引言 - Introduction

欢迎，GNU Radio 的新手。能有机会阅读这篇文章，便不难猜出是位对 GNU Radio 的基本知识已有相当了解的爱好者，它 - GNU Radio - 是什么以及它可以干什么已是不言自语的。现在，也想进入这个令人兴奋的开源的数字信号处理（DSP）世界，对吧！

这是一篇教你如何为 GNU Radio 编写 Python 应用的教程。它不是牵手你如何编程、构建软件无线电或进行信号处理的教程，也无意涵盖通过构建新的功能块或为源码树添加新代码来扩展 GNU Radio。如果具备前述的技能也想染指 GNU Radio，这篇文章便是为你准备的。如果对什么是软件无线电？或者 FIR 滤波器是干什么的？等等问题一团雾水，也许应当先补补这方面的功课，储备一下更加坚实的信号处理理论知识。但也千万不要被这些困难吓住 - 学习新东西最好的方法是尝试的去做。

尽管此教程是想尽可能轻松地引导你进入 GNU Radio 世界，但它决不想充当一个完善的指导教程的角色。事实上，有时不得不委曲求全地掩盖事实以便使得解释更容易理解。这也使得有些地方同后来的章节会变得相互矛盾。无论如何用心用脑一直是开发 GNU Radio 的必须。

序言 - Preliminaries

在着手此教程之前，应确保 GNU Radio 已经被安装并在你的计算机上并成功地运行起来。USRP 不是必须必备的，但一些形式的“源”- source（如：USRP) 和“漏”- sink (如：audio)还是很有帮助的。GNU Radio 的例程一旦运行成功（比如： gnuradio-examples/python/audio 目录下的 dial_tone.py），便可以傲游“星空”了。

编程的基本功还是需要一些的 - 没有用过 Python？不用担心，它是一门十分容易被掌握的语言。

理解流程图 - flow graphs

在探索代码之前，理应首先理解一些关联 GNU Radio 的基本概念，诸如：流程图 - flow graphs (图论的一种) 和功能块- block。很多的 GNU Radio 应用程序中仅仅包含这些流程图。关联这些流程图的节点便被称为功能块（block），数据流便在其中的连线穿行着。

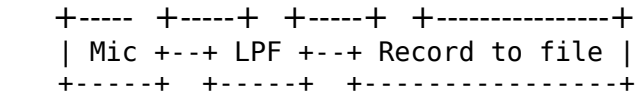
现实的信号处理都是在这些功能块（block）内完成的。理想的情况是，每个功能块仅仅完成一件工作 - 这样一来，GNU Radio 便能保持模块化（modular）和灵活化。功能块是由 C++ 构建的；编写新的功能块也不是一件很困难的工作（这将在其它地方阐述）。

数据可以以任何数据类型在功能块之间传递 - 实际上就是使用 C++ 能够被定义的任何数据类型。在现实使用中，最常用和最常见的数据类型是复合类型、长和短整型以及浮点型，数据从功能块之间传输可以是采样字节或位的形式。

举例 - Examples

下面通过一些实例来阐明这些晦涩的概念：

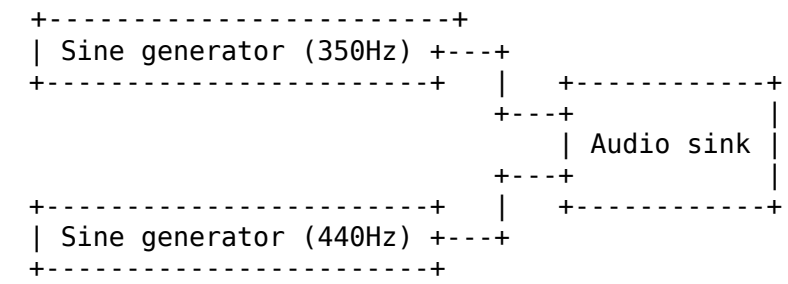
低通滤波器音频记录机 - **Low-pass filtered audio recorder**



首先，来自于麦克风（**Mic**）的音频信号通过 PC 的声卡被录制并且被转换为数字信号。这些采样的数据“流”向下一个功能块 - 低通滤波器（**LPF**）- 它可以用 FIR 滤波器来构建成。经过滤波的信号被传入最后一个功能块，功能块（**Record to file**）的功用是将被滤波的音频信号录制到一个文件之中。

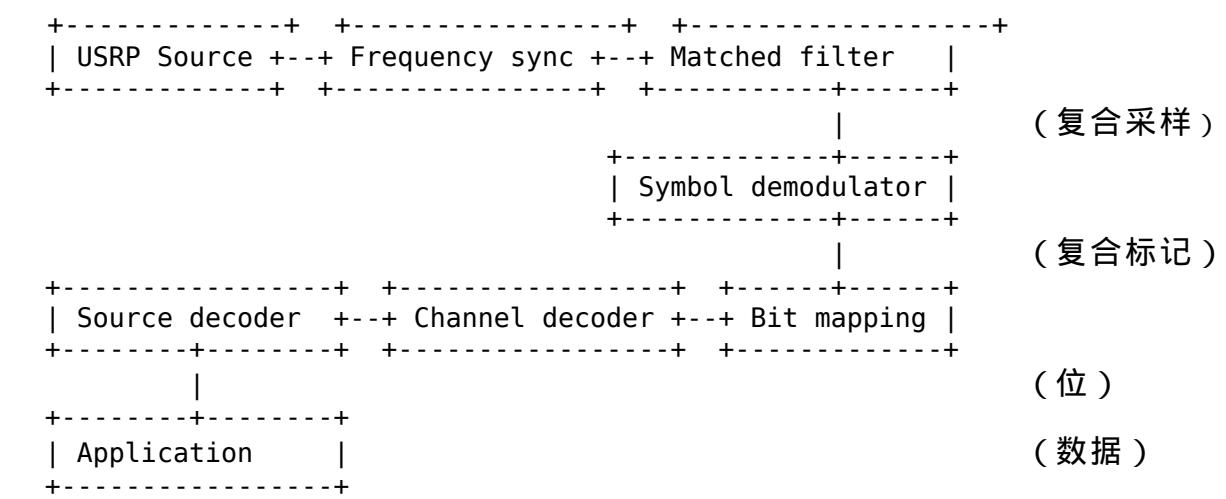
这是一个简单但完整的流程图。第一和最后一个功能块充当着特殊的功能：其功用作为“源” - source 和“漏” - sink。每个流程图都必须至少一个“源”和“漏”才能够完整地工作。

拨号音发生器 **- Dial tone generator**



这个简单的例程常常被称为"GNU Radio 的 Hello World“。同上一个例程不同的是，它有两个“源”。“漏”，在另一方面，有两个输入 - 在这里充当着声卡的左和右声道。在 gnuradio-examples/python/audio/dial_tone.py 目录下可以找到此例程的代码。

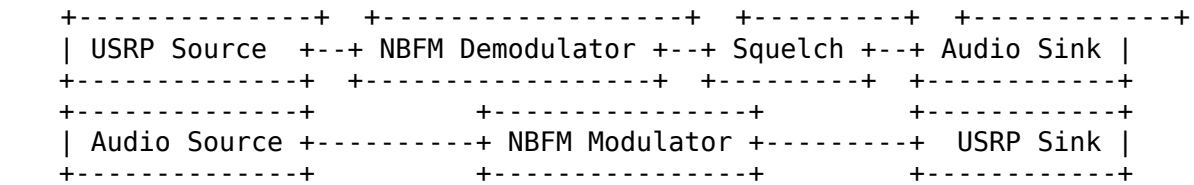
QPSK Demodulator（QPSK 解调器）



此例程便显得更“老道”一些，但 RF 工程师对此不会很陌生吧。USRP 在此充当着“源”的角色，它同天线相连接。这就是那种向下一个功能块“流”传复合采样（数据类型）的“源”。

让人饶有兴趣的是，此流程图展示了一个这样的现象：数据类型在流程图中间发生了转换。首先，复合形式的基带采样 - **complex samples**（数据类型）向下传递着。然后，通过此基带采样信号流解调出复合标记 - **complex symbols**。随后，这些标记转换成“位” - **bits** 继续着行程。最后，被解码处理过的位被传递到应用程序中被加以利用。

对讲机 **- Walkie Talkie**



此例由两个独立的流程图组成，它们工作在并行方式下。一个用于处理 Tx（发射）链路，另一个用于 Rx （接收）链路。对此还需要一些额外代码（凌驾于流程图之外）在其中一个链路被激活时，禁音另外一个链路。每个流程图仍旧需要至少一个“源”和“漏”。在 gnuradio-examples/python/usrp/usrp_nbfm_ptt.py 目录下，可以找到例程的代码（略微成熟的例程）。

小结 - Summary

在此对流程图做个小结。下面是一些你必须知道的关键点：

- GNU Radio 的信号处理通过流程图 - flow graphs 来构建的。
- 流程图由功能块（block）构成。功能块进行着信号处理的操作，诸如：滤波、叠加、变形、解码、硬件读取及很多其它的工作。
- 数据在功能块以多变的形式传输，复合、实整数、浮点或者说，能够被定义的任何形式。
- 每个流程图都必须至少一个“源”和“漏”。

第一个可行的代码例程

下面的工作便是探索如何用 Python 来实现这些流程图。下面我们一行一行的来分析代码。如果你已经很熟悉 Python，可以略去一些解释，但还是不要匆忙“撞入”下一部分 - 因为这些解释不光是针对 Python 新手，也包括 GNU Radio 的新人。

如下代码是用来实现流程图的例程 2.同 gnuradio-examples/python/audio/dial_tone.py 目录下的代码相比，这是一个略加修改过的版本。

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

from gnuradio import gr
from gnuradio import audio, analog

class my_top_block(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)

        sample_rate = 32000
        ampl = 0.1

        src0 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 350, ampl)
        src1 = analog.sig_source_f(sample_rate, analog.GR_SIN_WAVE, 440, ampl)
        dst = audio.sink(sample_rate, "")
        self.connect(src0, (dst, 0))
        self.connect(src1, (dst, 1))

if __name__ == '__main__':
    try:
        my_top_block().run()
```

```
    except [[KeyboardInterrupt]]:
        pass
```

对于具有 Unix 或 Linux 背景的人而言，第 **1** 行看起来不会很陌生：它用来告知 shell 此文件是一个 Python 文档，理应使用 Python 来解释并运行它。如果想直接从命令行来运行此文件，此行便是必须的。

第 **3** 和 **4** 行通过导入（import）命令来导入 Python 所需的模块用来运行 GNU Radio。导入命令同 C/C++ 的 #include 十分相似。在此，从 gnuradio-package 导入了两个模块：gr 和 audio 。第一个 gr 是 GNU Radio 的基础模块。这将是运行 GNU Radio 应用时总要必须导入（import）的模块。第二个是用来载入音频设备的功能块。GNU Radio 具有很多模块，其后将会给出一个简单的列单。

第 **6-17** 行定义了一个名字为 my_top_block 的类，它是从另外一个名字为 gr.top_block 的类继承下来的。此类(函数)对流程图而言基本上是个容器类（container）。通过对类 gr.top_block 的继承，便可得到所需的添加和连接功能块的 hooks 和 functions 等。

在这个类中仅只定义了一个函数：该类的构造函数 +init+()。在此函数的第一行（程序的第 **8** 行）便调用了父类的构造函数（这一点，在 Python 中是必须明示的。在 Python 中，有一个很重要的规矩，大多数动作都是必须明确地展示的）。另外定义了两个变量参数：sample_rate 和 ampl。它们分别用来控制信号发生器的采样速率和信号的振幅的。

在进而解释下一行之前，再回顾一下前一部分的流程图草图：它包含三个功能块和两个连接线。功能块在程序的 **13-15** 行被定义：它生成了两个信号源（分别被命名为 src0 和 src1）。这些信号源以固定频率（350 和 440Hz）持续性的产生正弦波信号。其振幅是由变量 ampl 来控制并被设置为 0.1 。功能块 gr.sig_source_f 的后缀"f"表示其输出的是浮点型（float）数据类型，这样以来情况便变得十分理想，因为音频“漏”接纳在 -1 和 +1 区间变化的浮点类型的采样信号。尽管 GNU Radio 连接功能块时会做一些相应的检查，但是这些事情主要还是由程序员来掌握的。有相当的工作必须手动来完成。比如，如果想把整型的采样信号注入音频“漏”功能块（audio.sink）中去的话，GNU Radio 便回报一个出错信息（error）- 但是在上例中如果设置振幅大于 1 的话，尽管会得到一个变形但并没有出错的信息的回报。

信号“漏”在第 **15** 行被定义：功能块 audio.sink() 的行为正像一个声卡，它可以用来播放任何管流其内的信号。正如该功能块之前描述那样，信号源已经为它设置，但其采样信号频率还是必须明确表示。GNU Radio 无法从情景中推测出正确的采样频率，它也不是流经功能块之间的所应表白的信息。

第 **16** 和 **17** 行用来连接功能块。用来连接功能块的一般句法是 self.connect(block1, block2, block3, ...)，它把功能块 1 的输出同功能块 2 的输入相连、把功能块 2 的输出同功能块 3 的输入相连、以此类推。一个 connect() 的调用可以连接想连接数量的功能块。在此，必须使用一个特殊的句法，来展示所连接的是 src0 和 dst 的第一路输入；src1 和 dst 的第二路输入。self.connect (src0, (dst, 0)) 正是你所想要的：它明确地把 src0 同 dst 的端口 0 相连。(dst, 0) Python 术语称之为元组（tuple）。在 self.connect() 调用它是明确所要连接的端口的编号。当端口编号为 0 时，功能块便可单个被使用。这样，第 16 行便可以如下表示：

```
1 self.connect((src0, 0), (dst, 0))
```

构建一个流程图的工作便完成了。最后 5 行除了启动流程图没有任何别的功用。try 和 except 的简单功用是确保在 Ctrl+C 被键入时停止流程图（不然的话，它便无休止的运行下去）- 它是用来触发 Python 的异常函数 KeyboardInterrupt 。

对于 Python 新手而言，在此强调两个注意点：(第一)也许已经都意识到，类 my_top_block 的使用不需创建实例。在 Python 中，这种用法屡见不鲜，尤其是对于那些仅仅只能创建一个实例的类。当然，也可以创建该类的一或者二个实例，然后通过这些实例来调用函数 run()。

第二，缩进是代码的组成部分，但不是（参与程序功用的）核心部分。仅仅是为了程序员的方便。如果你试图修改此代码，请确保不要把 tabs 键和 space 键相混。每一级都确保一致的缩进。

如果想遵循教程深入下去，坚实的 Python 功底是第一必须。到如下的 Python 站点可以找到相关的或感兴趣的文档或库：

http://www.python.org/ ，

下一站点也许对具有一些程序背景的更有用

http://wiki.python.org/moin/BeginnersGuide/Programmers 。

小结 - Summary

- 必须使用 from gnuradio import 命令导入（import) 所需的 GNU Radio 模块。经常需要用到模块是 gr.
- 流程图被包含在继承于类 gr.top_block 的类函数内。
- 功能块通过调用诸如 gr.sig_source_f() 的函数来构建，其返回值保存在其变量参数中。
- 功能块通过包含流程图的类调用 self.connect() 相互连接在一起。
- 对于编写基本的 Python 代码感到困难的话，休息一下然后先去学习 Python 。

下一部分将会给出用 Python 编写 GNU Radio 应用更详细的概述。

编写 Python 的 GNU Radio 应用

上面的例程已经包含相当部分的如何用 Python 编写 GNU Radio 应用的内容。这一章节和后面的将试图展示 GNU Radio 应用的可能性以及如何使用它们。从现在开始，没有必要顺序一部分一部分读完，通过章节标题来感受那些需要细研。

GNU Radio 模块

GNU Radio 涵盖了相当多的库和模块。通常使用如下句法来导入模块：

```
from gnuradio import MODULENAME
```

某些模块的工作方式稍有不同，请参阅下面的列表最常见的模块。

gr	GNU Radio 主要库函数。 这是总要被用到的库。
analog	所有有关模拟信号和模拟调制的东西。
audio	声卡控制（“源”、“漏”）。 使用它给声卡来发送或接受音频，但是配合外部射频前端声卡只能用作窄带接收机。
blocks	该模块包含额外使用 Python 编写的模块，诸如常用的调制、解调、一些额外的滤波代码、重新采样、压缩等等。
channels	信道模型进行仿真。
digital	任何有关数字调制。
fec	任何有关的前向纠错（FEC）。
fft	任何有关 FFT。
filter	过滤块，像 firdes 和 optfir 设计工具。
plot_data	Matplotlib 绘制数据的一些功能。
qtgui	使用 QT 库的图形化工具来绘制数据（时间，频率，频谱，星座，直方图，时间光栅）。
trellis	用于构建网格，网格编码调制，和其他 FSM 的模块和工具。
vocoder	处理语音编码/解码。

wavelet	处理微波
wxgui	这实际上是一个子模块，内含实用程序来快速创建图形用户界面给你的流图。使用 from gnuradio.wxgui import *导入所有的子模块或者 from gnuradio.wxgui import stdgui2, fftsink2 来导入特定组件。请参见“图形用户界面”获取更多信息。
eng_notation	添加用来处理工程标记的诸如：‘100M’ for 100 * 10^6' 的函数。
eng_options	使用 from gnuradio.eng_options import eng_options 导入此功能。该模块扩展了 python 的 optparse 模块，用来了解工程符号（见上文）。
gru	功用杂类，算术和其它。

这是到目前为止的不完全列表，对模块的描述也不完全有意义。GNU Radio 的代码目前变化太大，所以编制一套静态的文档的时机还不成熟。

取而代之（静态文档），比较鼓励的做法是就像早期版本的星球大战箴言那样“使用资源 - Use the source!”来探索模块的奥妙。如果觉得 GNU Radio 应当具备一些想要的功能，要么去钻研一下 Python 目录所涵盖的模块、要么探究一下 GNU Radio 功能块所包含的源码。特别理应关注源码目录下以 gr- 开头的源码资源，诸如：gr-sounder 或 gr-radar-mono 。这些生成它们自身的代码，及相应的模块。

当然，Python 自身资源（模块）也极其丰富，它们中间的一些，尽管不是十分必要，但对于编写 GNU Radio 应用极其有用。敬请参阅 Python 的文档，以及到 SciPy 的站点寻求更多的资讯。

选择、定义和配置功能块 - Choosing, defining and configuring blocks

GNU Radio 包含极其丰富的预先定义的功能块。但是对于一个新手而言，常常会感到十分困惑的是如何对自己的应用选择合适的功能块并且能够正确的把它们搭配起来。

论及模块， GNU Radio 的代码的变化相当大，因此有关模块的静态文档目前没有现实意义。但是谈到功能块，情况便略有所不同。

下面便是关于文 档的自动生成的问题。它是源码通过 Doxygen 生成的。在此建议该文档在构建系统时一并生成。运行 make 命令的同时，在 ./configure 的命令中添加 --enable-doxygen 命令行选项便可产生此文档。这样的操作很是有益的，其文档包含便于浏览的 HTML 格式的文本。如果不想或没有机会自动生成这些文档，也可以到如下站点（当然不会是最新版本的）去浏览

http://gnuradio.org/doc/doxygen/hierarchy.html.

在 gnuradio-core/doc/html 目录下，可以找到这些自动生成的文档。

学习如何使用这些文档是学习如何使用 GNU Radio 的核心工作！

感受些特殊的。下面这三行代码是摘自前述的例程：

```
1 src0 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350, ampl)
2 src1 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 440, ampl)
3 dst = audio.sink (sample_rate, "")
```

在此简述代码被执行时背后的故事：

第一，模块 gr 的函数 sig_source_f 被执行时。它接纳了四个函数参数：

sample_rate, Python 的变量，

gr.GR_SIN_WAVE, 在模块 ‘gr’被定义的参数，

350, 常数，
ampl, 另外一个变量。

此函数产生一个结果被赋予 src0 的类。同样的故事发生在其它两行，只是“漏”是源于不同的模块（audio）。

此 中如此奥妙！还有一个疑问，如何得知该使用哪个功能块和哪些参数应当被传递到 gr.sig_source_f() 中的呢？这便就是该文档的所扮演的角色。如果你使用的是 Doxygen 生成的文档，点击左上角名称为“Modules"的键。进入到“Signal Sources". 便会发现一列信号发生器，其中包含 sig_source_* 组。后缀在此定义输出的数据类型：

f = float - 浮点型

c = complex float - 复合浮点
i = int - 整型
s = short int - 短整型
b = bits - 位 (实际上整型)

这些后缀的规则对于所有的功能块都适用。比如，gr.fir_filter_ccf() 则定义了一个 FIR 带复合输入、复合输出和浮点抽头的滤波器；而 gr.add_const_ss() 则定义了一个功能块用来叠加短整型输入数值同另一个短整型常数。

点击文档顶部名称为 "Classes" 的键便会得到一个带有简短描述的所有类的列表。那份非官方的 GNU Radio 手册按照模块分类列出所有的类，也可以使用所用 PDF 阅读器的选择性的查询。

你所用的大多数的功能块要么来自 gr，audio 要么来自 usrp 模块。如果能在自动生成文档中发现名称为 gr_sig_source_f 的类的文档的话，便可以在 Python 中生成名字为 gr.sig_source_f() 的类。

很 有必要在此探求一下 GNU Radio 幕后的故事。可以如此自如地在 Python 代码中使用由 C++ 构建的功能块，其原因是 GNU Radio 使用 SWIG 工具来生成 Python 和 C++ 的接口。每个由 C++ 构建的功能块都源自于构建一个名称为 gr_make_*** (在上面的例程中就是 gr_make_sig_source_f())的函数。此函数在被匹配的同时便在相同的页面被自动文档化，而且这也被输出到 Python 。可以这样理解，Python 的 gr.sig_source_f() 函数调用 C++ 的 gr_make_sig_source_f() 。基于同样的原因，它们使用同样的变量 - 这便是为何应当知道如何在 Python 进行功能块中初始化的原因。

浏览 Doxygen 版本的类 gr_sig_source_f 的文档，便会发现很多的表达式，诸如：set_frequency()。这些表达式也被输出到 Python。这样在产生一个信号源时，使用如下方法便可以改变频率（应用便可声称具有频率控制功能）。

```
 # We're in some cool application here
    src0 = analog.sig_source_f (sample_rate, analog.GR_SIN_WAVE, 350, ampl)
# Other, fantastic things happen here

src0.set_frequency(880) # Change frequency
```

（如上代码便）将第一个信号发生器的频率改为 880 Hz.

希望 GNU Radio 文档能够成长并且变得越来越完善。但是，要在细节上想完全理解功能块。或迟或早，必须去查看和理解代码。在这一点上，文本无论是否完善是无法替代的。

连接功能块 - Connecting blocks

使用 gr.top_block 的 connect() 来连接功能块。下面一些事项值得磋商：

1.相同数据类型的输入和输出才可以被相连。如果把浮点型输出同复合型输入相连便会导致错误。

2.一个输出可以同多个输入直接相连，并不需额外的功能块来复制该信号。

3.这些连接功能块的基本规则，基本上适应一般的情况。但一旦所要连接的数据类型被混淆的话，一些额外的注释值得在此表述。

4.GNU Radio 通过检查其大小来检查输入和输出数据类型是否匹配。如果凑巧连接了两个不同的数据类型但它们的数据大小一致，这肯定招致无用的数据。

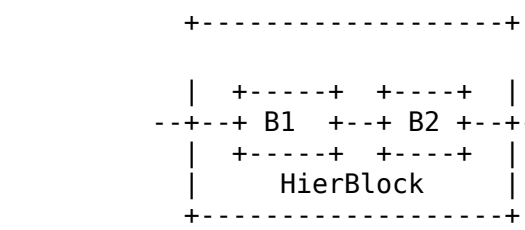
5.在处理单个数据位时，应格外谨慎。有时只是一般意义上的二进制；但有时便需对特定位置（或特定数量）的位进行处理。

也 应格外谨慎应对动态范围问题。对于使用浮点或复合数据类型的情况，它提供了一个足够大的范围让你根本无需考虑任何机器方面的问题，但对于一些“源”和 “漏”必须严守一些规则。比如，音频一类的“漏”要求采样在 +-1 之内，任何大于该范围的便被裁剪掉。另一方面，由于 DAC 的动态范围限制，USRP 作为“漏”其采样被限制在 +-32767 （16 位带符号数值）。

模块的阶层结构 **- Hierarchical blocks**

有时常常需要把几个功能块综合为一个新的功能块。可以这样来理解，有好几个应用都涉及一个通用信号处理部件它是由其它的几个功能块来构建的。这些功能块便可以被综合为一个新功能块，此新构建的功能块便可以被当作一个通用 GNU Radio 功能块被用于应用程序之中。

例子：假如有两个不同的流程图，FG1 和 FG2。两者都使用 - 或相互使用 - 功能块 B1 和 B2。便可以把它们合并成一种阶层结构的功能块 -HierBlock:



这正是应当做的工作：构建一个流程图其衍生（继承于） gr.hier_block2 进而使用 self 作为“源”和“漏”：

```
class HierBlock(gr.hier_block2):

    def __init__(self, audio_rate, if_rate):
        gr.hier_block2.__init__(self, "HierBlock",
                        gr.io_signature(1, 1, gr.sizeof_float),
                        gr.io_signature(1, 2, gr.sizeof_gr_complex))

        B1 = blocks.block1(...) # Put in proper code here!
        B2 = blocks.block2(...)

        self.connect(self, B1, B2, self)
```

如上所展示的，构建一个阶层结构的功能块同构建一个继承于 gr.top_block 的流程图十分相似。当然除了使用 self 作为“源”和“漏”这点之外，另外还有一个不同点：其父类的构建函数（第 3 行）需接收额外的信息。该调用 gr.hier_block2.+init+() 还涉及四个参数：

self （它总是作为第一个参数被传递给构建函数的），

字符串变量用于标记该阶层结构功能块（需要恰当相应地更名），

一个输入签名，

和一个输出签名．

对上面最后的两个参数需一些额外的解释。除非已经写好相关的 C++ 的功能块（第 7，8 行）。否则 GNU Radio 需要知道功能块所用到的输入和输出的数据类型。正如上例展示的，通过调用输入和输出的签名 gr.io_signature() 便可应对该问题。该函数的调用同时也涉及如下三个问题：

端口的最小数量

端口的最大数量

输入/输出元素的大小。

对于阶层结构功能块 HierBlock 而言，不难发现它有一个输入和一个或两个输出。输入到功能块的对象是浮点型 - float，这样功能块便以真实浮点数值来处理输入。在 B1 或 B2 某个位置数据类型被转换成复合浮点型，这样输出签名便声明输出的类的对象是 gr.sizeof_gr_complex 。gr.sizeof_float 和 gr.sizeof_gr_complex 等同 C++ 调用 sizeof() 的返回值。以下是其它数据类型的

gr.sizeof_int

gr.sizeof_short

gr.sizeof_char

使用 gr.io_signature(0, 0, 0) 便可以产生一个 null 类型的 IO 签名（signature），比如，（这种方法）便可以用来定义阶层结构的功能块作为“源”或“漏”。

工作到此告一段落，现在便可把 HierBlock 作为一个通常功能块来使用。如下例所展示，它是如何被用在同上面相同的例程中。

```
class FG1(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self)

    ... # Sources and other blocks are defined here
    other_block1 = blocks.other_block()
    hierblock    = HierBlock()
    other_block2 = blocks.other_block()

    self.connect(other_block1, hierblock, other_block2)

    ... # Define rest of FG1
```

当然，模块化的使用 Python，还应把 HierBlock 的代码放入另外一个名称为 hier_block.py 的文件。从另外一个文件来使用该功能块，简单的做法是通过导入标识符（import）的使用来实现：

```
      from hier_block import HierBlock
```

这样便可以如上使用 HierBlock。

并行流程图 - Multiple flow graphs

有时，需要完全分离（并行的）的流程图。比如，发射和接收链路（如上的'Walkie-Talkie' 例程）。目前（2008 年 6 月），并行运行顶层功能块 - top_block 还不现实。但是可以使用 gr.hier_block2 如上所述，借用阶层功能块的概念。来生成一个 top_block 来持有如下例程所构建的流程图：

```
class transmit_path(gr.hier_block2):
    def __init__(self):
                gr.hier_block2.__init__(self, "transmit_path",
                                gr.io_signature(0, 0, 0), # Null signature
                                gr.io_signature(0, 0, 0))

        source_block = blocks.source()
        signal_proc  = blocks.other_block()
        sink_block   = blocks.sink()

        self.connect(source_block, signal_proc, sink_block)

class receive_path(gr.hier_block2):
    def __init__(self):
                gr.hier_block2.__init__(self, "receive_path",
                                gr.io_signature(0, 0, 0), # Null signature
                                gr.io_signature(0, 0, 0))

        source_block = blocks.source()
        signal_proc  = blocks.other_block()
        sink_block   = blocks.sink()

        self.connect(source_block, signal_proc, sink_block)

class my_top_block(gr.top_block):
    def __init__(self):
        gr.top_block.__init__(self)

    tx_path = transmit_path()

    rx_path = receive_path()

    self.connect(tx_path)
    self.connect(rx_path)
```

这样一来，只要启动 my_top_block，两个流程图便被并行运行。不难注意到此阶层结构的功能块明示没有输入和输出，是一个 null 类型的 IO 签名。这样一来导致的结果是没有把 self 作为“源”和“漏”而同它相连。相反，它们各自定义自己的“源”和“漏”（就像定义阶层结构功能块作为“源”或“漏”一样）。 顶层功能块简单地把阶层结构功能块连向自身，这样它们便没有被连接起来。

GNU Radio 的扩展和工具 - GNU Radio extensions and tools

GNU Radio 不只是包含一些功能块和流程图 - 它还包含很多的工具和代码用来帮助编写 DSP 应用。

在 gr-utils/. 下便可以发现一整套设计用来帮助构建 GNU Radio 应用的应用。

到 gnuradio-core/src/python/gnuradio 通过浏览源代码来寻找有用的应用，诸如：滤波器设计代码、调制应用以及更多。。。

流程图的控制 **- Controlling flow graphs**

到目前为止如果一直沿着教程的话，便不难发现流程图总是以类的形式存在，并继承于 gr.top_block。这样一来，问题便呈现出来：如何来控制这些类？

前面已经提到，从 gr.top_block 继承下来的类包含了其中的所有可能被用到的函数。这样便可以使用如下一些方法（函数）来运行和停止流程图：

run()	最简单的方法来运行一个流程图的方法是：先调用 start()再 wait()。一般地启动一个流程图，直到它自行停止。或者永久性的运行下去直到接收到 SIGINT 后（停止）。
start()	启动一个内嵌的流程图。线程（threads）一产生便返回调用者。
stop()	停止正在运行的流程图。值得注意的，该线程由调度生成来关闭流程图，然后返回调用者。
wait()	等待流程图完结。流程图的在如下两种情况下完结： (1) 所有的功能块表白任务被完结，(2) 通过调用 stop 来关闭流程图。
lock()	闭锁流程图来准备被重新配置。
unlock()	在准备被重新配置的过程中开锁。一旦同等数量的闭锁 - lock() 和开锁 unlock() 的调用发生的话，流程图将会自动被重新启动。

详细信息请参阅 gr_top_block 功能块的文档。

例子：

```
class my_top_block(gr.top_block):

    def __init__(self):
        gr.top_block.__init__(self)
    ... # Define blocks etc. here

if __name__ == '__main__':
    my_top_block().start()
    sleep(5) # Wait 5 secs (assuming sleep was imported!)
    my_top_block().stop()
    my_top_block().wait() # If the graph is needed to run again, wait() must be
called after stop
    ... # Reconfigure the graph or modify it
    my_top_block().start() # start it again
    sleep(5) # Wait 5 secs (assuming sleep was imported!)
    my_top_block().stop()  # since (assuming) the graph will not run again, no
need for wait() to be called
```

这些方法（函数）可以帮助你从外部来控制流程图。但 对好多问题这是不够的：不能只是简单地来启动或停止流程图，最好是能通过重新配置来改变其行为。比如，设想应用中包含一个音量控制器它在流程图的某个地 方。这个音量控制器通过在采样数据流中插入乘法器来实现。这个乘法器是 gr.multiply_const_ff 类型。如果查阅此功能块的文档，便会发现函数 gr.multiply_const_ff.set_k() 是用来设置乘法系数的。

应当使此设置从外部可视这样便可以控制它。简单的做法是把功能块作为流程图的类的属性。

例子：

```
class my_top_block(gr.top_block):
```

```
    def __init__(self):
        gr.top_block.__init__(self)
    ... # Define some blocks
    self.amp = blocks.multiply_const_ff(1) # Define multiplier block
    ... # Define more blocks

        self.connect(..., self.amp, ...) # Connect all blocks

    def set_volume(self, volume):
    self.amp.set_k(volume)

if __name__ == '__main__':
    my_top_block().start()
    sleep(2) # Wait 2 secs (assuming sleep was imported!)
    my_top_block.set_volume(2) # Pump up the volume (by factor 2)
    sleep(2) # Wait 2 secs (assuming sleep was imported!)
    my_top_block().stop()
```

此例程运行流程图 2 秒钟，然后通过名称为 set_volume() 的成员函数读取功能块 amp 来倍增音量。当然也可以忽视该成员函数直接访问属性功能块 amp。

提示：把功能块作为流程图的类的属性是一个很好的观念，它使得通过成员函数来扩展流程图变得更加容易。

非流程图框架的应用 - Non-flow graph centred applications

到目前为止，此教程的 GNU Radio 的应用都是围绕着这个从 gr.top_block 的类继承下来的类为中心来阐述的。尽管如此，这并不表示 GNU Radio 必须被这样来使用。GNU Radio 被设计成使用 Python 作为工具来开发 DSP 应用， 这样便理应尽 Python 之能来尽 GNU Radio 之用。

Python 的功能异常强大，而且新的库和函数在持续性的增加。同样的 GNU Radio 也以强大的、实时能力的 DSP 库来扩展 Python。轻轻地动一下手指，便可以把这些库想结合让你拥有庞大的函数库。比如，把 GNU Radio 同 SciPy ， 一个 Python 的科学函数库集合，相结合便可实时采集射频信号然后进行十分强大离线数学运算、把统计结果存储到数据库等等。这一切都可以在同一应用上完成。如果把这些库 都结合在一起，诸如 Matlab 之类的昂贵的工程软件也可能变得多余了。

高级主题 - Advanced Topics

如果仔细的研读了前面的部分，这已经足以能开始编写一个 GNU Radio 的 Python 应用程序。这部分将阐述一些 GNU Radio 的 Python 应用程序的略微高深的话题。

动态流程图的生成 - Dynamic flow graph creation

一般而言，前述构建流程图的方法足以解决大多数问题。但想要应用更加灵活，所要做的便是从（正在构建的）类的外部对流程图进行更多的控制。

这可以通过把函数+init+()的代码提取到外部，然后简单地把 gr.top_block 当作容器来使用

```
    ... # We are inside some application
```

```
tb = gr.top_block() # Define the container

block1 = blocks.some_other_block()
block2 = blocks.yet_another_block()

tb.connect(block1, block2)

... # The application does some wonderful things here

tb.start() # Start the flow graph

... # Do some more incredible and fascinating stuff here
```

如果想编写一些需要动态停止流程图（便于重新配置、重新启动等等）。此法不失是一个可行的选择。

相关例程：

`gnuradio-examples/python/apps/hf_explorer/hfx2.py`

命令行选项 - Command Line Options

Python 有一些用来语法分析（parse）命令行选项的库。参阅模块 optparse 的文档来查看如何使用它。

GNU Radio 扩展了 optparse 的命令行选项。使用 from gnuradio.eng_option import eng_option 来导入这种扩展。籍助 eng_option 扩展了如下内容：

eng_float	同原始的 float 选项一样，但它接纳工程标记诸如：101.8M
subdev	仅接纳有效的子设备描述符诸如 A:0 (用来表述某个 USRP 上的某个子板)
intx	仅接纳整型

应用程序一旦支持命令行选项的功能，这样该应用便可以遵从 GNU Radio 固有的命令行的习惯。在文档 README.hacking 里可以找到同这些（还有更多为开发者的建议）相关的内容。

现实中几乎每个 GNU Radio 的例程都利用了该特性。参阅一下 dial_tone.py 便对此一目了然。

用户图形界面 - Graphical User Interfaces

正如前面多次提到的，GNU Radio 仅仅延伸了 Python 的 DSP 功能而已。这样编写 GNURadio 的 GUI 应用时，只管先编写一个纯粹的 GUI 应用、然后再把流程图添加上、随后再定义一些接口来把 GNU Radio 的信息传递到此应用中；反之亦然。最后想输出图形的话，使用 Matplotlib 或者 Qwt 即可。

然而，有时只需简单快速的编写一个带有 GUI 的 GNU Radio 应用而不必繁琐地去配置 widgets、以及定义所有的菜单等等。为此 GNU Radio 本身包含了一些预定义的类可以用来帮助快速编写图形化的 GNU Radio 应用。

这些模块是基于 wxWidgets（准确地说，wxPython），一套与平台无关的 GUI 工具包。掌握它还是需要一些 wxPython 的背景知识 - 但不要担心，首先它不是多么复杂

的工作而且网络上也有几个教程。到如下 wxPython 的网站可以找到一些文档。

http://www.wxpython.org/

首先需要导入一些模块，这样才能够使用 GNU Radio 的 wxWidget 工具：

```
from gnuradio.wxgui import stdgui2, fftsink2, slider, form
```

上面的例程展示，从子模块 gnuradio.wxgu 中导入了 4 个部件。下面列单一个简单的模块清单（再强调一下，这不是全部的模块清单。在 gr-wxgui/src/python 目录下便可浏览更多的模块或源码）。

stdgui2	基本的 GUI，这也是经常被用到的部分
fftsink2	绘制数据的 FFTs 波形图用来进行频谱分析之类
scopesink2	示波器输出
waterfallsink2	瀑布式输出 - Waterfall output
numbersink2	显示输入数据的数字量
form	常常使用的输入形式

如下是一个新的流程图的定义的代码块。在此流程图中，被定义的类不是继承于 gr.top_block 而是 stdgui2.std_top_block：

```
class my_gui_flow_graph(stdgui2.std_top_block):

        def __init__(self, frame, panel, vbox, argv):
                stdgui2.std_top_block.__init__ (self, frame, panel, vbox, argv)
```

正如此例程中所展示的，还有一点不同：构建函数中包含了一些新的参数。这是因为 stdgui2.std_top_block 不光包含流程图的函数（它们是从 gr.top_block 继承下来的），也包含直接生成一个窗口所需的基本元素（如：菜单等）。对于那些仅需快速的图形应用的开发者这是个好消息：GNURadio 可以生成窗口及相关的一切，所要做的只是把插件（widgets) 添加进去便可。下面也给出一个清单来简单的描述这些新的对象的功用（如果对于 GUI 编程不了解的话，这些概念可能会感觉很晦涩）：

frame	框架（frame），新窗口的 wx.Frame。使用 frame.GetMenuBar() 可以得到一个预先定义好的菜单。
panel	面板（panel），位于框架‘frame’之中，用于包容所有的 wxControl widgets。
vbox	一个垂直的对象 box sizer (wx.BoxSizer(wx.VERTICAL) 如此被定义), 用来在 panel 内垂直对齐 widgets。
argv	命令行命令的参数

现在构建 GUI 的万事具备。只要简单地把新的 box sizers 和 widgets 对象添加到 vbox，便能改变菜单等等诸如此类。而且 GNU Radio 的 GUI 的库 form 还把一些常规的功能进行了进一步的简化。

form 具有丰富数量的输入插件 widgets: form.static_text_field() 用于

静态文本类 (仅用于显示)、form.float_field() 用于浮点型数值输入、form.text_field() 用于文本输入、 form.checkbox_field() 用于 checkboxes、form.radiobox_field() 用于 radioboxes 等等。 在 gr-wxgui/src/python/form.py 下查看所有这些的代码。这些中的大多数调用把参数传递到相关的 wxPython 对象中，而且函数的参数命名含义十分明示它的含义。

下面的例程展示如何使用 form 添加插件（widgets）。

gnuradio.wxgui 最有用的部分大概就是可以直接描绘输入的数据。实现这需要使用以 gnuradio.wxgui 为输入的一个“漏”诸如： fftsink2 。这些“漏”的行为同 GNU Radio 的其它“漏”没有什麽不同。也具有使用 wxPython 所需的特性。例子：

```
from gnuradio.wxgui import stdgui2, fftsink2


# App gets defined here ...

# FFT display (pseudo-spectrum analyzer)
my_fft = fftsink2.fft_sink_f(panel, title="FFT of some Signal", fft_size=512,
             sample_rate=sample_rate, ref_level=0, y_per_div=20)
self.connect(source_block, my_fft)
vbox.Add(my_fft.win, 1, wx.EXPAND)
```

首先，定义功能块(fftsink2.fft_sink_f)。除典型的 DSP 参数诸如：采样速率之外，还需把 panel 对象传递到构建函数。其次，把功能块同“源”相连。最后，把 FFT 窗口 (my_fft.win)置入 vbox BoxSizer 来显示其值。请记住，信号功能块的输出可以同任何数量的输入相连。

最后，所作的一切便可以被启动。运行 GUI 需要 wx.App()，这一点不同于传统的流程图：

```
if __name__ == '__main__':

    app = stdgui2.stdapp(my_gui_flow_graph, "GUI GNU Radio Application")
    app.MainLoop()
```

stdgui2.stdapp() 使用 my_gui_flow_graph (第一个参数)来产生 wx.App。窗口的标题也被设置为 "GUI GNU Radio Application"。

一些简单的 GNU Radio GUIs 的例程：

gr-utils/src/python/usrp_fft.py
gr-utils/src/python/usrp_oscope.py
gnuradio-examples/python/audio/audio_fft.py
gnuradio-examples/python/usrp/usrp_am_mw_rcv.py