**测量Python代码的路径覆盖率**

张钧天、朴乾浩

**摘要**

我们在开始这个项目时，考虑了实现100%的路径覆盖率的好处。首先，它使部署和修改程序变得更容易，也使程序员更容易理解内部程序的代码。

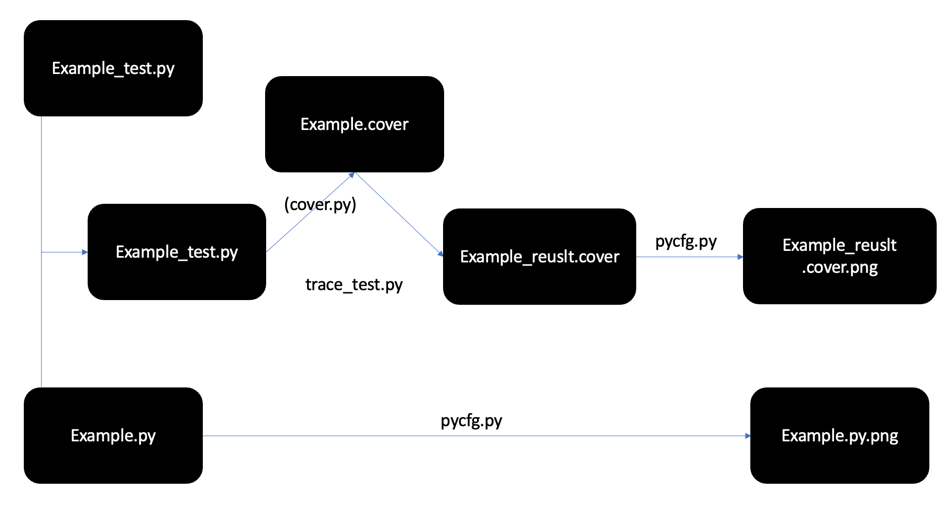
目前，在python中没有测量路径覆盖率的相关库，只有语法或分支覆盖率，所以我们决定创建一个测量路径覆盖率的库。主要的思想是利用trace库跟踪python代码，从所跟踪的代码上使用ast库去分析ast-树结构并算出其路径数量。

项目地址：<https://github.com/pagh2322/path-coverage>

**假设**

假设程序员在某个python文件（Example.py）里编写几个简单的函数，之后想要测试那些函数。于是，又编写了测试该函数的文件（Example\_test.py）。为了让后面的跟踪稳定执行，保证Example.py文件不能有非必要的空行。

**整个流程**



逻辑流程图

1. 程序员在Example.py中写了一个程序，然后创建了Example\_test.py，其中写有测试该程序的代码。

2. trace\_test.py文件读取该测试文件(\_test文件)并创建Example.cover文件，后者又通过cover.py文件创建Example\_result.cover文件。这些文件包含纯粹的执行代码。

3. 使用pycfg.py来计算该代码中的所有可能的路径。

**生成源代码的CFG**

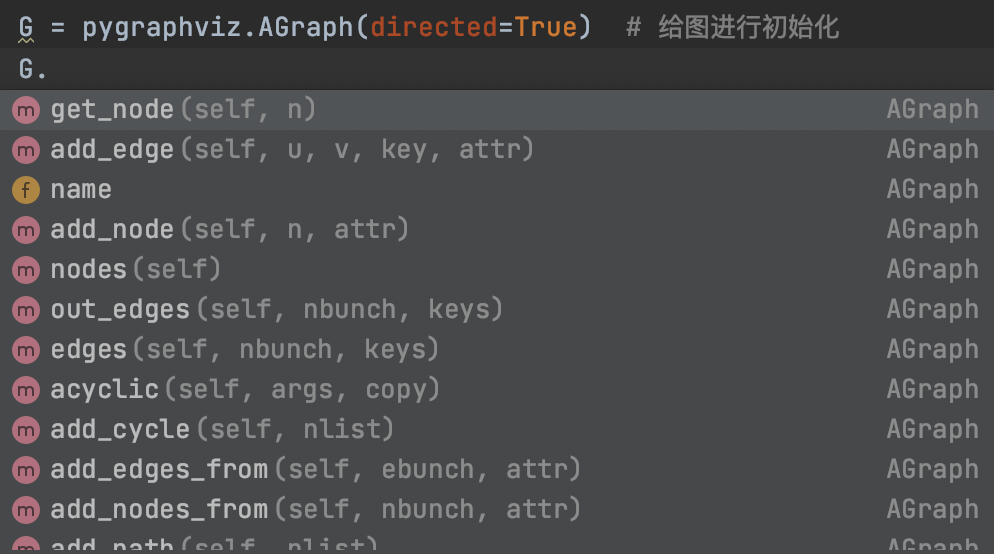
通过Python的astunparse库来实现对被测试程序Example.py的语法解析，同时，我们可以用到pygraphviz库来实现Control Flow Graph（控制流图）的可视化，使最终的结果更为直观。

具体而言，我们可以定义一个CFGNode类，存储被测试程序的诸多信息，例如父节点列表parents、调用函数列表calls、节点唯一id序号rid、节点缓存cache等变量，这对于后续生成控制流图、深度优先遍历节点等步骤都带来便利。



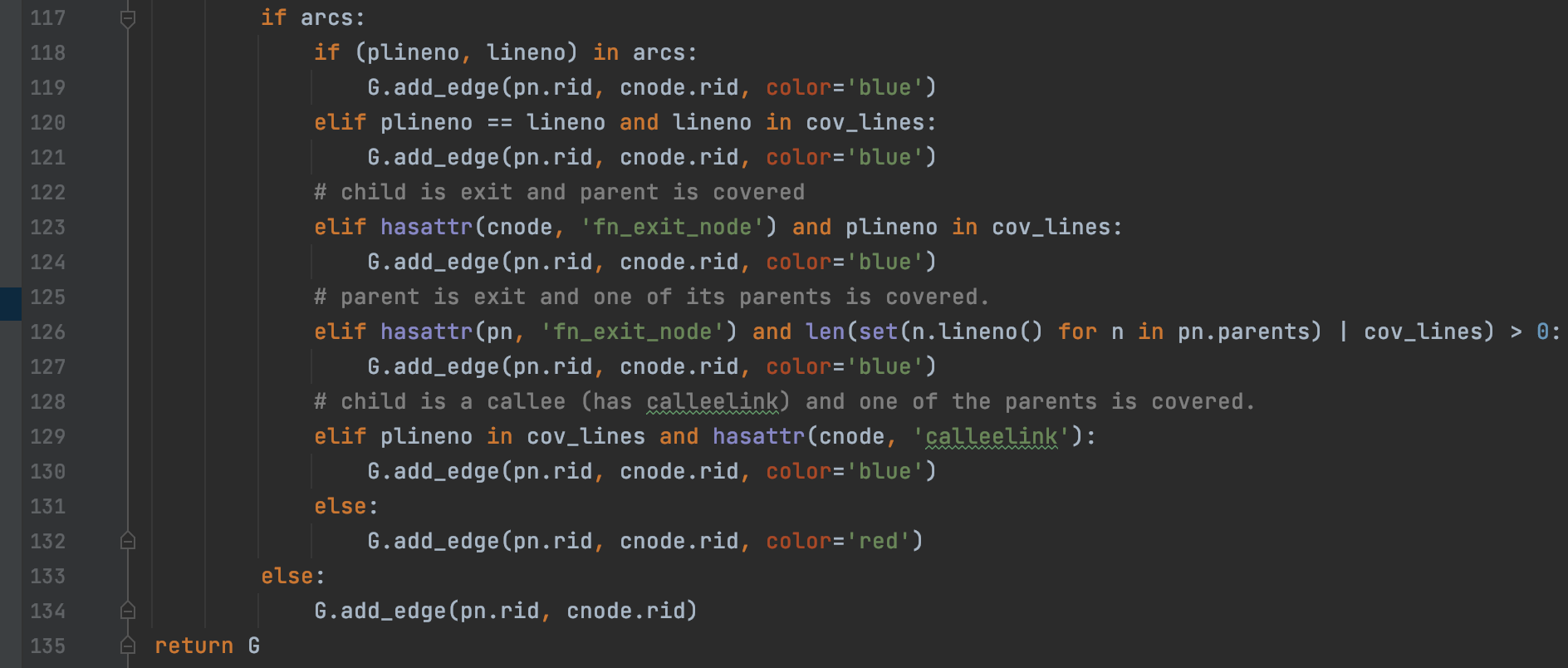
CFGNode类

针对图的生成，我们利用的是AGraph结构，它是一个经典的图的结构，并且具备多样的方法，可以有效且良好地存储图的信息。



AGraph方法

对于语句中不同的情况，我们可以分别进行对应的判定并处理，最终存储到G中。在具体处理程序中，我们运用两种颜色来标记节点，blue表示已经执行的代码路径，red表示还未执行到的代码路径。最终将其可视化为一个图形。



标记节点

此外，针对deminator算法，它用于计算控制流图中每个节点的支配者集合。其中，支配者集合表示的是，在控制流图中，某个节点的所有路径上都必须经过的节点集合。

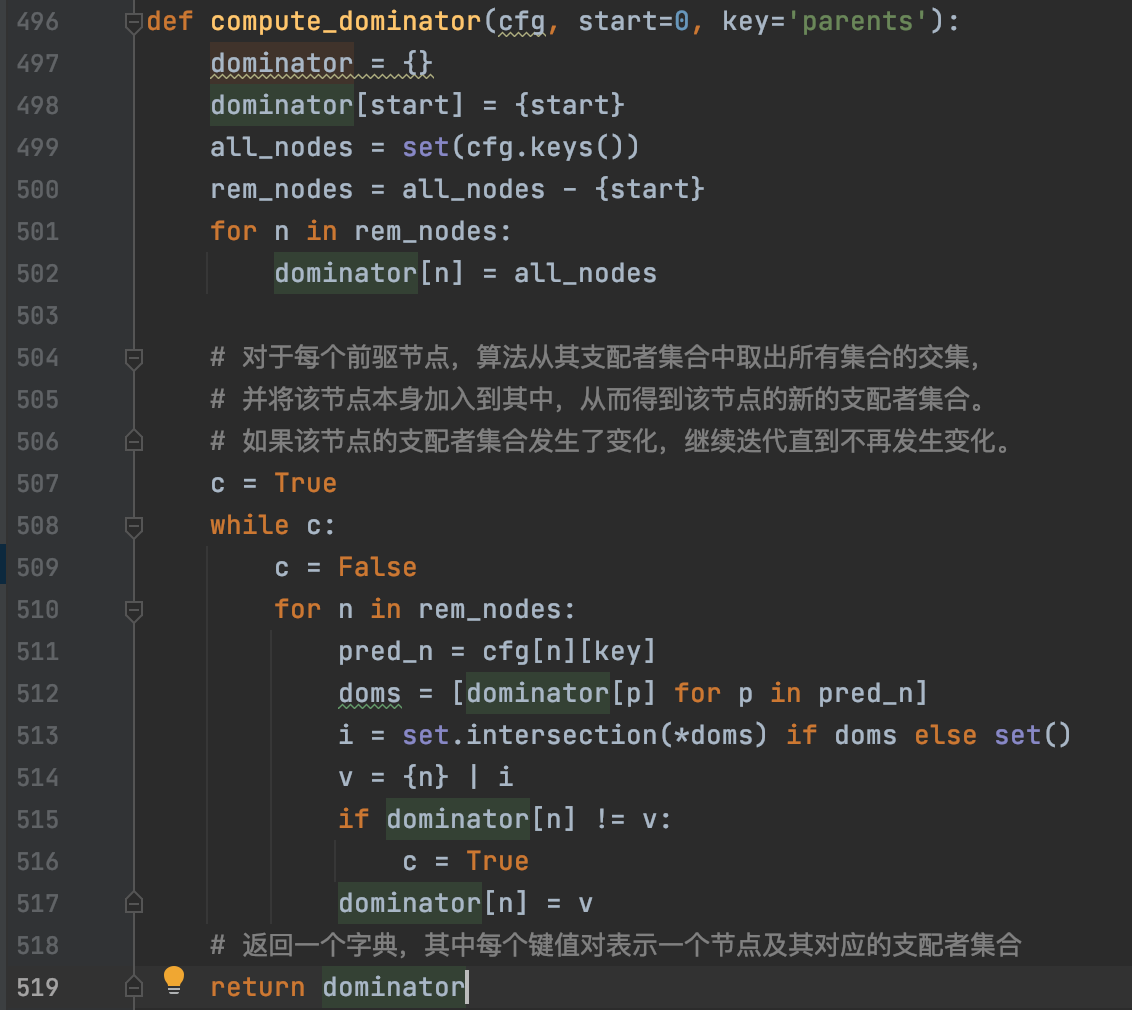
首先，初始化支配者字典(dominator)，将起始节点(start)映射到包含它自己的集合，并且初始化所有节点的支配者集合为包含所有节点的集合，除了起始节点。

然后，对于每个非起始节点n，将其支配者集合初始化为所有节点的集合。

重复一下操作直至不再变化：

* 对于每个非起始节点n，获取其前驱节点集合。
* 从前驱节点的支配者集合中取出所有集合的交集，并将节点n本身加入其中，得到节点n的新支配者集合。
* 如果节点n的支配者集合发生了变化，则继续迭代。

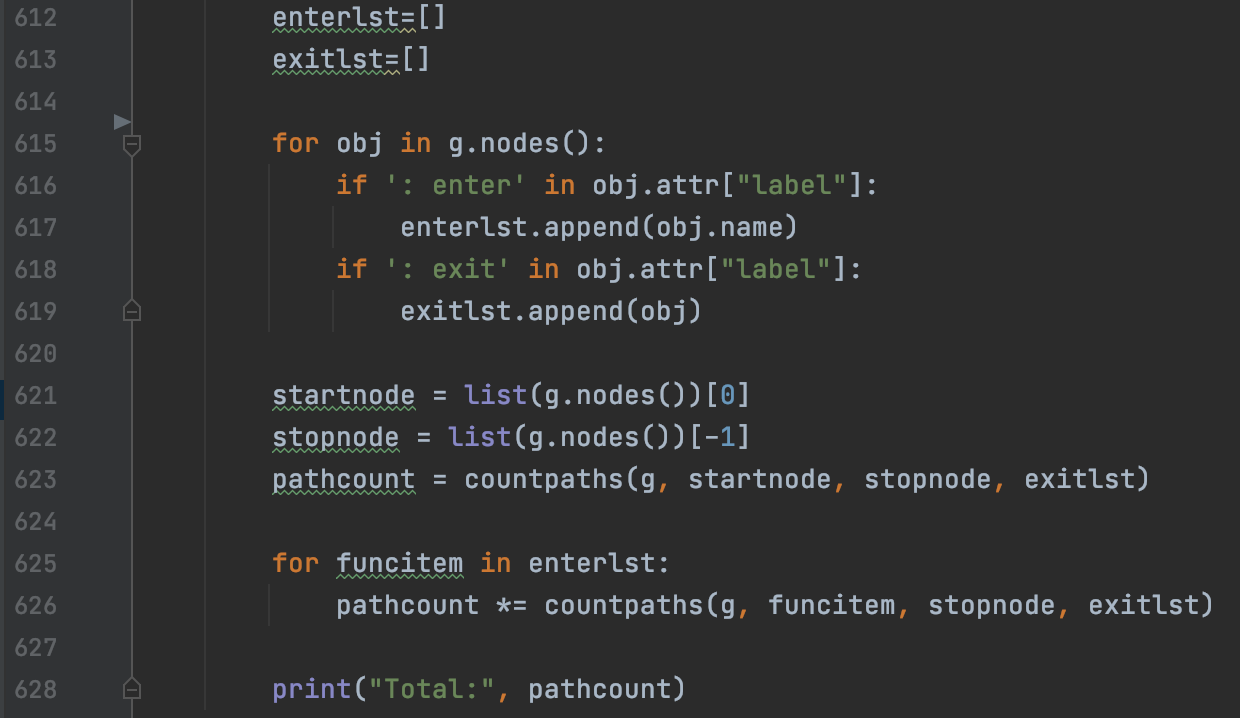
最后，就可以返回一个字典，其中每个键值对表示一个节点及其对应的支配者集合。



支配者字典

**给出源代码的总路径数量**

在获取控制流图之后，我们需要对于图中的节点/边信息进行进一步处理，计算出所有可能的路径。我们考虑最经典的深度优先算法来处理图中信息。



DFS

对于被测试程序，基于上文的假设，可能出现一个主程序入口以及若干函数出入口。在上一步，我们已经对所有节点进行了标记，主程序出入口节点分别是‘start’与‘stop’，函数的出入口节点标记为‘enter’与‘exit’，针对这些标记，我们获取到相对应的enterlst与exitlist，对每个enterlst的节点进入深度优先函数进行遍历判定。针对简单程序的情况，我们将每一个函数遍历的分支数进行累乘，得到最终的总路径数量。

**语句跟踪**

首先，我们需要使用python的trace库来跟踪一个python文件中的哪些代码被执行。这个库会生成一个覆盖率报告，关键是要修改那个报告文件（.cover 文件），创建一个包含纯粹的执行代码的文件。

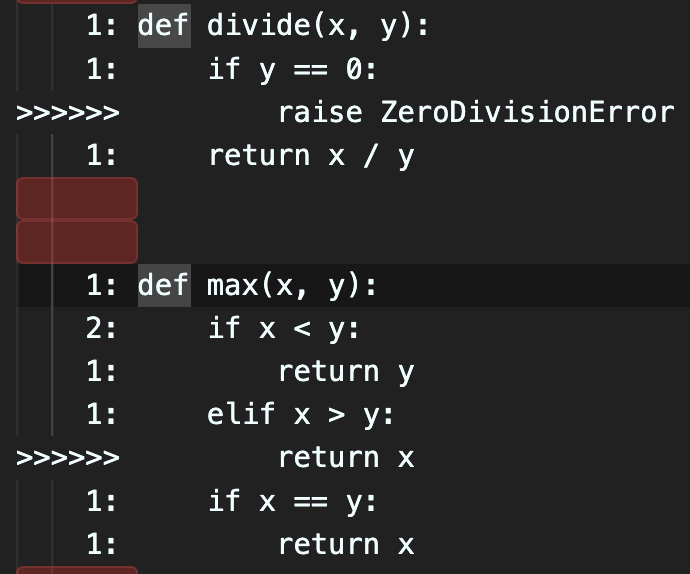
首先是trace\_test.py。它的代码较短，其功能主要是生成.cover文件。



trace库所提供的Trace类包含run函数和results函数。

* run(str)：str是要执行的代码的字符串，在执行该代码时，trace库会跟踪它，并生成相关的跟踪信息。
* results()：返回CoverageResults类的变量，它包含跟踪结果信息。通过执行该变量的write\_results函数，会生成最初的.cover文件。
* getPureName(str)：提取测试文件名。例如，str为Example\_test.py时，返回Example。

之后，执行cover.py文件的makeCoverFile函数，生成\_result.cover文件。下面是最初的.cover文件的例子。

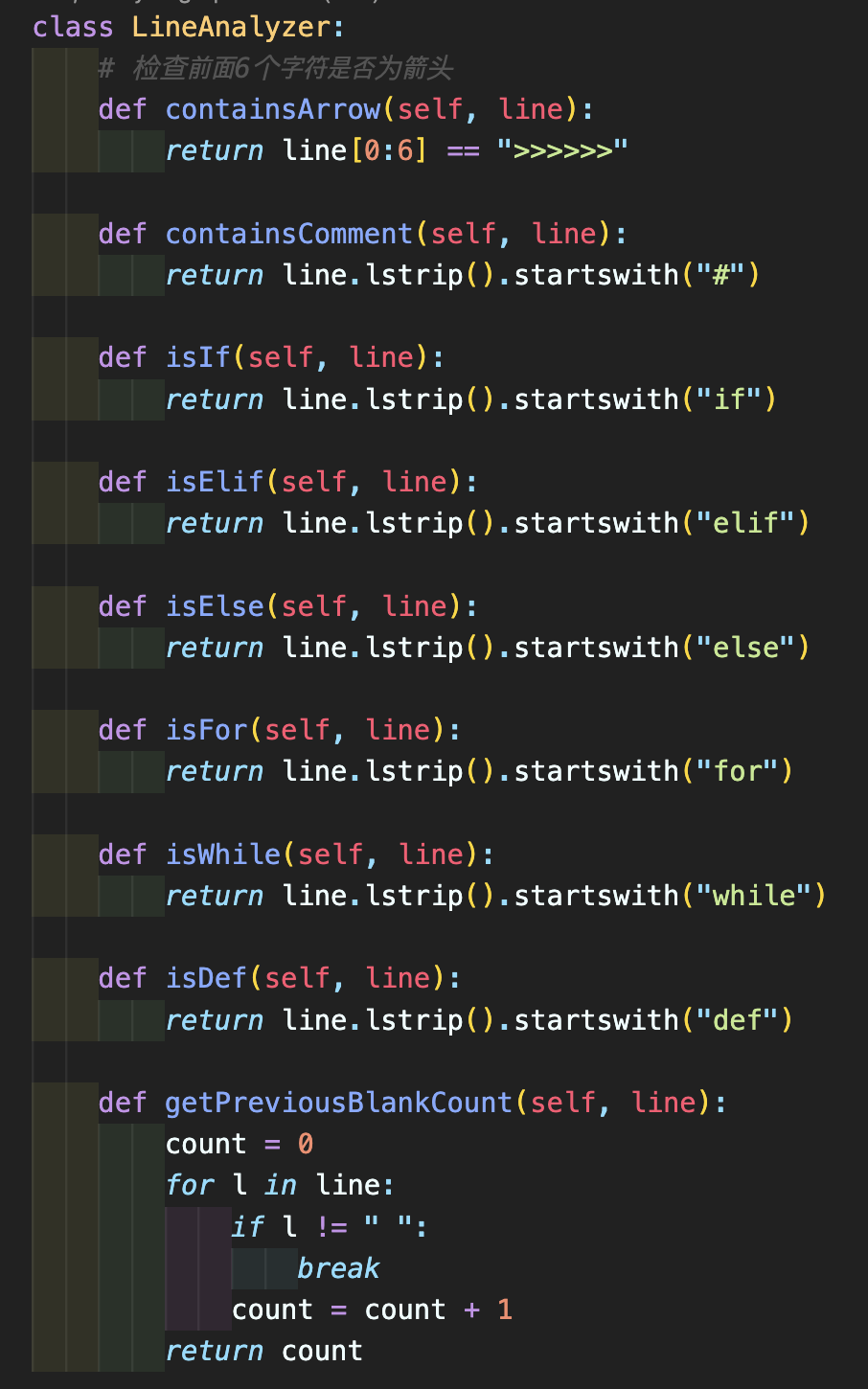


以下是cover.py文件中的代码，先看一下makeCoverFile函数。



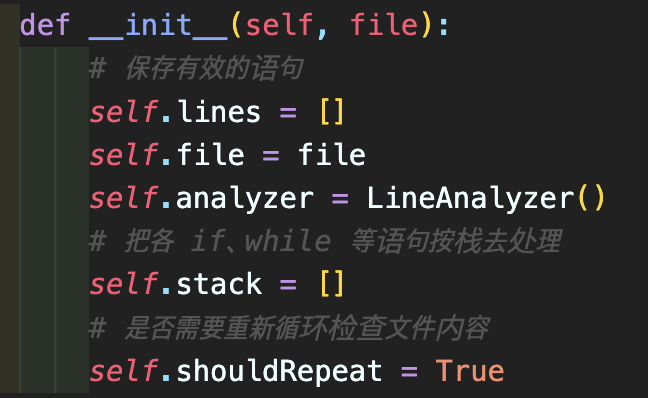
本函数先读取trace库所生成的.cover文件。之后，利用我们编程的CoverFileGenerator，输出最终的\_result.cover文件，同时删除初时的.cover文件以及其他不必要的文件。

在修改.cover文件时，会使用2个类：LineAnaylzer和CoverFileGenerator。以下代码是LineAnaylzer部分。



它是为了分析文件中每一行语句是什么类型，比如if语句，while语句等。它还可以计算该语句的indent值为多少，这是根据python的语法特性去判定该语句是否在if（while、for等）语句内的。

之后是CoverFileGenerator类，init函数为如下。



之后，执行gen函数生成我们所要的\_result.cover文件。该函数会调用preProcessing函数以及repeatProcessingh函数。

preProcessing函数的作用是对最初的.cover文件进行处理。因为它包含不需要的数字以及6个箭头（>)，需要合理的处理。preProcessing函数读取.cover文件的每行，并利用analyzer分析它是否包含箭头，若是，则跳过膝一行。若不是，从第7个字符开始读取，检查该语句是否为def语句。

* 如果不是，则意味着该行是在某个函数块里，直接加到stack。
* 如果不是，则意味着该行是新的函数块的开始。需要调用processDef函数把stack里的所有元素（某个函数的代码）进行处理，并加到lines里。最后，把新的函数块加到空的stack里。

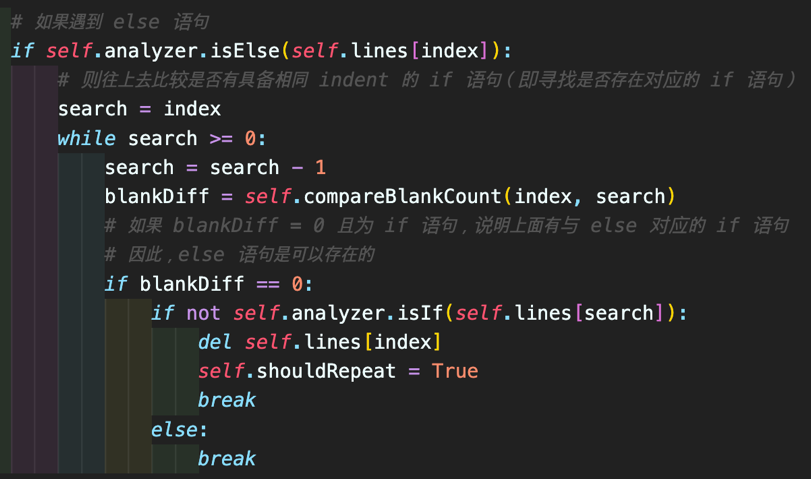
这时，lines会包含有效的语句。

repeatProcessing函数将反复遍历lines，直到标志值（shouldRepeat）为False时，停止执行。代码为如下。



当遇到空行时，prcessBlankLine会检查该行是否删除，如果是，则把该行从lines中删除掉。

当遇到if（for、while等）语句时，processNormal会去进行处理。这也是实现起来最难的部分。举一个例子：如果该行是else语句时，需要检查它所对应的if语句是否在lines里。以下是实现的代码。



这时，可以往上去查找是否有相同indent的if语句。因为indent相同意味着该if语句对应目前的else语句。遇到elif语句时，做类似的过程。

在processNormal函数里处理各种语句时，会出现各种情况。比如if语句里又包含多层的if语句时，需要合理的处理indent，来保证每个if-else语句块的代码。经过处理之后，有可能出现if或else等语句块是空的，即：

if bool\_var:

else:

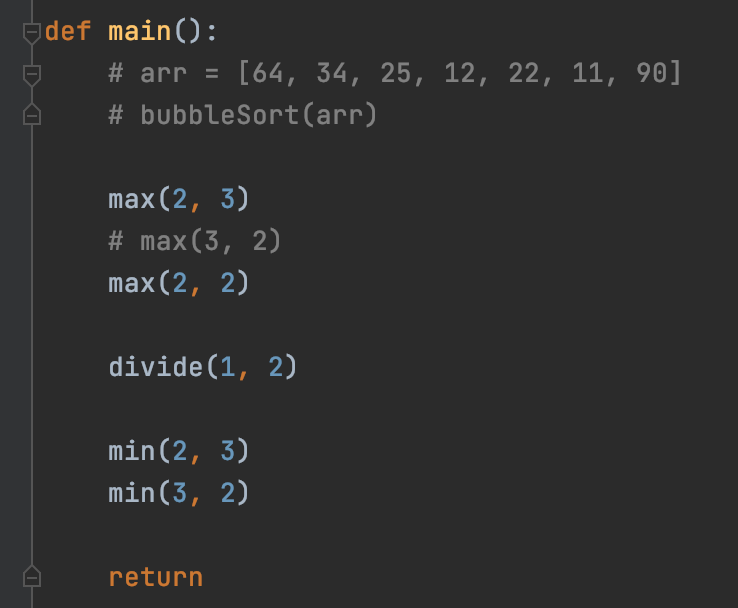
# …

这时候，需要把if语句删除，同时要把else语句去改变。最好是写成：if not bool\_var。目前暂时实现成删除else语句，这显然会导致错误，尤其是在if-else之间存在elif语句的情况。这时，只删除else语句会导致else语句块内的代码自然加入到elif语句。

最后，repeatProcessing函数结束并生成\_result.cover文件。并把该文件传递给pycfg.py去执行，可以计算出所有可能的路径。

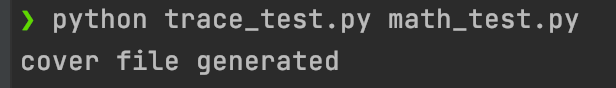
**样例结果测试**

以“math.py”为例，“math\_test.py”是测试数据。

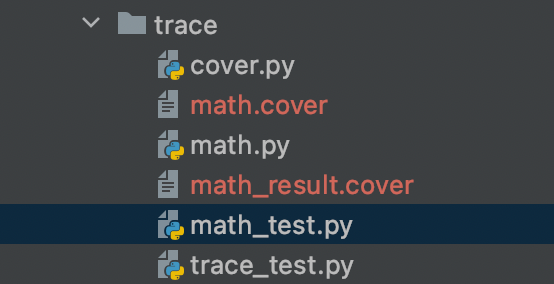


测试数据 math\_test.py

首先，对数据进行追踪，在pycfg/trace目录下，利用“trace\_test.py”，执行下列语句，生成处理前的“math.cover”与处理后的“math\_result.cover”。

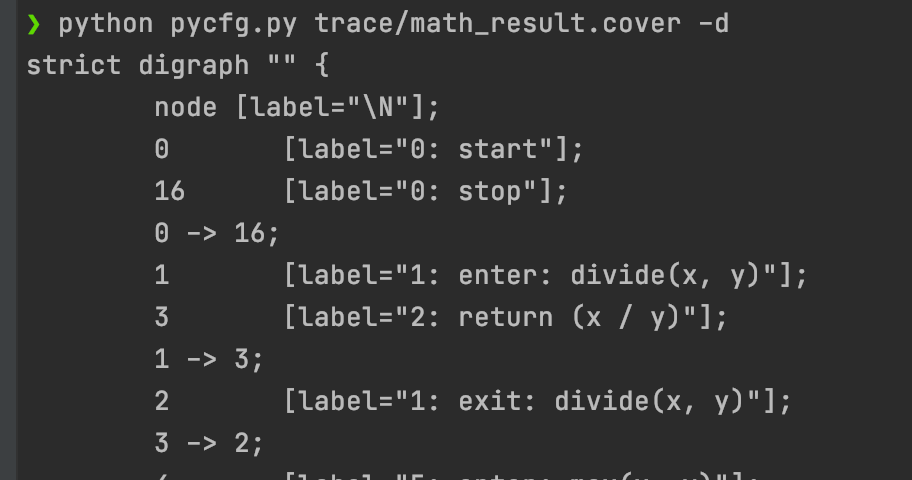


追踪数据路径

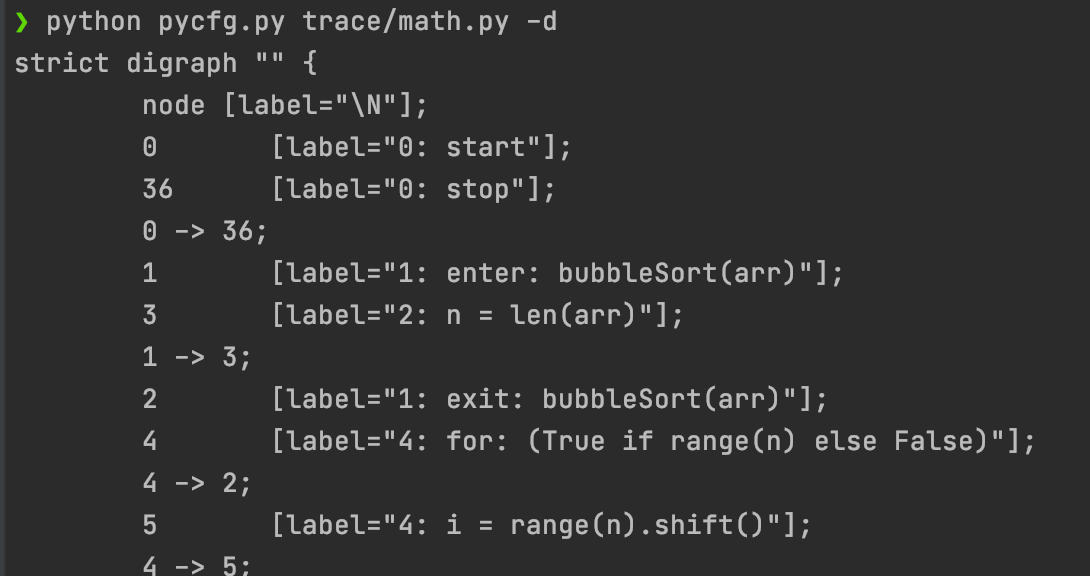
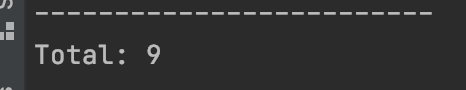


执行后目录

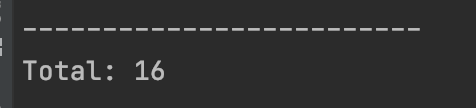
之后，在pycfg目录下，利用“pycfg.py”，对原文件和处理后文件，分别获得其cfg数据，二者相除即为所求的路径覆盖率（9/16）。

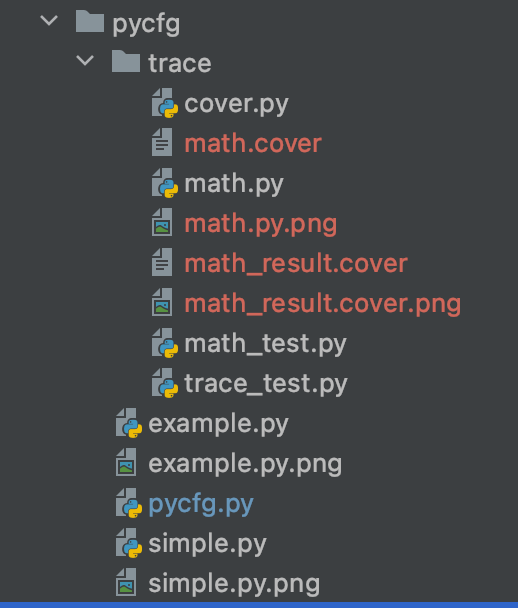


获取分子控制流图以及分子路径数量



获取分母控制流图以及分母路径数量





执行后目录文件

**结论**

该项目仍有待完善之处，例如跟踪估量花费、进一步优化路径、多文件的情况处理等。但对于简单的预设情况，该项目可以顺利得出路径覆盖率，从而更好地评估测试数据。

**参考文献**

1. <http://pymotw.com/2/trace/>
2. <https://docs.python.org/2/library/trace.html#trace.CoverageResults>
3. <https://astunparse.readthedocs.io/en/latest/>
4. <https://pygraphviz.github.io/>