

最全 Pycharm 教程

转自山在岭就在博客 <http://blog.csdn.net/u013088062/article/category/5935163>,之间花了一周多的时间把 Pycharm 官方帮助文档翻译了一遍,一共 43 篇博客,累得要屎,感悟颇多。发牢骚之前先总结点干货,这里把所有的翻译文档列成如下目录,方便大家索引:

[最全 Pycharm 教程 \(1\) ——定制外观](#)

[最全 Pycharm 教程 \(2\) ——代码风格](#)

[最全 Pycharm 教程 \(3\) ——代码的调试、运行](#)

[最全 Pycharm 教程 \(4\) ——有关 Python 解释器的相关配置](#)

[最全 Pycharm 教程 \(5\) ——Python 快捷键相关设置](#)

[最全 Pycharm 教程 \(6\) ——将 Pycharm 作为 Vim 编辑器使用](#)

[最全 Pycharm 教程 \(7\) ——虚拟机 VM 的配置](#)

[最全 Pycharm 教程 \(8\) ——Django 工程的创建和管理](#)

[最全 Pycharm 教程 \(9\) ——创建并运行一个基本的 Python 测试程序](#)

[最全 Pycharm 教程 \(10\) ——Pycharm 调试器总篇](#)

[最全 Pycharm 教程 \(11\) ——Pycharm 调试器之断点篇](#)

[最全 Pycharm 教程 \(12\) ——Pycharm 调试器之 Java 脚本调试](#)

[最全 Pycharm 教程 \(13\) ——Pycharm 部署](#)

[最全 Pycharm 教程 \(14\) ——Pycharm 编辑器功能总篇](#)

[最全 Pycharm 教程 \(15\) ——Pycharm 编辑器功能之自动生成格式](#)

[最全 Pycharm 教程 \(16\) ——Pycharm 编辑器功能之代码自动生成](#)

[最全 Pycharm 教程 \(17\) ——Pycharm 编辑器功能之自动导入模块](#)

[最全 Pycharm 教程 \(18\) ——Pycharm 编辑器功能之代码拼写提示](#)

[最全 Pycharm 教程 \(19\) ——Pycharm 编辑器功能之代码折叠](#)

[最全 Pycharm 教程 \(20\) ——Pycharm 编辑器功能之模板应用](#)

[最全 Pycharm 教程 \(21\) ——Pycharm 编辑器功能之代码快速修改](#)

[最全 Pycharm 教程 \(22\) ——Pycharm 编辑器功能之窗口选项卡管理](#)

[最全 Pycharm 教程 \(23\) ——Pycharm 编辑器功能之代码高亮显示及错误提示机制](#)

[最全 Pycharm 教程 \(24\) ——Pycharm 编辑器功能之宏定义](#)

[最全 Pycharm 教程 \(25\) ——Pycharm 编辑器功能之查看帮助文档](#)

[最全 Pycharm 教程 \(26\) ——Pycharm 搜索导航之文件名、符号名搜索](#)

[最全 Pycharm 教程 \(27\) ——Pycharm 搜索导航之跳转到声明与定义](#)

[最全 Pycharm 教程 \(28\) ——Pycharm 搜索导航之搜索应用实例](#)

[最全 Pycharm 教程 \(29\) ——再探 IDE, 速成手册](#)

[最全 Pycharm 教程 \(30\) ——Pycharm 中的 File Watchers](#)

[最全 Pycharm 教程 \(31\) ——Pyhcarm 实战](#)

[最全 Pycharm 教程 \(32\) ——根据 FHS 在 Linux 上安装 Pycharm](#)

[最全 Pycharm 教程 \(33\) ——使用 Pycharm 编写 IPython Notebook 文件](#)

[最全 Pycharm 教程 \(34\) ——Pycharm 内置终端以及远程 SSH 工具的使用](#)

[最全 Pycharm 教程 \(35\) ——Pycharm 中使用 Vagrant](#)

[最全 Pycharm 教程 \(36\) ——Pycharm 中 Vagrant 高级技巧](#)

[最全 Pycharm 教程 \(37\) ——Pycharm 版本控制之基础篇](#)

[最全 Pycharm 教程 \(38\) ——Pycharm 版本控制之远程共享](#)

[最全 Pycharm 教程 \(39\) ——Pycharm 版本控制之本地 Git 用法](#)

[最全 Pycharm 教程 \(40\) ——Pycharm 扩展功能之捆绑插件 TextMate](#)

[最全 Pycharm 教程 \(41\) ——Pycharm 扩展功能之便签注释](#)

[最全 Pycharm 教程 \(42\) ——Pycharm 扩展功能之 Emacs 外部编辑器](#)

[最全 Pycharm 教程 \(43\) ——Pycharm 扩展功能之 UML 类图使用](#)

一、不妥之处

说实话这个教程翻译得还是有一些不完善的地方，主要有以下几点：

1、教程中的跳转链接没有处理好

在翻译过程中只是按照目录翻译了官方文档的内容，至于文档内部的链接地址则没有仔细处理，都是直接跳转到了原有的英文原版的链接界面。对于教程各个篇章之间的链接跳转也没有做详细处理，也是直接跳到了原版。没办法，因为链接实在太多太复杂了，大家直接从这个目录上进入到对应模块吧。

2、部分名词未进行翻译

由于译者经验有限，在加上这里观法给出的帮助文档所涉及的范围实在是太广，以致某些专有名词译者之前并没有接触过，有道、谷歌等翻译机构给出的译文又不尽合理，因此选择了保留原有名词，望相关领域的大牛给予帮助。

3、解释性语言经过适当改动

教程的前半部分一般都是严格按照官方文档的内容进行翻译，包括以下幽默的引用、辅助性的提问等等。在后半部分教程，尤其是后十篇，由于期间回了趟家，课题安排又比较紧，所以加快了翻译进度。对于一些重复性的解释、意义不大的提问等等都予以省略，但主要的操作解释部分的说明依然是完全按照官方文档的说明进行了直译，避免发生歧义。

二、注意事项

1、速成手册

不得不说，这个教程规模实在是有点庞大，不过其中有两篇是独立的速成手册，分别是[最全 Pycharm 教程 \(29\) ——再探 IDE, 速成手册](#)和[最全 Pycharm 教程 \(31\) ——Pycharm 实战](#)，如果你希望尽快掌握 Pycharm 的基本功能，建议直接阅读这两篇教程即可。

2、版本问题

Pycharm 目前已经更新到了 5.0 版，相对于之前的 2.x、3.x 版本，在界面设计上有了较大改变，所以在实际操作中可能会出现选项卡找不到的情况。不过 Pycharm 提供了设置搜索功能，在不能顺利某个设置页时，可以直接在设置对话框的搜索栏中输入对应设置页的名称来进行定位。

最全 Pycharm 教程（1）——定制外观

Pycharm 作为一款强力的 Python IDE, 在使用过程中感觉一直找不到全面完整的参考手册, 因此决定对官网的 Pycharm 教程进行简要翻译, 与大家分享。

1、准备工作

首先确定你安装了 2.7 或者更高版本的 Pycharm ([下载地址](#)), 下载时请根据你的系统平台下载对应版本即可。

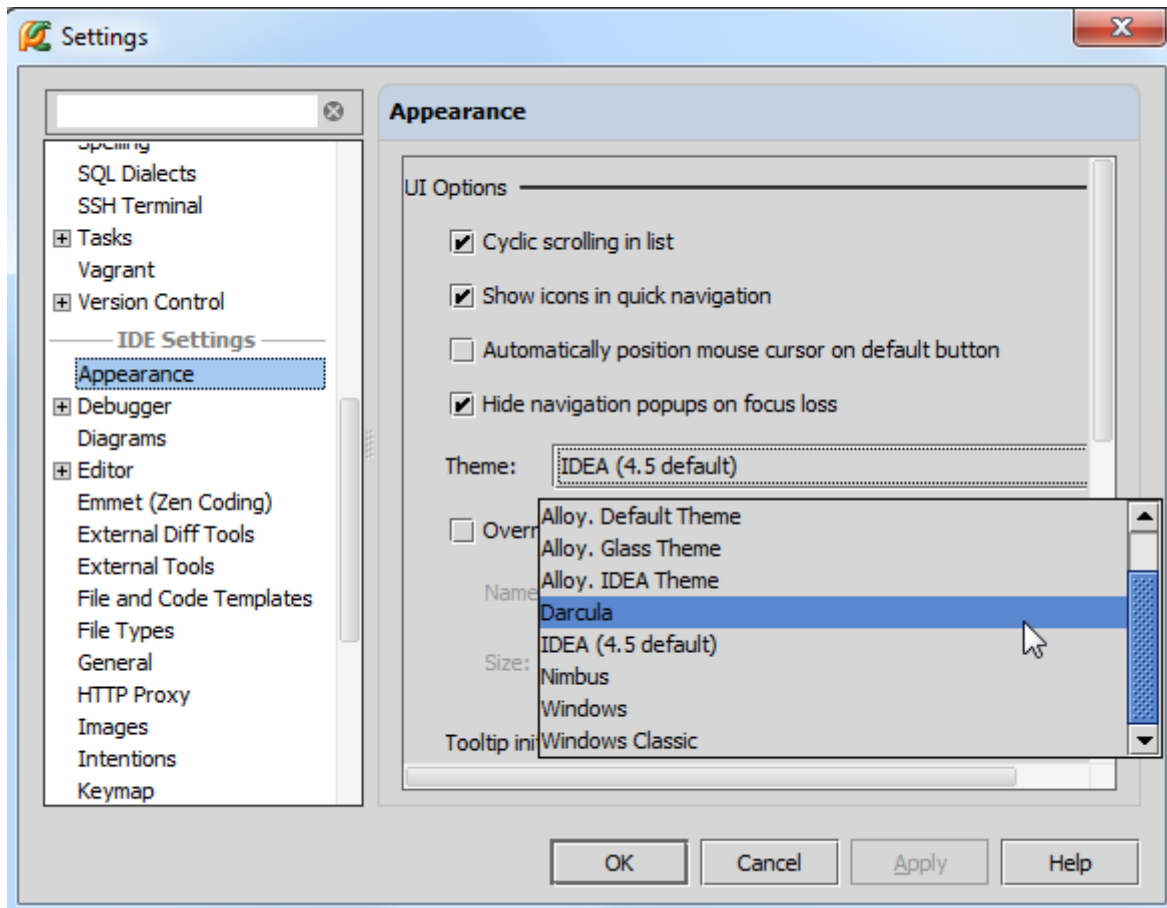
2、如何选择 Pycharm 的外观

Pycharm 预定义了几种主题模式, 可用主题的数量与操作系统类型有关, 你可以参照外观说明 ([参照说明](#)) 在 “Settings/Preferences” 对话框中进行相关设置。

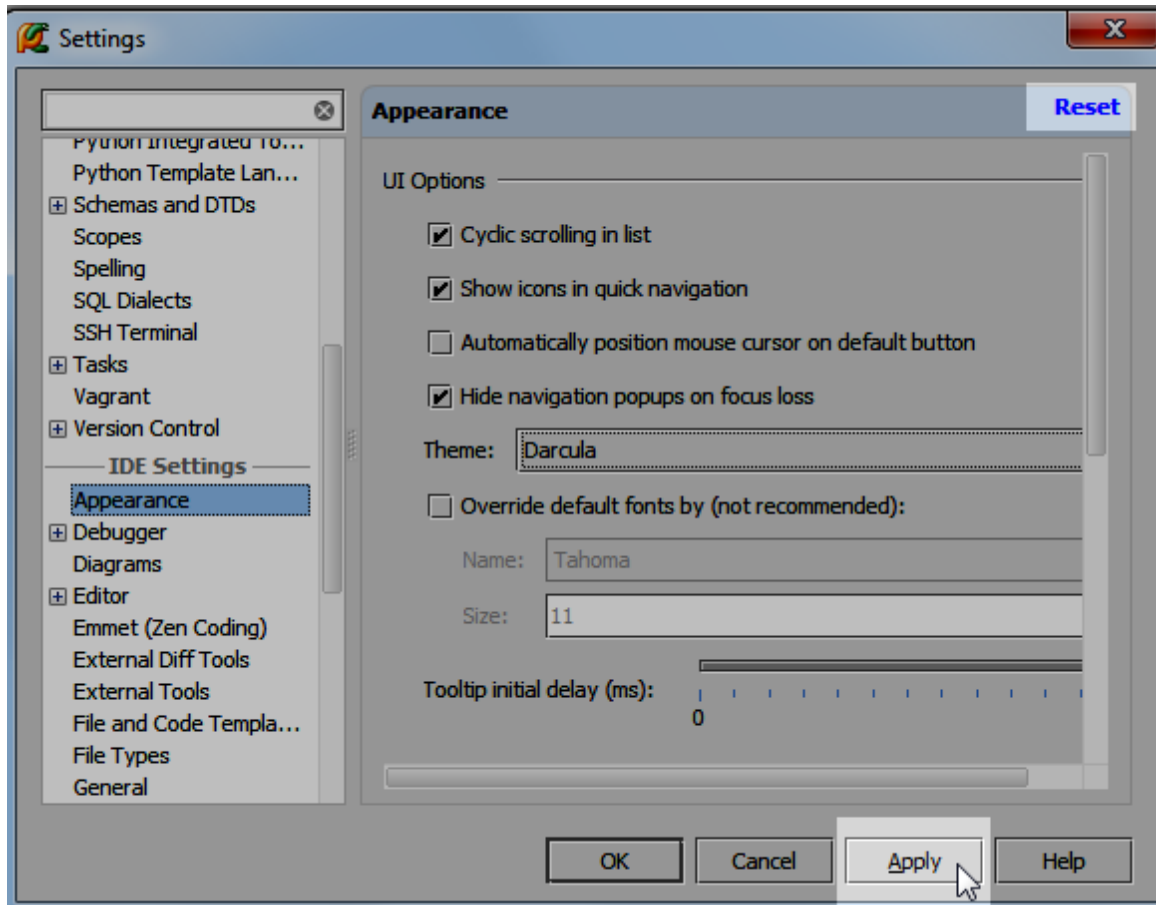
背景主题的具体设置方法如下:

(1) 在主工具栏中, 单击  来打开 “Settings/Preferences”, 然后单击 [参照说明](#):

(2) 在下面的对话框中, 单击 “Theme” 对应的下拉菜单, 然后选择一个你喜欢的主题:

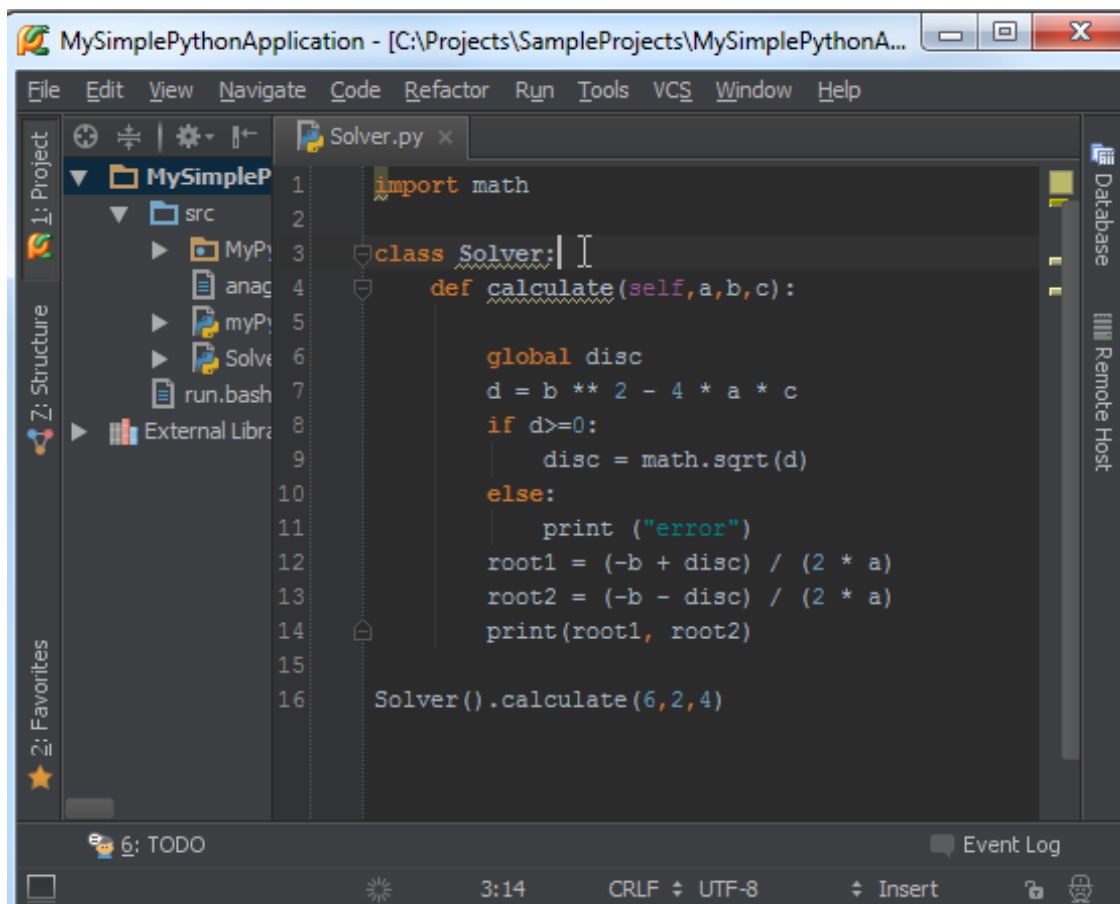


注意此时位于对话框右上角的 Reset 按钮, 如果你改变了注意, 可以通过单击这个按钮来恢复之前的设置。同时当你将鼠标移动至 Apply 按钮时, 它将变为可用状态:



当然你也可以更改其他外观设置，例如字体和字号、窗口属性等。

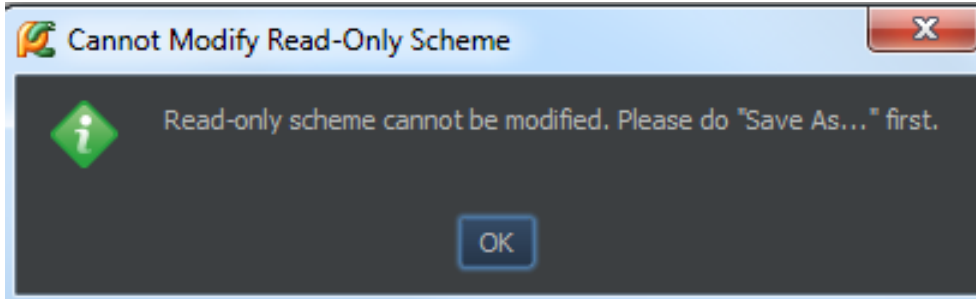
3、应用更改设置，建议重启 Pycharm 软件（例如当你将主题改为 Darcula 时，冲击之后将是下面这种效果）：



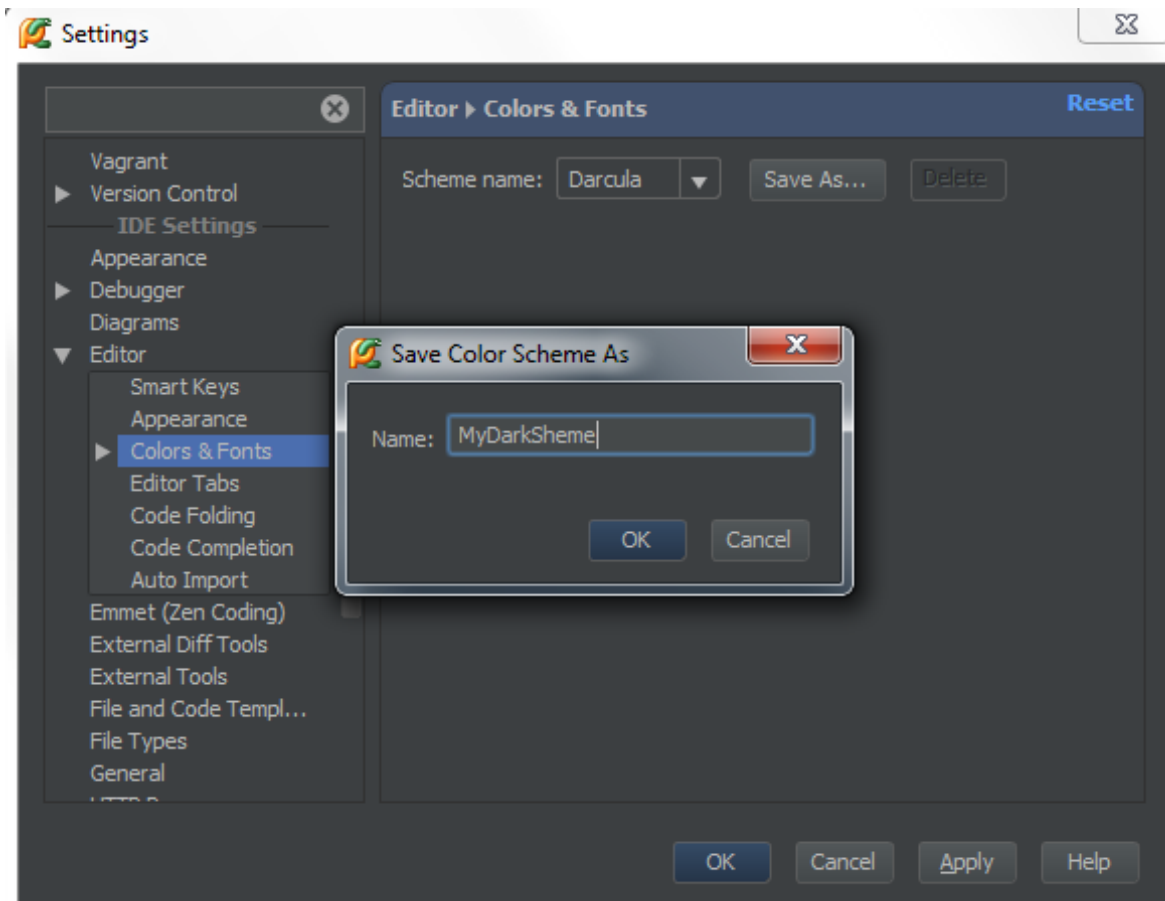
4、如何更改编辑框的主题颜色

在更改完 Pycharm 的主题背景之后,你可能对编辑器的外观仍不满意,例如你希望将文档字符串改变为另外一种颜色,下面介绍具体更改方法:

首先,我们再次单击工具栏上的  图标进入“Settings/Preferences”对话框,展开“Editor”节点,然后单击“Color and Font”,发现系统提示我们当前并不能更改预定义的字体主题,也就意味着我们必须先要对它复制一份:

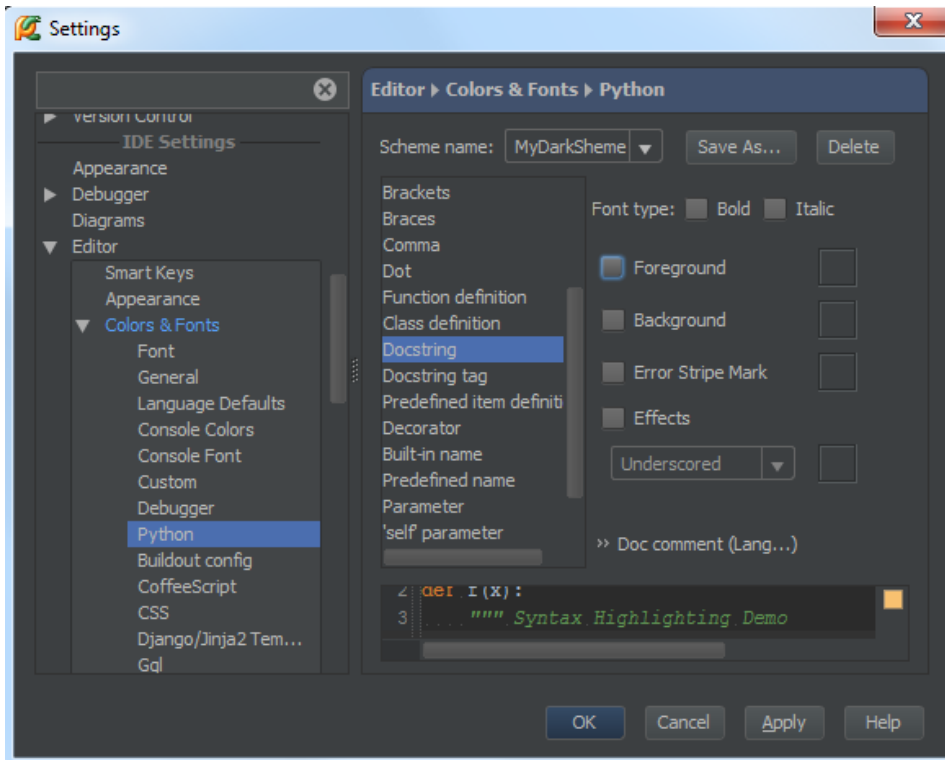


单击“Save as”按钮,然后键入一个新的字体框架名称:



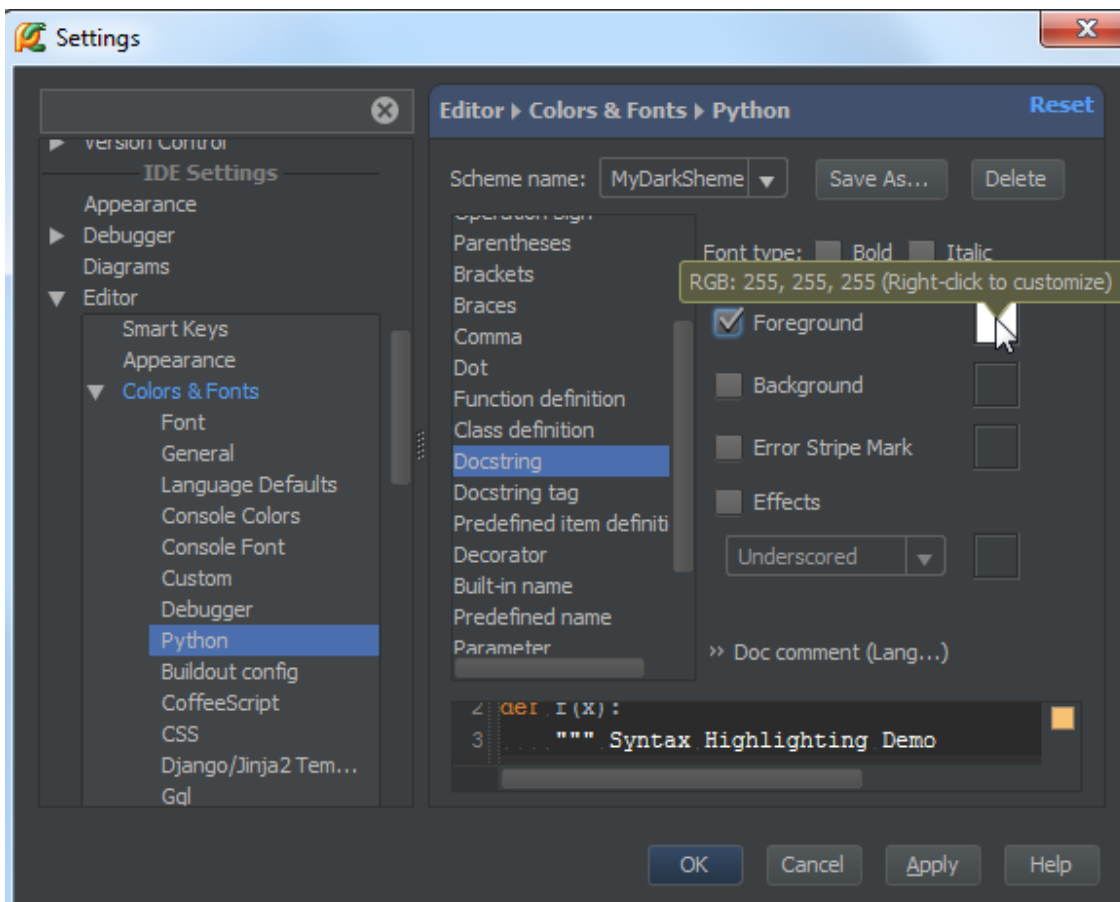
现在我们发现我们所新建的字体类型变为可编辑状态,我们可以根据自己的喜好对它进行修改。

展开“Color and Font”节点,进入编辑器设置对话框:



首先，在语言空间的下拉列表中，单击选择当前文档字符串的字体类型，通过预览窗口我们可以简单预览当前设置的视觉效果。

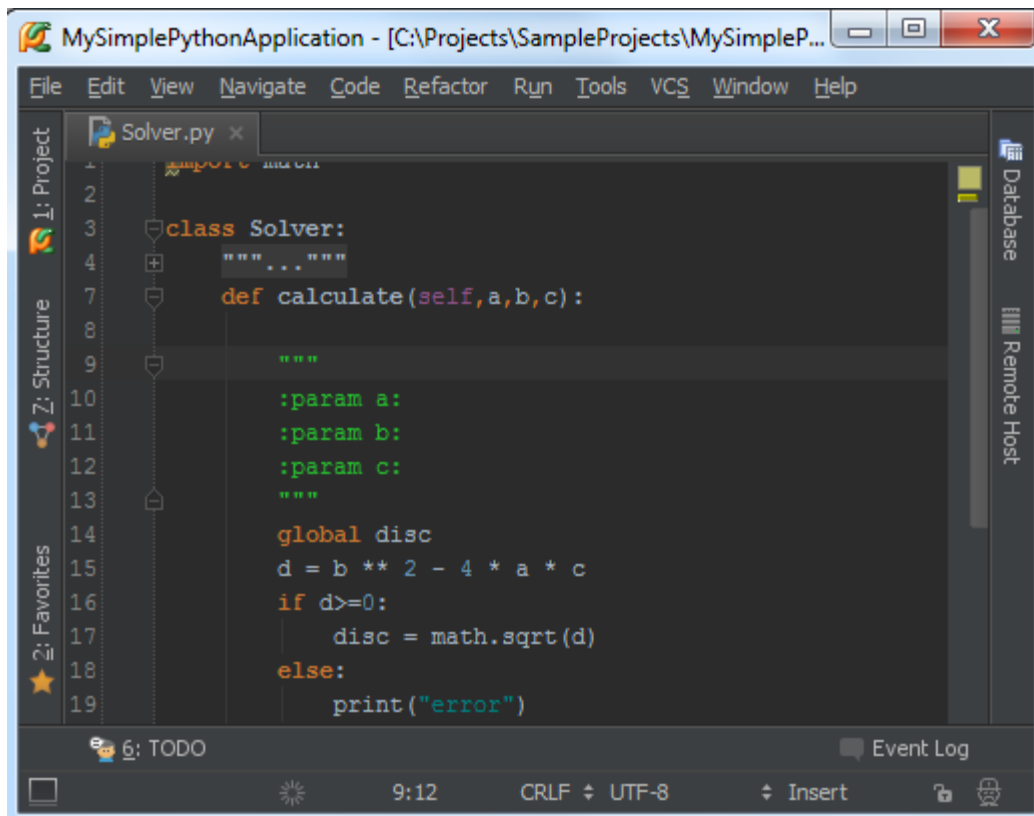
值得一提的是，如果你想将字体颜色由深绿色改为其他颜色的话，只需勾选“Foreground”复选框，然后右击或者双击右侧的颜色块：



接下来再弹出的调色板中选择一种颜色：

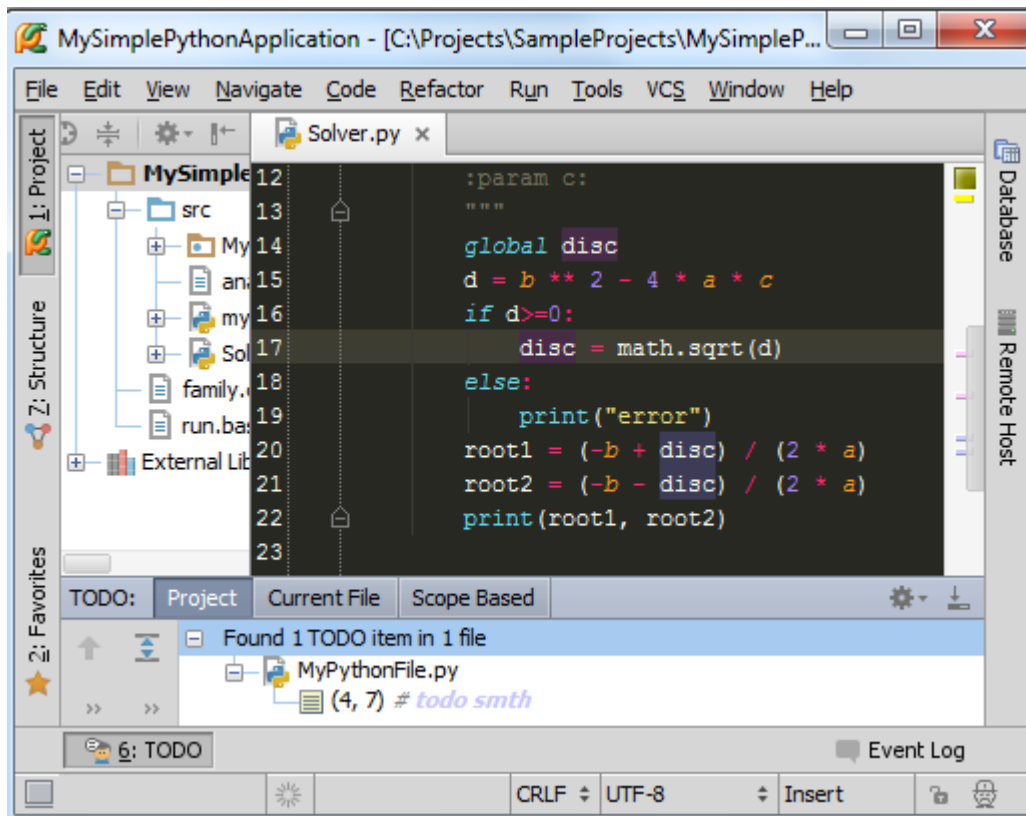


OK，应用设置，然后我们发现我们选中的颜色成功用于字体的显示：



5、软件主题和编辑框字体主题的区别

留意这两个主题之前的区别，前者是于整个软件相关的，而后者只是应用于编辑框部分的属性更改，我们完全可以将这个 IDE 的主题设置为亮色（例如 Default or Alloy）同时将编辑框设置为一个深色主题（例如 Twilight or Monokai），这样的效果就是 Pycharm 的控件都显示为亮色，而编辑窗口显示为暗色：



最全 Pycharm 教程（2）——代码风格

1、主题

这部分教程主要介绍如何创建一个 Python 工程并使其具有 Pycharm 的代码风格。你将会看到 Pycharm 使你的源码变得非常简洁美观，带有合适的缩进、空格等等，因此 Pycharm 也是一款代码质量管理的利器。

这部分教程并不会介绍如何使用 Python 进行编程，更多有关 Python 编程的知识请参照：[Python 编程](#)

2、准备工作

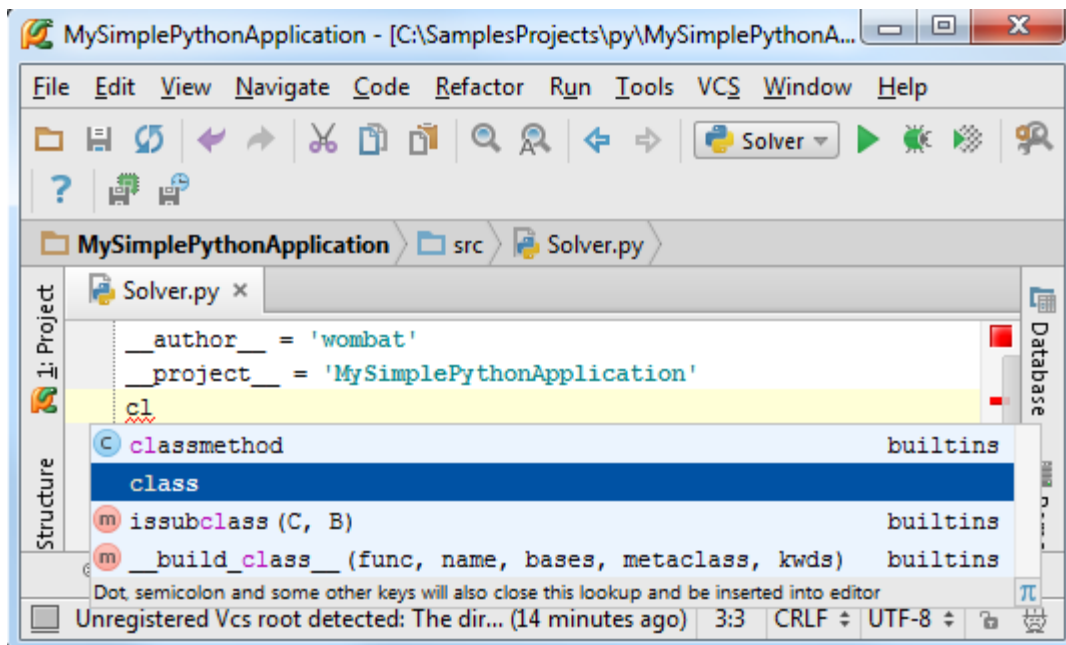
在开始之前，请确认一下情况：

- (1) 安装了 Pycharm2.7 或更高版本的软件
- (2) 已经新建了一个 Python 工程（File→New Project），详情参照：[Pycharm 新建工程文件](#)
- (3) 已经在工程下添加了两个目录：src 和 test_dir（File→New or Alt+Insert），详情参照：[Pycharm 新建工程文件](#)
- (4) 已经向工程目录下添加了对应的 Python 文件（File→New or Alt+Insert），详情参照：[Pycharm 新建工程文件](#)

3、代码报错的高亮模式

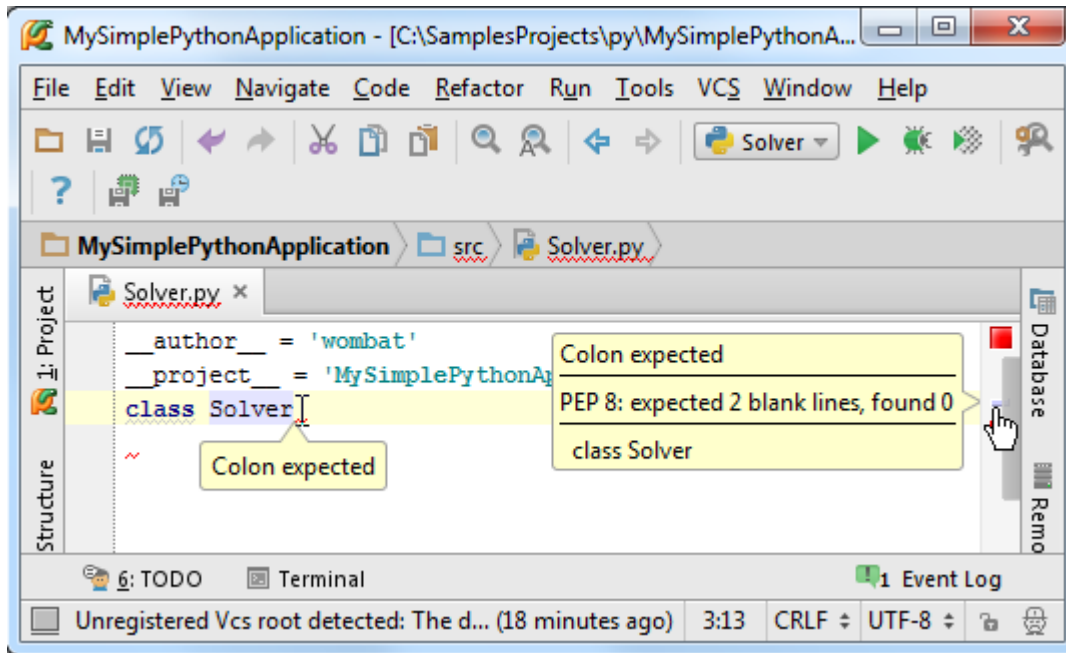
打开一个新建的 Python 文件进行编辑（F4），这个文件中默认有两行代码：作者姓名和工程名称。之所以会出现这两行代码，是因为 Python 文件在创建时是基于[文件模板](#)进行创建的，因此会预定义这两个变量。

接下来输入关键字 class，当你开始输入时，Pycharm 的拼写提示机制会立即列出候选项来帮助你完成代码：




（参照 [Pycharm 拼写提示](#) 来了解 Pycharm 更多关于拼写提示的信息）

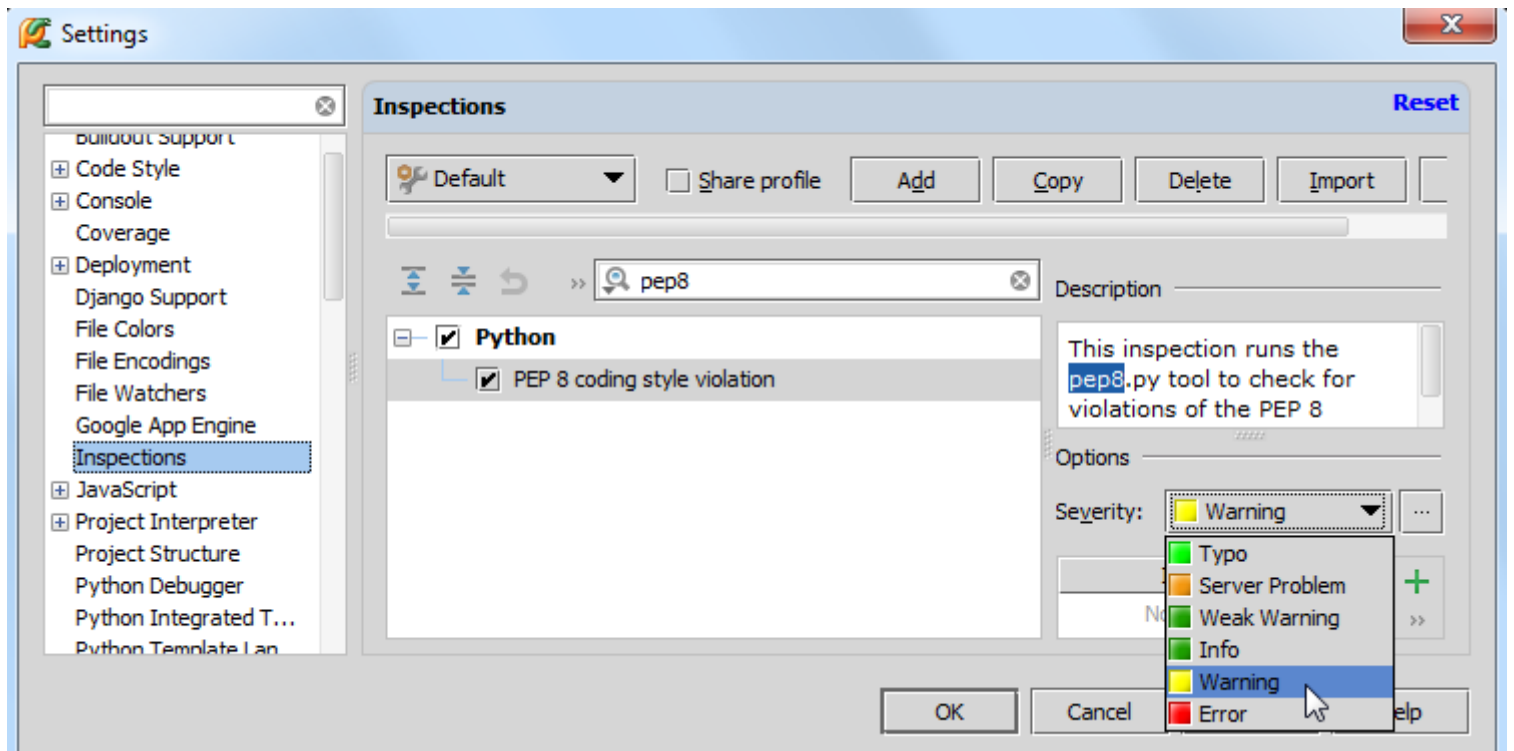
这个红色波浪线标记了下次代码输入的期望位置，在这种情况下，它是一个预输入定义符。键入类名 Solver，红色波浪线将会移动到类名之后。如果你将鼠标指针悬停在波浪线上，将会看到所提示的错误信息（"Colon expected"），当然，此时位于右侧滚动栏的红色标志也会给出相同的错误信息。



OK，输入冒号，回车。根据 [Python 代码风格标准](#)，需要定义下一个类声明，当然此时我们可以通过输入空格来取消它。

4、聚焦 PEP8 代码风格检查

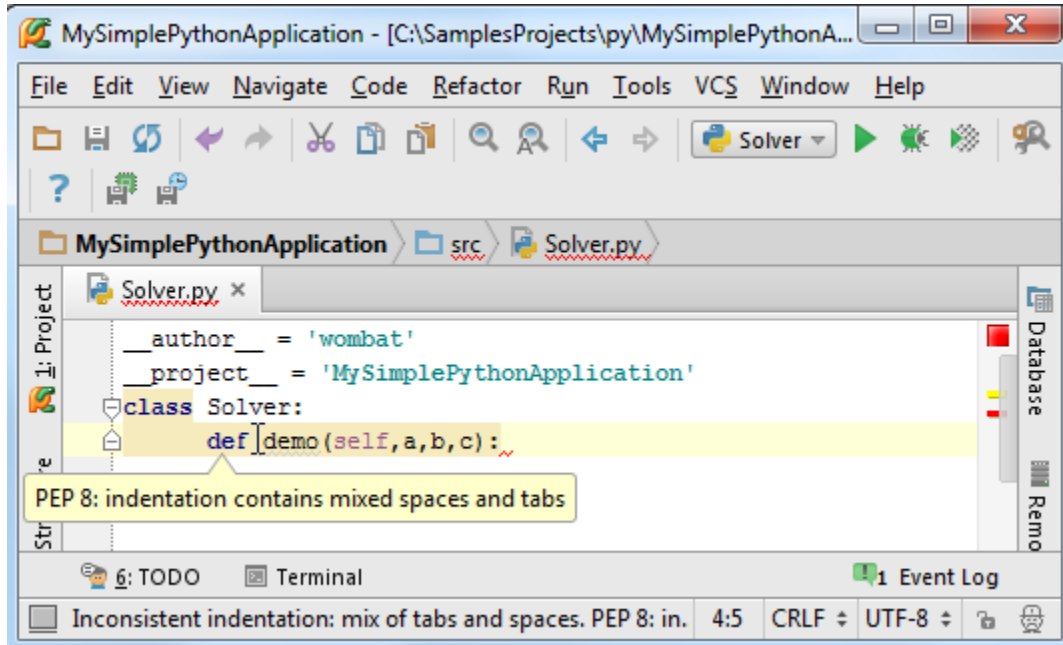
然而，在默认情况下这些警告提醒是不可见的，所以首先需要做的就是提升它们的优先级以进行显示。单击  设置按钮，然后在 Settings/Preferences 对话框中的 [Inspections](#) 页面，键入 PEP8 来找到所有相关选项，在对应的下拉菜单中选择 warning 选项：



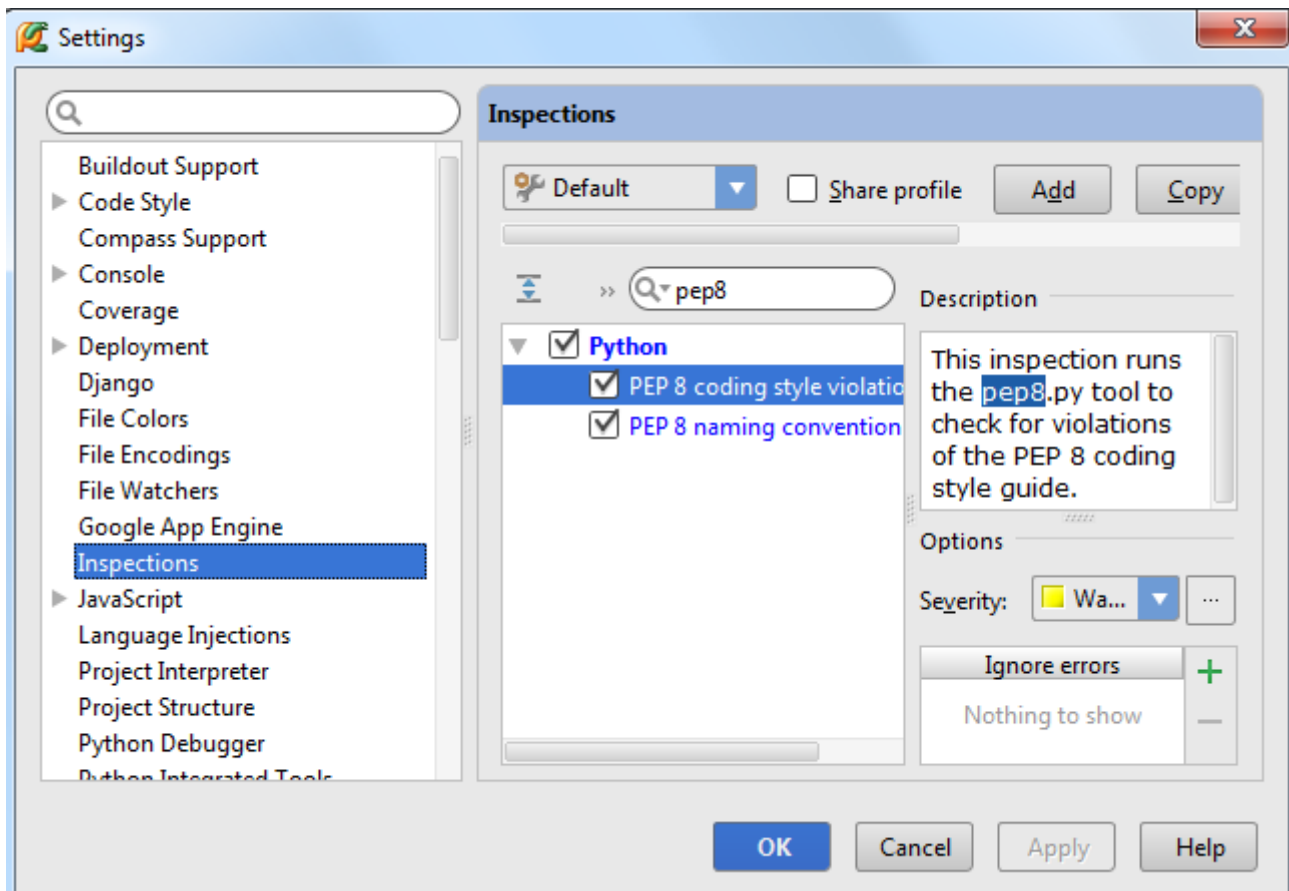
单击应用，关闭对话框，返回源码编辑界面。

5、详解 PEP8 代码风格

现在 Ptcharm 已经能够正常显示它的代码规范，确保你编写的代码格式的完整性。接下来当我们输入下一条语句（例如 `def demo(self,a,b,c):`），Pycharm 将根据 PEP8 的代码规范机制来报告当前存在的格式问题。



正如你所见到的那样，Pycharm 将其所支持的 PEP8 规范设置为默认的正规 Python 代码格式标准。如果你打开 inspections 的列表，（`Ctrl+Alt+S`→Inspections），可以看到 Pycharm 在你的代码中加载了 `pep8.py` 工具，用来精确定位你的代码风格问题。



6、代码检查以及相关设置

顺便说一下，如果你仔细观察 [Inspections page](#) 页面中 [inspection profile](#) 的缺省设置（如果你是第一次进行设置的话）会发现，Pycharm 已经将所有的代码规则用于当前的工程中了。

接下来我们对代码检查机制做两方面的改动：

- (1) 在测试脚本中，将拼写错误标记为绿色
- (2) 在说明文档（注释）中，将拼写错误改为红色提示

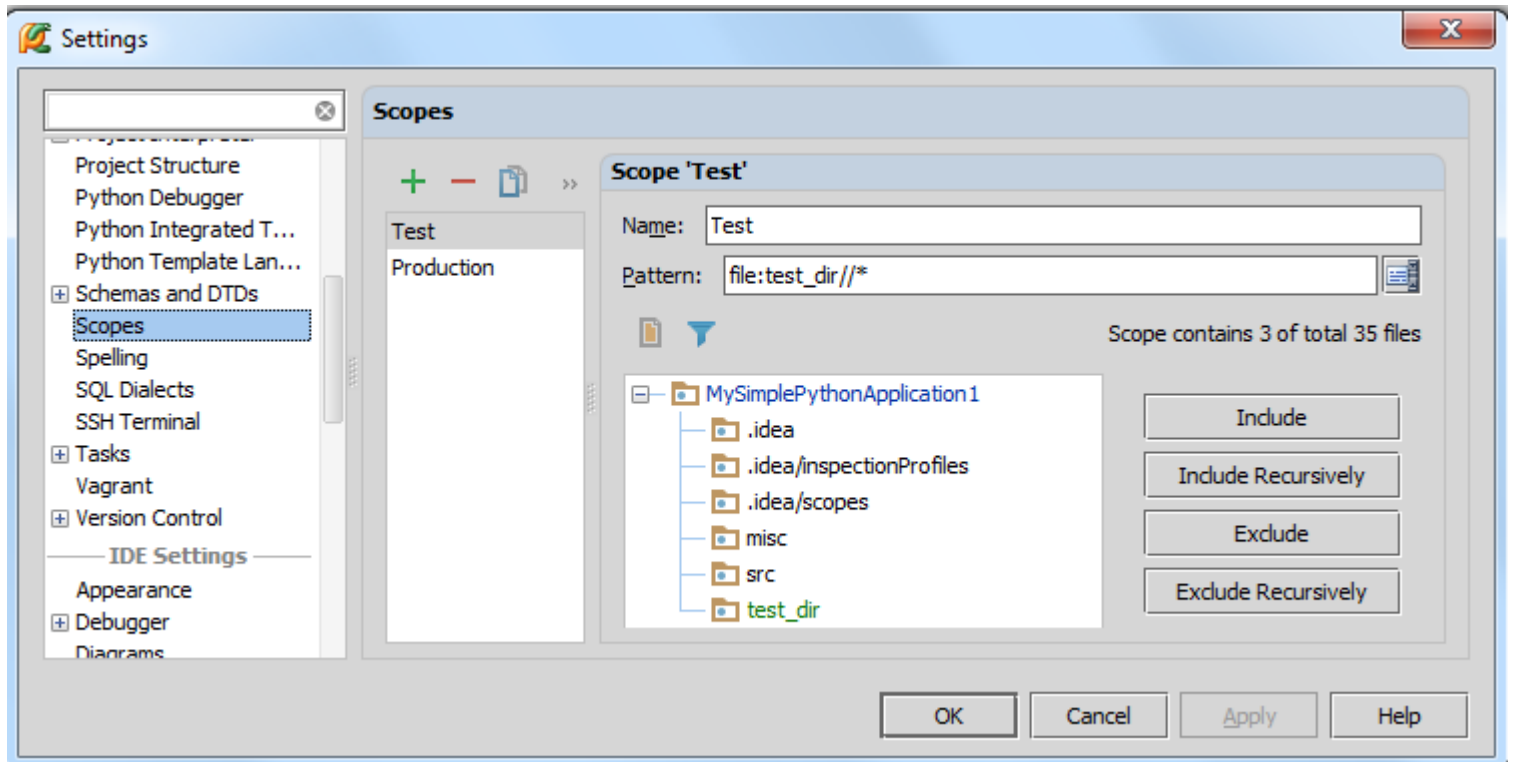
接下来我们一一进行介绍

7、创建一个作用域

首先我们需要创建两个作用域来进行两个不同应用范围的设置。单击设置按钮进入 Settings/Preferences 对话框，打开 [Scopes](#) 页面，单击上方绿色的加号来创建一个局部类型的作用域：



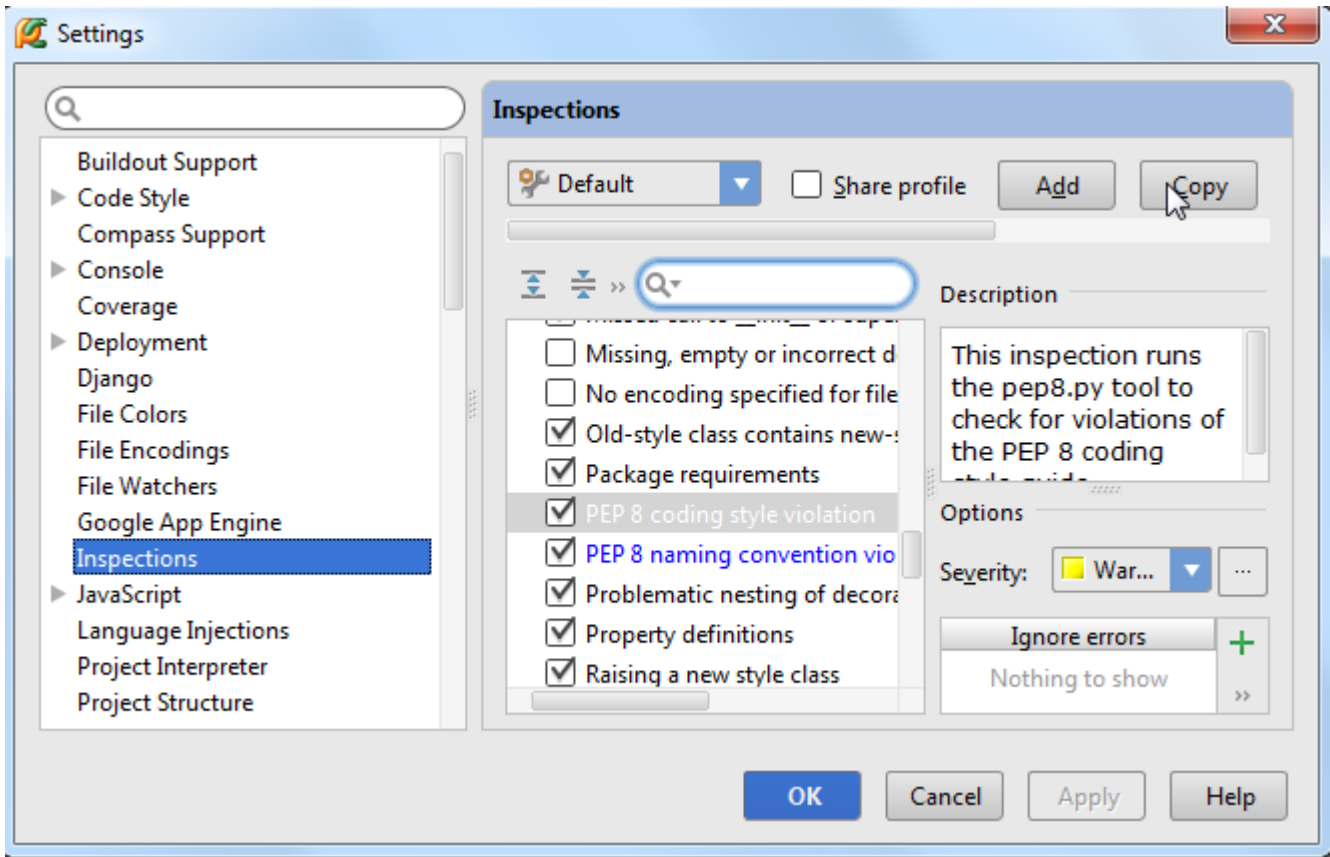
在 Add New Scope 对话框中，键入作用域名称，然后在工程管理器（树型结构）中选择需要包含到当前作用域中的目录：test_dir，注意此时的 Pattern 栏已经自动显示加载路径：



重复上述步骤再新建一个 Production 作用域。

8、在新建的作用域中创建代码检查控制文件

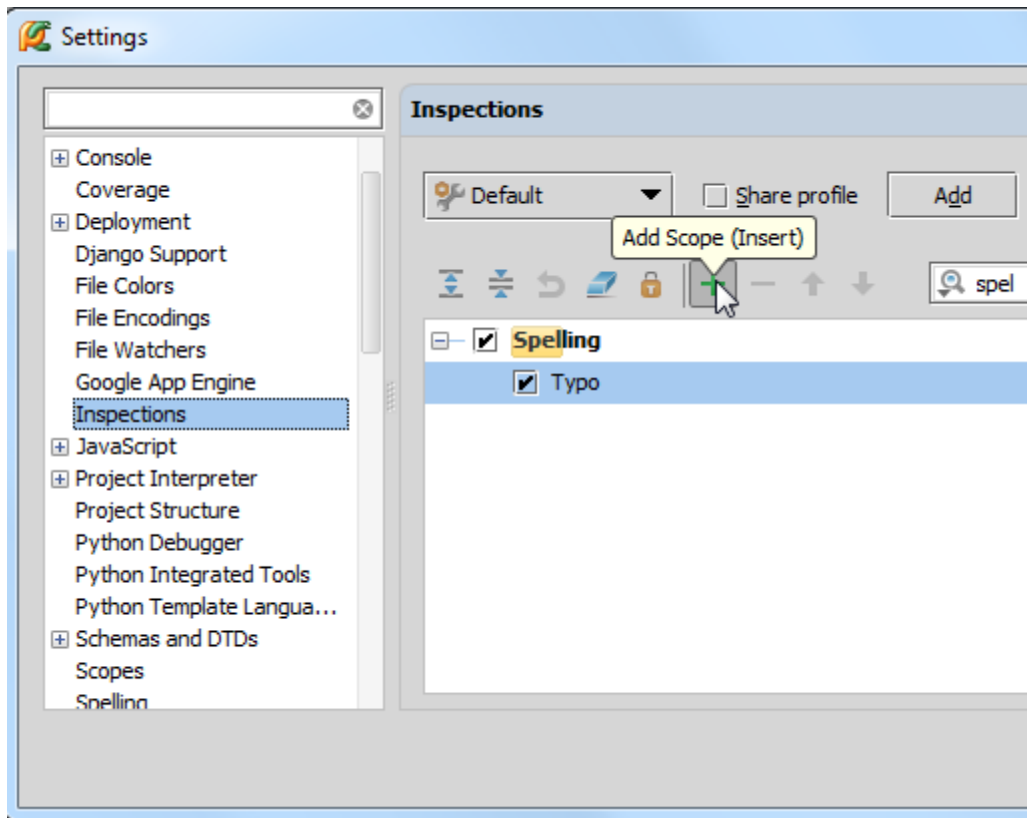
接下来，创建一份缺省代码控制文件的[拷贝文件](#)（处于安全考虑）：



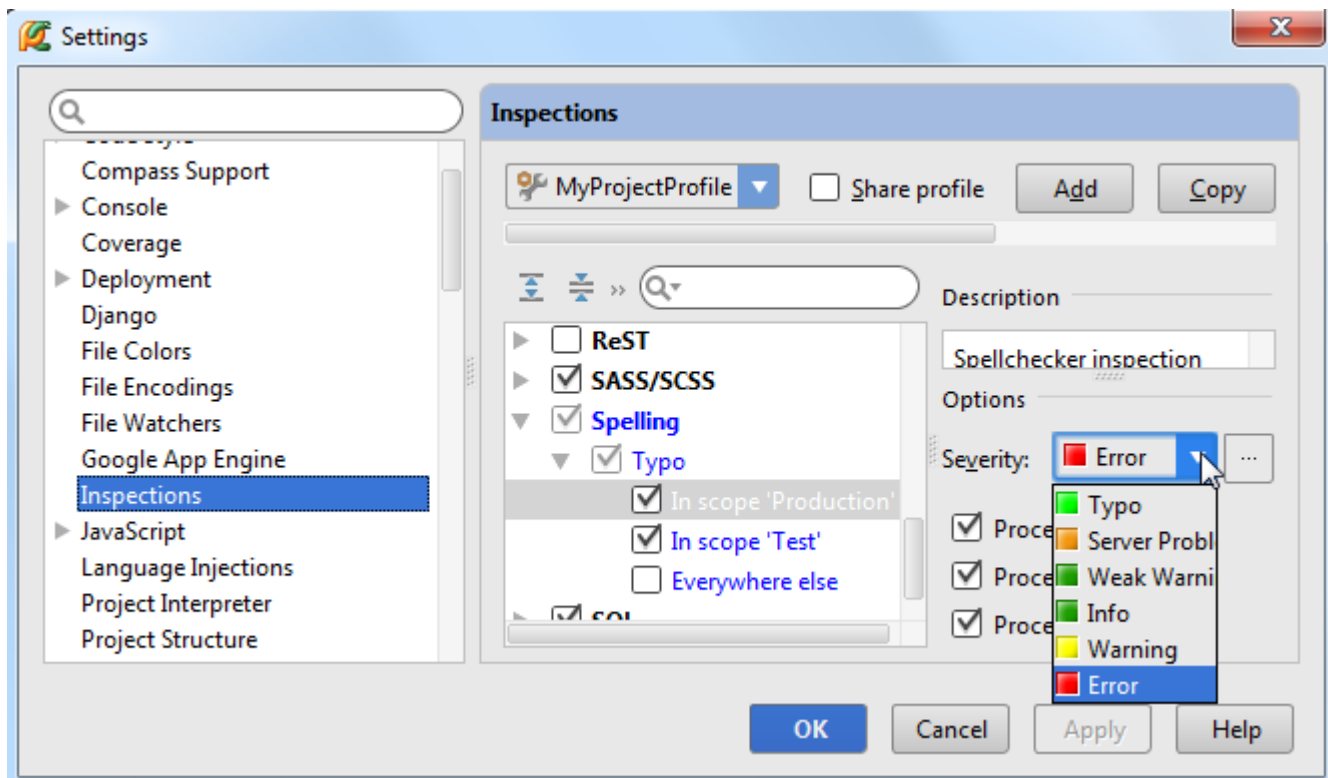
然后在对它进行命名，例如我们这里命名为 MyProjectProfile。这个新的配置文件是之前默认缺省配置文件的复制版，两者的设置内容完全相同。

接下来选中我们拷贝的代码控制文件，定位到 Spelling 项进行相应改动。为了快速找到 Spelling 选项叶，只需在搜索栏中输入 Spel 即可。

然后通过单击绿色的加号来添加我们之前新建的 Test 作用域，然后再次单击添加 Production 作用域：



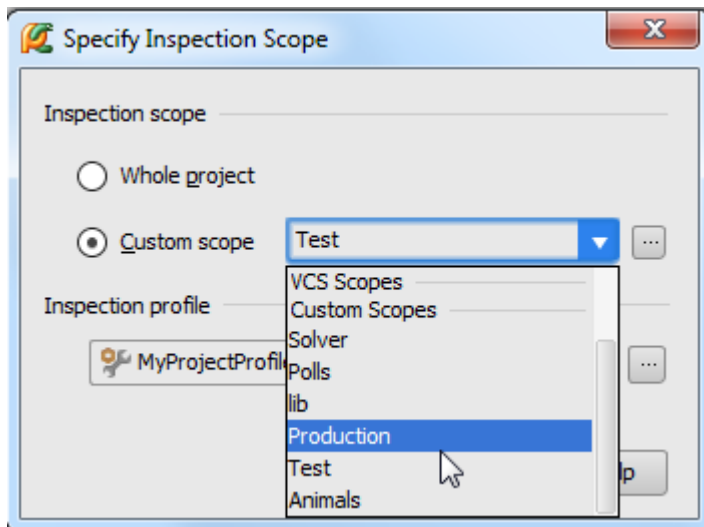
在 Test 作用域中，代码检查的严格等级如图中左侧所示，Production 作用域中有类似设置，不过所选择的下拉列表中的安全等级不同：



留意对话框中作用域名称的字体颜色，如果为灰色则说明未做改动，若是蓝色则说明已经更改了相关设置。

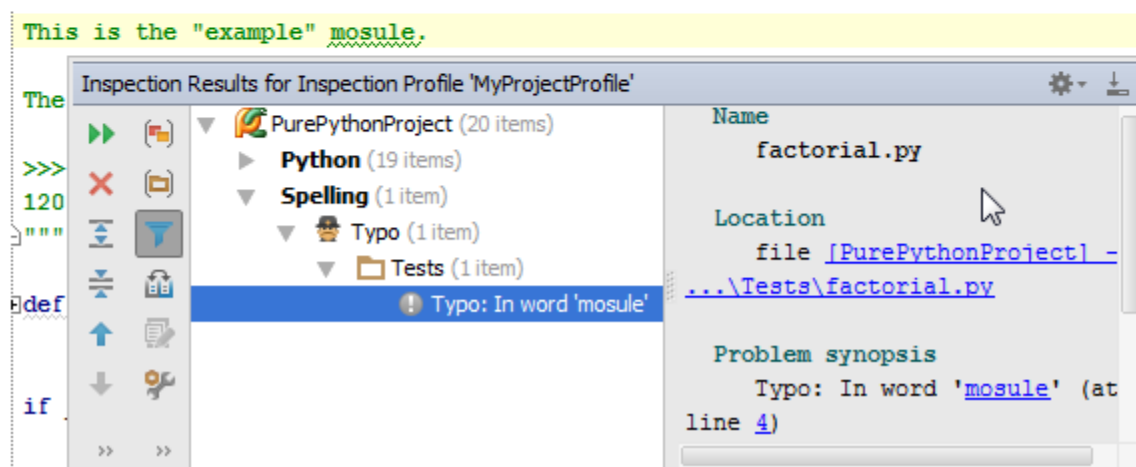
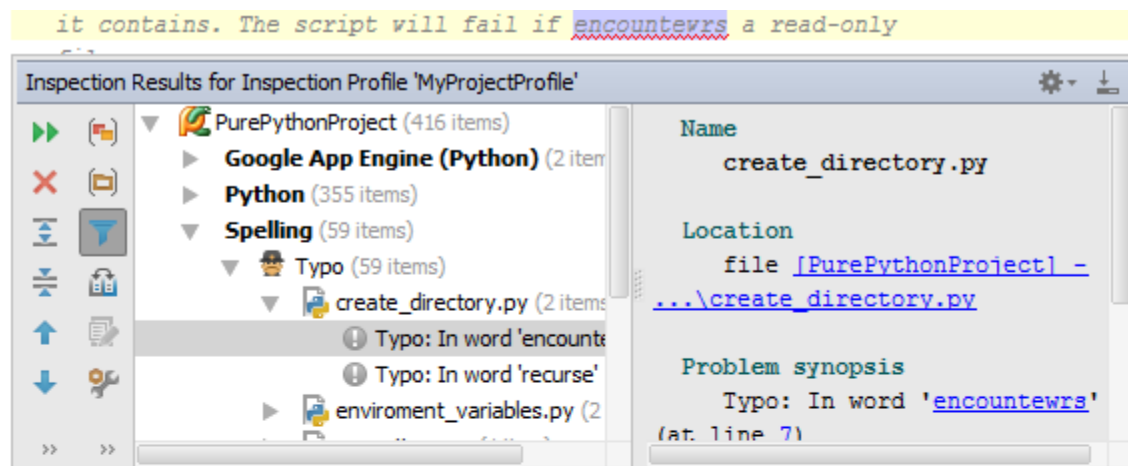
应用更改设置然后关闭对话框。

此时，按照要求修改后的配置文件已经完成，名为 MyProjectProfile，其在 Test 作用域和 Production 作用域中有不同的拼写检查设置。接下来我们将这个配置应用于对应代码区域，在主程序菜单中选择 Code→Inspect Code，在对话框中指定已经定义好的作用域和配置文件：



当然我们需要操作两次，因为有两个定义域需要进行相关配置的更改，并且可以将相关的配置清单导出。

比较一下这两个作用域的拼写检查结果：

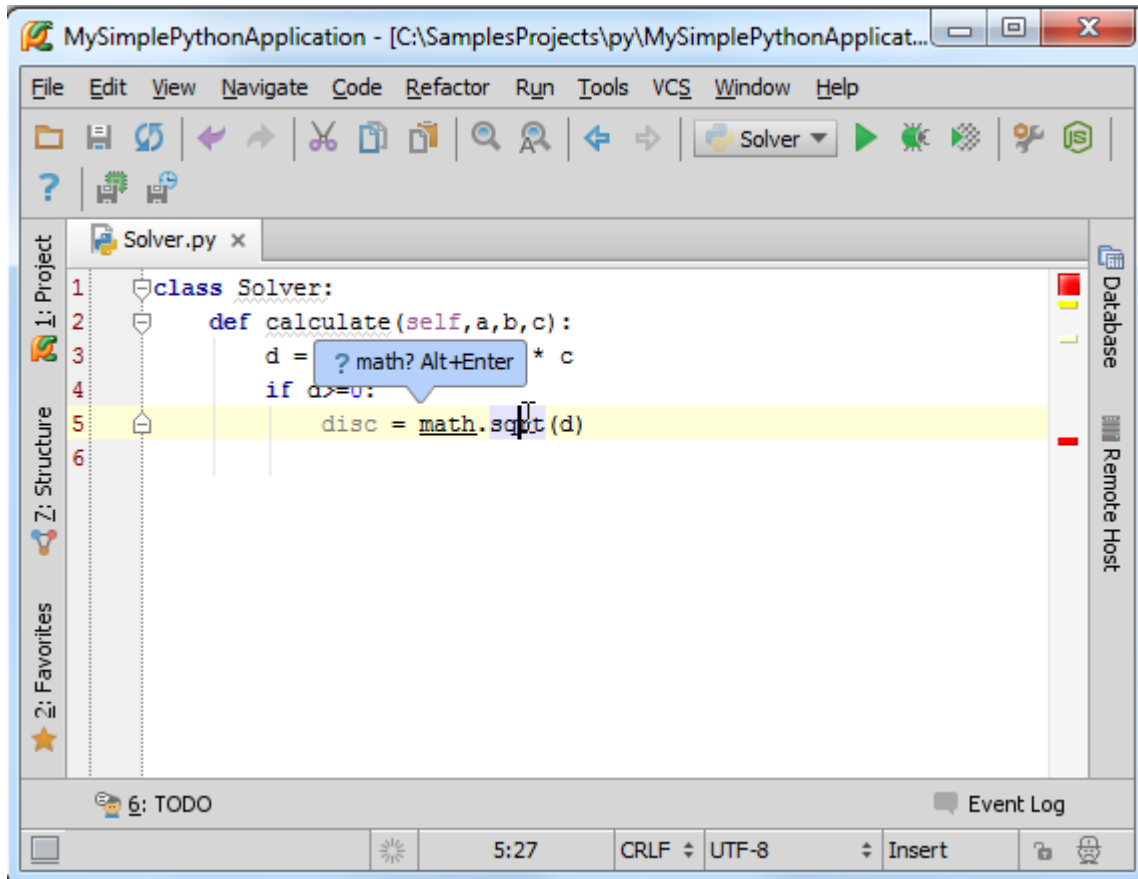


正如你所见，在 Production 作用域为红色波浪线，在 Test 作用域为绿色波浪线。

9、错误提示的高亮代码显示

除此之外，Pycharm 还会根据配置文件控制，对当前的一些错误进行高亮显示处理。

举个例子，如果你的拼写检查配置文件中包含"Unresolved references"这条检查规则，同时你又使用了一条尚未进行 import 的符号，Pycharm 就会用下划线标出无法解释的符号来提示你导入相关模块：



参考 [auto-import tutorial](#) 来完成相关模块的导入工作

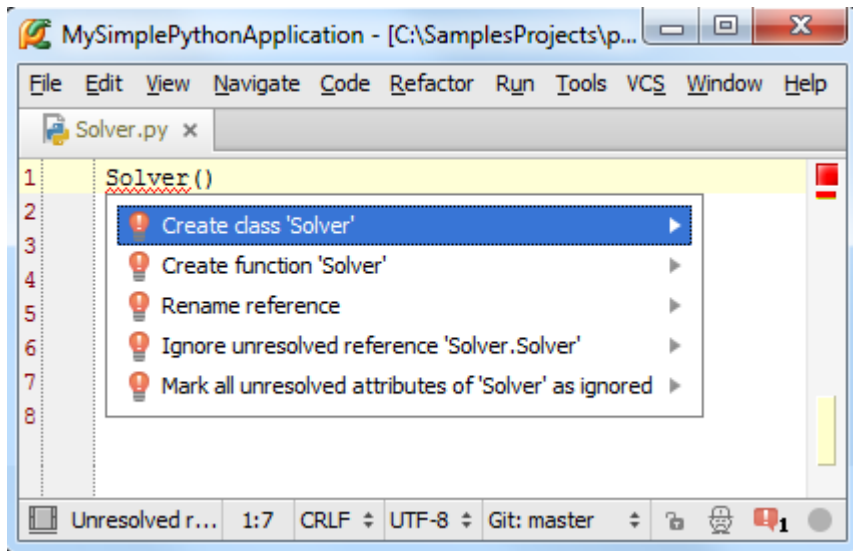
10、快速成型以及多次提示

你是否已经注意到在代码左端经常出现一个亮起的黄色或者红色的灯泡然而你却并不希望看到它？

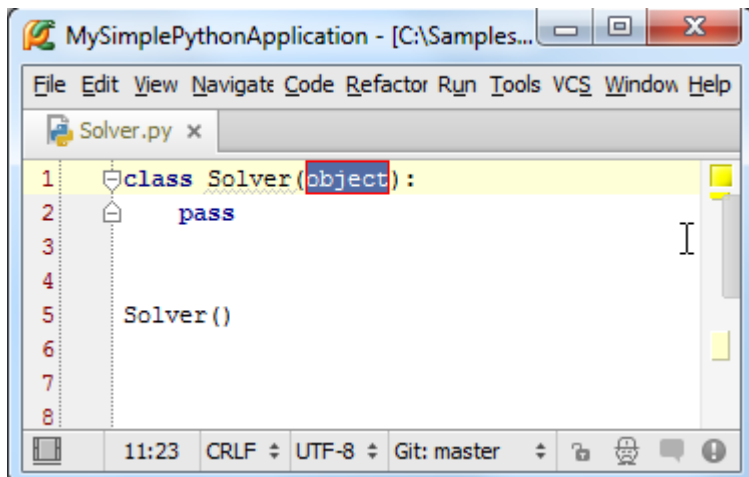
11、源码自动生成

Pycharm 提供了很多代码自动生成机制，你可以参照 [product documentation](#) 中有关自动生成代码的介绍：[Auto-generating code](#)，接下来我们探讨一下 Pycharm 的主代码生成机制。当然我们需要先将 Solver.py 中已有的内容删除，重新开始。

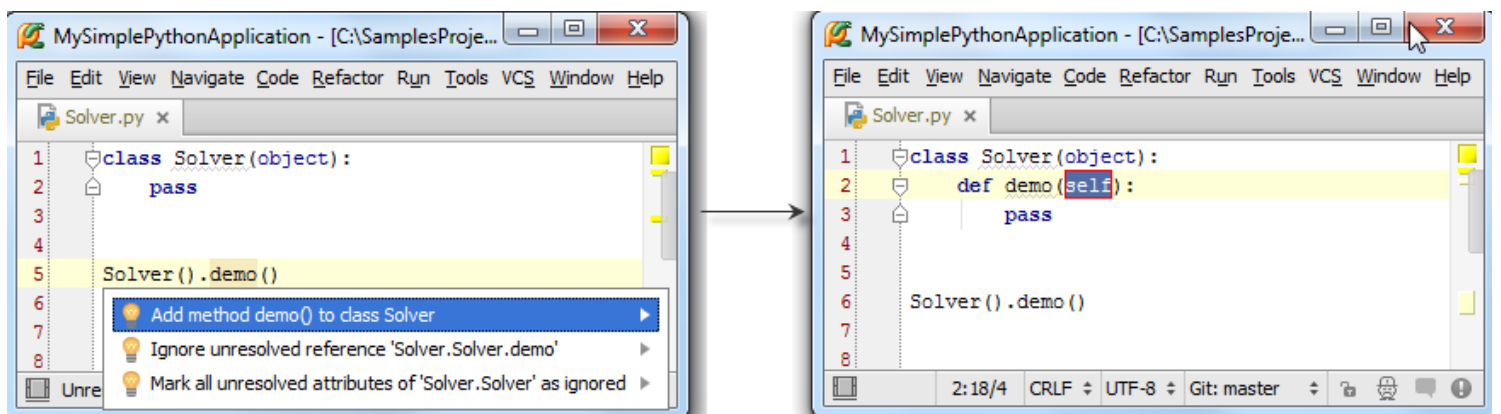
首先，创建一个类实例：



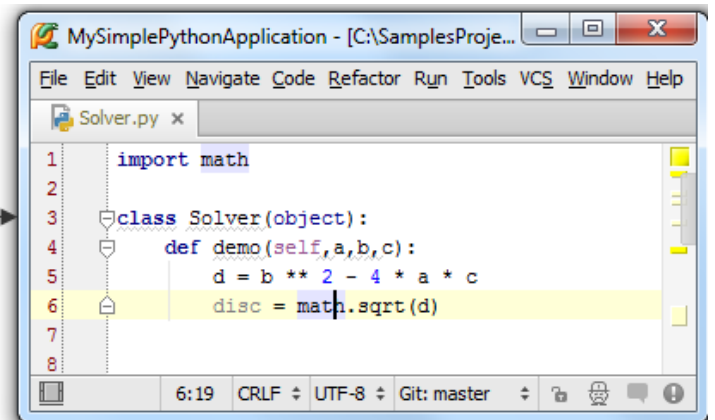
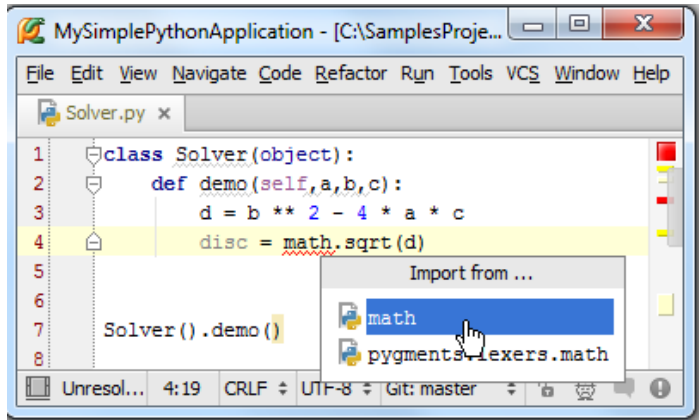
OK, Pycharm 成功创建出了一个类:



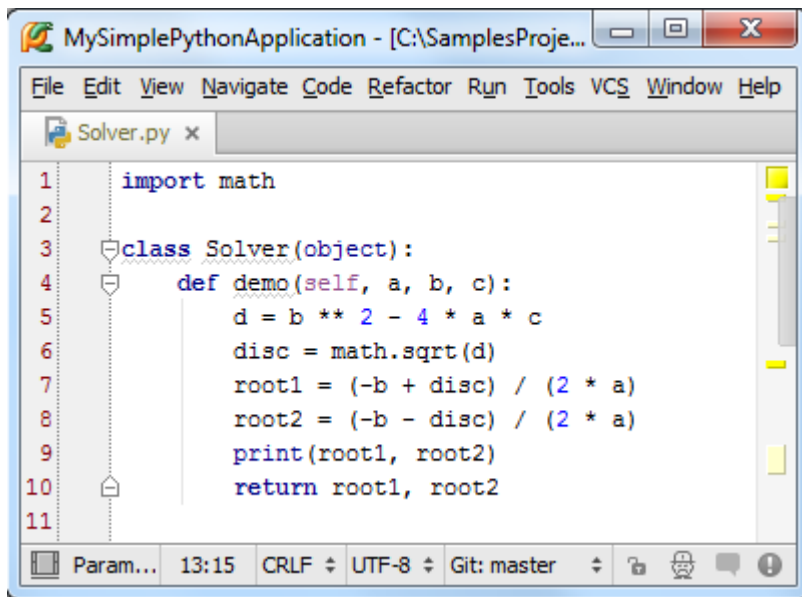
接下来我们向类中添加一个成员方法，为了达到这个目的，首先需要在类实例后面输入一个点号，然后键入成员函数名称。此时这个成员函数是未定义的，因此 Pycharm 会提示我们来创建一个：



然后在函数体中手动输入源码，例如我们输入一段计算二次方程判别式的程序，其中有一个函数 `sqrt()` 来自 `math` 模块，但目前尚未被包含，我们继续输入，看 Pycharm 如何解决这个问题：

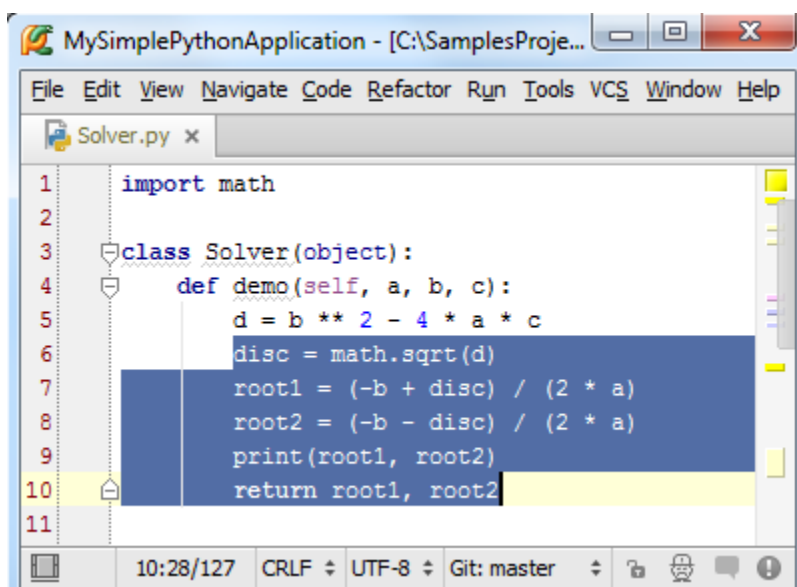


因此，我们源码最终如下：

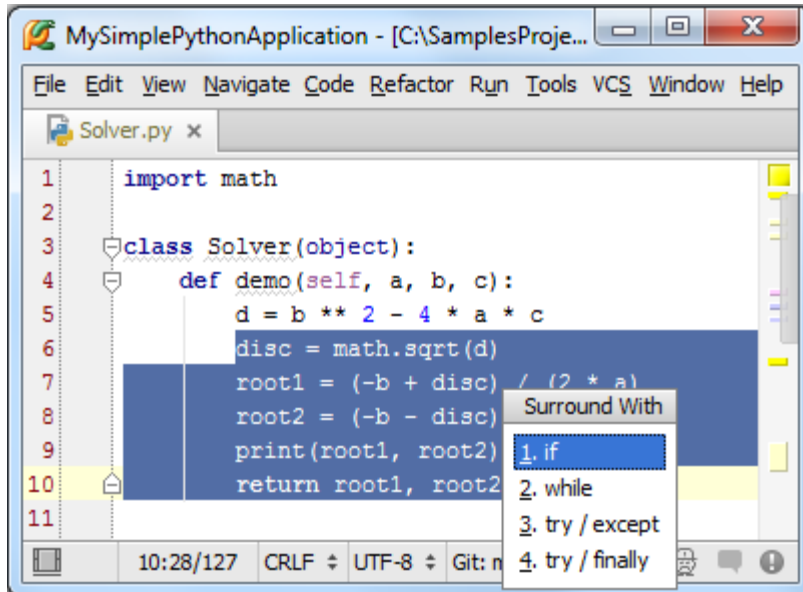


然而，代码缺少一些重要的逻辑分析。我们需要分析判别式结果 d ，如果它是零或者正数，则正常求解方程的根；如果其为负数，我们需要抛出一个异常，Pycharm 会如何帮助我们完成这个任务？

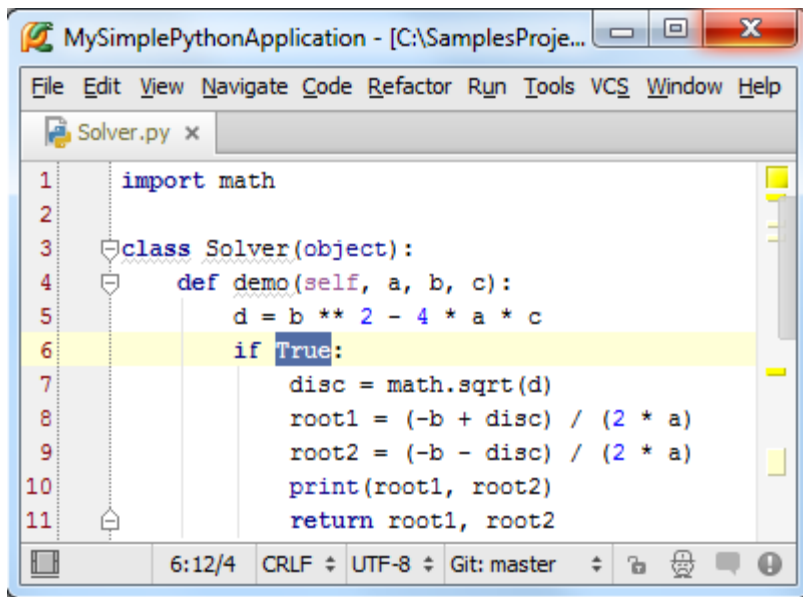
让我们用 `if` 语句来包含一块代码，即选中当 d 为非负数时需要执行的语句：



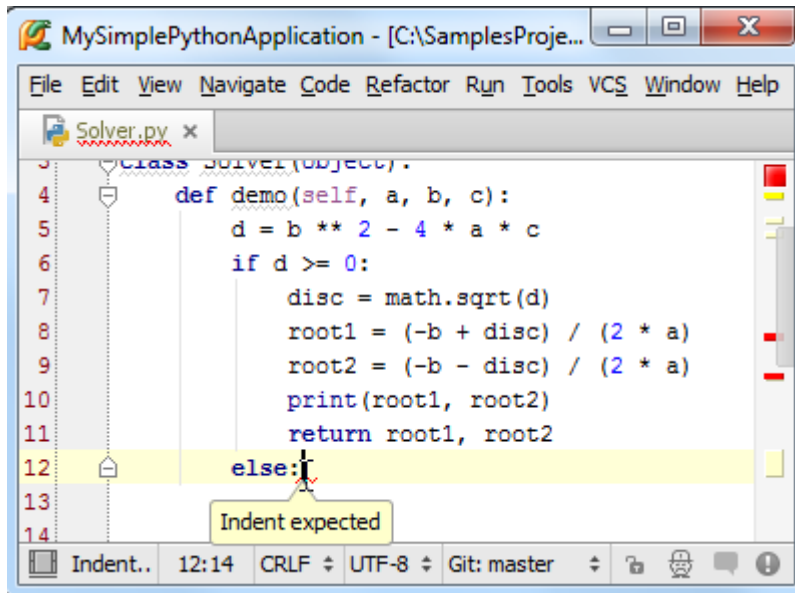
然后按下 Ctrl+Alt+T，或者单击主菜单中的 Code→Surround With 选项，Pycharm 将会弹出一个下拉菜单，显示当前情况下可用的范围控制结构：



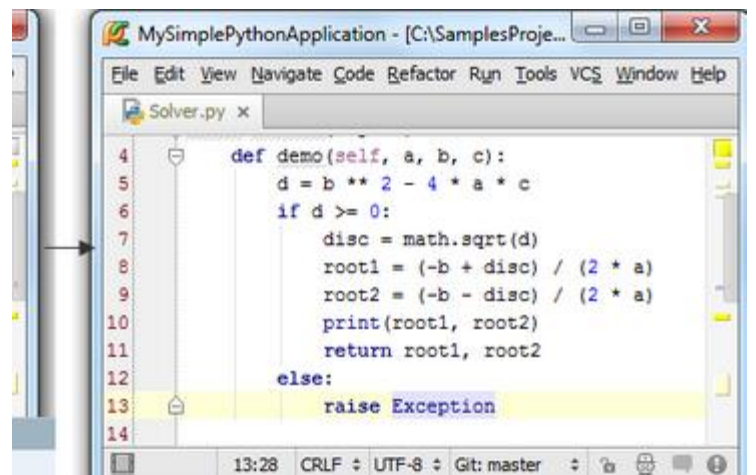
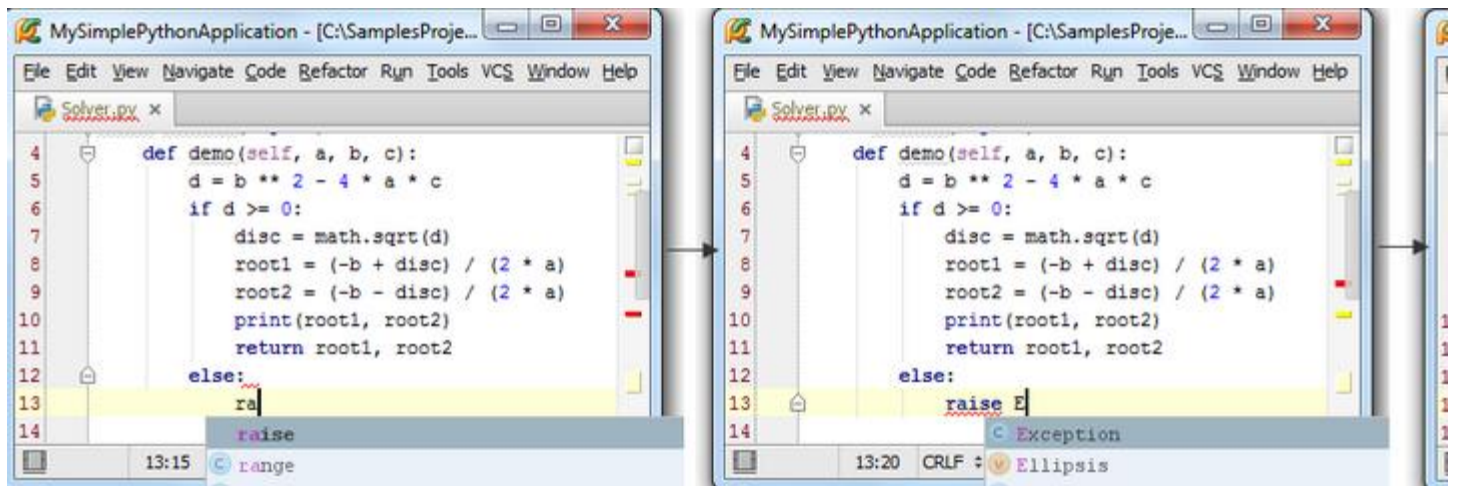
选择 if 选项，Pycharm 会自动添加 if True: 语句到选中的行：



这里我们并不对布尔表达式做过多解释，根据需要我们直接将 True 替换成 $d \geq 0$ ，接下来将光标定位到最后一行，回车，光标将会出现在下一行，和 if 保持相同的缩进，输入 else: ，然后观察 Pycharm 给出的预输入提示：

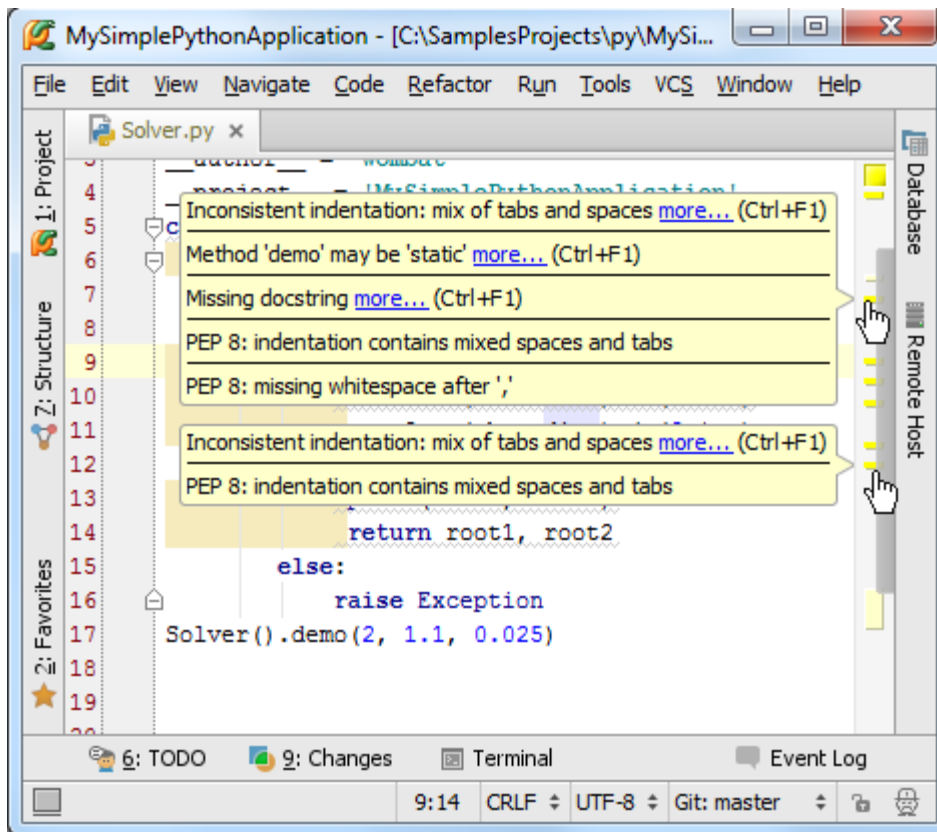


再次回车，移动光标，这里我们在 Pycharm 强大的拼写提示下输入抛出异常的代码：



12、代码格式修改

再次观察 Solver.py 文件会发现，右边滚动槽中显示了很多黄色标记，将鼠标悬停在上边，Pycharm 将会显示对应的代码格式问题：

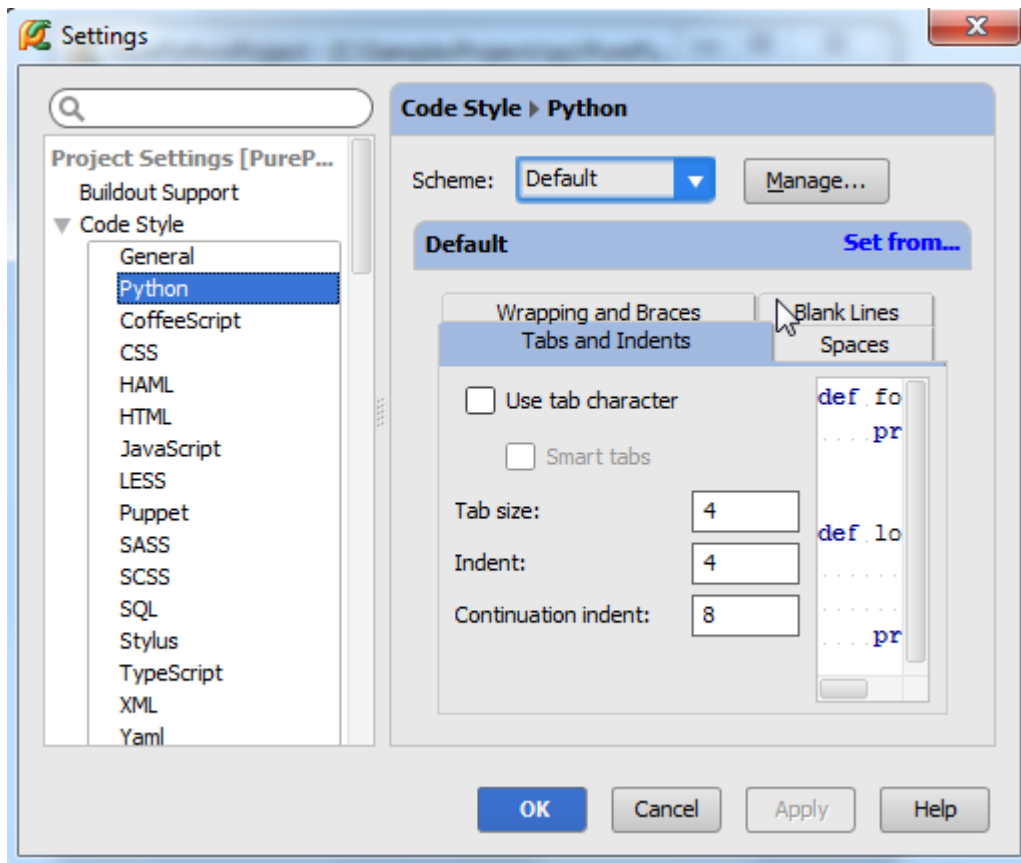


好在这些信息都是警告信息，并不会影响到代码的运行结果，但是格式问题实在是太多了，那么如何把代码格式调整得更为美观规范呢？

这里所用到的就是 [code reformatting](#) 了，不妨尝试一下。

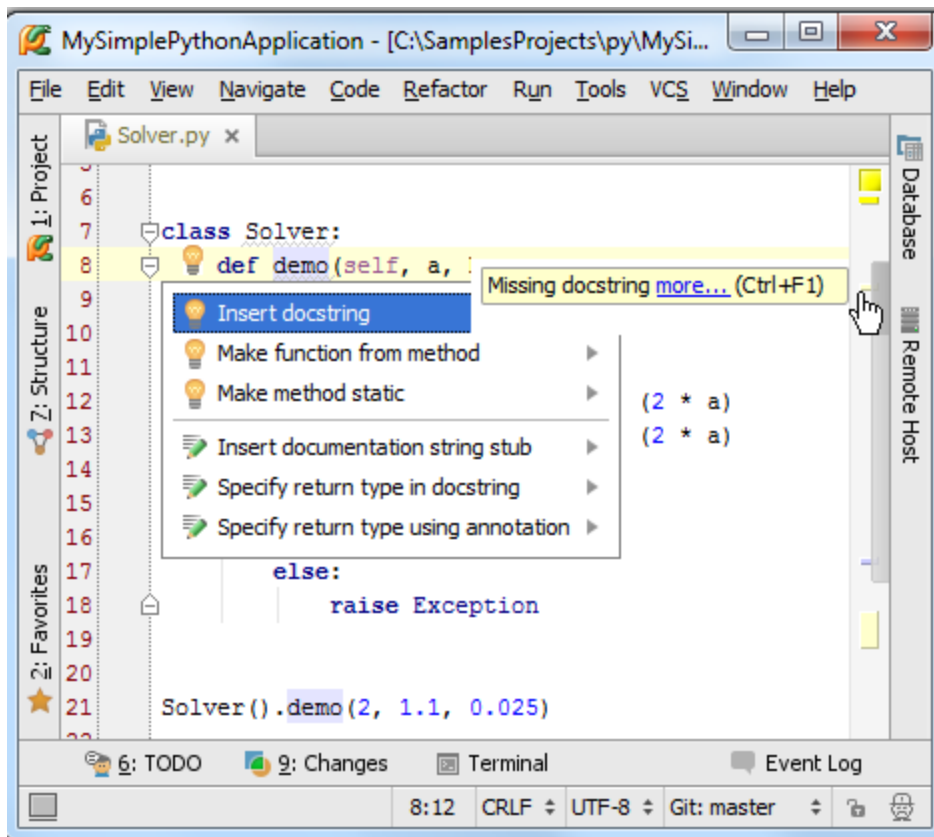
为了调用格式化操作，只需按下 `Ctrl+Alt+L` 快捷键，或者在主菜单中单击 `Code→Reformat Code`，此时我们惊奇发现所有的 PEP8 类格式问题都已经消除。

当然我们可以自定义格式化标准，打开 [code style settings](#) 对话框，选择指定语言（Python），进行必要的更改即可：

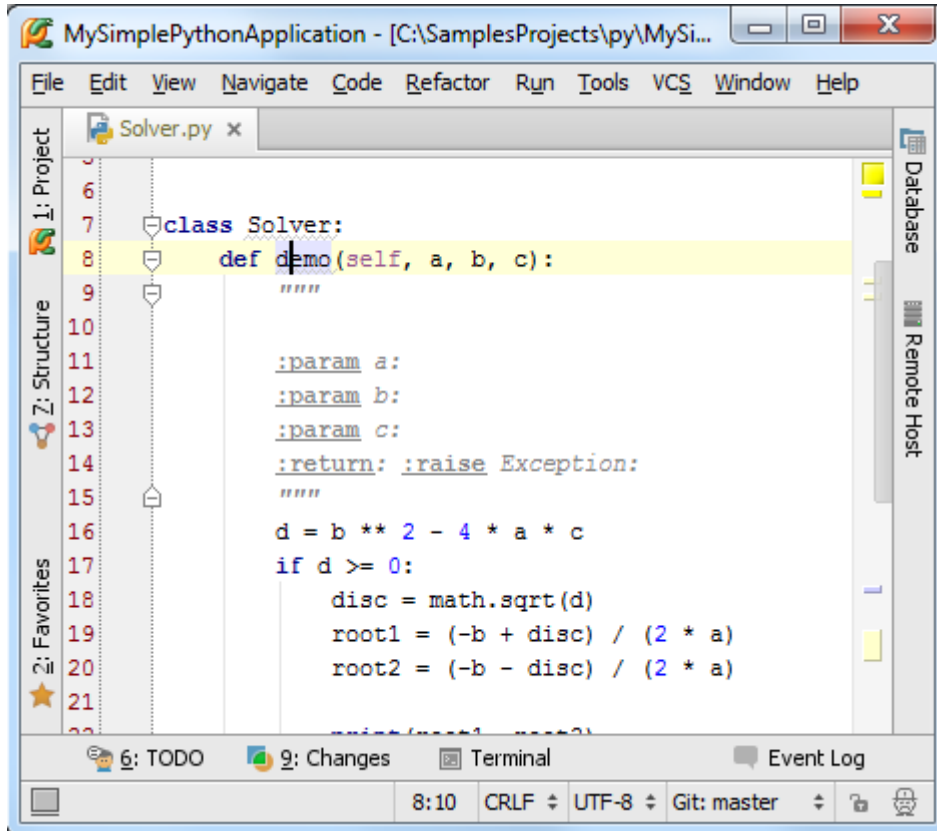


13、添加注释文档

代码格式调整完之后，左侧仍然留有一些黄色的标志位，鼠标悬停后提示类似于"Missing docstring"的警告信息，代码前方亮着的小黄灯泡也提示同样的信息：



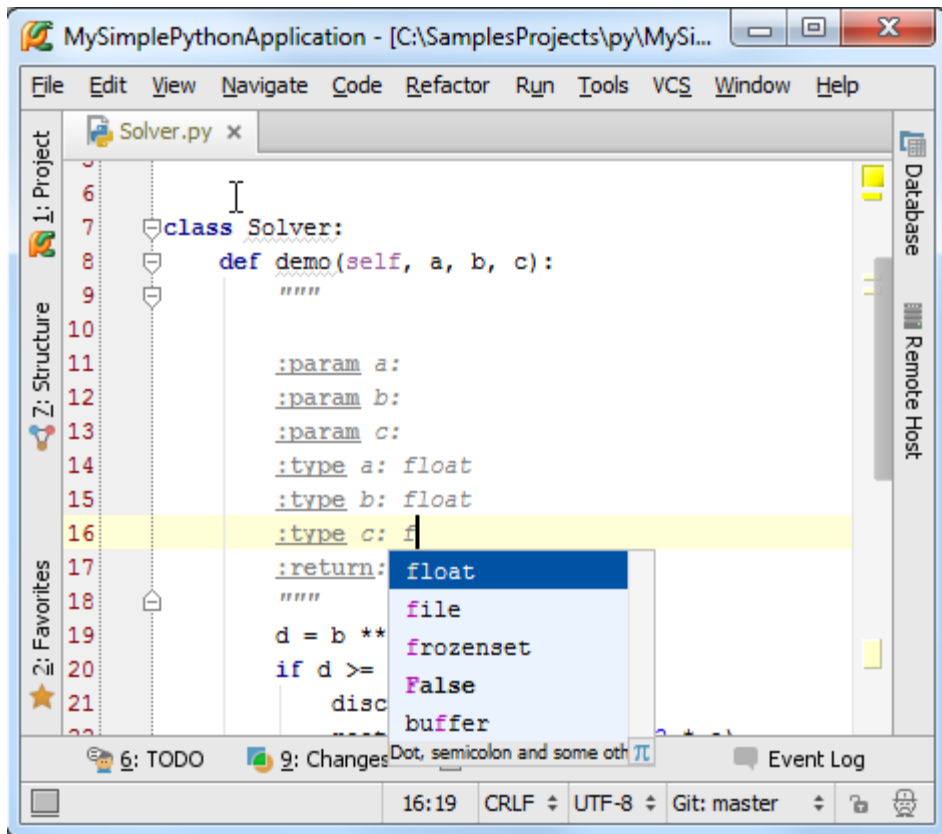
解决方法也很简单，在弹出的下拉菜单中选择 Insert docstring，Pycharm 就会自动添加一段带格式的文本作为注释文档：



注意这里有若干中注释文档的格式,你可以在 [Python Integrated Tools](#) 页面中设置当前需要插入哪种格式的注释文档,例如 Epytext、plain text 等

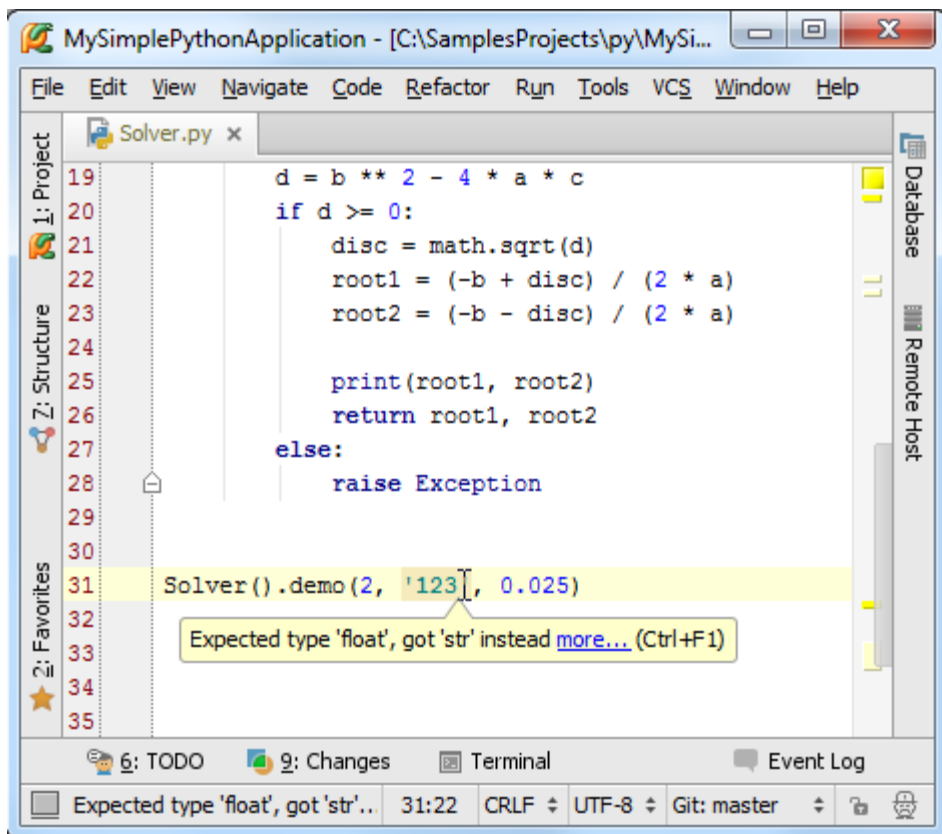
14、输入注释

注释文档用以解释函数的参数、返回值、变量的类型及含义。举个例子，我们需要控制 demo()的输入参数类型，我们就需要在注释文档中添加相应的注释信息：



至此，主函数的注释文档完成。

接下来在函数调用的过程中，若出现参数类型不匹配的情况，Pycharm 会依据注释文档来给出响应的错误提示信息：



更多有关 Pycharm 注释文档的信息参见：[type hinting](#)。

最全 Pycharm 教程（3）——代码的调试、运行

[最全 Pycharm 教程（1）——定制外观](#)

[最全 Pycharm 教程（2）——代码风格](#)

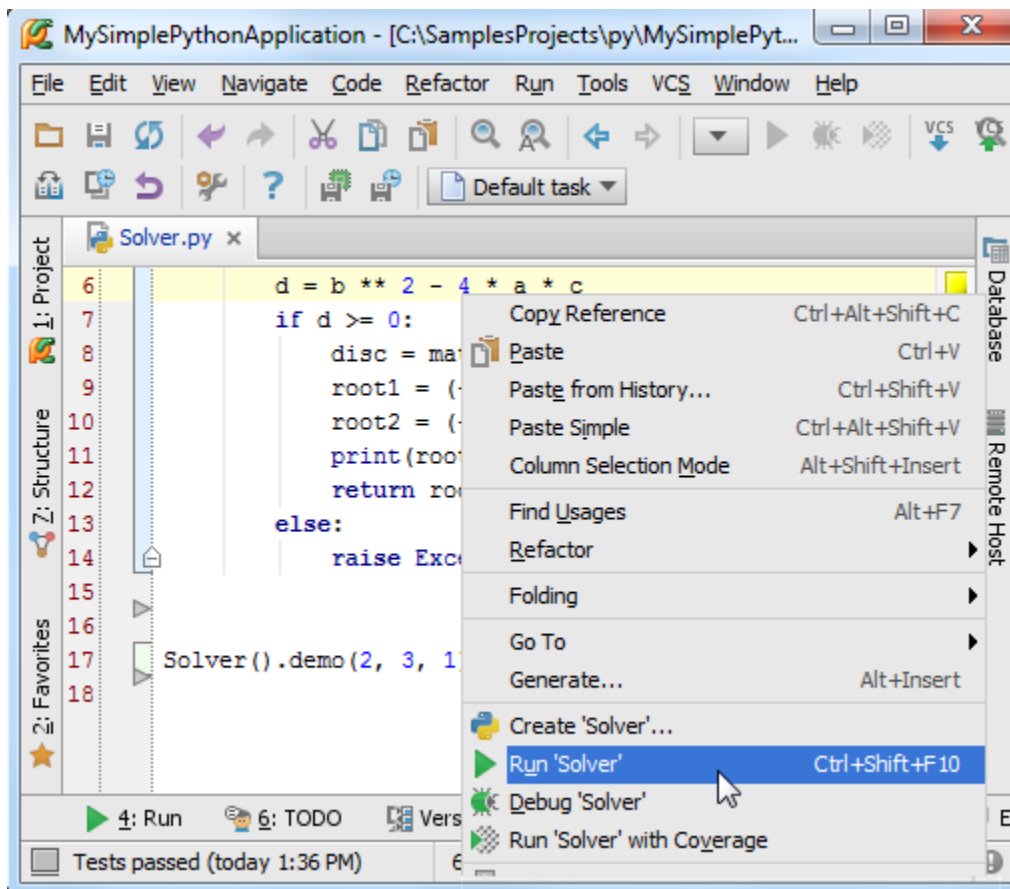
1、准备工作

(1) Python 版本为 2.7 或者更高版本

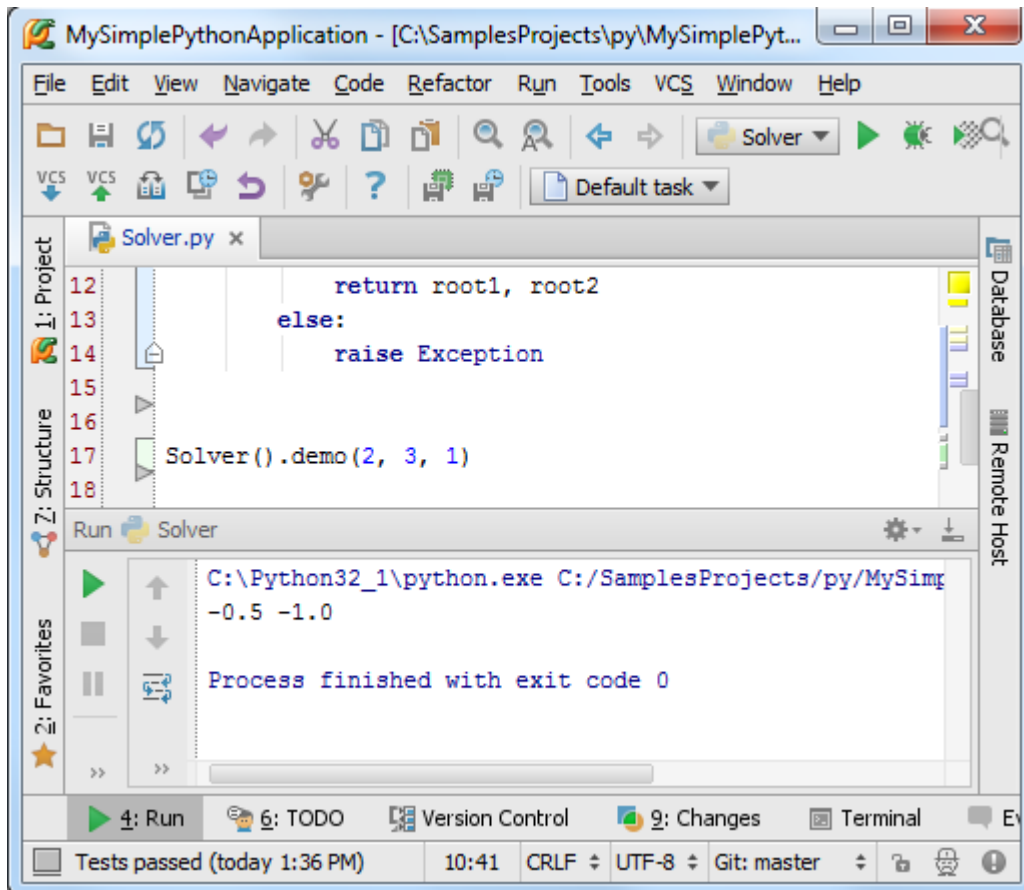
(2) 已经创建了一个 Python 工程并且添加了内容，具体参考：[Getting Started tutorial](#)

2、第一步——运行代码

打开之前编写的 Solver.py 文件，在编辑框中右键，选择快捷菜单中的“Run 'Solver'”选项。



此时脚本文件正常运行并在调试工具窗口中显示程序的输出值：



接下来我们对这两步操作的具体内容做详细的解释。

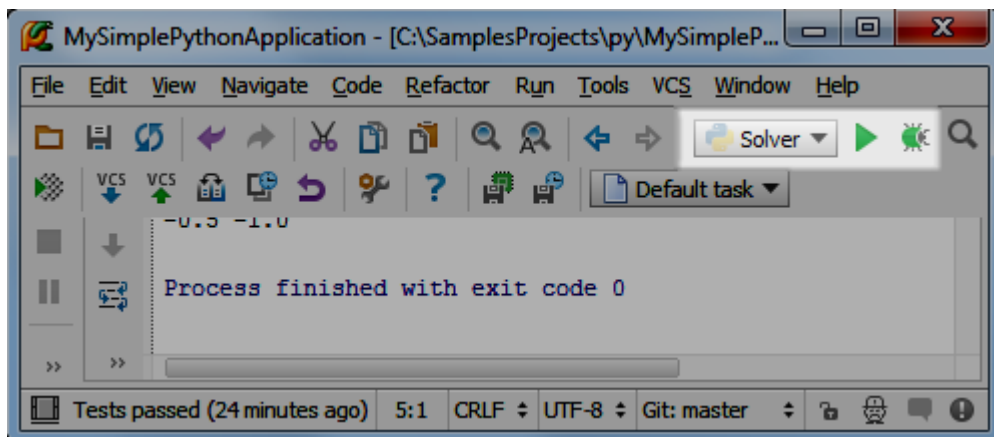
3、什么是 Run/Debug 模式

每个需要运行/调试的脚本文件都需要一个特殊的配置文件来指定其脚本名称、所在目录以及其他重要的运行调试信息。Pycharm 已经集成了这种配置文件，避免用户手动去创建。

每次当你单击 Run 或者 Debug 按钮时（或者在快捷菜单中执行相同操作），实际上都是将当前的运行/调试配置文件加载到当前的调试模型中。

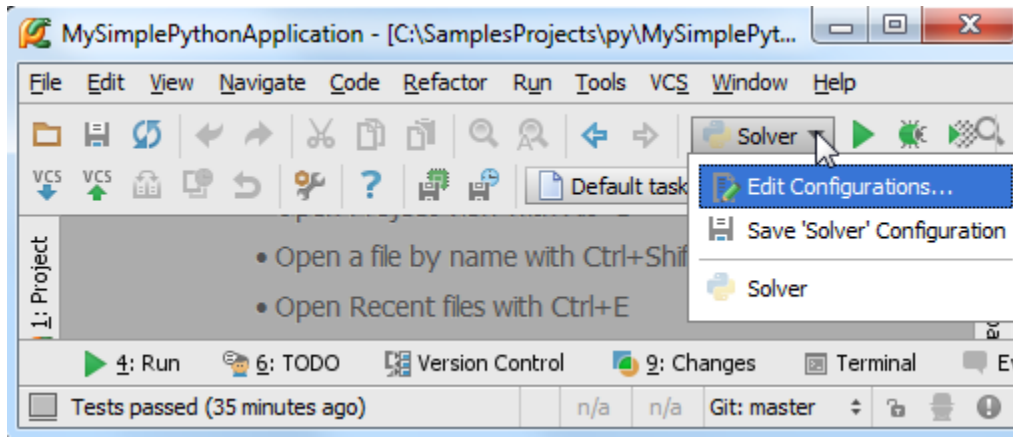
如果你仔细观察第一张图片就会发现，在组合框中根本就没有 run/debug 的相关信息，知道第二张图片中它们才出现。这就意味着当执行运行/调试命令的时候，Solver 脚本的 run/debug 配置文件才会自动生成，正如现在所显示的这样。

此时再主工具栏中 Run（绿色箭头按钮）和 Debug（绿色甲壳虫按钮）两个按钮变得可用：



同时这两个图标还是半透明，也就意味着他们临时的，即由 Pycharm 自动创建的。

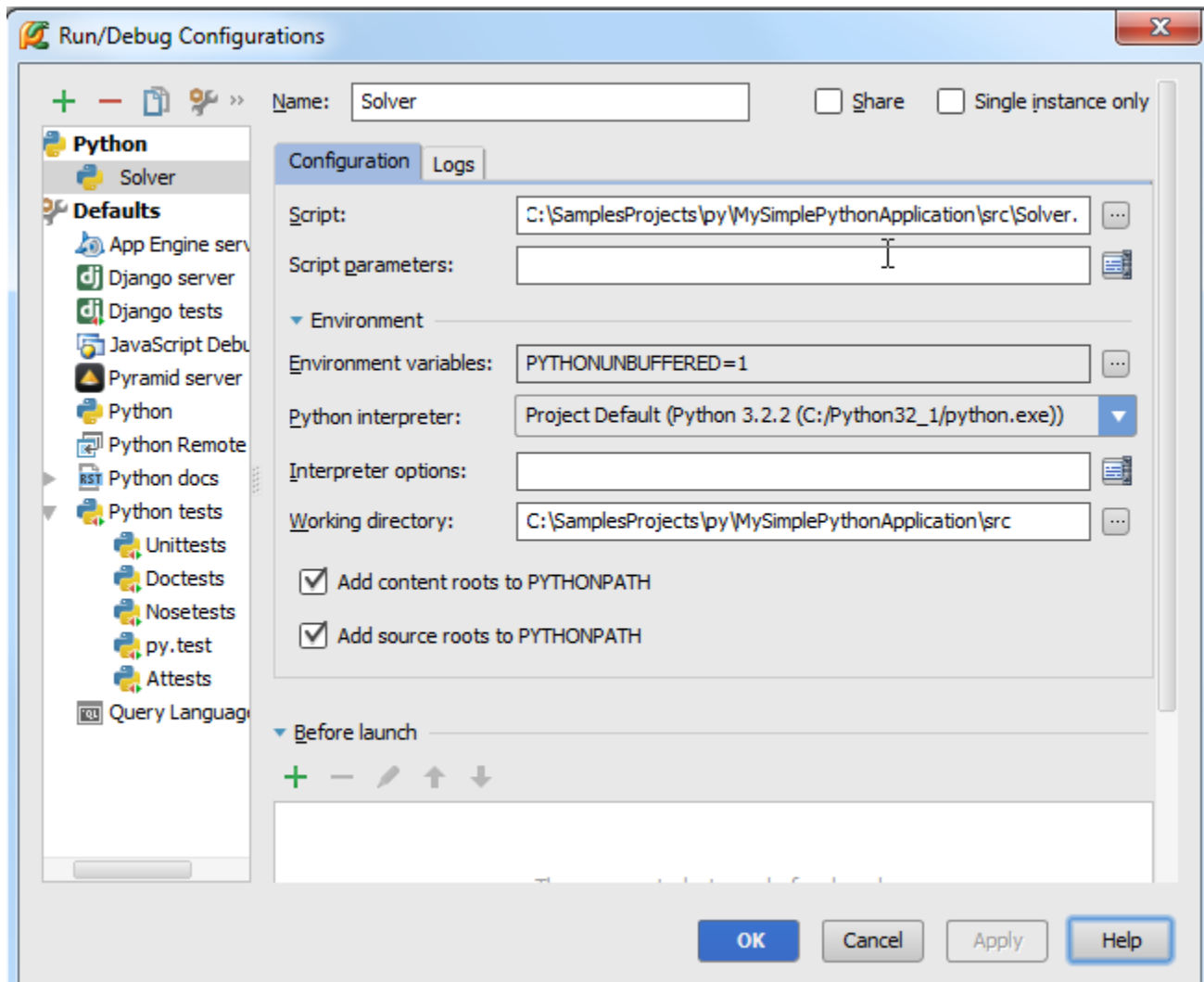
OK, 单击下拉箭头查看当前的可用命令操作:



如果你已经设置了多个 run/debug 配置方案, 它们将都会显示在这里下拉列表中, 单击选中一个作为当前工程的 run/debug 配置文件。

4、保存 run/debug 配置信息

在上图的下拉列表中, 单击 Edit configuration 选项, 打开 run/debug 配置编辑窗口:



在左侧目录中将会出现两个节点: Python 和 Default。在第一个节点目录下有一个唯一的配置选项'Solver', 在第二个选项下则有很多配置信息。

这意味着什么呢？

在 Default 节点下，你只能看到框架的名称或者模式名称，如果你**创建一个新的 Run/Debug 配置文件**，它将会在所选中的模式分支下进行创建，如果你更改了 Default 节点下的设置，相应的与其相关的所有配置文件都会更改。

例如，你想将 Pycharm 中所用到的 Python 解释器替换成远程或者本地解释器，就可以通过更改 Python 页面下的解释器设置，这样所有新建的调试配置文件都会使用这个新的解释器。

早 Python 节点下，只用单一的配置选项'Solver'，它属于 Python 类型的配置，但与 Default 节点下的 Python 机制并不相同，它使用一个非透明的图标进行的表示，这是用来指示当前配置文件的保存状态的，当你保存配置文件之后图标即变为非透明状态。例如，我们在 Python 类型下为当前的 Solver 脚本新建一个配置文件，取名'Solver1'。

如果你对已存在的配置文件做了任何更改，这些更改只会应用于对应的脚本区域。

5、正式运行

我们已经能够通过一种**非常直接的方式**，接下来我们寻求其他方法来运行脚本。

正如我们所知，运行脚本意味着加载当前的调试配置文件，因此，运行脚本主要遵循以下流程：

(1) 在主工具栏中，单击 run/debug 组框，确认当前的调试配置文件信息

(2) 做下面的工作（三选一即可）：

单击运行按钮，加载配置文件

按下 Shift+F10 快捷键

在主菜单上，选择 Run → Run

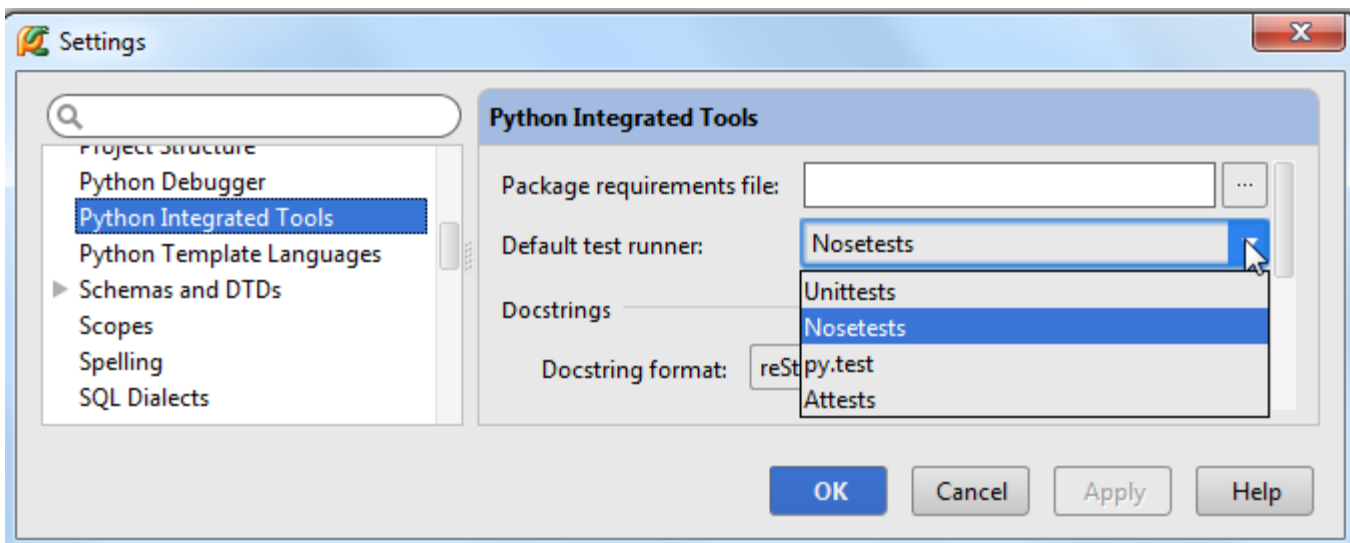
此时，我们可以在 [Run tool window](#)窗口中观察程序的运行结果。

6、运行测试程序

我们这里并不讨论代码测试的重要性，而是探讨 Pycharm 如何帮助我们完成这一功能。

7、选择一个测试器

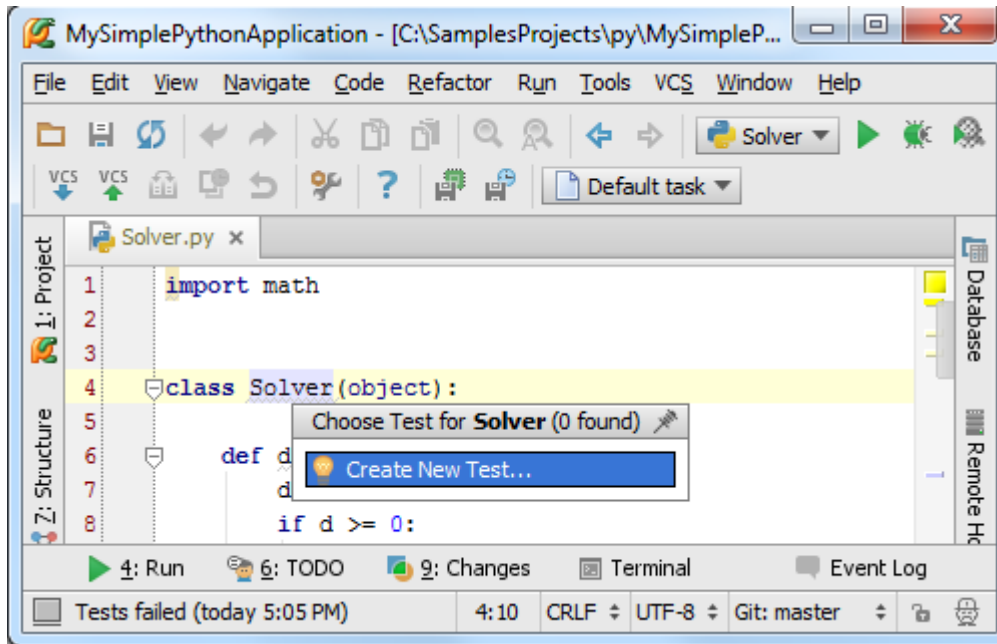
首先，需要指定一个测试器。单击工具栏的设置按钮，打开 Settings/Preferences 对话框，然后单击进入 [Python Intergated Tools](#) 页面（可以通过搜索功能找到），默认选择如下：



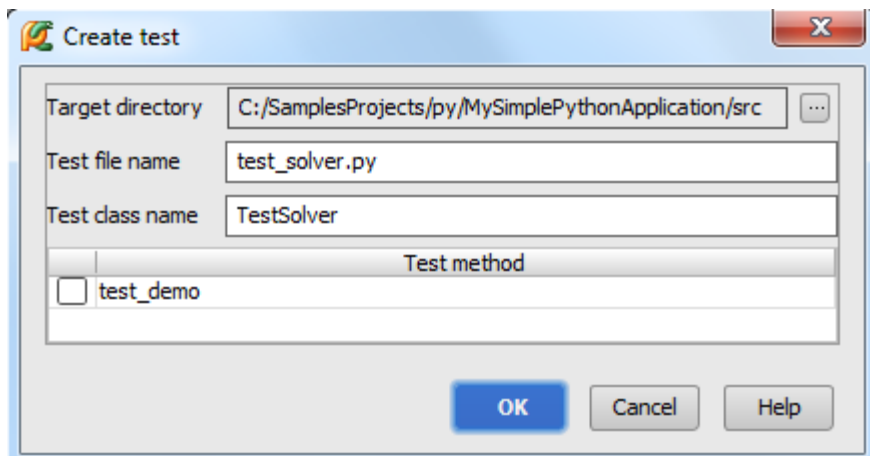
在这里我们选择 Nosetests，保存并关闭对话框。

8、创建一个 test 程序块

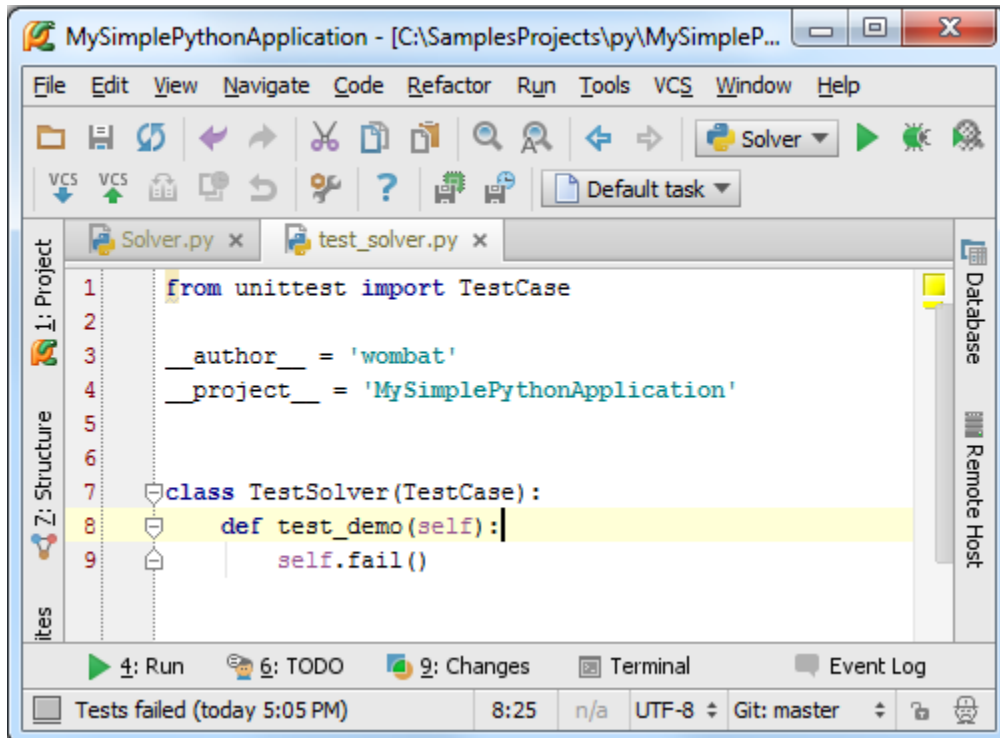
首先我们创建一个 test 实例。Pycharm 提供了一种非常智能的创建测试代码的方法：单击选中类名然后按下 Ctrl+Shift+T 快捷键，或者在主菜单中选择 Navigate → Test，如果 test 程序已存在，则会直接跳转到对应代码，否则创建它：



按照系统提示进行操作，Pycharm 会显示如下对话框：



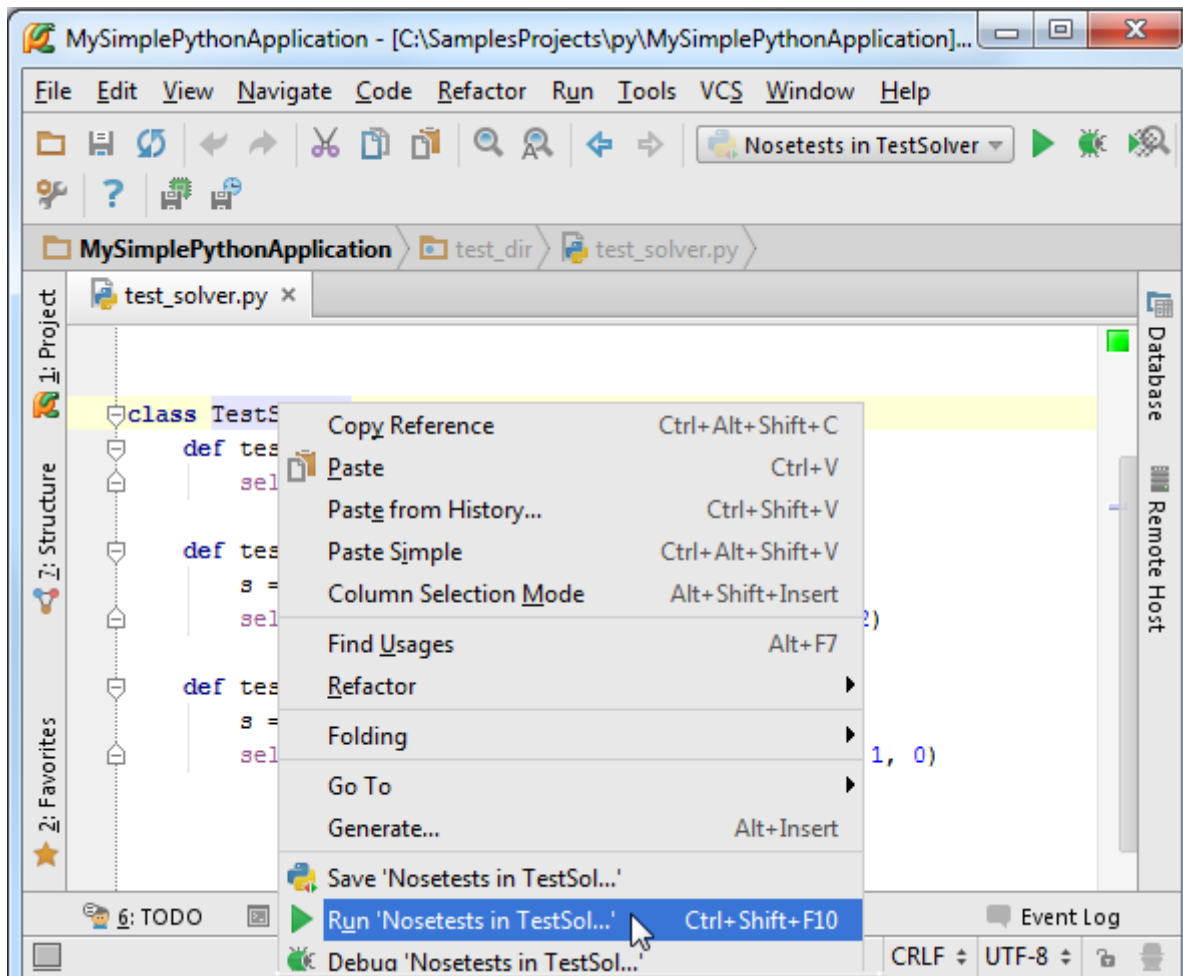
单击 OK 按钮，查看创建结果：



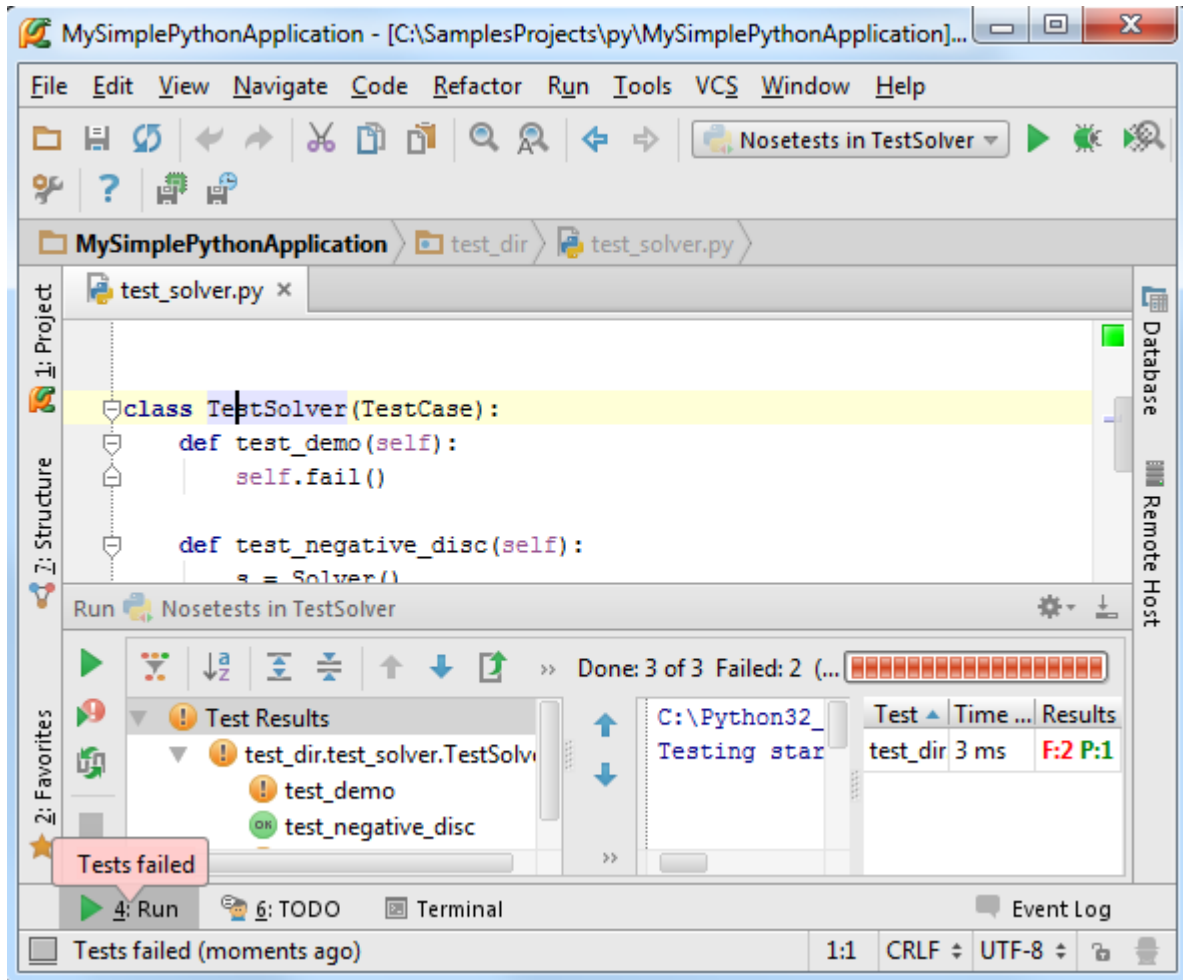
此时 Pycharm 已经自动创建了一个测试类，当然这只是一个类框架，需要我们手动编写测试函数。

9、运行测试代码

一切就绪后，右击测试类名，在弹出的快捷菜单中选择运行命令：



观察运行状态栏中 Test Runner tab 的输出结果：



10、调试运行

首先要弄清楚，为什么要进行调试？假设我们的程序在运行过程中命中了一个错误，那我们如何定位错误发生的位置？这就需要进行调试。

在 Pycharm 中我们可以在其中直接对程序进行调试，唯一需要做的准备工作就是在程序必要的地方加设断点，接下来我们进行详细的介绍：

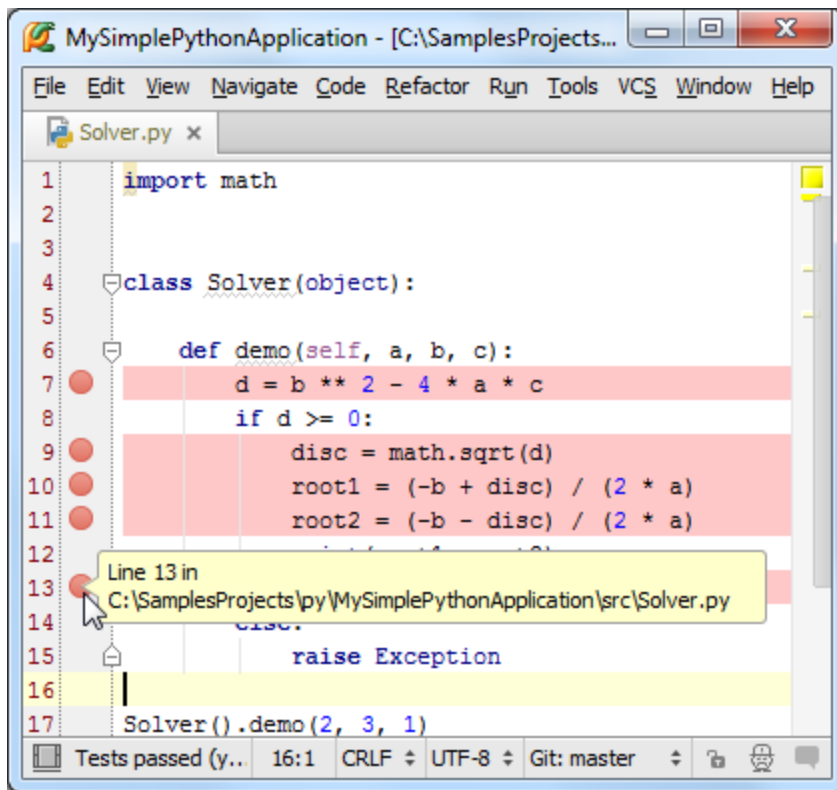
11、什么是断点？

一个 `breakpoint` 标记了一个行的位置，当程序运行到该行代码的时候，Pycharm 会将程序暂时挂起以方便我们对程序的运行状态进行分析。Pycharm 支持若干中类型的断点 `types of breakpoints`，可以通过对应图标进行分辨。

这里我们采用 Python 的行断点为例进行介绍

12、设置断点

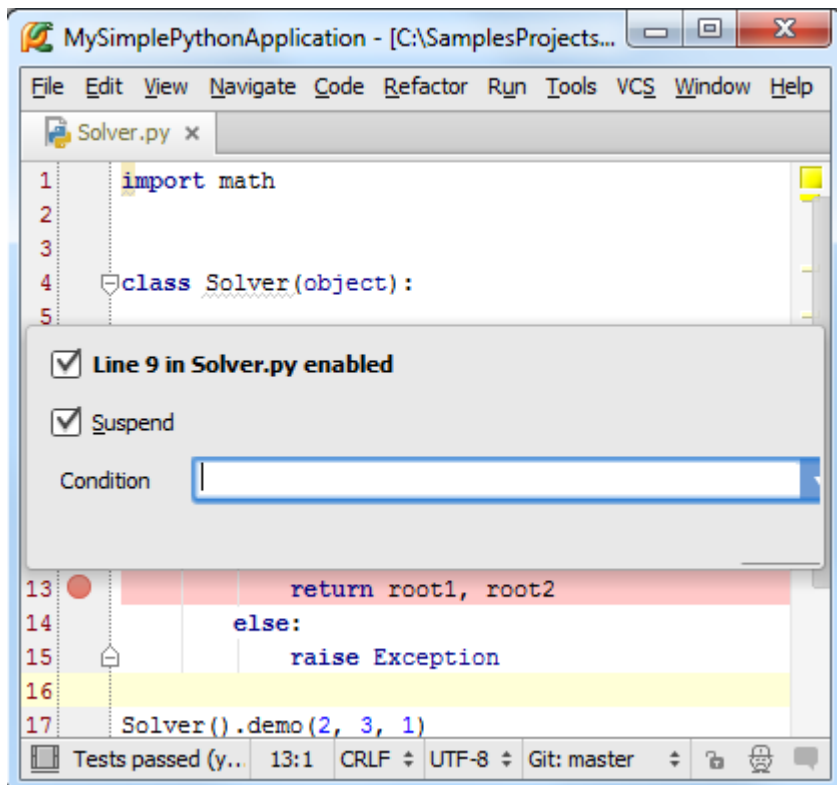
方法非常简单，单击代码左侧的空白灰色槽即可：



注意断点会将对应的代码行标记为红色，这种颜色标记目前还不能被用户所更改，我们会尽快出台解决方案。

顺便说一句，取消断点的操作也很简单，在同样位置再次单击即可。

当你将鼠标指针悬停在断点上方时，PyCharm 会显示断点的关键信息，行号以及脚本属性，如果你希望更改该断点的属性，右击断点：

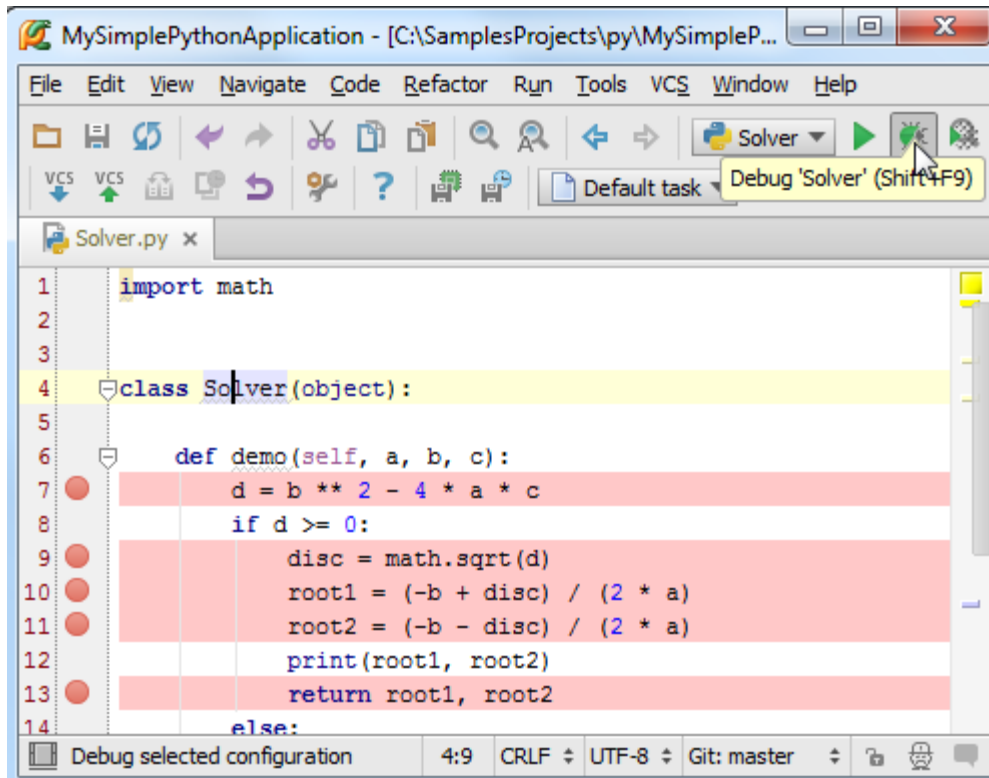


可以尝试对断点属性进行个性化更改，然后观察图标的变化。

13、代码调试

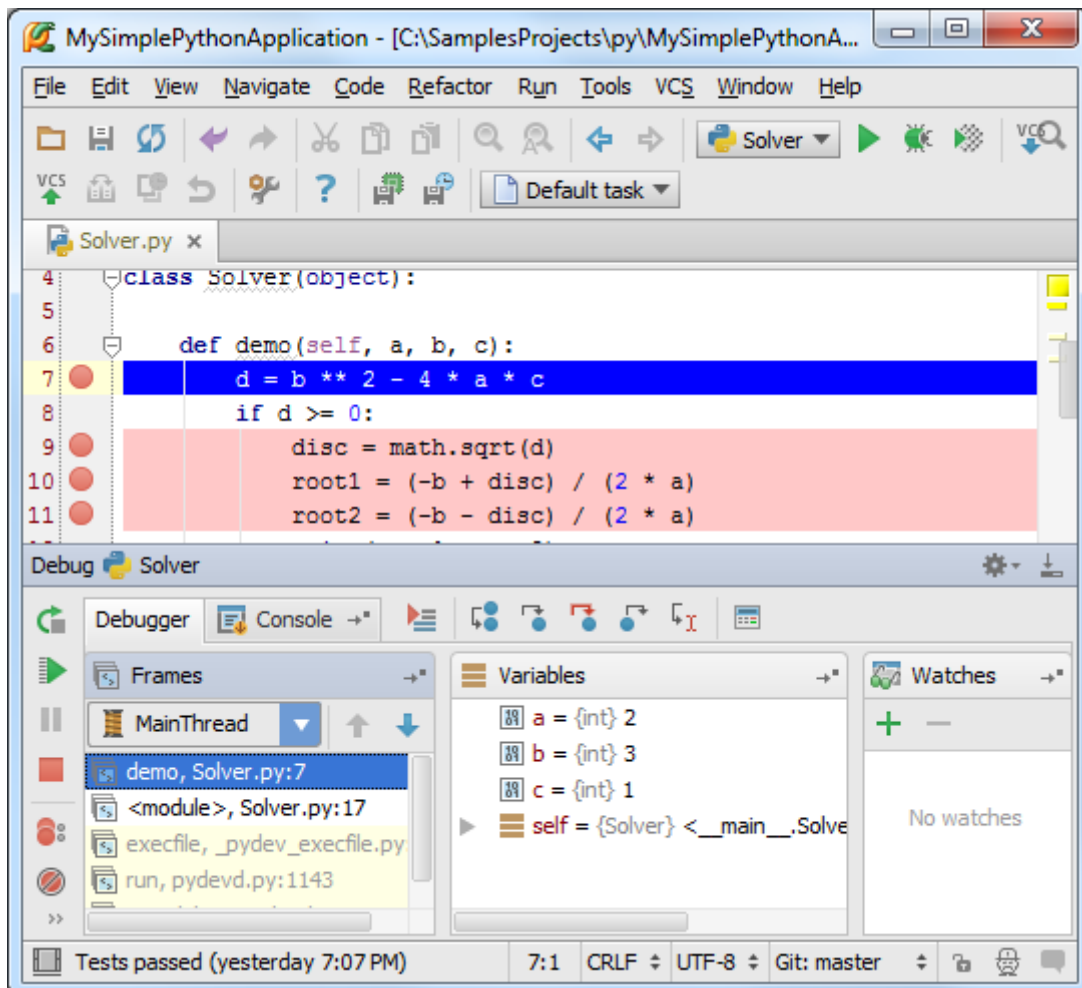
接下来，我们正式开始对代码进行调试。

首先从配置文件组框中选择同名的'Solver'文件作为当前调试的配置文件，然后单击调试按钮（绿色甲壳虫样式的按钮）：



接下来会 Pycharm 会执行以下操作：

- (1) PyCharm 开始运行，并在断点处暂停
- (2) 断点所在代码行变蓝，意味着 Pycharm 程序进程已经到达断点处，但尚未执行断点所标记的代码。
- (3) [Debug tool window](#) 窗口出现，显示当前重要调试信息，并允许用户对调试进程进行更改。



虽然 Pycharm 使用手册中已经完整提供了调试窗口中所有控件的功能信息，我们这里仍然对其进行简要介绍。我们发现窗口分为两个选项卡：Debugger tab and the Console tab。

(1) Debugger 窗口分为三个可见区域：Frames, Variables, 和 Watches。这些窗口列出了当前的框架、运行的进程，方便用户查看程序空间中变量的状态等。当你选择一个框架，就会显示出相关的变量信息，当然这些区域都是可以折叠隐藏的。

(2) Console 窗口显示当前的控制台输出信息，默认这个窗口位于 Debugger 之下，可以通过单击其标签将其前置显示。

当然我们可以改变这些窗口的摆放位置，如果你不喜欢程序的默认排版的话。具体参加 [Moving tabs and areas](#) 章节。

Debugger 窗口的工作模式：

OK，现在程序暂停在了第一断点处，Frames 窗口显示的是 Solver 脚本的第 7 行代码所对应的进程 demo，相关变量 a、b、c 已经定义，但变量 d 尚未进行定义。接下来？

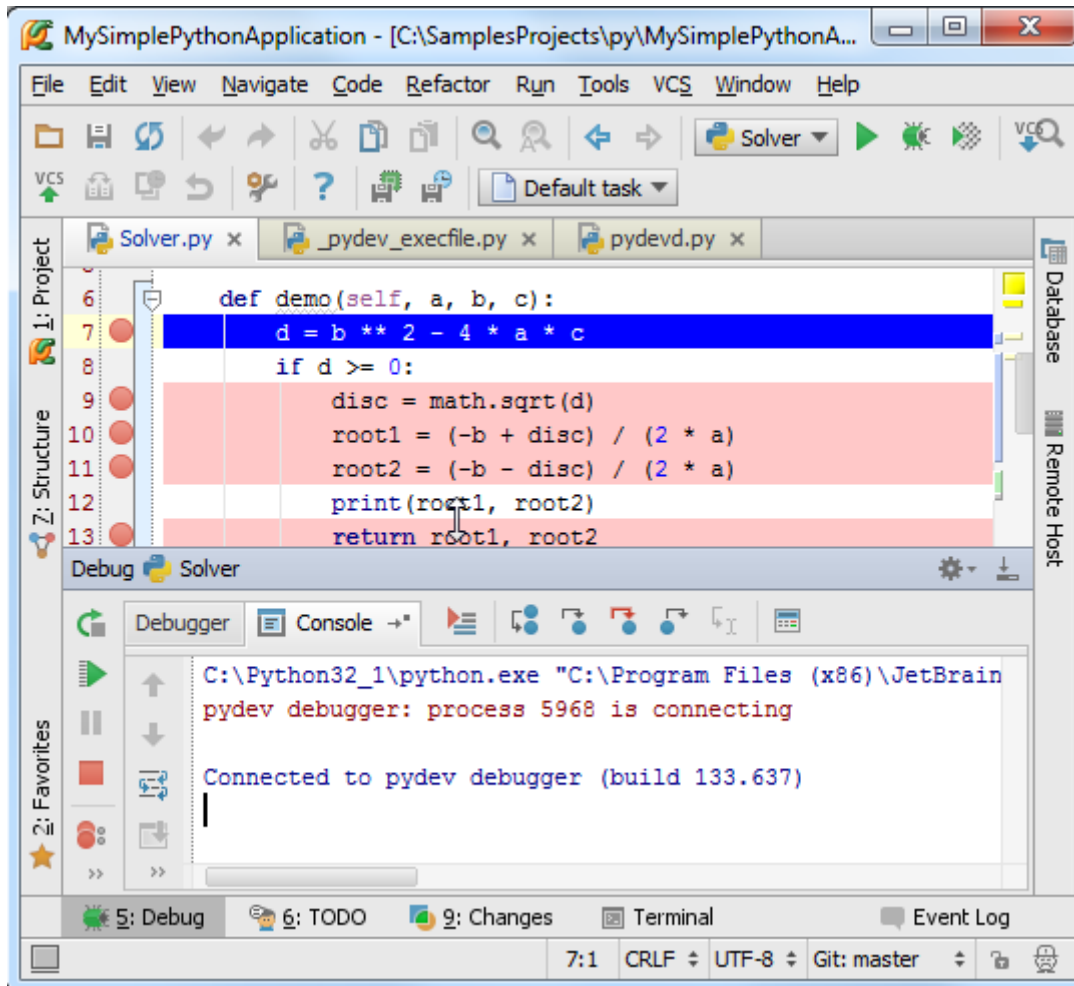
按下 F9（或者左侧工具栏的绿色箭头），程序会继续运行到下一断点处，通过这种方式你可以将每个断点都运行一遍，观察变量的变化。

更多有关 Debugger 窗口的信息参见软件手册：[product documentation](#)

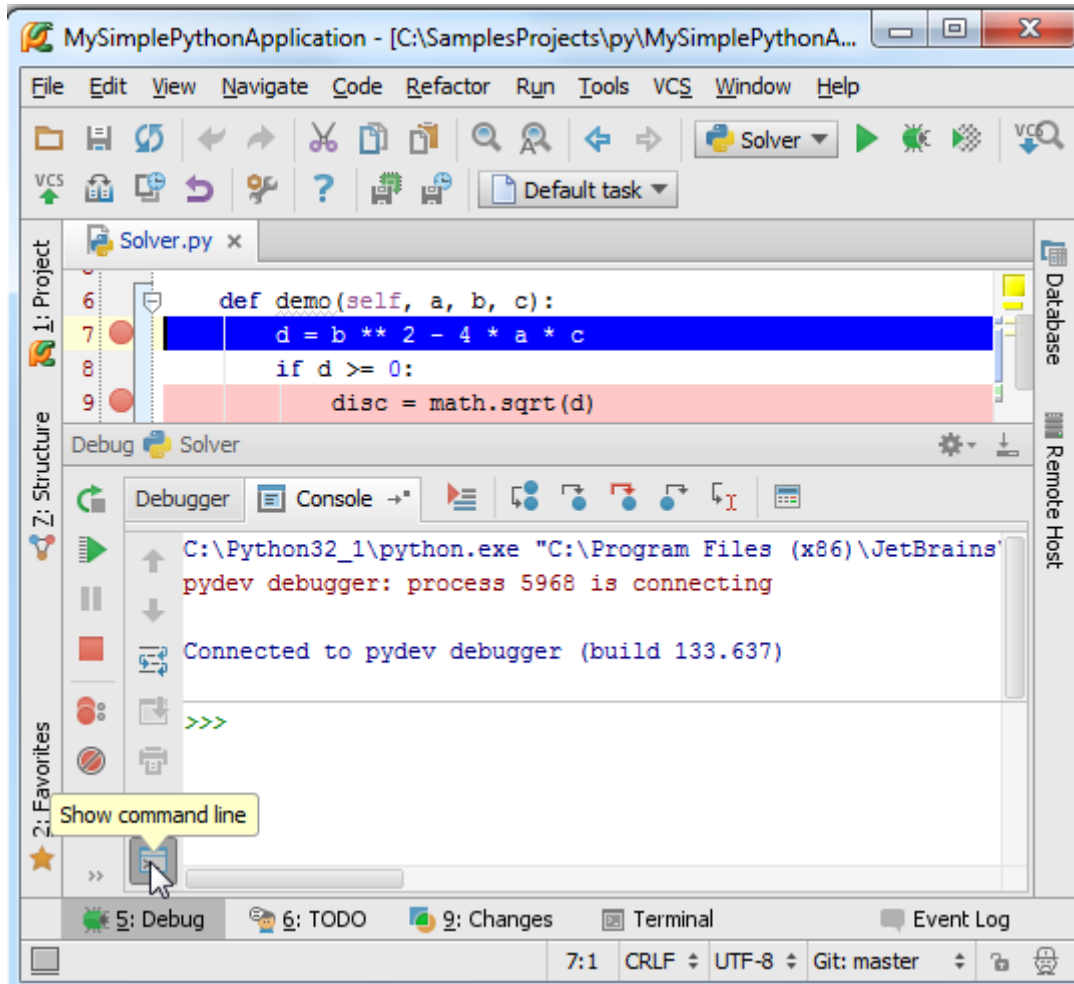
Console 窗口的工作模式：

为什么需要用到 Console 窗口呢？当我们需要查看程序给出的错误信息，或者进行一些额外的临时运算时，就需要在这个窗口里面进行。

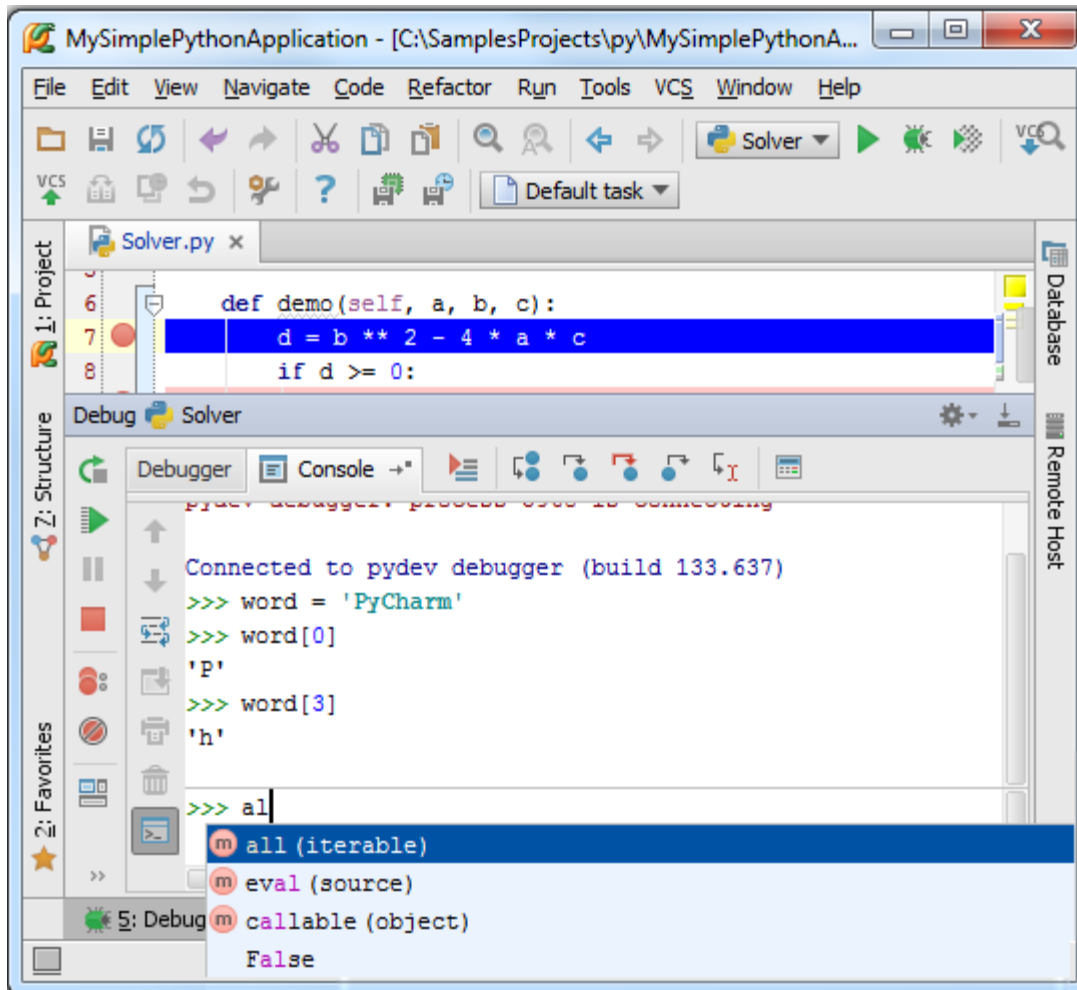
单击 Console 选项卡使其前置：



然后单击左侧工具栏中的命令符按钮，显示 Python 的命令提示符：

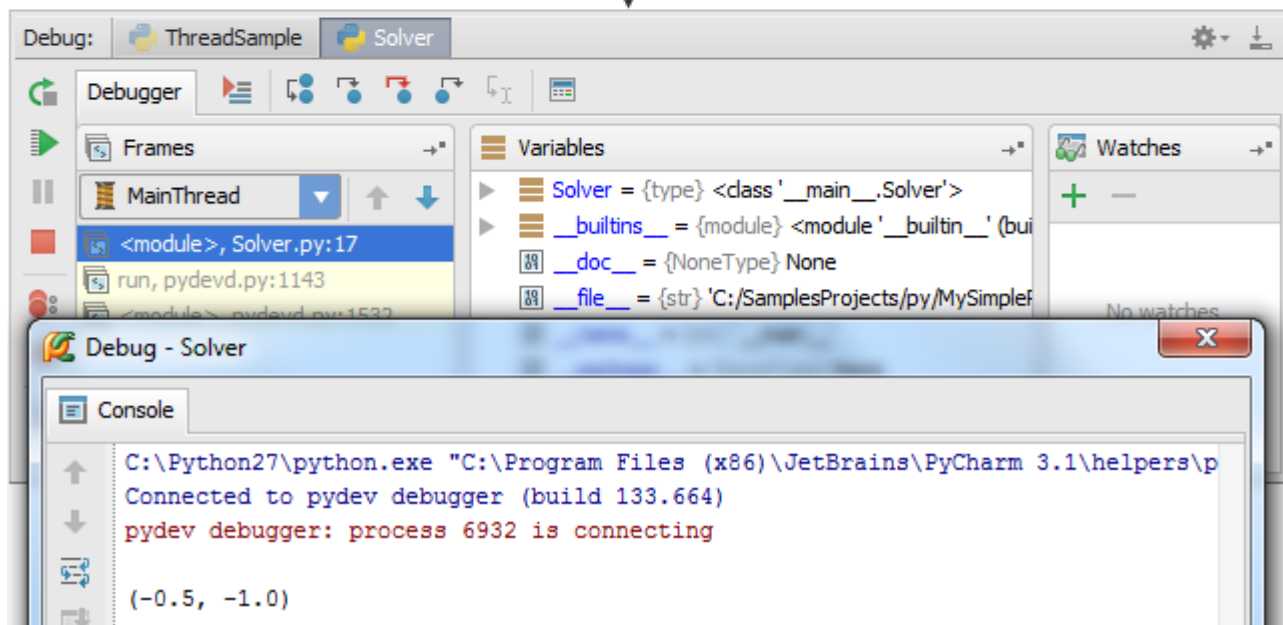
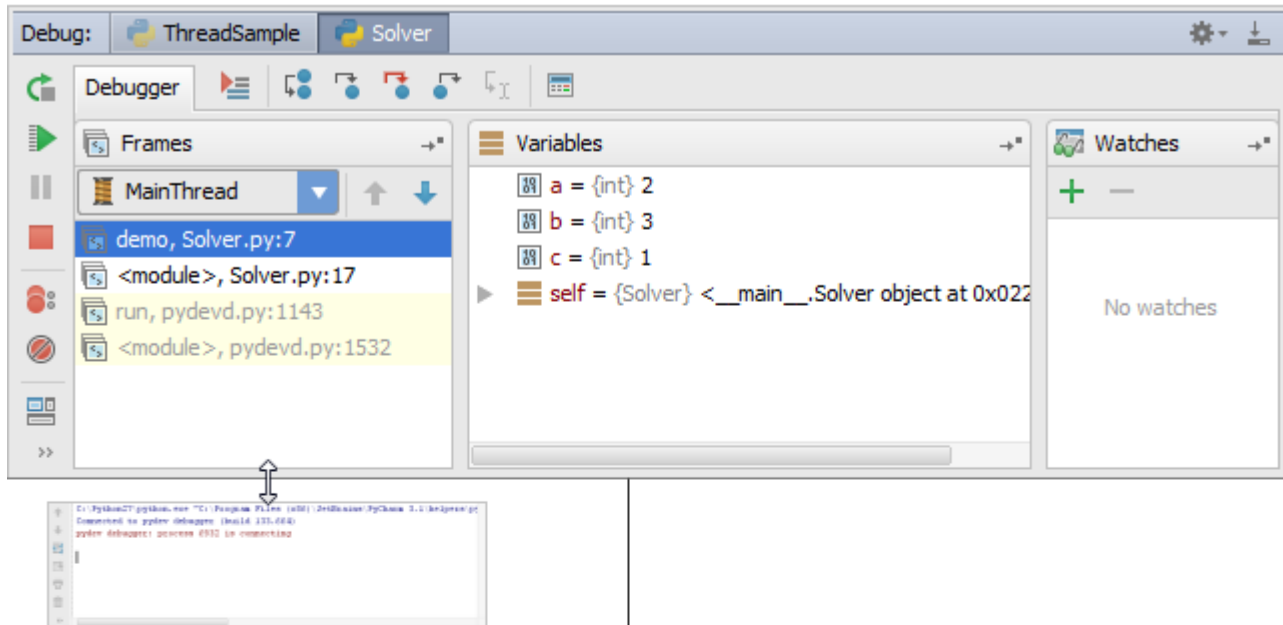


此时激活了控制台机制，尝试在其中执行一些 Python 命令：



注意到控制台窗口提供了代码的拼写提示 (Ctrl+Space) 以及历史记忆 (Up/Down keys) 功能, 更多信息参见: [Using Debug Console](#)

最后, 如果你希望 Console 窗口一直处于可用状态, 只需将其移动成为一个单独的窗口即可:

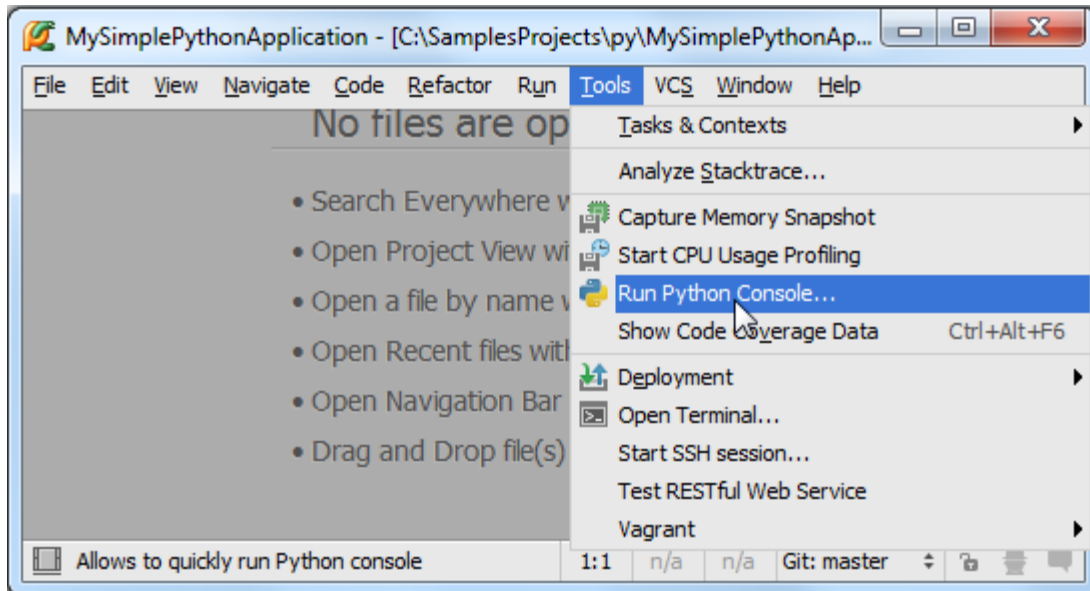


14、再次运行

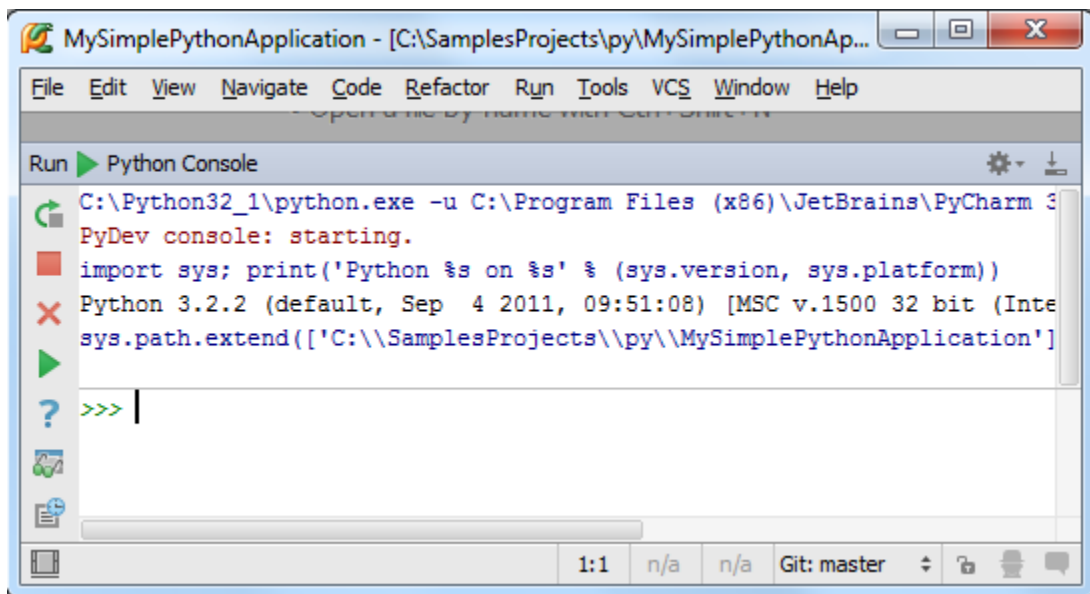
在完成了本次调试运行并再次加载调试配置文件之后，我们可以再次运行调试，单击工具栏的 run 按钮即可。

15、REPL——在控制台界面调试程序

最后，如果你更习惯工作于控制台环境下，也可以将 Pycharm 设置成为控制台模式。在主菜单中选择 Tools → Run Python Console...来加载控制台：

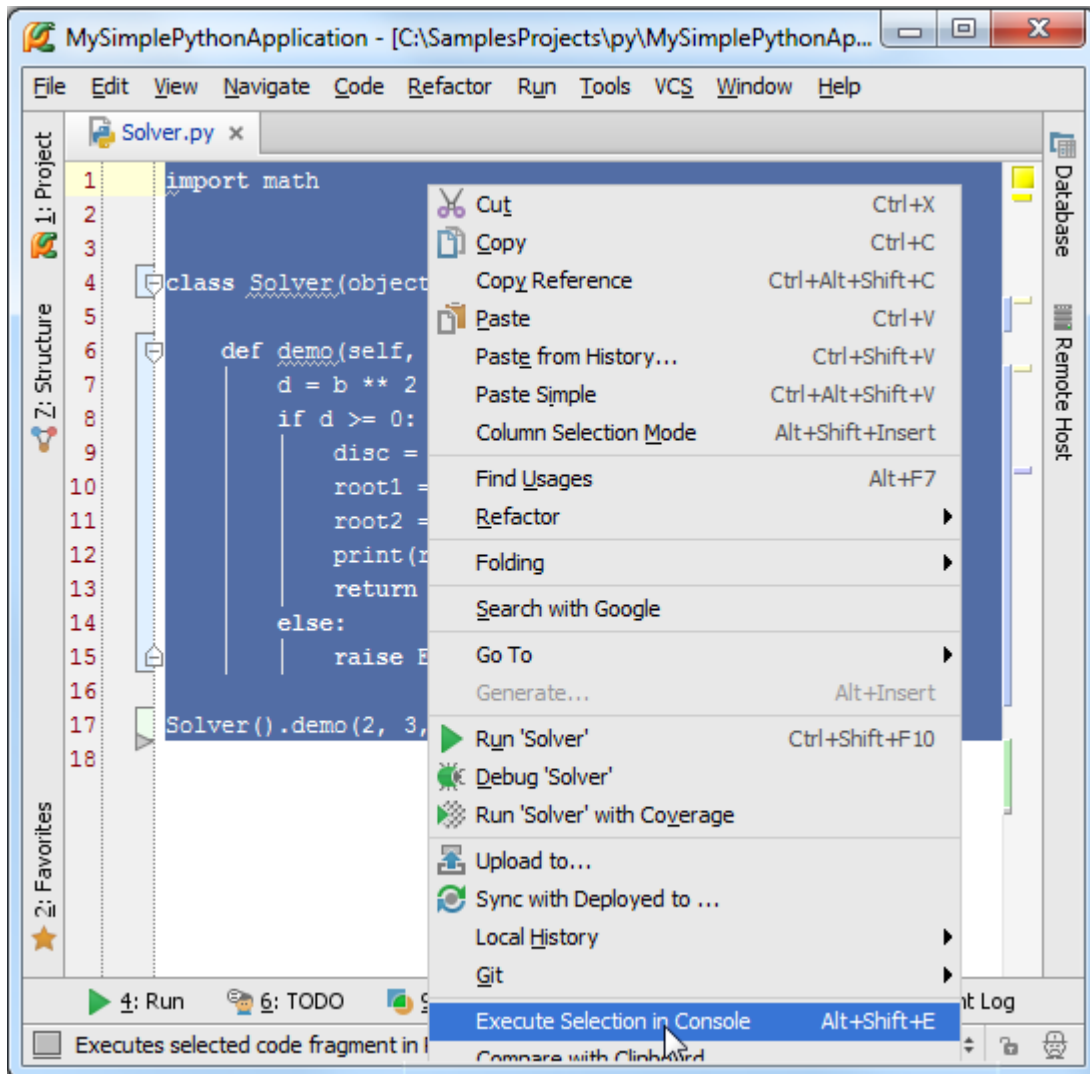


此时 console 窗口将会被激活，并显示为一个单独的窗口：

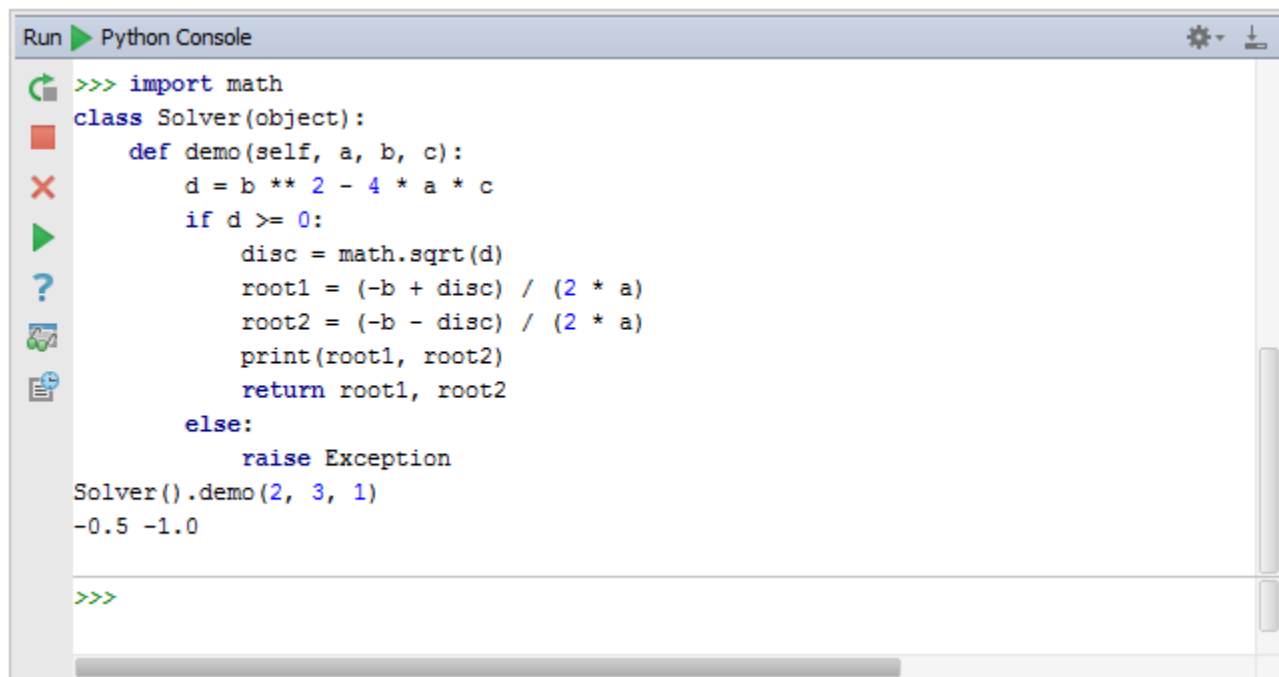


在这个控制台窗口中我们可以做很多有意思的事情，接下来我们演示如何将最近编写的 Solver.py 文件中的代码导入到控制台：

打开 Solver.py 文件(打开的方法多种多样,例如 Ctrl+E - View → Recent Files), 全选文件中的代码内容(Ctrl+A, or Edit → Select All) , 然后按下 Alt+Shift+E (或者右击在弹出的快捷菜单中选择 Execute Selection in Console) :



此时，Pycharm 就会自动将选中的代码导入到控制台界面，方便我们对其进行编辑：



最全 Pycharm 教程（4）——有关 Python 解释器的相关配置

[最全 Pycharm 教程（1）——定制外观](#)

[最全 Pycharm 教程（2）——代码风格](#)

[最全 Pycharm 教程（3）——代码的调试、运行](#)

1、准备工作

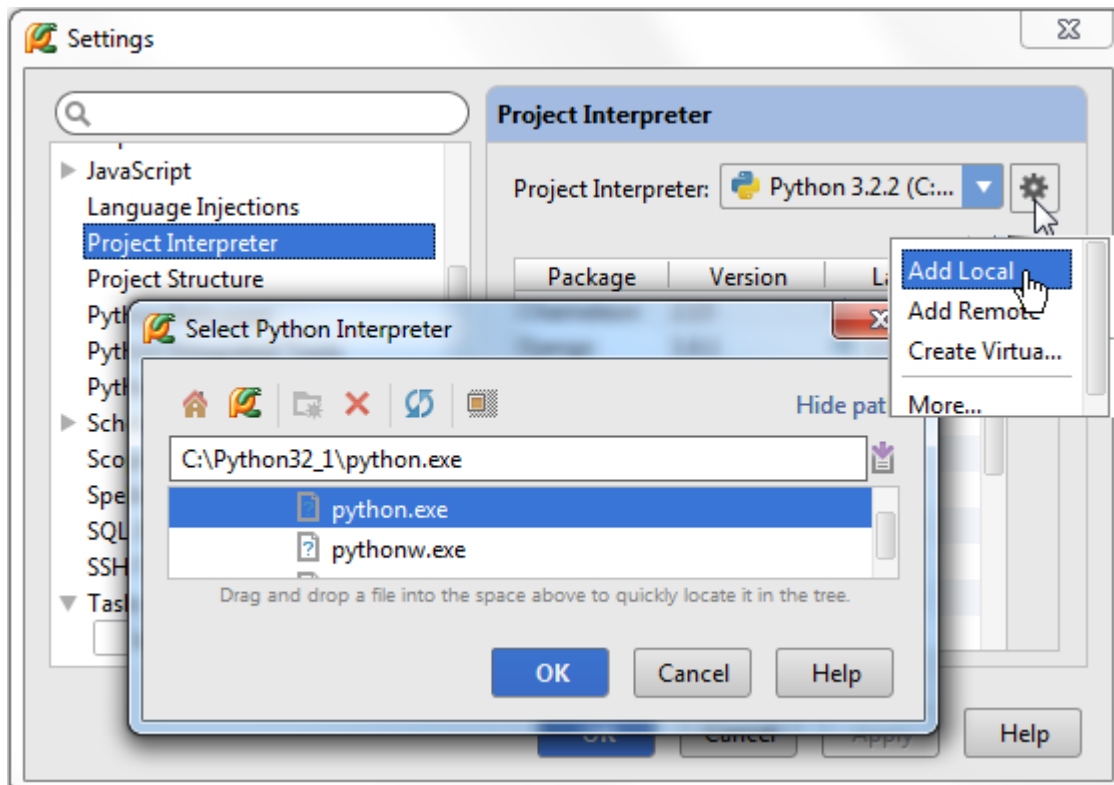
- (1) Pycharm 版本为 3.4 或者更高。
- (2) 电脑上至少已经安装了一个 Python 解释器。
- (3) 如果你希望配置一个远程解释器，则需要服务器的相关支持。

2、本地解释器配置

配置本地解释器的步骤相对简洁直观：

- (1) 单击工具栏中的设置按钮。
- (2) 在 Settings/Preferences 对话框中选中 **Project Interpreter** 页面，在 Project Interpreter 对应的下拉列表中选择对应的解释器版本，或者单击右侧的设置按钮手动添加。
- (3) 在接下来的情况下，选择 Add Local 选项，然后选择预期的解释器（Python 的可执行文件）。

值得一提的是，对于一些预先定义好的虚拟环境，同样可以将其作为解释器进行添加。

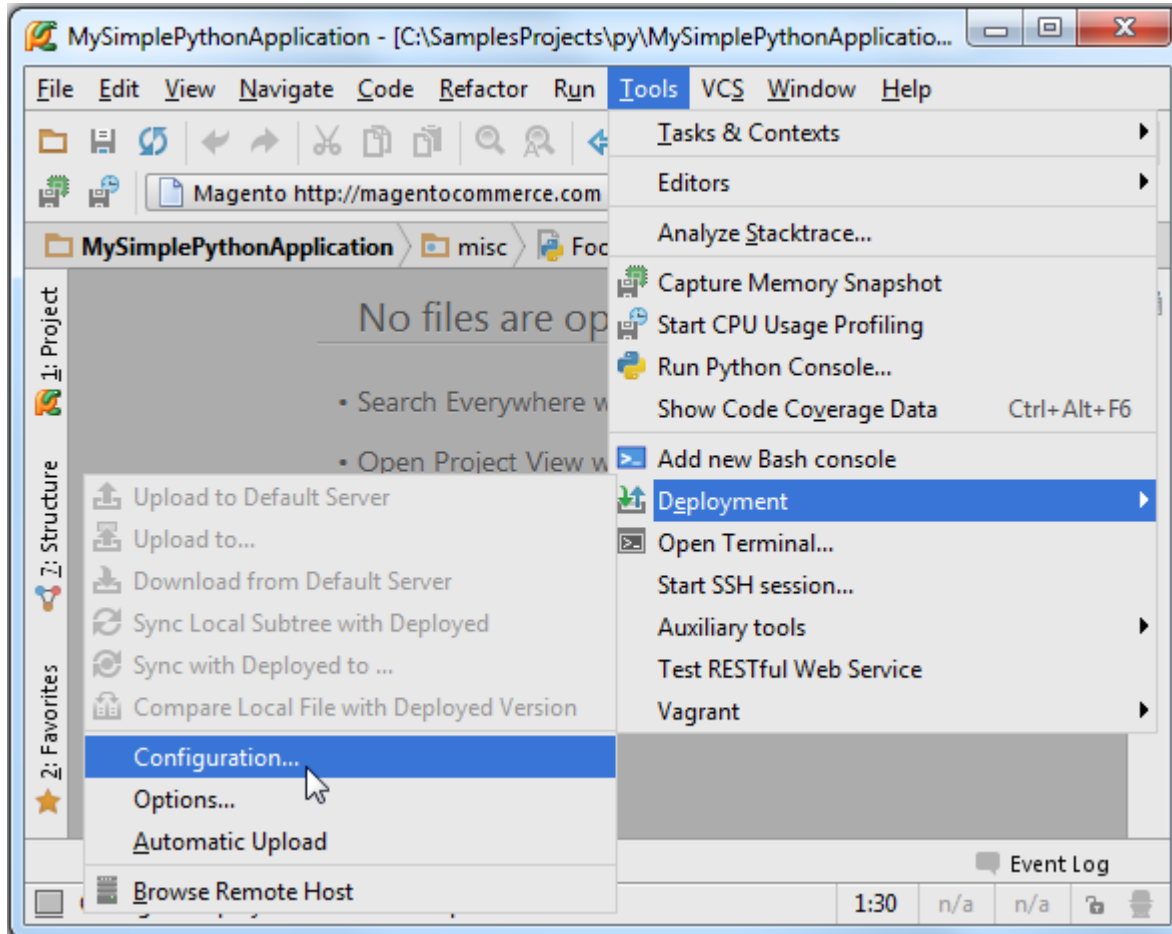


3、远程解释器配置

在配置远程解释器时，使用基于 SSH 连接的设置方法（确保服务器已经提供了响应的远程支持）。

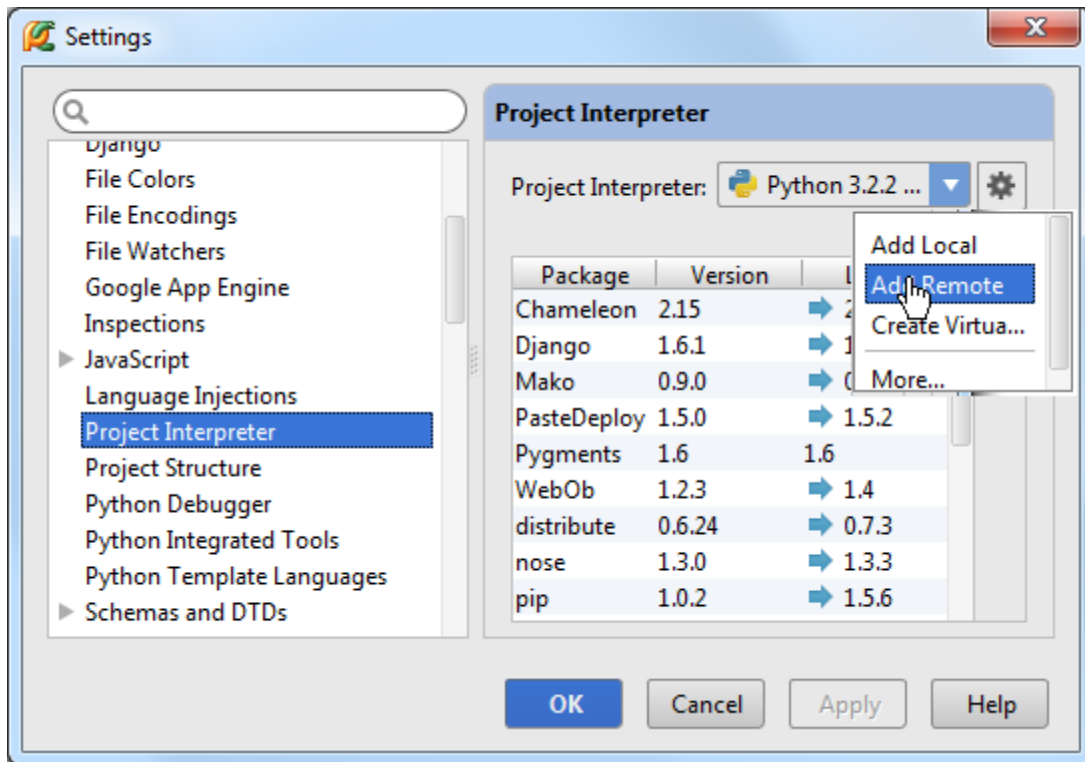
4、通过已有的部署设置来配置远程解释器

首先，我们需要一台服务器，可以通过主菜单的 Tools | Deployment，然后单击 Configuration 来定义一个：



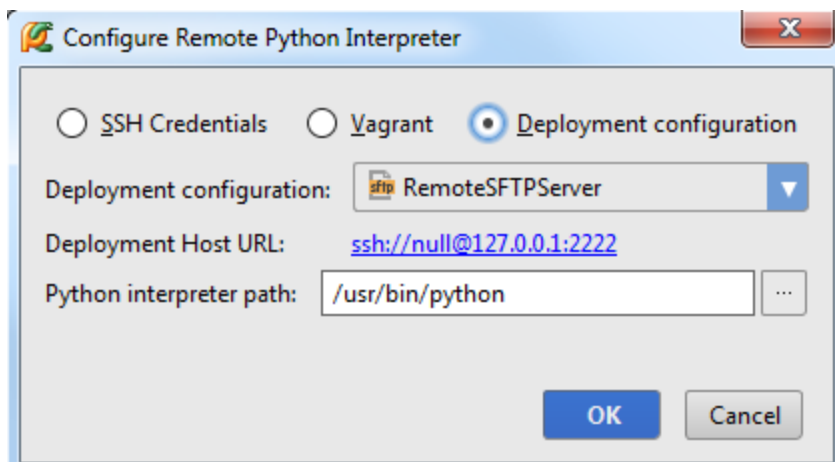
接下来在 [Deployment](#) 对话框中单击绿色的加号来创建一个服务器，输入名称，选择对应类型（一般为 SFTP），以及其他一些必要设置（host, port, login name 等等）。确认无误后单击 Test connection，弹出 Connection successful! 的消息框后说明解释器连接成功。

接下来，在主工具栏中单击设置按钮，在 Settings/Preferences 对话框中打开 [Project Interpreter](#) 页面，单击设置图标然后选择 Add Remote：



在 Configure Remote Python Interpreter 对话框中, 单击 Deployment configuration 按钮来设置已有的 SSH 服务器 (比如你之前定义的这个)。

在列表中选择想要的远程服务, 选择完成后会发现所有的服务器设置已经自动完成填充。

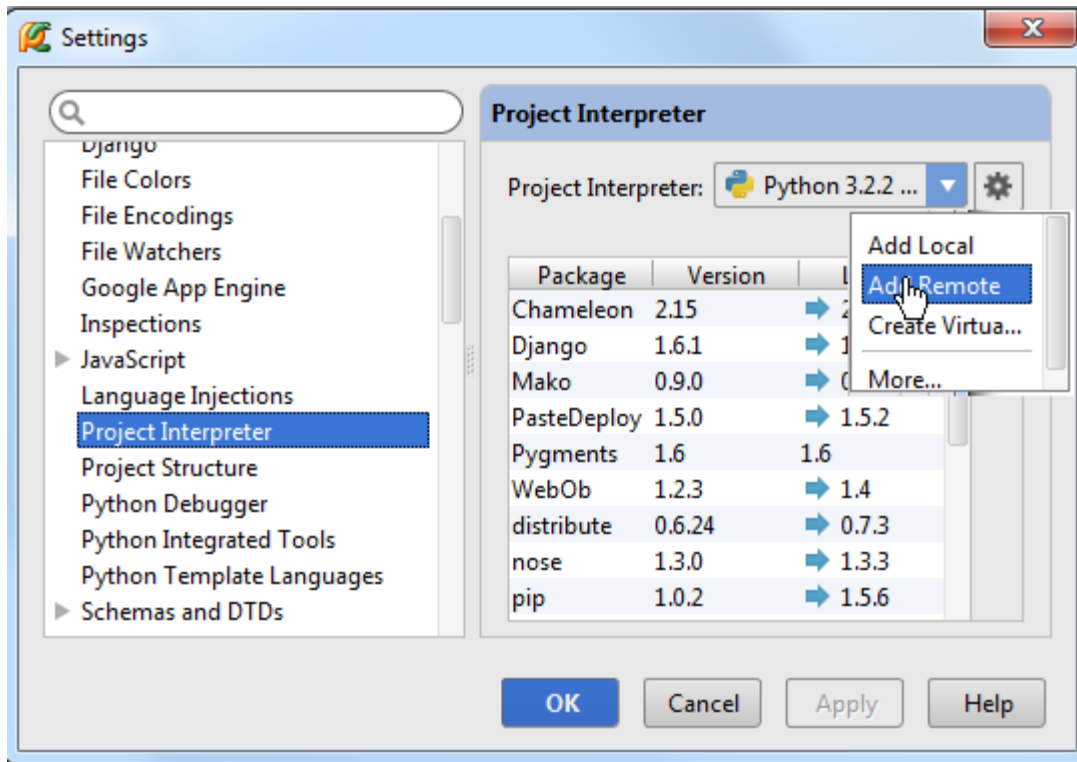


此时这个远程解释器可以作为当前工程的解释器来使用了, 注意这里所有的远程解释器在命名时都会加一个前缀 "Remote"。

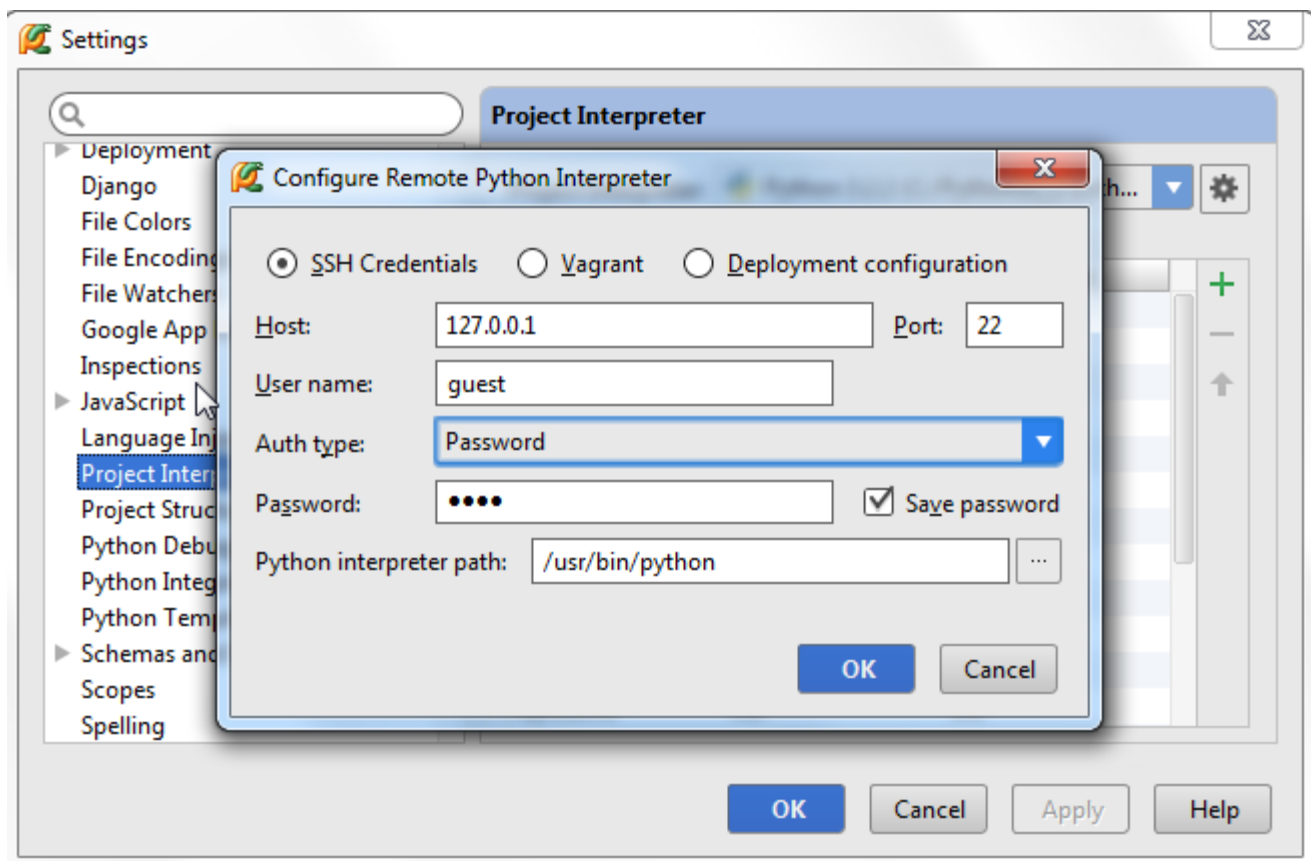
5、基于 SSH 证书的远程解释器

如果你没有提前定义服务器设置, 你也可以按照以下步骤手动建立特定的连接:

1、单击主工具栏的设置按钮打开 Settings/Preferences 对话框, 选择 Project Interpreter 页面, 单击设置按钮然后选择 Add Remote:



接下来，在 Configure Remote Python Interpreter 对话框中，选中 SSH credentials，然后键入服务器的 Host、端口号、用户姓名等等：



此时这个远程解释器可以作为当前工程的解释器来使用了，注意这里所有的远程解释器在命名时都会加一个前缀 "Remote"。

6、基于 virtual box 的远程解释器

另外一种定义远程解释器的方法是通过 Vagrant configuration 文件。在使用 virtual boxes 之前提前准备一些工作，所以在开始之前先确认一下几方面问题：

- (1) Vagrant 已经在电脑上正确安装，并且已经创建好了相关的基础结构。
- (2) Oracle's VirtualBox 已经在电脑上正确安装。
- (3) 确保将以下两个可执行文件的路径添加到系统的环境变量中：

Vagrant 安装目录下的 vagrant.bat 文件，这一步应该会由安装程序自动来完成。

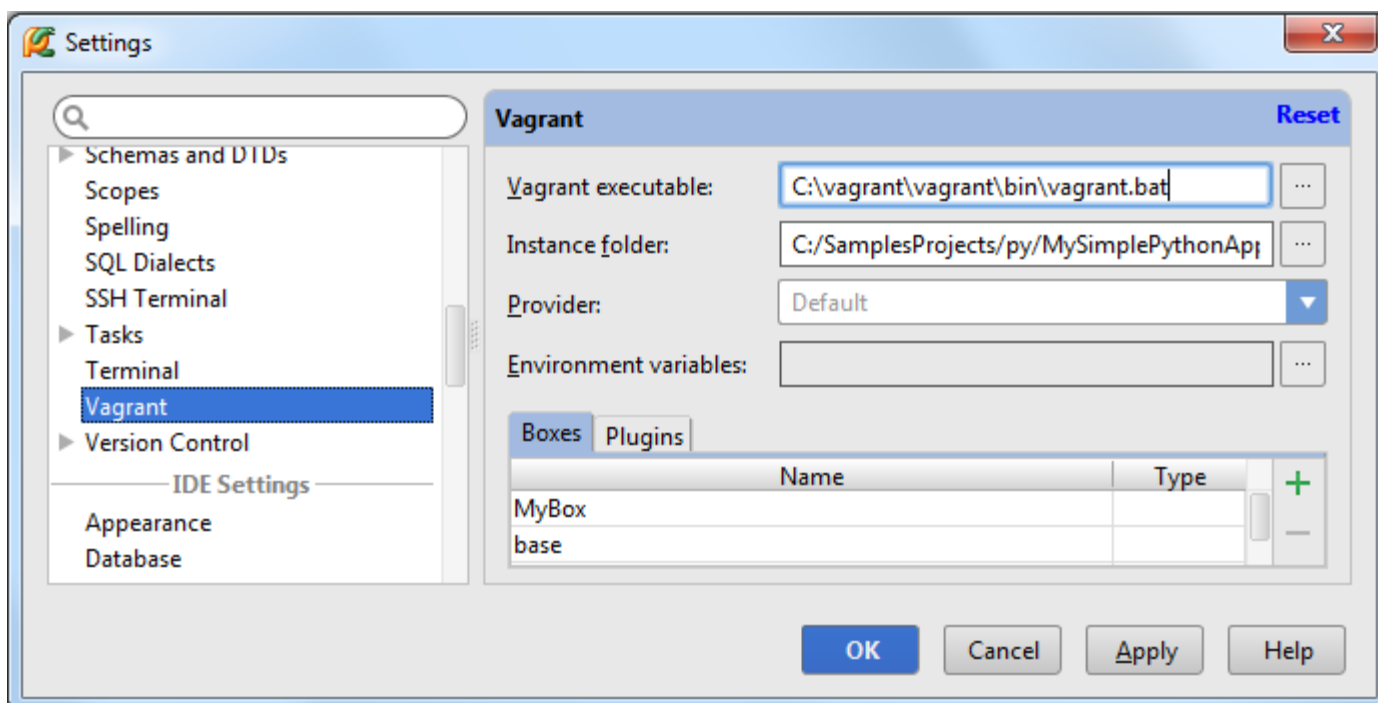
Oracle's VirtualBox 安装目录下的 VBoxManage.exe 文件路径。

最后还要确保 Vagrant 的相关插件能够正常使用。

首先，你需要一个 virtual box，这需要我们手动进行配置，但 Pycharm 提供了一些列辅助工具来是的我们可以在当前 IDE 环境下完成设置。

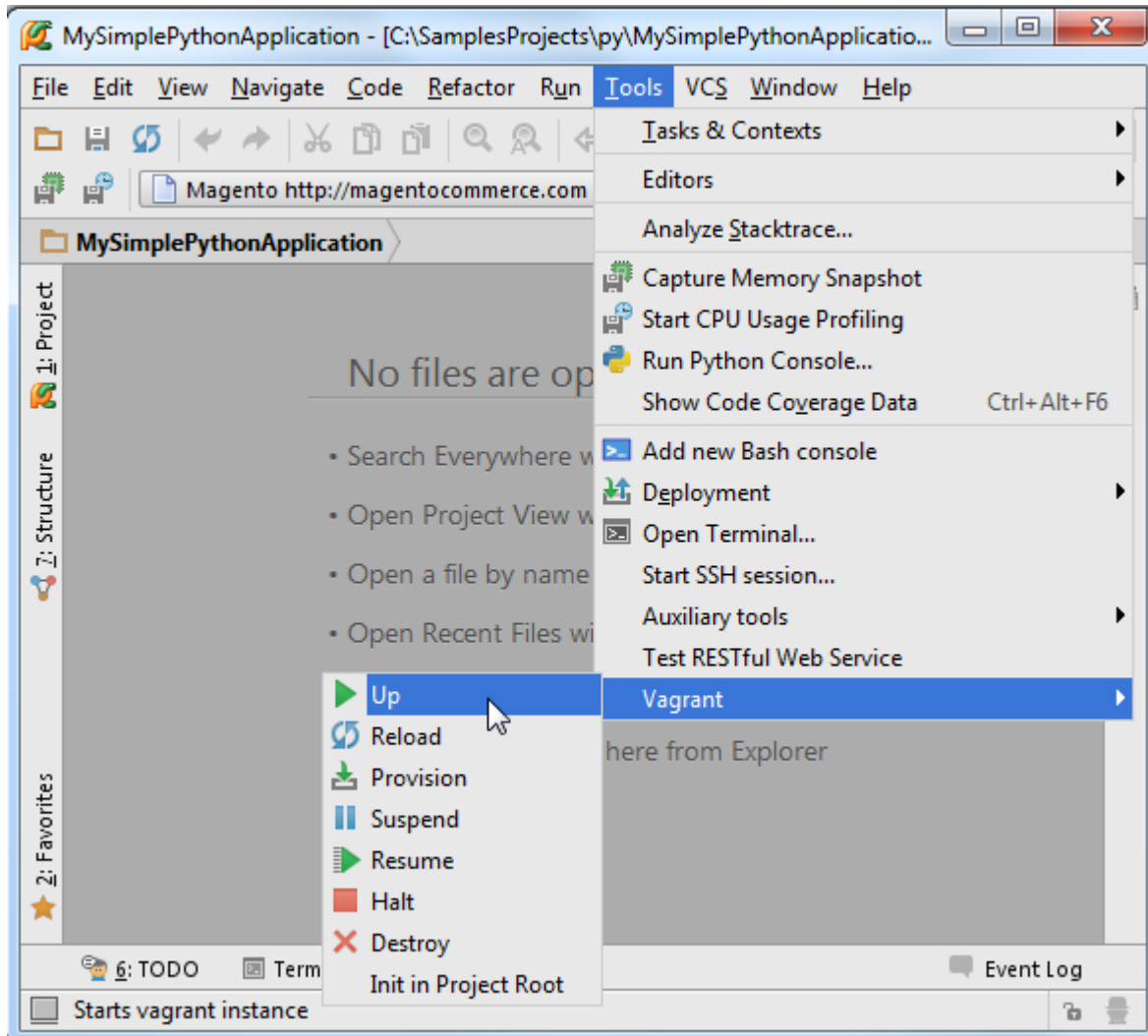
在主工具栏中单击设置按钮，进入 Settings/Preferences 对话框，打开 Vagrant 页。

留意 Vagrant 可执行文件的路径以及 Vagrant 实例所在的文件夹路径：

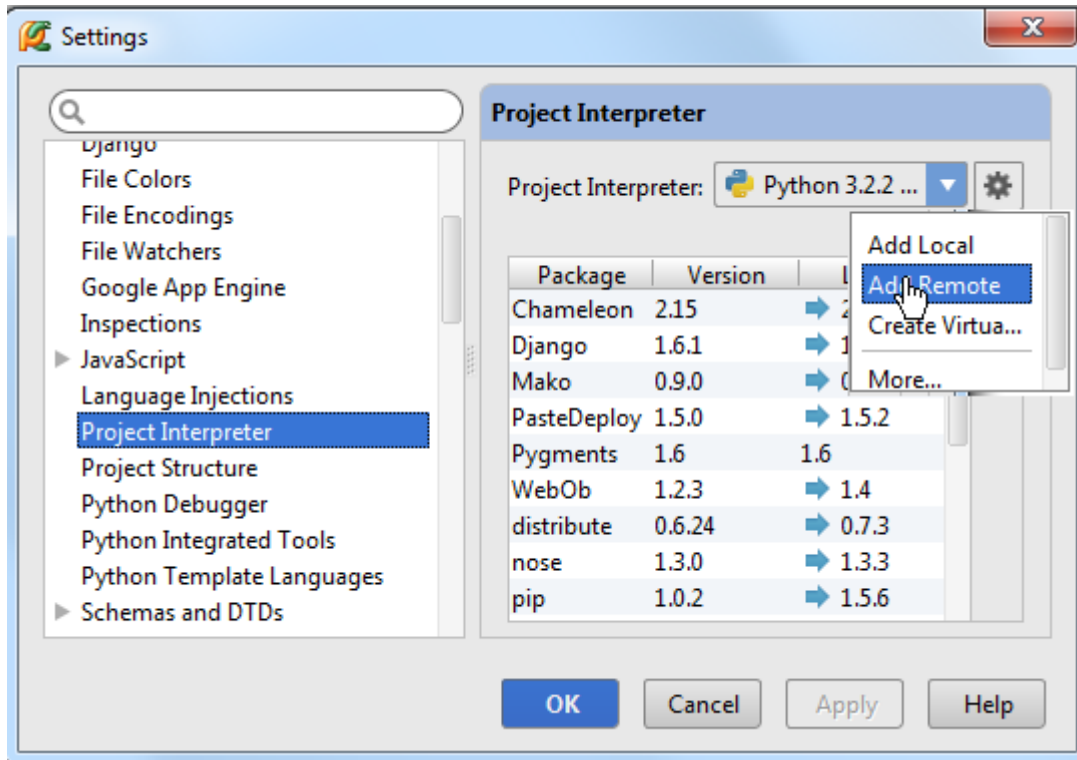


如果已经实现定义了 virtual box，它将会出现在下拉列表中以便我们进行选择。如果当前没有合适的 virtual box 可选，则可以通过单击绿色的加号来新建一个。

接下来需要初始化 Vagrant box。在主菜单上单击 Tools | Vagrant，选择 Init in Project Root，选择 **vagrant up** 命令：

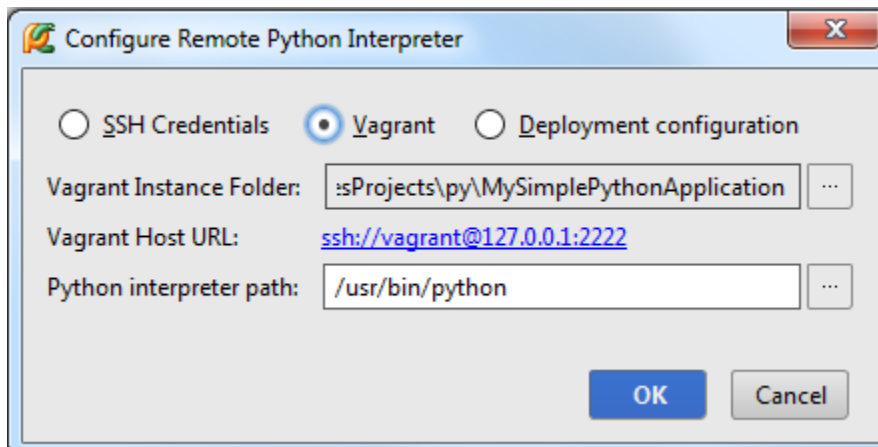


接下来再次进入 Settings/Preferences 对话框，打开 [Project Interpreter](#) 页面然后选择 Add Remote:



在 Configure Remote Python Interpreter 对话框中自定义服务器的相关设置, 这些设置可以通过之前定义的配置文件进行替换, 因此选中 Vagrant 选项。

所有的服务器设置会自动填充如下:

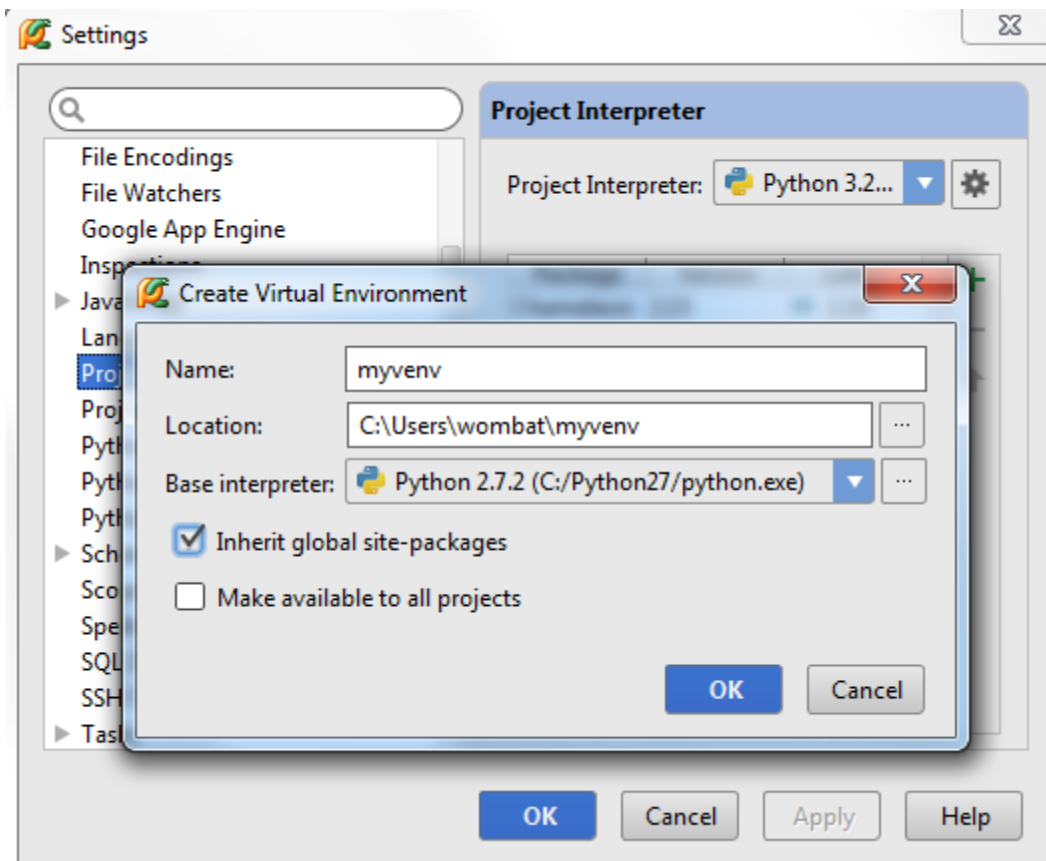


此时这个远程解释器可以作为当前工程的解释器来使用了, 注意这里所有的远程解释器在命名时都会加一个前缀 "Remote"。

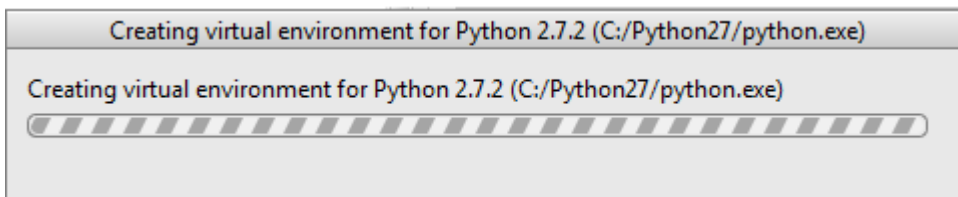
更多信息参见 [dedicated Vagrant tutorial](#)。

7、创建虚拟环境

- (1) 打开 [Project Interpreter](#) 页面 (通过单击工具栏上的设置按钮)。
- (2) 单击设置图标并选择 Create Virtual Environment。
- (3) 在 Create Virtual Environment 对话框中输入新的虚拟环境的名称、位置, 同时制定虚拟环境所依赖的 Python 解释器:

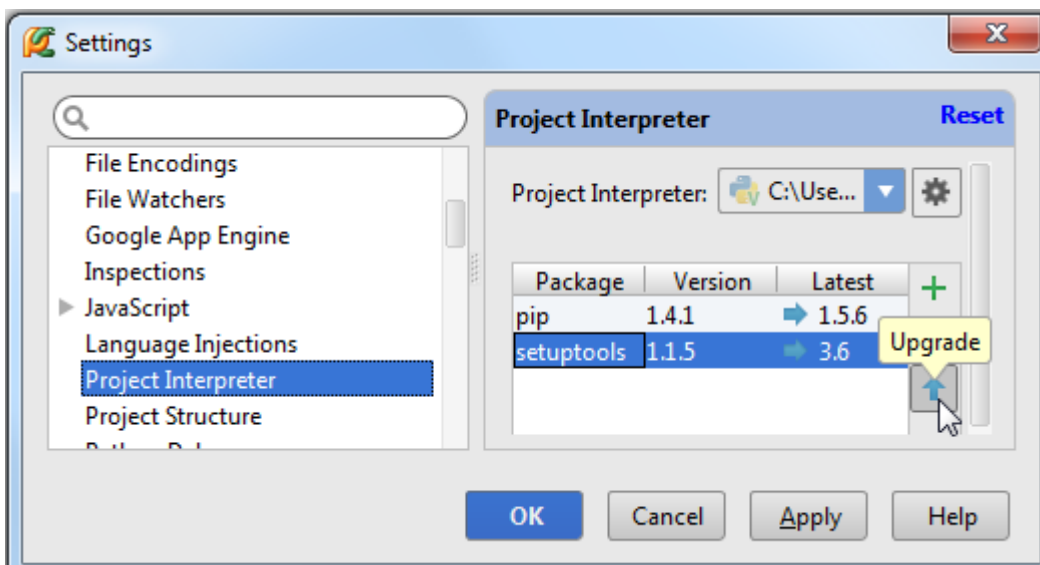


在创建虚拟环境的过程中可能需要花费一些时间，Pycharm 会给出进度条来指示当前的创建进程：

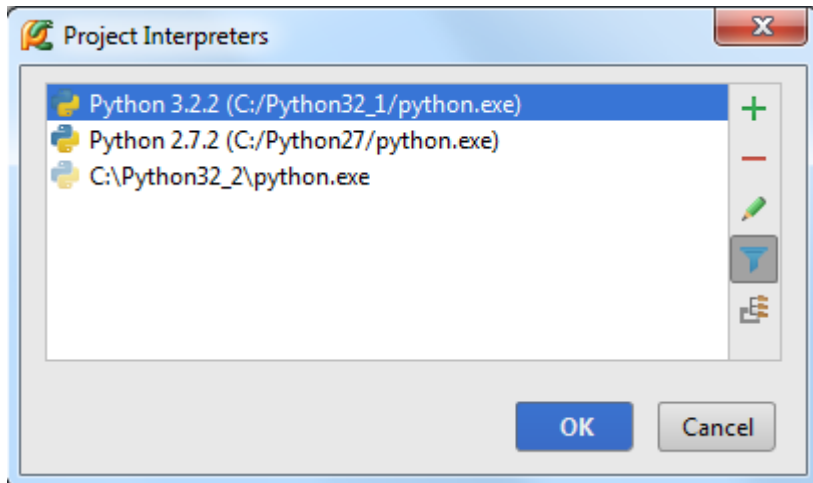



8、第三方库以及路径的配置

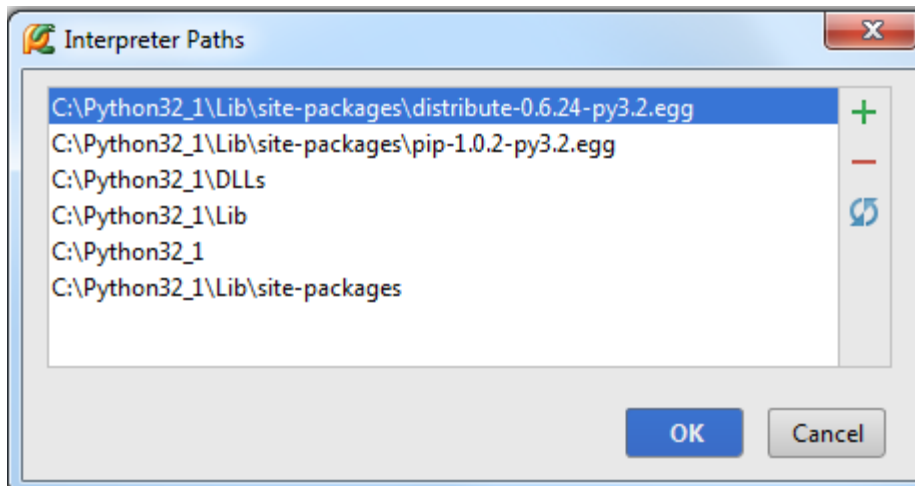
在配置好虚拟环境之后，你将会看到当前安装的所有第三方库，Pycharm 会列出当前安装的每个第三方库的版本以及响应的最新版本，你可以决定是否对其进行升级：




为了查看安装路径，可以通过单击对话框中的设置按钮，选择 More，此时可以查看所有可用的 Python 解释器：



选中一个解释器，然后单击右侧工具栏中的  按钮来查看其对应的路径结构：



如果一个解释器已经更新过，最好通过单击  来更新其路径。

最全 Pycharm 教程（5）——Python 快捷键相关设置

[最全 Pycharm 教程（1）——定制外观](#)

[最全 Pycharm 教程（2）——代码风格](#)

[最全 Pycharm 教程（3）——代码的调试、运行](#)

[最全 Pycharm 教程（4）——有关 Python 解释器的相关配置](#)

1、主题

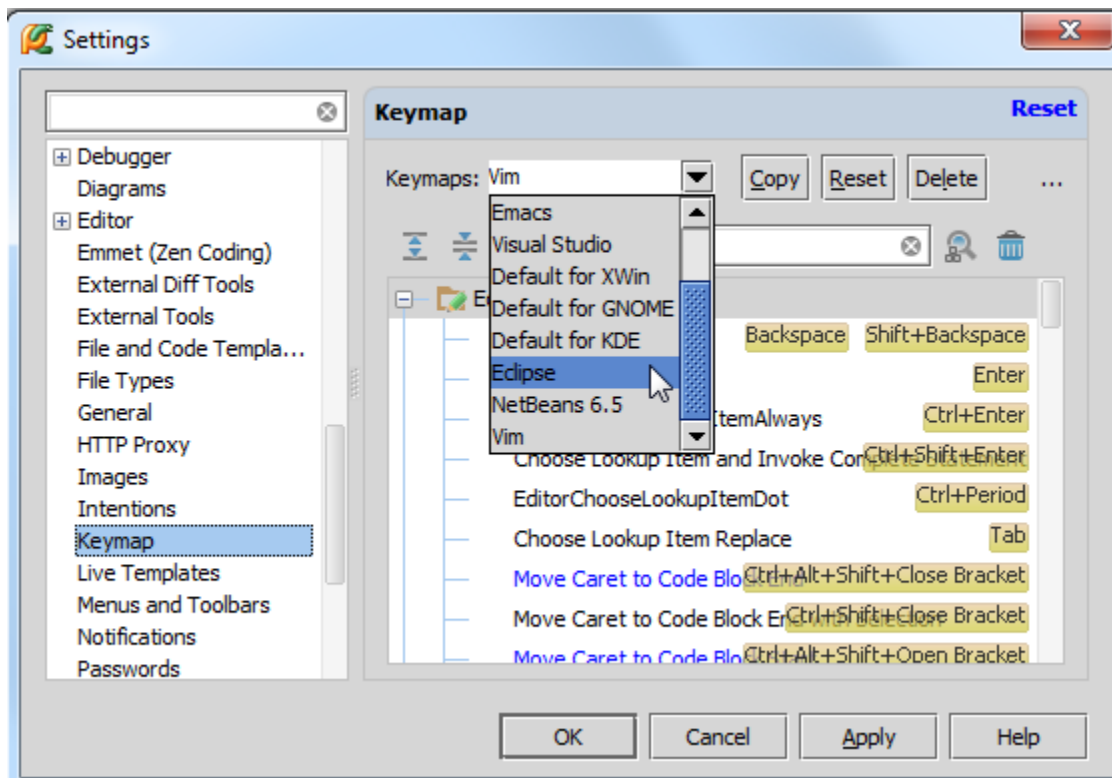
毫无疑问 Pycharm 是一个具有强大快捷键系统的 IDE，这就意味着你在 Pycharm 中的任何操作，例如打开一个文件、切换编辑区域等，都可以通过快捷键来实现。使用快捷键不仅能提高操作速度，看起来也会非常酷。然而，如果你已经习惯使用一些快捷键方案，你的手指就会习以为常，改变起来就比较困难。这部分教程即是介绍如何根据你的习惯来量身定制 Pycharm 快捷键设置，使你用起来得心应手。

对于基本的快捷键的组合、用法这里不再赘述，详情可参考 [Configuring keyboard shortcuts](#) 或者 [Keymap](#)。

2、选择一个快捷键配置方案

这一步非常简单，在主工具栏中单击设置按钮，在设置对话框中单击 [Keymap](#)。

在对应页面的下拉列表选择一个快捷键配置方案：

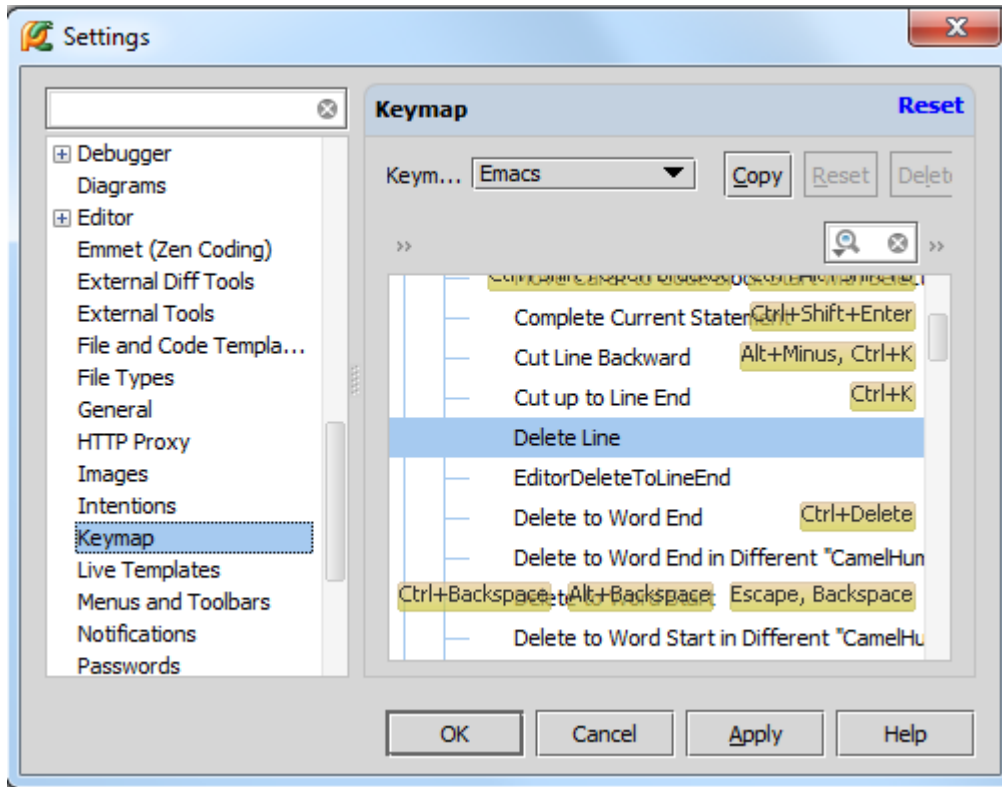


单击应用保存更改。例如我们这里选择了 Eclipse 方案，因此删除一行的快捷键就是我们所习惯的 Ctrl+D 了。

3、改变快捷键配置方案。

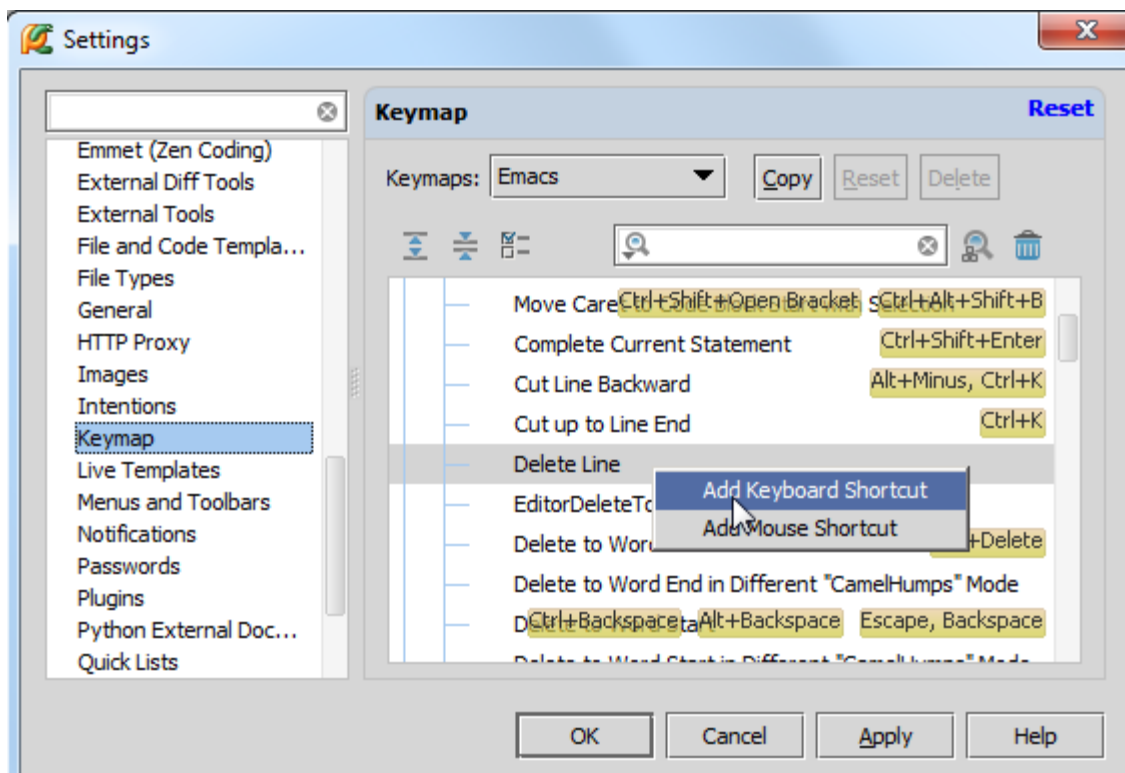
在这里我们介绍如何改变快捷键的配置方案。

设想一下情况：你选择了 Emacs 方案的快捷键配置，但这个配置方案里面并没有预先定义好如何通过快捷键来实现删除一行：

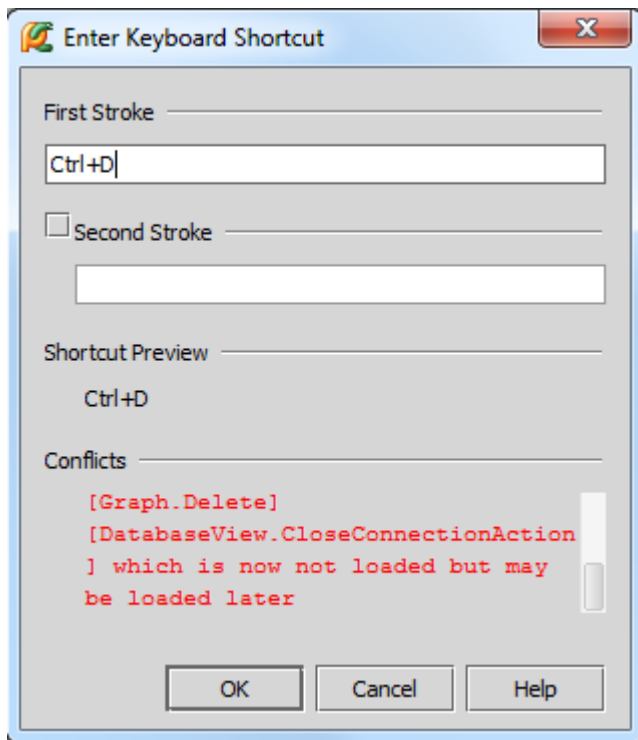


假设你希望将这个功能指定快捷键为 Ctrl+D，操作如下：

- (1) 在主工具栏中单击设置按钮，在设置对话框中单击 [Keymap](#)。
- (2) 在对应下来列表中选择 Emacs。
- (3) 在下方的快捷键功能列表中，展开 Editor Actions 节点，定位到 **Delete Line** 功能。
- (4) 右击，在快捷菜单中选择 Add Keyboard Shortcut



(5) 在打开的 **Enter Keyboard Shortcut** 窗口的 **First Stroke** 中输入你想要的快捷键组合：

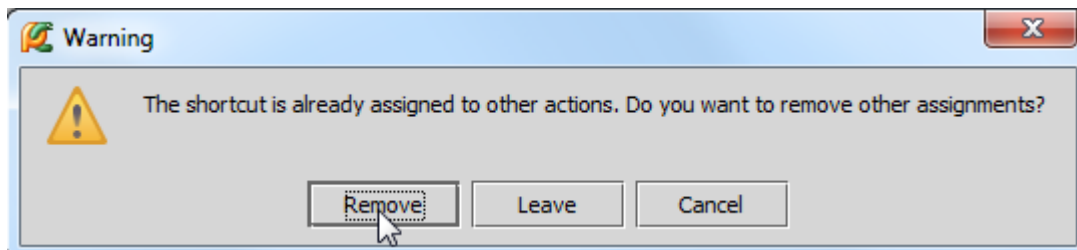


需要注意的一点是，在输入预期的快捷键时，所有来自键盘的输入都会被识别为用户的快捷键设置，举个例子，如果你想设置一个快捷键组合 **Ctrl+D**，则需要一次按下 **Ctrl** 和 **D** 键。

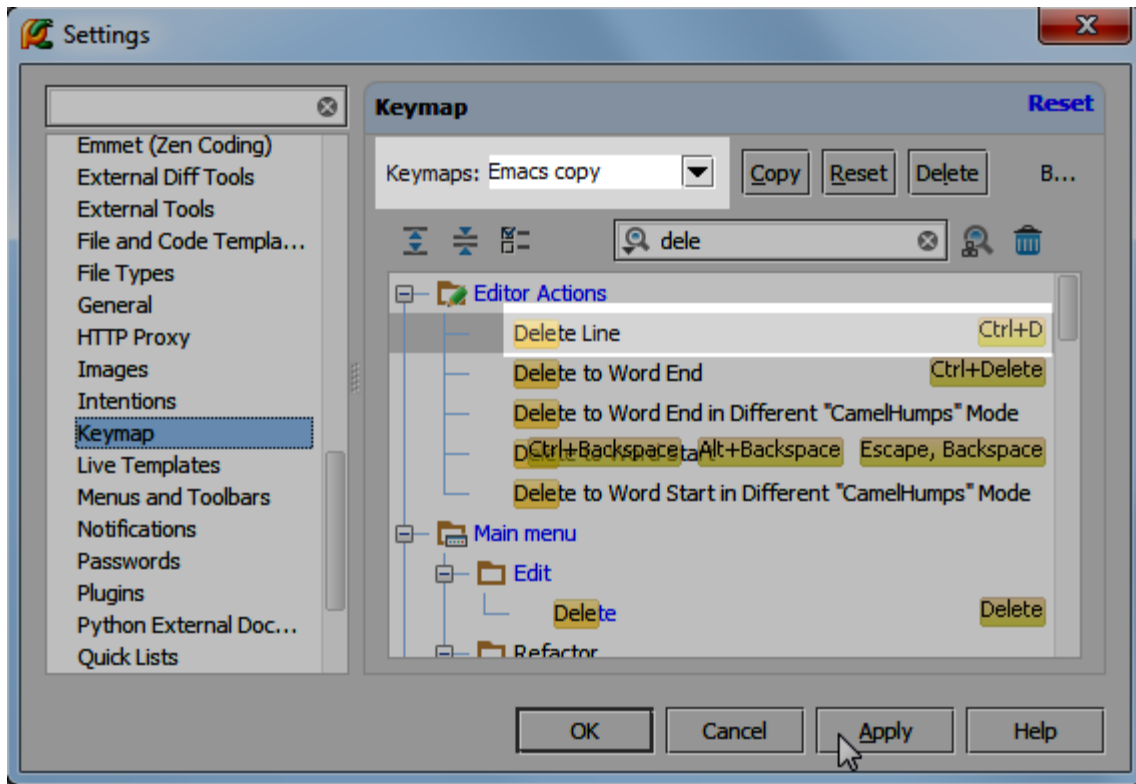
如果你按错了键，你只能使用鼠标指针来删除错误的输入，因为此时来自键盘的输入 **Backspace** 也会被系统认定为一个快捷键设置。

(6) 接下来需要注意系统的警告提示：这个 **Ctrl+D** 的快捷键组合已经存在，说明 Pycharm 已经将这个快捷键组合关联到别的设置中去了，好在之前的设置并不常用，可以将它替换掉，因此单击 **OK** 确定。

在替换时 Pycharm 会给出警告提示：



(7) 再次浏览一下设置好的快捷键方案：



你将会发现此时的配置方案自动更名为 Emacs copy。这是因为 Pycharm 并不允许用户更改其预定义好的快捷键配置方案，只能更改其拷贝文件。单击应用并关闭对话框。

最后测试一下我们的更改是否生效。打开一个 py 文件，将光标定位到待删除行，按下 Ctrl+D，该行代码顺利删除，设置更改成功。

最全 Pycharm 教程（6）——将 Pycharm 作为 Vim 编辑器使用

1、主题

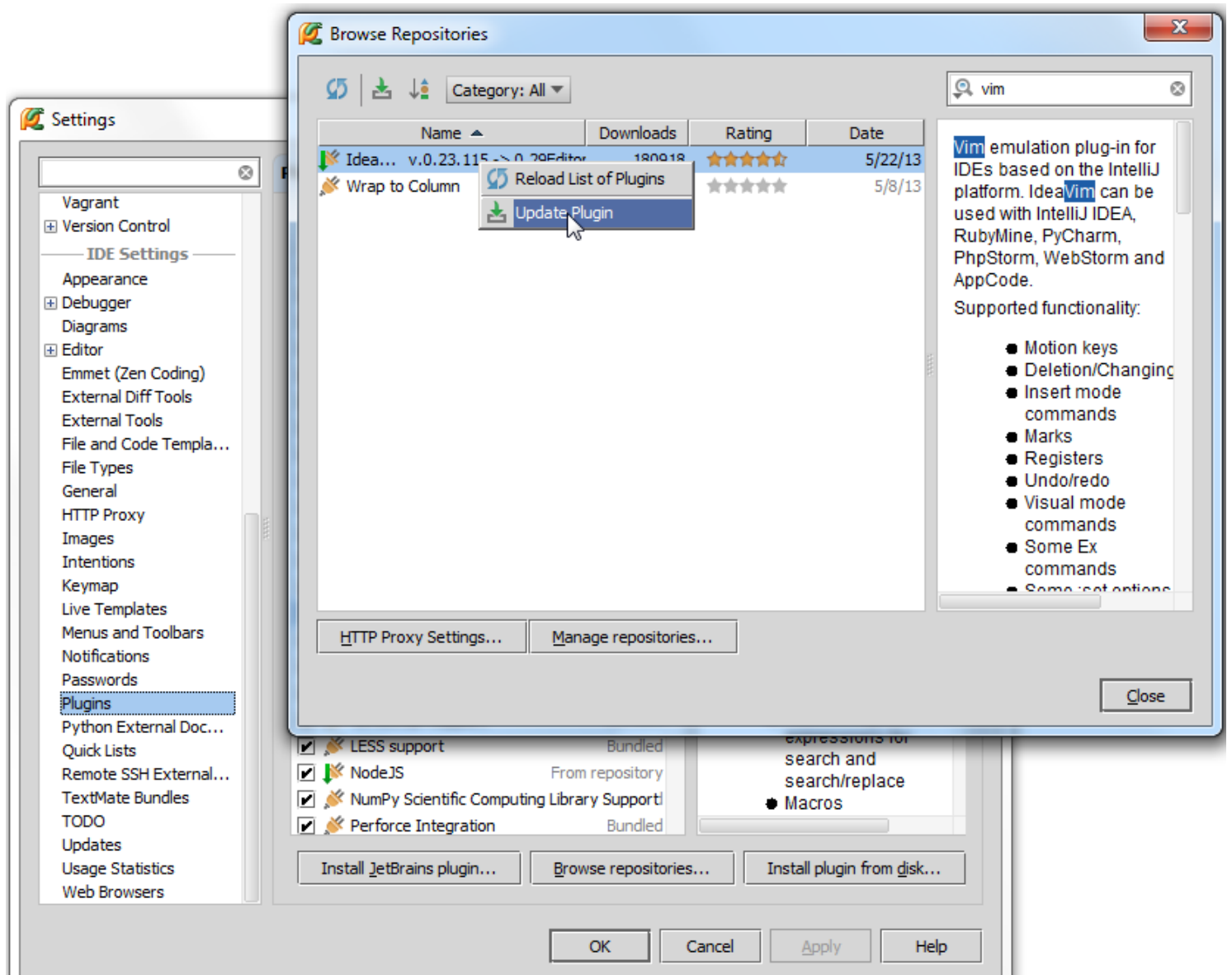
如果你是 Vim 的粉丝，并且不打算使用其他类型的编辑器，那么这篇教程将会比较适合你。这里将会详细介绍如何在 Pycharm **IdeaVim** 插件的帮助下下载、安装、使用 Vim。至于有关 Python 编程以及 Vim 的用法，详见 [official website](#)，[Vim documentation](#)。

2、准备工作

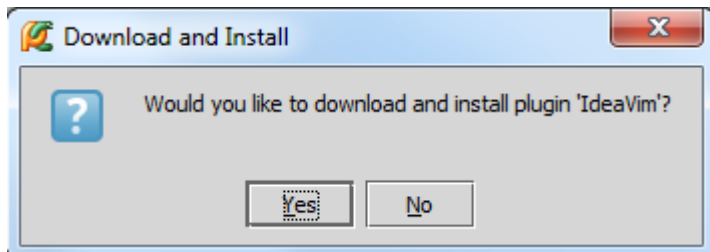
安装 2.7 或者更高版本的 Pycharm

3、下载安装 IdeaVim 插件

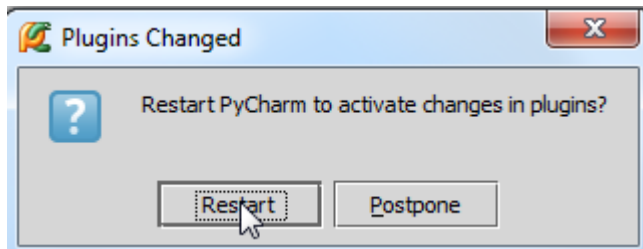
在 Pycharm 的主工具栏中单击设置按钮，在 **IDE Settings** 界面下选择 **Plugins** 页面。此时将会显示当前平台下安装的所有插件。然而 IdeaVim 并不在其中，此时需要单击 **Browse JetBrains plugins** 按钮，在搜索栏中键入 vim 来找到对应的插件：



安装插件：



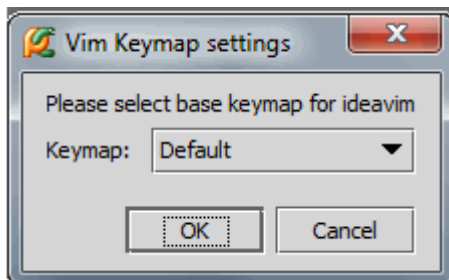
重启 Pycharm 后即可使用：



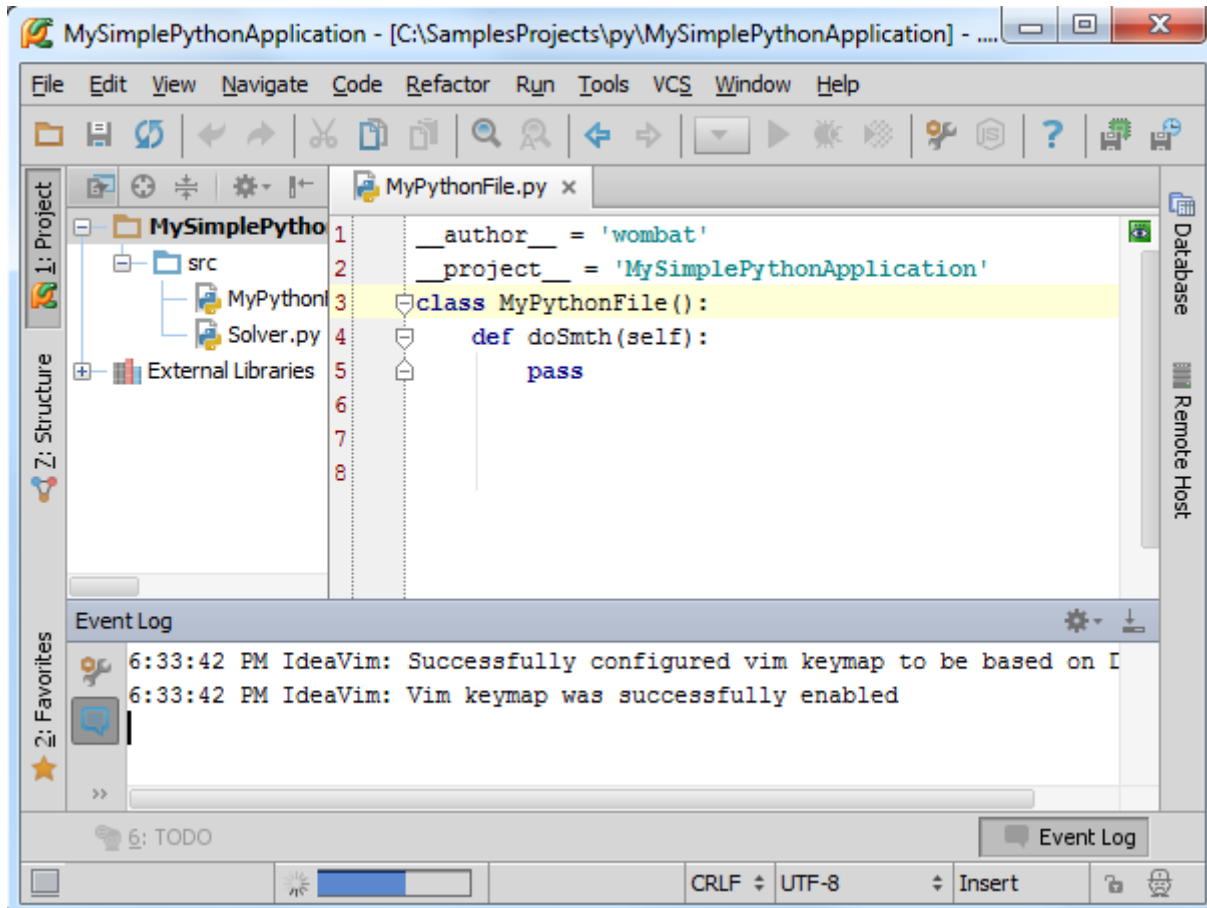
4、重启后的变化

Vim 和 Pycharm 都是基于键盘输入的文本编辑软件，响应的 IdeaVim 插件的快捷键设置很可能与 Pycharm 的快捷键设置相冲突，这也是为什么 Pycharm 需要根据已有的快捷键配置来重新创建一套 Vim 版的快捷键方案。

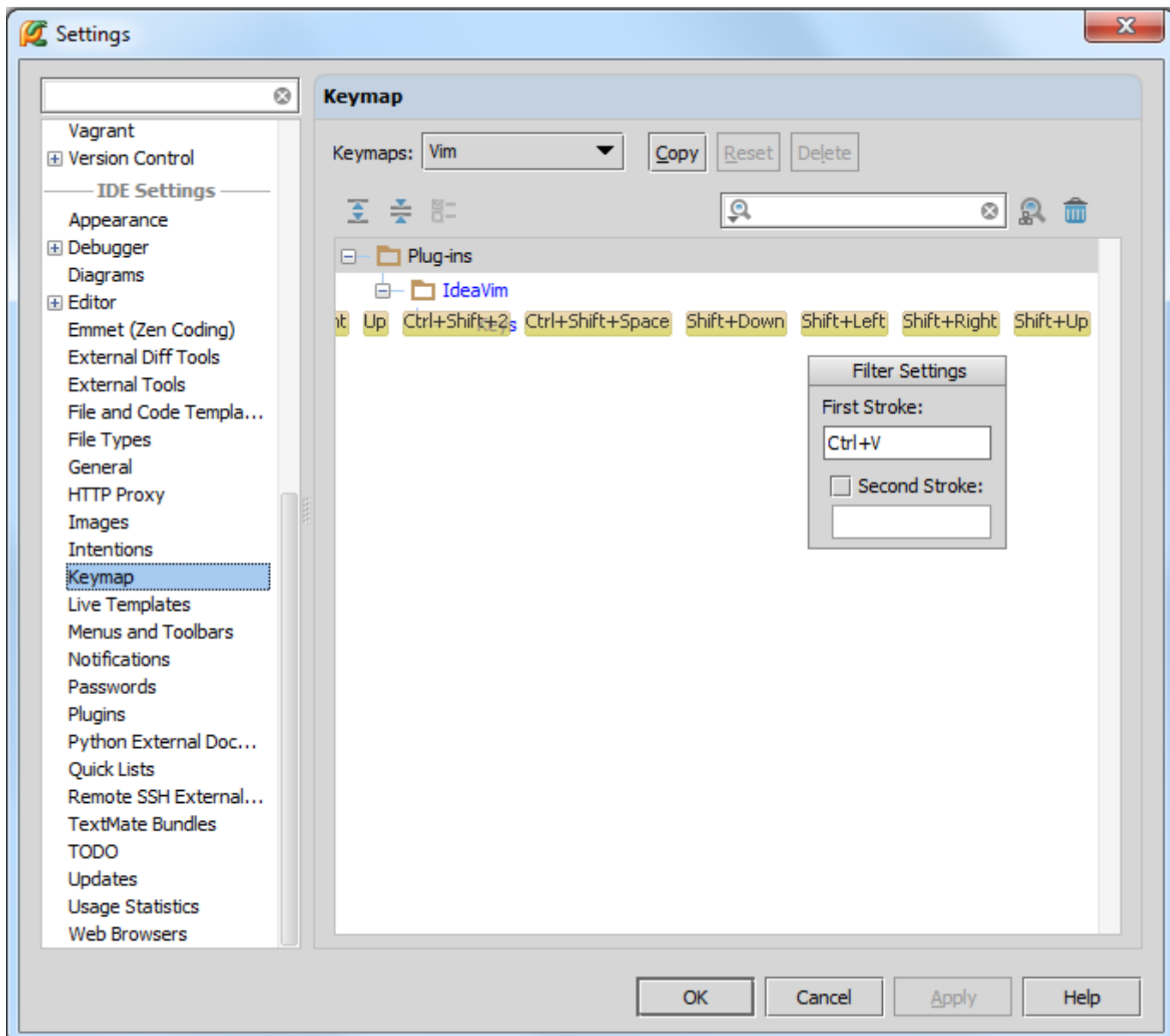
当 Pycharm 重启之后，你将会看到 **Vim Keymap settings** 对话框，在这里选择一个快捷键配置方案来作为当前 Vim 环境下的快捷键方案：



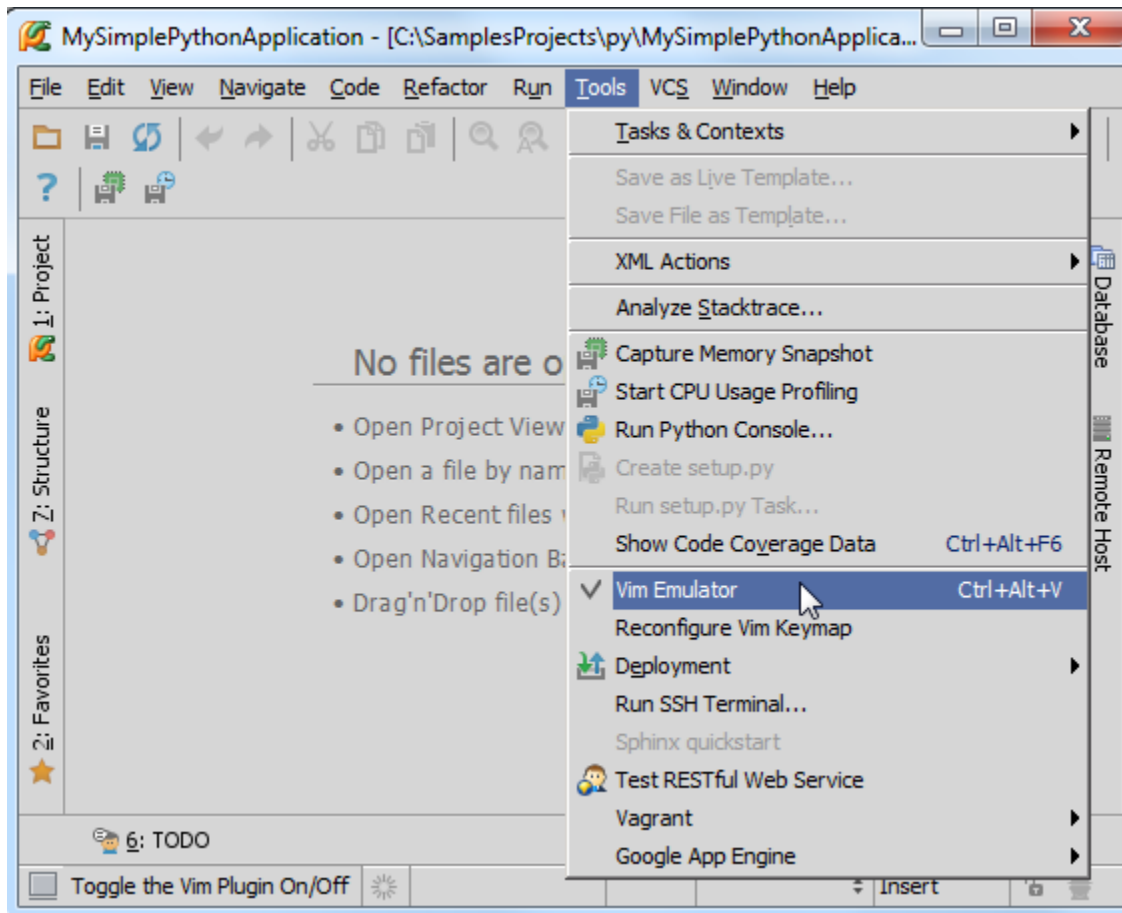
我们这里选择默认的配置，单击 OK，可以看到 Pycharm 创建了一份新的快捷键配置：



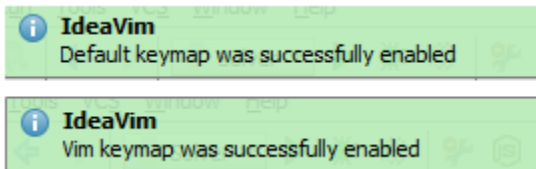
接下来再次查看快捷键设置界面。单击设置按钮进入 **Settings** 对话框，在 **IDE Settings** 下单击 **Keymap** 页面。在对应的快捷键配置下拉列表中会出现一个名为 Vim 的方案名称，其中定义了一些默认的快捷键组合，例如 Ctrl+V 代表粘贴等等：



然而这里还有一个更重要的 Pycharm 界面设置，在主工具栏中单击 **Tools**，选择 **Vim Emulator** 命令：

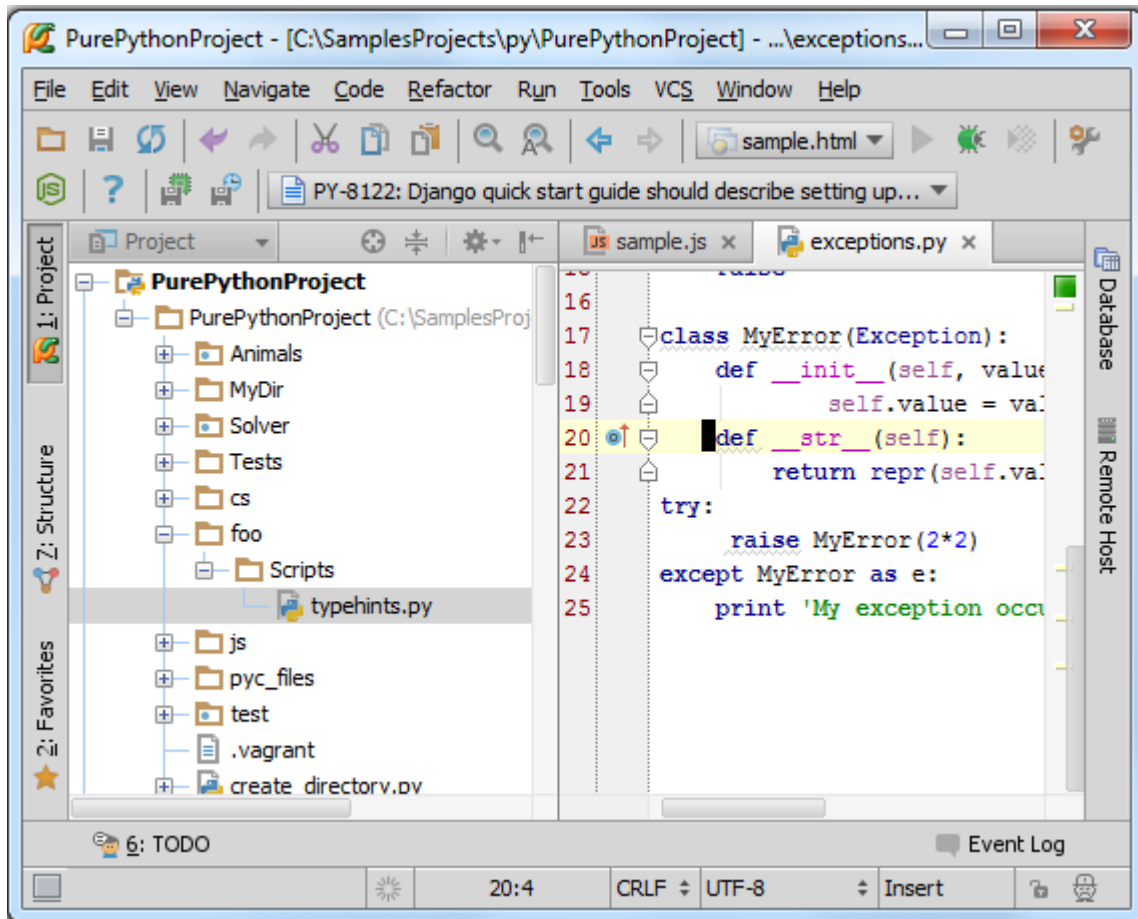


当你安装了 IdeaVim 插件之后，这条命令是默认选中的（前面有对号），也就是意味着当前的 Vim 仿真器可用。取消勾选，Pycharm 就会回到正常的快捷键状体；再次勾选，Pycharm 就会重新加载 Vim 版的快捷键配置方案：

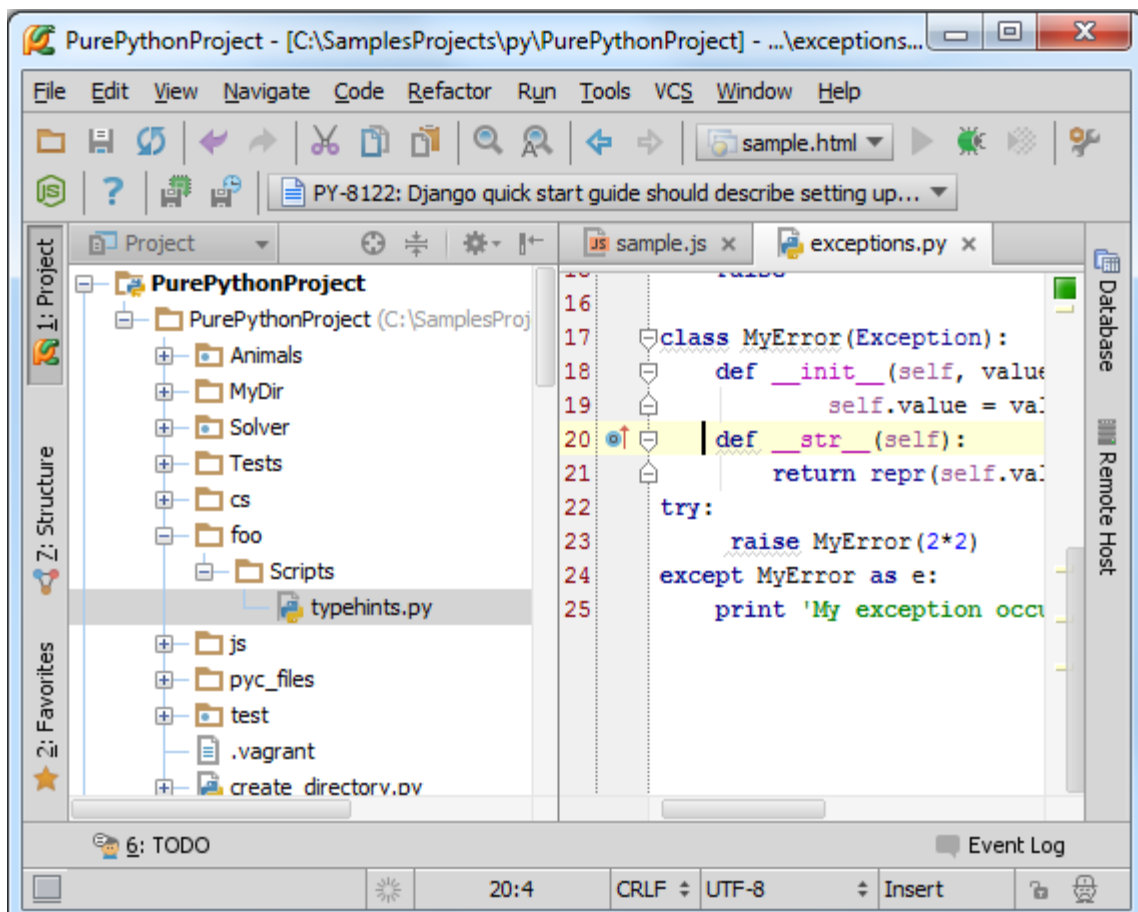


5、编辑模式

现在，我们就可以在 Vim 模式下进行编辑了，此时我们的输入光标为一个黑色小块，意味着我们当前处于 Normal 模式：

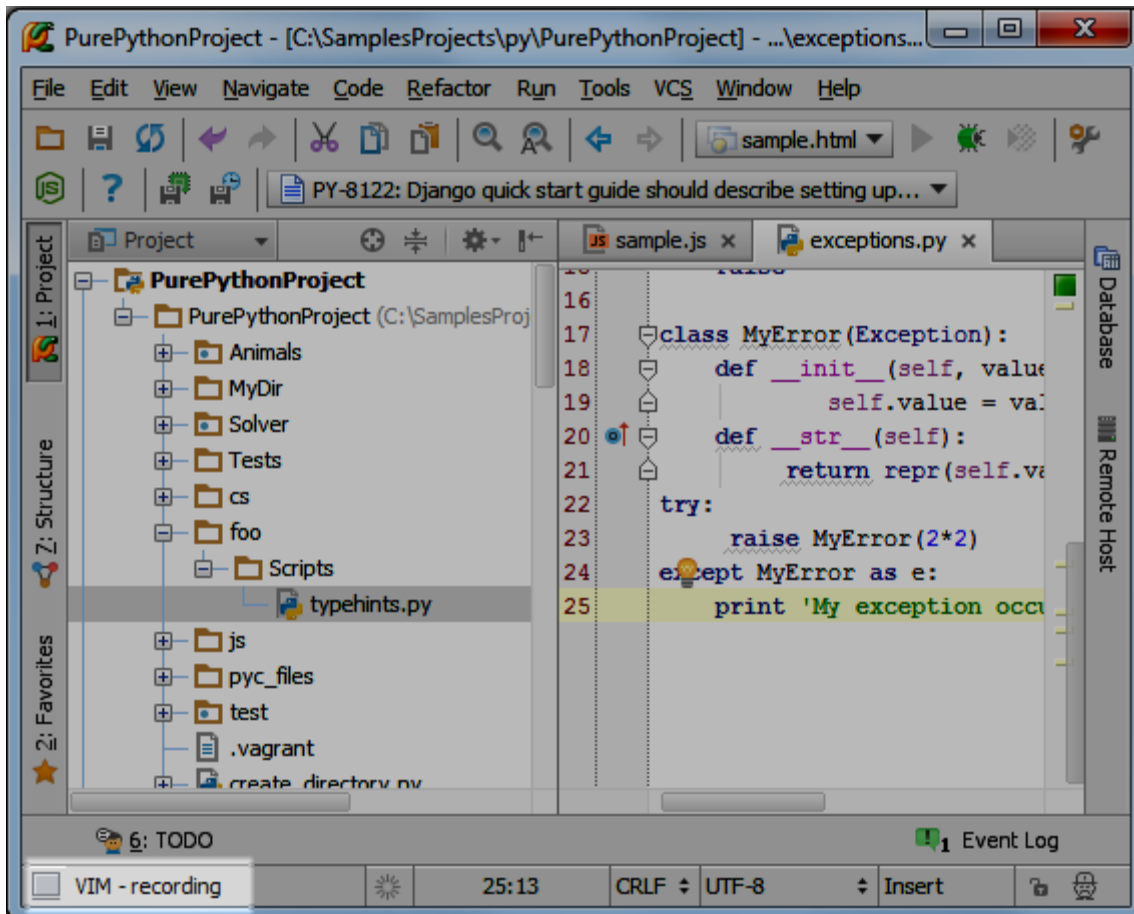


如果你想变为 `Insert` 模式，按下 `i` 键，光标将会变为一条竖线：



在这个模式下可以输入新的代码或者对当前代码进行修改。当然你还可以选择其他模式，例如按 `r` 键进入 [Replace](#) 输入模式。

顺便说一句，为了快速进入 Vim 仿真，可以查看状态栏的相关信息：[Status bar](#)



按下 `Esc` 键，回到正常编辑模式。

IdeaVim 拥有很多 Vim 编辑器的特征，例如缩写形式、快捷键组合、各种各样的命令等等，详见：[a lot more](#)。

最全 Pycharm 教程（7）——虚拟机 VM 的配置

设想这样一种情况，你在一个平台上操作你的工程，但你希望在另外一个平台上完善并运行它，这就是为什么 Pycharm 做了很多工作来支持远程调试。

在虚拟机上运行一个工程主要包含以下步骤：

- (1) 定义一个虚拟框架 [define a virtual box](#)
- (2) 需要在虚拟框架下配置一个远程的解释器 [configure a remote interpreter](#)
- (3) 在远程控制台加载当前工程 [launch your script in the remote console](#)

1、准备工作

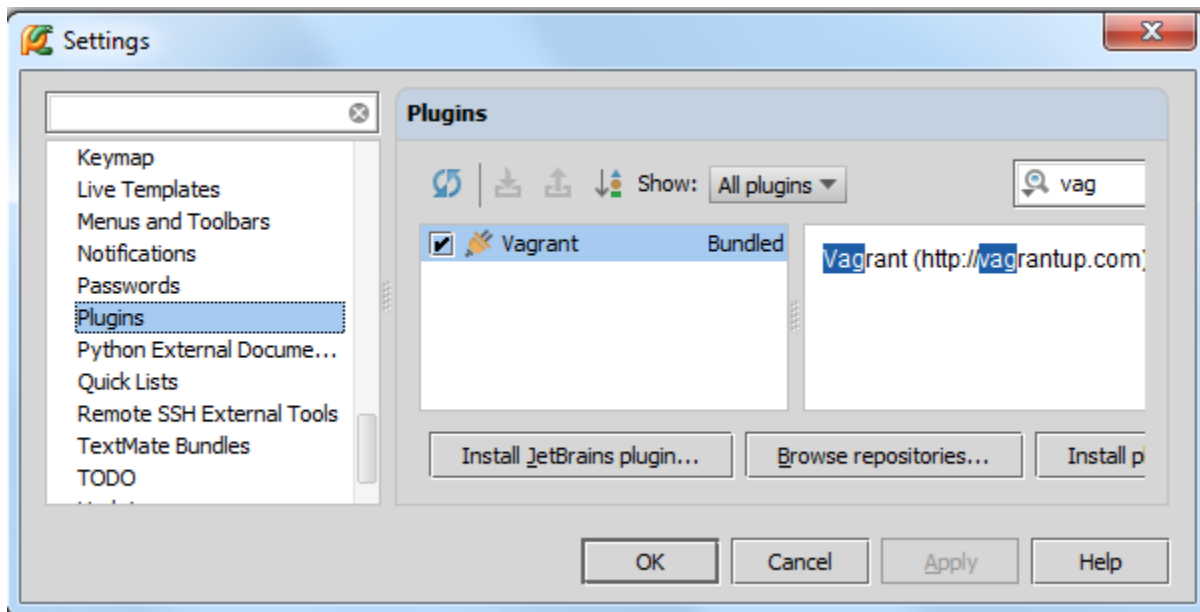
确定你的 Pycharm 已经拥有以下环境：

- (1) 安装了 [Oracle's Virtual Box](#)
- (2) 安装了 [Vagrant](#)
- (3) 将一下这些可执行文件添加到环境变量中

Vagrant 安装文件下的 vagrant.bat 文件，这部分工作应该由安装程序自动完成

Oracle's VirtualBox 安装文件下的 VBoxManage.exe 文件。

确保 Pycharm 的 **Vagrant** 插件可用：单击主工具栏中的设置按钮，在设置对话框中打开 [Plugins](#) 页面，显示插件默认可用：

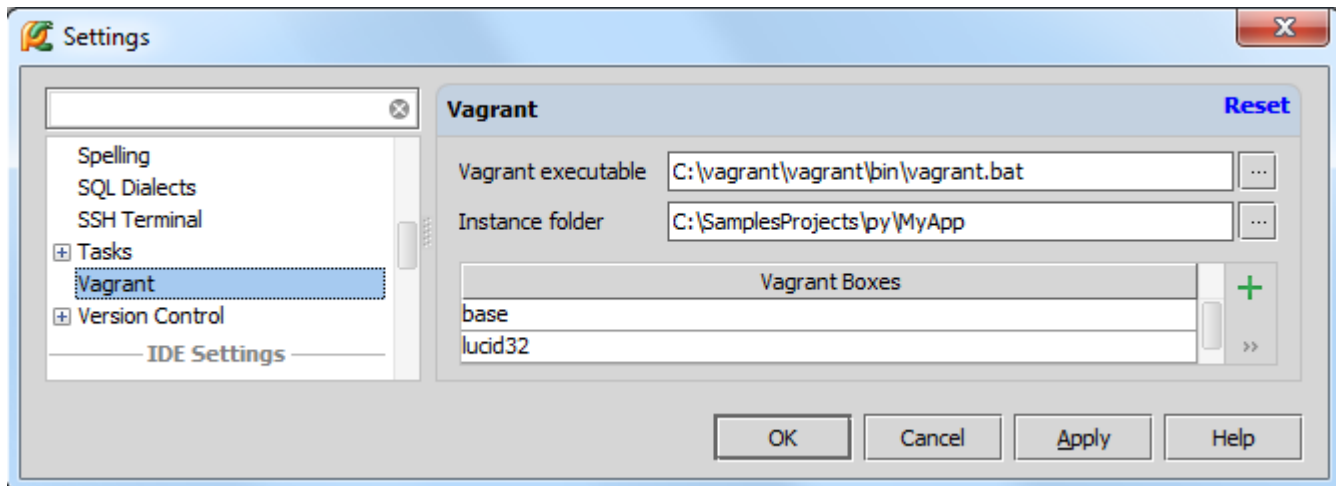


至此准备工作完成，正式开始。

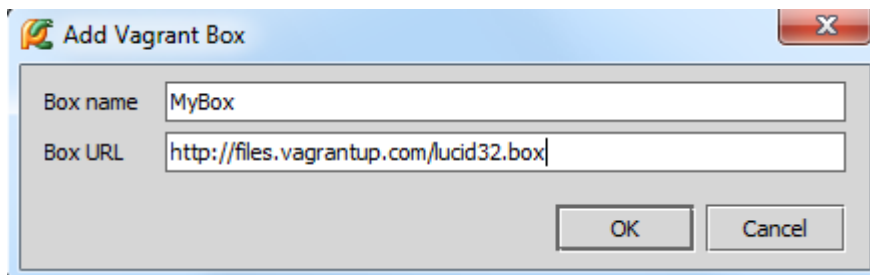
2、创建一个虚拟的 virtual box

在设置对话框中（单击主工具栏的设置按钮），单击 [Vagrant](#) 界面，然后输入可执行文件路径以及实例路径。

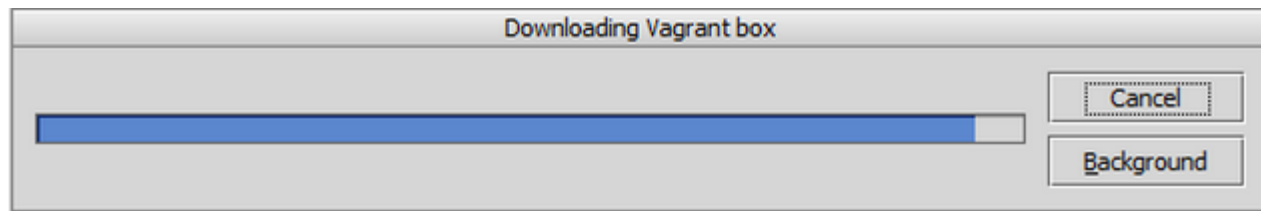
如果框架已经预先定义好，以上选项就会出现在一个下拉列表中，从中选择一个即可：



如果当前没有合适的 virtual box，则可通过单击绿色的加号来添加一个，输入框架名称和下载地址：



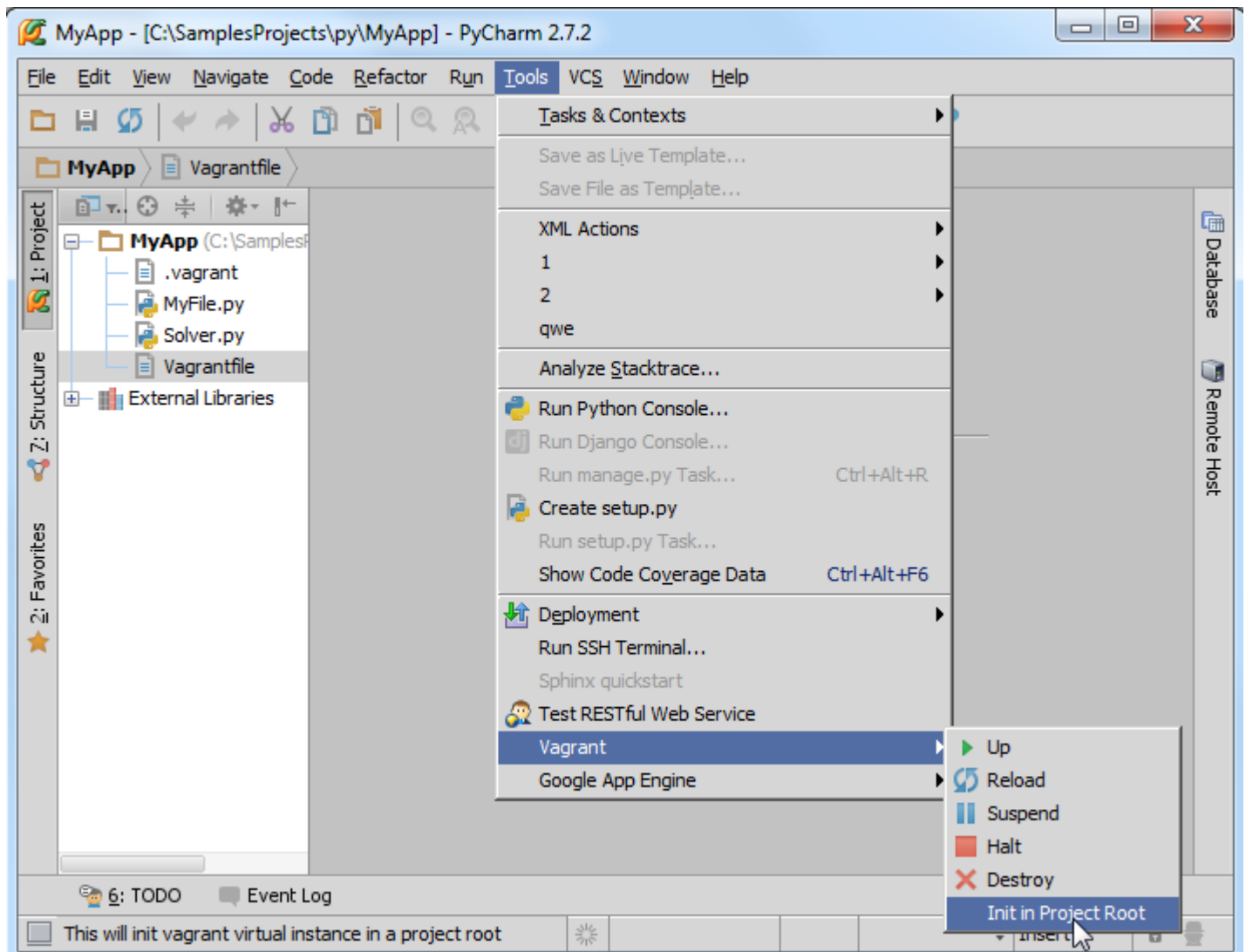
单击 OK，Pycharm 开始自动下载 VM 模板：



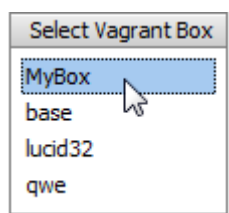
至此你已经新建了一个 virtual box 并已经将其添加到了当前环境中。

注意 **Tool** 菜单下的 **Vagrant** 命令，这个命令与标准的 Vagrant 行为相关联。

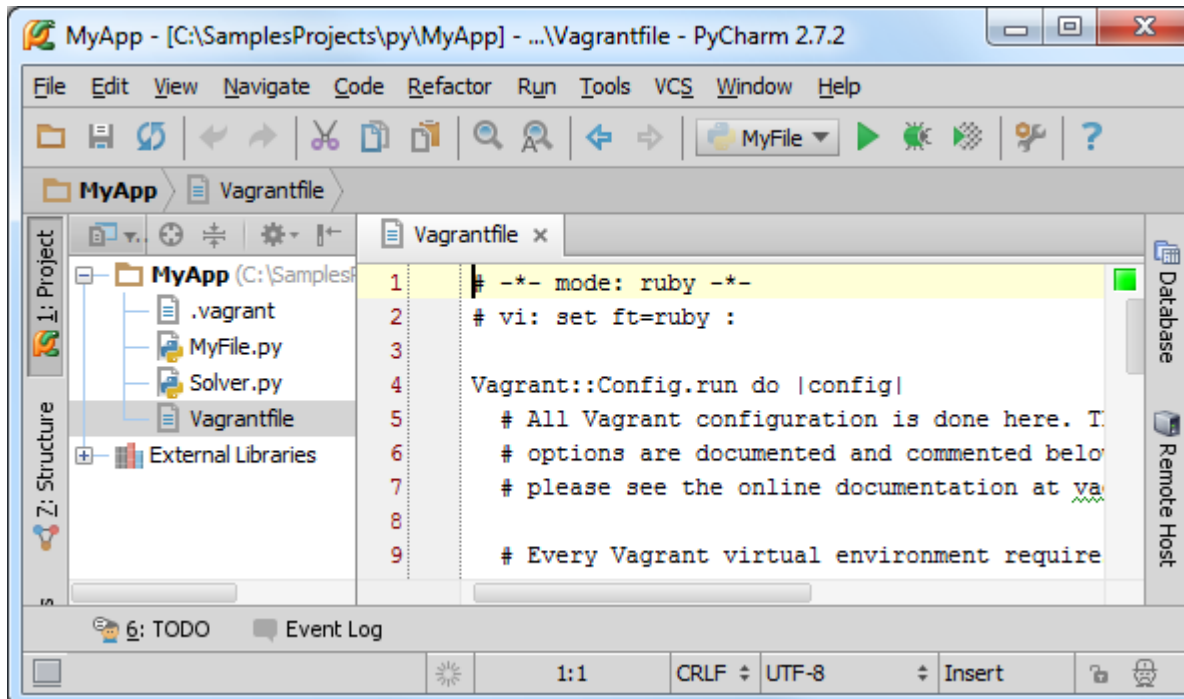
一旦创建了 Vagrant box，就需要在工程存根下对其进行初始化。在主菜单上单击 **Tools | Vagrant**，选择 **Init in Project Root**：



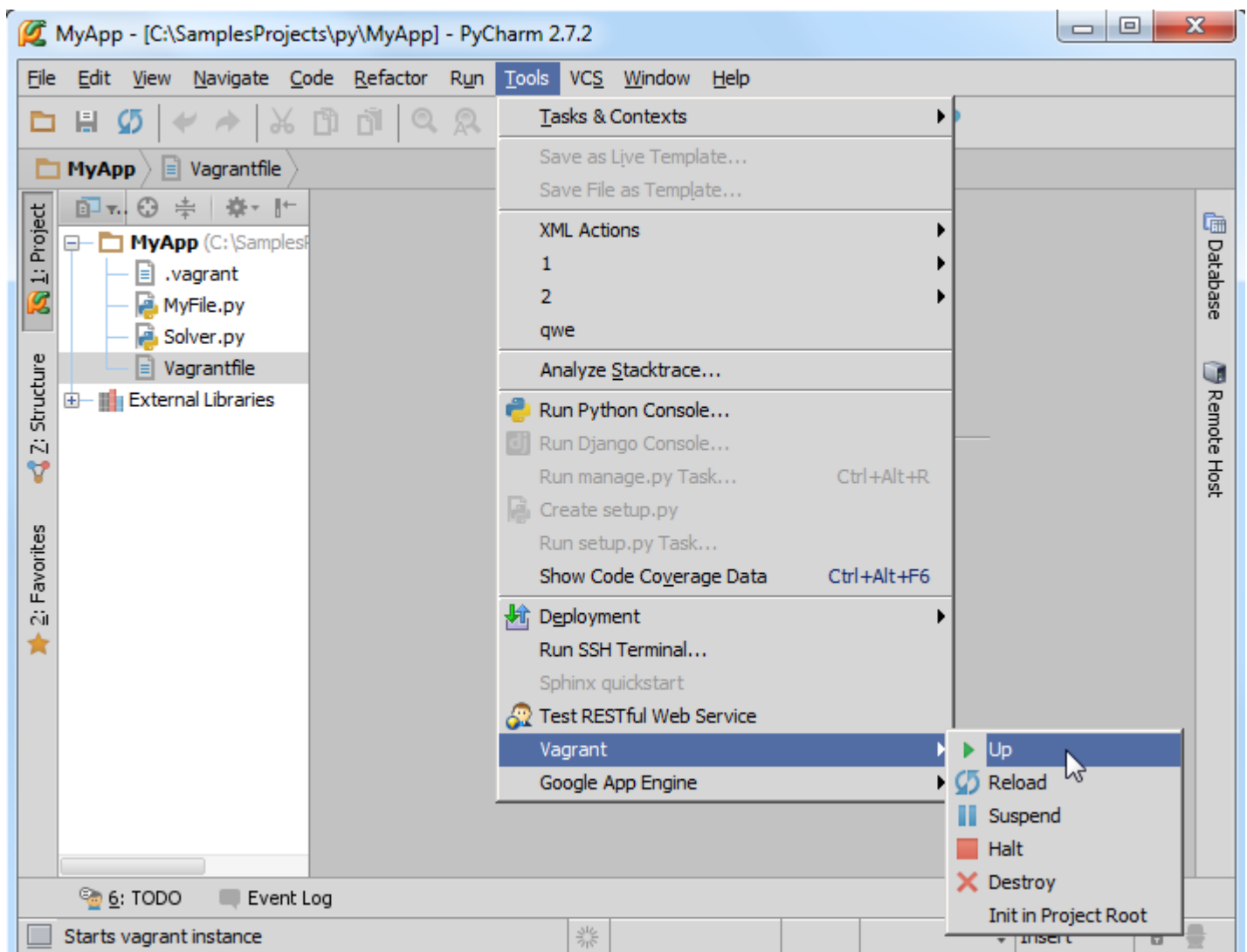
选择你准备初始化的 Vagrant box:



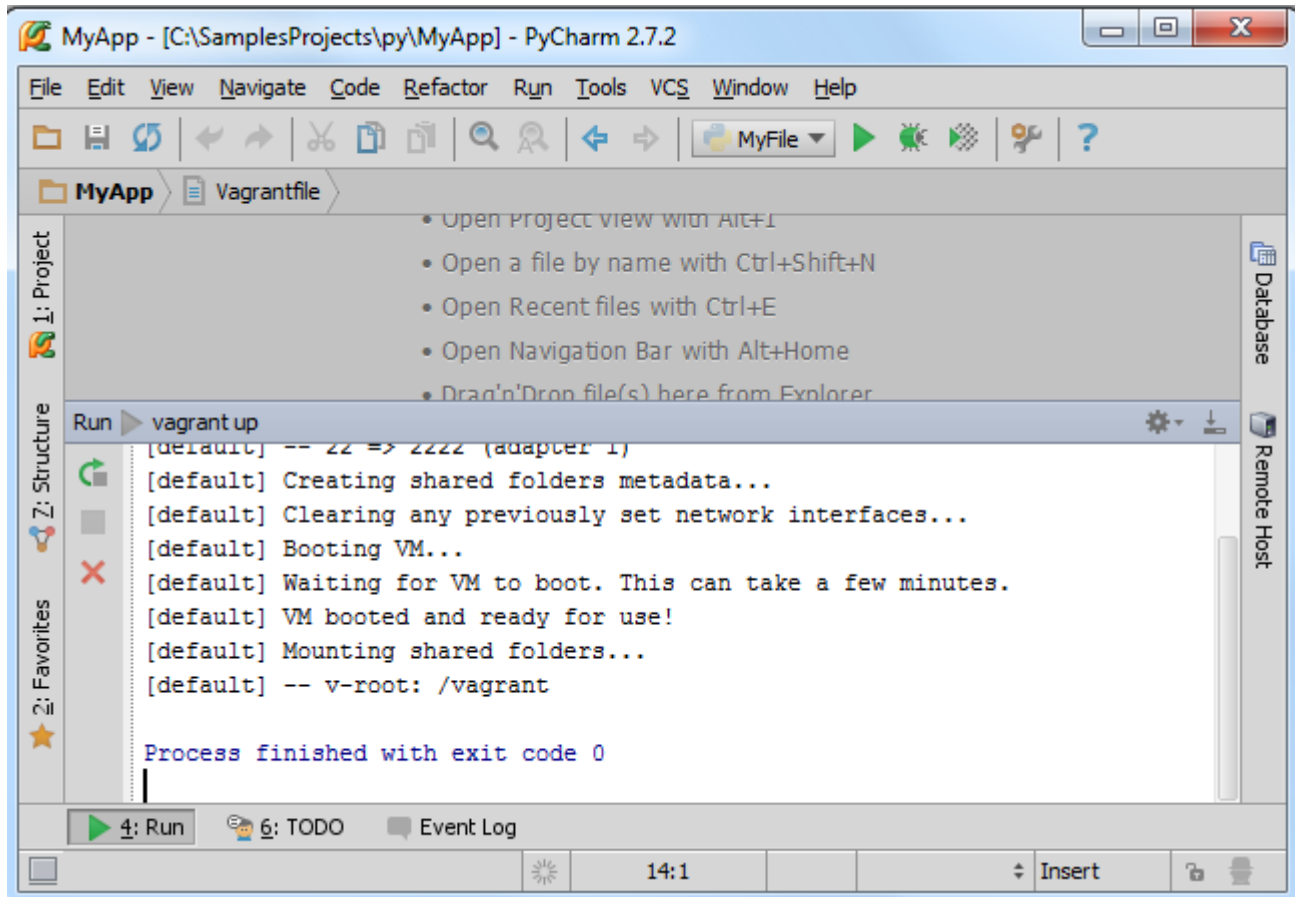
此时会创建对应的 Vagrantfile 文件，可以根据要求对其进行更改：



初始化完成后，执行 `vagrant up` 命令（在 **Vagrant** 菜单中选择 **Up** 命令）：

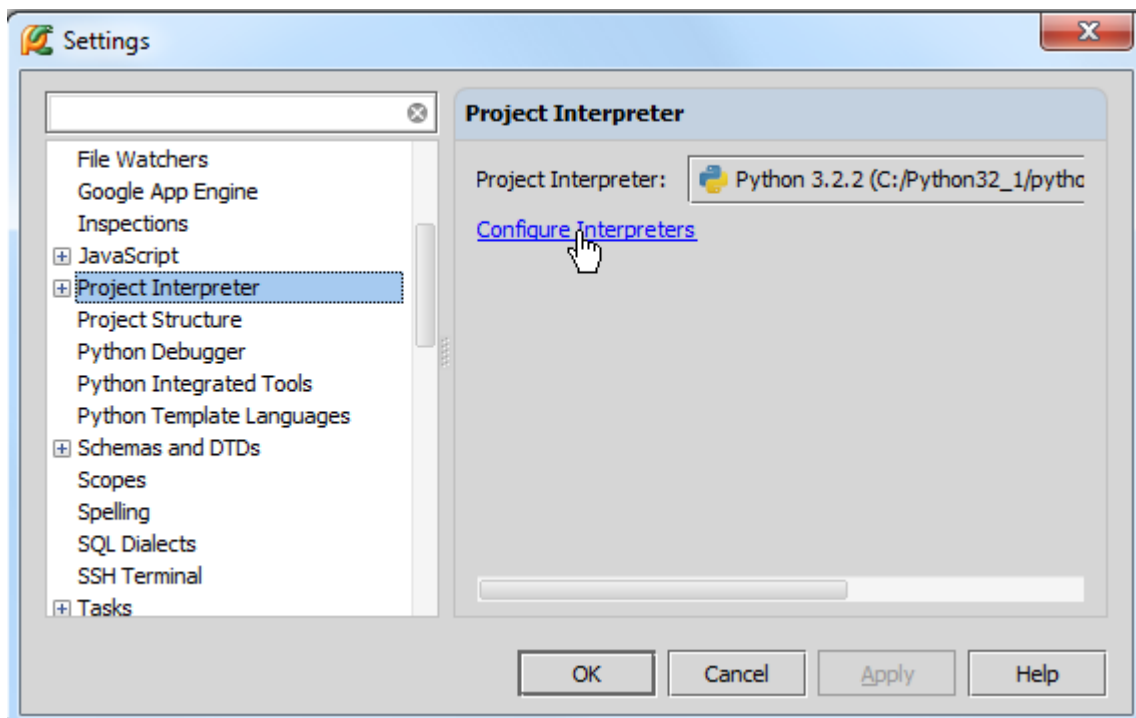


Pycharm 会自动运行 `vagrant up` 命令，并在控制台界面显示输出结果：

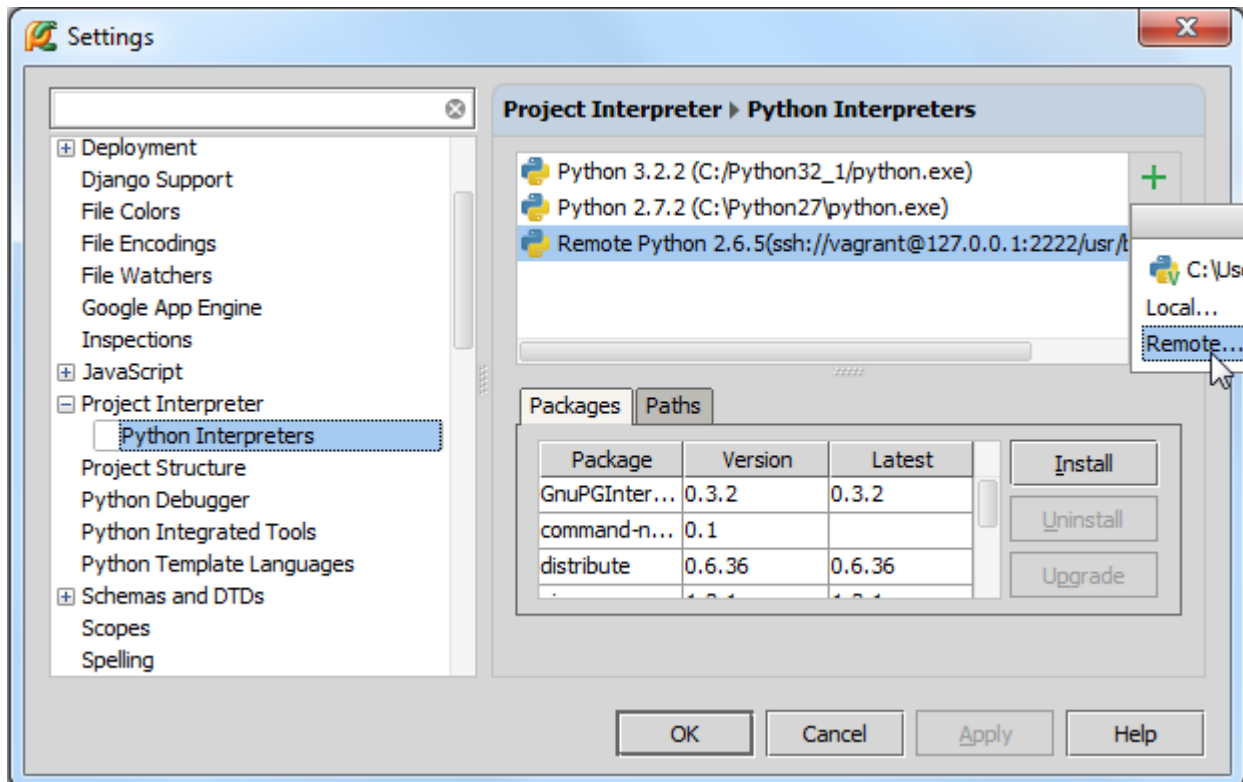


3、在虚拟机上配置远程解释器

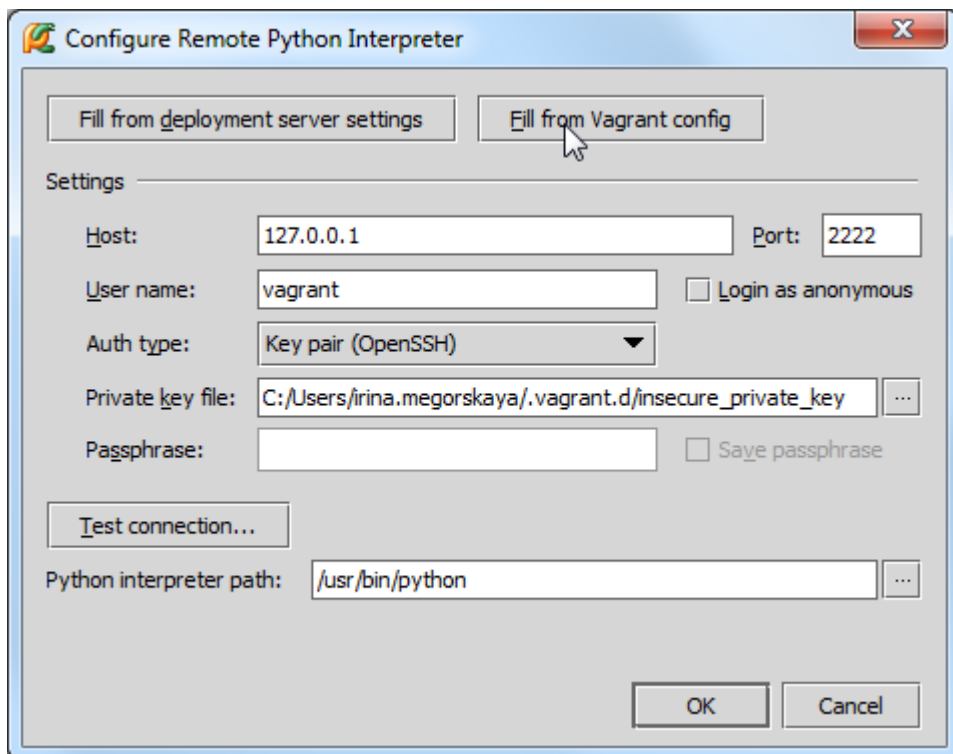
再次打开设置对话框（单击主工具栏上的设置按钮），选择 **Project Interpreter** 页面，在这里你可以从下拉列表中选择一个对应的解释器，但是如果当前没有可用的解释器，我们就需要单击 **Configure Interpreters** 来自定义一个：



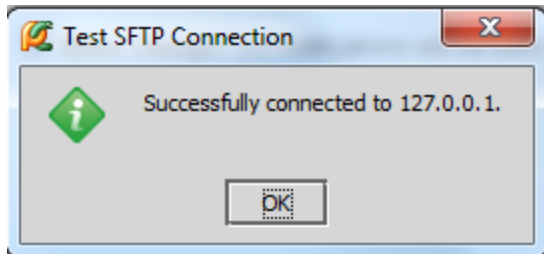
此时会打开 **Python Interpreters** 界面，单击绿色的加号来选择一個远程的解释器：



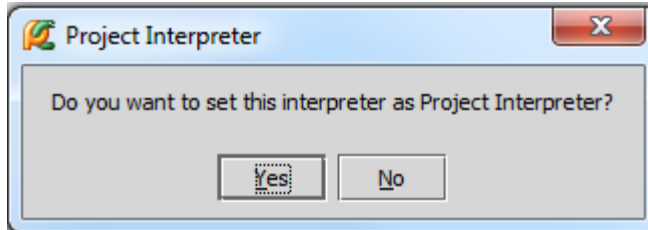
在 **Configure Remote Python Interpreter** 对话框中，需要进行服务器配置。这些设置可以手动设定，也可以从已经定义好的 Vagrant 配置文件中导入，在这里我们选择第二个方式。单击 **Fill from Vagrant configuration** 按钮，将会根据配置文件的内容自动填充相关设置属性值：



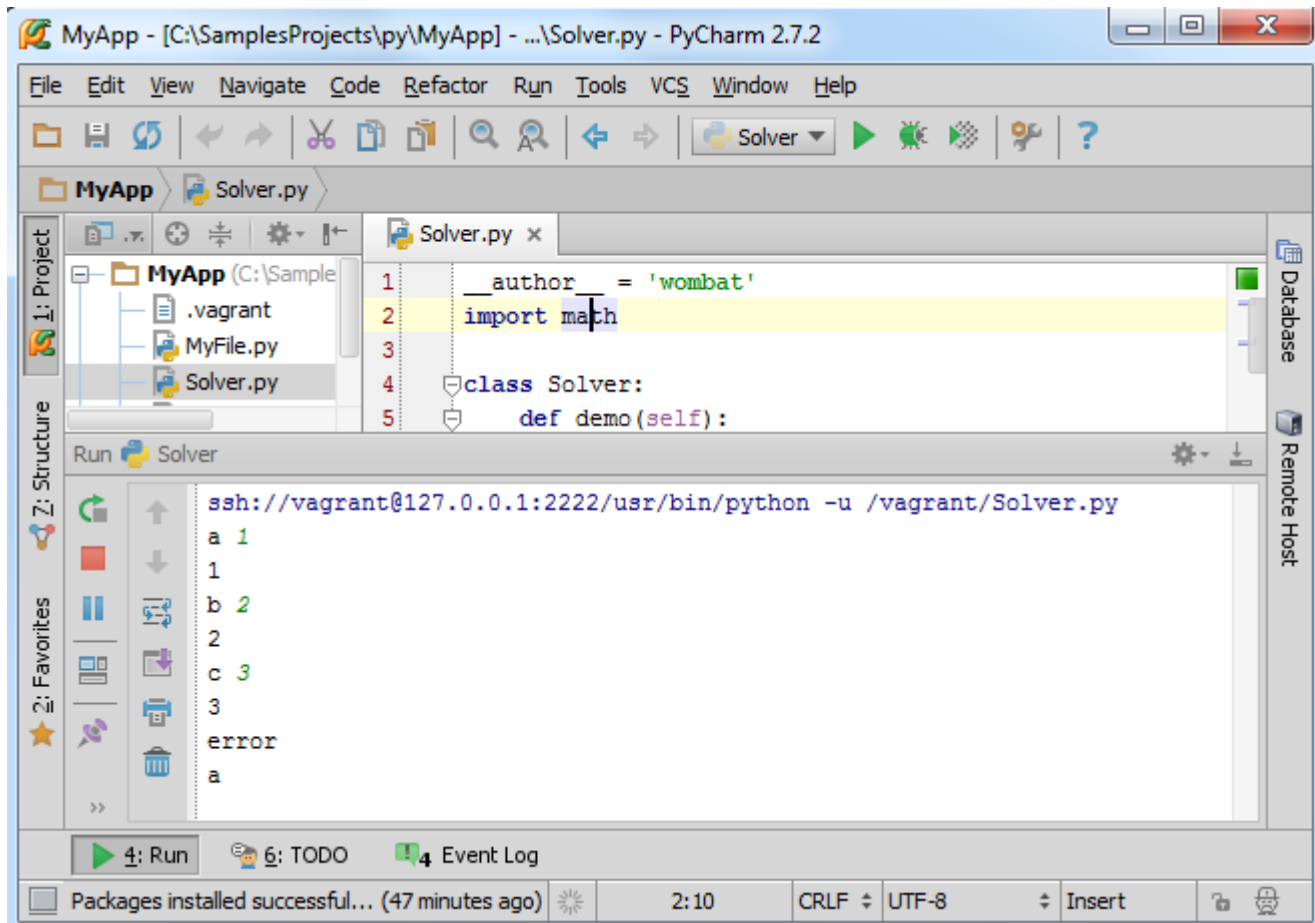
为了确认是否配置成功，单击 **Test connection...** 按钮：



将其设置为默认解释器：



从现在开始我们就可以在 VM 虚拟机上运行所有的脚本工程了：

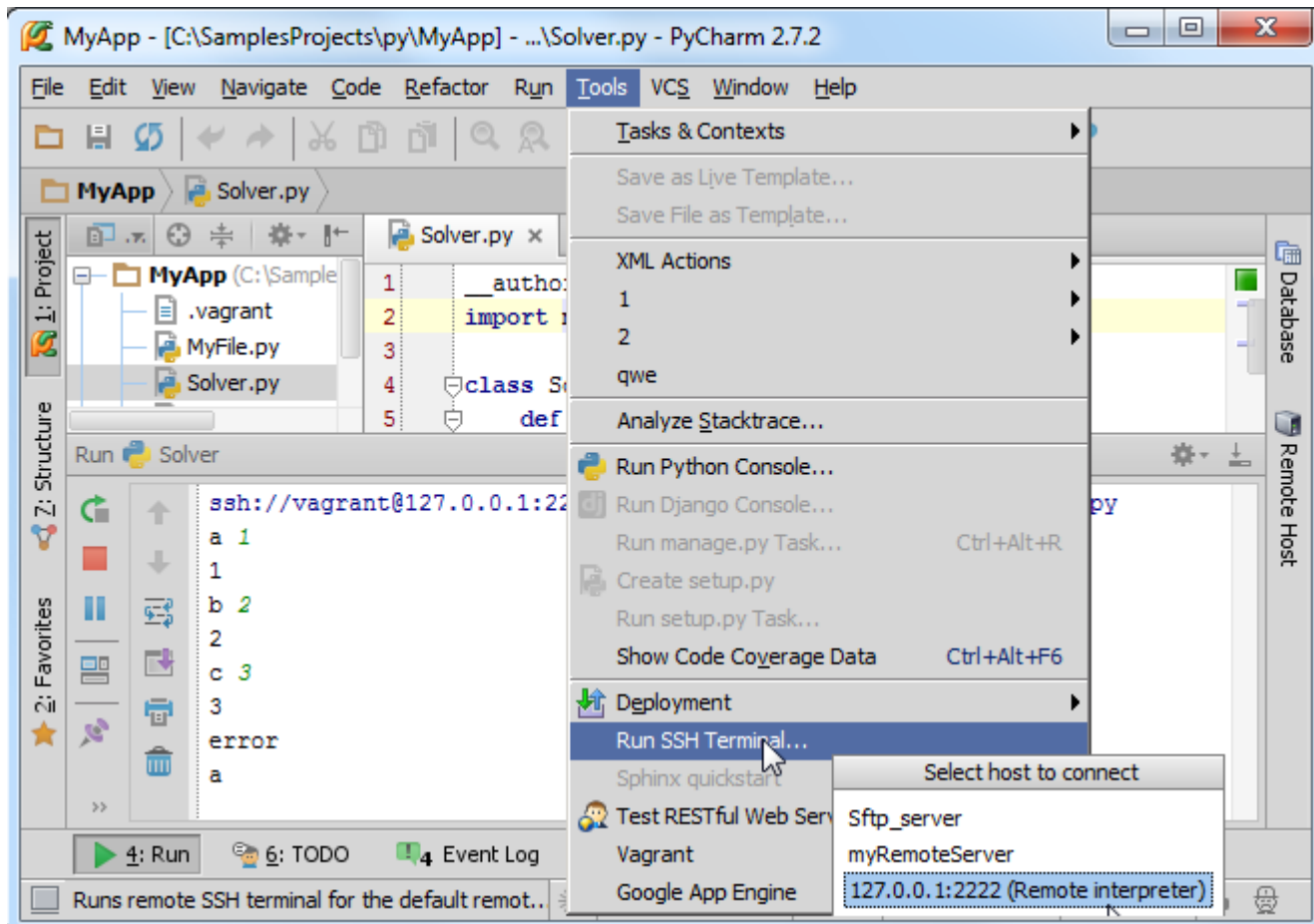


接下来我们通过 SSH 来登录 virtual box。

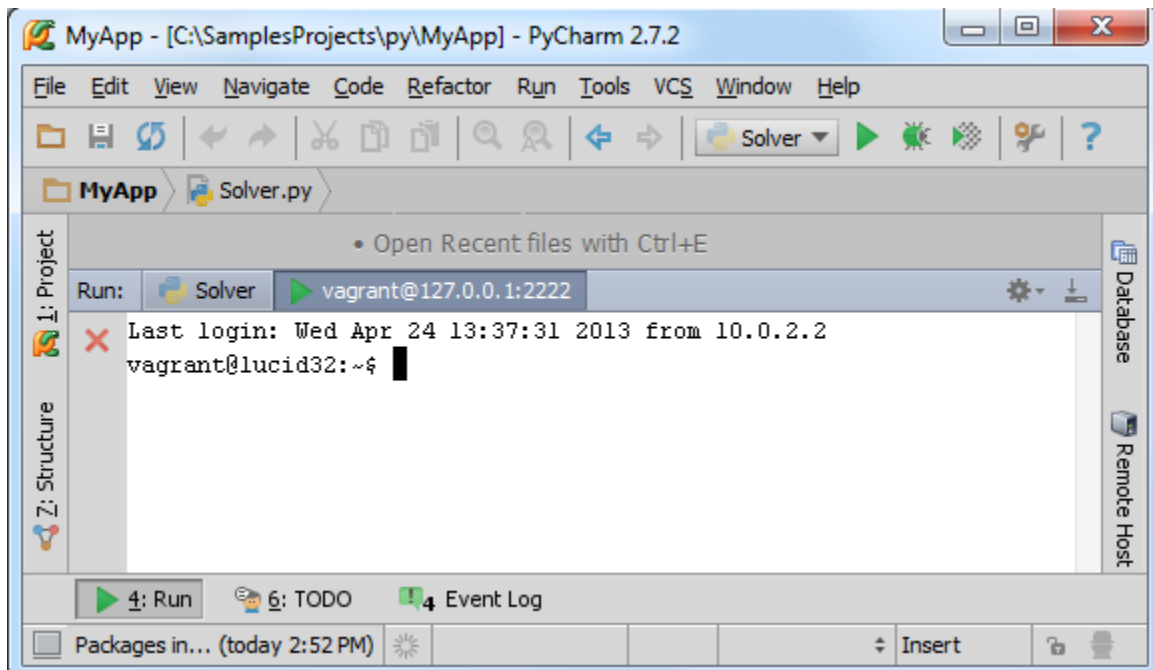
4、链接 SSH 终端机

为什么需要登录呢？因为 Pycharm 要求你这么。

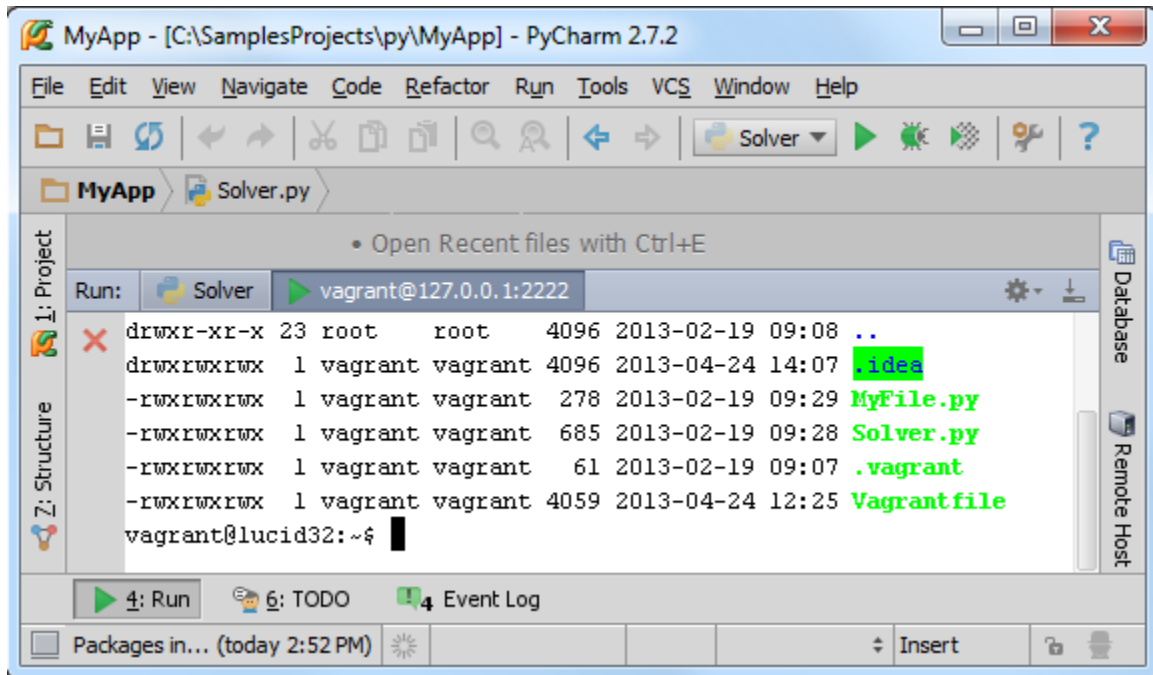
在主菜单中选择 **Tools | Run SSH Terminal**，如果你定义了不止一个主机 (host)，则选择一个你想要建立链接的（我们这里选择远程解释器）：



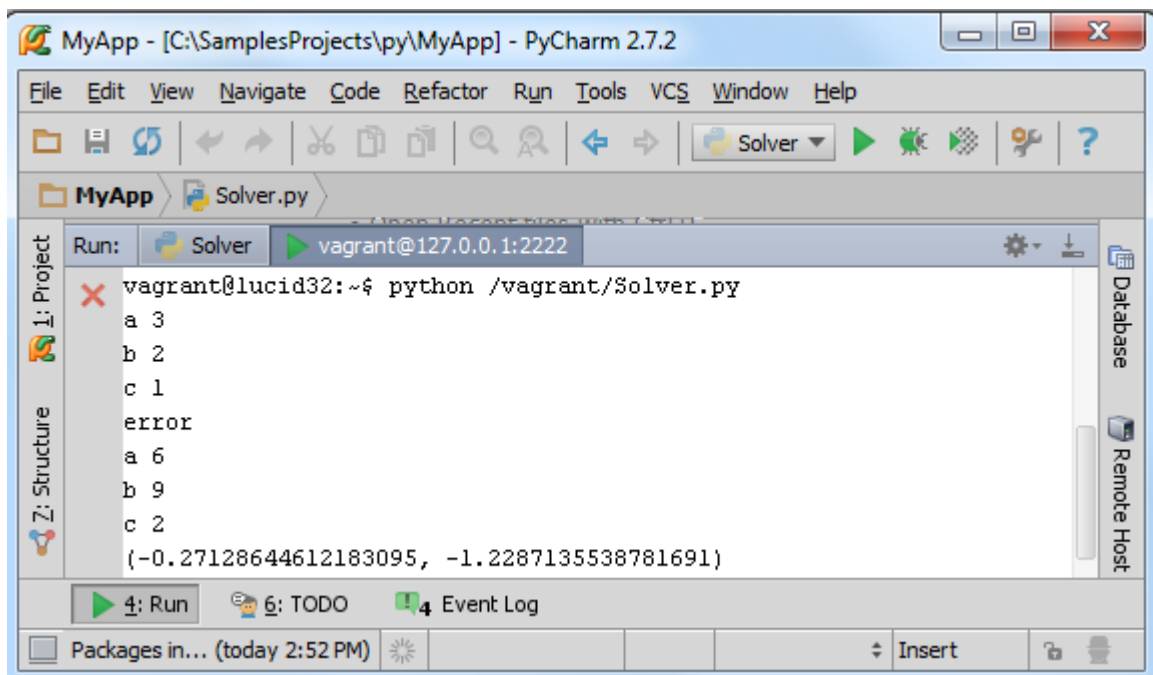
观察 Run tool window 窗口的控制台运行信息：



现在你已经能够和 virtual box 进行直接交互了，首先我们先确认你的工程目录是否进行了完整映射。只需观察 vagrant's 默认的共享文件夹信息：



接下来运行一个可用的脚本文件，例如 Solver.py:



大功告成。

最全 Pycharm 教程（8）——Django 工程的创建和管理

1、主题

这部分教程主要介绍如何通过 Pycharm 创建、管理、运行一个 Django 工程。对于 Django 模块的相关知识大家可以参考 Python 社区。

2、准备工作

- (1) Pycharm 为 3.0 或者更高版本。
- (2) 电脑上至少安装了一个 Python 解释器，2.4 到 3.3 版本均可。

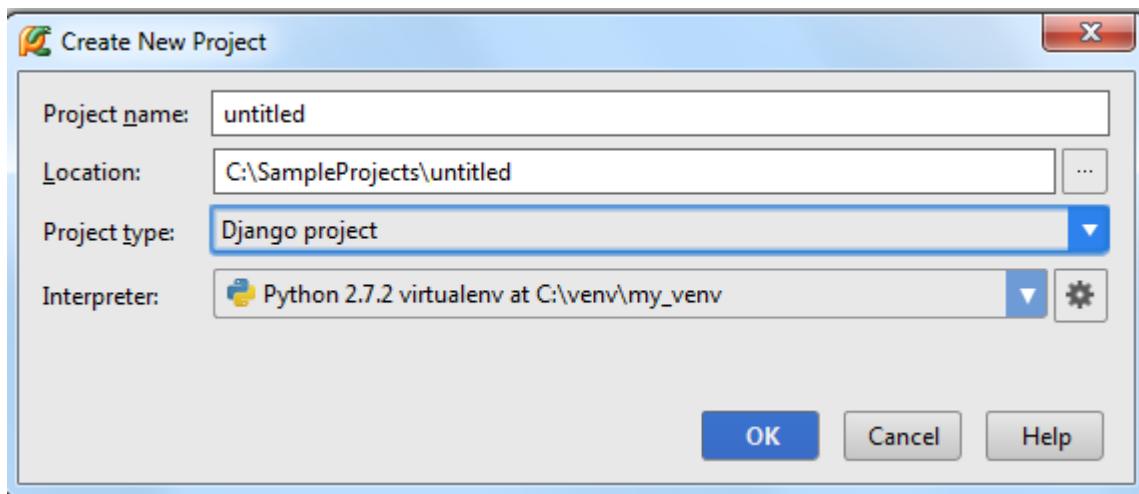
这部分教程所用的环境配置如下：

- (1) Django 模块的版本为 1.6.5
- (2) 默认为 Windows 模式下的快捷键配置
- (3) 这部分例子与 Django 模块的说明文档中所用的实例相同 [Django documentation](#)

3、创建一个新工程

实际上所有工程的创建都可以通过单击 [Welcome screen](#) 界面上的 Create New Project 按钮来实现。

如果你已经打开了一个工程,可以通过菜单栏 File → New Project...来创建一个新的工程。接下来在 [Create New Project dialog](#) 对话框中输入工程名字、选择类型以及用到的解释器版本：



单击 OK，工程的个性化设置完成。

这就意味着对应目录已经创建完成，并且预先定义了一个.idea 目录用来保存配置信息 [project settings](#)。

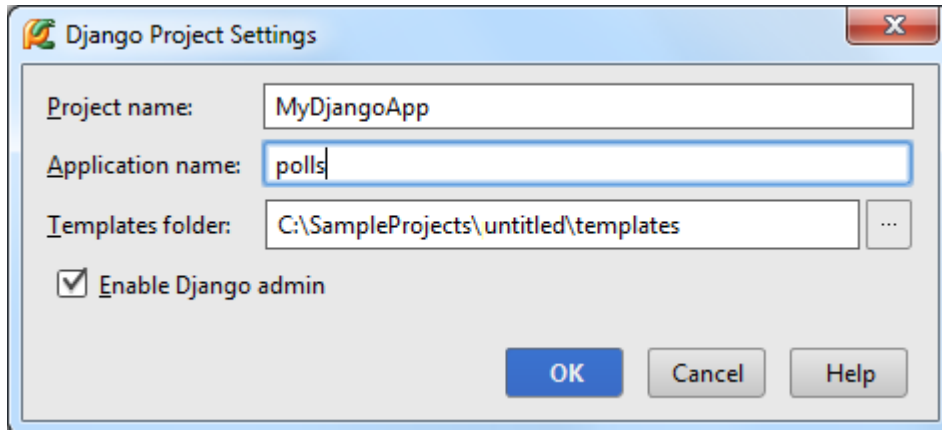
对于一个空的工程 [empty project](#)，创建的环节已经完成了。接下来你就可以开始编写程序。但对于一些所支持的第三方框架，还有一些工作要做。根据所选择的工程类型，Pycharm 会提示我们进行一些额外的框架设置。

在本实例中，让我们来创建开发一个 Django 应用。

4、创建一个 Django 工程

因此，在 [Create New Project](#) 对话框中我们的工程类型选择为 Django，注意 Pycharm 会提示我们安装 Django 框架，如果当前环境中没有可用的话。

接下来我们进行 Django 工程的相关设置：



单击 OK，设置完成。

5、工程目录结构

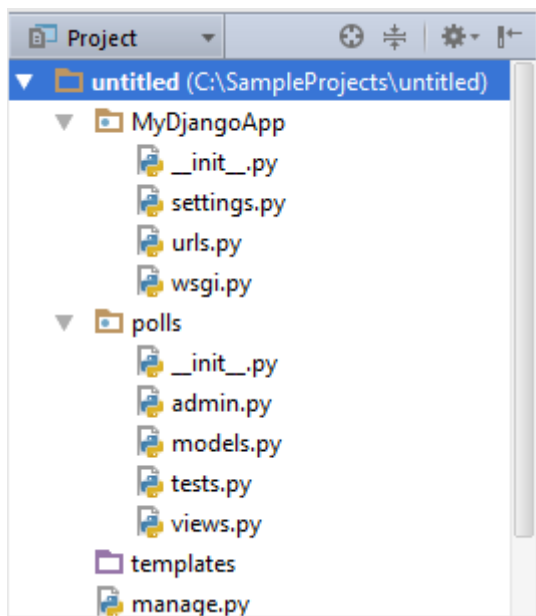
正如上面所说，工程的根目录结构已经创建完成，主要包含基本的框架配置文件和目录，当你创建其他类型的工程时也会有类似操作，如 t Pyramid，或者 Google App Engine。

接下来我们研究如何在 Project 窗口中显示工程结构。

6、Project 窗口中的目录结构

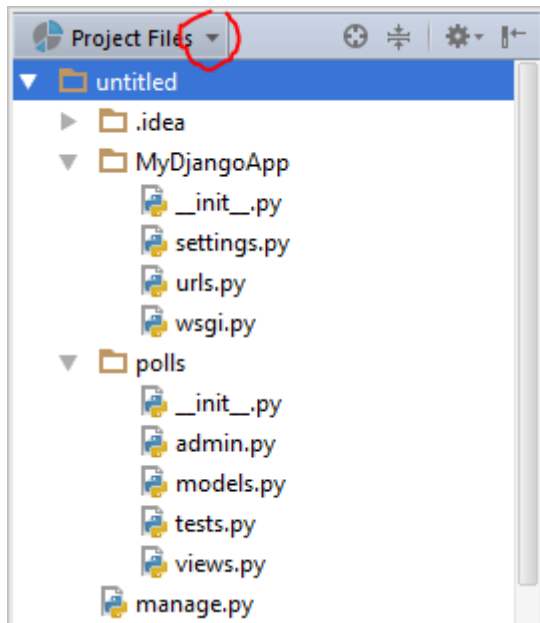
这是默认显示模式。窗口中将会显示的结构有 polls 和 MyDjangoApp 目录，当然还有两个 Python 文件：manage.py 和 settings.py。

在这个窗口中你是无法看到.idea 目录结构的。



7、Project 窗口下的工程文件

如果你想看到 idea 目录，只需选择 view Project Files 模式，这个视图所显示的文件和之前一样，只是多了 idea 目录：



Ok,回到之前的视图模式。

8、Project 窗口中所显示的文件都是干什么用的？

- (1) untitled 目录是工程的容器，在窗口中以加粗字体显示。
- (2) manage.py 是一个命令行文件，帮助你操作你的 Django 工程，详见 [product documentation](#)
- (3) 嵌套子目录 MyDjangoApp 充当了当前工程的库
- (4) MyDjangoApp/__init__.py 是一个空文件，用来指示当前目录应该作为一个库来使用。
- (5) MyDjangoApp/settings.py 包含了当前工程的相关设置 [configuration for your Django project](#)
- (6) MyDjangoApp/urls.py 包含了当前工程响应的 url 信息 [URL declarations for your Django project](#)
- (7) MyDjangoApp/wsgi.py 定义了 WSGI 兼容模式下 Web 服务器的入口，详见 [How to deploy with WSGI](#)
- (8) polls 目录下包含了完善 Django 应用的所有文件（此时为空）：

polls/__init__.py 指示当前目录应该作为一个库来使用

polls/models.py 保存我们所创建的应用程序模型

polls/views.py 保存我们的 views

- (9) templates 为空，用以包含响应的模板文件

值得一提的是你可以创建很多 Django 应用，通过运行 manage.py 文件的 startapp 任务来将其添加到当前工程中（主菜单上的 Tools→Run manage.py task）命令。

9、配置数据库

根目录生成后，我们需要做一些微调。打开文件 settings.py（选中后按 F4）。

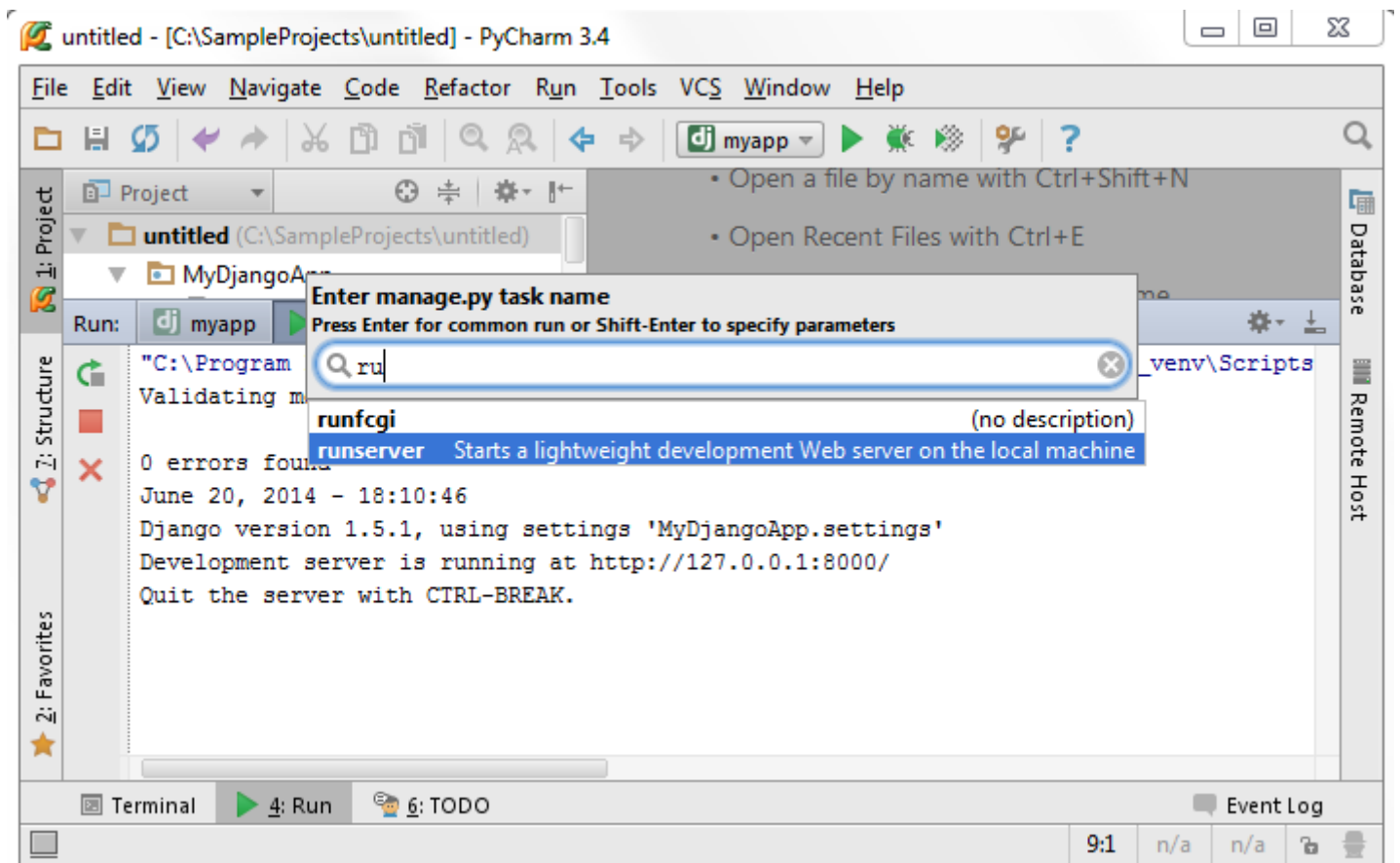
首先确定准备在应用程序中使用哪种数据库。可以通过以下方法定位 DATABASES 变量：按下 Ctrl+F，然后在搜索栏中输入需要查找的字符串，然后在'ENGINE'行的冒号后边输入使用的数据库管理系统（这里暂且设置为 sqlite3）。

在'NAME'行，输入预定义数据库的名称（无论其是否已经存在）：

```
settings.py x
data
9
10 MANAGERS = ADMINS
11
12 DATABASES = {
13     'default': {
14         'ENGINE': 'django.db.backends.sqlite3'
15         'NAME': 'MyDatabase',
16         # The following settings are not used
17         'USER': '',
18         'PASSWORD': '',
19         'HOST': '', # Emp
20         'PORT': '', # Set
21     }
22 }
```

10、加载 Django 服务

由于我们在这里处于谨慎选择了 sqlite3 数据库。因此这里无需再定义其他变量（如用户证书、端口号、POST 文件等）。接下来我们核实一下我们的设置是否正确，做法相当简单，至于要加载并运行 manage.py 文件：按下 Ctrl+Alt+R，在弹出的消息框中输入任务名称：



11、创建一个模型

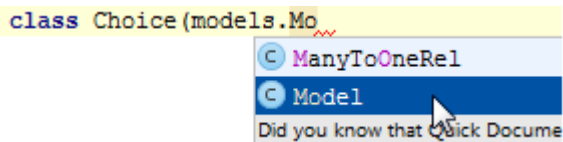
接下来，打开并编辑（open for editing）models.py 文件，注意此时 Pycharm 已经实现导入好了相关库，然后键入以下代码：

```
from django.db import models

class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice = models.CharField(max_length=200)
    votes = models.IntegerField()
```

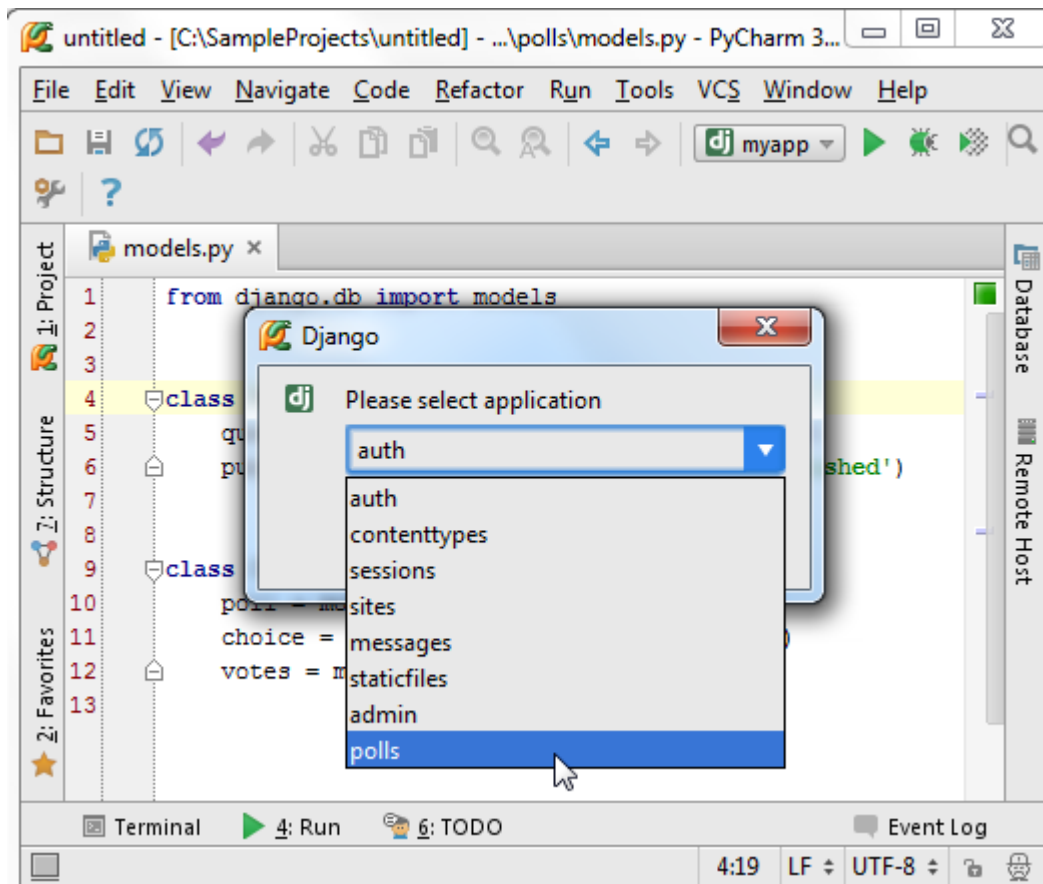
事实上直接对上述代码进行复制粘贴即可，不过这里推荐大家手动输入以体会 Pycharm 强大的拼写提示功能：



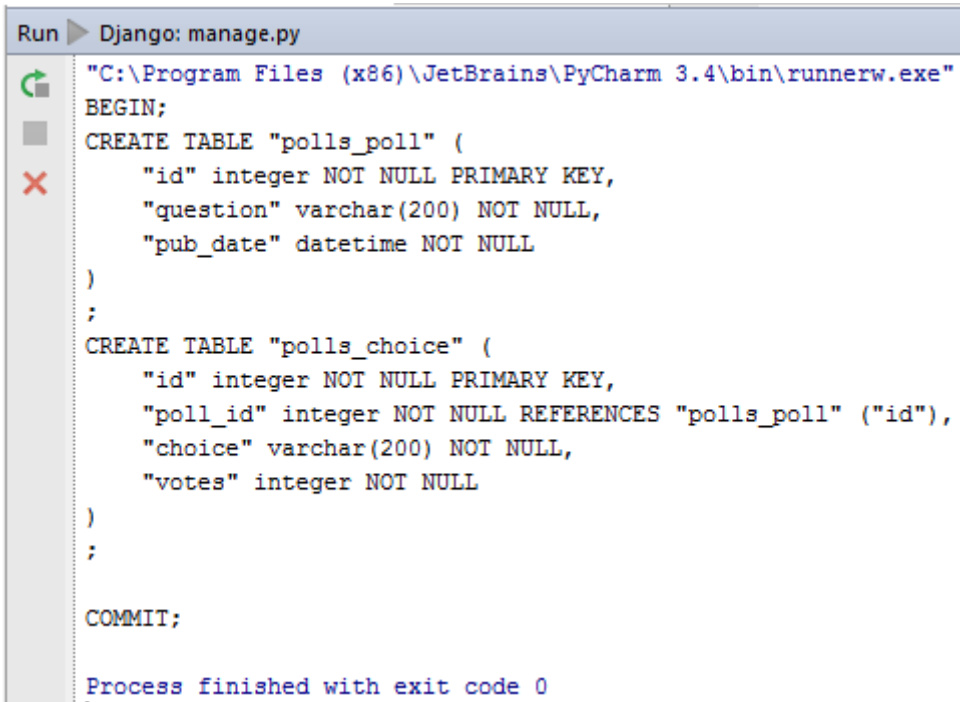
12、创建一个数据库

接下来我们需要为新建模型添加一个表单。再次使用 Ctrl+Alt+R 快捷键：

首先从提示列表中选择 sql，然后选择预期的应用名称：

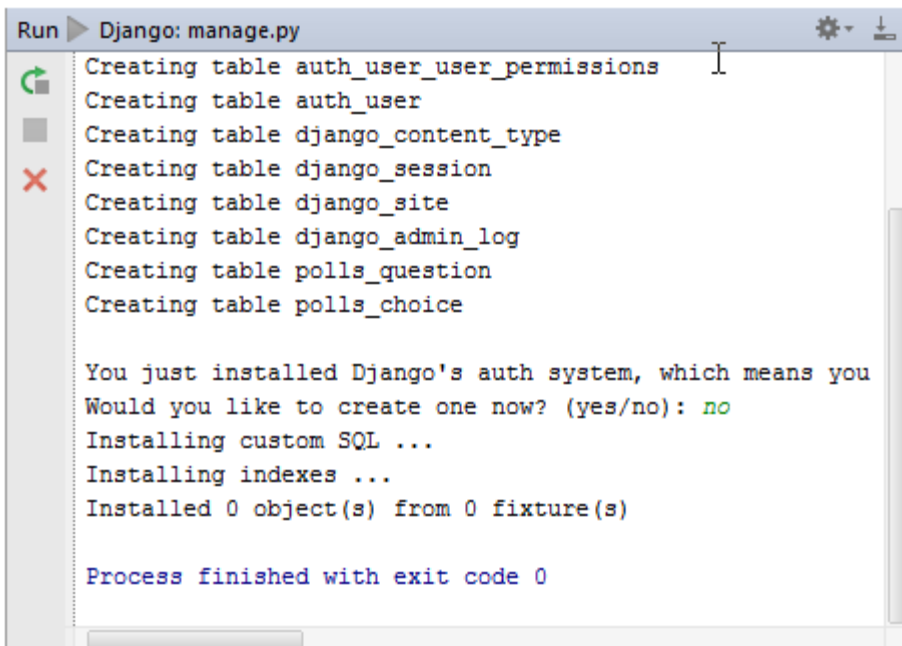


这条命令会为当前类自动添加 SQL 声明:



```
Run ▶ Django: manage.py
"C:\Program Files (x86)\JetBrains\PyCharm 3.4\bin\runnerw.exe"
BEGIN;
CREATE TABLE "polls_poll" (
  "id" integer NOT NULL PRIMARY KEY,
  "question" varchar(200) NOT NULL,
  "pub_date" datetime NOT NULL
)
;
CREATE TABLE "polls_choice" (
  "id" integer NOT NULL PRIMARY KEY,
  "poll_id" integer NOT NULL REFERENCES "polls_poll" ("id"),
  "choice" varchar(200) NOT NULL,
  "votes" integer NOT NULL
)
;
COMMIT;
Process finished with exit code 0
```

第二步, 在提示列表中选择 syncdb 语句, 在提示列表中进行列表的创建, 显示结果如下:



```
Run ▶ Django: manage.py
Creating table auth_user_user_permissions
Creating table auth_user
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table polls_question
Creating table polls_choice

You just installed Django's auth system, which means you
Would you like to create one now? (yes/no): no
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)

Process finished with exit code 0
```

13、完善管理控制函数

由于我们需要对应用进行账户管理, Pycharm 已经在 urls.py 文件中定义好了相关命令。

然而, 我们需要编辑函数的 admin 功能。在 polls 文件夹下创建一个 admin.py 的文件 (Alt+Ins), 然后输入一下代码:

```

from polls.models import Poll, Choice
from django.contrib import admin

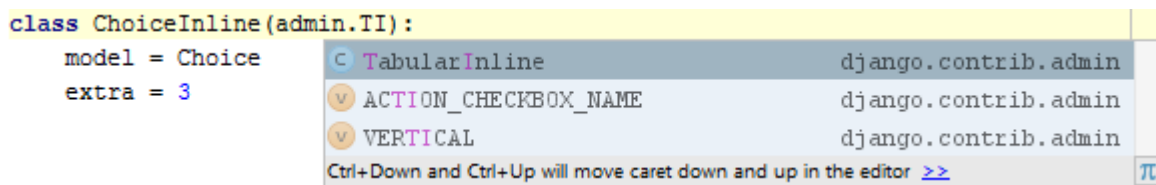
class ChoiceInline(admin.TabularInline):
    model = Choice
    extra = 3

class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']})
    ]
    inlines = [ChoiceInline]

admin.site.register(Poll, PollAdmin)

```

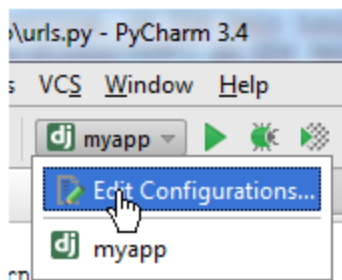
再次展示一下 Pycharm 强大的拼写提示功能：



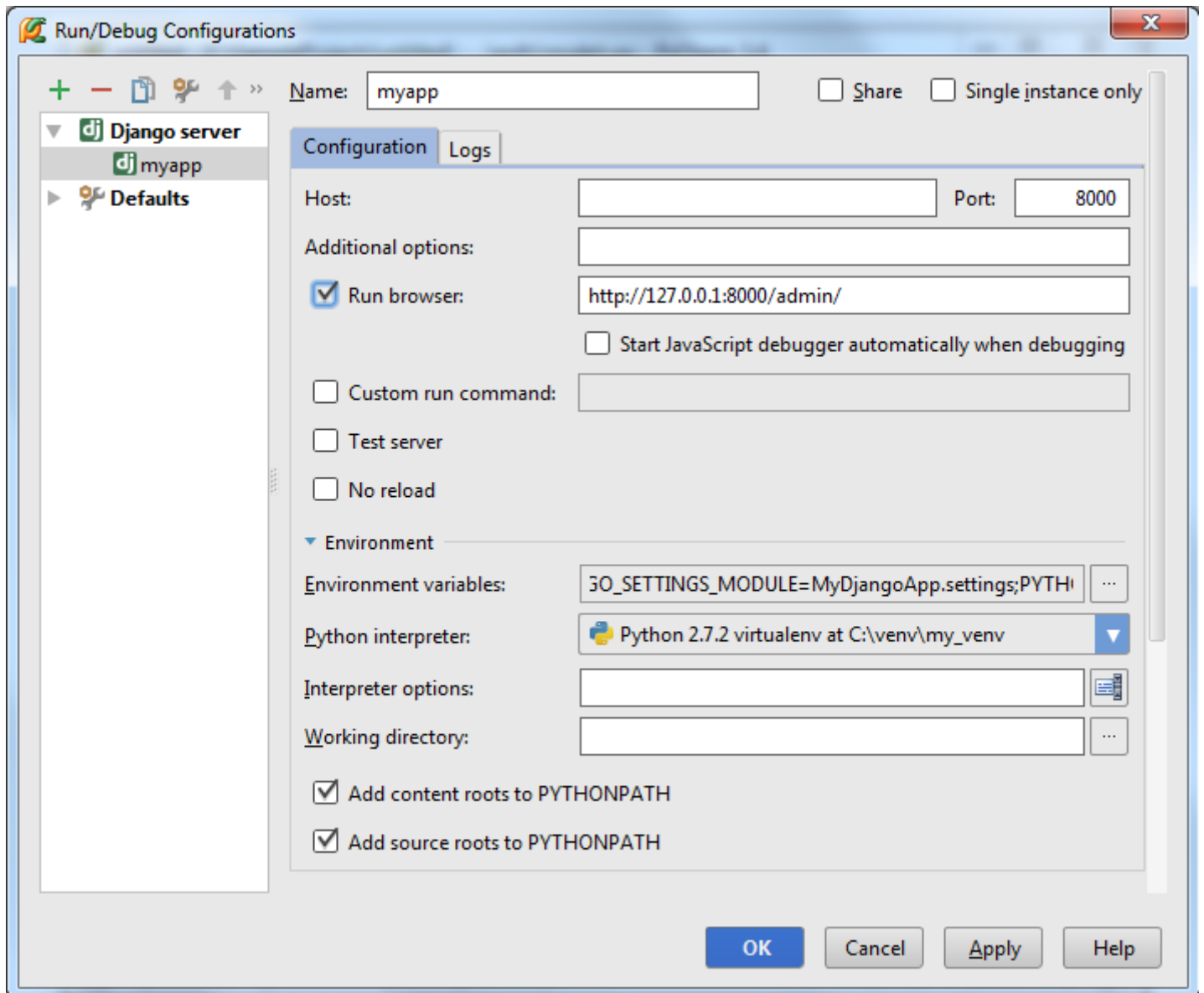
14、运行与调试

现在我们准备前往 admin 界面进行一些设置了。当然，我们很可能需要先运行 Django 服务，进入对应文件目录，在地址栏输入完整的 URL 地址。不过这里 Pycharm 提供了一个轻量级的修改方法：[Django server run configuration](#)

单击主工具栏的 run/debug configurations 选项来进入调试配置模式，然后选择 Edit Configuration（或者在主菜单中选择 Run→Edit Configurations）：

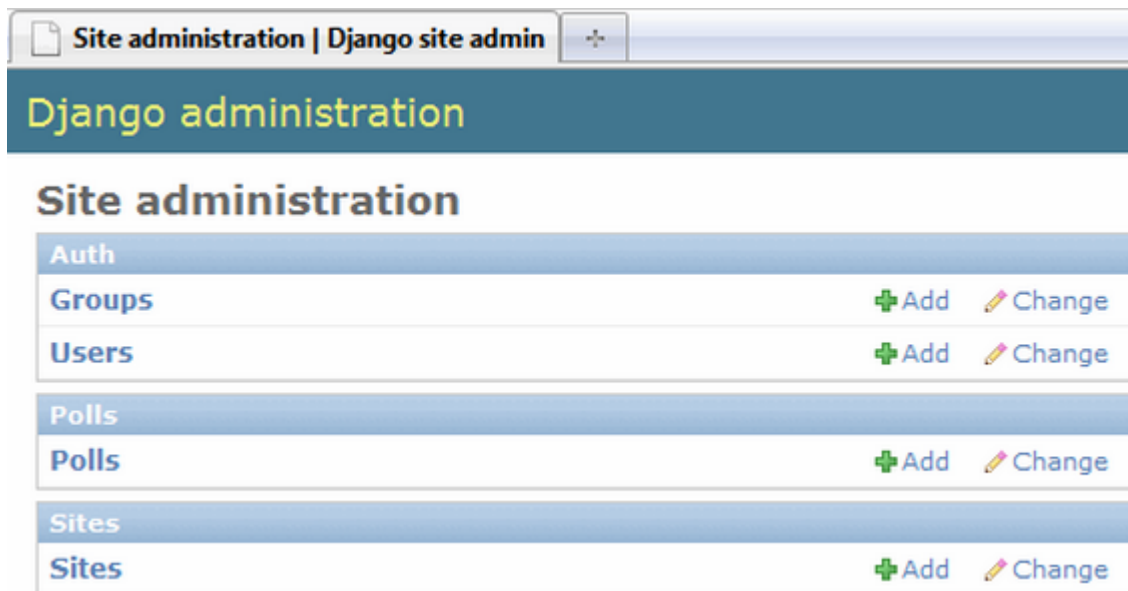


在 [Run/Dug Configuration dialog box](#) 对话框中，输入配置方案名称（这里为 myapp）、默认的浏览器（勾选 Run browser 选项），个性化定制我们的节点界面：



15、加载用户界面

加载并运行这个应用，按下 Shift+F10 或者主工具栏中的 run 按钮，打开标准的 administration 页面，而且必须登录。接下来你可以创建一些 polls 并为其制定相应的问题及候选项：

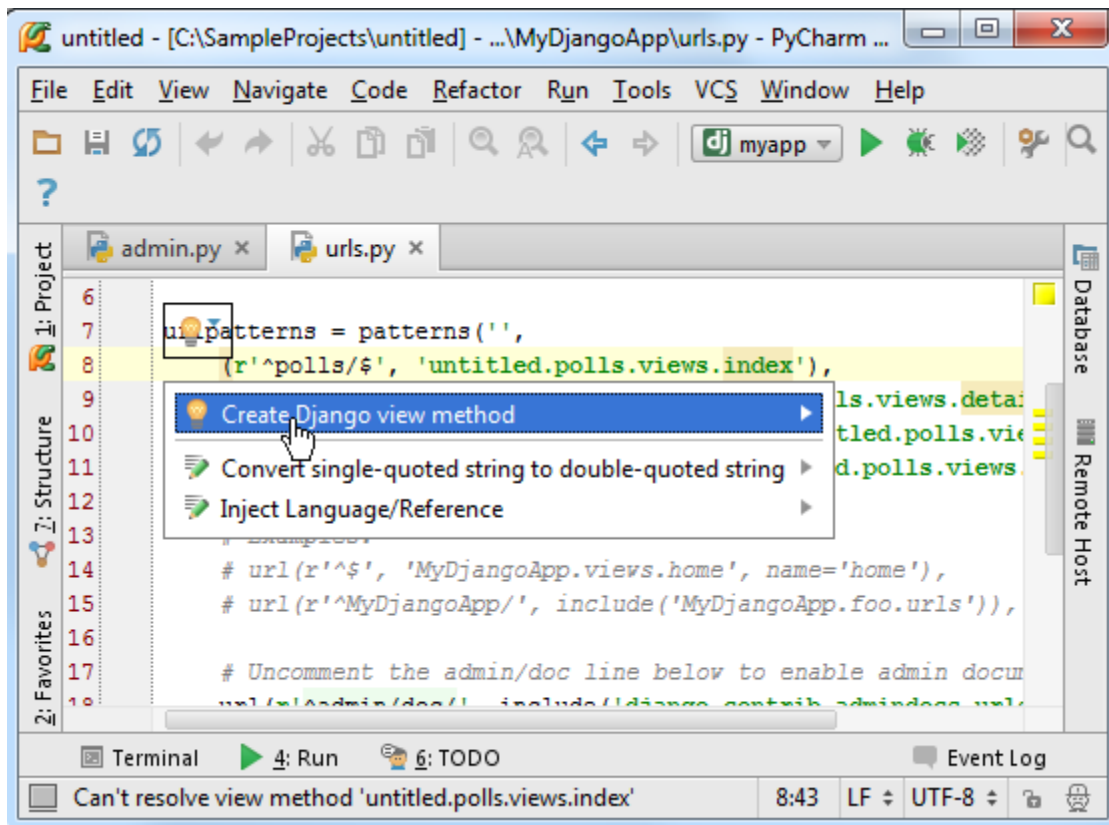


16、创建视图链接

接下来我们准备为应用添加一些子视图，让它拥有"index"、"details"、"results"、"votes"等子页面。首先，我们向 `urls.py` 文件中添加这些子页面的模式（在 Project 窗口中选中该文件然后按 F4）：

```
(r'^polls/$', 'polls.views.index'),  
(r'^polls/(?P<poll_id>\d+)/$', 'polls.views.details'),  
(r'^polls/(?P<poll_id>\d+)/results/$', 'polls.views.results'),  
(r'^polls/(?P<poll_id>\d+)/vote/$', 'polls.views.vote'),
```

这些模式所涉及的页面目前还并不存在，因此需要手动向其中添加一些方法并进行模板关联，这些操作在 Pycharm 的帮助下会变得异常简单：你只需将鼠标指针悬停在一个未定义（Pycharm 会高亮显示那些 unresolved reference 的代码），这是会亮起一个黄色的小灯泡，这就意味着 Pycharm 在此准备了一个快速提示，单击小灯泡（或者按下 Alt+enter）：

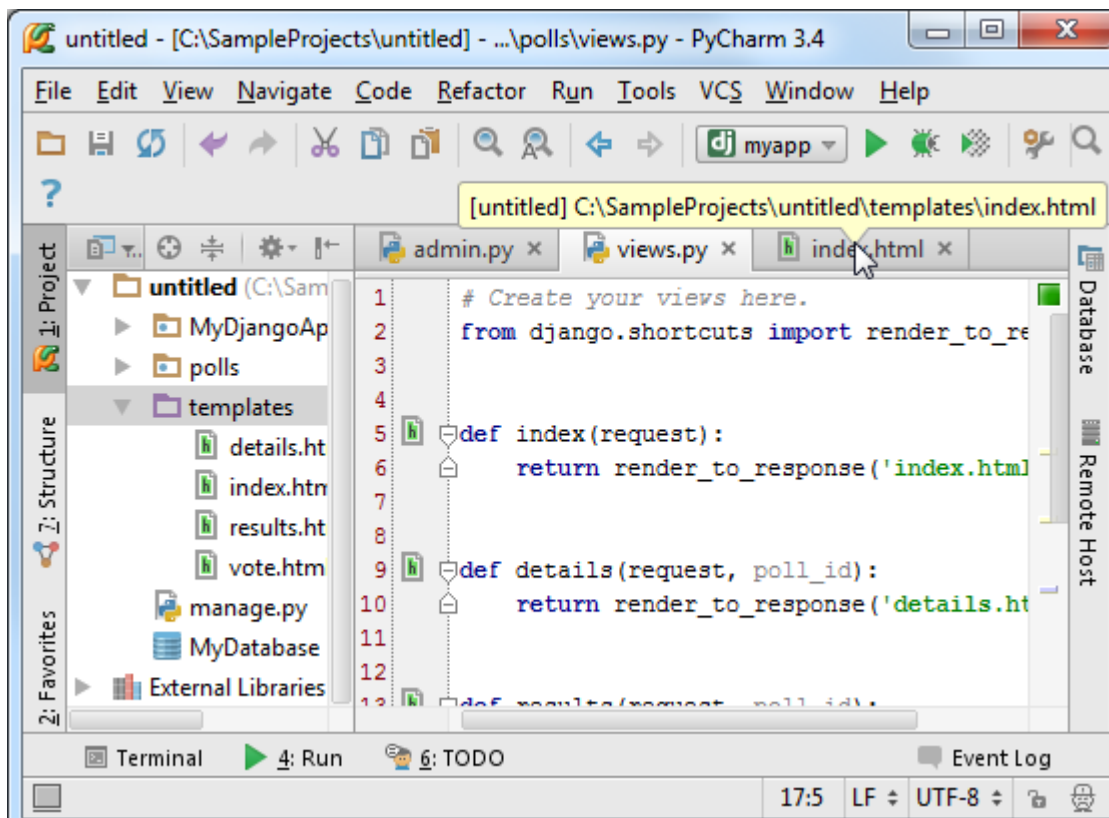


选择 Create Django view method 选项来在 views.py 文件中创建一个视图的成员方法，并与特定的模板文件相关联。

接下来我们会看到以下变化：

templates 目录不再为空，其中包含了我们创建的根模板文件。

views.py 文件中已经包含了根视图的相关方法。



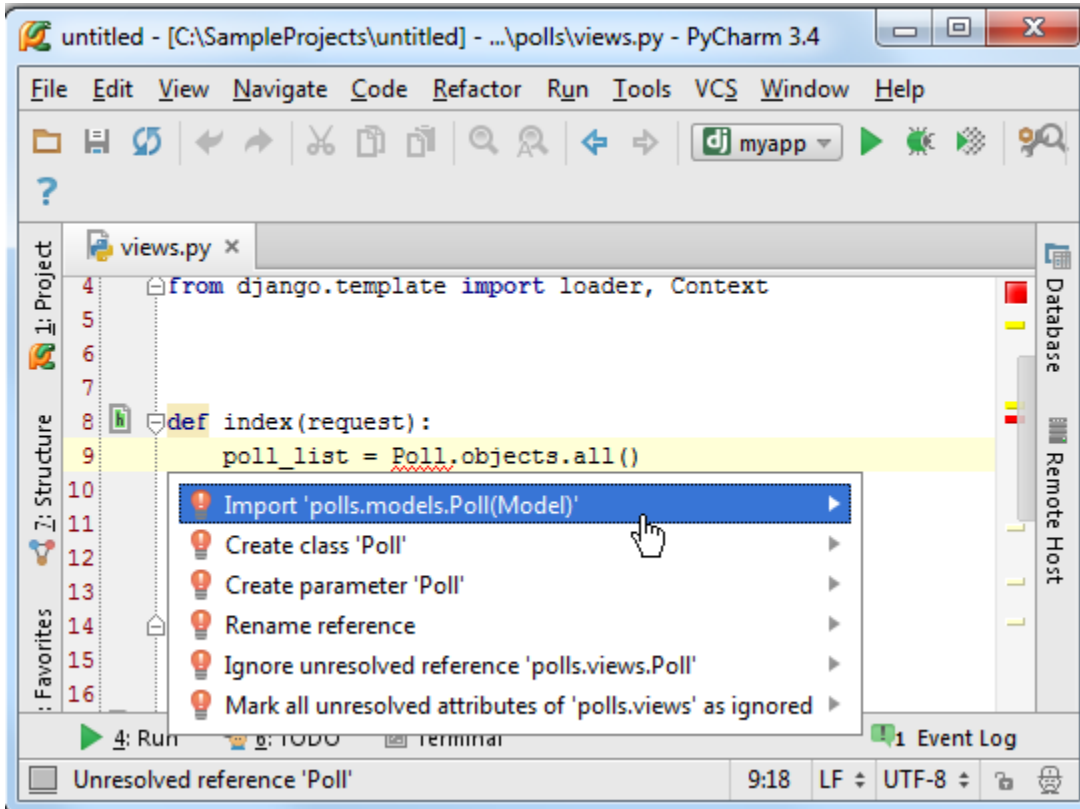
除了添加了 view 的相关方法外，Pycharm 还自动导入的 Django 中的相关操作，并用 `render_to_response` 来标记。

注意 view method 名称左侧的图标，可以通过该图标来查看该方法对应的模板。可以通过 `Create template<name>` 命令在快速创建视图以及对应模板，接下来我们向其中写入代码。

例如我们希望看到 polls 的可用列表，打开 `views.py`，输入以下代码：

```
def index(request):
    poll_list = Poll.objects.all()
    t = loader.get_template('index.html')
    c = Context({
        'poll_list': poll_list,
    })
    return HttpResponse(t.render(c))
```

Pycharm 会给出快捷的拼写提示：



完成后将会有如下显示：

```

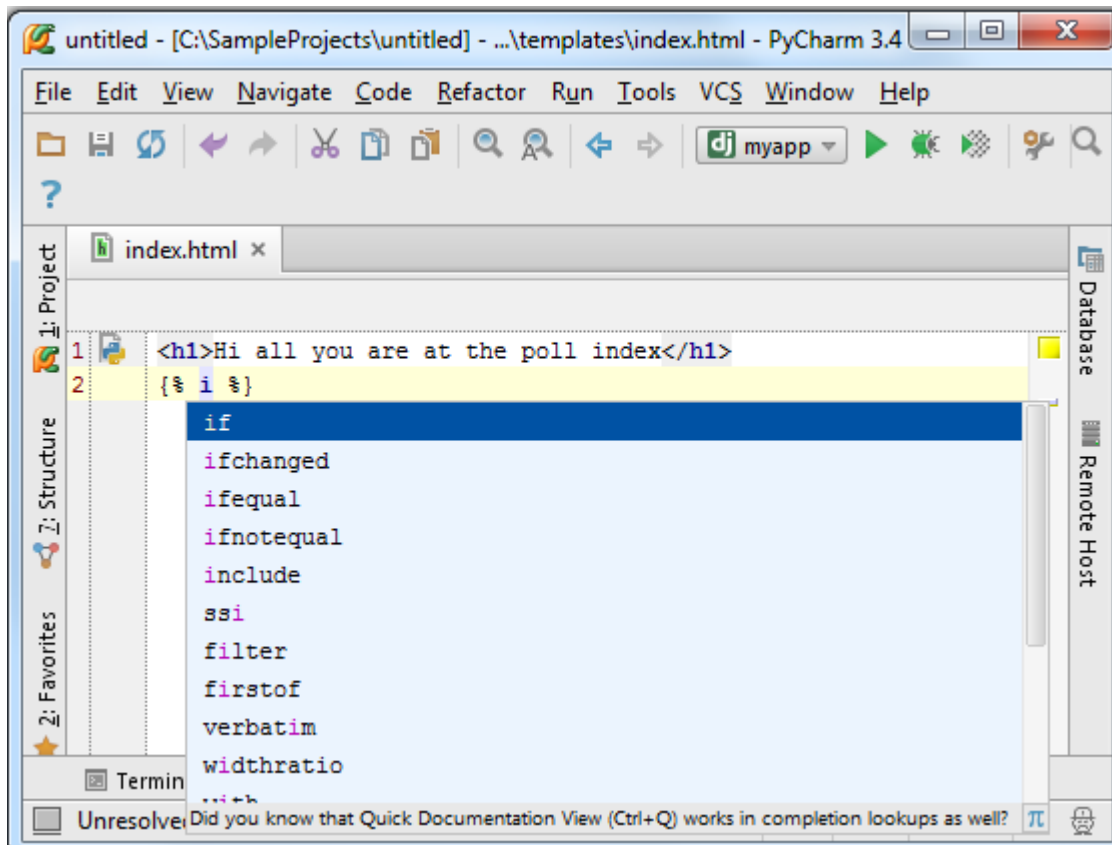
from django.http import Http404, HttpResponse
from django.shortcuts import render_to_response
from django.template import loader, Context
from polls.models import Poll

def index(request):
    poll_list = Poll.objects.all()
    t = loader.get_template('index.html')
    c = Context({
        'poll_list': poll_list,
    })
    return HttpResponse(t.render(c))

```

17、创建模板

接下来我们向模板中添加一些代码。打开 index.html 文件，输入模板代码。这里需要注意的是大括号一定要成对出现，当你输入{%，Pycharm 会在输入光标的后面自动添加另一个括号。这里你可以通过 Ctrl+Space 来进行拼写提示。

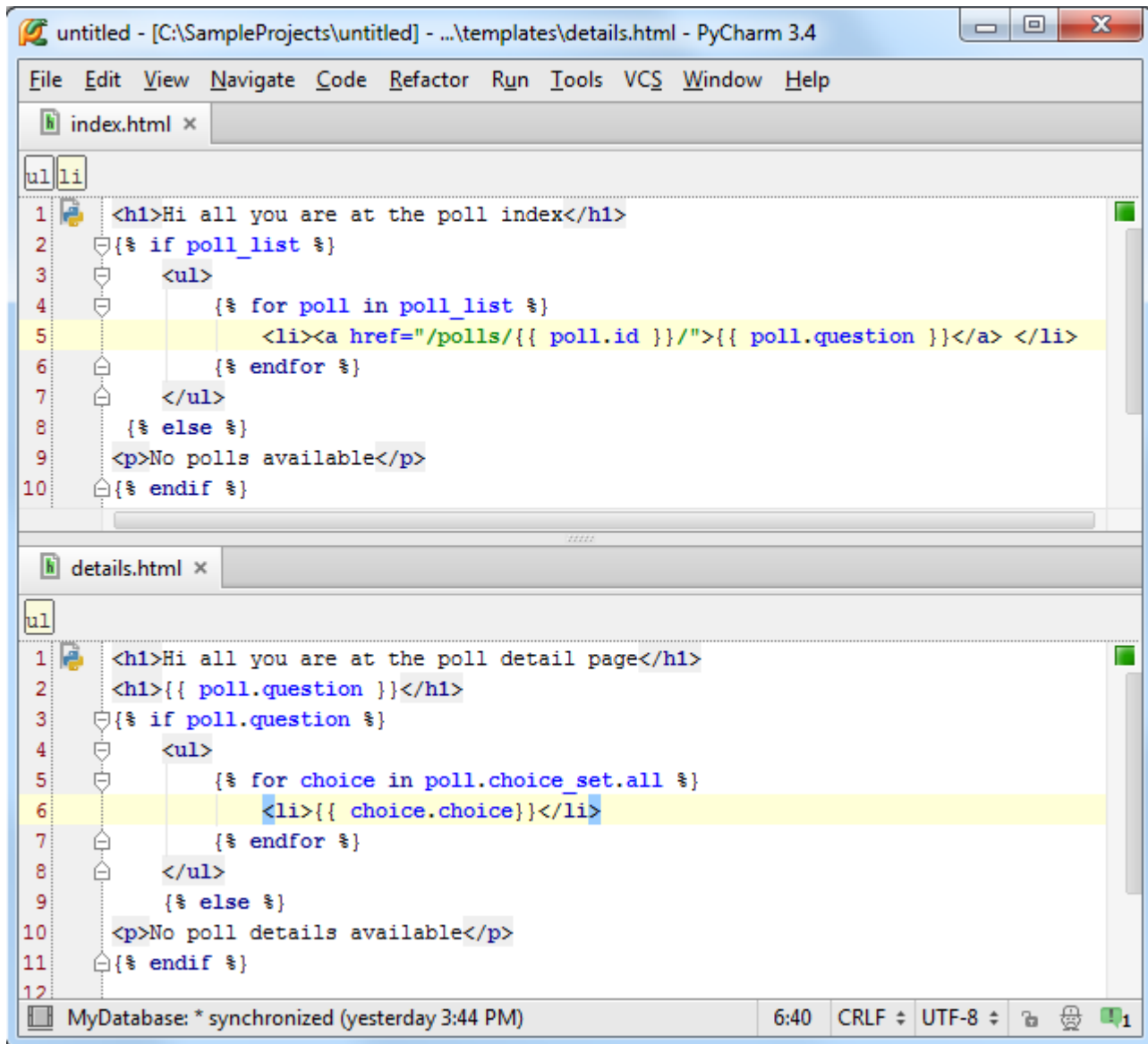


当需要输入 HTML 类型标签时，PyCharm 同样设计了帮助系统：

Ctrl+Space 调用拼写提示功能。

当输入一个括号时，会自动生成另一个括号以进行匹配

接下来拟至于一步一步晚上你的模板代码，最终结果如下：

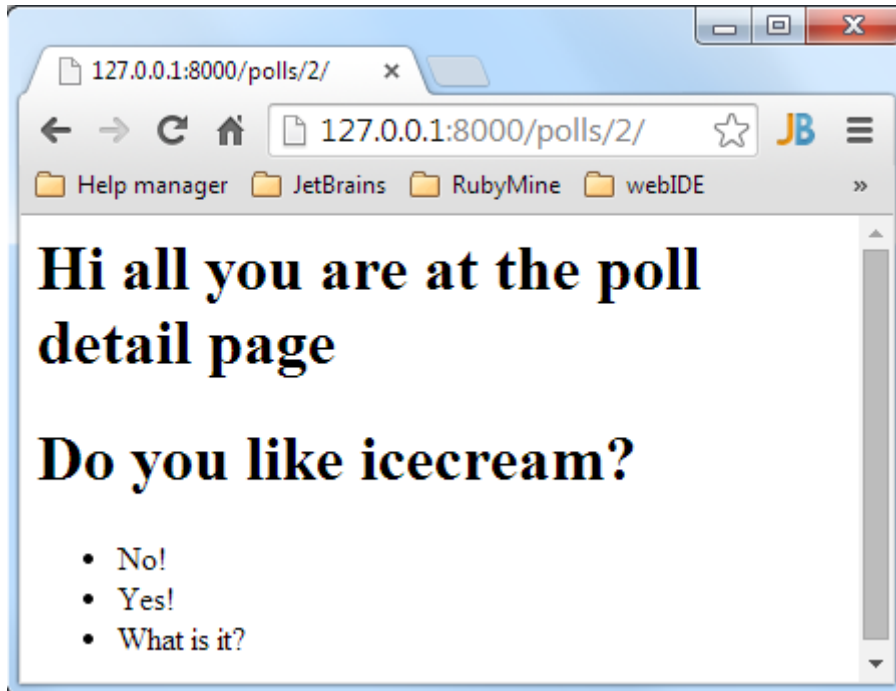


18、大功告成

让我们核实一下 polls 的变量列表,发现我们的 admin 能够正常使用,并且能够在地址栏中显示对应的 URL 地址(/admin/type/polls/):



单击以查看详细信息:



最全 Pycharm 教程（9）——创建并运行一个基本的 Python 测试程序

1、主题

这里我们着重介绍 Pycharm 如何帮助我们创建并运行一个基本的测试程序。至于如何编写具体的测试程序，参见之前的文章。

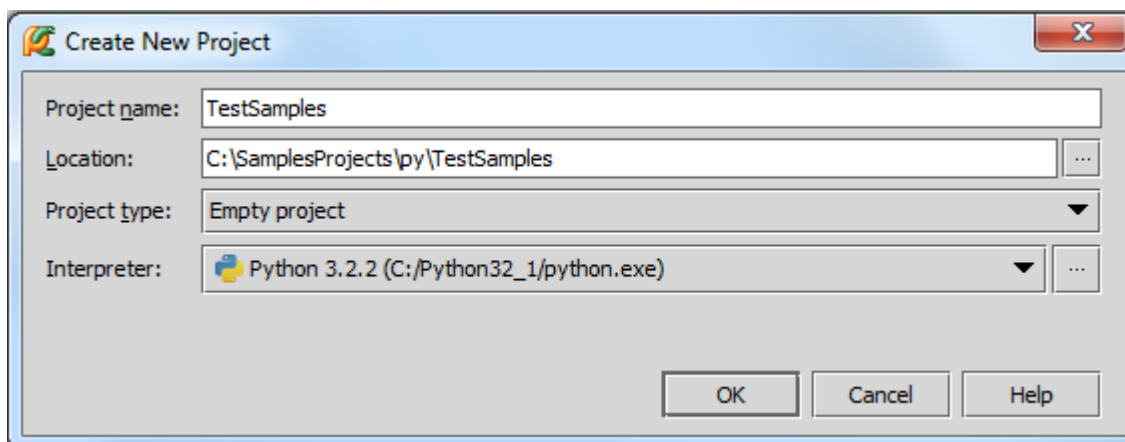
2、准备工作

确认你电脑上已经安装了 Python 解释器，2.4 到 3.3 的版本均可。

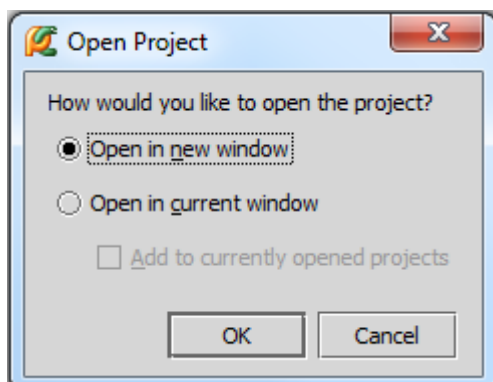
3、创建一个简单的 Python 工程

在主菜单中，选择 **File | New Project**

在创建工程对话框中，输入工程名称（这里暂定为 TestSamples），选择工程类型（这里选择一个空的工程），并指定 Python 解释器版本：

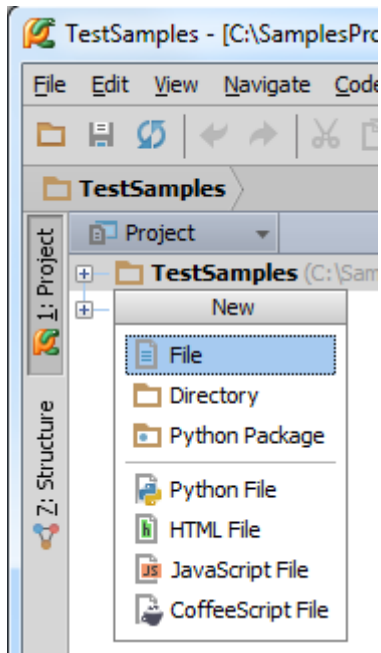


单击 OK，选择显示工程所需窗口，这里我们选择第一个选项——在一个独立的新窗口打开我们的工程：

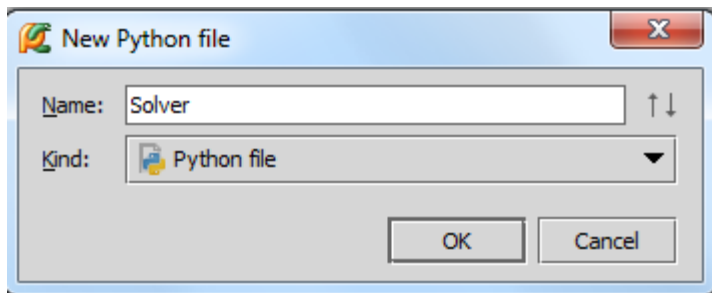


4、创建一个 Python 类

按下 **Alt+Insert**，选择 Python file：



在新建 Python 对话框中，输入文件名称：



可以看到新建的 Python 文件中已经定义好了 `__author__` 以及 `__project__` 变量，接下来我们创建一个简单的脚本来实现解二次方程的功能：

```

import math

class Solver:

    def demo(self, a, b, c):

        d = b ** 2 - 4 * a * c

        if d >= 0:

            disc = math.sqrt(d)

            root1 = (-b + disc) / (2 * a)

            root2 = (-b - disc) / (2 * a)

            print(root1, root2)

        else:

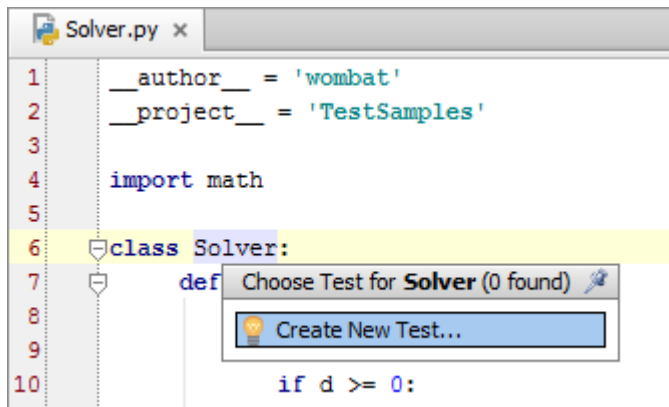
            raise Exception

Solver().demo(2,1,0)

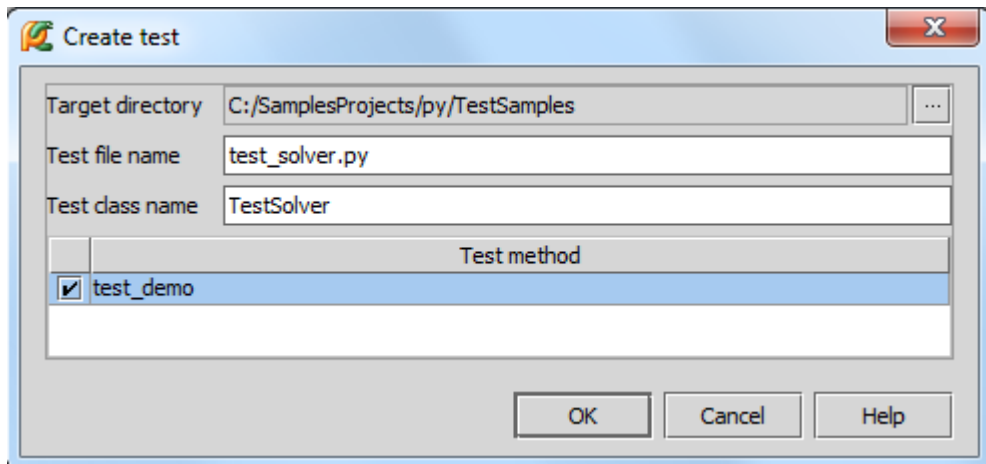
```

5、创建测试程序

右击类名，在快捷菜单中选择 **Go to | Test**（也可以直接按 **Ctrl+Shift+T**）：



在 **Create test** 对话框中，输入路径和名称，勾选复选框中的 **test_demo** 函数选项：



结果如下：

```

1  from unittest import TestCase
2
3  __author__ = 'wombat'
4  __project__ = 'TestSamples'
5
6
7  class TestSolver(TestCase):
8      def test_demo(self):
9          self.fail()

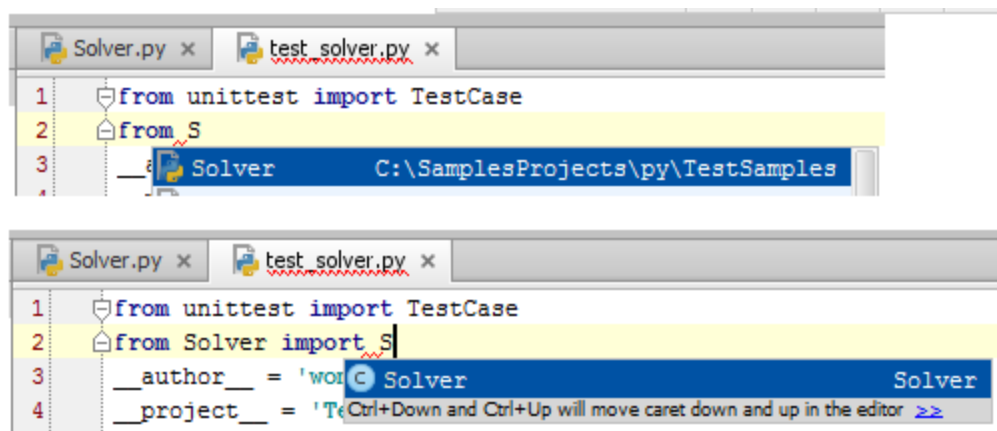
```

如你所见，所创建的测试程序满足 [Python unit testing framework](#) 标准——从 `unittest` 模块中导入响应的测试类，并将测试函数名称加上“test”前缀。

然而，目前的测试单元只是一个基本的框架，需要进行修改。首先导入相关模块：

```
from Solver import Solver
```

在输入时建议使用拼写提示功能，通过 **Ctrl+Space** 组合键，PyCharm 将给出合适的模块及类名称提示：



如果 `import` 声明的语句显示为灰色，则说明当前导入的模块还没有被使用。

接下来我们创建一个函数用来抛出判别式为负数的异常，向 `test` 类中加入以下代码：


```
def test_negative_discr(self):

    s = Solver

    self.assertRaises(Exception,s.demo,2,1,2)
```

```
class TestSolver(TestCase):
    def test_negative_discr(self):
        s = Solver
        self.ar
def test_assertRaises(self, excClass, callableObj, args, kwargs) TestCase
```

测试单元的最终代码如下：

```
from unittest import TestCase

from Solver import Solver

class TestSolver(TestCase):

    def test_negative_discr(self):

        s = Solver

        self.assertRaises(Exception,s.demo,2,1,2)

    def test_demo(self):

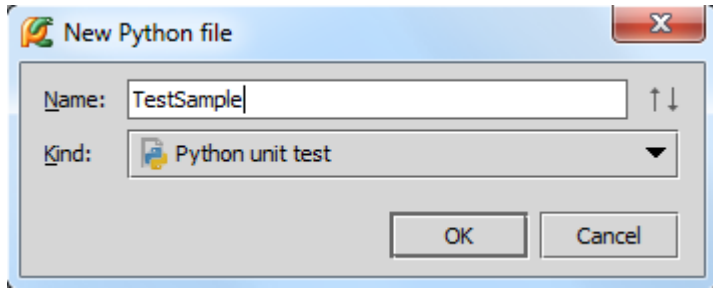
        self.fail()
```

我们的测试方案包含两个测试方法：test_negative_discr and test_demo，当然后者通常不会被执行。

注意此时的 import 语句已经不再变灰色，因为我们已经在 test_negative_discr 函数中用到了 Solver 类。

6、创建测试方案的其他方法

让我们尝试其他方法来创建测试方案。按下 Alt+Insert 组合键，在弹出的菜单中选择 **Python file**，接下来在 **New Python file** 对话框 Kind 中选择 Python unit test 选项，然后键入测试方案名称：



此时 Pycharm 会创建并初始化一个解决方案程序，打开并编辑它：

```
TestSample.py x
1  __author__ = 'wombat'
2
3  import unittest
4
5  class MyTestCase(unittest.TestCase):
6      def test_something(self):
7          self.assertEqual(True, False)
8
9
10 if __name__ == '__main__':
11     unittest.main()
```

和之前的步骤类似，最终的测试代码为：

```
import unittest

from Solver import Solver

class MyTestCase(unittest.TestCase):

    def test_negative_discr(self):

        s = Solver

        self.assertRaises(Exception)

    def test_something(self):

        self.assertEqual(True, False)

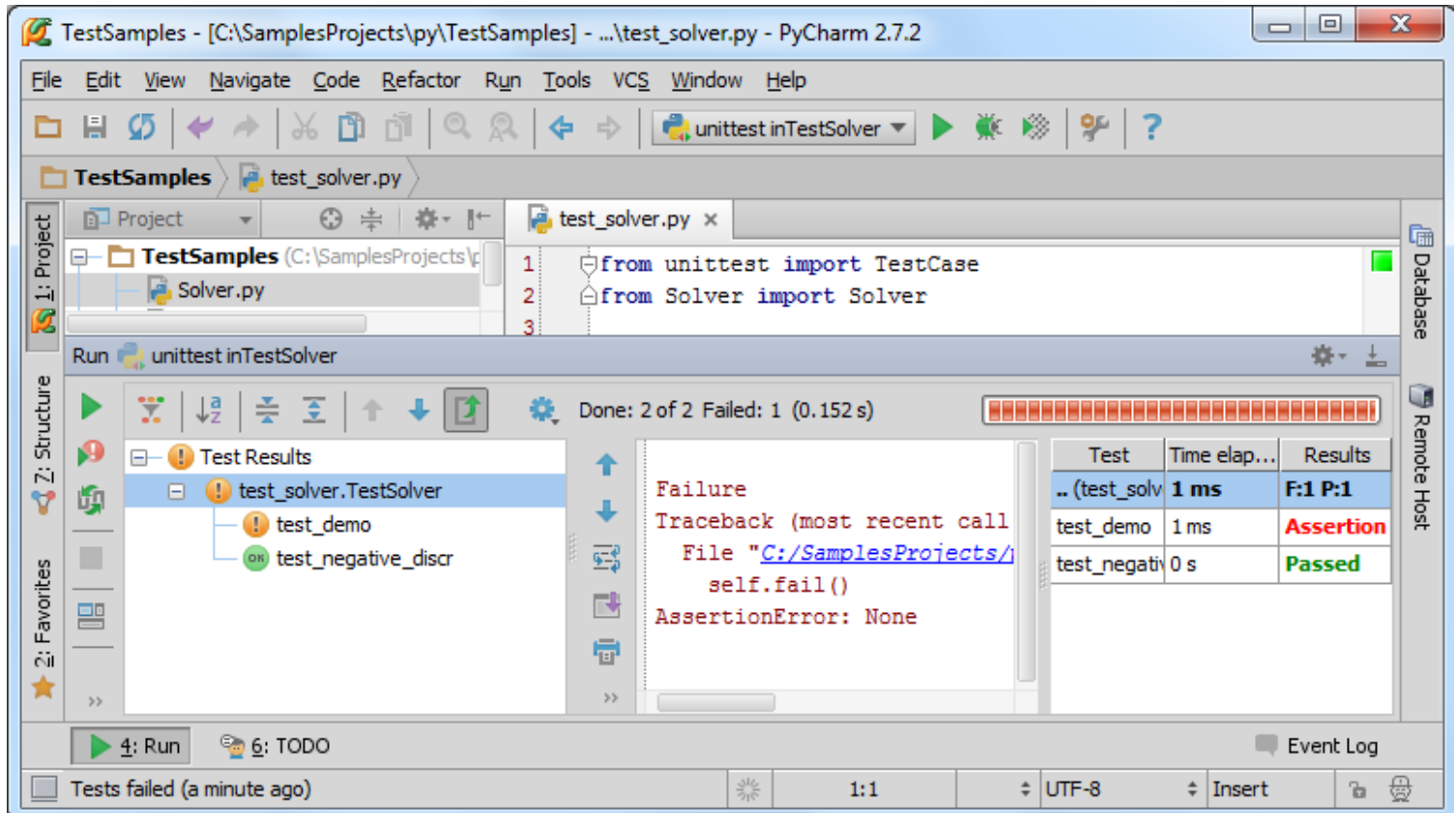
if __name__ == '__main__':

    unittest.main()
```

7、运行测试单元

为了执行我们的测试单元，Pycharm 建议使用一个新的配置文件 `run/debug configuration`，这个配置模板已经是预先定义好的，我们直接使用即可。按下 **Ctrl+Shift+F10**，或者在类内右击，选择 **Run unittests in test_solver**：

运行结果如下：



最全 Pycharm 教程（10）——Pycharm 调试器总篇

1、准备工作

- (1) Pycharm 版本为 3.0 或者更高版本
- (2) 至少安装了一个 Python 解释器并且已经正确配置
- (3) 已经创建了一个 Python 工程

2、主要内容

介绍如何通过 Pycharm 来调试脚本文件，以及各个工具按钮的作用等等，至于 Python 编程方法，请参见 [Python documentation](#)。

3、待复习知识

为了完成本篇教程的内容，需要用到前面的两个重要知识点：

- (1) Run/debug 配置文件的相关知识
- (2) 断点的相关知识

4、Run/debug 配置知识

每当你通过 Pycharm 来运行或者调试一个脚本文件时，都需要一个特殊的文件来记录脚本的名称、工作目录以及其他的重要调试信息。Pycharm 已经针对特定模式预先创建好了配置文件，避免我们手动去创建。

每次你单击运行或者调试按钮时（或者通过快捷菜单执行相同的操作），我们实际上都是在当前工作模式中加载了对应的配置文件。详见 [product documentation](#)。

5、断点

一个断点标记了一个代码行，当 Pycharm 运行到该行代码时会将程序暂时挂起。Pycharm 提供了几种形式的断点 [types of breakpoints](#)，其图标 [icon](#) 各不相同。详见 [product documentation](#) 中的 [Breakpoints tutorial](#)。

6、实例

在你的 Python 工程中，创建一个新的 Python 文件，命名为 ThreadSample.py，然后输入以下代码：

```
import threading
import time

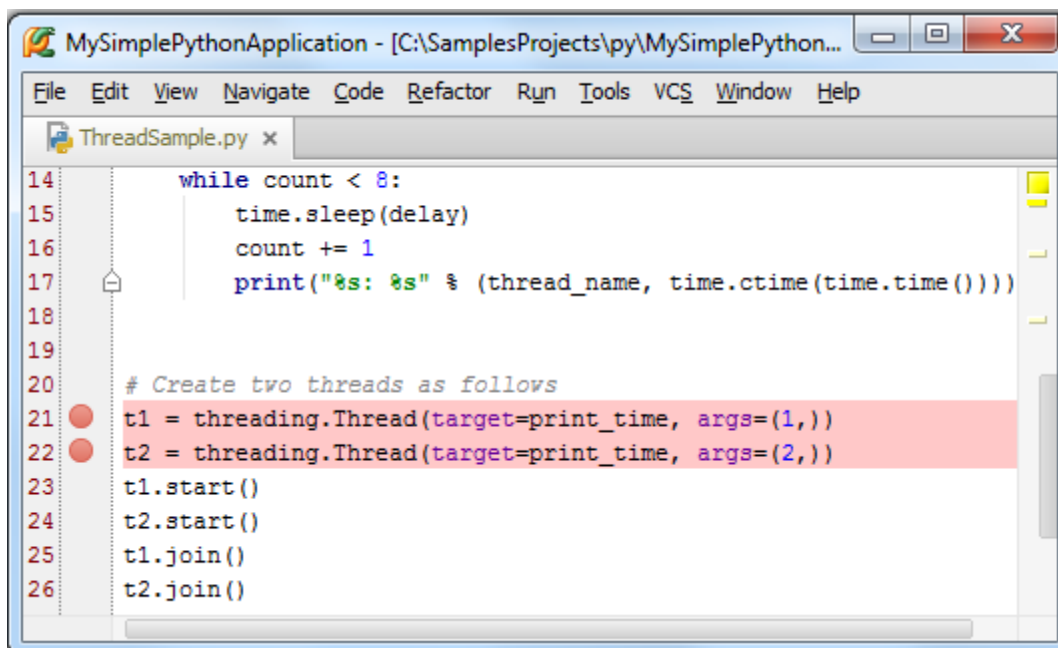
def get_thread_name():
    t = threading.current_thread()
    return t.name

def print_time(delay):
    """Define a function for the thread."""
    thread_name = get_thread_name()
    count = 0
    while count < 8:
        time.sleep(delay)
        count += 1
        print("%s: %s" % (thread_name, time.ctime(time.time())))

# Create two threads as follows
t1 = threading.Thread(target=print_time, args=(1,))
t2 = threading.Thread(target=print_time, args=(2,))
t1.start()
t2.start()
t1.join()
t2.join()
```

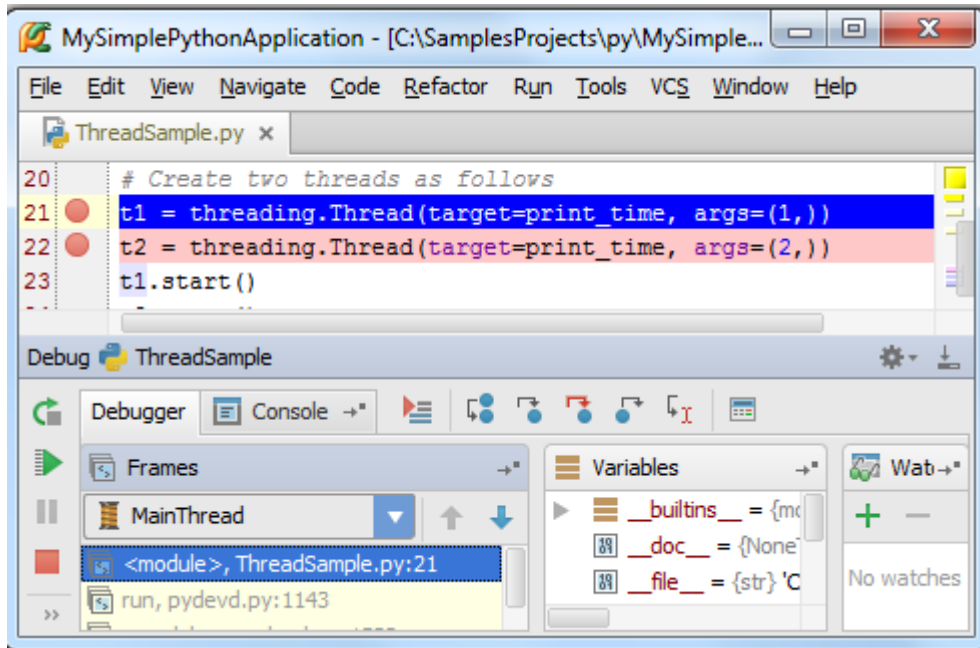
7、设置断点

首先，在源码中设置断点。通过单击代码左侧的空白槽来在对应位置生成断点：



8、开始调试

选择 run/debug configuration "ThreadSample", 然后按下 Shift+F9 (或者单击工具栏中的绿色蜘蛛形式的按钮), 调试开始, 并在第一个断点处停止:



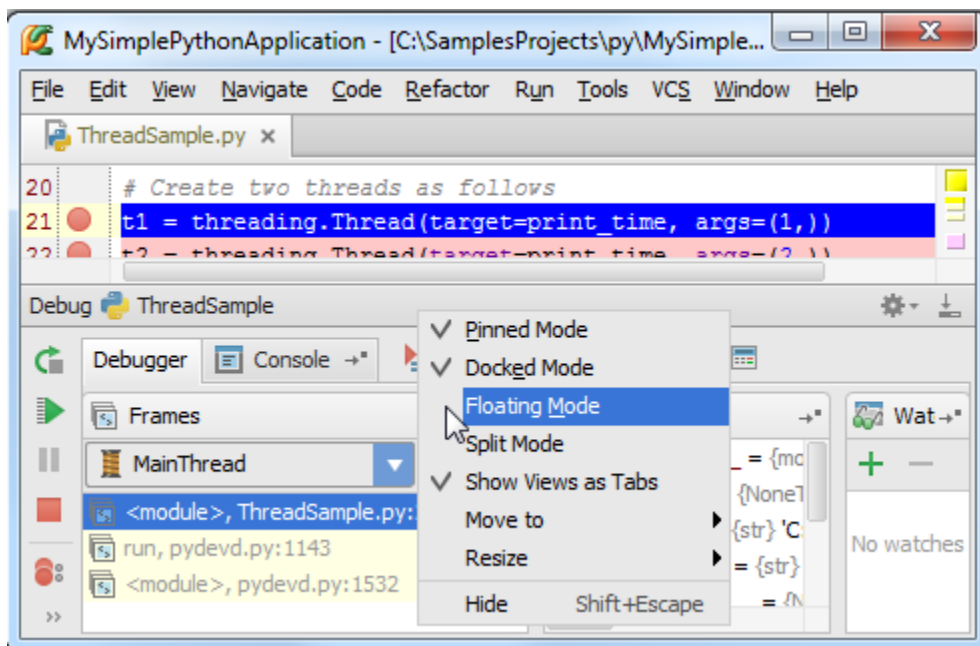
断点所在的行变为蓝色, 说明 Pycharm 已经击中了这个断点, 但尚未执行这行代码。

9、更改调试窗口的布局

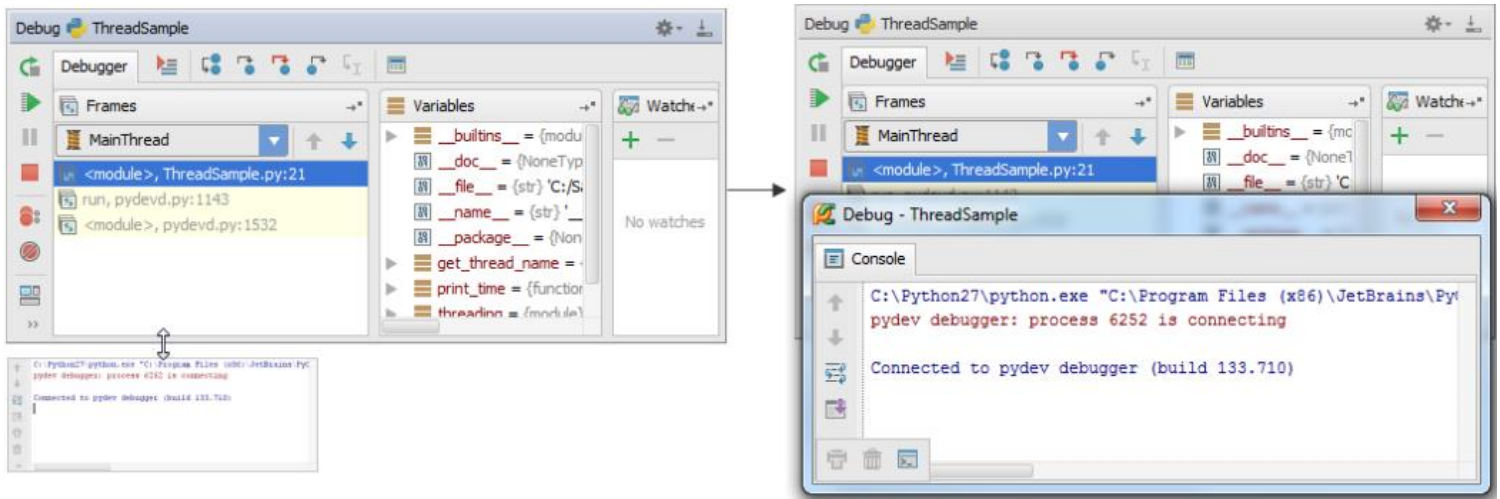
此时 Pycharm 进入调试界面模式 [Debug tool window](#), 各个控件的具体功能详见 [know how to use this tool window](#)。

加入你并不喜欢当前的默认布局, 例如你希望将调试器输出窗口作为一个独立的窗口显示以便更方便的观察当前的调试状态, 你可以对布局进行个性化定制。

首先, 我们先把调试工具窗口独立出来, 只需要右击窗口的标题栏, 然后选择 **Floating mode**:



接下来将控制台窗口移动成为一个独立的窗口, 只需要拖动控制台窗口将其拖出测试窗口即可:



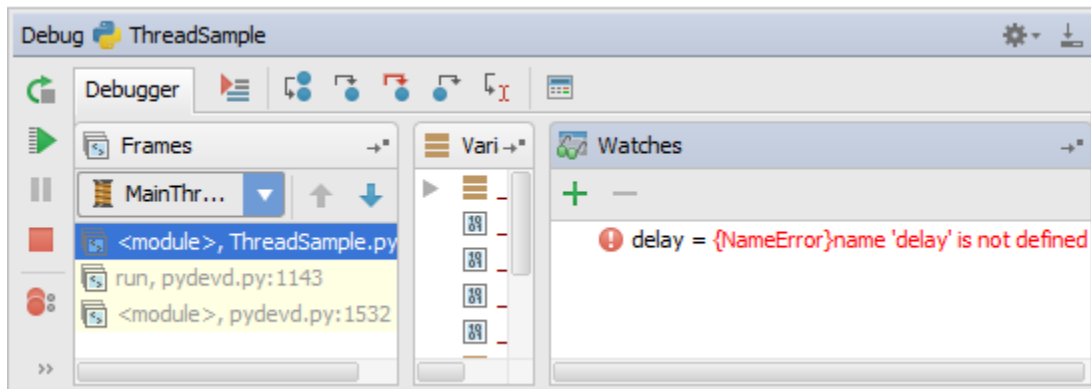
可以通过单击工具栏上的  按钮来恢复到默认布局。

更多信息请参见 [PyCharm Tool Windows](#) 和 [Moving tabs and area](#)

10、添加一个变量查看器

接下来我们介绍如何在调试过程中观察变量的状态。我们需要对其设置一个查看器。在 [Watches](#) 窗口中，单击绿色的加号，输入期望查看的变量名称，例如这里输入 `delay`，然后回车。当然你也可以采用另外一种方式：在编辑窗口中右击变量名，在快捷菜单中选择 **Add to watches**：

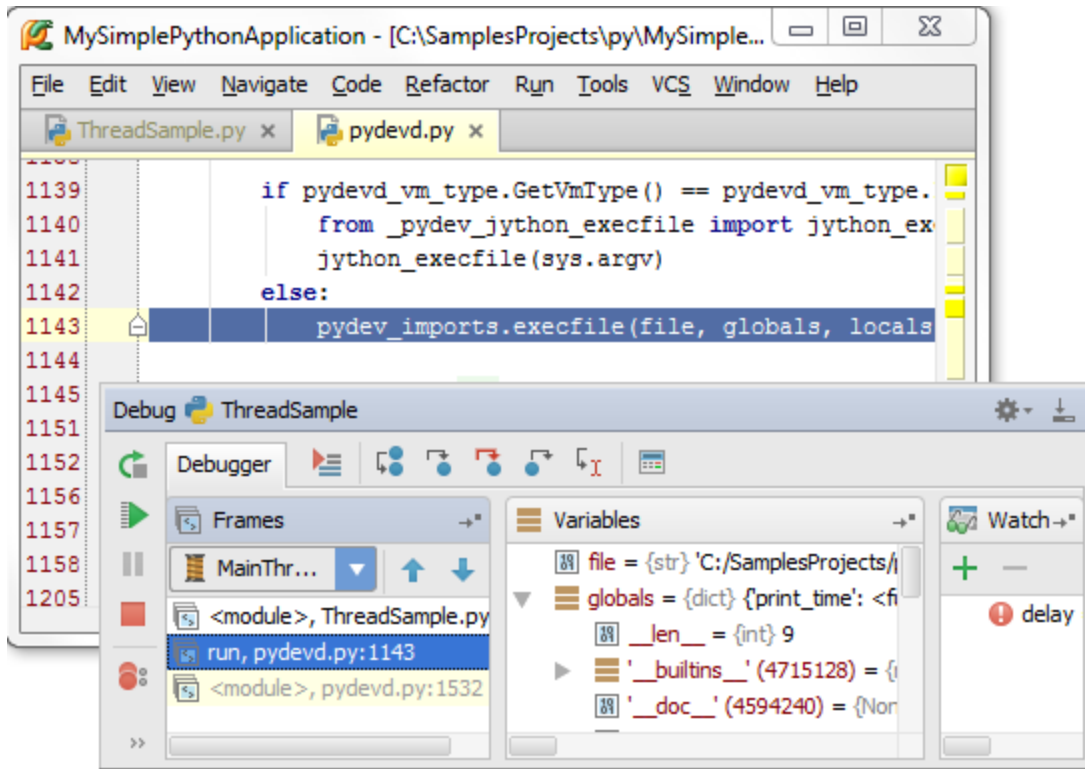
此时观察 [Watches](#) 窗口，发现 `delay` 变量目前尚未定义：




稍后你将会看到如何对这个变量进行赋值，以及其在 [watches](#) 窗口中的变化。接下来我们为 `get_thread_name()` 函数设置一个查看器作为练习。

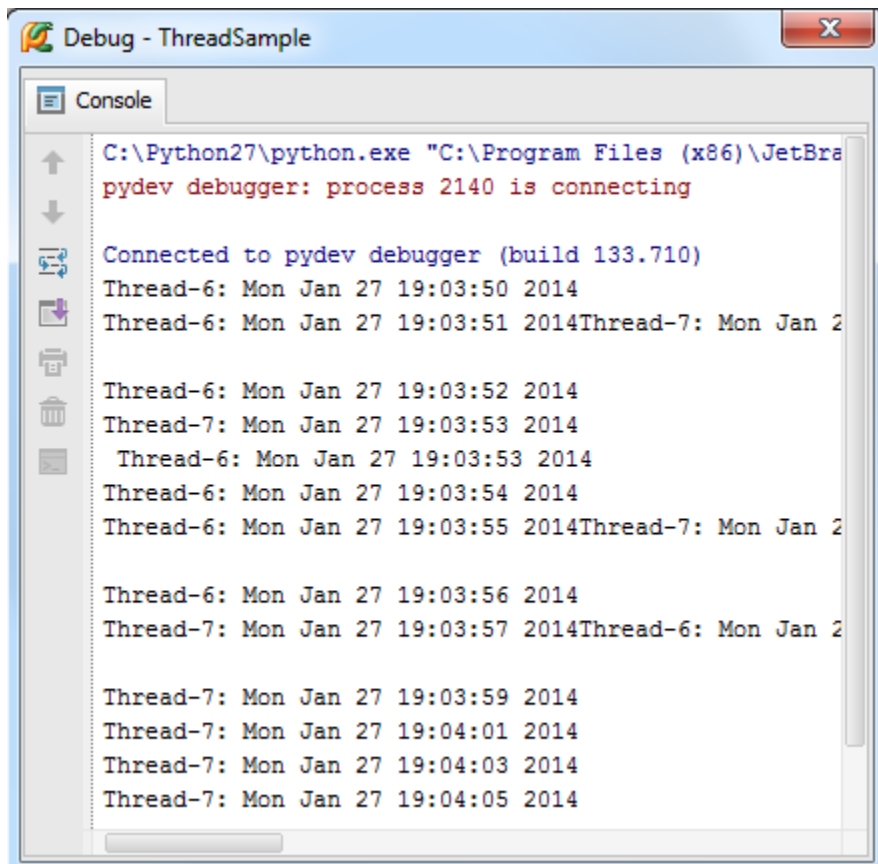
11、浏览帧

此时你能看到一个名为 `MainThread` 的进程，其中包含三帧。单击每一帧来显示其变量状态以及相对应的 `py` 文件，同时会对有问题的代码行以高亮显示：




12、简单的调试

在每个断点出都单击  按钮来时程序继续运行，观察控制台的脚本输出：

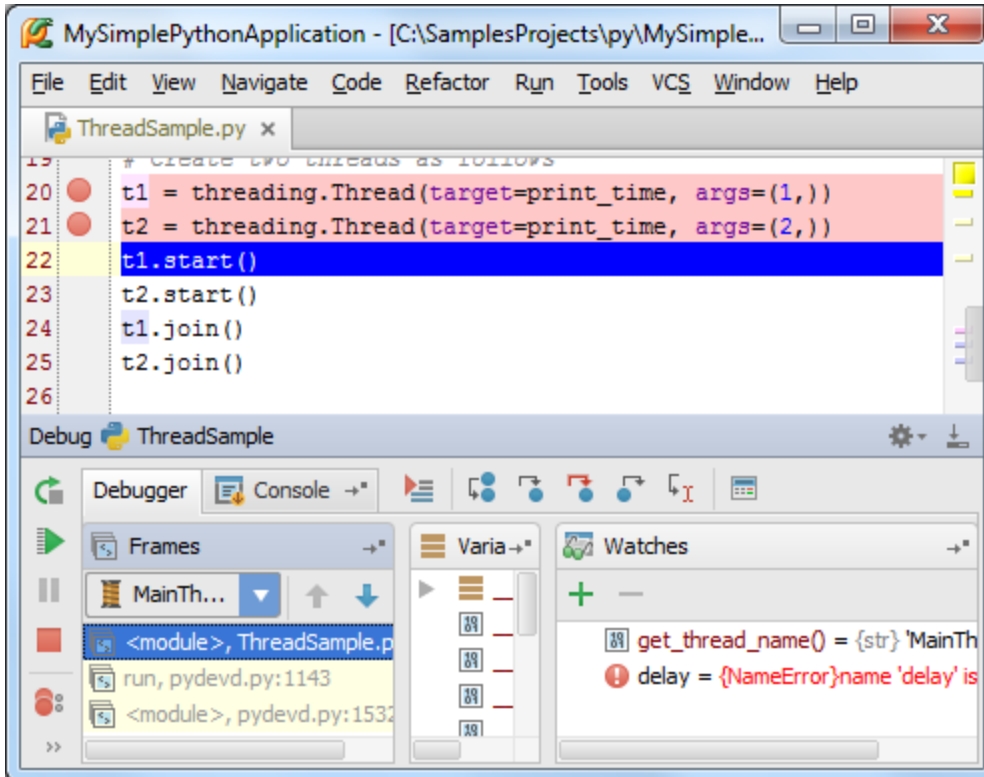



13、步进式脚本调试

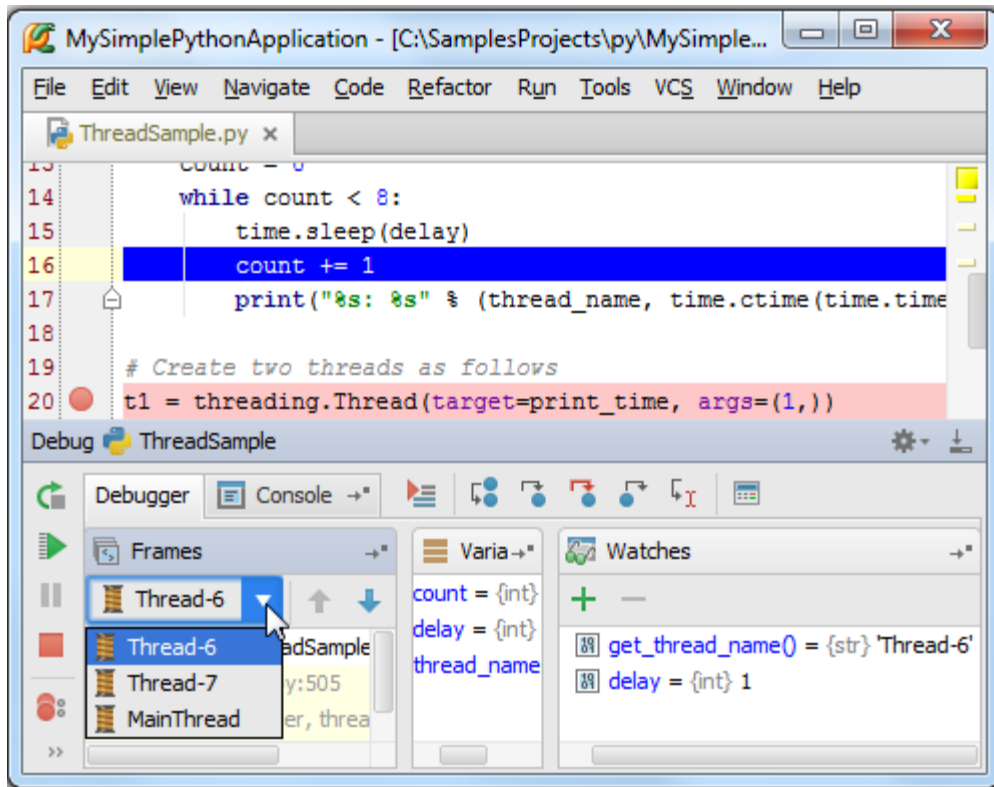
让我们对当前脚本进行进一步的调试。单击  重启调试进程，程序再次运行到第一个断点处并暂停。

在调试窗口的顶端，你会看到一系列包含步进调试按钮的工具栏 [stepping buttons](#)。

单击 ，或者按下 F8，你会发现蓝色标记移动到了下一行：



与此同时，当你暂停了脚本执行时（单击  按钮），你能看到高亮表示的函数 `print_time()`，你可以选择其中的任何一个进程，并观察变量的变化：



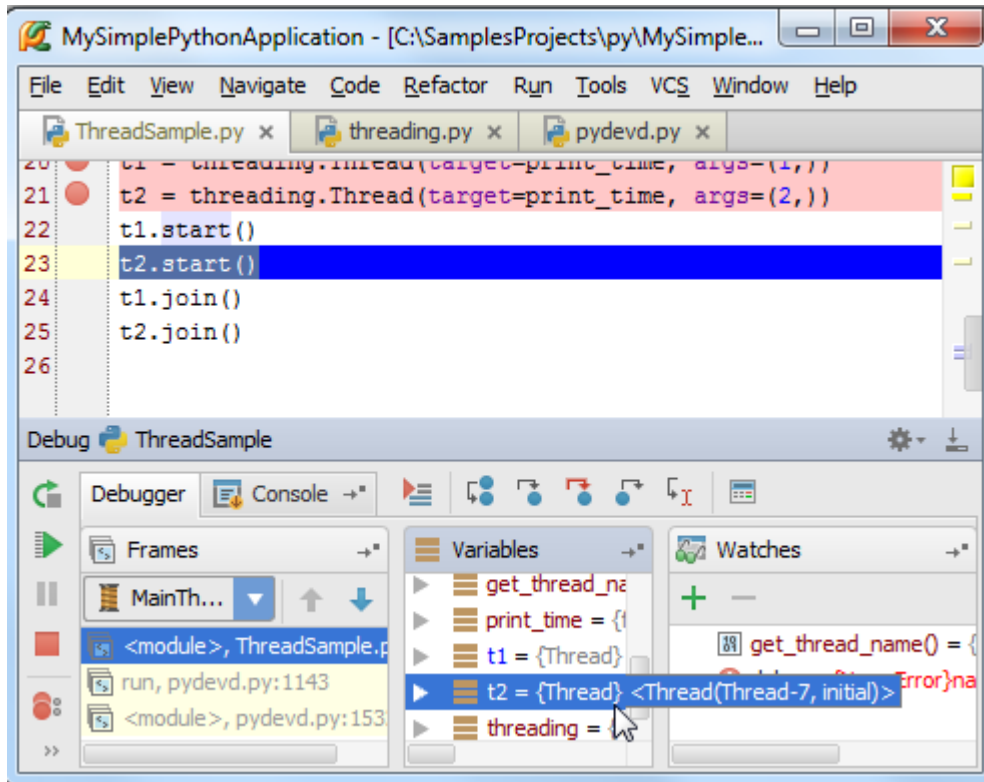
14、运行到当前光标处

假如你不想加入断点，但仍希望代码能够调试运行到某一指定的代码行，如何操作？

返回调试界面的第二个断点处，在对应行插入输入光标。









`t2.start()`

单击  按钮，或者按下 Alt+F9 快捷键，该行代码变为高亮显示：



15、如何调用 Debug 命令

值得一提的是所有的调试操作不仅仅可以通过调试工具栏的对应按钮来完成，还可以通过主菜单中 Run 菜单下的命令来实现，以及相关的快捷菜单项。下面给出一些常用的调试菜单命令及其对应的快捷键列表：

Command	Icon	Keyboard shortcut
Toggle breakpoint on the line at caret		Ctrl+F8
View all breakpoints		Ctrl+Shift+F8
Resume debugging session		F9
Rerun debugging session	 or 	Ctrl+F5
Stop		Ctrl+F2
Step over		F8
Step into		F7
Run to cursor		Alt+F9

更多控件描述参见 [Debug tool window](#)。

最全 Pycharm 教程（11）——Pycharm 调试器之断点篇

1、准备工作

- (1) Pycharm 版本为 3.0 或者更高
- (2) 至少安装了一个 Python 解释器
- (3) 已经创建了一个 Python 工程
- (4) 使用 [Debugger tutorial](#) 正在调试处理一个实例

2、什么是断点

Pycharm 提供了多种不同类型的断点 [types of breakpoints](#)，并都有特定的图标 [icon](#)。本篇教程中我们将详细介绍行断点，即标记了一行待挂起的代码。

3、断点属性

有两种方法来浏览和改变代码属性：

- (1) 首先，有一个单独的置顶窗口来显示断点属性。
- (2) 有一个断点对话框 [Breakpoints dialog](#) 来显示当前应用中所存在的所有断点及其属性。

4、设置断点

选中对应代码行，然后执行下面步骤（二选一）：

- (1) 单击左边空白槽
- (2) 按下 `Ctrl+F8` 快捷键

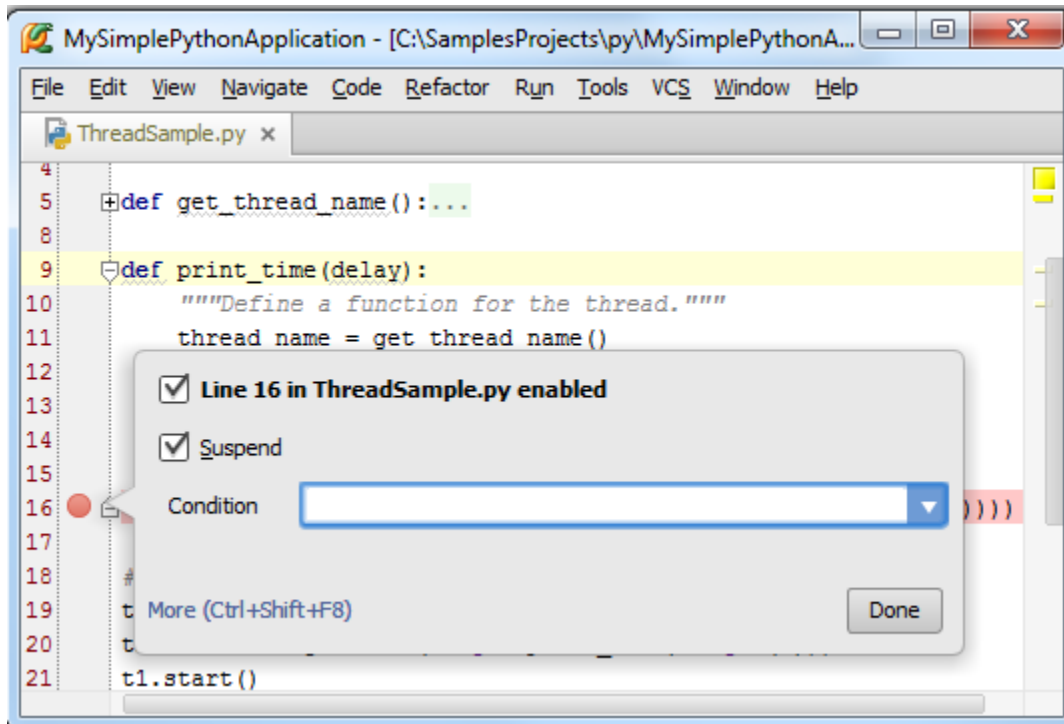
这是一个触发开关式的操作，即可以不断的重复操作以移除/添加断点。

作为练习，在第 16 行设置断点：

```
print("%s: %s" % (thread_name, time.ctime(time.time())))
```

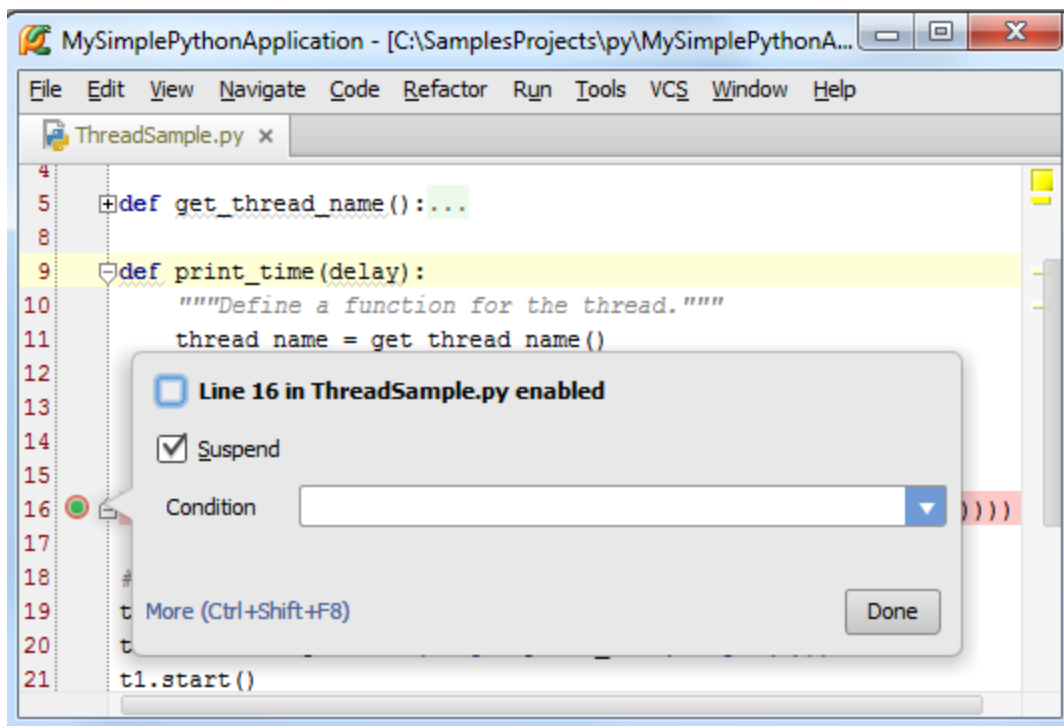
5、浏览并改变断点属性

为了浏览单独一个断点的属性，只需右击它即可：



6、启用和关闭断点

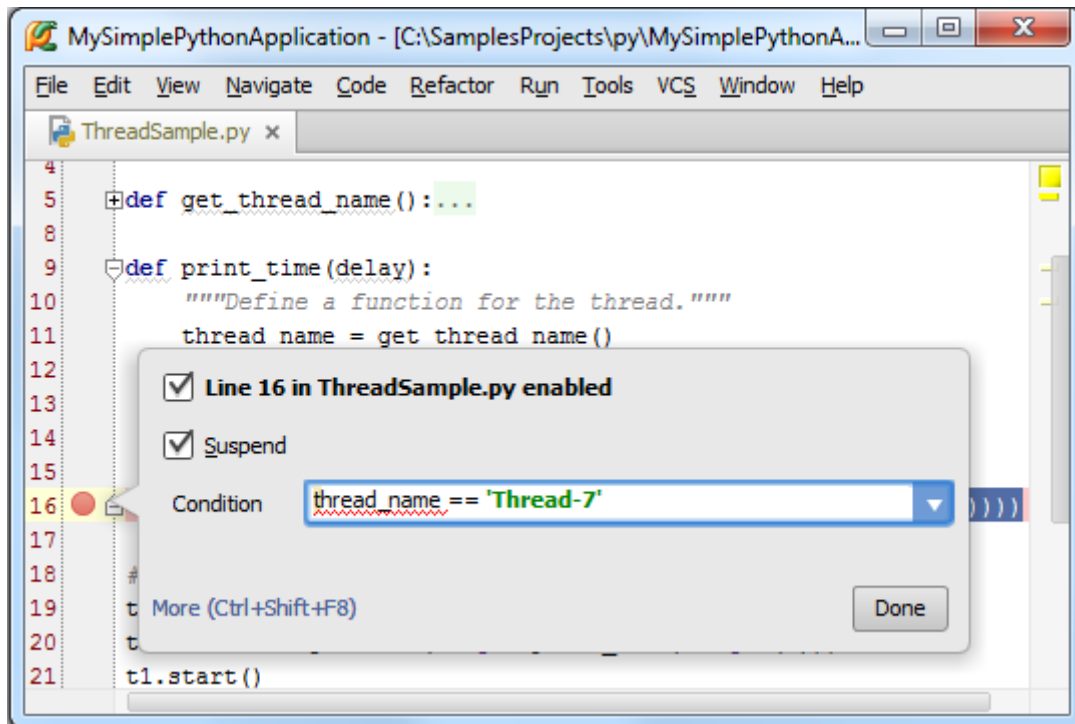
首先，清空复选框中的 **Line 16 in ThreadSample.py enabled** 选项，此时断点会失效，同时图标中心会变为绿色：




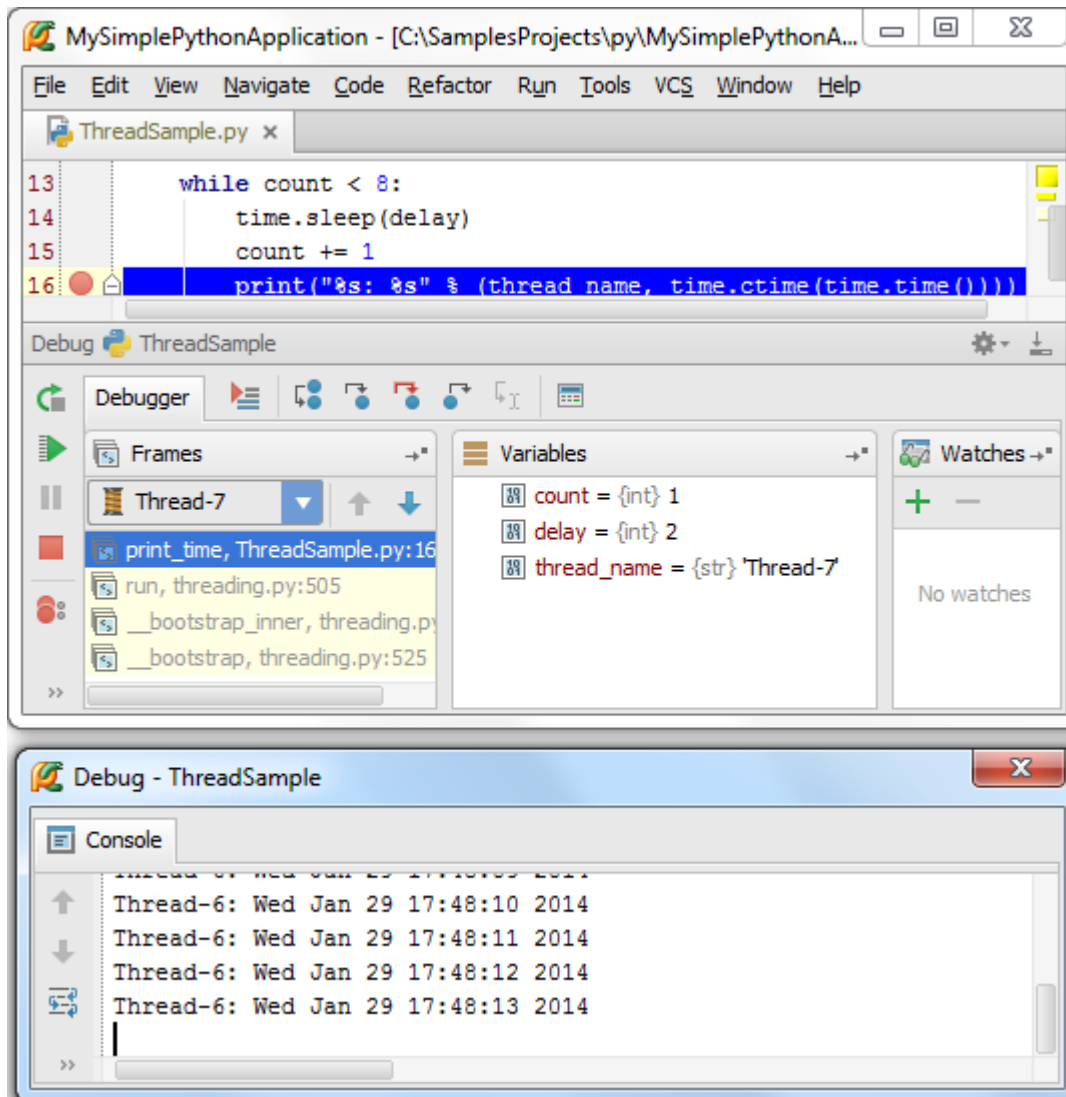
OK，再次勾选复选框，图标恢复原状，断点生效。

7、条件性代码悬挂

假设你希望 Pycharm 在当前代码处只挂起某一个线程，而其他线程能够正常通过断点，操作非常简单，为断点添加一个条件即可：



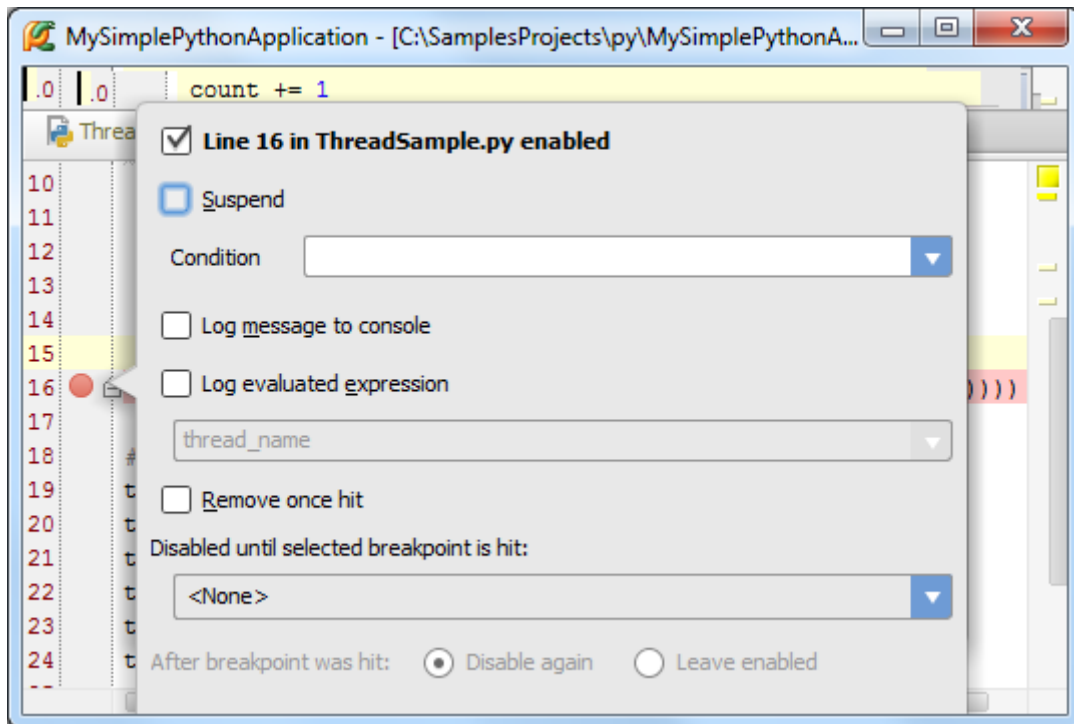
接下来我们开始脚本调试（单击工具栏中的  按钮），你会看到实际上 Pycharm 只在当前断点处挂起了 Thread-7，而 Thread-6 正常通过了该断点：



8、如何忽略某些断点？

通常情况下，你只需要在某一个断点处停下来，而希望在其他断点能够正常通过。换句话说就是我们并不希望 Pycharm 在断点处停止，但我们仍需要保留对应断点来作为标记，如何实现？

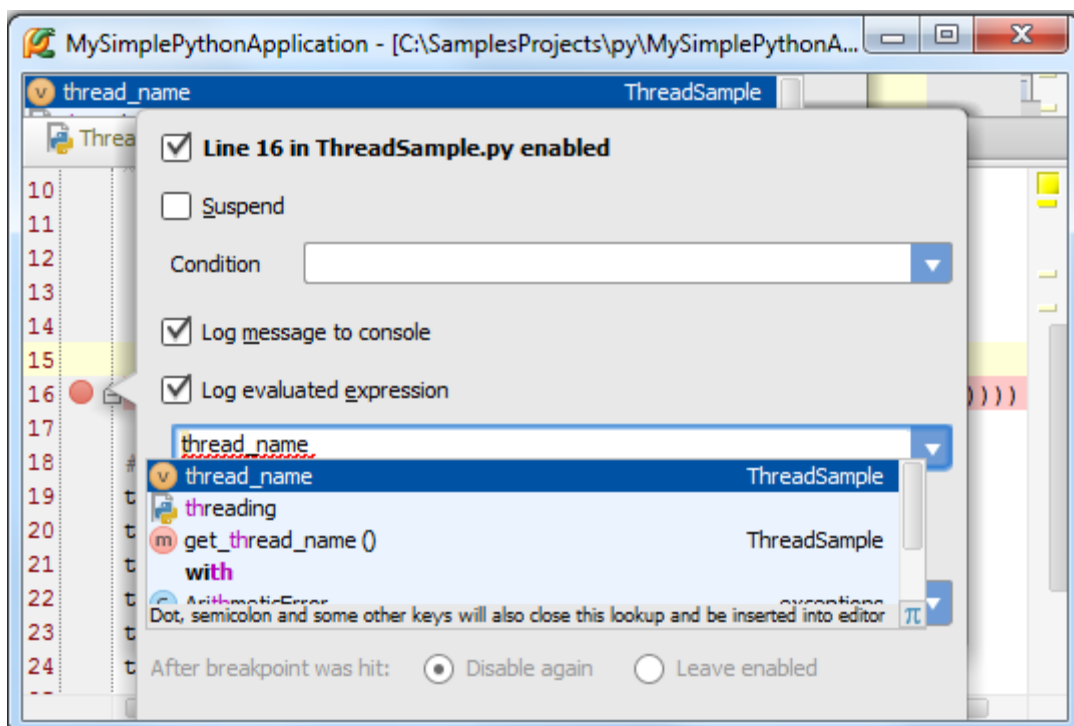
在对话框中取消 **Suspend** 复选框的勾选，此时断点属性对话框中会显示更详细的信息：



勾选一下两个选项：

(1) **Log messages to console**：选中这个选项后，Pycharm 会在命中该断点后在控制台上显示相关重要信息。

(2) **Log evaluated expressions**：选中这个选项之后，Pycharm 会计算下方输入框中给出的特定表达式并显示在控制台上。

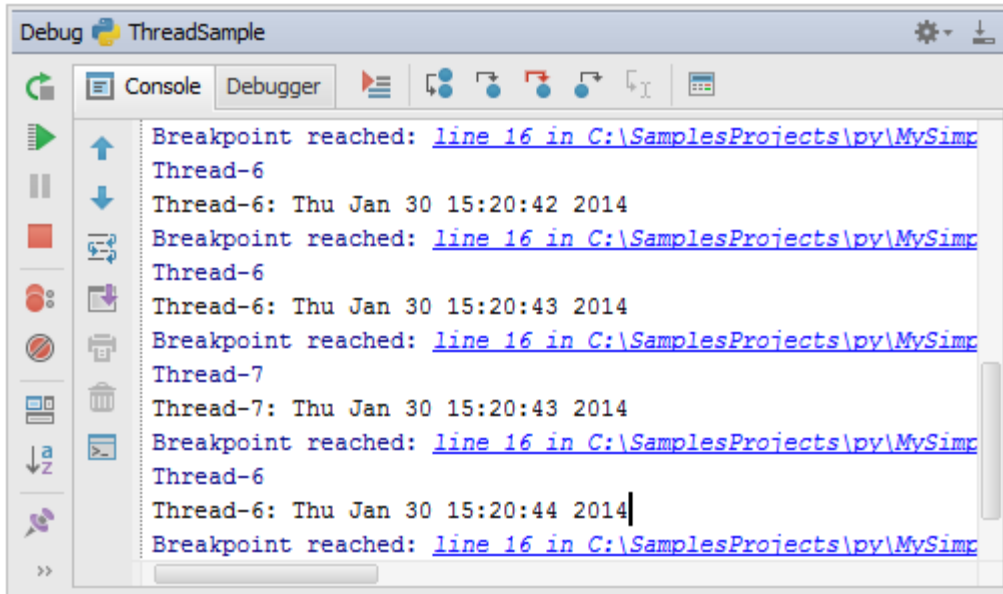


顺便说一下，在下发输入框中输入表达式时 Pycharm 同样提供了拼写提示功能（Ctrl+空格）。OK，重新运行调试，Pycharm 会在调试控制台中显示一下信息：

应用的基本信息。

击中断点时的注册信息

表达式的计算结果



9、浏览所有断点

以上你使用到的所有断点相关的对话框都能帮助你获取断点的大部分重要信息。不过我们通常使用 [Breakpoints dialog](#) 对话框来完成大部分断点处理工作，它会显示当前项目中的所有断点的属性信息，并方便我们对其进行更改。

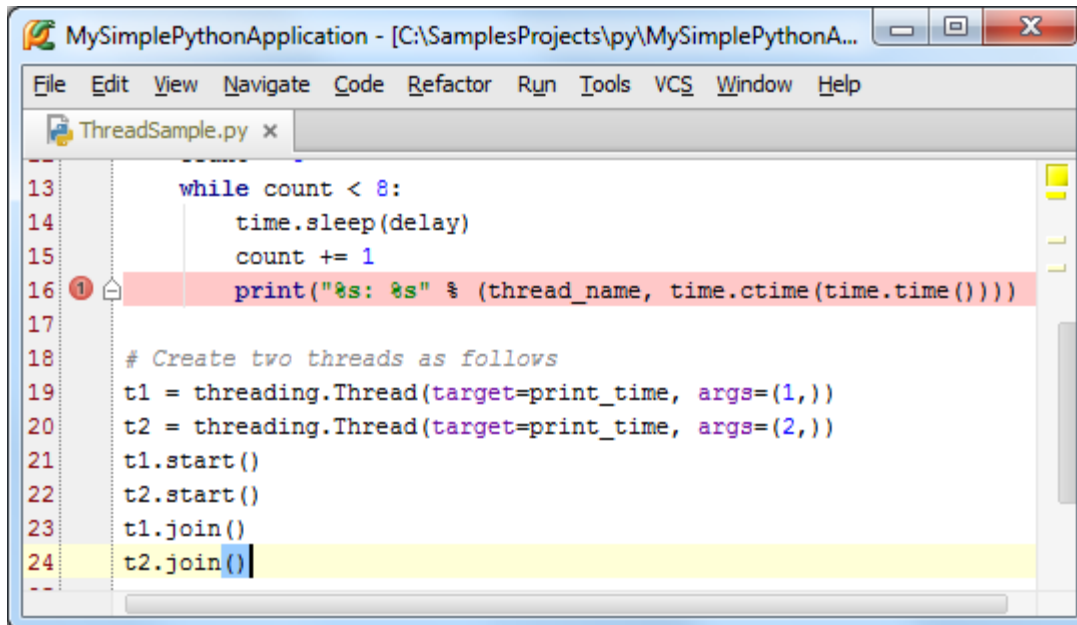
我们可以通过以下几种方式来打开这个对话框：

- (1) Ctrl+Shift+F8 快捷键
- (2) 在主菜单中选中 Run → View Breakpoints
- (3) 在断点属性快捷对话框中单击 [More \(Ctrl+Shift+F8\)](#)
- (4) 在调试窗口中单击  按钮。

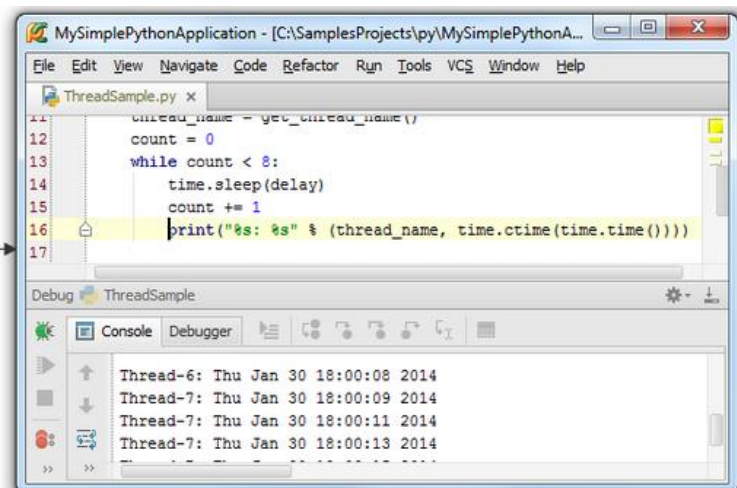
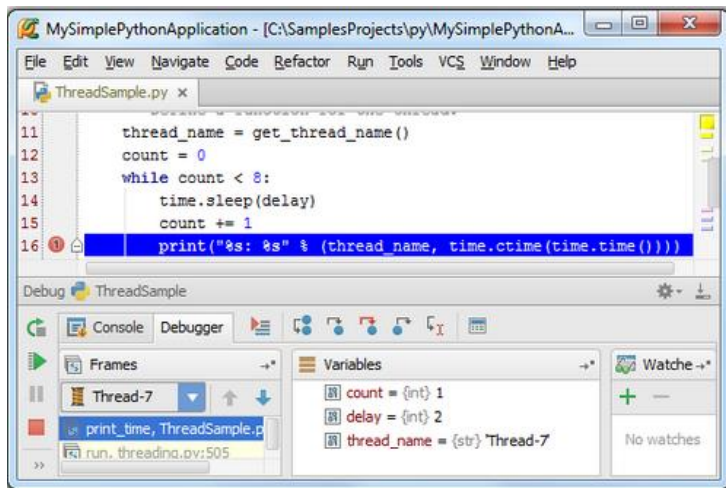
10、创建一个临时断点

假设你希望这样一个断点，即Pycharm在命中它之后就将命中的断点删除。Pycharm 提供了一个特殊的复选项 **Remove once hit**，默认情况下是不会启动这个机制的。

加入 **Suspend** 复选框已经选中，接下来再选中 **Remove once hit** 复选框，观察断点的图标的变化：



重新调试，发现 Pycharm 只会在断点处暂停一次，之后便会将其移除，以保证之后的程序执行不会受到阻碍：



最全 Pycharm 教程（12）——Pycharm 调试器之 Java 脚本调试

1、总览

对于 Web 开发而言，调试 Java 脚本是十分重要的。为了显示 Pycharm 对于 Java 脚本的强大调试能力，我们这里创建一个非常简单的脚本，用以展示一些简单的浏览器页面，然后在服务器上对其进行调试。

为了能够在外部服务器上进行调试，需要在上面运行程序文件，同时在你的电脑上拷贝一份。不过没关系，无论 Web 服务器是运行在物理远端还是运行在你的电脑上，其中的应用程序都可以看成是一个远程服务程序。

当一个使用 JavaScript 生成的远程文件打开时，调试器会告诉 Pycharm 当前处理文件的名称以及对应的行号。Pycharm 会打开本地副本并定位到相应的行。Pycharm 的这种行为使得服务端和客户端的文件进程保持同步。这种通信机制成为映射（mapping），在 debug 配置文件中保留了相关的设置信息。

2、准备工作

(1) Pycharm 版本为 3.0 或者更高。

(2) 使用谷歌浏览器（这篇教程是基于谷歌 Chrome 的）

(3) 你已经安装了 JetBrains IDE Support 外部插件。如果你是第一次加载调试器，Pycharm 会给出你关于安装 JetBrains IDE Support 外部插件的重要性。

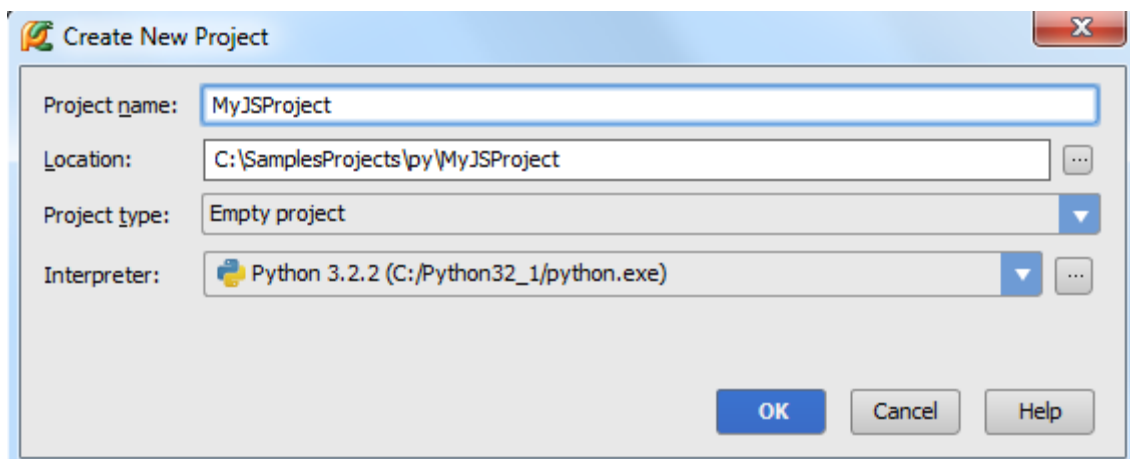
举个例子，对于浏览器，当在地址栏的右侧显示  图标，并且非透明，说明插件已经安装并且成功激活。

可以在 <https://chrome.google.com/webstore/detail/jetbrains-ide-support/hmhgeddbohgjknpmjagkdomcpobmlji> 下载和安装 JetBrains IDE Support 外部扩展。

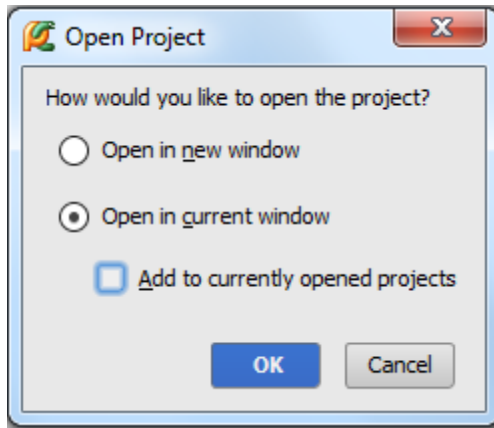
建议使用 XAMPP 作为服务端程序（文章是针对 XAMPP 的）。

3、创建一个简单的工程

在主菜单中选择 File→New Project，选择一个空工程，命名为 MyJSProject：

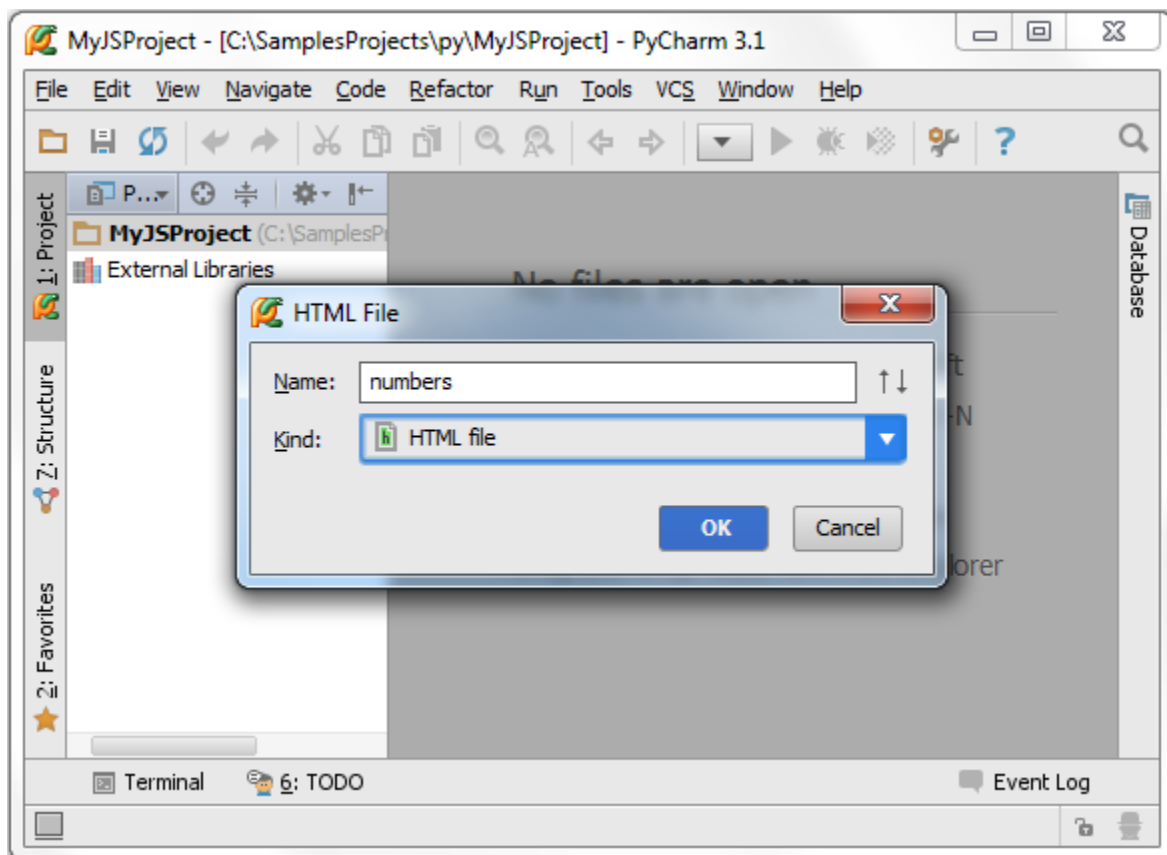


在一个独立的窗口中打开这个工程：



4、实例准备

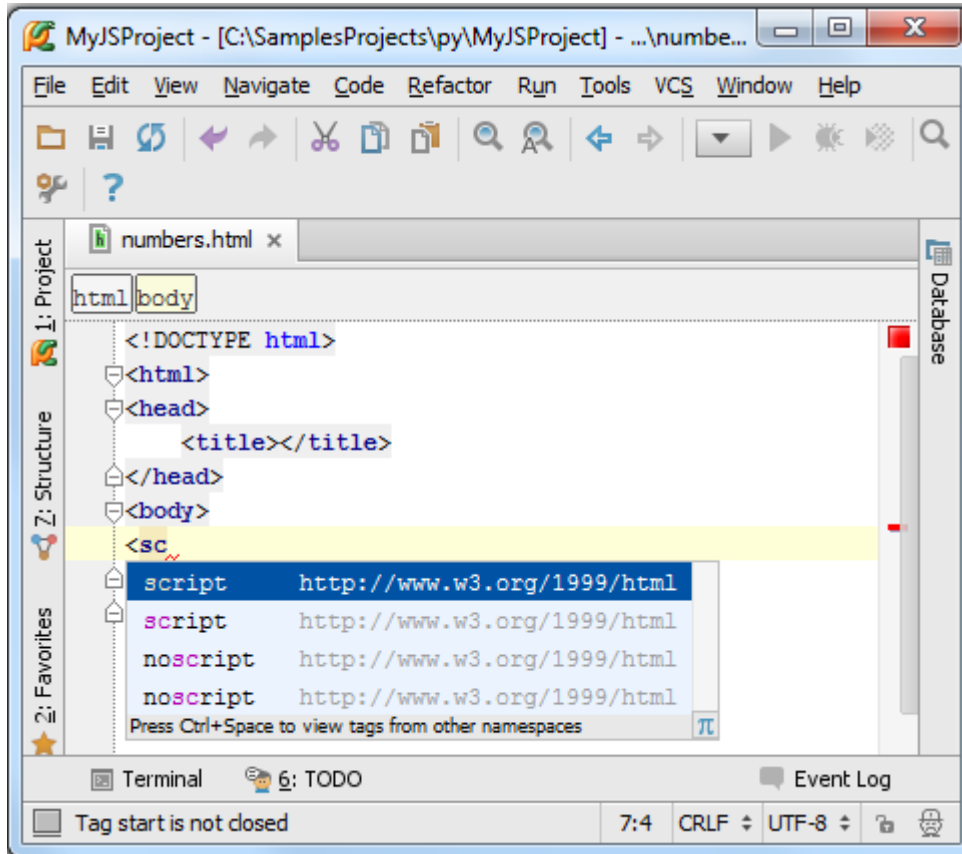
首先，创建一个 HTML 页面。在工程管理窗口中按下 Alt+Insert，在弹出的快捷菜单中选择 HTML 文件类型，命名为 numbers:



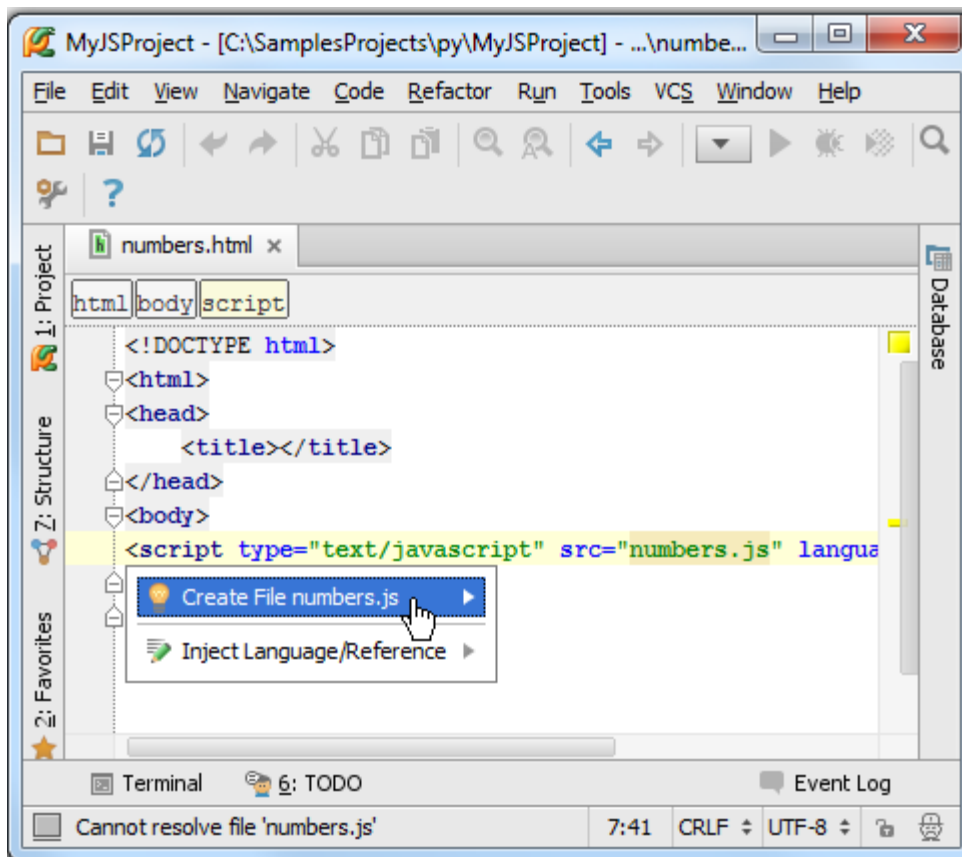
Pycharm 会在生成的 HTML 文件中添加一些原始内容。接下来，向其中嵌入一个 JavaScript 文件，在<body>标签下输入一下代码：

```
<script type="text/javascript" src="numbers.js" language="JavaScript"></script>
```

在输入代码时注意体会 Pycharm 的拼写提示功能：



完成后，注意文件名 **numbers.js** 以高亮的形式给出。这说明当前的 JavaScript 文件不存在。在将光标定位在名称“numbers”上，按下 Alt+Enter（或者单击左侧的小黄色灯泡）；会给出快捷提示——创建一个缺省文件：

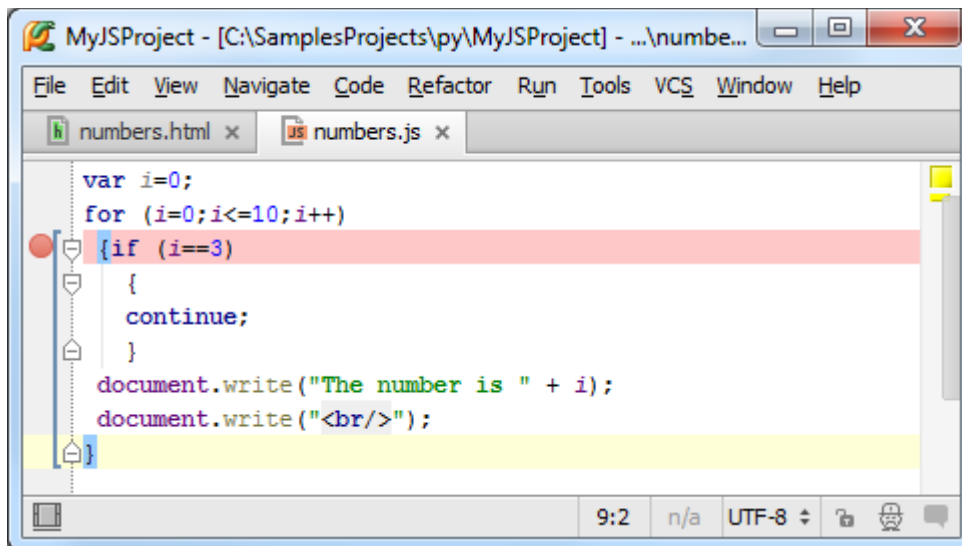


通过这个快捷方法，已将创建好了原始的 JavaScript 文件，接下来输入下面代码：

```
var i=0;
for (i=0;i<=10;i++)
{if (i==3)
{
continue;
}
document.write("The number is " + i);
document.write("<br/>");
}
```

5、设置断点

接下来在 JavaScript 文件中插入断点，非常简单，在右侧单击即可：



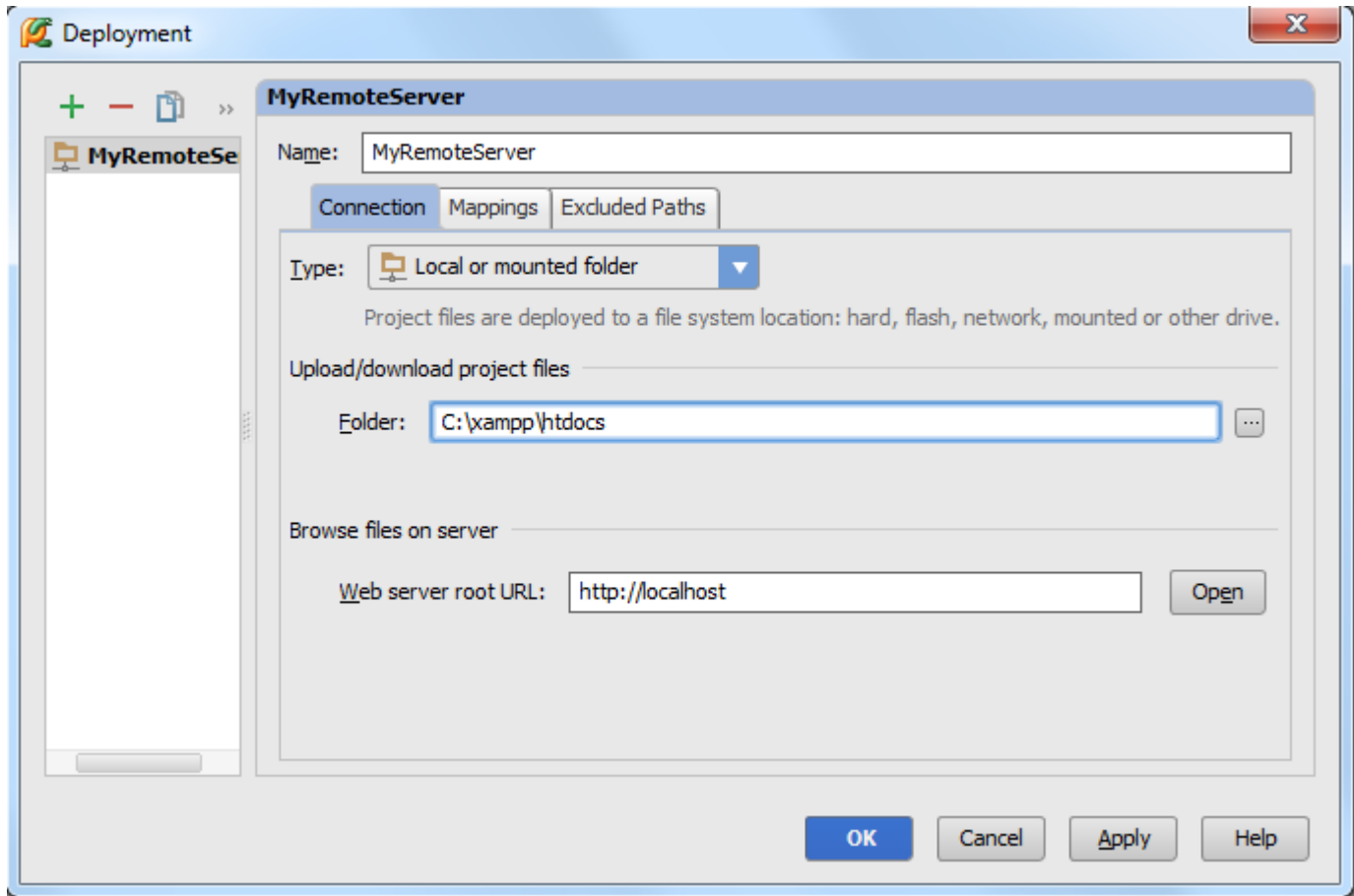
6、配置服务器

7、创建一个服务器

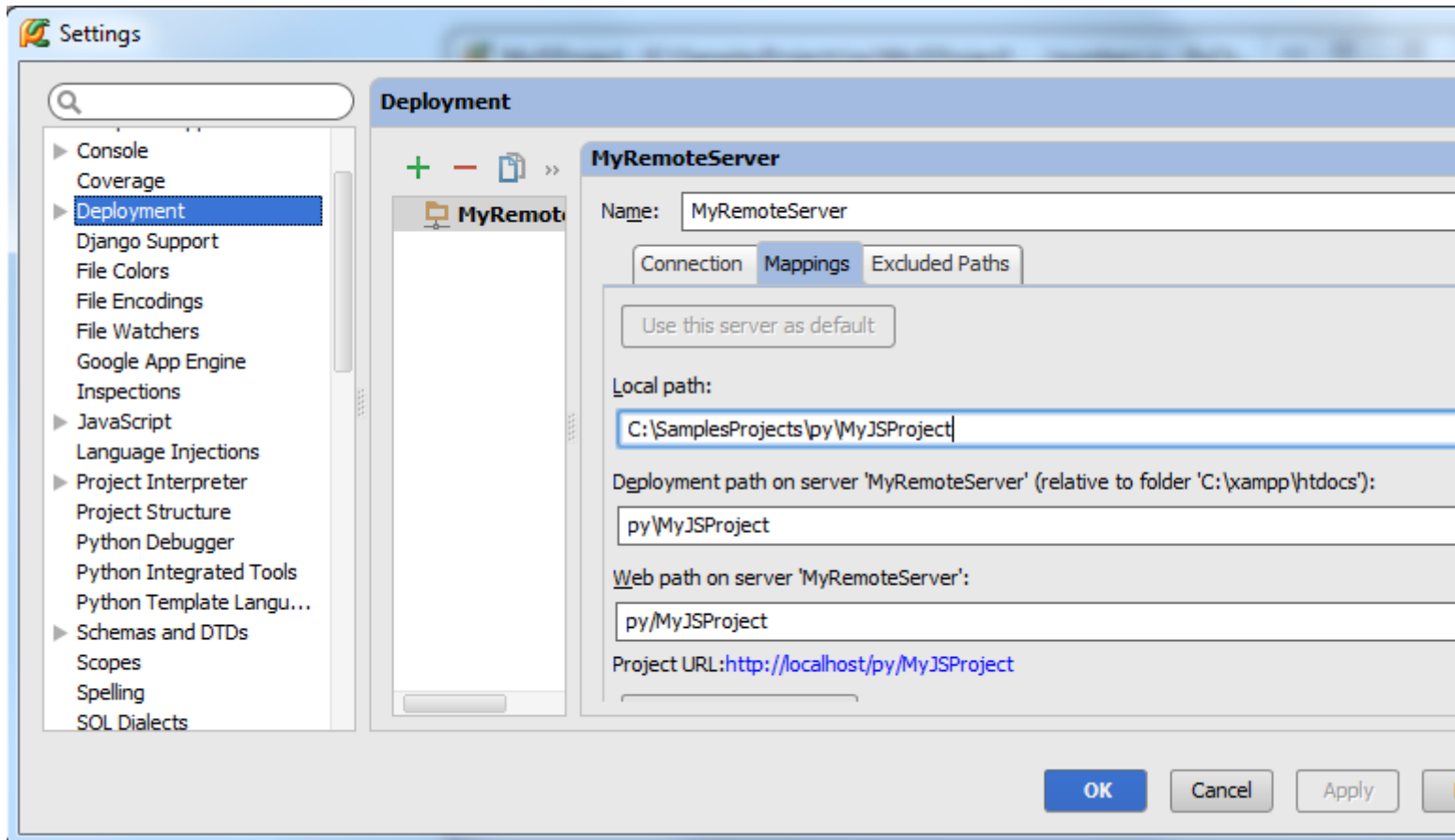
打开 Settings/Preferences 对话框（Ctrl+Alt+S 或者单击主菜单的设置按钮），单击 [Deployment page](#) 页面的绿色的加号，将服务器命名为 MyRemoteServer，指定类型为 local or mounted server。

8、配置映射连接


接下来配置创建的服务器。在 Connection 选项卡中，输入需要加载的本地文件的目录，这里为 C:\xampp\htdocs，也就意味着将从这个目录来上传本地文件：



单击 Mappings 选项卡，在这里定义本地路径，服务器的部署路径（与 Connection 选项卡的设置相同），以及服务器的 Web 路径：

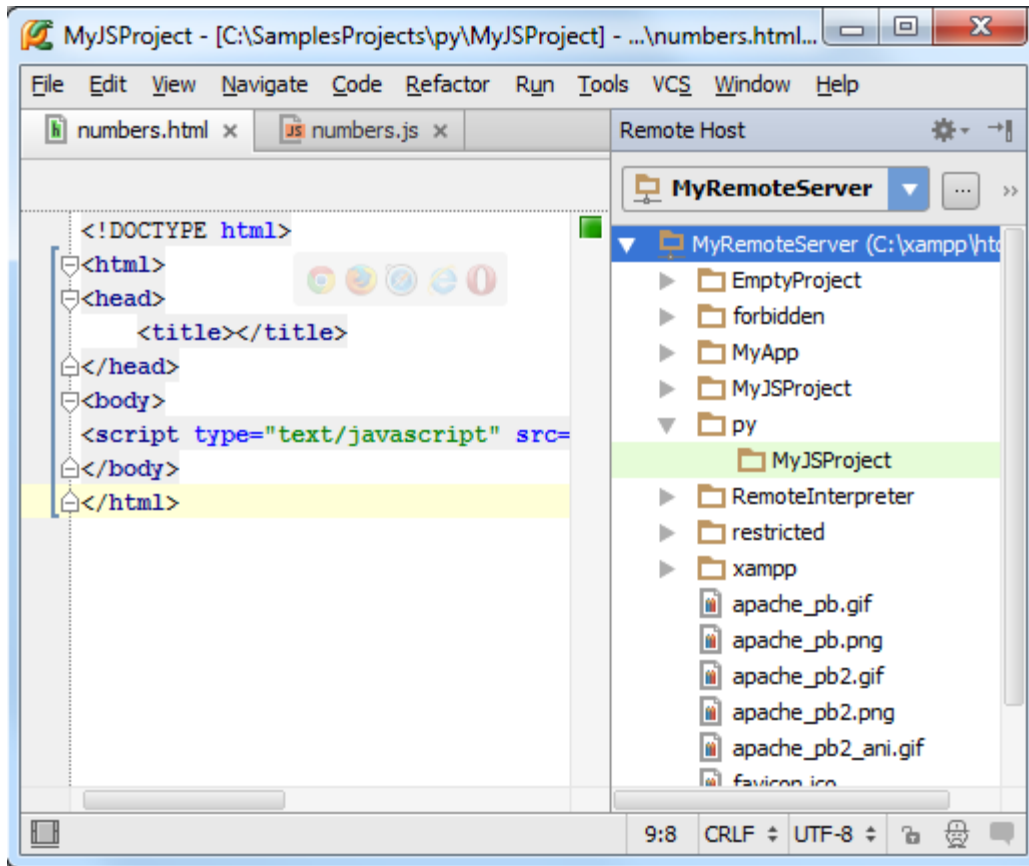


9、定义项目的默认服务器

指定创建的服务器为当前项目的默认服务器，只需在 **Deployment toolbar** 中单击  按钮。

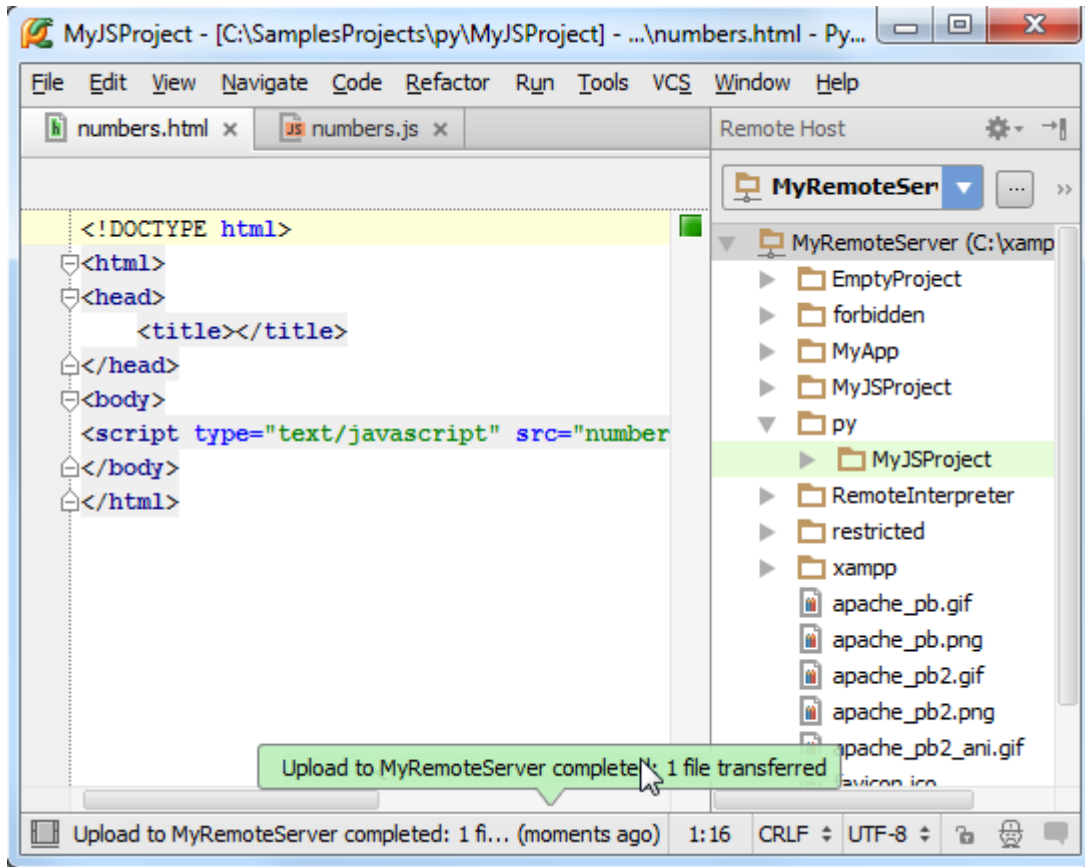
10、预览服务器

接下来需要确认我们的服务程序能够在 Pycharm 的可视化窗口中顺利上传和运行。在主菜单中选择 **Tools** → **Deployment** → **Browse Remote Hosts**。远程 Host 控制窗口会显示当前新开启的服务：



11、向服务器应用中部署文件

在 Pycharm 中这个操作非常简单。在主菜单中选择 **Tools** → **Deployment** → **Upload to MyRemoteServer**，确保新目录 **C:\xampp\htdocs** 已经在服务器上顺利创建。

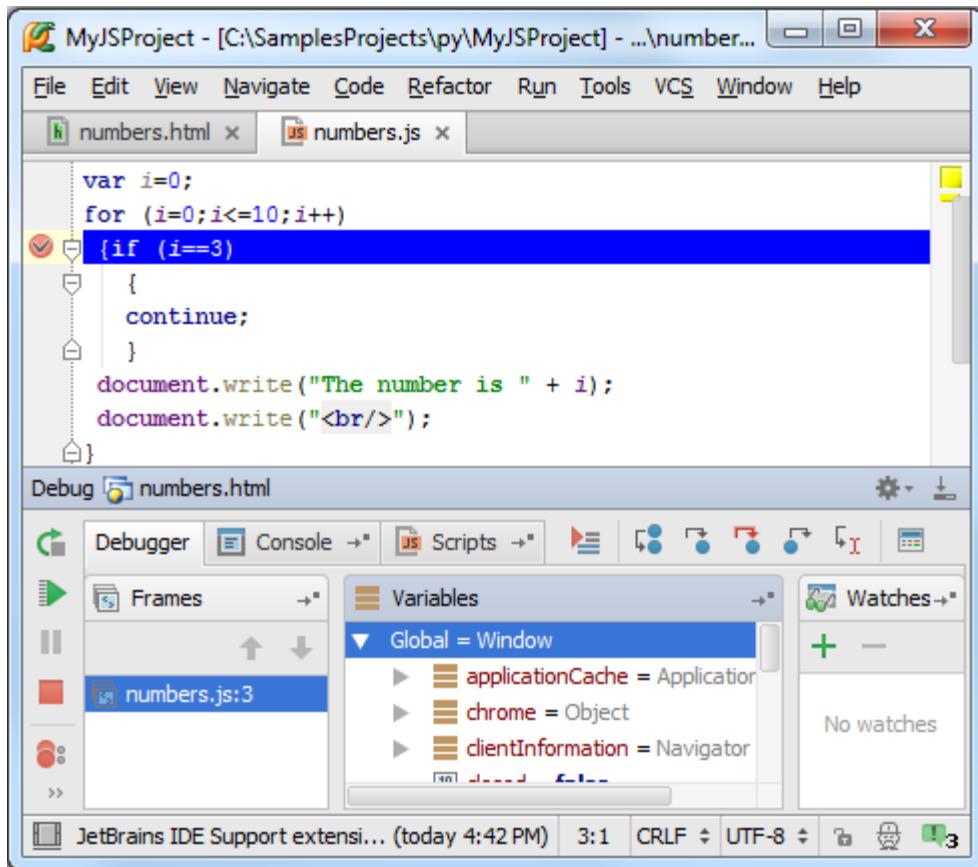


另一种方式可以通过右击文件，在快捷菜单中进行操作。当然二者都需要通过 Upload to MyRemoteServer 命令来完成。

12、调试

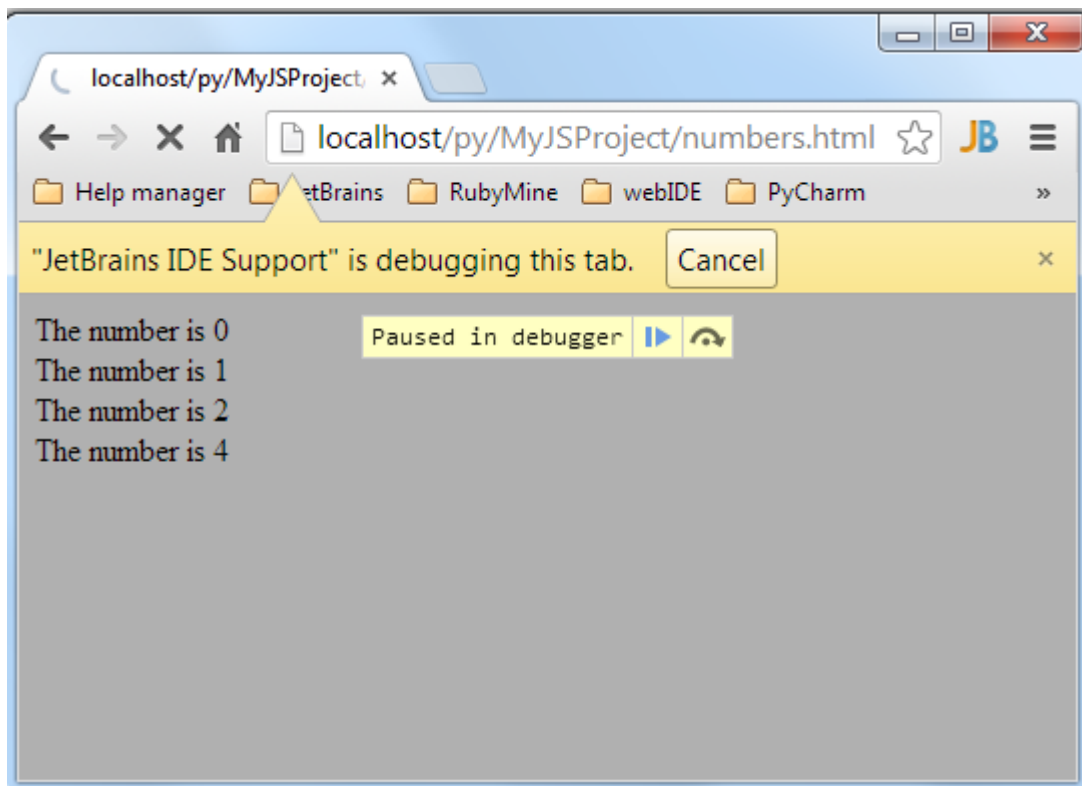
13、开始调试



开始调试后，将会在浏览器中显示你的 HTML 页面，同时调试窗口 [Debug tool window](#) 开启。应有程序会在命中第一个断点的时候停止，并用蓝色标记代码行：



更多有关断点的信息参见 [product documentation](#) 中的 [Breakpoints](#) 部分。

对应用程序进行更为深入的调试，相关的调试信息会显示在调试窗口以及浏览器上：



单击  和  控制代码的调试进程，通过关闭黄色标志或者单击 Cancel 按钮来终止调试。

最全 Pycharm 教程（13）——Pycharm 部署

1、主题

这篇教程将逐步介绍如何通过 Pycharm 将你的代码部署在远程服务器上。

2、准备工作

(1) Pycharm 版本为 3.0 或者更高

(2) 拥有待部署远程服务器的访问权限

强调这篇教程是针对 win7 操作系统，使用了默认的快捷键配置方案，不同系统下的默认快捷键配置方案可能不同。

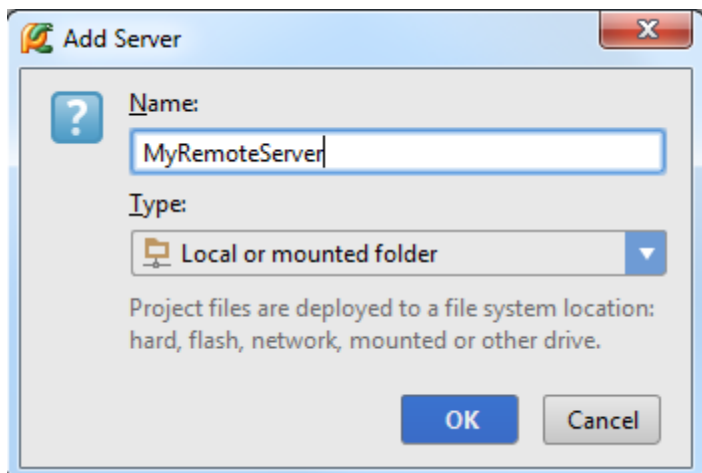
3、准备实例

建议使用 [Getting Started](#) 篇所介绍的实例，那里面已经将主要步骤描述清楚，直接借用即可。

4、配置一个部署服务器

单击主工具栏中的设置按钮来打开 Settings/Preferences 对话框，选择 [Deployment](#) 页面（也可以通过主菜单上的 Tools→Deployment→Configuration 命令来完成）。

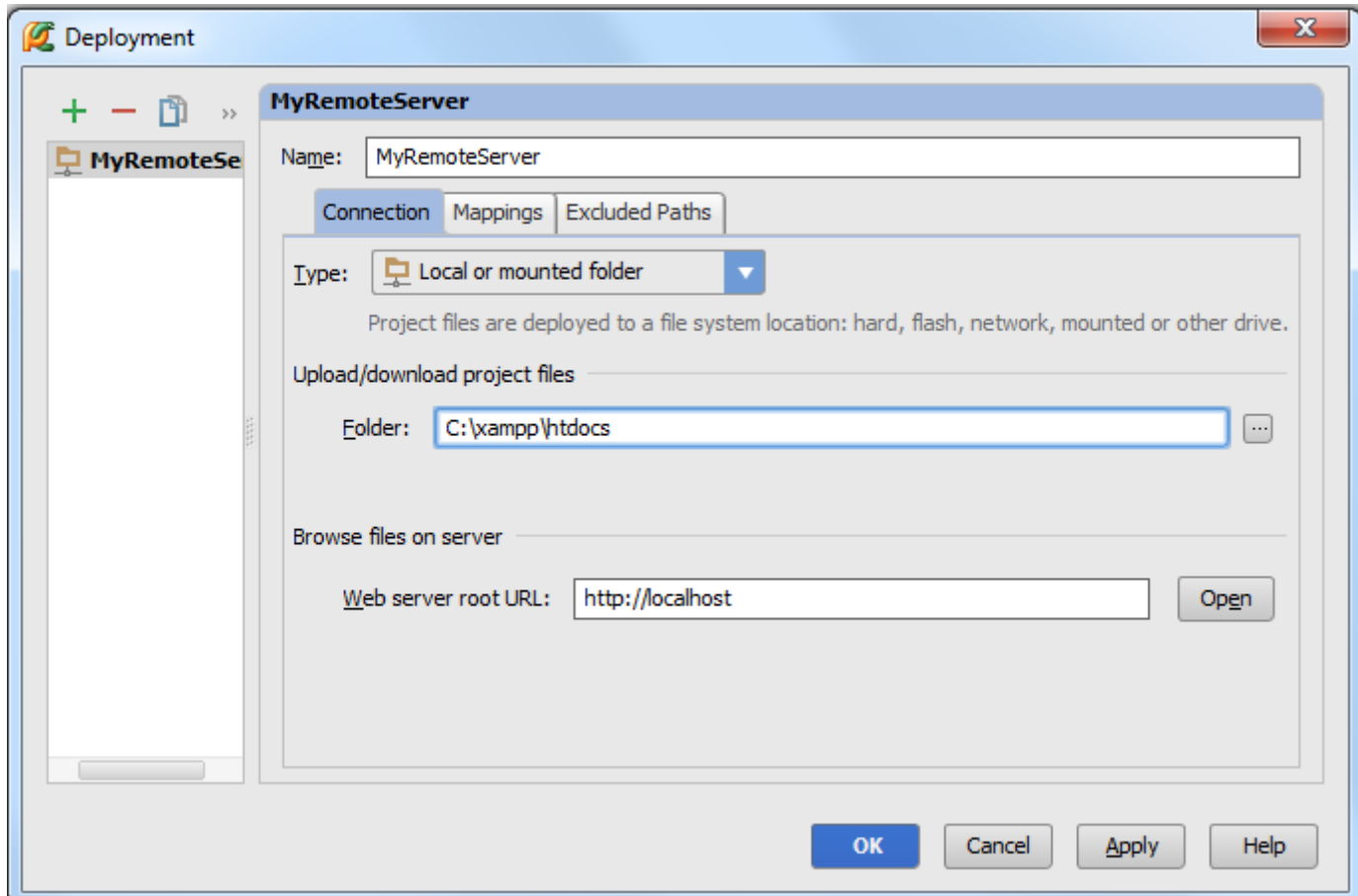
单击绿色加号，在 Add Server 对话框中，输入服务器的名称（MyRemoteServer）并指定其类型（此处选择 Local or mounted folder）：



此时已经添加了一个新的服务器，但其内容为空。它只显示了 Web 服务的根域名（<http://localhost>），你需要在其中上传你的文件。

5、如何定值连接表单

选择带上传文件所在目录，这里为本地目录 C:\xampp\htdocs（既可以手动输入，也可以通过 Shift+Enter 快捷键来打开 [Select Path](#) 对话框进行输入），详见 [product documentation](#)：

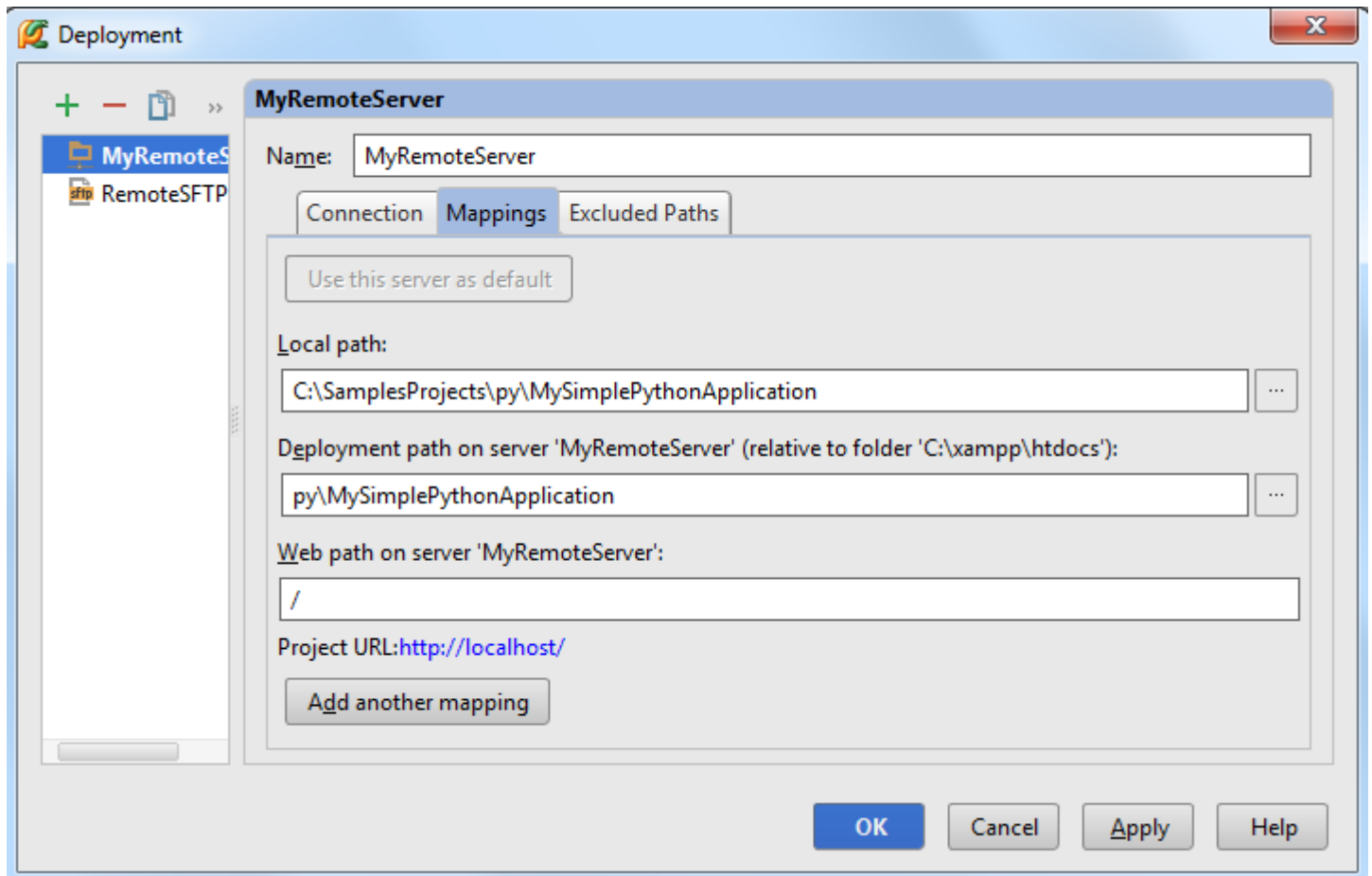


6、如何定制映射表单

接下来，选择 [Mappings tab](#)，其中的 Local path 栏默认包含了工程根目录。当然你可以选择你的工程树中的任意目录，这里我们使用它的缺省值。

在 Deployment path 栏中（缺省值为空），需要定义服务器所在文件夹，将来 Pycharm 会从 Local path 文件夹向其中上传文件。这里选择为 C:\xampp\htdocs。

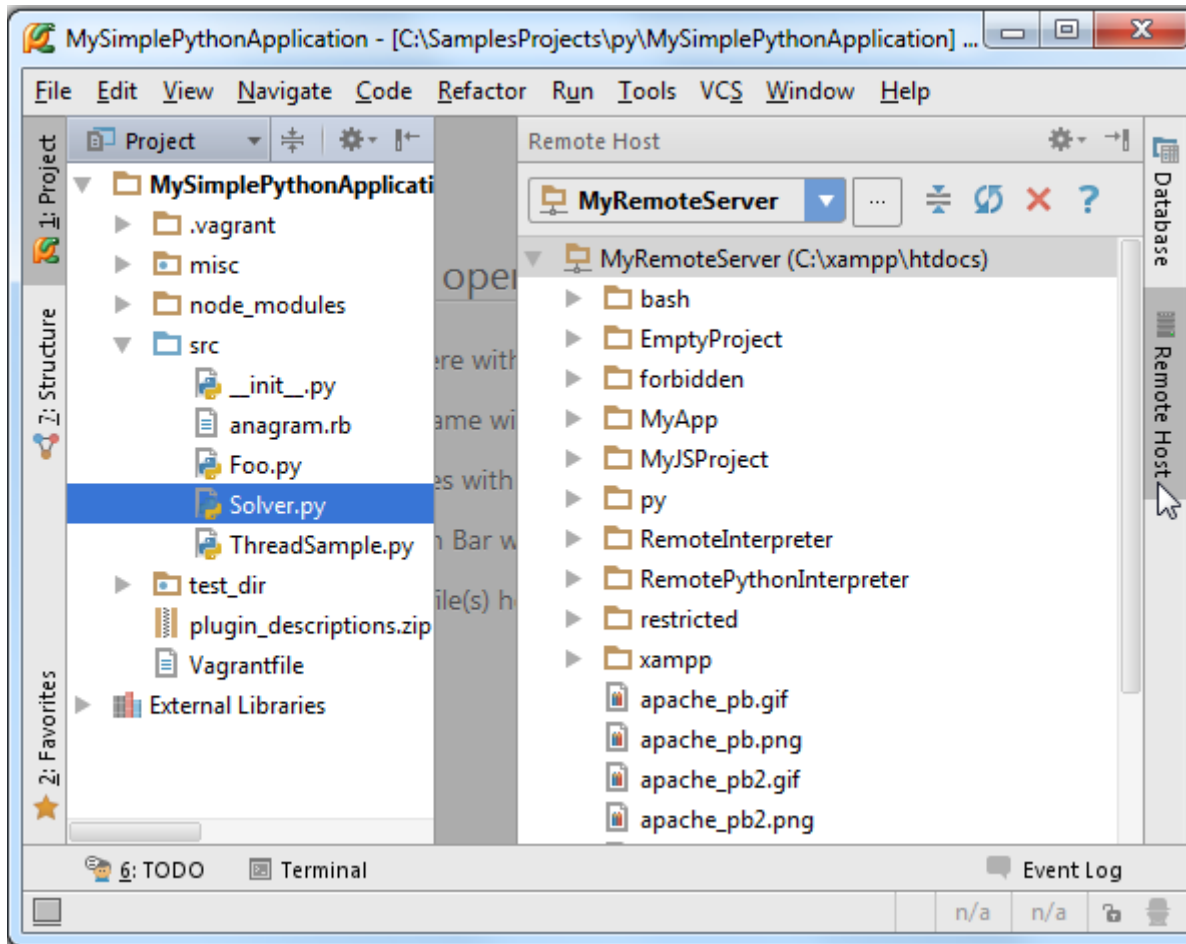
最后在 Web path on the server *MyRemoteServer* 栏中使用其缺省值：



OK，保存设置，服务可用。

7、浏览远端主机

你需要简单的确认一下你的服务器是否已上线并正常运行。打开 [Remote Hosts tool window](#)（位于 PyCharm 窗口的右边缘）：



当然，也可以通过 `Tools`→`Deployment`→`Browse Remote Hosts` 菜单命令来打开这个窗口。

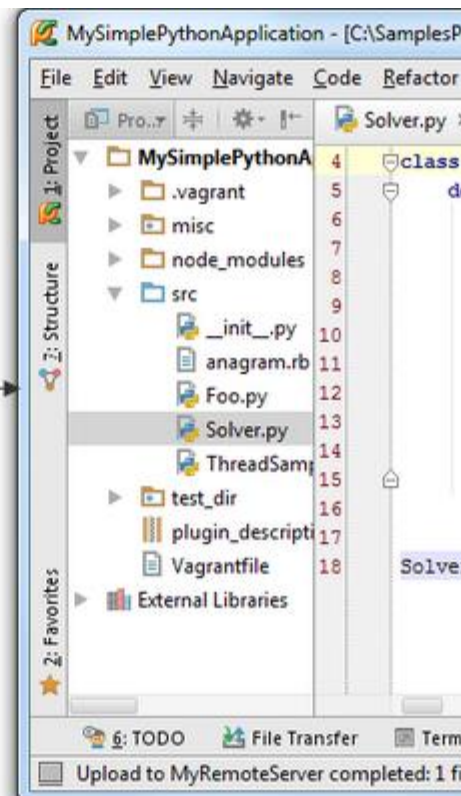
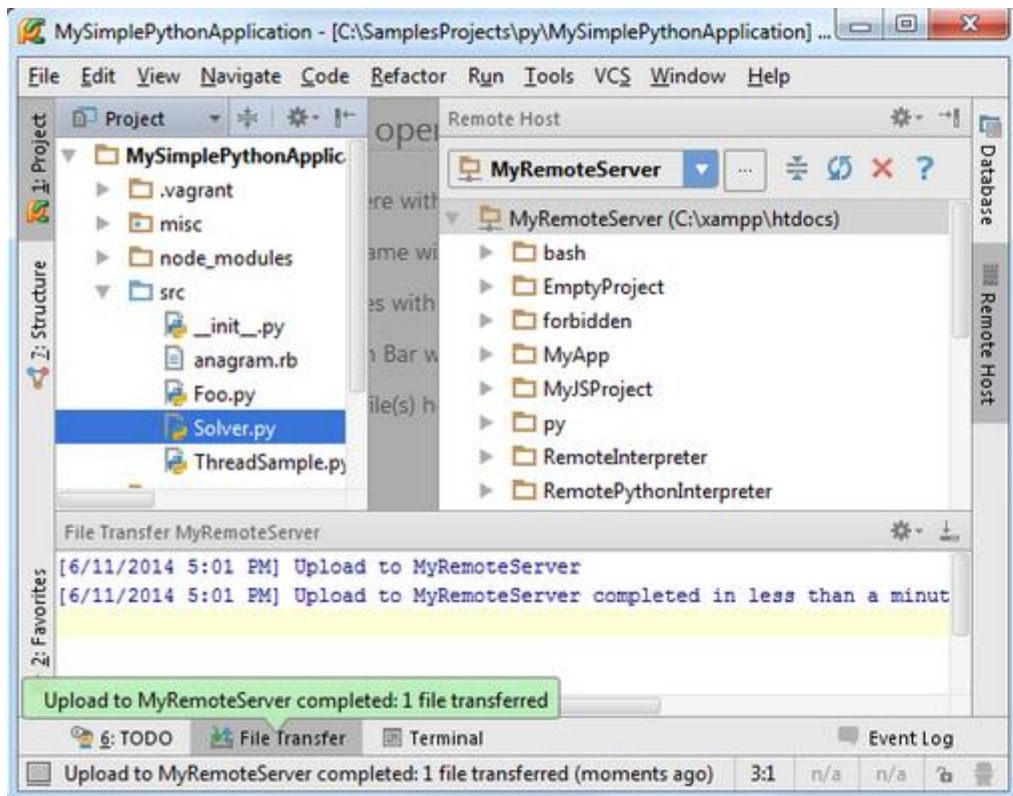
8、部署工具

接下来开始执行部署操作。

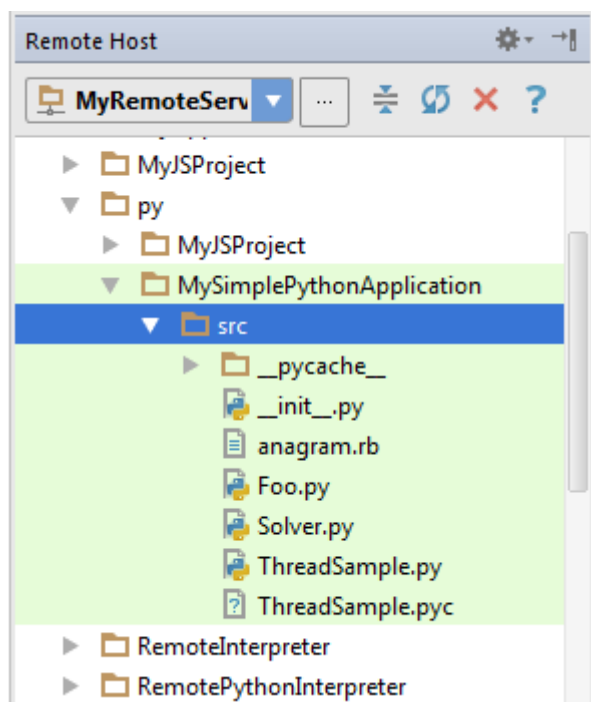
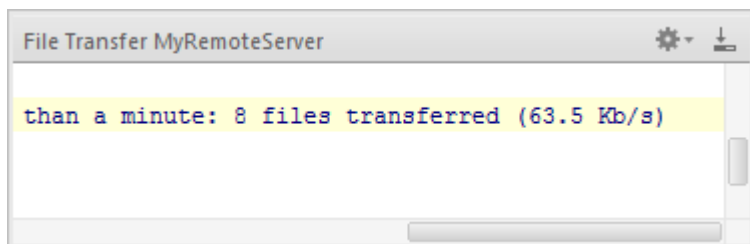
9、上传

首先，向远端服务器上传文件，做法如下：

在 `Project tool window` 窗口中，右击待上传的文件，这里为 `Solver.py`。在弹出的快捷菜单中，选择 `Deployment`→`Upload to MyRemoteServer`，观察上载结果：



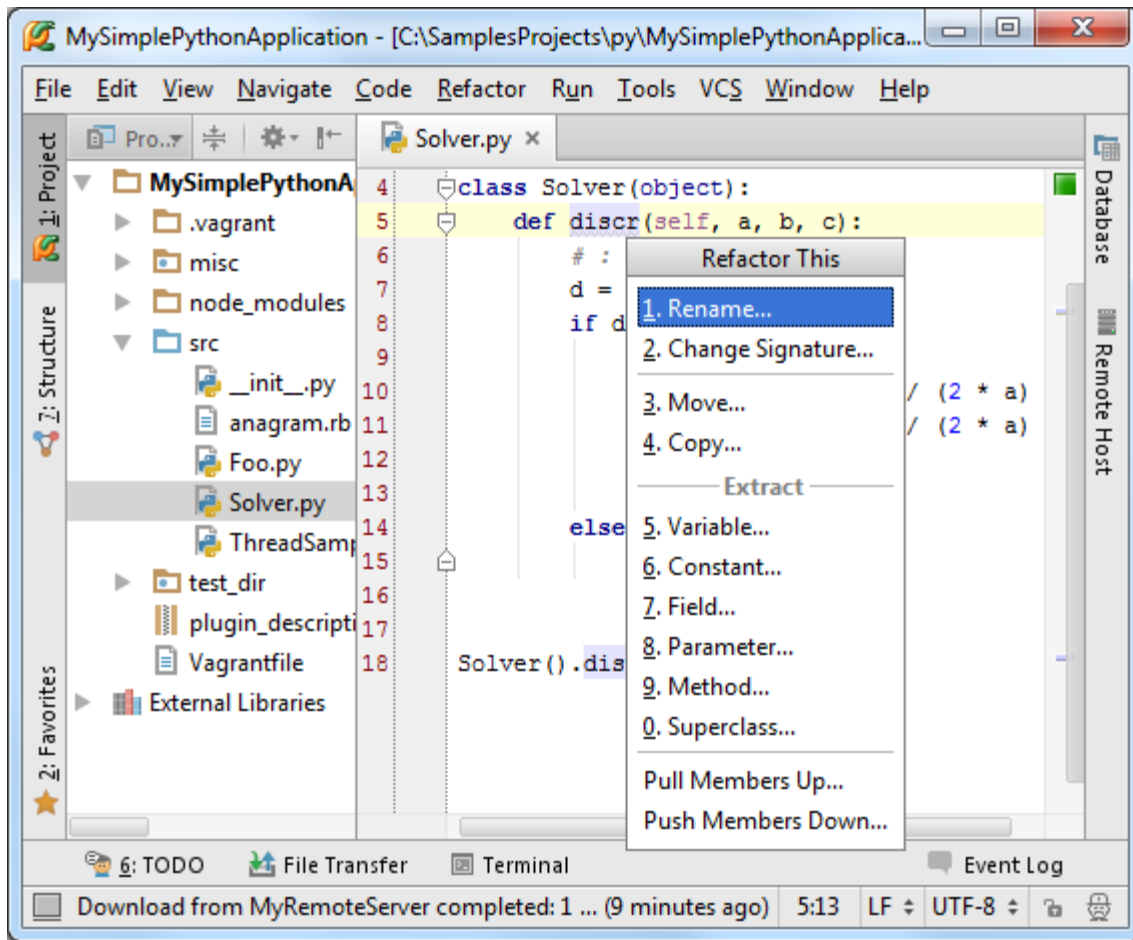
当然我们也可以上传工程目录下的所有文件。例如，右击 Solver.py 文件所在父目录 (src)，在快捷菜单中选择 Upload to MyRemoteServer，就能够将目录下的所有文件上传到服务端：



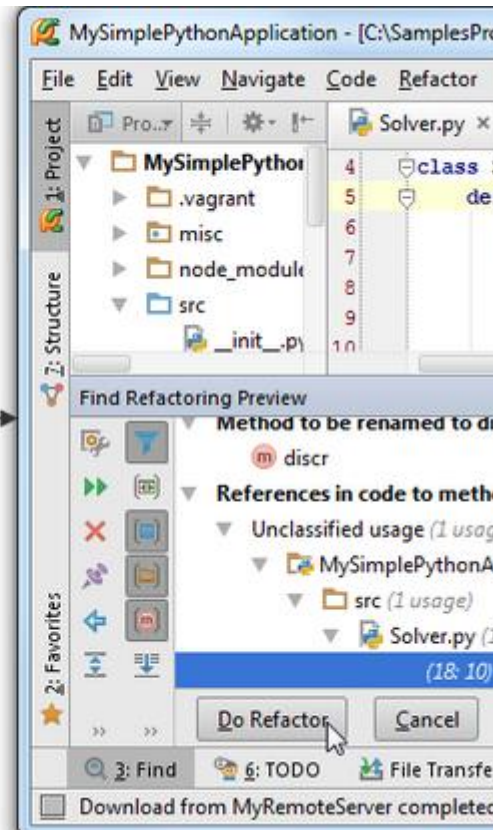
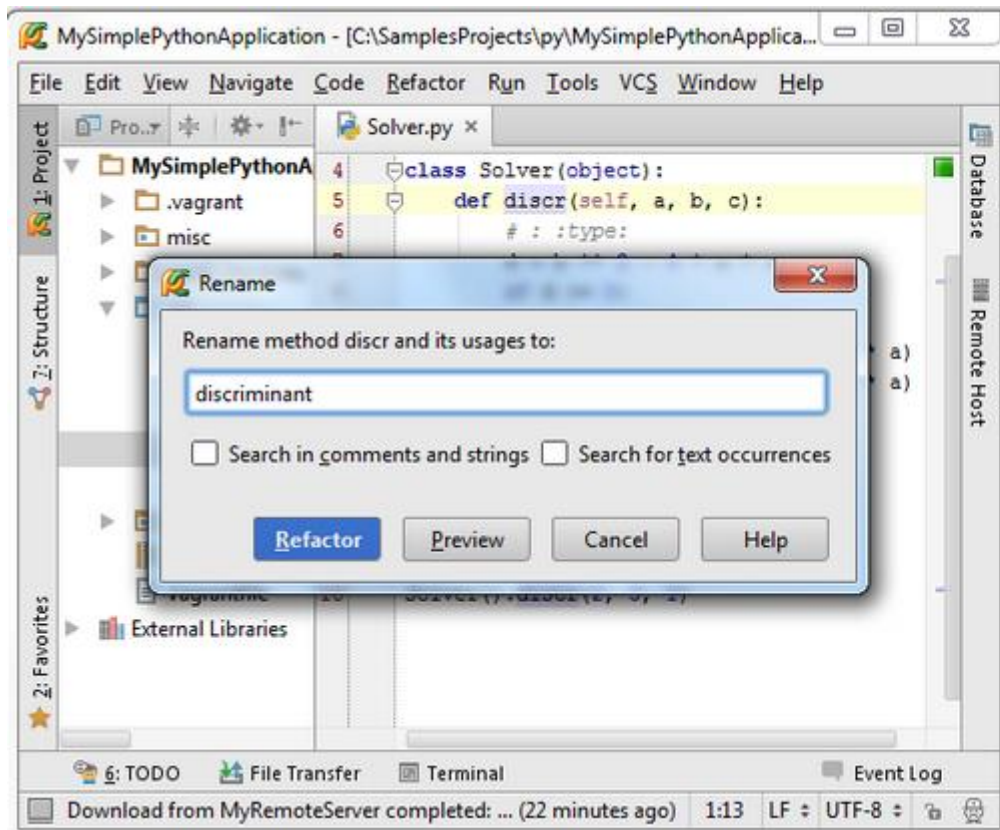
10、远程、本地版本比较

此时再远程服务端和本地服务端都有一份 Solver.py 文件，它们是完全相同的。此时切换到本地视图。

做法很简单，将光标定位在函数声明语句上，按下 Ctrl+Alt+Shift+T（或者主菜单上的 Refactor→Refactor This 命令）：

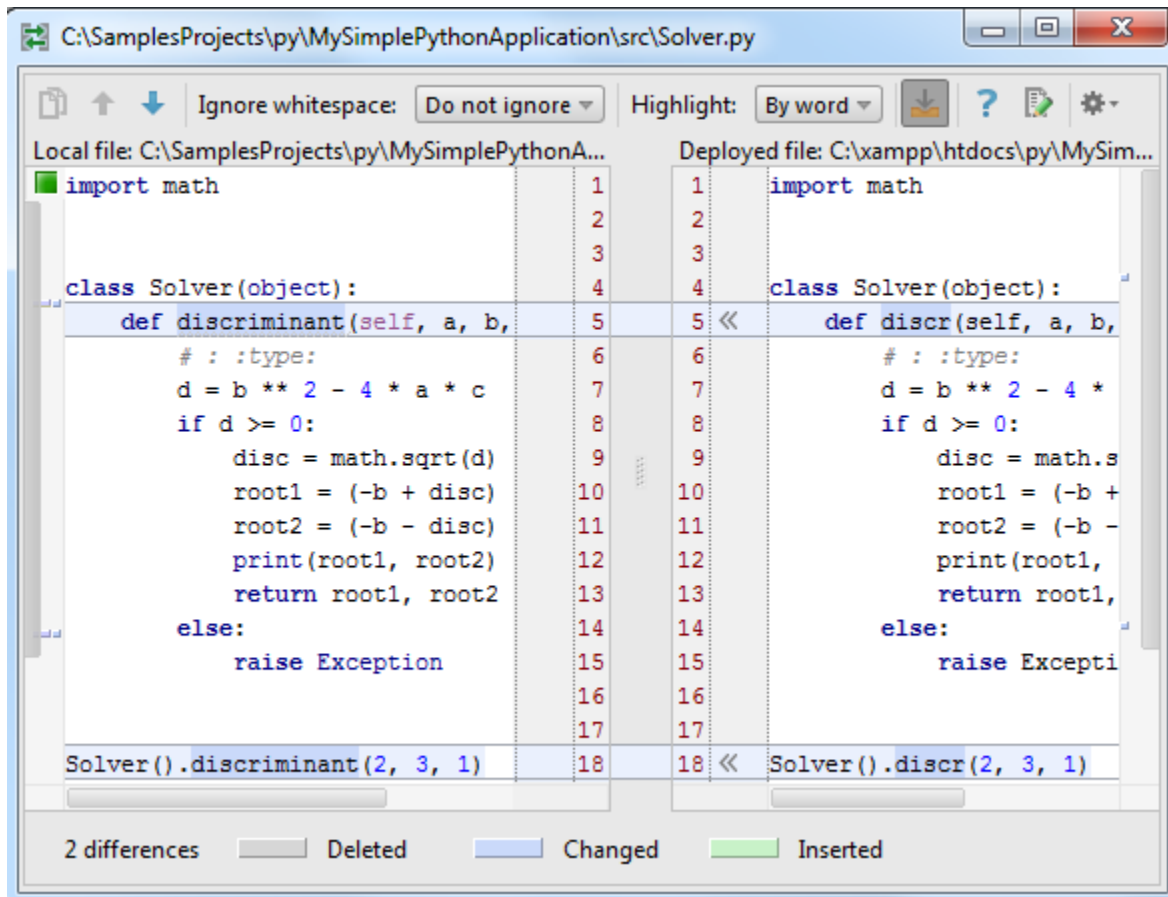


如你所见，快捷菜单中列出了当前文本下可用的所有代码重构，我们这里选择 Rename refactoring，并且 `rename a method`：



单击 Do Refactor，观察函数名称以及用法的改变。

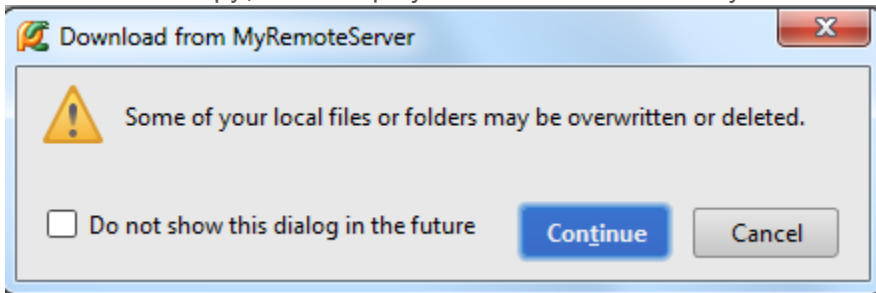
此时我们已经对一个本地版本进行了更改，接下来我们需要做的就是让 Pycharm 接收到这些更改。再次进入 [Project tool window](#)，右击 Solver.py 文件，在快捷菜单中选择 Deployment→Compare with Deployed Version on MyRemoteServer。Pycharm 会打开 [differences viewer for files](#) 对话框，在这里你可以通过 *shvron* 按钮来核对所做的改变：



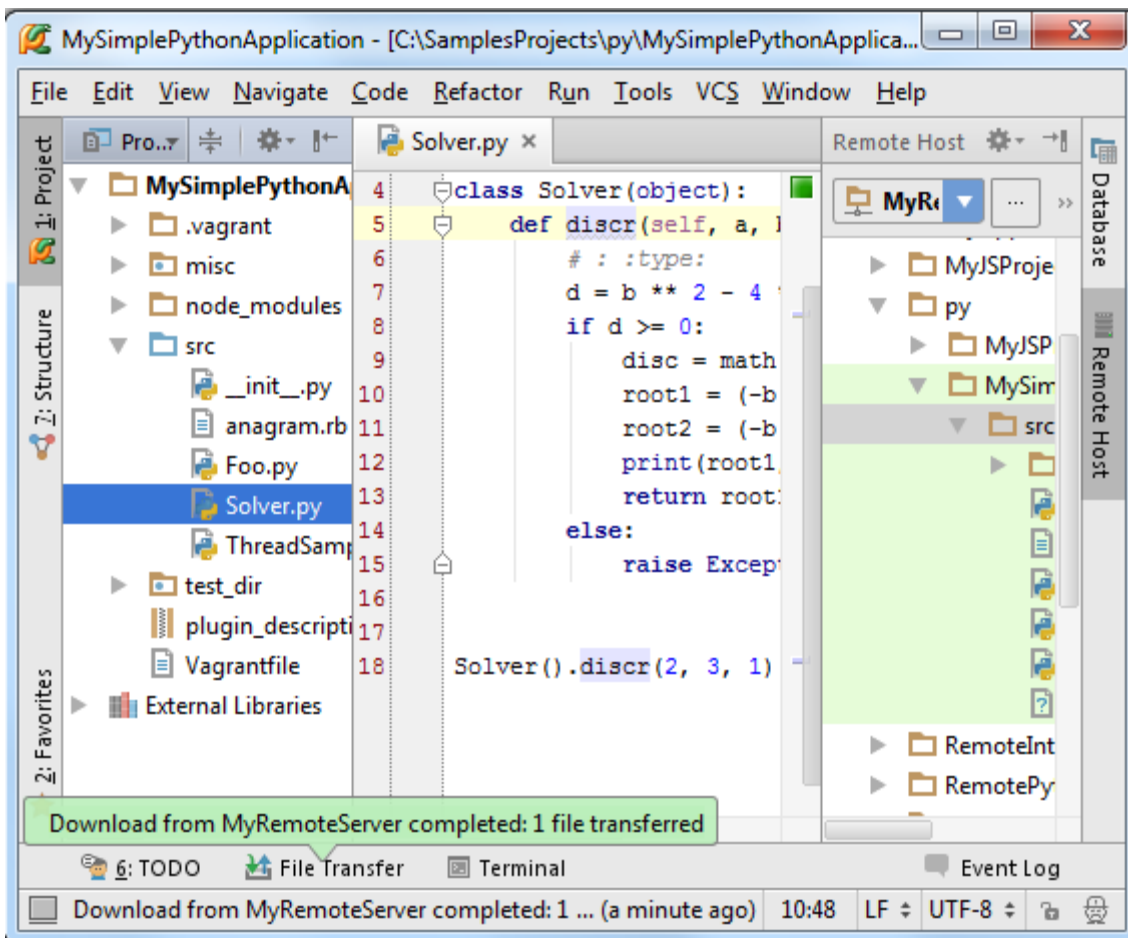
更多信息参见 [product documentation](#)。

11、下载

右击 Solver.py，选择 Deployment→Download from MyRemoteServer，Pycharm 会立即给出警告提示：

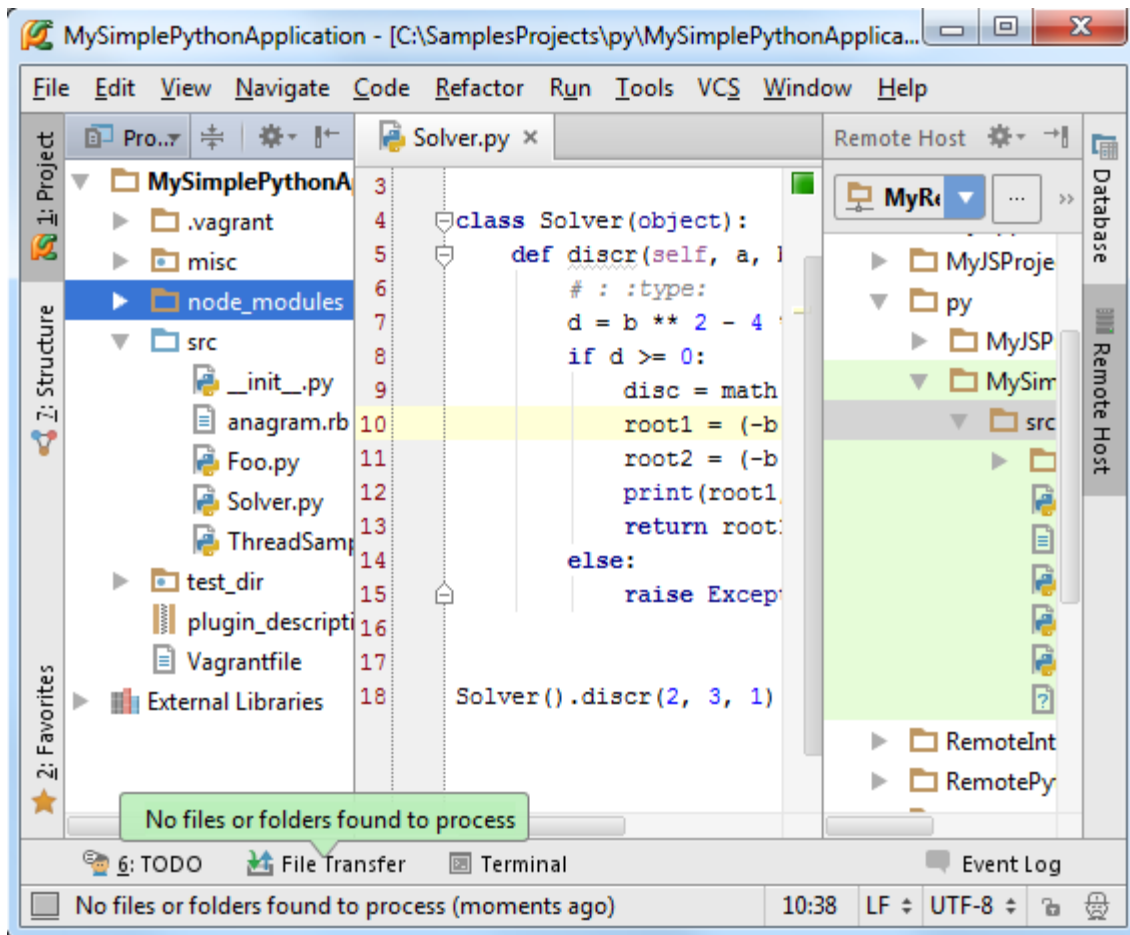


不必惊慌，直接单击 Continue 即可：



对应的你也可以下载整个目录下的文件，前提是这些文件已经上传完成。例如，右击父目录 src，执行相同操作，即会自动下载目录中所有嵌套文件。

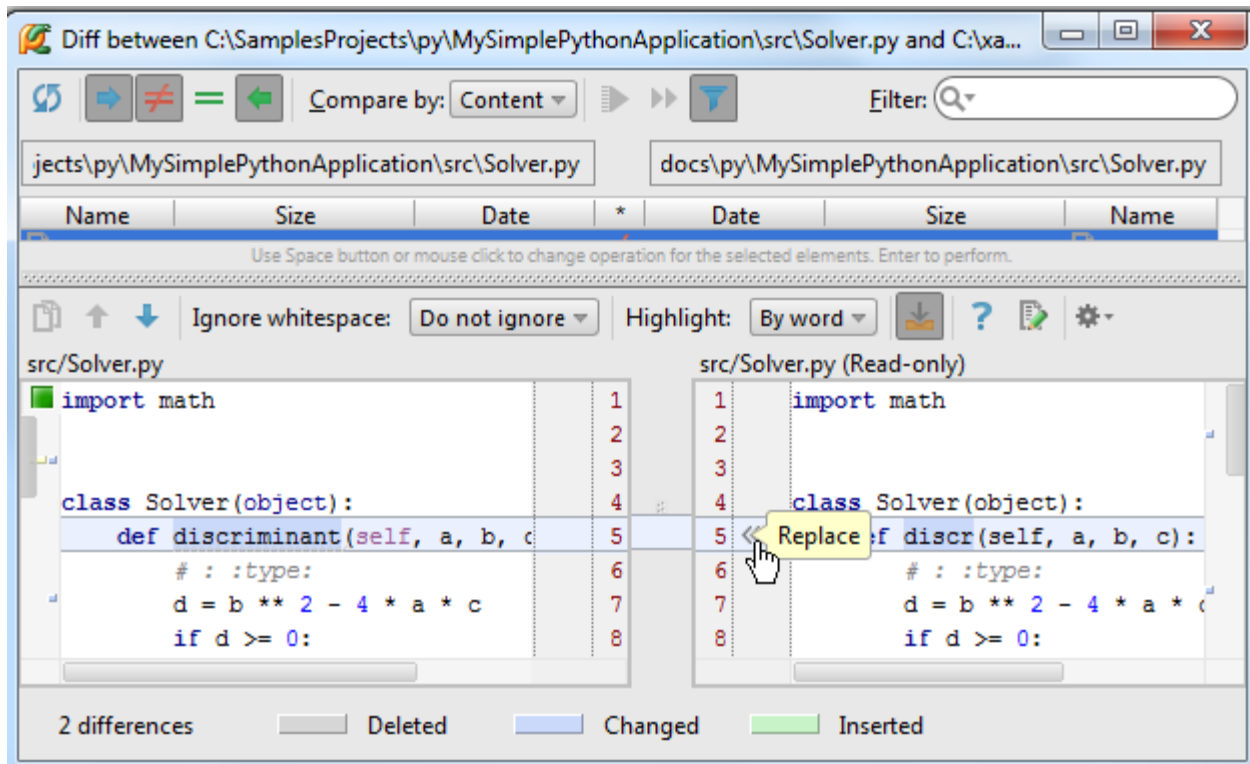
然而如果你试图下载一个还未上传的文件，Pycharm 会给出下载失败的提示：




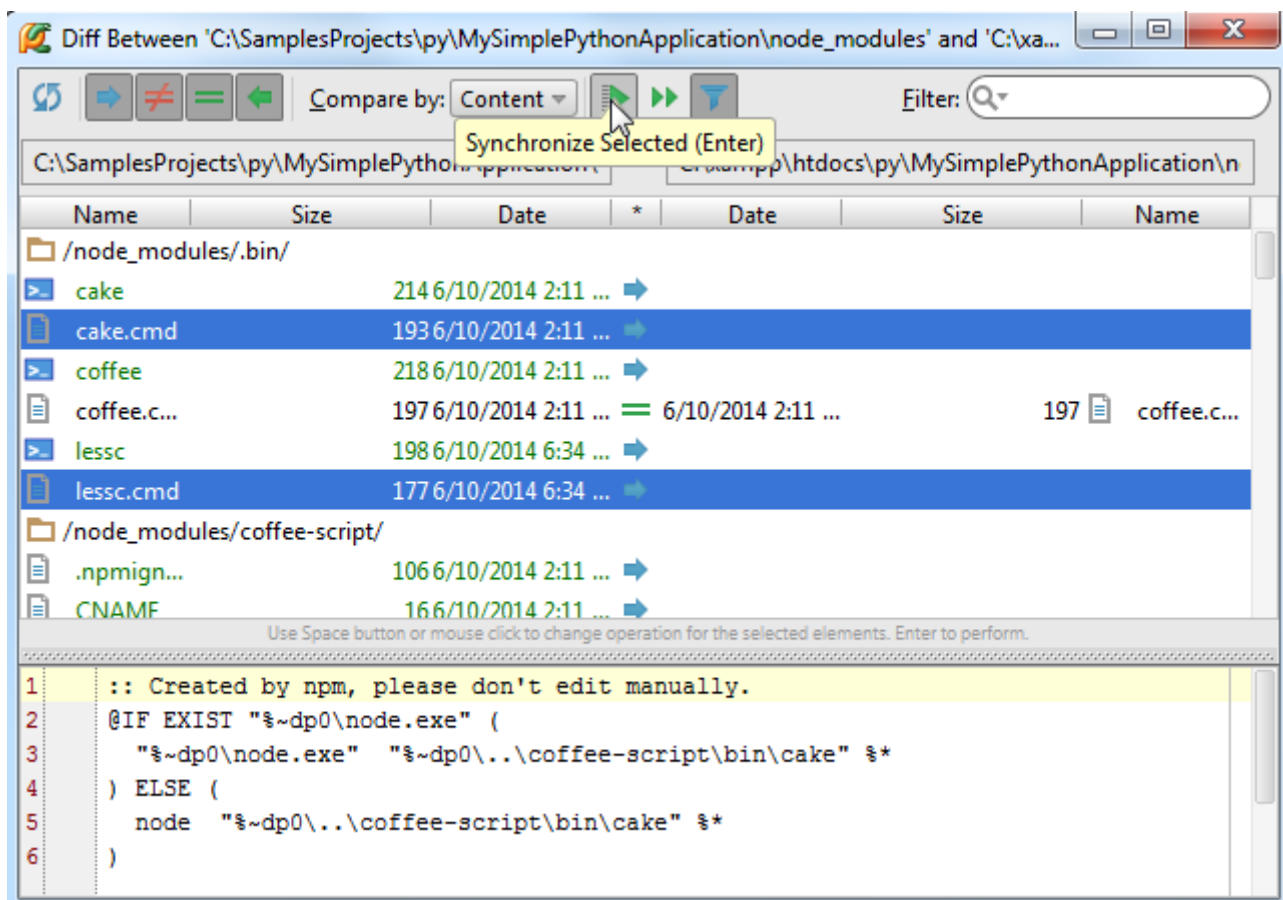
12、同步机制调整

首先需要进行一步准备工作，就是将对 Solver.py 文件所做的更改撤销（Ctrl+Z），此时会再次看到 Solver.py 文件中类的成员函数重构名称。

接下来右击 Solver.py, 选择 Deployment→Sync with Deployed to MyRemoteServer, Pycharm 弹出 [differences viewer for folders](#) 窗口，在这里你可以通过 *shevron* 按钮来核对所做的单独改变：



当然也可以对整个文件夹进行同步操作。例如右击 `node_modules` 选择 `Deployment`→`Sync with Deployed to MyRemoteServer`，此时会在左窗口中显示 `node_modules` 文件夹中的内容，但右侧窗口为空，因为该目录下的文件并未上传到服务端。我们可以在此处进行上传。选择待同步的文件，单击工具栏上的  按钮：



13、自动上传至缺省服务器

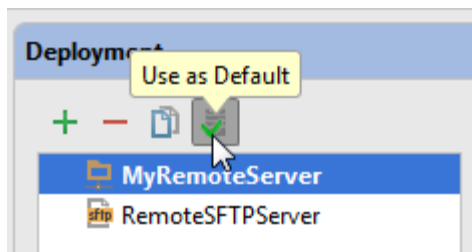
当用户需要在服务端使用与 Pycharm 工程中完全相同的文件时，自动上传功能就显得很用帮助。自动上传功能意味着无论在 IDE 中对代码进行了何种改变，Pycharm 都会自动将其保存在已部署的默认的服务端。

14、将服务器指定为缺省服务器

缺省服务器的最大优点就是可以使用自动上传功能，指定方法如下：

(1) 在 [Deployment page](#) 页面上选择一个服务器。有两种打开 [Deployment page](#) 页面的方法：要么使用 Settings/Preferences→Deployment 菜单命令，要么使用 Tools→Deployment→Configuration 菜单命令。

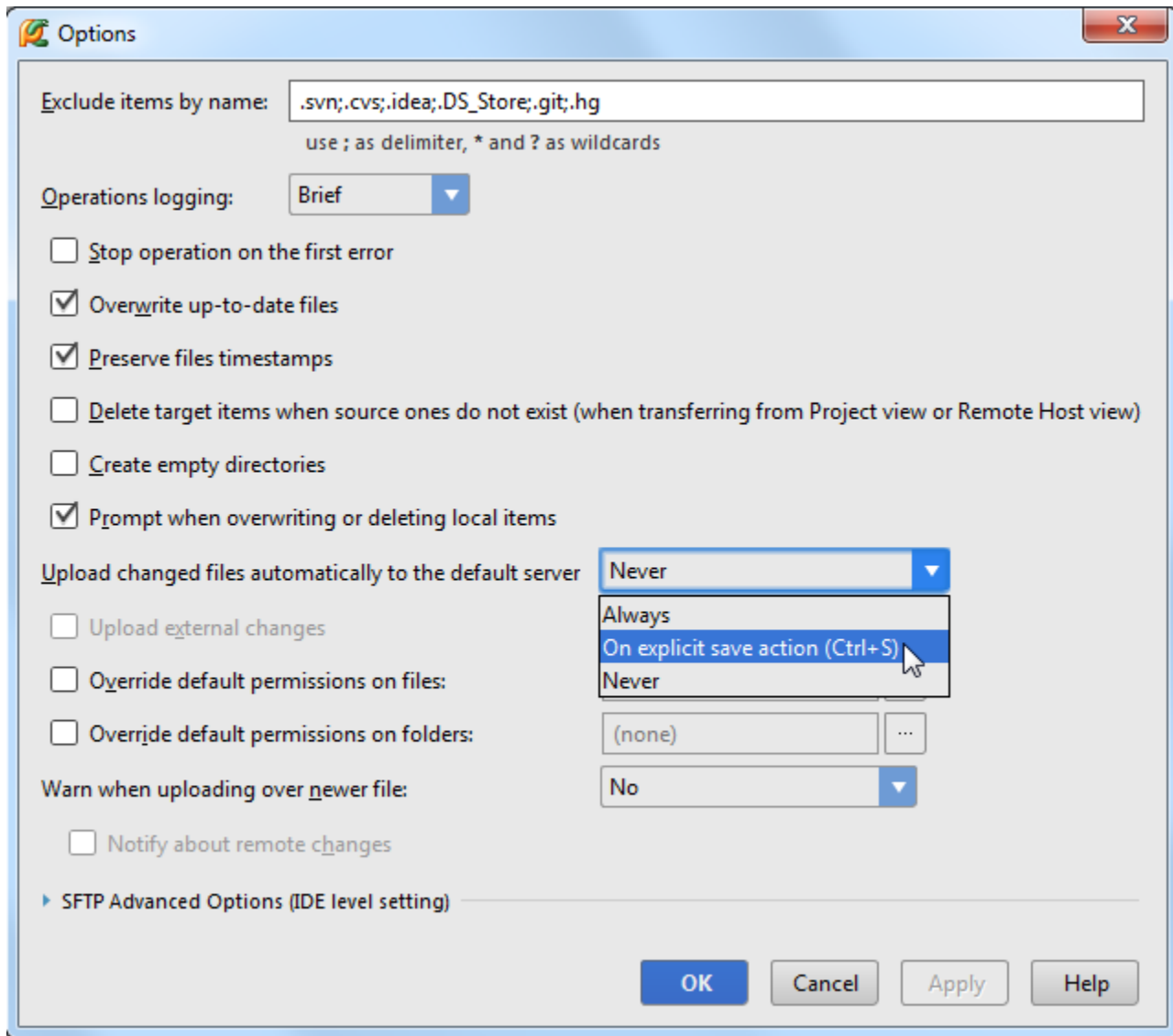
(2) 在 server configurations 列表中，单击缺省按钮：



15、启用自动上传功能

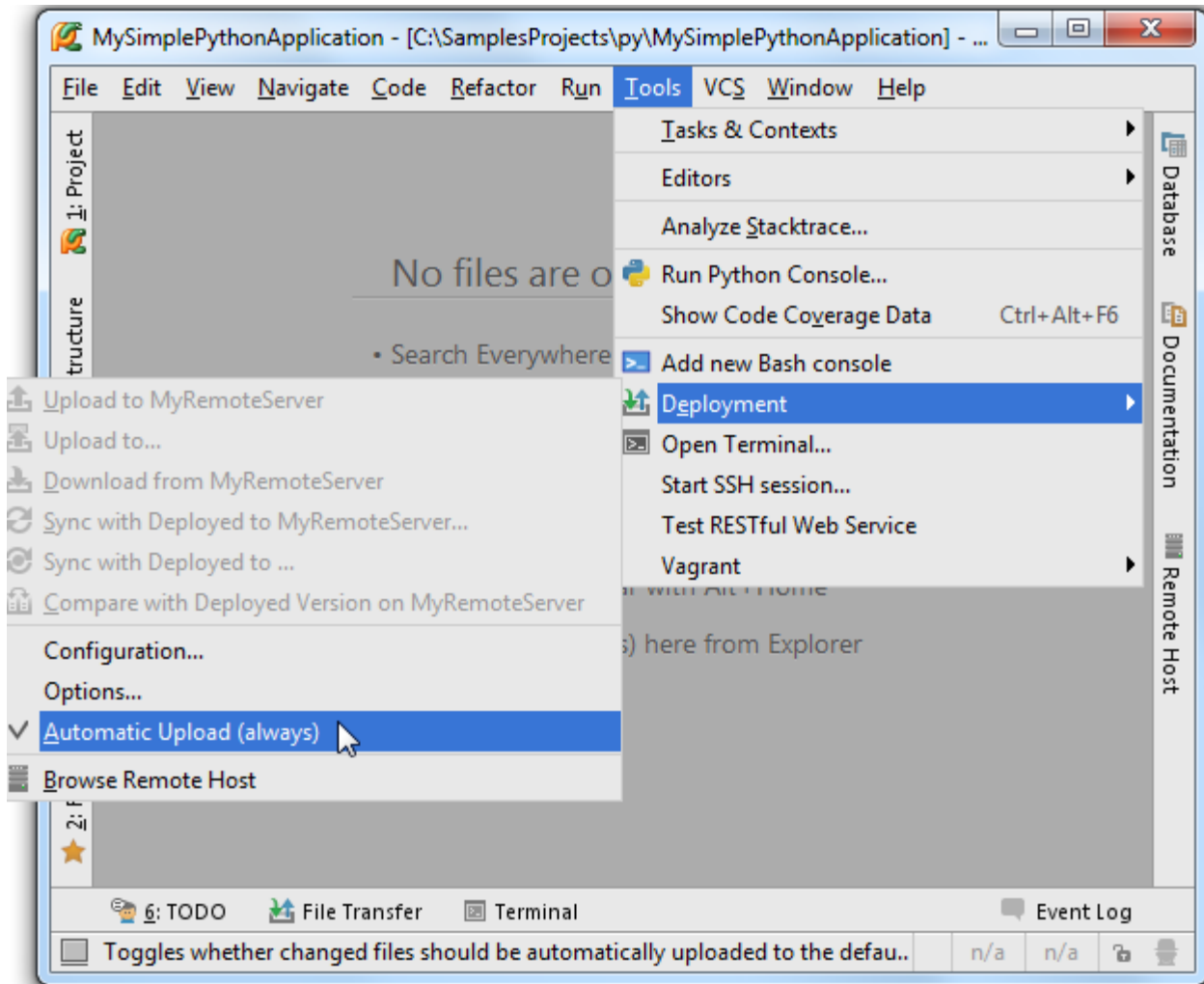
设置完缺省服务器之后，接下来就是开启自动上传功能。做法如下：

首先，打开 [Options](#) 部署选项（主菜单中的 Settings/Preferences→Deployment→Options 或者 Tools→Deployment→Options 命令），在 Upload files automatically to the defaylt server 选项中选择 Always，或者 On explicit save action 选项：



以上两个选项之间的差别详见 [in the field description](#)。

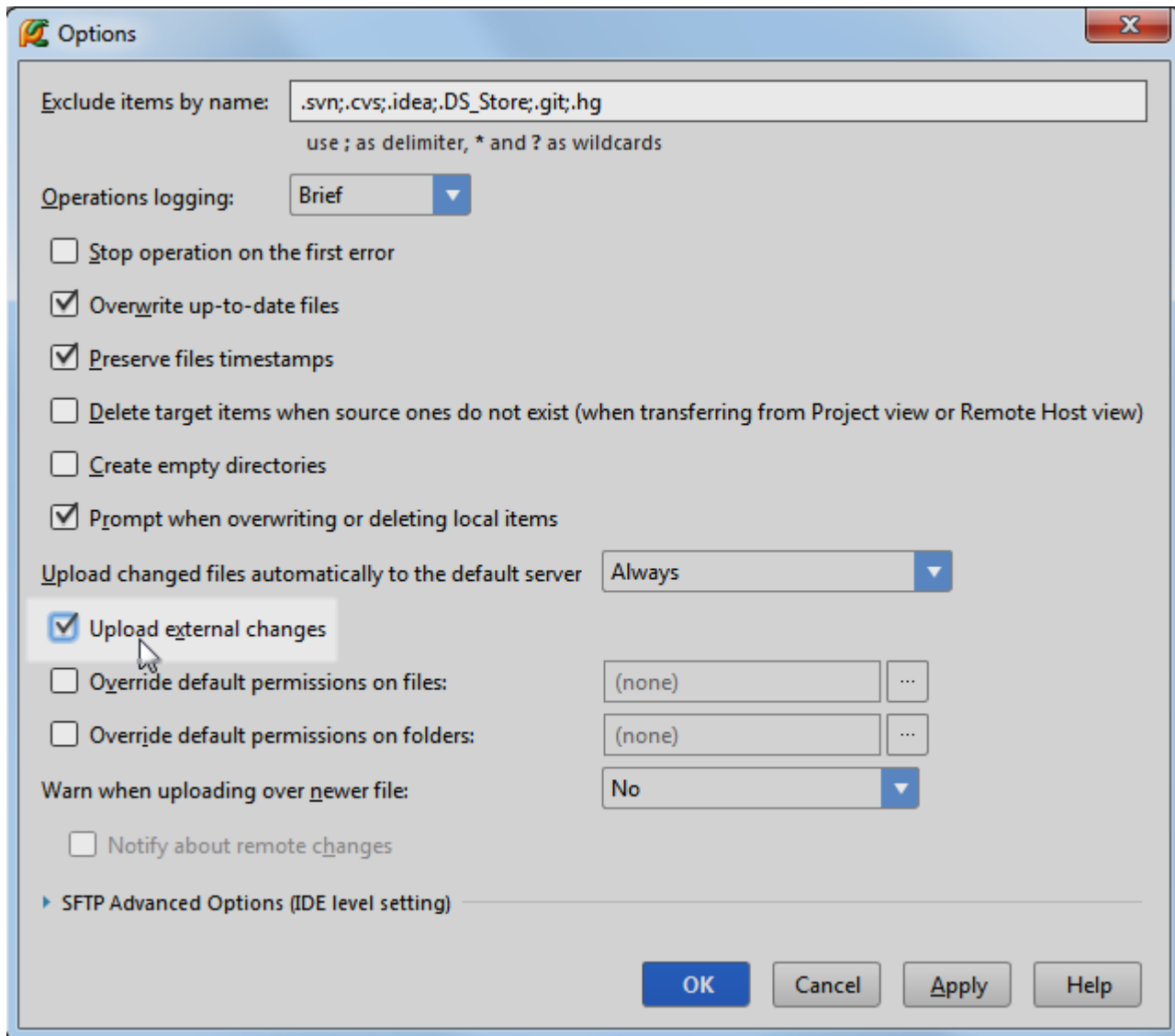
第二步，勾选主菜单 Tools→Deployment→Automatic upload 复选框，注意此时的 automatic apload 选项已经处于 *Always* 的模式：



值得一提的是，我们并不推荐在发布的产品中使用 *Always* 模式，避免我们在部署过程中不经意上传了未开发完整的代码，从而破坏了产品的稳定性。

16、上传外部更改

默认情况下 Pycharm 只上传文件自身的更改情况，如果我们通过其他途径对文件进行了更改，例如通过 VCS branch、transpilation of SASS 或者 LESS or a File Watcher 进行的更改，Pycharm 是不会将这些更改自动上传的，为了保证这些更改也能顺利上传，需要启用 Upload external changes 功能：



最全 Pycharm 教程（14）——Pycharm 编辑器功能总篇

1、主题

在编写代码的过程中，大部分时间都花在了编辑框的交互中。为了能够更高效的使用这个工具，我们将对其一下特点进行分节介绍：

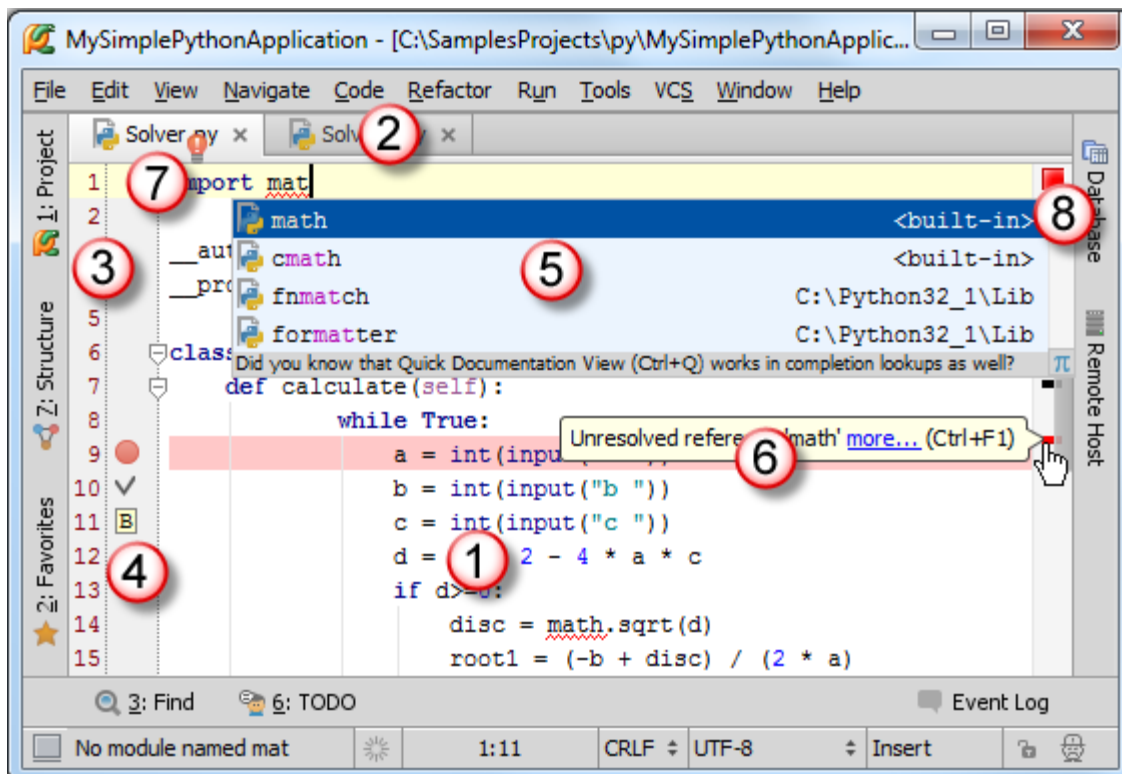
- Viewing documentation
- Using macros
- Syntax highlighting and error indication
- Managing editor tabs
- Intention actions and quick fixes
- Creating and applying live templates (code snippets)
- Code folding
- Code completion
- Auto-import
- Auto-generating code
- Auto-editing with PyCharm

更多内容详见 [online help](#)。至于编程方面的知识参见 [Python](#) 和 [Django](#) 文档。

2、准备工作

- (1) Pycharm 版本为 2.7 或者更高。
- (2) 已经创建并配置了一个工程。

3、鸟瞰编辑框



- (1) 主编辑区，用来编写代码。

(2) 编辑框的标题栏，标记了当前处于激活状态下的编辑框选项卡，每个选项卡中都隐含了大量的相关的快捷菜单命令，详见 [Managing Tabs](#)。

(3) 行号，默认显示，如果需要可以通过右击行号取消 **Show line numbers** 复选框来隐藏行号。当然通过 **Settings | Editor | Appearance - Show line numbers** 主菜单命令也能到达同样目的。

(4) 俗称左槽，此时这里显示了三种图标，分别为用于调试的断点 [breakpoint](#)，用来导航的书签 [bookmarks](#)。

(5) 弹出的拼写提示 [code completion](#) 窗口

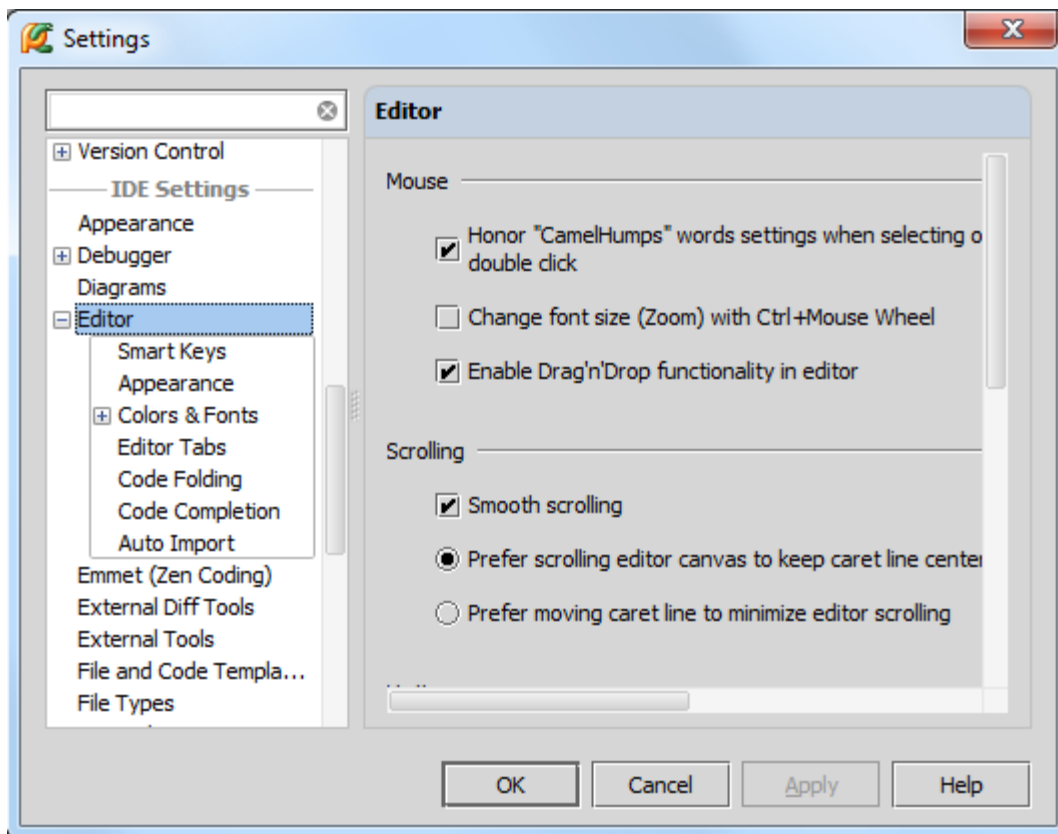
(6) 俗称诊断标志球。当代码出现错误时，Pycharm 会以红色波浪线标记错误代码行，在右槽对应行显示诊断标志球，并给出具体提示。

(7) 在拼写提示菜单的左侧，有一个红色的灯泡，功能参见 [quick fix](#)。

(8) 右槽，处于编辑框的右侧，显示各种颜色的标志来指示代码状态，是否存在错误、警告等等。同样具有导航功能，方便我们快速定位代码出错位置、转到指定代码行等等。

4、配置编辑器

Pycharm 的编辑器配置十分灵活，在设置对话框中 (**Ctrl+Alt+S - IDE Settings - Editor**) 可以对其外观和行为进行各种各样的更改：



在这里你可以找到关于鼠标、滚轮、代码范围、错误高亮显示方案等相关的设置命令。

虽然你可以直接使用默认的编辑器设置，但如果你希望修改其中的部分设置的话，可以参考 [editor options](#) [here](#) 或者单击界面的 help 按钮。

举个例子，你可能希望使用“ctrl+鼠标滚轮 来改变字体大小”这一功能，这个功能默认情况下是关闭的，需要手动打开。再比如你可能已经习惯了“Autoparse delay”功能，这些都需要手动进行设置。

接下来我们会分解对编辑器的功能进行介绍。

最全 Pycharm 教程（15）——Pycharm 编辑器功能之自动生成格式

1、主题

之前已经介绍过，Pycharm 具有强大的拼写提示功能，包括 [basic](#) 和 [smart type](#)，除此之外 Pycharm 还能自动生成一些结构化代码，插入成对的花括号和引号，接下来我们一探究竟。

2、自动缩进

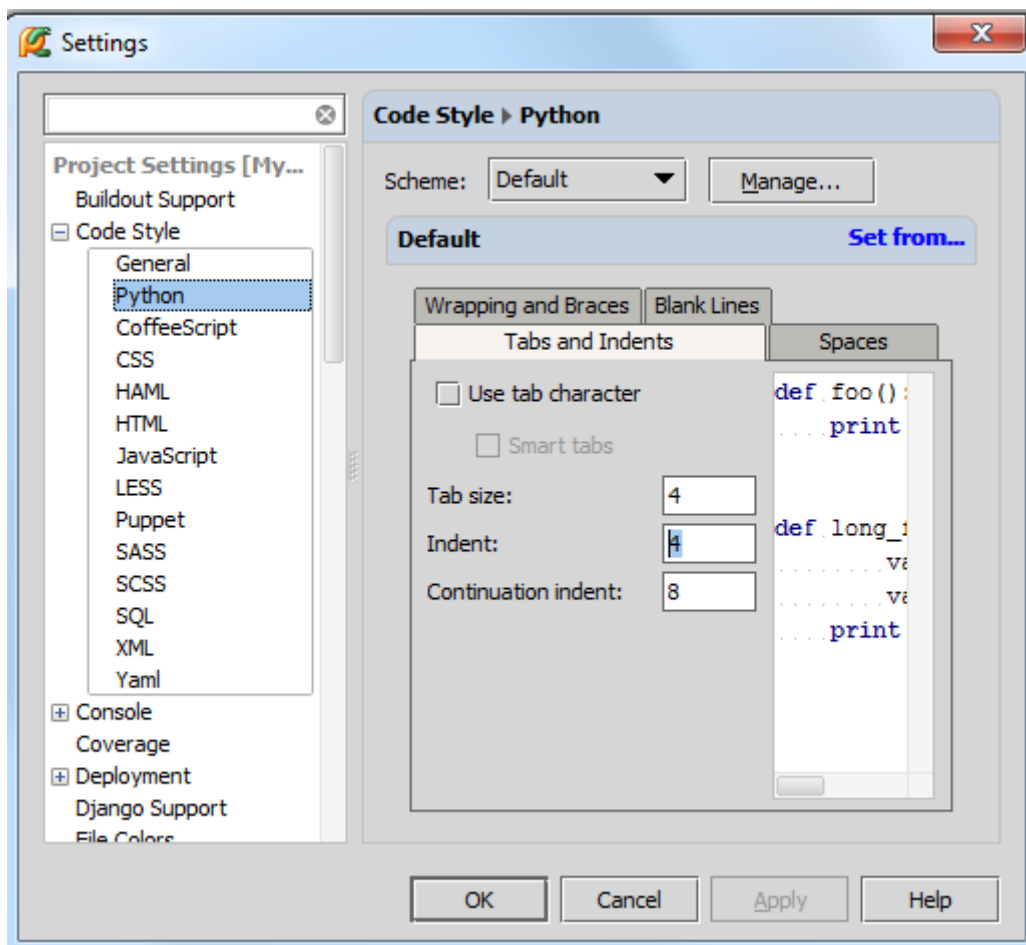
键入类声明，然后回车：

```
class Solver:
```

Pycharm 光标会自动预留缩进，此处可以开始输入函数声明：

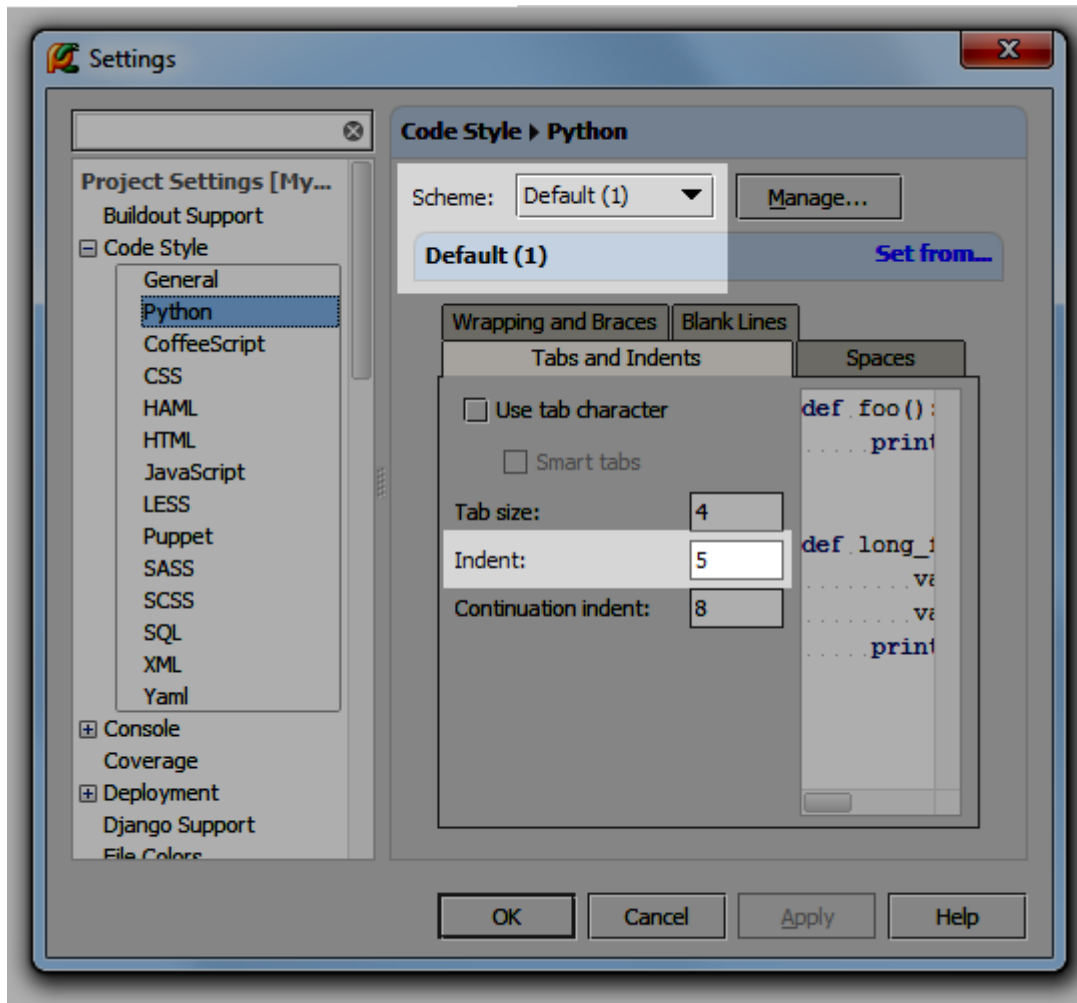
```
class Solver:
    def demo
```

此时自动缩进的机制已经启动。这里函数声明（以及所有的嵌套操作）默认从左边缩进了 4 个空格位置。这里的缩进规则涉及了 Python 相对固定的代码风格标准，并且这个标准是根据你的团队定制的（即一个公司需要遵循统一标准），指定好标准后，Pycharm 会帮助我们按照标准来维护代码风格。接下来我们介绍 Pycharm 具体是如何实现这个功能的。同样以这段简单的代码为例，打开设置对话框，展开 Code Style 节点，打开 [Python](#) 页（Ctrl+Alt+S→Project Settings→Code Style→Python）：



如你所见,这里规定的缩进尺寸为4个空格,Pycharm按照这个规则来进行代码的自动生成以及格式检查 [reformatting](#)。当然我们更改缩进规则(就和修改其他代码规则一样)。假设你希望缩进个数为5个空格,即生成的所有新的代码相对于上层代码的缩进个数均为5个空格。浏览 [Smart Keys](#) 页面获得更详细的信息。

然而 Pycharm 预设的代码风格框架是不可更改的,在进行私人订制之前 Pycharm 会自动拷贝一份,拷贝的副本供我们修改:



这份新的配置框架保存在你用户目录下,有必要的话你可以与其他队友分享,方便整个团队代码的风格统一。更多有关代码风格设置的信息详见 [Project and IDE Settings](#)。

3、语法自动补全

当你输入一个圆括号时(Pycharm 会试图补全另一半),Pycharm 会补全系统参数 self。回车之后光标会移动到下一个输入位置(根据当前制定的缩进规则),在此处输入需要打印的信息,输入一个圆括号,Pycharm 会自动补全另一半括号,并将光标至于括号内部以供我们输入待打印的信息:

```
class Solver:
    def demo(self):
        print("|")
```

接下来我们测试一个更复杂的情况——编辑一个 Django 模板。对于语法非常复杂的 Django 模板,语法自动补全功能就显得至关重要。OK,开始输入一个 Django 标签

```
{%
```

一旦你键入一个花括号，Pycharm 就会自动补全另外一个：

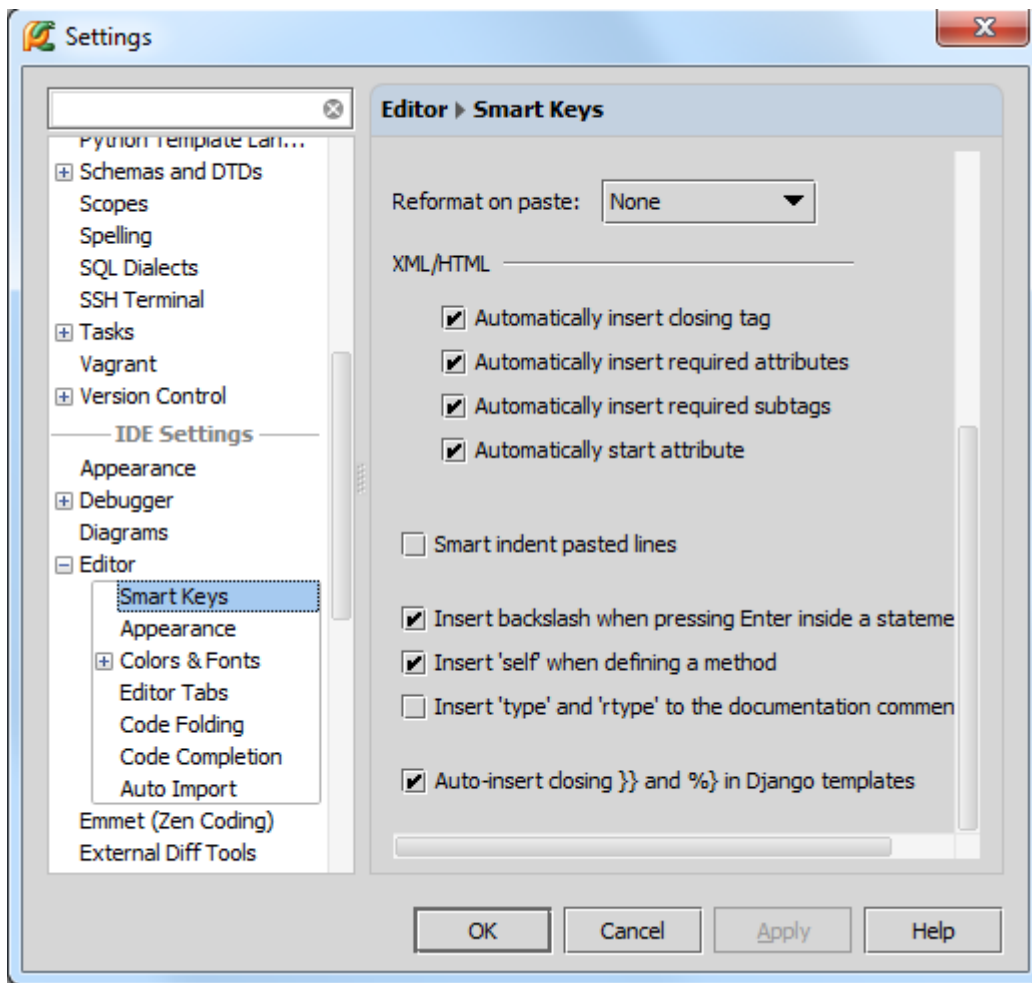
```
{% | %}
```

并且光标会停在两个%之间，方便我们输入代码：

```
{% if poll_list %}
```

类似的，对于{{标志 Pycharm 同样会自动补全另外一半。

语法自动补全功能的相关设置位于编辑器设置的 [Smart Keys](#) 页面：Ctrl+Alt+S→IDE Settings→Editor→Smart Keys:



举个例子，如果希望 Pycharm 能够自动补全圆括号、方括号等，需要勾选 **Insert pair bracket** 复选框；要想自动补全引号则需要勾选 **Insert pair quote** 复选框。对于 Django 模板，有一个专门的复选框 **Auto-insert closing }} and %} in Django templates** 以供选择。

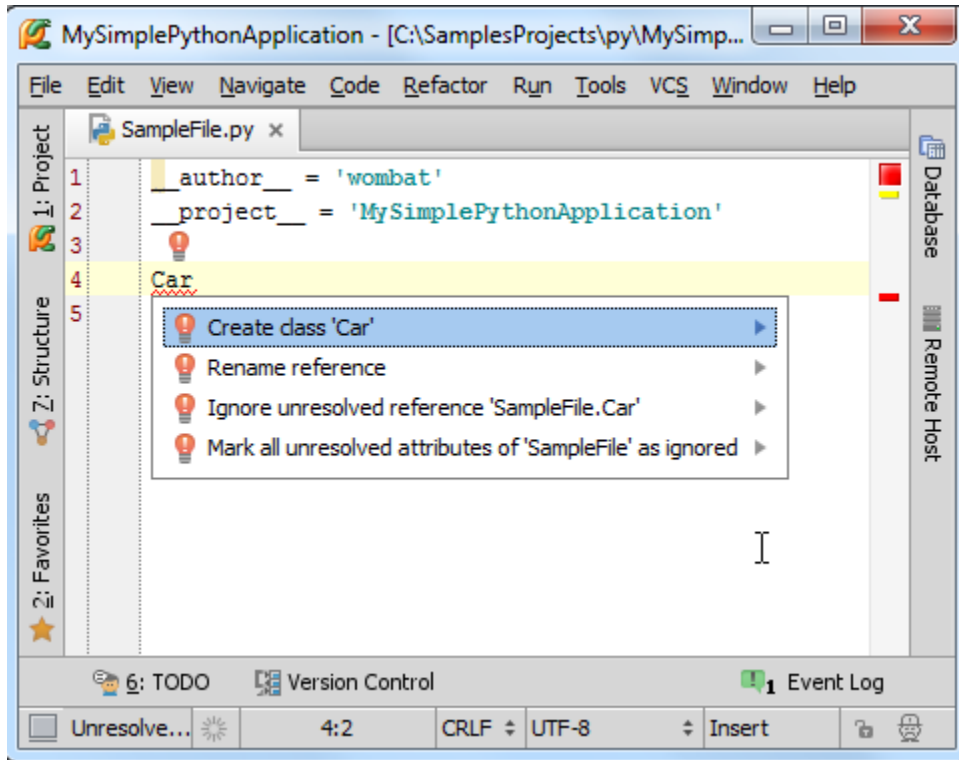
最全 Pycharm 教程（16）——Pycharm 编辑器功能之代码自动生成

1、准备工作

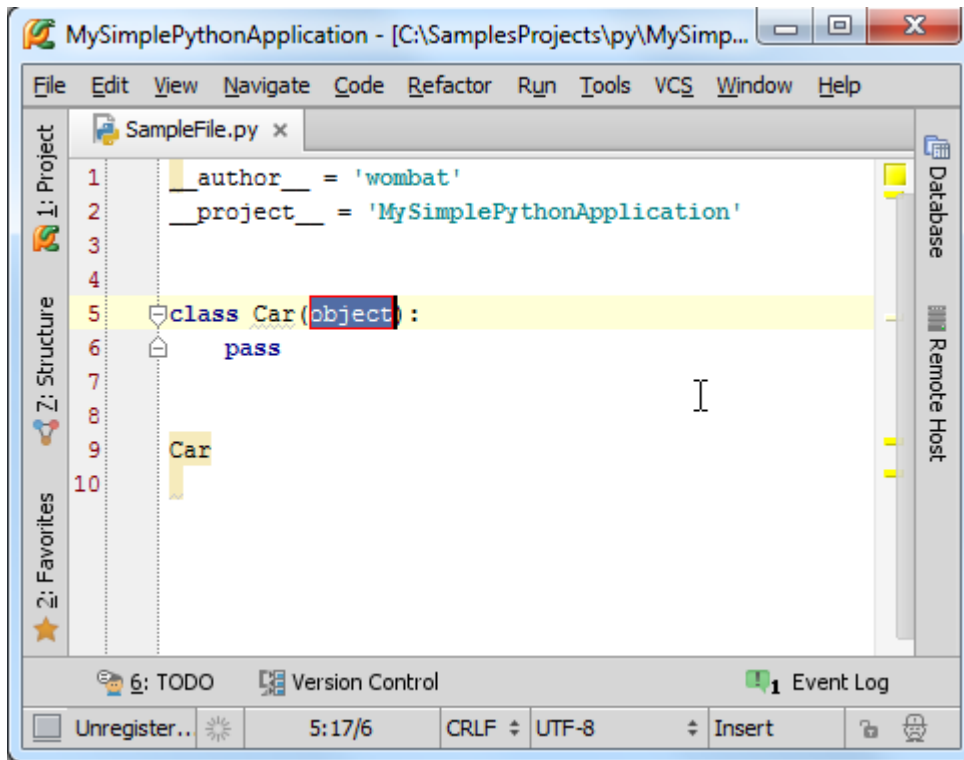
- (1) Pycharm 版本为 2.7 或者更高。
- (2) 已经创建一个工程。
- (3) 创建 Python 文件（Alt+Insert→Python File）

2、生成源码

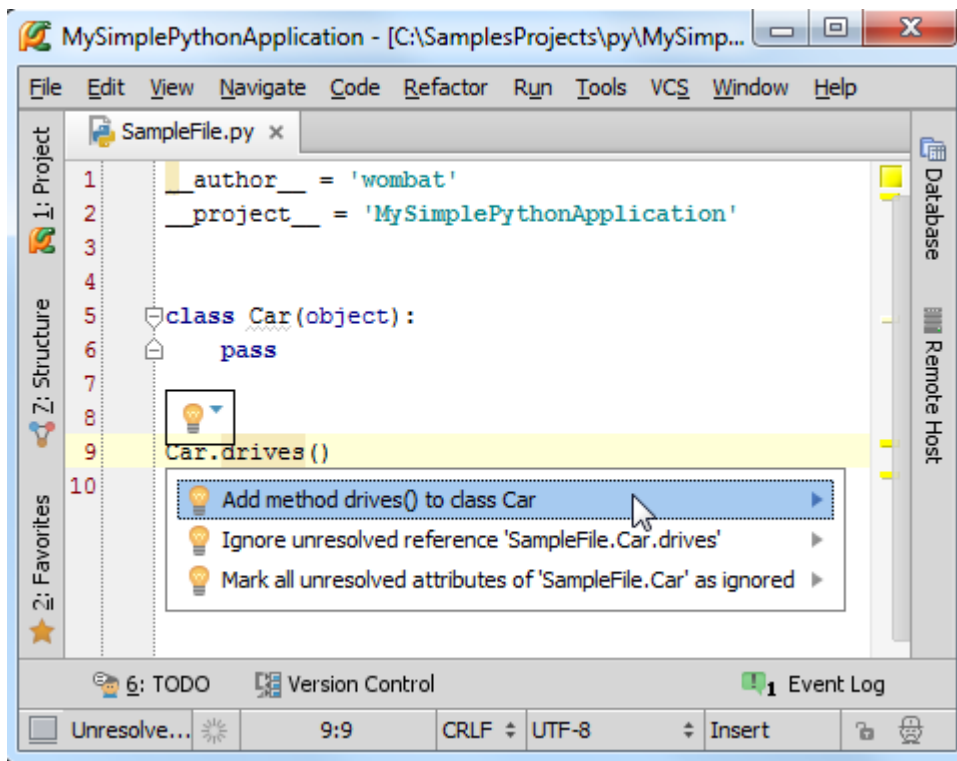
首先，我们实例化一个类，Pycharm 会立即显示一个红色灯泡来给出快速补全的建议：



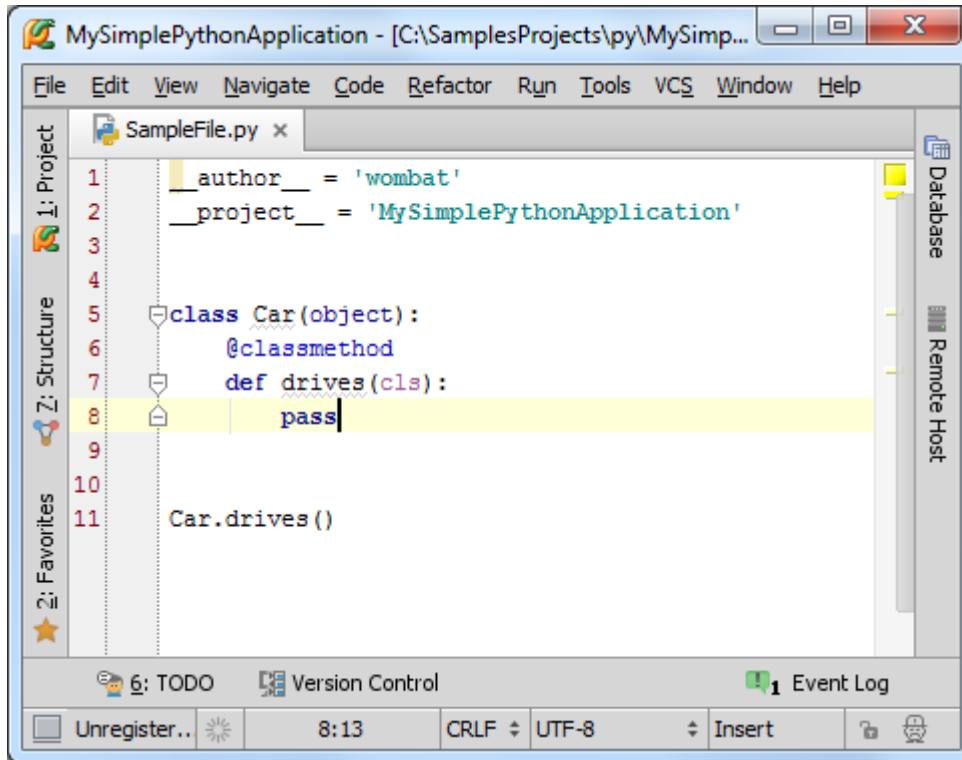
这里我们希望创建一个类，选择对应的提示命令，Pycharm 会根据名称自动创建一个类：



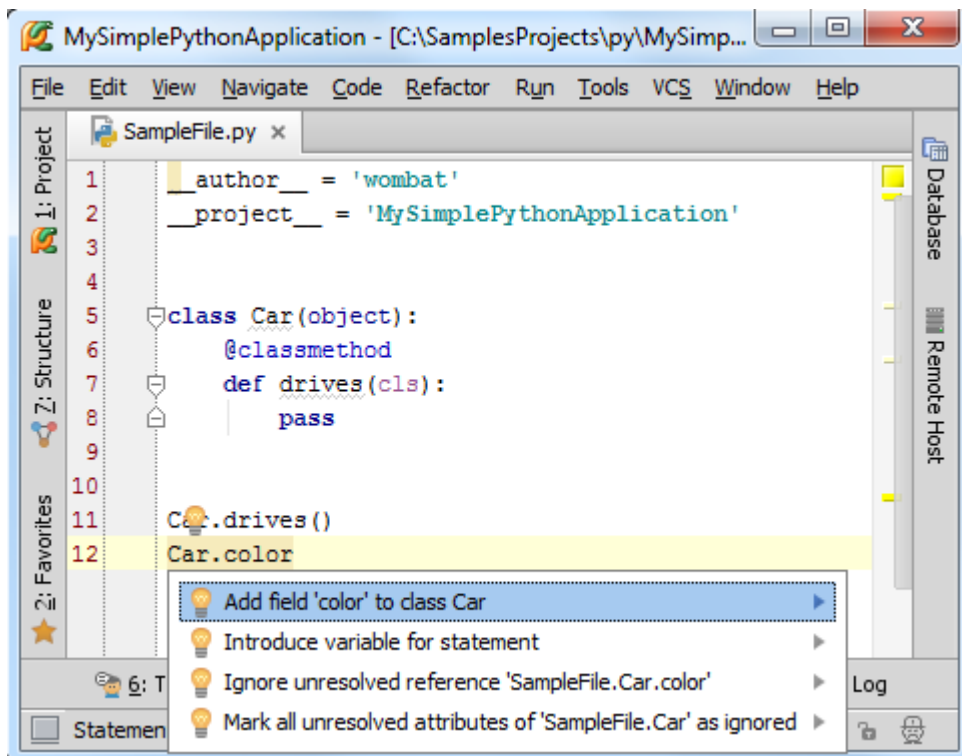
接下来我们调用这个类的成员函数（一般情况下，一旦你在类名后面输入一个点号，Pycharm 的代码补全机制就会列出当前可用的函数名称，然而在这里我们所用的成员函数还没有在类中进行创建）：



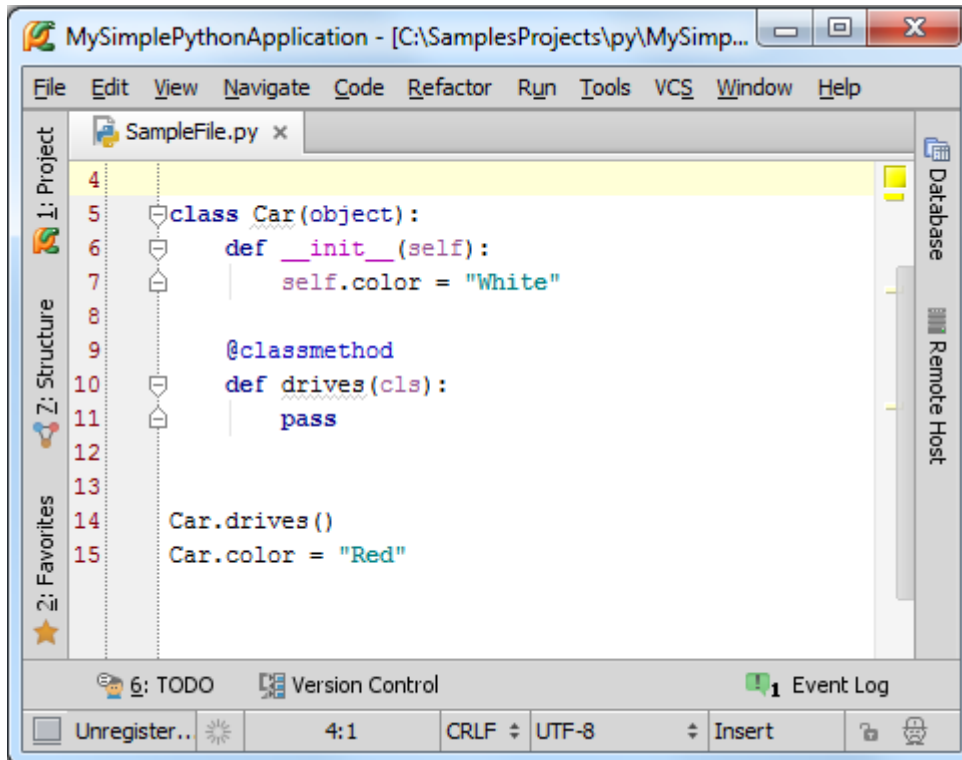
同样的情形，选择创建该方法，然后观察 PyCharm 如何生成成员函数：



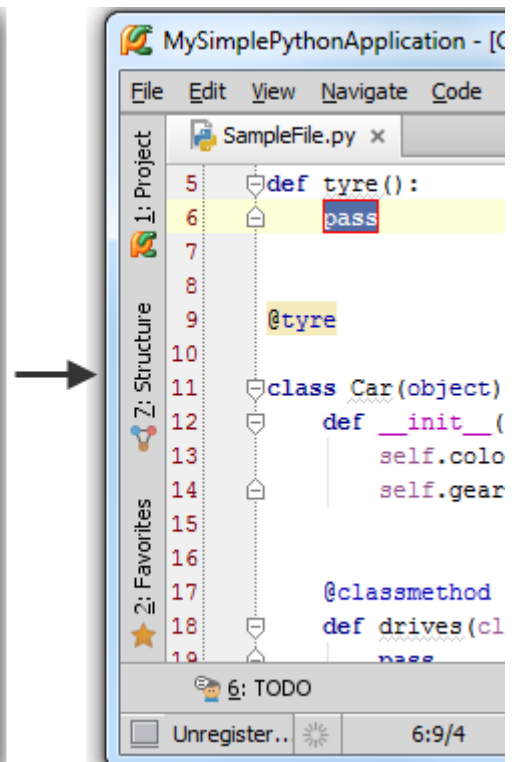
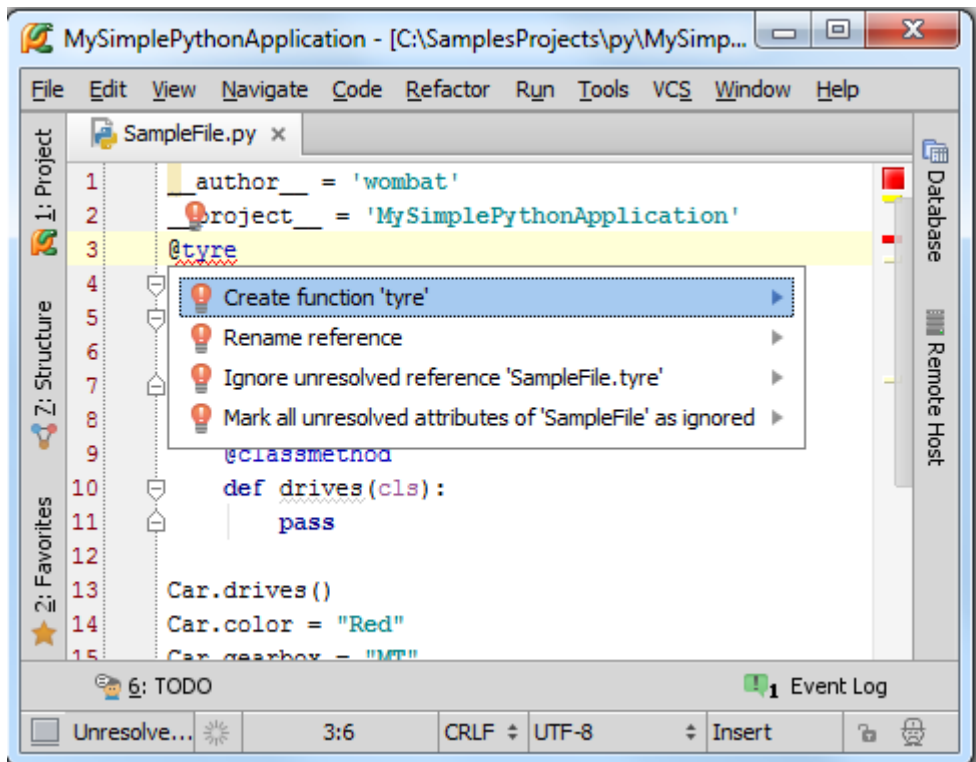
接下来我们准备向类中添加一个成员变量"color"，Pycharm 会提示我们创建一个成员变量：

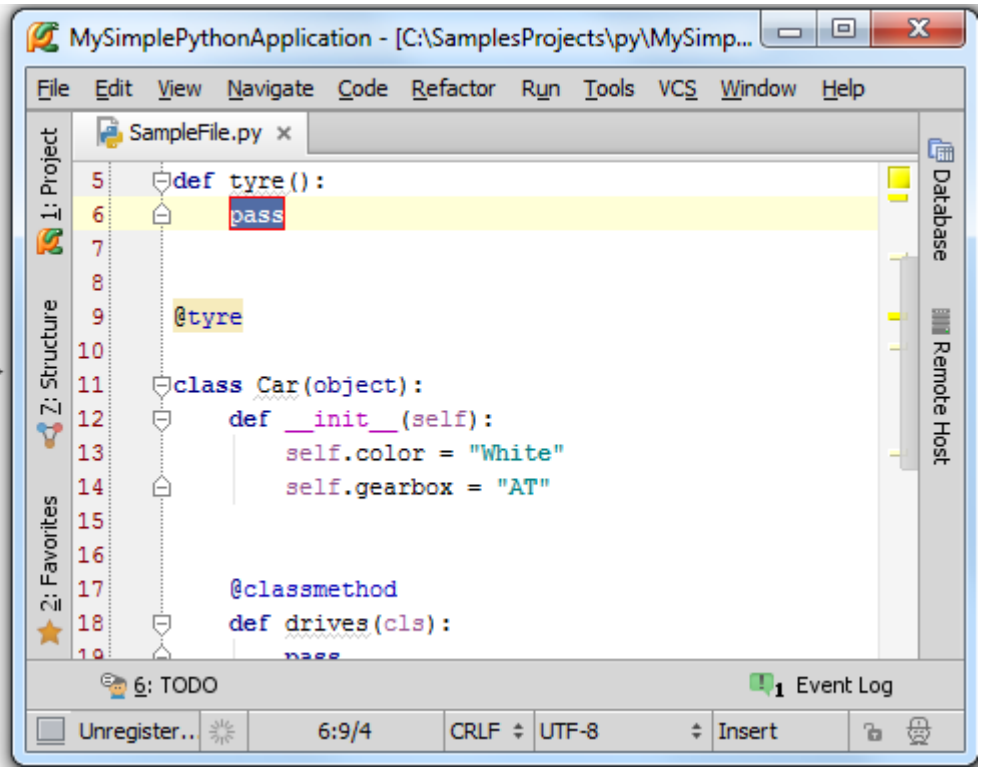
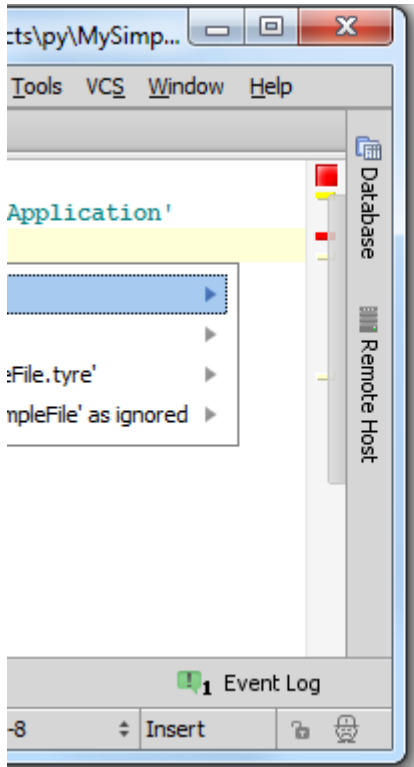


OK, Pycharm 智能的完成了变量的创建添加——创建了一个构造函数：



最后，也可以通过 Pycharm 智能创建一个全局函数：





最全 Pycharm 教程（17）——Pycharm 编辑器功能之自动导入模块

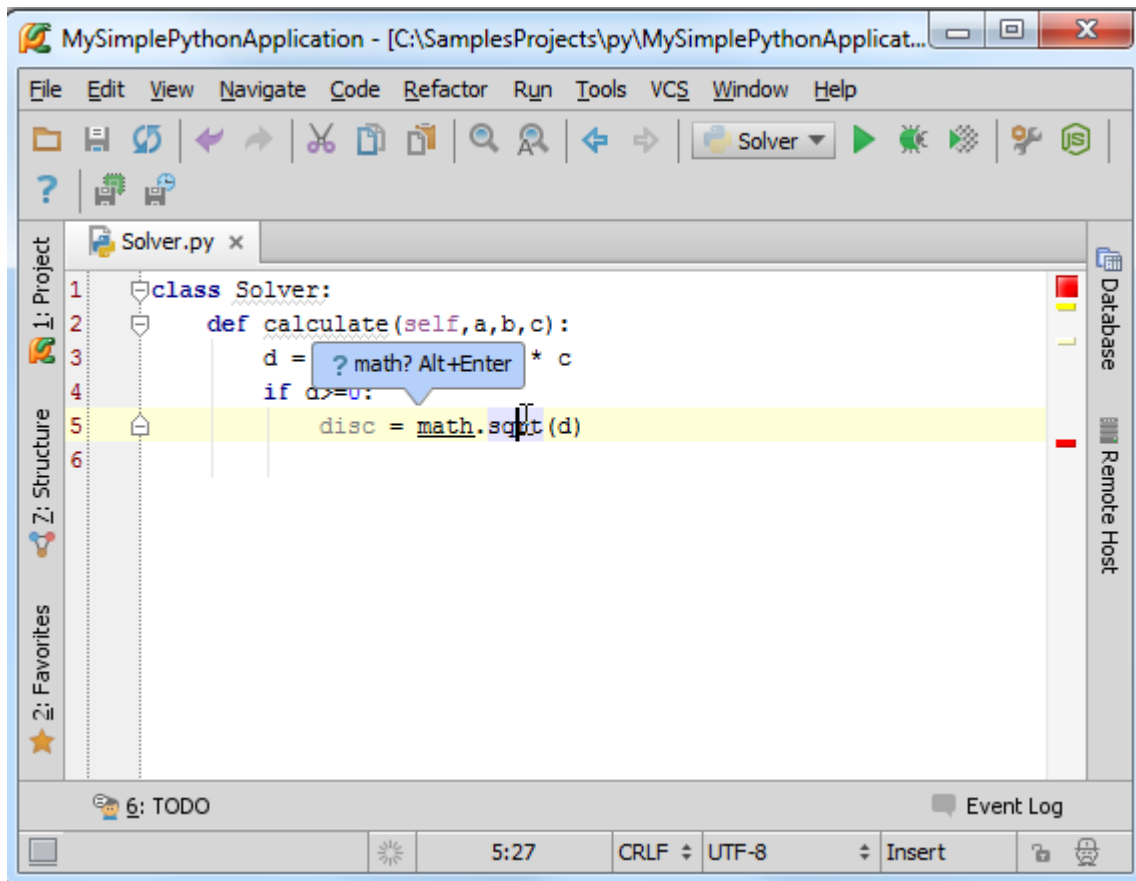
1、导入模块

我们在编程过程中经常会不经意的使用到一些尚未导入的类和模块，在这种情况下 Pycharm 会帮助我们定位模块文件位置并将其添加到导入列表中，这也就是所谓的自动导入模块功能。

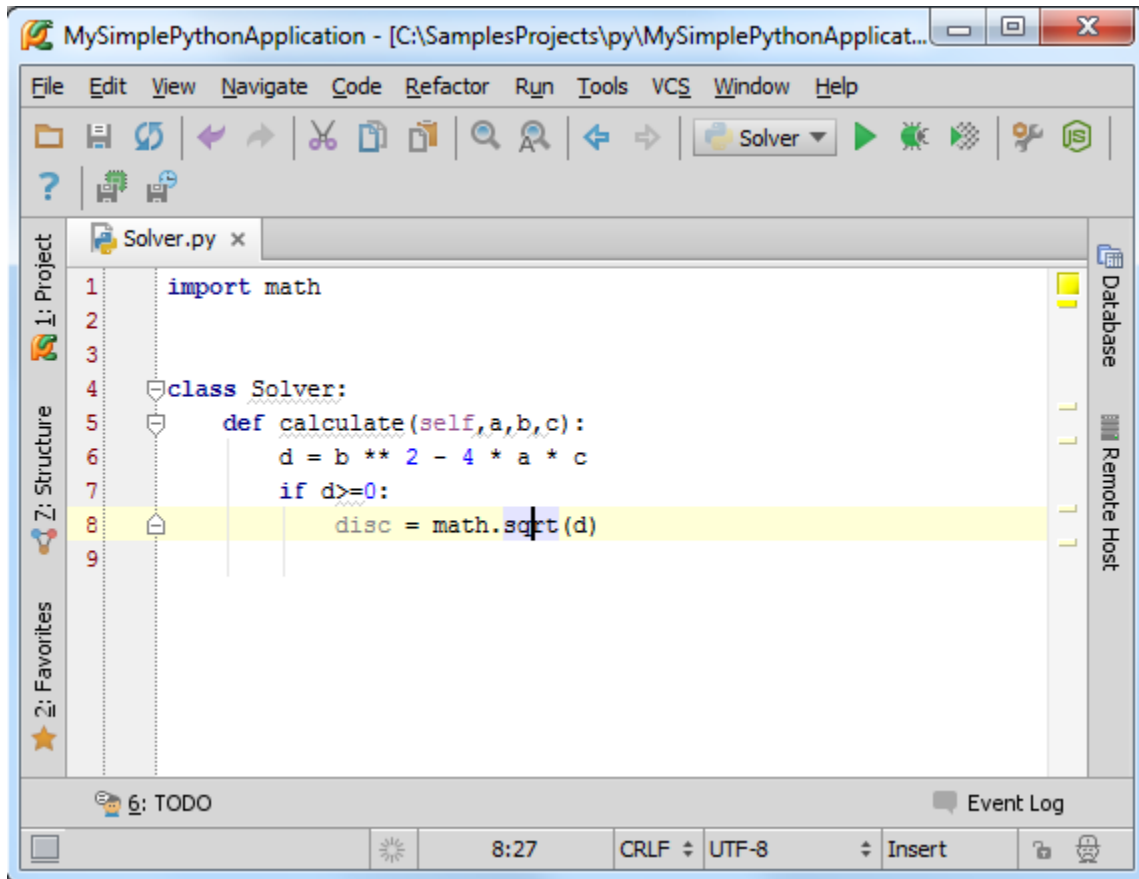
为了研究这个功能，我们借用之前已经编写好的 Solver 类，输入以下代码：

```
class Solver:
    def calculate(self,a,b,c):
        d = b ** 2 - 4 * a * c
        if d>=0:
            disc = math.sqrt(d)
```

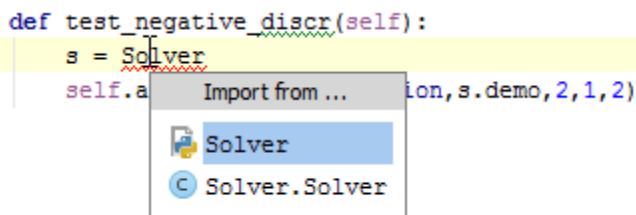
在输入 `math.sqrt(d)` 的时候，Pycharm 会弹出一个菜单来提示你导入缺失的模块：



按下 `Alt+Enter`，采取快捷菜单中的建议，此时 `import` 命令会被添加到导入模块的代码部分，并且输入光标仍留在原位，方便我们继续输入而无需重定位：



值得一提的是，如果当前有多个可选的导入资源，Pycharm 会给出提示列表来供用户选择：

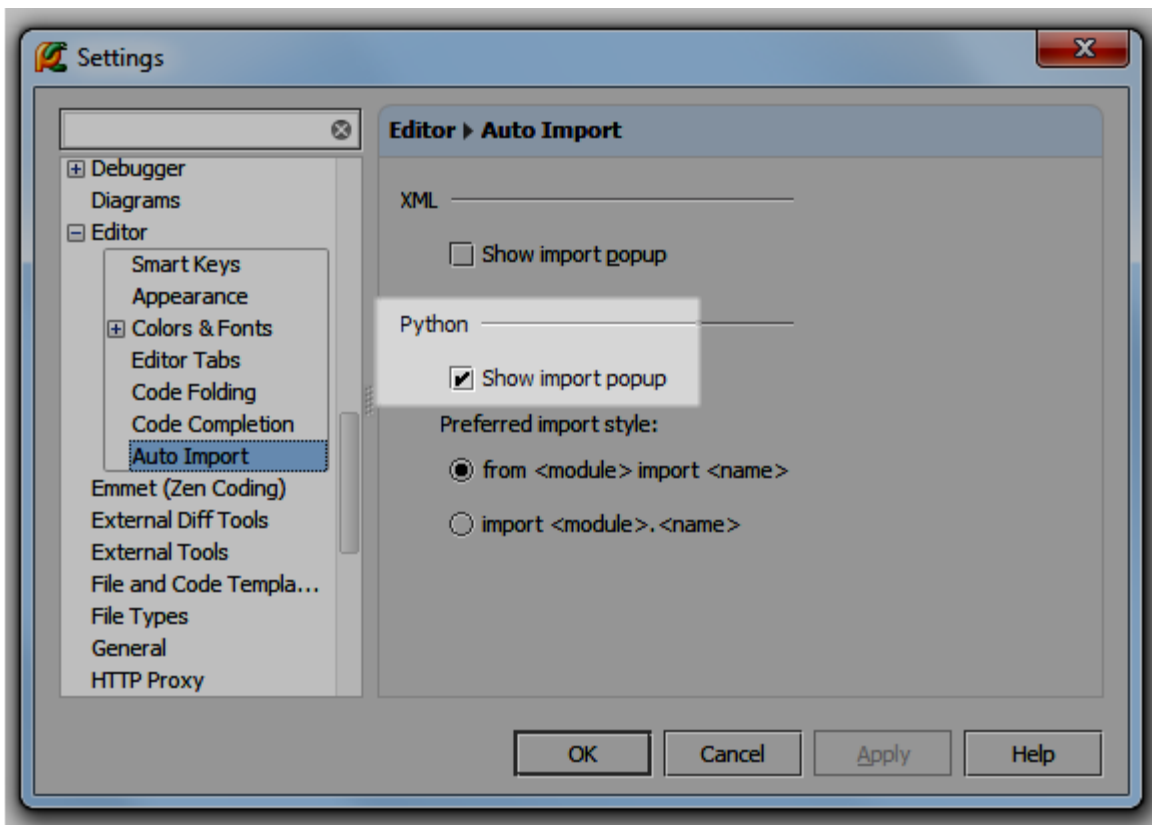


2、配置导入助手

这里有一个小问题，如果这个窗口出现让你很烦恼，不要着急，单击右下角那个帅哥就可以关闭它了：



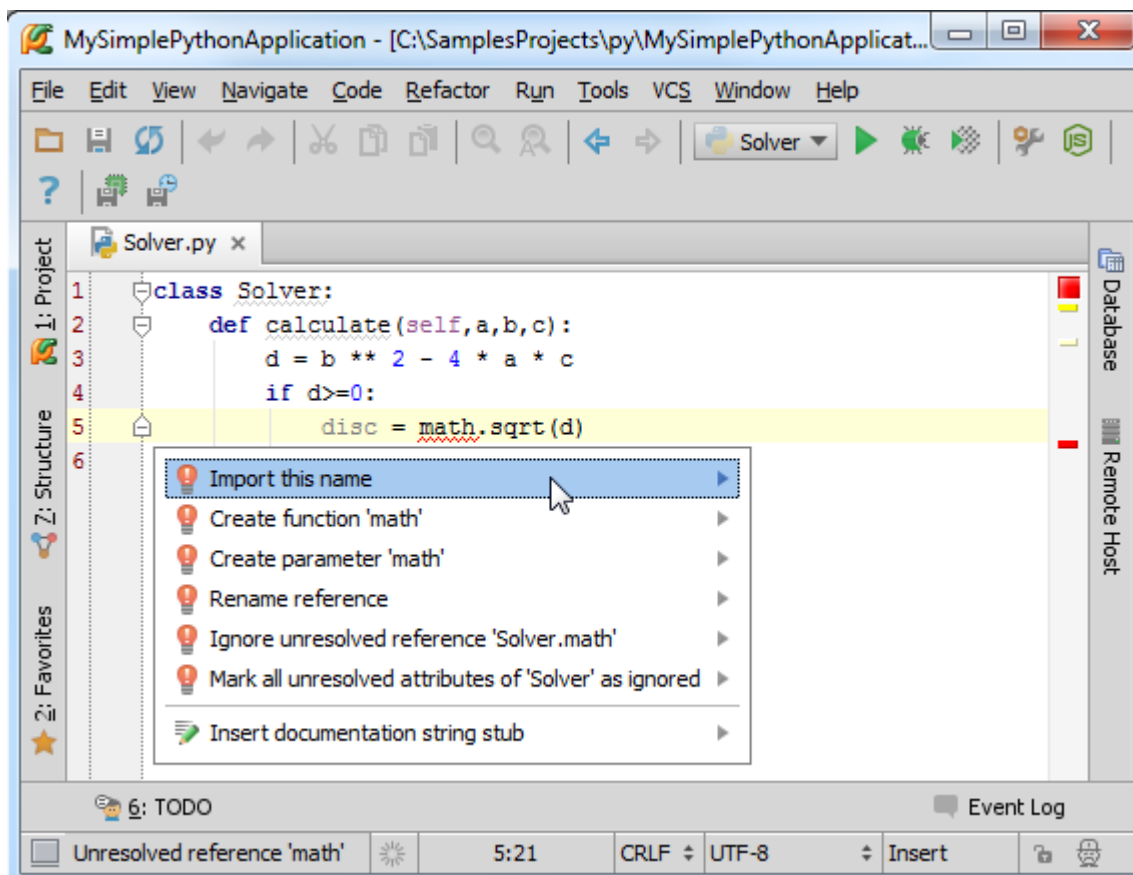
单击那个帅哥头像会弹出一个窗口，取消 **Import popup** 复选框，此时就取消了自动导入的功能。当然在配置对话框的 [Auto-Import page](#) 页面也可以进行同样操作（Settings → Editor → Auto-Import）：



当然，如果你希望关闭导入助手，直接取消这一项的勾选即可（Settings → Editor → Auto-Import）。

3、快速导入

当导入助手关闭时，不必惊慌。此时 PyCharm 不会直接给出提示，但会以红色波浪线标记缺失模块的代码位置，同时在左侧显示一个红色灯泡，单击这个灯泡，或者按下 Alt+Enter 快捷键：



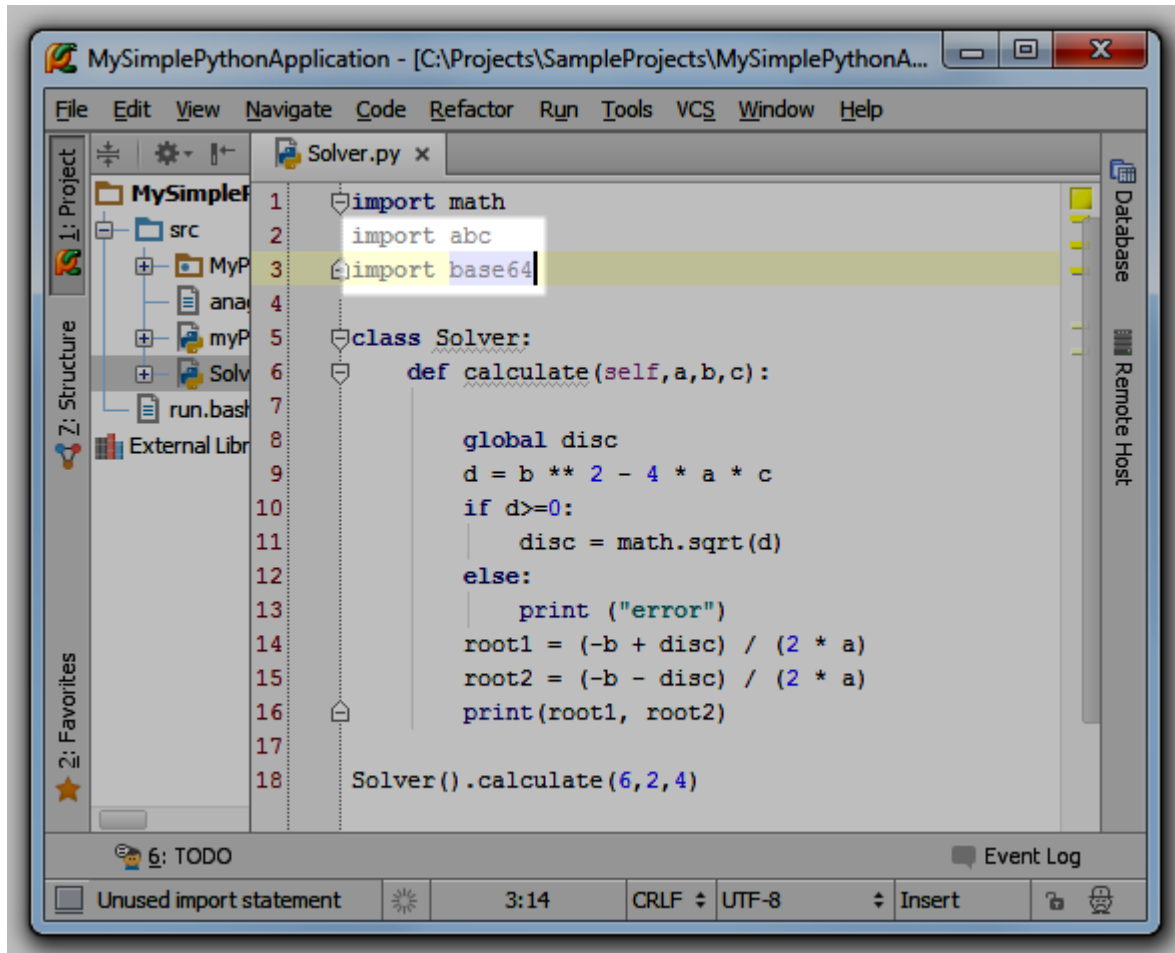
在我们这个例子中应选择导入对应的缺省库，导入完成，红色波浪线消失。

4、导入助手的优化

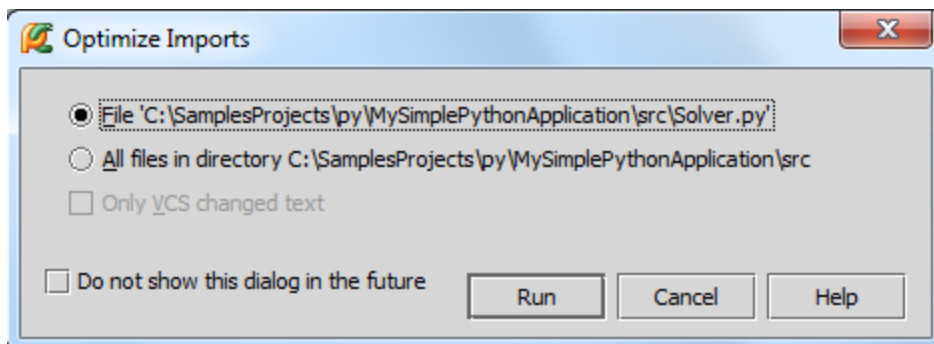
当你在完善代码的过程中，总会停止使用一些导入声明（例如调试代码所对应的库，在调试完成后就不再起作用）。然而这些 import 声明仍然存在于你的工程中，你不得不停下来从头搜索，找出并删除这些声明语句，这种做法不仅效率低而且容易出错，很可能会多删或者漏删。

Pycharm 能够帮助我们处理掉这些冗余的 import 声明语句，也就是所谓的 **Optimize Imports** 功能，这个功能能够帮助你随时删除工程中的冗余 import 声明语句。

注意到，在 Pycharm 编辑环境中那些冗余的 imports 语句都是灰色显示的：



为了移除这些冗余语句，按下 Ctrl+Alt+O（或者选择 Code → Optimize Imports 菜单命令），Pycharm 弹出如下对话框，提示你选择需要清理的文件（当前文件还是当前目录下的所有文件）：



单击 OK，清理完成：

```
MySimplePythonApplication - [C
File Edit View Navigate Code
?
1 import math
2
3
4 class Solver:
```


最全 Pycharm 教程（18）——Pycharm 编辑器功能之代码拼写提示

1、主题

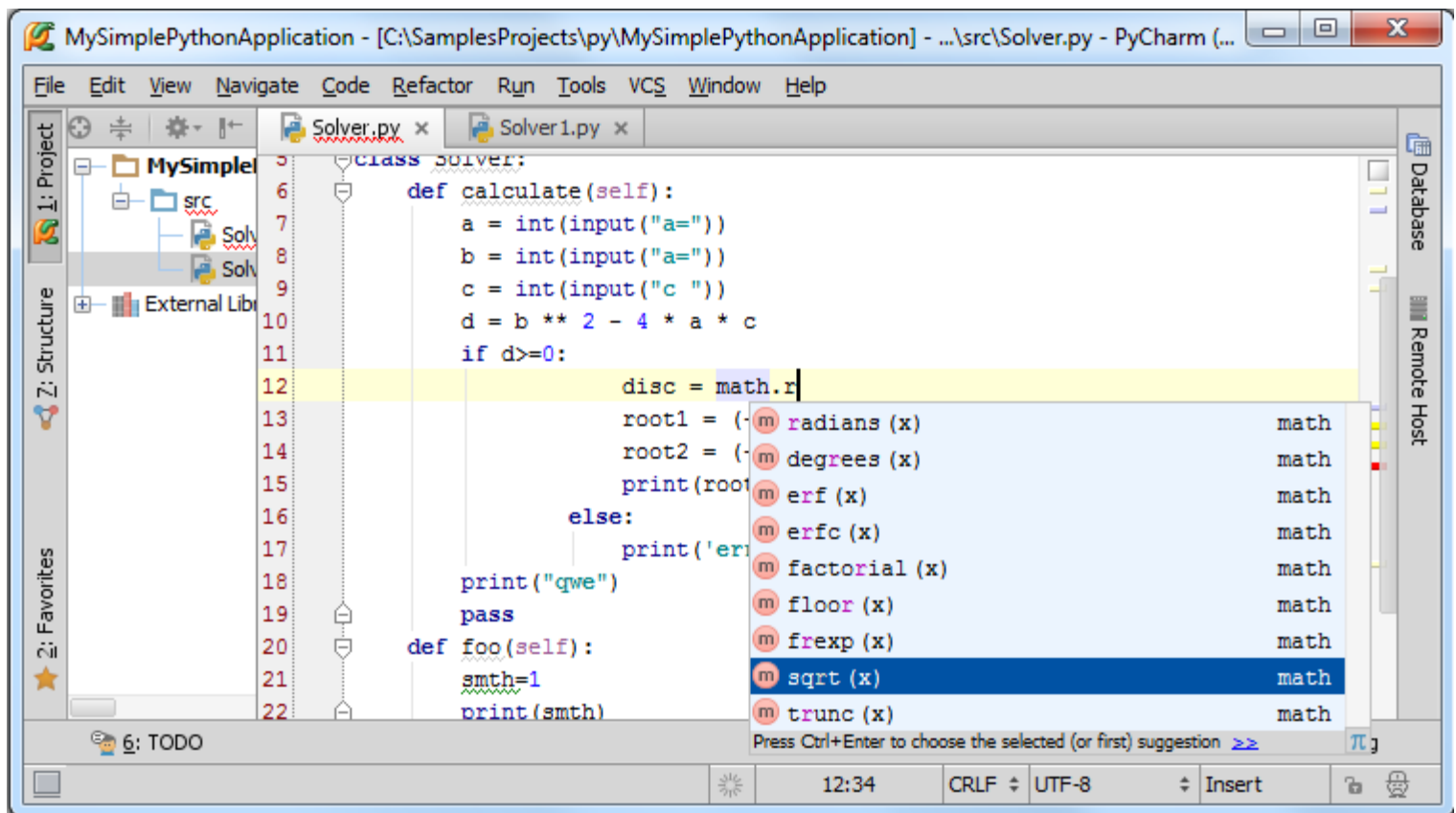
之前我们已经在 [Getting Started Guide](#) 接触了代码拼写提示这一帮助功能，这里我们将详细的介绍它。

拼写提示就是以列表的形式给出当前位置下可用的单词（函数名、类名、变量名等）。

2、启用拼写提示功能

在任何情况下我们都可以通过 `Ctrl+Space` 快捷键来启动拼写提示功能。

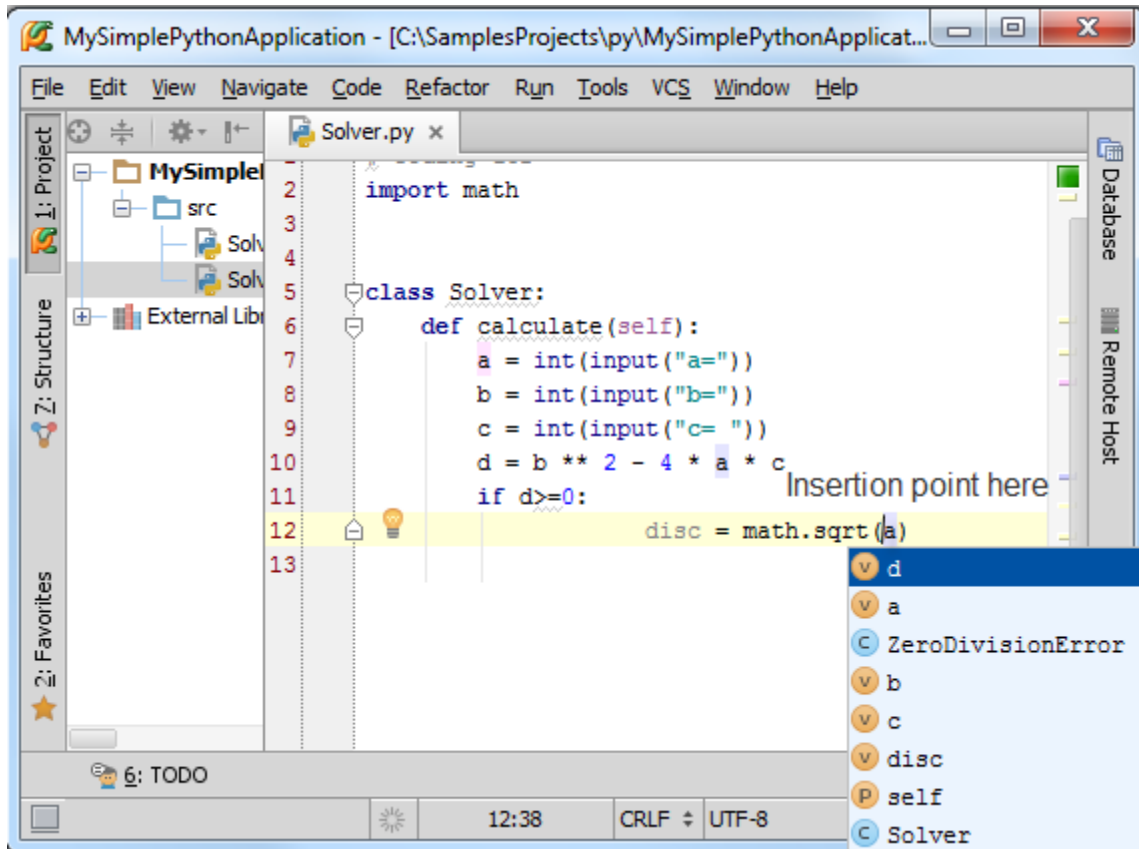
事实上，拼写提示功能在你输入代码的过程中是默认启动的，当然也包括输入点号之后的域成员提示功能。当你继续输入时，拼写提示列表会缩小范围以匹配你输入的字符。由此可将，拼写提示功能也是相对智能的，会在任意位置匹配当前已输入的字符串以供用户选择：



当然如果你想关闭拼写提示功能，可以通过 `Ctrl+Alt+S` → `Settings` → `Editor` → `Code Completion` 命令打开拼写提示功能对话框，取消 **Autopop code completion in (ms)** 复选框的勾选即可。

3、如何使用拼写提示列表

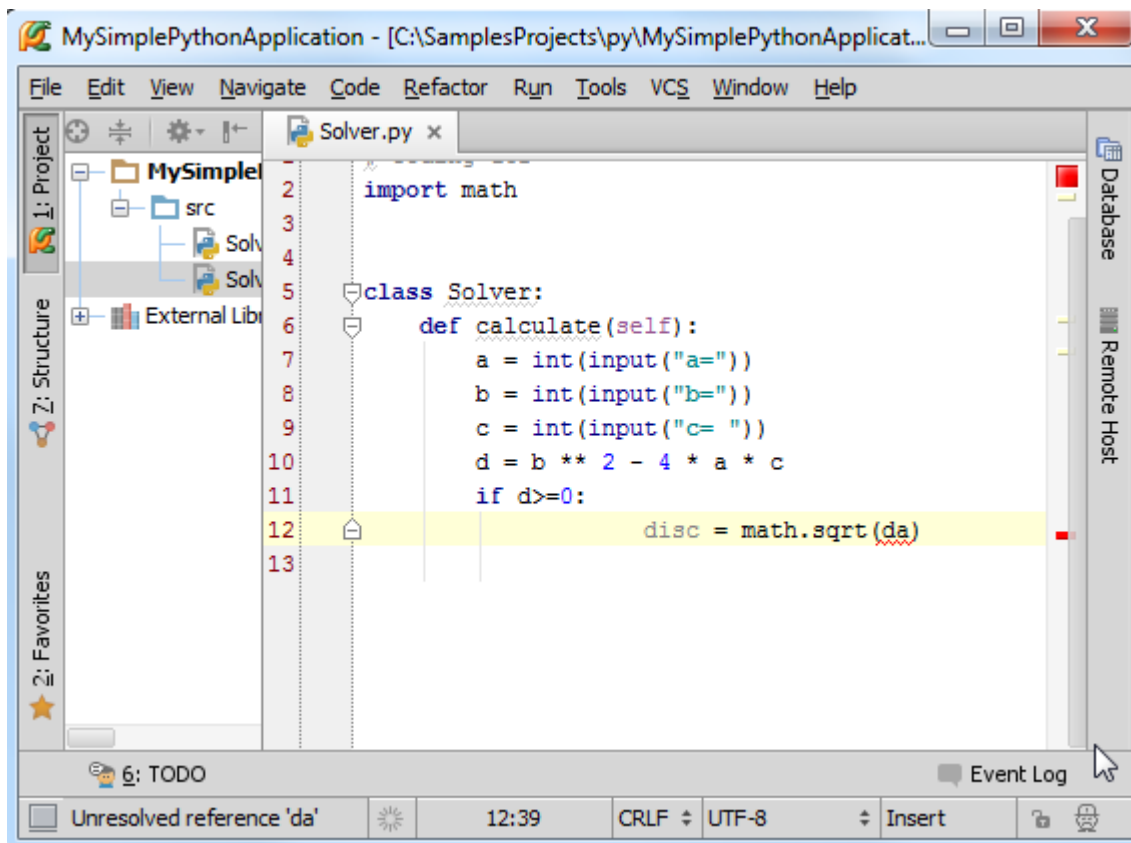
使用鼠标指针或者 `up/down` 键在拼写提示列表中选择期望输入的名称：



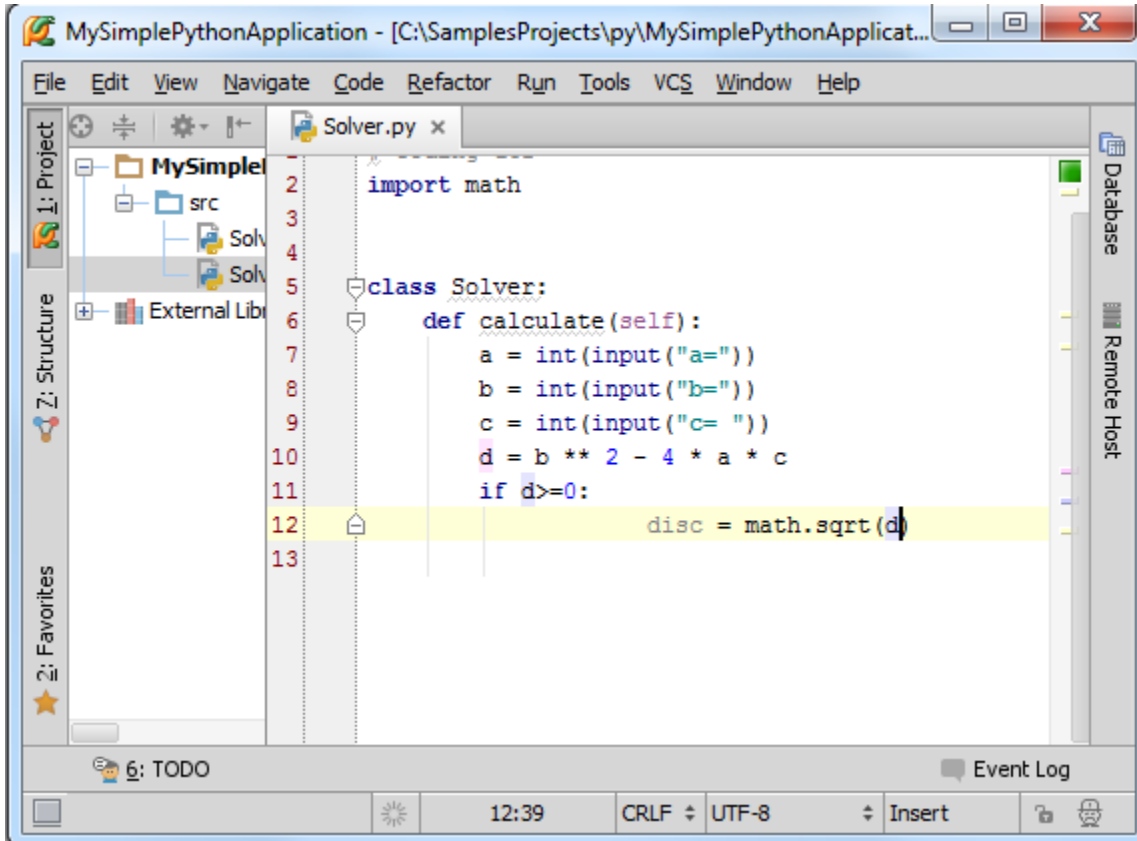
这里你可以通过 right/left 键来调整拼写列表的宽度。

然后你就需要将选中的名称插入到当前位置，Pycharm 提供了两种插入方法：

(1) 按下回车，对应变量会添加到当前光标的位置：

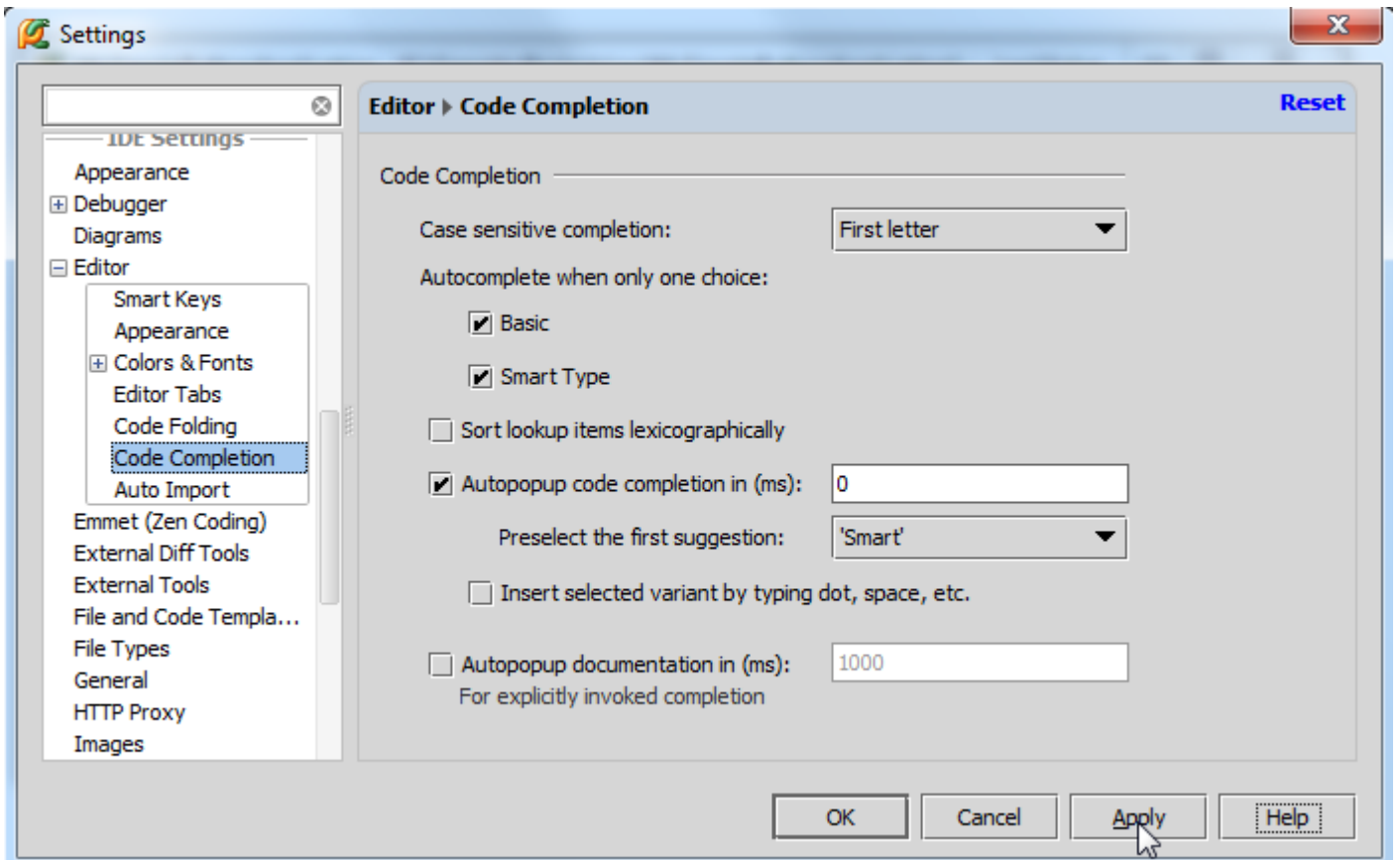


(2) 按下 Tap 键，选中的名称会替换掉当前光标右侧的字符串：



4、拼写提示功能更改

按下 Ctrl+Alt+S 打开设置对话框，展开 Editor 节点，单击 Code Completion 页：



如你所见，Pycharm 允许我们对拼写提示功能做各种各样的更改以满足需求，如果你不清楚某些选项的具体功能，单击 Help 按钮获取帮助信息。




更多拼写提示功能相关信息参见 [this link](#)。

最全 Pycharm 教程（19）——Pycharm 编辑器功能之代码折叠

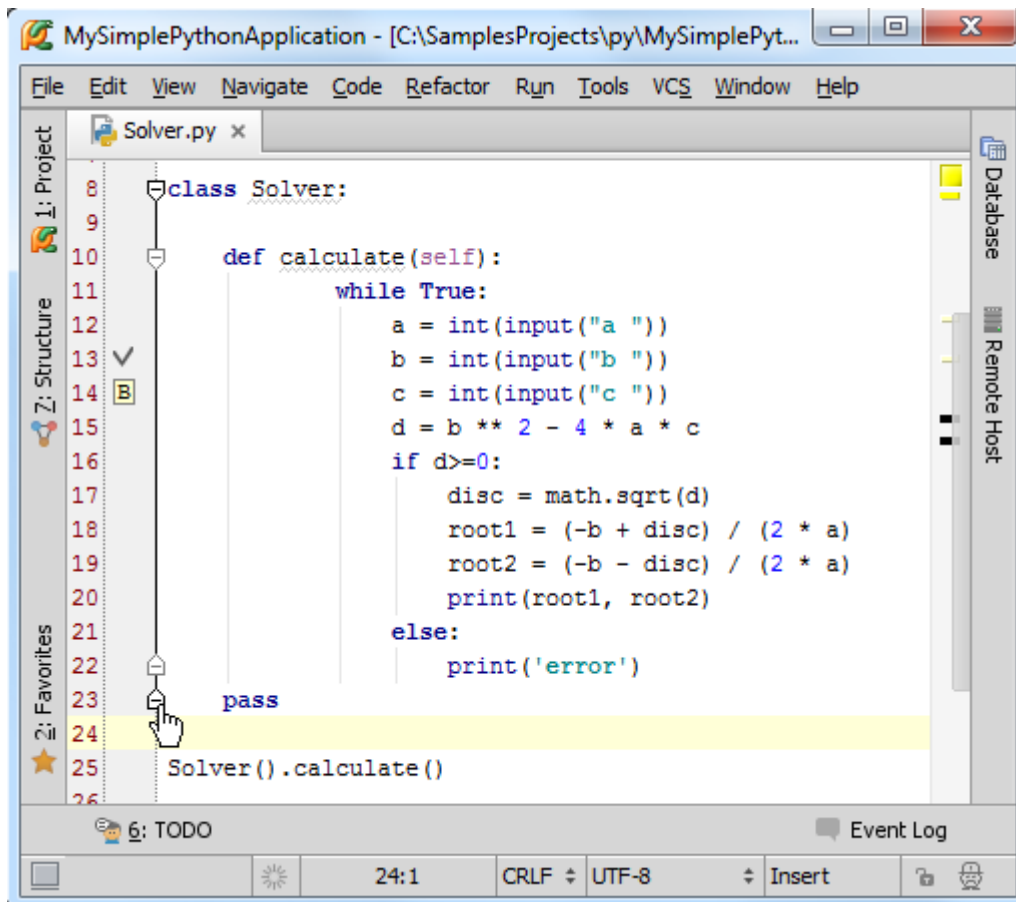
1、主题

在一些情况下，如果某些代码显得不太重要，我们可以通过 Pycharm 的代码折叠功能将其折叠为一行。在接下来的部分我们将介绍代码折叠功能基本用法。

2、代码可折叠轮廓线以及折叠开关

首先，我们观察一下代码左侧的折叠线。这条细线显示在代码左侧，标记了代码块区域。当代码处于未折叠状态时，线的开头和结尾分别显示折叠开关  和 ；当代码块折叠之后，两个开关标志将会合并成一个 ，折叠后的代码只显示第一行，其他行隐藏在三个点号后面。单击这个加号的标志即可将折叠代码展开。

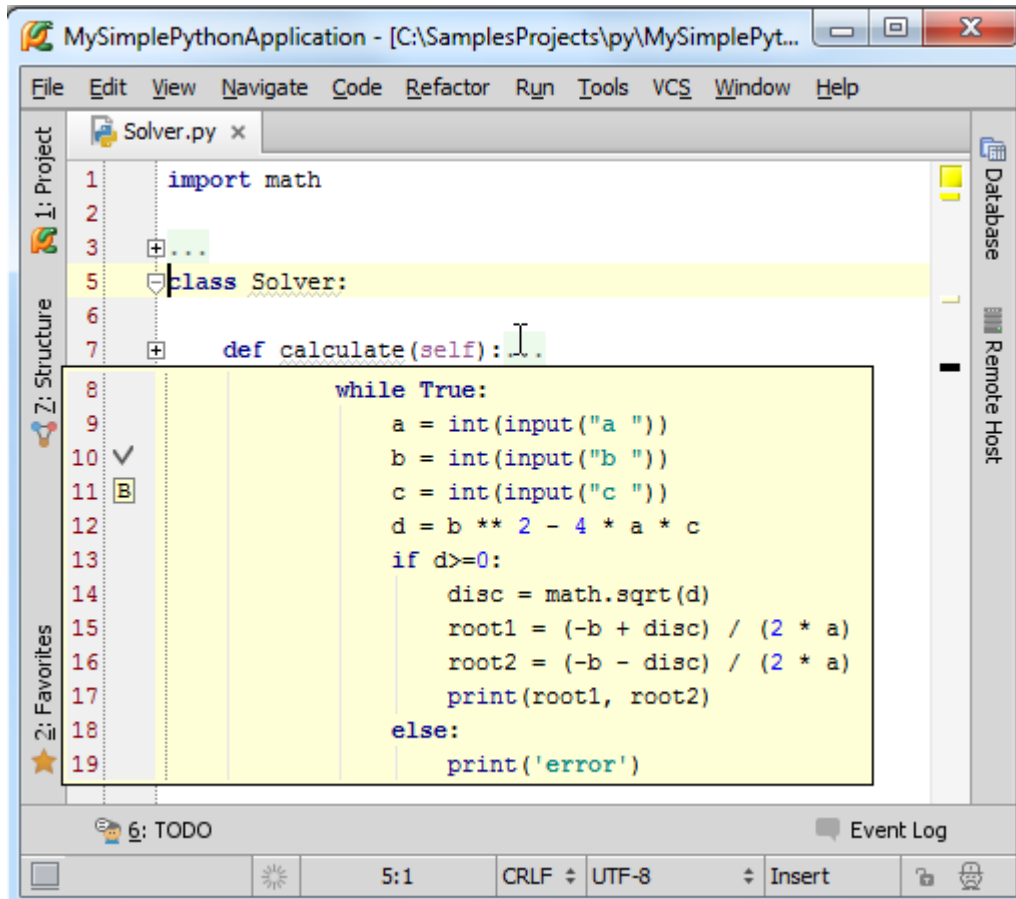
将鼠标指针悬停在折叠线上，折叠线会加粗显示：




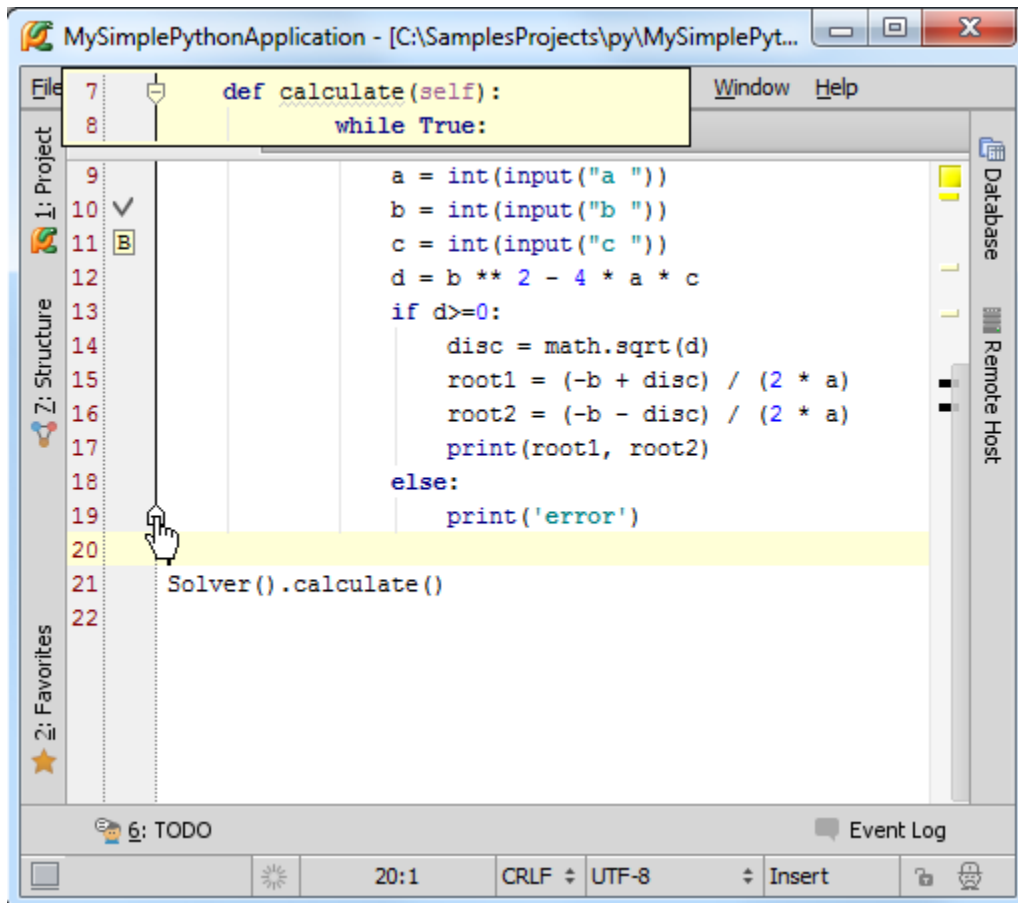
注意在这里我们可以取消折叠线的显示。单击设置对话框，在 **Editor** 节点下，单击 **Code Folding** 页面，取消 **Show code folding outline** 复选框的勾选。当折叠线不可见时，我们只能通过菜单命令（**Code | Folding | Expand/Collapse**）或者快捷键来实现代码的折叠。

3、浏览折叠后隐藏的代码

加入你希望能够在不展开代码块的情况下查看折叠代码的内容，操作非常简单，只需将鼠标指针悬停在三个点号上，Pycharm 就会弹出一个临时窗口来显示折叠的代码内容：



Pycharm 也会默认将超出当前编辑区域的代码块显示完整。如下图所示：文件开头的代码部分在当前编辑环境下不可见（超出当前编辑框的显示范围），不过我们不必拖动滚动条来浏览那部分未显示的内容，只需将鼠标悬停在代码块结束标记 ，Pycharm 会自动弹出窗口来补全显示当前的代码块：



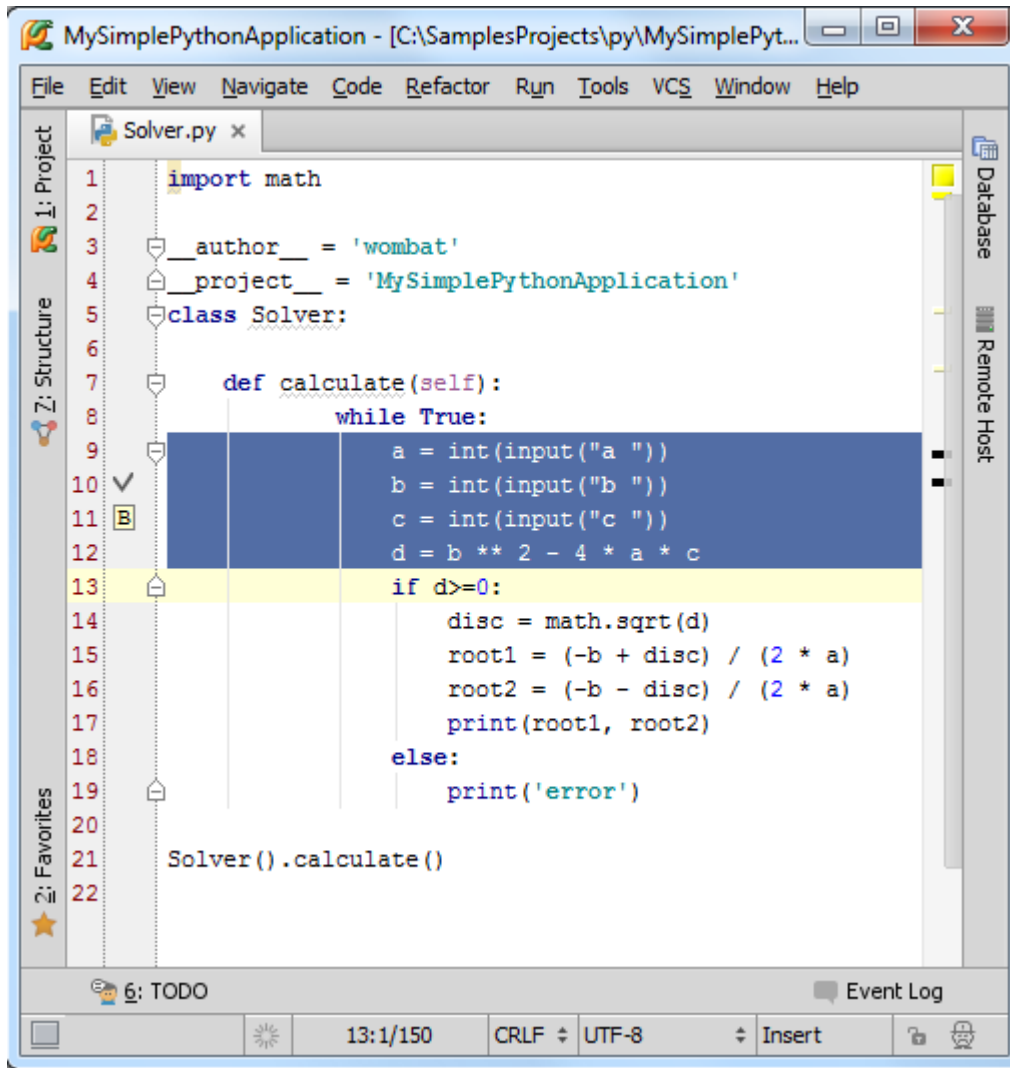
4、默认代码块的折叠规则

默认情况下折叠先会标记类和函数的实现部分，折叠也是针对这部分代码块进行的，即默认折叠一个类、一个函数。

5、折叠任意代码片

假设，你希望折叠几句零散的程序，而这些语句并不属于默认可折叠的代码块（不是一个完整的类或函数），如何做到？

首先选中你希望折叠的代码片

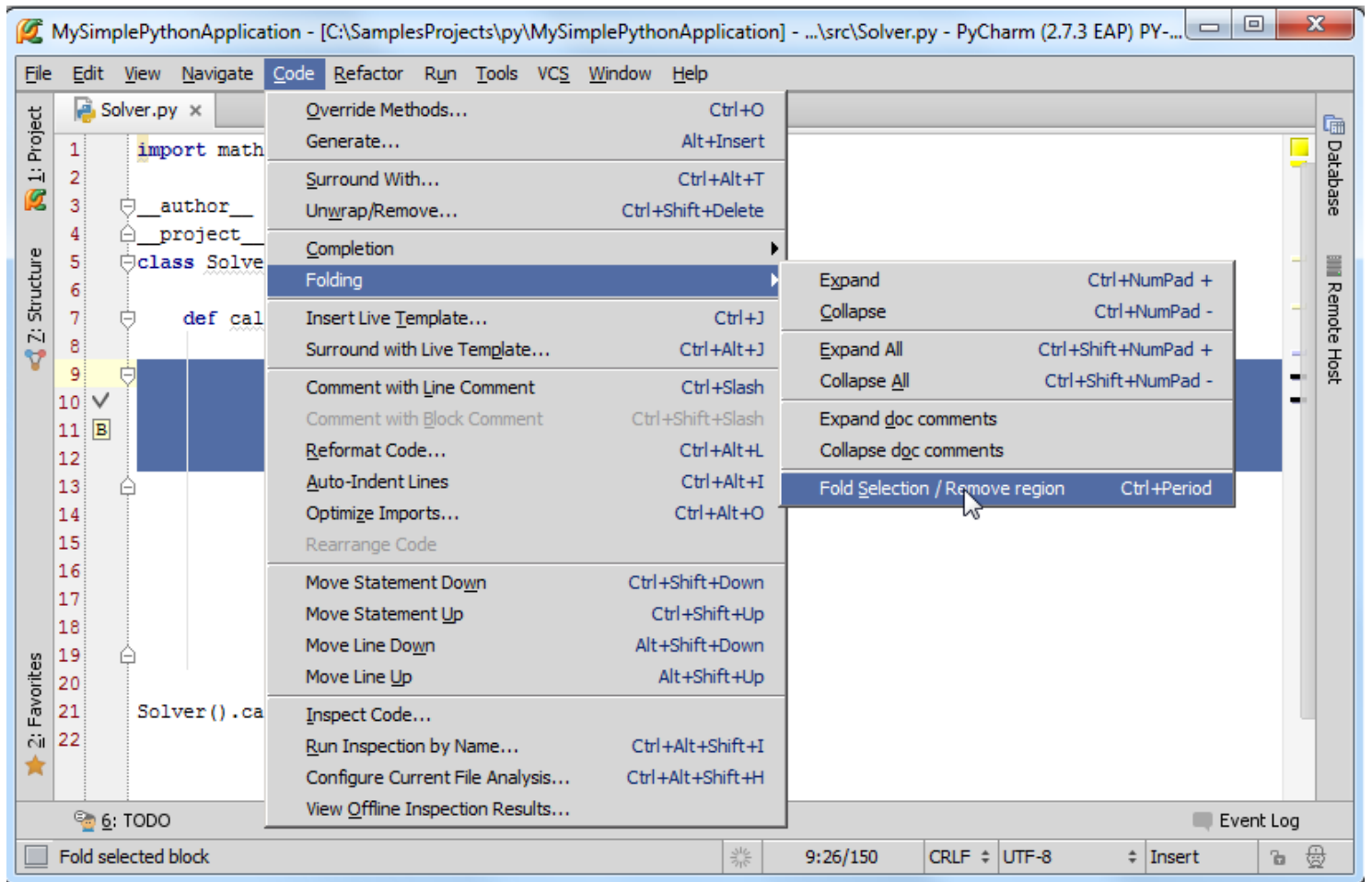


然后进行以下操作（三选一即可）：

在主菜单选择 **Code | Folding | Fold Selection/Remove Region** 菜单命令。

右击选中的代码片，在快捷菜单中选择 **Folding | Fold Selection/Remove Region**

按下 **Ctrl+Period** 快捷键



此时选中的代码片被折叠隐藏。

需要注意的就是所选代码片应该位于类体或者函数体内，如果我们选择了类或函数开头的定义部分，是无法对这部分代码进行折叠隐藏的。

6、使用双行注释来注释代码

Pycharm 提供了两种逻辑代码块环绕注释方式 `surround`：

VisualStudio 模式：

```

#region Description
Your code goes here...
#endregion

```

NetBeans 模式：

```

// <editor-fold desc="Description">
Your code goes here...
// </editor-fold>

```

具体操作如下：

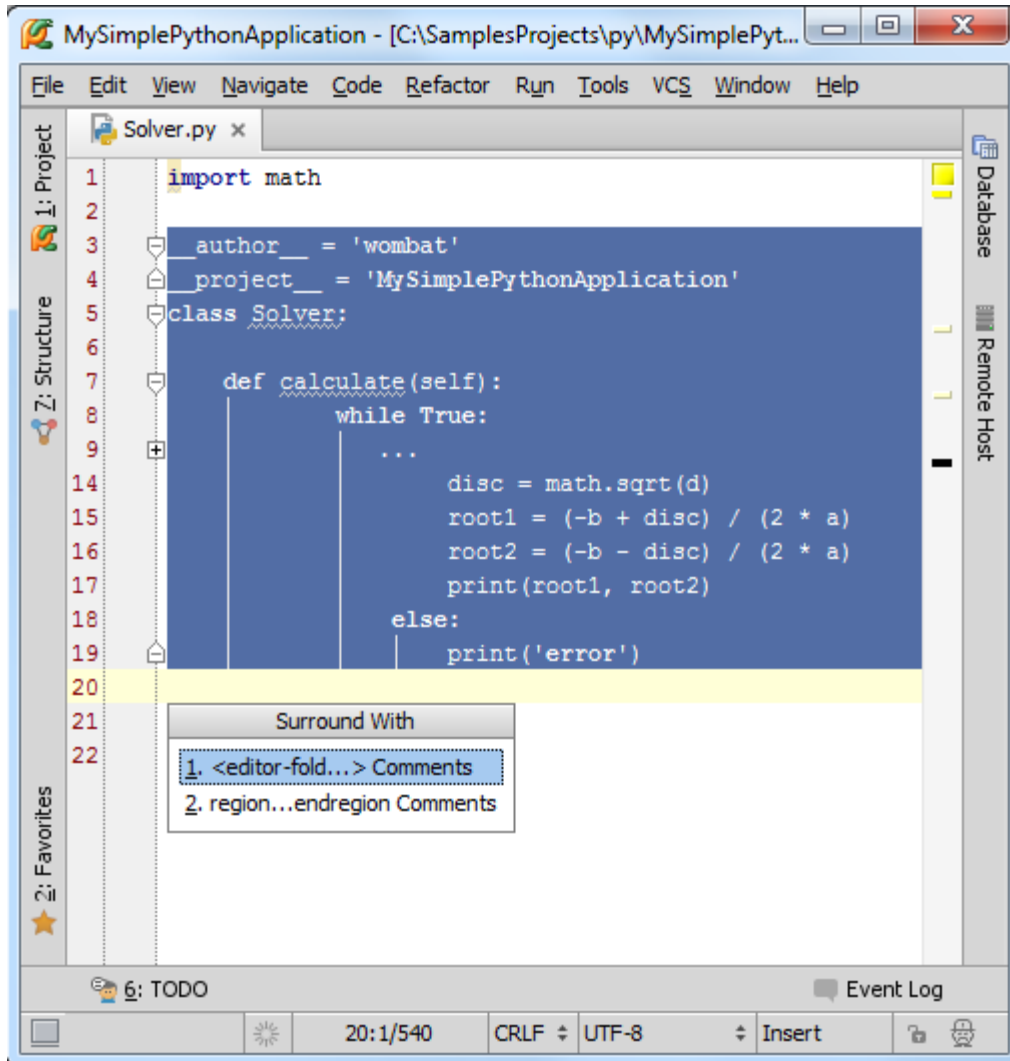
- (1) 选中待环绕注释的代码块

(2) 以下操作二选一

使用 **Code | Surround with** 主菜单命令

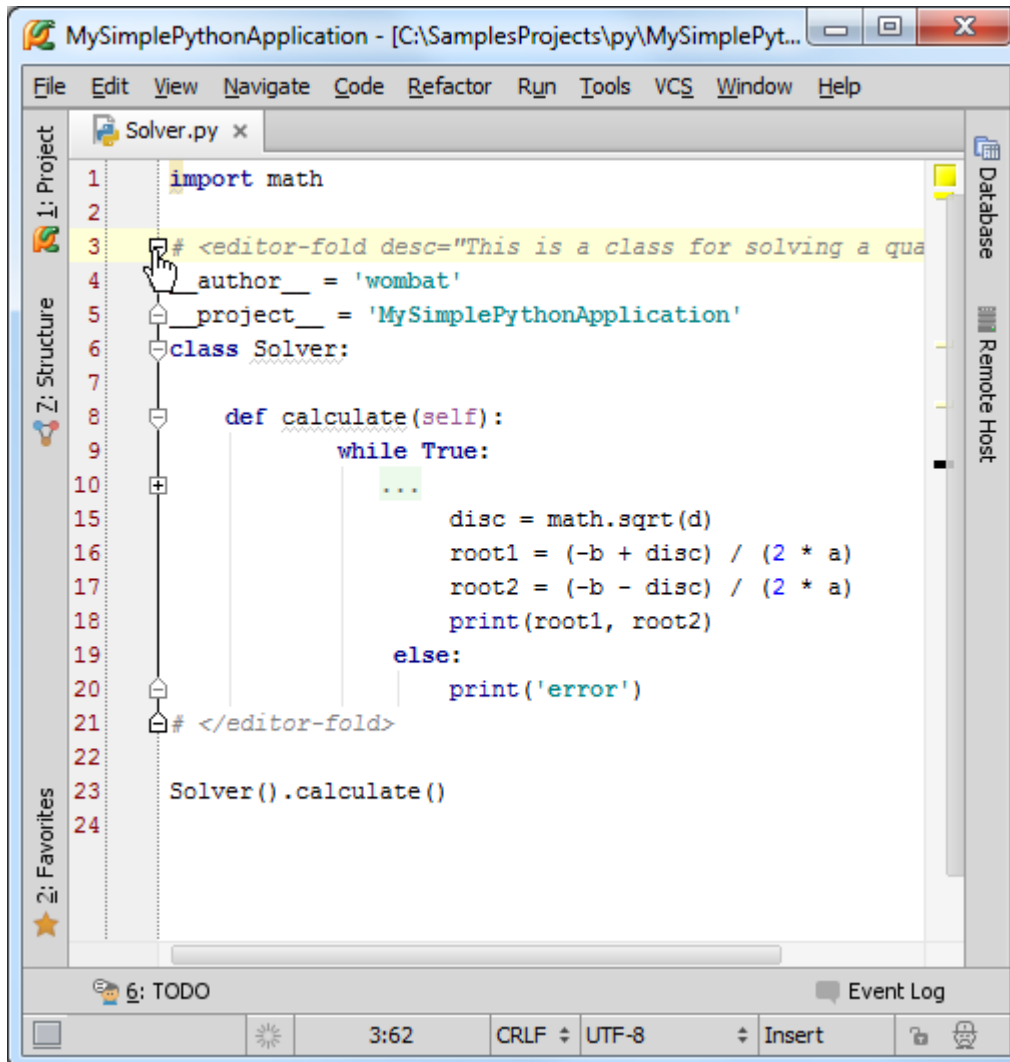
按下 **Ctrl+Alt+T**

(3) 在弹出的快捷菜单中选择需要的注释风格:

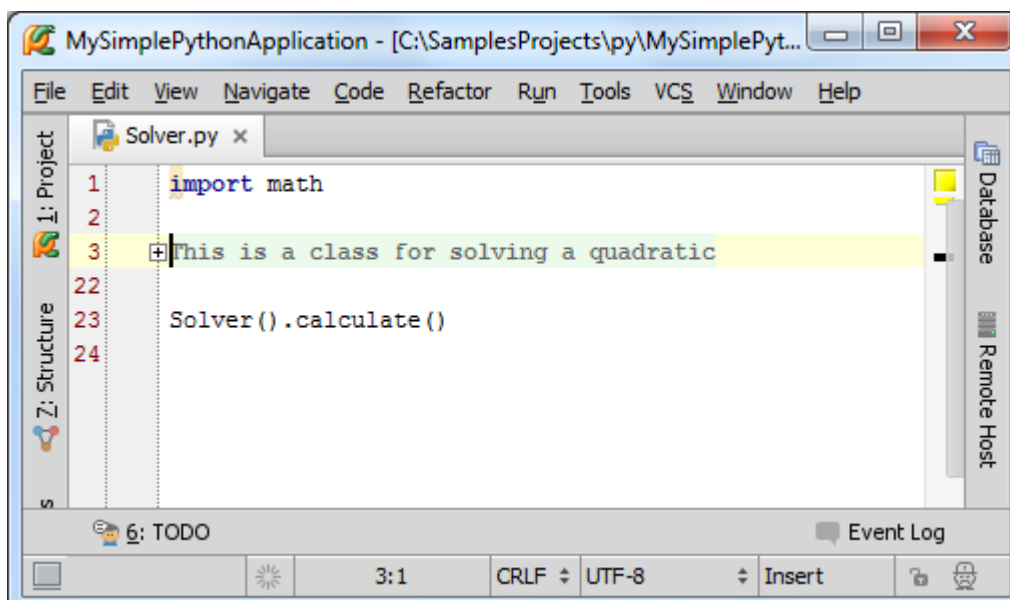


(4) 输入一些必要的注释描述

当通过这种方法对代码片进行环绕注释之后, 折叠开关会显示在注释行左侧:



单击折叠开关可折叠对应代码片，只显示添加的环绕注释信息：



更多有关代码折叠功能的信息参见 [here](#)。

最全 Pycharm 教程（20）——Pycharm 编辑器功能之模板应用

Pycharm 自带了很多灵活的模板，但针对 python 本身的模板只有一个。这里我们将详细介绍如何针对 Python 类来创建模板并使用。

对于 Python 编程以及基本模板的使用我们这里不再赘述。关于模板的类型、缩写、变量名、以及存储的相关信息参见 [Live Templates](#)；模板的使用方法参见 [Creating Code Constructs by Live Templates](#)。

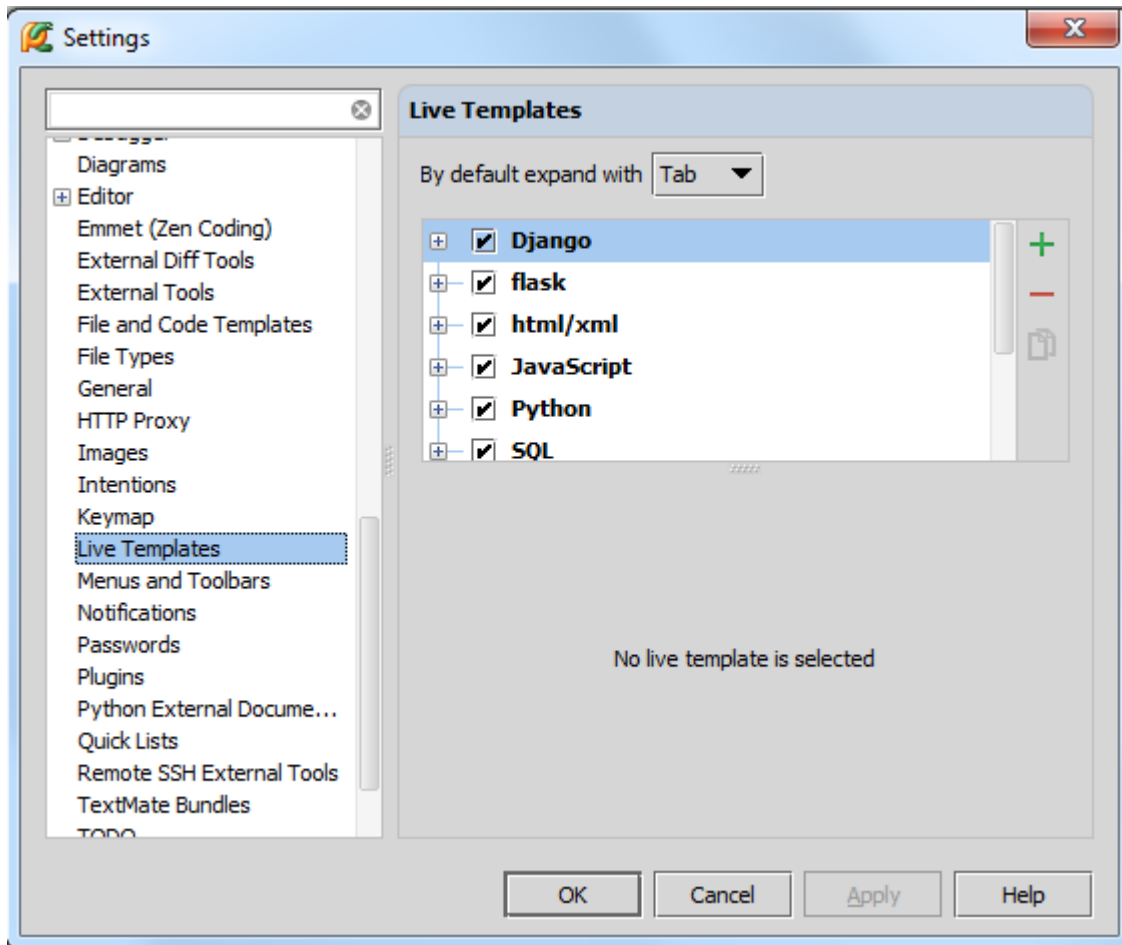
2、准备工作

Pycharm 版本为 2.7 或者更高，请大家自行安装。

3、模板的私人订制

4、创建一个根模板

打开设置对话框（单击工具栏的设置按钮，或者按 Ctrl+Alt+S 快捷键），在 IDE Settings 设置下单击 [Live Templates](#)：

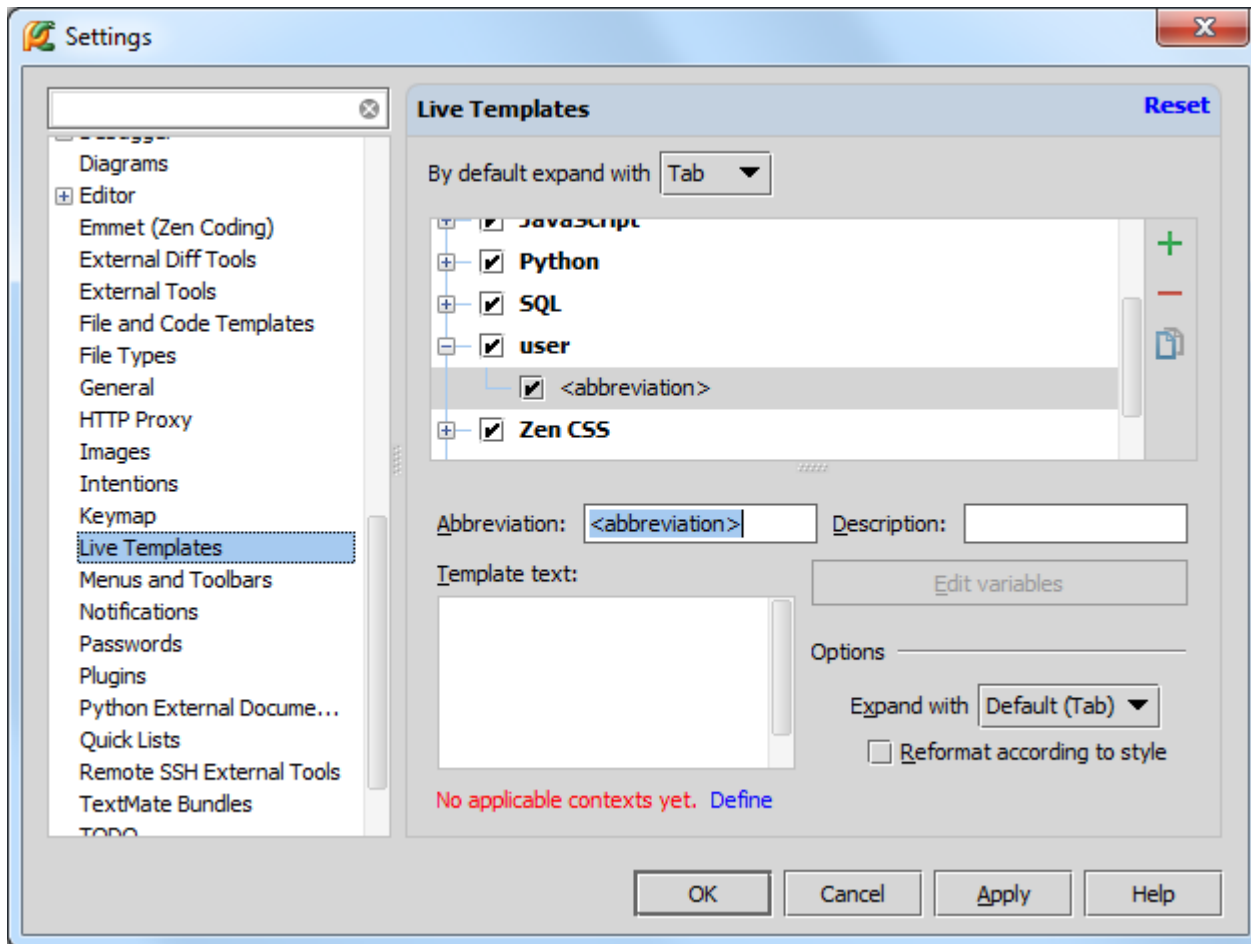


单击绿色加号，等待奇迹。

首先，注意到在 user 下面出现了一个新的分组。

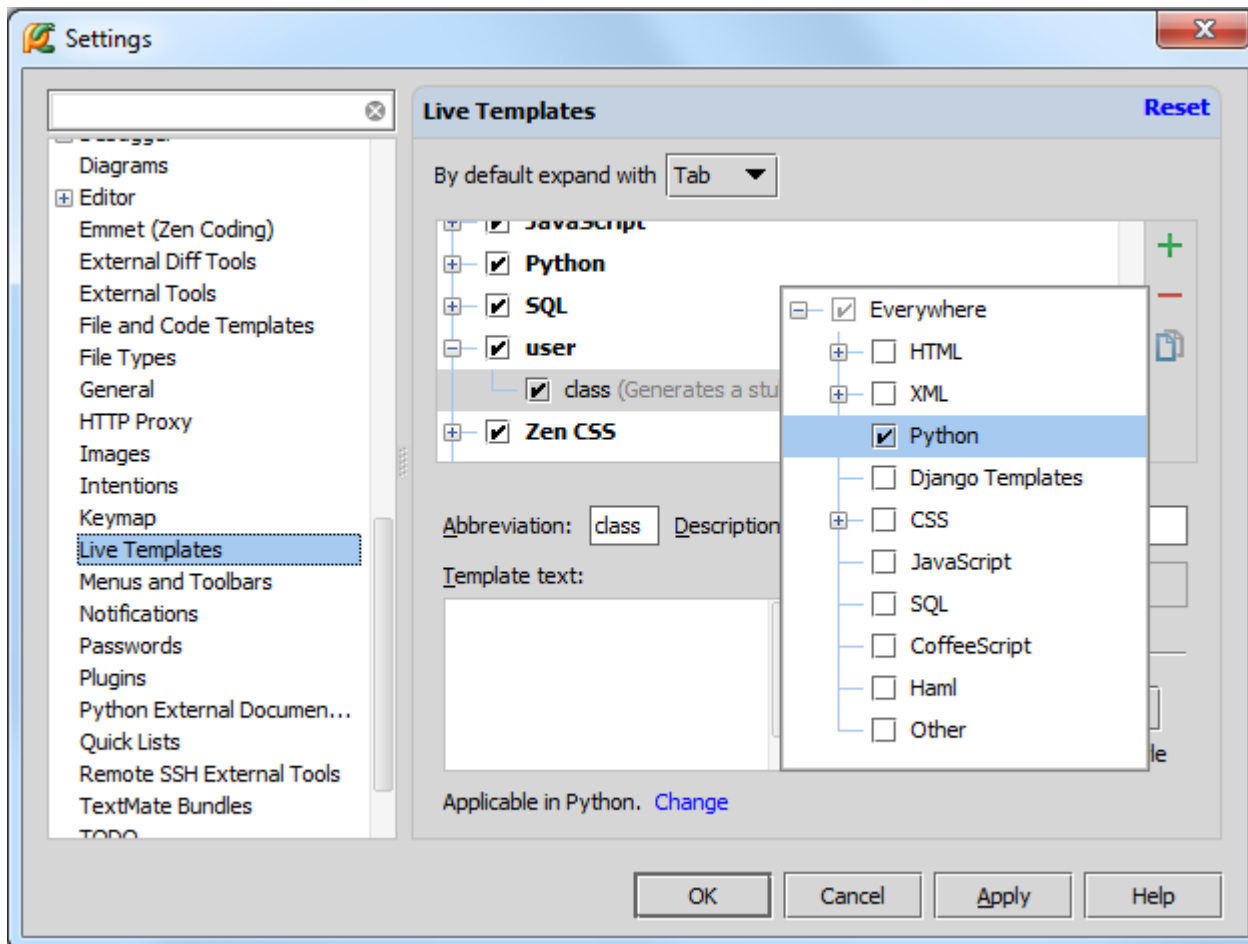
其次，选中 user 组后出现一个名为<abbreviation>的根模板

最后，界面上有缩写、描述说明、模板内容等输入窗口。



5、指定模板的缩写和上下文环境

第一步，输入模板缩写 [template abbreviation](#)，这里定义为 class。然后输入描述说明 description（可选），指定模板应用的上下文环境（这里选择 Python）：



最上方 expansion 值选择默认的 Tab 即可。

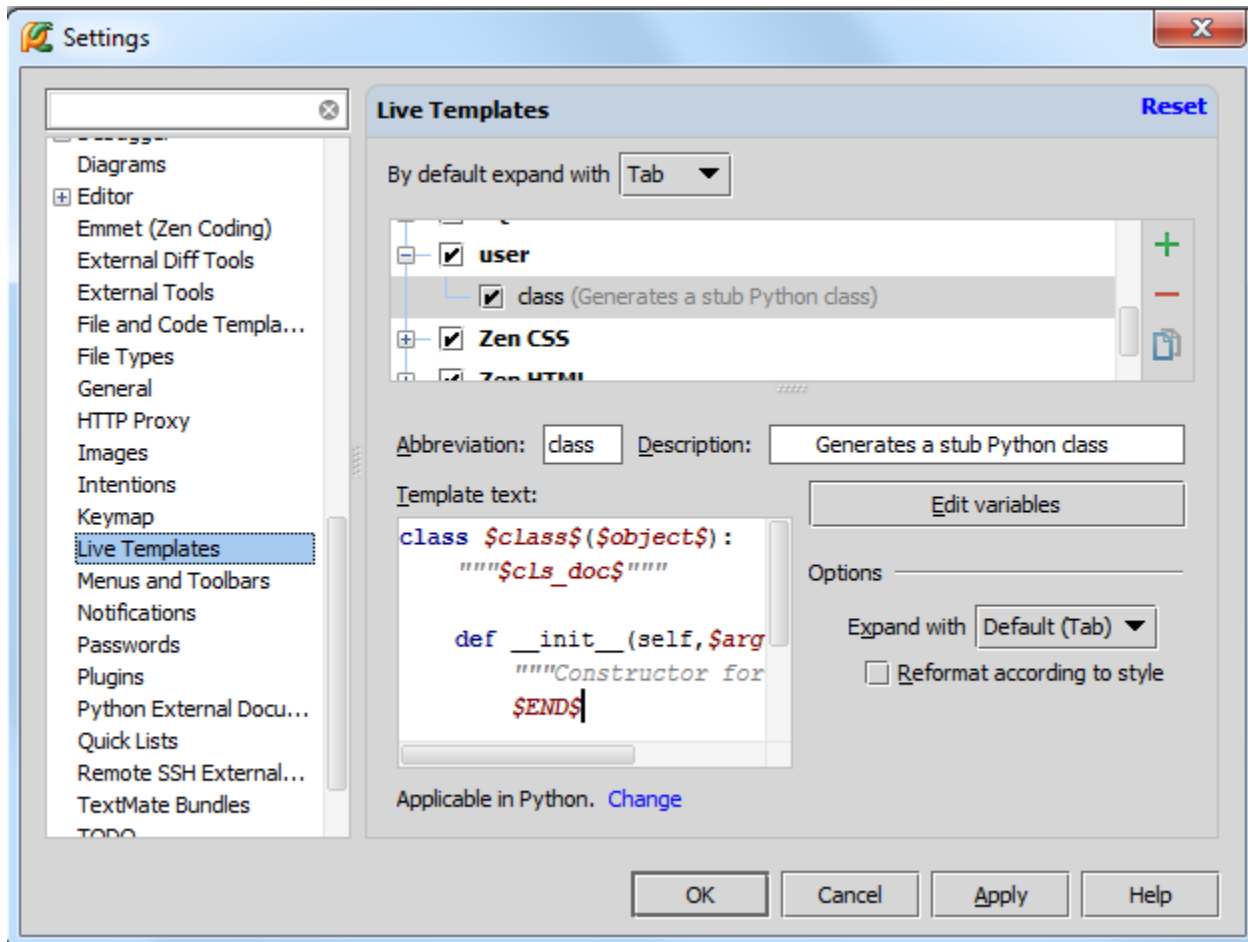
6、定义模板文本

在模板文本栏中输入以下代码：

```
class $class$($object$):
    """$cls_doc$"""

    def __init__(self,$args$):
        """Constructor for $class$"""
        $END$
```

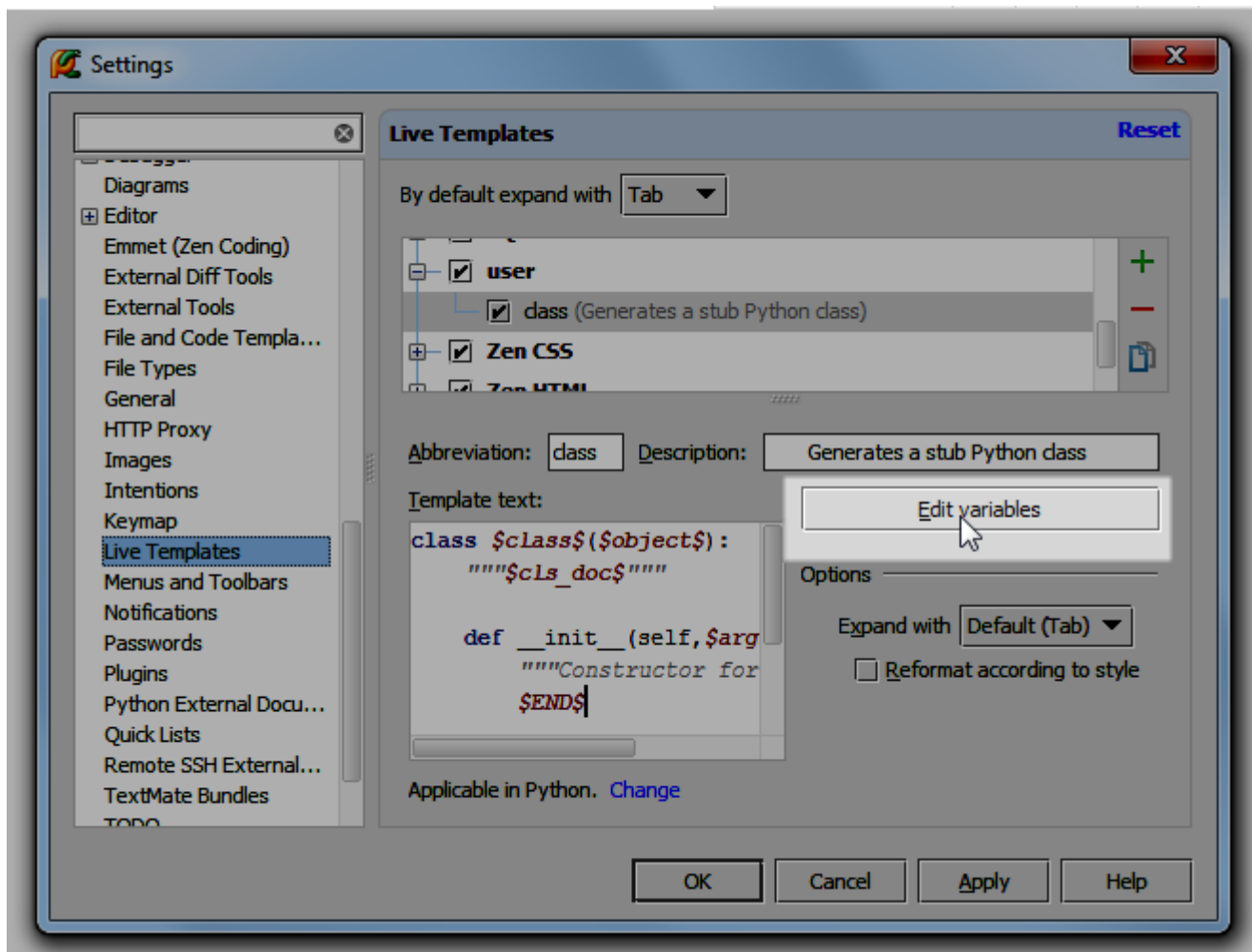
其中使用\$标记包围的部分为模板变量 [template variables](#)，Pycharm 将其标记为红色方便我们辨认：



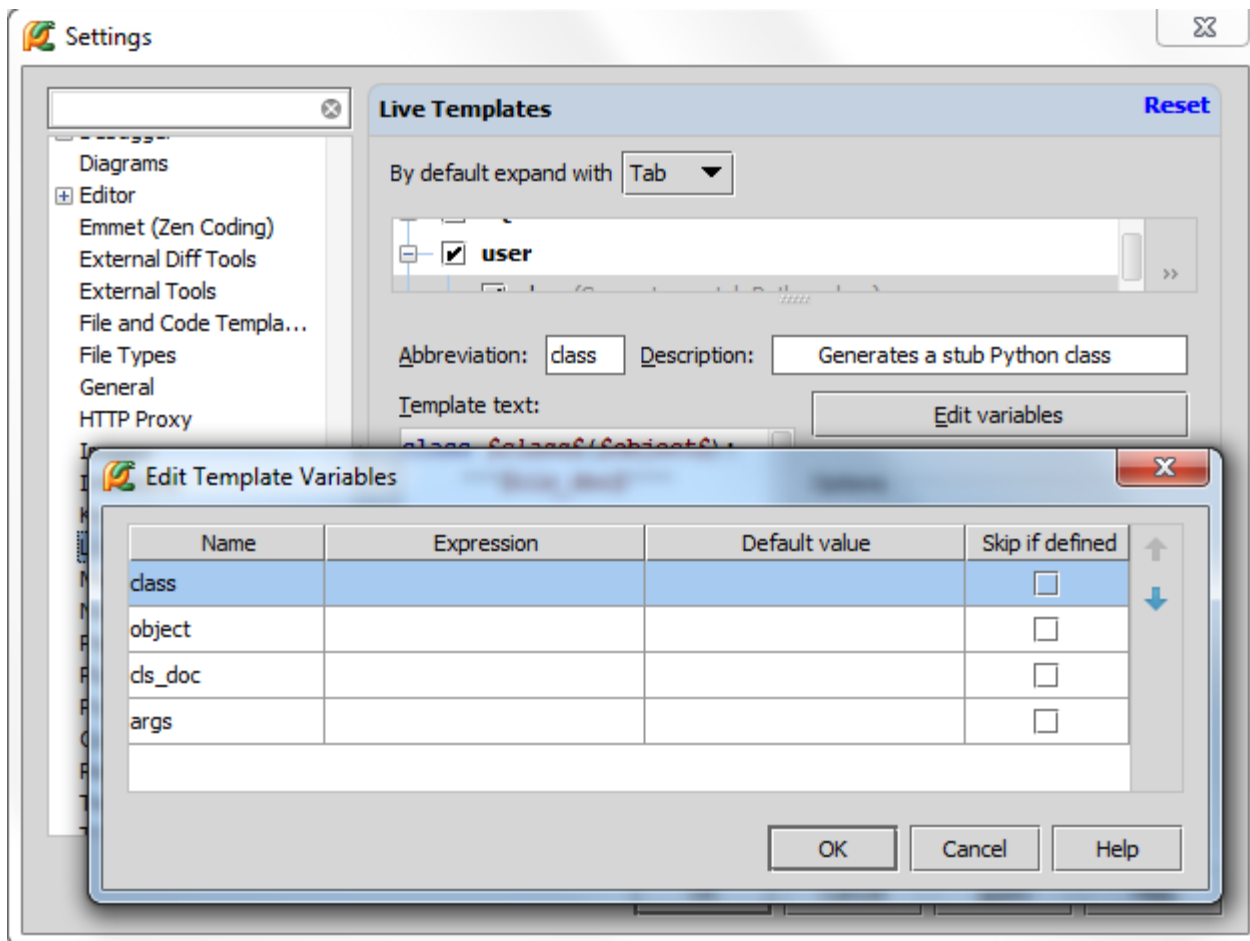
这些模板变量目前为空，接下来我们对其进行定义。

4、编辑模板变量

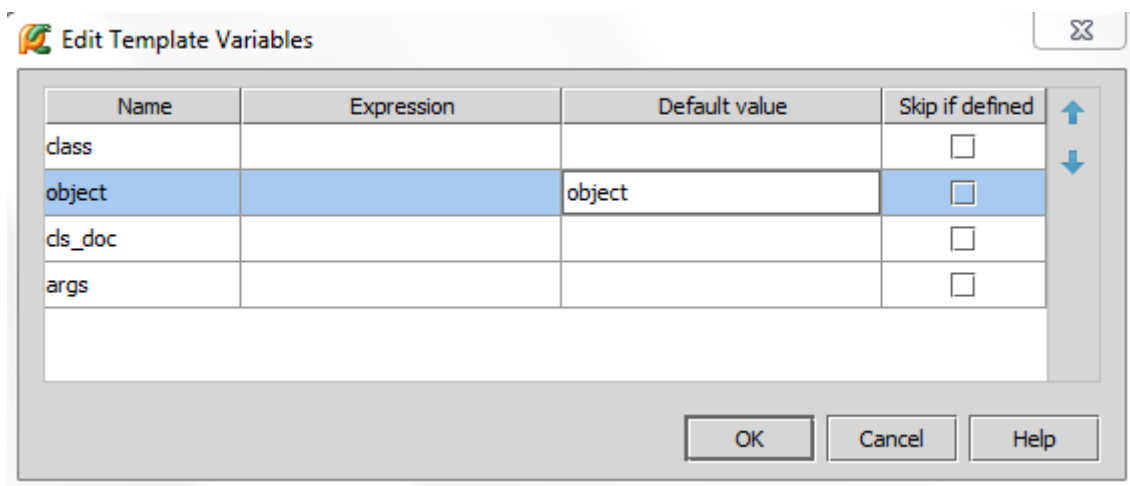
点击 **Edit variables** 按钮：



在 **Edit template variables** 对话框中显示了当前的模板变量列表：



对于变量\$object\$, 我们给出其缺省值 (object), 单击 OK 按钮:



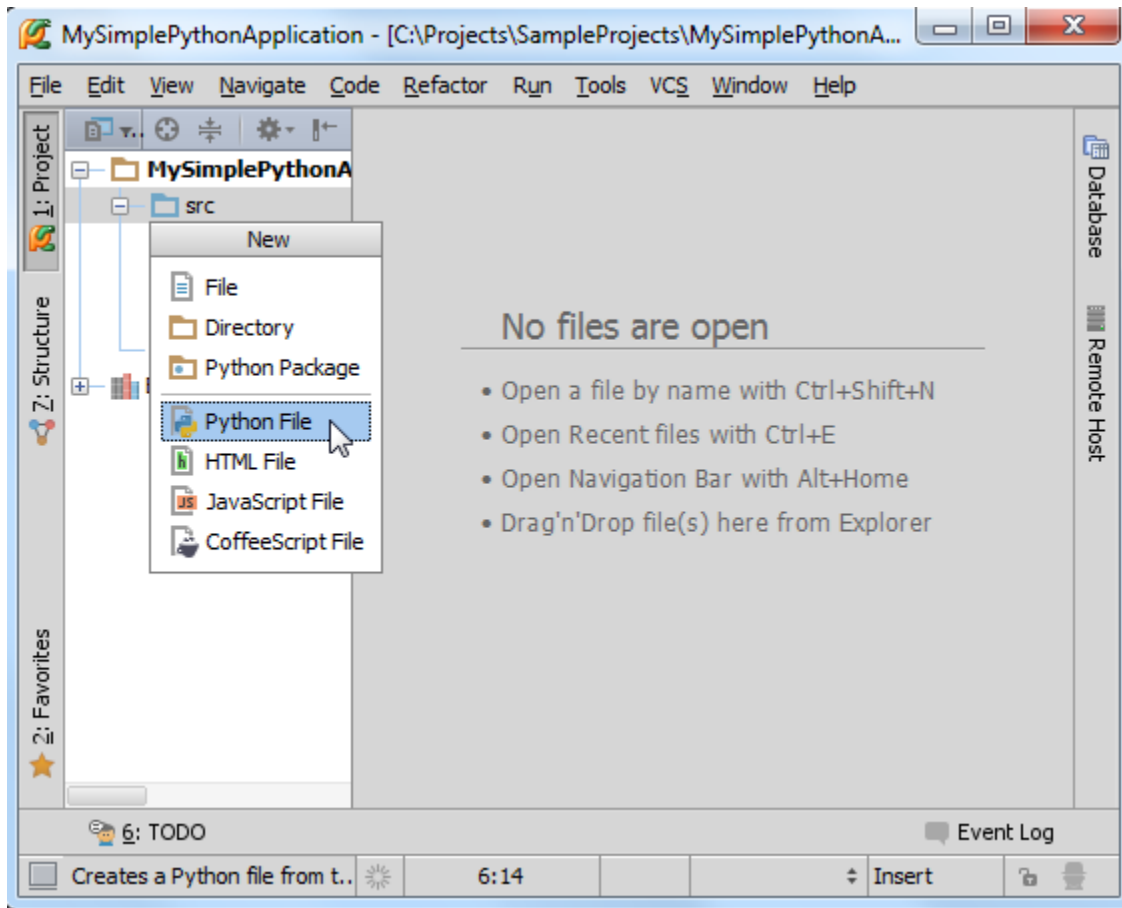
上面列表中所显示的变量并不包含 \$END\$, 可见 Pycharm 不希望我们对其进行更改。这是因为 \$END\$ 已经进行了预定义, 因此是不可编辑状态。它用来指示模板展开后输入光标的默认位置, 方便我们对模板对应的代码进行完善。我们这里这个光标会默认置于类声明之后。

5、保存自定义模板

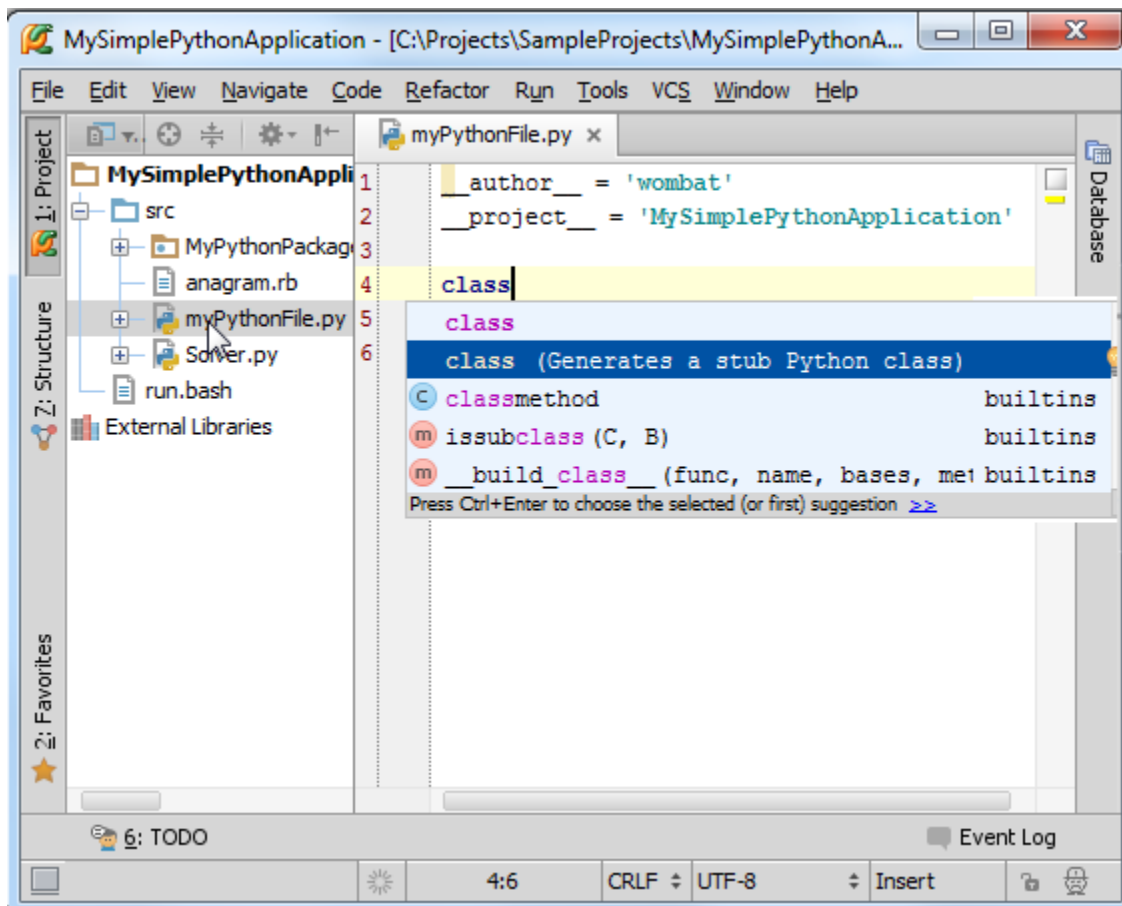
非常简单, 单击设置窗口中的 OK 按钮即可。

6、使用自定义模板

首先, 创建一个 Python 文件, 这里命名为 myPythonFile:

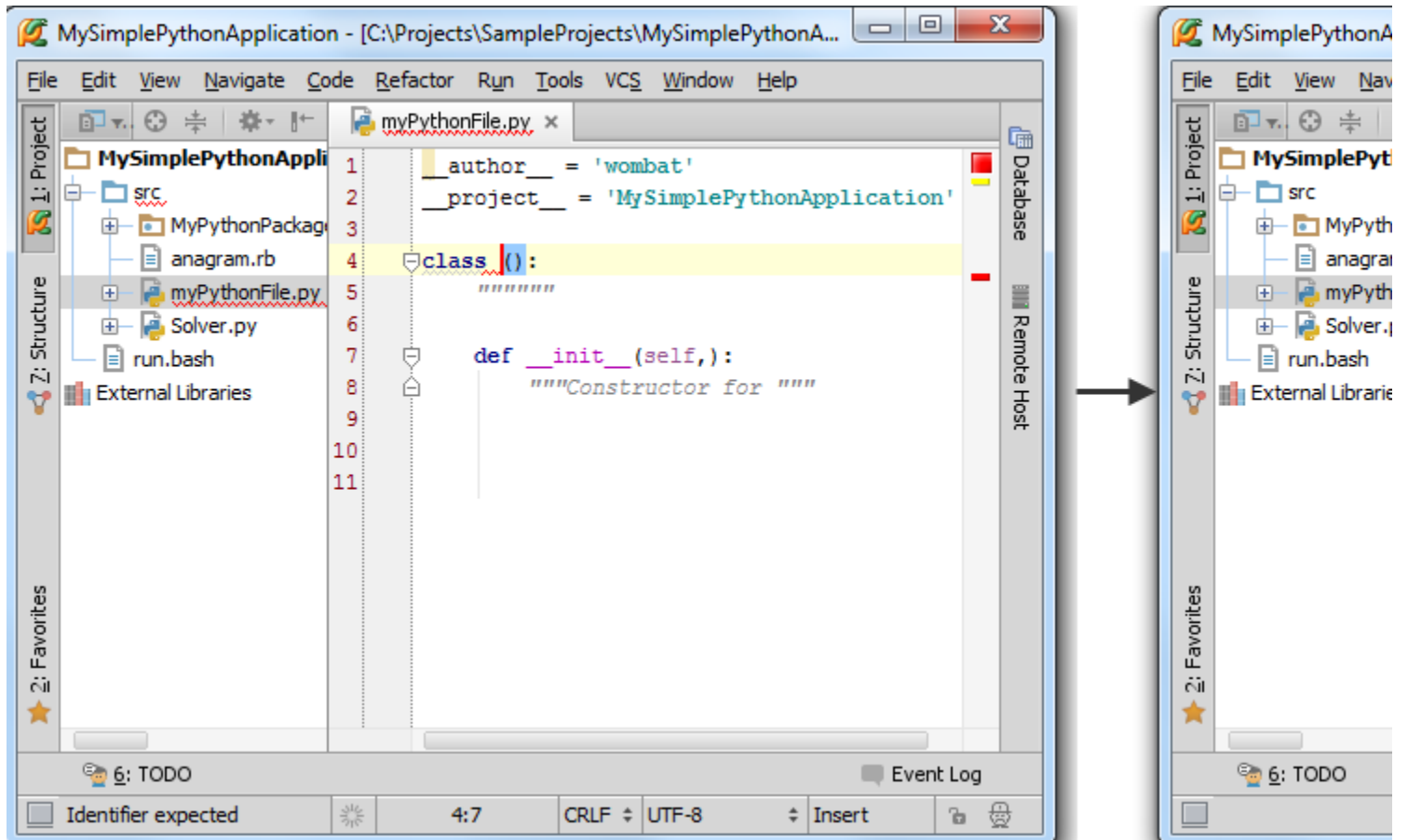


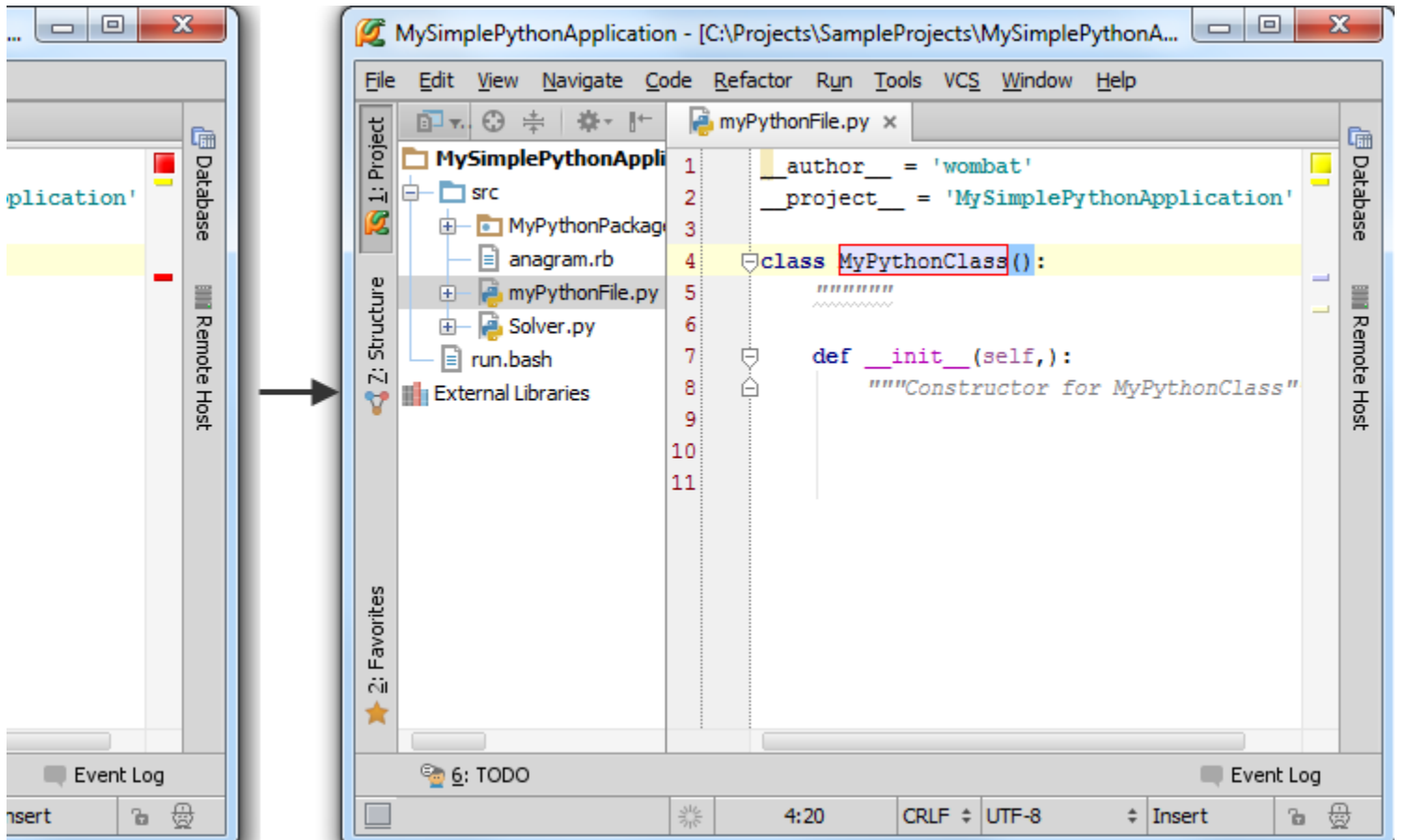
编辑这个新建的 Python 文件。接下来我们在其中创建一个类声明。输入模板缩写 `class`，会发现我们自定义的模板已经出现在提示列表中了，没错，就是我们刚才定义的那个：



按下 Tap 键选择该项。

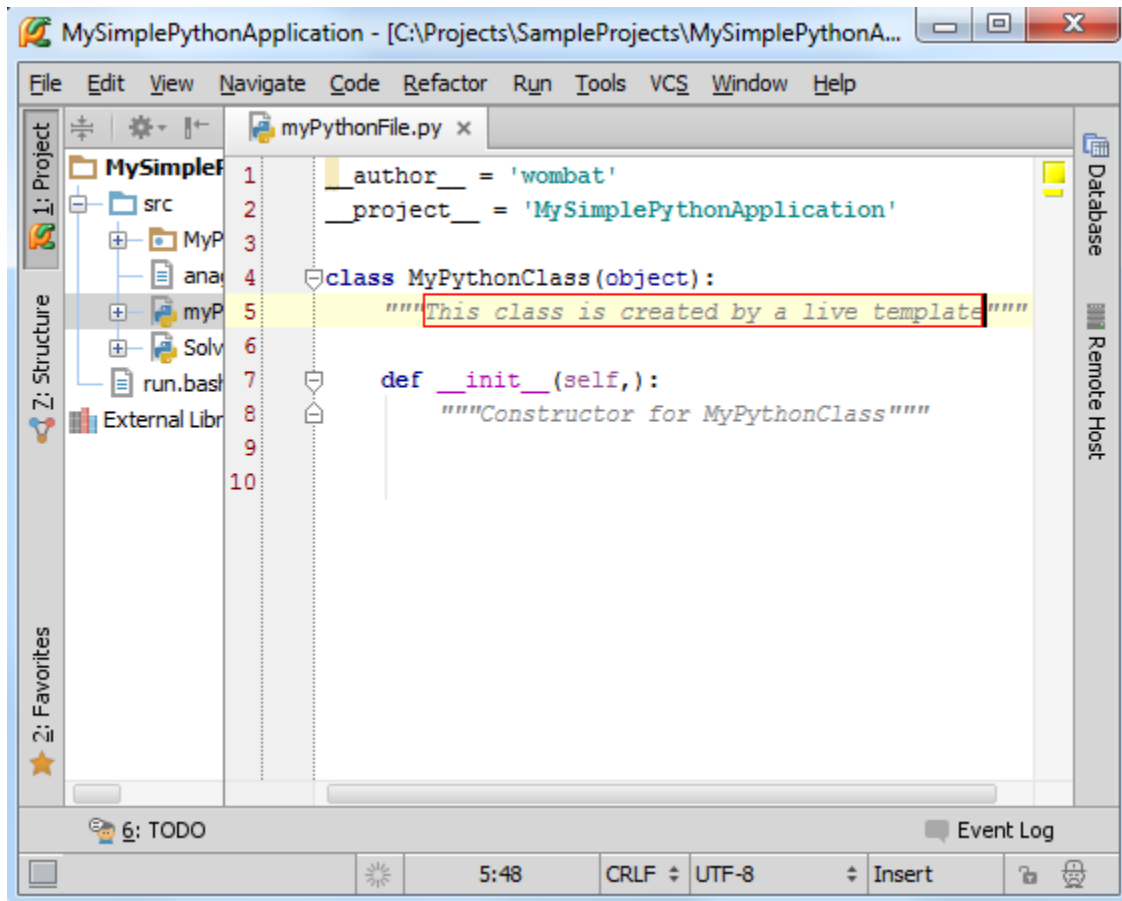
正如期望的那样，缩写名成功扩展成为了一个基本的 Python 类。红色下划线标记了接下来期望输入的位置，当你输入类名时（对应模板变量 class），就会插入到当前红色波浪线所在位置：





注意这里的模板变量 class 已经应用两次了：在类的声明语句和构造函数中。在构造函数中 Pycharm 对其进行了自动填充（填充为 self）。

输入类名，回车，红色波浪线移动到下一行，输入对应内容，最后回车：



最终，光标定位在类末尾。

更多自定义模板信息参见 [Creating Code Constructs by Live Templates](#)。

最全 Pycharm 教程（21）——Pycharm 编辑器功能之代码快速修改

1、主题

我们经常发现在程序中会弹出一个亮着的灯泡，它是用来干什么的？

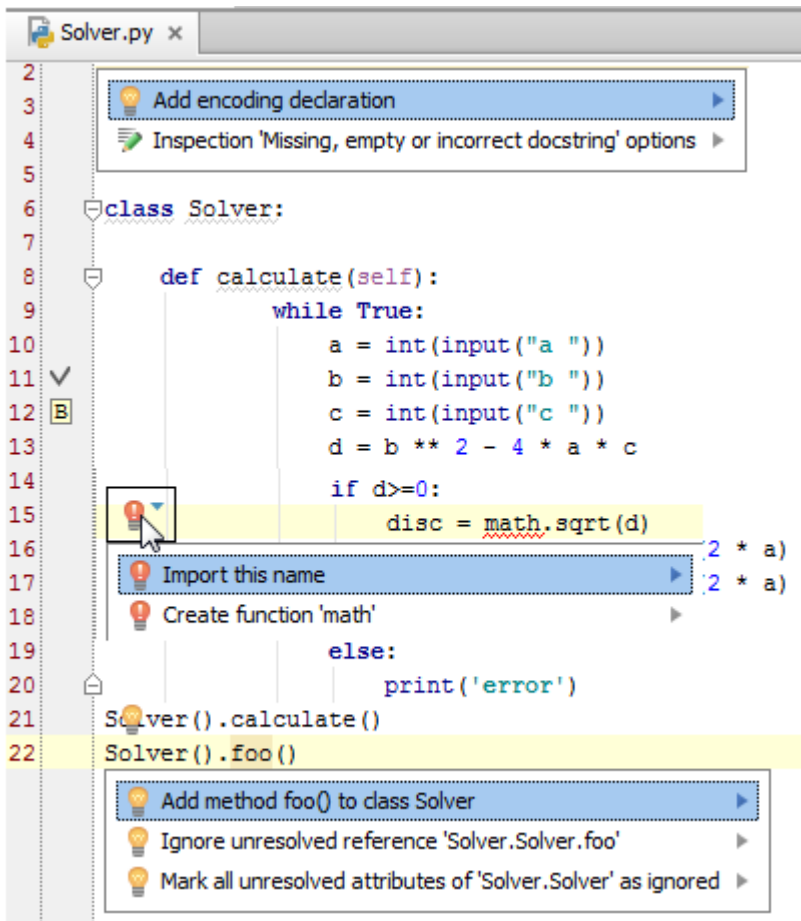
黄色灯泡意味着 Pycharm 对你当前编写的代码提出了一些建议，此时的程序并没有什么错误，但是可以对其进行一些改进，例如添加几行说明文档等等。另外一个作用就是创建使用源，比如当你使用了一个尚不存在的函数，Pycharm 会通过这种方式来提醒你去创建它。

红色灯泡则意味着 Pycharm 发现当前代码中存在错误而给出的修正建议，例如需要导入缺失的第三方库、源文件丢失等等。Pycharm 会给出快捷方便的纠错提醒。

Pycharm 将给出的各种各样的修改建议显示在建议列表中，可以通过以下方式打开建议列表：

单击灯泡图标。

按下 Alt+Enter 快捷键。



更多相关信息参见 [Intention Actions](#)。

最全 Pycharm 教程（22）——Pycharm 编辑器功能之窗口选项卡管理

1、主题

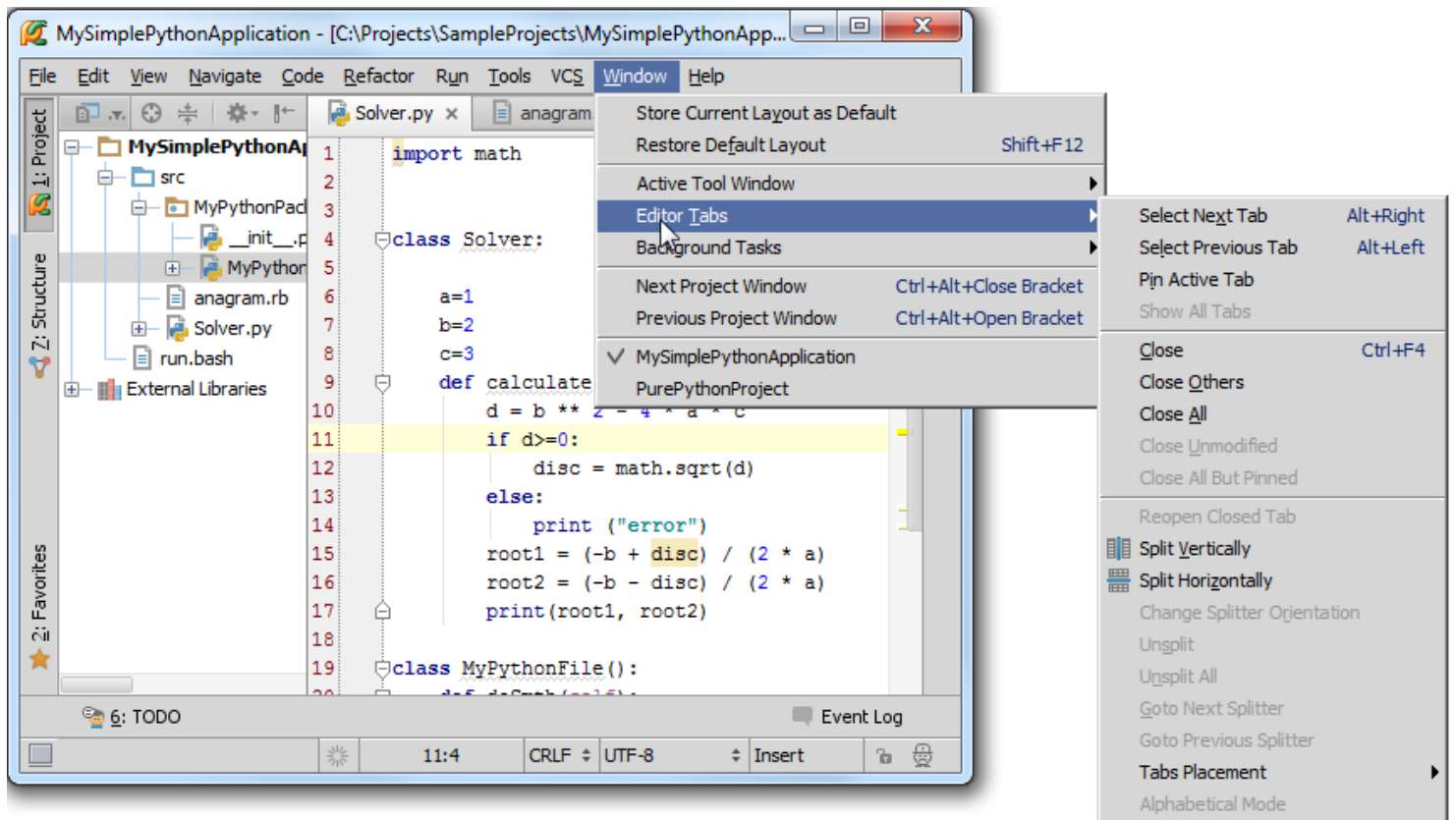
我们已经注意到 Pycharm 的主编辑框是基于窗口选项卡机制显示的，Pycharm 选项卡多种多样，这里我们将详细介绍这种选项卡机制。

2、激活的选项卡

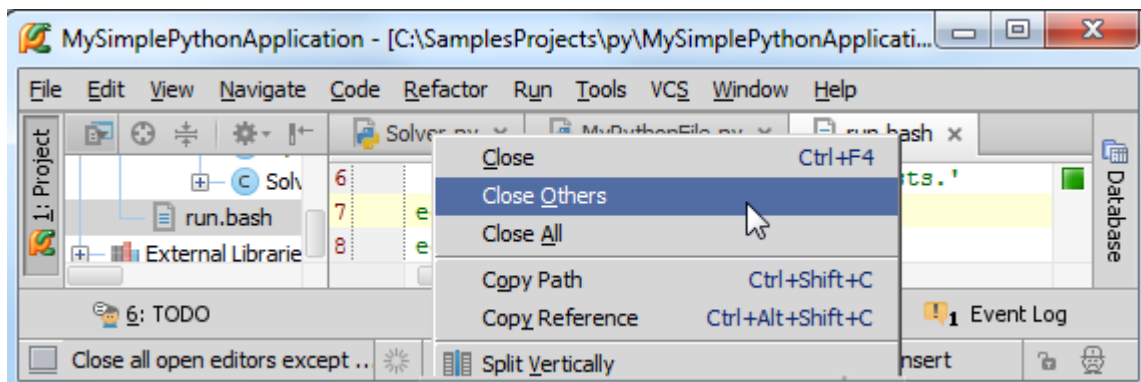
每当我们打开一个 Python 文件时 [open a file for editing](#)，它都会对应打开一个选项卡窗口，当前处于操作状态的选项卡成为激活选项卡。

3、选项卡行为属性

在主菜单中找到选项卡列表（Window → Editor Tabs）：



或者使用选项卡标题栏区域的快捷菜单：

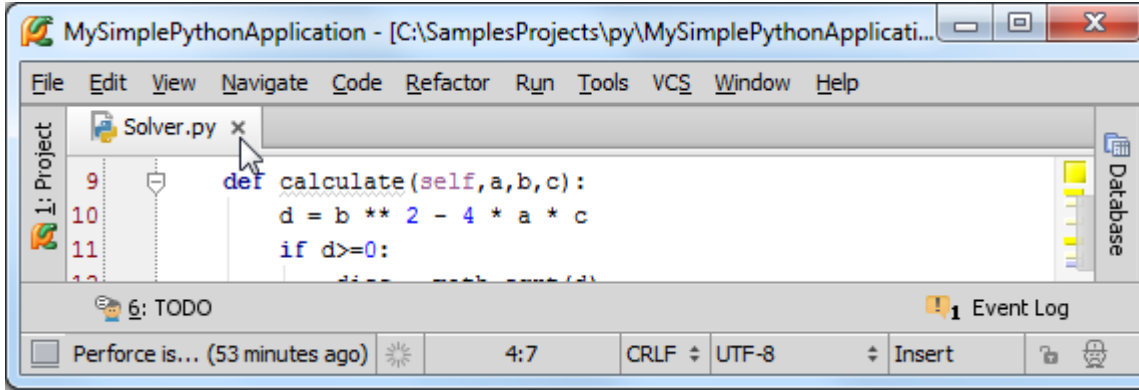


4、关闭编辑选项卡

关闭编辑选项卡的方法多种多样，依据不同设置而不同 [settings](#)：

(1) 在超出窗口容纳的最大数量时，选项卡会自动关闭。例如 Tab limit 定义了一次所能显示选项卡的最大数量，当一个新的选项卡被打开后，Pycharm 就会根据已有的管理策略来关闭一个已经打开的选项卡。

(2) 我们可以单击标题旁边的叉号来手动关闭对应选项卡。



注意这个叉号的显示也是可以设置的（勾选 editor 选项卡中的 Show "close" button 选项）

(3) 按下 Ctrl+F4 快捷键来关闭当前激活的选项卡。

(4) 使用主菜单命令 Window → Editor Tabs 或者快捷菜单对应命令。

5、选项卡之间的切换

Pycharm 提供多种方式来完成多个选项卡之间的跳转：

(1) 使用鼠标单击对应的选项卡标题栏完成切换

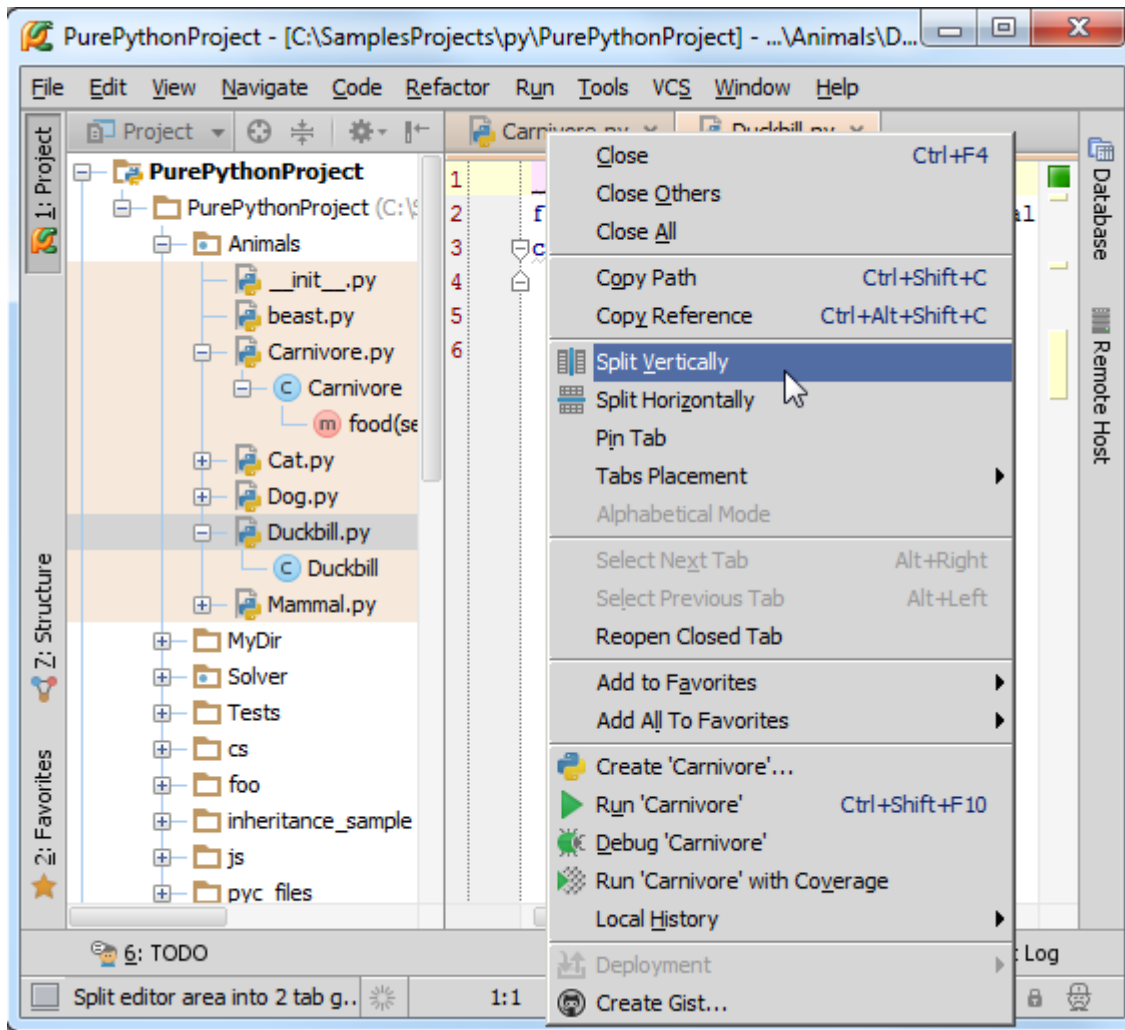
(2) 使用 Alt+Right 快捷键依次打开右侧的编辑器选项卡，使用 Alt+Left 快捷键依次打开左侧的编辑器选项卡。

(3) 使用标题区域快捷菜单的导航命令 Select Next Tab/Select Previous Tab，或者主菜单上的 Window → Editor Tabs 命令。

6、选项卡的拆分与合并

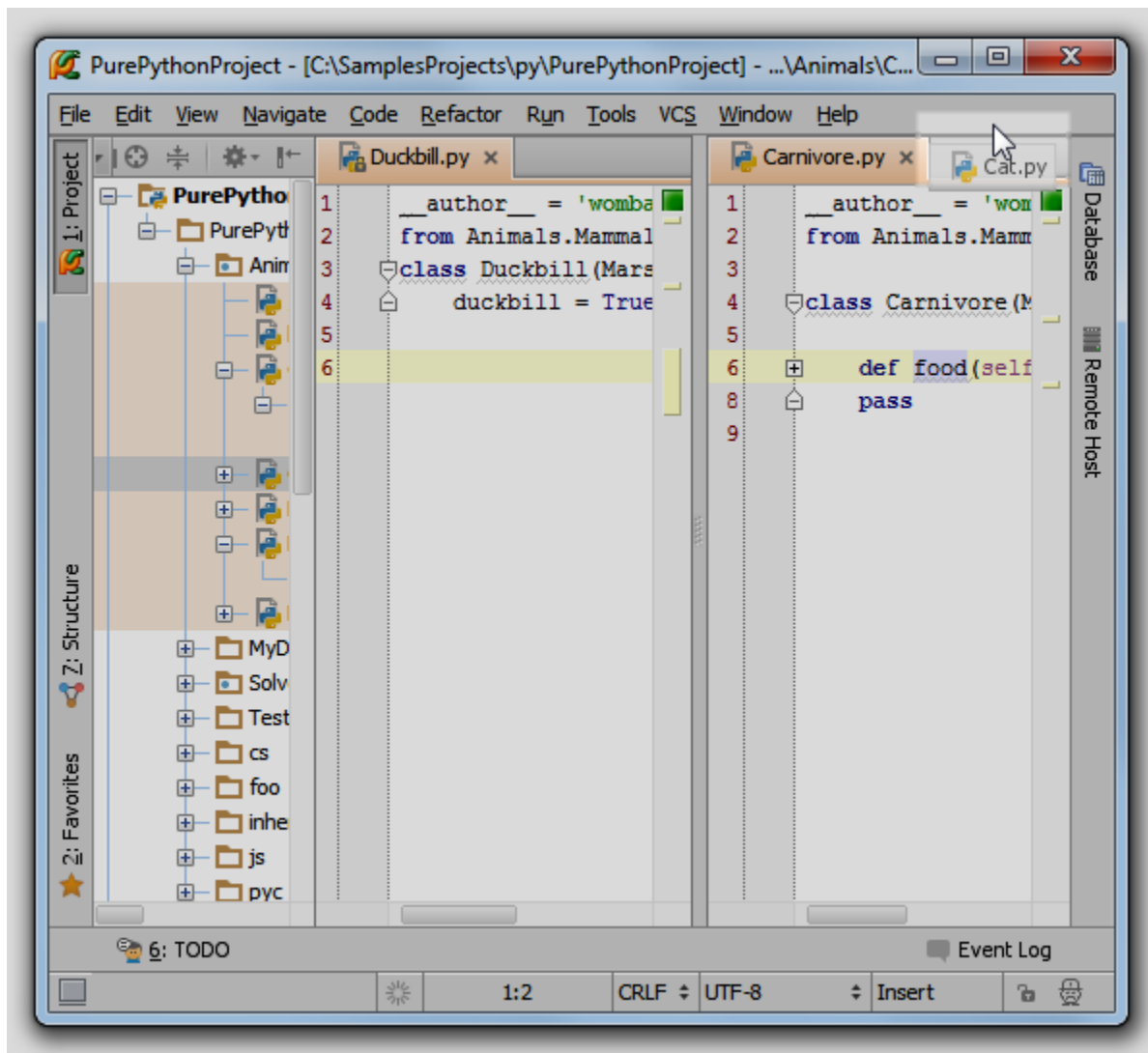
什么情况下需要对选项卡进行拆分呢？假设我们编写的文件非常长，我们希望同时显示这个文件的不同部分，或者希望在一个窗口中同时查看多个文件，这种情况下就需要对选项卡进行拆分。另外一种情况就是在创建选项卡组 [groups of tabs](#) 的时候也会用到拆分技术。

在选项卡标题区域的快捷菜单中有拆分命令：



需要强调的是在系统定义的快捷键配置方案中（比如说默认的 Windows 快捷键方案）是没有与拆分命令相关联的快捷键的，我们向其中自定义添加对应的快捷键设置。详见 [Configuring keyboard schemes](#) 和 [Configuring keyboard shortcuts](#)。

拆分后的窗口共享一个剪贴板，因此可以很方便的在各个选项卡之间进行复制粘贴，当然也可以将一个选项卡组的文件拖动到另一个选项卡组中。



也可以在已拆分和未拆分的选项卡组中进行切换，使用主菜单命令 Window → Editor Tabs → Goto Next Splitter/Goto Previous Splitter。

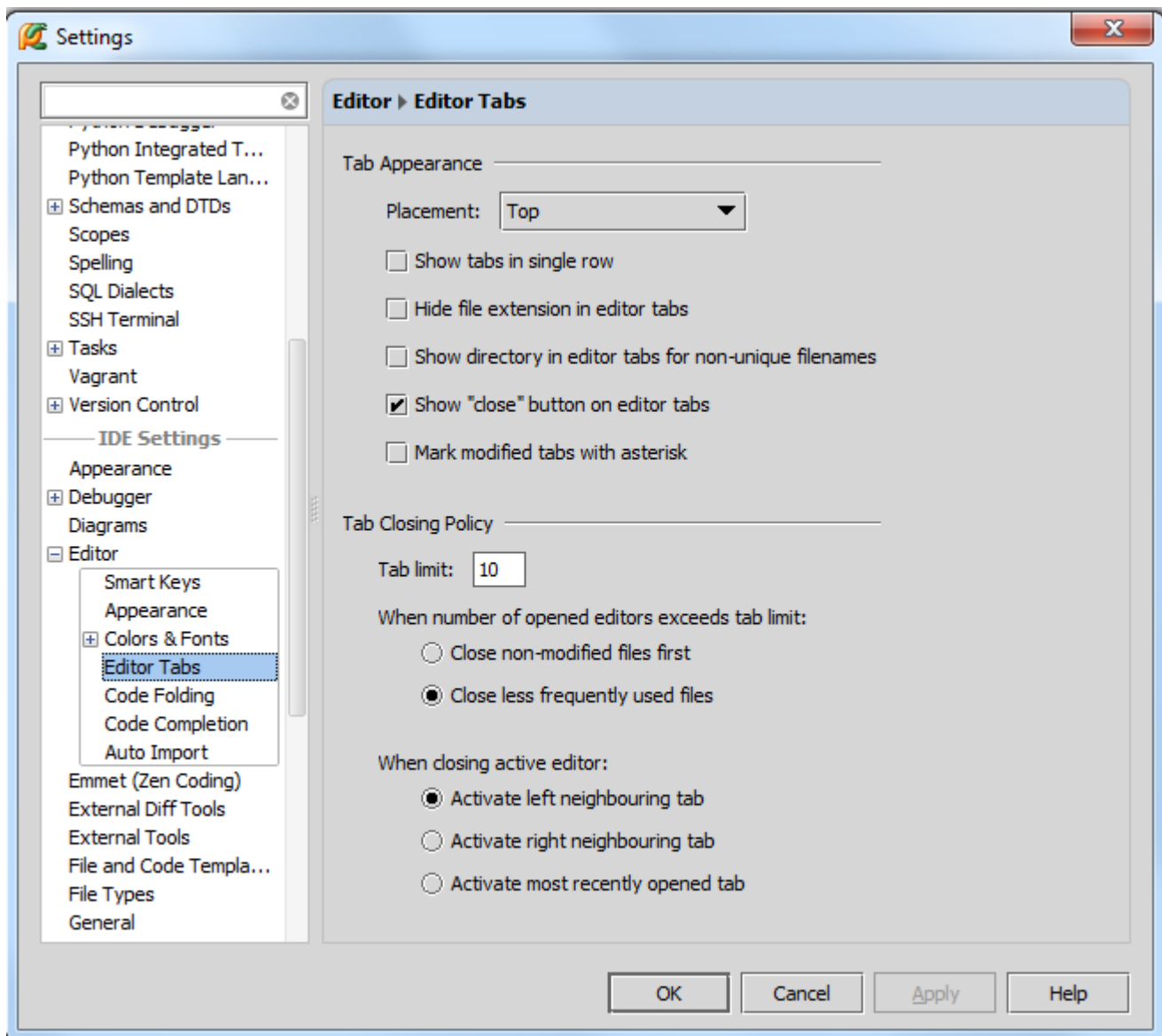
需要强调的是在系统定义的快捷键配置方案中（比如说默认的 Windows 快捷键方案）是没有与拆分命令相关联的快捷键的，我们向其中自定义添加对应的快捷键设置。详见 [Configuring keyboard schemes](#) 和 [Configuring keyboard shortcuts](#)。

Pycharm 允许我们更改拆分方向。例如我们已经创建了一个垂直方向的拆分窗口，并且不想再看到水平拆分，只需选择标题区域快捷菜单命令 Change Splitter Orientation，或者主菜单命令 Window → Editor Tabs。

当我们厌倦了拆分的窗口后，可以通过标题区域快捷菜单命令 Unsplit 来取消拆分，或者是主菜单命令 Window → Editor Tabs。当然这样只会取消当前选项卡组的拆分，如果希望取消所有拆分，选择 Unsplit All 命令即可。

7、选项卡属性配置

我们可以在 Pycharm 设置对话框中更改编辑选项卡的属性配置。打开 [Editor Tabs](#) 页面（单击主工具栏的设置按钮，或者选择 File → Settings 主菜单命令，展开 Editor 节点，单击 Editor Tabs），依据个人习惯进行更改：



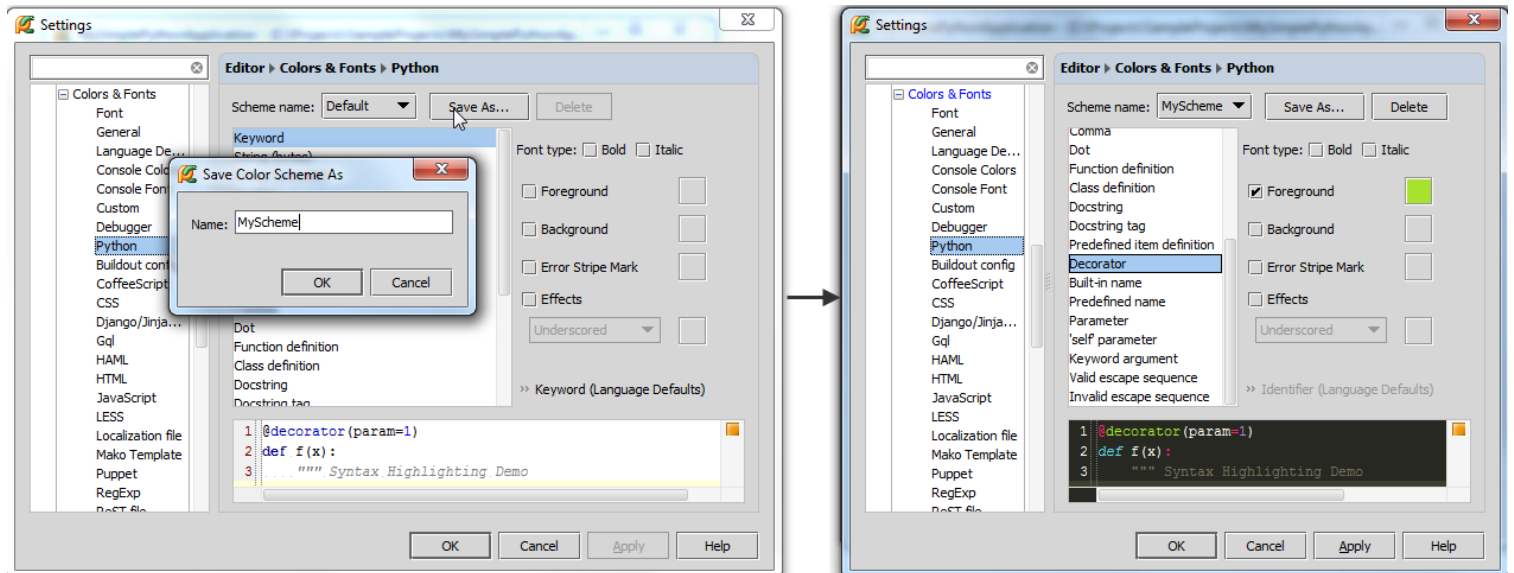
更多选项卡信息参见 [Managing editor tabs](#)。

最全 Pycharm 教程（23）——Pycharm 编辑器功能之代码高亮显示及错误提示机制

1、代码高亮显示

当你在编辑框中输入代码时，Pycharm 会在后台对其进行分析。这个 IDE 能够智能的识别出关键字、变量、字符串、注释等，并以不同的字体颜色进行显示。Pycharm 的符号配色方案定义在 [Colors and Fonts settings](#) 中（Ctrl+Alt+S→IDE Settings→Editor→Color and Fonts）。

我们先选择编程语言，这里选择 Python，然后根据个人习惯来设置字符配色方案。然而这里 Pycharm 自带的配色方案是不可更改的，我们需要先创建一个拷贝，然后更改：

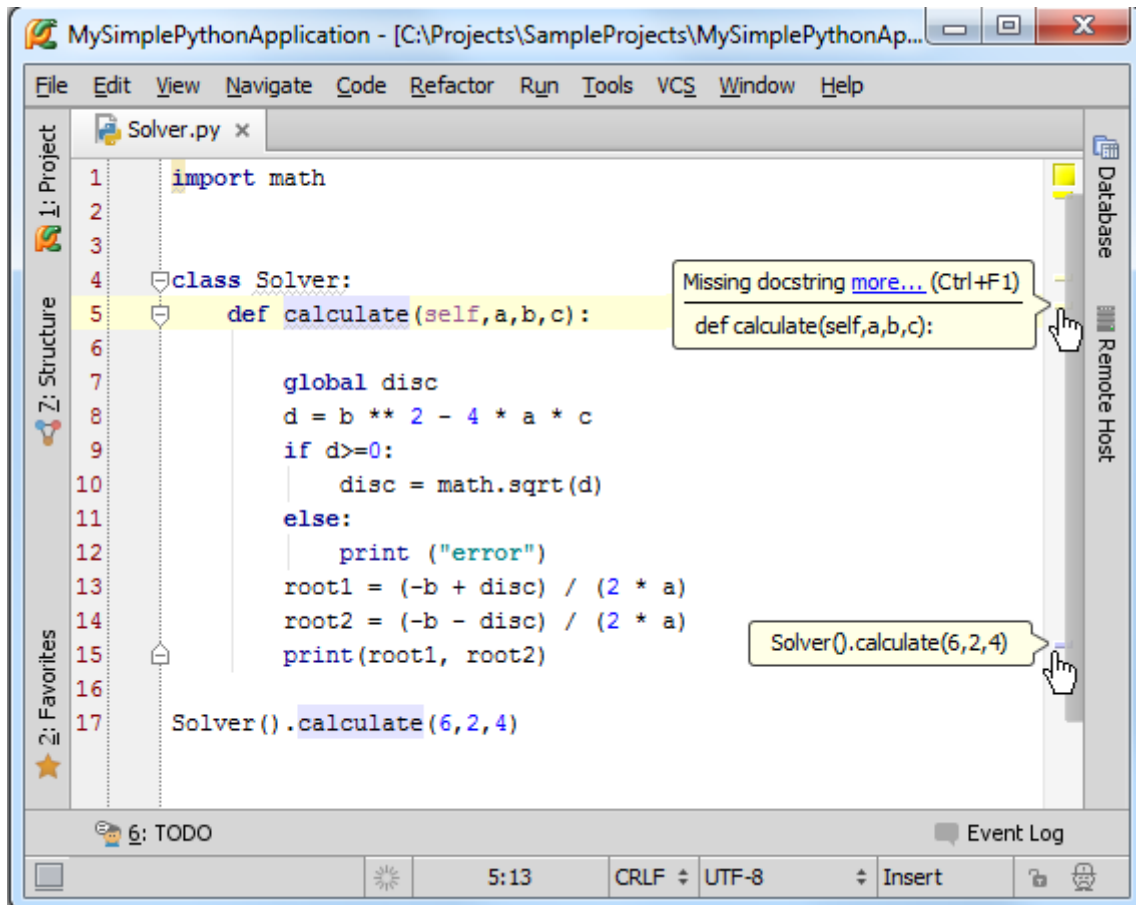


在预览窗口中会实时展现当前的更改设置，直到你满意为止。单击应用，关闭设置对话框。

详细信息参见 [Configuring Colors and Fonts](#)。

2、触发标志

回到代码部分，如果你想知道某个符号已经使用的次数，只需将光标定位在其中的一处符号上，其他位置的相同符号就会以高亮形式显示，并且会在右槽显示对应的发生标志（条纹），如果将鼠标指针悬停在这些条纹上，Pycharm 会给出详细信息：

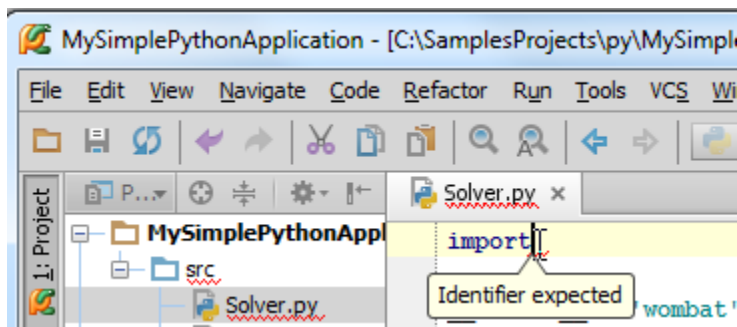


3、代码错误提示

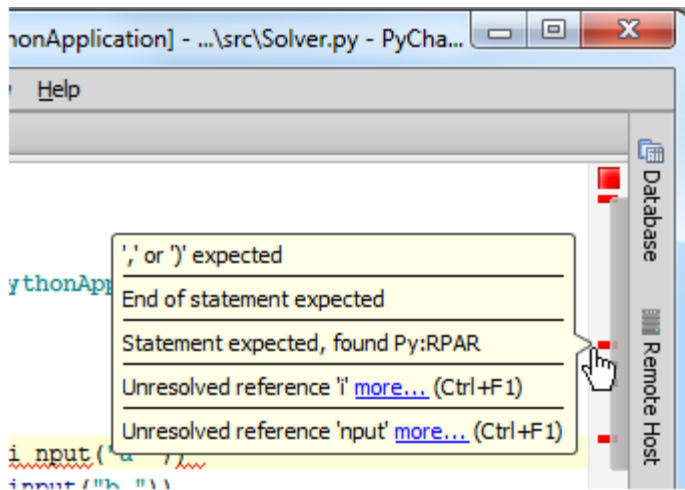
如果 Pycharm 在对代码分析的过程中发现了错误，则会按照以下方式给出提示：

用红色波浪线标记错误代码，鼠标悬停在波浪线上时会给出详细的错误信息。

标题栏的文件名也会用红色波浪线标记，在项目窗口中的目录也会有同样的标记。



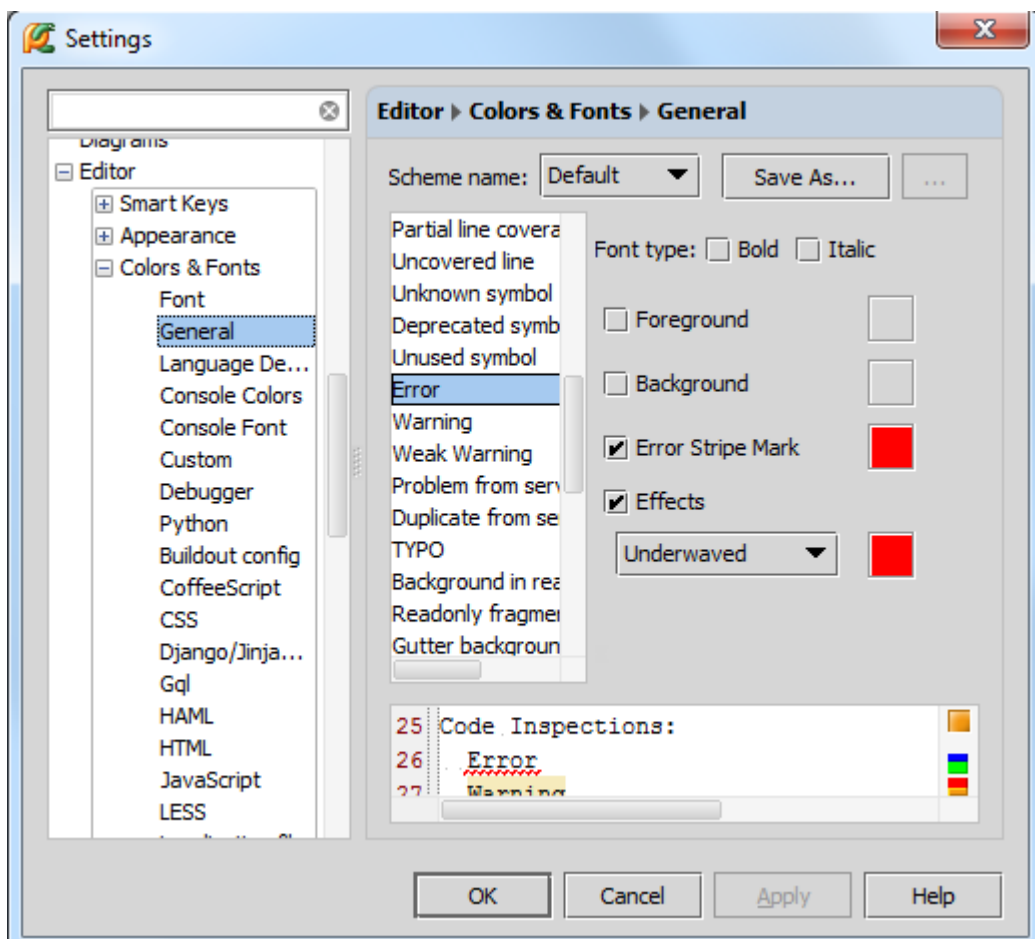
在右槽的对应位置显示错误标识，鼠标指针悬停在上方时会给出详细错误信息。



这些错误标识也可以起到导航作用，帮助我们快速定义错误发生的位置。

在右槽顶端显示整个文件的状态标识。绿色带便一切正常，黄色代表存在一些警告，红色代表存在错误。

当然这种错误提示机制的配色方案也是可以更改的，在字体及颜色设置对话框中的 **General** 页面进行设置（Ctrl+Alt+S→IDE Settings→Editor→Colors and Fonts→General）：

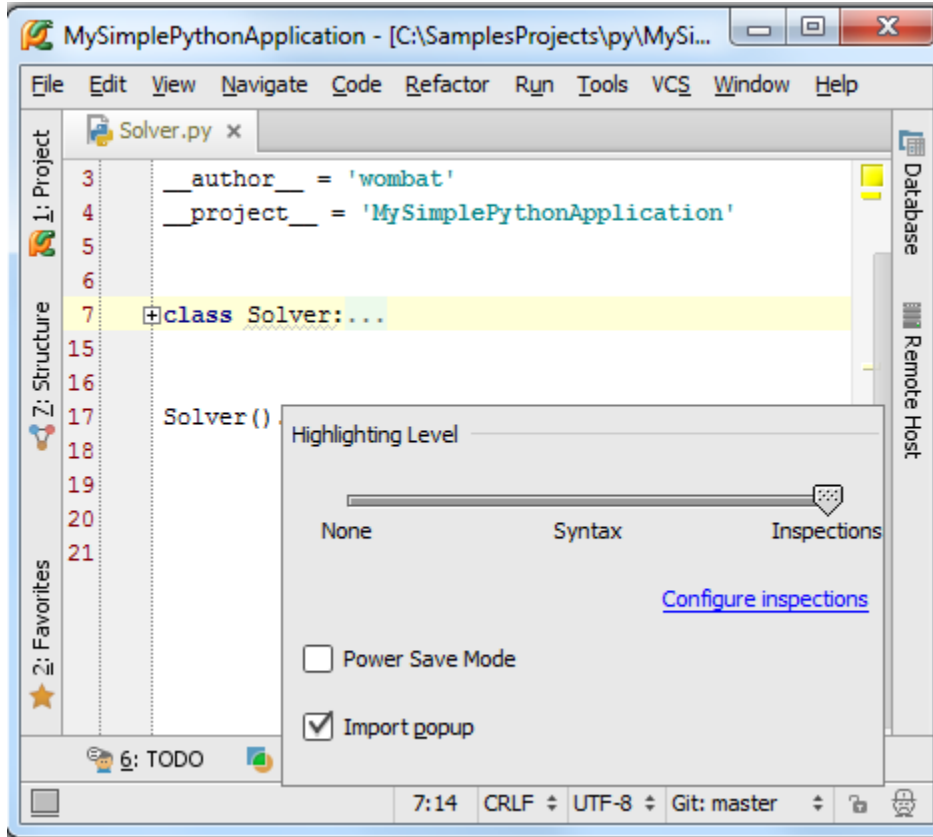


4、再见帅哥

注意窗口左下角的帅哥头像：



他的名字叫 Hector-the-Inspector，负责代码的纠错检查，单击这个头像，会弹出一个窗口，里面有代码的高亮程度调节器：



如果你对自己的编程水平很有自信，那么你可以向左拖动到 Syntax 或者 Hector 头像 None。在 Syntax 位置，系统只会对语法错误给出高亮提示，Hector 头像只剩一半。在 None 位置，代码纠错功能完全关闭，编程速度更快，但也更容易出错，对应 Hector 头像消失。

不过这种更改只对当前文件有效。

如果你想更改代码纠错机制，单击 **Configure inspections** 链接，更多有关代码纠错机制的信息参见 [Code Inspections](#) 中的 [code inspection tutorial](#)。

最全 Pycharm 教程（24）——Pycharm 编辑器功能之宏定义

1、为什么使用宏

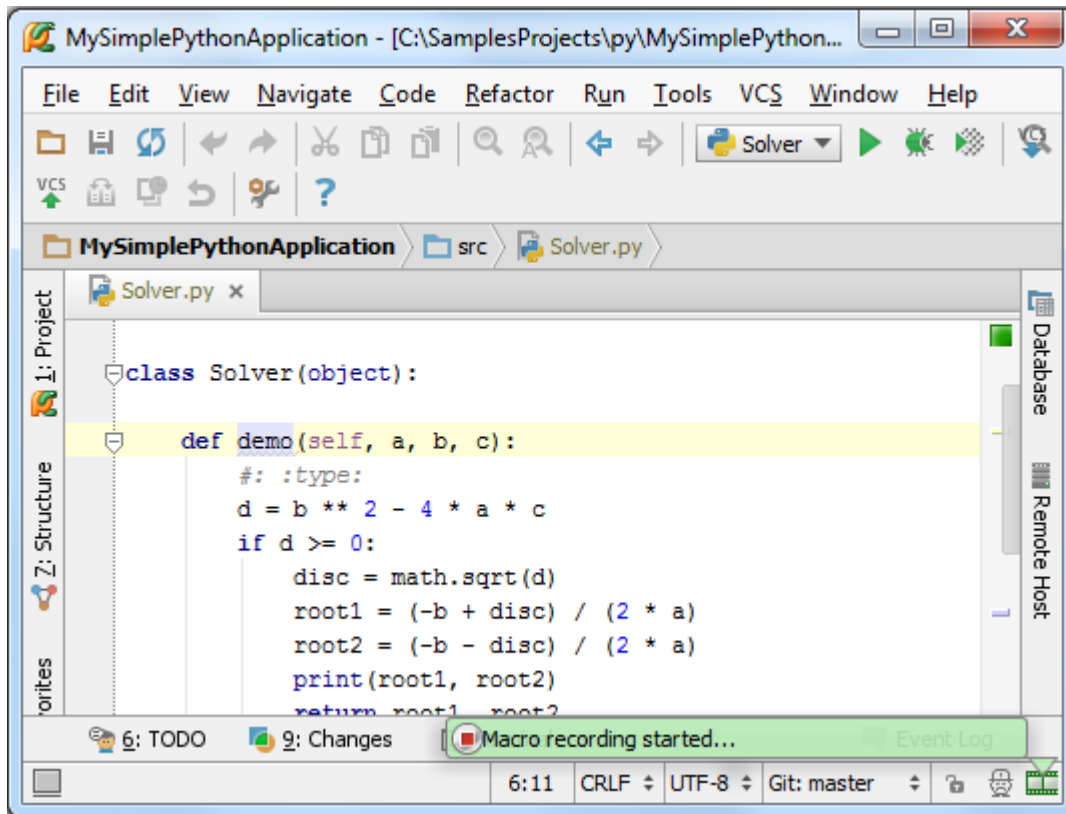
加入你需要重复某种操作很多次，例如选中源码并将其发送到控制台端调试，我们能不能将着一系列操作简化为一步，甚至用一组快捷键来代替呢？

2、准备工作

- (1) Pycharm 版本为 2.7 或者更高。
- (2) 与 [product documentation](#) 中的行为保持一致。
- (3) 创建了一个项目，并向其中添加了至少两个脚本，详见 [Getting Started](#) 和 [Debugger](#)

3、录制宏

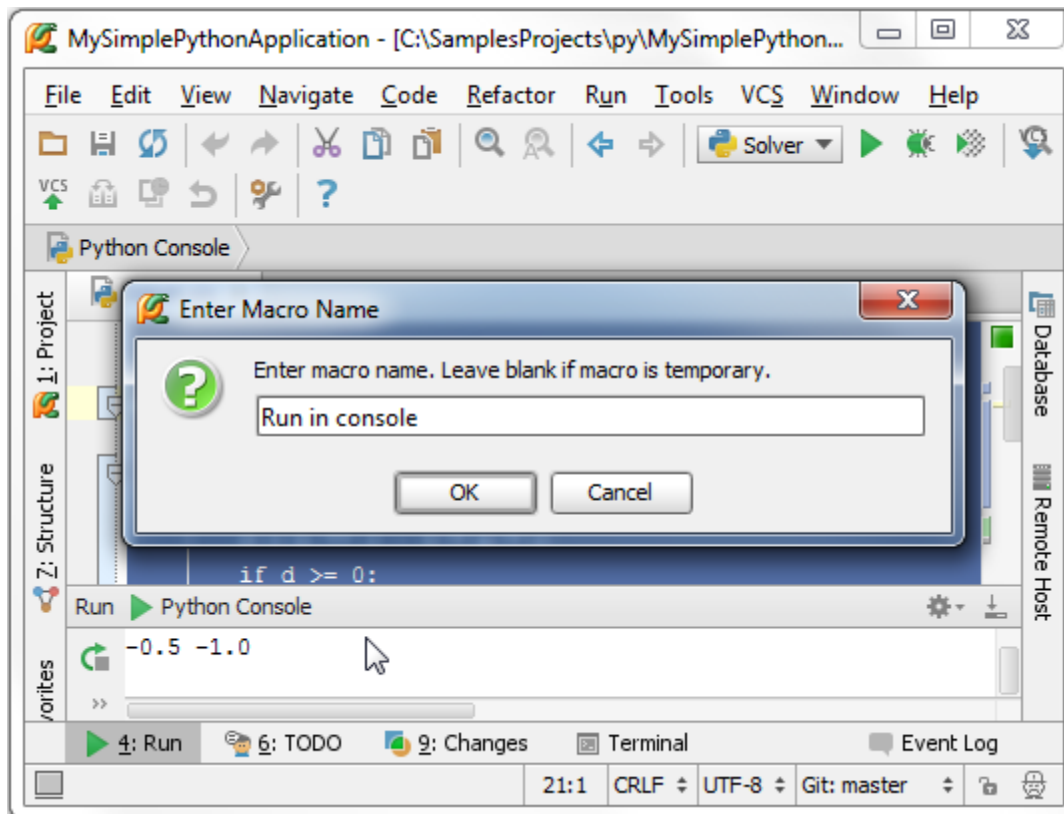
在主菜单上选择 Edit→Macros→Start Macro Recording 命令，在窗口底部出现 Macro recording started 的提示信息。



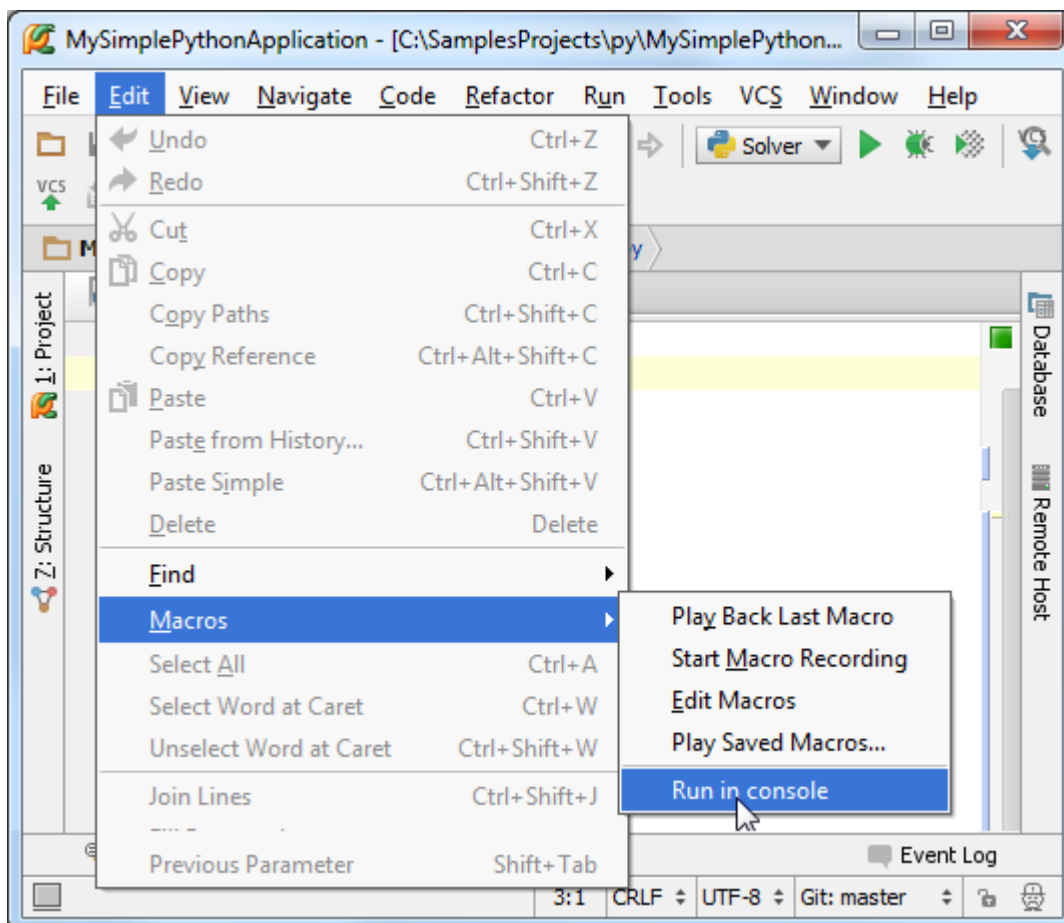
打开你想要执行的脚本文件（注意 [Using Macros in the Editor](#) 中所描述得列表限制），然后进行对应的需求操作：

- (1) 全选编辑器中的代码（例如在编辑窗口中按下 Ctrl+A）
- (2) 右击，在弹出的快捷菜单中选择 Execute selection in console 命令

然后单击主菜单的 Edit→Macros→Stop Macro Recording 命令，Pycharm 会提示你保存当前记录的宏。此时如果未指定宏名，Pycharm 会将其设定为一个临时的宏命令，这里我们将这个宏命名为 "Run in console"：



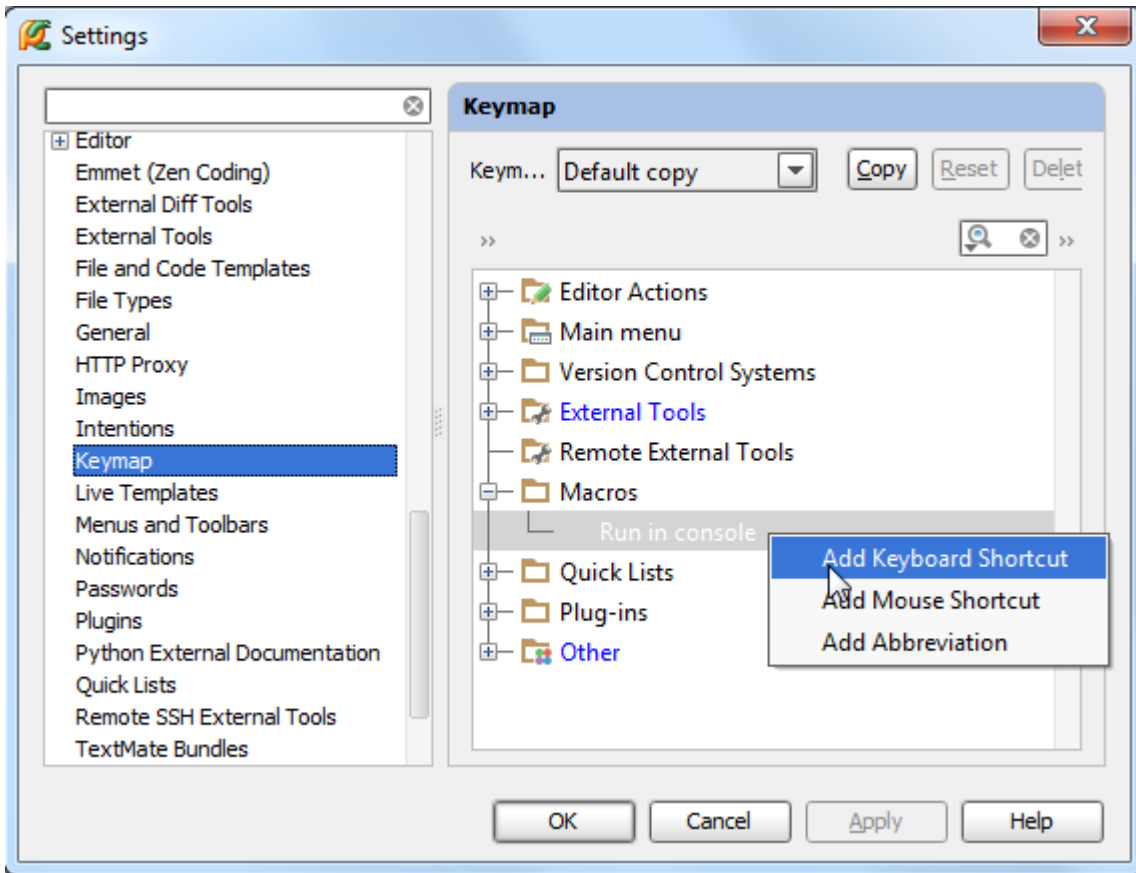
此时，再次查看 Edit→Macros 菜单，我们会在列表中发现我们刚刚定义的宏命令：



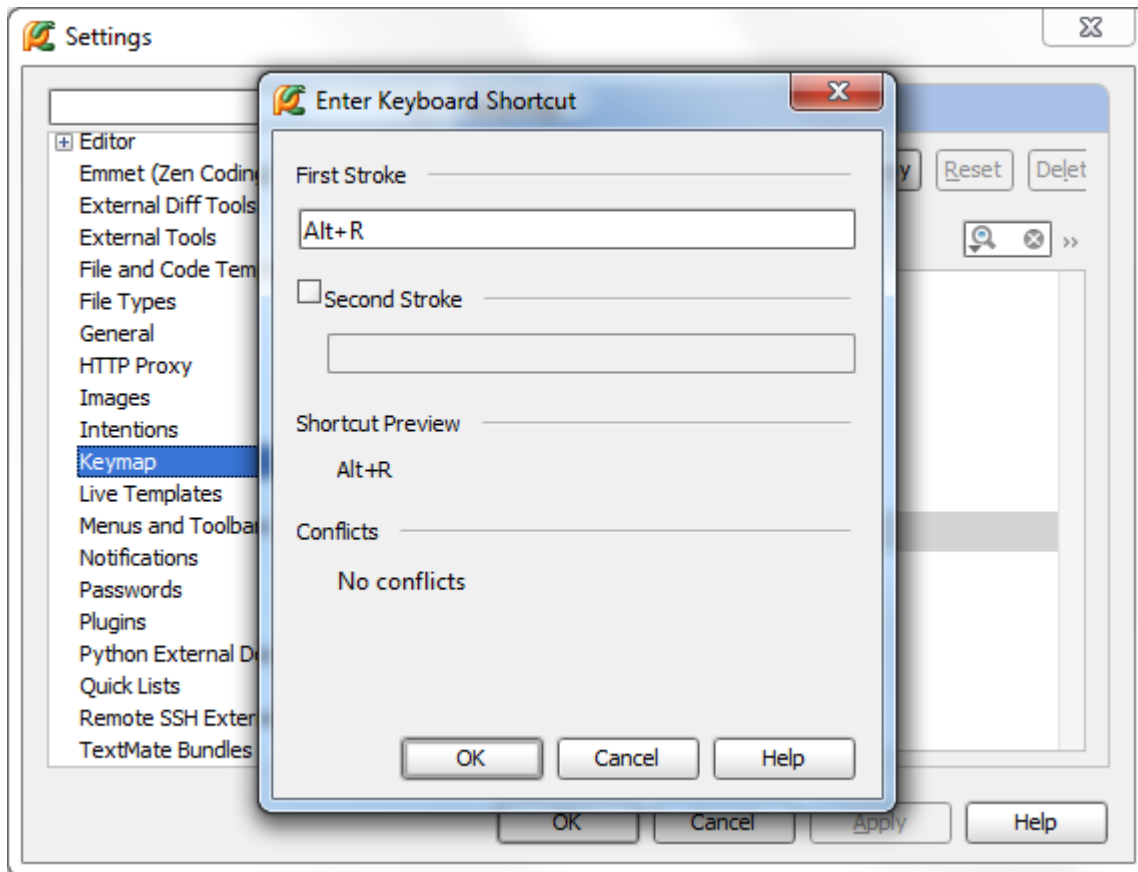
4、为宏命令指定快捷键

接下来我们为这条宏命令指定一个快捷键组合，做法如下。

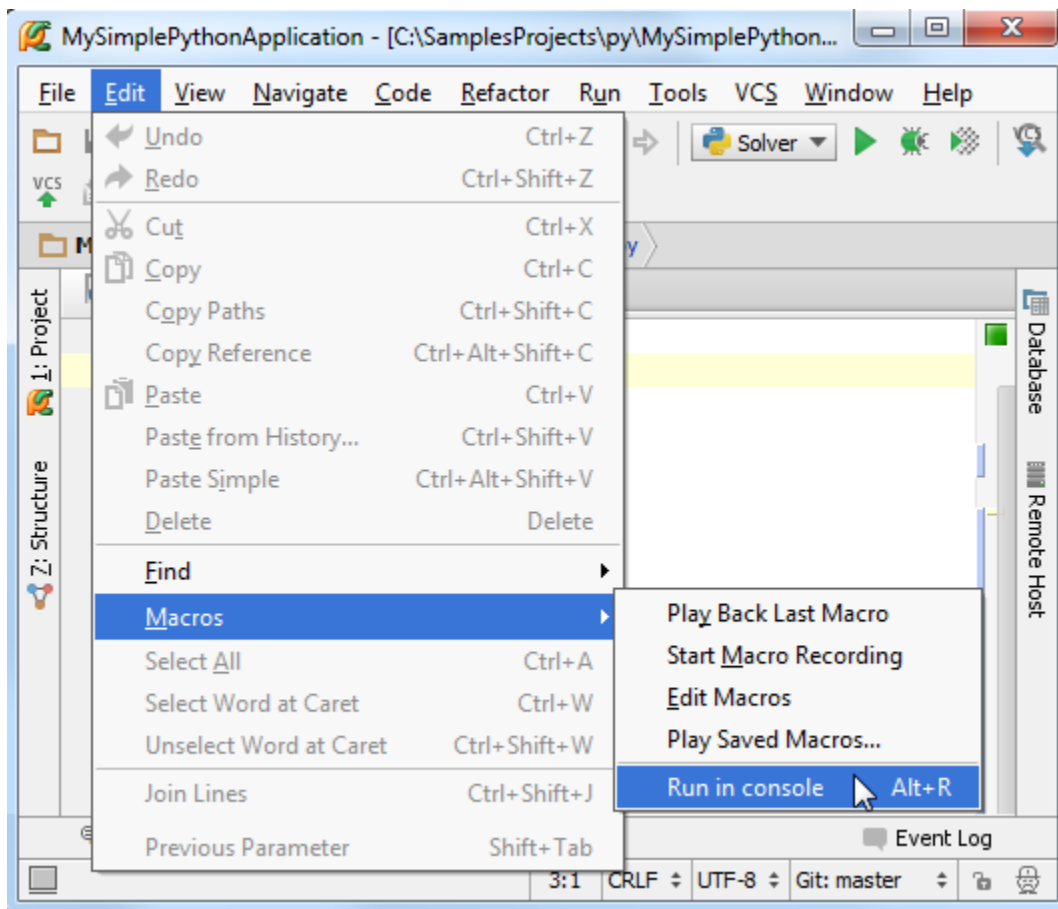
在设置对话框中，打开 [Keymap](#) 页，展开 Macros 节点，找到我们新添加的宏命令"Run in console"，右击，在弹出的快捷菜单中选择 Add keyboard shortcut：



接下来，在 [Enter keyboard shoctrut dialog](#) 对话框中指定期望的快捷键组合。注意此时我们只能通过鼠标指针来单击对话框中的控件，任何键盘操作都会被认为是快捷键的设置内容。



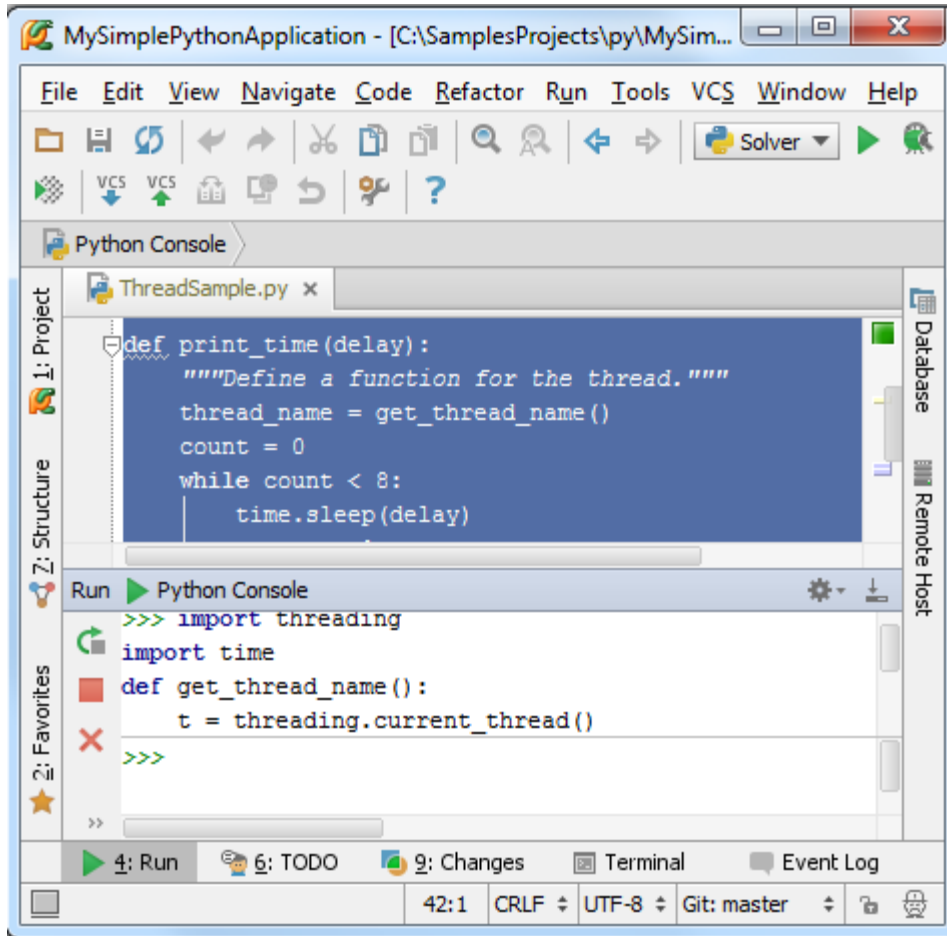
如你所见，系统并未提示相关快捷键冲突，我们的设置可用，单击应用并关闭对话框。此时新增的快捷键会显示在菜单中：



5、宏命令的使用

现在我们完成了一个宏命令的私人订制。此时我们可以在控制台端运行任何已打开的脚本文件。我们可以通过菜单命令 Edit→Macros→Run in console 来实现，也可以通过快捷键 Alt+R 来更为快捷的完成这个功能。我们尝试一下：

在编辑器中打开另外一个脚本文件，按下 Alt+R，OK，脚本被自动加载到了控制台中并运行：



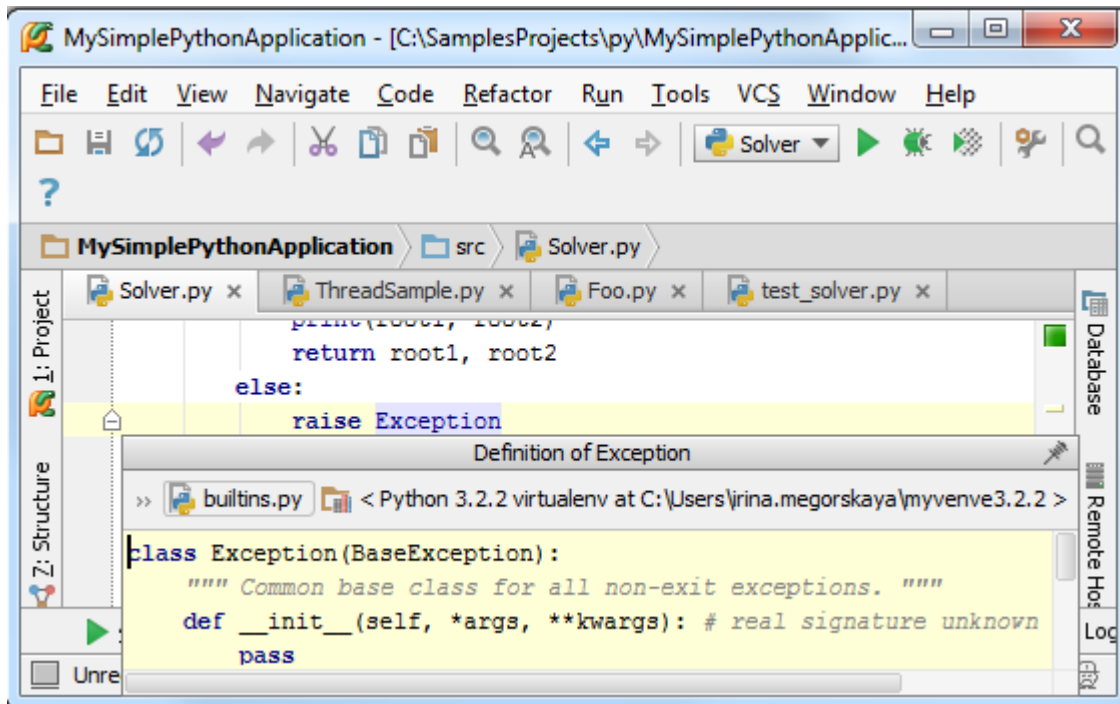
最全 Pycharm 教程（25）——Pycharm 编辑器功能之查看帮助文档

1、准备工作

- (1) Pycharm 版本为 2.7 或者更高
- (2) 与 [product documentation](#) 的注意事项保持一致
- (3) 已经创建了一个工程并且至少向其中添加了两个脚本文件，详见 [Getting Started](#)

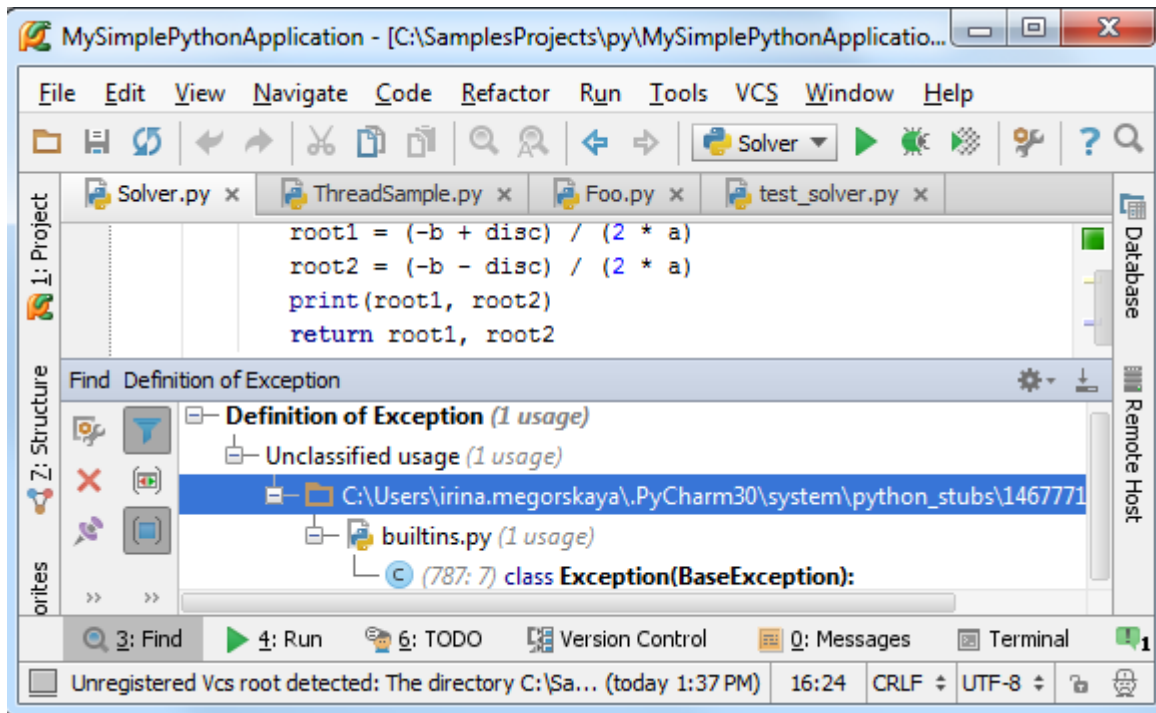
2、快速查看定义

当你只是想知道相关的声明信息时，不妨使用 [Quick definition](#)，而无需跳转到实际的定义位置。例如，将光标定位在一个表达式上，然后在主菜单中选择 View→Quick Definition：



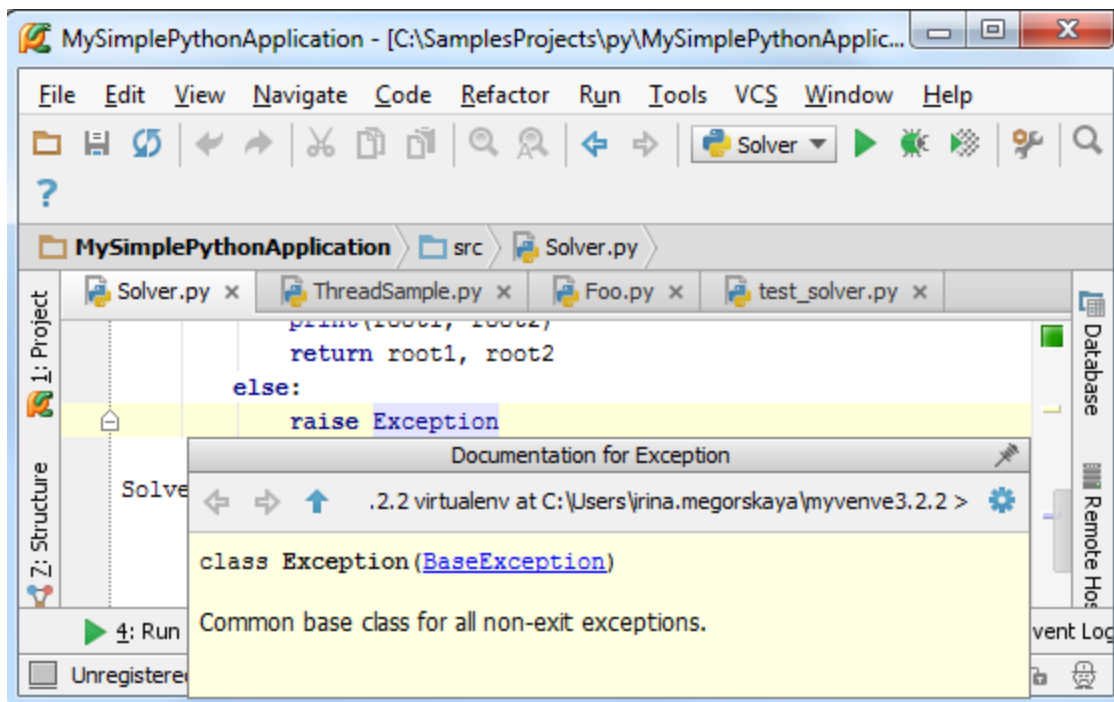
你可以在弹出的窗口中看到相关的快速定义信息，然后通过方向键来移动浏览整个提示信息。

单击  按钮，在 [Find tool window](#) 窗口中找到快速定义信息：

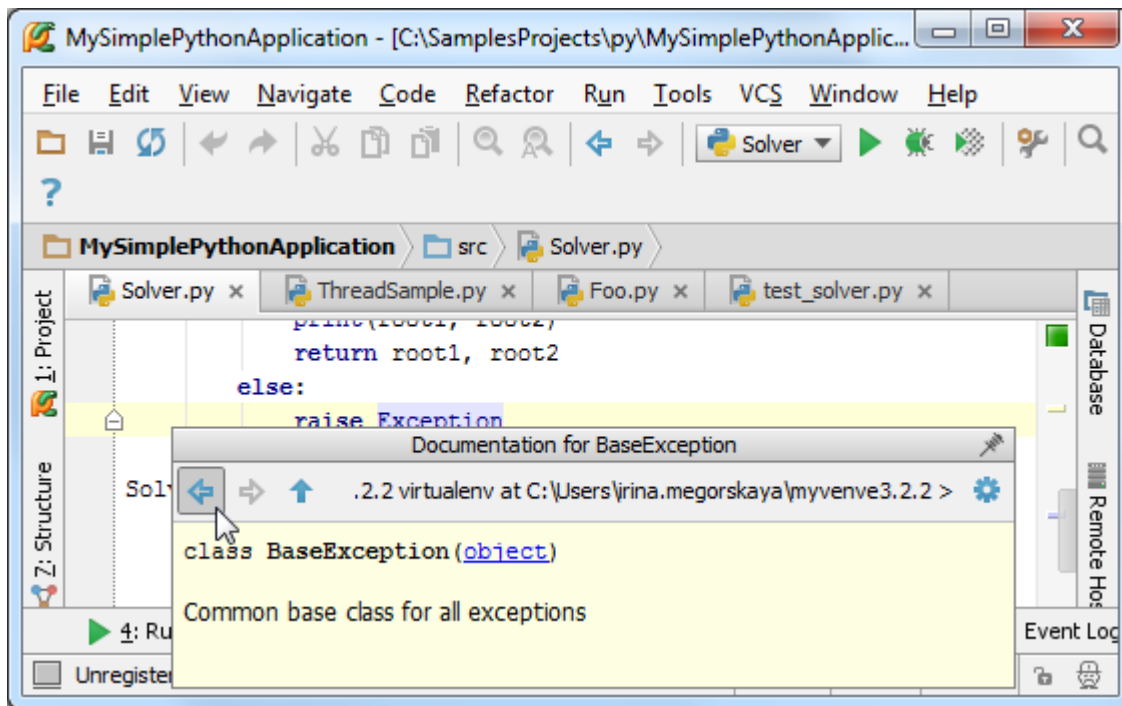



3、快速查看帮助文档

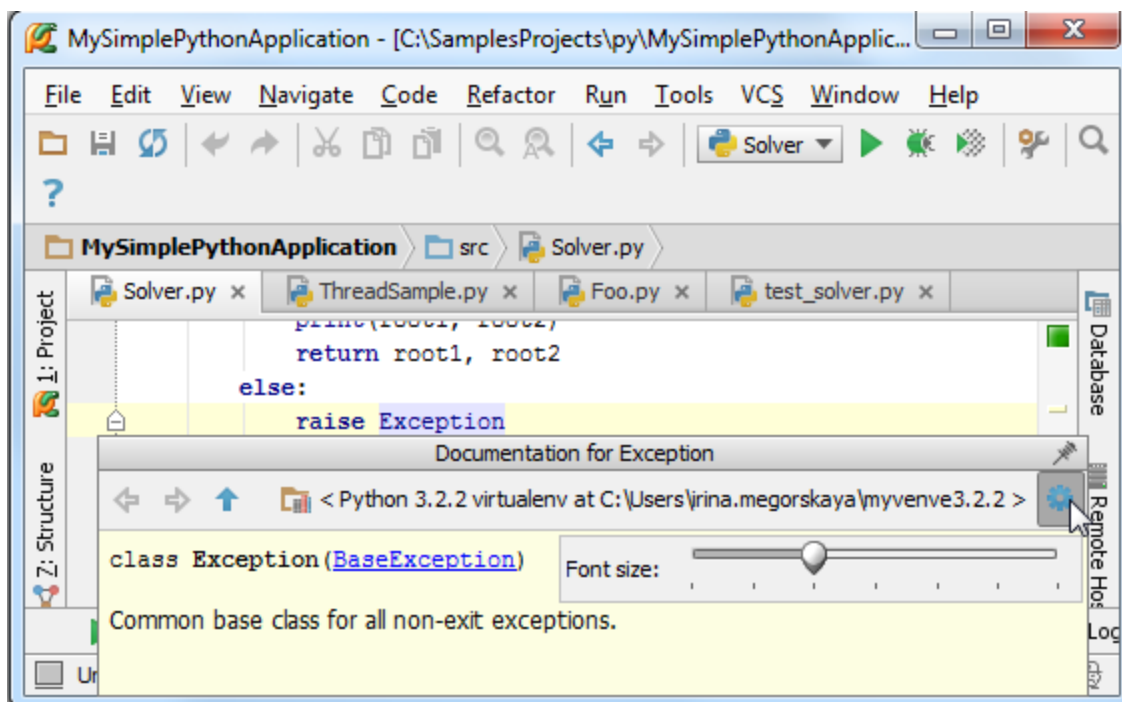
快捷（帮助）文档显示了当前符号的文档注释以及相关注释符。我们再次将光标定位在某个表达式上，这次我们使用 View→Quick Documentation 的菜单命令。



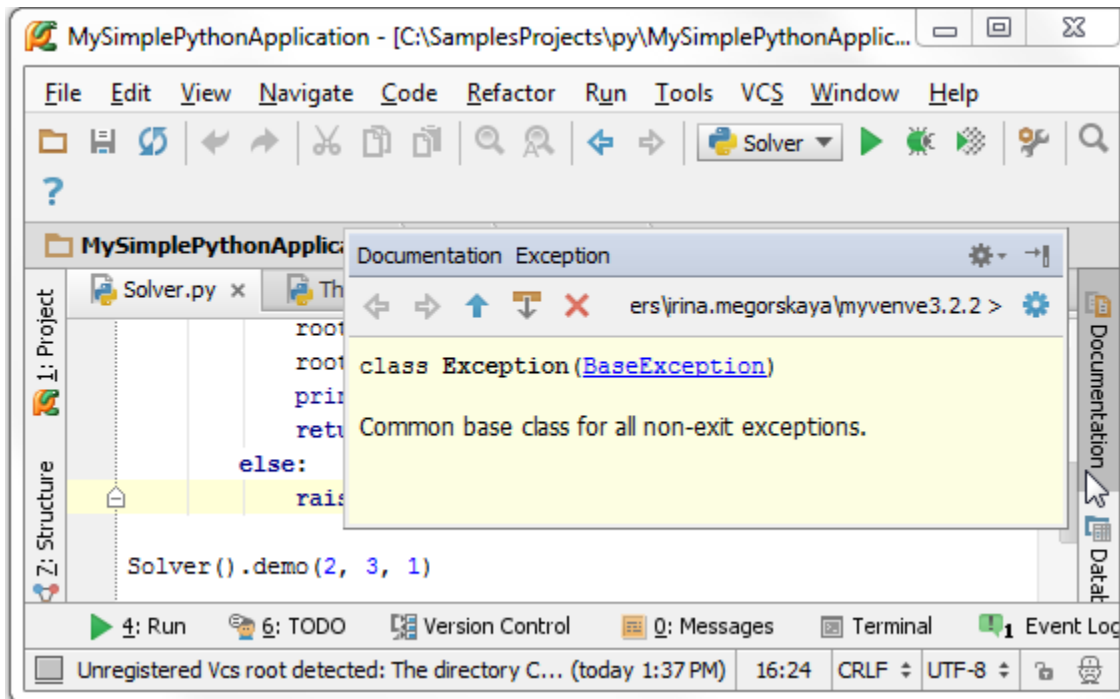
在弹出的窗口中我们同样可以通过左右方向键来浏览这个文档信息：



当然我们可以调整弹出窗口的大小。单击  弹出尺寸调节滚动条，拖动滑块来改变当前尺寸：



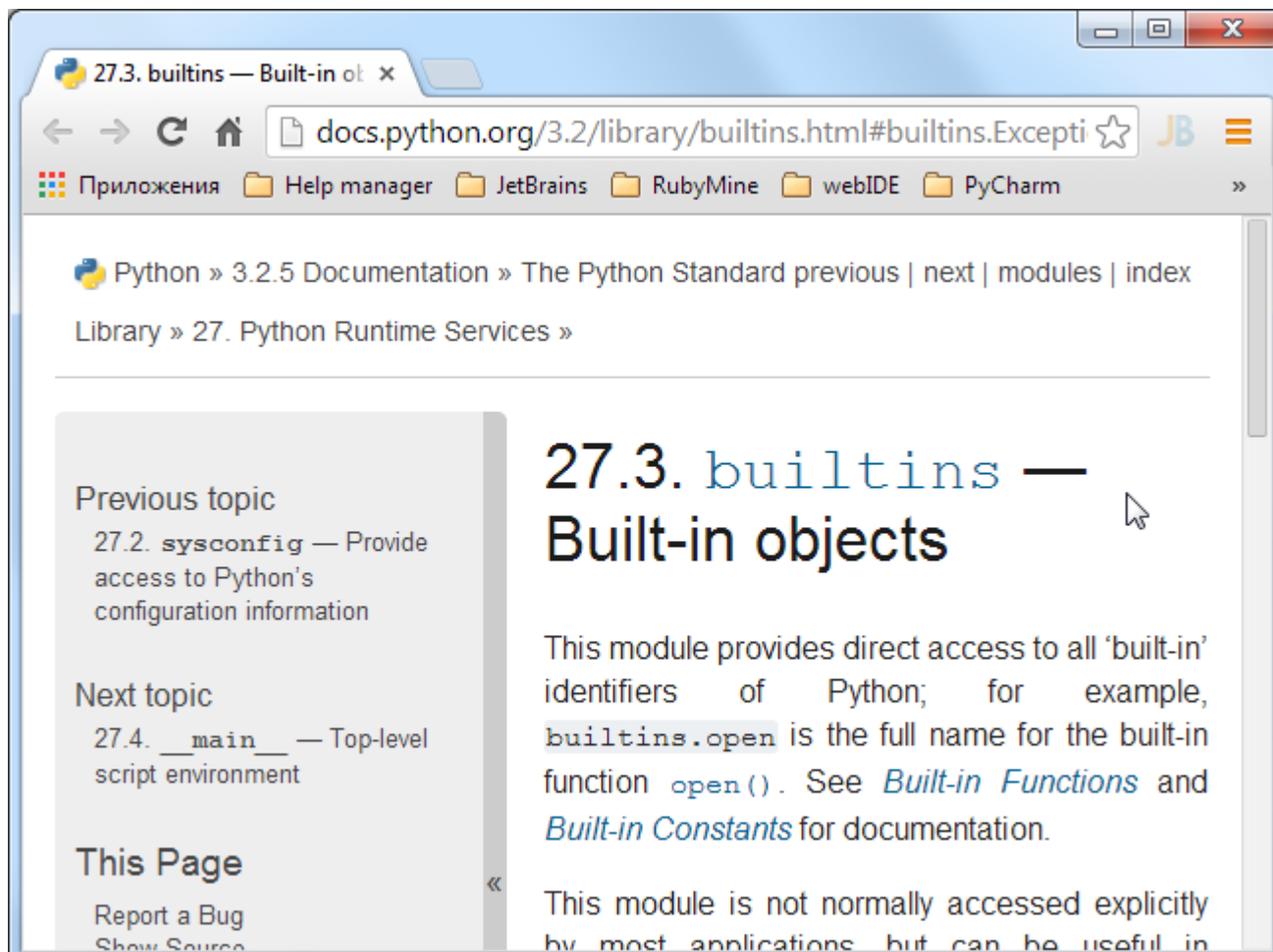
单击  按钮，在 [Documentation tool window](#) 窗口中打开快速帮助文档：




单击 **X** 恢复到原来的窗口形式。

4、查看外部文档

这个命令允许你通过默认浏览器查看详细帮助文档信息：

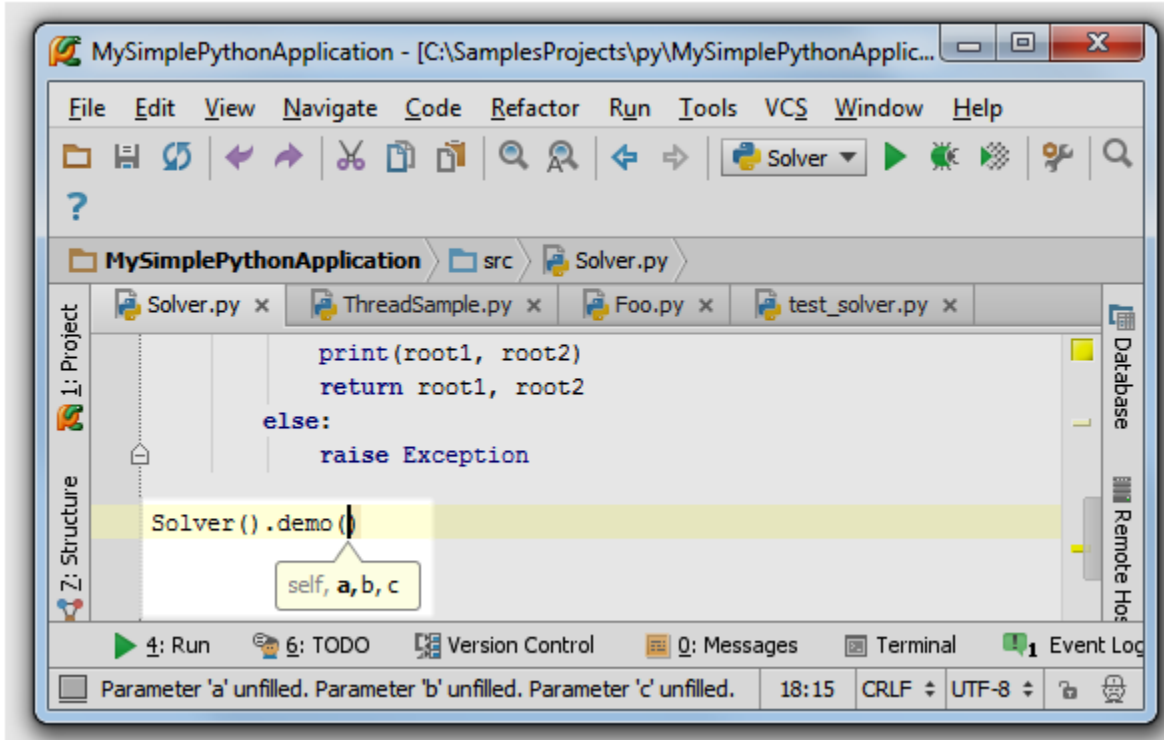


顺便提一句，你可以通过 [quick documentation pop-up window](#) 来打开外部帮助文档，可以单击  或者按下 Shift+F1 快捷键。

外部文档用到的 PyQt4, PySide, gtk, wx, numpy, scipy, 和 kivy 等第三方库都是默认版本的，如果你想查看其它版本下的帮助文档，例如 Pyramid，请到 [Python External Documentation](#) 对应位置参考。

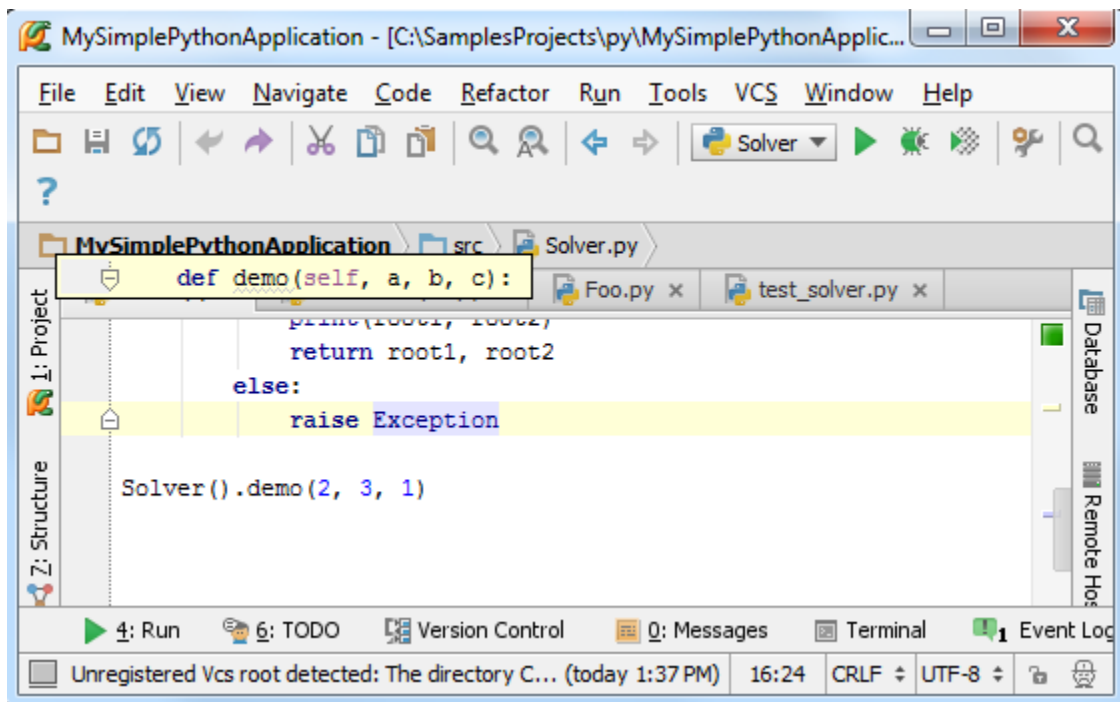
5、浏览参数信息

这个命令可以显示函数方法的形参信息：



6、浏览环境上下文信息

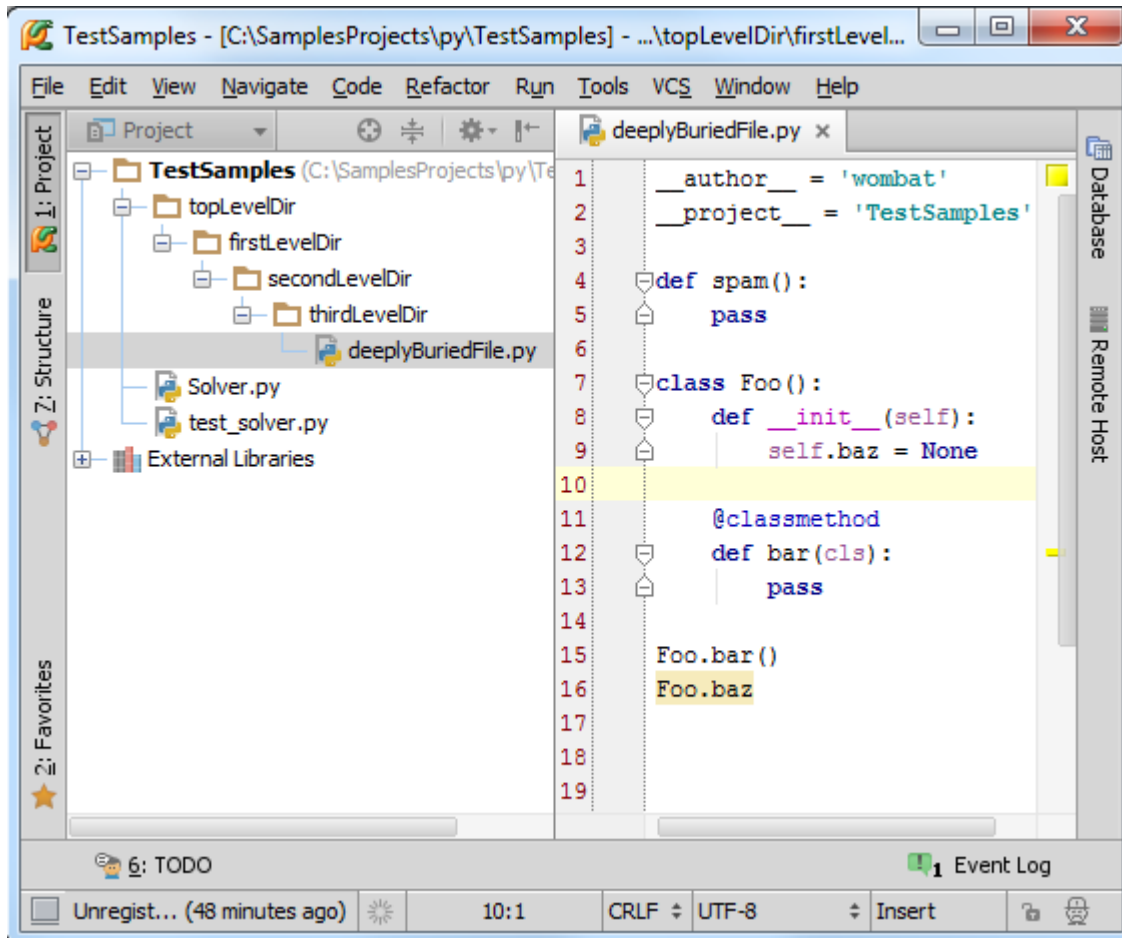
最后，你可以通过弹出窗口来浏览代码块开始部分的特定的符号信息。举个例子，我们将光标定位在一个 `exception` 上，而函数的声明位于当前可视编辑范围之外，然后按下 Alt+Q 或者 View→Context Info 菜单命令：



最全 Pycharm 教程（26）——Pycharm 搜索导航之文件名、符号名搜索

1、准备一个工程

向你的工程中添加一个 Python 文件，并输入一些源码，例如：

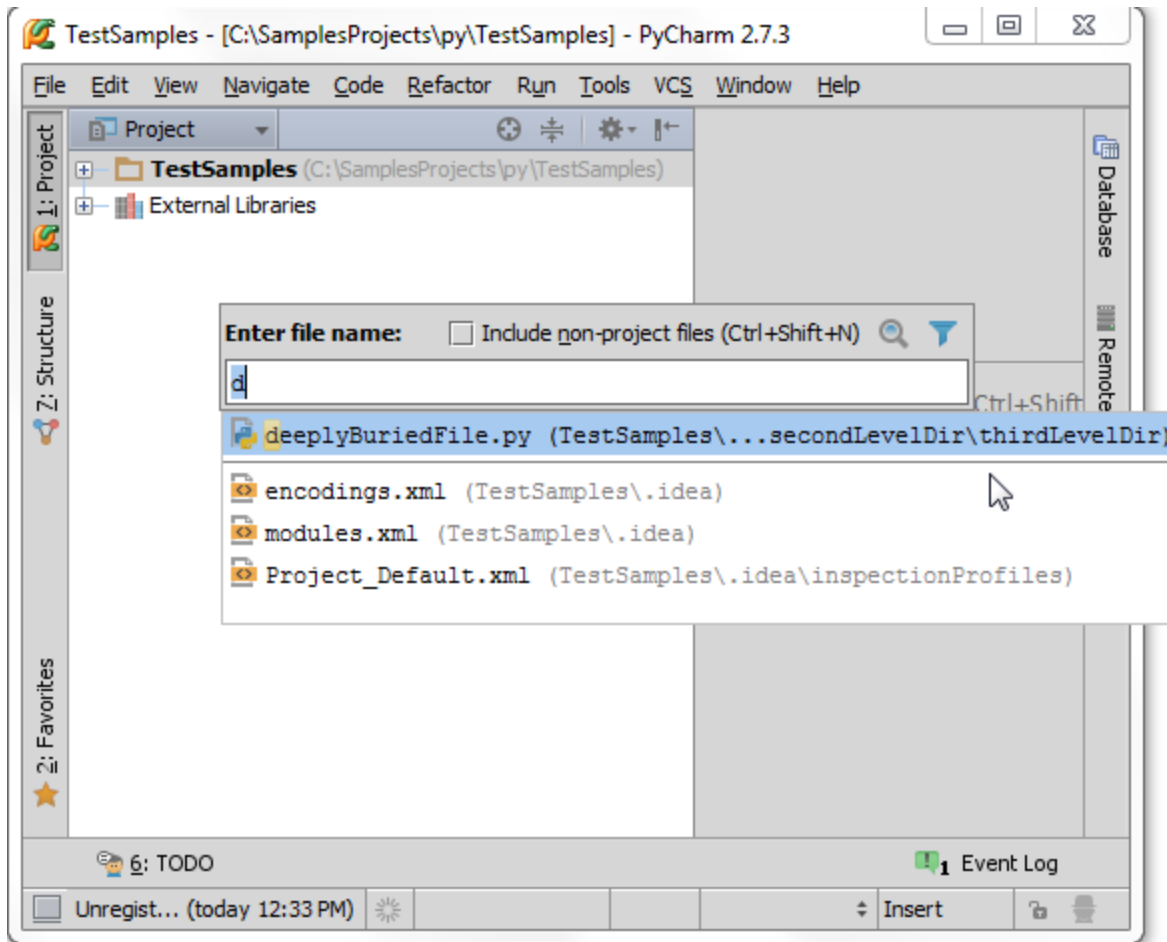


2、转到对应文件、类、符号

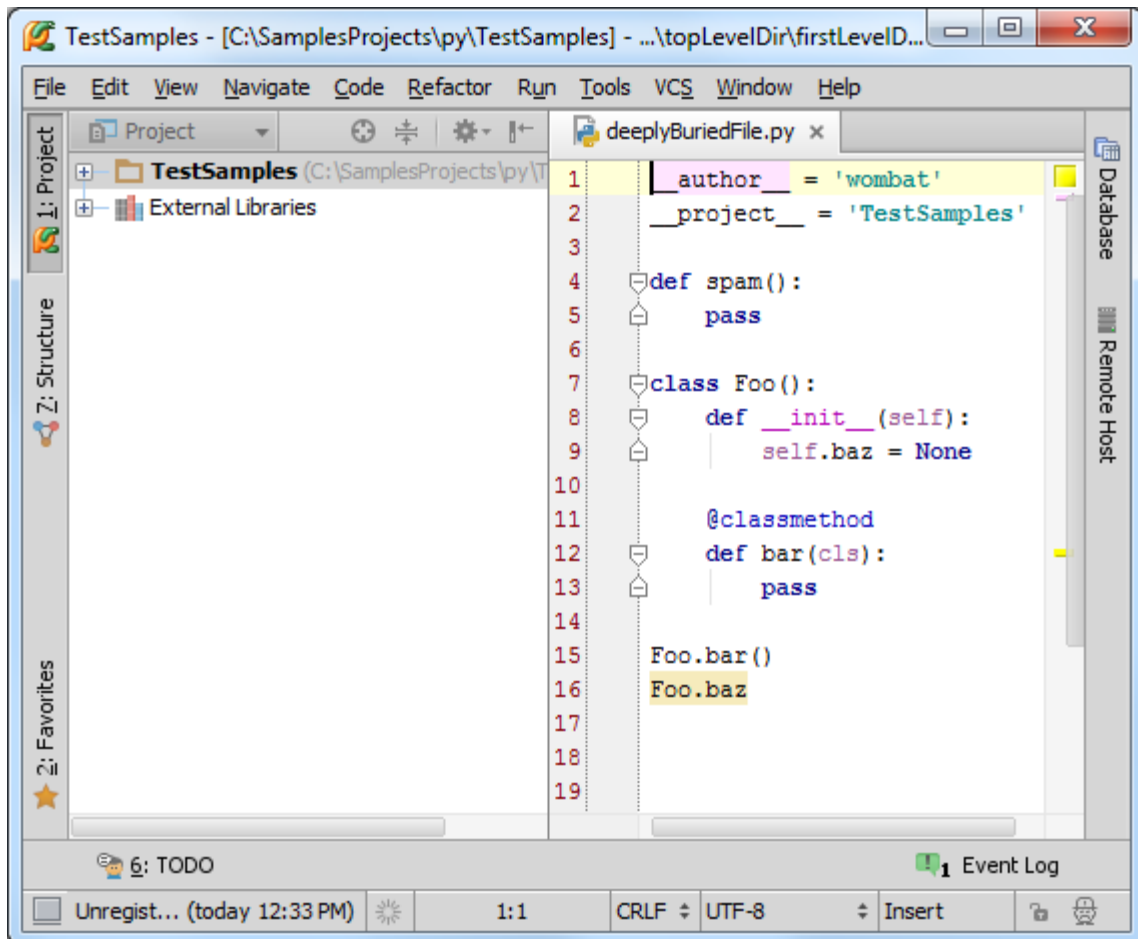
Pycharm 提供的一个很强大的功能就是能够根据名称跳转到任何文件、类、符号所在定义位置。

3、跳转到文件

按下 Ctrl+Shift+N 快捷键，在弹出的窗口中输入 d:

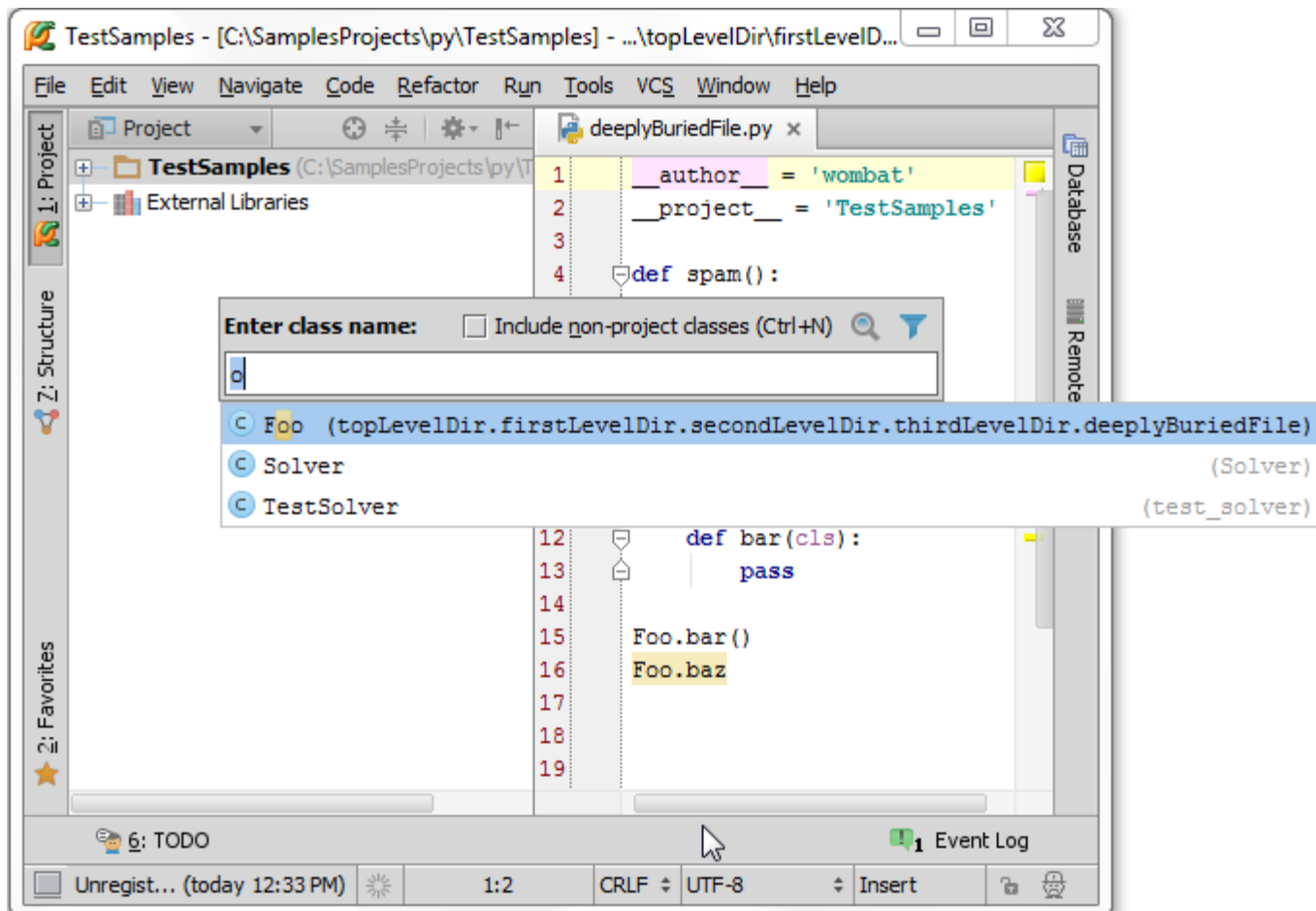


下面所显示的下拉列表中包含了所有名称中有字母 d 的文件。这里选择 `deeplyBuriedFile.py`，回车，打开对应的 `deeplyBuriedFile.py` 文件：

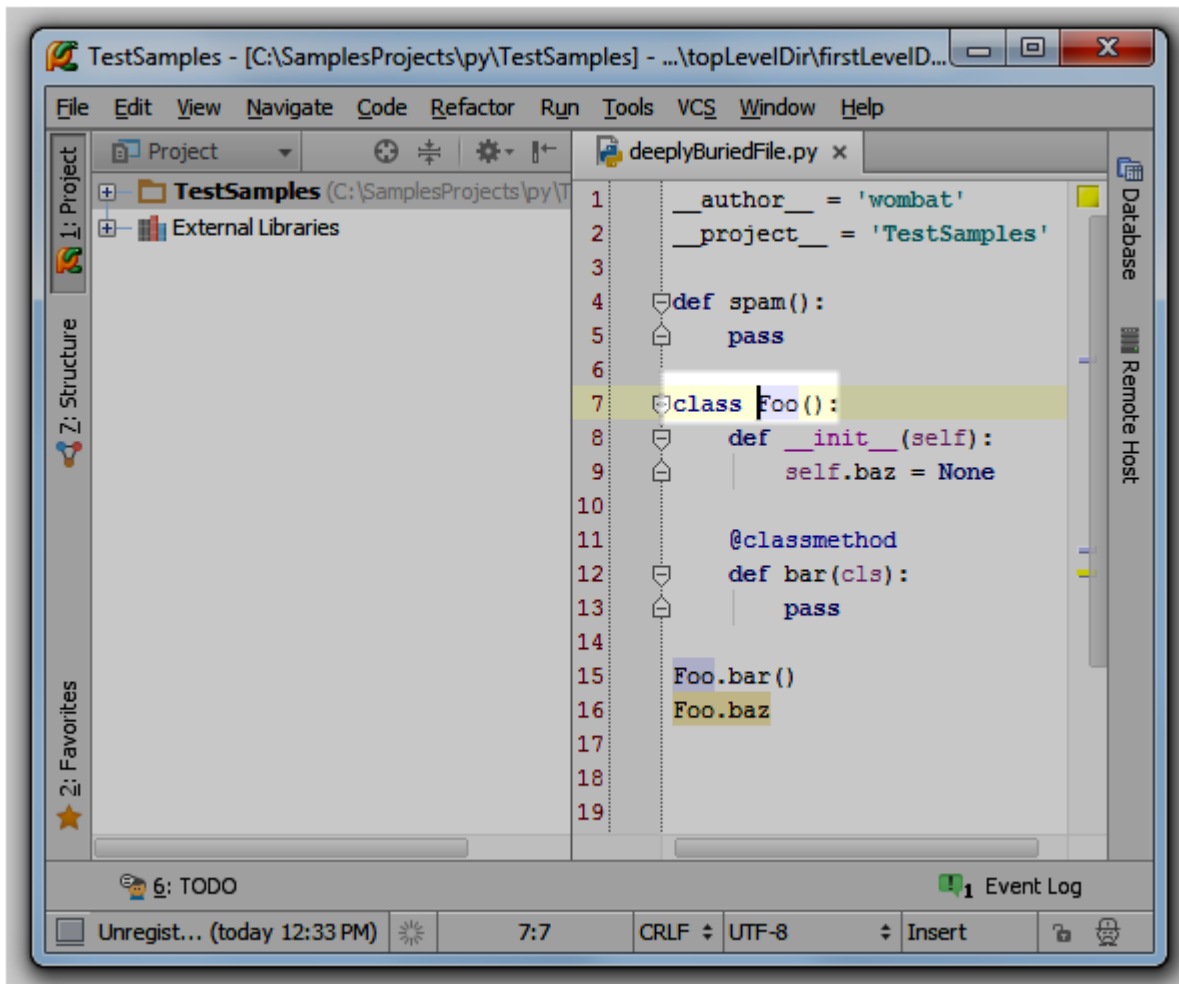


4、跳转到类

接下来我们尝试跳转到一个特定的类定义。按下 Ctrl+N，输入 o，注意这里你可以输入*来代表所有的文件名。在下拉列表中列出了所有名字中包含字母 o 的类，并且给出了对应的全路径：

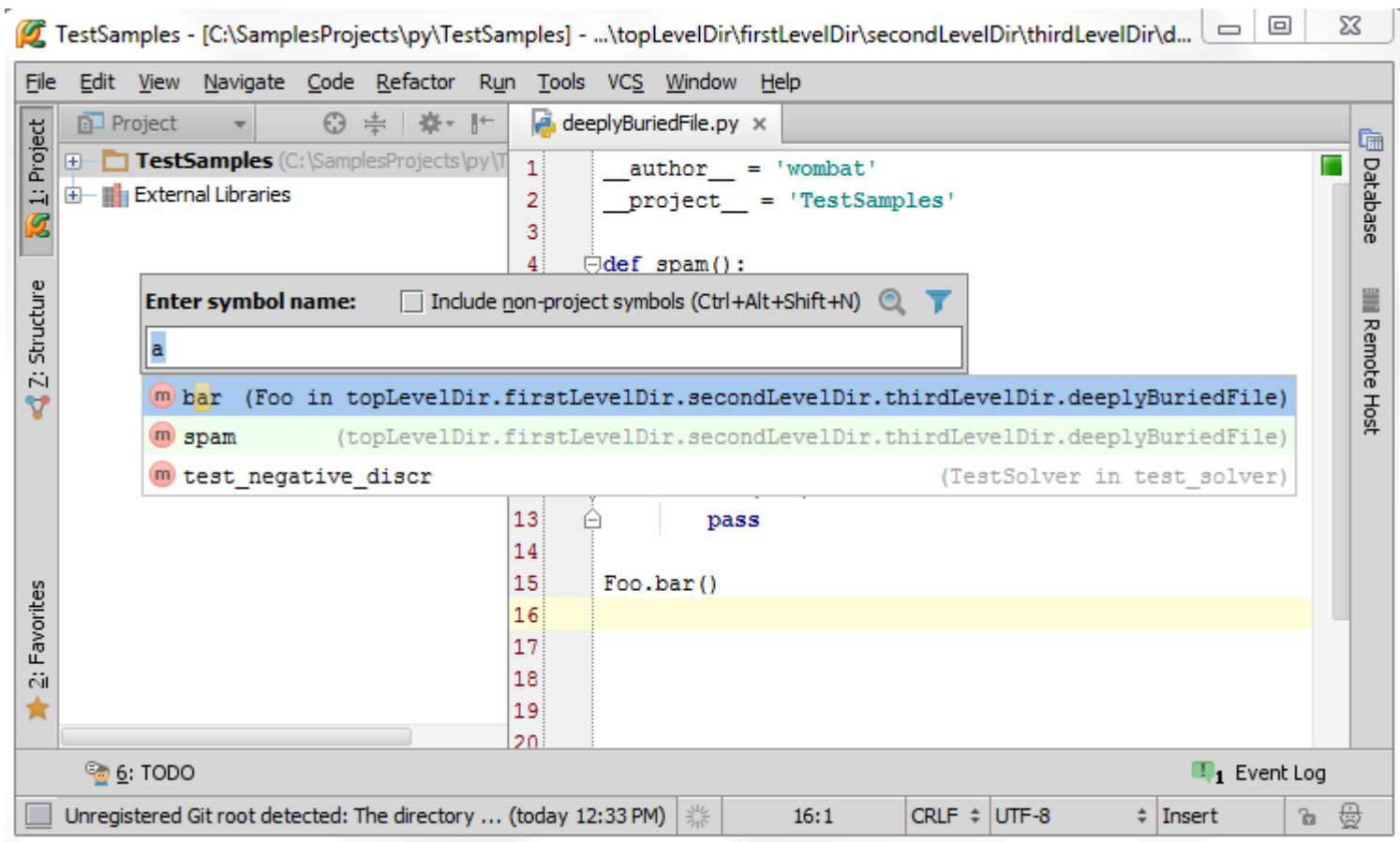


选择提示列表中的 Foo，回车，这次会在光标所在的类声明处打开所在文件：

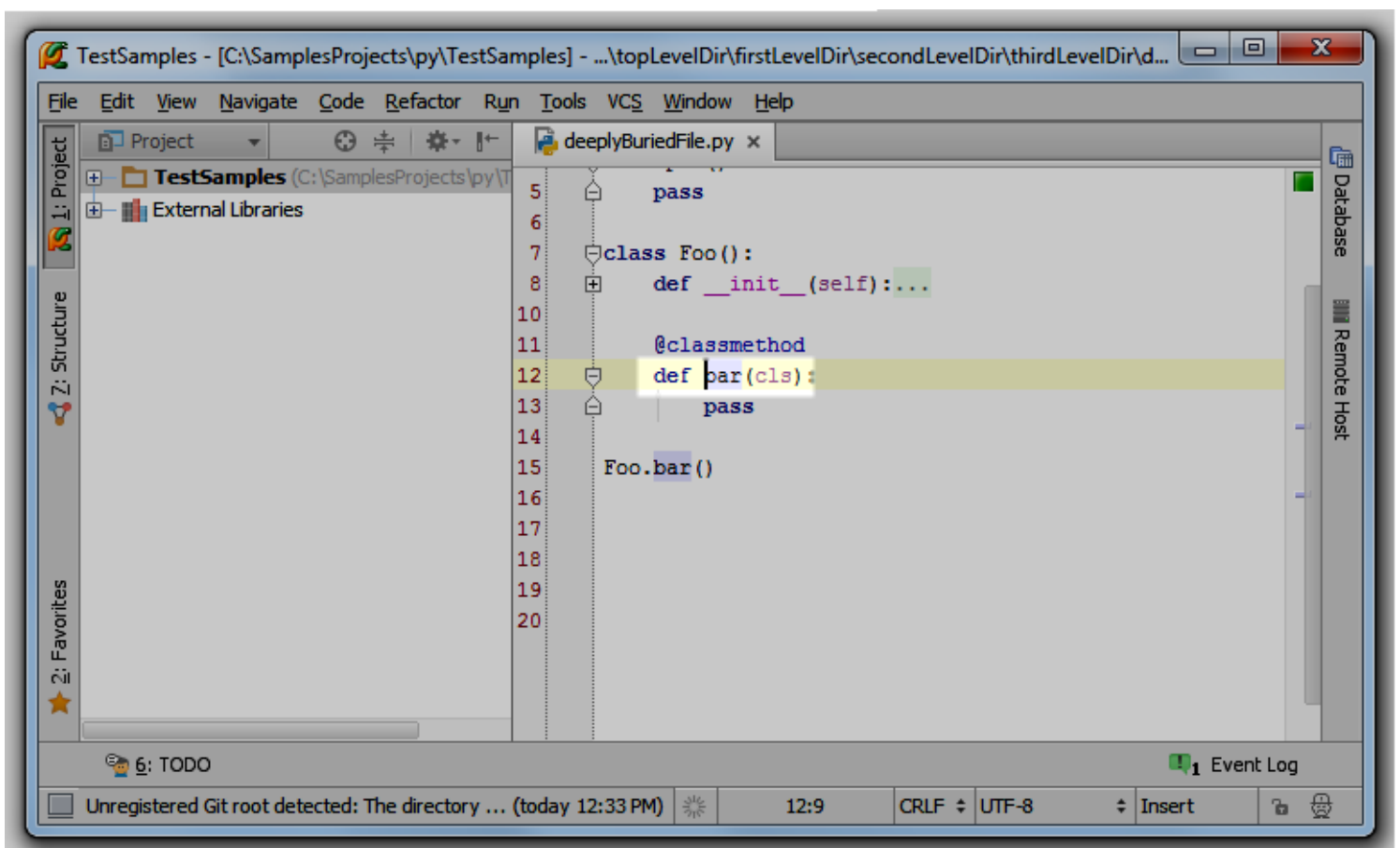


5、跳转到符号

接下来我们尝试跳转到类中一个特定的成员处。按下 `Ctrl+Alt+Shift+N`，输入 `a`：



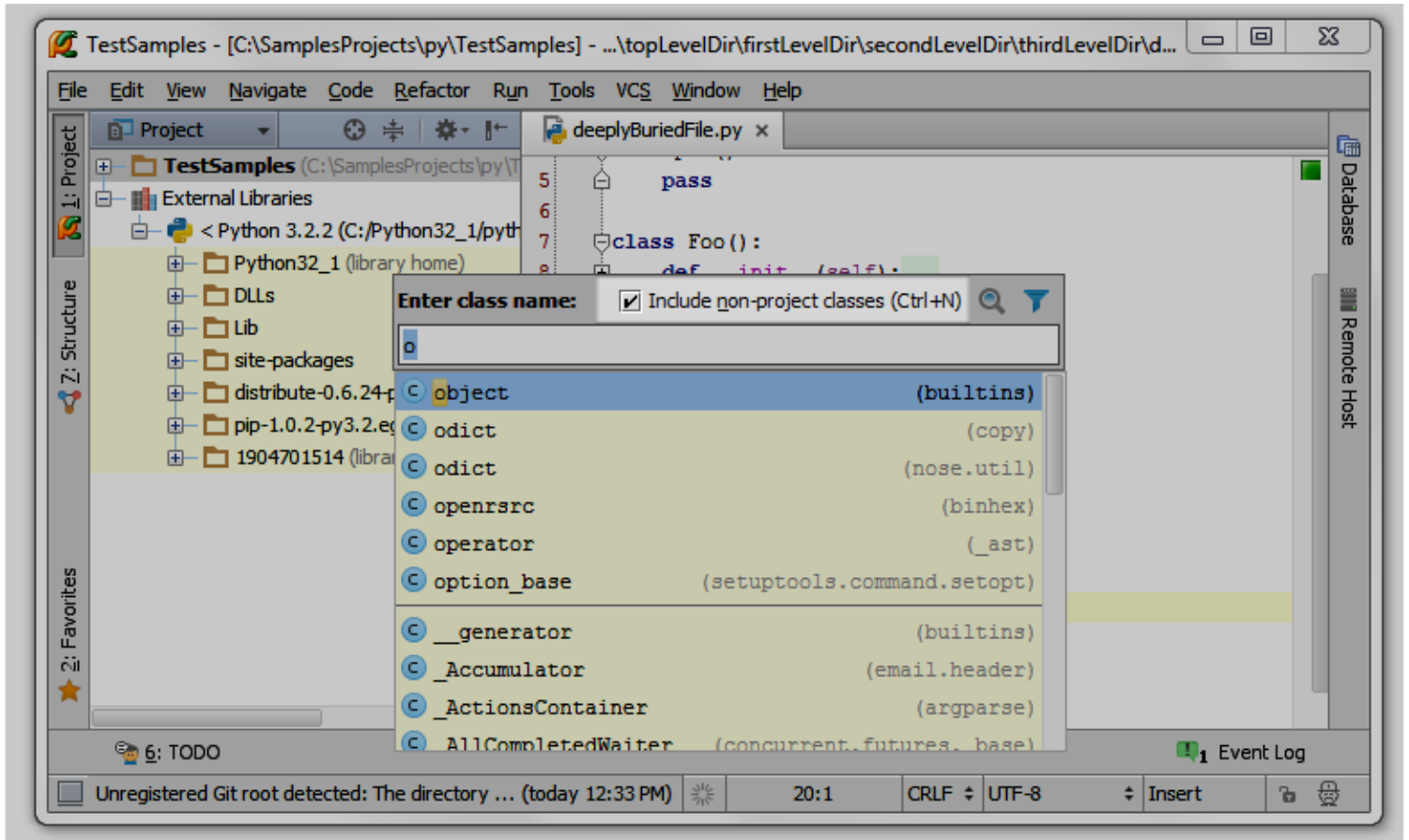
选择 bar，然后回车，转到对应定义：



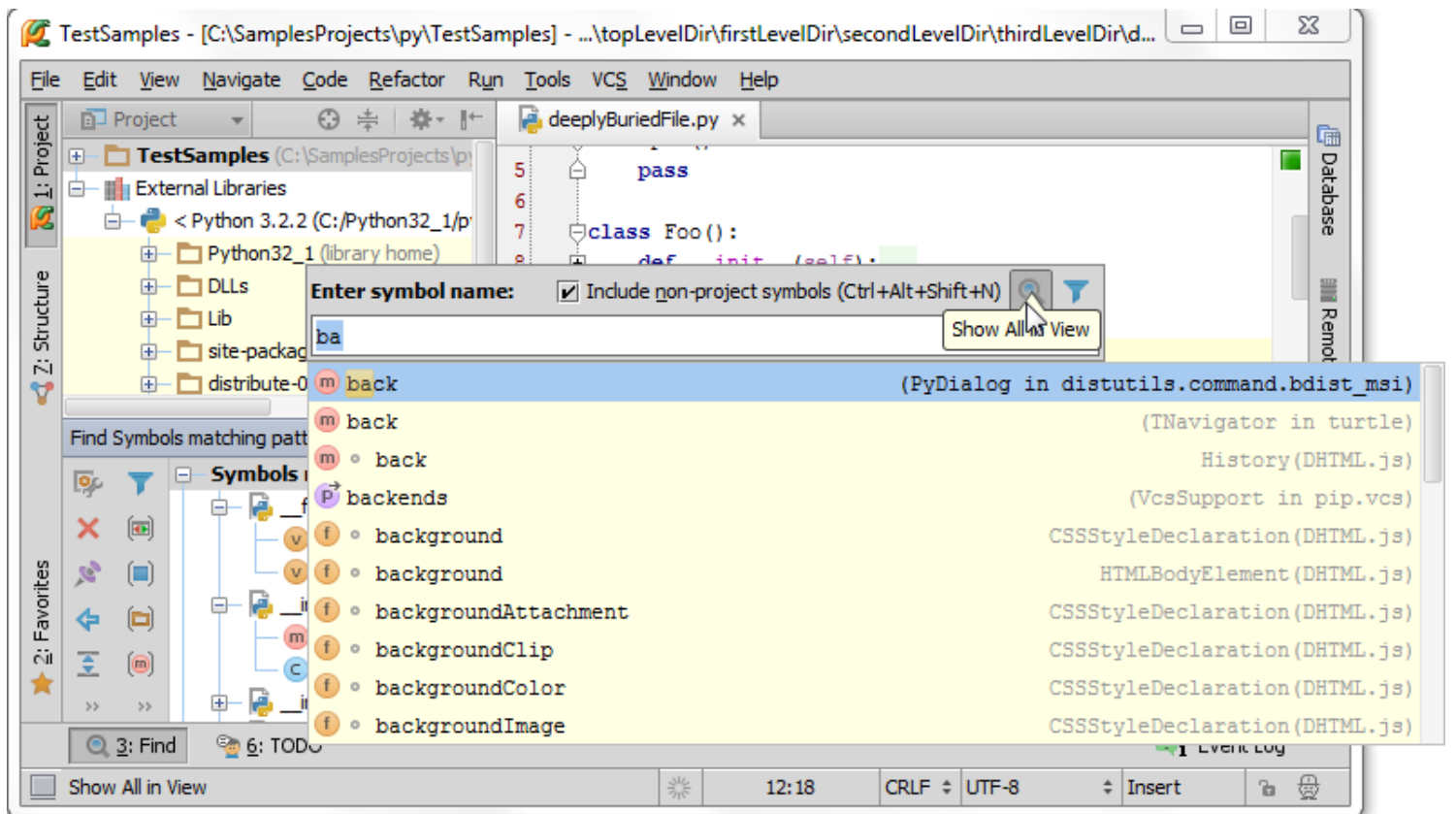
6、跳转窗口的额外功能

你可能注意到所弹出的窗口中还有其他控件为我们提供额外功能。

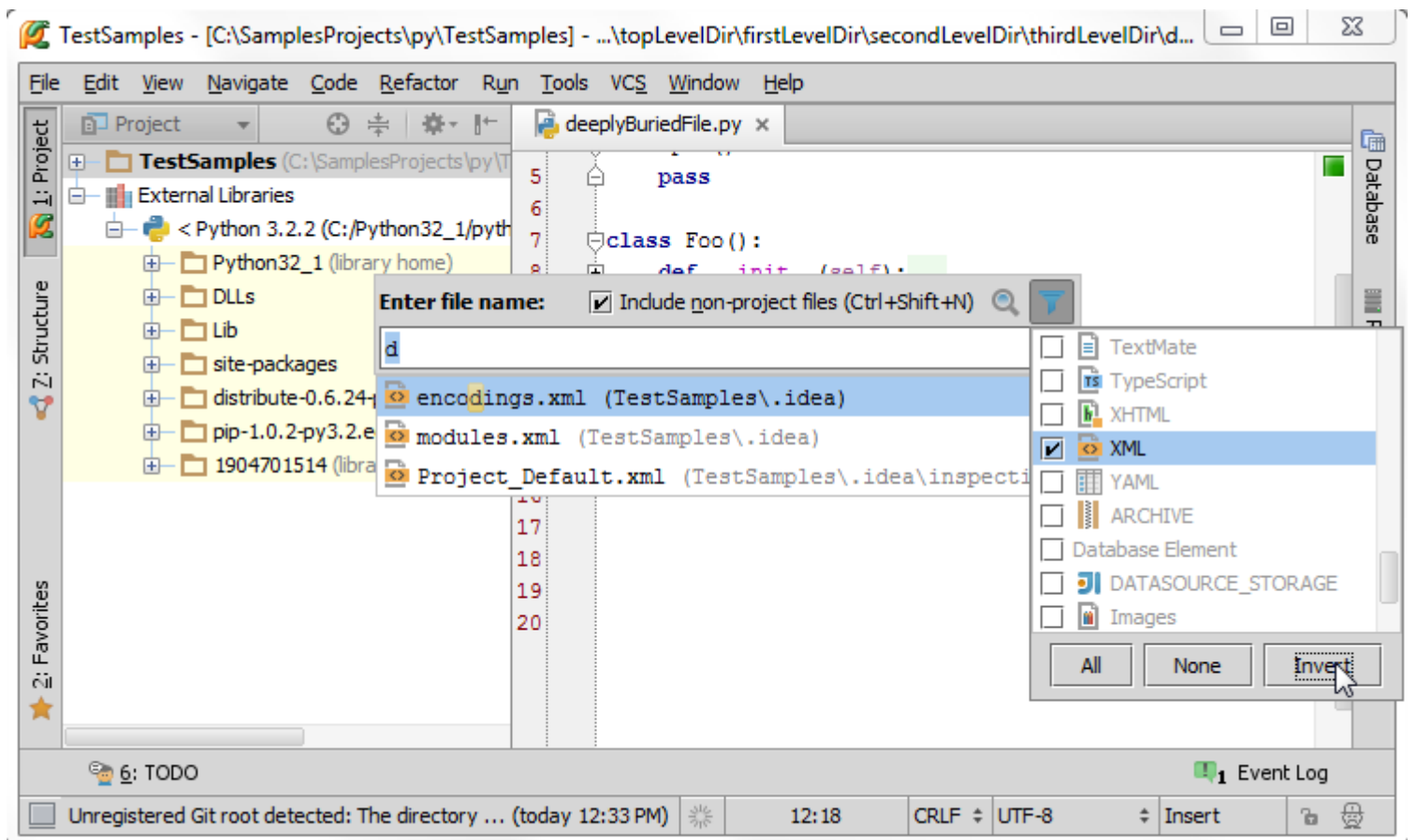
首先，我们介绍如何跳转到当前工程之外的文件、类、符号处。勾选 **Include non-project files/classes/symbols** 复选框或者按下相关快捷键以打开外部提示列表。与之前的 [suggestion list](#) 不同的是这个列表包含了外部库中的相应的符合搜索条件的项：



接下来拟可以在一个独立的窗口 [Find tool window](#) 来显示当前多匹配的条目，当你在进行多重搜索时，这种独立的显示模式变得很重要。同时我们也希望能够保存搜索结果方便下次查看：



最后，你可以通过单击提示列表旁边的漏斗图标来指定当前需要显示哪种文件类型的搜索结果：



最全 Pycharm 教程（27）——Pycharm 搜索导航之跳转到声明与定义

1、准备实例

(1) 在工程目录下创建 Animals 模块（Alt+Insert→Python Package）：

(2) 创建一个 Python 文件（Alt+Insert→Python File）：

在 Mammals.py 文件中输入以下代码：

```
from Animals.Carnivore import Carnivore
from Animals.Herbivore import Herbivore

class Mammalia(object):
    extremities = 4
    def feeds(self):
        print ("milk")
    def proliferates(self):
        pass
class Marsupial(Mammalia):
    def proliferates(self):
        print("poach")
class Eutherian(Mammalia):
    def proliferates(self):
        print("placenta")

class TasmanianDevil(Marsupial, Carnivore):
    pass

class Duckbill(Marsupial, Herbivore):
    pass

class Cat(Carnivore, Eutherian):
    pass

class Tiger(Eutherian,Carnivore):
    pass

class Cow(Eutherian, Herbivore):
    pass

Cat.feeds()
```

在 Carnivore.py, Herbivore.py 文件中输入以下代码：

```
from Animals.Mammal import Mammalia

class Carnivore(Mammalia):
    def food(self):
        print("meat")

    pass
```

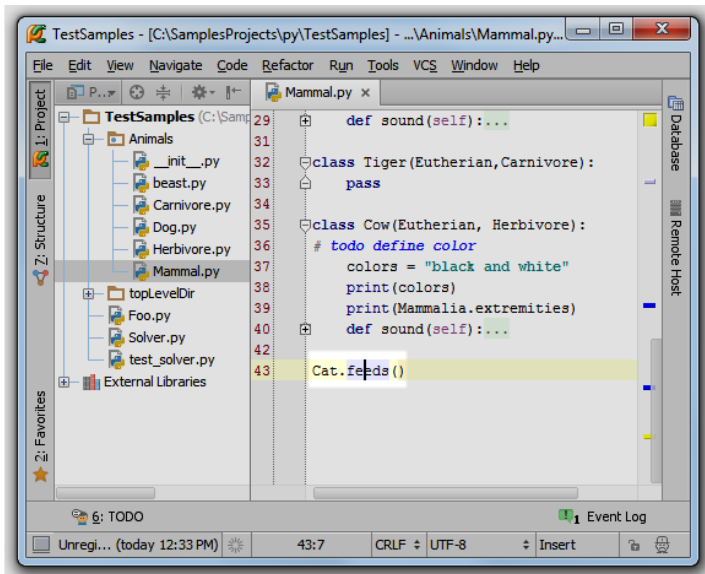
```
from Animals.Mammal import Mammalia

class Herbivore (Mammalia):
    def food(self):
        print("grass")

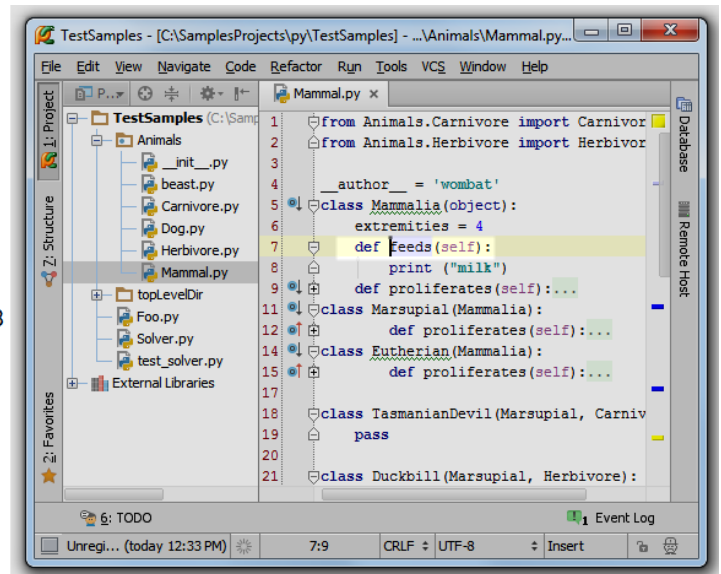
    pass
```

2、转到声明

将光标定位在 Cat 类实例的 feeds () 函数名处，按下 Ctrl+B。Pycharm 会自动跳转到 Mammalia 类 feeds 成员的定义：

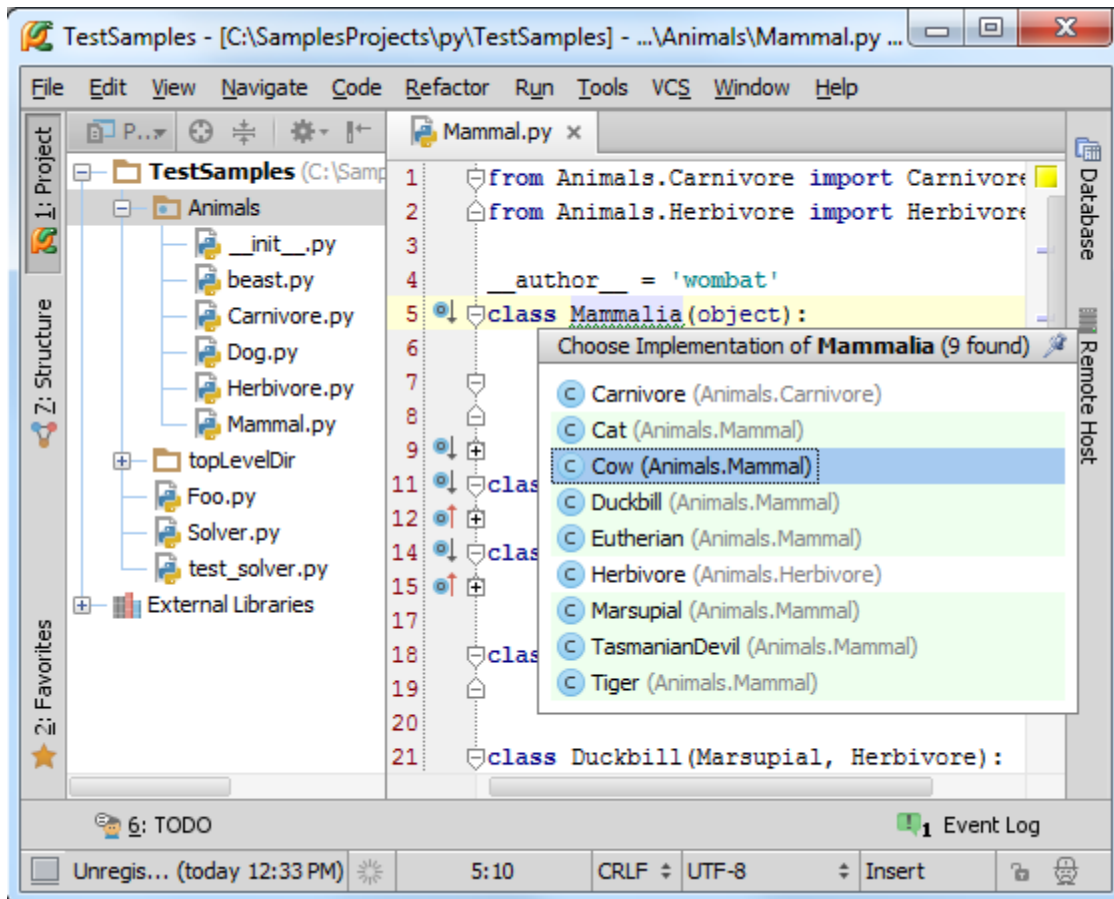


Ctrl+B

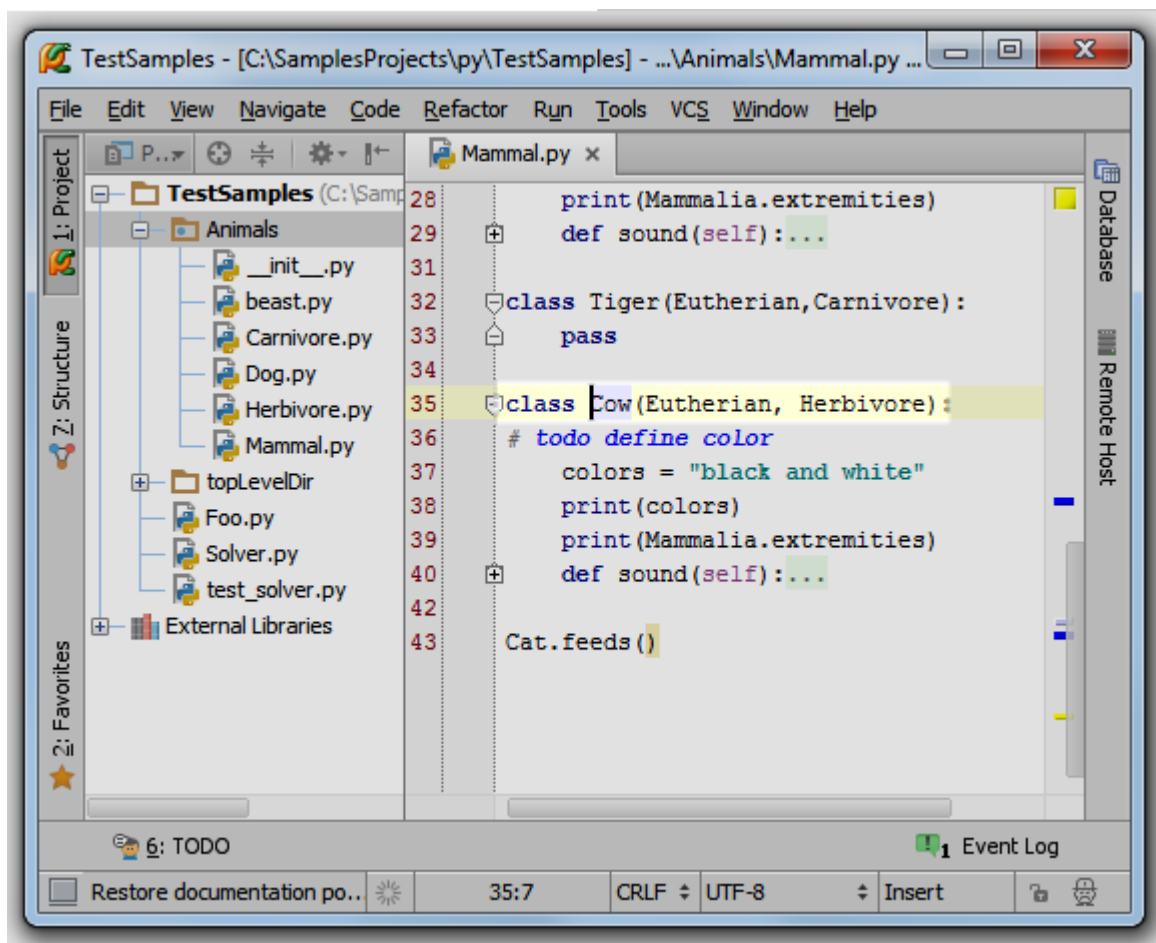


3、转到引用

接下来吧光标至于 Mammalia 类名称处，查找其所有引用。按下 Ctrl+Alt+B，会看到 Mammalia 类的引用列表：



选择你想查看的引用（例如这里选择 Cow），回车，Pycharm 会跳转到对应引用位置：



如果你选择了位于另外一个文件中的引用，如 Carnivore，Pycharm 会在一个单独的编辑选项卡中打开它。

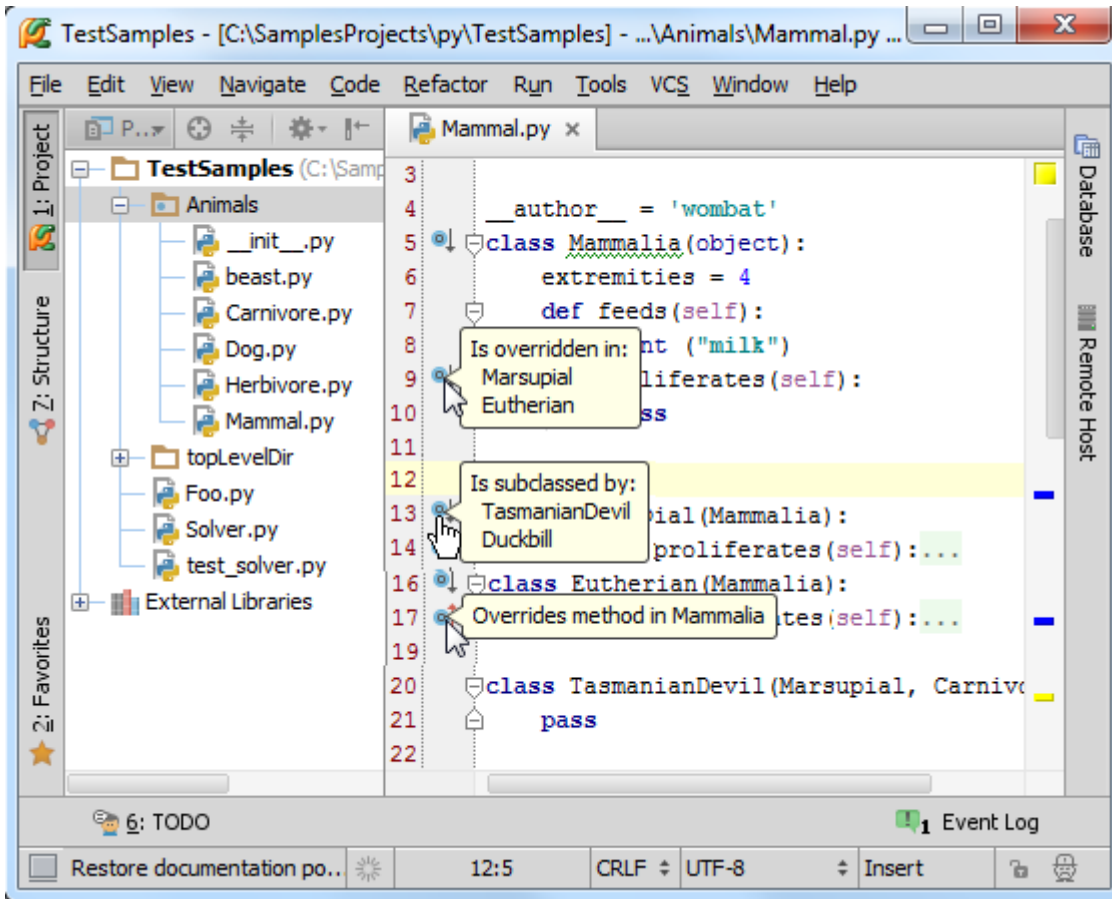
4、注意侧边图钉图标

之前我们已经预见类似的图钉图标，例如在查看快捷文档时（Ctrl+Q），如果你单击这个图标，则整个弹出窗口会被固定。在这里则意味着所遇到的引用已经添加到 Find tool window 窗口中进行显示。

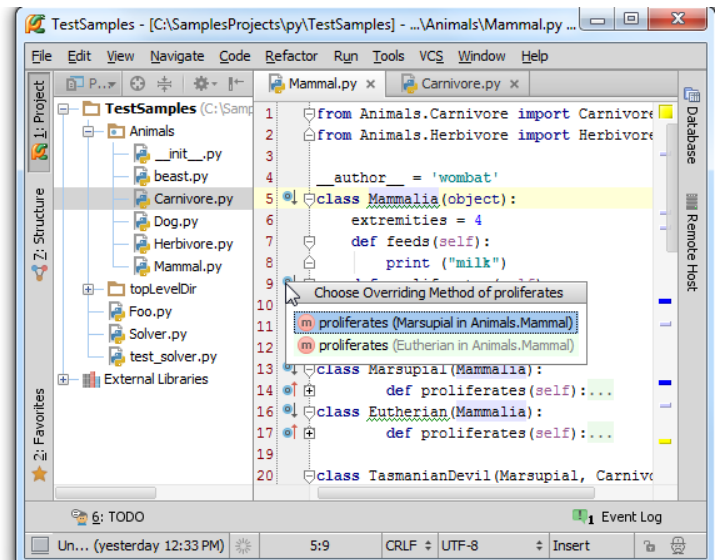
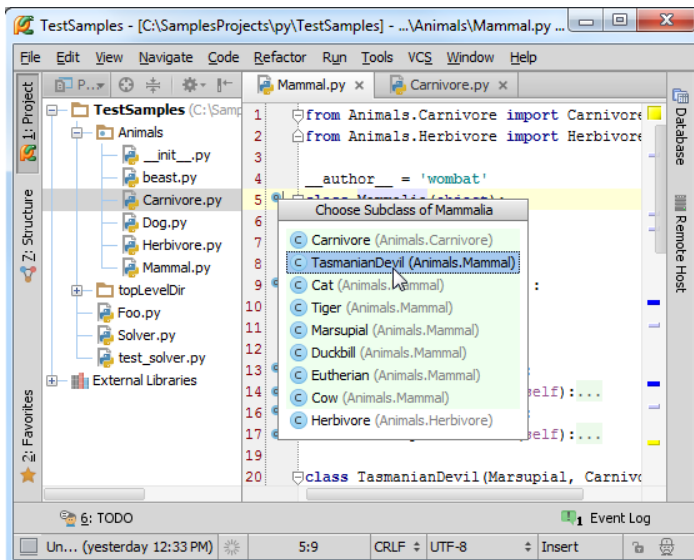
5、左槽图标的导航功能

最后我们观察窗口左槽，这里有很多附有指向箭头的图标，意味着？

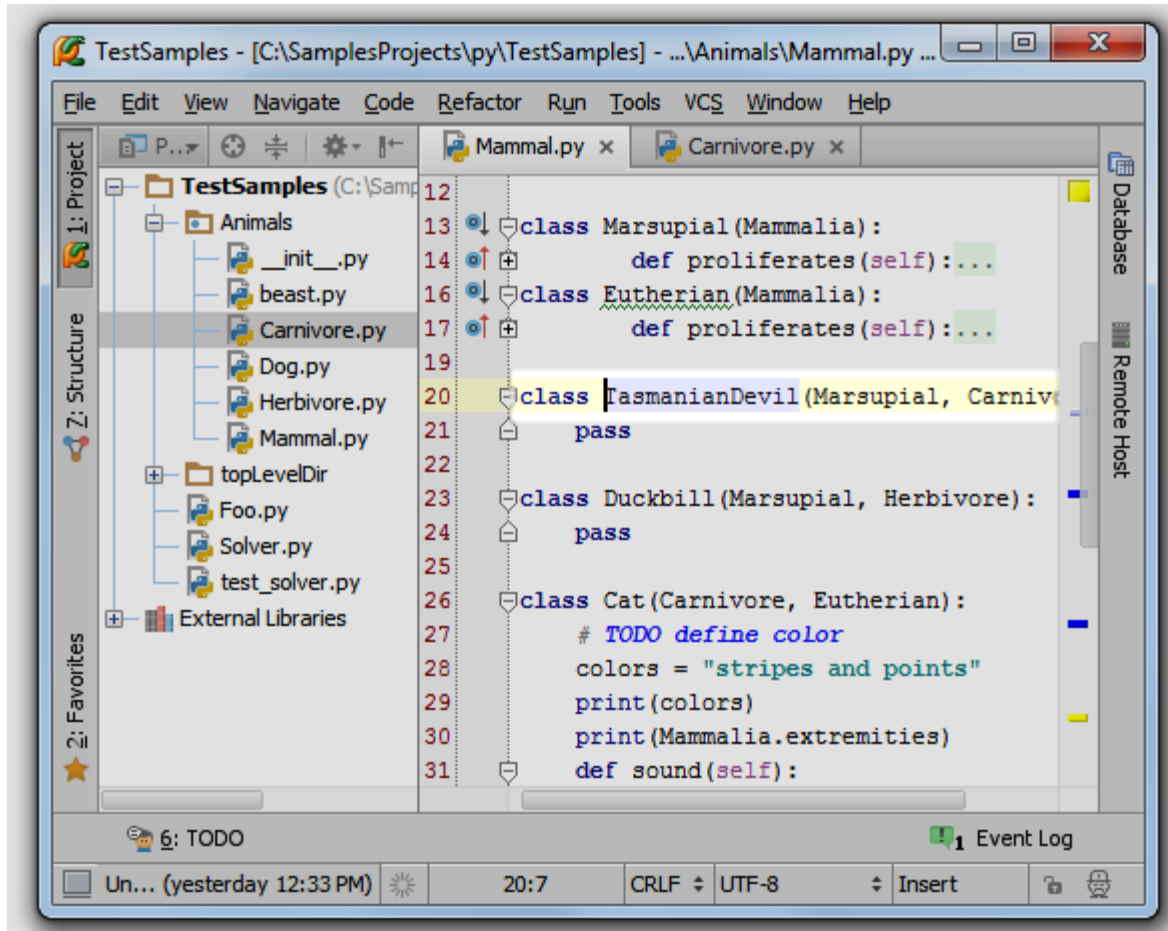
如果你将鼠标指针悬停在图标上面，Pycharm 会显示其子类或者重载方法（向下的箭头）、父类（向上箭头）：



当你单击这个图标时，Pycharm 会列出其子类或者重载方法（如果有的话）：



之后 Pycharm 会跳转到所选条目，并将光标至于对应的声明部分。如果只存在一个子类、父类或者重载函数，则默认执行跳转操作：



最全 Pycharm 教程（28）——Pycharm 搜索导航之搜索应用实例

1、主题

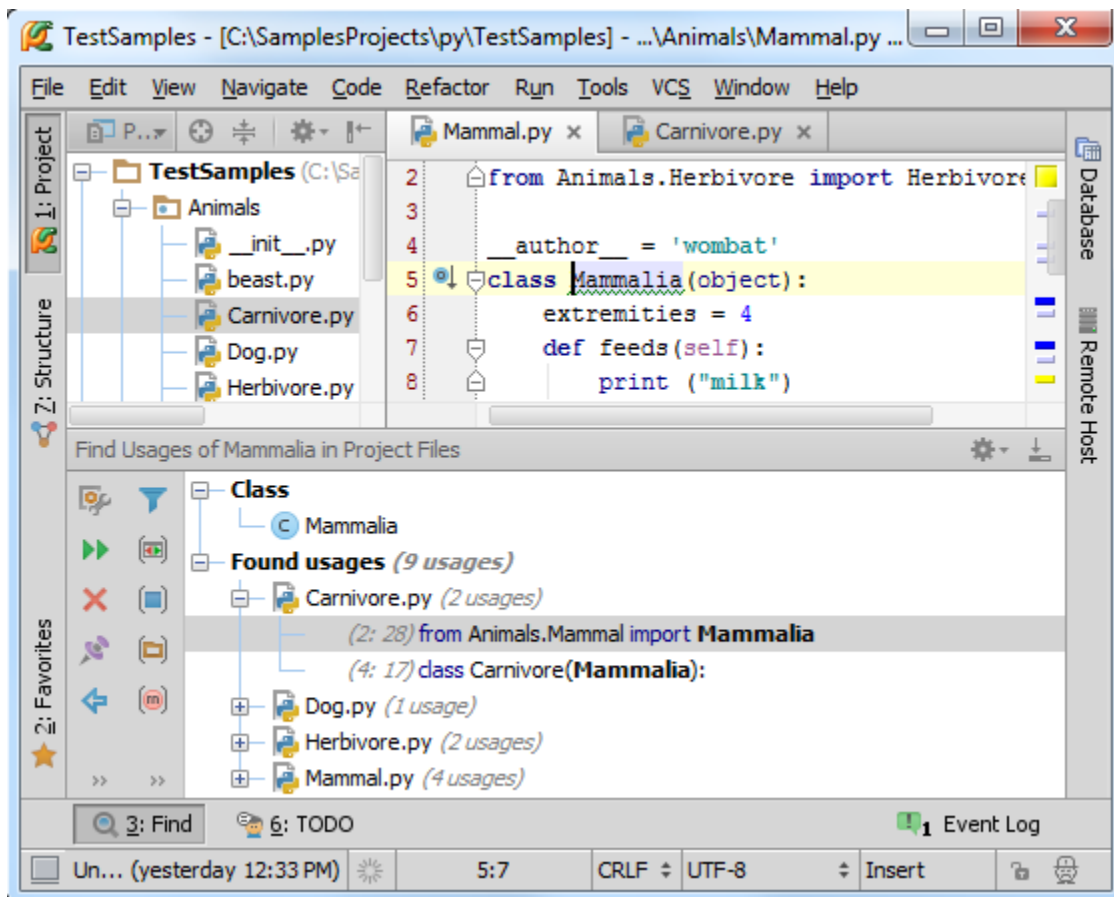
这里我们将介绍 Pycharm 另外一项强力的搜索导航功能。假设你希望知道某个特定的类或方法都在工程中的哪些地方发挥了作用，也就是找出其所有的 usages，这将是一个非常巨大而繁琐的工程，不过这里 Pycharm 会帮助我们实现。

2、准备工作

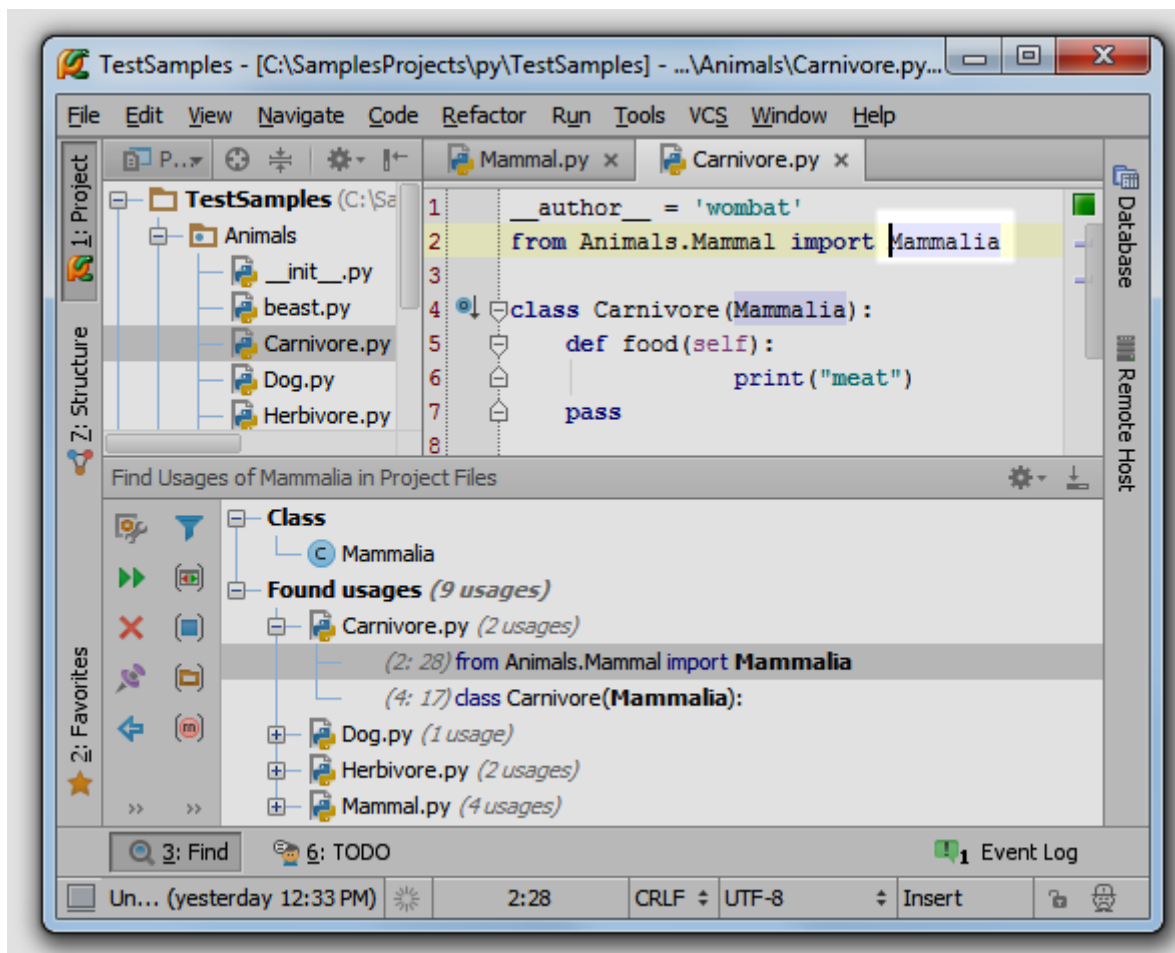
使用上一篇博客的 Animals 模块。

3、搜索所有引用 usages

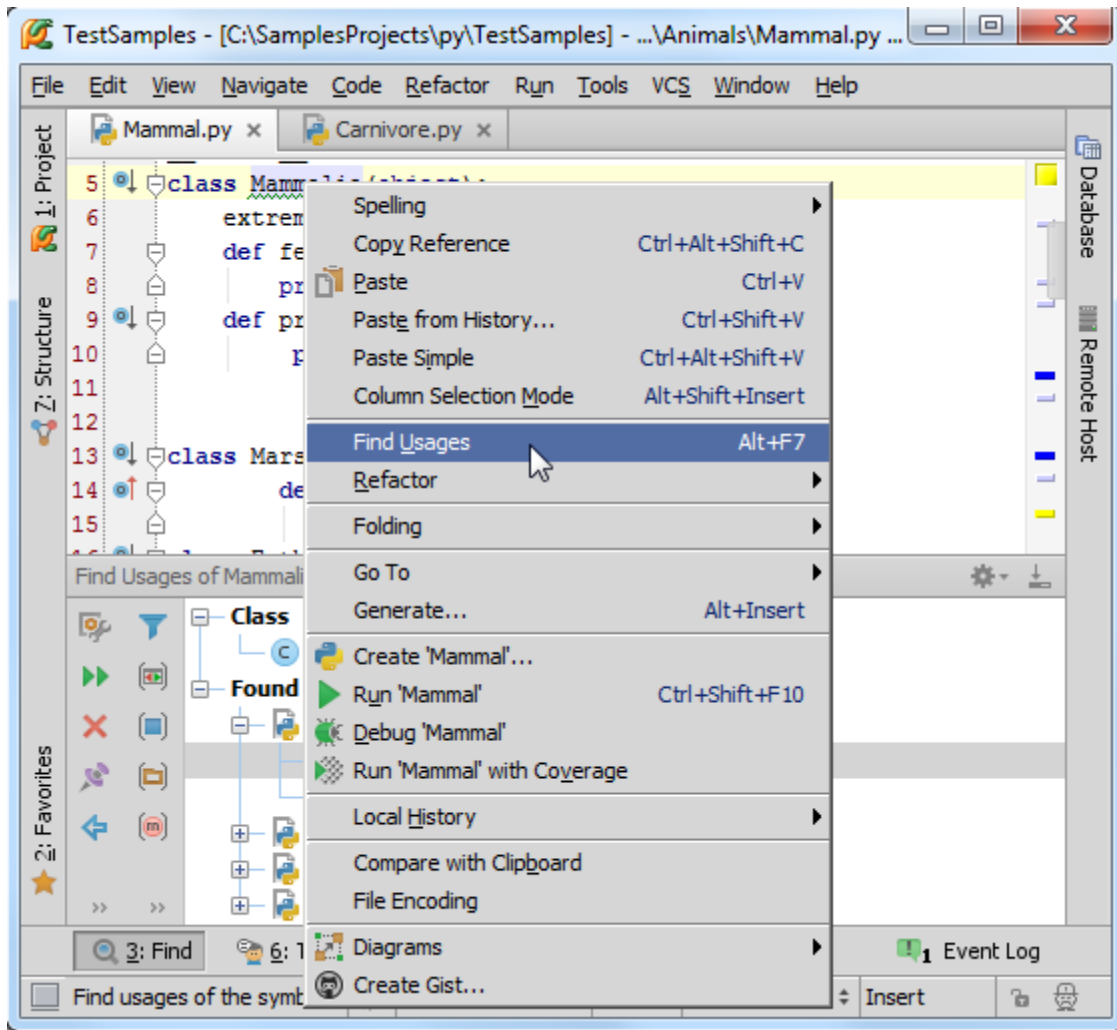
假设我们希望找出 Mammalia 类的所有使用环境，然后跳转到其中一处。将光标定位在类声明处，按下 Alt+F7，在 [Find tool window](#) 窗口中显示当前类的 usages：



选择了一处引用之后，回车，Pycharm 会打开相关文件：



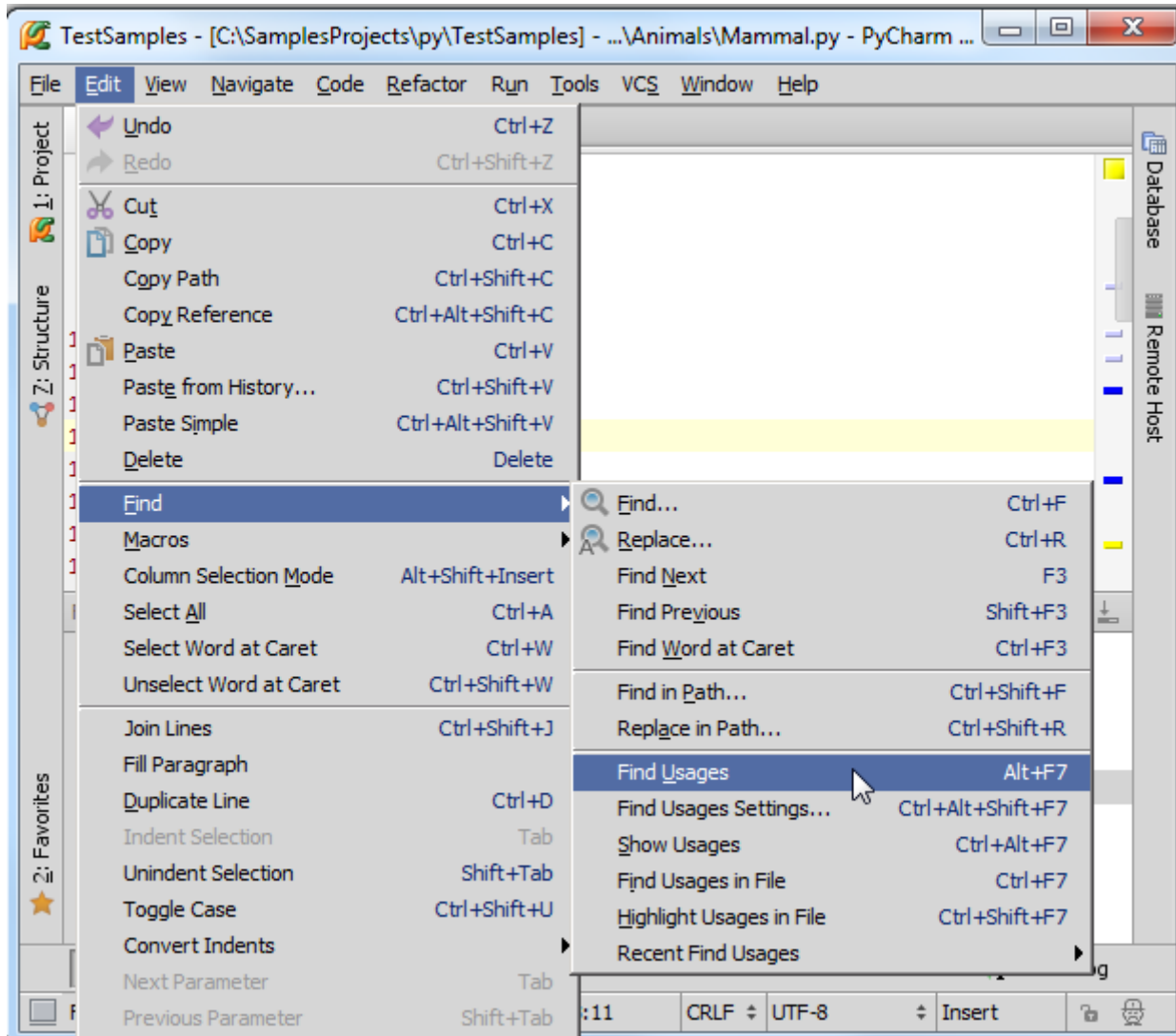
顺便提一句，你可以通过快捷菜单来实现这个功能，例如在 Mammalia 右击，观察弹出的快捷菜单内容：



通过这种方式你可以在默认设置下找到某个符号的所有 usages。

4、其他查询方式

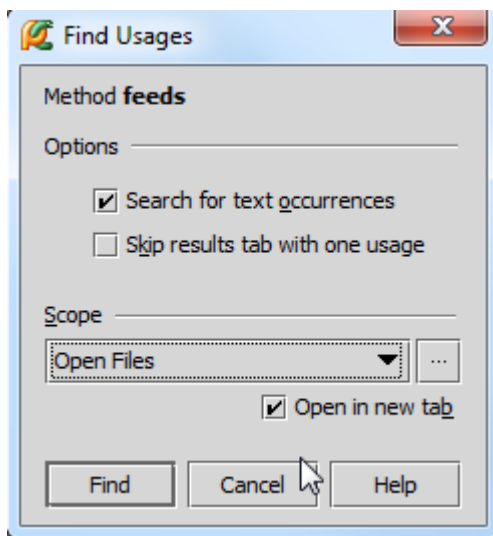
处理查找 usages，Pycharm 还提供了几种其他的查找方式，操作相似，目的相同，请看主菜单（Edit → Find）：



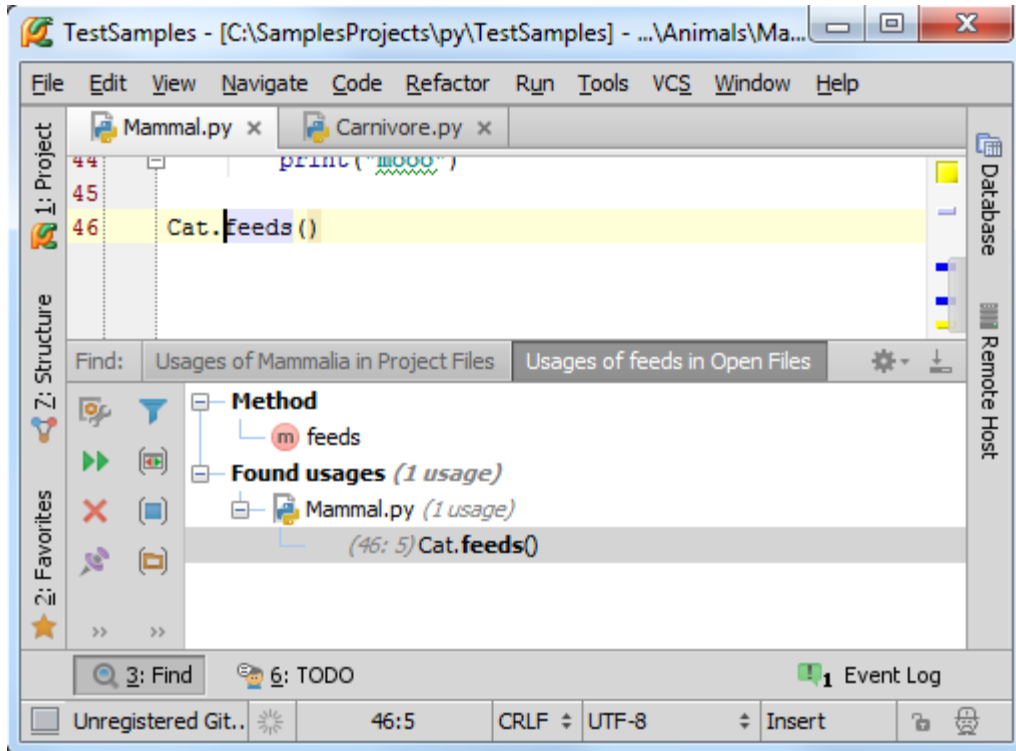
这些操作一部分已经制定了相关快捷键，接下来我们详细介绍。

5、改变搜索模式，通过对话框查找 usages

将输入光标置于符号声明处，例如函数名 `feeds`，按下 `Ctrl+Alt+Shift+F7`，弹出对话框 `dialog box`，在这个对话框中可以更改搜索选项。例如希望在当前打开的文件范围内来搜索，并且在新的编辑框中显示搜索结果：



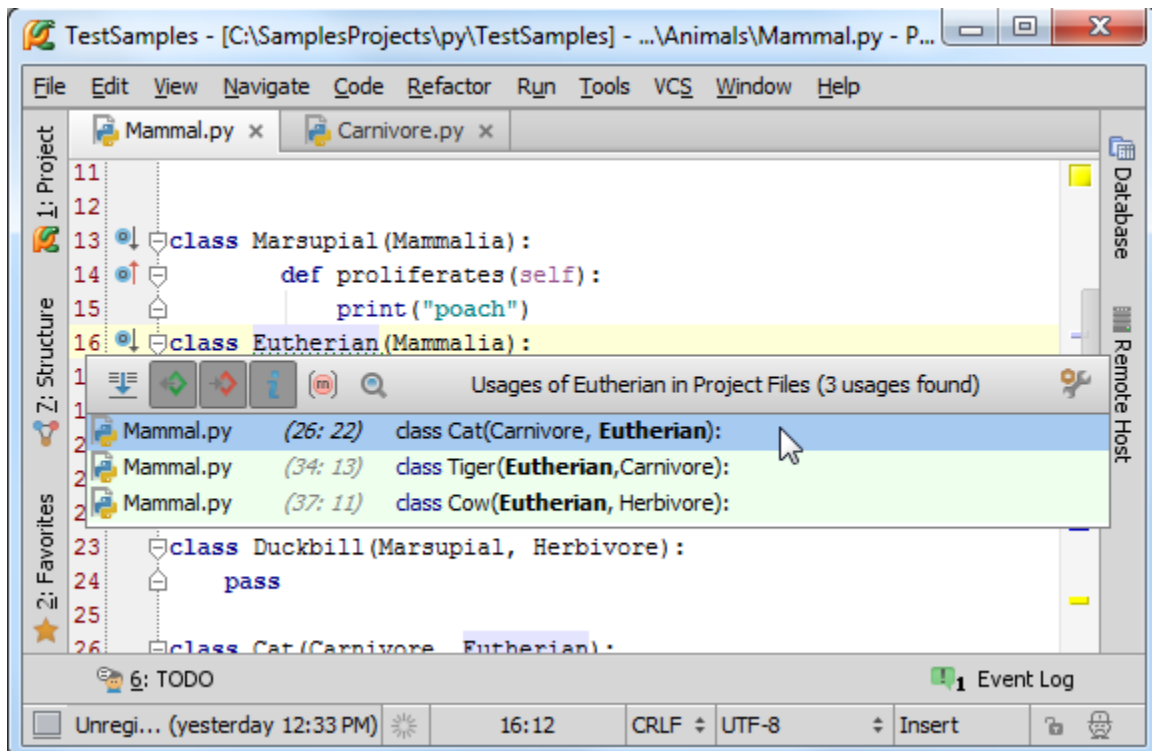
单击 Find 按钮：Pycharm 会通过一个新的编辑选项卡来显示 feeds 的搜索结果。双击（或者使用方向键选中并回车），Pycharm 打开相关文件：




6、以列表形式显示 usages


在某些情况下通过搜索窗口来查看搜索结果并不是很方便，Pycharm 允许将搜索结果以弹出列表的形式反馈出来。例如我们想查找 Eutherian 类的所用使用环境：

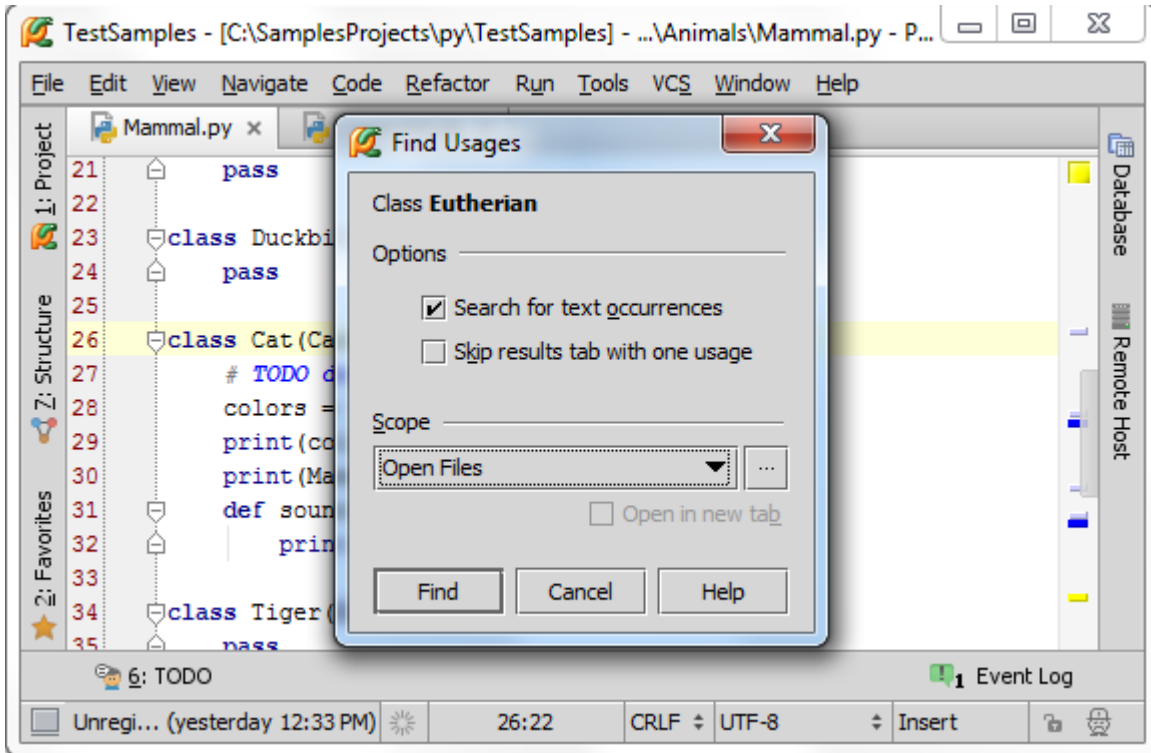
将光标置于类声明处，按下 Ctrl+Alt+F7（或者使用 Edit→Find→Show Usages 菜单命令）：



如果你通过方向键选中对应条目（例如 Cat 类）然后回车，Pycharm 会跳转到相关文件。

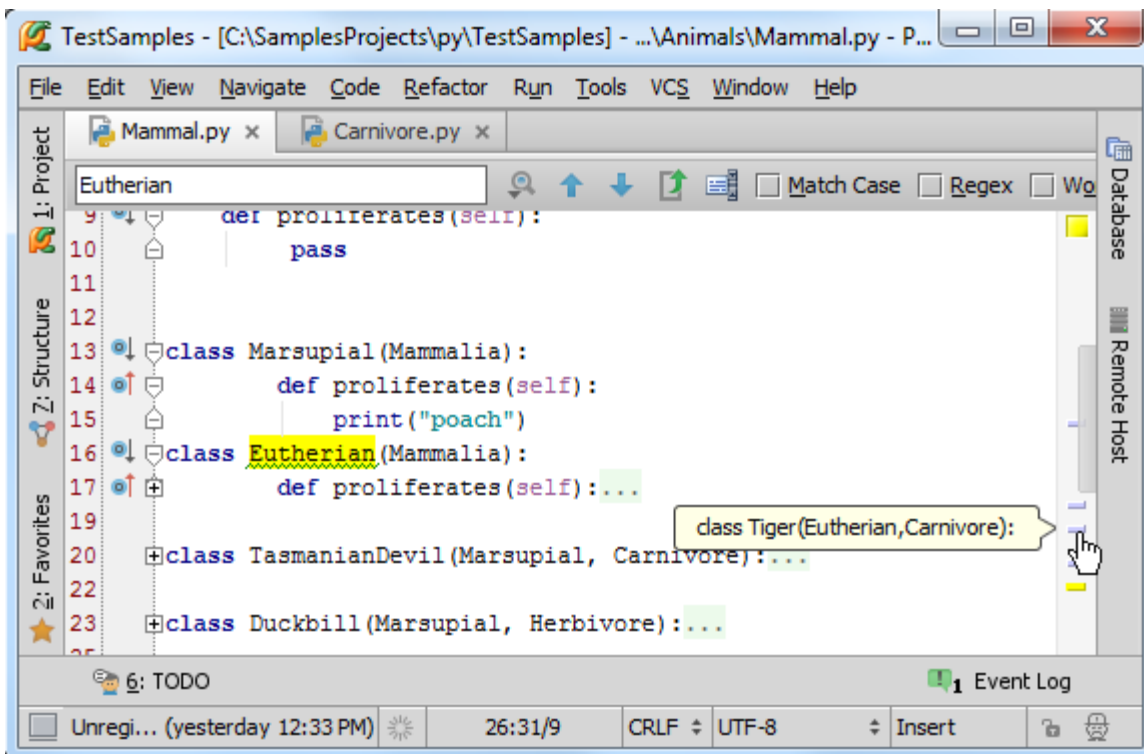
现在回到弹出列表窗口，如果你想恢复到之前的 Find tool window 模式，只需单击  即可。

最后，如果你对当前的搜索模式不满意，单击  按钮打开设置对话框 `dialog box`：



7、在当前文件中查看 usages

更简单的，只需按下 `Ctrl+Shift+F7`，或者使用 `Edit→Find→Highlight Usages` 主菜单命令，问题解决：



正如你所见，每个 usages 在右槽对应都有一个标记，当将鼠标指针悬停在对应标记上时，Pycharm 会给出简要的提示声明。单击这些标记来实现不同 usages 间的切换。

最全 Pycharm 教程（29）——再探 IDE，速成手册

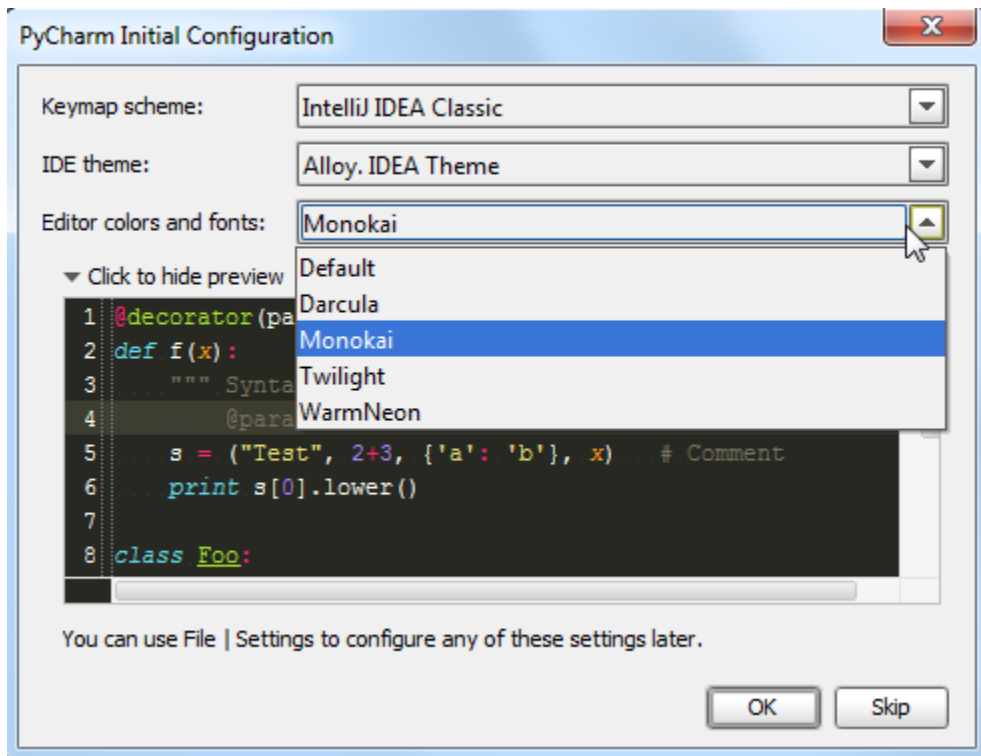
1、准备工作

- (1) 确认安装了 Python 解释器，版本 2.4 到 3.4 均可。
- (2) 注意 Pycharm 有两个发布版本：社区版和专业版，详见 [Edition Comparison Matrix](#)

2、初始化安装

第一次安装 Pycharm 时，安装程序会咨询你几个重要问题：

- (1) 是否已经预先保存了设置信息（例如早期版本的配置信息）
- (2) 许可证信息
- (3) 选择何种快捷键配置和背景主题



注意这里 Pycharm 预设了好几种快捷键方案，有诸如 Eclipse 或者 Visual Studio 的，也有针对 Emacs 粉丝量身打造的 GNOME、KDE 等等。具体参见设置对话框中 [Keymap page](#) 页的快捷键方案列表。

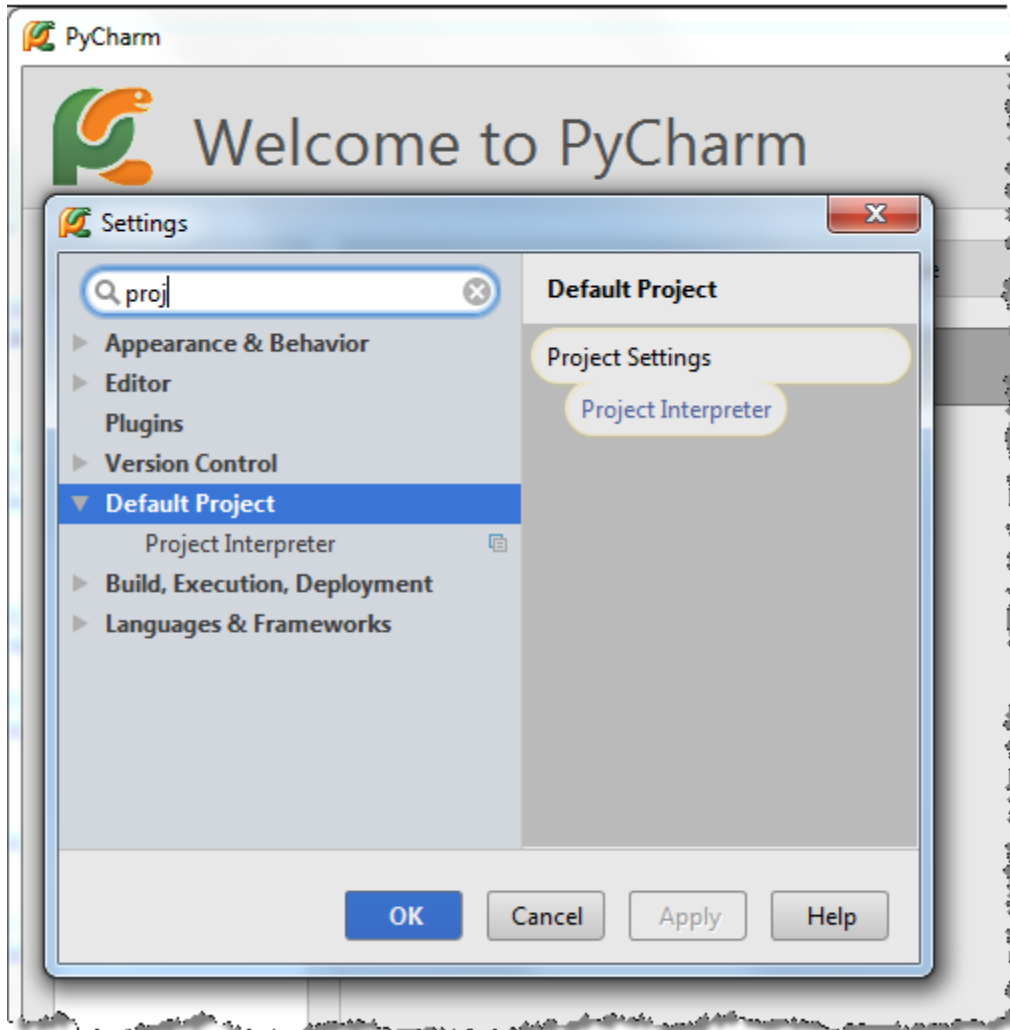
对于 Vim 专业户，PyCharm 建议使用 [IdeaVim plugin](#) 插件。同样对于那些习惯 Emacs 开发的用户，Python 同样提供了相关外部插件 [use it as an external editor](#)。

当然我们可以在后期对初始设置进行更改，详见 [documentation](#) 以及以下两篇教程：

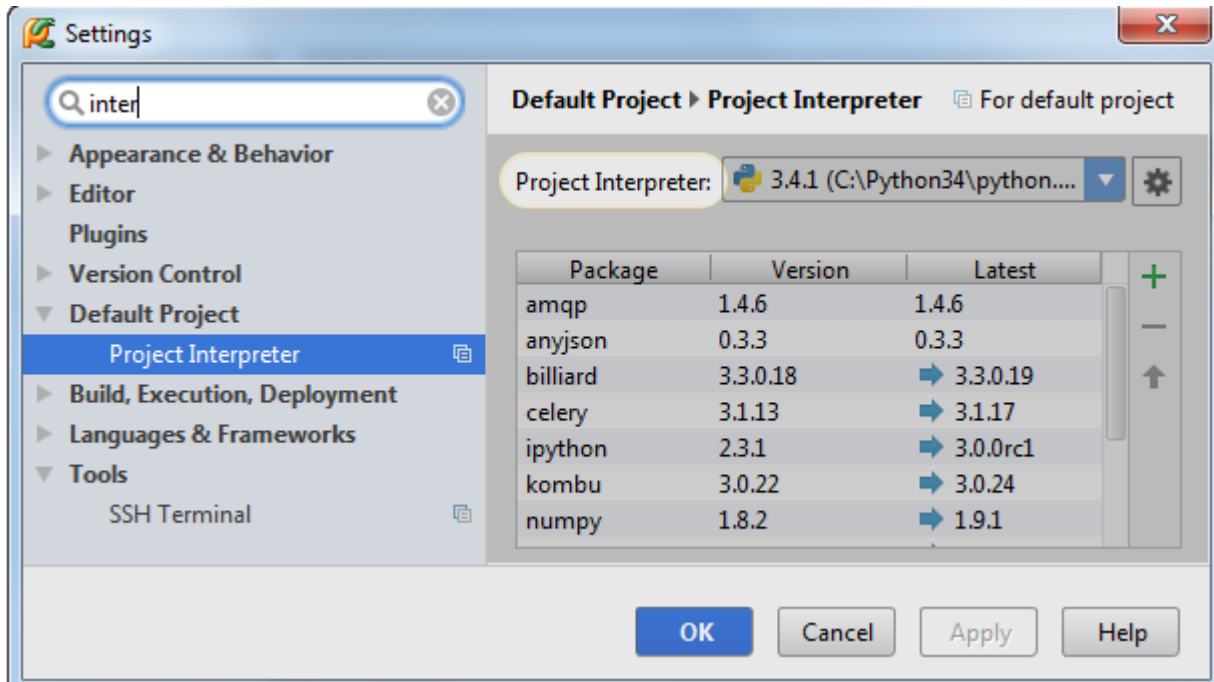
- [Getting started with PyCharm](#)
- [Configuring keyboard schemes](#)

3、欢迎界面设置

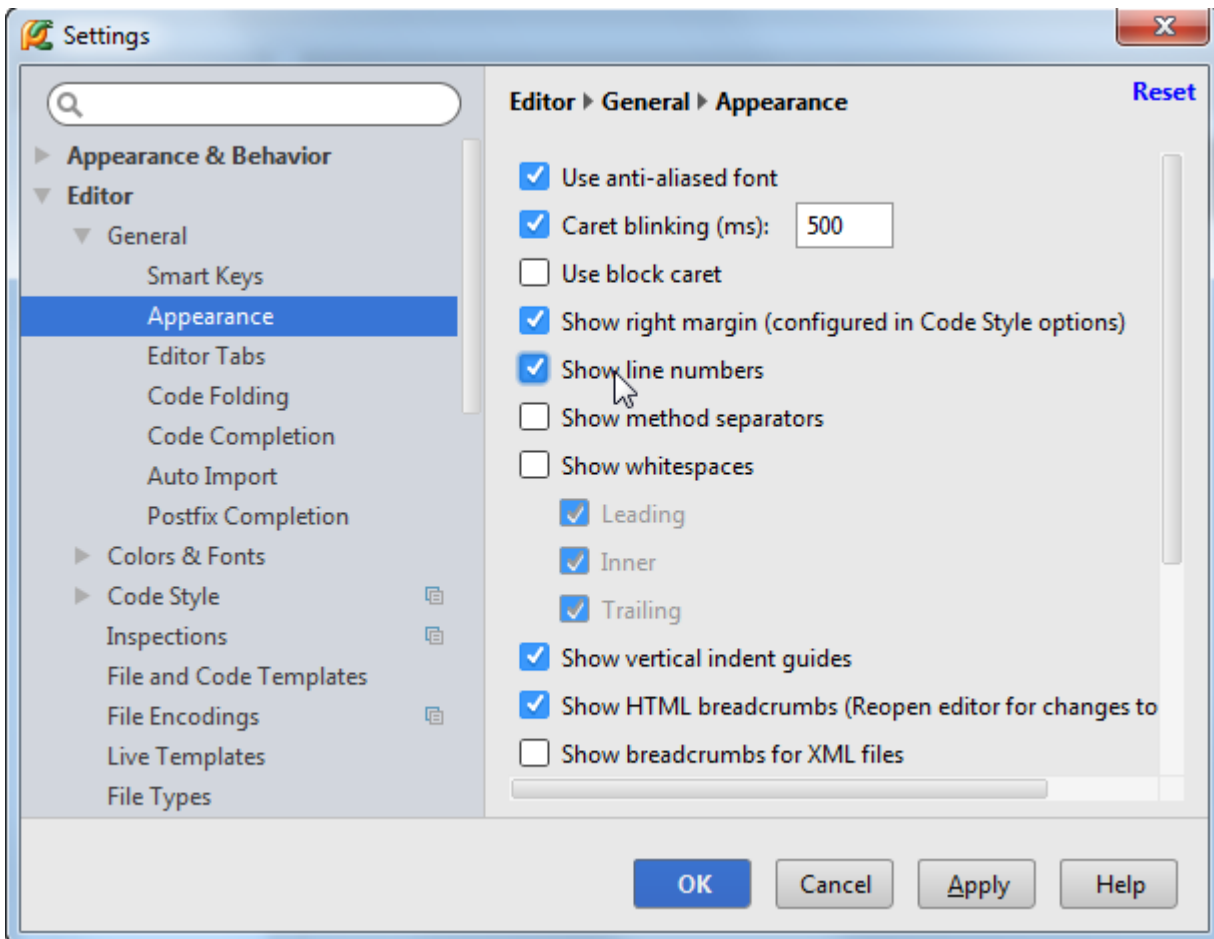
如果你第一次使用 Pycharm (尚未创建工程), 则你会首先进入欢迎界面 [Welcome screen](#)。单击 [Configure](#), Pycharm 会提示你来核实当前有关环境、插件、导入导出以及其他相关的外部配置。再次单击 [Configure](#), 进入 [Settings/Preferences dialog](#) 配置对话框, 注意这里对话框标题默认为“Default Project”:



这就意味着每次你创建新的工程时都会默认使用如上配置。假设你希望所有新创建的工程都使用相同的解释器, 可以在 [Default Project settings](#) 中设置 [define such an interpreter](#) :



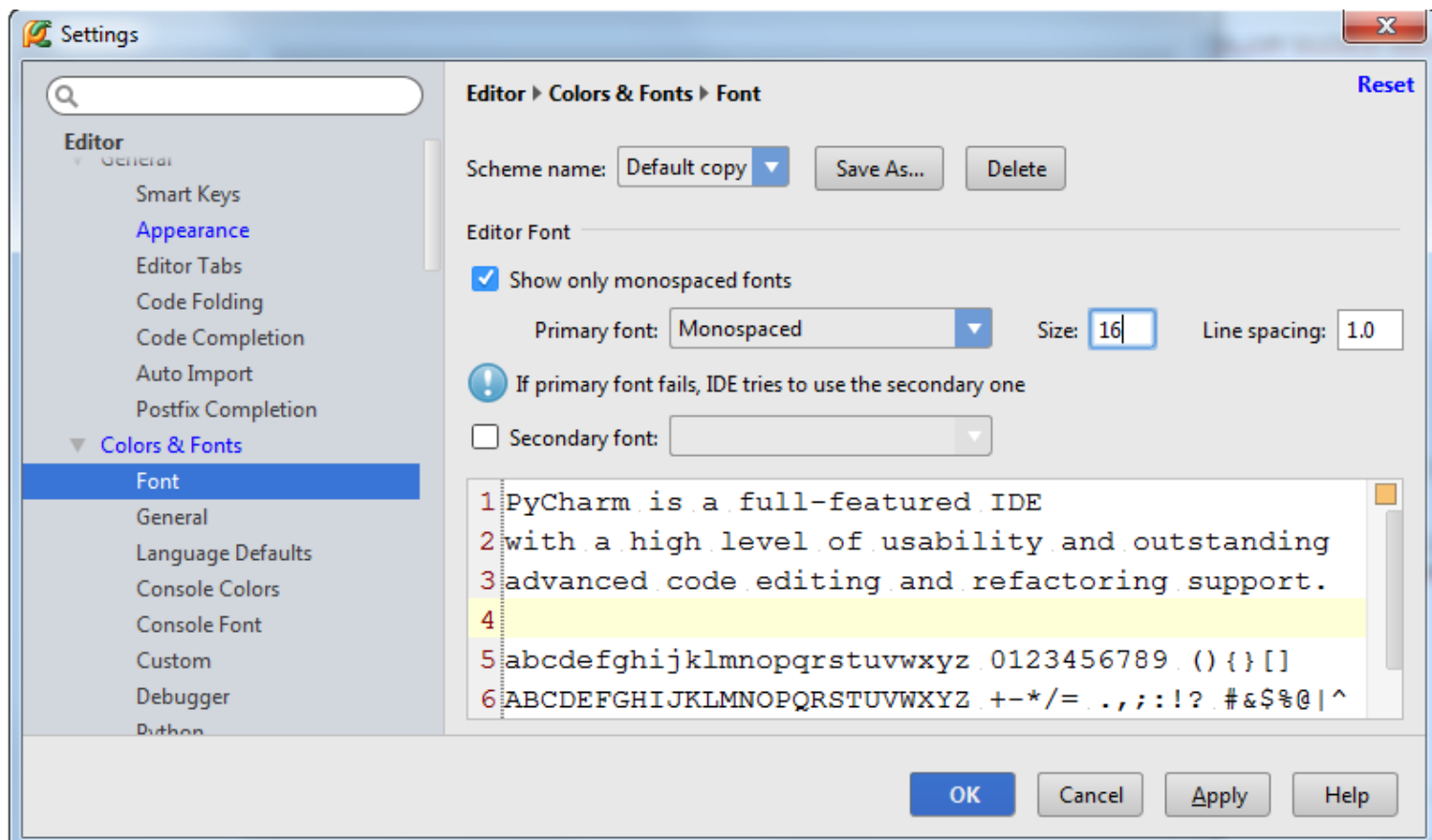
这里同样可以设置默认的编辑环境。例如你希望一直显示代码的行号，则需要在设置对话框中，展开 Editor 节点，在 [Appearance page](#) 页面将“Show line numbers”所对应的复选框勾选：



接下来假设你希望使用特定的颜色主题，OK，选择基本主题，拷贝，然后改变配色方案即可（系统预设的颜色主题是不可更改的）。

当然字体大小也是可以改变的。这些都需要在 [Colors and Fonts settings](#) 页面进行操作。同样的你需要先创建一个主题备份，定义编辑器的字体大小，这些设置会作用于编辑器字体，但不会对其他控件区域的字体造成改变。

我们可以在预览窗口预览更改后的效果：



更多详细的外观主题设置参见教程 [What my PyCharm looks like](#)。

当然在一个项目创建完成后，我们仍可以随时对其进行更改，这将在下面的章节 [What my PyCharm looks like](#) 进行讨论。

最后，你可以选择隐藏/显示用户界面的一些控件：工具栏按钮、菜单栏按钮、主工具栏按钮等。Pycharm 还允许你选择视图模式，详见：

- [PyCharm tool windows](#)
- [Presentation and Full Screen viewing modes](#)

4、工程

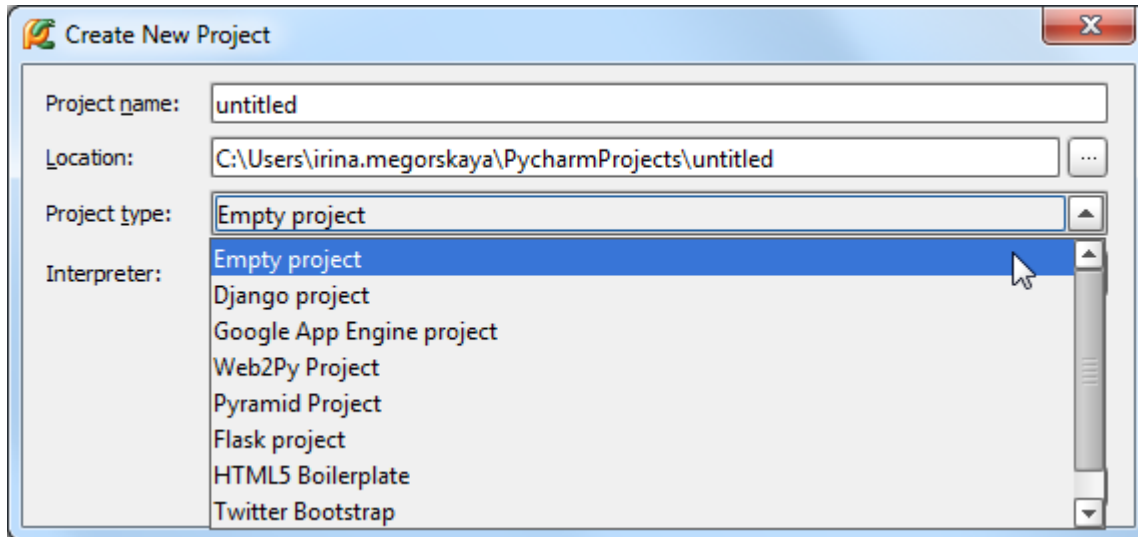
你在 Pycharm 中所做的任何操作都应该属于某个工程。最有意思的是 Pycharm 的工程管理器，它允许我们在一个框架下打开多个工程 [open multiple projects in one frame](#)。当你创建了一个新的工程（File → New Project）或者打开一个现有的工程（File → Open），Pycharm 会咨询你用哪种方式打开：单独在一个新窗口，还是添加到当前窗口。

你可以根据需要在一个窗口中打开多个工程。此时第一个工程被认为是主工程，其他工程的符号在主工程中均可用。

尝试创建一个新的工程作为练习，并输入一些代码。详见 [Getting started with PyCharm](#)，重点参见 [Creating a simple project](#) 部分。

5、工程类型

Pycharm 提供了各种各样的工程类型：Django、Flask、Pyramid、web2py 等等。创建工程时根据需要在列表中选择对应的工程类型。

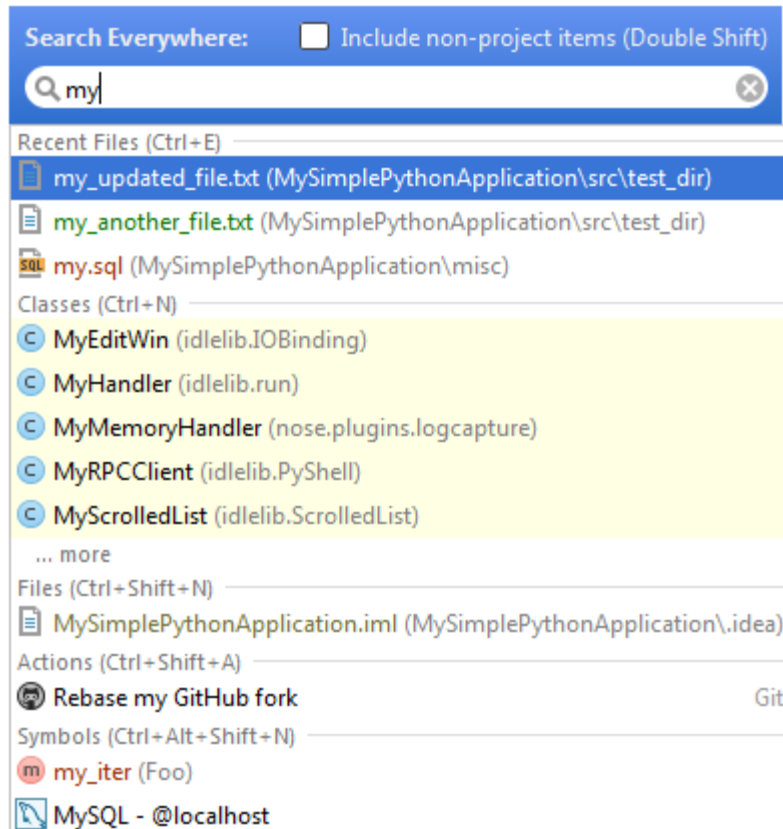


可见 Pycahrm 支持所有的 Python 主流框架，并会帮助生成对应的文件结构以及必要组件，详见：

- [Django](#)
- [Flask](#)
- [Pyramid](#)
- [Google App Engine](#)
- [Web2Py](#)

6、开始

此时工程已经创建完毕，在开始工作之前，按两下 Shift 键，会弹出一个窗口，供我们查找和跳转：



如你所见，在这里可以搜索 Pycharm 的任何信息，命令、设置信息、文件信息、控件等等。这只是 Pycharm 搜索导航功能的一部分。

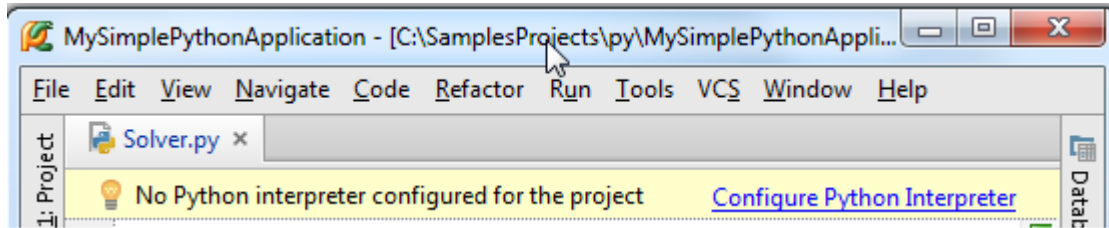
7、工程的私人订制

单击主工具栏的设置按钮进入设置对话框，[Settings/Preferences dialog box](#)，在这里可以改变项目结构、控件安装、调节开发环境。

一些设置是主要针对具体工程的，如工程解释器类型、配置属性、文件颜色等。其他例如编辑框设置、快捷键、生成模板等则是针对整体的开发环境，无需依赖某个具体工程。

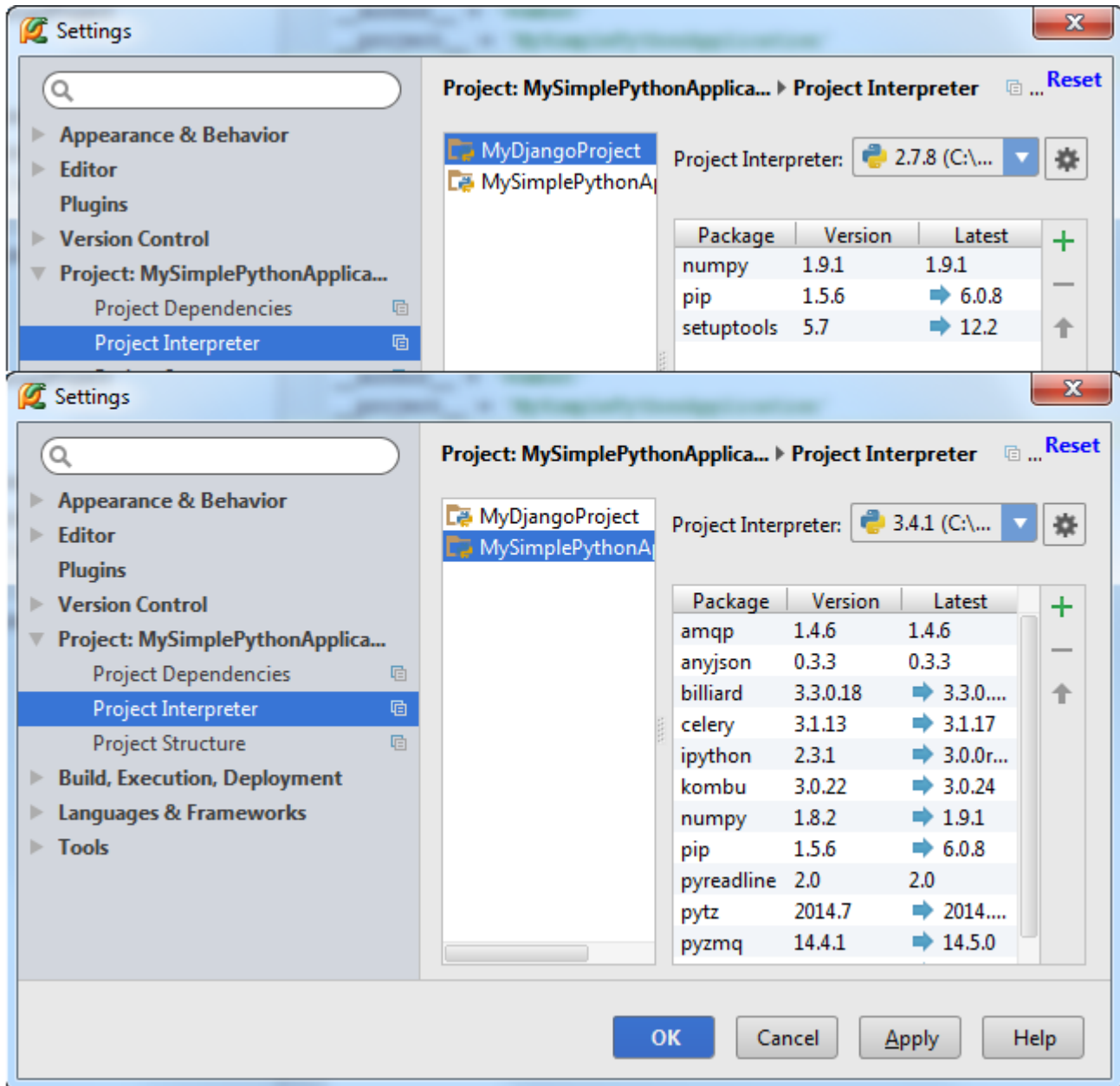
8、工程解释器

Python 解释器时必需的，若解释器配置失效，Pycharm 会给出如下错误提示：



在 Pycharm 你可以同时定义若干解释器，然后选择一个应用于工程中。

总之，必须明确告诉 Pycharm 使用哪个解释器以方便其 [use a different interpreter for each project](#)：



Pycharm 解释器类型包括以下几种：

- Local
- Remote
- Virtual environments

9、本地解释器

最直接的使用解释器的方式，下载 Python 解释器，安装到本地，执行.....，详见 [tutorial](#) 或者 [product documentation](#)。

10、远程解释器

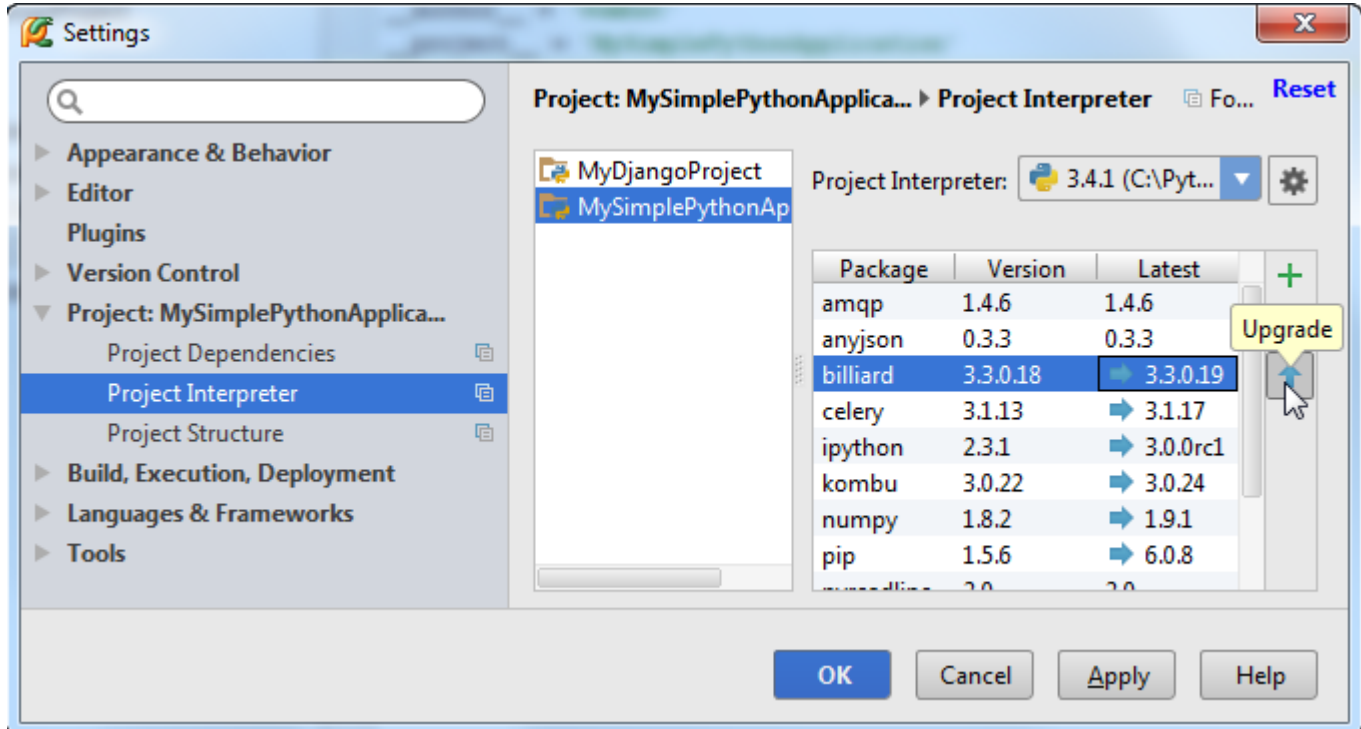
Pycharm 允许使用远程解释器，例如通过可靠的服务器来完成。此前 Pycharm 需要先通过 [SSH connection](#) 或者 [Vagrant box](#) 来进行远程解释器配置，详见 [Configuring interpreters with PyCharm](#)。

11、虚拟环境

重要性？假设你正在使用 Django 1.6 编写一个工程，同时你需要支持另外一个要求使用 Django 1.2 的工程，此时你需要通过某种手段来保证你操作环境的安全性和一致性，也就是通过一个工具创建一个 Python 编译器的拷贝。具体如何创建虚拟环境参见 [documentation](#) 以及 [tutorial](#)。

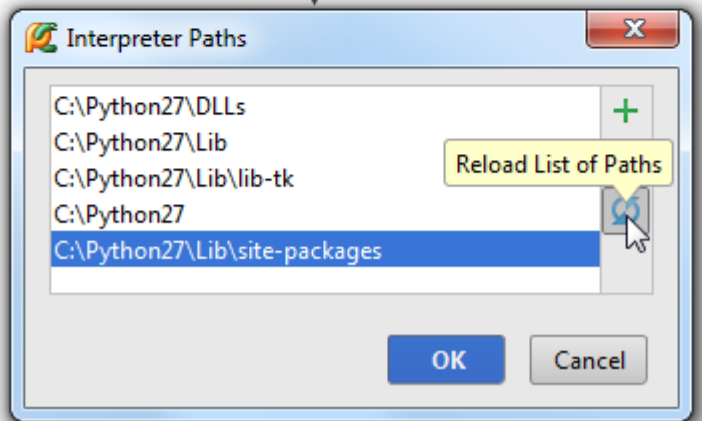
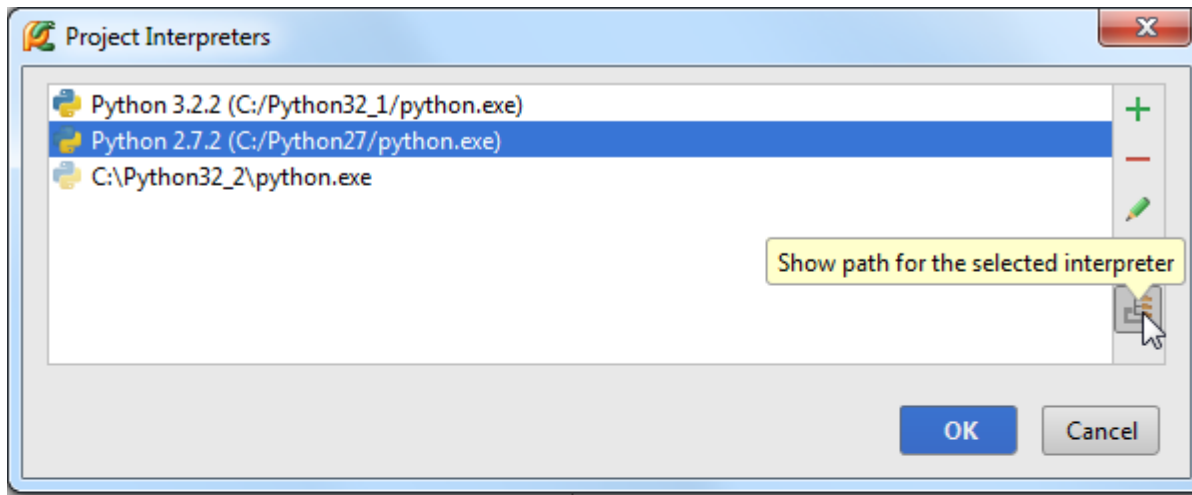
12、第三方库及其路径

如论哪个版本的解释器，Pycharm 能够帮助我们安装和更新一些必需的第三方库。例如当 Pycharm 检测到一些缺失的或者待更新的第三方库之后，会提示我们进行安装和更新：



对于路径的更新也同样适用。如果安装或更新了库，建议重新扫描一下 Python 的安装目录。单击 按钮，选择 more。

然后在工程解释器设置对话框中，选择要浏览的解释器，单击 按钮，在解释器路径对话框中，单击 按钮进行库更新。



其他详细信息参见 [product documentation](#)

13、VCS（版本控制系统）

Pycharm 帮助我们对已有工程进行版本控制。

当然我们可以对其进行微调。在 [Settings/Preferences dialog](#) 对话框中单击 Version Control 节点（Project Settings→Version Control），默认情况下只能看到工程的根目录，但是你可以将你的应用分割成更小的块进行管理。

在 [Settings/Preferences dialog](#) 对话框中，更改的命令行为将会应用于所有的版本控制系统：创建/删除文件的确认信息、后台控制方式等等。

更多内容参见：

- [Version control basics](#)
- [Using PyCharm's Git integration locally](#)
- [Sharing via a remote repository](#)

也可以参考 Pycharm 文档：

- [Version control with PyCharm](#)
- [Version control procedures](#)

14、文件颜色

你的工程中可能包含若干工程，每个工程目录下的文件名都可能相同（例如 `init.py`, `models.py`, `tests.py`, `views.py` 等），当它们同时在编辑器中打开时，为了方便区分其各自归属于哪个工程，PyCharm 通过其标签颜色来进行标记（Settings/Preferences→Project Settings→File Colors）。尝试将你的工程分片，然后分别制定其文件配色方案。

更多信息参见 [Configuring scopes and file colors](#)

15、IDE 和编辑器

IDE 的职责在于改善编程环境，优化视觉体验，提高操作效率。因此 PyCharm 允许我们对 IDE 以及编辑器进行各种各样的私人订制，如快捷键、滚动条、高亮显示等等。详情参见

• [Configuring project and IDE settings](#)

以上这些设置都是在设置对话框中完成的 [Settings/Preferences dialog](#)。单击主工具栏的设置按钮，打开对应页面进行设置即可，推荐在优先在这些页面中作调整：[Appearance](#)、[Keymap](#)。

16、外观

在 Settings/Preferences→Appearance and Behavior→Appearance 页面中定制外观。单击 Look and feel 下拉列表，选择喜欢的主题。这里单击 Apply 按钮预览选中主题效果，无需关闭对话框，直到满意为止。

更多信息参见 [How do I choose look and feel for my PyCharm?](#) 以及 [product documentation](#)。

17、编辑器

所有编辑器相关设置都在 [Editor](#) 节点下（Settings/Preferences→Editor），包括配色方案、字体、高亮显示机制等等。并且可以在预览窗口快速预览所作的更改。

更多信息参见 [How do I change color scheme of the editor](#) 以及 [product documentation](#)。

18、快捷键

快捷键的设置取决于你的操作习惯。

通过 Settings/Preferences→Appearance and Behavior→Keymap 打开快捷键设置窗口，选择对应的快捷键方案，设置特定的快捷键组合。

这里有一个强力的快捷键 `Ctrl + Back Quote`，能够在不同主题间切换而无需打开设置对话框。

更多信息参见 [Configuring keyboard schemes](#) 以及 [product documentation](#)。

19、外部编辑器

PyCharm 可以借助外部插件来转换为其他编辑器形式，例如 Emacs 等，详见 [Using Emacs as an external editor](#)。

20、后台任务

PyCharm 对一些耗时较长的任务会给出进度条，我们可以将其隐藏在后台（仍保持可见），详见 [Working with Background Tasks](#)。

21、代码智能

PyCharm 在编写代码过程中会给出一些方便智能提示，主要体现在：

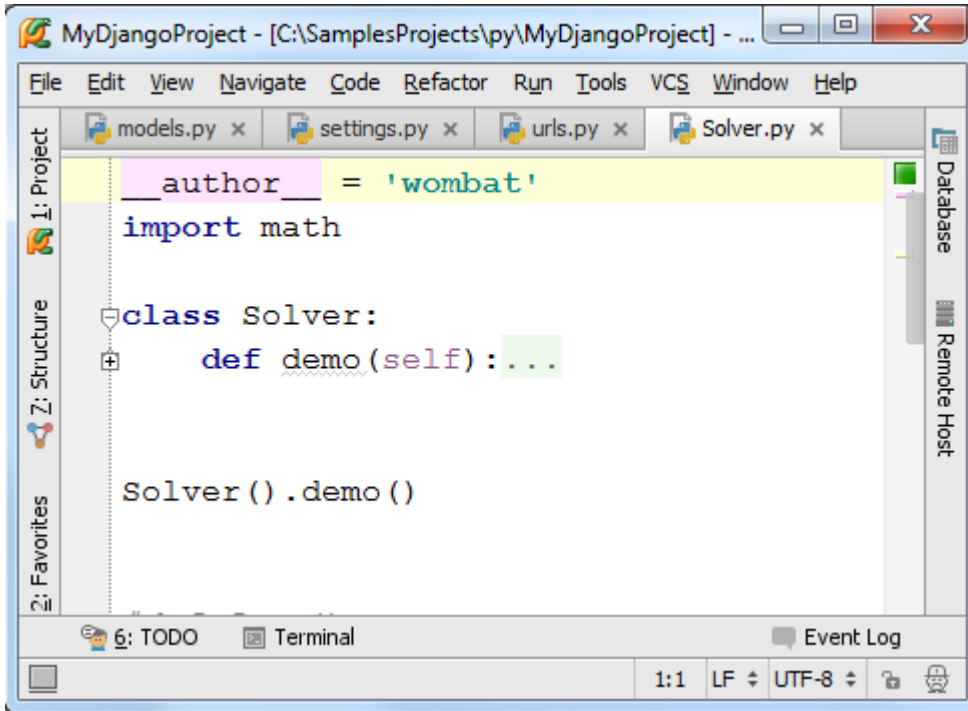
- (1) 拼写提示（`Ctrl+Space`）[code completion](#)
- (2) 使用生成模板 [Creating and applying live templates \(code snippets\)](#)，[product documentation](#)
- (3) 代码快速定型 [Quick fixes and intention actions once more](#) 和 [product documentation](#)

22、宏的使用

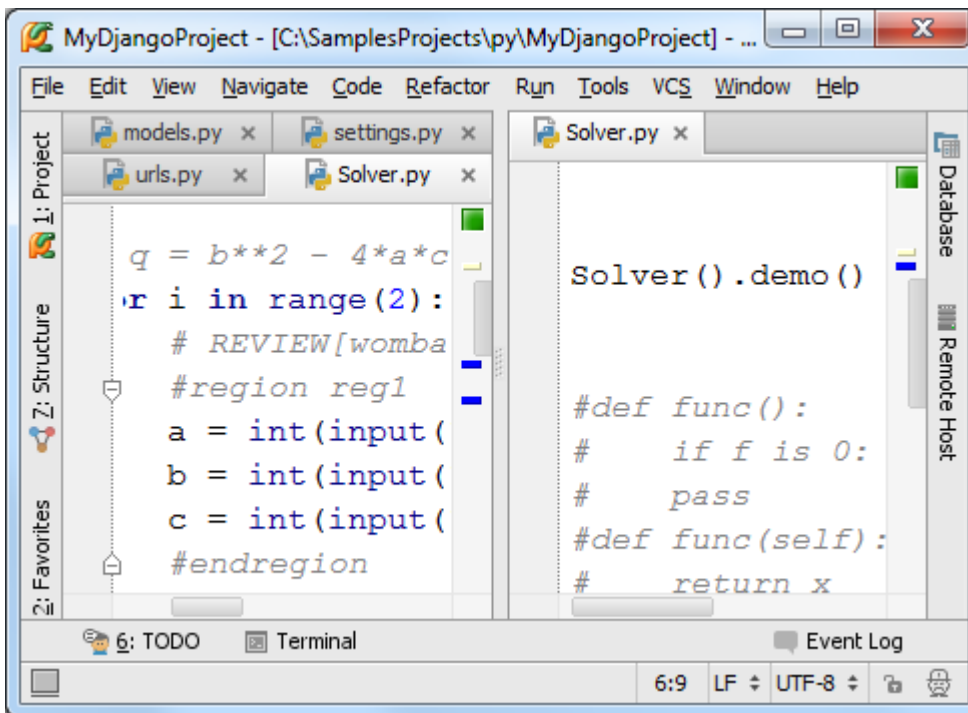
使用宏能够简化编辑过程，详见 [Using macros](#) 和 [documentation](#)。

23、多文件工作环境

默认情况下，Pycharm 通过单独编辑选项卡来打开文件：



我们可以手动交换这些选项卡的顺序、固定或者非固定、靠边显示、拆分等等：

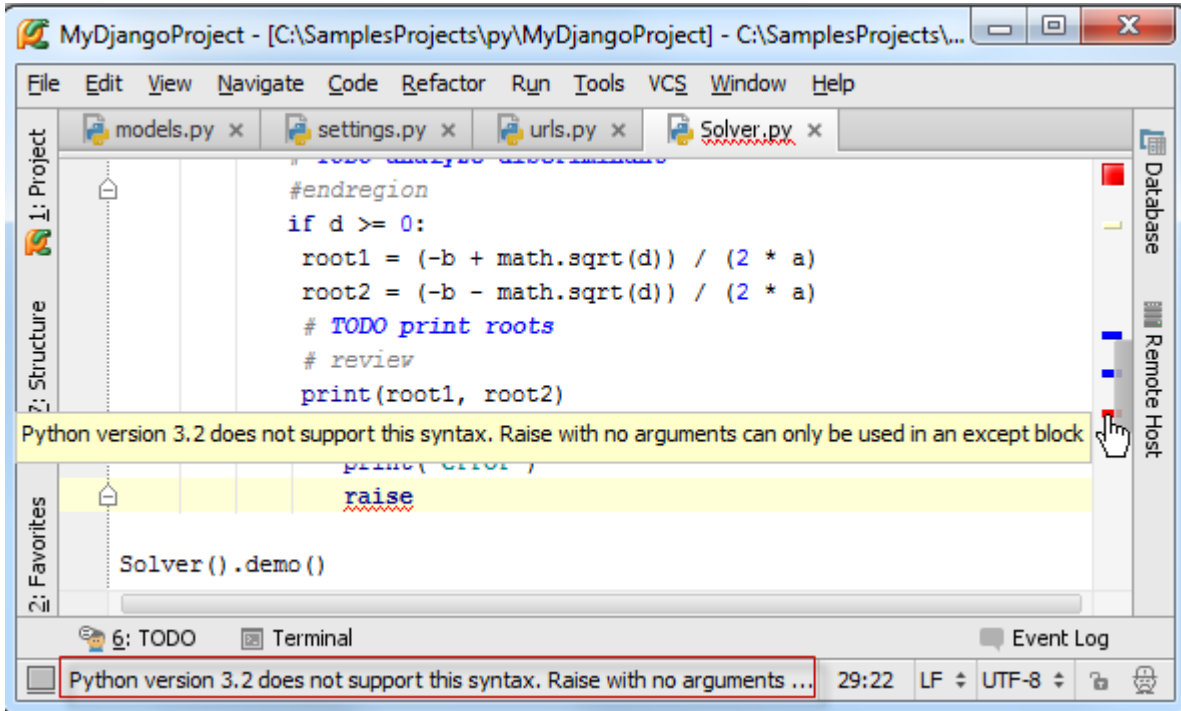


详见 [section Managing editor tabs](#) 以及 [tutorial](#)。

24、显示代码分析

Pycharm 会给出各种各样的提示来帮助你发现并改正代码中的错误。

首先，在输入代码时它会以红色波浪线标记所有的语法错误，同时在右槽对应行给出标记，在下方状态来显示错误信息，鼠标悬停在对应标记上时也会显示简要的错误提示信息：



同时 Pycharm 还会通过在左侧显示红色、黄色灯泡图标来指示错误和警告。

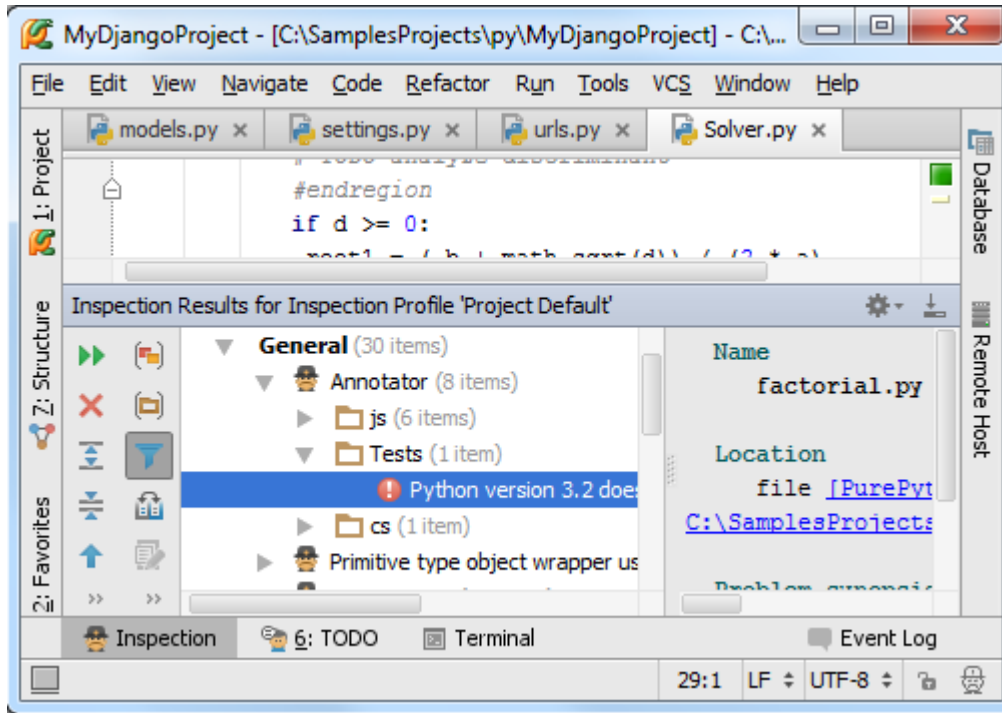
接下来是静态代码分析，也就是代码检查，并在右槽给出对应标记。红色代表错误，黄色代表警告，右槽顶部代表该文件的整体错误情况。绿色代表一切正常，红色和黄色代表有错误发生。

同时右下方状态栏的 Hector 图标用来指示当前的纠错等级。



如果你对自己的编程能力很有自信，可以单击它以关闭代码检查。

然而如果你希望对整个工程进行代码检查以提高代码质量，选择 Code→Inspect Code，在 [Inspection tool window](#) 显示代码检查结果：



更多信息参见 [Syntax highlighting and error indication](#)。

25、创建高质量代码

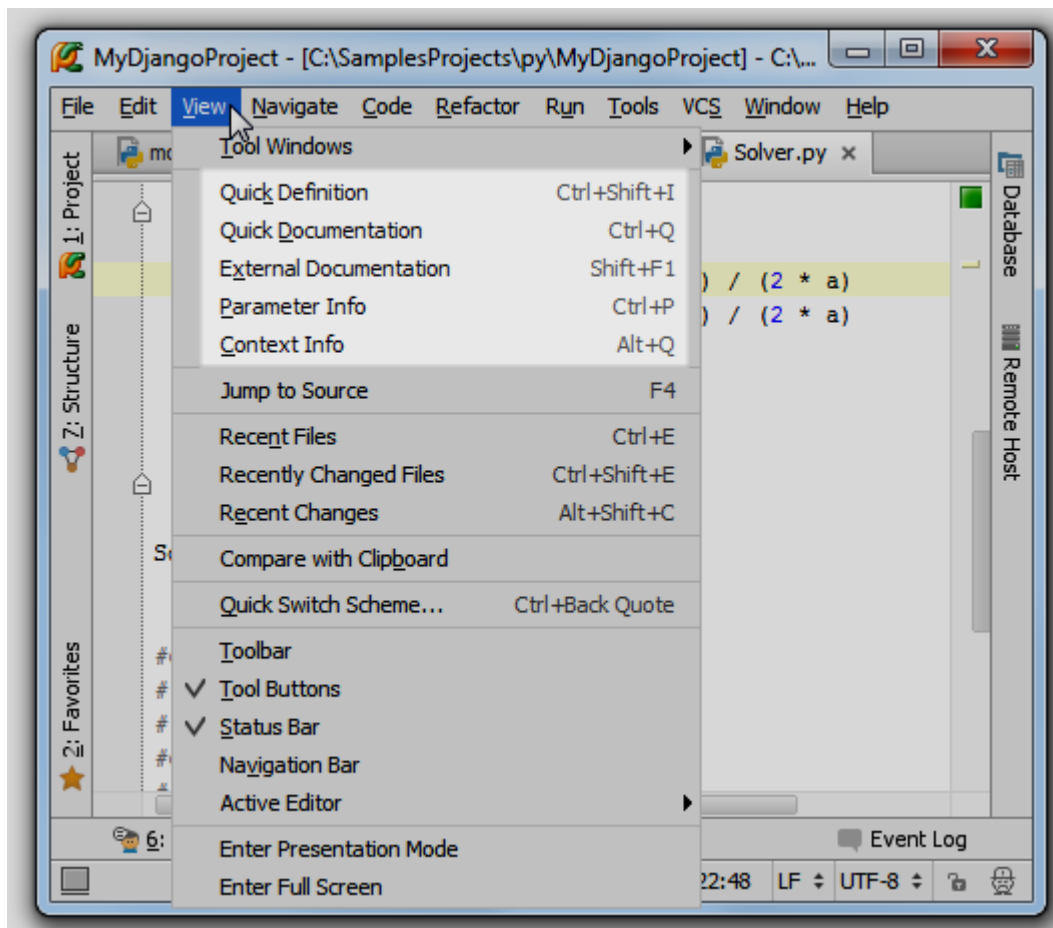
详见 [Code Quality Assistance Tips and Tricks, or How to make your code look pretty?](#)

26、浏览文档信息

Pycharm 有以下几种浏览文档的方式：

- Quick definition
- Quick documentation
- External documentation
- Parameter info
- Error description

这些方式都有直接的菜单命令相对应，以及快捷键设置：

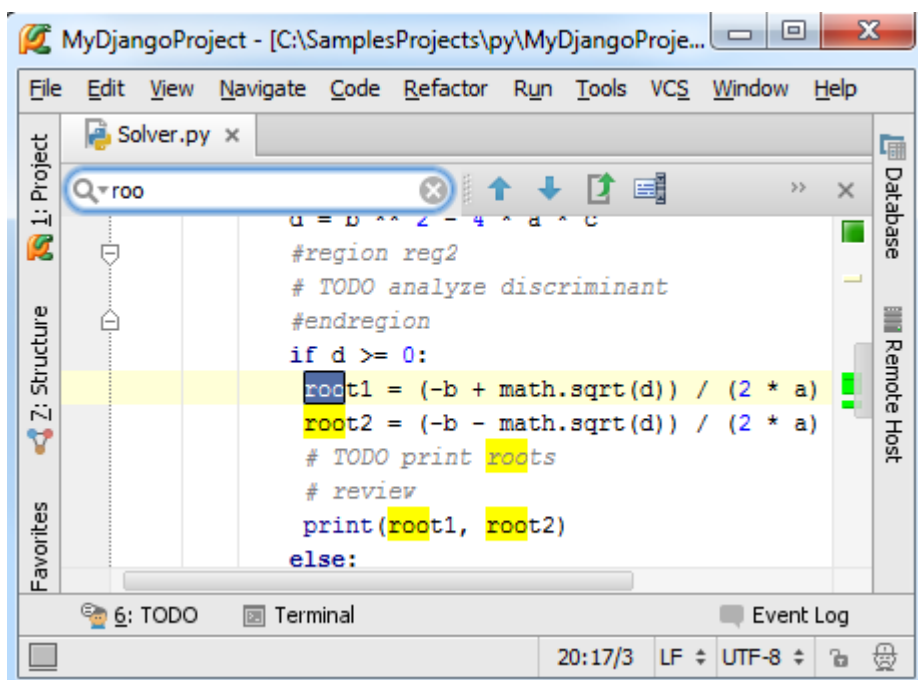


详见 [Viewing documentation](#) 以及 [Viewing reference information](#)。

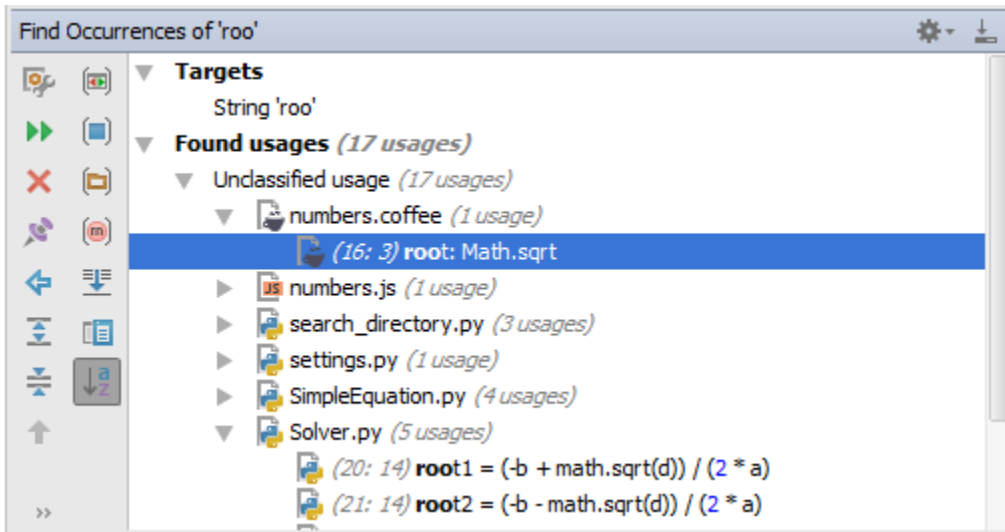
27、搜索与查找

28、源码搜索

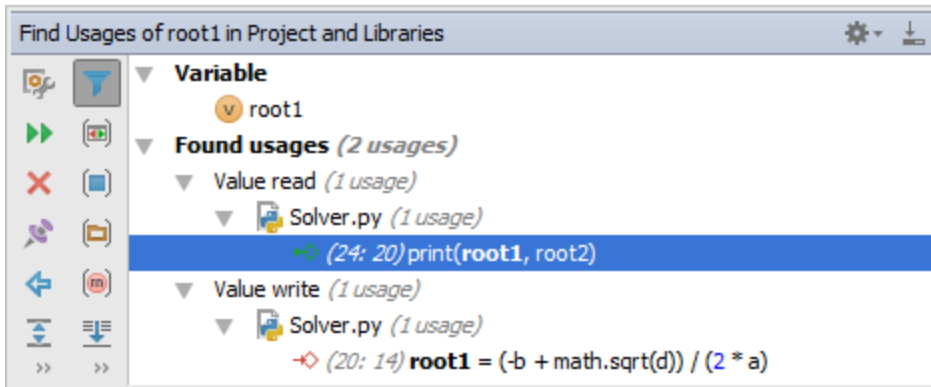
最基本的查找方式就是 Ctrl+F 命令：



Pycharm 运行进行更深层次的查找：特定目录、任何范围、整个工程（Ctrl+Shift+F）：



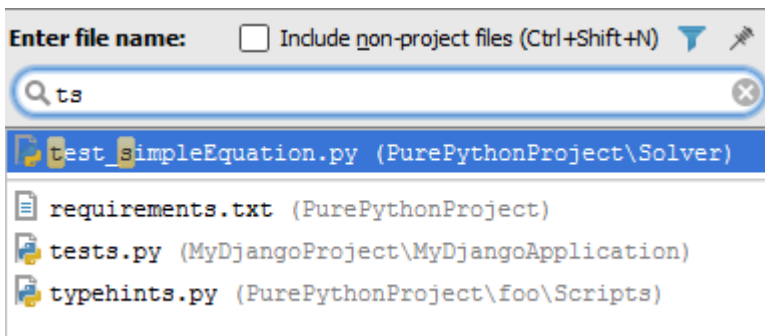
Pycharm 还可以查找某个符号的所有应用环境 [search for usages](#)，按下 Alt+F7，或者使用快捷菜单的 Find Usages 命令：



还有其他类似的命令：Ctrl+F7 实现跳转；Ctrl+Shift+F7 以代码着色的方式浏览 Usages；Ctrl+Alt+F7 以弹出窗口的形式搜索整个工程。

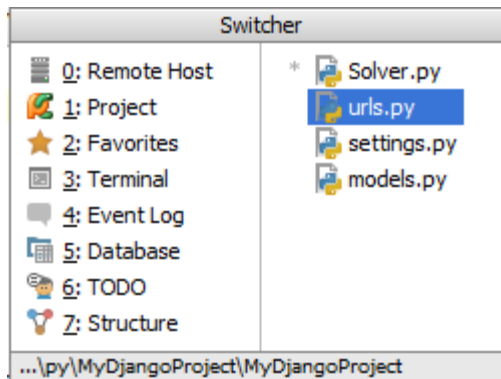
可以通过中间键单击或者 Ctrl+B 跳转到对应声明。

Ctrl+N: 按名称快速查找一个类；Ctrl+Shift+N: 按名称快速查找一个文件；Ctrl+Shift+Alt+N;按名称快速查找一个符号，这些查找均可以使用名称通配符。

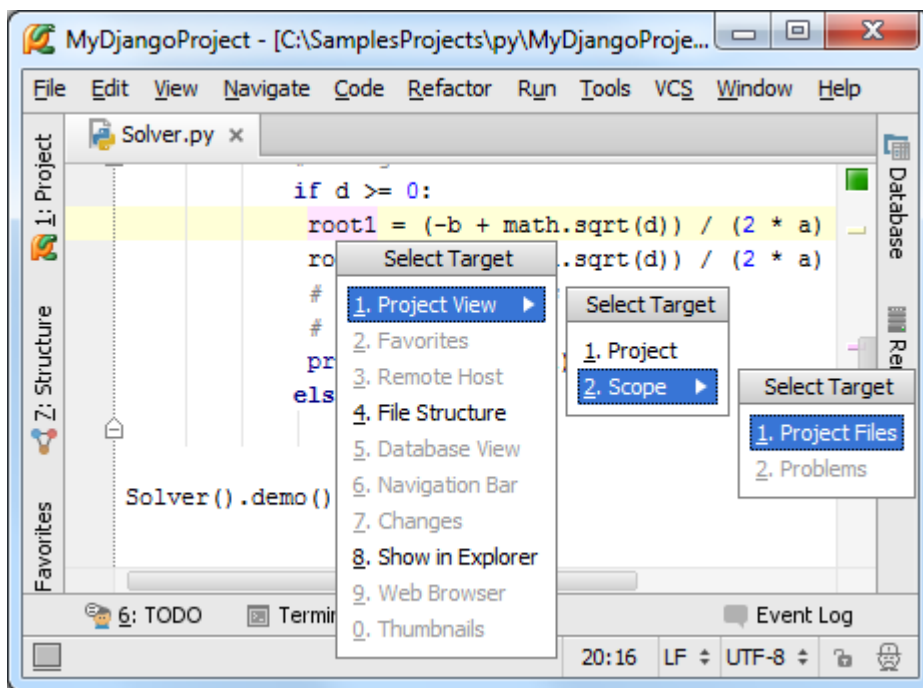


29、IDE 控件搜索

我们这里只是简要介绍。以 [switcher](#) 为例，按下 Ctrl+Tab 显示 switcher，按住 Ctrl 不放，使用 Tab 键或者方向键来滚动到目标位置：



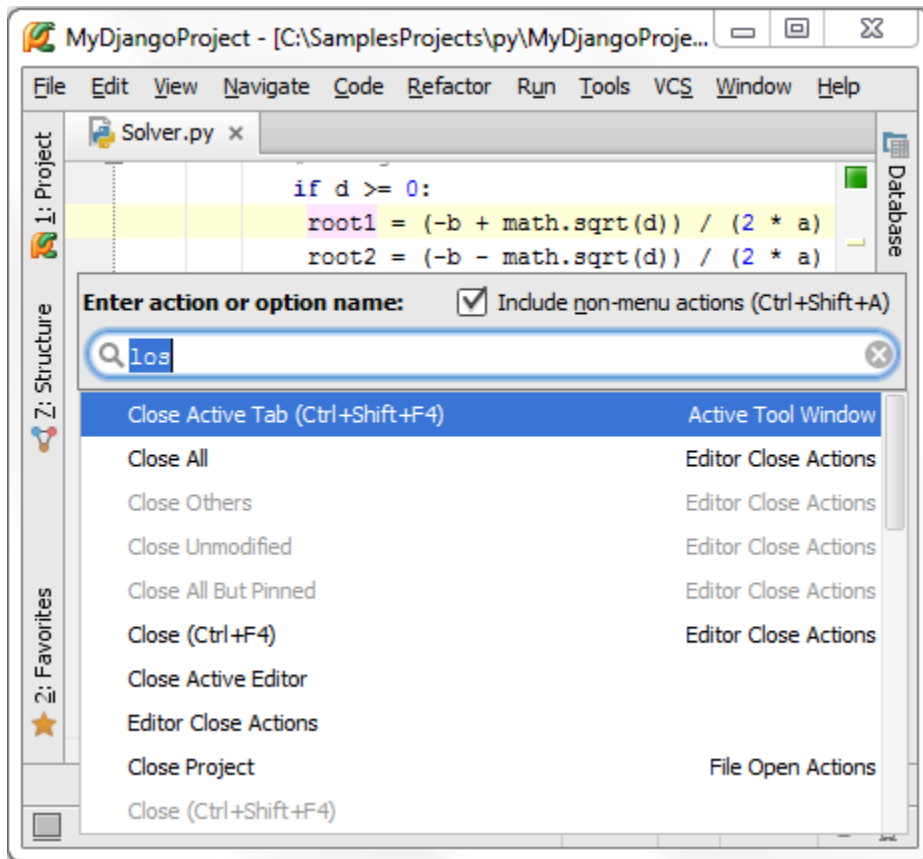
如果你选择一个 IDE 组件，然后想在其他组件中浏览它，使用 [Select Target](#) (Alt+F1)：



最后通过 Esc 键返回编辑框。

30、命令搜索

可以通过简单的命令搜索 [invoke it by name](#) 来找到并使用相应的命令（不使用主菜单）。按下 Ctrl+Shift+A，输入对应字母：



注意你可以在这里搜索主菜单中并不包含的命令，只需再次按下 Ctrl+Shift+A 即可。

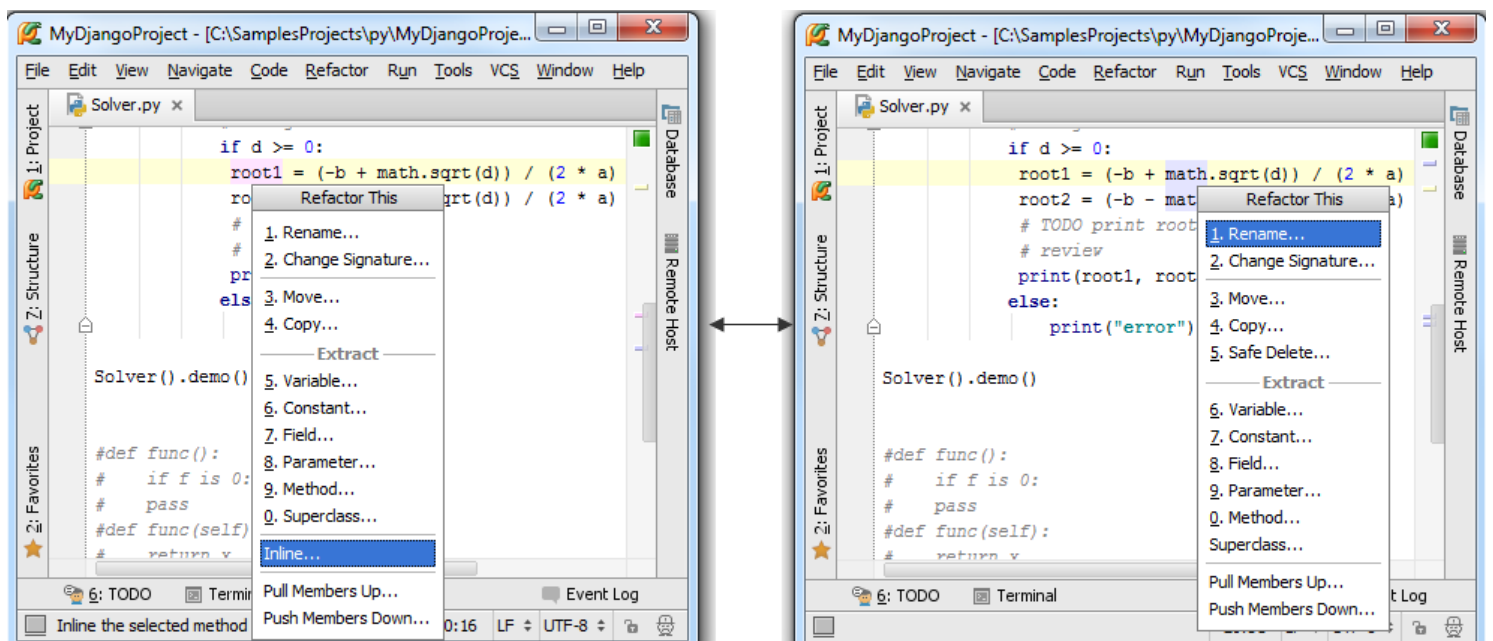
31、搜索导航任意位置

按两下 Shift，或者单击主工具栏右端的  按钮，打开 [Search everywhere](#) 对话框，在这里面可以查找任何内容。

更多信息参见 [Exploring navigation and search](#), [Navigating through the source](#), [Searching through the source code](#)。

32、重构代码

Pycharm 建议根据当前代码环境进行重构。



33、运行、调试、测试程序

34、运行代码

通过右键快捷菜单或者 Ctrl+Shift+F10 快捷键运行代码,如果想使用其他配置信息来运行,需要在主菜单栏中进行选择,然后按下 Shift+F10。

详情参考 [Code running assistance](#) 以及 [Running](#)。

Pycharm 允许远程调试运行 [virtual machine and Vagrant boxes](#)。

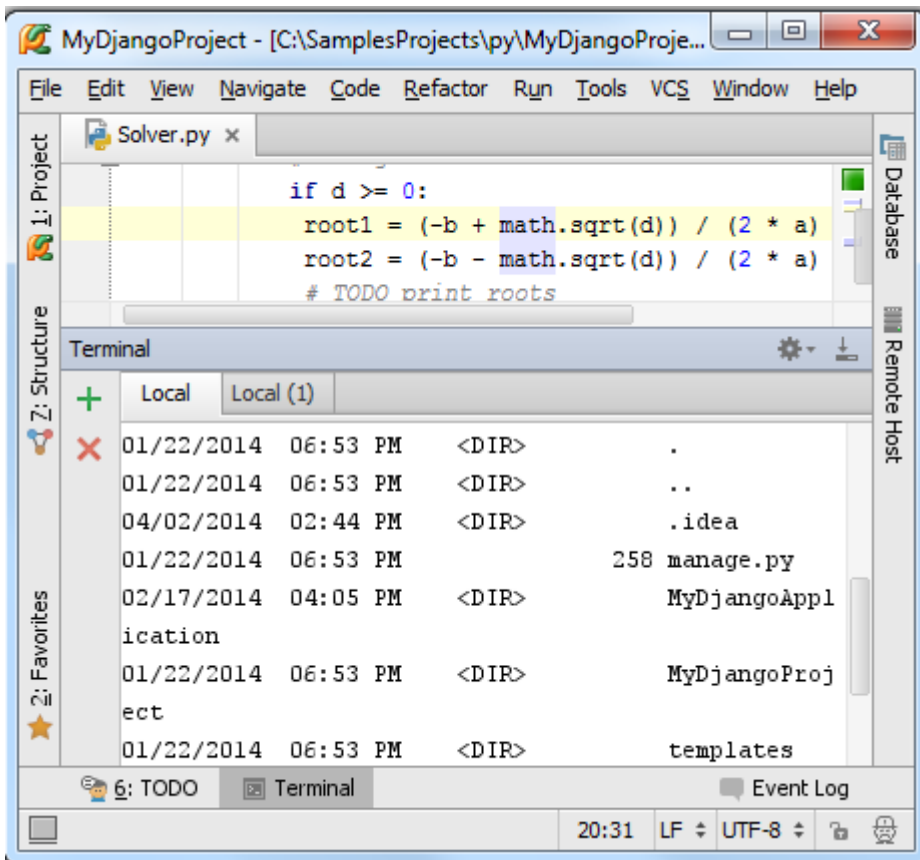
35、REPL 控制台窗口

在 Tools 菜单下,选择加载 Python 或者 Django 控制台的命令。Pycharm 控制台能够高亮显示代码、拼写提示、显示历史命令 (Ctrl+Up/Down)

Pycharm 也可以将编辑器中的代码放到控制台中运行。选中代码,然后按下 Alt+Shift+I 快捷键 (也可以使用快捷菜单中的命令),更多信息参见 [REPL - running an interactive console](#) 以及 [Working with consoles](#)。

36、本地终端

选择 Tools→Open Terminal 的主菜单命令,进入命令行操作模式:



更多信息参见 [product documentation](#)。

37、调试

调试方法很简单,选择待调试脚本,指定配置文件,按下 Shift+F9。

选择运行/调试配置文件的快捷键: Alt+Shift+F10/ Alt+Shift+F9

更多信息参见 [Debugger](#)、[Breakpoints](#)、[Debug run](#) 以及 [Debugging](#)。

38、测试

Pycharm 提供如下测试途径：

- (1) 创建测试类
- (2) 指定测试用的配置文件
- (3) 运行/调试测试类
- (4) 查看结果

Pycharm 支持以下主流的 Python 测试框架：[Unittest](#), [Doctest](#), [Nosetest](#), [py.test](#) 和 [Attest](#)，并预先设定好了相关配置文件。

更多信息参见 [Creating and running a Python unit test](#)、[Unit tests](#)、[Testing](#)。

39、远程操作

首先，将本地脚本文件上传至远端服务器，具体参见 [Configuring Synchronization with a Web Server](#)。

接下来就可以远程调试运行相关文件了。具体参见 [compare local and remote folders, and synchronize local copy with that deployed on the server](#)

40、数据库以及 SQL 支持

Pycharm 在数据库方面提供了相关支持。一旦你获得了某个数据库的权限，你就可以借助 Pycharm，利用已有许可证向其中存储更多相同格式的数据源。Pycharm 在数据连接过程中提供帮助。

更多信息参见 [product documentation](#)

41、多编程语言支持

Pycharm 支持 Python、[JavaScript](#)、[CoffeeScript](#)、[HTML](#)、[XML](#) 等编程语言。其强力的代码助手能够满足各种编程语言的需求。

同时 Pycharm 还支持 JavaScript 的代码调试，详见 [JavaScript-Specific Guidelines](#) 以及 [Debugging JavaScript with PyCharm](#)

42、大功告成

这是 Pycharm 的俗称手册，简要介绍了其各个特征，方便大家快速入门，请开始实践吧。

最全 Pycharm 教程（30）——Pycharm 中的 File Watchers

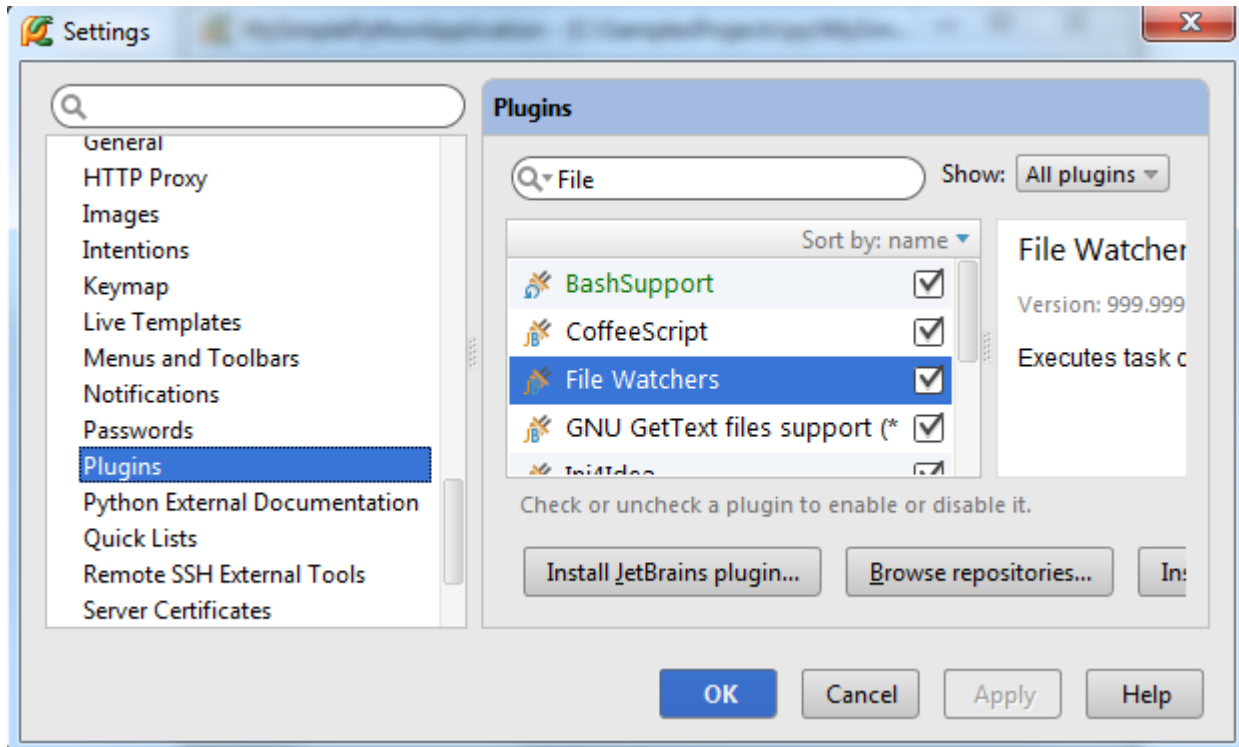
1、主题

详细介绍 PyCharm 中 file watcher 的使用。

2、准备工作

(1) Pycharm 版本为 3.1 或者更高

(2) File Watchers 插件可用，这个插件在安装 Pycharm 应该会默认安装（若没有则需手动安装，参见 [product documentation for details](#)）：

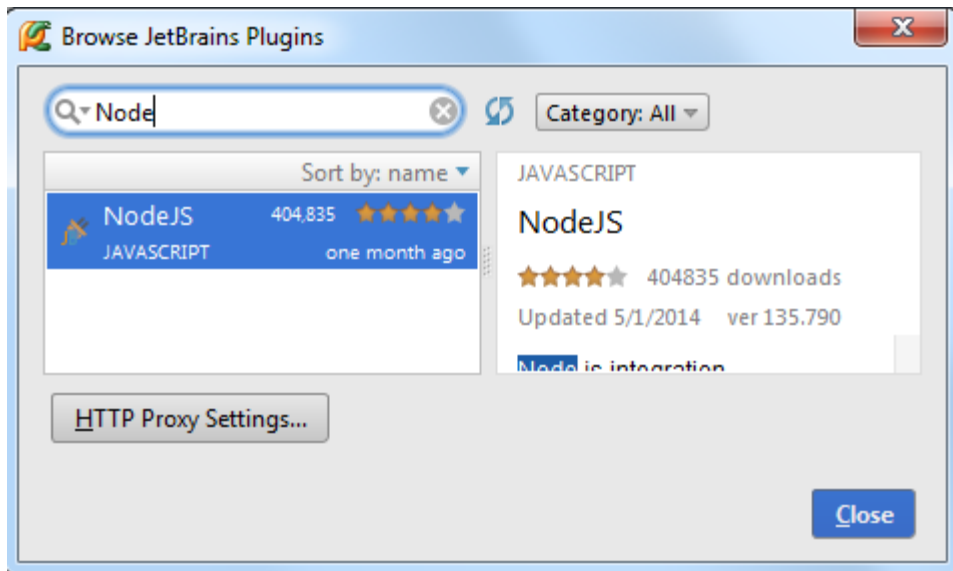


(3) 提前熟悉相关使用要领，参见 [Using File Watchers](#)

(4) 这篇教程中我们会处理 [LESS](#) 以及 [CoffeeScript](#) 文件。建议提前做一些预习工作。

3、安装 Node.js 插件

首先需要下载安装 Node.js 插件。打开 IDE 设置的 [Plugins](#) 页面，搜索对应插件：

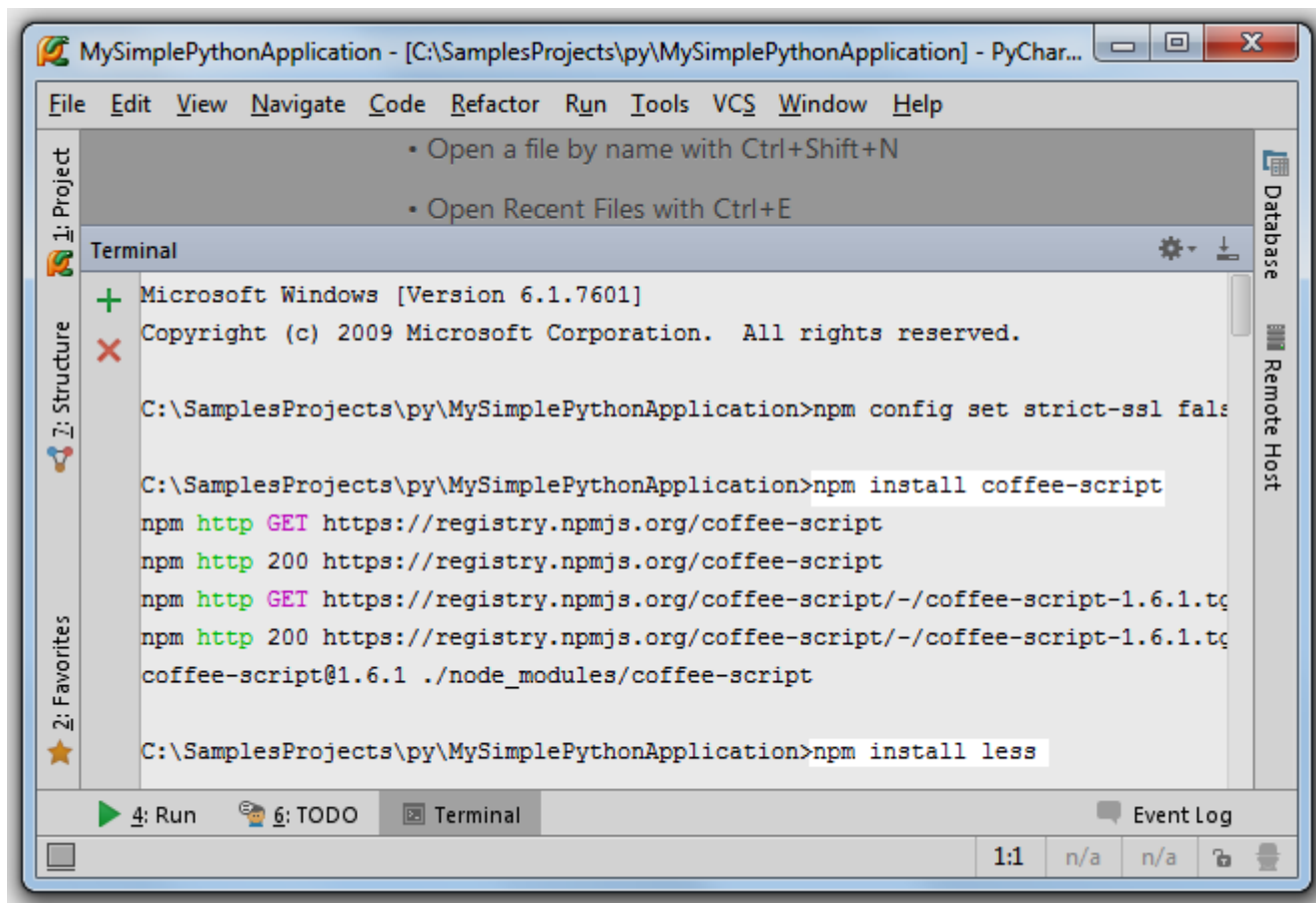


安装完成后，重启 Pycharm。

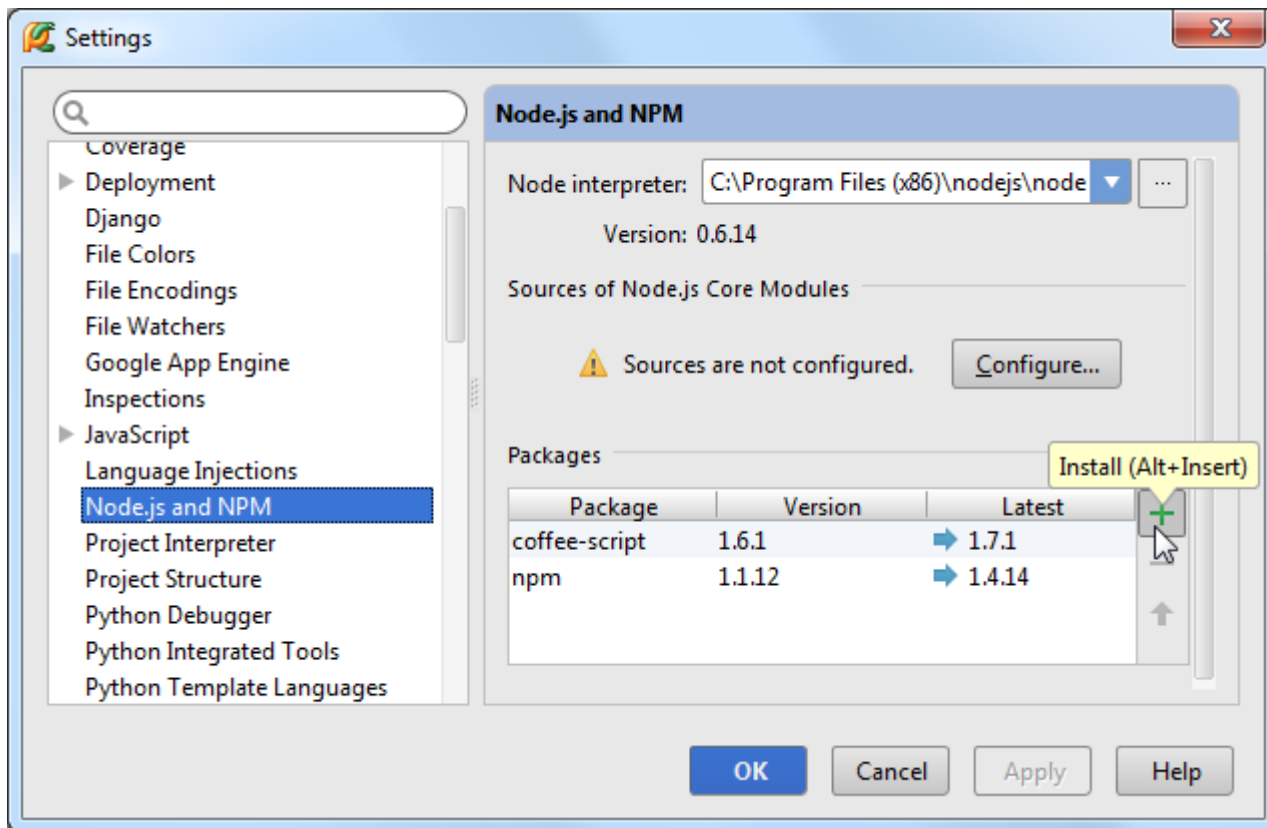
4、安装 LESS 和 CoffeeScript 编译器

这里有两种方式安装这两个编译器：

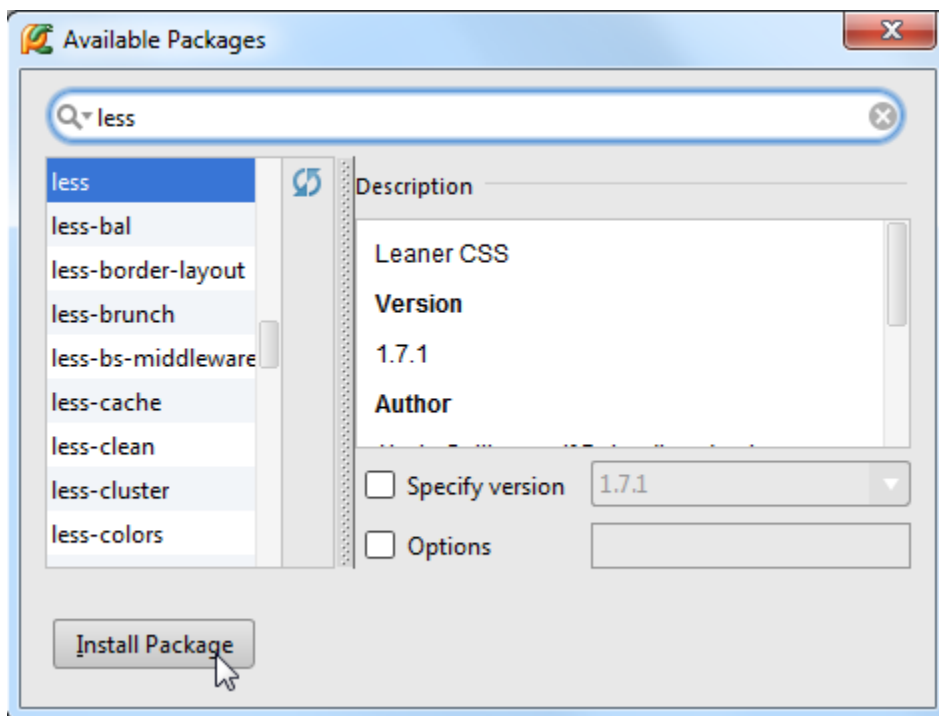
(1) 第一，手动安装。打开 [local terminal](#)（单击 PyCharm 窗口下边缘的 Terminal 窗口按钮），使用 npm 命令安装 LESS 和 CoffeeScript：



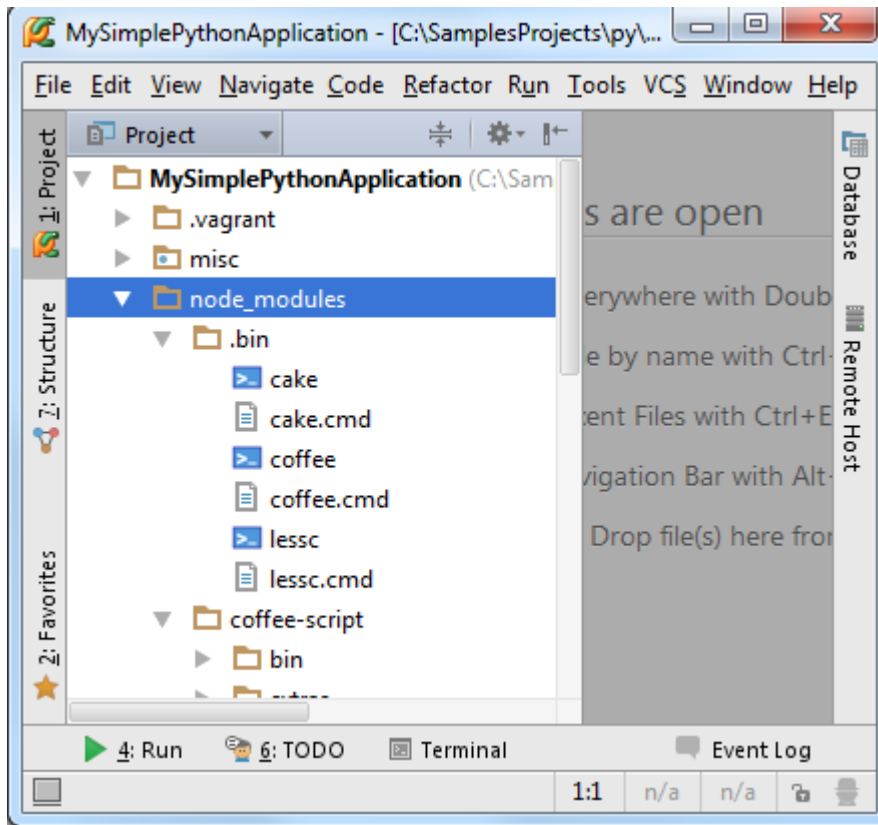
(2) 第二，使用 Pycharm 界面安装：单击主工具栏的设置按钮，在 [Node.js and NPM](#) 页面单击绿色加号：



在 Available Packages 对话框中，选择要安装的库（这里选择 less），单击 Install Package：



安装完成后，相关编译器文件会写入工程的根目录下：



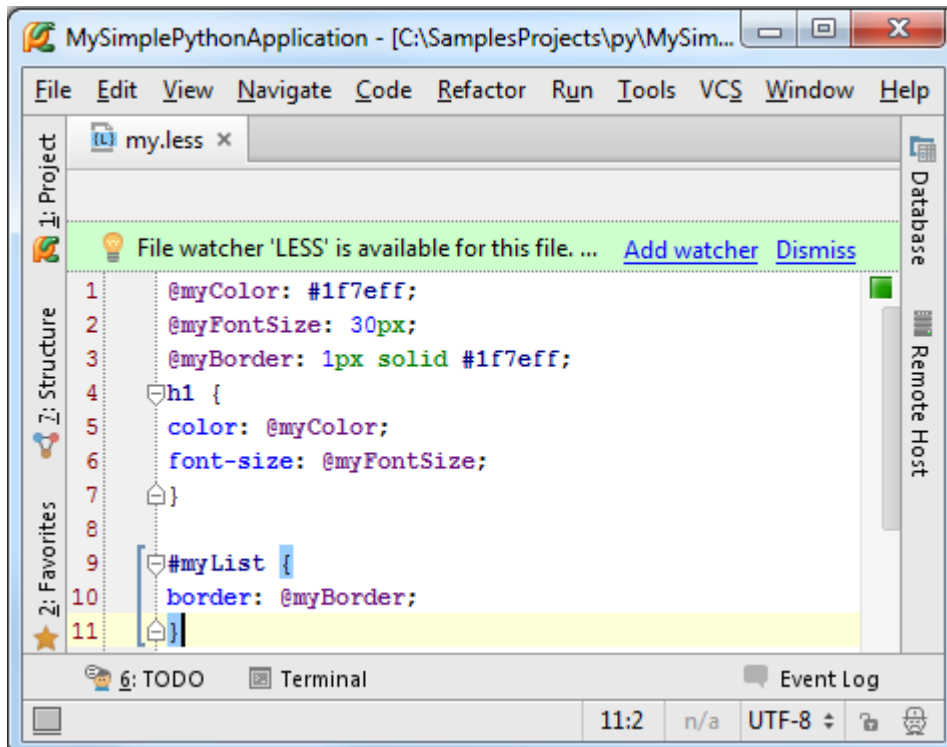
稍后用到这些文件。

5、配置 File Watchers

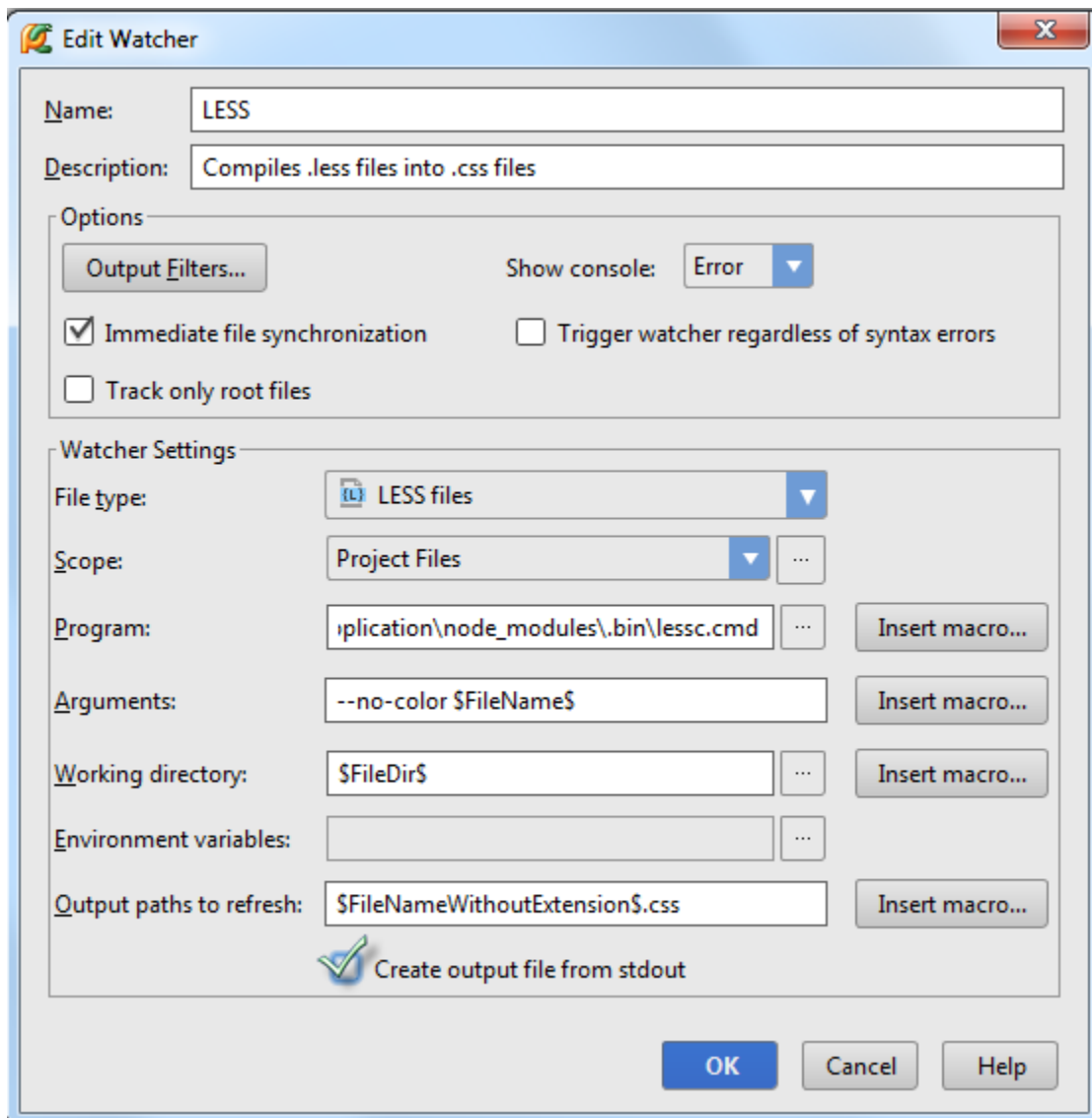
这一步由 Pycharm 自动完成。

6、为 LESS 文件配置相应的 file watcher

在没有进行相应配置是，打开 LESS 文件后 Pycharm 会给出错误提示：



单击 Add watcher 连接按钮，在弹出的对话框中选择 file watcher 的可执行类型（这里选择 LESS），并制定从标准输出中指定生成格式：

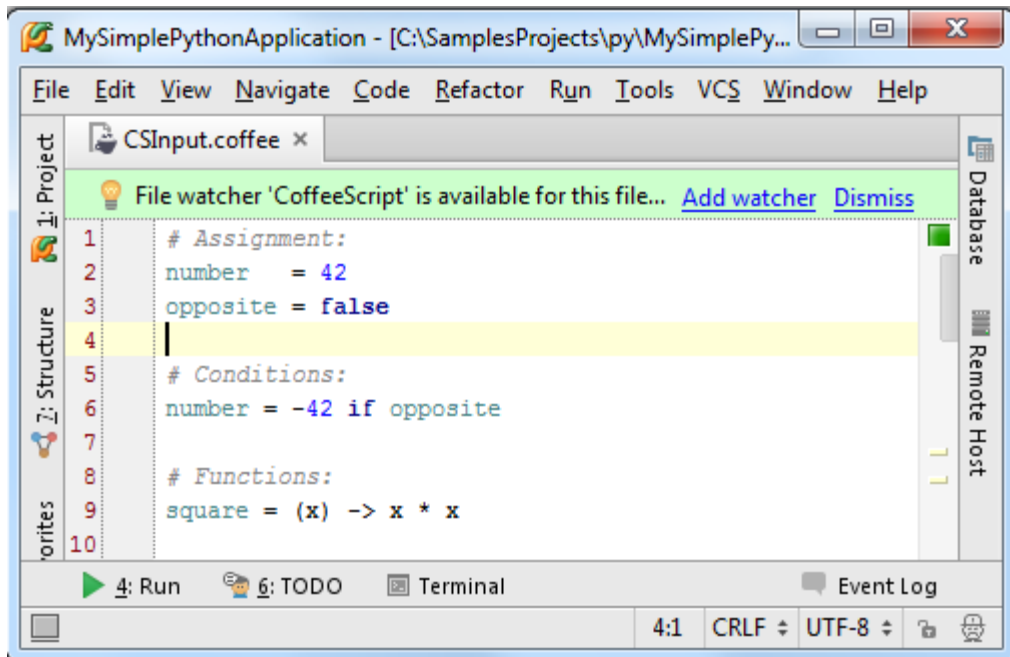


通过这个配置窗口你能够大致了解 file watcher 的实际功能；

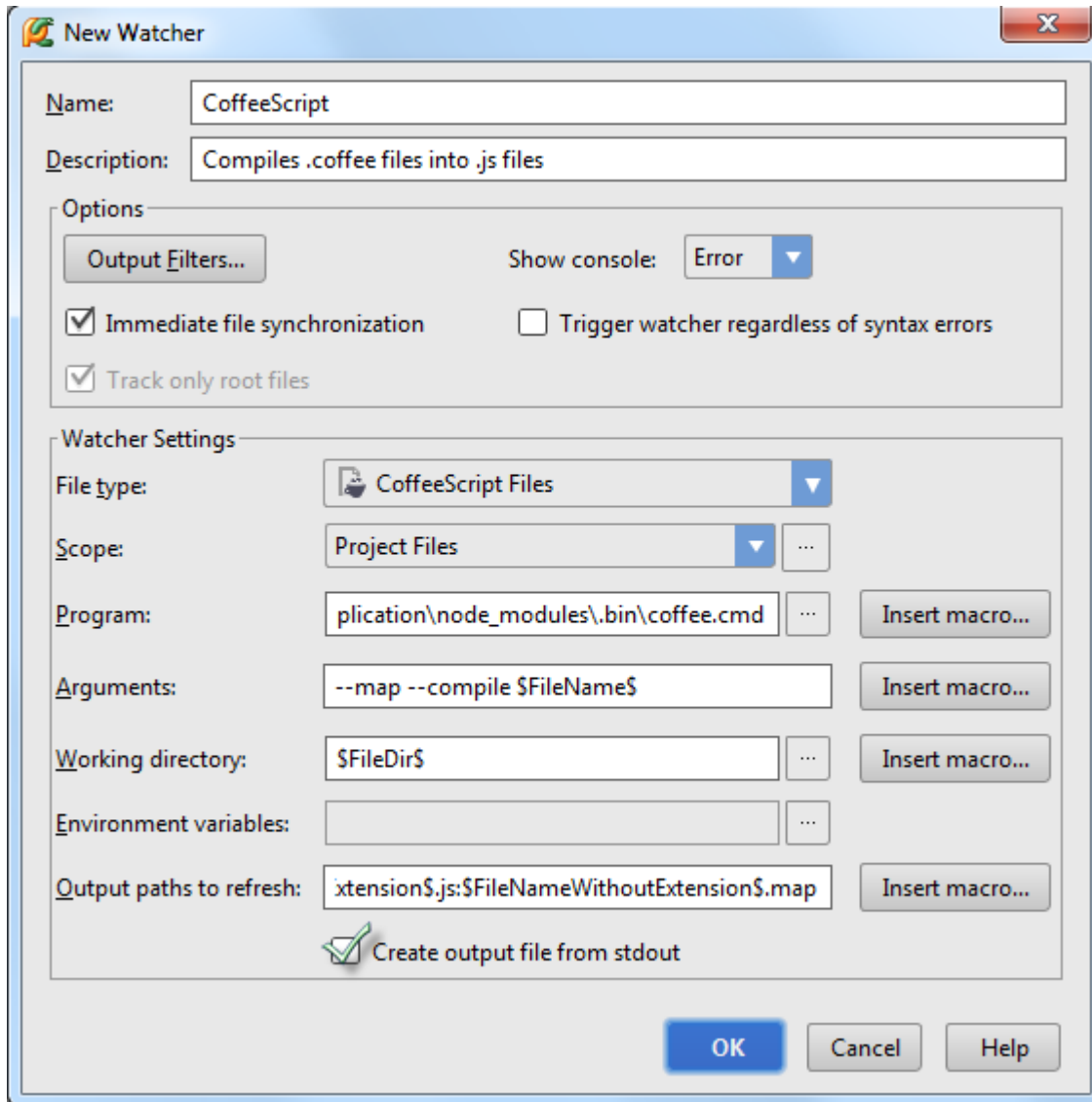
- (1) 监视项目中所有 LESS 文件的变化
- (2) 使用外部扩展来编译文件，一般使用定义在 Program 域中的 *lessc.cmd* 编译器。

7、为 CoffeeScript 文件配置相应的 file watcher

在没有进行相应配置是，打开 CoffeeScript 文件后 Pycharm 同样会给出错误提示：



单击 Add watcher 连接按钮，在弹出的对话框中选择 file watcher 的可执行类型（这里选择 CoffeeScript）：



通过这个配置窗口你能够大致了解 file watcher 的实际功能：

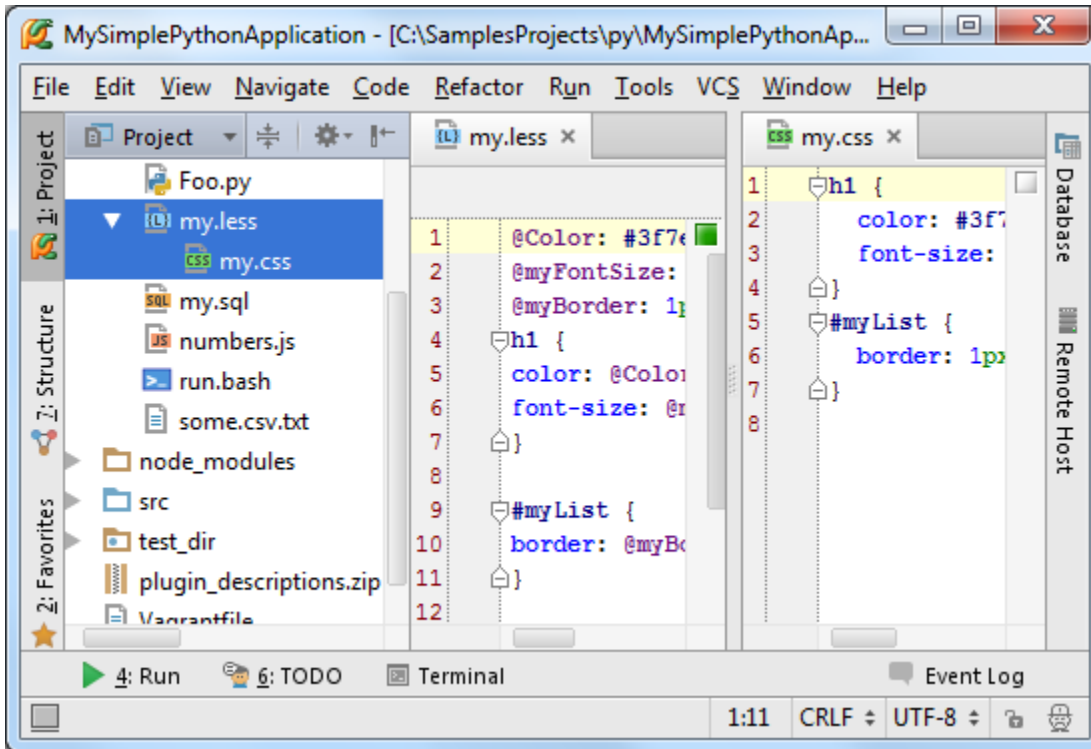
(1) 跟踪项目中所有 CoffeeScript 文件的变化。

(2) 使用外部扩展 `.coffee` 来编译处于外部扩展 `extension .js` 的文件，一般使用定义在 Program 域中的 `coffee.cmd` 编译器。

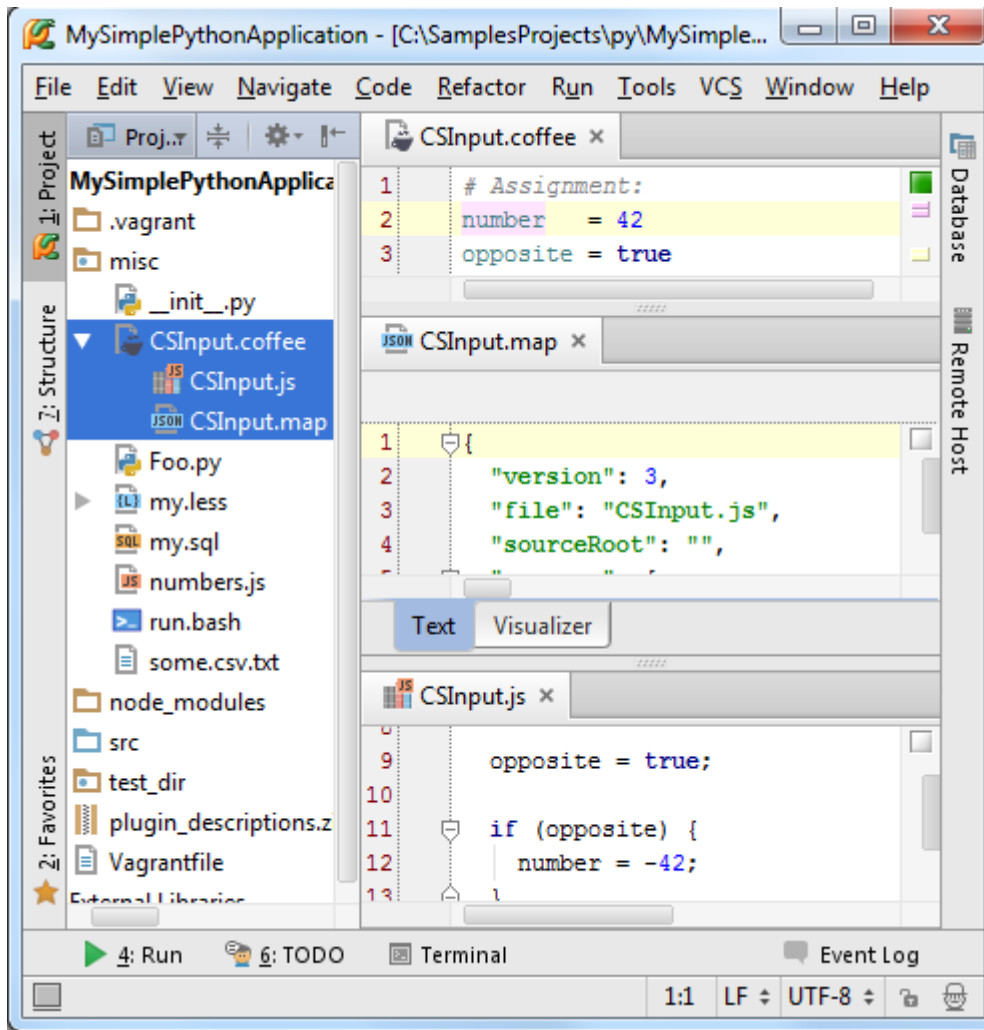
(3) 使用外部扩展 `.coffee` 来编译处于外部扩展 `extension .map` 的文件，一般使用定义在 Program 域中的 `coffee.cmd` 编译器。

8、编辑 file watcher

打开 LESS 文件，做一些改动，例如重命名变量 `@myColor` 为 `@Color`，并改变它的值。file watcher 会立即处理改动后的文件，并通过外部 `.css` 将其变成一个输出文件：



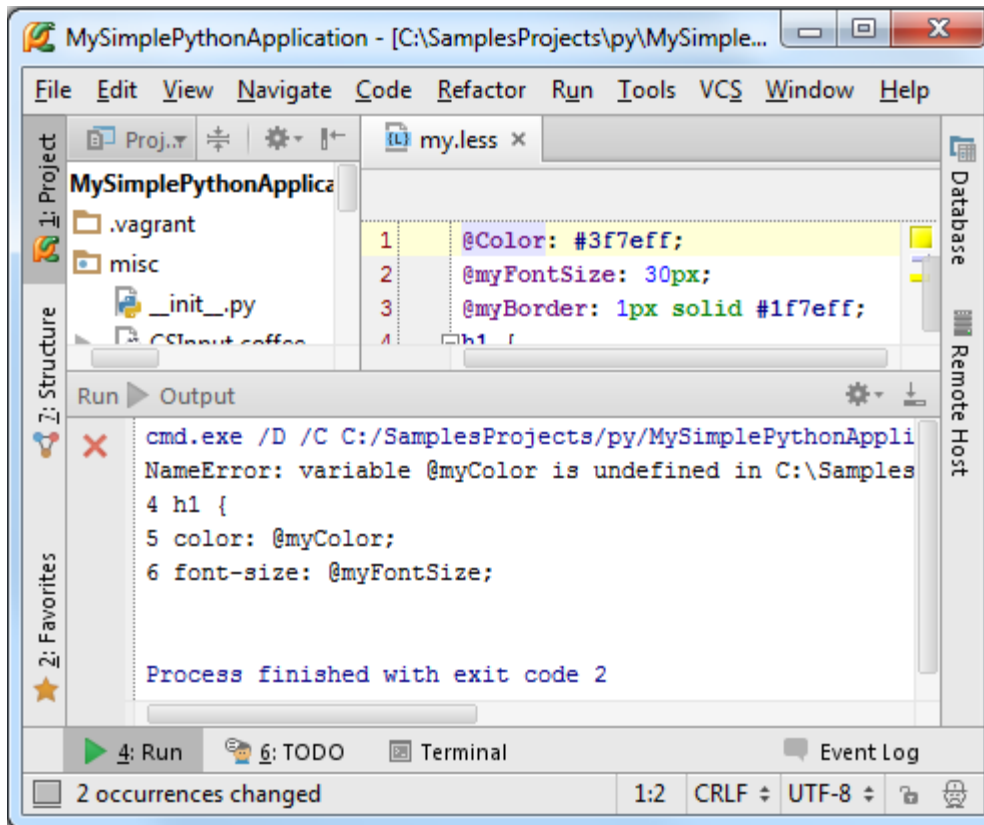
接下来打开并编辑 CoffeeScript 文件，做一些修改，例如更改 `opposite` 变量的值，file watcher 会生成一个 JavaScript 文件以及一个映射源文件：



注意 PyCharm 会在项目窗口中同时显示生成文件和源文件。

9、有问题？

File Watcher 执行失败，PyCharm 会给出提示以帮助修正：



最全 Pycharm 教程（31）——Pyhcharm 实战

1、主题

介绍如何用 Pycharm 实打实的创建、运行、调试程序。

2、准备工作

Pycharm 版本为 2.7 或者更高。

至少安装一个 Python 解释器，2.4 到 3.3 均可

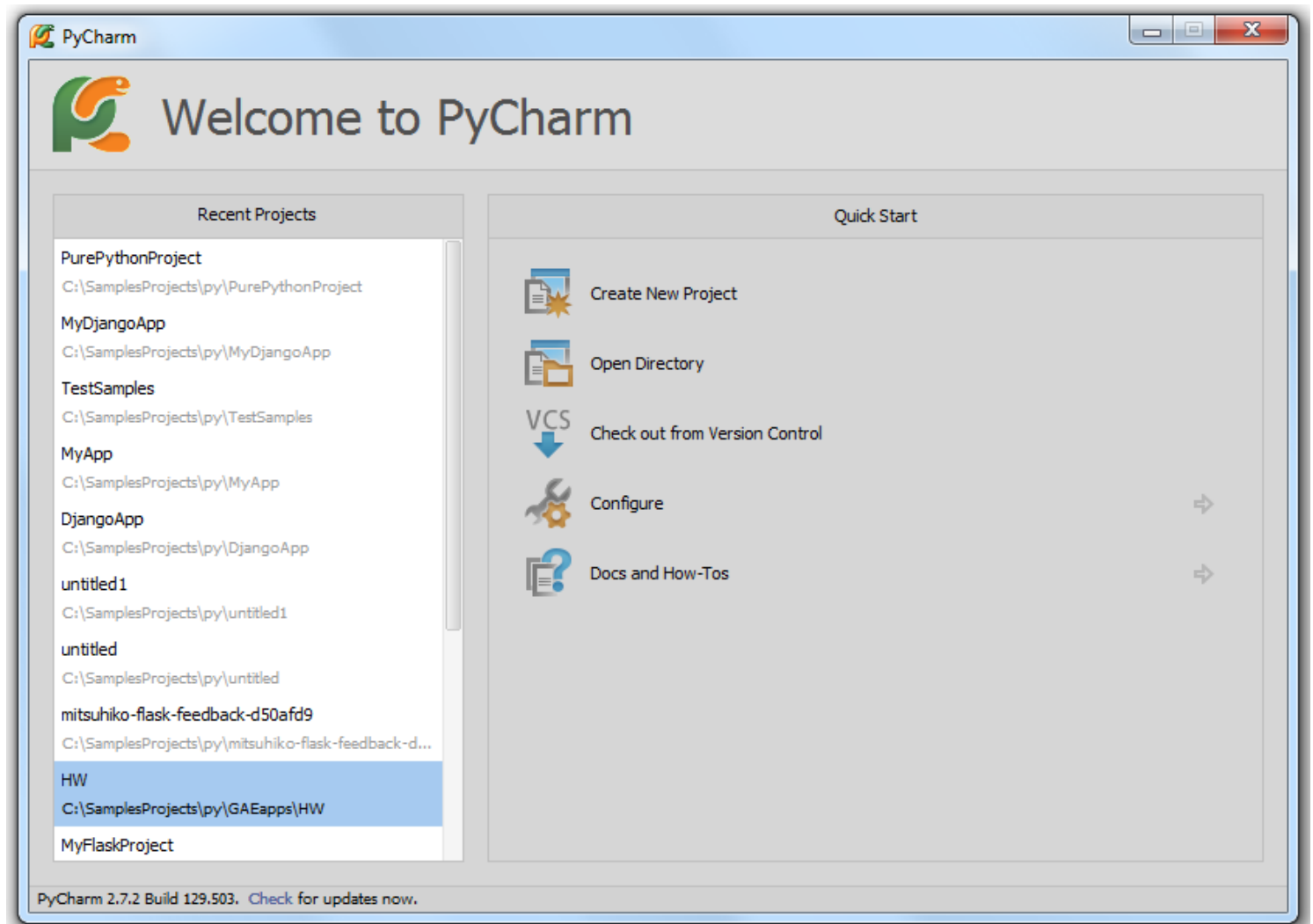
3、下载安装 Pycharm

下载地址：[this page](#)

4、启动 PyCharm

双击快捷方式（windows 中为 pycharm.exe 或者 pycharm.bat；MacOS and Linux 为 pycharm.sh），进入欢迎界面

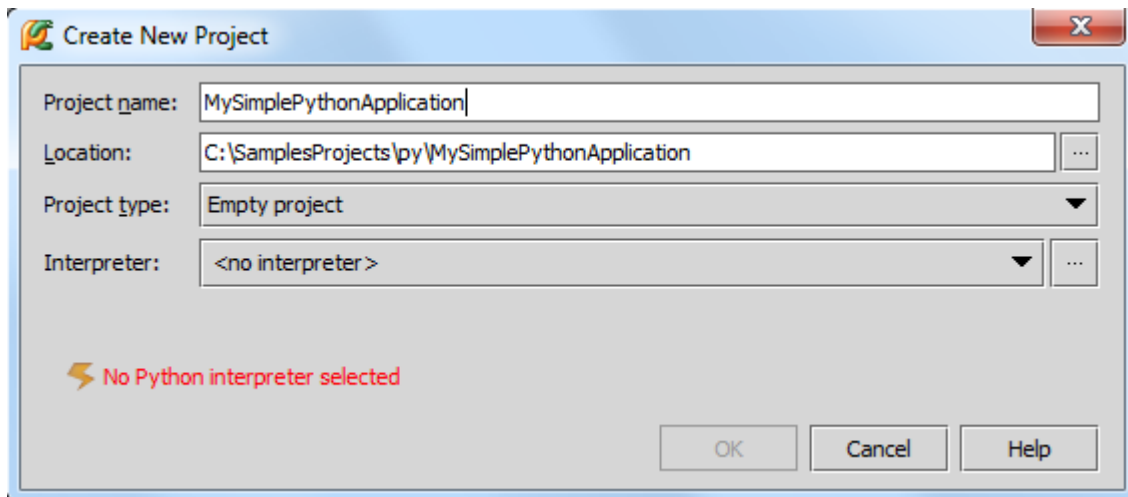
Welcome screen：



5、创建一个简单工程

单击 Create New Project 链接，进入创建工程对话框，进行相关工程设置。

当然，也可以通过主菜单命令 File → New Project 来随时创建新的工程：

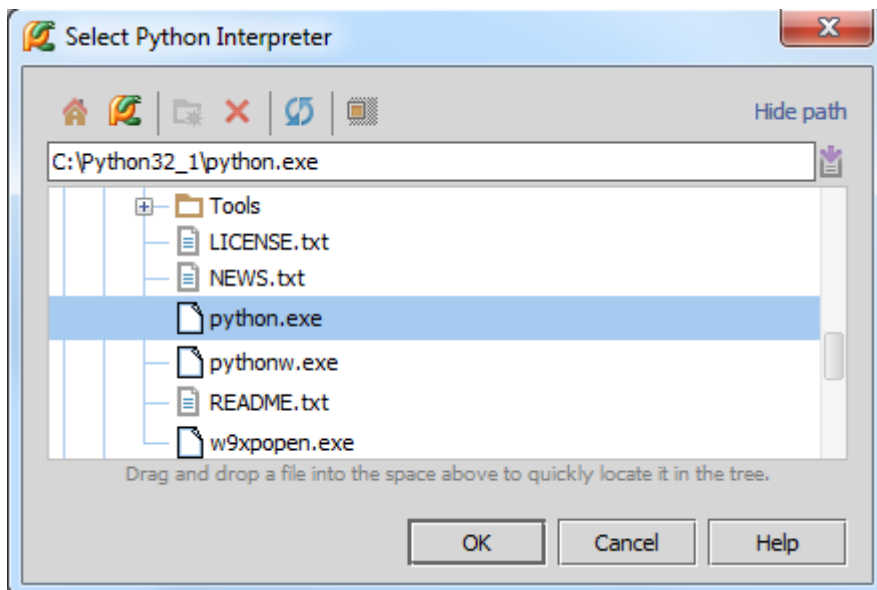


首先为工程命名，这里命名为 `MySimplePythonApplication`。然后更改工程位置，既可以使用默认的位置，也可以通过单击浏览按钮来指定。

接下来选择工程类型，Pycharm 预设了若干类型模板（Django, Google AppEngine 等等），并默认创建相关文件。

这里我们选择 *Empty project* 类型（比较适合简单的 Python 工程），不需要 Pycharm 预设任何文件。

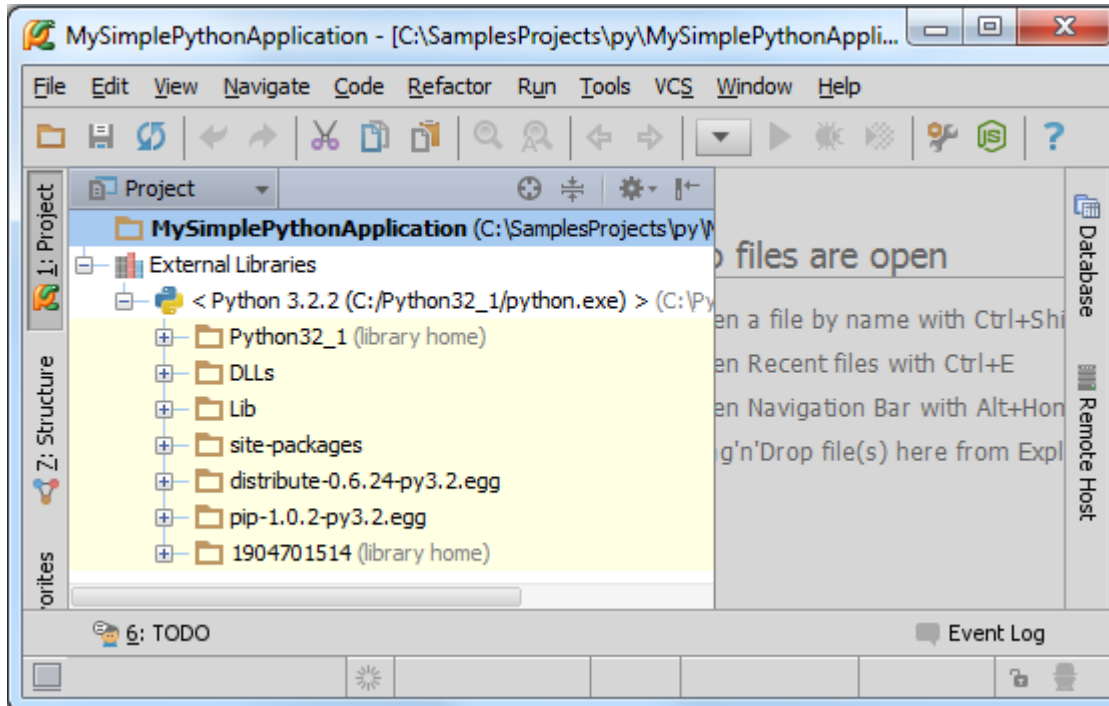
最后，指定 Python 解释器，在下拉列表中选择即可。



单击 OK 按钮，工程创建完毕。

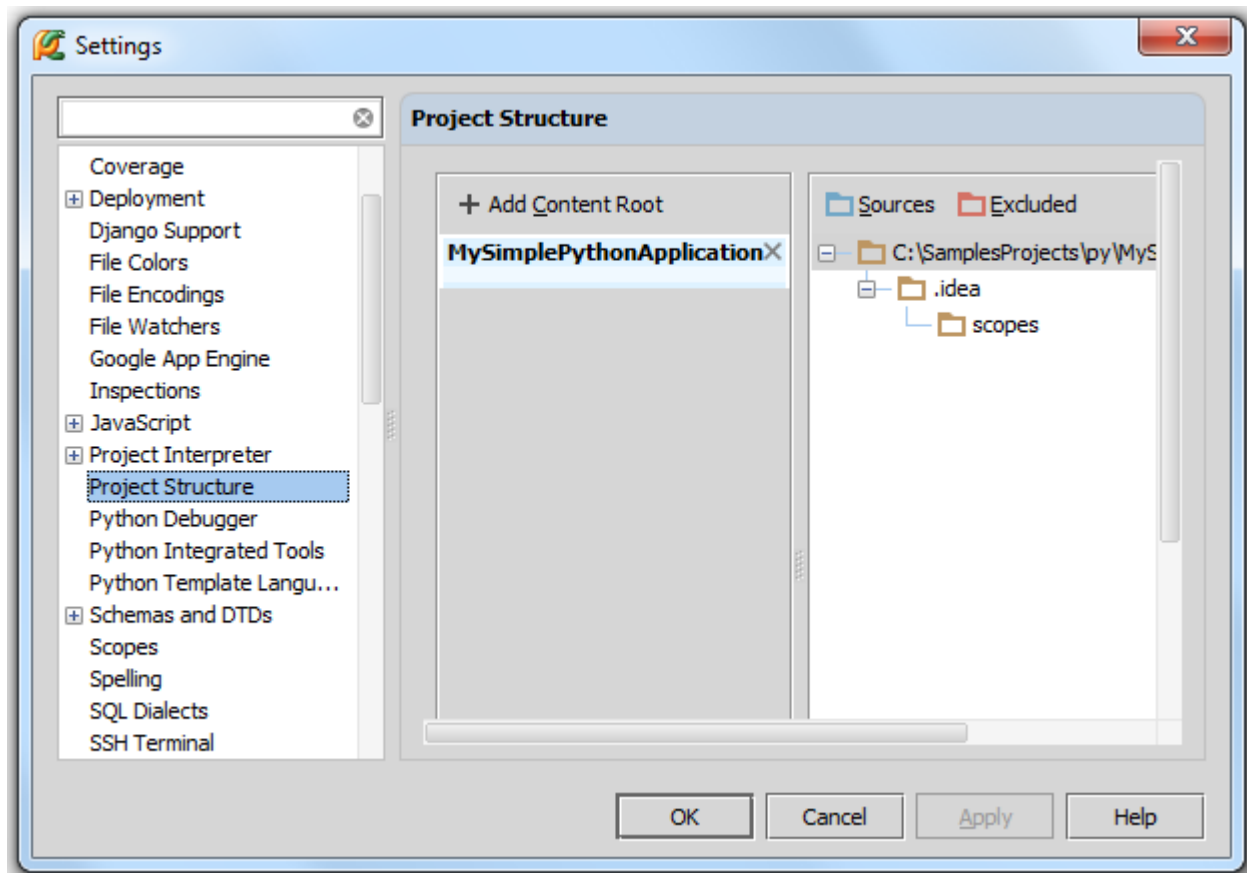
6、浏览工程目录结构

初始工程目录（[Project tool window](#) 中）如下：



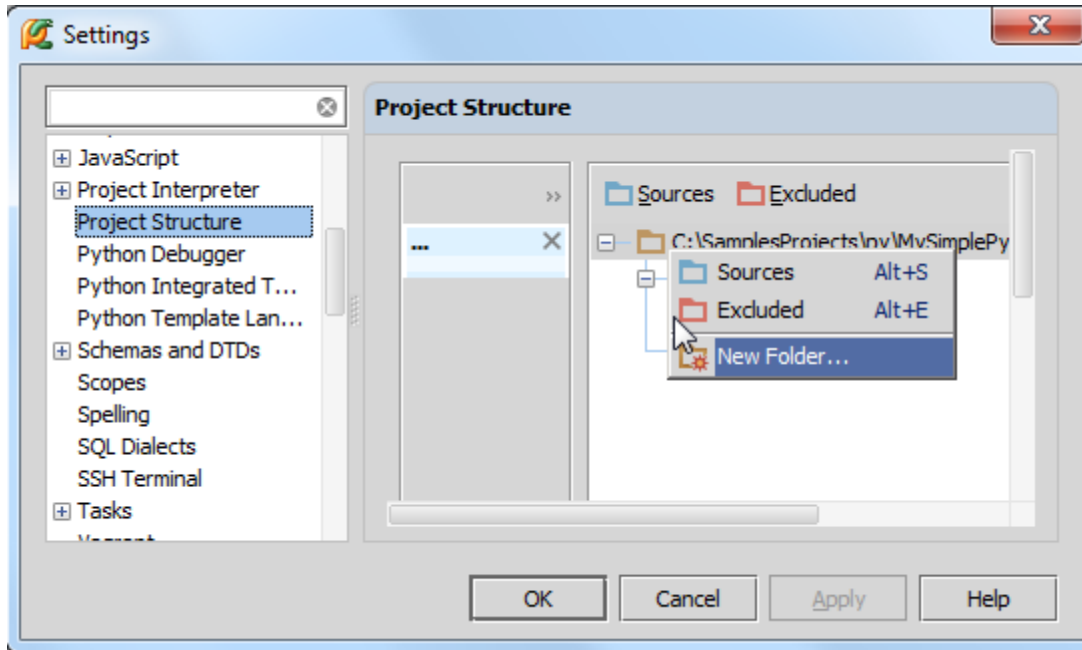
此时只存在工程根目录以及定义了 Python 解释器的 External Libraries 目录。

单击主工具栏的  按钮，选择 **Project Structure** 页，查看详细工程目录信息：

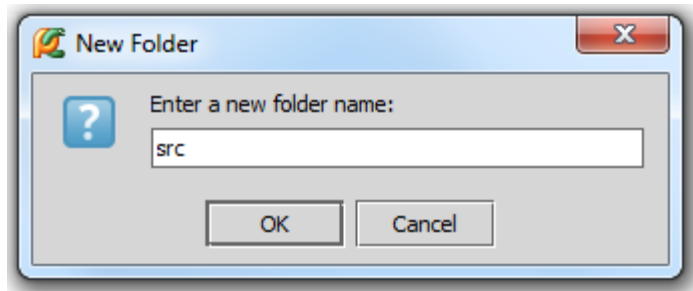



在工程根目录下的 idea 目录下存放了 MySimplePythonApplication.iml 文件，用以记录当前的工程结构；目录下还有若干 XML 文件，保存着相关的配置信息。idea 目录在 Project tool window 窗口中是不可见的。

接下来向根目录中添加工作目录。在 Project Structure 页，右击工程根目录，选择 New Folder：



输入目录名称：



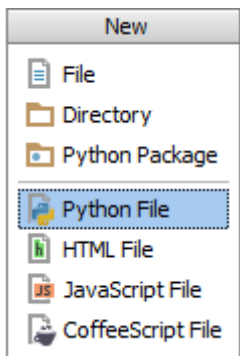
最后，将该目录标记为源文件根目录：选择 src 目录，单击 ，标记完成。

单击 OK，关闭设置对话框。

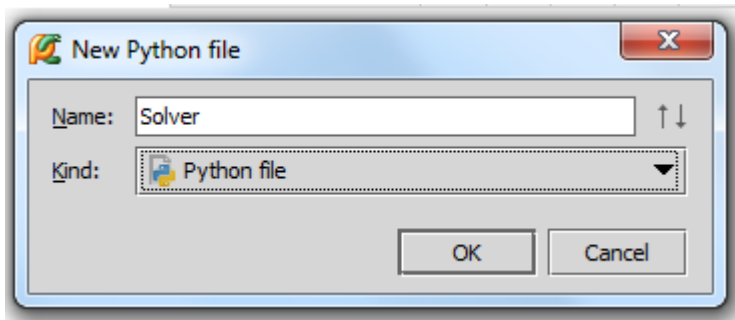
当然添加目录的方式并不是唯一的，也可以直接在工程根目录下创建 Python 文件，此时工程目录被默认为源文件根目录。

7、创建 Python 类

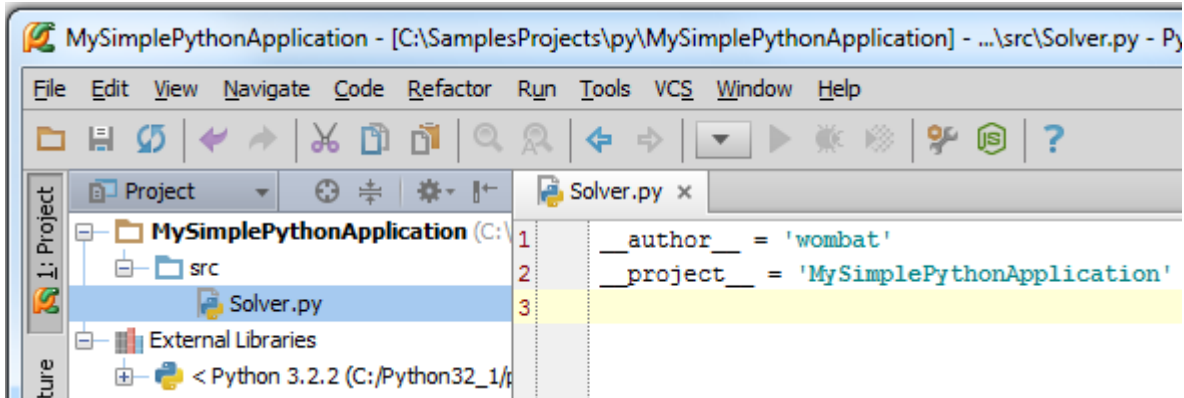
在 project tool window 窗口中选择 src 目录，按下 Alt+Insert：



选择 Python file，输入名称（Solver）：



类创建完成，打开编辑：



8、编辑源码

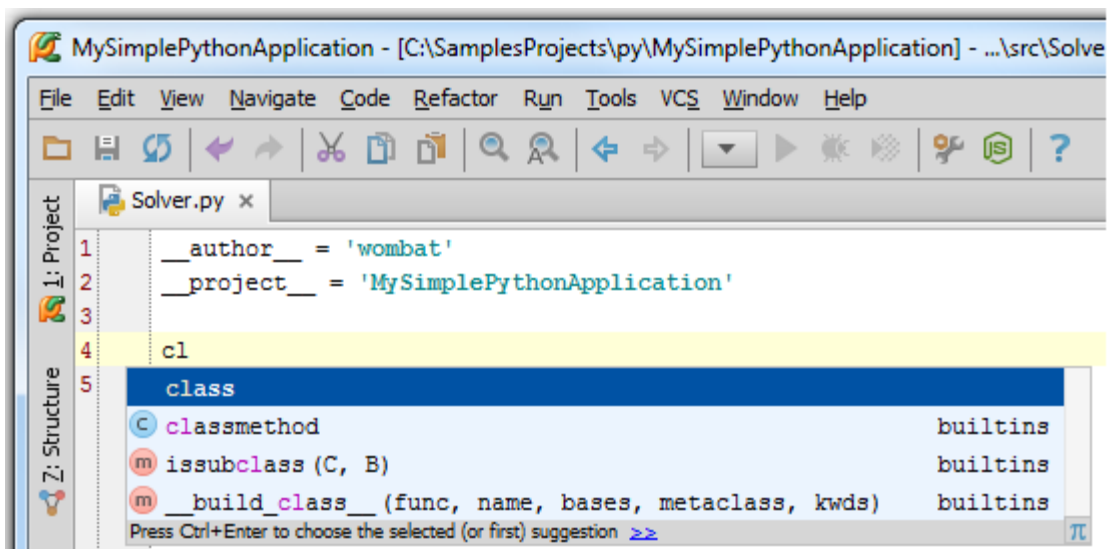
首先，文件中有两行默认生成的代码：

```
__author__ = 'wombat'  
__project__ = 'MySimplePythonApplication'
```

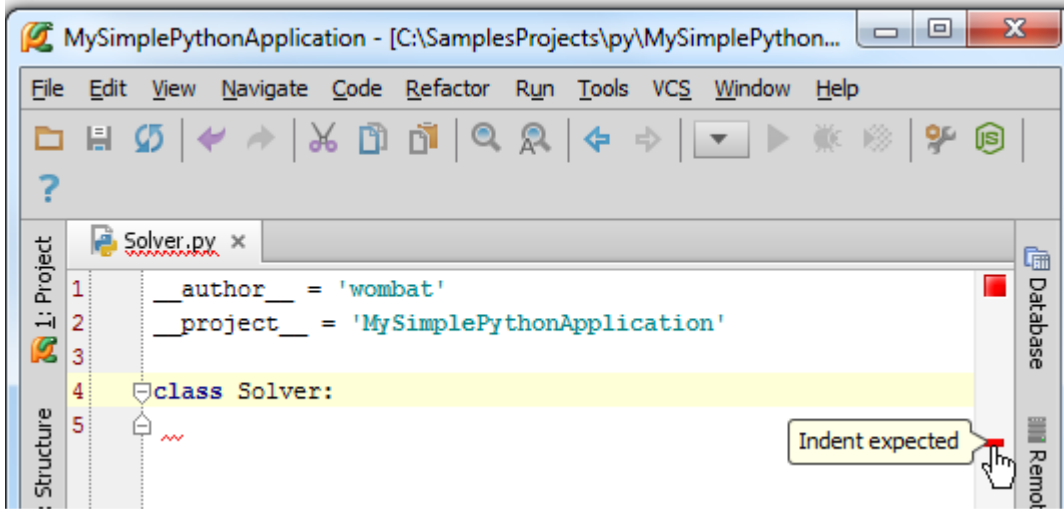
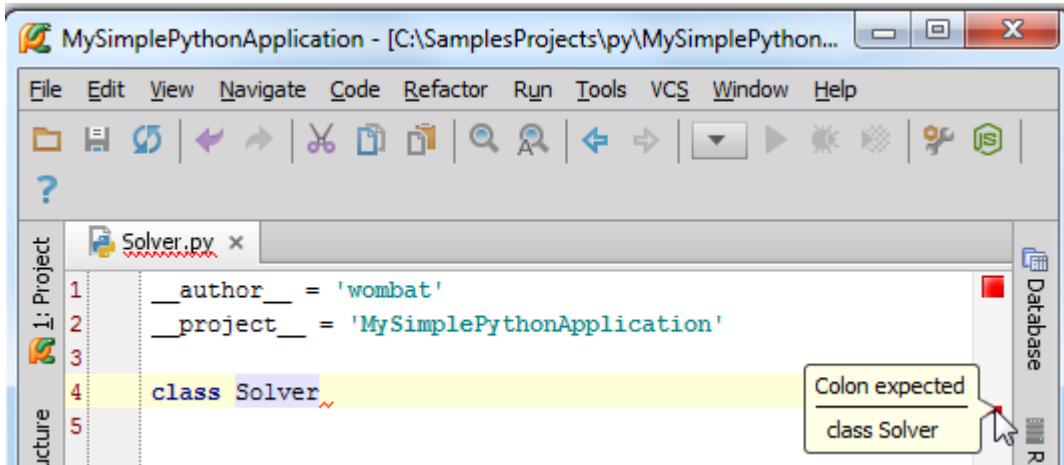
这是 Pycharm 根据模板 [file template](#) 生成的，并自动替换了形式变量 \$PROJECT_NAME 和 \$USER。

接下来我们编写一个求解二次方程的小程序。

在编写代码的过程中 Pycharm 会提供各种各样的提示帮助，例如在创建类时，只需输入关键字，则会弹出提示列表：

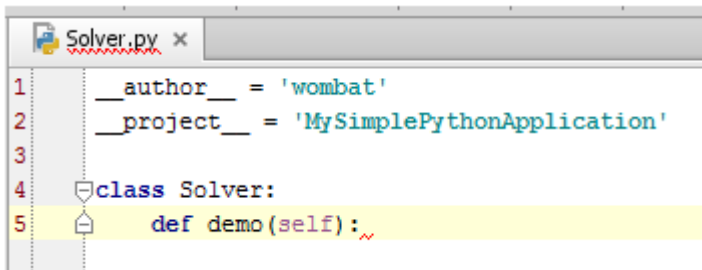


选择关键字 class，输入类名（Solver），Pycharm 会提示你继续输入：

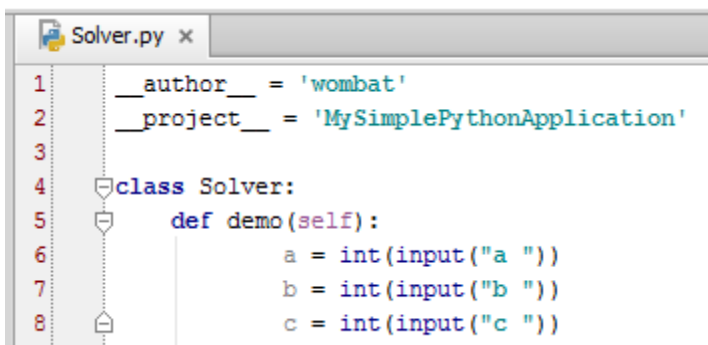


此时右槽会显示错误标记，鼠标悬停在上方时会给出错误提示。右槽顶部的指示灯标记了当前的代码检查状态，绿色代表一切正常，黄色代表有警告，红色代表有错误。

继续创建“demo”成员函数，体会 Pycharm 代码的自动补全功能：



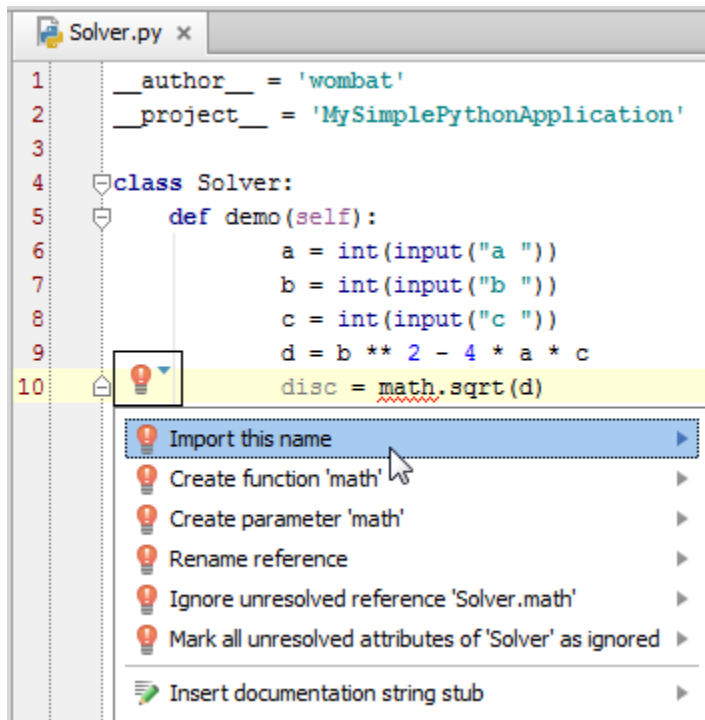
继续输入，未用到的变量以灰色显示：



接下来计算判别式，用到了 `math` 模块，由于尚未导入，Pycharm 会报错（红色波浪线和红色灯泡）。

红色灯泡的使用机制参见 [intention actions and quick fixes](#)，格式问题为黄色灯泡，出现错误即为红色灯泡。

按下 `Alt+Enter` 查看 Pycharm 给出的提示：



选择导入 `math` 库，然后晚上 `Solver` 类中的 `demo` 函数，计算判别式：

```
import math

__author__ = 'wombat'
__project__ = 'MySimplePythonApplication'

class Solver:
    def demo(self):

        a = int(input("a "))
        b = int(input("b "))
        c = int(input("c "))
        d = b ** 2 - 4 * a * c
        disc = math.sqrt(d)
        root1 = (-b + disc) / (2 * a)
        root2 = (-b - disc) / (2 * a)
        print(root1, root2)

Solver().demo()
```

按下 `Ctrl+Shift+F10` 运行脚本文件，出现一个控制台，输入 `a`、`b`、`c` 的值，发现 Pycharm 遇到一个错误：


```
Run Solver
C:\Python32_1\python.exe C:/SamplesProjects/py/MySimplePythonA
a 1
b 2
c 3
Traceback (most recent call last):
  File "C:/SamplesProjects/py/MySimplePythonApplication/src/Sc
    Solver().demo()
  File "C:/SamplesProjects/py/MySimplePythonApplication/src/Sc
    disc = math.sqrt(d)
ValueError: math domain error

Process finished with exit code 1
```

这里是当 d (判别式) 为负数时, 程序报错。为了避免这种情况, 加入判断语句 `Ctrl+Alt+T` (Code→Surround with) :

```
d = b ** 2 - 4 * a * c
disc = math.sqrt(d)
root1 = (-b + disc) / (2 * a)
root2 = (-b - disc) / (2 * a)
print(root1, root2)

Solver().demo()
```

- 1. if
- 2. while
- 3. try / except
- 4. try / finally

Pycharm 会自动创建一个 `if` 语句结构。最后如果你希望多次执行该程序, 需要再在外层嵌套一个 `while` 循环, 代码最终效果如下:

```
import math

__author__ = 'wombat'
__project__ = 'MySimplePythonApplication'

class Solver:
    def demo(self):
        while True:
            a = int(input("a "))
            b = int(input("b "))
            c = int(input("c "))
            d = b ** 2 - 4 * a * c
            if d >= 0:
                disc = math.sqrt(d)
                root1 = (-b + disc) / (2 * a)
                root2 = (-b - disc) / (2 * a)
                print(root1, root2)
            else:
                print('error')

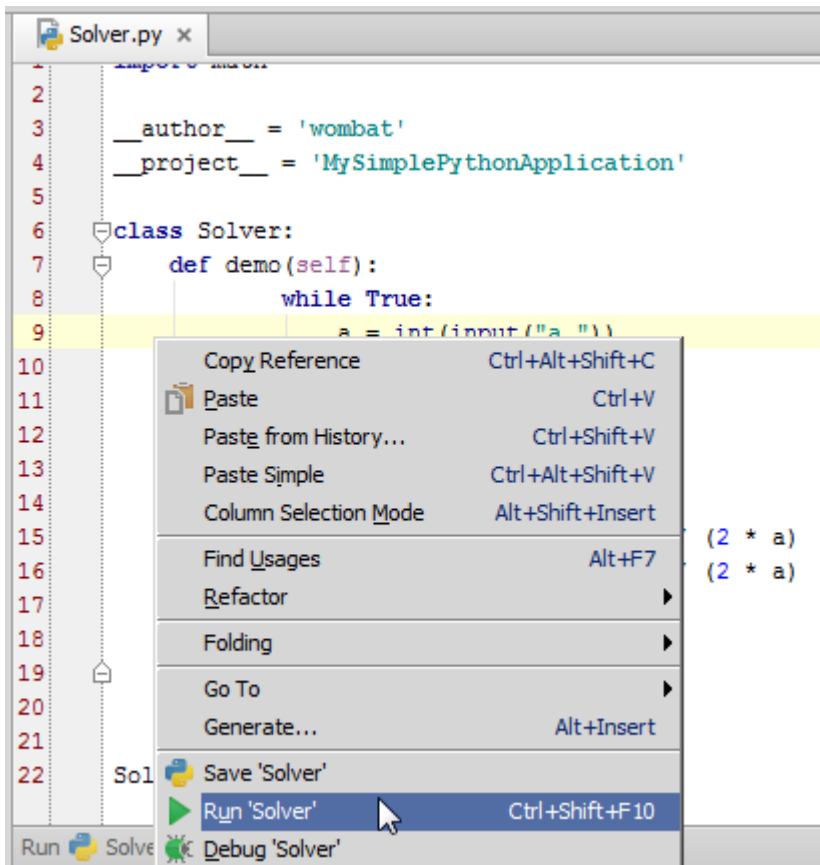
Solver().demo()
```

接下来，准备调试。

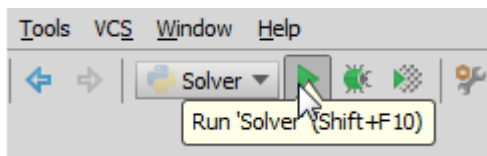
9、运行程序

三种运行脚本文件的方式：

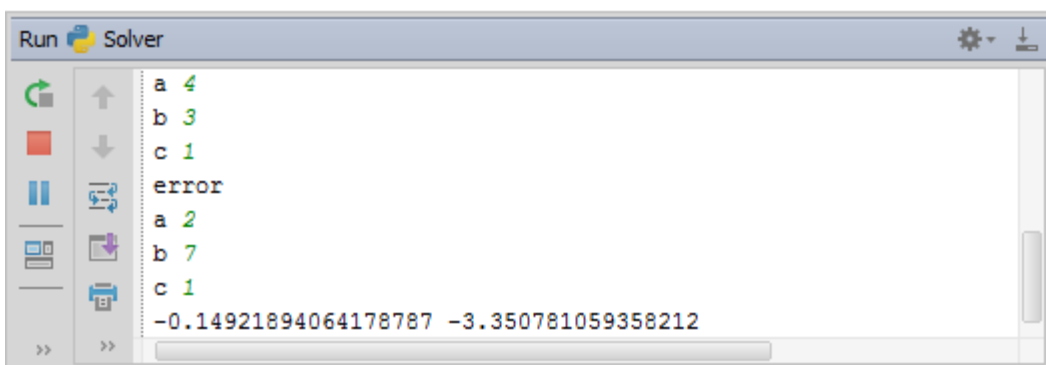
- (1) Ctrl+Shift+F10 快捷键
- (2) 使用快捷菜单选项



(3) 使用主菜单的运行按钮



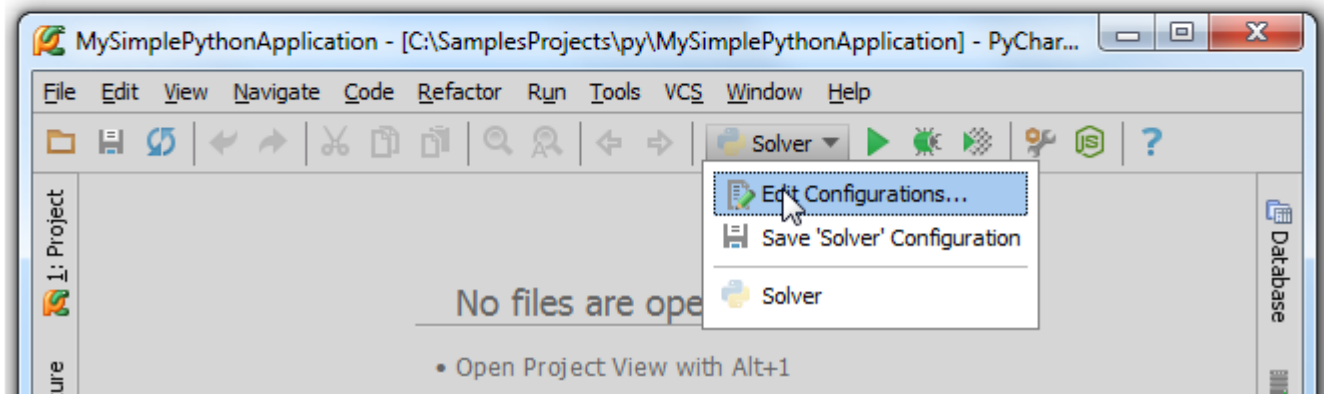
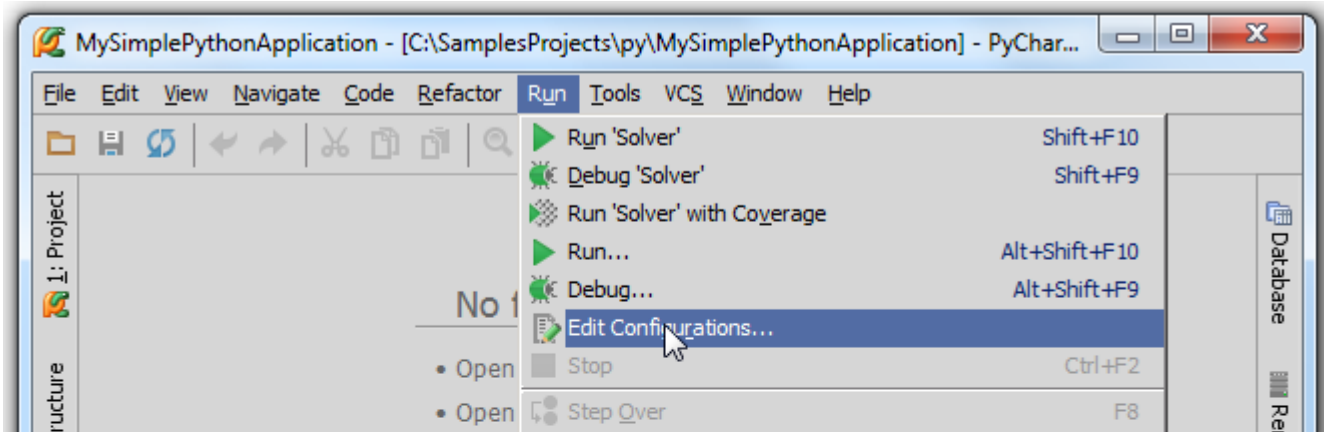
查看运行结果：



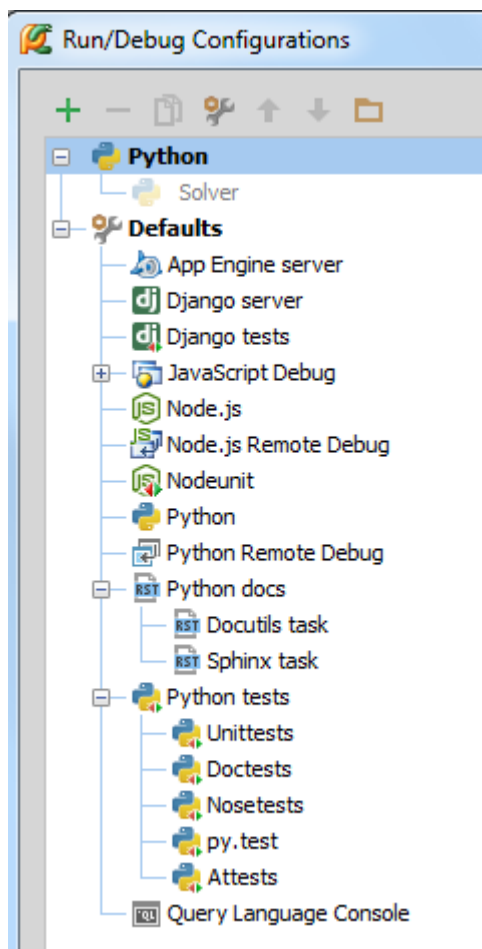
10、运行/调试相关配置

每个脚本文件在运行和调试时都按照指定的配置文件的规定（[run/debug configuration](#)）执行，包括脚本名称、工作目录、预处理等等。

Pycharm 已经预设了若干中常规的配置文件的类型（针对 Python scripts, Django applications, tests, 等等），可以在 [Run/Debug Configurations dialog](#) 对话框中浏览这些配置。可以通过 `Run→Edit Configurations...` 命令或者单击主工具栏 `Run` 区域的下拉列表来打开这个对话框：



详细查看 Edit Configurations 对话框，其包含两个主要部分：Python 和 Default



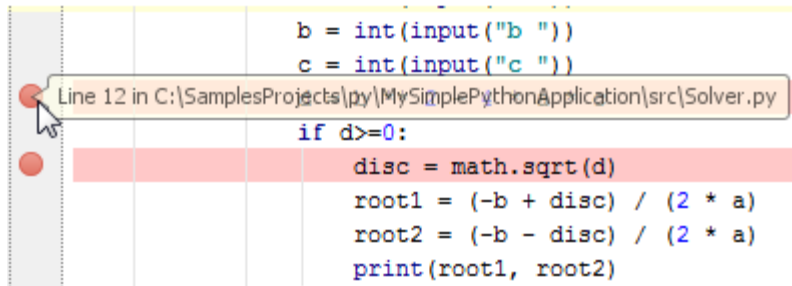
default run/debug 列表下的内容是默认的配置信息，他们没有具体名字，但会根据类型自动加载使用。

上方名为 Python 的节点只包含一个灰色显示的配置文件 *Solver*。它是一个临时配置文件 *temporary profile*，是 Pycharm 创建的缺省配置 *default configuration of the Python type*。

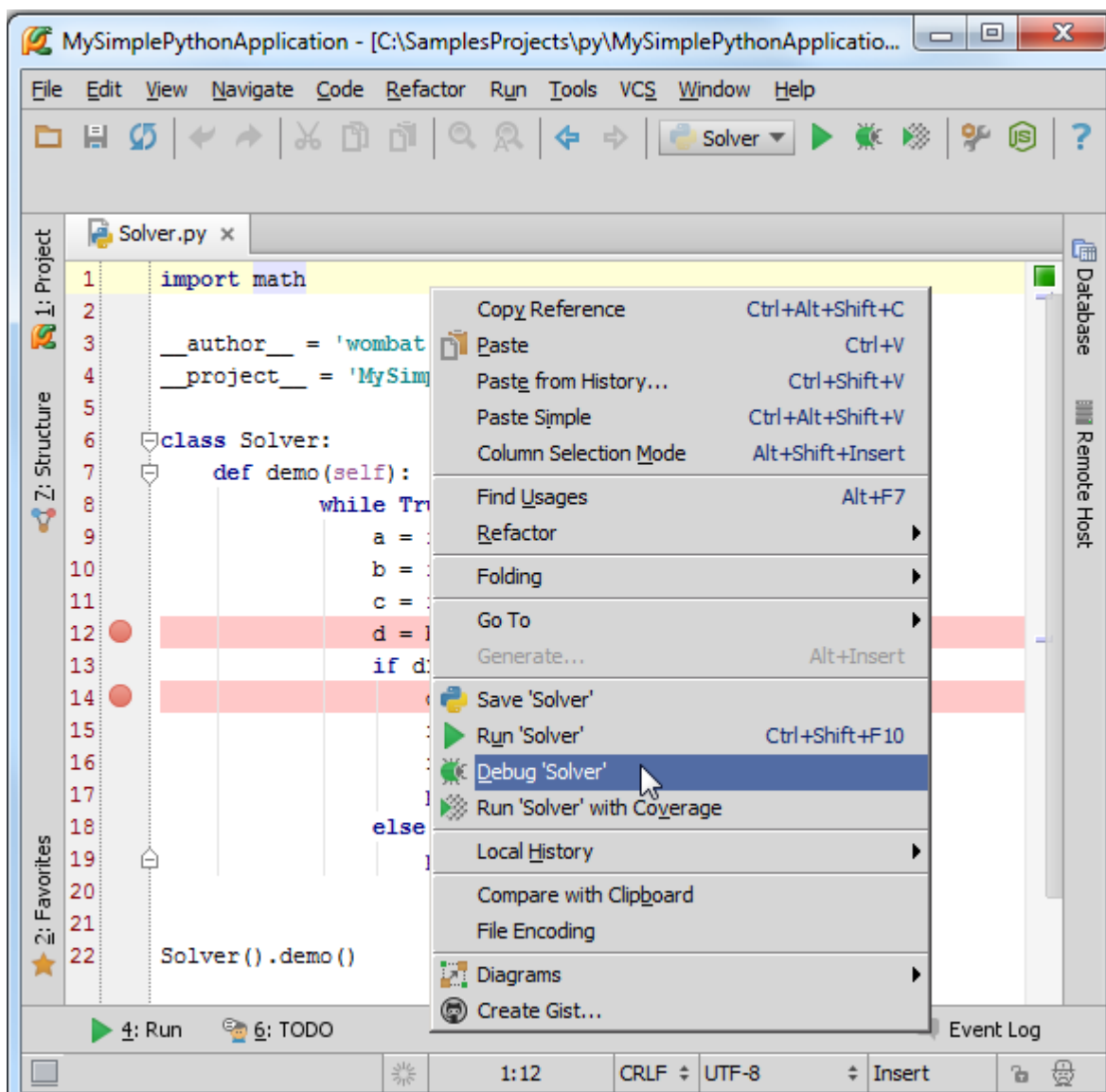
你可以永久保存你的配置文件，数量不限。

11、调试程序

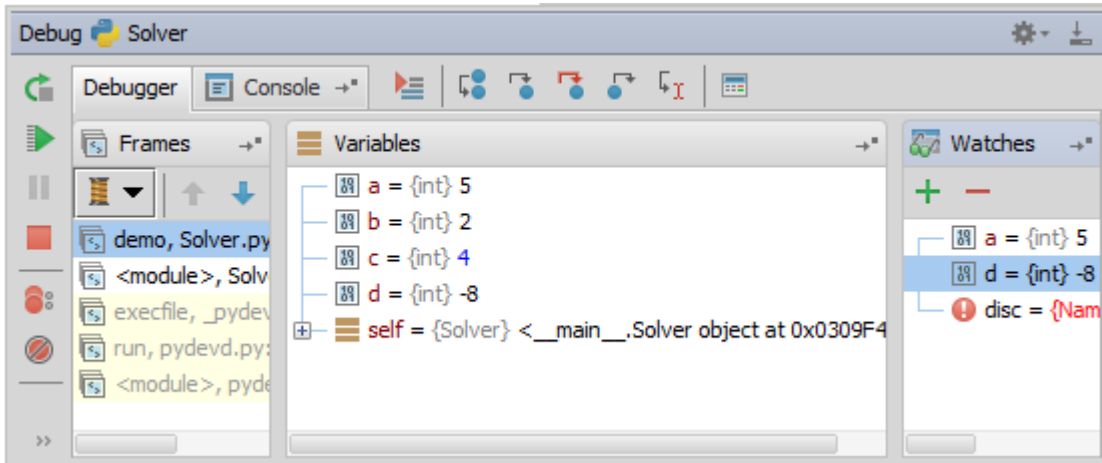
调试之前先设断点 *breakpoints*，单击左槽即可：



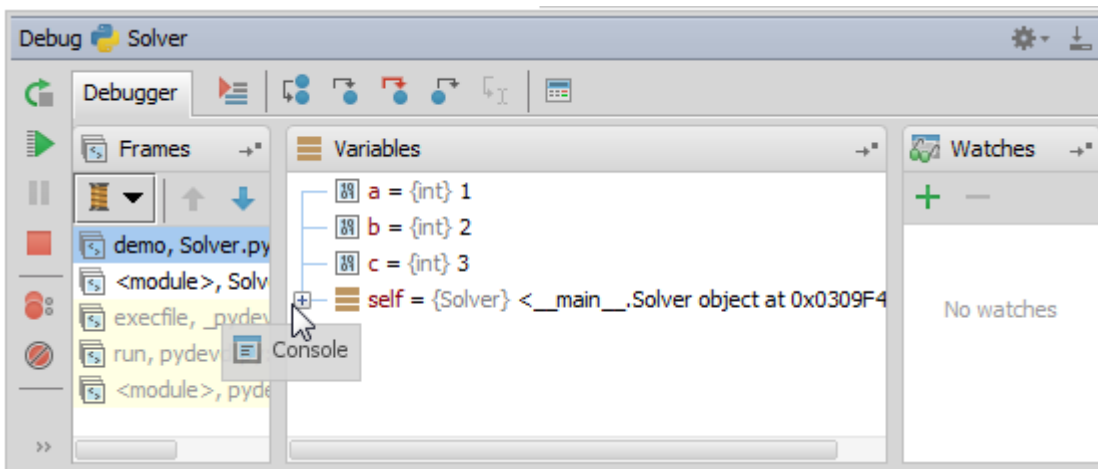
然后右击编辑区域，选择 *Debug 'Solver'*：



显示 *Debug tool window* 窗口，调试开始，调试窗口的默认布局如下：



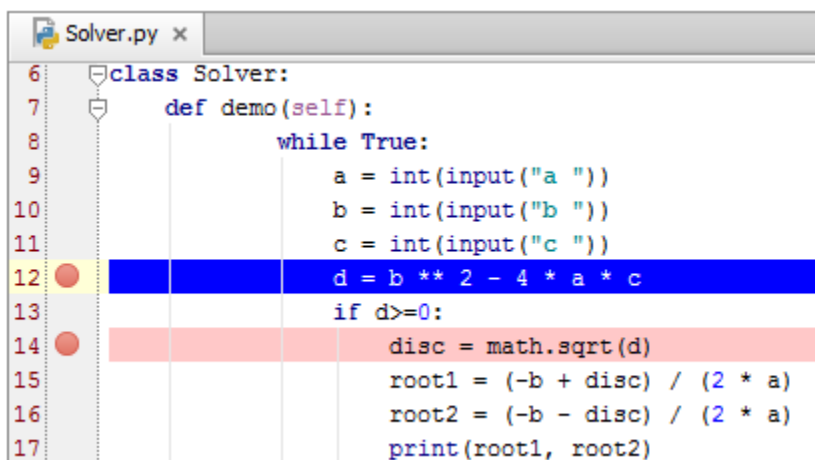
这里显示了框架、变量、控制台等。当然如果你希望控制台一直可见的话，将其拖动到指定区域即可：



使用 [stepping toolbar buttons](#) 来单步调试：



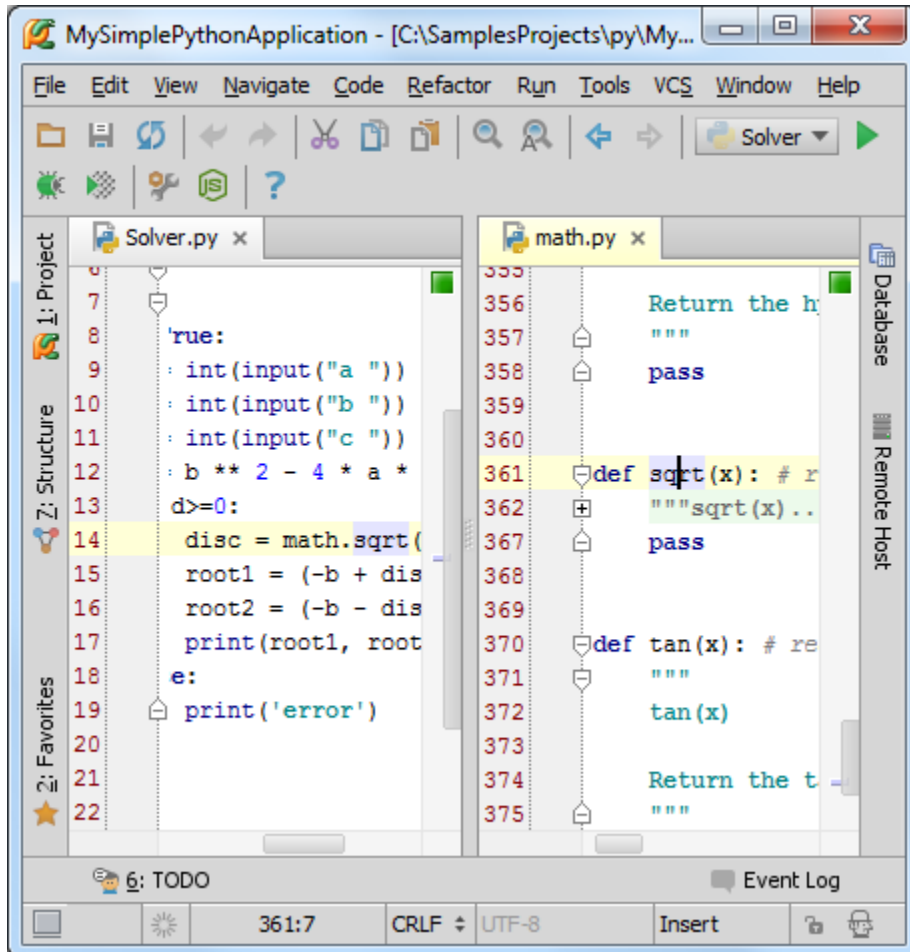
命中断点，对应行变蓝：



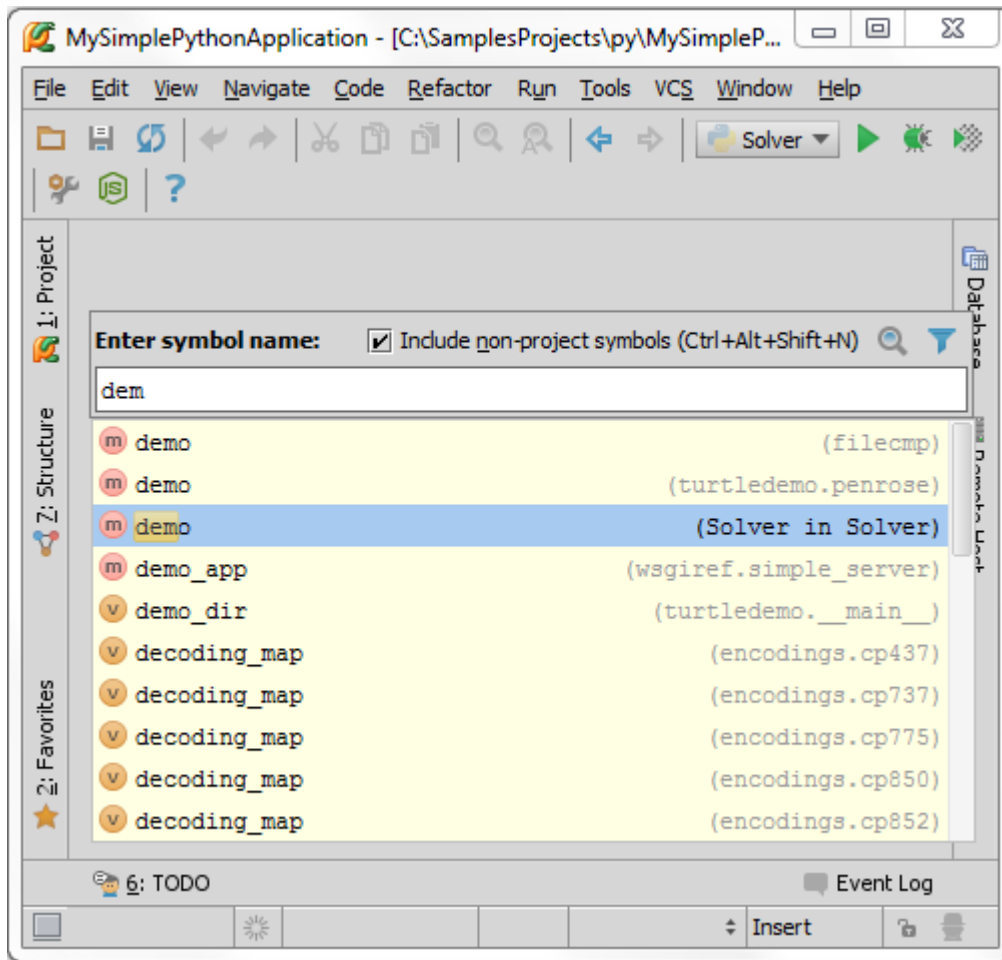
12、导航功能初探

假如你编程过程中中途中断，回来时不知道该从哪里继续开始，这就用到一个重要的导航功能：跳转到最后编辑位置。按下 `Ctrl+Shift+Backspace` 即可。

快速查看符号定义，例如将光标定位在 `sqrt` 的调用处，按下 `Ctrl+B`，Pycharm 会跳转到 `math.py` 的指定定义位置：



快速查找符号、类、文件。按下 `Ctrl+Alt+Shift+N`，输入名称即可：

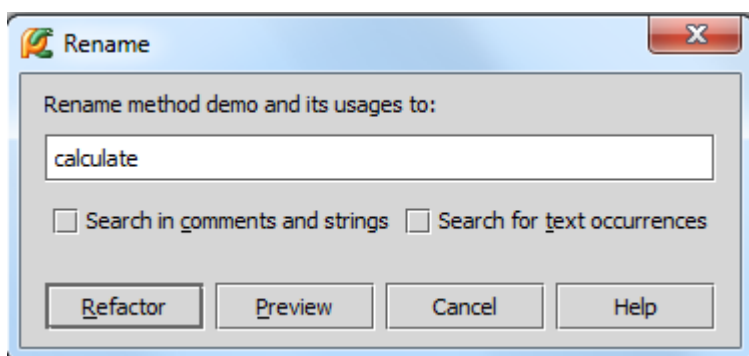


更多详情参见 [here](#)。

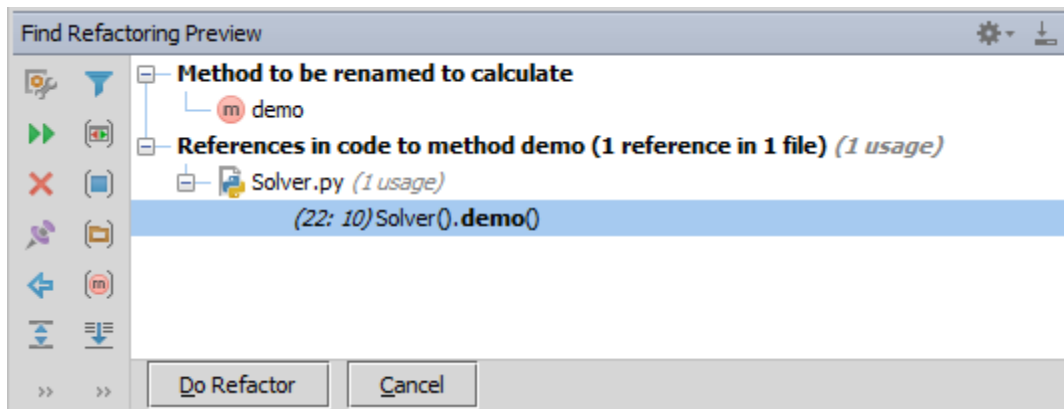
13、代码重构

假如你更改了一个函数 `demo` 的名称，理论上对其所有的调用都需要进行更改，这里 Pycharm 提供了代码重构功能。

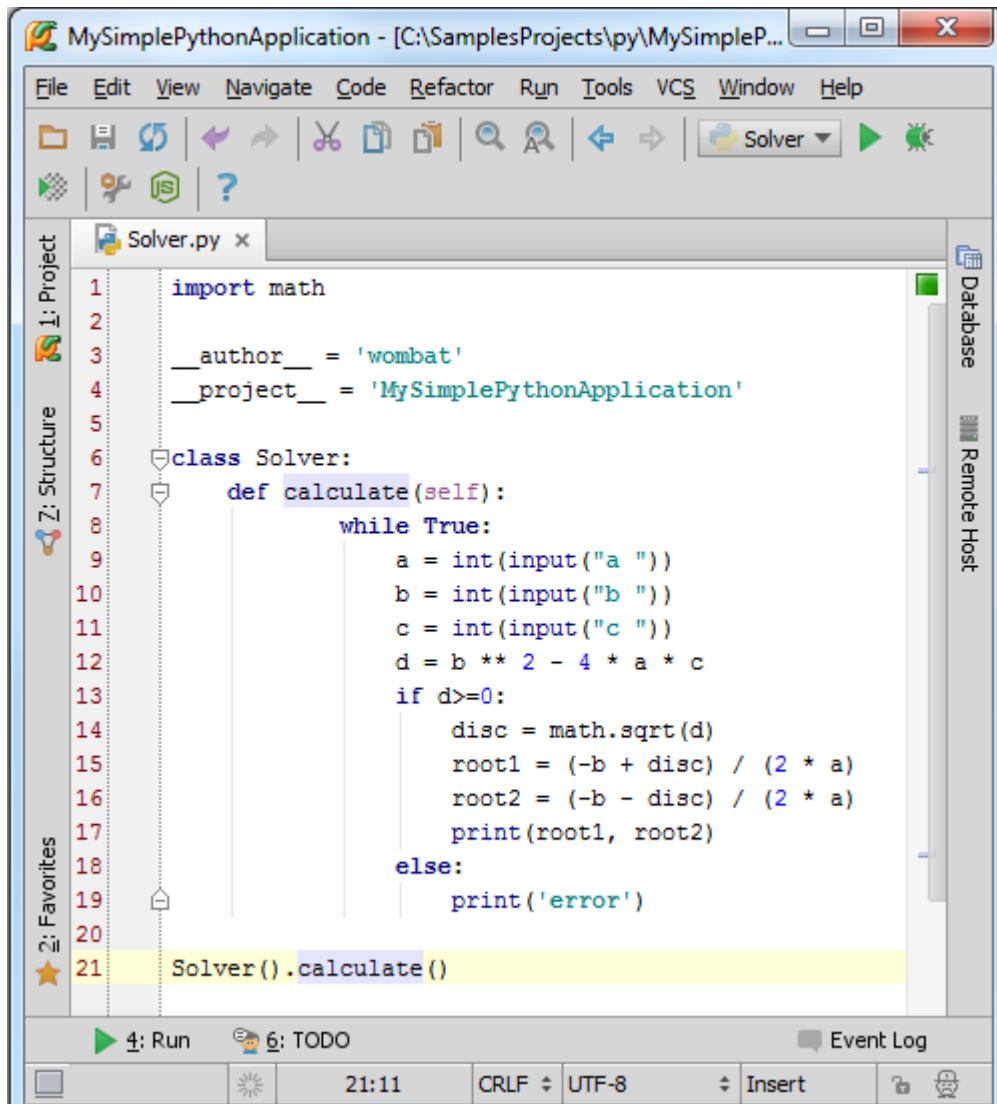
按下 `Shift+F6`，在对话框中输入新名称：



单击 Refactor，在 [Find tool window](#) 显示查找结果：



单击 Do Refacto 按钮完成替换:



当然我们还可以做更多改变，例如移动文件位置、改变函数的参数结构、提取变量等。这些都属于各种各样的重构。我们会在以后的教程中详细介绍。

最全 Pycharm 教程（32）——根据 FHS 在 Linux 上安装 Pycharm

1、主题

如何在 Linux 上安装 Pycharm，根据 [FHS](#)。

2、平台要求

Intel Pentium III/800 MHz 或更高。

内存最小 512M，建议 1G 以上

屏幕最小分辨率 1024x768

[Oracle \(Sun\) JDK 1.6](#) 以及 [Open JDK 1.7](#) 或者更高版本

[GNOME](#) 或者 [KDE](#) 桌面

Python2.4 或更高，Jython、PyPy 、IronPython

确认已经获得目标目录/opt 的读写权限。

3、下载安装文件

现在 Linux 版本 [Download PyCharm](#):



Download PyCharm



根据需要下载安装文件

`pycharm-professional-<version number>.tar.gz`

或者

`pycharm-community-<version number>.tar.gz`

4、解压并安装

将安装文件拷贝或移动到安装目录/opt:

```
sudo mv pycharm-professional-4.0.1.tar.gz /opt/
```

进入目录:

```
cd /opt/
```

使用如下命令解压目标文件:

```
sudo tar -xzvf /opt/pycharm-professional-4.0.1.tar.gz
```

对应字母含义:

- x - 从压缩包中提取文件
- f - 处理文件
- z - 使用压缩程序
- v - 使用贪婪模式

最后, 移除下载的安装包:

```
sudo rm pycharm-professional-4.0.1.tar.gz
```

安装完毕。

5、运行

安装完成后使用如下命令加载/opt/pycharm-<version number> 目录:

```
bin/pycharm.sh
```

更为快捷的方法是为这个文件设置一个链接:

```
sudo ln -s /opt/pycharm-4.0.1/bin/pycharm.sh /usr/bin/pycharm
```

至此可以在 Ubuntu 主菜单中正常使用 Pycharm 了。

最全 Pycharm 教程（33）——使用 Pycharm 编写 IPython Notebook 文件

1、主题

详细介绍如何使用 PyCharm 创建一个 IPython Notebook（基于 Web 技术的交互式计算文档格式）并运行。

2、准备工作

(1) 已经创建一个工程，这里使用 C:/SampleProjects/py/IPythonNotebookExample 目录下的工程。

(2) 在设置对话框的 [Project Interpreter page](#) 页面中，

创建一个虚拟环境 [created a virtual environment](#)，这里创建了针对 Python.2.7.8 的虚拟环境

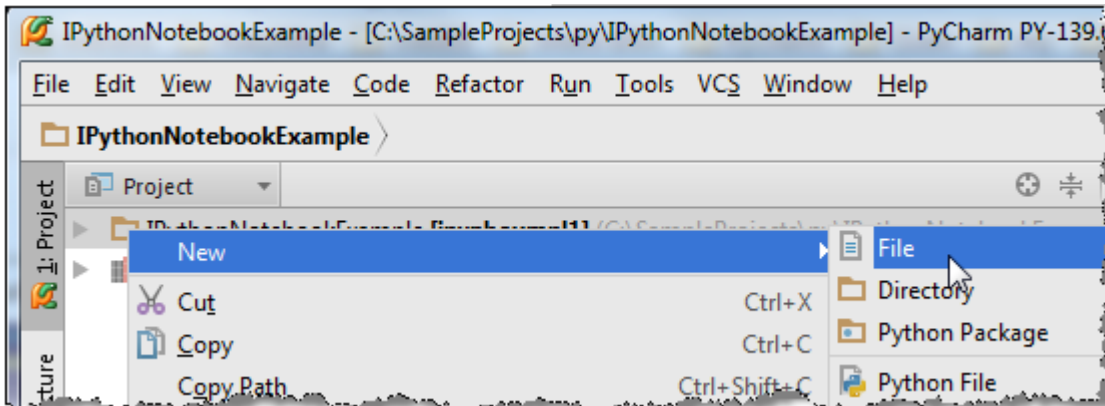
安装了以下库 [installed the following packages](#)：

- Jinja2
- ipython
- matplotlib
- numpy
- pyzmq
- tornado
- sympy

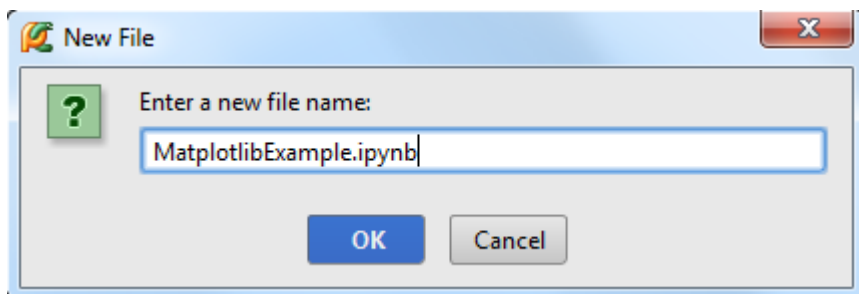
当然 Pycharm 可以帮助我们自动进行安装。

3、创建一个 IPython Notebook 文件

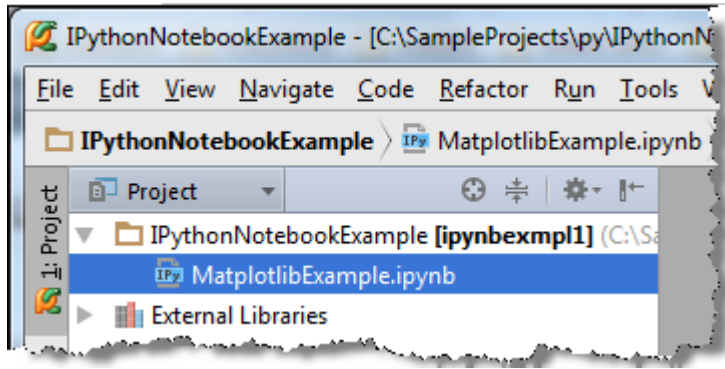
右击工程名，选择 New → File：




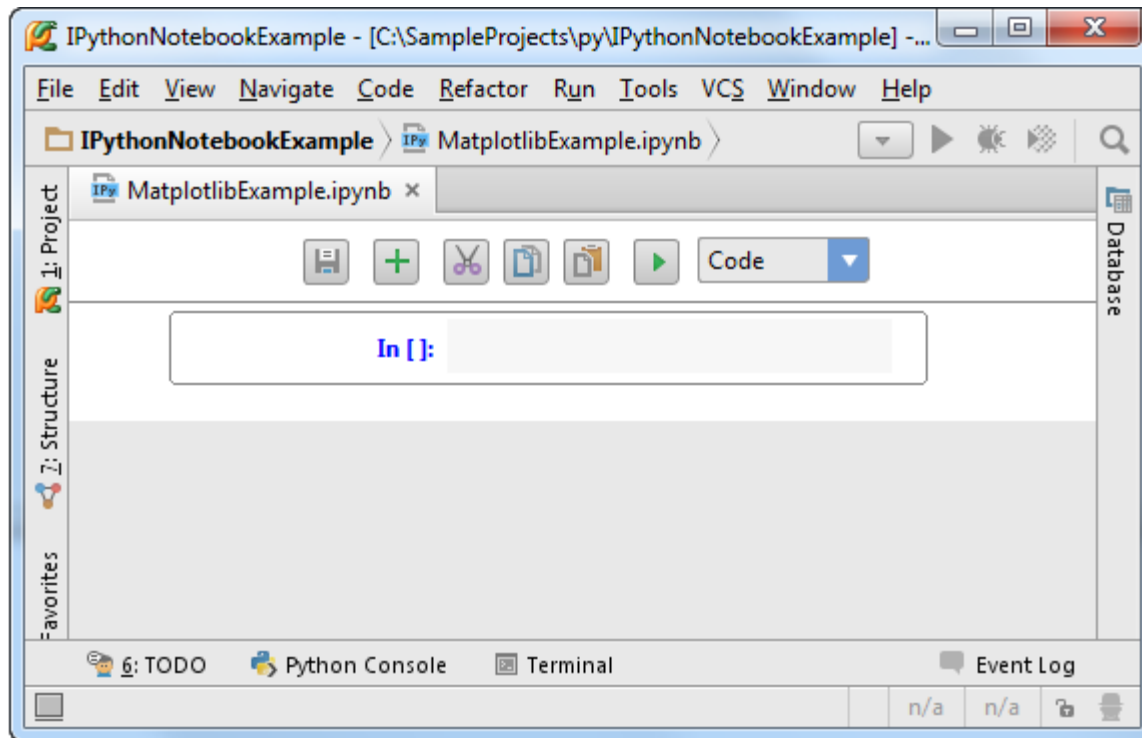
输入名称及其外部扩展 MatplotlibExample.ipynb：



此时在 [Project tool window](#) 显示了一个新创建的文件：




双击打开它，这是一个空的 IPython Notebook 文件，以  为标记，并有着与真正的 IPython Notebook 非常相似的工
具栏：

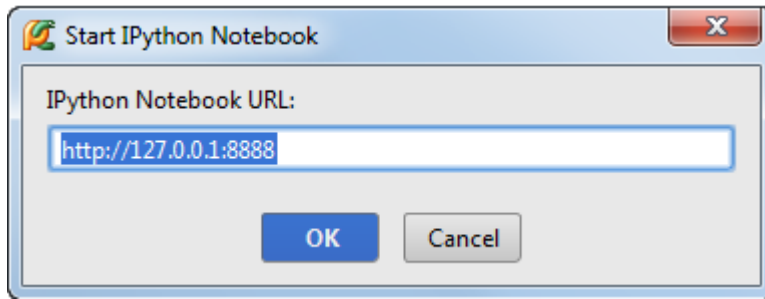


4、填充并运行一个文件胞

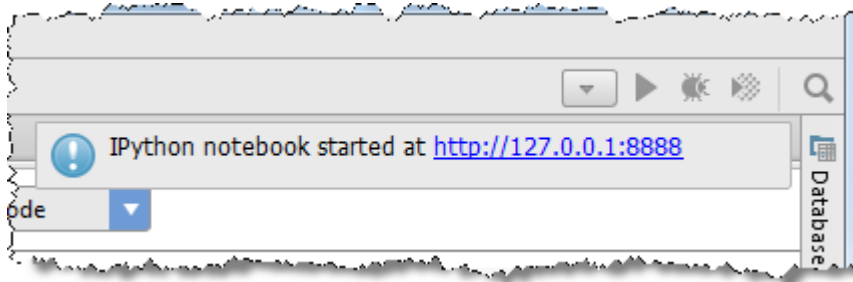
单击第一个元胞，输入代码，这里键入 matplotlib 库的配置代码：

```
%matplotlib inline
```

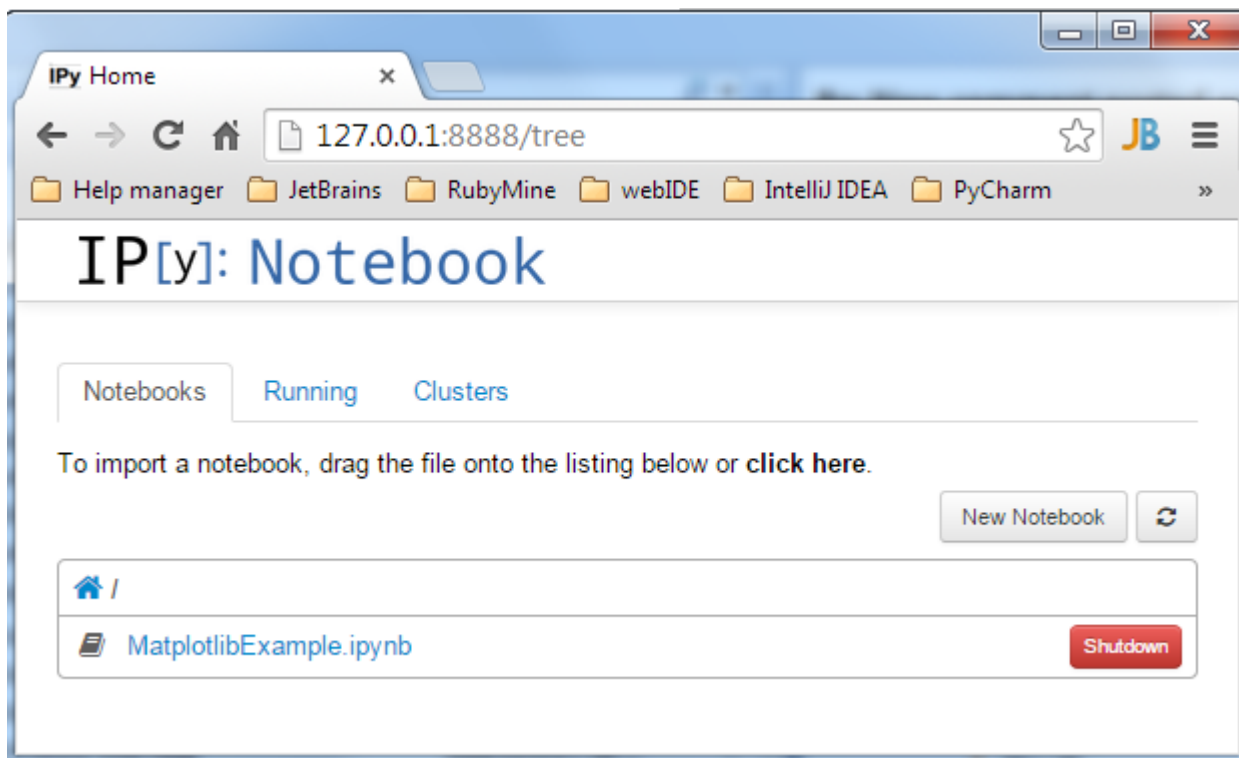
单击工具栏上的 （或者 Shift+Enter）运行，Pycharm 会弹出一个对话框显示 IPython Notebook 服务运行的 URL
地址：



单击 OK:



可以通过浏览器来打开这个链接:



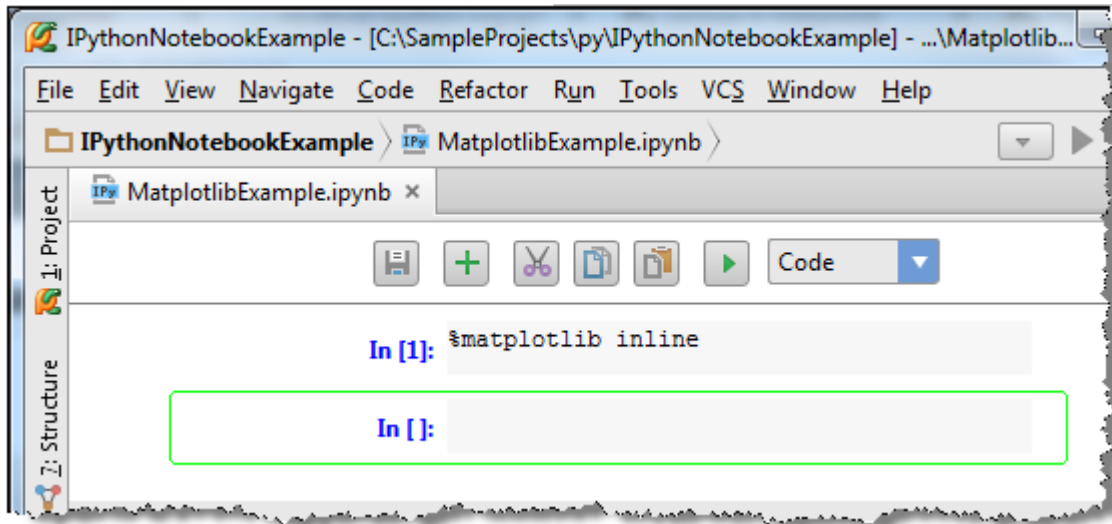
在设置对话框中的 IPython Notebook 中指定了其默认的 URL。接下来我们开始真正使用 IPython Notebook。

5、使用 cells

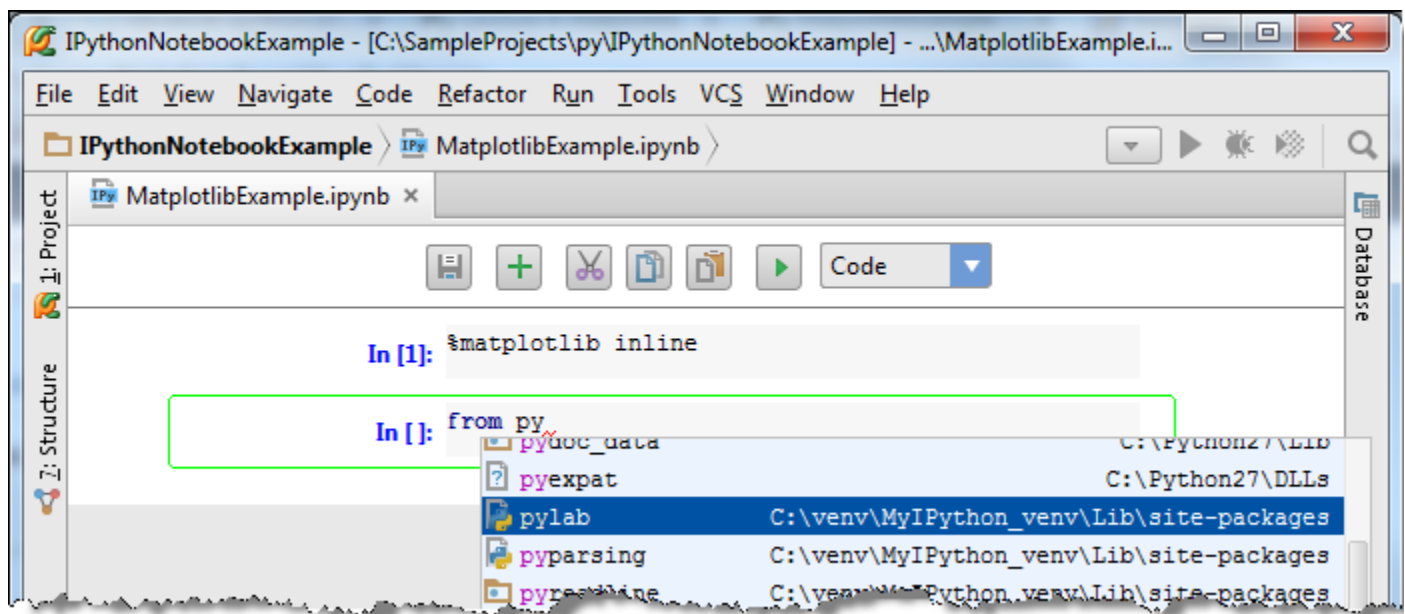
首先，写上 import 语句:

```
from pylab import *
```

当你运行第一个 cell 时，Pycharm 会默认创建下一个空的 cell:



输入代码（体会拼写助手功能）：

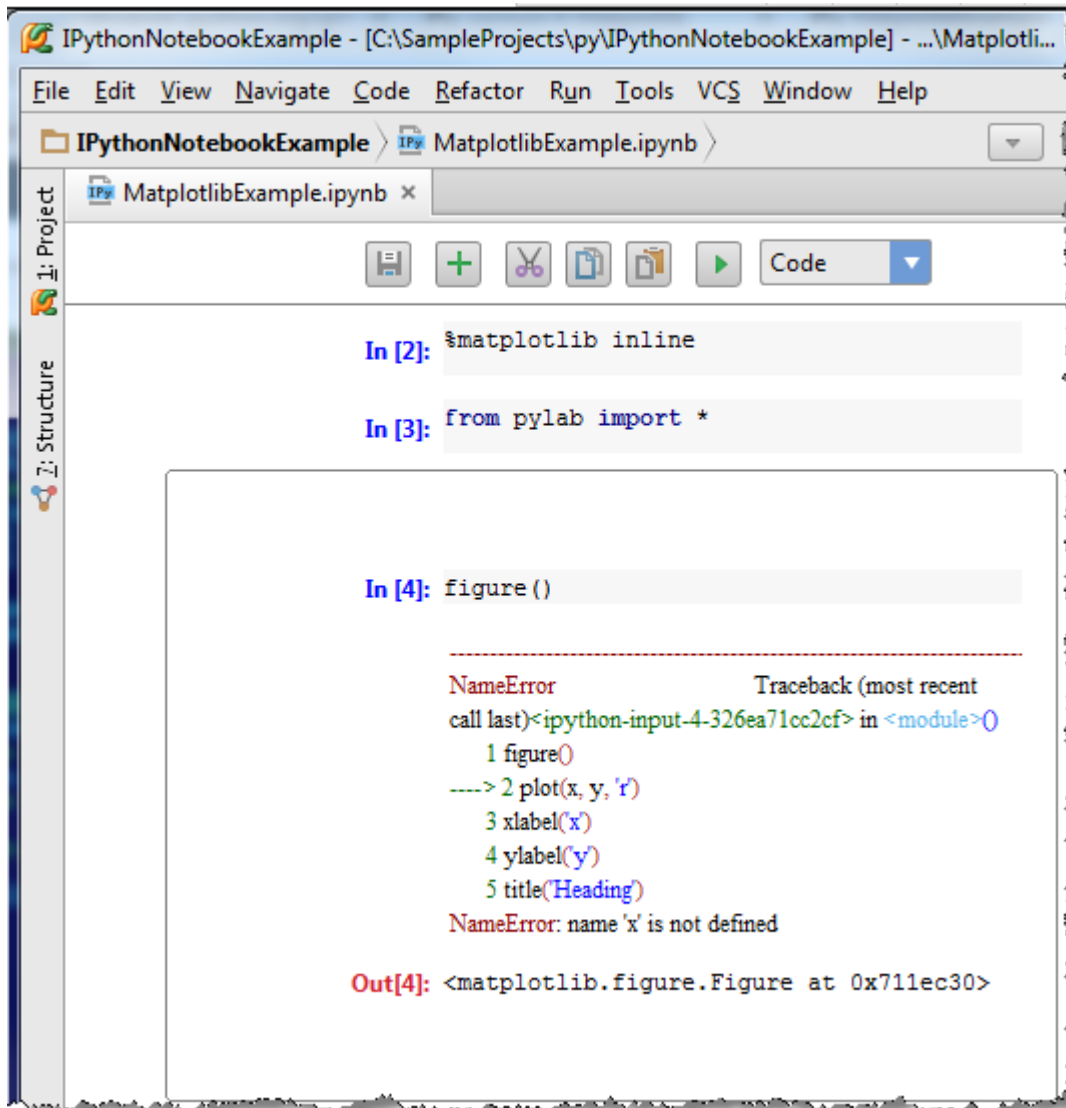


单击  再次运行，无输出，但有创建了一个新的 cell。

在新的 cell 里面输入如下代码；

```
figure()
plot(x, y, 'r')
xlabel('x')
ylabel('y')
title('title')
show()
```

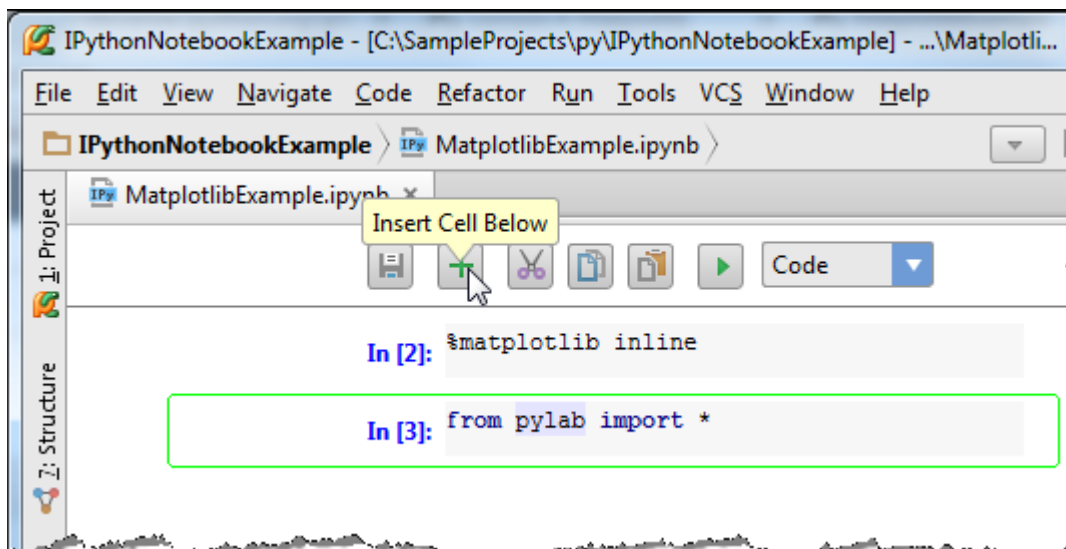
运行这个 cell，报错：



变量需要提前定义，因此我们再添加一个新的 cell。

6、添加 cell

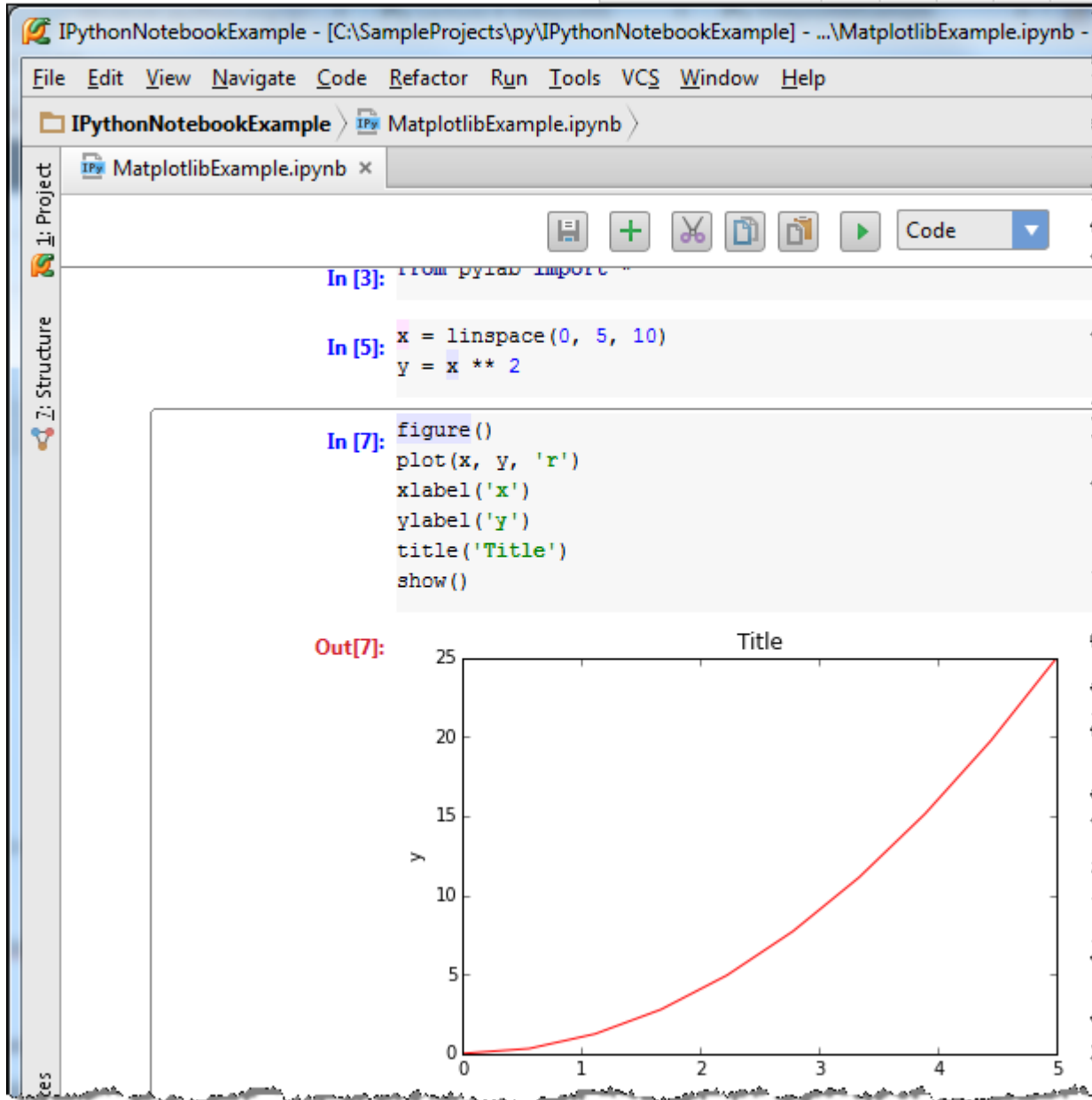
单击 import 语句所在 cell（变绿框），单击绿色的加号（或者是 Alt+Enter 快捷键）：








输入变量定义代码：




```
x = linspace(0, 5, 10)
y = x ** 2
```

先运行这个 cell，在运行下一个 cell，输出正常：




7、cell 的剪贴板操作

在工具栏中，除了  和  按钮，还有  (Ctrl+X)、 (Ctrl+C) 以及  (Ctrl+V) 按钮，如果单击 ，则删除当前 cell，并将其缓存于剪贴板上。

 是粘贴， 是复制，其余按钮功能大家自己体会。

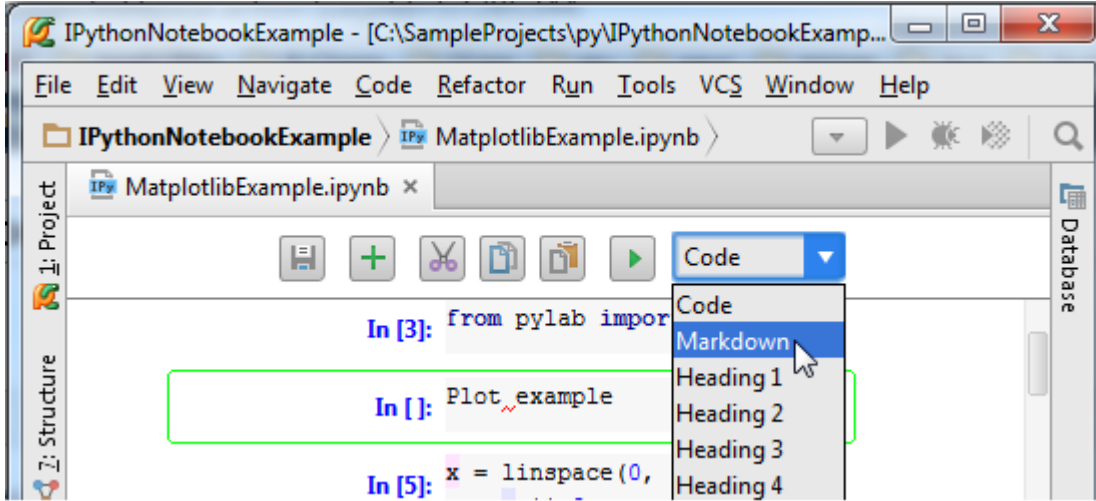
8、选择风格

工具栏右侧下拉菜单用于指定界面风格，这里默认为 code 分割。

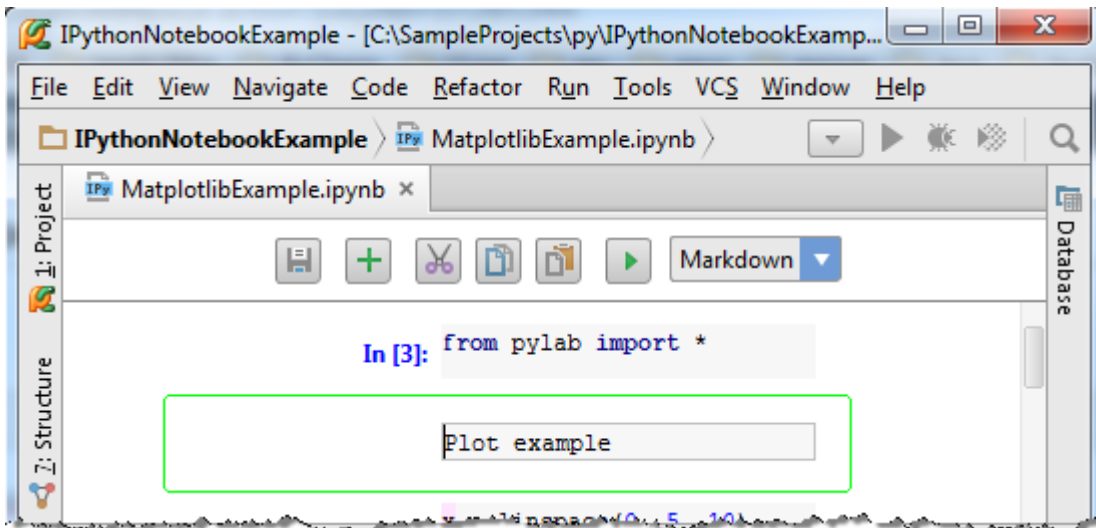
单击 import 语句的 cell，单击 ，默认创建 code 风格的 cell，输入一下代码：

Plot example

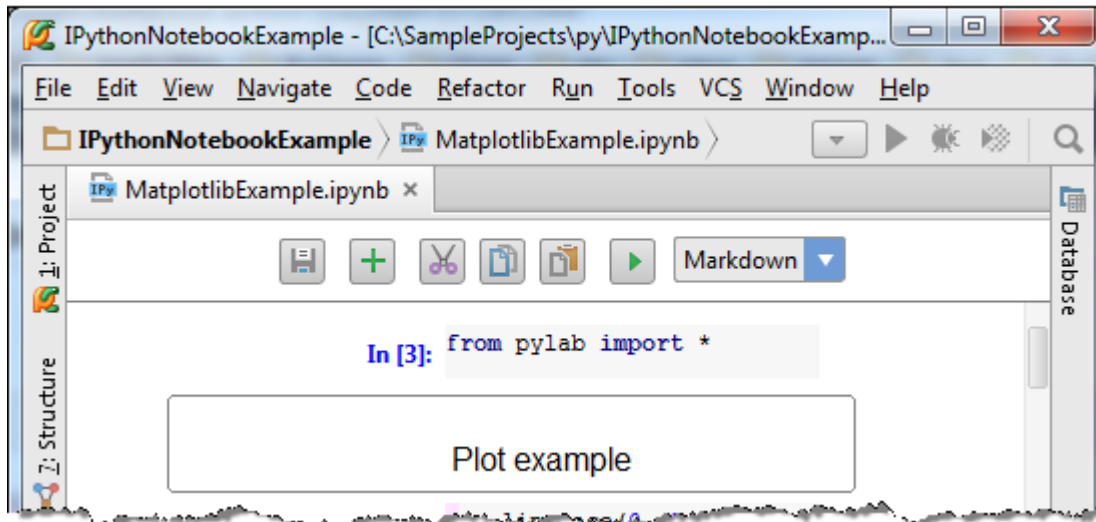
单击下拉箭头选择 Markdown 模式：



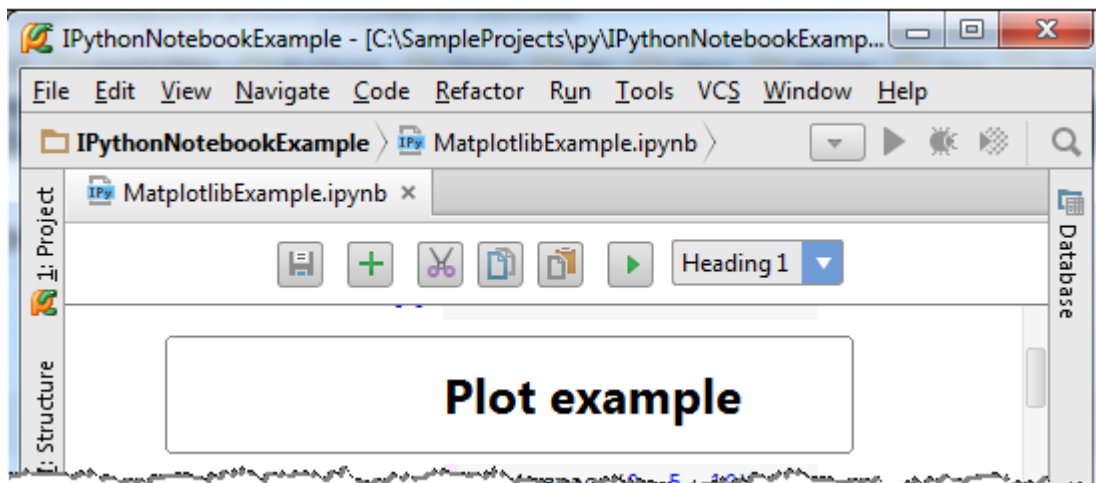
cell 外观发生响应改变：



单击 ：



OK, 接下来可以选择你喜欢的 style 类型:

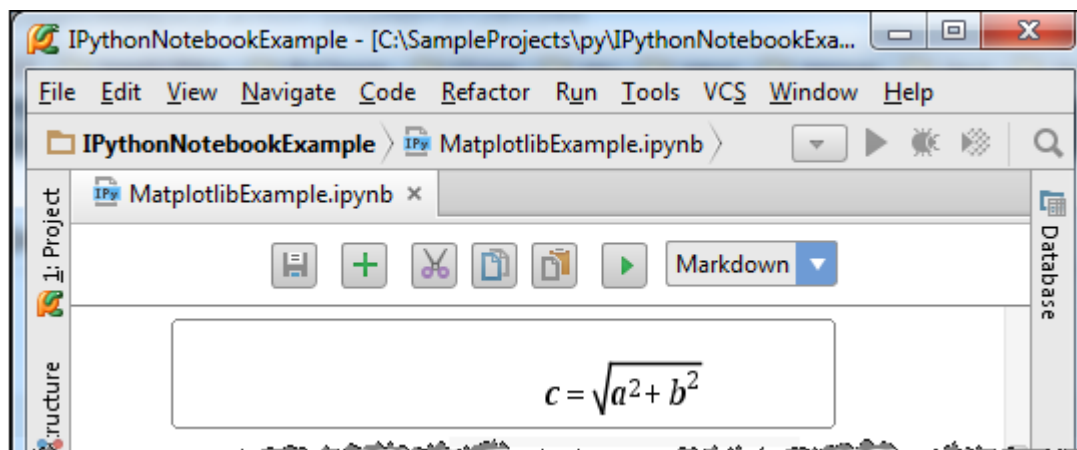


9、输入格式

添加一个新的 cell, 选择 Markdown 格式, 输入以下代码:

```
$$c = \sqrt{a^2 + b^2}$$
```

单击 :



正如你所见，IPython Notebook 可以实现 Latex 格式的公式编辑。

接下来尝试更复杂的功能，显示公式的计算结果。添加一个 cell，输入一下代码（来自 [SymPy: Open Source Symbolic Mathematics](#)）：


```
from __future__ import division
from IPython.display import display

from sympy.interactive import printing
printing.init_printing(use_latex='mathjax')

import sympy as sym
from sympy import *
x, y, z = symbols("x y z")
k, m, n = symbols("k m n", integer=True)
f, g, h = map(Function, 'fgh')
```

运行，无输出，继续添加 cell，输入代码：

```
Rational(3,2)*pi + exp(I*x) / (x**2 + y)
```

单击 ，享受结果吧：

IPythonNotebookExample - [C:\SampleProjects\py\IPythonNotebookExample] - ...\MatplotlibExample.ip

File Edit View Navigate Code Refactor Run Tools VCS Window Help

IPythonNotebookExample > MatplotlibExample.ipynb >

MatplotlibExample.ipynb x

Code

```
In [31]: from __future__ import division
from IPython.display import display

from sympy.interactive import printing
printing.init_printing(use_latex='mathjax')

import sympy as sym
from sympy import *
x, y, z = symbols("x y z")
k, m, n = symbols("k m n", integer=True)
f, g, h = map(Function, 'fgh')
```

```
In [34]: Rational(3,2)*pi + exp(I*x) / (x**2 + y)
```

$$\frac{3\pi}{2} + \frac{e^{ix}}{x^2 + y}$$

最全 Pycharm 教程（34）——Pycharm 内置终端以及远程 SSH 工具的使用

1、主题

如何使用 Pycharm 内置终端以及远程 SSH 工具。

2、准备工作

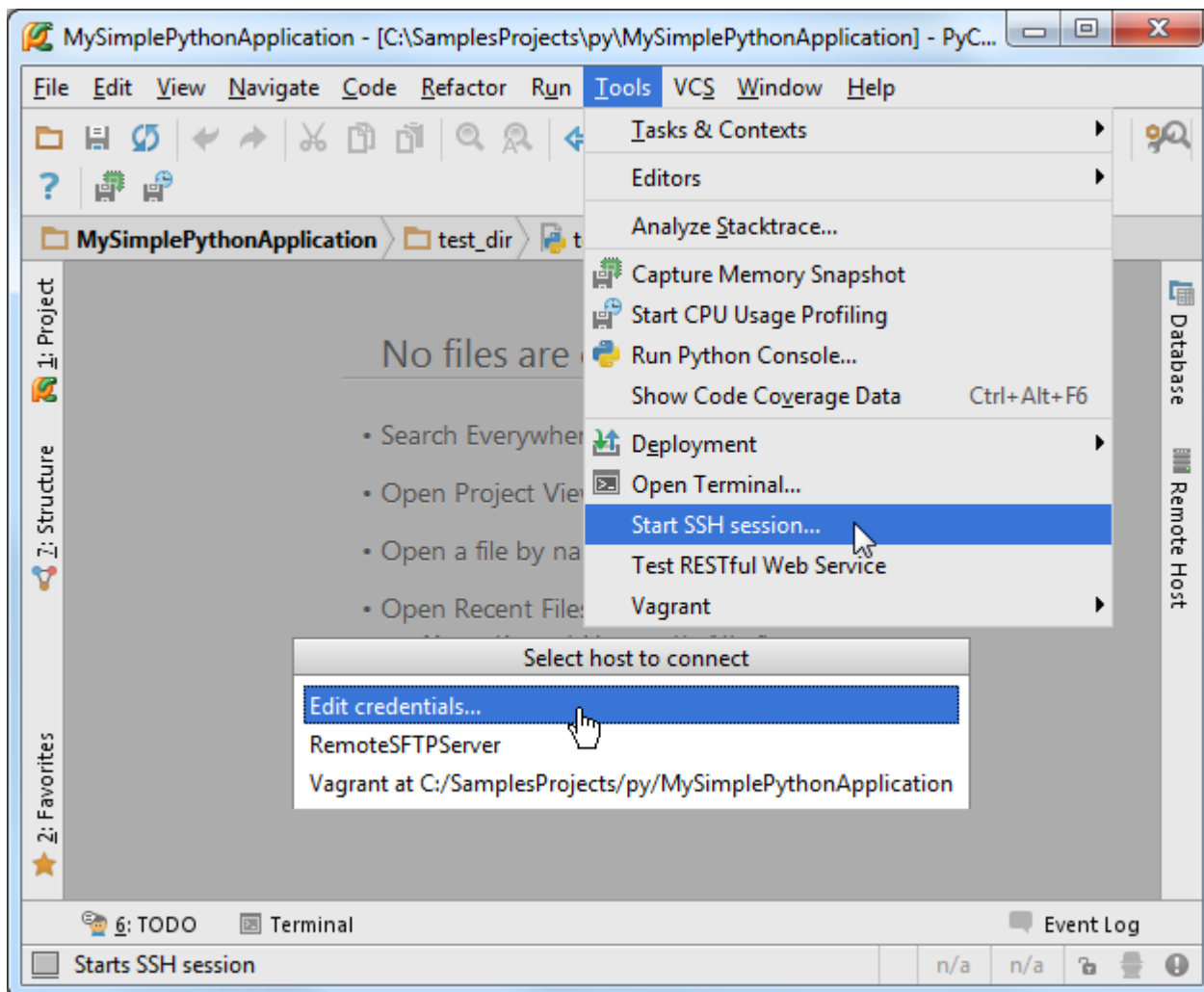
Pycharm 版本为 3.0 或更高

连接 SSH 服务器

3、使用 SSH 客户端

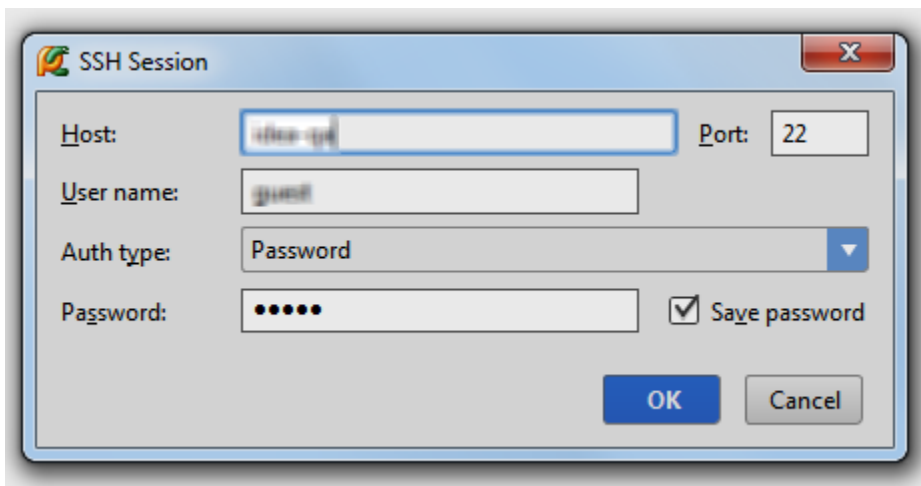
4、开启连接

选择 Tools | Start SSH Session...的主菜单命令，单击 Edit credentials:



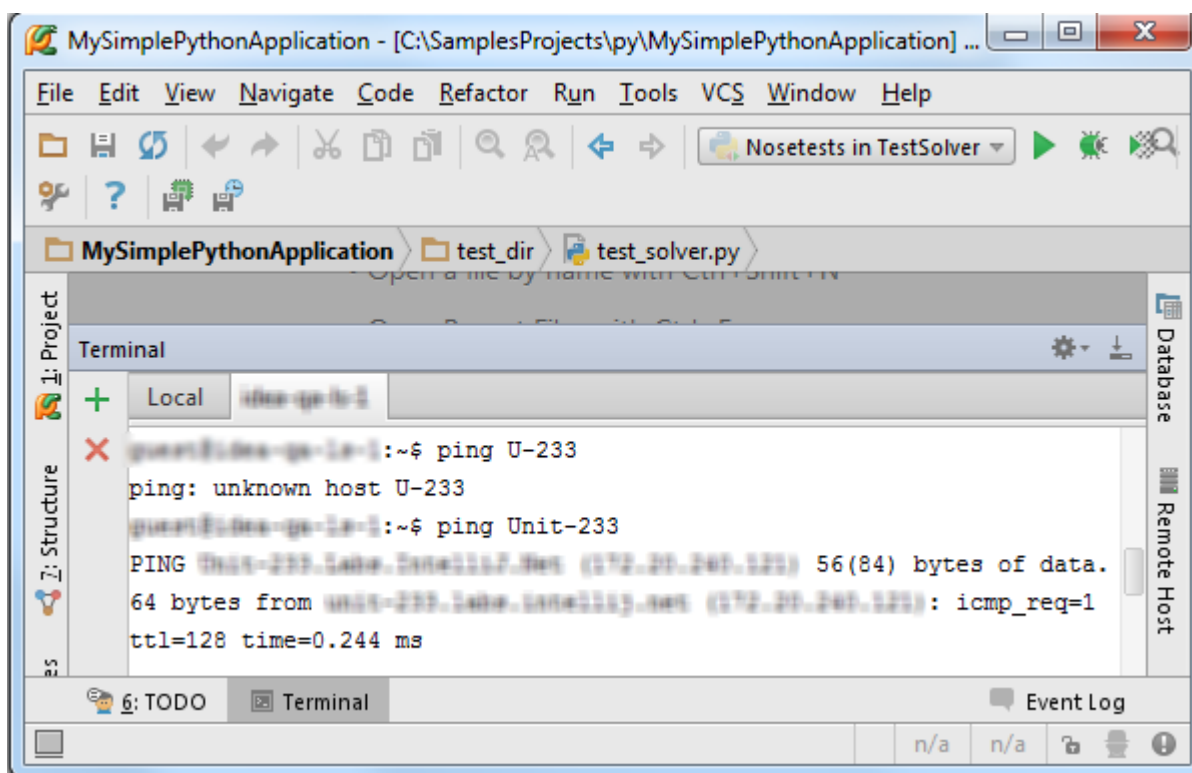
5、提供连接信息

在 Session 对话框中输入建立连接所需相关信息：



6、建立连接

单击 OK，连接开始：



7、SSH 会话功能

运行命令、粘贴复制、浏览历史命令……

8、远程 SSH 外部工具的作用

定义一个外部工具用来脱机运行命令，如显示一个日历。

9、配置一个 SSH 外部工具

打开设置对话框，在 IDE Settings 节点下，选择 [Remote SSH External Tool](#) 页面，单击绿色加号创建一个新的远程工具，在 [Create Tool dialog](#) 窗口中进行一些必要设置：

(1) Name

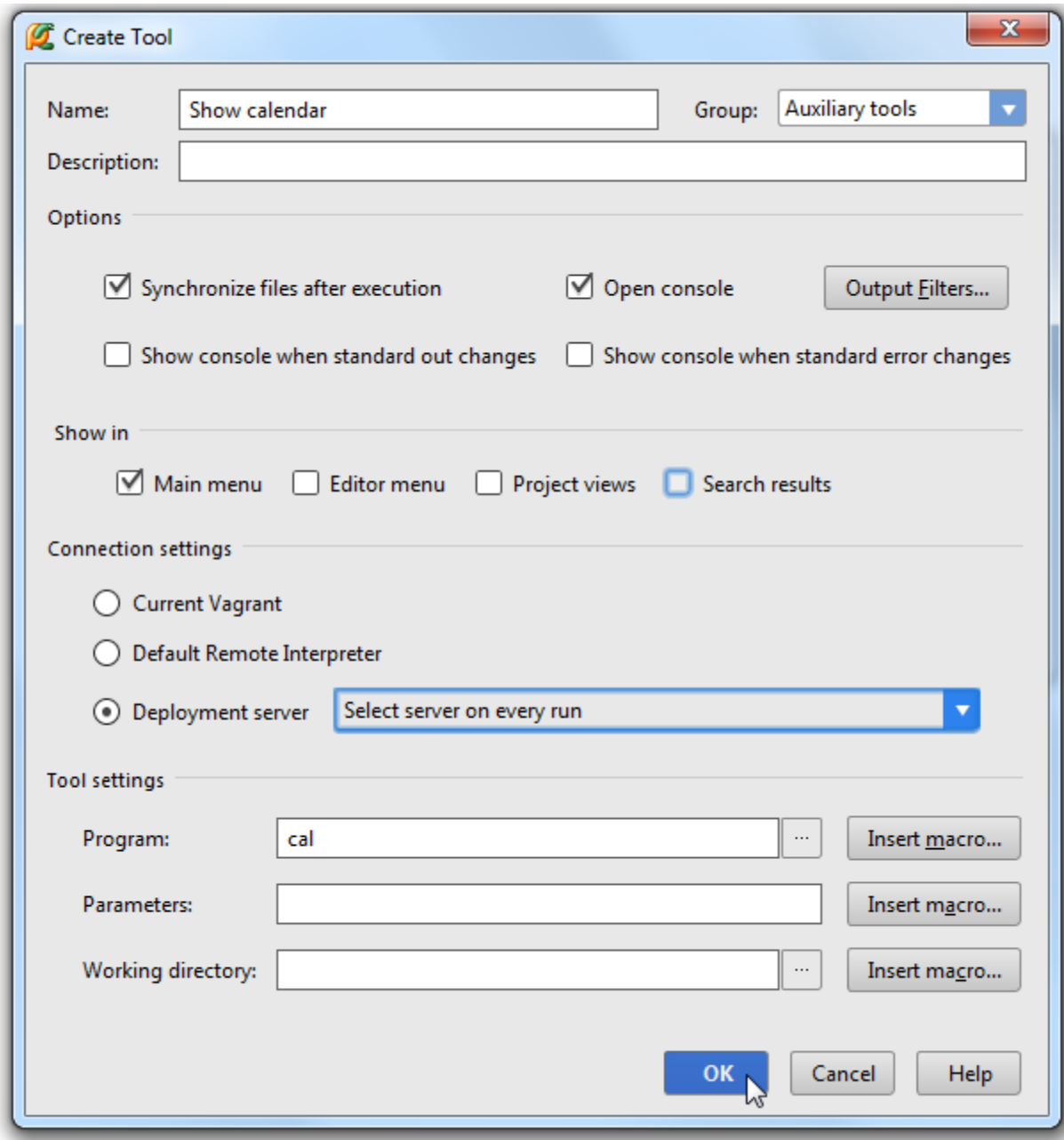
Description 栏提供多种类型的描述。

Group 指定创建级联菜单，用以存放外部工具名

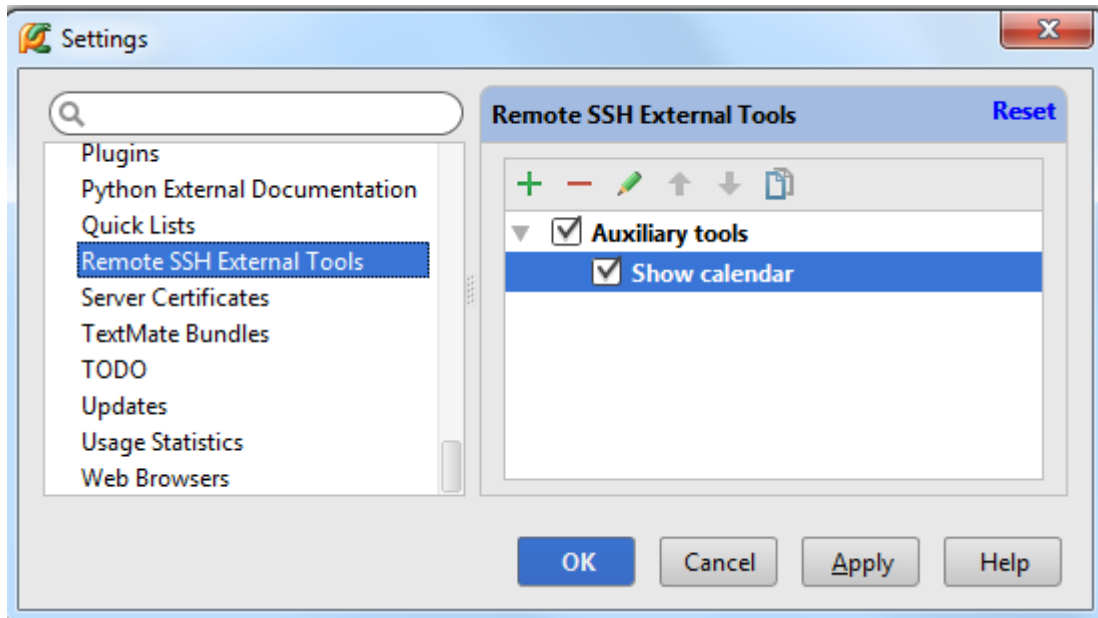
(2) Show in 指定新工具的显示位置，这里只勾选 Main Menu

(3) Connection settings 区域选择 Select server on every run 的单选按钮，即每次运行服务都需要进行详细设置。

(4) 在 Tool settings 区域，指定待远程执行的工具，参数和工作目录是可选的，也可以用宏来代替当前命令：



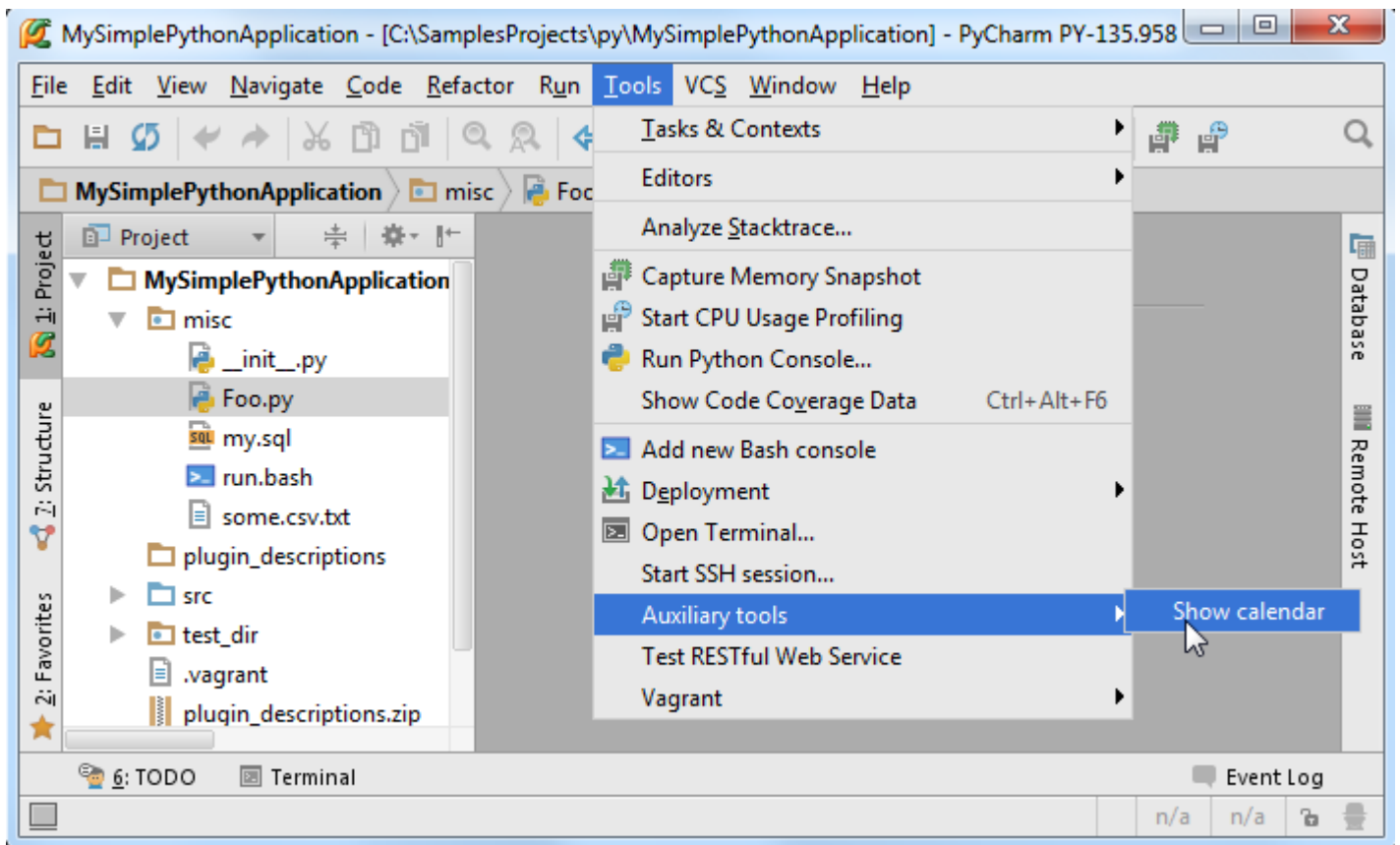
单击 OK，关闭 [Create Tool dialog](#) 对话框，返回 [Remote SSH External Tool](#) 页，在下拉列表中可以看到新创建的工具：



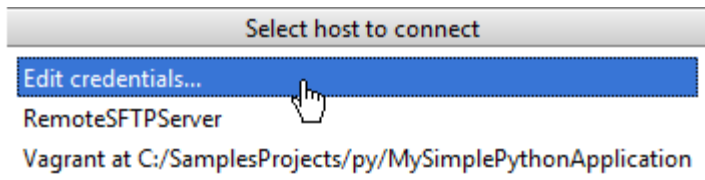
其他控件信息详见 [product documentation](#)。

10、加载 SSH 外部工具

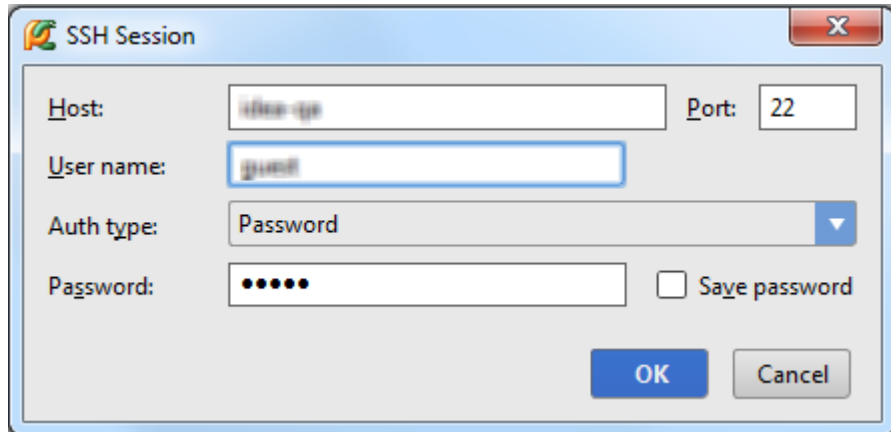
设置好的工具会在 menus 中显示以供选择，这里显示在 Tools 菜单下：



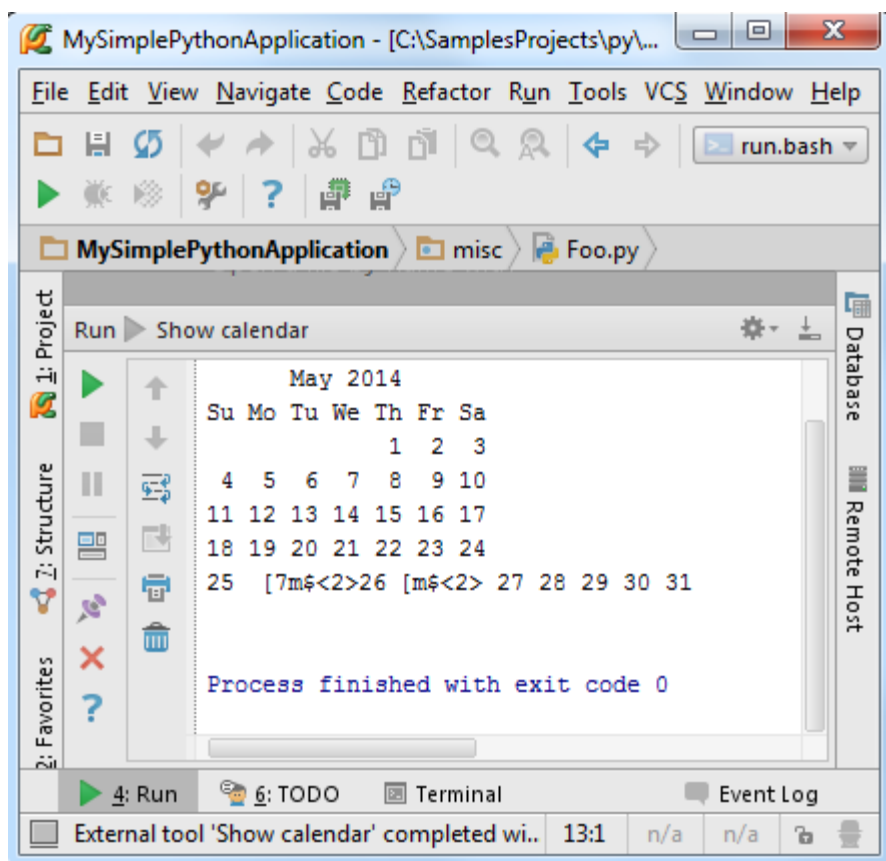
接下来尝试使用这个命令。首先选择要连接的服务器：



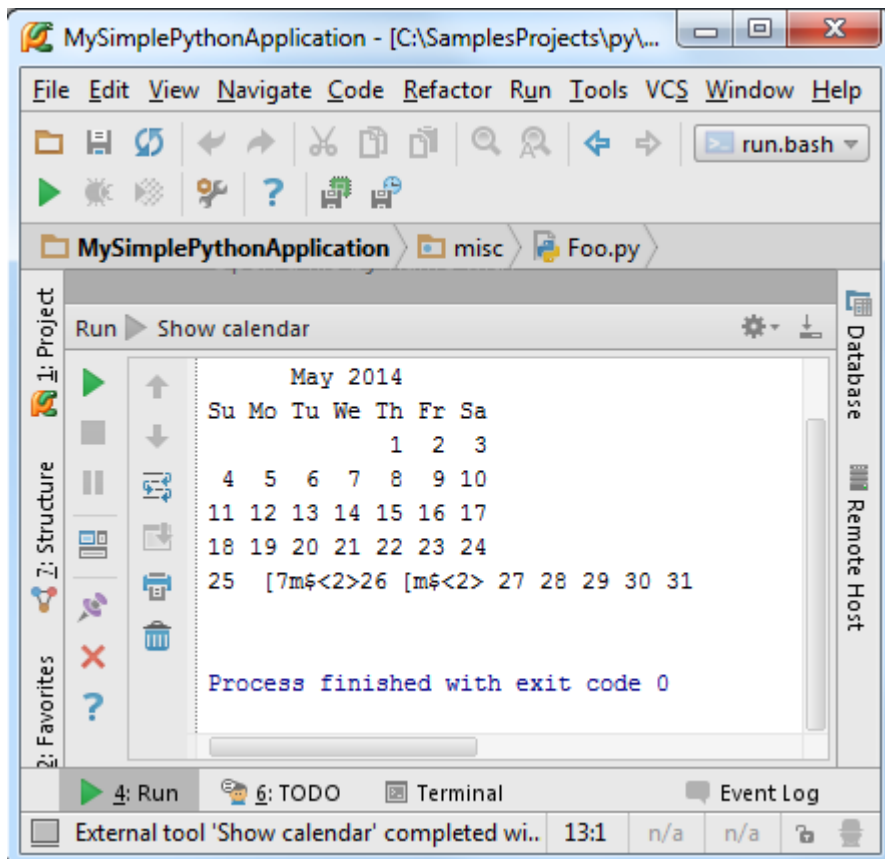
进行相关设置：



最后，顺利显示日历：



SSH 会话运行在终端工具窗口，外部工具在 [Run tool window](#) 窗口中进行加载：



最全 Pycharm 教程（35）——Pycharm 中使用 Vagrant

1、主题

介绍如何在 Pycharm 中使用 Vagrant

2、准备工作

确认电脑上安装了 Vagrant 和 Oracle VirtualBox。

3、下载安装 Vagrant

下载地址：[from the official website](#)

4、下载安装 Oracle VirtualBox

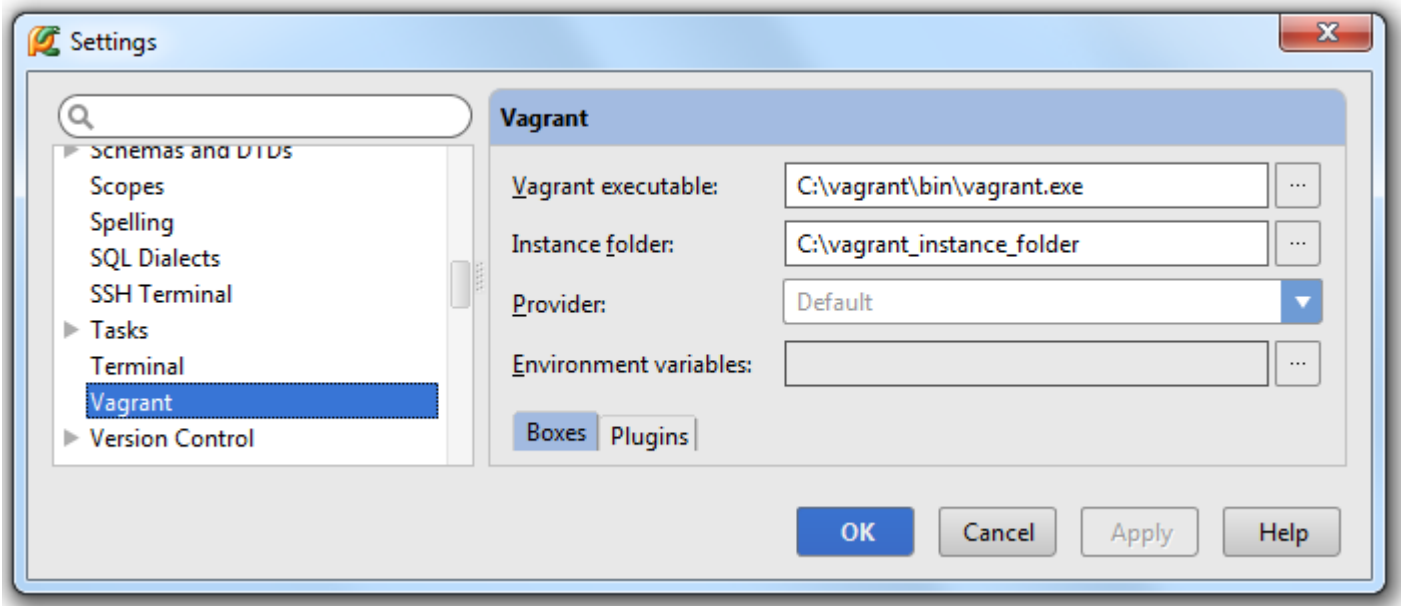
下载地址：[download and install](#)

5、设置环境变量

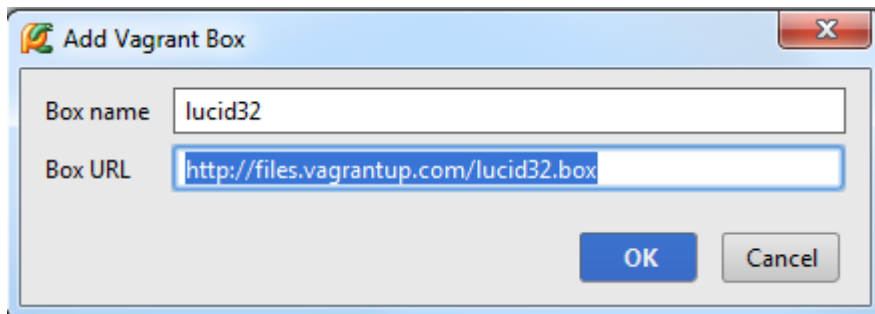
将 vagrant 的可执行文件和 VBoxManage.exe 添加到环境变量中。

6、配置 Vagrant 工程

在工程设置窗口，打开 [Vagrant](#) 页，指定相关参数。其中必须指定 Vagrant 可执行文件路径以及工程实例所在目录：

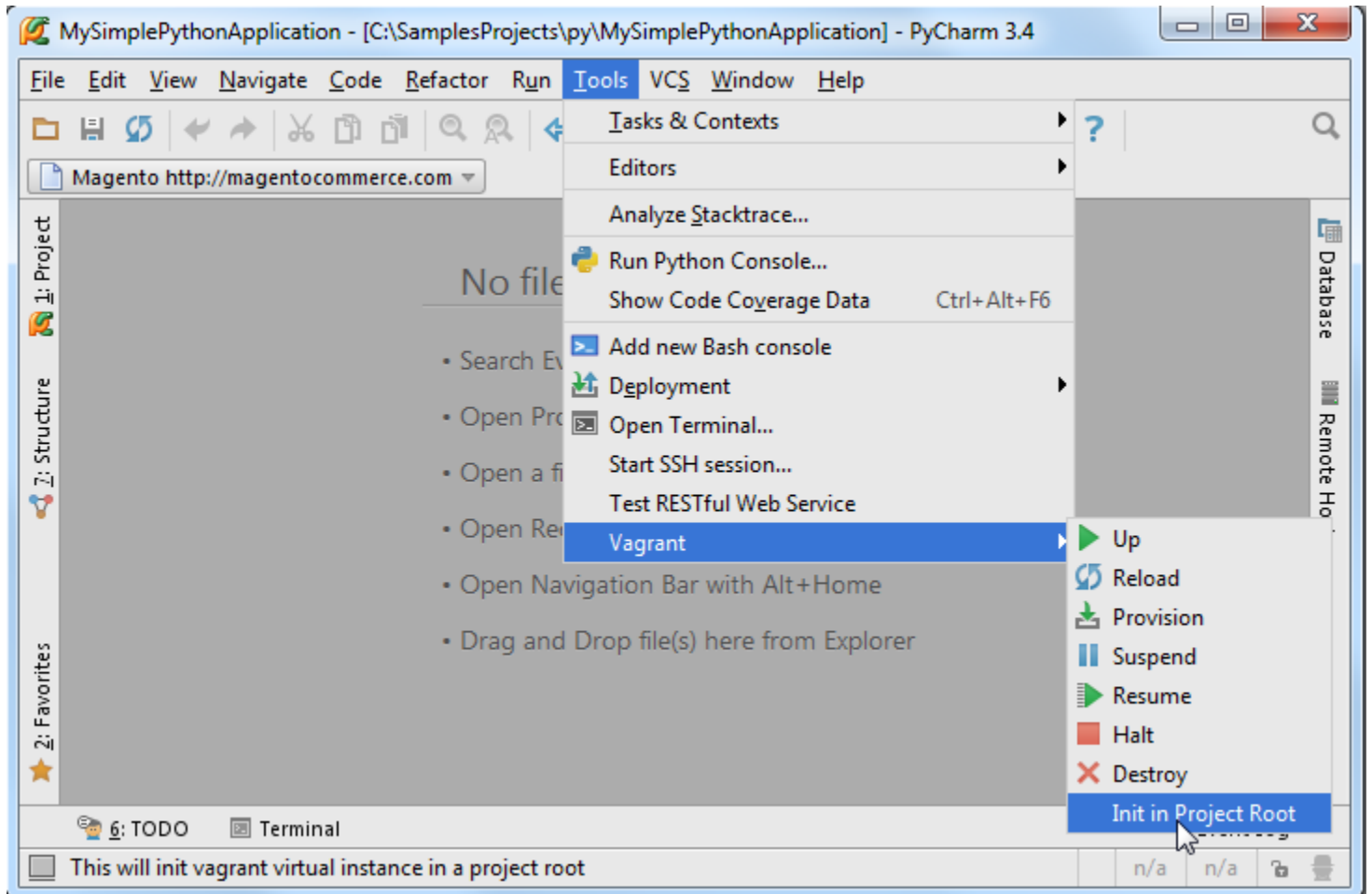


单击绿色加号，指定名称以及 URL：

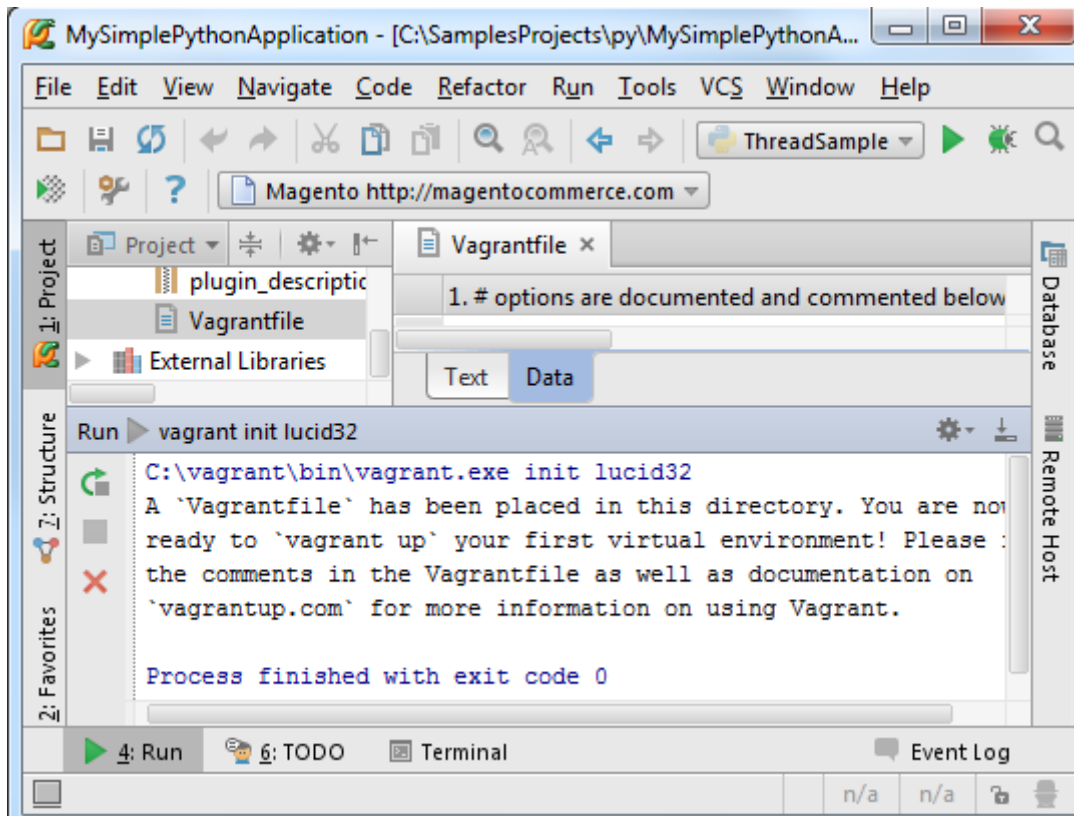


7、初始化 Vagrant 文件

VagrantFile 包含了虚拟机的所有信息。VagrantFile 文件可以手动创建，也可以通过 Tools→Vagrant→Init in Project Root 的主菜单命令自动创建：

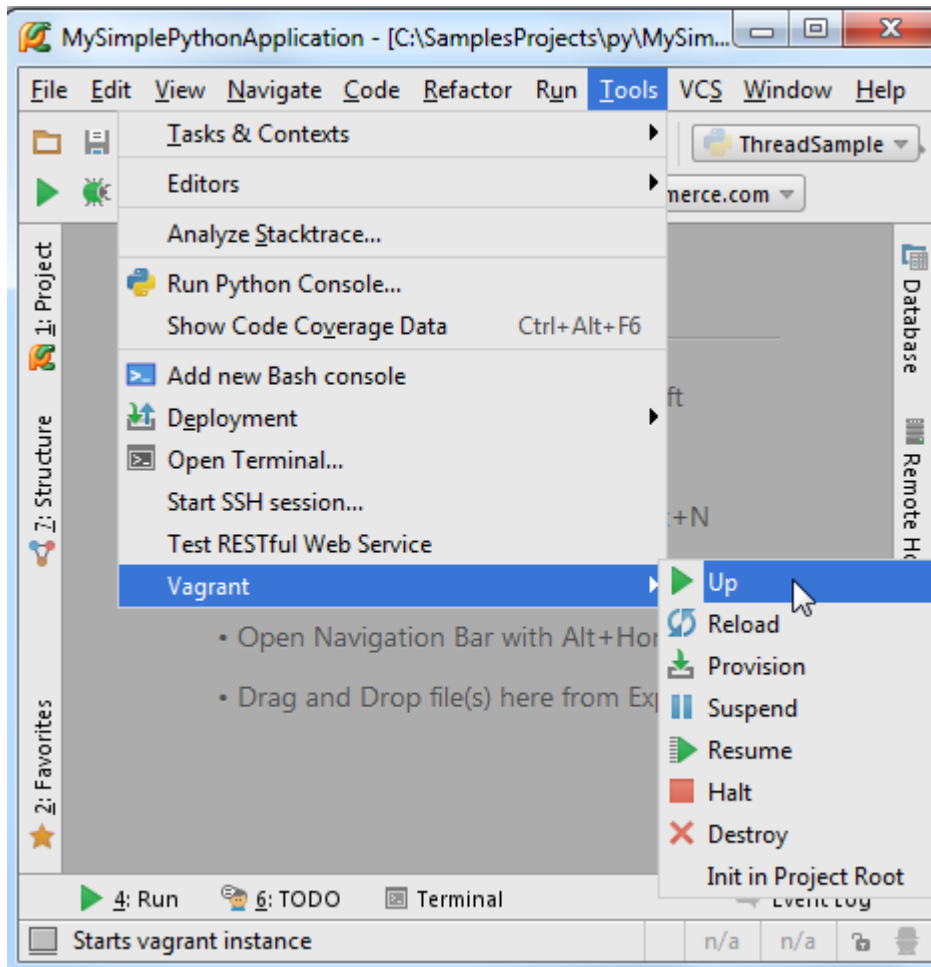


这样就创建了一个默认的 VagrantFile：

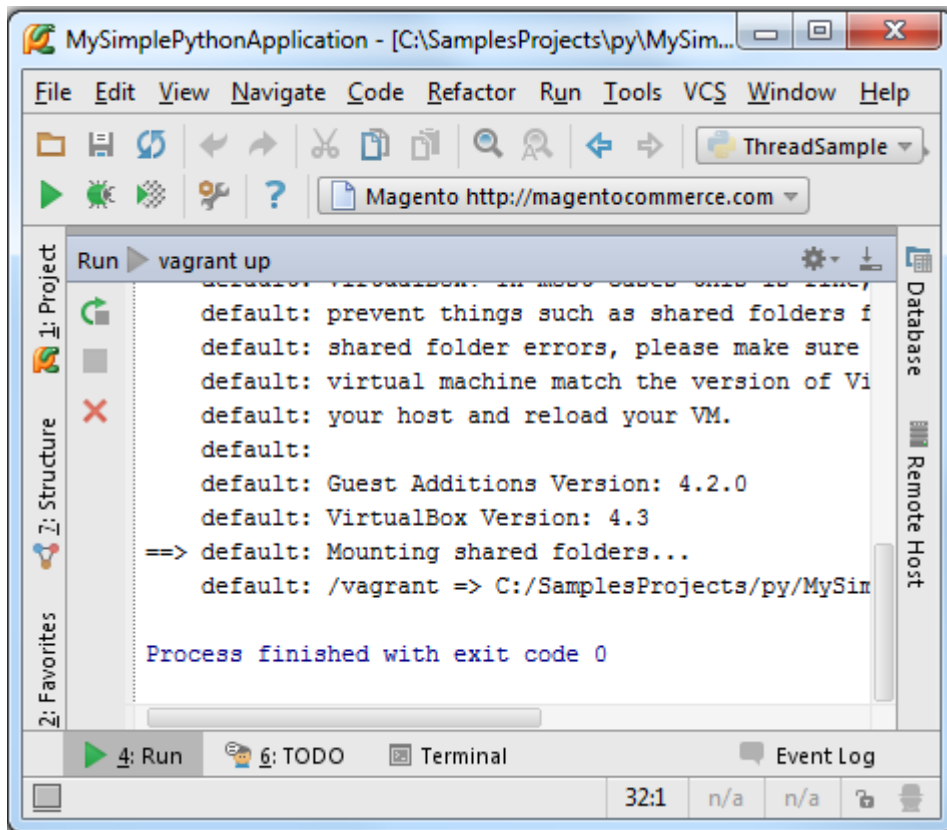


8、开启 Vagrant

选择 Tools→Vagrant→Up 主菜单命令启动 Vagrant，首次启动会自动配置虚拟机的 VirtualBox 并进行引导：



在 [Run tool window](#) 显示操作进程：



配置一次即可。

最全 Pycharm 教程（36）——Pycharm 中 Vagrant 高级技巧

1、主题

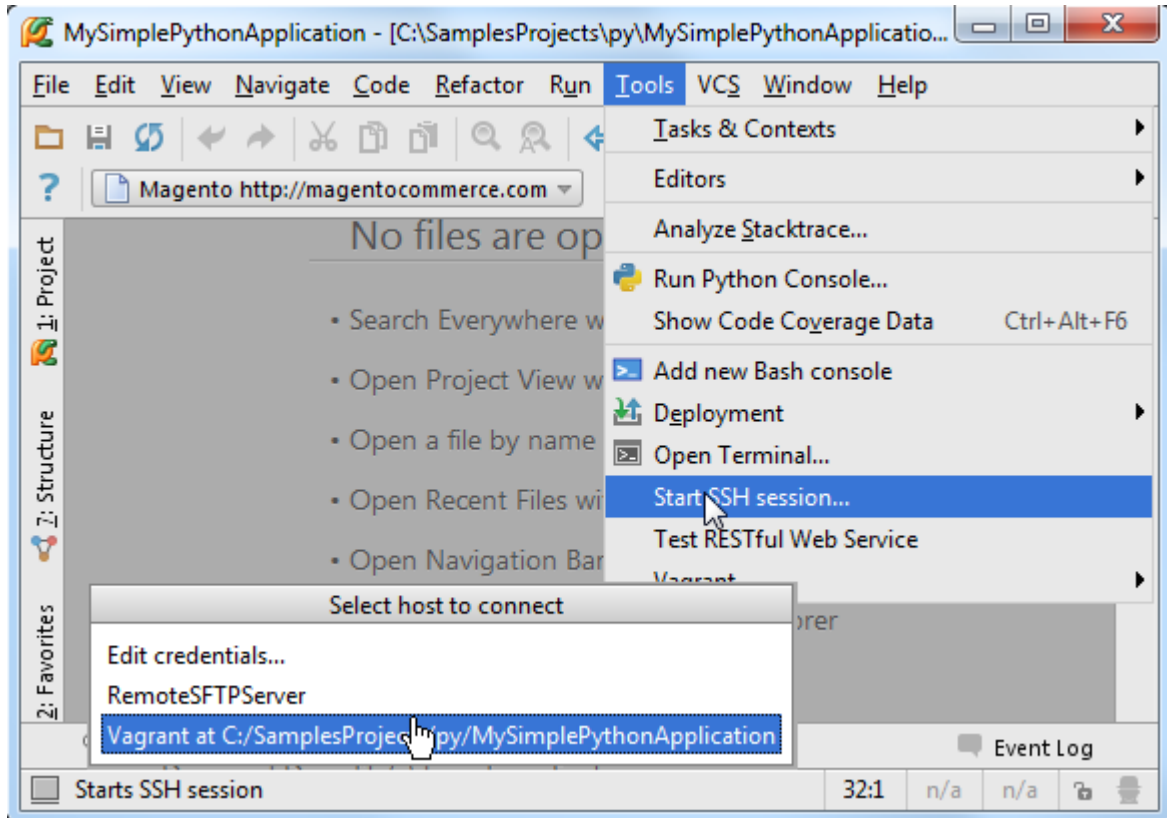
介绍 Pycharm 中 Vagrant 高级使用技巧。

2、使用内置 SSH 连接一个 Vagrant

[built-in SSH terminal](#)

3、开始连接

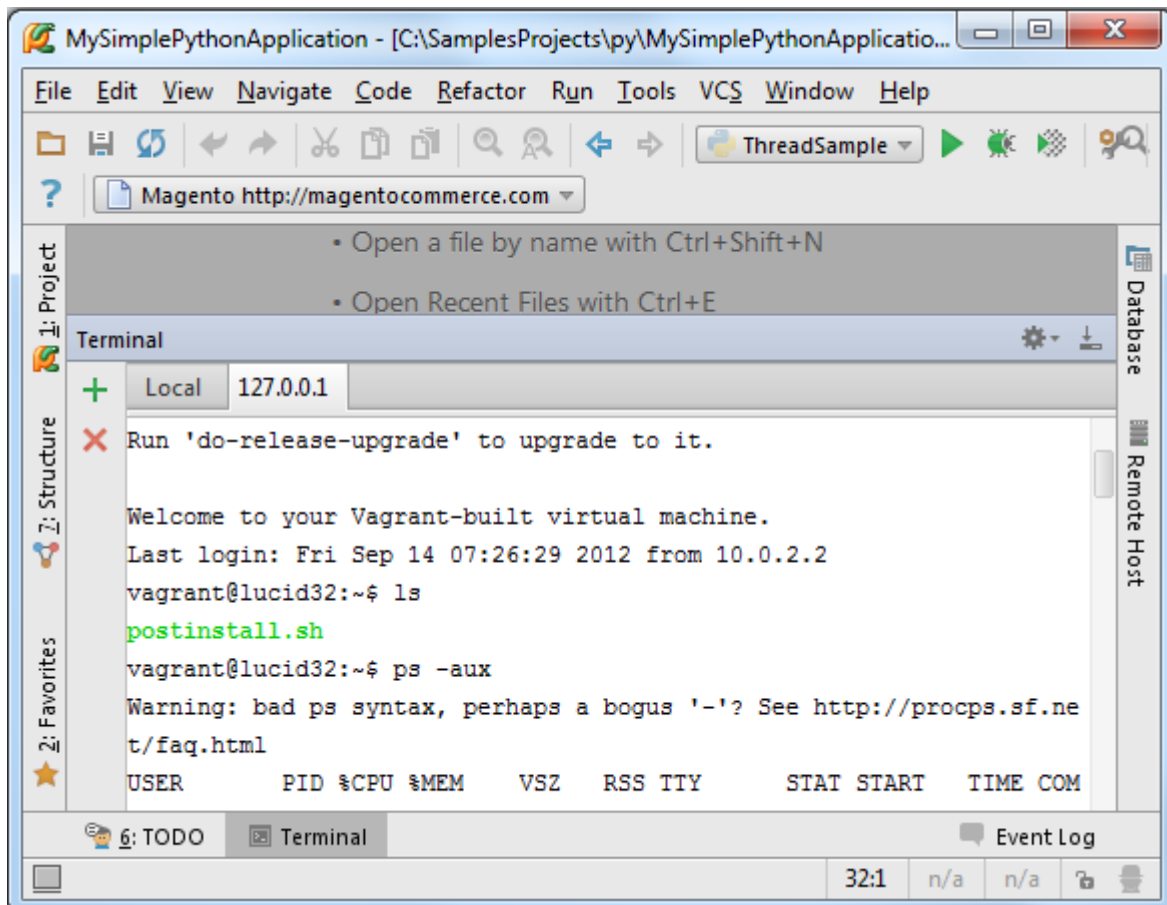
使用 Tools→Start SSH session...主菜单命令：



The configured Vagrant machine 已经被自动添加到了列表中，单击连接。这里的 Edit credentials...菜单项是用于用户手动输入连接信息，我们这里只使用 virtual box。

4、使用 SSH

选定 Vagrant 之后，Pycharm 使用 SSH 连接到 Vagrant，并在本地终端显示相关信息：



在 SSH 终端可以进行以下操作：

使用上下键浏览历史命令

使用剪贴板功能

5、使用共享文件夹

Pycharm 允许在 host 和 Vagrant 共享文件夹。

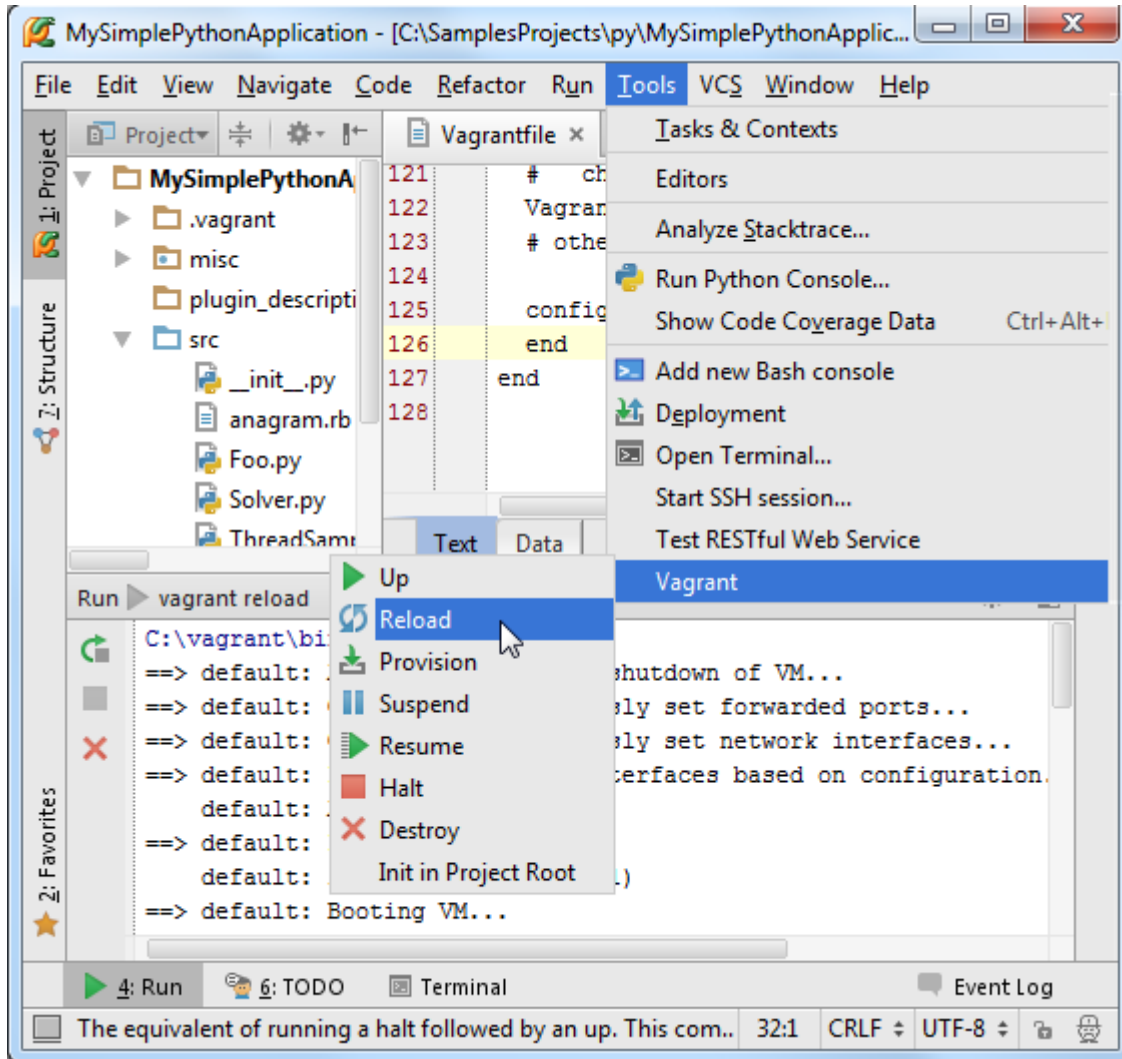
6、添加映射路径

打开 Vagrantfile 文件 [Open Vagrantfile for editing](#)，添加一个映射路径配置参数：

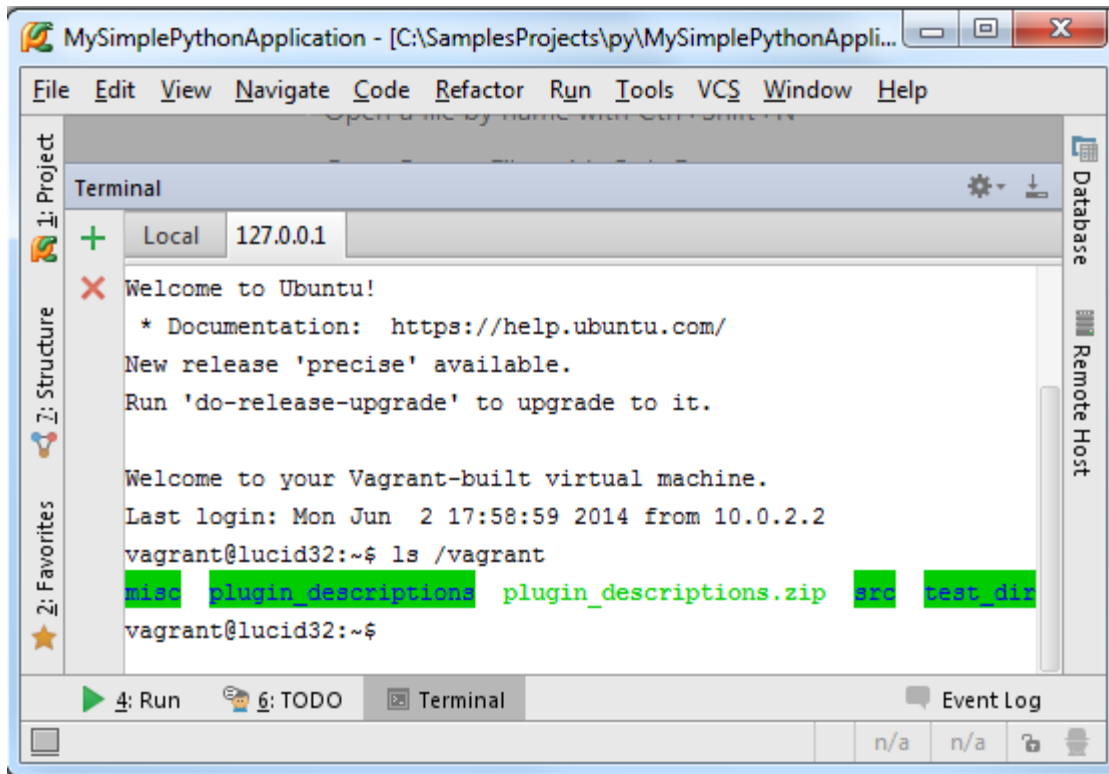
```
Vagrant.configure("2") do |config|
  config.vm.synced_folder "src/", "/srv/website"
end
```

7、重载 Vagrant

可以使用 Tools→Vagrant→Reload 主菜单命令来重新装载 Vagrantfile：

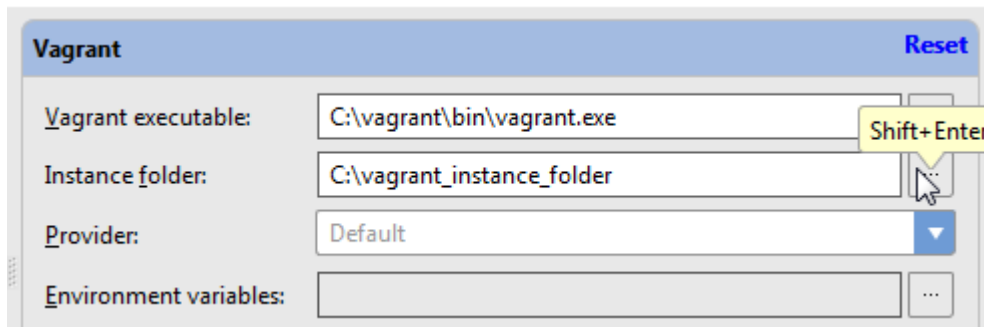


一旦重新装载了 Vagrant 之后，一个新的映射路径变得可用。例如我们使用 SSH 内置终端连接 Vagrant 之后，我们看到 Pycharm 将 `/vagrant` 文件夹中的内容映射到本地工程文件夹中，注意这里如果删除一处文件夹中的文件，两边的文件夹都会删除相应文件。



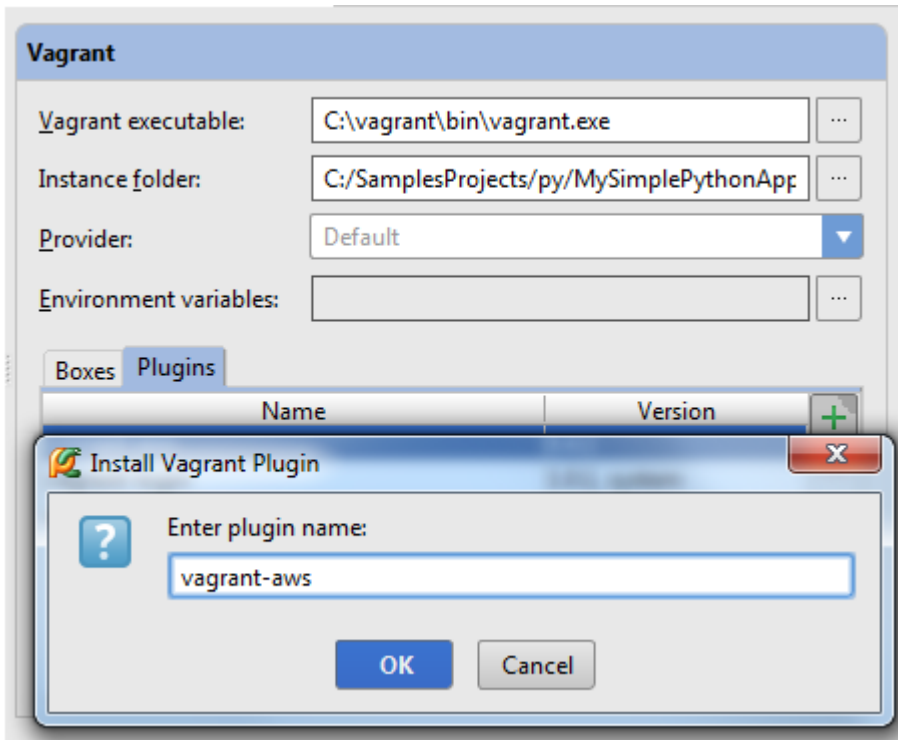
8、指定 Vagrant 实例文件夹

默认情况下 Vagrant 相关文件被放置在工程的根目录下，我们可以在工程设置窗口的 [Vagrant](#) 页面对其进行更改：



9、在设置窗口管理 Vagrant 插件

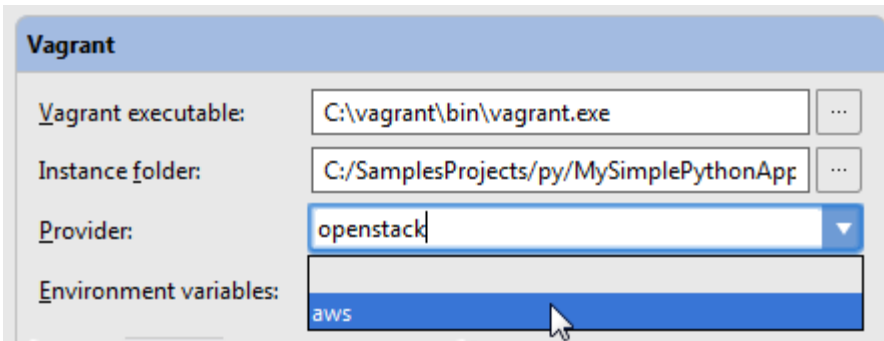
可以在 [Vagrant](#) 设置中管理相关插件，在 [Plugins](#) 页，使用安装/卸载/更新插件 按钮来进行管理。例如通过 [VMWare Fusion Provider](#) 插件可以允许我们在 VMWare 环境中运行 Vagrant。



10、供应商支持

Vagrant 默认与 [Oracle VirtualBox](#) 协同工作，这里我们可以更换供应商平台，例如 [VMWare](#)、[Amazon EC2](#)，具体参见供应商列表 [Vagrant plugins list](#)。

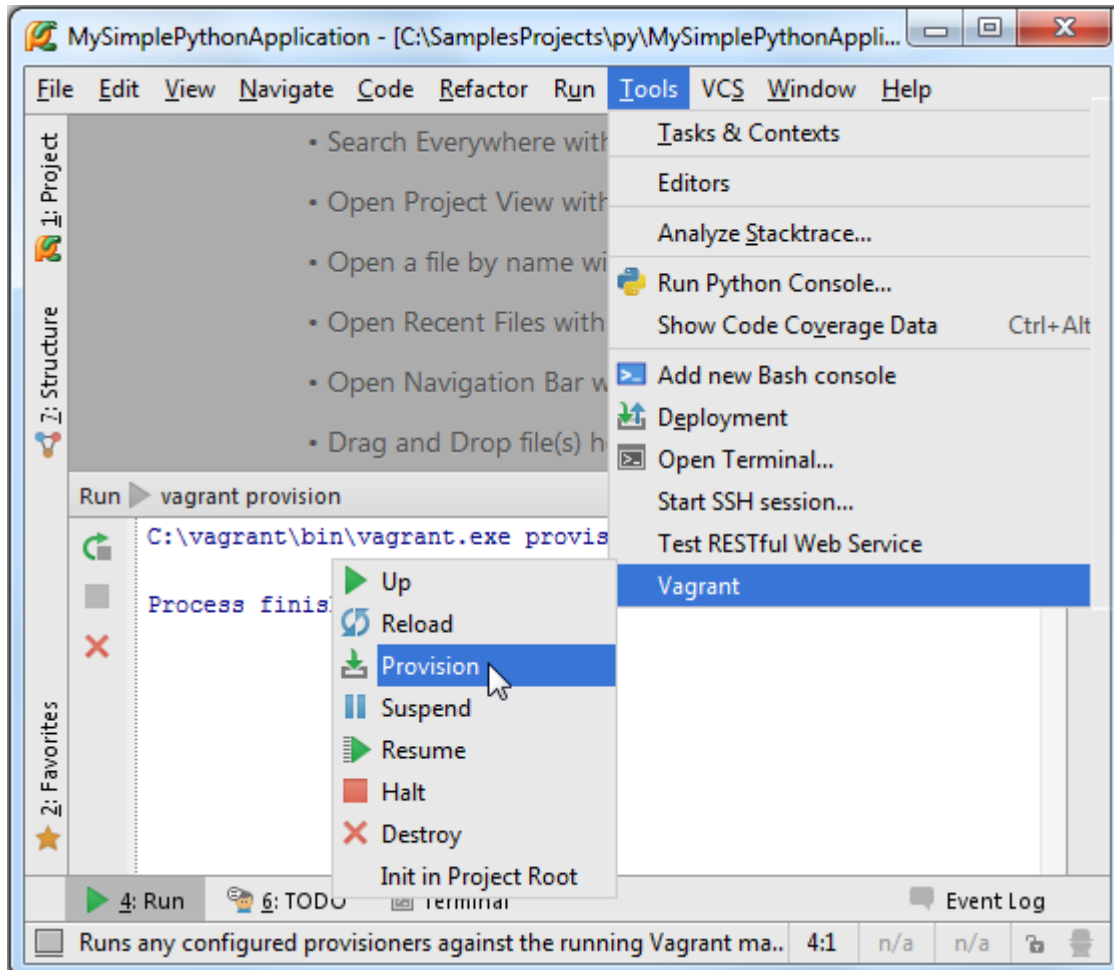
Pycharm 将供应商名称自动绑定到命令行，方便通过其来使用 Vagrant。选中一个电脑上安装并设置好的供应商之后，将会通过它来执行所有的 Vagrant 命令：



11、更换供应商支持

Tools→Vagrant→Provision 用以切换供应商设置，无需关闭虚拟机。

使用 Tools→Vagrant 菜单命令，运行 [provisioning on a running environment](#)：

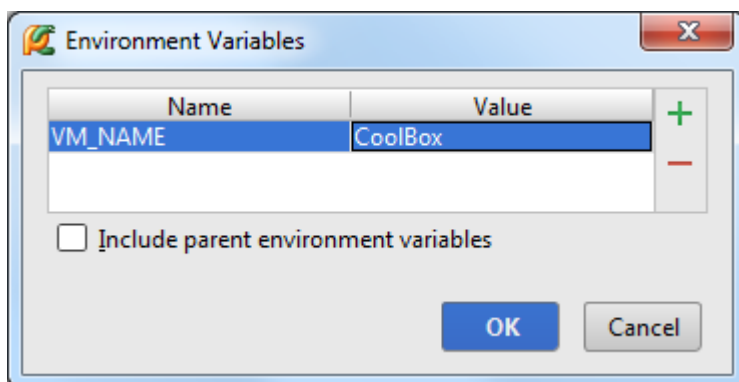


12、环境变量功能

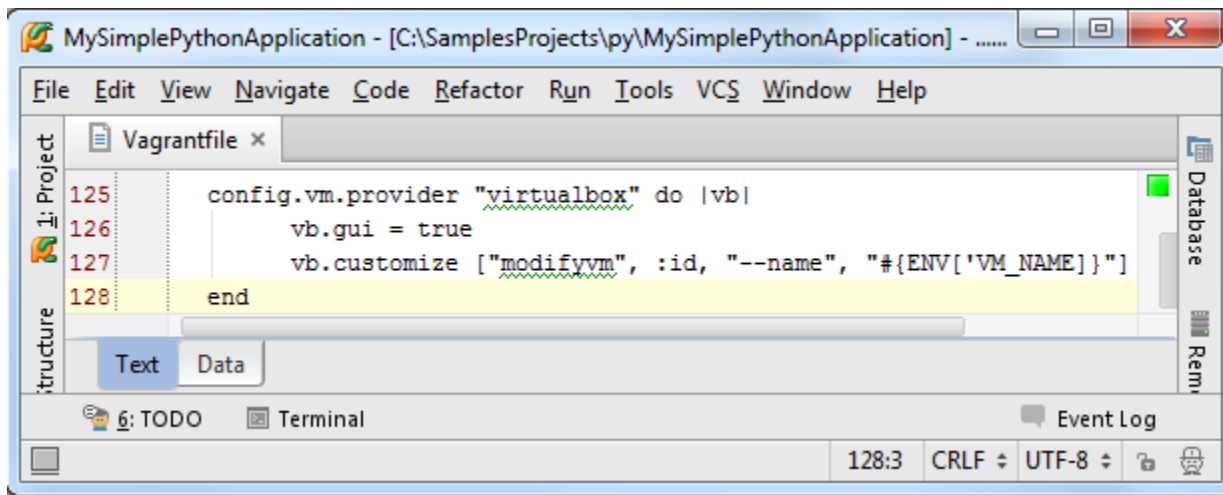
环境变量有以下功能：

- setting the Puppet node
- setting the Puppet environment
- setting custom facts
- setting AWS keys
- ...

在 [Vagrant](#) 页面设置项目相关环境变量：



设置完成后，这些环境变量就会被加入到 Vagrantfile，以 `{ENV['name_of_variable']}` 的形式：



最全 Pycharm 教程（37）——Pycharm 版本控制之基础篇

1、主题

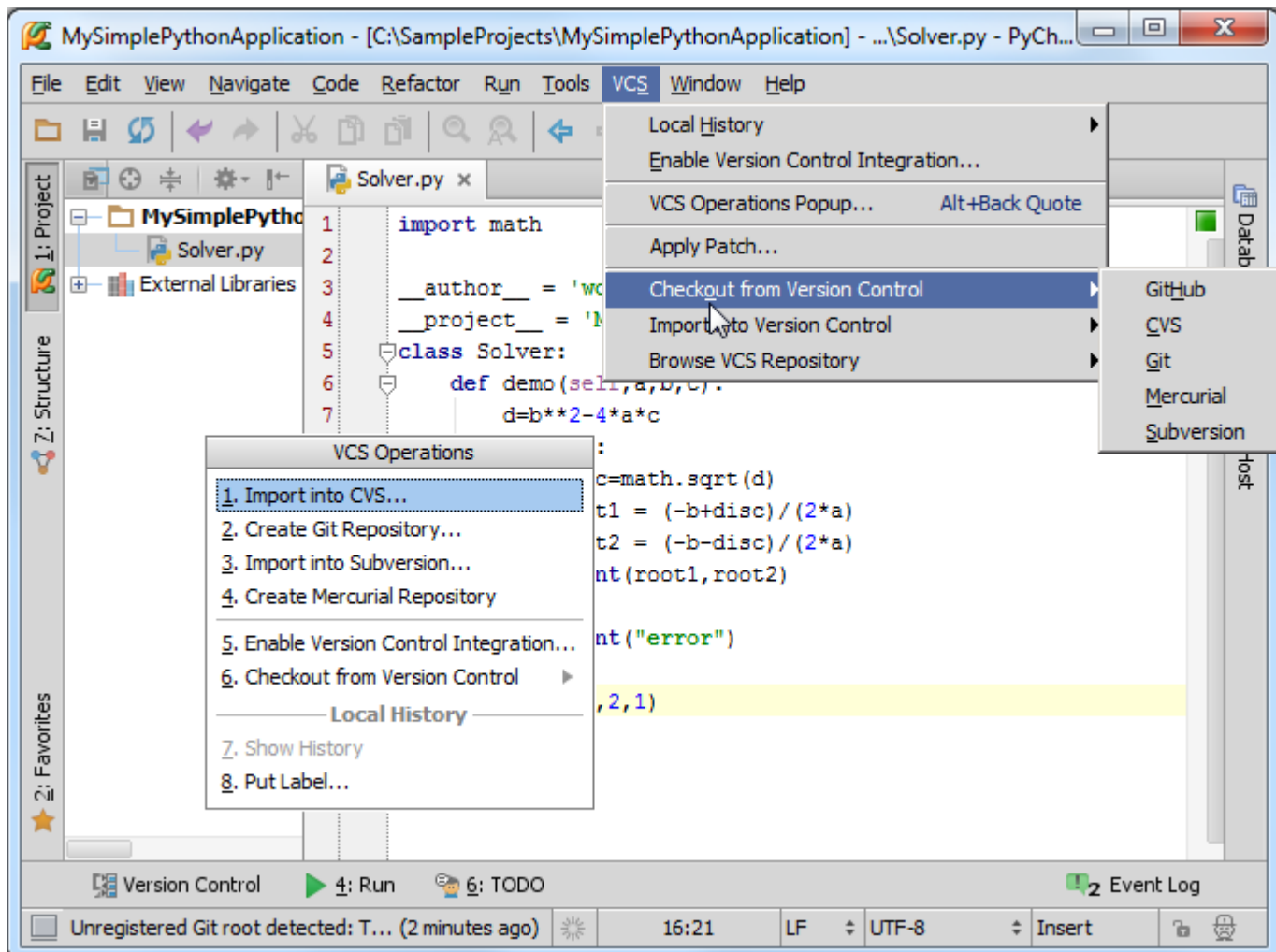
介绍 Pycharm 的版本控制系统

2、准备工作

- (1) Pycharm 版本为 2.7 或者更高
- (2) 已经创建一个工程，参见 [Getting Started tutorial](#)
- (3) 安装并配置了 VCS，有远程接入权限

3、启用 VCS

Pycharm 默认版本控制不可用，不过可以在 VCS 菜单中找到相应命令：

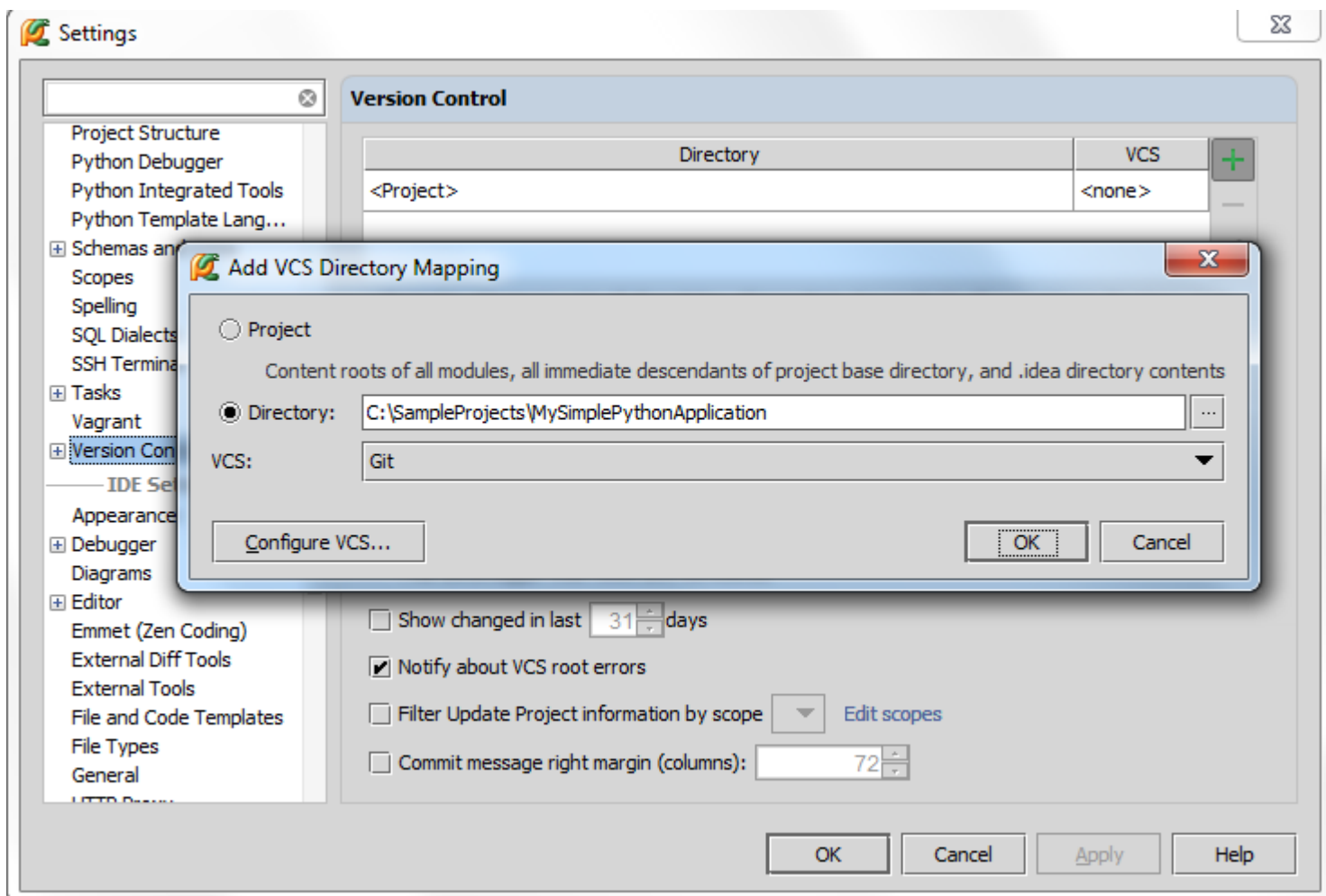


此时 Pycharm 会根据本地历史来浏览外部资源，甚至可以从这个菜单中启用版本控制系统，不过此时版本控制只针对工程根目录，跟多具体配置参见下面章节。

4、启用版本控制

单击主工具栏上的控制按钮，单击 [Version Control](#)，默认情况下这里只有根目录<project>。

首先，单击绿色加号，在 Add VCS Directory Mapping 对话框中，单击省略号按钮，选择对应目录，在下面的下来菜单中指定版本控制系统（这里使用 Git）。





当然，可以对每个目录指定对应的版本控制系统。

然后，单击应用，关闭对话框

5、对 Pycharm 外观的影响

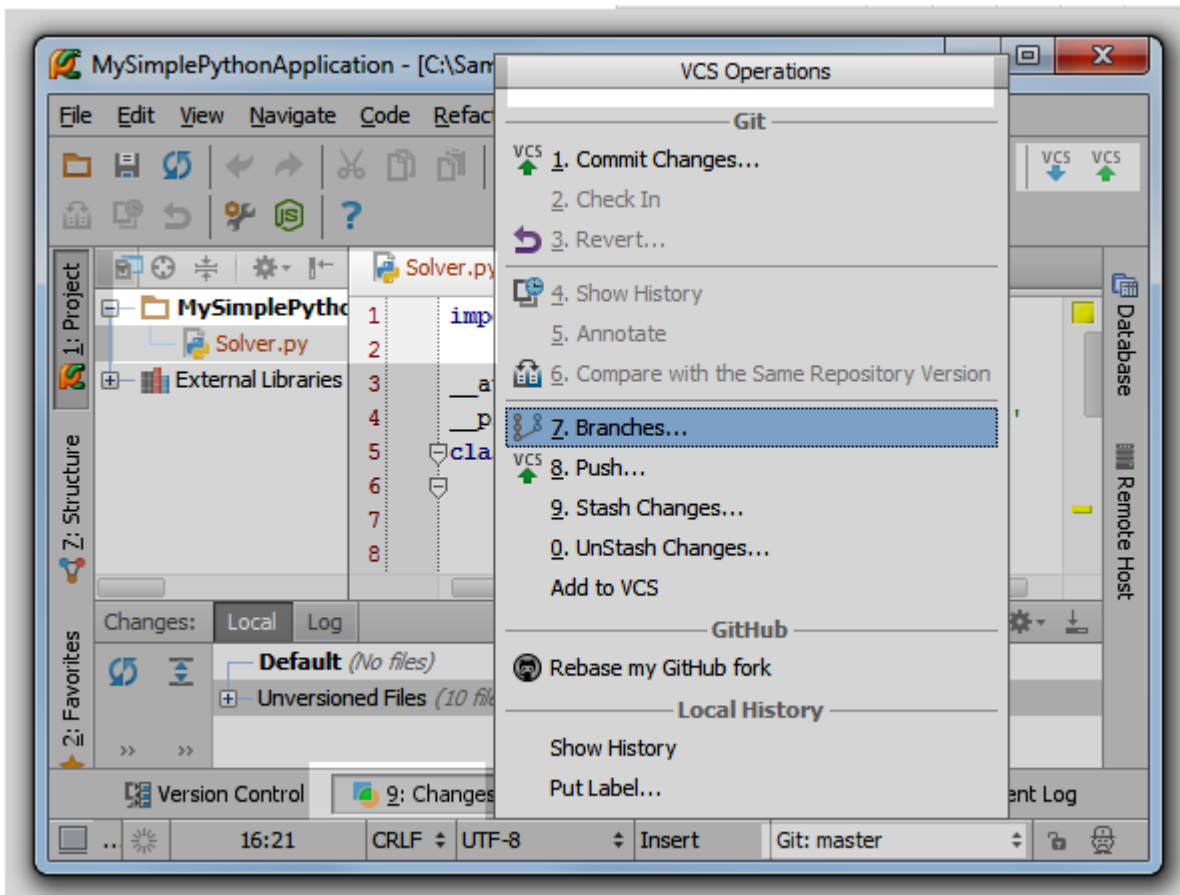
(1) Solver.py 文件颜色改变，意味着其尚未添加版本控制

(2) 出现 [Changes tool window](#) 窗口，类似于一个按钮位于 Pycharm 下边缘。单击它可以打开窗口，查看尚未添加版本控制的文件。

(3) VCS 菜单上出现更多命令，并且允许查看工程的更改以及更新整个工程，对应  和  按钮。主工具栏对应也会添加响应命令

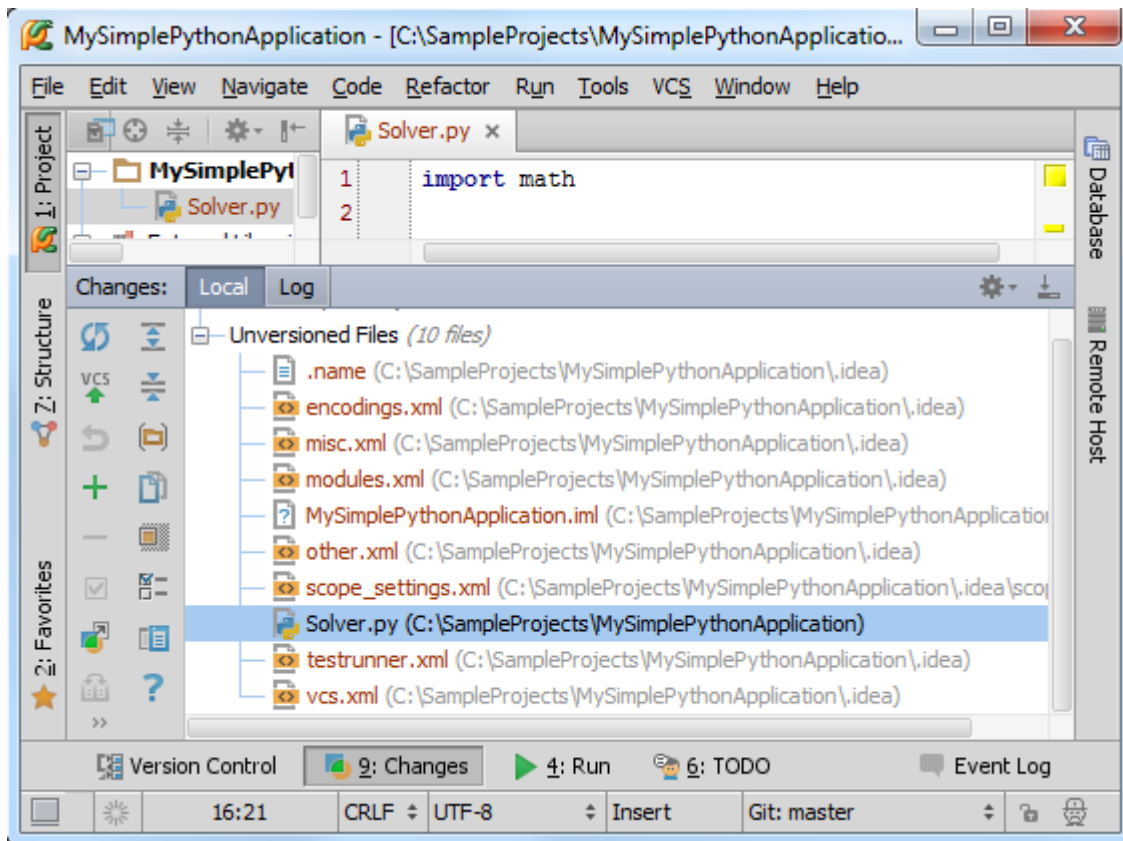
(4) VCS 菜单以及工程快捷菜单中出现 Git 节点（取决于所选择的版本控制），其中包含特定的 VCS 命令


(5) 状态栏出现 [Git widget](#)

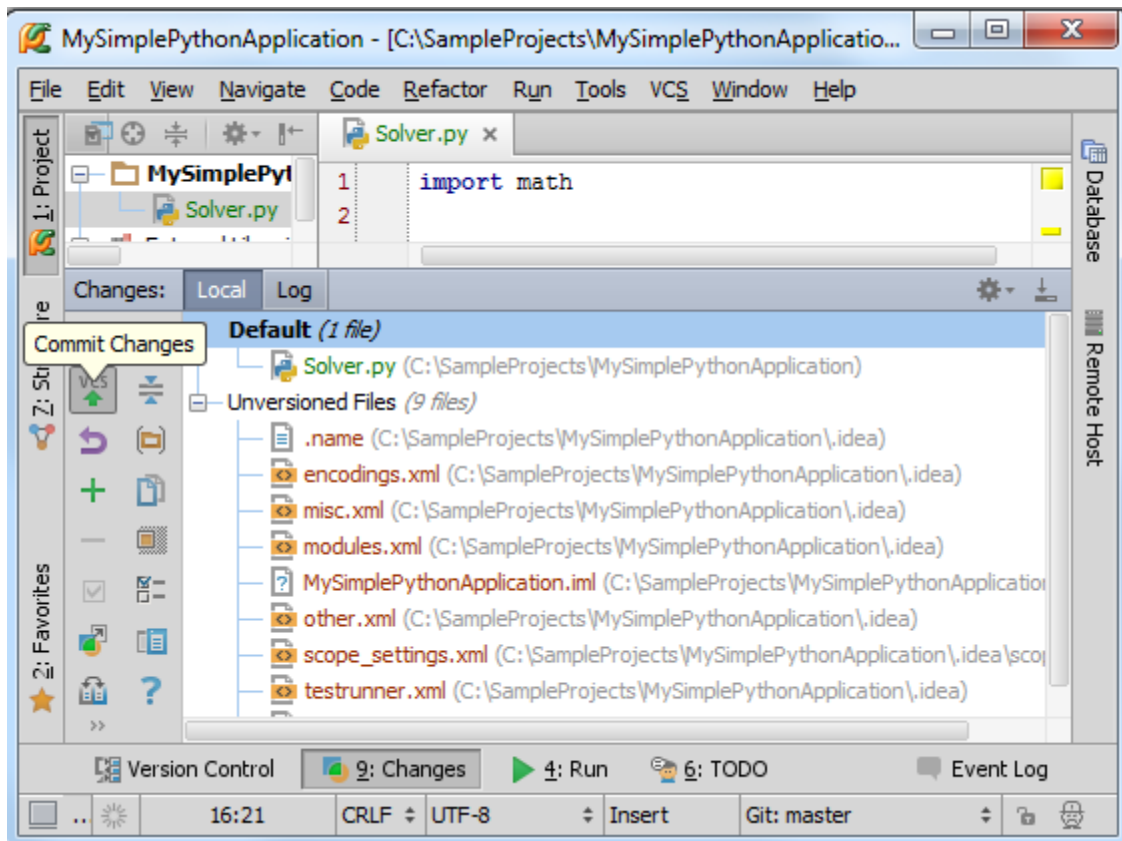


6、对一个文件添加版本控制

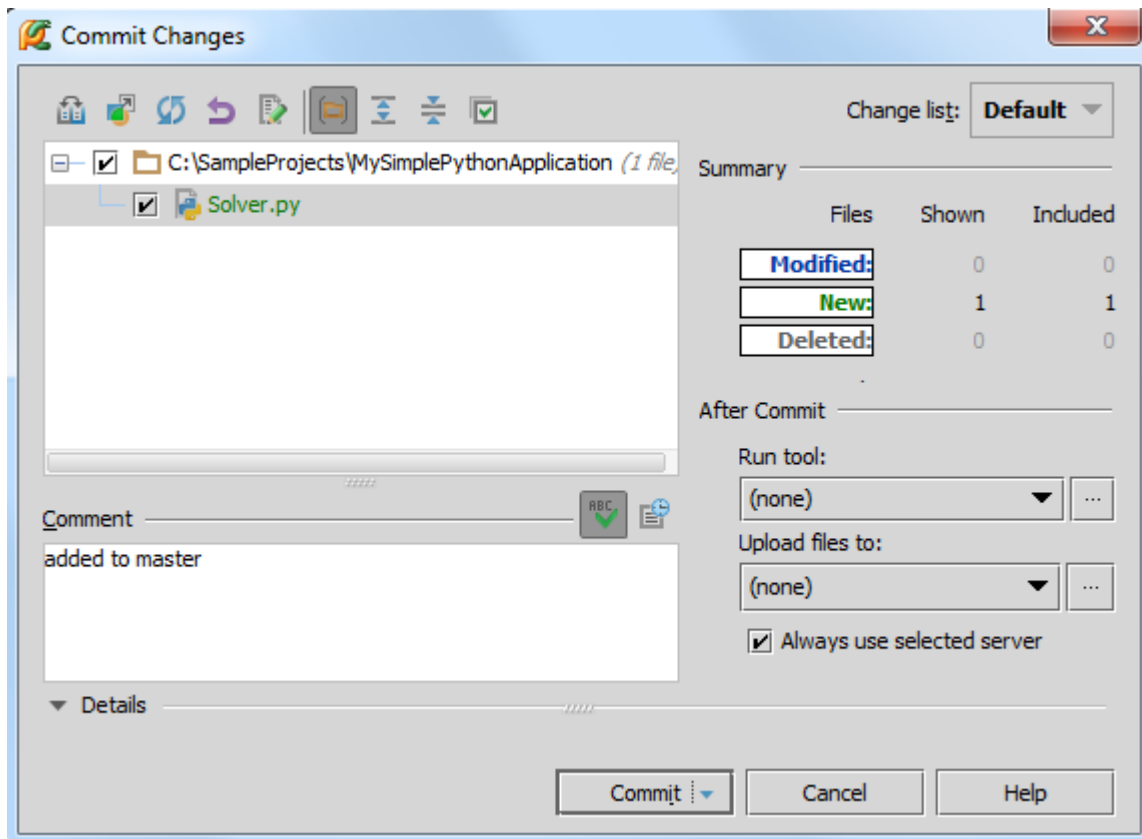
在 Changes tool window 窗口中未进行版本控制的文件显示如下:



选中 Solver.py，按下 Ctrl+Alt+A 来添加对应版本控制，文件颜色变为绿色，添加成功，但尚未进行托管。将文件移动到 Default 变更表下，按下 Ctrl+K（或者 ）来进行托管：




核实无误，单击 Commit 按钮：



大功告成，Solver.py 文件名再次变为黑色，意味当前没有未更新的更改。

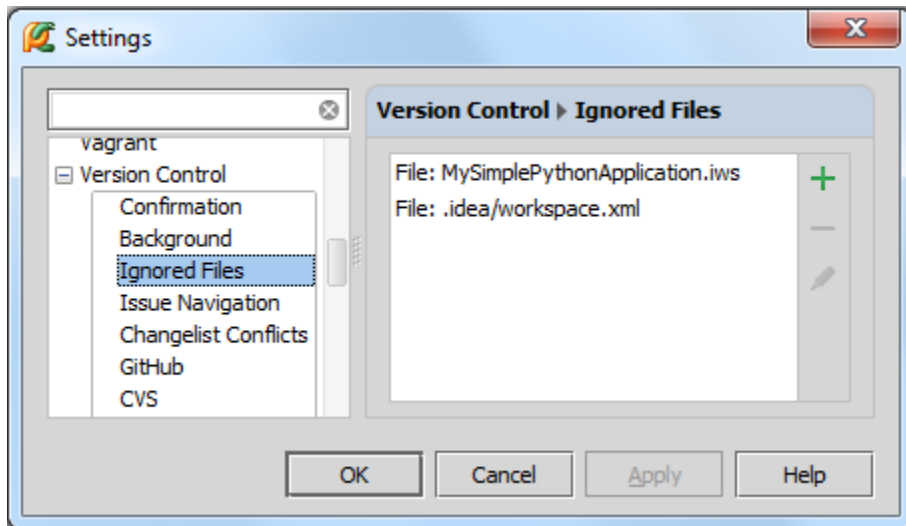
7、.idea 目录下的文件

单击 Changes tool window 窗口中的  按钮，发现工程根目录下的 idea 目录下的文件都未进行相关版本控制，并且这部分文件在工程窗口中不可见。

这部分文件保存了工程的配置信息，我们需要对其进行替换，除了版本控制文件。

8、忽略文件

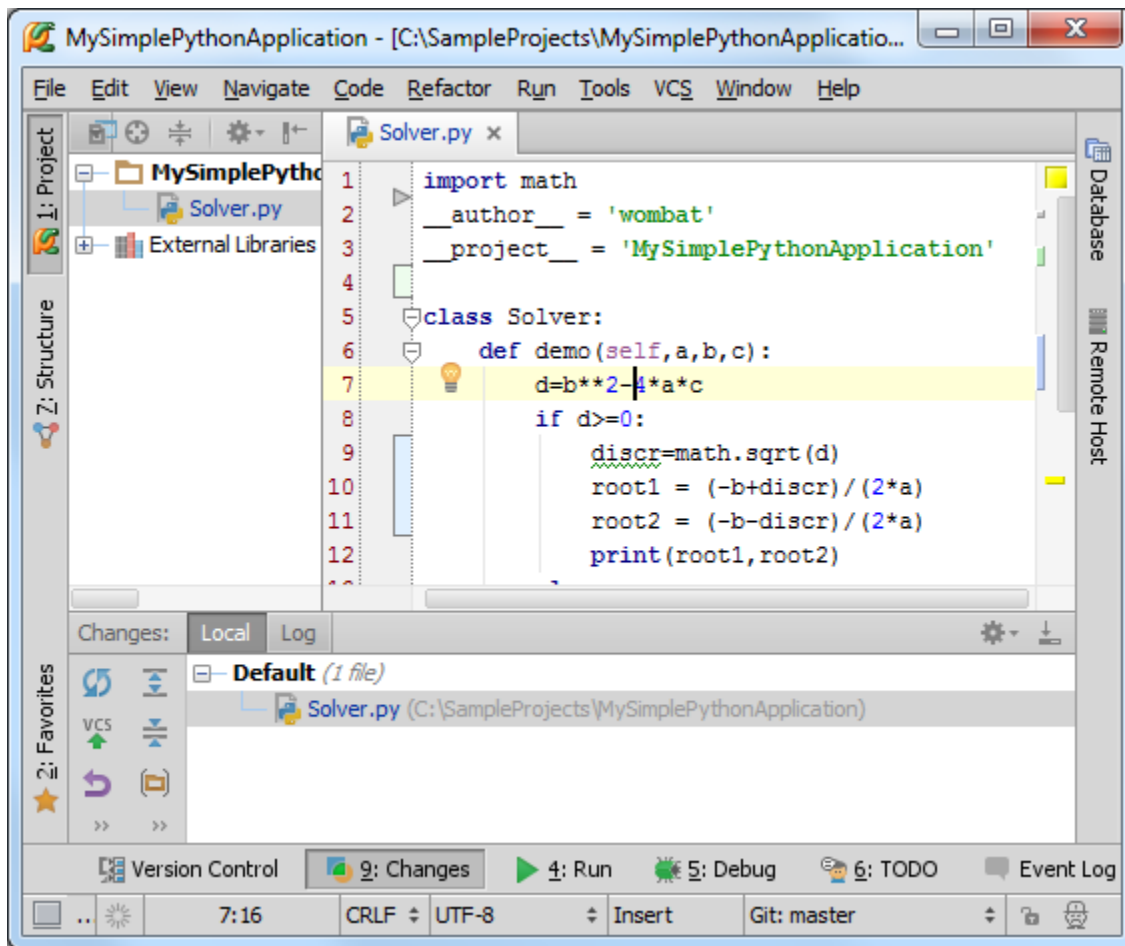
打开设置对话框的 [Ignored Files page](#) 页（设置→Version Control→Ignored Files），查看默认忽略文件列表：



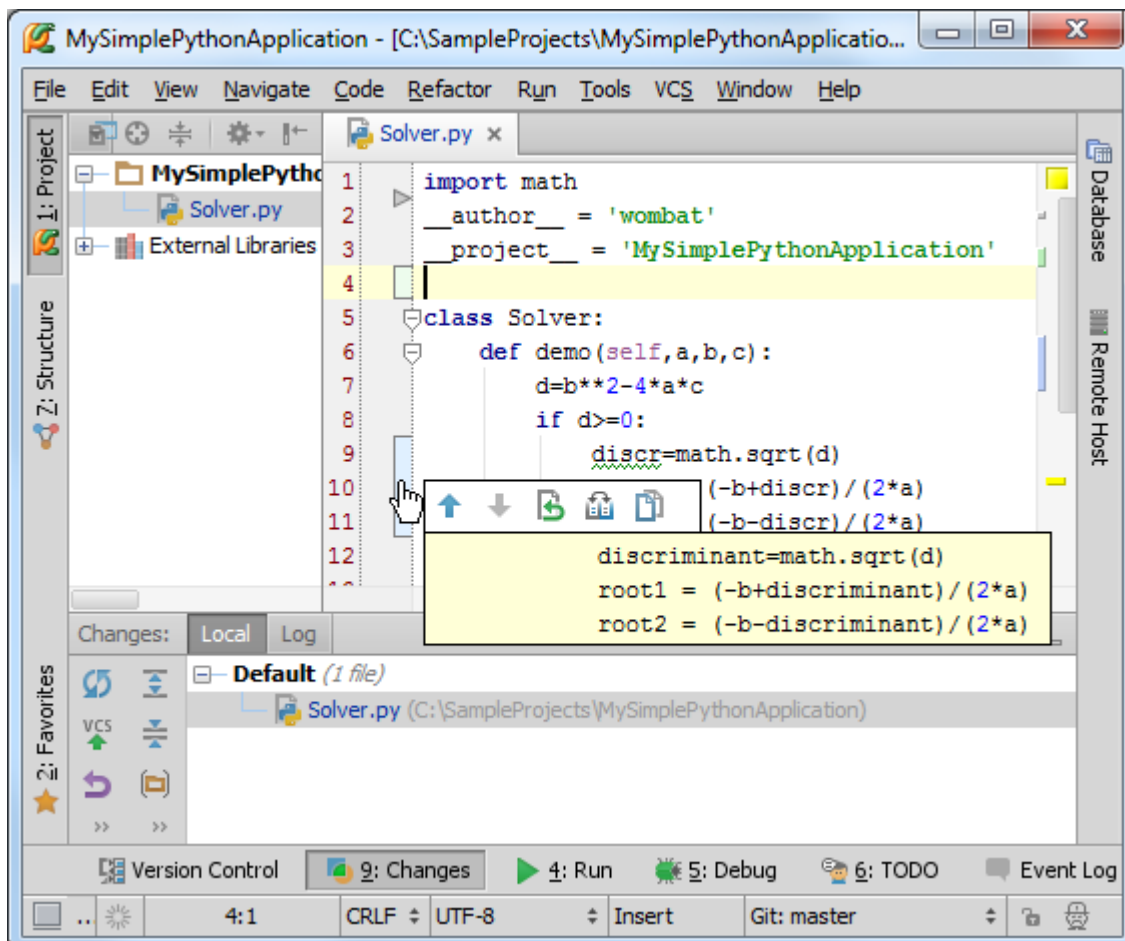
单击绿色加号选择忽略更多文件，详见 [Configuring Ignored Files](#)。

9、编辑器的变化

编辑代码，左槽会标记出所有更改：



单击一个标记，会弹出一个工具窗口：



这个弹出的工具栏能够帮助你进行导航、浏览更改等操作。

更多信息参见 [Using Change Markers to View and Navigate Through Changes in the Editor](#)

此时代码文件名变为蓝色，意味着当前存在尚未托管的更改。

10、获取更新日期

按下 Ctrl+T

单击  按钮

使用 VCS→Update Project...菜单命令

最全 Pycharm 教程（38）——Pycharm 版本控制之远程共享

1、主题

介绍如何通过 GitHub 共享你的本地 Git 版本库

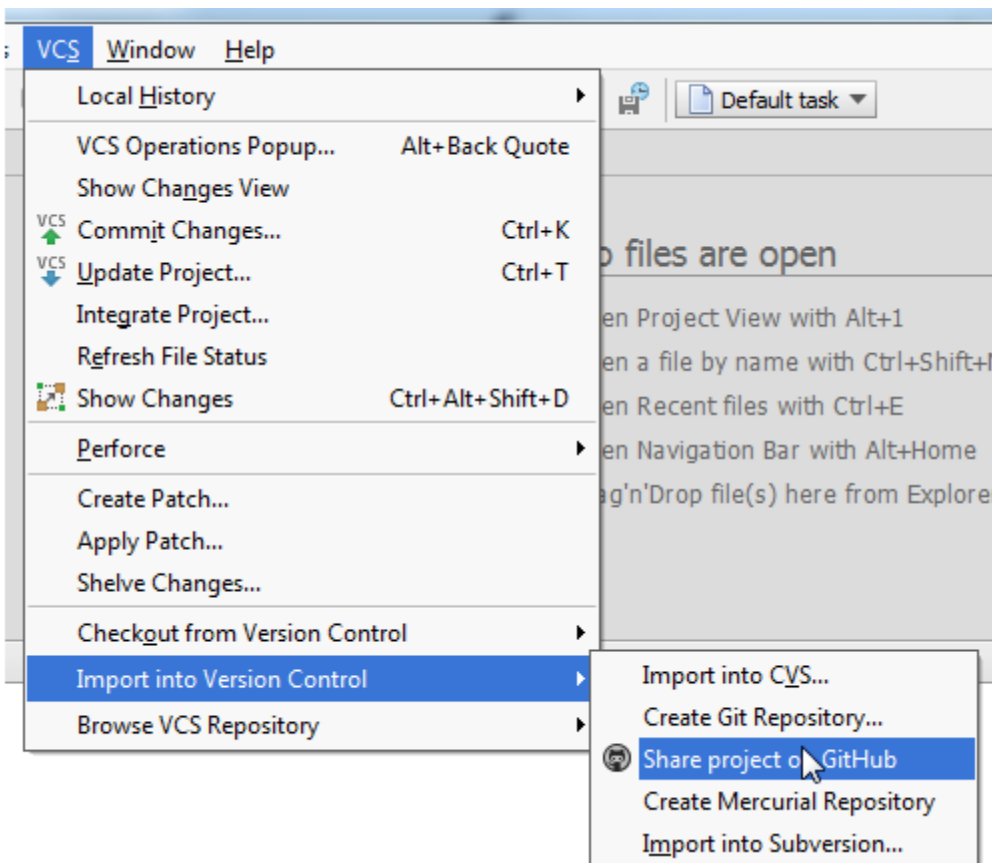
2、准备工作

- (1) Pycharm 版本为 2.7 或者更高
- (2) Git 以及 GitHub 可用
- (3) 有 [GitHub storage](#) 的读写权限，因此需要先创建一个账号以及远程版本库
- (4) 在 "[Using PyCharm's Git integration locally](#)" 教程基础上

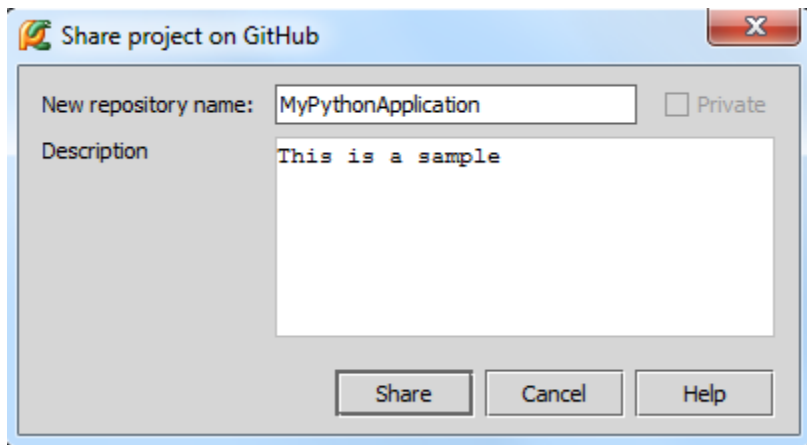
3、分享源码的两个方式

4、在 GitHub 上分享

单击 VCS→Import into Version Control→Share project on GitHub 主菜单命令：



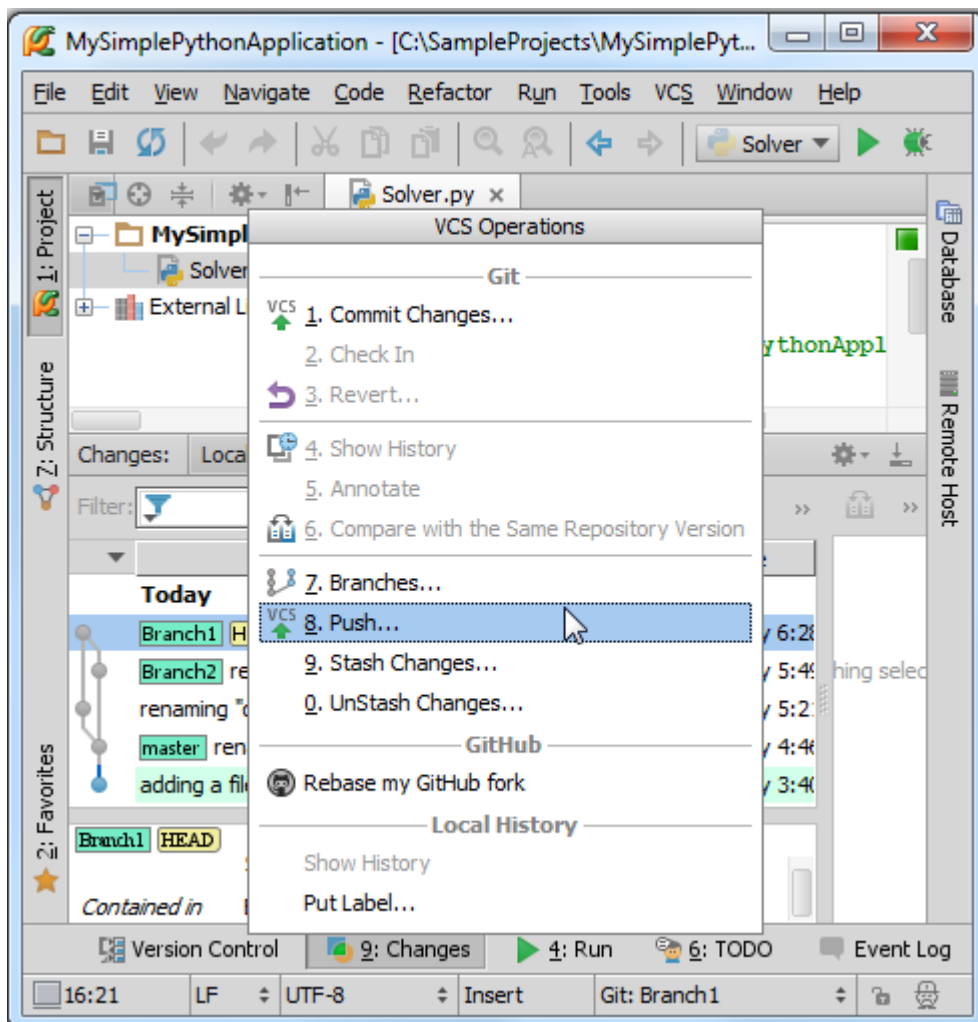
如果你已经配置好的 Github 账户，其将会显示在下面列表中，对其命名以及进行简要说明：



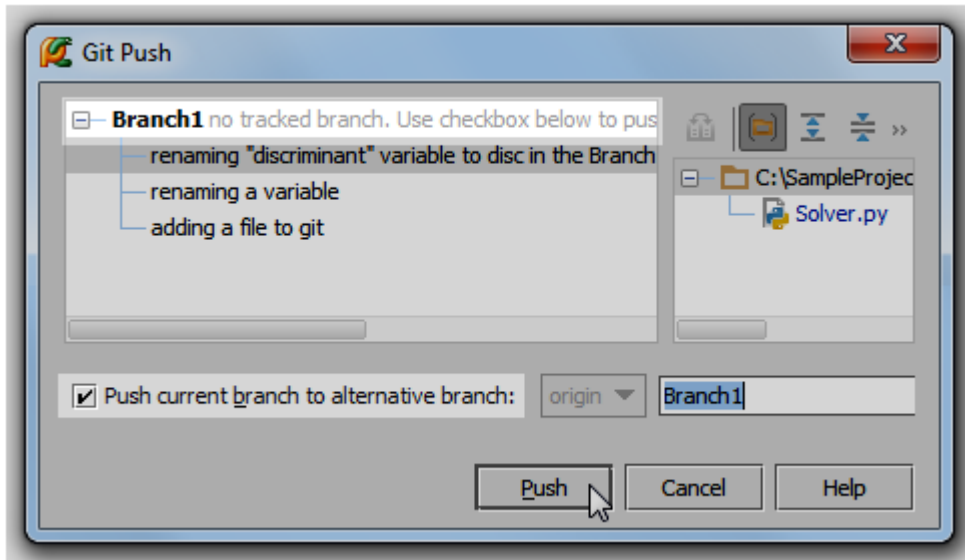
单击 **Share**，共享完成。

5、源码推送

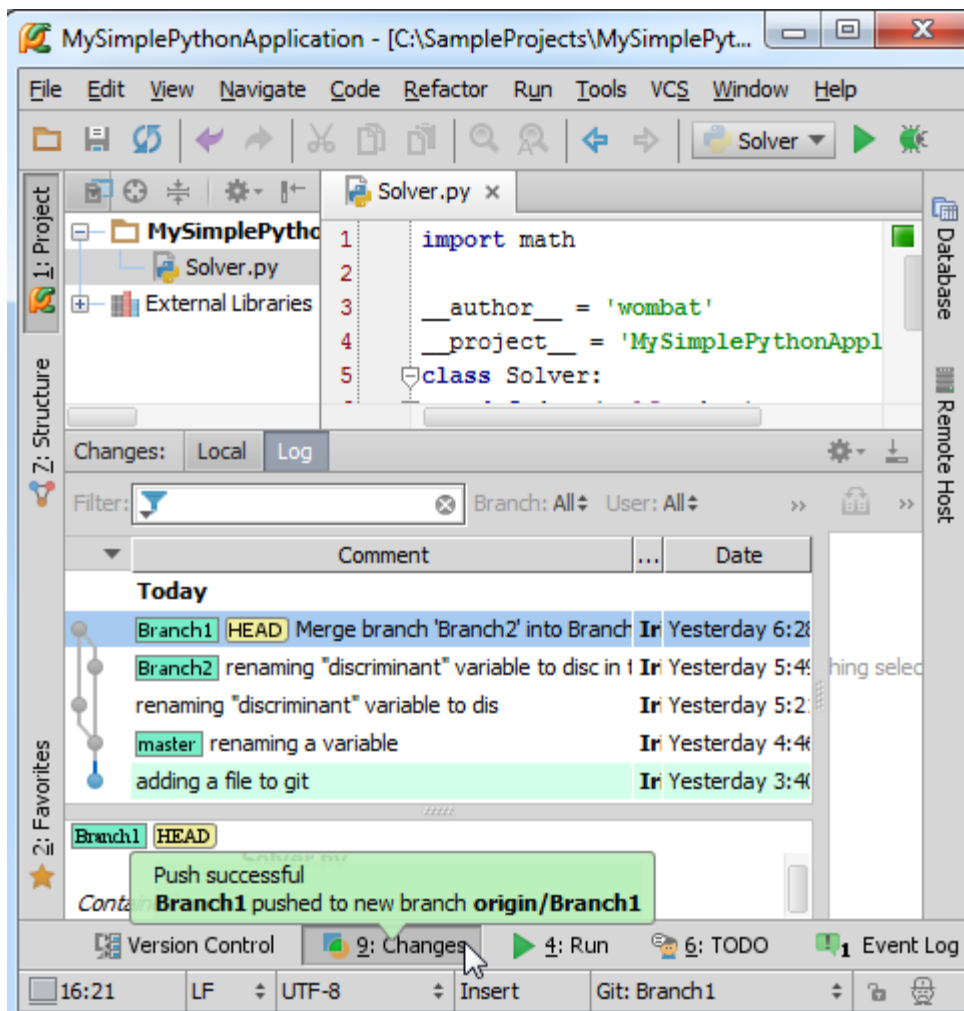
按下 `Alt+``，在弹出窗口中按下 `8` 来调用 `Push` 命令：



在 `Git Push` 对话框中，选择待推送的版本，在做分支的首次推送时，需要勾选相应的复选框：



推送过程中 Pycharm 会给出相关提示球：



推送完成。

最全 Pycharm 教程（39）——Pycharm 版本控制之本地 Git 用法

1、主题

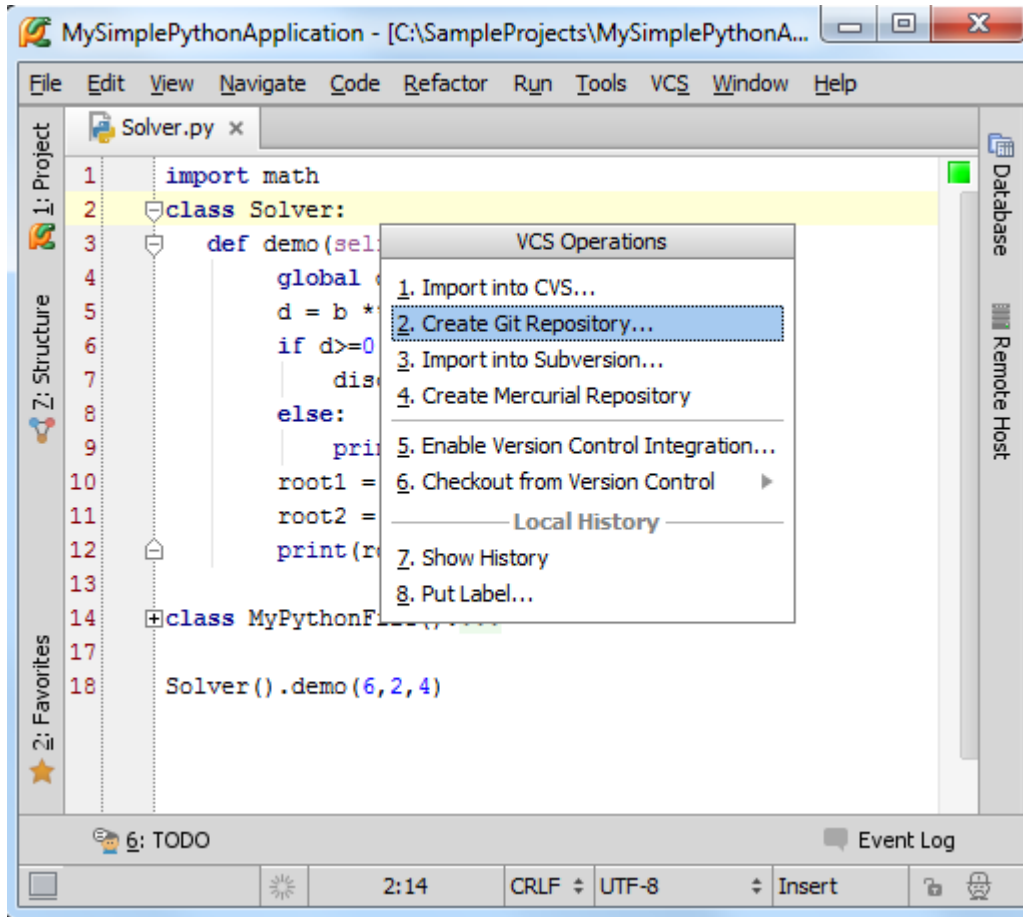
介绍如果通过 Pycharm 使用本地 Git 集。

2、准备工作

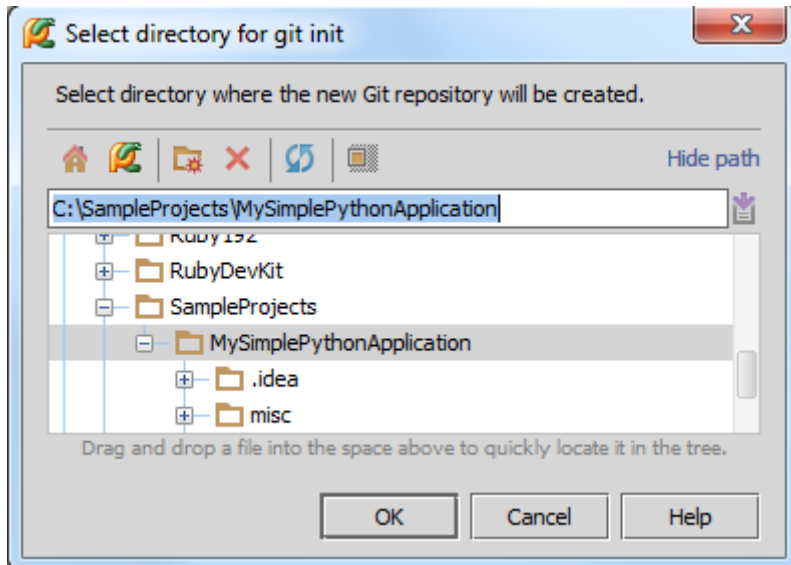
- (1) PyCharm 版本为 2.7 或更高
- (2) 已经创建一个工程
- (3) Git 插件可用，对应可执行文件在 [Git page](#) 页面正确配置

3、创建一个 Git 集

按下 Alt+` 显示常用的 VCS 命令（也可以通过主菜单 VCS→VCS Operations Popup），选择 Create Git repository 命令：

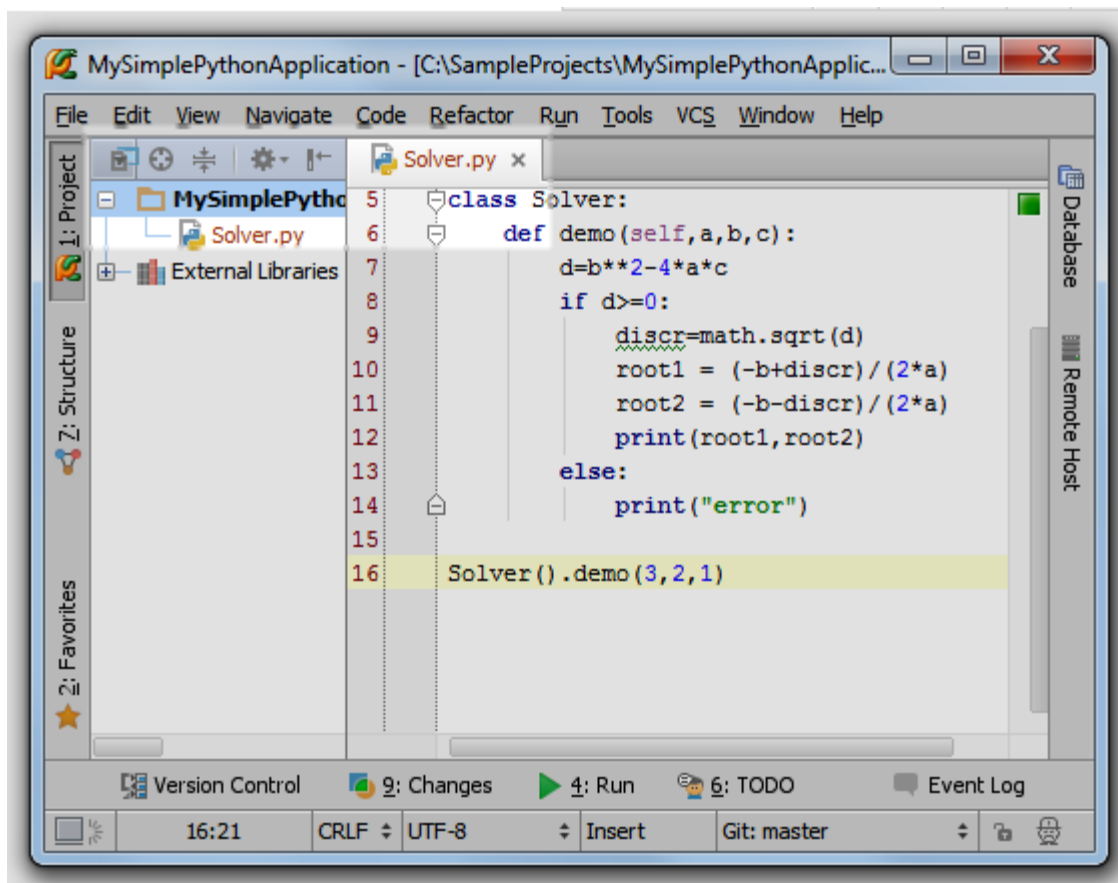


Git 通过在父目录下创建一个.git 文件夹来安装本地版本库。此处我们选择在根目录下创建.git 目录：



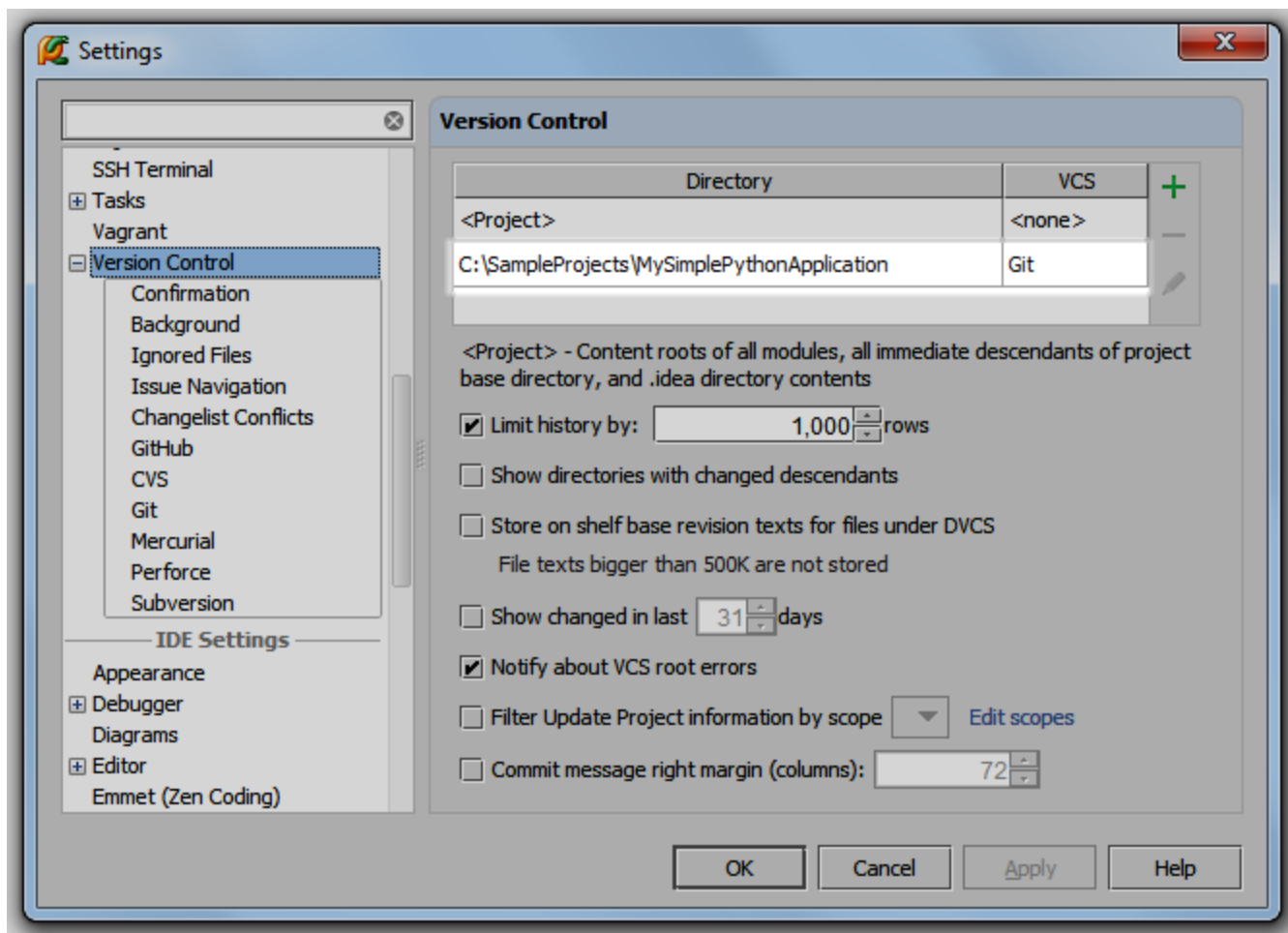
4、Pycharm 用户界面变化

- (1) 出现 [Changes tool window](#) 窗口
- (2) Solver.py 文件名变色



这也意味着这个新的文件尚未添加版本控制（稍后介绍）。

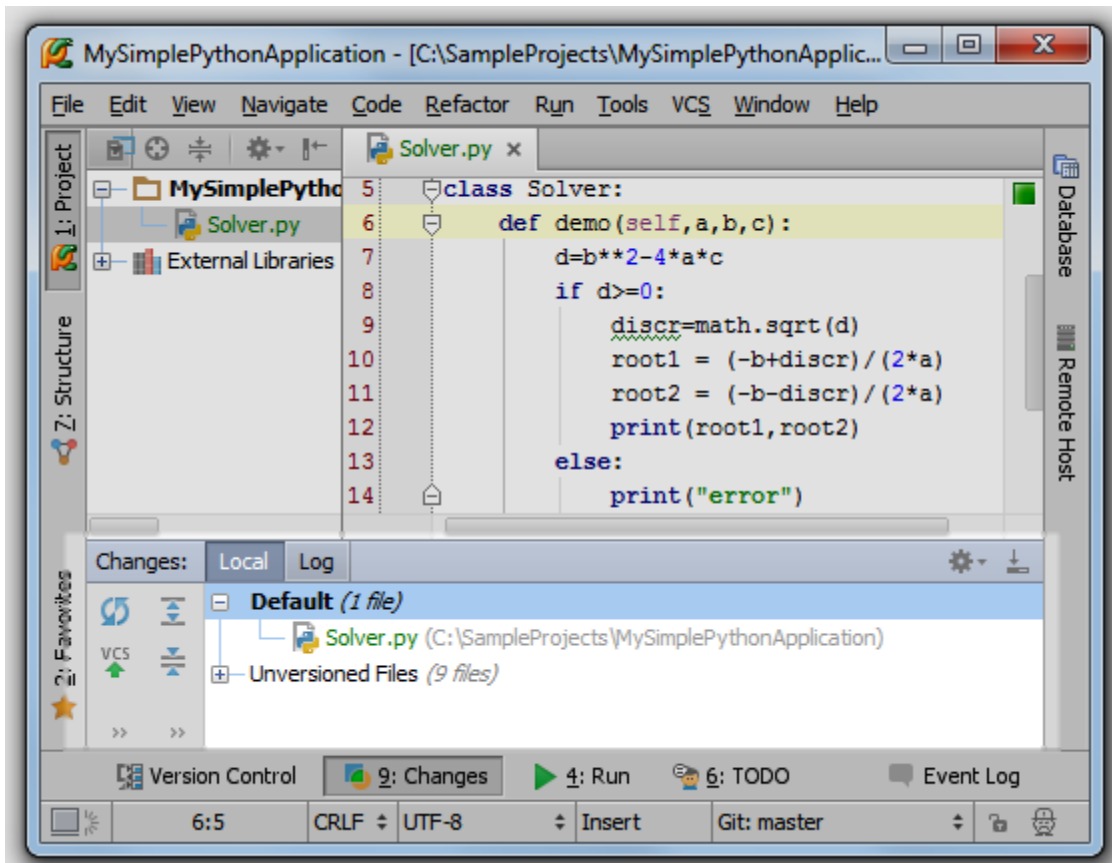
- (3) 打开设置对话框（Ctrl+Alt+S），单击 [Version Control](#)，发现 MySimplePythonApplication 目录已经和 Git 关联：



5、为文件添加版本控制

方法 [put a file under version control](#)，这里例举一种。选择 Solver.py 文件，按下 Ctrl+Alt+A。

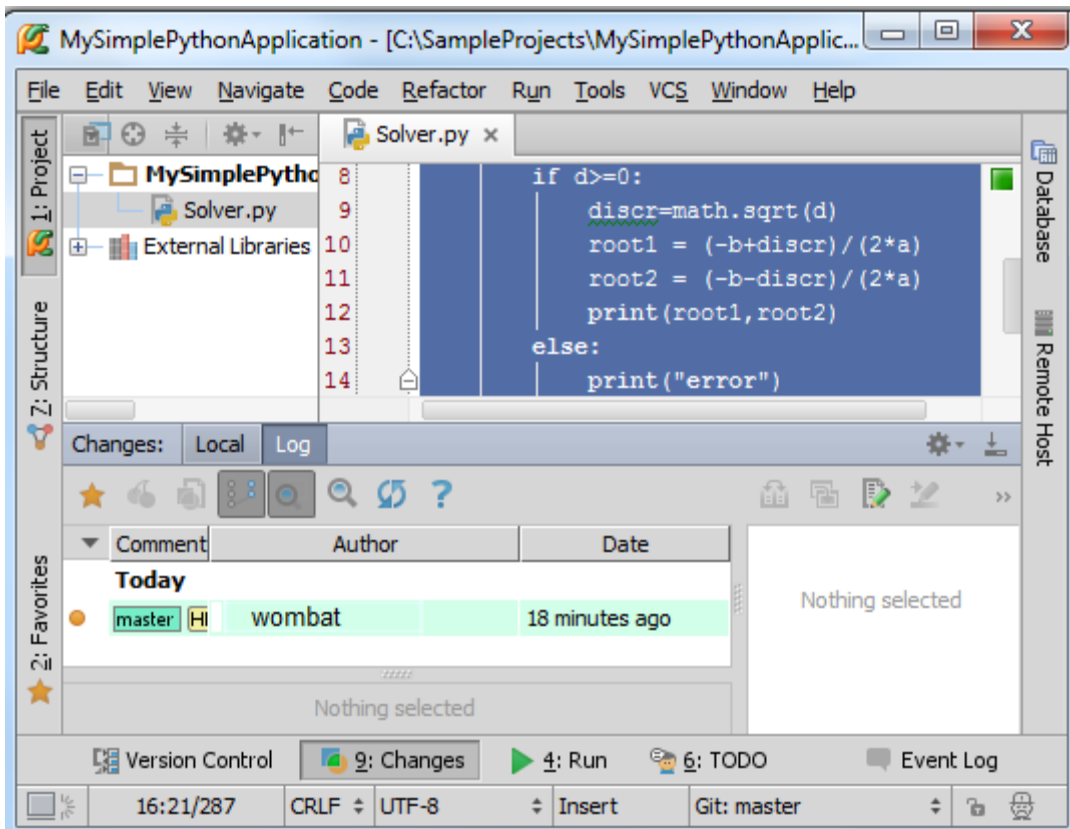
Solver.py 文件变为绿色，意味着已经进行了版本控制，但尚未托管：



6、提交本地版本库

在 Changes tool window 窗口中选择 Solver.py 文件，按下 Ctrl+K，输入信息，单击 Commit。

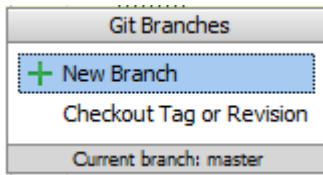
打开 Changes tool window 的 [Log tab](#) 选项卡查看；



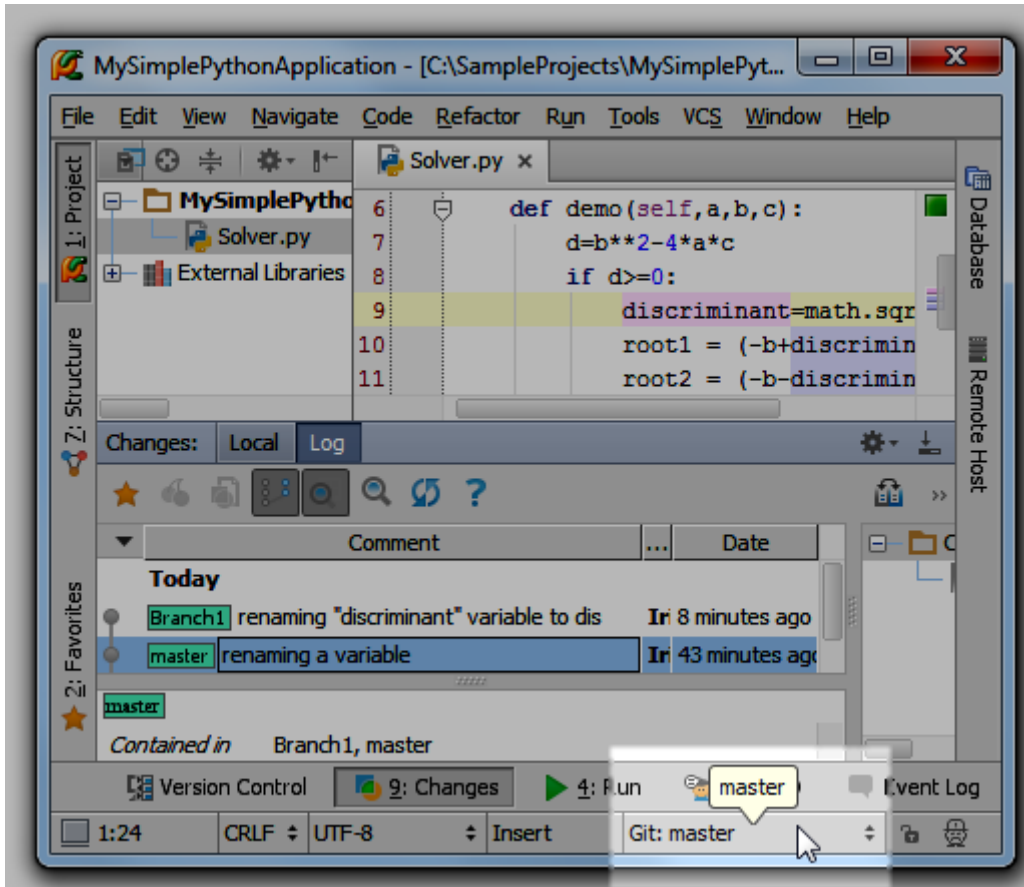
7、查看当前分支

两种方式：

第一，使用主菜单命令 VCS→Git→Branches，在弹出的窗口中查看：



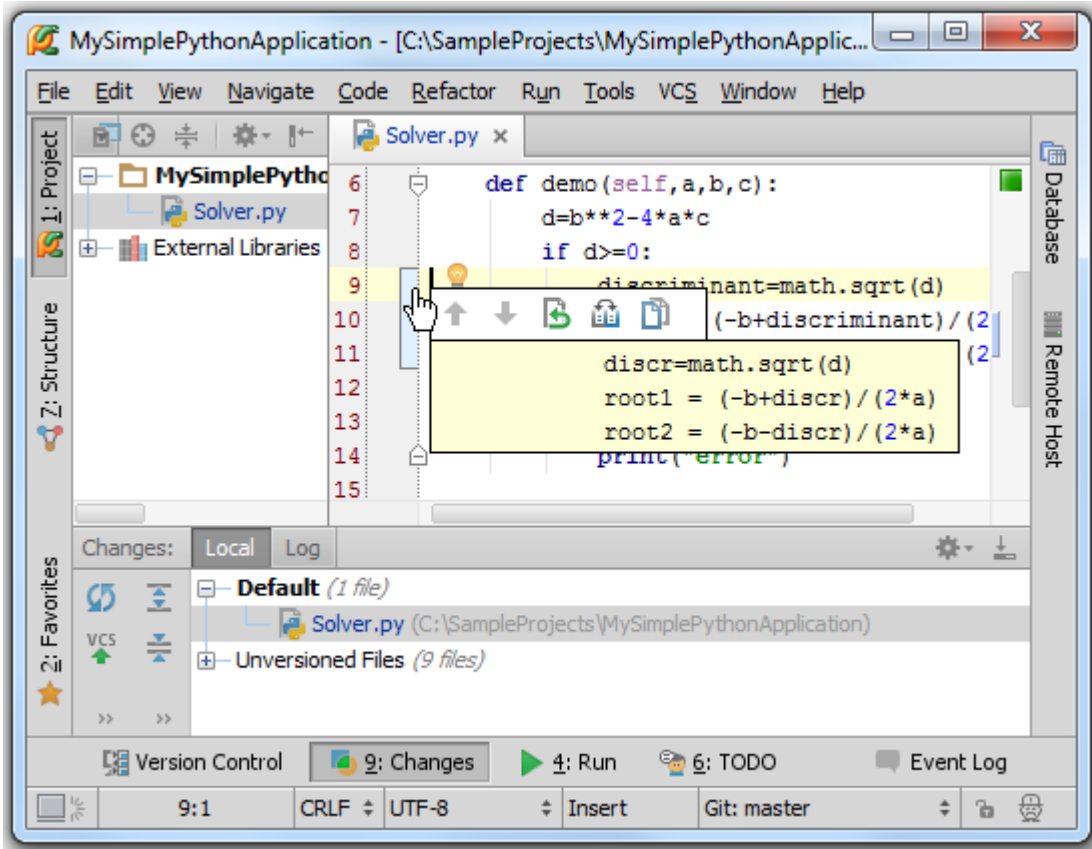
第二，使用状态栏上的 Git 组件：



8、更改主分支代码

以重命名为例。将光标定位在 discr 符号上，按下 Shift+F6，输入新名字 discriminant。

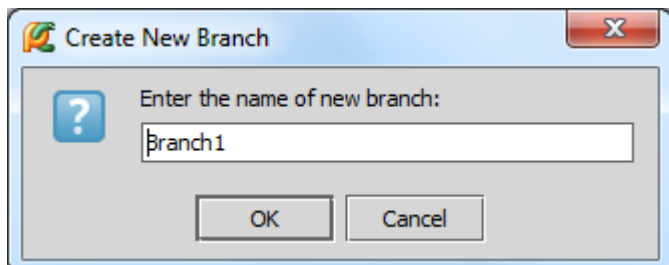
注意此时左槽会产生相应标记：



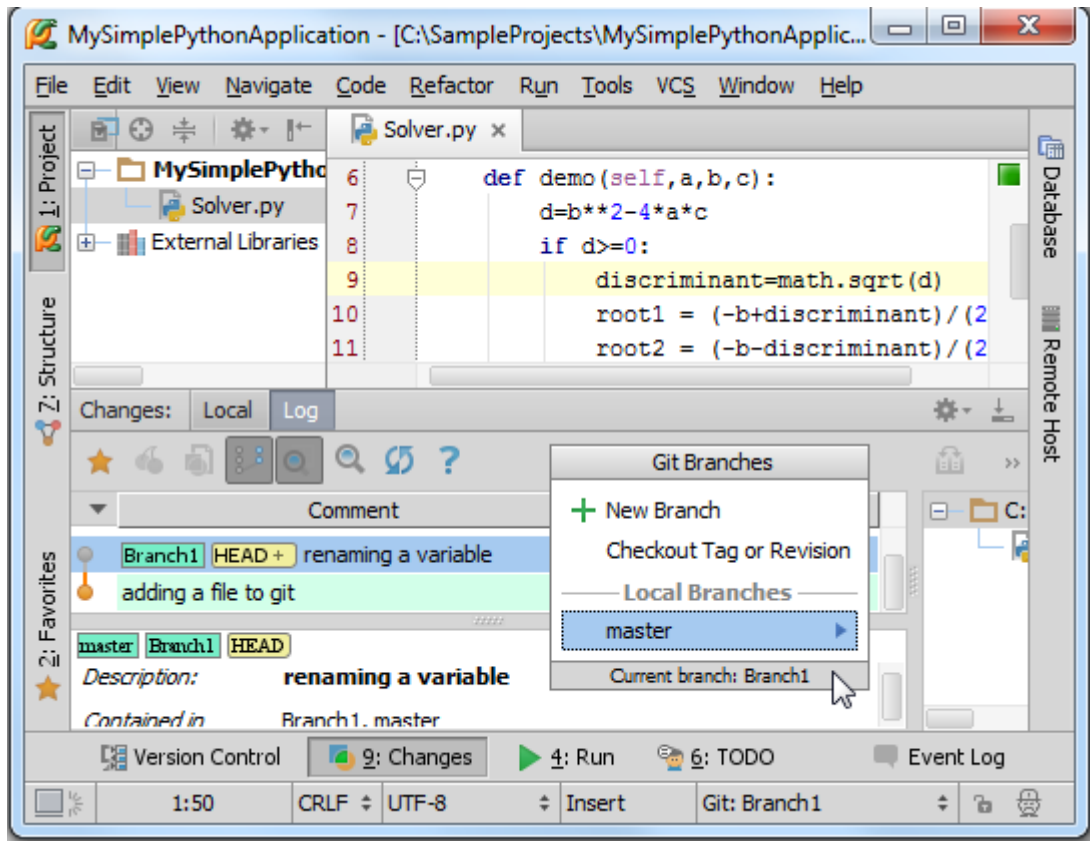
单击这个标记，Pycharm 会弹出窗口提示当前所做更改。按下 Ctrl+K 快捷键更新代码。

9、创建一个新的分支

单击状态栏上的分支图标，输入名称：



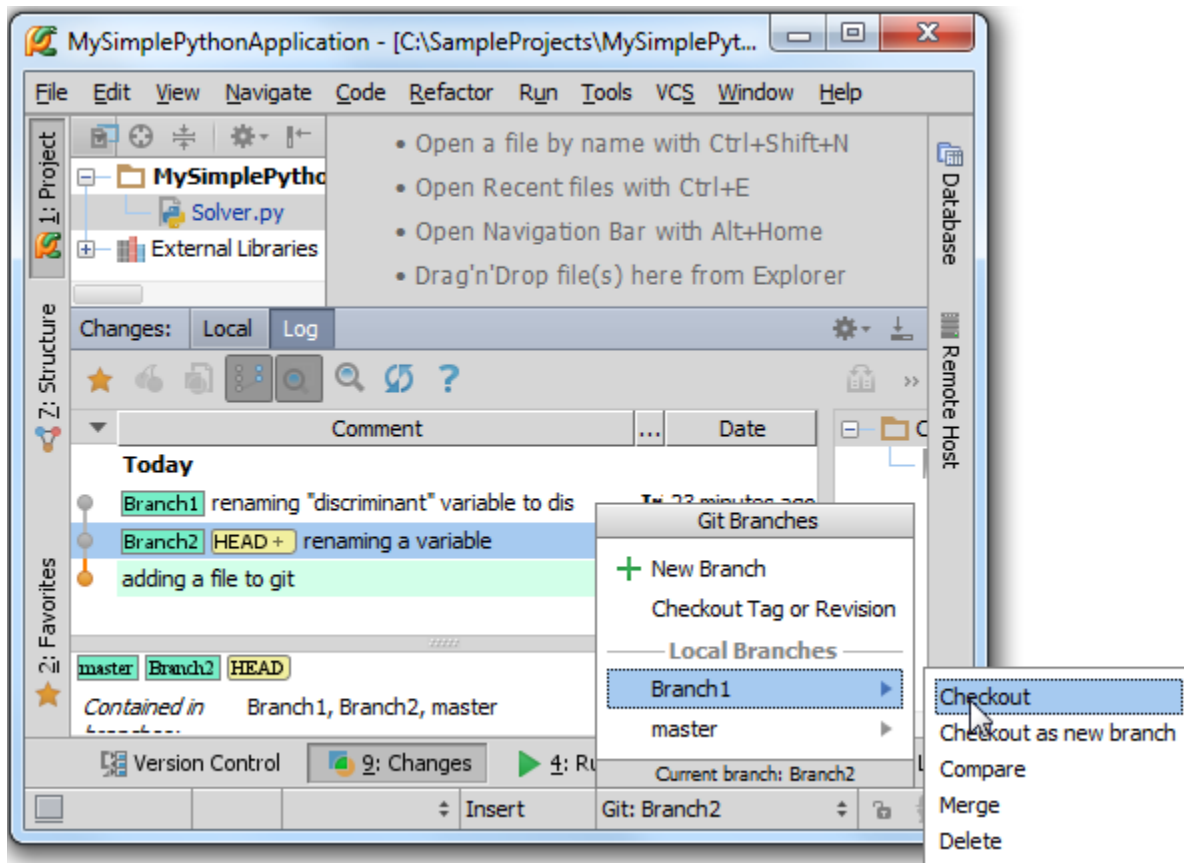
新分支现在在当前 Changes tool window 窗口中，作为一个选项卡：



接下来再从主分支中创建第二个分支（Branch2）。

10、更改新分支中的代码

切换到分支 1：

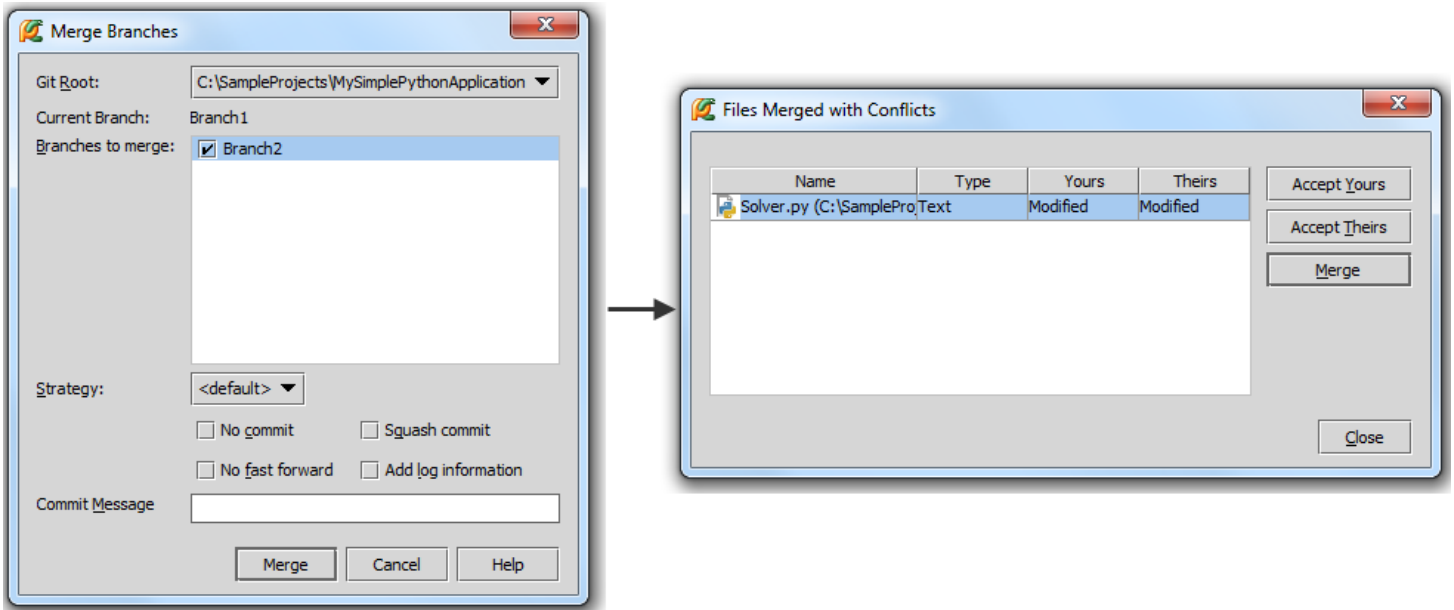


将光标定位在 discriminant 符号上，按下 Shift+F6，输入简写，例如 dis，然后按下 Ctrl+K 托管更改。

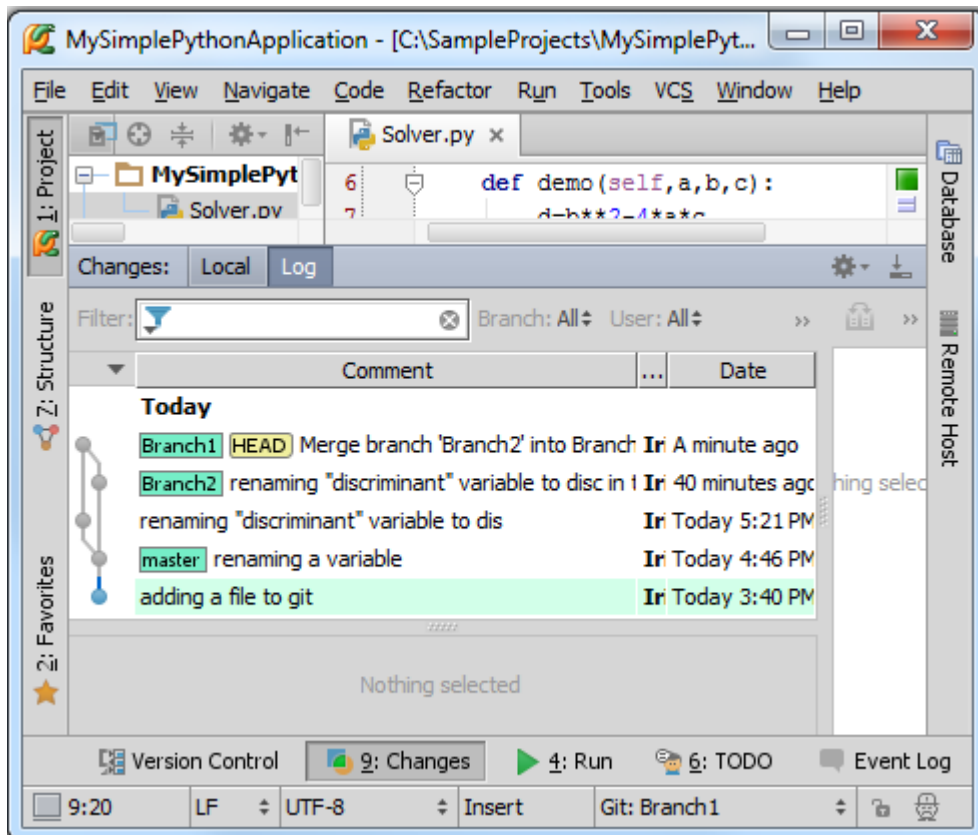
对分支 2 进行同样操作。

11、合并分支

只能将当前分支合并到其他分支上。使用 VCS→Git→Merge Changes 的主菜单命令：



选择接受这些更改并托管，再次查看，发现比之前更复杂了：



最全 Pycharm 教程（40）——Pycharm 扩展功能之捆绑插件 TextMate

1、主题

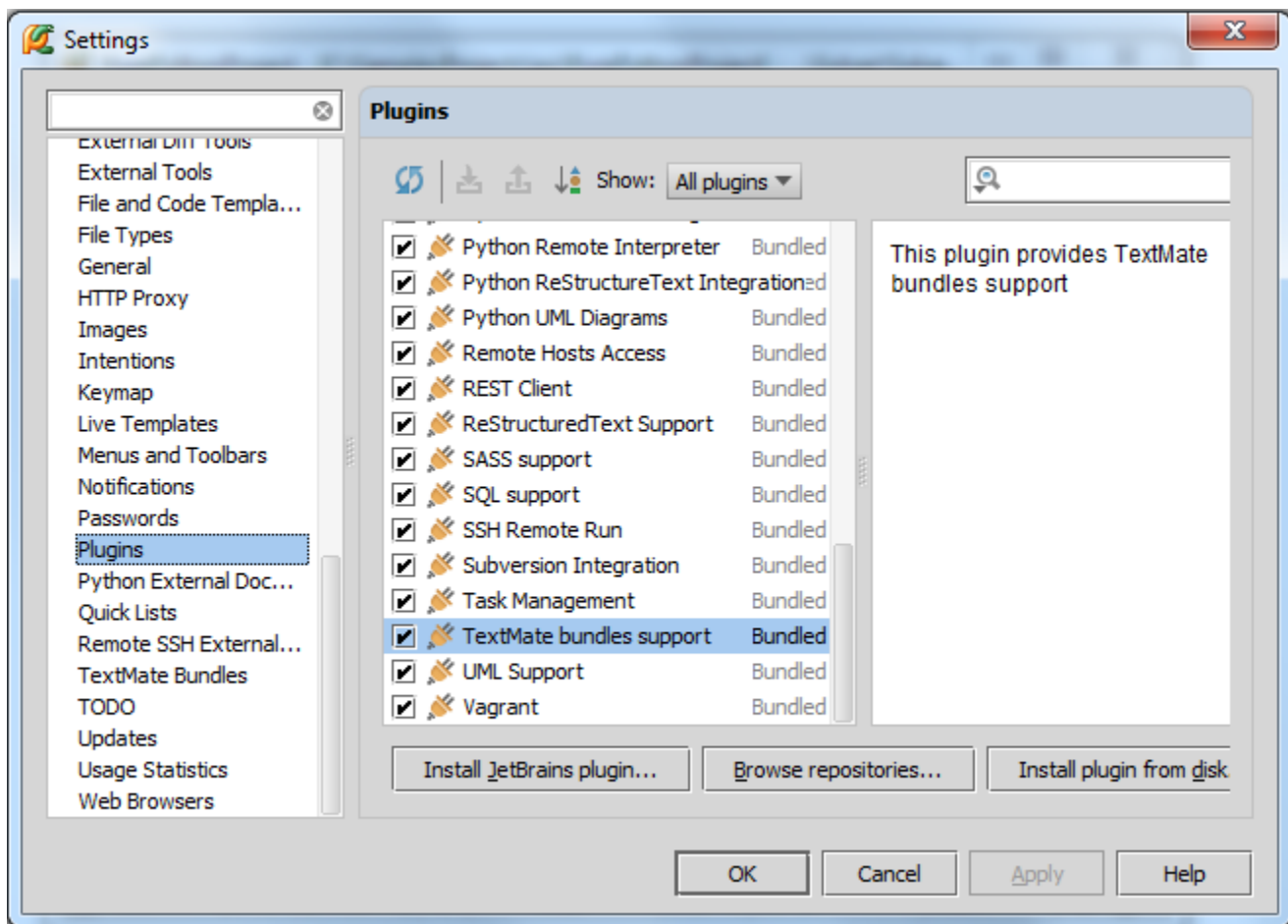
介绍如何在 Pycharm 中使用 TextMate 来格式化编辑各种代码文件

2、准备工作

(1) 已经下载了相关捆绑软件，如 [GitHub](#) 或者 [Subversion](#)

(2) Pycharm 版本为 2.7 或更高

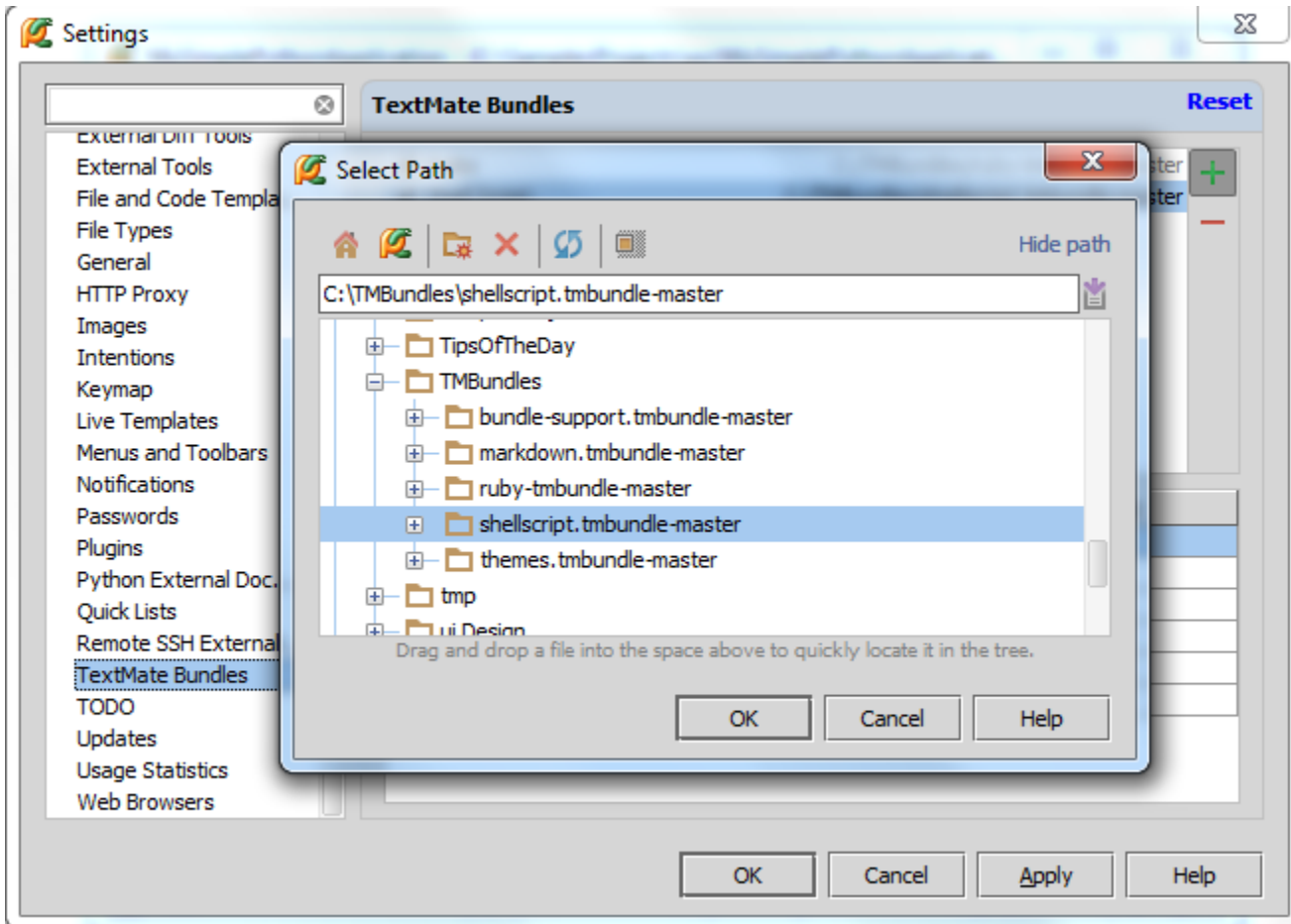
(3) **TextMate bundles support** 插件正常工作。打开设置对话框，在 **IDE Settings** 节点下，单击 **Plugins** 页，勾选对应插件复选框：



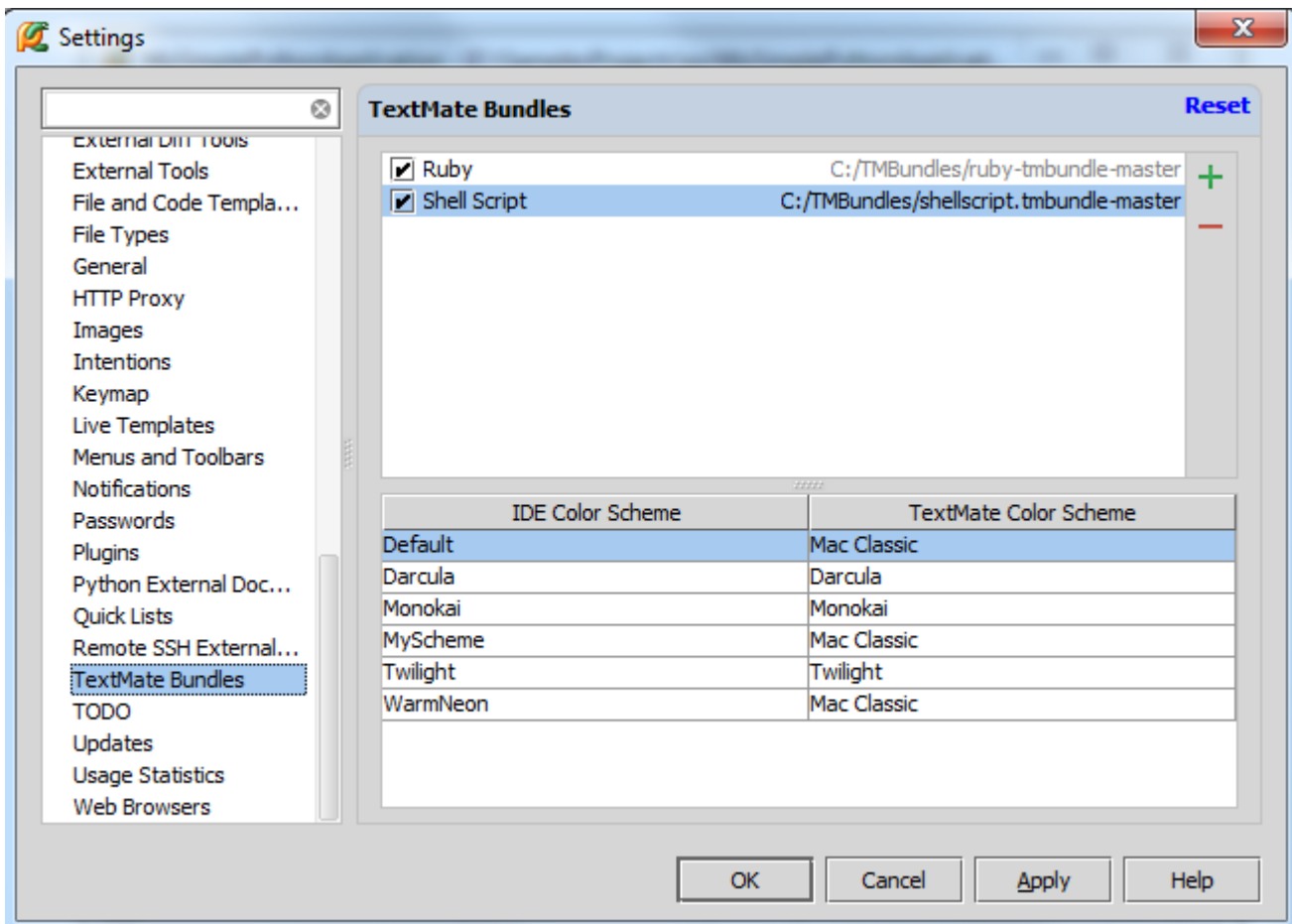
3、导入捆绑软件集合

假设你希望使用高亮语法来编辑 Shell Script 文件，此时需要先下载 [Shell Script TextMate Bundle](#)，然后导入。

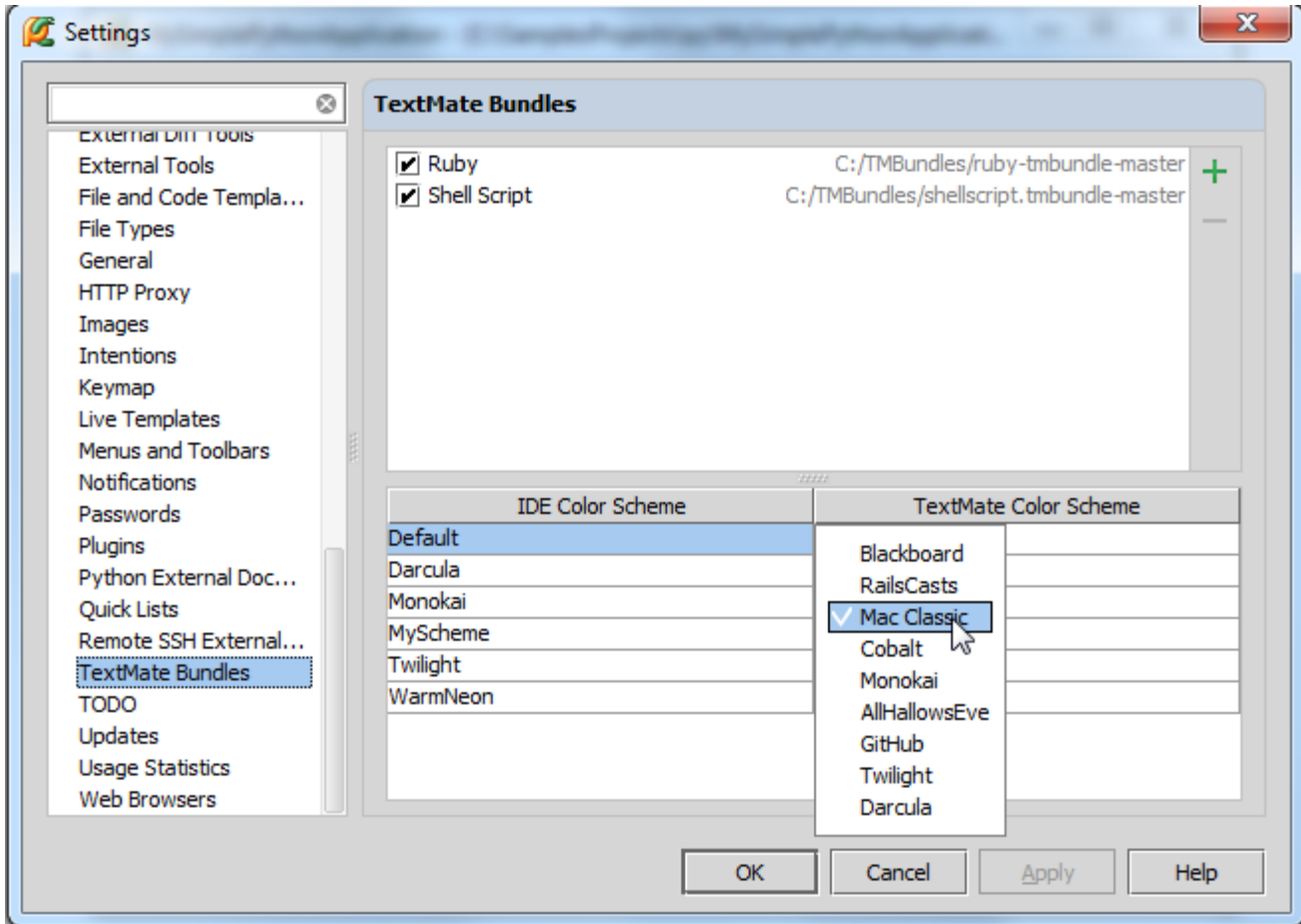
在主工具栏单击设置按钮，在 **IDE Settings** 节点下单击 [TextMate Bundles](#)，在 **TextMate Bundles** 区域单击绿色加号，定位待加载路径：



单击 OK 按钮，Shell Script 出现在绑定列表且本地路径可见：



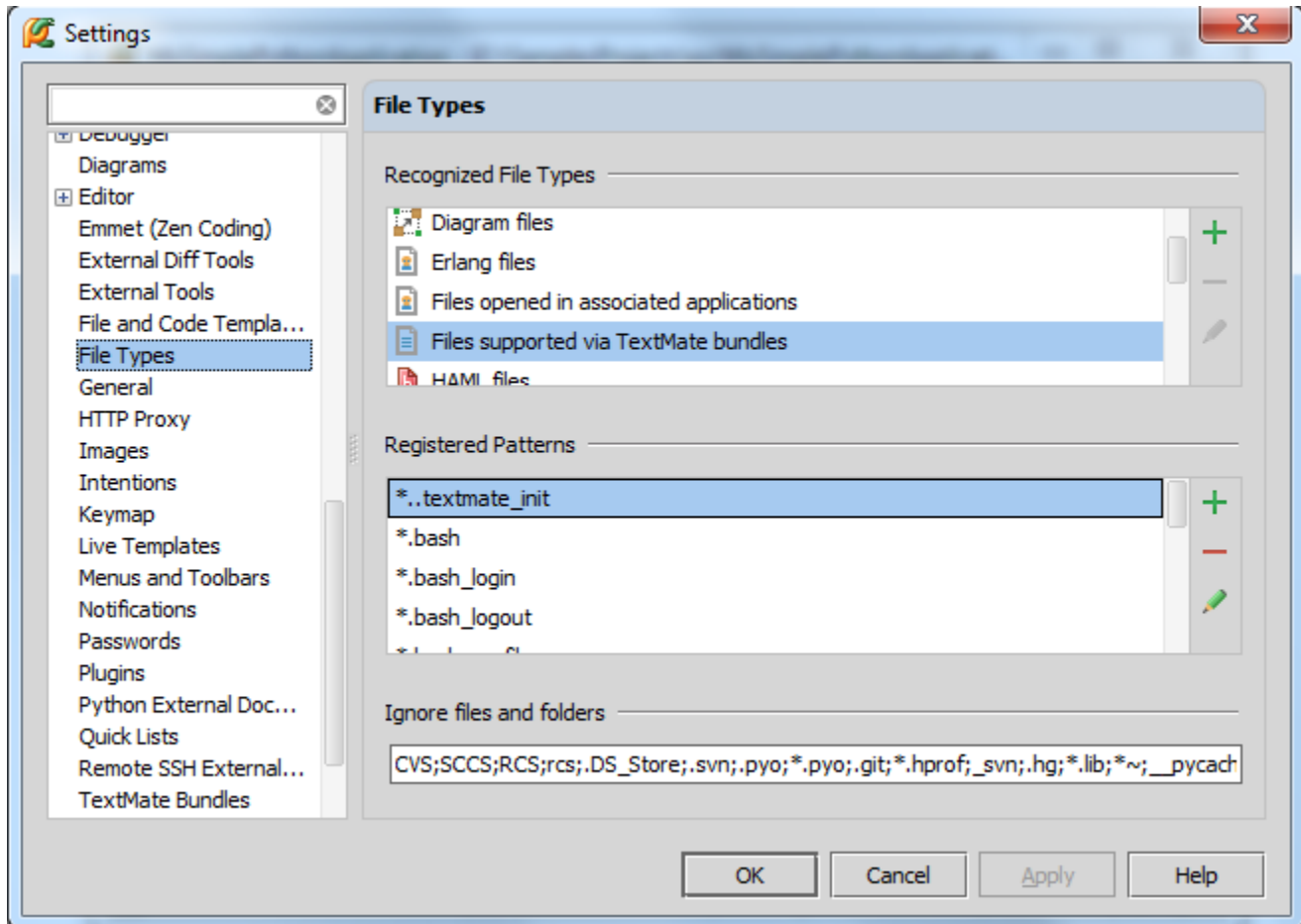
默认主题方案为 Mac Classic，可以单击对应表单进行更改：



4、增加文件类型

大部分 TextMate 绑定本身已经添加了很多文件扩展，当然我们可以继续增加它。

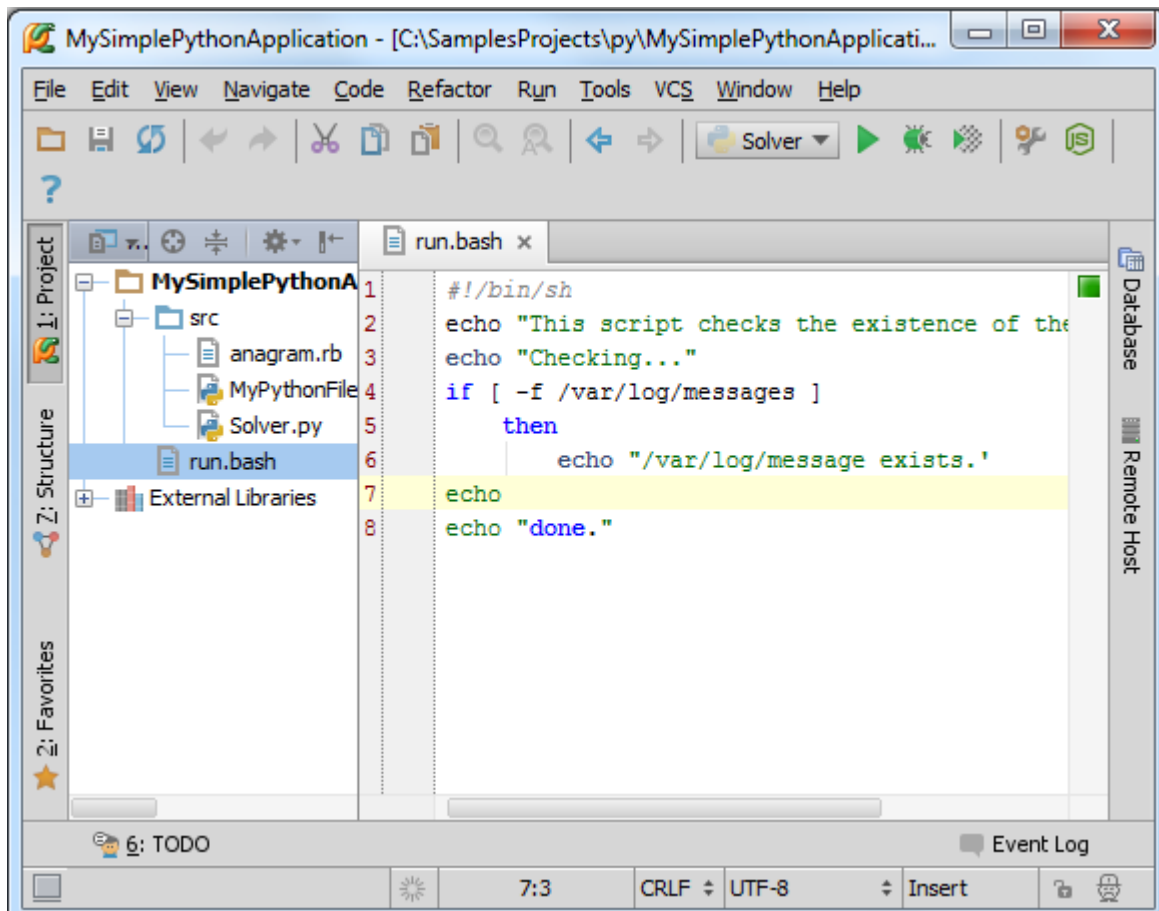
在设置对话框单击 [File Types](#)，在页面的 **Recognized File Types** 区域，定于到 **Files supported via TextMate Bundles** 选项，在下方的 **Registered Patterns** 中，显示了当前绑定所支持的所有文件类型。单击绿色加号，在 **Add Wildcard** 对话框中执行添加操作。



单击 OK 按钮关闭设置对话框。

5、测试

这里有一个测试脚本：



最全 Pycharm 教程（41）——Pycharm 扩展功能之便签注释

1、主题

在开发项目的过程中经常需要创建任务列表，对于一些小任务，在代码中插入便签注释会有奇效。

这种注释的格式如下：

```
#TODO <some text goes here>
```

2、创建便签注释

非常简单：

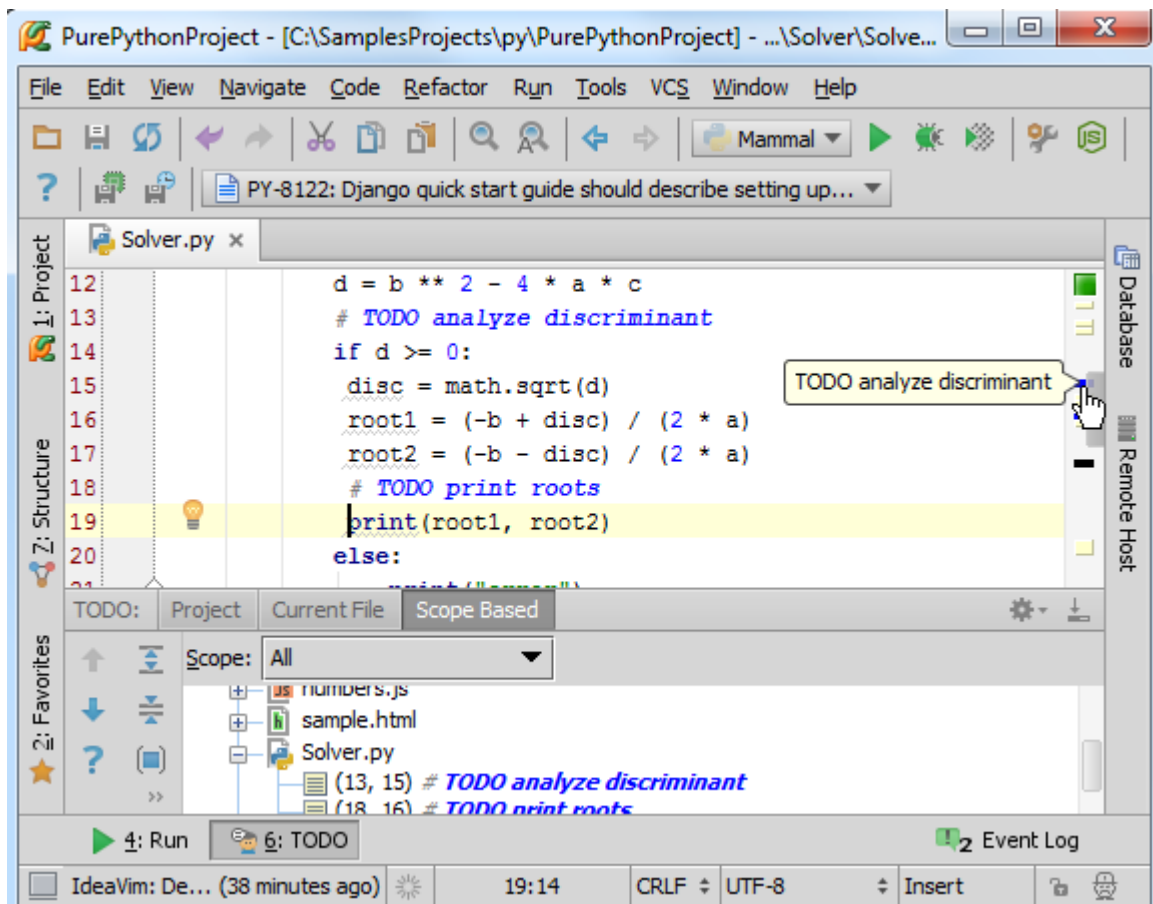
首先，将光标定位到相应行。

然后，创建一行注释（Ctrl+Slash）。

最后，在#后面，输入 **TODO** 或者 **todo**，然后输入注释

3、浏览便签

在 **TODO tool window** 窗口中进行浏览（Alt+6 或者 View→Tool Windows→TODO）：



更多窗口相关信息参见 [Using Tool Windows](#)。

4、便签条目之间的导航

一种方法是通过单击窗口右槽便签标记实现切换。

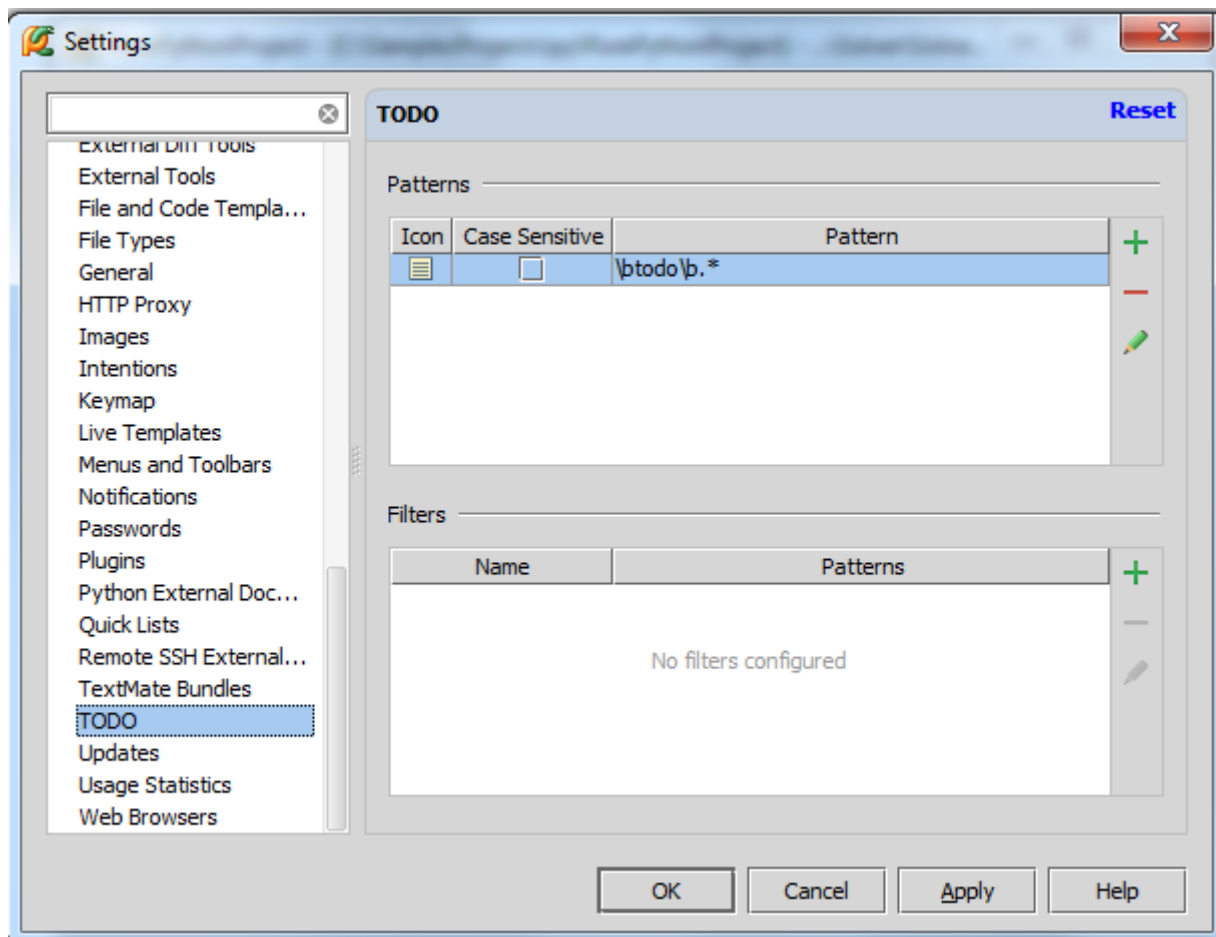
另一种方法是在 **TODO tool window** 窗口中切换。使用窗口中的快捷菜单命令 Jump to Source，或者选中一条便签信息后按 F4。

5、便签模式的私人订制

Pycharm 自带两种便签模式，以 *todo* 开头或者以 *TODO* 开头。只要注释满足以上格式，就会在 TODO tool window 窗口中显示。

这里我们介绍如何创建其他格式的便签。

单击主工具栏的设置按钮，在 IDE Settings 节点下，单击 TODO 打开 [TODO page](#) 页：

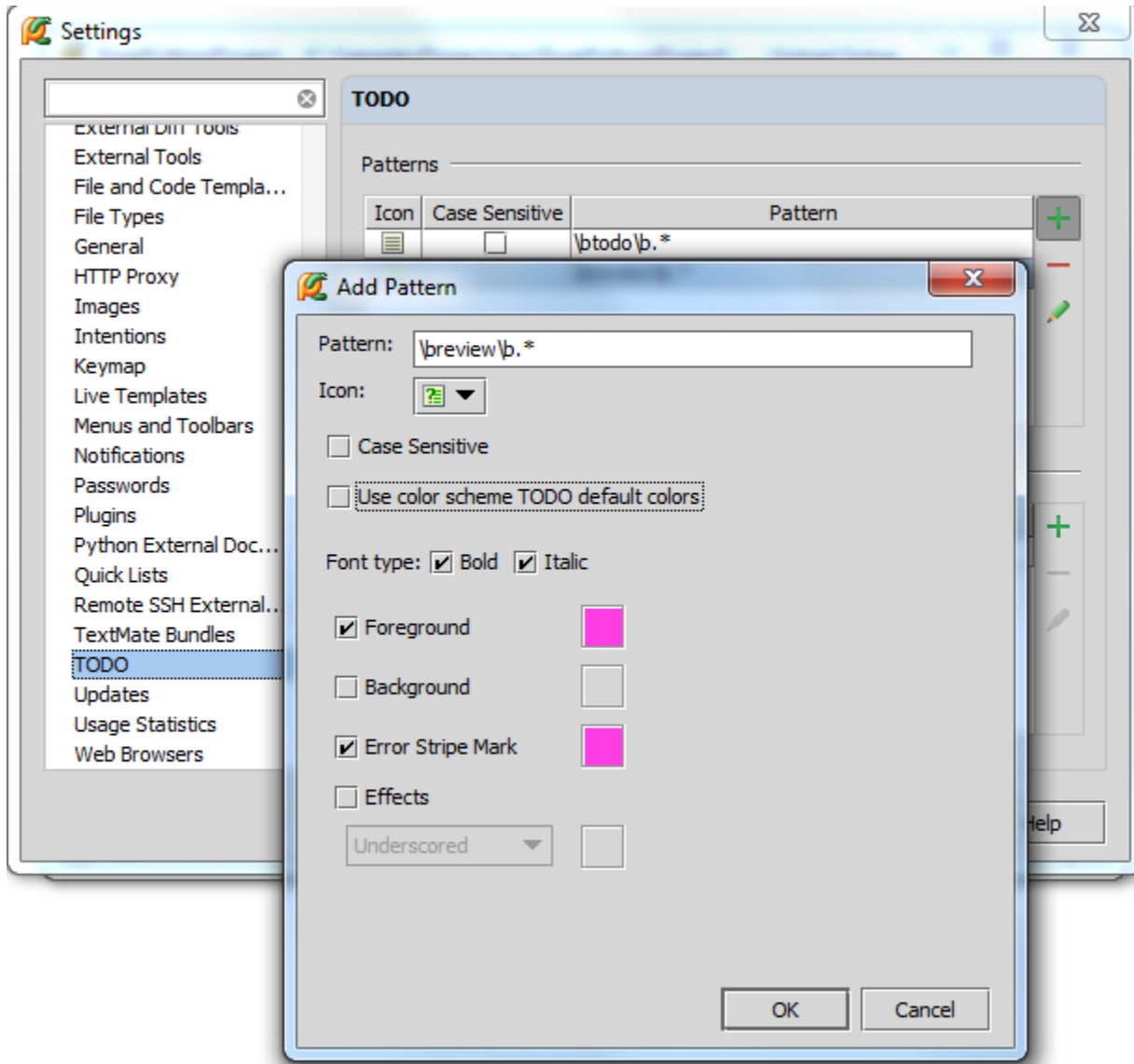


假设我们希望在特定行创建代码浏览的相关便签，包含“review”关键字，接下来创建这种便签模式。

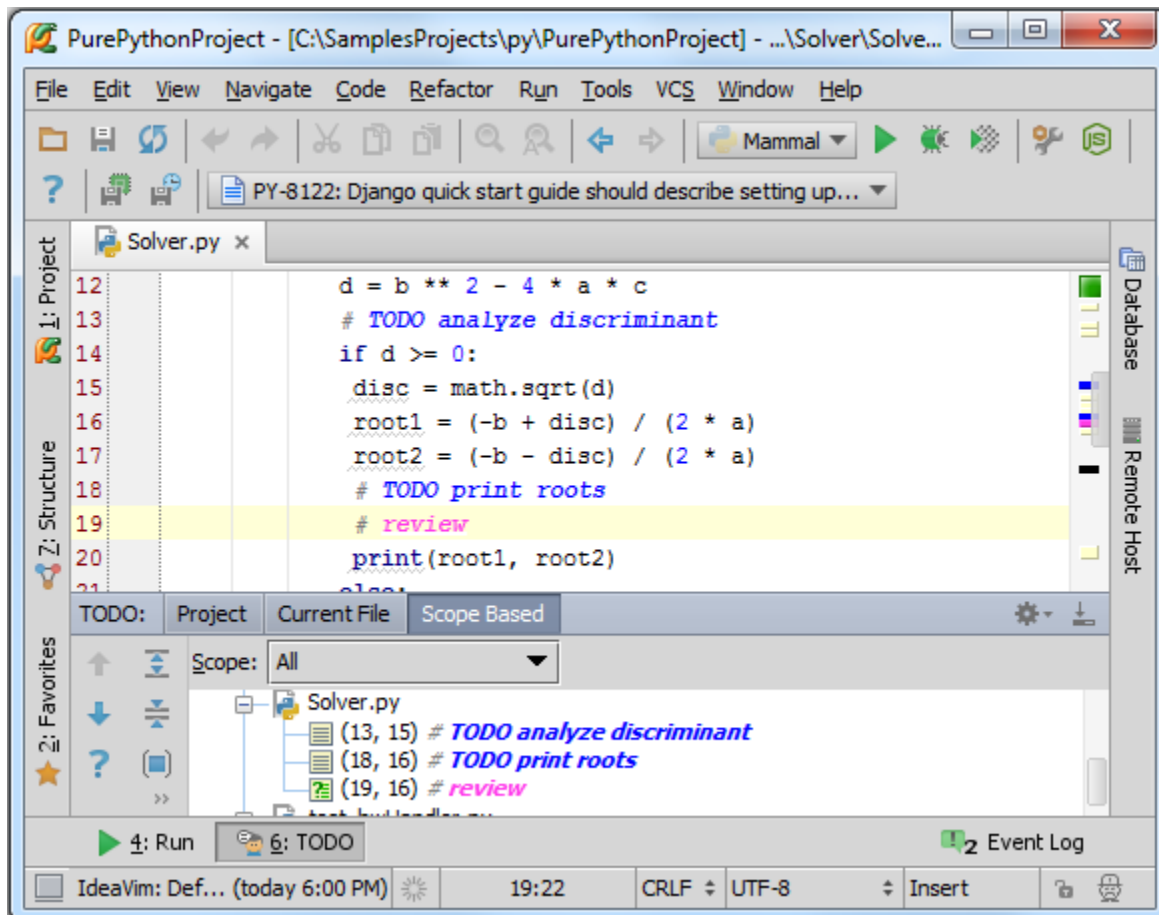
在 **Patterns** 部分单击绿色加号，输入如下正则表达式：

```
\\breview\b.*
```

同时更改图标和配色方案：



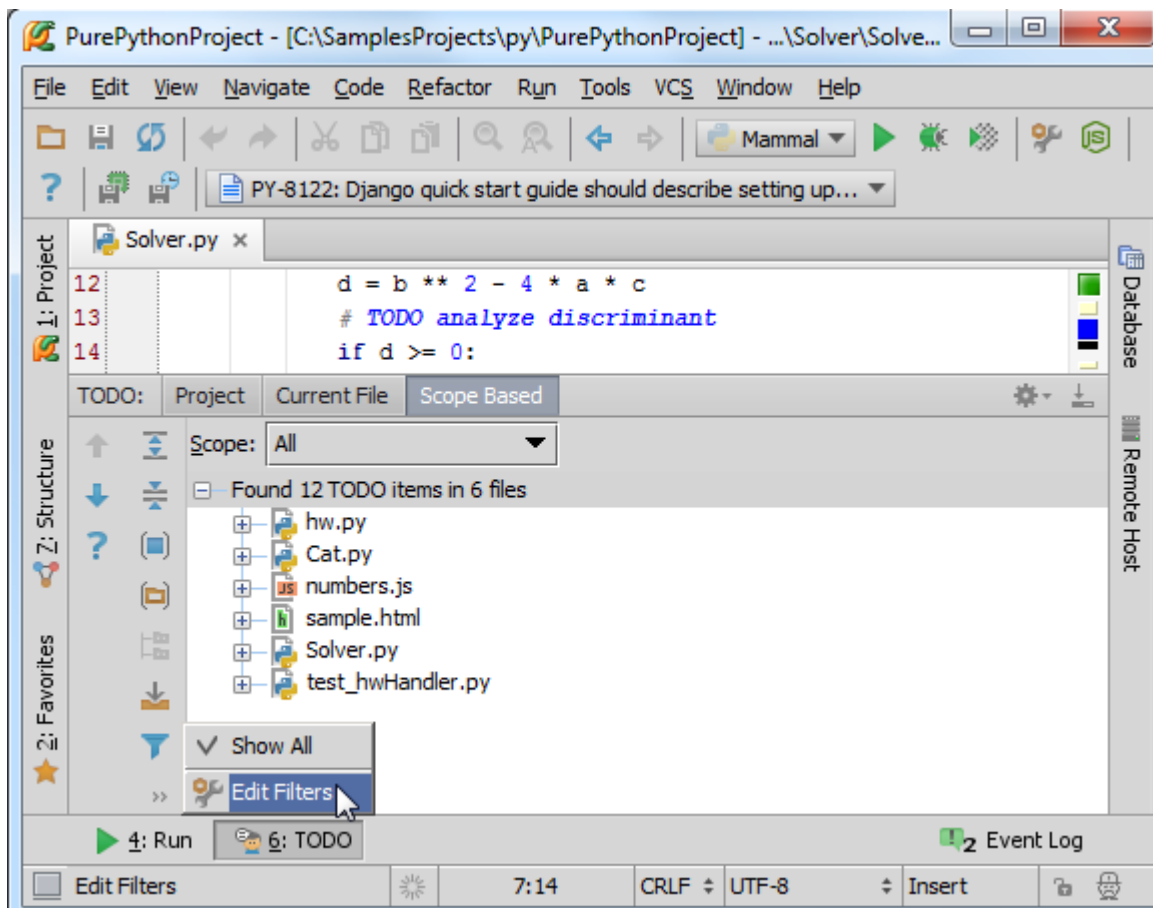
设置完成后 Pycharm 会自动检测当前工程中的所有注释，若发现符合条件的便签注释，则会以新的配色方案在 TODO tool window 窗口中显示：



6、创建便签过滤器

假设我们只希望看到与代码浏览相关的便签信息，将其他便签隐藏，这里需要用到便签过滤器。

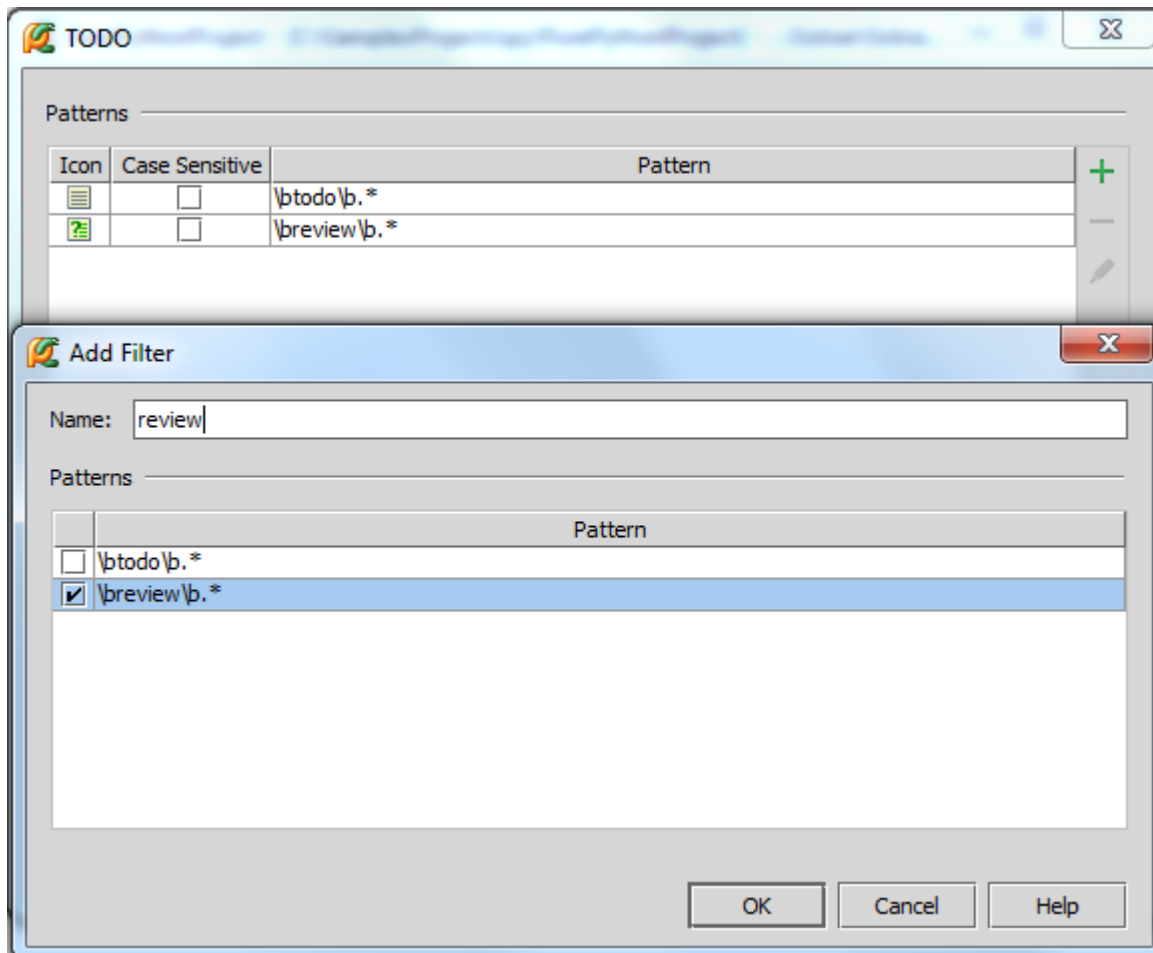
单击 [TODO tool window](#) 的  图标，选择 Edit Filters：



当然也可以通过设置对话框中的 **TODO page** 页完成相关操作（设置→IDE Settings→TODO）。

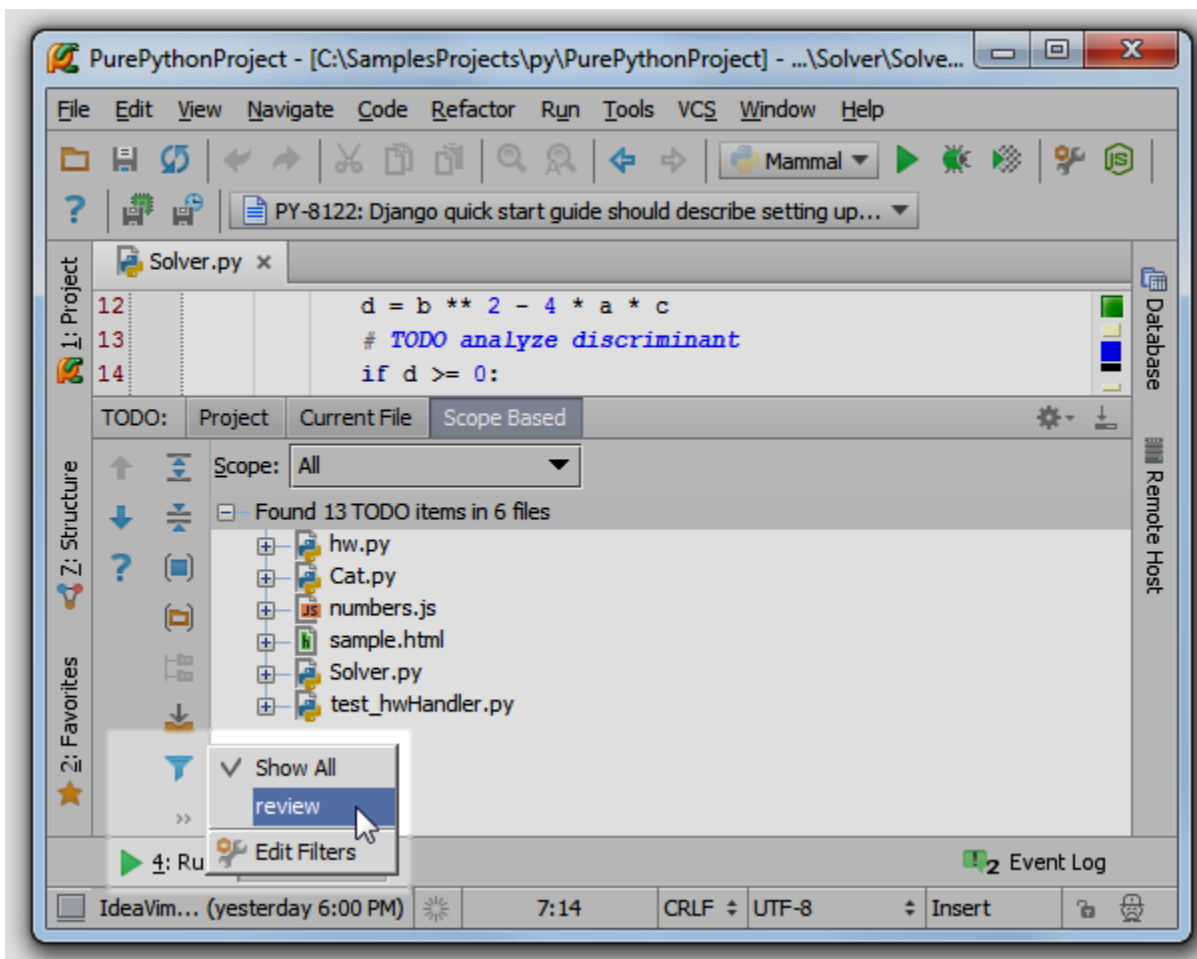
在 **Filters** 区域单击绿色加号。

在打开的 **Add Filter** 对话框中显示两种模式 `\btodo\b.*` 和 `\breview\b.*`。这里我们选择隐藏传统的模式，只保留 `\breview\b/*` 模式。选中这个模式，输入过滤器名称：

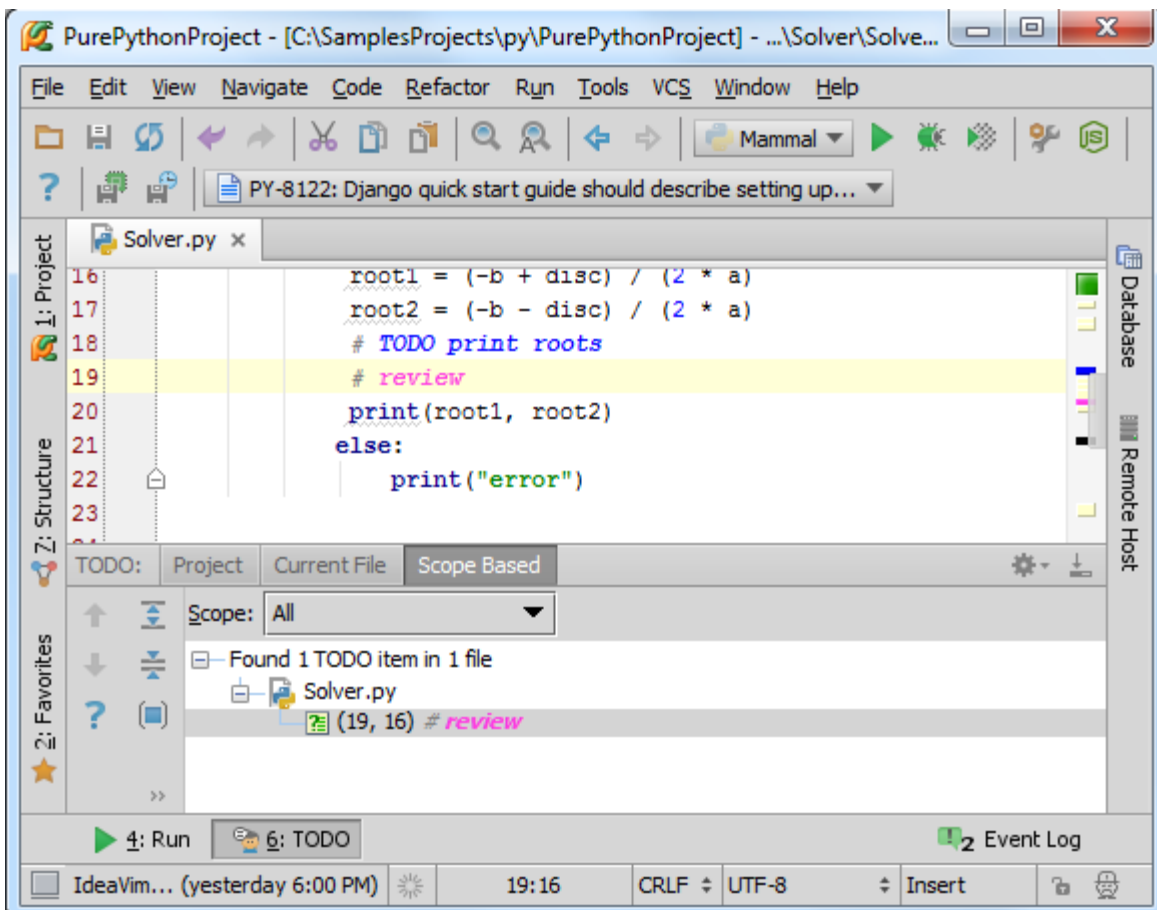


7、过滤便签

返回 TODO tool window 窗口，单击 ：



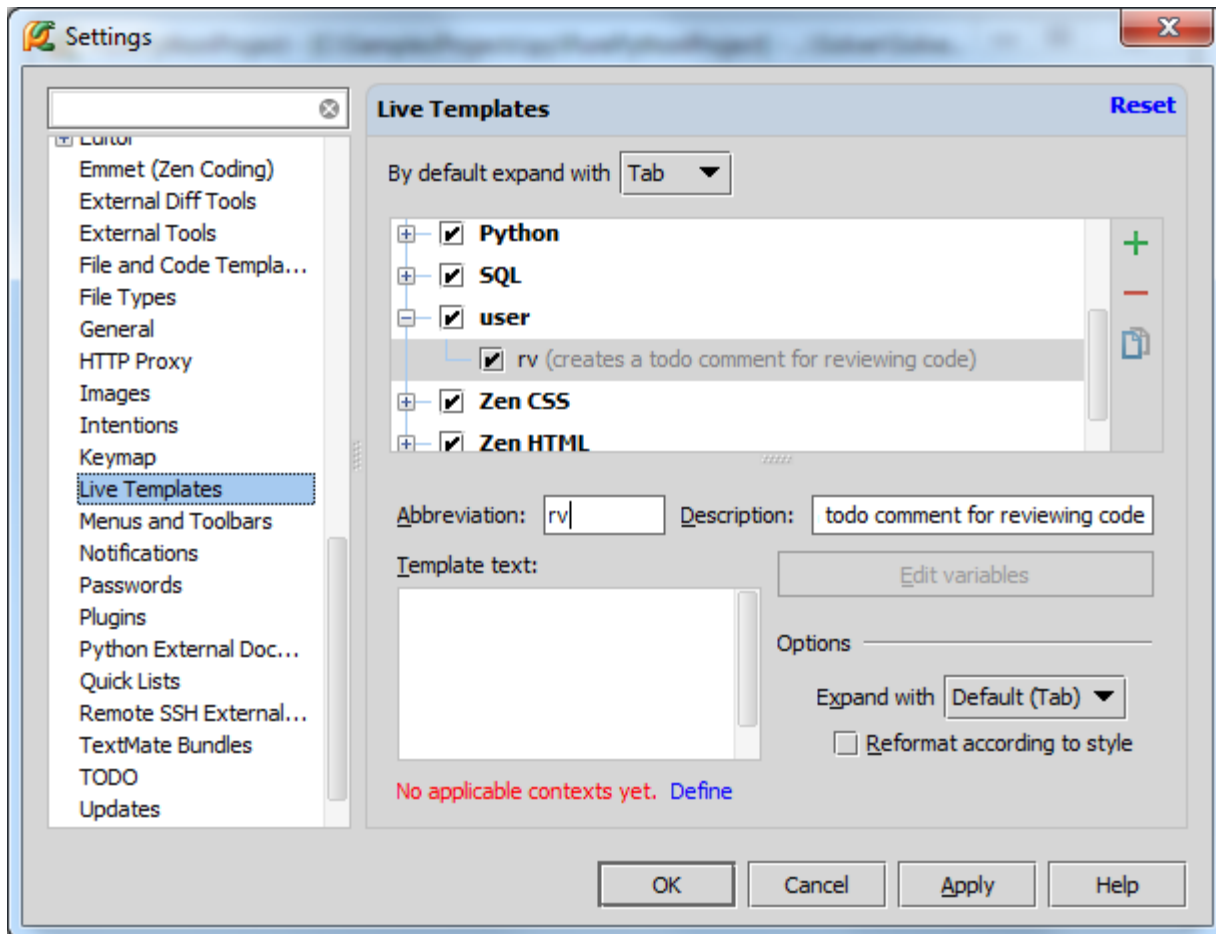
选择 review 命令，在 tool window 窗口中显示过滤结果，双击对应条目可实现快速跳转：



8、为便签注释创建模板

创建模板有助于快速编写便签信息。

打开设置对话框，在 **IDE Settings** 节点下，单击 **Live Templates**：设置→IDE Settings→Live Templates：单击绿色加号添加模板，输入相关信息：



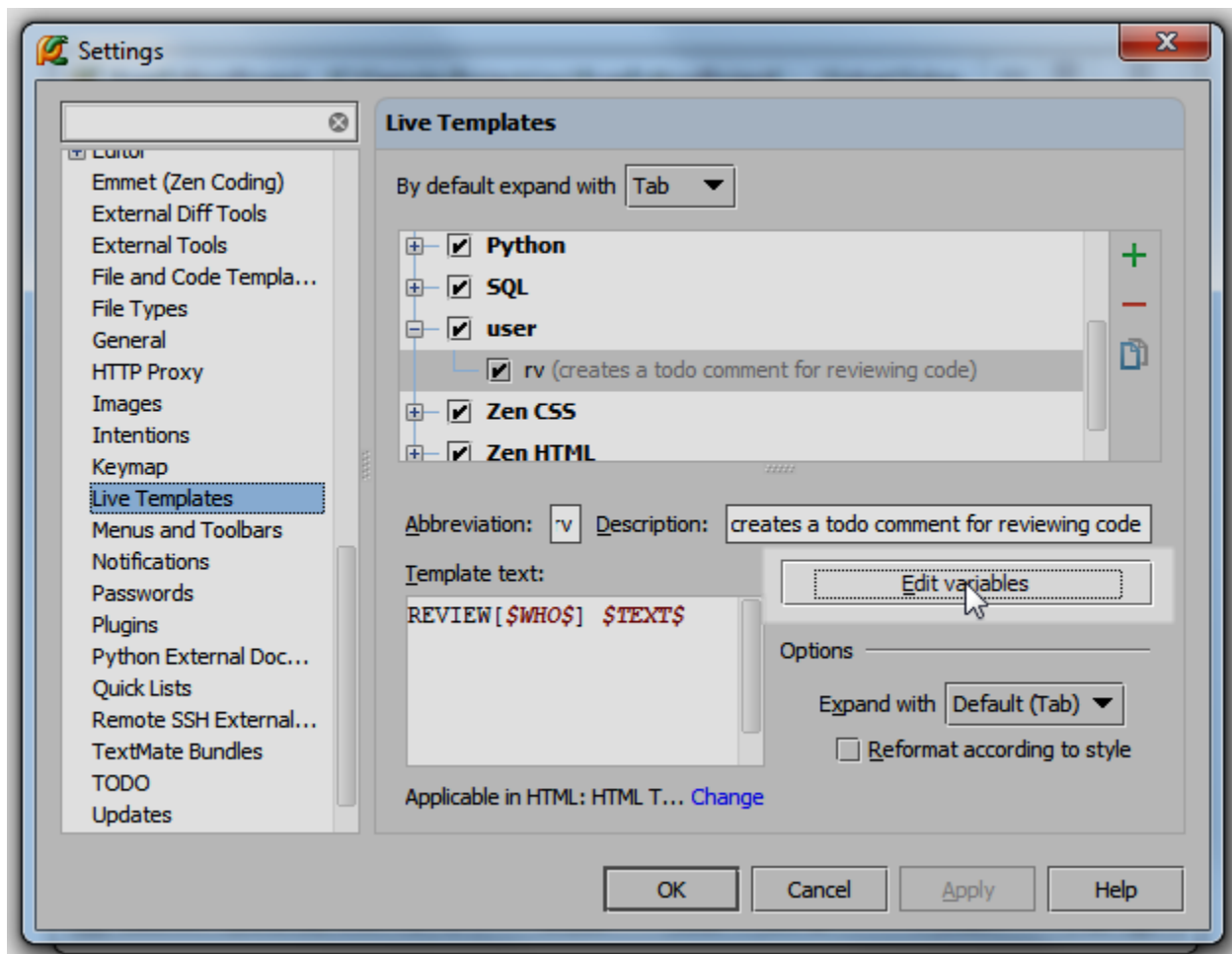
注意到新建的模板会自动添加到用户组。

注意底部的红色提示，单击 **Define**，查看模板内容的可选信息。

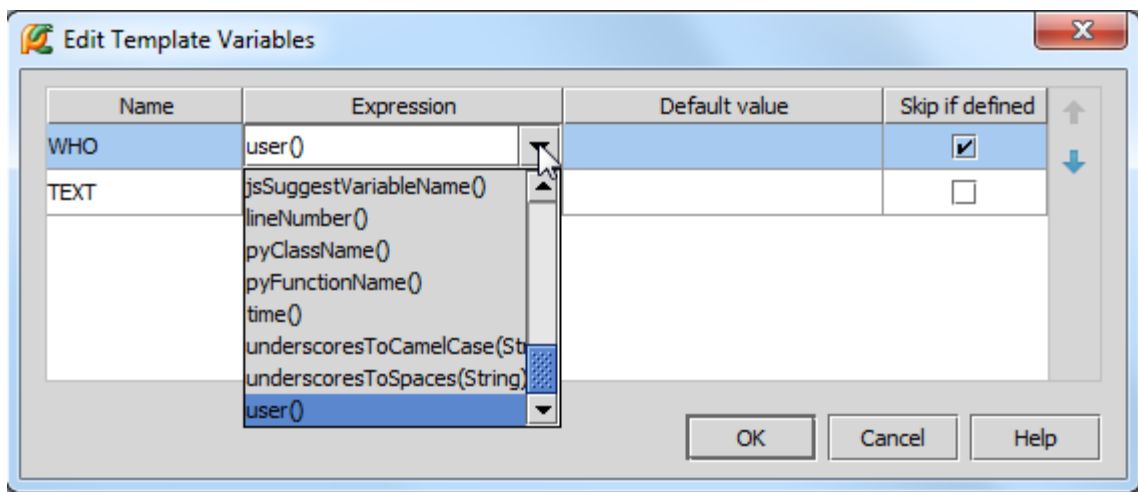
最后，定义模板体：

```
REVIEW[$WHOS] $TEXT$
```

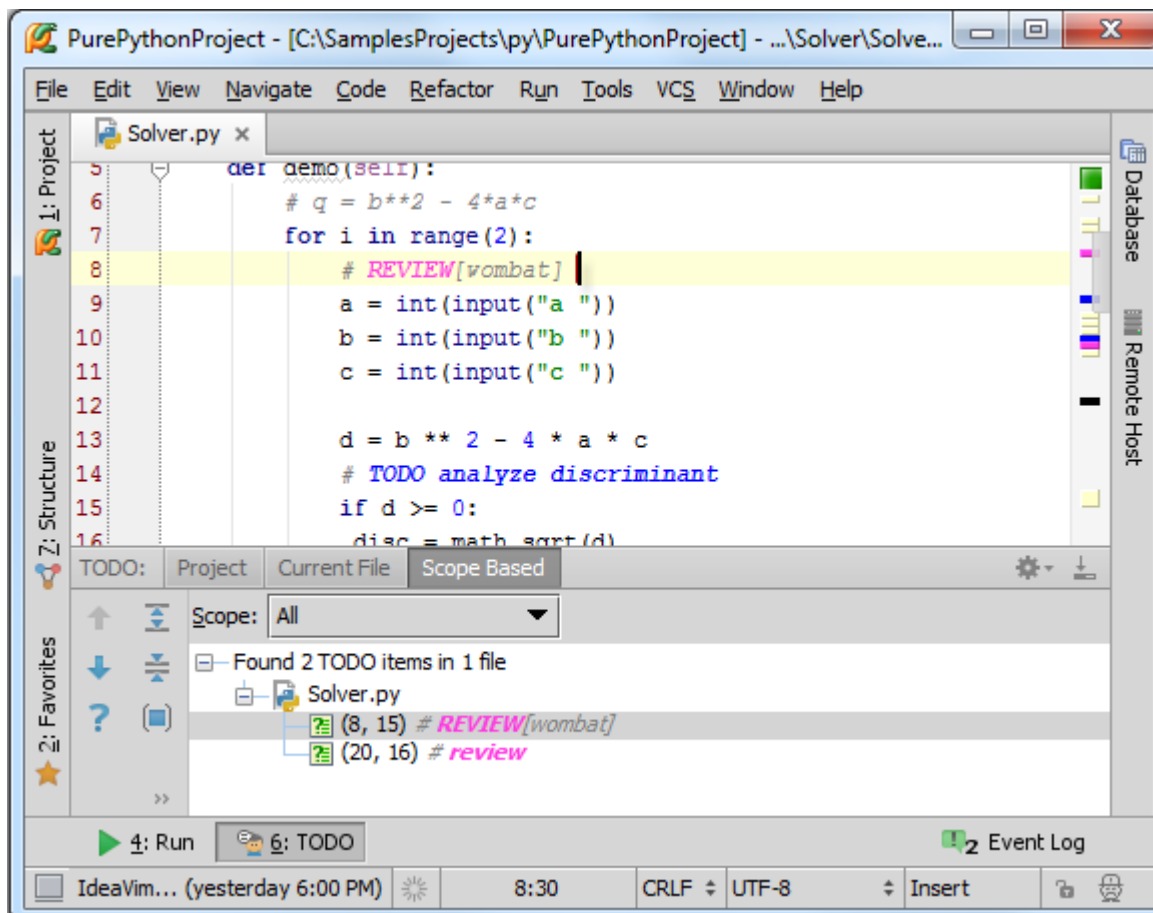
这里有两个变量\$WHOS\$以及\$TEXT\$。前者为输入区域，后者自动填充。单击 **Edit variables** 按钮：



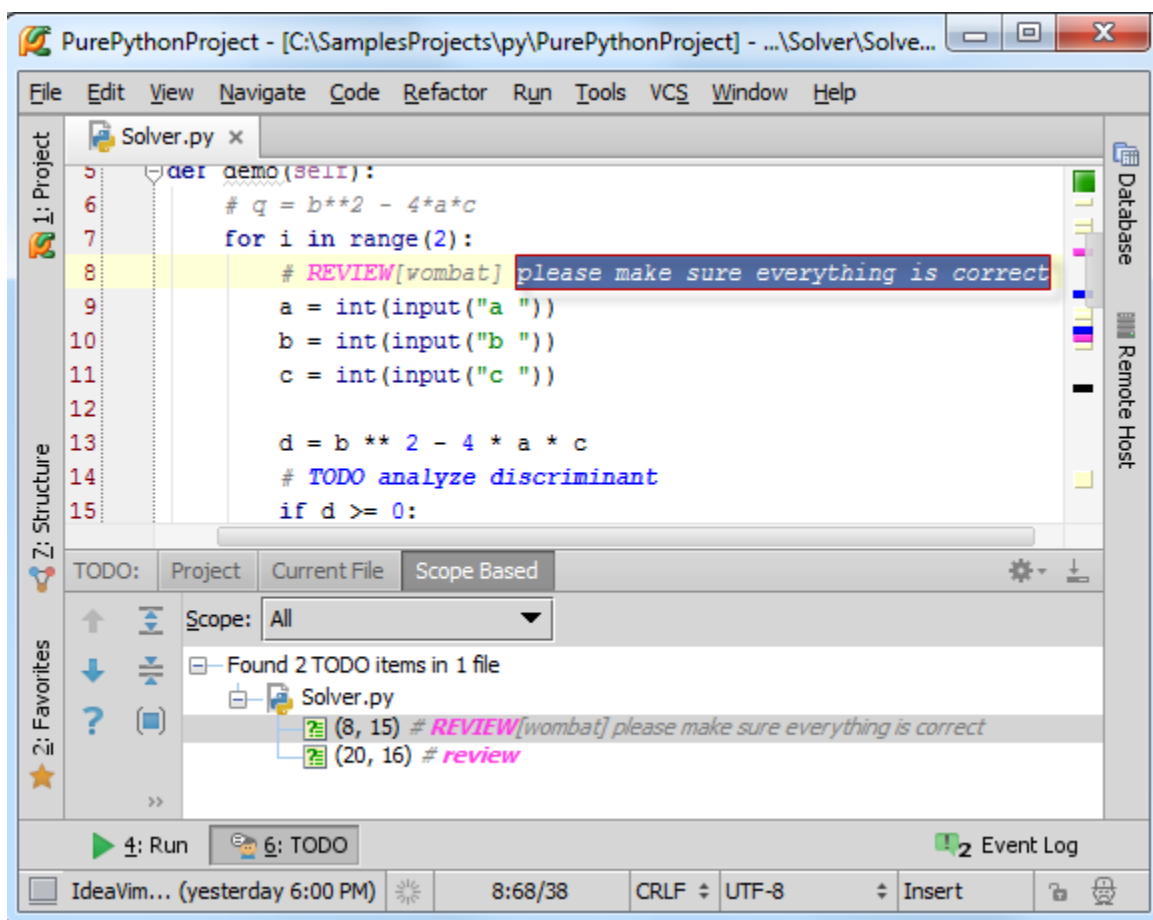
在 [Edit Template Variables dialog box](#) 对话框，选择变量\$WHO\$的内容：



测试一下模板。创建一个注释行，输入 rv，按下 TAB：



输入文本，回车：



模板生效。

最全 Pycharm 教程（42）——Pycharm 扩展功能之 Emacs 外部编辑器

1、主题

介绍如何将 Emacs 定义为一个 Pycharm 外部编辑器。

2、准备工作

- （1）Pycharm 版本为 2.7 或更高
- （2）下载了 [downloaded](#) Emacs 并正确安装

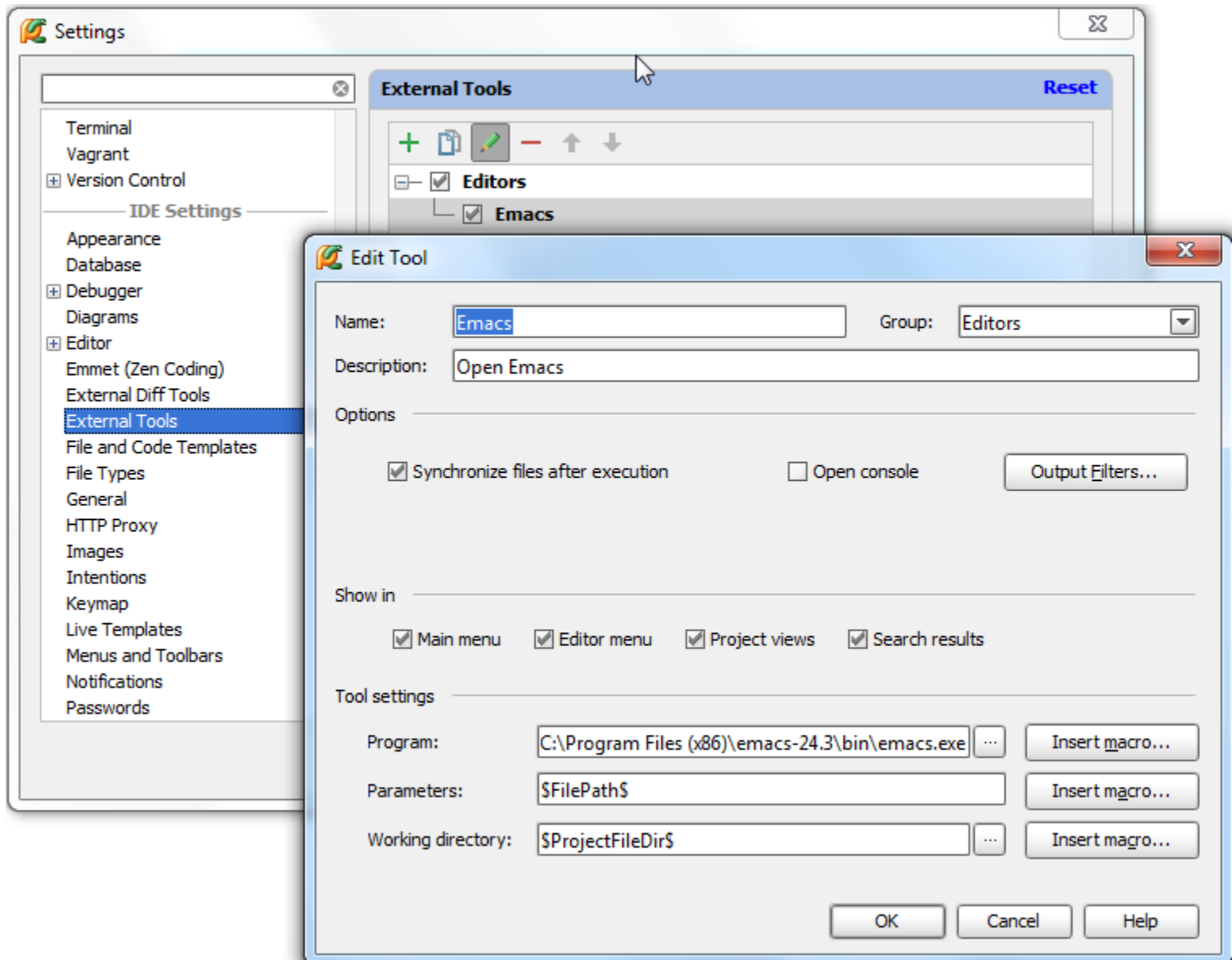
3、配置 Emacs

打开设置对话框，在 IDE settings 节点下，打开 [External tools](#) 页，确认 Emacs 的安装情况。

首先，在 [External tools](#) 页，单击绿色加号，打开 [Create/Edit tool dialog box](#) 对话框：

做以下工作：

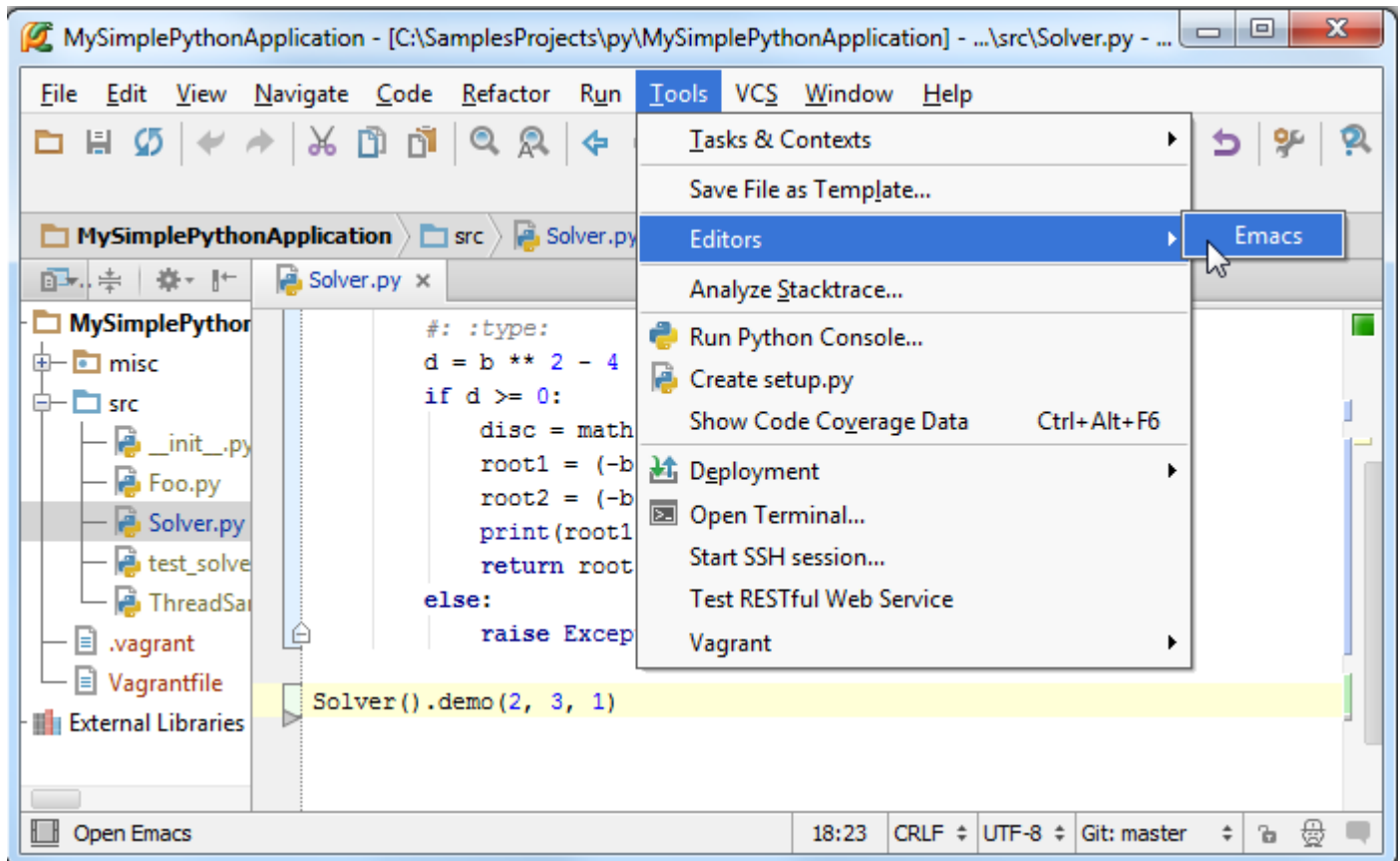
- （1）输入工具名（Emacs）、简要描述（open Emacs）
- （2）指定出现的菜单栏工具组。这里放到 Editors 菜单组中，也可以放到主菜单栏
- （3）取消 Open console 勾选
- （4）定义 Emacs 二进制文件位置，可以手动输入也可以浏览选择
- （5）在 Parameters 栏输入默认打开的文件路径 \$FilePath\$
- （6）指定工作目录（例如 \$ProjectFileDir\$）
- （7）单击 OK



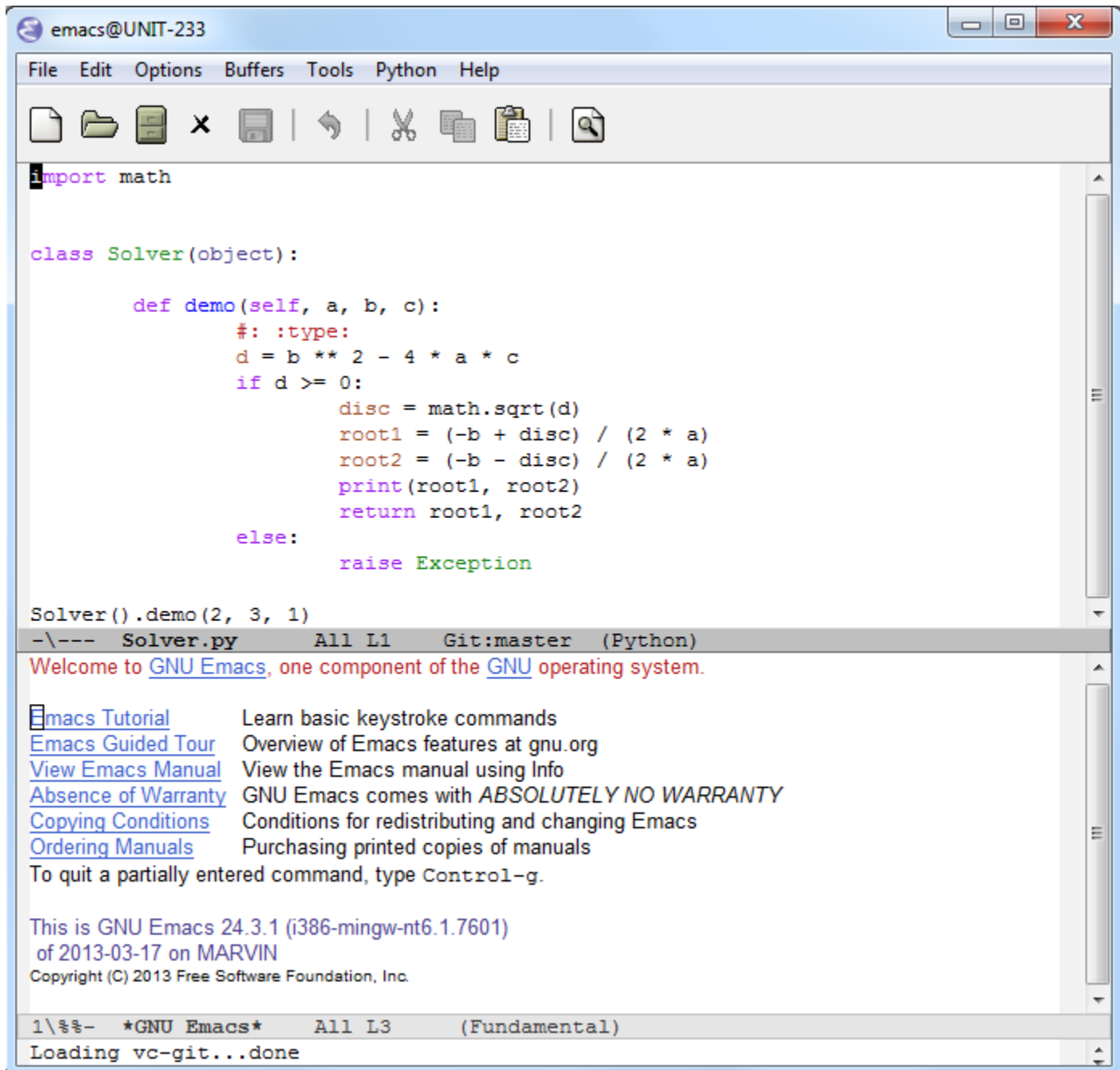
设置生效。

4、在 Emacs 中打开当前文件

单击 Tools 菜单栏中的 Editors 节点中的 Emacs 命令：

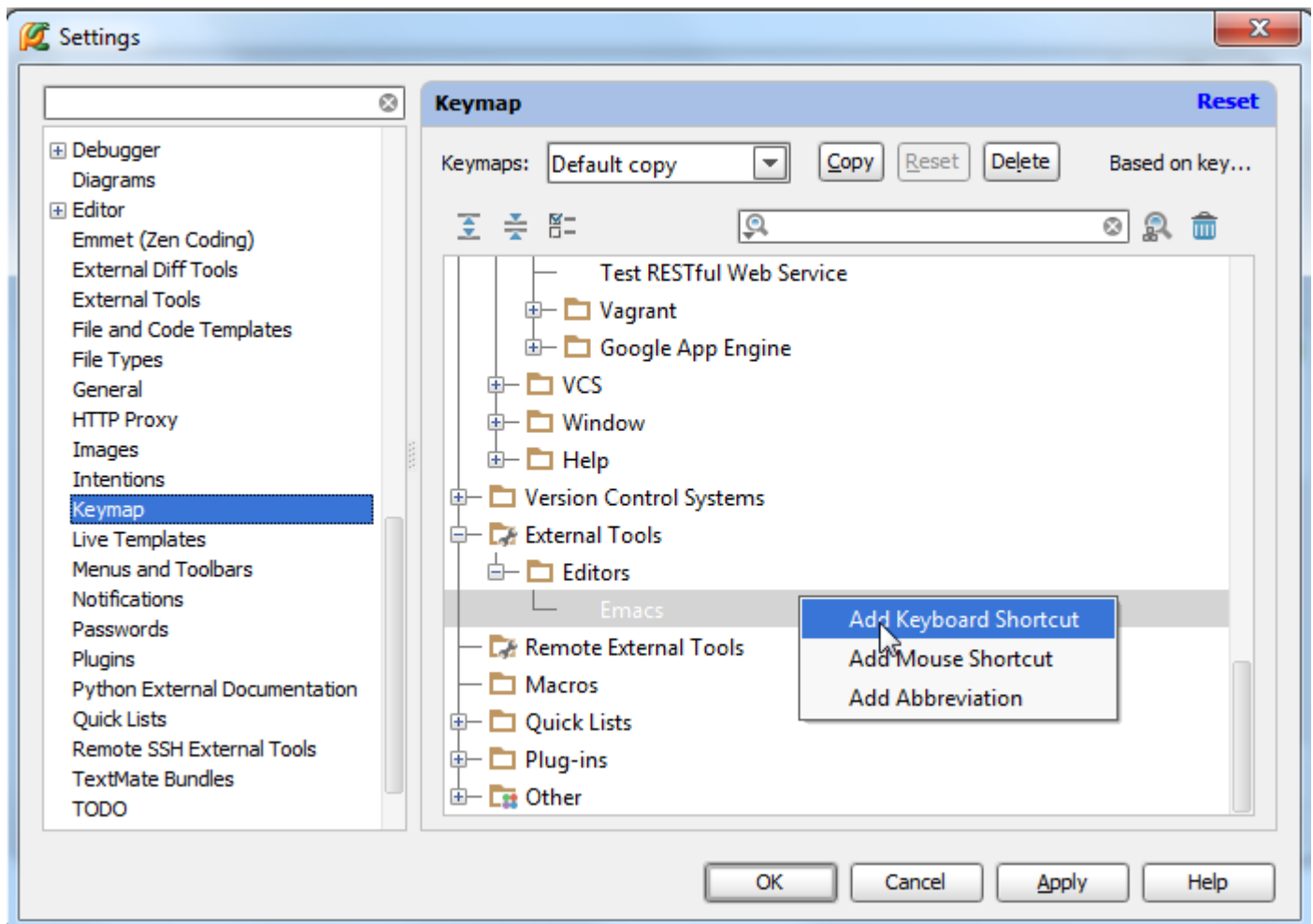


打开 [open a file in the PyCharm editor](#) 对话框，在 Tools 菜单，选择 Editors→Emacs：

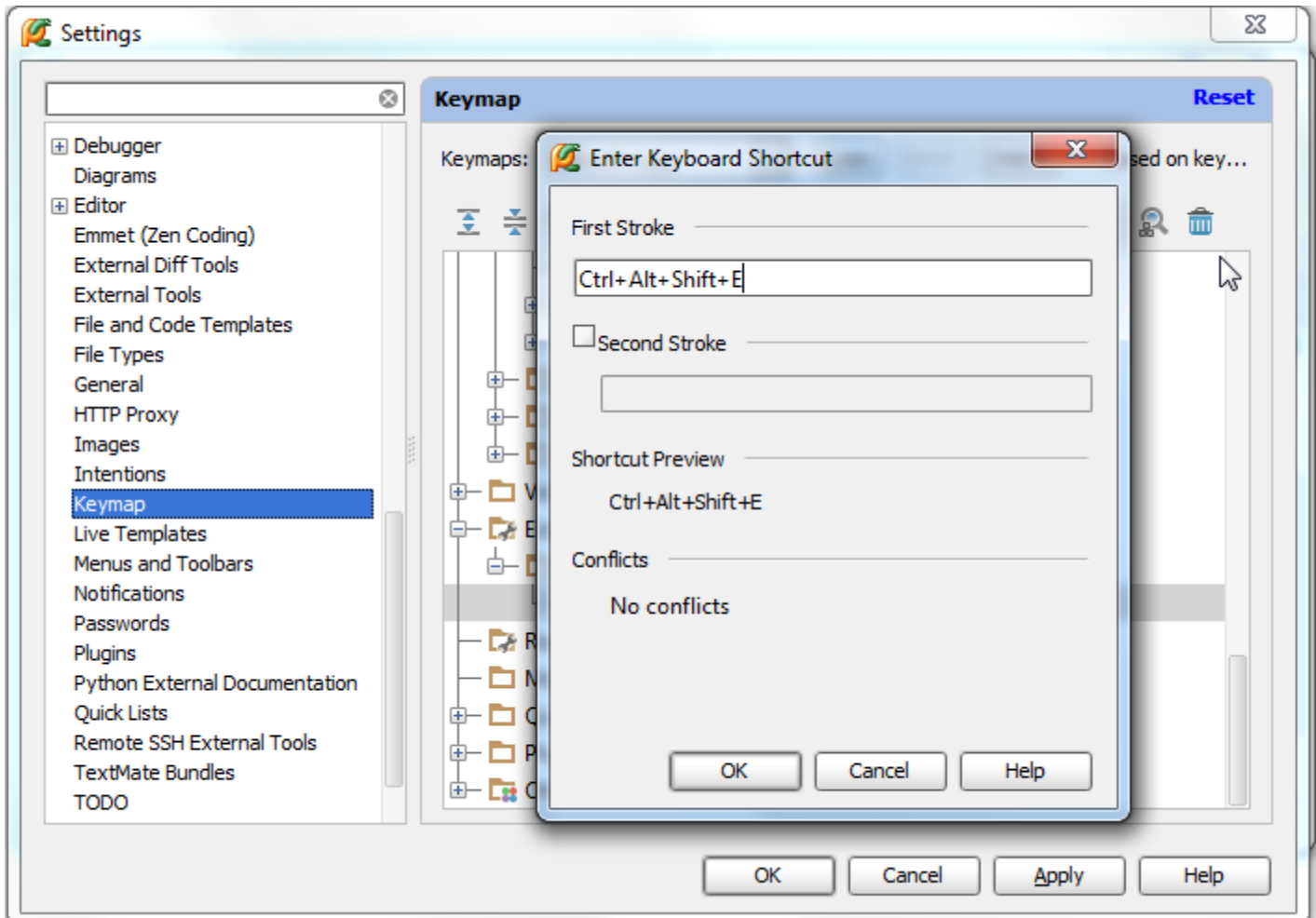


5、指定快捷键

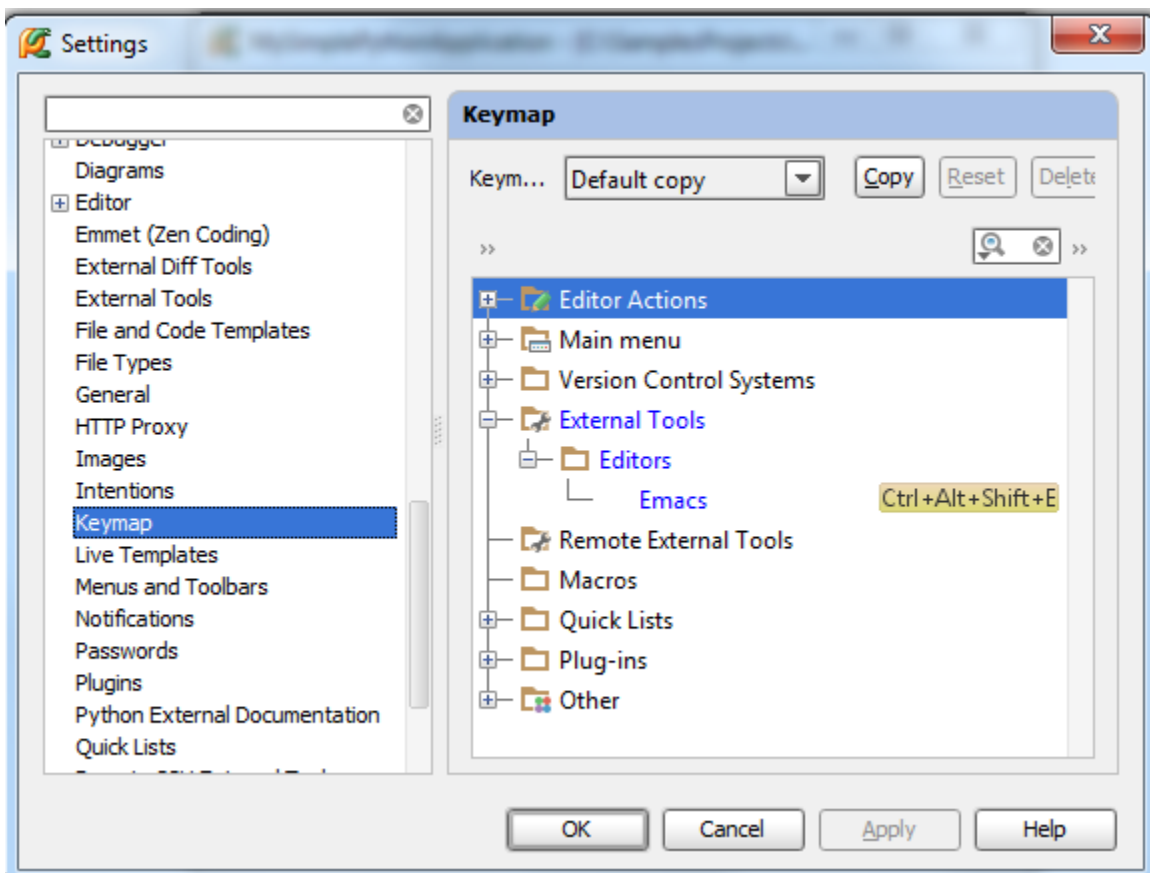
打开设置对话框，在 [Keymap](#) 页的下拉列表中找到 External Tools 节点，展开 Editors，右击 Emacs 节点，选择 Add keyboard shortcut:



打开 [Enter keyboard shortcut](#) 对话框，这里输入 Ctrl+Alt+Shift+E 组合：



没有提示冲突，单击 OK 按钮，快捷键设置完成：



最全 Pycharm 教程（43）——Pycharm 扩展功能之 UML 类图使用

1、什么是 UML

UML 类图能够快速检查代码结构。

2、主题

UML 类图在 Pycharm 中的用法。

3、准备工作

- (1) Pycharm 版本为 2.7 或更高
- (2) 安装 Python 解释器
- (3) UML 插件以及 UML 类图插件安装并能正常使用

4、准备一个例子

创建一个 **Animals** 模块以及 Mammal.py 文件（Alt+Insert→Python Package, Python File），输入以下代码：

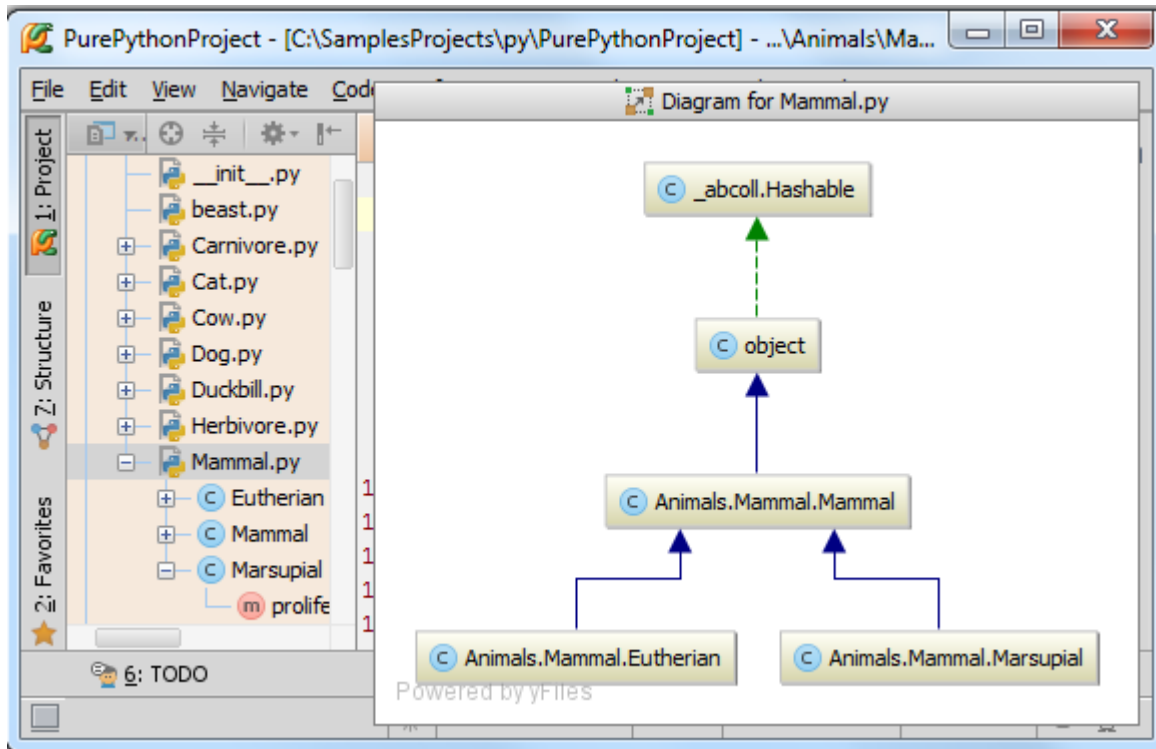
```
class Mammal(object):
    extremities = 4
    def feeds(self):
        print ("milk")
    def proliferates(self):
        pass
class Marsupial(Mammal):
    def proliferates(self):
        print("poach")
class Eutherian(Mammal):
    def proliferates(self):
        print("placenta")
```

继续向其中添加更多类，例如 **Carnivore** 和 **Herbivore**。

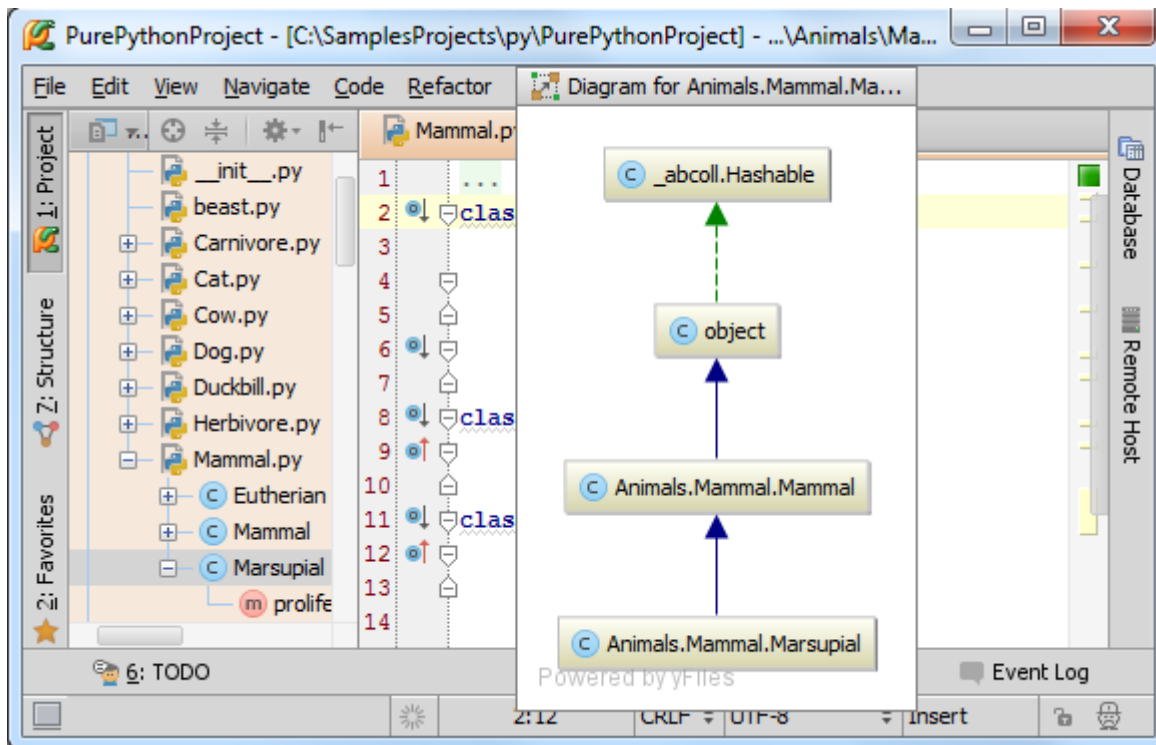
5、通过 UML 类图浏览程序结构

假设我们想查看 mammals 的派生结构, 在 Project tool window 窗口中, 右击 **Mammal.py**, 指向快捷菜单中的 **Diagrams** 节点, 选择如下可用命令:



- (1) Show Diagram: 在当前编辑选项卡内打开 UML 类图
- (2) Show Diagram Popup: 在单独窗口中显示 UML 类图

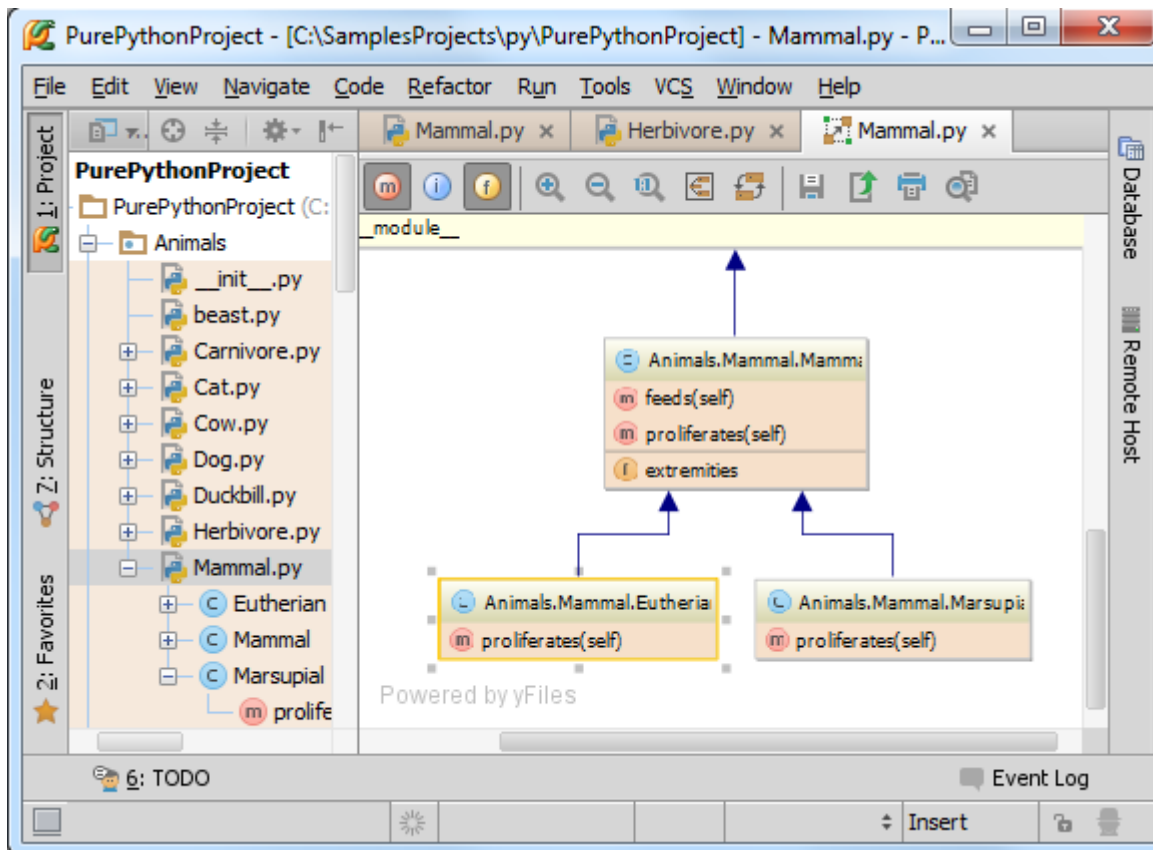


选择 mammals 中的一个类（例如 marsupials），用 UML 类图查看它：

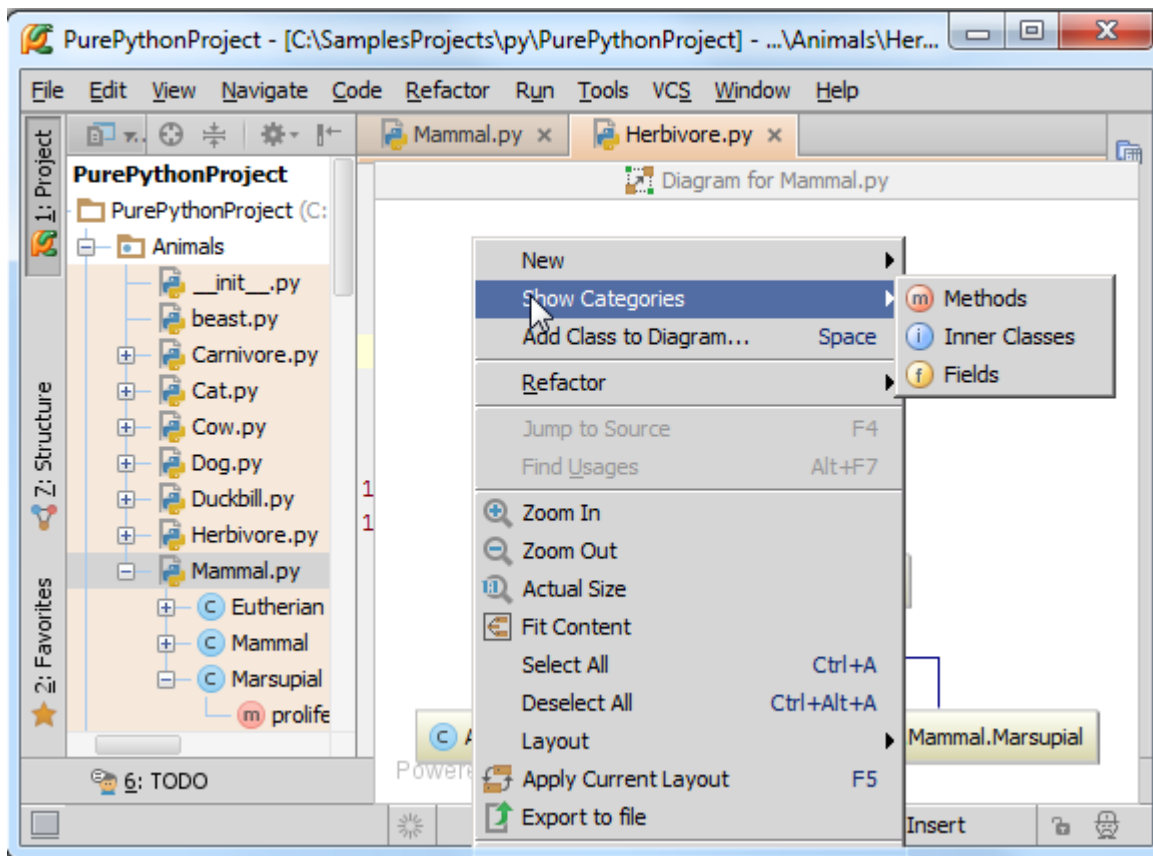


也可以使用快捷键 `Ctrl+Alt+U`（当前窗口）或 `Ctrl+Alt+Shift+ U`（独立窗口）。

单击  和  按钮查看详细信息：

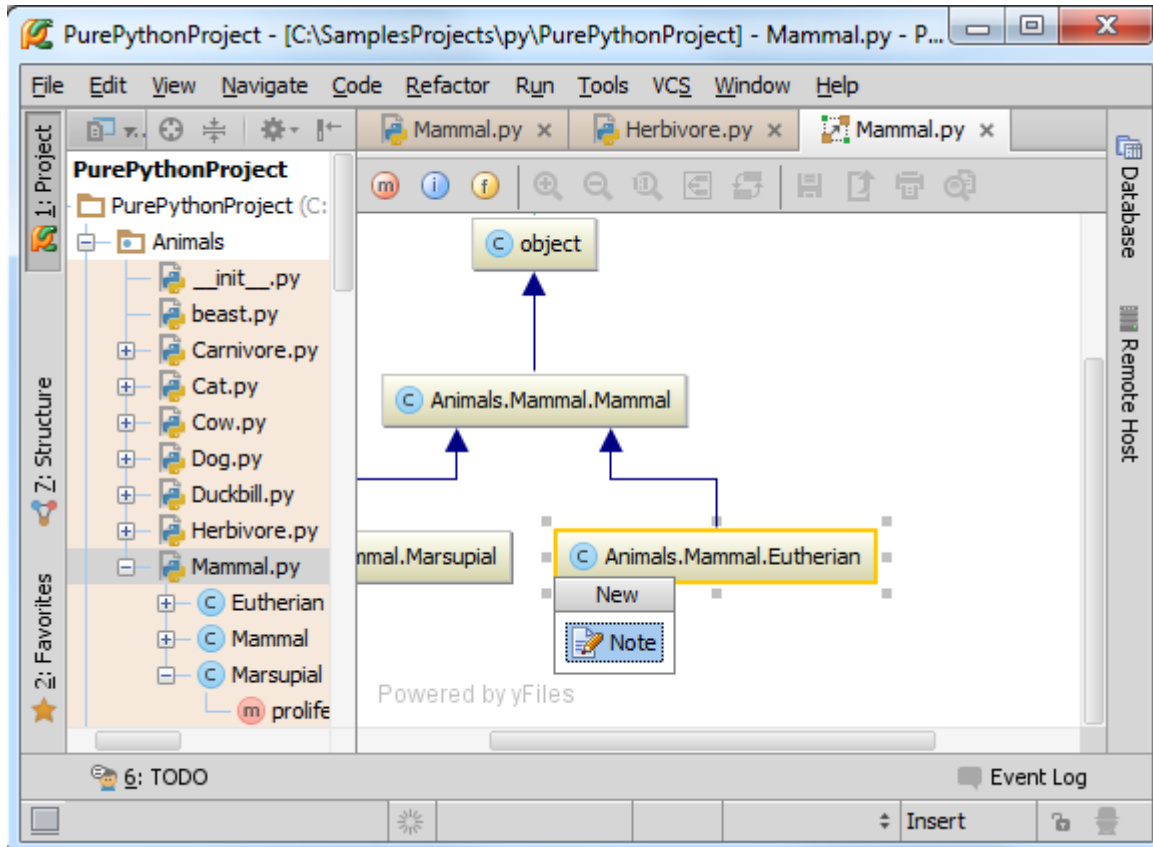


也可以使用快捷菜单：

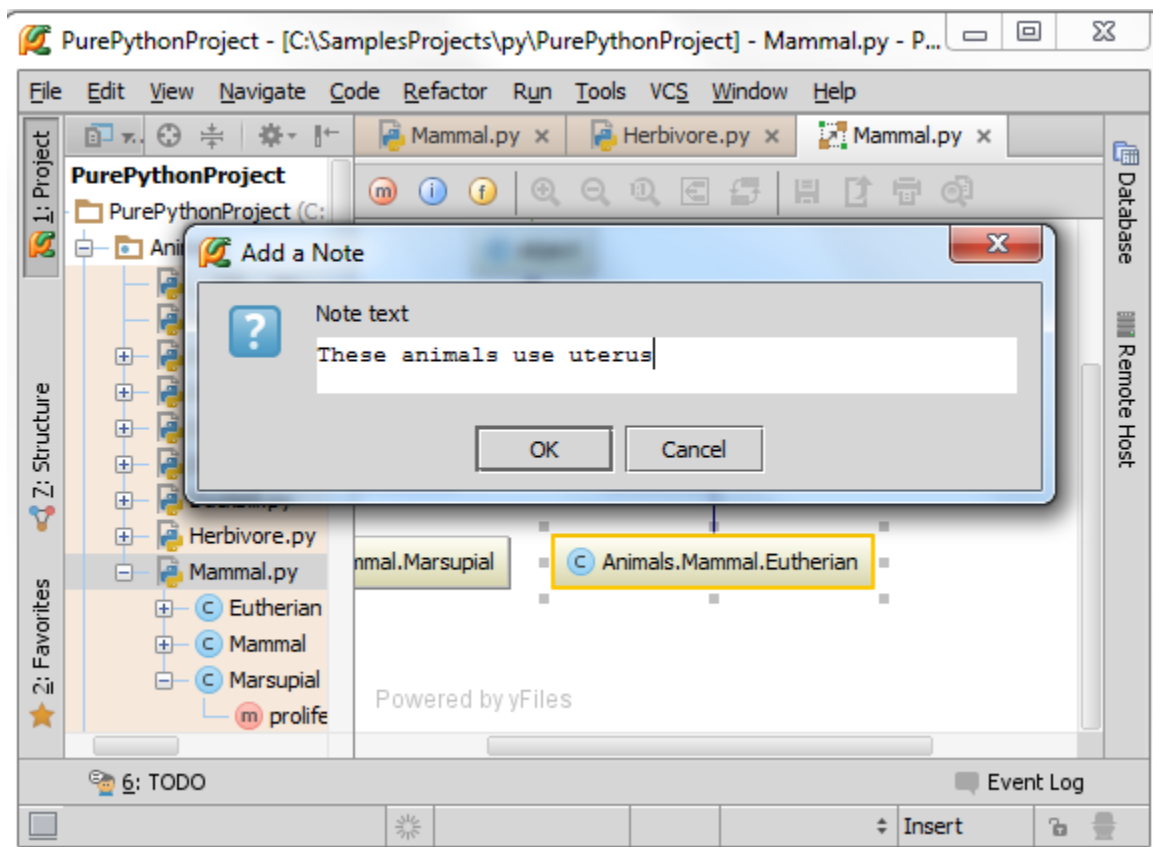


更多工具栏及快捷菜单的功能参见 [Class Diagram Toolbar and Context Menu](#) 页。

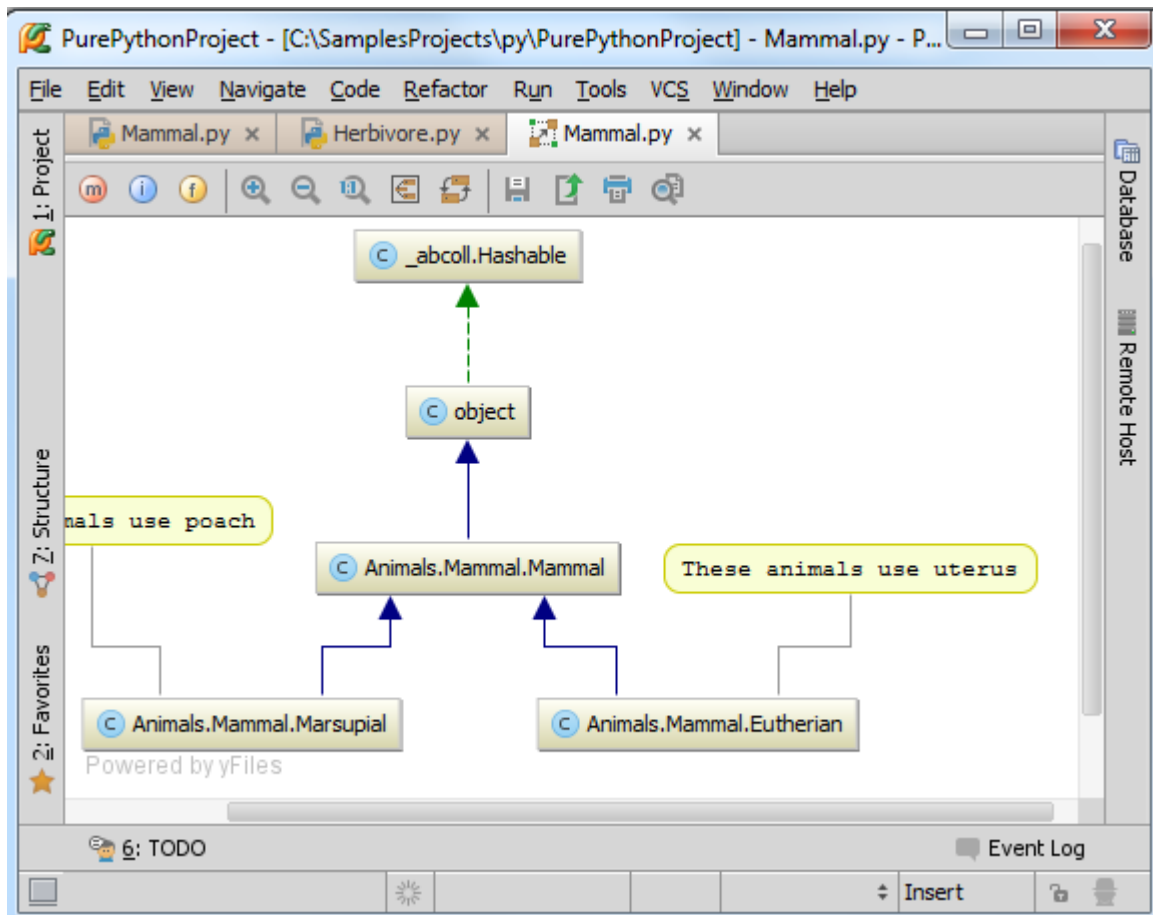
接下来向其中加入注释。选择想要注释的元素，按下 Alt+Insert：



回车，输入注释内容：



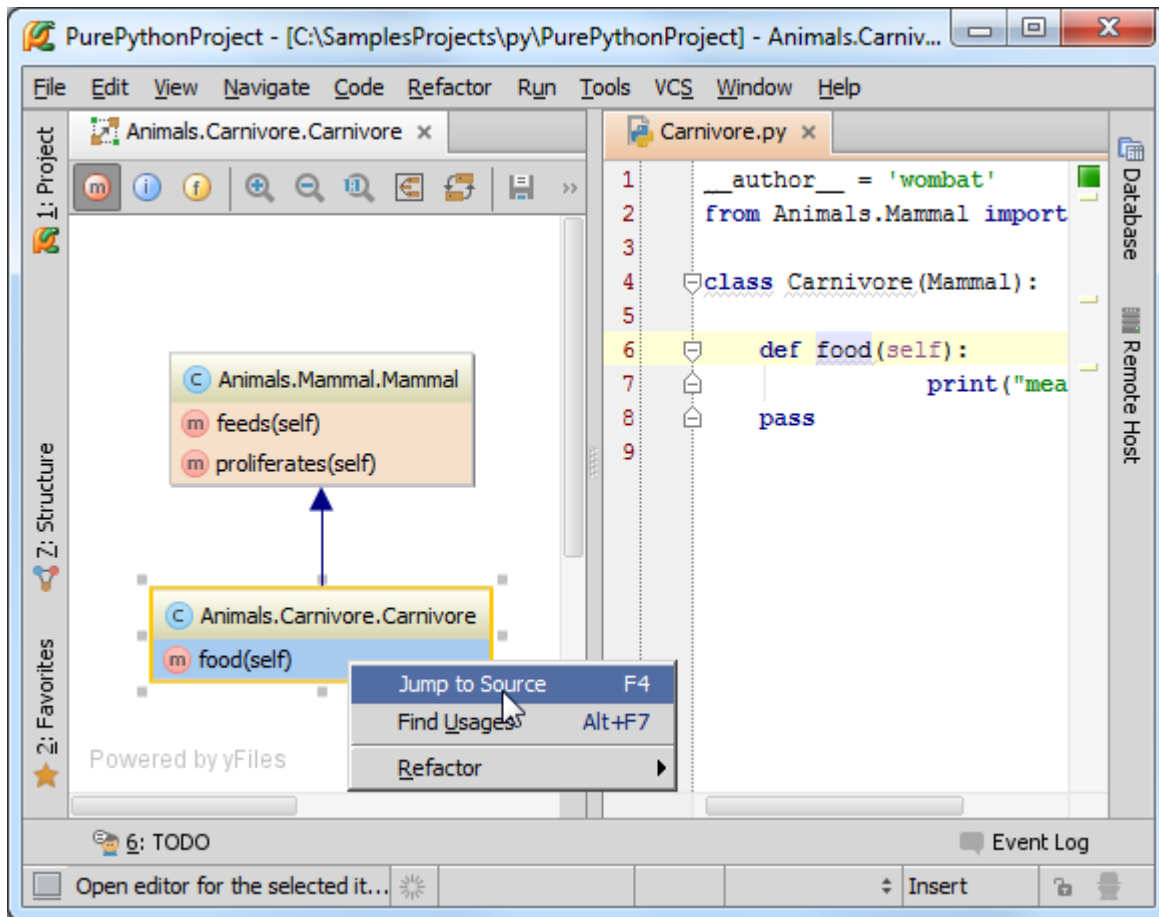
单击 OK，注释添加完毕：



6、结构图与源码间导航

如何从结构图跳转到响应的源码位置？

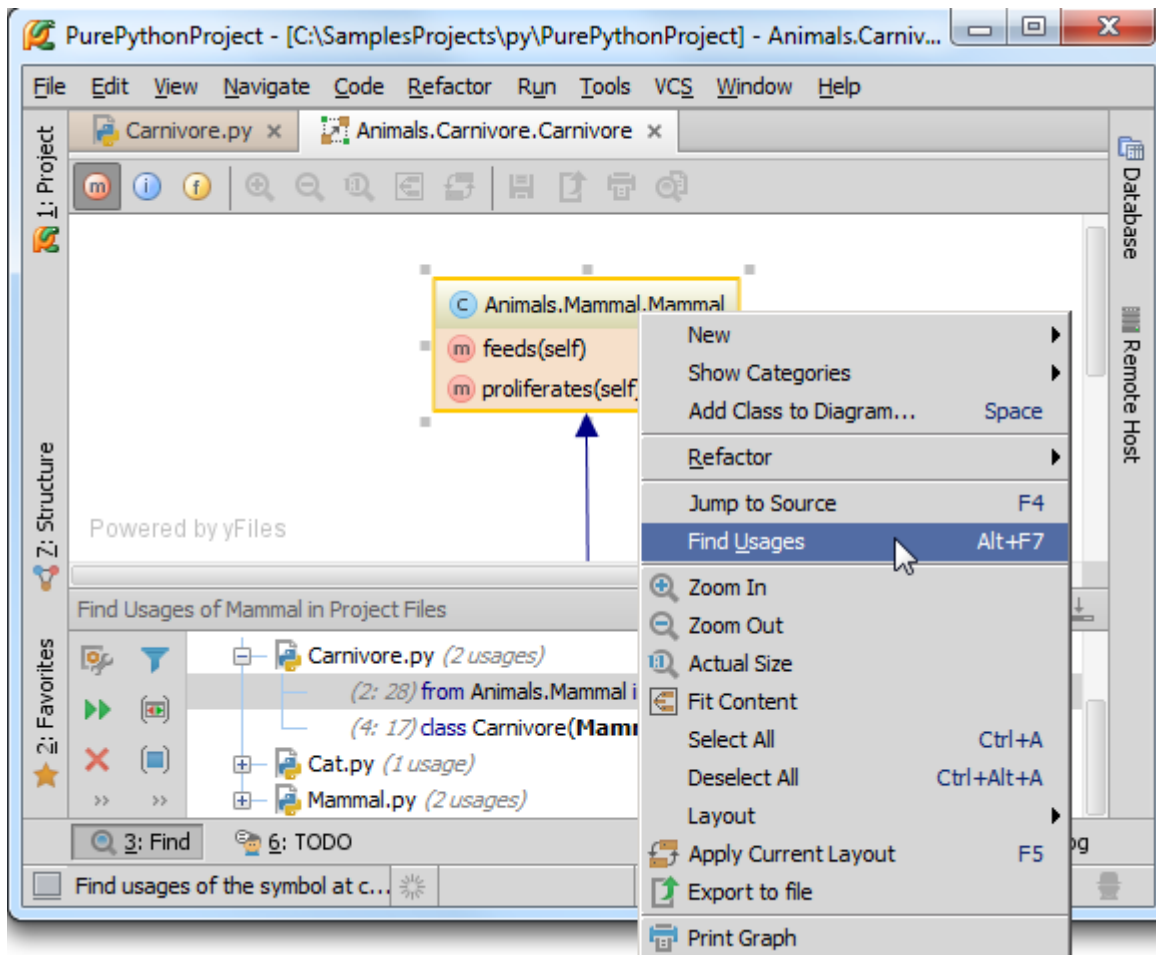
在结构图中选择一个节点元素，按下 F4 或者右键快捷菜单选择 Jump to Source 命令：



7、UML 结构图作用

8、查找所有引用

在结构图中选择一个节点元素，右击，在快捷菜单中选择 Find Usages，或者使用 Alt+F7 快捷键：

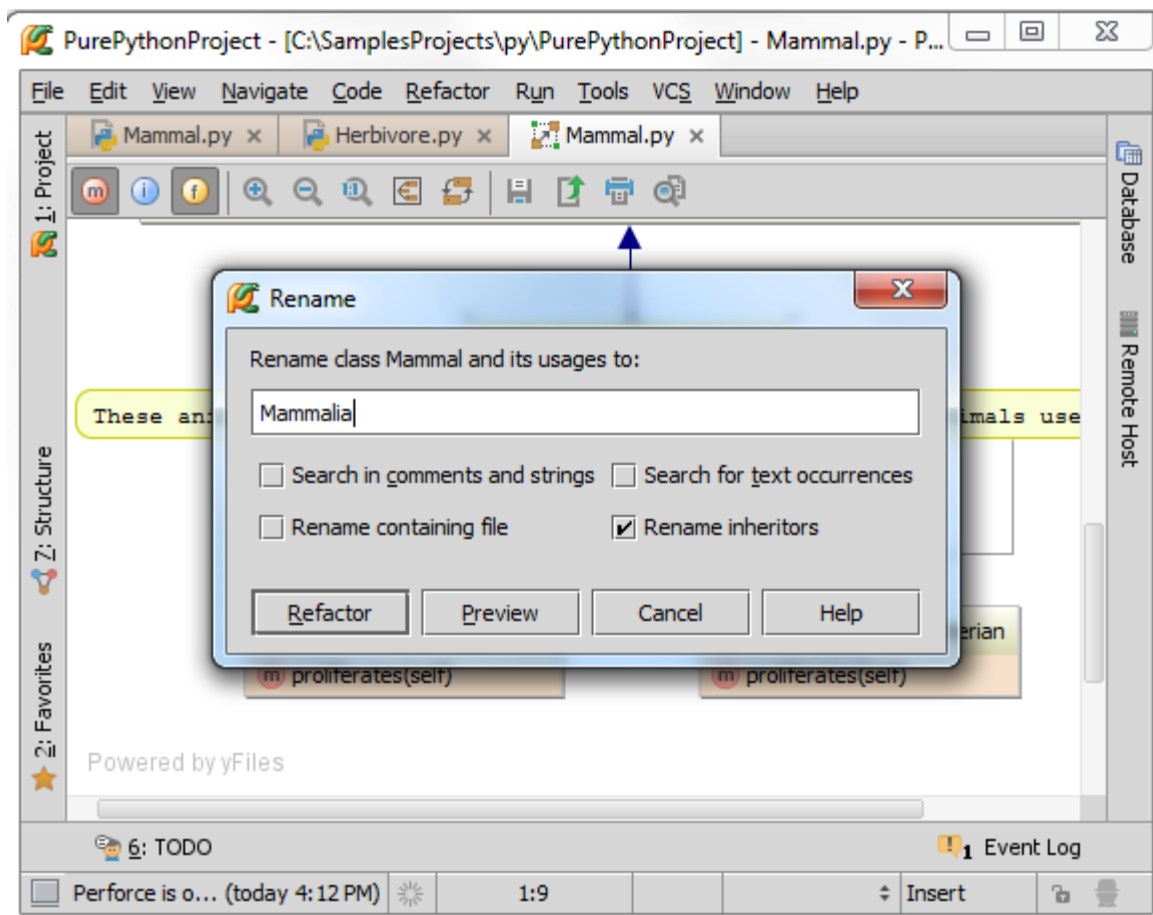


9、重构

可以直接在类图中对类或者成员就行重构（重命名等）。

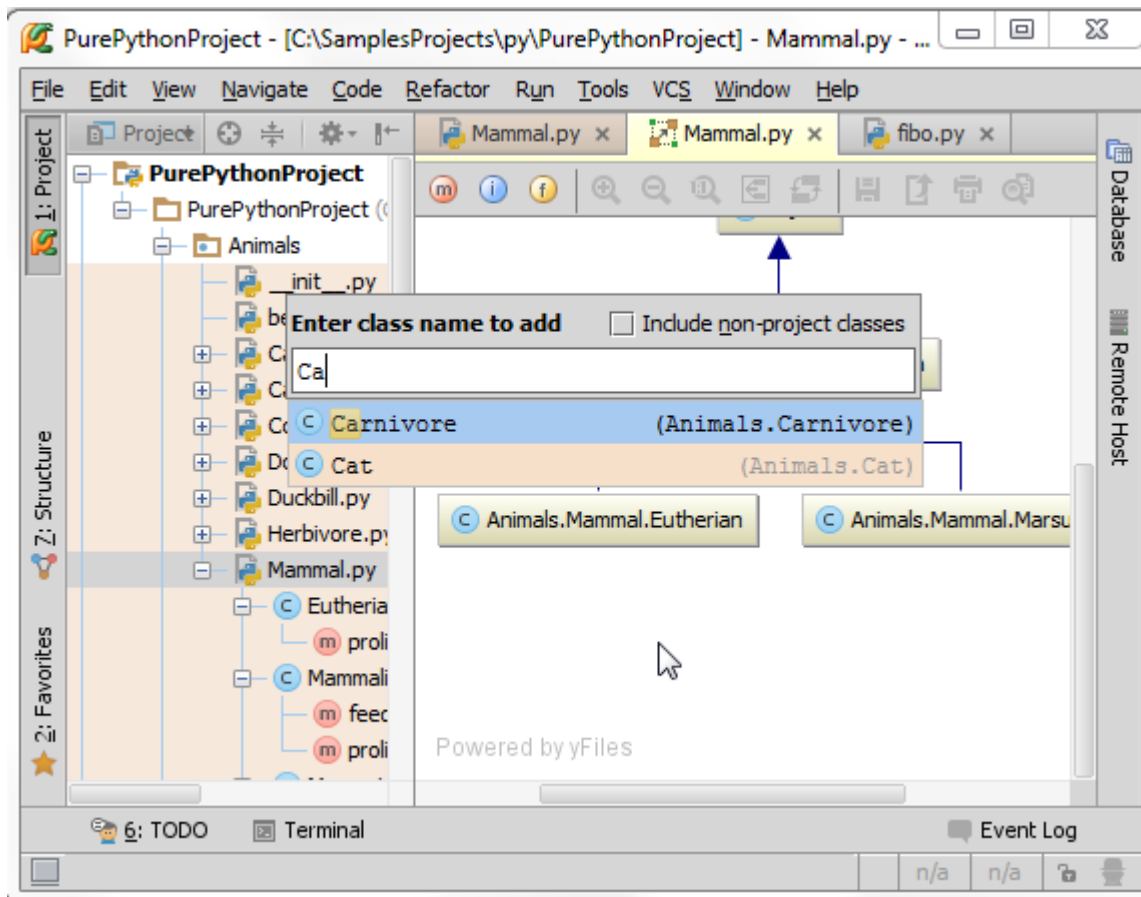
在类图中右击待修改的节点，指向 Refactor 命令，在子菜单中选择对应命令。

例如你希望对一个类进行重命名，需要在类图中选中它，然后选择 Refactor→Rename 快捷菜单命令或者按下 Shift+F6：

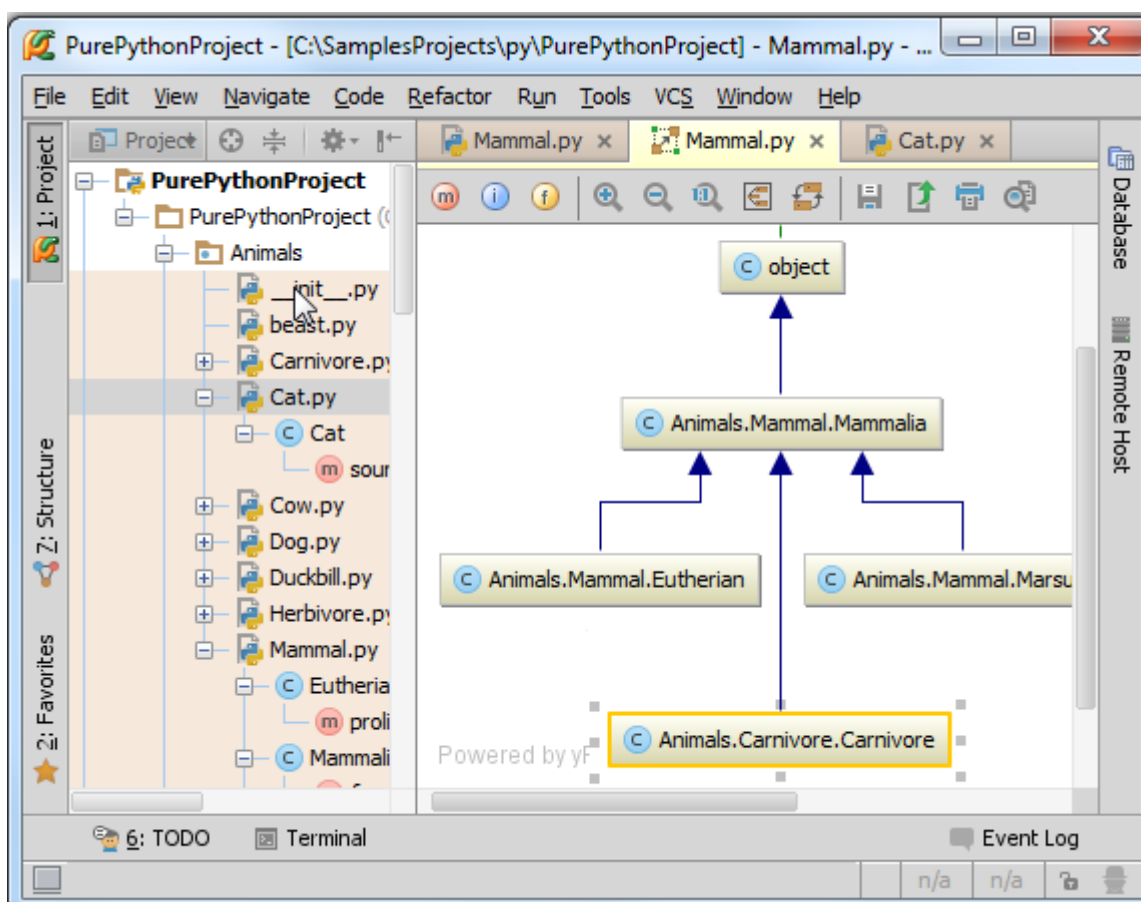


10、向模型中添加元素

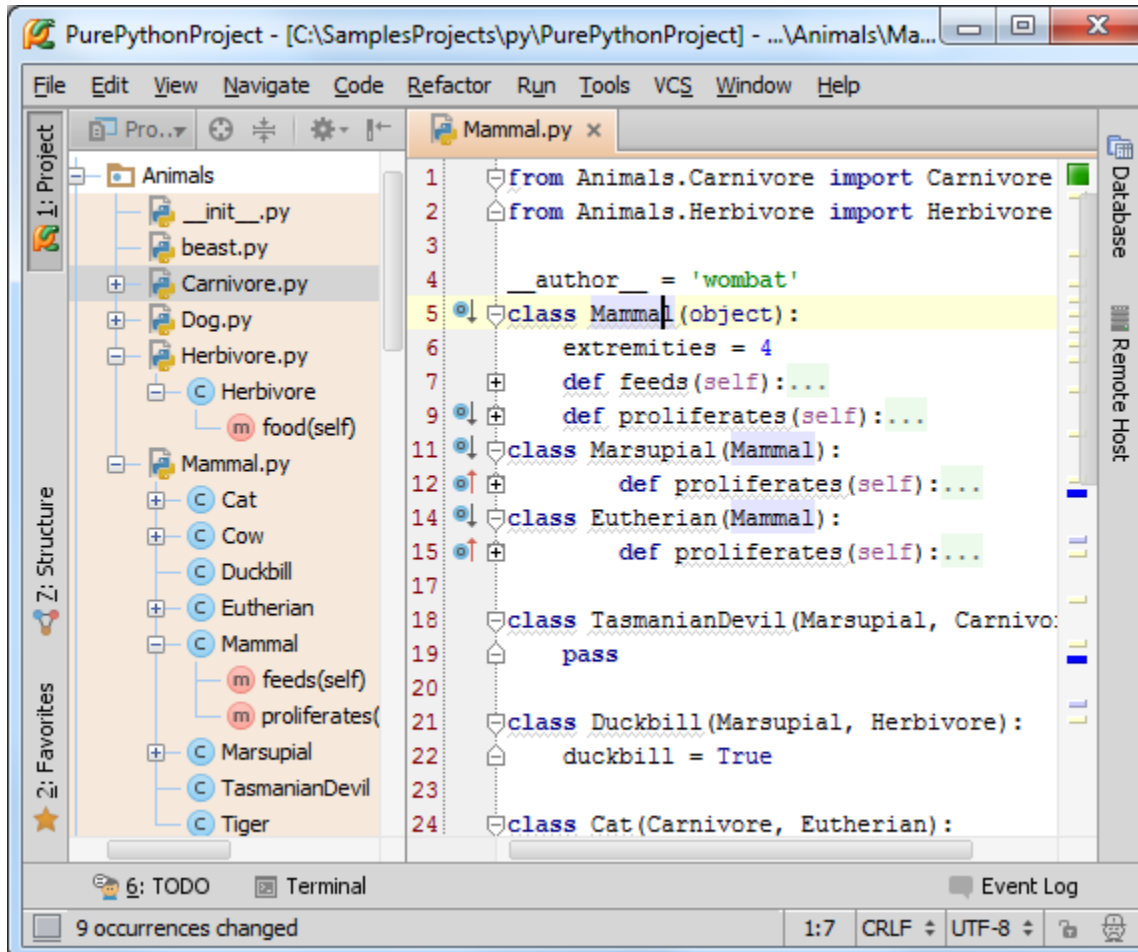
按下 Space，在弹出的窗口中输入类型和名称：



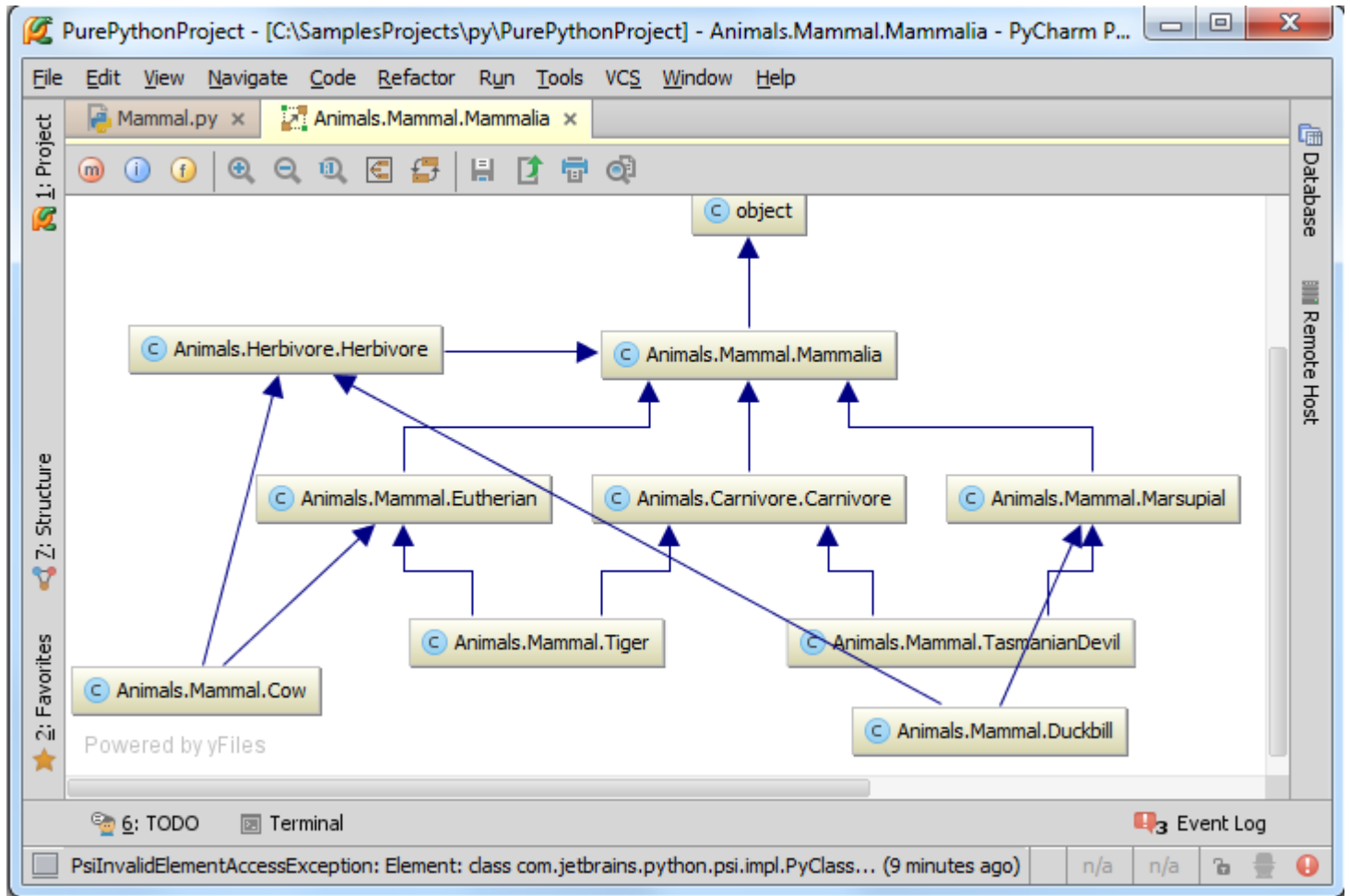
添加 Carnivore:



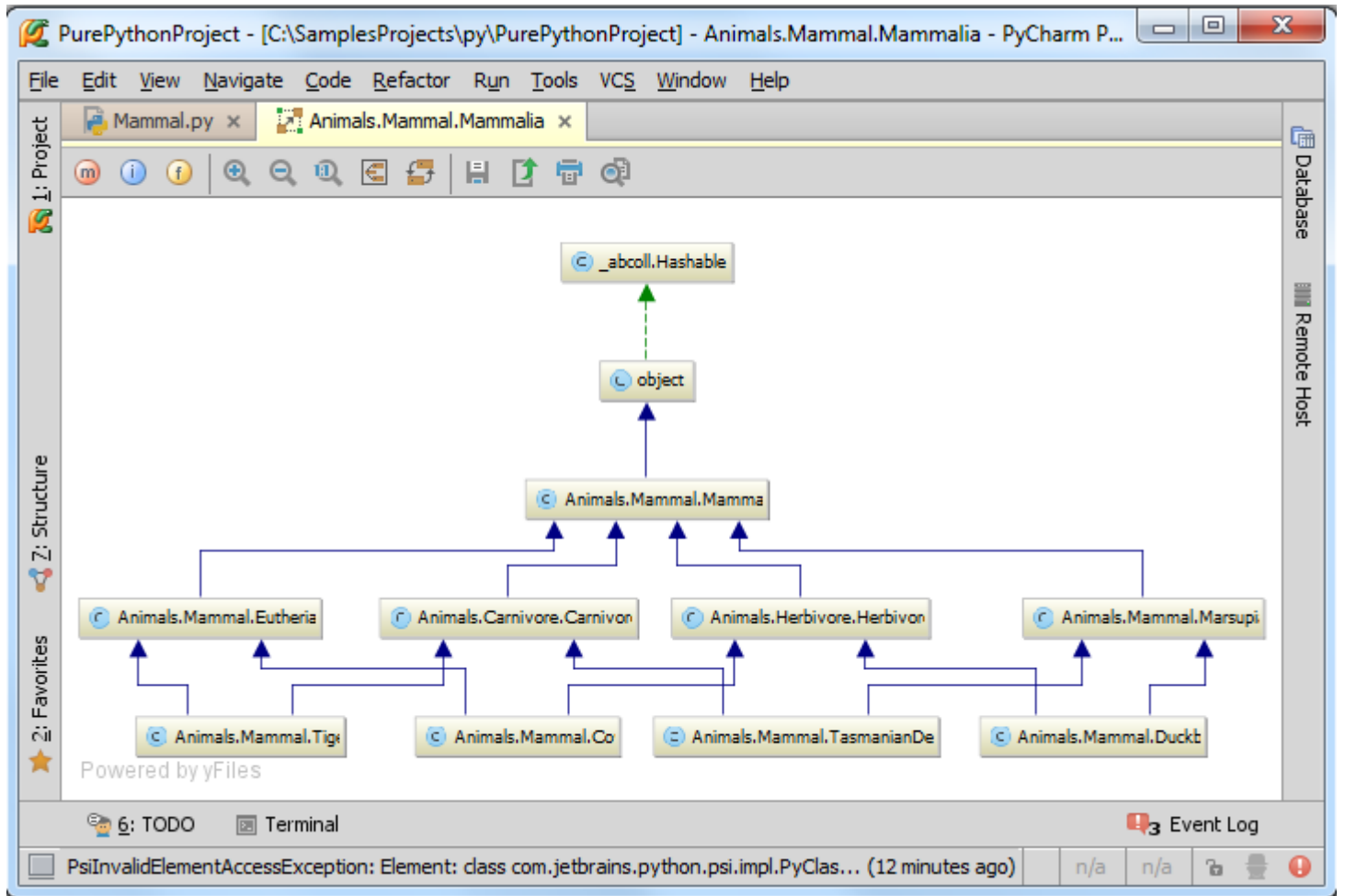
接下来手动创建更多类,如 **Carnivore** 或 **Herbivore**,指定 **mammal** 类型。例如 **Cow** 继承了 **Herbivore** 和 **Eutherian**。**Tiger** 继承了 **Carnivore** 和 **Eutherian**。**Duckbill** 继承了 **Herbivore** 和 **Marsupial**,**TasmanianDevil** 继承了 **Carnivore** 和 **Marsupial**:



按下 Space, 将这些类加入类图:

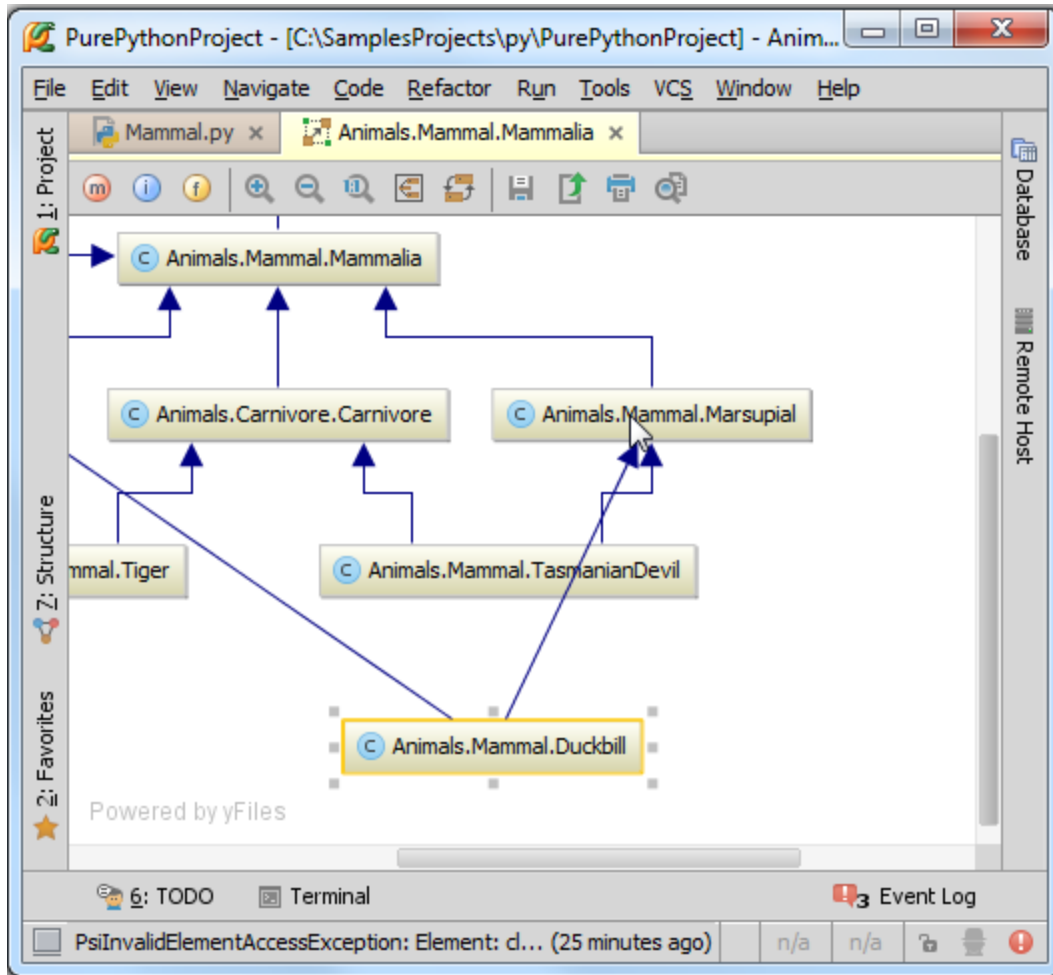


单击  按钮，优化显示风格：

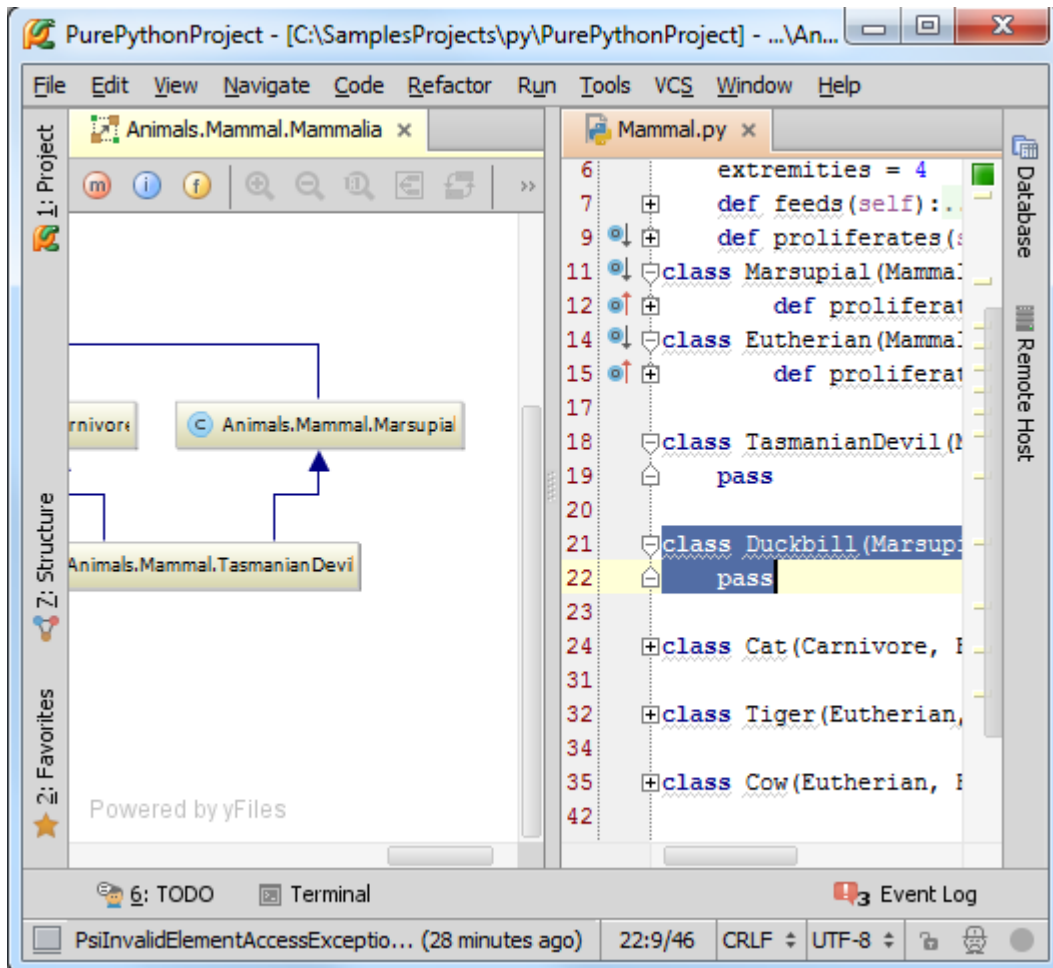


11、从类图中移除元素

在类图中选中一个元素，Delete：



此时删除的类仍存在于源码中，只是在类图中不可见：



12、通过类图浏览代码更改

推荐使用 `Ctrl+Alt+Shift+D` 快捷键，或者  按钮。