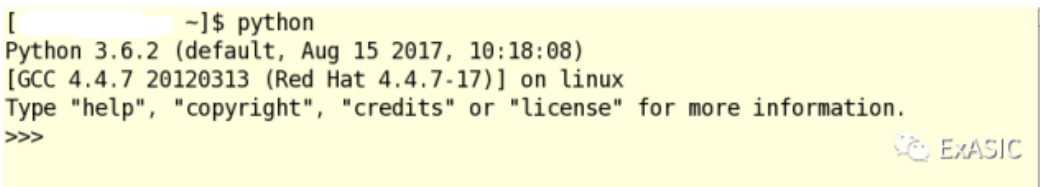
```
Python 3.4.4 (command line - 32 bit)
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 19:28:18) [MSC v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```



```
C:\WINDOWS\system32\cmd.exe - python
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\chenf.NEU>python
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 19:28:18) [MSC v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

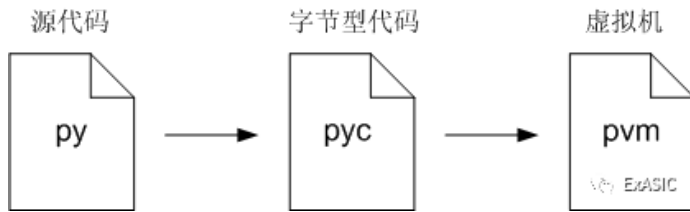
Linux 平台，打开 Terminal，输入 python 命令。



```
[ ~ ]$ python
Python 3.6.2 (default, Aug 15 2017, 10:18:08)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-17)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python 是如何运行脚本的

我们的 python 脚本，先被编译成 Byte Code，Byte Code 再被 Python 虚拟机解释执行。所以说 python 是解释型语言。如下图所示。



为了帮助理解这个过程，我们手动把 py 编译成 pyc。编译命令如下：

```
python -m py_compile helloworld.py
```

```
└─ helloworld.py
└─ __pycache__
    └─ helloworld.cpython-36.pyc
```

1 directory, 2 files ExASIC

经过编译之后，自动创建了一个临时目录 `__pycache__`，这个临时目录里产生了一个文件 `helloworld.cpython-36.pyc`。这个 pyc 文件就是 Byte Code。我们注意文件的命名，“原文件名.cpython-36.pyc”，36 是编译时使用的 python 版本号。python 在运行时，会根据源代码的修改时间来决定要不要重新编译，这样会大大加快编译速度。跟 Makefile 有异曲同工之处。

有些童鞋可能会问 Python 用了虚拟机的机制会不会速度很慢啊？

其实不用担心，因为一是 Python 虚拟机已经优化得很好了，二是因为我们 ASIC 领域的脚本也不会写得特别大。验证仿真的速度主要还是和芯片的规模和仿真模型的抽象级别有关。

准备工作已经完成，下一次，我们学习写第一个 python 程序。

我的第一个 Python 程序

原创2017-08-16ExASICExASICExASIC

今天我们来学习写第一个 Python 程序。

目标: 从命令行传入一个参数, 比如人名, 然后打印出欢迎信息。

我们大学期间都学过 C 语言或 C++。那先来想想如果用 C++该如何实现?

在 C++中, 命令行传参数是用 `char * argv[]`, 即字符串的指针的数组。打印是用 `cout` 输入输出流。下面是 C++实现的代码。

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(int argc, char *argv[]){
6     cout << "hello " << argv[1] << "!" << endl;
7
8     return 0;
9 }
10
```

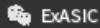


编译命令 `g++ -o hello hello.cpp`, 编译后生成可执行文件 `hello`。

运行命令 `./hello bill`, 终端里输出 `hello bill!`。

所有的编程语言都只是思想或逻辑的描述, 所以不同语言之间肯定有相同之处。我们用对比的方法来学习第一个 Python 程序。下面是 Python 的实现代码。

```
1 #!/usr/bin/env python
2
3 import sys
4
5 def main(argv) :
6     print("hello " + argv[1] + "!")
7
8 if __name__ == '__main__' :
9     main(sys.argv)
10
```



第 1 行：#!/usr/bin/env python 用来指定执行本脚本所用的软件。是这 Linux、Unix 脚本的共同特性，与 Python 无关，Shell、Perl 等其它脚本语言都有。操作系统先读取脚本的第一行，来决定用哪个软件来执行脚本，不依赖文件的后缀名。因此 Linux 中的脚本可以不带用后缀名。通常好的编程习惯是加上特定的后缀名，如 Python 加 py，Perl 加 pl。

第 3 行：import sys 类似 C++的#include 语句，用来包含（准确点叫导入）库文件。sys 是与系统相关的库，包括 argv、stdin、stdout 等。我们这个例子里用到了 argv，所以需要导入 sys 库。

第 5 行：def main(argv) : 定义一个函数。def 是定义函数的关键字，main 是函数名，圆括号里定义了函数参数。这个例子里定义了一个名叫 main 的函数，函数有一个参数 argv。

我们注意到第 5 行的最后有一个冒号 “: ”，这是 Python 语言所特有的标记，相当于 C++的大括号{}，表明一个层次关系。

第 6 行：print("hello " + argv[1] + "!")是调用 python 语言内置的函数 print()来向终端（屏幕）打印信息。括号里的 “+” 的作用是字符串的拼接，其中 argv[1]是数组 argv 的第 1 个元素（从 0 开始计数）。

第 8 行：if __name__ == '__main__' : 意思是判断当前模块的名字是不是 '__main__'，如果是则执行后面的语句，如果不是则忽略后面的语句。一般情况下 __name__ 的值总是等于 '__main__'。只有一种例外的情况，当我们写的脚本被当作库文件导入到其它脚本时，

`__name__` 等于我们的脚本文件名。例如，我们这个脚本名叫 `hello.py`，被导入到其它脚本时，`__name__` 等于 `hello`。

当然现在觉得复杂不好理解没有关系，照着写就行了。等以后学习写库文件的时候再深入学习。

第 9 行：`main(sys.argv)`调用了上面自定义的 `main()`函数，并把命令行参数的字符串数组 `sys.argv` 传入 `main()`函数。

运行 Python 脚本

方法一：

```
chmod +x hello.py
```

```
./hello.py bill
```

方法二：

```
python hello.py bill
```

结果输出：

```
hello bill!
```

我们来总结一下 Python 脚本的特点：

1. 第 1 行用 `env python` 来指定运行脚本的命令。比直接指定 `/usr/bin/python` 要更通用，因为用 `env python` 时写的代码与 `python` 安装路径、版本无关。
2. 定义函数用 `def` 关键字

3. 需要表明层次关系时需要用冒号 “:”，如 def、if 的行尾加冒号。即使 def 或 if 内部层次只有一个语句。并且同一层次的语句必须保持相同的缩进。if 语句块后续的语句必须与 if 保持相同的缩进。例如：

```
if a == 0:
```

```
    b = 1
```

```
    c = 2
```

```
d = 3
```

b = 1 和 c = 2 属于 if 内部层次，只有当 if 条件成立时才执行。而 d = 3 与 if 并列，if 语句块过后，始总会执行。

4. 利用 `if __name__ == '__main__':` 来实现类似 C++ 的 main 函数的作用，让代码看起来更整洁，更利于被当作库文件来复用。
5. 与 C++ 的不同之处一：每个语句末尾不需要加分号 “;”
6. 与 C++ 的不同之处二：变量的类型不需要提前定义，给它赋什么类型的值就是什么类型。
7. 与 C++ 的不同之处三：不需要编译，可直接执行。

"我的第一个 Python 程序"介绍了一个较规范的 Python 脚本的例子，并且跟 C++ 语言作了对比。希望大家对文章末尾的总结多看几遍，慢慢体会。

练习：

亲自写一遍这个 hello.py 脚本，并在电脑上执行一下。

个人下脚本执行过程:

1) 第一次, 第 8 行的 if 语句, 由于呈上面的 print 惯性, 自动缩进和 print 一致, 如下面所示, 所以, 执行时, 没有显示任何结果。

```
#!/tool/pandora64/hdk-4.8.1/21/bin/python
```

```
import sys
```

```
def main(argv):
```

```
    print("hello " + argv[1] + "!")
```

```
    if _name_ == '_main_':
```

```
        main(sys.argv)
```

2) 第 2 次, 修改第 8 行的 if 语句, 回到行首, 和 def 并列, 此时, 出现下面的结果:

Traceback (most recent call last):

```
File "hello.py", line 8, in <module>
```

```
    if _name_ == '_main_':
```

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

经过多次试验, 猜测可能是 name 和 main 前后应该不是一个下划线, 而是两个下划线, 于是, 修改为:

```
#!/tool/pandora64/hdk-4.8.1/21/bin/python
```

```
import sys
```

```
def main(argv):
```

```
print("hello " + argv[1] + "!")
```

```
if __name__ == '__main__':
```

```
    main(sys.argv)
```

此时，正确打印出如下信息：

```
cybhen-nveu0784:/home/junq/python[ 94 ] --> python hello.py bill
```

```
hello bill!
```

所以，有两点经验教训：

1) 注意缩进，尤其是有时候自动缩进的，避免犯错误

2) 注意 `__name__` 前后都是双下划线。

python 的数据类型

原创2017-08-17ExASICExASICExASIC

谢谢合位童鞋的热情捧场，经过几天的学习，大家撩起了学习 Python 的热潮。今天来学习 Python 的数据类型（或者叫对象的类型）。

Python 是面向对象的语言。在 Python 里，数据总是以对象的形式存在，或者 Python 内建的对象，或者是我们创建的类。

对象是什么？

对象就是一片内存。在这片内存区域里，存放着变量的值，及与其相关的一些操作方法。Python 里的一切都是对象，甚至一个数字，如 99，也是一个对象。数字 99 支持加、减等操作。

我们先来看看 Python 内建对象的基本类型有哪些？

对象的类型	例子
数字	123, 3.14, 0b0101, 0x5
字符串	'Bob', "Bob's"
列表	[1, 'one', 2.5]
字典	{'apple': 'red', 'orange': 'yellow'}
元组	("Jiangsu", "Shanghai", "Beijing"), {"110", "112", "119"} ERASIC

数字与字符串比较好理解。

数字就是指

- 整数
- 实数
- 复数

可以用不同进制来表示，如 **0b0101** 表示二进制的 5，**0o5** 表示八进制的 5，**0x5** 表示十六进制的 5。

为什么说数字也是对象？

以整数为例，执行 `a = 123` 时，我们来看看发生了什么事情：

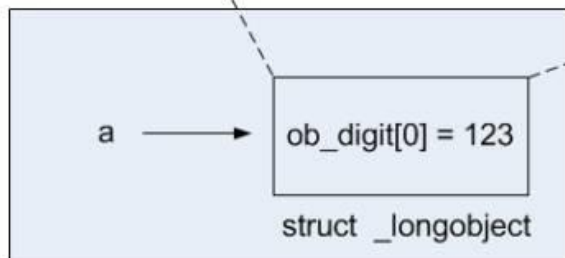
1. 在内存中创建一个整数的对象
2. 初始化这个对象的成员变量的值为 123
3. a 指向刚创建的对象

我们打开 python3.6.2 的源码，Include/longintrepr.h，找到整数结构体_longobject 的定义如下：

```
struct _longobject {
    PyObject_VAR_HEAD
    digit ob_digit[1];
};

PyAPI_FUNC(PyLongObject *) _PyLong_New(Py_ssize_t);
/* Return a copy of src. */
PyAPI_FUNC(PyObject *) _PyLong_Copy(PyLongObject *src);
```

a = 123



可以看出其实 Python 是基于 C++ 的，由 C++ 定义的结构体对整数类型进行了封装。至此，我们应该理解了为什么说 Python 的数值也是对象了吧。

字符串是指用

- 单引号' ... '
- 双引号" ... "
- 三引号""" ... """和''' ... '''

括起来的字符序列。

如 'Bob' , 'apple' , "yellow" , "'I love you!'".

引号中的引号

单引号里可以包括双引号，双引号里也可以包括单引号。三引号里可以包括单引号和双引号，反之不行。例如，'ab"c', "Bob's", "'ab"cBob's'". 但'ab'''c'则是非法的。

引号中的转义

单引号、双引号、三引号中都可以转义，且效果相同。如'ab\nc', "ab\nc", ""ab\nc"" 三者是等价的。

三引号的特殊作用

三引号中可以加换行，即可以表示多行字符串。在用 print()打印时按原样输出，所以三引号通常用来输出大段的文本。如：

```
strHelp = '''
```

```
This is the help doc of simulation tools.
```

```
-v To specify verilog library file
```

```
-y To speicy verilog library search path
```

...

```
print(strHelp)
```

总结一下字符串与 Perl 语言的对比:

	Python	Perl
单引号	可以包含双引号 可以转义	可以包含双引号 不可以转义, 按原样输出
双引号	可以包含单引号 可以转义	可以包含单引号 可以转义
三引号	可以包含单引号和双引号 可以转义 可以换行	没有三引号, 可以用 Here doc 实现 Here doc 内部可以转义, 可以换行

下次我们再慢慢介绍列表、字典、元组等其它对象类型。

练习题:

对照 Python3 的源代码, 理解为什么数字是对象。

编几个小程序, 验证一下单引号、双引号、三引号的差别。

个人编写的小程序:

```
#!/tool/pandora64/hdk-4.8.1/21/bin/python
```

```
import sys
```

```
def main(argv):  
    dauther_name = "" 3)hello,  
        My dauther's name is "" + argv[3]+ "!"  
    print("1)hello, my name is " + argv[1] + "!"  
    print("2)hello, my wife's name is " + argv[2] + "!"  
    print(dauther_name)  
if __name__ == '__main__':  
    main(sys.argv)
```

执行结果为:

```
cybhen-nveu0784:/home/junq/python[ 101 ] --> python data_type.py QinJunyan  
WuXianghui Sophia
```

```
1)hello, my name is QinJunyan!
```

```
2)hello, my wife's name is WuXianghui!
```

```
3)hello,
```

```
        My dauther's name is Sophia!
```

python 的数据类型 (二) : 数字

原创2017-08-22ExASICExASICExASIC

上一次我们讲了 python 数据类型的“数字”和“字符串”，了解了数字和字符串的定义。今天我们来深入学习数字的基本操作方法。

数字的基本操作是加、减、乘、除四则运算。例如下面的脚本：

```
#!/usr/bin/env python
```

```
a = 10
```

```
b = 2
```

```
c = a + b
```

```
d = a - b
```

```
e = a * b
```

```
f = a / b
```

```
print(c)
```

```
print(d)
```

```
print(e)
```

```
print(f)
```

结果输出：

```
12
```

```
8
```

```
20
```

5.0

从这个例子我们可以看出，四则运算跟我们想像的差不多，很接近 C++ 语言。但除法有点特殊，结果是 5.0 而不是 5。这是因为 python 在进行除法运算时，默认会作带小数的除法运算，即使被除数和除数都是整数。

那如果一定需要整除该怎么办呢？python 还提供了专门的整除运算符//，例如：

```
a = 10 // 2
```

```
b = 10 // 3
```

```
c = 10 // 3.0
```

```
print(a)
```

```
print(b)
```

```
print(c)
```

结果输出

```
5
```

```
3
```

```
3.0
```

从这个例子可以看出，//运算符相当于取/运算符结果的整数部分，直接把小数部分给舍掉。但是当被除数或除数带小数时，//的结果也带小数部分，这个时候与/相同。

当然我们还有一种方法，利用 int() 做类型转换，把小数转成整数。如：

```
a = int(10/3)
```

```
print(a)
```

理解数字四则运算符的本质

我们这里讲的四则运算符+ - * /本质上**运算符重载**。怎么讲？我们知道数字是对象，两个对象进行加减乘除必然要通过运算符重载来实现。

我们来看两个例子。例子一：

```
a = 10
```

```
b = 2
```

```
c = a + b
```

```
print(id(a))
```

```
print(id(b))
```

```
print(id(c))
```

结果输出：

```
9323424
```

```
9323168
```

```
9323488
```

这个例子中，我们分别打印了 a, b, c 三个对象的 ID，三个 ID 均不相同。

例子二:

```
a = 10
print(id(a))
a = a + 2
print(id(a))
```

结果输出:

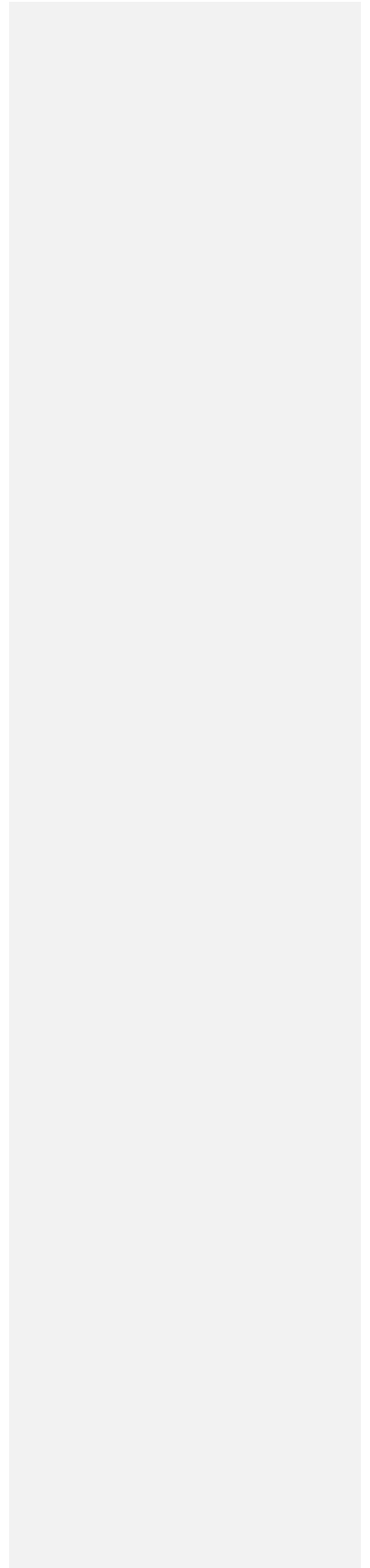
```
9323424
9323488
```

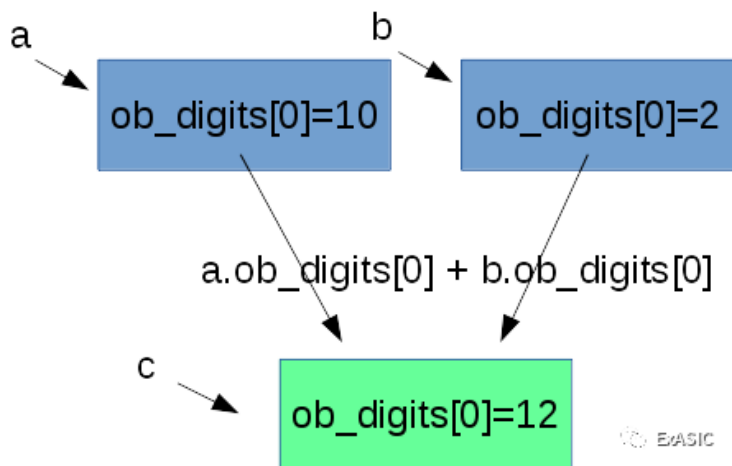
这个例子，我们在加法前后打印了两次对象 a 的 ID，两次 ID 并不相同。

这是因为 python 语言里的数字对象一旦初始化，就不能更改了，类似 C++ 语言的 const 常量。当执行 $c = a + b$ 时，大致经过下面这几步：

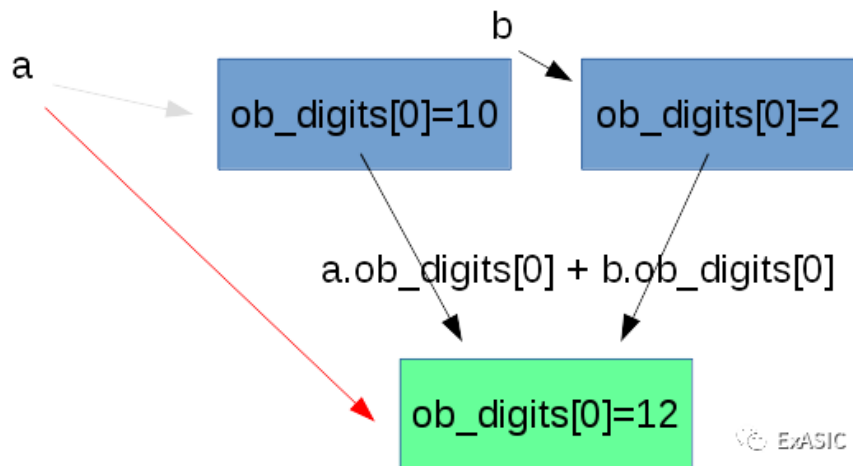
1. 取出对象 a 的值，取出对象 b 的值
2. 两者相加，并赋值给一个临时对象
3. 把 c 指向这个新的临时对象

上面的例子一可用下图来表示：





上面的例子二可用下图来表示:



由于数字对象 a 是不可修改的, 当 $a = a + b$ 时, 最终结果 a 其实是指向这个新创建的临时对象, 而不是修改 a 指向的对象本身。

总结:

我们再次强调 python 是面向对象的编程语言，所以在基础语法学习阶段我们会一直传递对象的思想。只有理解了一切皆对象，才能更熟练地使用 python 来编程。这一点与学习 perl 的方法完全不同，我们需要尽快调整过来。

下一次，我们学习数据类型"字符串"的操作方法，这是非常重要的部分哦！

python 的数据类型（三）：字符串

https://mp.weixin.qq.com/s?__biz=MzlyMjYxNzA4NQ==&mid=2247483822&idx=1&sn=4efffa6cca10ea0f64255593fa8541ca&chksm=e82b8d3cdf5c042acc3b2a7995698472c58133cd16b6297f6f80c4daab1d9e5745a77aef756c&mpshare=1&scene=1&srcid=101338mXipxT45Z1Cbpz8NOg&pass_ticket=quu0pEOegr0TmMjeXcQDfEspQizm8fgH4sq5lirgEjRu16t8RbaFkJjqZ%2BPv%2Bqa#rd

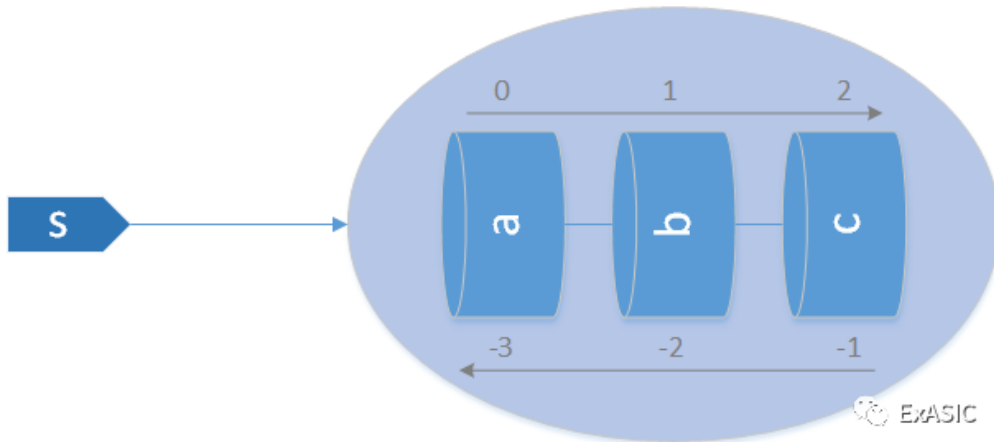
原创2017-09-25ExASICExASICExASIC

各位童鞋很抱歉，由于这段时间比较忙，好久没更新了。今天终于有空来继续跟各位童鞋一起学习 python。

之前讲到，字符串的定义就是把一些字符序列用引号引起来。例如：

```
s = 'abc'
```

上面这行代码的执行过程：先创建一个字符串对象，并初始化里面字符串的值为'a', 'b', 'c'的序列，再把指针 s 指向这个对象。如下图所示：



字符串的序列操作

我们知道字符串内部是一个字符序列。作为序列，我们可以用下标来索引。例如：

```
c = s[0] # c = 'a'
```

```
c = s[1] # c = 'b'
```

还可以从字符序列的末尾开始索引，-1 代表最后一个字符，-2 代表倒数第二个字符，...
例如：

```
c = s[-1] # c = 'c'
```

```
c = s[-2] # c = 'b'
```

另外字符串还支持切片 (slice) 操作，取出给定两个下标之间的字符序列（包括开始下标的字符，但不包括结束下标的字符。用数学区间的[a:b)来描述是不是更清楚点呢）。

例如：

```
c = s[0:2] # c = 'ab'
```

```
c = s[1:2] # c = 'b'
```

```
c = s[1:-1] # c = 'b'
```

这一点很像 verilog 的位宽操作, 例如: `reg s[7:0] = 8'h5a`, 那我们可以 `c = s[3:0]` 来取出 3 到 0 位的 4'ha。只是 verilog 是闭区间的。

在切片时, 当开始下标是 0, 冒号前的下标可以省略; 结束下标指向最后一个元素, 冒号后的下标也可省略。所以上面的切片可以简化成:

```
c = s[:2] # c = 'ab'
```

```
c = s[1:] # c = 'bc'
```

当开始下标和结束下标都省略时, `s[:]` 表示整个字符序列 'abc'。

注意理解 `s[1:-1]` 和 `s[1:]` 的区别, 前者不包括最后一个字符, 而后者包括。

作为字符序列, 还支持拼接和重复操作。例如:

```
s1 = 'abc'
```

```
s2 = s1 + 'def' # s2 = 'abcdef'
```

```
s3 = s1 * 3 # s3 = 'abcabcabc'
```

另外我们要注意字符序列的只读性。我们不可以对字符赋值, `s[0] = 'd'` 是非法的。但我们可以迂回:

```
s = 'abc'
```

```
s = 'd' + s[1:]
```

但要注意，第二句并不是简单的修改 *s* 对象的元素值，而是用拼接产生了一个全新对象，并让 *s* 指向新的对象。

字符串类型的操作

除了序列操作外，字符串本身作为一种类型，自带了很多操作函数（类的方法）。下表列出了一些常用的操作：

查找子字符串	index, find
替换	replace
拆分	split
转大小写	upper, lower
截头尾	strip, lstrip, rstrip
开始结束	startswith, endswith
格式化	format

Python BASIC

index, find

```
position = index(substr, begin=0, end=len(string))
```

```
position = find(substr, begin=0, end=len(string))
```

描述：index 和 find 函数的作用相同，都是查找子字符串。可以指定开始和结束索引，在一个范围内查找。

返回值：子字符串的起始索引值。index 和 find 的区别是，当没有找到子字符串时，index 报错，而 find 返回-1。

例如：

```
s = 'abcdefdef'
```

```
p1 = s.find('de') # p1 = 3
```

```
p2 = s.index('de', 5) # p2 = 6
```

replace

```
str_new = replace(substr_old, substr_new[, max])
```

描述：替换函数，如其名，查找子字符串 substr_old，替换成 substr_new。第三个参数是可选的，指定替换的最大次数，默认是全部替换。

返回值：返回替换后的新字符串。

例如：

```
s = 'abcdefdef'
```

```
s1 = s.replace('de', 'gh') # s1 = 'abchgfhgf'
```

```
s2 = s.replace('de', 'gh', 1) # s1 = 'abchgfedf'
```

split

```
list = split(str=' ', num)
```

描述：split 函数用分隔字符 str 把字符串拆分成若干个子字符串。num 指定拆分多少次，若没有指定次数，则为全部拆分。

返回值：拆分后的子字符串列表（[下一次我们将要学习列表](#)）。

例如：

```
s = 'I am learning python'
```

```
list1 = s.split(' ') # list1 = ['I', 'am', 'learning', 'python']
```

```
list1 = s.split(' ', 2) # list1 = ['I', 'am', 'learning python']
```

upper, lower

```
str_new = upper()
```

```
str_new = lower()
```

描述: 把字符串转成大写或小写。

返回值: 大小写转换后的新字符串。

例如:

```
s = 'abc'
```

```
s1 = s.upper() # s1 = 'ABC'
```

```
s2 = s1.lower() # s2 = 'abc'
```

strip, lstrip, rstrip

```
str_new = strip(char='')
```

```
str_new = rstrip(char='')
```

```
str_new = lstrip(char='')
```

描述: strip 函数用来去除头或尾部的指定字符, 默认是去掉空格。

返回值：返回处理后的新字符串。

例如：

```
s = ' abc\n'
```

```
s1 = s.lstrip() # s1 = 'abc\n'
```

```
s2 = s1.rstrip('\n') # s2 = 'abc'
```

startswith, endswith

```
boolean = startswith(str, begin=0, end=len(string))
```

```
boolean = endswith(str, begin=0, end=len(string))
```

描述：检查字符串是否以 str 开头或结尾，可以在指定范围内检查。

返回值：如果检查到，返回 True，否则返回 False。

例如：

```
s = 'clk_a'
```

```
b1 = s.startswith('clk') # b1 = True
```

```
s = 'rst_n'
```

```
b2 = s.endswith('_n') # s2 = True
```

format

```
str_new = '{}{}...'.format(arg1, arg2, ...)
```

描述: format 用来把其它数字、字符串、甚至对象等格式化成字符串。大括号{}用来指定名称、位置、数字的格式等。

返回值: 格式化后的新字符串。

例如:

```
s = 'I am learning {lang}'.format(lang='python') # s = 'I am learning python'
```

```
s = '{0} {1} {0}'.format('face', 'to') # s = 'face to face'
```

```
s = '{} {} {}'.format('I', 'love', 'python') # s = 'I love python'
```

第一种, 按名称替换。


第二种, 按位置替换。

第三种, 默认按位置替换, 也是最常见的替换方式。

是不是有点像 verilog 的模块例化? 可以按名称, 也可以按位置。

format 数字格式化

数字格式化成字符串的规则如下表:

<code>{:b}, {:o}, {:d}, {:x}</code>	二、八、十、十六进制
<code>{:#x}, {:#X}</code>	0xab, 0XAB
<code>{:.nf}, {:+.nf}</code>	小数的格式化, n 为小数点保留位数 +为带符号位
<code>{:x>nd}, {:x<nd}</code>	>宽度不足 n 时, 左边补 x <宽度不足 n 时, 右边补 x
<code>{:,}, {:.n%}, {:.ne}</code>	逗号, 百分数, 科学计数法
<code>{:>nd}, {:<nd}, {:^nd}</code>	右对齐, 左对齐, 居中对齐 n 为宽度
<code>{(,)}</code>	表示字符 '(' 和 ')' 时, 需要转义  BY-NC-SA

例如：

```
s = "8'h{:0>2x} ".format(15) # s = "8'h0f"
```

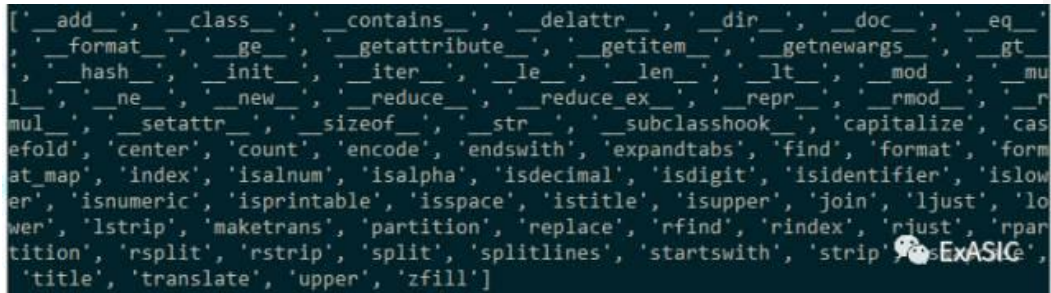
```
s = '{:.2%}'.format(3 / 9) # s = "33.33%"
```

可能有童鞋要问了“字符串内置这么多函数，一下也记不住啊？”所以下面内容非常重要（敲黑板）。

1. 怎么看 string 还内置其它什么函数？

```
s = 'abc'  
  
print(dir(s))
```

`dir()`是一个内置函数，能够查看类的所有属性和方法。结果如下：



```
['_add_', '_class_', '_contains_', '_delattr_', '_dir_', '_doc_', '_eq_',  
'_format_', '_ge_', '_getattr_', '_getitem_', '_getnewargs_', '_gt_',  
'_hash_', '_init_', '_iter_', '_le_', '_len_', '_lt_', '_mod_', '_mul_',  
'_ne_', '_new_', '_reduce_', '_reduce_ex_', '_repr_', '_rmod_', '_rmul_',  
'_setattr_', '_sizeof_', '_str_', '_subclasshook_', 'capitalize', 'cas  
efold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'form  
at_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islow  
er', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lo  
wer', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpar  
tition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'title',  
'translate', 'upper', 'zfill']
```

2. 那怎么查看具体函数的使用方法呢？

python 已经考虑到这个问题了，不需要百度、不需要查看源代码，只需要调用 `help()` 函数。例如：

```
print(help(s.find))
```

将打印出下面的内容：


```
Help on built-in function find:

find(...) method of builtins.str instance
  S.find(sub[, start[, end]]) -> int

  Return the lowest index in S where substring sub is found,
  such that sub is contained within S[start:end]. Optional
  arguments start and end are interpreted as in slice notation.

  Return -1 on failure.

None
```



有没有被 python 的贴心功能感动呢？

下一次我们将学习 python 的列表。

批注 [QJ1]: 2/8 看到此处

python 的数据类型（四）：列表 List 和元组 Tuple

Original2017-10-31ExASICExASICExASIC

各位童鞋，上一次的字符串都学会了吗？如果还有疑问欢迎在文章后面留言，或者直接在 ExASIC 公众号里留言讨论。今天与各位分享 python 的另外两种数据类型：列表（Lists）和元组（Tuples）。

列表是什么？

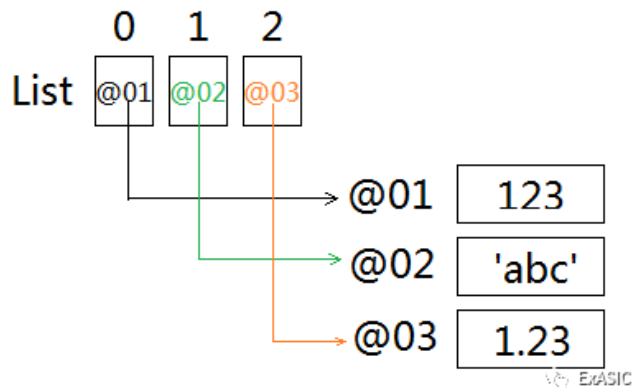
列表是一组对象的有序集合。

与 C 语言的数组有什么区别呢？

区别：

- C 语言数组的成员类型相同，而 Python 列表的成员类型可以不同（注）。
- C 语言数组的大小是固定的，而 Python 列表的大小可扩大、可缩小。

注：从语法上准确的讲，Python 的列表里保存的是指向各个成员数据对象指针的数组，并不是链表的形式。因此，根据索引从列表里取一个成员的速度与 C 语言的数组一样快。



例如：

```
L = [123, 'abc', 1.23, ['dev', 'mgr']]
```

这个列表有四个成员，分别是整数、字符串、小数、子列表。列表 L 里面保存的是指向这四个数据的指针（地址），并不是四个数据本身。

怎么定义一个列表？

用中括号[]把一组数据括起来，并且这些数据之间用逗号隔开。例如：

```
L = ['module_a', 'module_b', 'module_c']
```

```
L = ['wire1', 'reg2', 'logic3']
```

```
L = [] 定义一个空列表
```

列表有哪些操作方法？

用 len() 计算列表的长度，例如：

```
L = [1, 2, 3]
```

```
len(L)  # = 3
```

列表作为一种序列，与字符串一样，支持序列的基本操作，如+（拼接）、*（重复）、index 索引、slice 切片、。例如：

```
L1 = [1, 2, 3]
```

```
L2 = [4, 5, 6]
```

```
L3 = L1 + L2  #L3 = [1, 2, 3, 4, 5, 6]
```

```
L4 = L1 * 2   #L4 = [1, 2, 3, 1, 2, 3]
```

```
L1[0]        # = 1
```

```
L1[0:2]      # = [1, 2]，与字符串切片一样，是前开后闭型的[a, b)
```

列表支持原位修改。与字符串是只读的不一样，列表是可变的。例如：

```
L1 = [1, 2, 3]
```

```
L1[0] = 4     #L1 = [4, 2, 3]
```

```
L1[0:2] = [5, 6] #L1 = [5, 6, 3]
```

```
L1[:0] = [1, 2, 3] #L1 = [1, 2, 3, 5, 6, 3]，在开头插入
```

```
L1[len(L1):] = [7, 8, 9] #L1 = [1, 2, 3, 5, 6, 3, 7, 8, 9]，在末尾插入
```

列表作为对象本身的操作方法有 `append`、`extend`、`insert`、`remove`、`pop`、`sort`、`reverse`、`count`、`index`、`clear`。例如：

```
L1 = [1, 2, 3]
```

```
L1.append(4)      #L1 = [1, 2, 3, 4]
L1.remove(4)     #L1 = [1, 2, 3]
L1.extend([4, 5, 6]) #L1 = [1, 2, 3, 4, 5, 6]
L1.pop(0)       #L1 = [2, 3, 4, 5, 6]
L1.pop()        #L1 = [2, 3, 4, 5]
L1.insert(0, 1) #L1 = [1, 2, 3, 4, 5]
L1.reverse()    #L1 = [5, 4, 3, 2, 1]
L1.sort()       #L1 = [1, 2, 3, 4, 5]
L1.count(2)     #= 1
L1.index(3)     #= 2
L1.clear()      #L1 = []
```

注：列表没有 `push` 函数，不要想当然哦。不信？你可以用 `print(dir(L1))` 查查列表有哪些内置函数。

下面介绍一个特别的数据类型——**元组 (Tuple)**。元组可以理解成只读的列表。不能进行上面几乎所有操作，除了 `index` 和 `count`。只读属性有点像字符串。正因为定义后不能修改，元组是定长的。

元组的定义用圆括号()而不是方括号[]。例如：

```
T1 = ('192.168.0.1', 9002)
```

```
T2 = ('eco_module_a', 'eco_module_b', 'eco_module_c')
```

如这些例子，元组可用于防止误修改的情况，比如作为函数的输入参数，并不希望被函数内部误修改到。

例如：我们需要读取 tc 目录下的 test case 列表，并转化成元组。

```
#!/usr/bin/env python
```

```
import os
```

```
tc = os.listdir('tc')
```

```
print(type(tc))
```

```
tc_ro = tuple(tc)
```

```
print(type(tc_ro))
```

```
print(tc_ro)
```

运行结果如下：

```
[chenf@localhost python]$ ls tc
tc_func_a.svh tc_func_b.svh tc_func_c.svh
[chenf@localhost python]$ ./test.py
<type 'list'>
['tc_func_c.svh', 'tc_func_a.svh', 'tc_func_b.svh']
<type 'tuple'>
('tc_func_c.svh', 'tc_func_a.svh', 'tc_func_b.svh')
```

下次我们将介绍 python 的另一个重要的数据类型“字典”。在 TCL 语言里也有字典，在 C 和 Perl 里叫哈希，在 SystemVerilog 里叫关联数组，我们会介绍他们有哪些异同点。

python 的数据类型（五）：字典 Dict

Original2017-11-02ExASICExASICExASIC

今天开始介绍 python 的另一种重要的数据类型——字典（Dictionary）。顾名思义，就是像字典一样，根据“单词”查找“解释”的一种数据结构。在 python 里，“单词”就是索引，叫键（Key），“解释”就是值（Value）。

字典是无序的（不像列表，不支持序列操作，如下标索引 index、切片 slice 等）不能用位置 0,1,2, ...来索引，只能通过键名来索引。这一特点，非常适合给物品的属性建模，在数学上适合给稀疏数据结构建模。

字典在本质上，是建立了一个 hash 表，存放着键和指向数据的指针（地址）。python 优化了 hash 查找算法，使得查找键的速度非常快。

如何定义一个字典？

例如，我们描述一个 STDCCELL，

name : BUF

area : 1um2

delay : 1ns

那么，我们用 python 的字典就可以表示成，

```
stdcell = {'name':'BUF', 'area':'1um2', 'delay':'1ns'}
```

这就是定义字典的一个例子，语法上，把冒号:分隔的键值对（key:value pairs）用逗号分开，再用大括号{}括在一起。

第二种定义方法，调用字典类的构造函数 dict(**kwargs)。如，

```
stdcell = dict(name='BUF', area='1um2', delay='1ns')
```

这种方法就像函数传参数一样。

在大部分情况下，在字典定义的时刻并不知道所有的数据，所以最常用的方法还是像下面这样：

```
stdcell = {}
```

```
stdcell['buf0'] = '0.2um2' #注意，这里是方括号[]
```

```
stdcell['buf1'] = '0.6um2'
```

随着读入外部文件的一行行的解析，逐渐的构建字典。

字典有哪些操作？

最基本的操作就是根据索引（用方括号）取值，如

```
stdcell['buf0']
```

还有一种根据索引取值的方法，用 `get()` 函数，如

```
stdcell.get('buf0')
```

注：`get()` 函数，当键不存在时，返回空 `None`。如果希望当键不存在时，返回一个默认值，可以这样用：`stdcell.get('buf2', '0um2')`，这有时候很有用。

用 `len()` 计算字典的元素个数，如

```
len(stdcell)
```

修改元素的值，如


```
stdcell['buf0'] = '0.3um2'
```

```
#{'buf0':'0.3um2', 'buf1':'0.6um2'}
```

另一种方法是用 `update()` 函数，如

```
stdcell_new = {'buf0':'0.4um2', 'buf2':'0.8um2'}
```

```
stdcell.update(stdcell_new)
```

```
#{'buf0':'0.4um2', 'buf1':'0.6um2', 'buf2':'0.8um2'}
```

这种方法可以一次更新多个值，甚至增加新的键值对。

删除元素用 `pop()`，如，

```
stdcell.pop('buf0') #{'buf1':'0.6um2', 'buf2':'0.8um2'}
```

还有清空函数 `clear()`，如，

```
stdcell.clear() #{} 
```

分别提取字典的键和值的列表：`keys()`、`values()`

```
stdcell.keys()  #['buf0', 'buf1']
```

```
stdcell.values() #[0.3um2', '0.6um2']
```

`keys()`可以配合 `for` 循环对字典有序输出，在后面介绍循环的时候会重点介绍。

最后介绍判断键是否存在的操作 `in`，这是一个关键字，不是函数。如

```
'buf0' in stdcell
```

如果存在返回 **True**，不存在返回 **False**。一般配合 if 条件语句来用，后面介绍条件语句时有重点介绍。

字典与列表的异同？

通过上面的介绍，可以看出字典与列表有两个相同点：**可变长度和可修改**。

整理下字典与列表的异同点：

	字典	列表
序列	无序，通过键来索引	有序，用位置来索引。支持序列操作，如 slice。
可变长度	是	是
可修改	是	是
常用函数	get, keys, values, pop, update, clear 等	count, extend, index, insert, pop, remove, reverse, sort, clear 等

与其它编程语言的异同？

最后我们用表格来整理一个 python 字典、perl 哈希、tcl 字典、C 哈希的区别，方便大家学习：

	python 字典	perl 哈希	tcl 字典	C 哈希
定义	<code>d = {'name': 'bill'}</code>	<code>my %d = ('name' => 'bill');</code>	<code>set d [dict create name 'bill']</code>	c 语言本身不自带（一般是

				基于“结构体+链表” 从零开始 码)
判断 是否 存在 键	'name' in d	exists \${'n ame'}	dict exists \$d name	自己写函 数实现
是否 可变 长	是	是	是	是
常用 操作	get, keys, value s, pop, update, c lear	仅有 keys, values, exis ts, delete 等	append, cre ate, exists, f ilter, for, ge t, incr, info, keys, lappe nd, merge, r emove, repl ace set, size, un set, update, values, with (函数最 丰富)	自己写函 数实现

到此为止，主要的数据类型我们就介绍完了。剩下的其它类型如，文件、集合等后面实际用到时再作介绍。

下一次我们将进入下一个章节，介绍语句（if else、for、while）。

python 的条件和循环

https://mp.weixin.qq.com/s?__biz=MzlyMjYxNzA4NQ==&mid=2247483854&idx=1&sn=9a597b39d341b390af756f4694b9f369&chksm=e82b8d5cdf5c044ad3ef200b4c99960044b9dde0f29335a5cc3912c50cd79044a298919c149b&mpshare=1&scene=1&srcid=1228L8h8aPbERcvIOEAciqGC&pass_ticket=yGGnUzynyDHx%2BBqD18qh8gJ2LpHzm4gG%2BqquUw8O4yl%2FdMIBVGoDufQ9ipxE23P#rd

Original2017-12-04ExASICExASICExASIC

今天开始介绍 python 语言的控制流语句——条件 (if-else, if-elif-else) 和循环 (while, for)。

有了基本的数据 (砖瓦), 我们还需要按一定的逻辑 (水泥), 把数据合理的组织起来, 才能实现特定的功能 (房子)。这部分内容与其它语言, 比如 C++ 语言, 很接近, 我们主要通过对比介绍它们的异同点。

条件

	C++	Python
两个分支	<pre>if(判断条件){ 执行语句 A } else{ 执行语句 B }</pre>	<pre>if 判断条件: 执行语句 A else: 执行语句 B</pre>

多 个 分 支	if(判断条件){	if 判断条件:
	执行语句 A	执行语句 A
	}else if(另一个判断条件){	elif 另一个判断条件:
	执行语句 B	执行语句 B
	}else{	else:
	执行语句 C	执行语句 C
	}	

可以看出 python 有两点不一样，一是 if、elif、else 后有冒号:，二是 else if 简写成 elif。else if (C++) -> elsif (Perl) -> elif (Python)，越来越简洁。

例如，下面是一段网表处理的代码片段。当不需要 else 分支时，可以省略。

```

if opt == '-h':
    print('simplify_sram_netlist.py -i <inputfile> -o <outputfile>')
    sys.exit()
elif opt in ("-i", "--ifile"):
    ifileName = arg
elif opt in ("-o", "--ofile"):
    ofileName = arg

```

条件 if-else 语句可以嵌套，用缩进来控制。下面的示例中，if-else 嵌套了三级。

```

if not allmodule:
    print("Can not find any module!")
else:
    for m in allmodule:
        searchModName = re.search(r'module\s([\w_]+\s', m, re.S)
        mName = searchModName.group(1)
        if mName in libMod:
            continue
        else:
            if mName == "rwckts0":
                strTimescale = "`timescale 1ns/100ps\n"
            elif mName == "inoutbuf":
                strTimescale = "`timescale 1ns/10ps\n"
            else:
                strTimescale = "`timescale 1ns/1ns\n"

```

循环

	while	for
定义	while 条件成立: 执行语句	for 成员 in 列表: 执行语句

例如，下面的 while 代码段从 10 打印到 1。for 代码段把列表 sModKeys 元素逐个写入文件 outfile 里。

```

cnt = 10
while cnt > 0:
    print("cnt = {}".format(cnt))
    cnt -= 1

```

```

for k in sModKeys:
    outfile.write(sMod[k])

```

while 一般用于有条件的循环，而 for 一般用来对序列或列表遍历操作。例如，我们知道字符串本身就是序列，所以字符串是可以逐个字符遍历的。

```
for c in "abc":  
    print(c, end=' ')
```

将打印出：

a b c

for 循环与 range

range 是 for 循环的最佳搭档，利用 range 我们可以递增或递减的、指定步长的循环。

range(start=0, stop, step=1)

start 是开始点，默认是 0。

stop 是结束点，但不包括结束点。

step 是步长，默认是 1。

当开始点是 0、步长是 1 时，可以简写成 range(stop)

例如，下面的代码将逐个输出字符串的字符。

```
s = "abc"  
for x in range(len(s)): #等价于 for x in [0, 1, 2]:  
    print(s[x])
```

又如，下面的代码逆序、间隔输出，将输出 f、d、b。

```
s = "abcdef"
for x in range(len(s)-1, -1, -2): #停止点用-1 是因为 s[0]有可能需要输出
    print(s[x])
```

for 循环与 zip

zip 可以把两个列表打包, for 可以并行对两个列表遍历。例如,

```
L1 = ["AND2", "NOR2", "XOR2"]
L2 = [10, 20, 30]
for (x,y) in zip(L1, L2):
    print("Cell {} used {}".format(x, y))
```

上面的代码将输出:

```
Cell AND2 used 10
Cell NOR2 used 20
Cell XOR2 used 30
```

这种方法, 常常可以遍历字典的 keys 和 values 列表。

for 循环与 enumerate

enumerate 可以同时取出序列的索引和元素, 当遍历序列时, 如果需要索引值时, 这将会比较方便。例如,

```
L1 = ["AND2", "NOR2", "XOR2"]
```



```
for (i,c) in enumerate(L1):  
    print("{}: {}".format(i, c))
```

将输出:

```
0: AND2  
1: NOR2  
2: XOR2
```

[for 循环与 map](#)

map 把序列的元素按照指定函数进行转换，并返回新的序列。例如，

```
s = "abc"
```

```
for x in map(ord, s):  
    print(x)
```

这个例子中，map 把字符串 s 转换成对应的 ASCII 码序列。其中 ord 可以替换成自定义的函数，这给了我们很大的便利。

[退出循环](#)

最后来看看如果中止和退出循环。对于循环我们可以用 `break` 和 `continue` 来退出循环。区别在于 `break` 退出整个 while 或 for，而 `continue` 会立即结束本次循环，进入下一次循环。

例如，下面的代码打印所有使用个数大于等于 5 个的 Cell。

```
L1 = ["NAND2", "NOR2", "NXOR2", "AND2", "OR2", "XOR2", "INV2", "BUF2"];
L2 = [10, 20, 30, 1, 6, 3, 60, 7]
i = 0
for (x,y) in zip(L1, L2):
    if y < 5:
        continue
    print("Cell {} used {}".format(x, y))
    i += 1
```



将输出:

```
Cell NAND2 used 10
Cell NOR2 used 20
Cell NXOR2 used 30
Cell OR2 used 6
Cell INV2 used 60
Cell BUF2 used 7
```

又如，下面的代码打印出使用最多的 6 种 Cell。先使用 `sorted()` 对字典按 value 排序，打印输出时，当输出次数达到 6 后退出 for 循环。

```
D1 = {"NAND2":10,
      "NOR2" :20,
      "NXOR2":30,
      "AND2"  :1,
      "OR2"   :6,
      "XOR2"  :3,
      "INV2"  :60,
      "BUF2"  :7}
L2 = sorted(D1.items(), key=lambda d : d[1], reverse=True)
D3 = dict(L2)
i = 0
for k in D3.keys():
    if i == 6: break
    print("Cell {} used {}".format(k, D3[k]))
    i += 1
```



将输出

Cell INV2 used 60
Cell NXOR2 used 30
Cell NOR2 used 20
Cell NAND2 used 10
Cell BUF2 used 7
Cell OR2 used 6

练习题

写一个 python 脚本统计网表中的 module 和 stdcell 的个数，并按照使用个数从多到少进行排列。

[参考脚本](#)

```

#!/Python-3.6.2/python

import re

#read in P&R verilog netlist
infile = open("top_pr.v", "r")
netlist = infile.read()
infile.close()

#find instances
all_inst = re.findall(r'^\s+([A-Z][A-Z0-9]+)\s+\w+\s+\(',
                    netlist,
                    re.M)

if not all_inst:
    print("Can not find any instance!")
else:
    total = len(all_inst)
    d_stdcell = {} #define dict type variable
    for m in all_inst:
        if m in d_stdcell:
            d_stdcell[m] += 1
        else:
            d_stdcell[m] = 1

    #sort by count value
    l_stdcell_sorted = sorted(d_stdcell.items(),
                             key=lambda d : d[1],
                             reverse=True)
    d_stdcell_sorted = dict(l_stdcell_sorted)

    #print report, format as below:
    print("Cell Name\tCount\tPercentage") #print title
    print("-" * 40) #print divide line
    for k in d_stdcell_sorted.keys():
        print("Cell {} \t{} \t{:.2f}%".format(k,
                                             d_stdcell[k],
                                             (100*d_stdcell[k])/total))

```

Cell Name	Count	Percentage
Cell ND2D00	2244	15.82%
Cell ND2D0	1681	11.85%
Cell DFCN2Q	1381	9.73%
Cell MUX2D0	1321	9.31%
Cell INV0	681	4.80%
Cell BUF4	527	3.71%
Cell XNR2D0	431	3.04%
Cell AN2D0	421	2.97%
Cell IND2D0	350	2.47%
Cell ND3D0	317	2.23%
Cell ANTENNA	310	2.19%
Cell ND4D00	308	2.17%
Cell NR2D00	299	2.11%

ExASIC

预告

下一次，我们将介绍 python 的函数。

python 的函数（一）：基本概念

https://mp.weixin.qq.com/s?_biz=MzlyMjYxNzA4NQ==&mid=2247483866&idx=1&sn=09aa5fc284fe23521318741eb4a973b4&chksm=e82b8d48df5c045e71ef89c210041b13eae8cbea3277bfd6e9ca235347d66115ce181ad3eb3b&mpshare=1&scene=1&srcid=1228DWhLg9hgXChfvx9r4wGh&pass_ticket=yGGnUzynyDHx%2BBqD18qh8gJ2LpHzm4gG%2BqqxuUw8O4yl%2FdMIBVGoDufQ9ipxE23P#rd

Original2017-12-22ExASICExASICExASIC

我们之前学了一些基础的过程语句，如 if else、while、for。随着我们 python 程序的功能越来越复杂，代码也就越来越长，因此我们就需要用“函数”来简化代码。我们通常把功能单一的、可重复利用的代码写成函数。函数的优点就是定义一次，可多次调用，提高的代码的**可复用性**、**可阅读性**、**可维护性**。

函数的定义

函数的定义用 `def` 关键字，一般格式如下：

```
def name(arg1, arg2, ...argN):  
    statement  
  
    return value
```

注：

1. `def` 创建了一个**函数对象**，把函数名字 `name` 指向这个函数对象。
2. 参数是可选的，可以没有参数，也可以有任意多个参数。参数的类型是任意的，可以是数字、字符串、列表，甚至是对象。
3. **参数传递时是对象引用**的方法，就是说传递的是指针。当函数内部修改输入参数时，外部的参数就真的被修改了。
4. `return` 可以返回一个值，也可以仅仅是 `return`，不带 `value`，甚至可以没有 `return`。当后两种情况时，函数会默认返回 `None`。

5. return 语句可以出现在函数主体的任意位置，一旦遇到 return 语句，函数就执行结束了。

函数也是对象？

与 C++、Perl 等语言不同，python 里的函数本身也是可执行代码（不是函数声明或者预定义）。python 解释器看到 def 时，先创建一个函数对象，然后把函数名字指向刚创建的函数对象。因此，函数与数字、字符串、列表等一样，也是一种数据类型。

函数定义在运行时 (Runtime) 执行？

我们来理解 python 的函数与 C++、Perl 等语言的另一个区别。上面讲到 python 解释器遇到 def 就立即创建对象，所以说 python 没有预先定义的说法，也就没有 compile 的说法。因此，python 的函数定义可以出现在任意的地方，例如下面的例子也是合法的：

```
def func_a ():    #创建对象 func_a

    a = 1

    def func_b (): #在 func_a 里面又创建了对象 func_b

        b = 2
```

又如:

```
if a > 10:
```

```
    def func():
```

```
        print("great than")
```

```
else:
```

```
    def func():
```

```
        print("less than")
```

```
func() #此处调用的函数是动态的
```

参数类型是可变的?

我们来看一个例子:

```
def times(x, y):
```

```
    return x*y
```

```
times(2, 4) # = 8
```

```
times(3.14, 2) # = 6.28
```

```
times("-", 20) # = "-----"
```



```
l = [1, 2]
```

```
times(l, 4)  #[1, 2, 1, 2, 1, 2, 1, 2]
```

从这个例子是不是感觉 python 到与 Perl 有不一样的地方。参数类型由实际传递的对象类型决定。在 python 里，对象类型不同，操作符*乘号就做不同的事情。这其实就是操作符的重载（C++里也是这样叫），实现了多态。

总结

到这里，我们把函数基础概念就讲完了。是不是学得特别累，python 的函数竟有这么多不同之处。（我们既然学 python，就必须一步一个脚印打好基础！）

我们总结一下要点：

1. 函数也是对象。
2. 函数定义是动态执行的，没有编译的过程，所以使用之前必须先定义。
3. 函数定义可以出现在任意地方，甚至在另一个函数内部。
4. 函数的参数是对象引用，是指针传递。
5. 函数的参数类型由传入的对象决定，利用操作符重载，实现了多态。

预告

下一次，我们学习函数的作用域。

python 的函数（二）：作用域

https://mp.weixin.qq.com/s?_biz=MzlyMjYxNzA4NQ==&mid=2247483872&idx=1&sn=cf39d2d46e1e88520a33605a6f73d78b&chksm=e82b8d72df5c04645ac8431b376099747055b069f05cc71becd66cfd20519bd7a06de98081ab&mpshare=1&scene=1&srcid=1228IAMlpE2O812w8Vp4fH7g&pass_ticket=yGGnUzynyDHx%2BBqD18qh8gJ2LpHzm4gG%2BqqxuUw8O4yl%2FdMIBVGoDufQ9ipxE23P#rd

Original2017-12-28ExASICExASICExASIC

我们在写函数时，时常需要引用全局的变量，或对全局变量赋值。又或者偶尔遇到局部变量与全局变量同名。在处理这些问题时，python 语言的游戏规则是怎样的？今天我们就来学习这方面的内容。

什么是作用域？

在 python 语言里，在函数内部定义的变量，仅在函数内有效。在函数外面定义的变量对全局有效。我们把这种变量的有效范围叫作变量的作用域。

在 python 语言里，当引用变量时，会按照由内向外、由近及远的找变量的定义及赋值。例如下面的代码：

```
a = 1
```

```
def func():
```

```
    b = a
```

```
c = a
```

当执行到 `b = a` 时，

第一步，会先在函数内部查找变量 `a`。

第二步，如果第一步没有找到，会继续向外、向上查找。在这里就是查找函数外部，查找函数定义之前的代码，找到 `a = 1`。（如果第一步查找到了 `a` 的定义，就忽略第二步。）

当执行到 `c = a` 时，向上跳过 `func()`，直接找到 `a = 1`。就是说会跳过同级的函数。也就是说，函数内的变量只对本函数有效，对外部没有影响。

三段代码对比

#示例一：函数内部

```
a = 1
def func_a ():
    a = 2
    def func_b ():
        a = 3
        print(a)
    func_b()
func_a() #打印结果 3
```

#示例二：向上一级

```
a = 1
def func_a ():
    a = 2
    def func_b ():
        print(a)
    func_b()
func_a() #打印结果 2
```

#示例三：全局

```
a = 1
def func_a ():
    def func_b ():
        print(a)
```

```
func_b()
func_a() #打印结果 1
```

global 和 nonlocal 是干什么的?

从上面的代码来看，python 会自动按照由里向外、由近及远的规则查找变量。
来看下面的几段代码。

#示例一：func_b 不能修改 func_a 里的变量

```
a = 1
def func_a ():
    a = 2
    def func_b ():
        a = 3
    func_b()
    print(a)
func_a() #打印结果仍然是 2
```

#示例二：func_b 修改了 func_a 里的变量

```
a = 1
def func_a ():
```

```
a = 2
def func_b ():
    nonlocal a
    a = 3
func_b()
print(a)
func_a() #打印结果 3
```

#示例三：func_b 修改了全局变量，而不影响 func_a 内部的变量

```
a = 1
def func_a ():
    a = 2
    def func_b ():
        global a
        a = 3
    func_b()
    print(a)
func_a() #打印结果仍然是 2

print(a) #打印结果 3
```

所以，当需要修改外部变量的值时 global 和 nonlocal 是必需的。一般建议是不管是引用还是修改，都使用 global 和 nonlocal。

for 循环没有单独的作用域

for 内部定义的变量在循环结束后变量仍然有效。如：

```
for i in range(3):  
    a = i  
print(a) #打印结果 2
```

总结

简单总结一下变量作用域的规则：

1. 由内向外、由近及远。
2. 可直接引用外部变量、全局变量（建议使用 nonlocal 和 global）。
3. 当需要修改外部变量和全局变量时必需使用 nonlocal 和 global 来定义。

预告

下一次，我们学习函数的参数传递。

python 的函数（三）：参数传递


https://mp.weixin.qq.com/s?_biz=MzlyMjYxNzA4NQ==&mid=2247483883&idx=1&sn=135678ba39506250b914d4148ac63b0b&chksm=e82b8d79df5c046f47c5d3da853feaa670c10ab71328320d5c9e01299119910fd7cd9dc550bb&mpshare=1&scene=1&srcid=0104mTKs6TZQ400xOAsWp2gC&pass_ticket=NUMTX4HBTHaF6kvI6%2FUeuqgSPJlrLKhPCae%2BGW2jBXDihUDd2AeHKRI%2FbKCo%2FQ6m#rd

Original2018-01-04ExASICExASICExASIC

今天我们来继续学习 python 的函数，学习参数传递的一些基本规则。

可修改型参数与不可修改型参数的区别

我们在学习数据类型的时候，知道 python 的数据类型有两类，不可修改型（数字、字符串等）和可修改型（列表、字典等）（*不记得的童鞋可以点击链*

接回忆一下 ）。可修改型和不可修改型作为函数参数时有较大的区别。我们下面通过几个简单的例子来说明。

#示例一：数字作为参数


```
def func(a):  
    print("id of a(initially):", id(a))  
    a = 88  
    print("id of a(assign):", id(a))  
    print("value of a:", a)
```

```
b = 99  
print("id of b:", id(b))  
print("value of b (before):", b)
```

```
func(b)  
print("value of b (after):", b)
```

输出结果:

```
id of b: 9086048  
value of b (before): 99  
id of a(initially): 9086048  
id of a(assign): 9085696 #创建了新对象  
value of a: 88  
value of b (after): 99 #函数外部的 b 不变
```

从输出结果我们可以清楚看到，刚进入函数时，a 指向了 b，也就是说 b 的对象传递到了函数内部。当把 a 赋值成 88 时，a 指向了一个新创建的对象，对象的值是 88。当函数 func 执行结束后，b 的值没有受到影响，仍是 99。

是乎有点绕，我们来提取核心内容：

1. 传递的是对象 `b`，而非数值 `99`。
2. 由于 `b` 是不可修改的数据类型，对函数的参数 `a` 赋值时，在函数内新建一个对象，然后用 `a` 指向这个对象。
3. 这个新对象是函数内部变量，作用域仅在函数内部，对外部全局变量没有影响。

那对于可修改类型的列表和字典有什么不一样的呢？我们通过下面的例子来看。

#示例二：

```
def func(a):  
    print("id of a(initially):", id(a))  
    a[0] = 4  
    print("id of a(assign):", id(a))  
    print("value of a:", a)  
  
b = [1, 2, 3]  
print("id of b:", id(b))  
print("value of b (before):", b)  
  
func(b)  
print("value of b (after):", b)
```

输出结果：

```
id of b: 140298248703240
value of b (before): [1, 2, 3]
id of a(initially): 140298248703240
id of a(assign): 140298248703240 #没有创建新对象
value of a: [4, 2, 3]
value of b (after): [4, 2, 3] #外部的值被修改
```

示例二的输出结果表明函数内部并没有创建新的列表。当修改列表元素时，实际上修改的就是函数外部的列表。函数执行结束后，看到外部的列表确实被修改。

可修改型参数的风险及解决办法

使用可修改型参数给我们提供了修改全局变量的方法。但如果使用不当，可能会误修改。我们要牢记这一点。那么有没有好的方法来避免这种风险呢？有！

一般有两种方法：

1. 使用元组 (Tuple) 代替列表。
2. 函数一开头把列表拷贝到本地。

#示例三

```
def func(a):  
    a[0] = 4  
  
b = [1, 2, 3]  
func(tuple(b))
```

输出结果:

TypeError: 'tuple' object does not support item assignment

示例三表明，把列表强制转换成元组再传递给函数，当函数内部有修改数组的操作时就会报错，提示元组不能赋值。这种方法虽简单暴力，但有局限性。局限性是列表数据类型提供的方法，如 `append`、`remove` 等，都不能使用。

再看下面的示例四，示例中提供了一种更优雅的方案。

#示例四

```
def func(a):  
    print("id of a(initially):", id(a))  
    a = a[:]
```

```
a[0] = 4
print("id of a(assign):", id(a))
print("value of a:", a)
```

```
b = [1, 2, 3]
print("id of b:", id(b))
print("value of b (before):", b)
```

```
func(b)
print("id of b:", id(b))
print("value of b (after):", b)
```

输出结果:

```
id of b: 140122896936264
value of b (before): [1, 2, 3]
id of a(initially): 140122896936264
id of a(assign): 140122896936328 #创建了新对象
value of a: [4, 2, 3]
id of b: 140122896936264
value of b (after): [1, 2, 3] #外部列表没有被修改
```

在函数一开头，通过 `a = a[:]` 创建了一个新的列表对象，`a` 指向了这个新对象。那么函数内部接下来的所有操作都只针对这个新的列表对象。

两种参数调用模式：按位置顺序、按名称

我们知道 verilog 有两种实例化方式，按位置顺序和按端口名称。同样 python 也有这两种方式。例如下面的两个例子。

#示例五：

```
def func(a, b, c):
```

```
    print(a, b, c)
```

```
func(1, 2, 3)
```

#示例六：

```
def func(a, b, c):
```

```
    print(a, b, c)
```

```
func(a=1, b=2, c=3)
```

```
func(b=2, c=3, a=1) #与上一行效果相同
```

示例六里，按名称调用时，变量的先后顺序就不重要了。按名称调用的好处是一目了然。

另外，函数定义时还可以指定参数默认值。如下面的例子：

#示例七：

```
def func(a, b, c=3):
```

```
    print(a, b, c)
```

```
func(1, 2)
```

这个例子里，对于定义了默认值的参数，在调用时如果不需要修改就可以省略。这样可以**让代码更简洁**。

注意，函数定义时，**没有默认值的在前，有默认值的在后**。像 `def func(a, b=2, c)` 是不符合 python 语法的，会报错。

两种调用模式的混用

按位置顺序和按名称调用可以混用。如下面的示例：

#示例八:

```
def func(a, b, c=3):
```

```
    print(a, b, c)
```

```
func(1, 2, c=4)
```

注意，混合调用时要讲究顺序，先按位置、后按名称调用。否则语法会报错。

print()函数的可变个数参数是怎么实现的?

我们都知道 print()函数支持任意多个参数，倒底是怎么实现的？其实 python 的参数还有两种*和**。*表示把传递的参数看作一个元组，**表示把传递的参数看作是一个字典。例如：

#示例九:

```
def func(*a):
```

```
    print(a)
```

```
func(1, 2, 3) #输出(1, 2, 3)
```


#示例十:

```
def func(**a):
```

```
    print(a)
```

```
func(a=1, b=2) #输出{'a': 1, 'b': 2}
```

在示例九里, 如果我们在函数内部做元组的解析和打印, 是不是就可以实现自己的 print()函数了呢? (习题 2)

批注 [QJ2]: 2/9 看到此处

*、**和普通参数混用的顺序

由于*、**不确定参数的个数, 所以一般放在参数列表的最后, 表示剩下的其它参数。

#示例十一:

```
def func(a, *b):
```

```
    print(a, b)
```

```
func(1, 2, 3) #输出 1 (2, 3)
```

如果*不在最后，函数调用时*后面的参数需要按名称调用。不然没办法判断*的参数到哪里结束。如下面的例子：

#示例十二：

```
def func(a, *b, c):
```

```
    print(a, b, c)
```

```
func(1, 2, 3, c=4) #输出 1 (2, 3) 4
```

```
func(1, 2, 3, 4) #会报错, c 没有赋值
```

一般比较好习惯是，参数按位置 -> 名称 -> * -> ** 顺序定义及调用。

总结

函数的参数传递细节问题非常多，我们写代码时要多用简单、容易理解的编码风格。对于参数我们总结如下几点：

1. 注意可修改型（如数字、字符串）和不可修改型（如列表、字典）参数的区别，并学会利用拷贝避免风险。
2. 按位置和按名称两种调用方式，复杂的情况下多用按名称调用。
3. 学会用*和**实现参数个数不确定的函数。
4. 注意参数定义和调用时按位置、名称、*、**的顺序。

习题

1. 实现一个函数 `args_parse()`，解析命令行的参数，把结果存在一个字典变量中。
2. 实现自己的 `print_anything()`，支持任意个任意类型的参数。

注：参考答案晚一天公布。

参考答案

1. 我们假设有类似 `vcs` 的参数：`-R -full64 -sverilog -timescale=1ns/1ps -y rtl -f rtl.flist +warn=none +vpdfile+debug.vpd`

```

import sys

my_args = {}
my_args_plus = {}
def args_parse(args):
    global my_args
    global my_args_plus
    k = ""
    v = ""

    args = args[1:]
    while len(args) > 0:
        arg = args.pop()
        if not arg.startswith('-') and not arg.startswith('+'):
            arg2 = args.pop()
            k = arg2[1:]
            if arg2.startswith('-'):
                my_args[k] = arg
            elif arg2.startswith('+'):
                my_args_plus[k] = arg
        elif arg.startswith('-'):
            arg = arg[1:]
            if arg.find('=') != -1:
                lst = arg.split('=')
                k = lst[0]
                my_args[k] = lst[1]
            else:
                my_args[arg] = "true"
        elif arg.startswith('+'):
            arg = arg[1:]
            if arg.find('=') != -1:
                lst = arg.split('=')
                k = lst[0]
                my_args_plus[k] = lst[1]
            elif arg.find('+') != -1:
                lst = arg.split('+')
                k = lst[0]
                my_args_plus[k] = lst[1]
            else:
                my_args_plus[arg] = "true"

args_parse(sys.argv)
print(my_args)
print(my_args_plus)
if 'sverilog' in my_args:
    print('exists sverilog option')
if 'vpdfile' in my_args_plus:
    print('vpd file name: ', my_args_plus['vpdfile'])

```

结果输出:

```
$ python3 func_args_ans1.py -R -full64 -sverilog -timescale=1ns/1ps -y rtl -f rtl.flist
+warn=none +vpdfile+debug.vpd
{'f': 'rtl.flist', 'y': 'rtl', 'timescale': '1ns/1ps', 'sverilog': 'true', 'full64': 'true', 'R': 'true'}
{'vpdfile': 'debug.vpd', 'warn': 'none'}
exists sverilog option
vpd file name: debug.vpd
```


点评：主要的思路就是逆序判断参数前有无-或+。把解析后参数存在字典很方便后续脚本的识别和判断。

参考代码下载：http://www.exasic.com/example/func_args_ans1.py

2. 利用*参数类型，有函数内部拷贝列表到内部变量，再逐个打印。

```
def print_anything(*args):
    args = args[:]
    for arg in args:
        print(arg)

print_anything(1, "abc", [4, 5, 6], {"name": "dut", "area": "40um2"})
```



结果输出：

```
$ python3 func_args_ans2.py
1
abc
[4, 5, 6]
{'name': 'dut', 'area': '40um2'}
```

参考代码下载：http://www.exasic.com/example/func_args_ans2.py

预告

下一次我们介绍函数的一些高级话题，如递归函数、匿名函数、函数的属性和标注 (Annotation) 等。

python 的函数 (四)：递归函数、匿名函数等

https://mp.weixin.qq.com/s?__biz=MzlyMjYxNzA4NQ==&mid=2247483958&idx=1&sn=d905c2297e066cd7b92645f897da36f2&chksm=e82b8ea4df5c07b233b6b36e297852ddffa82a84760b1d19c83be03e92ac99751de0627cf7d2&mpshare=1&scene=1&srcid=0115cvCW334galGk9w3krxou&pass_ticket=Fmbn6xM46UufNx%2FuHK7ANYmAhUx5uSoPYkv%2F%2FVgbXF08dvzbPLp0rdAXm3ba63jM#rd

2018-01-15ExASICExASICExASIC

今天学习函数的最后一节。一般来说，前面三节的语法知识已经够用了，这一篇是几个高级一点的话题，主要包括递归函数、匿名函数、函数的属性与标注。由于是高级话题，我们就简单的介绍。

啥?  前三节是啥?  好吧。

不记得的童鞋点下面快速回顾一下!

python 的函数（一）：基本概念

python 的函数（二）：作用域

python 的函数（三）：参数传递

递归函数

递归函数，简言之就是“自己调用自己”的函数。有点绕口，用一个简单的例子来说明。对一个列表求和，你会想到什么方法？

- for 循环，逐个加
- 内置 sum()函数

上面的方法都不错。但今天我们用递归来对列表求和。

#示例一

```
def list_sum(l):  
    print(l)  
  
    if not l:  
        return 0
```

```
else:
```

```
    return l[0] + list_sum(l[1:])
```

#把 l[1:], 即列表除第一个元素外, 传递给自己

```
sum = list_sum([1, 2, 3, 4, 5])
```

```
print(sum)
```

结果输出:

```
[1, 2, 3, 4, 5]
```

```
[2, 3, 4, 5]
```

```
[3, 4, 5]
```

```
[4, 5]
```

```
[5]
```

```
[]
```

```
15
```

这个例子中, 把列表除第一个元素外的部分作为新列表, 传递给递归函数。当列表为空时, 就达到了递归结束的条件, 计算出列表元素的和。这种方法仅仅用来演示递归函数的使用方法, 大家在求和时还是要用前两种方法哈。

匿名函数

匿名函数，顾名思义就是没有名字的函数，临时的函数。下面举两个例子。

#示例二

```
l = [lambda x : 1 ** x,  
     lambda x : 2 ** x,  
     lambda x : 3 ** x,  
     lambda x : 4 ** x,  
     lambda x : 5 ** x,  
     lambda x : 6 ** x]
```

```
for f in l:  
    print(f(2), end=' ')  
print("")
```

```
for f in l:  
    print(f(3))  
  
print("")
```

结果输出：

```
1 4 9 16 25 36
1 8 27 64 125 216
```

lambda 是匿名函数的定义关键字，紧接着是参数 x，冒号后面是函数主体，但函数主体只能写一句话。lambda 语句执行后，创建了一个函数对象，但没有名字。上面这个例子里，我们把这些函数对象放在一个列表里。在使用时，用 f 指向这个对象，用 f(x)形式来调用匿名函数。

通常匿名用 inline 形式，就是说作为一个表达式嵌入在代码里。再看一个例子。

#示例三

```
import tkinter
```

```
top = tkinter.Tk()
```

```
str_e1 = tkinter.StringVar()
```

```
e1 = tkinter.Entry(top, textvariable=str_e1)
```

```
str_e1.set("hello")
```

```
e1.pack()
```

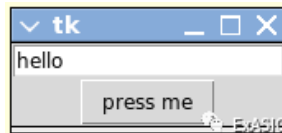
```
b1 = tkinter.Button(top, text="press me", command=lambda :
```

```
print(str_e1.get()))
```

```
b1.pack()
```

`top.mainloop()`

结果输出：



点击按钮，终端里输出 `hello`。

这个例子展示的是，在 GUI 编程时，可以用匿名函数实现一个简短的临时函数。比 `def` 要简洁很多。

函数的属性与标注

我们已经知道函数是一个对象，由 `<class 'function'>` 定义。那么作为一个对象，也可以用自己的属性（成员变量）和方法（成员函数）。我们用 `dir()` 来研究一下函数内部是什么样的。

假定有一个函数

```
def func(a, b):
```

```
    return a+b
```

执行 `dir(func)`显示如下:

```
['_annotations_', '_call_', '_class_', '_closure_', '_code_',  
'_defaults_', '_delattr_', '_dict_', '_dir_', '_doc_', '_eq_',  
'_format_', '_ge_', '_get_', '_getattribute_', '_globals_', '_gt_',  
'_hash_', '_init_', '_init_subclass_', '_kwdefaults_', '_le_', '_lt_',  
'_module_', '_name_', '_ne_', '_new_', '_qualname_', '_reduce_',  
'_reduce_ex_', '_repr_', '_setattr_', '_sizeof_', '_str_',  
'_subclasshook_']
```

`dir()`的返回值是一个函数对象内部属性和方法的列表。我们下面主要看看其中红色的 `_annotations_`, `_code_`, `_name_` 等。

`_name_`

`_name_` 里面保存的是函数的名字, `print(func._name_)` 输出 `func`。

`__code__`

`__code__` 是函数的主体对象，并包括函数的参数等信息。我们执行 `print(dir(func.__code__))` 输出如下：

```
['_class_', '_delattr_', '_dir_', '_doc_', '_eq_', '_format_', '_ge_',
 '_getattr_', '_gt_', '_hash_', '_init_', '_init_subclass_', '_le_',
 '_lt_', '_ne_', '_new_', '_reduce_', '_reduce_ex_', '_repr_',
 '_setattr_', '_sizeof_', '_str_', '_subclasshook_', 'co_argcount',
 'co_cellvars', 'co_code', 'co_consts', 'co_filename', 'co_firstlineno',
 'co_flags', 'co_freevars', 'co_kwonlyargcount', 'co_lnotab', 'co_name',
 'co_names', 'co_nlocals', 'co_stacksize', 'co_varnames']
```

例如，

`print(func.__code__.co_argcount)` 输出函数的参数个数 2。

`print(func.__code__.co_varnames)` 输出函数参数的列表 ('a', 'b')。

`__annotations__`

我们直接看下面的例子。

#示例四

```
def sum(a : int, b : int) -> int :  
    if not type(a) == sum.__annotations__['a'] :  
        print("Error: a should be type int")  
        return  
  
    return a + b  
  
s = sum(2, 5)  
print(s)  
s = sum('abc', 5)  
print(s)
```

结果输出

7

Error: a should be type int

None

这个例子，我们使用 annotation 来限制变量类型，当不符合变量类型时报错。其中：int 指定变量类型。函数的返回值的 annotation 写在最后面-> int。

至此，函数部分讲完了，《Python 在 ASIC 中的应用》的基础篇也结束了。各位童鞋学得怎么样？**编程没有捷径，一定要多写，多思考！**

下一次，我们一起学习《Python 在 ASIC 中的应用》的**中级篇**，学习一些模块、文件读写、正则表达式等。

python 的模块 module 介绍

Original2018-01-28ExASICExASICExASIC

https://mp.weixin.qq.com/s?__biz=MzIyMjYxNzA4NQ==&mid=2247483967&idx=1&sn=2730de41e00b4cf3e6f8ce1cca67b5f5&chksm=e82b8eaddf5c07bb051ee4b0565c2ab8c29bf084118c4abdc34d8bc42dfcfbbf4569150fb4da&mpshare=1&scene=1&srcid=01292ReChGrPS8UanzplZCX&pass_ticket=h6mJAbCsGZS83cqwyGU%2Fbo4wHLPb1dMHDiecrpSEnwqJsXEJajHCKFxUQql0Ep5#rd

引言

我们知道，Verilog 语言的模块是最底层的功能部件，底层模块在其它模块里
被例化，然后又在顶层模块里被例化。这样一级级、一层层构成了一个大项
目。

本文共计 2500 字，阅读大概需要 20 分钟。

概要：

- 模块是一块积木，是最基本的功能单位
- 一切都是以模块的形式存在的
- 再来理解 `if __name__ == '__main__':`
- 模块的搜索路径 `sys.path` 和 `PYTHONPATH`
- 如何安装模块（以 `numpy` 为例，重点讲离线、非管理员权限安装）
- 从源码开始安装 Tk 库

模块是一块积木

在 python 语言里也一样，模块是最基本的功能单位。我们写一个 python 脚本，通常需要导入（`import`）其它模块。比如，如果我们打算获取命令行参数，就需要 `import sys`。如果我们打算新建文件夹，就需要 `import os`。又如果我们打算使用科学计算，就需要 `import numpy`。在 python 的 PyPI (<https://pypi.python.org/pypi>) 里的第三方模块包已经超过 12 万个。所以绝大部分时候，我们并不需要从零开始造轮子。

一切都是以模块的形式存在的

在 python 里，一切都是以模块的形式存在的。可能有人反驳道，他写了一个最简单的 Hello world 脚本（如下面的 `hello_world.py`），并没有用到、也没有导入任何其它模块。

```
print("Hello world") #仅此一行
```


但实际情况是什么呢？python 解释器在启动的时候已经帮我们自动导入一个名叫 `builtins` 的模块。我们用 `dir()` 来帮助理解这一点。在 python 命令行里输入 `dir(__builtins__)`，输出如下的列表：

```
>>> dir(__builtins__)
```

```
[..., 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

是不是发现了这里面包括了 `print` 函数（红色）？此处稍微思考一会儿。。。很神奇是吧。是不是还发现了很多常用的函数（蓝色）？所以，不需要导入任何模块就可以直接使用的函数正是 `builtins` 模块中的函数。

之前讲 [函数作用域](#) 时讲过，查找变量的规则是“由内向外、由近及远”。这里的“远”就是指 `builtins` 模块。如果一直找到 `builtins` 模块，仍然没有找到变量的定义，就会报变量未定义的错误。

再来理解 `if __name__ == '__main__':`

一般来说，在 python 里模块名字就是文件名。我们可以通过内置变量 `__name__` 来读取模块名。例如：

```
>>> print(__name__)
```

```
__main__
```

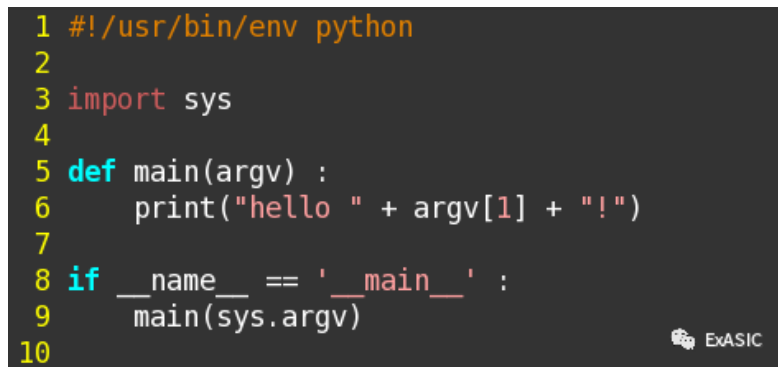
```
>>> import sys
```

```
>>> print(sys.__name__)
```

```
sys
>>> import os
>>> print(os.__name__)
os
```

当前正在执行的脚本被赋予的模块名字是__main__，相当于 C 语言的入口函数 main()。在《[我的第一个 Python 程序](#)》里，我们写了如下图的程序。我们判断模块的名字是不是__main__来决定要不要调用 main(argv)函数。这样写的好处是什么呢？当这个脚本被当作模块来做集成时，模块名是文件名 hello（而__main__变成是 import hello 模块的上层脚本），if __name__ == '__main__'条件里面的语句不会执行。

```
1 #!/usr/bin/env python
2
3 import sys
4
5 def main(argv) :
6     print("hello " + argv[1] + "!")
7
8 if __name__ == '__main__' :
9     main(sys.argv)
10
```



如果没有 if __name__ == '__main__'这一行，在被其它模块导入后，main(argv)会执行两次。

例如下面的例子，有 hello_world.py 和 test.py，位于同一个目录下。

```
#hello_world.py
def print_hello():
```

```
print("hello world")

print_hello() #没有放在 if 里

#test.py

import hello_world

hello_world.print_hello() #调用函数
```

结果输出

```
> python test.py
```

```
hello world
```

```
hello world
```

输出了两次 hello world? 你没看错。一次是 hello_world.py 里的，另一次是 test.py 里的。

所以，`if __name__ == '__main__':` 这句的作用就是，让脚本既可以直接当作执行脚本，又可以作当模块被导入。

模块的搜索路径

上面的例子里，我们导入自己写的 hello_world 模块时提到，须和 test.py 位于同一个文件夹下。就是说，test.py 在 import 时会在当前目录下查找叫做 hello_world.py 的文件。

我们可能会有疑问，如果项目做大了必须要分子目录了怎么办？安装第三方模块时是默认安装到了系统目录里，python 是如何找到的？又如，如果服务器上安装好几个版本 numpy 模块库，我们该怎么指定需要的版本？

想搞清楚这些问题，我们就需要先搞清楚 python 查找模块的规则。我们知道 linux 在查找可执行命令时是安照 PATH 环境变量的顺序。类似地，python 按照 sys.path 的顺序来查找模块定义。比如，windows 环境下，我们把 sys.path 打印出来，如下：

```
>>> import sys
```

```
>>> print(sys.path)
```

```
['', 'C:\\Python\\Python36\\python36.zip', 'C:\\Python\\Python36\\DLLs',  
'C:\\Python\\Python36\\lib', 'C:\\Python\\Python36', 'C:\\Python\\Python36\\lib\\site-  
packages']
```

我们可以看到，sys.path 其实是列表，里面存着一个一个路径。python 按照列表的先后顺序，逐个目录查找，直到找到模块定义。知道了这一点，我们就可以根据需要手工改造 sys.path 的列表，比如把我们自己写的模块目录 insert 到 sys.path 的一开头，比如 delete 掉某个不需要的目录，比如调换两个目录的顺序，等等。（点这里回顾一下[列表有哪些操作函数](#)）

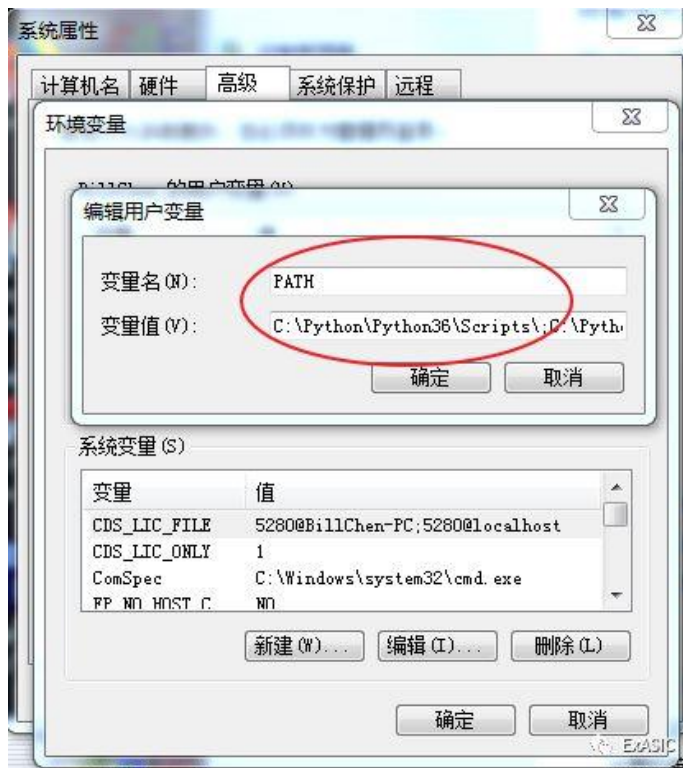
另外 python 启动时还会读取 PYTHONPATH 的环境变量，并 insert 到 sys.path 的一开头。因此，我们也可以修改 PYTHONPATH 来指定需要的模块所在目录。我们需要结合实际，决定是修改 PYTHONPATH 还是直接修改 sys.path。看哪个可行，看哪个更方便。

如何安装模块

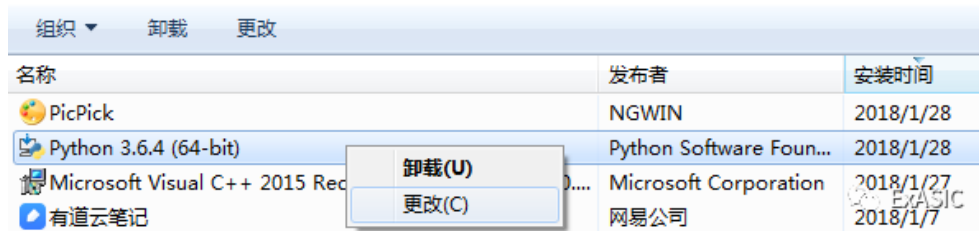
如果你的服务器可以联网，那么恭喜你，你只需要一个命令 pip install xxx 就可以安装第三方模块。pip 是 python 安装第三方模块的工具。比如，如果你需要安装 numpy，只需要 pip install numpy 即可，安装过程中会自动下载安装依赖包。

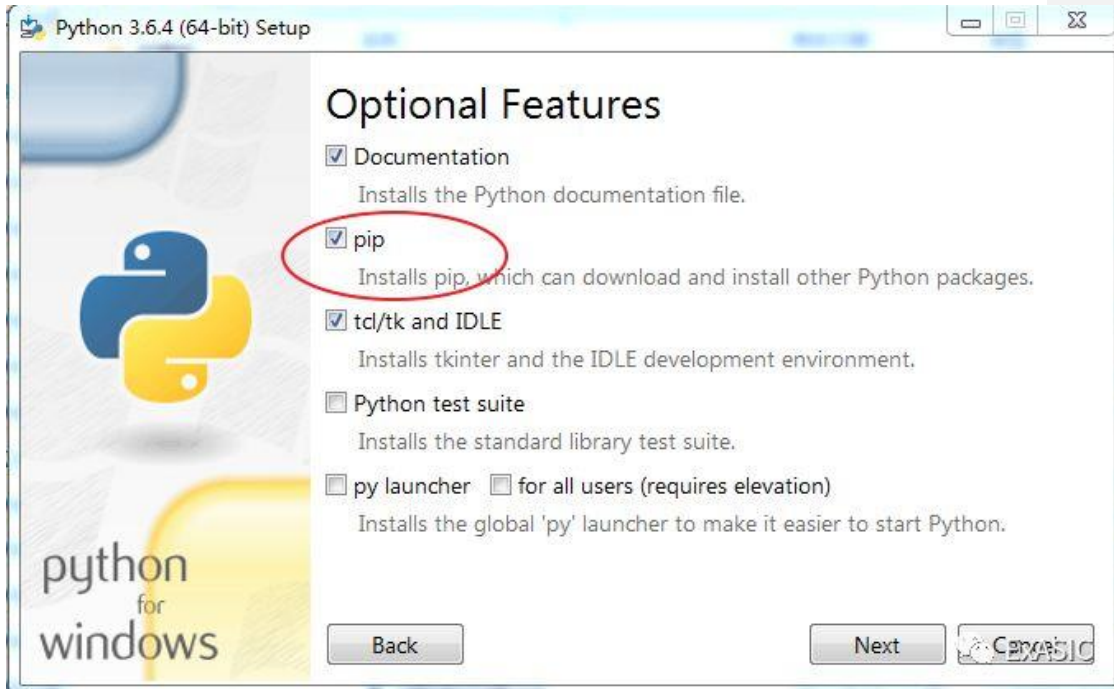
有些童鞋立即试了试，却发现“没有 pip 命令”？那很可能是没有安装 pip，或者没有正确设置 PATH 环境变量。

windows 环境下，先看看 python 安装目录下的 scripts 文件夹里有没有 pip.exe。如果有，就添加 C:\Python36\Scripts 到环境变量。如下图。



如果 Scripts 文件夹里没有 pip.exe，就需要安装 pip 了。在控制面板->添加删除软件的找到 python，右键点更改，再勾选上 pip。如下图。





但我们的服务器一般是 Linux，如果确认 Scripts 目录没有 pip，就需要去官网下载 pip 源码包来安装。下载地址：<https://pypi.python.org/pypi/pip/9.0.1>，找到 pip-9.0.1.tar.gz。解压缩后，用命令 `python setup.py install` 安装。

但是，但是，有不少童鞋说，“我们服务器不能连网，也没有管理员权限



”。不用担心，这种情况下，我们需要下载好 `numpy` 源码包和所有的依赖包，手动安装到自己的 HOME 目录或者指定的目录。

源码包去哪里下载？当然是去 python 官网，<https://pypi.python.org/pypi>。那么怎么知道要哪些依赖包呢？不用担心，如果缺少依赖包，安装时会报错。缺什么

补什么。我整理了安装 numpy 和 matplotlib 可能需要的依赖包，见百度网盘 <https://pan.baidu.com/s/1htmgfYG>。

一般来说，安装源码包安装方法是 `python setup.py install --prefix /home/xxx`，其中，`--prefix` 指定安装到自己的 HOME 目录下，因为我们没有管理员权限。

部分第三方模块，不需要安装，只要解压至一个目录，并加到 python 的模块搜索路径即可使用。如果遇到某些包实在无法安装，可以试试这种方法。

这里顺便介绍一下，如何安装 Tcl 和 Tk 包。一些童鞋在安装 matplotlib 模块库时，发现 python 缺少 Tk 库。分两种情况，如果 Linux 服务器能联网，且有管理员权限，则优先选用 `yum install xxx`（RHEL、CentOS）或者 `apt-get install xxx`（Ubuntu、Debian）。另一种情况就下没联网、没权限，我们下面简单介绍一下源码安装的步骤。

- 官网 <http://www.tcl.tk/software/tcltk/download.html> 下载源码包（tcl8.6.8-src.tar.gz 和 tk8.6.8-src.tar.gz），并解压目录 tcl8.6.8-src 和 tk8.6.8-src。
- 先安装 tcl，进入 tcl8.6.8-src 目录，阅读 INSTALL 和 README 等说明文件
- `./configure --prefix=/home/xxx`（xxx 是你的用户名）
- `make`
- `make test`
- `make install`
- 再安装 tk，进入 tk8.6.8-src，安装方法与 tcl 相同

一般情况下，到这里，你已经可以在 python 中 `import tkinter` 了。但如果你的 python 也是自己用源码编译安装的话，那很可能你需要重新编译安装 python。方法如下：

- 找到上次编译 python 的源码目录，如果不幸源码目录被删了，那就重新下载 python 的源码

- 进入源码目录，修改 Setup 和 Setup.dist 这两个文件中有关 tcl 和 tk 的库的路径，如下图白色文件部分。
- 重新 make、make test、make install。


```
# The _tkinter module.
#
# The command for _tkinter is long and site specific.
# Uncomment and/or edit those parts as indicated.
# specific extension (e.g. Tix or BLT), leave the
# commented out. (Leave the trailing backslashes
# experience strange errors, you may want to join
# lines and remove the backslashes -- the backslash
# done by the shell's "read" command and it may not
# every system.
```

```
# *** Always uncomment this (leave the leading under
_tkinter _tkinter.c tkappinit.c -DWITH_APPINIT \
# *** Uncomment and edit to reflect where your Tcl/Tk
-L/home/chenf/lib \
# *** Uncomment and edit to reflect where your Tcl/Tk
-I/home/chenf/include \
# *** Uncomment and edit to reflect where your X11 headers
-I/usr/X11R6/include \
# *** Or uncomment this for Solaris:
-I/usr/openwin/include \
# *** Uncomment and edit for Tix extension only:
-DWITH_TIX -ltix8.1.8.2 \
# *** Uncomment and edit for BLT extension only:
-DWITH_BLT -I/usr/local/blt/blt8.0-unoff/incl
# *** Uncomment and edit for PIL (TkImaging) extension
(See http://www.pythonware.com/products/pil/ for
-DWITH_PIL -I../Extensions/Imaging/libImaging
# *** Uncomment and edit for TOGL extension only:
-DWITH_TOGL togl.c \
# *** Uncomment and edit to reflect your Tcl/Tk version
-ltk8.5 -ltcl8.5 \
# *** Uncomment and edit to reflect where your X11 libraries
-L/usr/X11R6/lib \
# *** Or uncomment this for Solaris:
-L/usr/openwin/lib \
# *** Uncomment these for TOGL extension only:
-1GL -1GLU -1Xext -1Xmu \
```

预告

下一次，我们将介绍如何把自己的 python 脚本做成模块。

直接产生 verilog 的 testbench 的 python 脚本

<http://bbs.eetop.cn/thread-320584-1-1.html>

看到有人贴的直接生成 testbench 的 perl 脚本，也贴一个 python 脚本，用法同 perl 脚本。该脚本适用于对 verilog 代码，直接生成 testbench。

使用：在 python 编译环境中输入 `python vTbgenerator.py ModuleFileName.v`

注意：ModuleFileName.v 被替换成你需要的例化文件，另外要将 vTbgenerator.py ModuleFileName.v 放在同一个文件夹中，这样就可以生成 ModuleFileName.v 的测试文件了。

[eetop.cn_vTbgenerator.zip](#)

PythonTutor 一个帮助理解 python 执行过程的网站

Original2018-02-08ExASICExASICExASIC

https://mp.weixin.qq.com/s?__biz=MzIyMjYxNzA4NQ==&mid=2247483975&idx=1&sn=ff79eb77308fae6d975daf414db7c591&chksm=e82b8ed5df5c07c3525677e1f4263f589961ec7c00883f225bb88029d0b8e6b7d7c7ab511b2d&mpshare=1&scene=1&srcid=0208XrmNtwx0dBhWOByFibHv&pass_ticket=AiVdY3oDOAM5n8e1%2Fs4xLaNvfTkKf0oMnr9rhFRGdittcSlqiSwljJMPML2K0fpg#rd

如果你，

常常搞不懂 python 的**执行过程**？

常常不理解 python 的**对象**？

下面介绍的这个网站将会帮你搞定！

先看两个 GIF 动画。下面这个动画演示了函数调用的过程。

（注：原网页是 GIF 动画，但 copy 到此后为静止画面，需要到原网页看 GIF 动画）

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1
2 def hello(name):
3     print('hello', name)
4
5 if __name__ == '__main__':
6     hello('python')
7
```

Print output (drag lower right corner to resize)

Frames Objects

→ line that has just executed
→ next line to execute

<< First < Back Step 1 of 6 Forward > Last >>

又如下面这个动画演示了递归函数的执行过程。

（注：原网页是 GIF 动画，但 copy 到此后为静止画面，需要到原网页看 GIF 动画）

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1
2 def sum(l):
3     if not l:
4         return 0
5     else:
6         return l[0] + sum(l[1:])
7
8 if __name__ == '__main__':
9     l = [1, 2, 3]
10    s = sum(l)
11    print(s)
12
```

Print output (drag lower right corner to resize)

Frames Objects

→ line that has just executed
→ next line to execute

<< First < Back Step 1 of 21 Forward > Last >>

这么神奇的网站就是 **Python Tutor**！网址：<http://www.pythontutor.com>。下面是官网的介绍：

Python Tutor, created by Philip Guo (@pgbovine), helps people overcome a fundamental barrier to learning programming: understanding what happens as the computer runs each line of source code.

Using this tool, you can write Python 2, Python 3, Java, JavaScript, TypeScript, Ruby, C, and C++ code in your web browser and visualize what the computer is doing step-by-step as it executes.

提取下重点就是：用可视化图形帮助人们理解计算机在执行每一行代码时发生了什么。

可以上传自己的代码：

<http://www.pythontutor.com/visualize.html>

也可以直接在线写代码：

<http://www.pythontutor.com/live.html>

好了，废话不多说了，自己开始研究吧！

预告

这两天会写一篇文章，《写 Python 需要 IDE 吗》，敬请期待！

基于 Python 的数字信号处理初步

https://mp.weixin.qq.com/s?__biz=MzAxOTIxNTg0Mg==&mid=2650993039&idx=2&sn=c57a623037ab168ca68ea68881d37d1f&chksm=803c3502b74bbc144f8daccfc6bf0229c4904b4faaa22a8b4510d4f2b25c0c84a60d0accf71&mpshare=1&scene=1&srcid=12125xbCbTMMRI3B14cEQnbr&pass_ticket=CySkFwhyKgkXMXq28SINHeryhlea13xouc785gQolpyc%2FHvfK2GjELDHdv6%2FG1Bu#rd

Original2017-12-12许欢EETOPEETOP

欢迎加入EETOP行业群及区域群

加群方法：

1. 长按二维码，加群管为好友
2. 发送信息：加群
3. 接收群管发送的加群信息表，认真填写



区域群				行业群			
北京	上海	深圳	成都	IC设计	FPGA	IC验证	版图
福建	武汉	合肥	江苏	深度学习	物联网	汽车电子	射频微波
西安	浙江	东三省	天津	封测	EMC	信号完整	数字后端

来源：EETOP 行者无疆（论坛 username:ICNO.1）的博客

地址：<http://www.eetop.cn/blog/?xhsir520>

Python 是目前的热门语言，一直觉得掌握一门编程语言对作为搞技术的来说还是很有必要的，结合工作中能用到的一些数据处理和分析的内容，觉得从数据分析入手，争取能够掌握 Python 在数据处理领域的一些应用。下面是基于 Python 的 numpy 进行的数字信号的频谱分析介绍

一、傅里叶变换

傅里叶变换是信号领域沟通时域和频域的桥梁，在频域里可以更方便的进行一些分析。傅里叶主要针对的是平稳信号的频率特性分析，简单说就是具有一定周期性的信号，因为傅里叶变换采取的是有限取样的方式，所以对于取样长度和取样对象有着一定的要求。

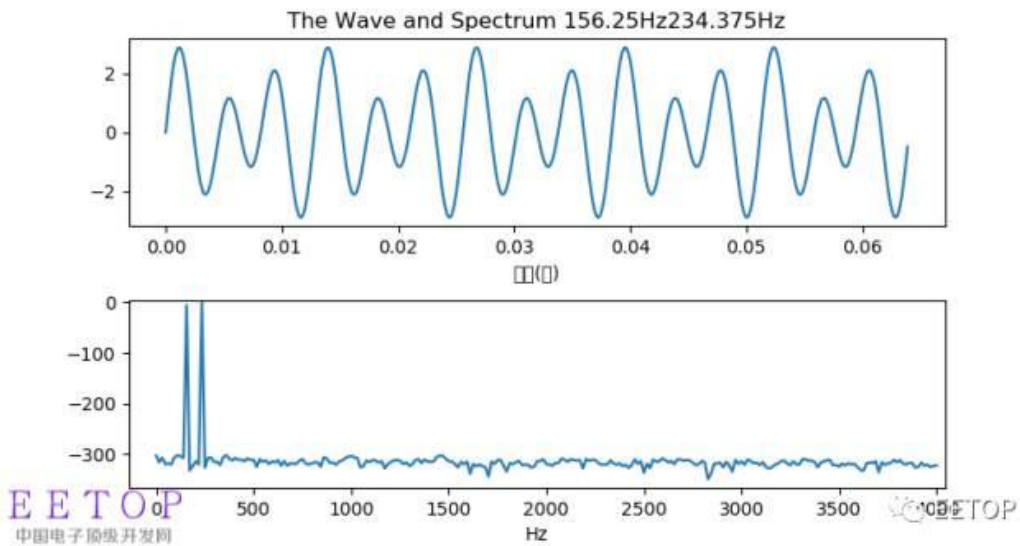
二、基于 Python 的频谱分析

```
# *_ coding:utf-8 *_  
  
import numpy as np #导入一个数据处理的模块  
  
import pylab as pl #导入一个绘图模块，matplotlib 下的模块  
  
sampling_rate = 8000 ##取样频率  
  
fft_size =512 #FFT 处理的取样长度  
  
t = np.arange(0,1,1.0/sampling_rate)  
  
#np.arange(起点， 终点， 间隔)产生 1s 长的取样时间  
  
x = np.sin(2*np.pi*156.25*t)+2*np.sin(2*np.pi*234.375*t)  
  
#两个正弦波叠加， 156.25HZ 和 234.375HZ， 因此如上面简单  
  
#的介绍 FFT 对于取样时间有要求，  
  
#N 点 FFT 进行精确频谱分析的要求是 N 个取样点包含整数个  
  
#取样对象的波形。  
  
#因此 N 点 FFT 能够完美计算频谱对取样对象的要求
```

```
#是  $n \cdot F_s / N$  ( $n \cdot$ 采样频率/FFT 长度) ,  
  
#因此对 8KHZ 和 512 点而言,  
  
#完美采样对象的周期最小要求是  $8000 / 512 = 15.625$  HZ,  
  
#所以 156.25 的  $n$  为 10, 234.375 的  $n$  为 15。  
  
xs = x[:fft_size]# 从波形数据中取样 fft_size 个点进行运算  
  
xf = np.fft.rfft(xs)/fft_size # 利用 np.fft.rfft()进行 FFT 计算, rfft()是为了更方便  
  
#对实数信号进行变换, 由公式可知/fft_size 为了正确显示波形能量  
  
# rfft 函数的返回值是  $N/2 + 1$  个复数, 分别表示从 0(Hz)  
  
#到  $\text{sampling\_rate}/2$ (Hz)的分。  
  
#于是可以通过下面的 np.linspace 计算出返回值中每个下标对应的真正的频率:  
  
freqs = np.linspace(0,sampling_rate/2,fft_size/2+1)  
  
# np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)  
  
#在指定的间隔内返回均匀间隔的数字  
  
xfp = 20*np.log10(np.clip(np.abs(xf),1e-20,1e1000))  
  
#最后我们计算每个频率分量的幅值, 并通过  $20 \cdot \text{np.log}_{10}()$   
  
#将其转换为以 db 单位的值。为了防止 0 幅值的成分造成  $\text{log}_{10}$  无法计算,  
  
#我们调用 np.clip 对 xf 的幅值进行上下限处理  
  
pl.figure(figsize=(8,4))  
  
pl.subplot(211)  
  
pl.plot(t[:fft_size], xs)
```

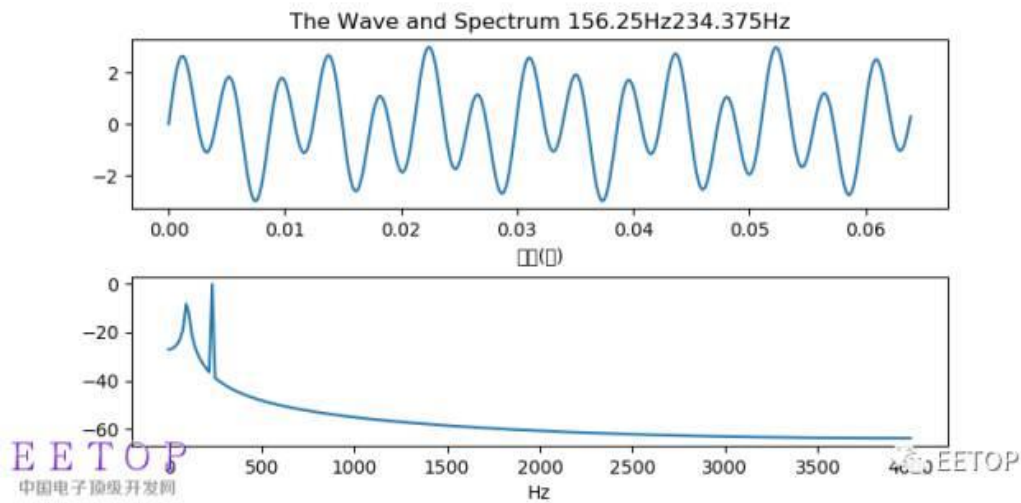
```
pl.xlabel(u"时间(秒)")  
  
pl.title(u"The Wave and Spectrum 156.25Hz234.375Hz")  
  
pl.subplot(212)  
  
pl.plot(freqs, xfp)  
  
pl.xlabel(u"Hz")  
  
pl.subplots_adjust(hspace=0.4)  
  
pl.show()
```

#绘图显示结果



现在来看看频谱泄露，将采样对象的频率改变


```
x = np.sin(2*np.pi*100*t)+2*np.sin(2*np.pi*234.375*t)
```



我们明显看出，第一个对象的频谱分析出现“泄露”，能量分散到其他频率上，没法准确计算到计算对象的频谱特性。

窗函数

上面我们可以看出可以通过加“窗”函数的方法来处理，尽量保证 FFT 长度内的取样对象是对称的。

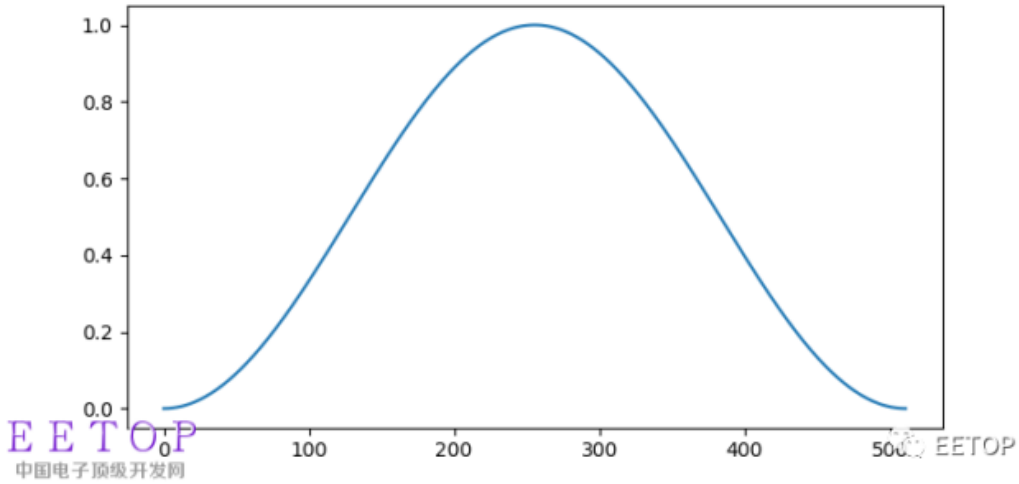
```
import pylab as pl
```

```
import scipy.signal as signal
```

```
pl.figure(figsize=(8,3))
```

```
pl.plot(signal.hann(512))#汉明窗函数
```

```
pl.show()
```



对上述出现频谱泄露的函数进行加窗处理，后面会介绍一下各种加窗函数的原理和效果。

用 Python 设计芯片

2017-12-22 [EETOP](#) [EETOP](#)

欢迎加入EETOP行业群及区域群

加群方法：

1. 长按二维码，加群管为好友
2. 发送信息：加群
3. 接收群管发送的加群信息表，认真填写



区域群				行业群			
北京	上海	深圳	成都	IC设计	FPGA	IC验证	版图
福建	武汉	合肥	江苏	深度学习	物联网	汽车电子	射频微波
西安	浙江	东三省	天津	封测	EMC	信号完整	数字后端

看到论坛里网友上传了一个关于 python 开发硬件的资料

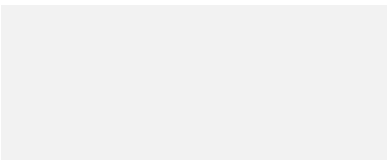
链接地址：

<http://bbs.eetop.cn/thread-666923-1-1.html>

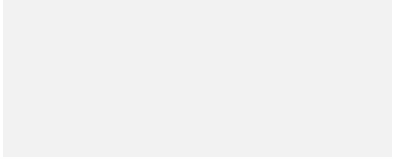
我已下载，见《eetop.cn_PythonMyHDL.pdf》

可以登录论坛下载。

资料目录截屏：



- 1 Overview**
- 2 Background information**
 - 2.1 Prerequisites
 - 2.2 A small tutorial on generators
 - 2.3 About decorators
- 3 Introduction to MyHDL**
 - 3.1 A basic MyHDL simulation
 - 3.2 Signals, ports, and concurrency
 - 3.3 Parameters and hierarchy
 - 3.4 Bit oriented operations
 - 3.5 Some concluding remarks on MyHDL and Python
 - 3.6 Summary and perspective
- 4 Modeling techniques**
 - 4.1 Structural modeling
 - 4.2 RTL modeling
 - 4.3 High level modeling
- 5 Unit testing**
 - 5.1 Introduction
 - 5.2 The importance of unit tests
 - 5.3 Unit test development
- 6 Co-simulation with Verilog**
 - 6.1 Introduction
 - 6.2 The HDL side
 - 6.3 The MyHDL side
 - 6.4 Restrictions
 - 6.5 Implementation notes
- 7 Conversion to Verilog and VHDL**
 - 7.1 Introduction



7.7	Excluding code from conversion	
7.8	User-defined code	
7.9	Template transformation	
7.10	Conversion output verification by co-simulation	
7.11	Conversion of test benches	
7.12	Methodology notes	
7.13	Known issues	

8 Conversion examples

8.1	Introduction	
8.2	A small sequential design	
8.3	A small combinatorial design	
8.4	A hierarchical design	
8.5	Optimizations for finite state machines	
8.6	RAM inference	
8.7	ROM inference	
8.8	User-defined code	

9 Reference

9.1	Simulation	
9.2	Modeling	
9.3	Co-simulation	
9.4	Conversion to Verilog and VHDL	
9.5	Conversion output verification	

Module Index

Index



关于 Python 开发硬件小编实在是外行，有兴趣的可以到 EETOP 论坛讨论和下载资料。

关于 Python 开发硬件的讨论链接：

<http://bbs.eetop.cn/thread-358628-1-1.html>

MyHDL 官网：

<http://www.myhdl.org>

以下是摘自官网的简单介绍：

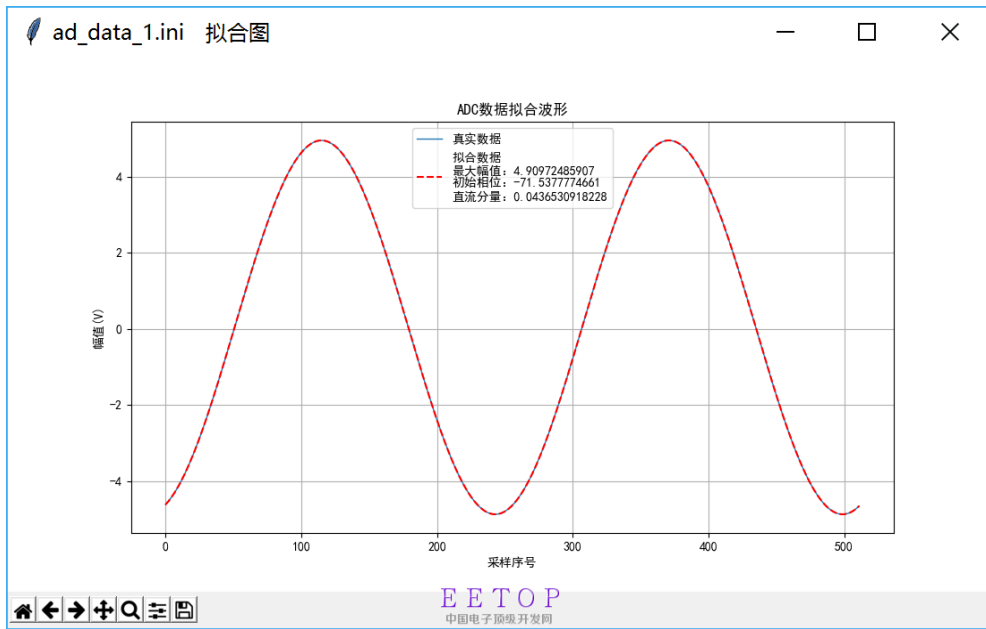
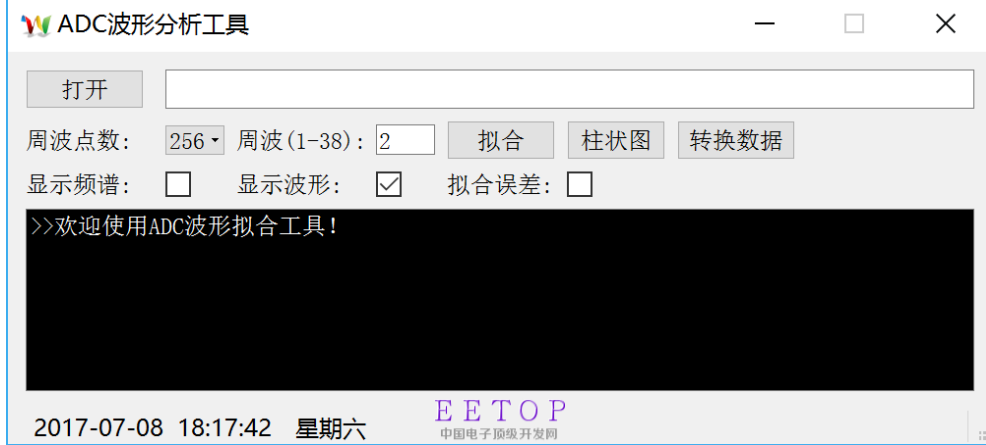
MyHDL

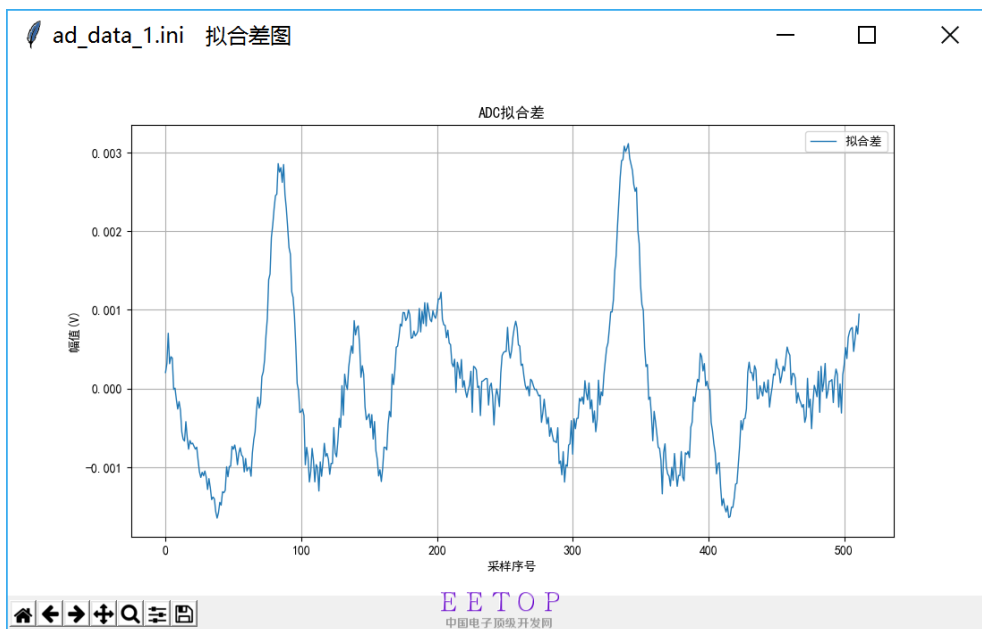
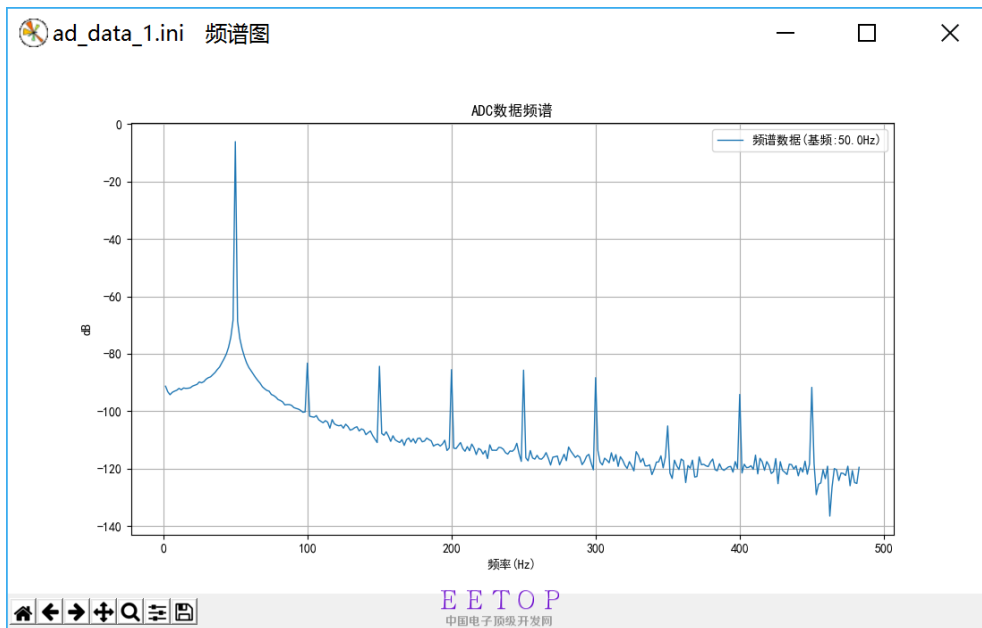
From Python to Silicon!

Design hardware with Python

MyHDL turns Python into a hardware description and verification language, providing hardware engineers with the power of the Python ecosystem.

程序基于 PyQ5, Python 3.6.1 注意: 数据输入是二进制格式, 每个数据点为 16 位, 10000 点数据。





源程序:



[ADCAnalyse.zip](#) (4.18 KB)

用 Python 给头像加上圣诞帽

https://mp.weixin.qq.com/s?__biz=MzU2MDAyNzk5MA%3D%3D&mid=2247483833&idx=1&sn=0106ed1947a5be4c851f8a94ce44f857&scene=45#wechat_redirect

Original2017-12-23冰不语CVPyCVPy



引言

随着圣诞的到来，大家纷纷@官方微信给自己的头像加上一顶圣诞帽。当然这种事情用很多 P 图软件都可以做到。但是作为一个学习图像处理的技术人，还是觉得我们有必要写一个程序来做这件事情。而且这完全可以作为一个练手的小项目，工作量不大，而且很有意思。

用到的工具

- OpenCV (毕竟我们主要的内容就是 OpenCV...)
- dlib (前一篇文章刚说过，dlib 的人脸检测比 OpenCV 更好用，而且 dlib 有 OpenCV 没有的关键点检测。)

用到的语言为 Python。但是完全可以改成 C++ 版本，时间有限，就不写了。有兴趣的小伙伴可以拿来练手。

流程

一、素材准备

首先我们需要准备一个圣诞帽的素材，格式最好为 PNG，因为 PNG 的话我们可以直接用 Alpha 通道作为掩膜使用。我们用到的圣诞帽如下图：



我们通过通道分离可以得到圣诞帽图像的 alpha 通道。代码如下：

```
1. r,g,b,a = cv2.split(hat_img)
2. rgb_hat = cv2.merge((r,g,b))
3.
4. cv2.imwrite("hat_alpha.jpg",a)
```

为了能够与 rgb 通道的头像图片进行运算，我们把 rgb 三通道合成一张 rgb 的彩色帽子图。Alpha 通道的图像如下图所示。



二、人脸检测与人脸关键点检测

我们用下面这张图作为我们的测试图片。



下面我们用 dlib 的正脸检测器进行人脸检测，用 dlib 提供的模型提取人脸的五个关键点。代码如下：

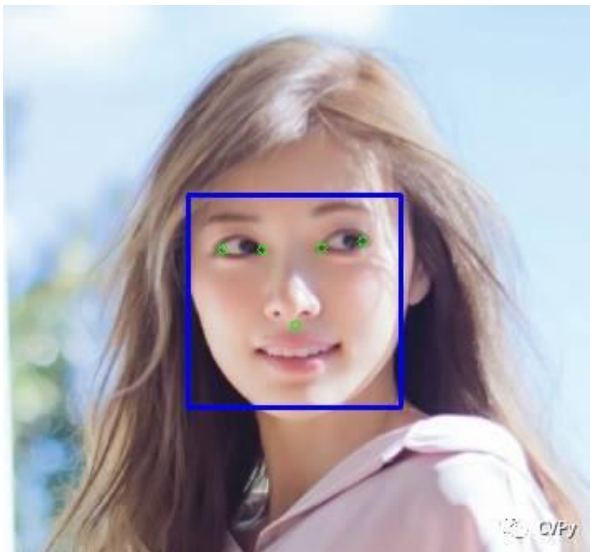
```
1. # dlib 人脸关键点检测器
```

```

2. predictor_path = "shape_predictor_5_face_landmarks.dat"
3. predictor = dlib.shape_predictor(predictor_path)
4.
5. # dlib 正脸检测器
6. detector = dlib.get_frontal_face_detector()
7.
8. # 正脸检测
9. dets = detector(img, 1)
10.
11. # 如果检测到人脸
12. if len(dets)>0:
13.     for d in dets:
14.         x,y,w,h = d.left(),d.top(), d.right()-d.left(), d.bottom()-d.top()
15.         # x,y,w,h = faceRect
16.         cv2.rectangle(img, (x,y), (x+w,y+h), (255,0,0), 2, 8, 0)
17.
18.         # 关键点检测, 5个关键点
19.         shape = predictor(img, d)
20.         for point in shape.parts():
21.             cv2.circle(img, (point.x,point.y), 3, color=(0,255,0))
22.
23.         cv2.imshow("image",img)
24.         cv2.waitKey()

```

这部分效果如下图:



三、调整帽子大小

我们选取两个眼角的点, 求中心作为放置帽子的 x 方向的参考坐标, y 方向的坐标用人脸框上线的 y 坐标表示。然后我们根据人脸检测得到的人脸的大小调整帽子的大小, 使得帽子大小合适。

```

1.     # 选取左右眼眼角的点
2.     point1 = shape.part(0)
3.     point2 = shape.part(2)
4.
5.     # 求两点中心
6.     eyes_center = ((point1.x+point2.x)//2, (point1.y+point2.y)//2)
7.
8.     # cv2.circle(img,eyes_center,3,color=(0,255,0))
9.     # cv2.imshow("image",img)
10.    # cv2.waitKey()
11.
12.    # 根据人脸大小调整帽子大小
13.    factor = 1.5
14.    resized_hat_h = int(round(rgb_hat.shape[0]*w/rgb_hat.shape[1]*factor))
15.    resized_hat_w = int(round(rgb_hat.shape[1]*w/rgb_hat.shape[1]*factor))
16.
17.    if resized_hat_h > y:
18.        resized_hat_h = y-1
19.
20.    # 根据人脸大小调整帽子大小
21.    resized_hat = cv2.resize(rgb_hat, (resized_hat_w, resized_hat_h))

```

四、提取帽子和需要添加帽子的区域

按照之前所述，去 Alpha 通道作为 mask。并求反。这两个 mask 一个用于把帽子图中的帽子区域取出来，一个用于把人物图中需要填帽子的区域空出来。后面你将会看到。

```

1.     # 用 alpha 通道作为 mask
2.     mask = cv2.resize(a, (resized_hat_w, resized_hat_h))
3.     mask_inv = cv2.bitwise_not(mask)

```

从原图中取出需要添加帽子的区域，这里我们用的是位运算操作。

```

1.     # 帽子相对与人脸框上线的偏移量
2.     dh = 0
3.     dw = 0
4.     # 原图 ROI
5.     # bg_roi = img[y+dh-resized_hat_h:y+dh, x+dw:x+dw+resized_hat_w]
6.     bg_roi = img[y+dh-resized_hat_h:y+dh, (eyes_center[0]-
7.         resized_hat_w//3):(eyes_center[0]+resized_hat_w//3*2)]
8.
9.     # 原图 ROI 中提取放帽子的区域
10.    bg_roi = bg_roi.astype(float)
11.    mask_inv = cv2.merge((mask_inv, mask_inv, mask_inv))
12.    alpha = mask_inv.astype(float)/255
13.
14.    # 相乘之前保证两者大小一致（可能会由于四舍五入原因不一致）
15.    alpha = cv2.resize(alpha, (bg_roi.shape[1], bg_roi.shape[0]))
16.    # print("alpha size: ", alpha.shape)
17.    # print("bg_roi size: ", bg_roi.shape)
18.    bg = cv2.multiply(alpha, bg_roi)
19.    bg = bg.astype('uint8')

```

这是的背景区域 (bg) 如下图所示。可以看到，刚好是需要填充帽子的区域缺失了。



然后我们提取帽子区域。

```
1. # 提取帽子区域
2. hat = cv2.bitwise_and(resized_hat, resized_hat, mask = mask)
```

提取得到的帽子区域如下图。帽子区域正好与上一个背景区域互补。



五、添加圣诞帽

最后我们把两个区域相加。再放回到原图中去，就可以得到我们想要的圣诞帽图了。这里需要注意的就是，相加之前 resize 一下保证两者大小一致，因为可能会由于四舍五入原因不一致。

```
1. # 相加之前保证两者大小一致（可能会由于四舍五入原因不一致）
2. hat = cv2.resize(hat, (bg_roi.shape[1], bg_roi.shape[0]))
3. # 两个 ROI 区域相加
4. add_hat = cv2.add(bg, hat)
5. # cv2.imshow("add_hat", add_hat)
6.
7. # 把添加好帽子的区域放回原图
8. img[y+dh-resized_hat_h:y+dh, (eyes_center[0]-
    resized_hat_w//3):(eyes_center[0]+resized_hat_w//3*2)] = add_hat
```

最后我们得到的效果图如下所示。



回复“圣诞”或者“圣诞帽”或者加入下方知识星球皆可获取完整代码的
Github 地址。

