inspect4py:从软件库中自动提取知识的新型框架

2022年1月

1 摘要

这项工作提出了一个新的静态代码分析框架-- **inspect4py**，它可以让用户自动提取Python资源库的主要软件特征和组件（如函数、类、文档、调用列表和控制图），对它们进行分析，以自动检测层次结构、依赖关系和要求，并推断出新的知识，如软件调用或软件分类。通过这个框架，我们旨在提高对用户软件的理解能力，以改善软件研究人员的可持续性、采用性和有效性。我们已经通过以下方式验证了我们的框架。1）使用一组我们以前注释过的100个Python软件库；2）创建一个包含小型Python程序的合成基准套件来评估调用列表。我们的结果显示，在90%的时间里， **inspect4py**能正确分类软件的类型，70%的时间能正确检测软件的调用。

2 简介

现代研究依赖于软件。软件分析数据，模拟真实和设想的系统，并将结果可视化；几乎科学工作的每一步都取决于正确运用软件。虽然商业软件的使用在科学中很重要，但科学家、软件工程师和学生自己也在开发今天在学术环境中使用的大量软件。

然而，理解、采用、比较、执行、复制或扩展科学软件（来自科学界，甚至是科学家自己的软件）可能是一项困难的任务。因此，我们目前正在研究自动理解软件的新方法，通过创建一个 "智能 "框架来实现代码的理解[1]，并检测（不需要人类的干预）。

我们可以从软件资源库中获取主要特征、组件、目标、依赖性、控制和调用流程、类型和执行层次。

这项工作提出了 **inspect4py** [1]，一个新的静态代码分析框架，它允许用户自动检查一个Python软件项目（即软件库中包含的目录及其子目录），并提取最相关的信息（通过使用抽象语法树），如函数、类、方法、文档和导入。该框架还自动生成调用列表和控制图。

inspect4py不仅能提取以前的信息，还能对其进行分析，并将其转化为可操作的见解。 **inspect4py**能够自动检测层次结构、依赖关系和需求，并通过采用一些新颖的启发式方法来推断新知识。例如，给定一个软件项目（如GitHub仓库），目前 **inspect4py**能够对项目的类型（即包、库、脚本或服务）进行分类，并检测如何调用它
.这些信息以JSON和HTML格式存储在本地，最后有提取的不同软件功能的摘要，以及如何安装和/或运行它的信息。

本文的其余部分结构如下。第3节介绍了相关背景。第4节介绍了我们如何用 **inspect4py** 提取几个软件特征。第5节详细介绍了我们生成*call_lists*的启发式方法。第6节和第7节介绍了获得特定Python资源库的软件调用和软件分类的新方法。我们在第8节和第9节中介绍了在这项工作中进行的不同评估。第10节描述了相关的工作，而安装我们软件的说明可以在第11节找到。我们在第12节中总结了成就并概述了一些未来的工作。

3 背景介绍

inspect4py属于静态代码分析工具/框架的范畴，因为它的目的是在不执行Python资源库的情况下，通过解析它们为抽象语法树（*AST*）来自动提取和分析这些信息。在接下来的小节中，将对*AST*和Python静态代码分析工具进行审查。

3.1 抽象语法树)

抽象语法树（*AST*）是源代码的树状表示[2]。编译器在将源代码转换为二进制代码时使用*AST*（见图1）。

- 给定一些文本源代码，编译器首先对文本进行标记，以确定编程语言的关键词、变量、字面意义等。每个标记代表一条指令的 "原子"。

[1] https://github.com/SoftwareUnderstanding/inspect4py

- 然后，标记被重新排列成*AST*，这是一棵树，其中节点是指令的 "原子"，边缘是基于编程语言语法的原子之间的关系。例如，*AST*明确了一个函数调用的存在，相关的输入参数，组成该函数的指令等。
- 然后，编译器可以对*AST进行*多种优化，并及时将其转换为二进制代码。

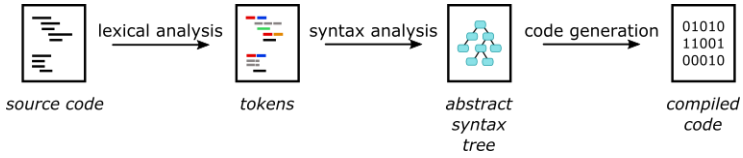


图1：编译器用于将代码转化为拜纳利代码的管道。

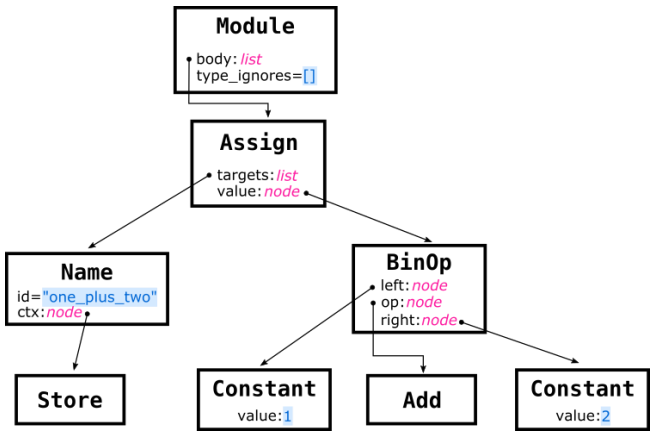


图2："one_plus_two = 1+2 "的*AST*。

在 **inspect4py** 中，我们使用 *AST* 来分析和提取目录/资源库中一个或多个给定文件的软件特征，而无需实际执行任何代码。更具体地说，我们使用 *ast* Python 软件包来生成我们的 *ASTs*。请注意，*AST*将源代码中的每个元素表示为一个对象。这些对象是各种*ast*子类[2]的实例，如Call、ClassDef、Name、BinOp或FunctionDef等等。图2显示了代码的*AST*：one_plus_two = 1+2。

一旦资源库的源代码被解析为*AST*， **inspect4py就会**遍历它以获得不同的软件特征（如类、方法、函数、依赖关系、调用列表等），这在第4节和第5节有详细描述。

[2] *ast*类和子类的完整列表可以在https://docs.python. org/3/library/ast.html找到。

3.2 静态代码分析工具

静态代码分析[3]是分析计算机程序的过程，以便在不实际执行该程序的情况下提取有关信息。一般来说，静态分析是在程序的源代码上进行的，用工具将程序转换成*AST*（见第3.1节）来了解代码的结构，然后找出其中的问题。有几种静态代码分析工具用于分析Python代码。这些工具可以根据它们提取的信息类型进行分类，我们可以看到如下内容。

- 模块（和包）的依赖性图。它显示了模块之间的依赖关系，通过寻找一个给定的资源库或文件的进口之间的关系。换句话说，它显示了 "进口 "是如何相互连接的。下面列出了一些生成这种类型的图的工具的例子。
  - *pydeps* [3]：它是一个Python模块，通过寻找Python字节码中的import-opcodes来发现进口。它还能够创建依赖关系图，并通过*Graphviz*将其可视化[4]。
  - *modulegraph2* [4]：它是一个用于创建和内省Python模块间依赖关系图的库。该图是使用静态分析法从源代码和字节码中创建的。
- 呼叫图（CG）。它显示了一个给定的资源库或文件的函数和方法之间的关系，通过创建一个依赖图来显示程序中哪些函数在调用其他函数。调用图也可以显示其他关系，如定义和文件分组）。下面列出了一些生成CG的工具的例子。
  - *pyan* [5]：它是一个Python模块，对Python代码进行静态分析，以确定函数和方法之间的调用依赖图。它进行静态分析，并构建一个组合源中的对象的有向图，以及它们如何定义或相互使用。
  - *pycg* [6]：它使用静态分析为Python代码生成调用图。它有效地支持高阶函数、扭曲的类继承方案、自动发现导入的模块以便进一步分析和嵌套定义。
- 呼叫列表。它列出了由另一个函数、方法甚至是一个给定源文件的 "主体 "所调用的所有函数和方法。下面列出了一些生成调用列表的工具的例子。
  - *inspect* [5]: inspect模块提供了几个有用的功能，以帮助获得有关实时对象的信息，包括调用列表，如

[3] https://github.com/thebjorn/pydeps [4] https://pypi.org/project/modulegraph2/ [5] https://docs.python.org/3/library/inspect.html

模块、类、方法、函数、回溯、框架对象和代码对象。

  - *pydoc* [6]: pydoc模块从Python模块生成文档。它可以用来生成不同类、方法和函数的调用列表。

- 控制流图（CFG）。它显示了程序或应用程序执行过程中的流程或计算。它为整个程序的结构，特别是子程序的结构提供更精细的 "细节"。下面列出了一些生成CFG的工具的例子。

  - ***staticfg*** [7]：它是一个可以用来为Python3程序生成CFG的软件包。生成的CFG可以很容易地用graphviz进行可视化，并用于静态分析。
  - ***cdm-flowparser*** [8]：它是一个模块，可以接受一个带有python代码或字符缓冲区的文件，对其进行解析，并以片段的形式提供代码的高层次表示。每个片段描述输入的一部分：一个起点（行、列和绝对位置）和一个终点（行、列和绝对位置）。

- Python依赖性管理（PDM）。PDM对于Python应用程序的健康发展至关重要。如果一个Python应用程序依赖第三方库和框架来运行，正确地管理它们可以帮助我们获得安全、可持续性和一致性的回报。下面列出了一些生成这种类型的图表的工具的例子。

  - ***pigar*** [9]：它是一个PDM工具，为给定的Python项目生成需求列表。该工具能够处理不同的Python版本，支持Jupyter笔记本。Pigar使用解析*AST的方法*，而不是正则表达式，可以很容易地从exec/Eval pa-rameters和文档字符串的文档测试中提取依赖库。它还能够通过导入名称搜索软件包。
  - *pipreqs* [10]：这是另一个PDM工具，它根据项目路径生成依赖文件，并列出文件中使用依赖库的位置。

与其他许多静态代码分析工具不同， **inspect4py**从一个给定的资源库中提取尽可能多的信息，获得不同类型的信息（或软件特征）。并对这些信息进行分析，以获得进一步的知识。

[6] https://docs.python.org/3/library/pydoc.html [7] https://pypi.org/project/staticfg/ [8] https://github.com/SergeySatskiy/cdm-flowparser [9] https://github.com/damnever/pigar 10 https://pypi.org/project/pipreqs/

为了提取一些软件特征（如控制流图、需求列表等），我们采用了上面回顾的一些最先进的静态代码分析工具（用黑体字突出的工具）。为了决定将哪些工具集成到 **inspect4py** 中，我们进行了几次实验，对它们进行比较，选择了更适合我们需求的工具。但对于 **inspect4py** 收集的大多数软件特征（如函数、类、文档、调用列表、软件调用等），我们已经开发了自己的方法，这些方法将在本文的下一节中介绍。

4 inspect4py

正如我们在第3.1节中介绍的那样，为了提取给定Python资源库的主要软件特征，**inspect4py**操作解析后的源代码的*AST*节点。它使用 ast.walk() [11] 函数，递归地产生树上的所有子节点。清单2显示了一个简化的代码片段，其中我们遍历*AST*树以提取清单1所示的*"Example.py "*文件的所有函数名称。

```
输入os

path=os . path . j o i n ( " / User " , " /home " , " f i l e . tx t " )

def width ( ) :
    返回 5

def area ( length , func ) 。
    print ( l e n g th * func ( ) )

面积 ( 5 , 宽度 )
```

清单1：计算物体面积的*Example.py*程序。

```
进口

with open ( " Example . py " , ' rb ' ) as s o u r c e :
     t r e e = a s t . p a r s e ( s o u r c e . read ( )
     )

f u n c t i o n s _ l i s t = [ node for node in a s t . walk ( t r e e ) ` ` ` ` ` ` 。
                  i f isinstance ( node , a s t . Function Def ) ] 。
for f in f u n c t i o n s _ l i s t :
         print ( " Function 身份证: 身份证% " % f . name )
```

清单 2: 通过检查哪些节点是 *ast.FunctionDef* 实例，提取 *Example.py* 文件的函数名称。这段代码打印出来。*函数名：宽度*和*函数名：面积*。

[11] https://docs.python.org/3/library/ast.html?highlight=walkast.walk

inspect4py采用了与清单2所示类似（但更复杂）的方法。在我们的案例中，**inspect4py** 使用不同的 *ast* 类（如 FunctionDef, Call, Assign, Return 等）来自动提取类、方法、函数和文档的所有细节以及给定源代码的其他特征。
inspect4py使我们能够表示整个系统（Python库）的代码结构，尽可能多地保留语义细节，并存储关于特定代码片在源代码中的确切位置的信息。下面，我们将描述由 **inspect4py** 提取的每个软件特征，这些特征存储在 *dir_info*、*dir_tree_info* 和 *call_list* 字典中。

- 文件：它在文件级别提取特征。对于找到的每个文件，**in- spect4py都会**（在dir_info[filePath][file]内）存储以下特征。
  - *path*: 文件的路径。
  - *fileNameBase*：不含扩展名的文件名。
  - *扩展名*：文件的扩展名；以及
  - *doc*：在文件级别上包含的文档。该方法支持多层次的*文档*提取，如文件的*长描述*、*短描述*和*完整描述*。

  .
- 依赖性。 它在依赖关系层面提取特征。 对于检测到的每个依赖关系，**inspect4py**（在dir_info[filePath][dependencies]中）存储一个*依赖关系*条目，其中包含以下特征。
  - *import*：模块名称（例如，import os）。
  - *from_import*: 从一个模块导入的函数或方法（例如从datetime导入datetime）。
  - *别名*：将每个模块作为一个别名导入（例如，将numpy导入为np）；以及
  - *类型*：当模块属于同一个资源库时，我们将其定义为 "本地 "依赖；例如，导入 inspect4py.utils。反之，当模块属于外部资源库时，我们定义为 "外部 "依赖；例如，import os。可能的值是。
    <*当地人*>和<外部人>
- 功能。它在函数层面上提取特征。对于检测到的每个函数， **inspect4py**（在dir_info[filePath][functions]内）存储了一个具有以下特征的*函数*条目。
  - *name*: 函数名称。

- *args*：函数参数。每个参数都会提取它们的*描述*、*类型*、*default_value*，以及它们是否是*可选的*。
- *doc*：在函数级别包含的文档。该方法支持几个级别的*docstrings*提取[12]，例如如函数*long_description*和*short_description*。
- *returns*：每个函数的返回值。每个返回值，它提取它们的*描述*、*类型*，或是否是一个生成器（*is_generator* ）。
- *min_max_lineno*：函数的起始和结束行；以及
- *call_list*：每个函数的调用列表。
- *store_vars_calls*：存储函数调用（如a=functionA()）或类实例（如b=MyClass()）值的变量。它既存储变量名（如a），也存储被调用的函数（functionA）。
- *nested*: 函数定义可以被嵌套，这意味着一个函数可以在另一个函数的上下文中被定义和调用。所以，*nested为*每个函数存储了一个嵌套函数的列表（如果它们退出）。这个列表里面的每个条目都是一个*函数*条目。

.

- 类。 它在类和方法层面提取特征。对于检测到的每个类，**inspect4py**（在dir_info[filePath][classes]内）存储了一个具有以下特征的*类*条目。
  - *名称*：类名称。
  - *extend*：该类继承自的类名。
  - *min_max_lineno*: 类起止线
  - *doc*：在类的层面上包含的文档。该方法支持多层次的文档提取，例如类的long_description和short_description。
  - methods[13]:每个类中定义的方法 列表。每个类的方法，它（在dir_info[filePath][classes][className][methods]里面）存储一个*方法*条目，其特征与*函数*条目相同。
- 主： 它 在"主"层面上提取特征。如果name == main，**inspect4py**就会存储（在dir_info[filePath]）一个*主*条目。我们把这些文件称为带"主"的脚本。在大多数情况下，带'main'的脚本会调用一个函数（即main()）来运行脚本的'主'功能，但这是可选的。

[12] https://www.python.org/dev/peps/pep-0257/

13方法是指作为类的一部分的函数。一个函数只是指一个独立的函数。

在这些情况下，我们在dir_info[filePath][main]中存储此类函数的名称。

- 身体：它在 "身体 "层面上提取特征。 如果检测到一个 "主体"，**inspect4py** 会在 *dir_info* 中存储一个*主体*条目，即dir_info[filePath][body]。我们将所有不是函数/类/方法或文件/模块中的导入的东西定义为 "主体"。我们把这些文件称为带有'body'的脚本。*body*条目具有以下特点。
  - *call_list*：主体调用的列表。
  - *store_vars_calls*：存储主体调用值的变量。
- 控制流：它以文本文件或图表（png/pdf/dot）的形式提取每个文件的控制流，取决于用户的偏好。我们使用了*cdmcf-解析器*和*staticfg*工具。这些工具生成每个文件的控制流。
- 调用列表。它将函数、方法和 "主体 "调用收集在它们的*调用列表*字典中（例如，dir_info[filePath][body][call_list]）。 **inspect4py**还存储每个函数调用（*嵌套*或*本地*）的上下文。关于创建这些*call_list的方法*的更多细节将在第5节描述。
- 要求。它通过使用 *pi- gar* Python 软件包来提取一个给定的源代码（文件、目录或资源库）的所有需求。每个发现的需求，它存储（在[dir_info][filePath][requirements] 内）一个具有以下特点的*需求*条目。
  - *名称*：要求（库/模块/工具）名称。
  - *version*: 需求的版本。
- 文件层次结构。它提取给定目录或软件库的文件层次结构，并将其存储在*dir_info_tree*中。
- 软件调用。它推断包含在目录或软件库中的软件是如何被调用的，并将最相关的文件标记为包、库、服务或脚本。有关这一功能的细节将在第6节描述。对于检测到的每个软件， **inspect4py** 都会将一个 *software_invocation* 条目存储为 dir_info[software_invocation]，具有以下特征。
  - *类型*：软件类型。
    可能的值是：<包‖库‖*service*‖script>。
  - *installation*: 安装软件包或库的命令
  - *运行*：调用一个软件的命令

- *has_structure*。表示一个文件是否有 "主"，或者没有。如果没有 "main"，它表示文件是否有 "body"。这个特征只适用于服务、脚本和测试。可能的值是。

  <main|body|without_*body*>

- *mentioned_in_readme*: 表示该软件是否在readme [14]中被提及。仅适用于服务、脚本和测试。可能的值是：<*True* | F *alse*>。

- 测试。它推断出软件是否是一个测试。关于这个功能的细节将在第6节描述。每发现一个测试，它就将一个*测试*条目存储为dir_info[test]，其特征与*software_invocation*相同。

- 软件类型。它推断出一个目录或软件库中所包含的软件类型（即包、库或脚本）。它将一个*软件类型*条目存储为dir_info[software_type]。关于这个功能的细节将在第7节描述。

文件、*依赖关系*、*函数*、*类*、*主要*和*主体*特征按文件（使用文件的完整路径）存储在*dir_info*中，以跟踪给定源代码的不同文件特征（例如dir_info["/User1/home/file1"][函数]）。另一方面，最后两个特征，*software_invocation*和*software_type*，需要前面所有的特征才能被计算。此外，这两个特征以及*dir_info_tree*（文件层次），只有在我们指出一个目录或一个软件作为输入源代码进行探索时才会被计算。

5 呼叫列表

正如我们在第4节中提到的，**inspect4py** 在给定一个文件或一个资源库（有多个文件）的情况下，为每个函数、方法或 "主体 "生成一个调用列表。在审查了几个生成调用列表的工具（见第3.2节）之后，考虑到我们已经从*AST*中提取的信息水平（如函数、类、main等），我们决定创建自己的方法来生成这些调用列表。这个方法由两个主要部分组成。

- 检测每个函数、方法或 "主体 "的所有*直接*[15]、*参数*[16]、[赋值]*17*和*动态*[18]调用。我们为此开发了**检测**启发式方法。见第5.1节。

[14] Readme文件通常是markdown文档，它提供了一个软件组件的基本描述 功能，如何运行它，以及如何操作它。
[15] 当我们使用一个函数或方法的名称直接调用它时。
[16]，当我们把一个函数作为参数传递时。
17[当]我们把一个函数或方法分配给一个变量[时]，我们用这个变量来进行调用。
[18] 当一个函数的执行只在运行时由另一个函数调用的输入参数决定时。

- 用定义了函数或方法的模块/文件来完成调用名称。为此，我们开发了 **completion_name** 启发式方法。见第5.2节。

我们的启发式方法允许我们处理高阶函数调用、模块、函数闭包、嵌套函数和多重继承。一旦调用列表被计算出来，这些信息就会被收集到*调用列表*字典中。这些信息也可以在每个dir_info[filePath]内跨越*函数*、*类*和*主体*特征（例如，dir_info["/User1/home/file1"][func1"][call_list]）。

请注意，我们还开发了一个基准套件（见第8节）来评估我们生成呼叫列表的启发式方法。

5.1 探测启发式

我们的**检测**启发式由两个主要阶段组成。在第一阶段，启发式开始检测每个类的方法、函数和 "主体"（如果我们有的话）的所有*直接*、*参数*和*赋值*调用（见第5.1.1节）。然后，**检测**启发式应用**completion_name**启发式来完成检测到的调用名称（见第5.2节）。在第一阶段结束时，启发式创建了初步版本的*调用列表*，其中包含几个子调用列表（每个类的方法、函数和 "主体"）。
在第二阶段，启发式检测所有使用预设*呼叫列表的动态*调用。结果，*call_list*被更新（用*动态*调用信息）并最终确定。在这个阶段，完成**名称**启发式也被应用于完成*动态*调用的名称。第5.1.2节详细解释了我们检测*动态*调用的策略。

5.1.1 检测呼叫

正如我们之前所介绍的，在**检测**启发式的第一阶段，它检测了给定源代码的所有*直接*、*参数*和*赋值*调用。启发式通过遍历和检查*AST*树来检测哪些节点是astr.Call的实例。一个简化的代码显示在清单3中。

```
进口
with open(" Example . py " , ' rb ' ) as s o u r c e :
        t r e e = a s t . p a r s e ( s o u r c e . read ( )
        )

c a l l s _ l i s t = [ node for node in a s t . walk ( t r e e )\ node in a s t .
                              i f i s i n s t a n c e ( node , a s t . Ca l l ) ]。
```

清单3：提取*Example.py*文件的调用的简化代码。*Example.py*的源代码显示在清单1中。
对于每一个检测到的呼叫，**检测**启发式又进行了两次检查。

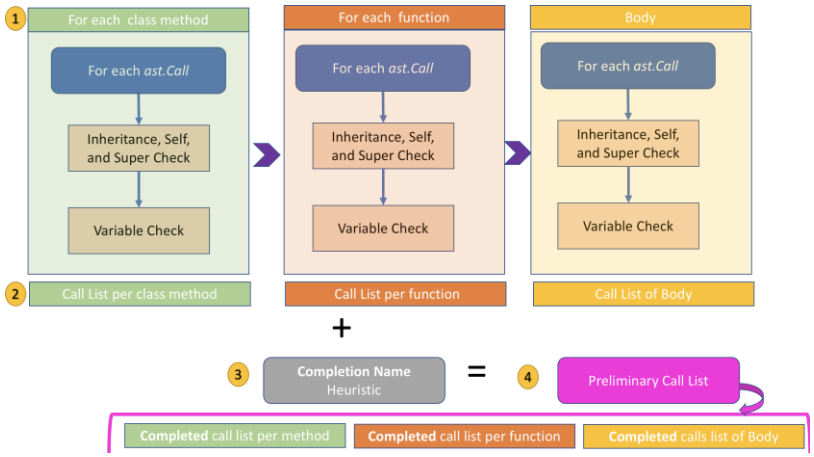


图3：**检测**启发式的第一阶段表示：（1）对于每个方法、函数和主体，它检测所有的调用（*动态*调用除外）。(2) 结果是，它为每个方法、函数和主体生成一个调用列表。(3) 然后，所有的调用名称被完成。(4) 最后得到第一个初步的调用列表，它折中了几个子调用列表。

- 继承。作为一种面向对象的语言，Python允许创建从其他类继承属性和方法的类。为了解决来自父类的继承方法，我们在**检测**启发式中实现了*方法解析算法*（*Method Resolution Or- der*）。当我们有一个对类方法的调用时，这个启发式算法会检查该类是否继承了几个基类（多级继承）。 如果是这样的话，它就会应用*MRO*算法在基类之间进行搜索（这个信息在dir_info[file][classes][className][extend]特征中捕获）。*MRO*使用*深度优先、从左到右的方案*，返回搜索过程中发现的第一个匹配对象。图16显示了一个多重继承的例子。此外，启发式还检测了一个调用是否使用了。
  - self.functionName: 在这种情况下，调用被存储在*call_list*中为className.functionName。
  - super.functionName：在这种情况下，调用被作为baseClassName.functionName保存在*call_list*中。这方面的例子见图16。
- 变量。函数或方法也可以分配给变量（例如cube=Cube(2)）。这些也被称为*赋值调用*。使用这些变量，我们可以多次调用函数。当 **inspect4py** 提取给定源代码的函数、方法和主体特征时，它也会在这些情况下存储变量和

他们在相应的*store_vars_calls*字段中分配的函数（例如，dir_info['/User1/home/super_test_5'][body][store_var_calls]）。在相应的*store_vars_calls*。如果检测到匹配，启发式会更新这些调用，方法是在相应的*call_list*中用其分配的函数替换变量名称。例如，在图16所示的例子中，变量'cube'存储了对'Cube'类的调用（例如，cube=Cube(2)）。这个信息被存储在'body'的*store_var_calls*中，所以当检测到对cube.surface_area()的调用时，这个启发式方法将其存储为Cube.surface_area()。

图3**显示了检测**启发式第一阶段的所有步骤。在这一阶段结束时，我们得到了初步的*调用列表*，该列表将在启发式方法的第二阶段用*动态*调用（如果有的话）信息进行更新。这将在下一章5.1.2节中介绍。

5.1.2 动态调用

当一个函数的执行不是在编写代码时决定的，我们称之为*动态*调用。而是在运行时由函数调用的输入参数决定。例如，在清单1中（显示了*Example.py*的源代码）， area() 调用 func() ，在运行时被 width() 替换。
在**检测**启发式的第一阶段，它将func()检测为对 area()函数的调用。然而，这是不准确的，因为我们在调用 area() 时是在调用 width() （将 width() 作为输入参数），而不是在调用 func() 。

因此，**检测**启发式的第二阶段旨在：检测*动态*调用的函数或方法；用作为输入参数的函数或方法更新其相应的函数或方法或主体调用列表。这些描述如下。

- 检测*动态*调用。
  - 对于包含在初步*调用列表*中的每一个调用，它只选择对repos- itory中开发的函数和方法的调用。我们把这些调用称为*本地调用*，其余的称为*外部调用*。因此，它跳过其余的*外部调用*。例如，给定以下调用os.path.join("/User", "/home", "file.txt")，该方法检测到它是对库os的外部调用，所以它不会探索其参数。
  - 对于每一个被选中的*本地调用*，它检查它们的输入参数，并检测其中是否有函数或方法，再次只选择以**函数或方法**作为输入参数的*本地调用*。

因此，在这一点上，我们已经检测到了所有被*动态*调用的函数和/或方法，对于这些函数和/或方法，我们必须更新其调用列表，同时考虑到这些调用时传递的输入参数。考虑到我们之前的例子，这只是area() 函数的情况，而不是 width() 函数的情况。

- 更新通话清单。
  - 对于之前选择的每个函数和方法（例如 area()）以及它们的每个输入**函数或方法参数**（例如 width()），该方法通过应用 **completion_name** 启发式方法（见 5.2）来提取这些**输入函数或方法参数**的完整调用名称。在我们的例子中，它将给我们 Example.width 作为输入参数 width() 的完整调用名称。
  - 之前过滤的每个函数或方法的调用列表接下来会用之前的信息（**输入函数或方法的参数**完成调用名称）进行更新。继续我们的例子，它将用Example.width更新区域调用列表。
  - 最后，这个方法从这些调用列表中删除了以前的错误条目。在我们的例子中，它将从 area 的调用列表中删除 func。

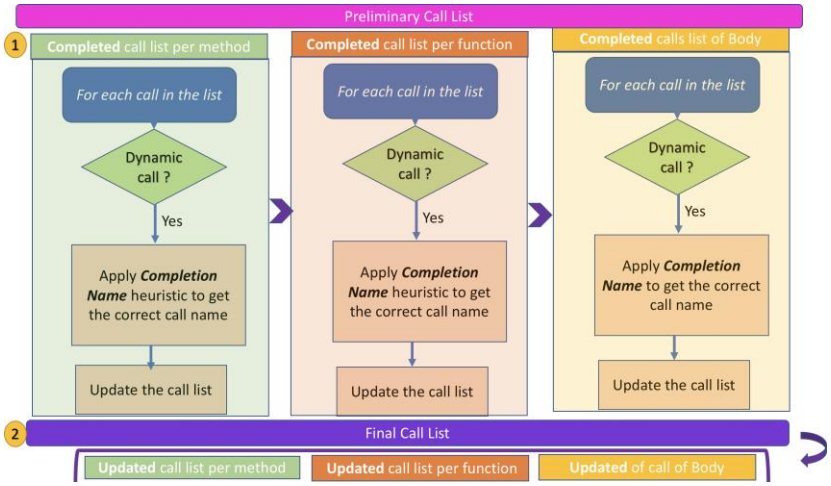


图4：**检测**启发式的第二阶段表示：（1）给定初步调用列表，它检测所有的*动态调用*，完成它们的调用名称并更新适当的子调用列表。(2) 它得到最终的*呼叫列表*，该列表也是由几个子呼叫列表组成。

图4记录了在**检测**启发式的第二阶段中为获得最终的*调用列表*而进行的所有步骤。图5显示了这个第二阶段是如何

*检测*启发式的阶段已经成功地用example.width而不是func()更新了 area() 的调用列表。

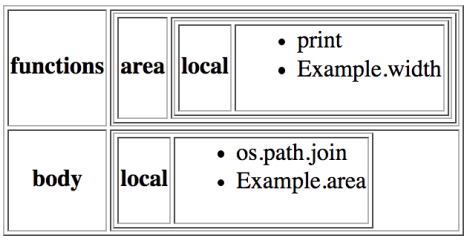


图 5: *Example.py 的*调用列表。*Example.py*的源代码显示在清单1中。

5.2 完成名称启发式

对于每个由**侦查**启发式的任何阶段识别的调用，启发式会返回其方法或函数的名称（<*classN ame*. [*methodN ame*]||*f unctionN ame*>）。

因此，**completion_name** heuristic的目的是找到函数或方法模块（它们在其中有定义）来完成它们的调用名称。因此，**完成度**启发式为每个调用返回以下信息。
<*moduleN ame*>.<*classN ame*.[*methodN ame*]||*f unctionN ame*>。

请注意，在有些情况下，**检测**启发式已经返回了模块的名称。例如在清单1所示的代码片段中，它导入了'os'模块（import os），随后它使用模块名称调用了一个函数（例如os.path.join()）。在这些情况下，调用名称已经完成，**completion_name** 启发式不必执行任何额外的步骤。然而，许多调用并不总是这样，在大多数情况下（见图5），**completion_name** 启发式必须为每个调用的函数或方法找到正确的模块。

Python是高度可扩展的，允许应用程序导入不同的模块。**completion_name**启发式能够识别来自给定源代码中导入的不同模块的函数和方法，并通过应用引入的*MRO*算法考虑其解析顺序。
对于每一个没有指定模块名称的调用，**comple- tion_name**启发式执行以下步骤（见图6）。

- 第1步：通过检查存储在*方法*或*函数*特征中的信息，检查该函数或方法是否已在本地（在当前模块/文件中）定义。

- 第2步：如果搜索没有返回任何结果，它将继续使用存储在*依赖性*特征中的信息检查导入的模块，过滤掉外部导入的模块（版本库外的模块）。对于每个被过滤的模块，启发式执行步骤1，检查该函数或模块是否在其*方法*和*功能*特征中被定义。

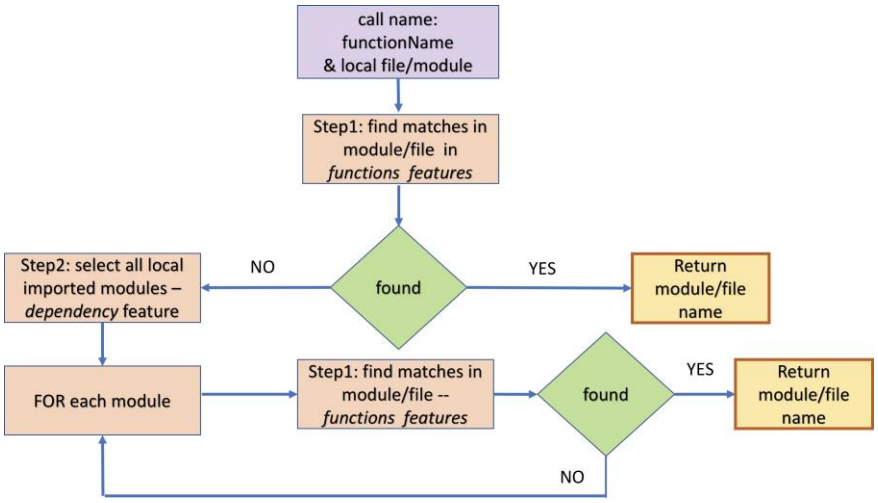


图6：用于查找调用中检测到的函数模块的complement_**name**启发式。

6 软件调用

一旦我们提取了给定目录或软件库的所有特征， **inspect4py就会**进一步分析它们，以检测它们的软件是否应作为*包*、*库*、*服务*或*脚本（s）*被调用。请注意，如果我们用单个文件运行 **inspect4py**，就会进行这种分析（既不分析软件类型）。

因此，我们设计了一个新的**软件定位**启发式，利用所有以前的信息来推断一个目录或软件库中包含的软件的新信息。

从本节开始，我们将专注于分析软件仓库。因此，我们将使用术语**repository**作为 **inspect4py**的输入源代码。

这个启发式方法使用了之前计算的三个结构（见第4节）。

- *dir_info*：包含所有提取的软件特征的字典。

- *dir_tree_info*: 包含目录和文件层次结构的字典。
- *call_list*：包含所有调用列表的字典。

Software_invocation启发式包括：。

1. 探索包和库。
2. 探讨 "主 "文件。我们所说的 "主 "文件是指那些包含 "主 "功能的文件。在这些文件中，我们将探索服务、测试和脚本。
3. 探索'主体'文件。体 "文件不包含 "主 "函数，但它们可以调用函数/方法和/或实例化类。在这些文件中，我们探索的是服务和脚本。
4. 探索没有'main'也没有'body'的文件来检测脚本。注意，这一步只对没有'main'或'body'文件的存储库执行。

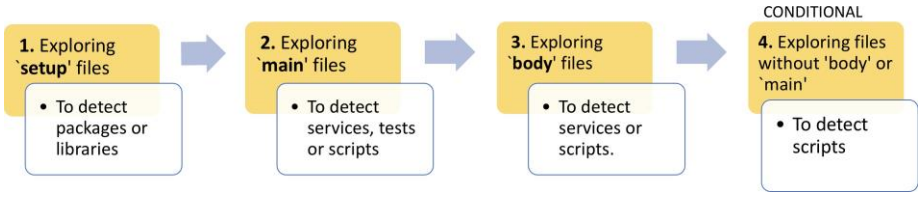


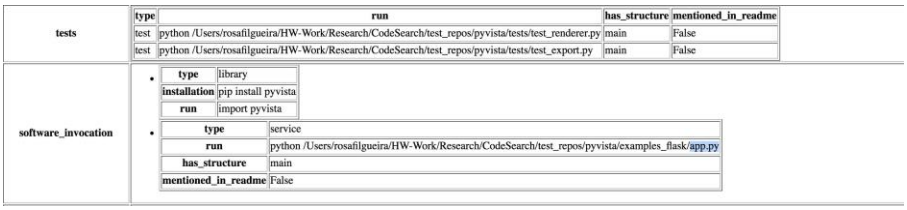
图7：**Software_invocation** 启发式的表示。请注意，启发式总是执行1到3的步骤。 第4步只在软件库中没有发现带有 "主 "或 "主体 "文件的软件包、库、服务或脚本的情况下执行。

这个启发式的模式在图7中显示。此外，对于每个脚本、服务和测试，启发式还检查它是否在 "readme "文件中被提及（如果有 "readme "文件的话）。

在执行了前面的所有步骤后，这个启发式方法将推断出的信息存储在两个字典中。

- *测试*：它存储了被归类为测试的 "main "文件的信息。
- *software_invocation*：它存储了关于其他文件的其余信息。

这两个字典（*测试*和*software_invocation*）都可以有几个条目，分析的信息来自一个给定资源库的不同文件。图8显示了**Software_invocation**启发式是如何发现两个测试的



tests	type	run	has_structure	mentioned_in_readme
	test	python /Users/rosafilgueira/HW-Work/Research/CodeSearch/test_repos/pyvista/tests/test_renderer.py	main	False
	test	python /Users/rosafilgueira/HW-Work/Research/CodeSearch/test_repos/pyvista/tests/test_export.py	main	False

software_invocation		
•	type	library
	installation	pip install pyvista
	run	import pyvista
•	type	service
	run	python /Users/rosafilgueira/HW-Work/Research/CodeSearch/test_repos/pyvista/examples_flask/app.py
	has_structure	main
	mentioned_in_readme	False

图8：通过**soft- ware_invocation**启发式从'pyvista'资源库中提取的信息。

('test_renderer.py'和'test_export'.py)，一个库('pyvista')和一个服务('app.py')，用于**pyvista**资源库[19]。

接下来的几个小节详细解释了在中国进行的每一项探索。
软件定位启发式。

6.1 包和/或库的探索

software_invocation 启发式首先通过使用 *dir_tree_info* 字典查看版本库中是否有 'setup.py' 或 'setup.cfg' 文件（或两者）。
如果它发现这两个设置文件中的任何一个，它会继续解析它们，并优先考虑 'setup.py'。只有在解析 "setup.py "出错或者 "setup.py "没有退出的情况下，它才会解析 "setup.cfg"。

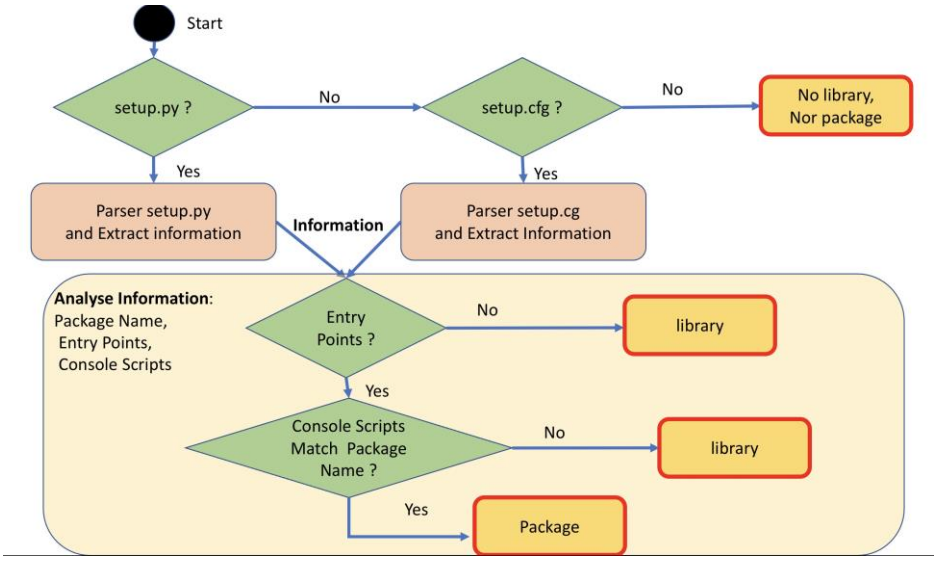


图9：探索 "setup "文件以检测软件包和库。

[19] https://github.com/pyvista

当启发式解析任何一个设置文件时，它获得：a）包的名称；b）入口点[20]；和c）控制台脚本[21]。如果它没有找到任何入口点，那么它就认为它已经找到了一个**库**。如果它找到了一个入口点，那么它将检查控制台脚本的名称，并将它们与包的名称进行比较（在对它们进行标准化处理之后）。如果它发现一个控制台脚本的名称与软件包的名称相匹配（或部分匹配），那么它就认为它已经找到了一个**软件包**。否则，它认为它找到了一个**库**。
我们已经开发了两种方法来解析**软硬件**启发式中的'setup.py'文件。第一种模拟[22]出'setuptools.setup'并检查其参数以提取名称、入口点和控制台脚本。在某些（少数）情况下，由于模拟库和'setup.py'文件之间的不兼容，这种方法不能获得所需的信息。对于这些情况，我们创建了一个替代方法，即逐行读取'setup.py'文件，并提取这些值。此外，我们还开发了一种解析'setup.cfg'文件的方法。图9显示了上述检测软件包和库的步骤。

一旦收集到这些信息，启发式就会将其储存在
dir_info[software_invocation] 字典中的一个条目，**类型为。**
<包 | 库>，安装： pip install 'package_name' (for pack- ages) 或 pip install library_name (for libraries)，**运行**： package_name (for packages) 或 import 'library_name' (for libraries)。

6.2 主要文件探索

dir_info 结构包含了哪些文件是带有'main'的脚本的信息。在这一点上，启发式只选择带有'main'的脚本，也就是那些在dir_info[filePath][main]中有条目的文件。并进一步进行三次探索，我们可以在图 10 中看到。
下文将详细介绍各项探索的细节。

1. 服务：启发式检查之前选择的文件（使用dir_info[filePath][dependencies]功能）所导入的模块和库是否与我们列出的任何服务相匹配 [23]. 对于每一个匹配，启发式都会在dir_info[software_invocation]中存储。
   特征是一个条目，**类型：服务**，**has_structure：main**，**mentioned_in_readme：<True|False>**。
2. test：启发式检查所选文件中是否有任何文件先前被归类为测试文件。这种分类发生在 **inspect4py**

[20] 一个入口点通常是一个函数 (或其他可调用的类似函数的对象，Python 包的开发者或用户可能想要使用它，尽管一个不可调用的对象也可以 作为一个入口点提供。
[21] 控制台脚本是一种类型的入口。它们指向开发人员使av ailable作为命令行工具的功能，供用户安装软件包。
[22] 模拟图书馆：https://docs.python.org/3/library/unittest.mock.html
[23] 个服务列表：flask, flask_restful, falcon, falcon_app, aiohttp, bottle, django, fastapi, locust, pyramid, hug, eve, connexion

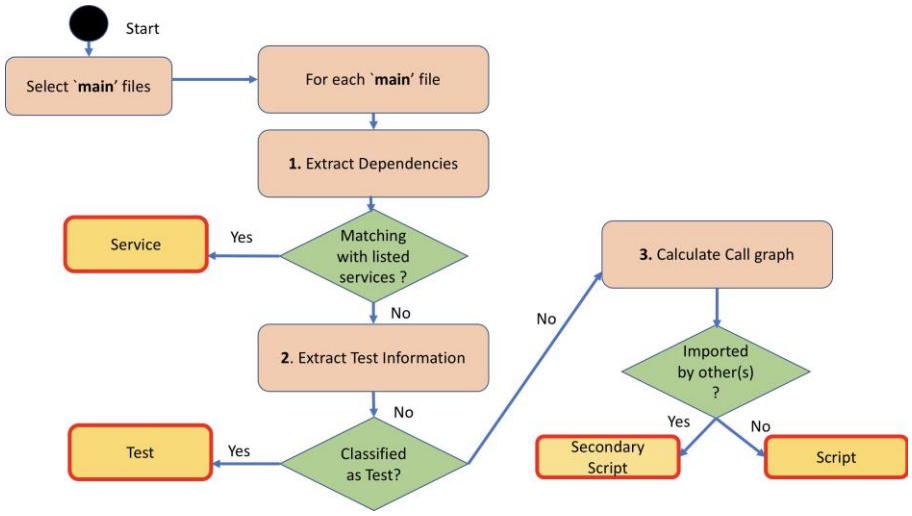


图10：探测服务、测试和脚本的 "主 "文件的探索。

提取软件特征（在创建*dir_info*结构时）。对于每个 "main "的文件，它检查它们是否导入了我们列出的任何测试库[24]。对于每一个先前被归类为测试的文件，**software_invocation**启发式在dir_info[test]结构中存储一个条目，其**类型为：test**，**has_structure：main**，并且**men-**<*True* | **F** *alse*>。

3. 脚本：启发式探索那些尚未被归类为服务或测试的文件。然后，它探索它们之间的关系，以检测哪些带有'main'的脚本被其他脚本（直接或间接）导入。请注意，如果一个带有 "main "的脚本被另一个带有 "main "的脚本（直接或间接）导入，那么导入的带有 "main "的脚本就会作为一个正常的脚本来运行。

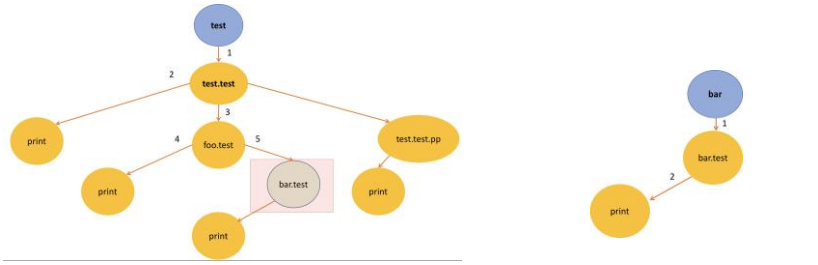
   为了探索这些关系，启发式使用*call_list*结构，为每个选定的文件即时创建一个*call_*graph[25]。*Call_graph*不仅代表了每个文件的调用，也代表了每个调用的调用，这可能意味着其他文件。 *Call_graph*节点有这样的命名模式：<module_*name*
   >.<*class_name*.[*method_name*]‖*f unction_name* > 。

   对于每一个带有 "main "的脚本，它都采用Deep-First-Search（DFS）算法来搜索它们的*call_graph*（见图11）。其目的是找到一个其模块名称与任何其他带 "main "的脚本相匹配的节点。在图11所示的例子中，我们有两个脚本的*Call_graphs*

[24] 测试库列表： unittest, pytest, nose, nose2, doctest, testify, behave, lettuce

[25]我们目前没有存储或预先计算*call_graph*，但在未来我们计划增加这一功能。

脚本有'main'、*test.py*和*bar.py*。当我们在 *test.py* 11.a 的 *call_graph* 中搜索时，DFS 算法在第 5 次迭代中发现了与 *bar.py* 匹配的 *bar.test* 形式。注意，我们正在寻找模块名称的匹配，所以DFS算法只在左边寻找每个节点的名称。另一方面，当我们在*bar.py*的*call_graph*中搜索时，我们无法找到与*test.py*的匹配。为了限制我们的



(a) *test.py call_graph* (b) *bar.py call_graph*

图11：为*test.py*和*bar.py*生成的*call_graphs*，它们是两个带有 "mains "的脚本。DFS算法发现*test.py*与*bar.py*有关系，但不是相反。蓝色节点代表模块/文件（例如 test 或 bar），而黄色节点代表调用。

如果启发式发现一个带有 "main "的脚本调用了另一个带有 "main "的脚本，它就会把这个脚本标记为 "secondary"（例如，在本例中*bar.py*被标记为secondary）。最后，启发式会在我们的dir_info[software_invocation]中为每个带有 "main "的脚本存储一个entry，其**类型为：script**，**has_structure：main**，**mention_in_readme。<*True* | F *alse*>, import_mains**:[带有'main'的脚本列表，它可以导入----。ports]和**imported_by**:[由其导入的带有'main'的脚本列表]）。按照前面的例子，可以在图12中看到生成的*软件信息（software_information）*。

6.3 身体文件探索

software_invocation启发式的下一步是探索带有 "body "的文件。因此，它选择那些在dir_info[filePath][body]中有条目的文件。并对它们进行两次进一步的探索，如图13所示。
这两个步骤详见下文。

1. 探索带有 "body "的脚本的主要原因是为了找到更多的服务。我们通过应用第6.2节中介绍的检测服务的技术来做到这一点，但要使用之前选择的带有'body'的文件。对于每一个匹配，启发式都会在dir_info[software_invocation]中存储。
   一个**类型为服务、结构为主体、内容为<*True* | F *alse*>**的条目

°

software_invocation	type	script
	run	python /Users/rosafilgueira/HW-Work/Research/CodeSearch/code_inspector/test/test_files/test_multiple_mains_indirect/test.py
	has_structure	main
	mentioned_in_readme	False
	import_mains	/Users/rosafilgueira/HW-Work/Research/CodeSearch/code_inspector/test/test_files/test_multiple_mains_indirect/bar.py
	ranking	1
	type	script
	run	python /Users/rosafilgueira/HW-Work/Research/CodeSearch/code_inspector/test/test_files/test_multiple_mains_indirect/bar.py
	has_structure	main
	mentioned_in_readme	False
	imported_by	/Users/rosafilgueira/HW-Work/Research/CodeSearch/code_inspector/test/test_files/test_multiple_mains_indirect/test.py
	secondary	1
	ranking	1

图12：使用'test/test_files/test_multiple_mains_indirect'目录的软件调用方法提取信息的例子。这里我们可以看到，有两个带有main的脚本被检测到（'test.py'和'bar.py'），并且'bar.py'被'test.py'导入。

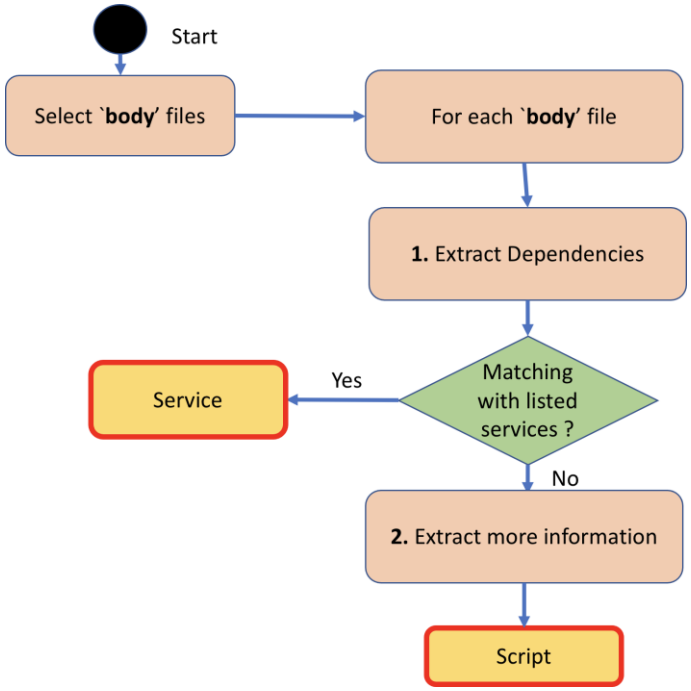


图13：探索用于检测服务和脚本的 "主体 "文件。

2. 只有在启发式在这一点上没有发现任何包、库或服务（在带有 "main "或 "body "的文件中）的情况下，才会继续探索，提取这类文件的信息。 对于每一个文件，它都会在dir_info[software_invocation]中存储一个条目，其**类型为：script**，**结构为：body**，以及**men-invocation。**

<*True* | **F** *alse*>。

6.4 其他文件的探索

最后，还有一些资源库只包含没有 "主 "函数或 "主体 "的文件。这些通常是一些函数或类的集合。

只有在这些情况下，启发式才会探索它们，并在我们的dir_info[software_invocation]中为每个文件创建一个条目，其中包含以下信息：**type: script**, **has_structure: without_body**, and **mentioned_in_readme: <*True* | **F** *alse* >**。图14显示了由**software_invocation**启发式（在某些场合）进行的最后一次探索。

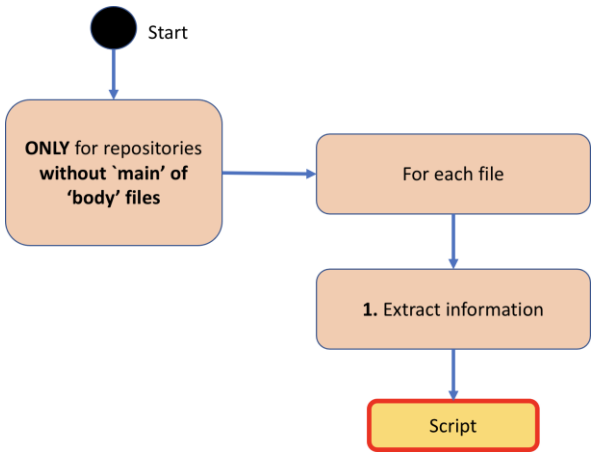


图14：探索没有 "body "也没有 "main "的文件，检测脚本。

7 软件类型和排名

一旦我们的文件分析完成，我们就通过估计其相关性对我们的调用结果进行排名。我们有一个评分函数，它根据之前分析得出的特征给每个结果分配权重。根据作者在90多个代码库中的经验观察，不同的特征有不同的权重。打分函数定义为：。

$$score(x) = w_{lib} * lib(x) + wreadme * \text{readme}(x) + \text{wservice} * service(x) + + \text{w}_{main} * main(x) + \text{w}_{body} * body(x) \quad (1)$$

其中，x表示被评分的调用，$lib(x)$=1，如果代码库被识别为库或包（否则为零），$service(x)$=1，如果x被识别为服务，$main(x)$和$body(x)$等于1，如果x有一个主函数或只有主体（否则它们等于零）。我们还考虑一个服务或脚本是否

在README文件中提到（*readme(x)* = 1），因为这表明它被作者认定为是重要的。与每个特征相关的权重（用w表示，如w_{lib}），包或库的权重最高，其次是服务，最后是脚本。如果一个可执行文件有一个*主要的*功能，它的权重就会比一个只有*主体*的脚本高。

```
"Software_invocation": [
    {
      "运行": [[
           "幽门"。
           "pylode.cli:main,"
        ]
      ],
      "类型": "包"。
      "installation": "pip install pyLODE
      ","ranking": 1
    },
    {
      "类型": "Service"。
      "run": "python pyLODE/pylode/server.py
      ","has_structure": "body"。
      "mentioned_in_readme": true,
      "ranking": 2
    }
  ],
  "软件_类型": "包"
```

清单4：显示两种调用方式的JSON片段

一旦所有的调用替代方案都被评估，我们就按分数从高到低进行排序。为了提高可读性，我们创建了一个升序的排名，其中第一个位置被分配给得分最高的条目。排名第一的元素被返回作为主要的软件类型。如果两个调用替代物具有相同的分数，它们就会被分配相同的排名号。清单4显示了一个软件包的简单例子，其中主要的调用是通过命令行，但也有一个作为服务的替代调用。Inspect4py检测到运行软件包的主要命令（可在代码库中找到的*setup.cfg*中找到），以及另一个脚本（*server.py*），在主README中提到。

8 呼叫列表评估

我们创建了一个基准套件，用于测试和评估本文中提出的生成*调用列表*的启发式方法，我们可以在表1中看到。

结果显示，**inspect4py**正确地生成了所有基准的*调用列表*。从这个表中，我们选择了三个基准，在本文中展示了由我们的启发式方法生成的*调用列表*。图15的代码片段说明了我们的启发式方法是如何识别本地和导入函数的。而图16所示的代码片断说明了如何成功地处理继承（简单和多重）。最后，图17显示了

第三个

类别	#Number	描述
继承性	6	继承方法的解决
级别	6	方法功能调用
进口产品	4	进口模块、函数类
变量	3	为参数赋值的功能
嵌套	2	嵌套函数调用
职能	3	香草函数调用
动态的	12	动态调用
争论	1	将函数作为参数传递
别名	2	作为别名导入函数

表1：用于测试 **inspect4py** *call_lists 的*基准套件。

代码片段，其中包含*动态*调用。从这些例子中，我们可以检查生成的*call_lists 是否*正确。

```
1 from test_classes import *
2 from test_functions import *
3
4 class MyClass_D:
5     def __init__(self):
6         print("Class D")
7         funct_C()
8         funct_D()
9         MyClass_E()
10
11 class MyClass_E:
12     def __init__(self):
13         print("Class E")
14         MyClass_B()
15
16 def funct_D():
17     print("Function D")
18     funct_A()
19
20 a1=MyClass_A()
21 b1=MyClass_B()
22 d= MyClass_D()
23 funct_A()
24 funct_D()
25 MyClass_C()
26 c=funct_D()
```

(a) test_import.py文件

functions	**funct_D**	**local**	• print • test_functions.funct_A
body	**local**		• test_classes.MyClass_A • test_classes.MyClass_B • test_import.MyClass_D • test_import.funct_D • test_functions.funct_A • test_import.funct_D • test_classes.MyClass_C
MyClass_D	**__init__**	**local**	• print • test_functions.funct_C • test_import.funct_D • test_import.MyClass_E
MyClass_E	**__init__**	**local**	• print • test_classes.MyClass_B

(b) 调用列表 示例1

图15：*导入基准*。为test_import.py中包含的python代码生成的每个函数、方法和主体的调用列表。这个例子使用了另外两个文件：test_functions.py和test_classes.py。在这里，我们可以看到名称是如何被相应地填充的，能够识别一个方法或函数是否属于一个导入的模块，每次都能捕捉到相应的模块。

```python
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)

class VolumeMixin:
    def volume(self):
        return self.area() * self.height

class Cube(VolumeMixin, Square):
    def __init__(self, length):
        super().__init__(length)
        self.height = length

    def face_area(self):
        return super().area()

    def surface_area(self):
        return super().area() * 6

cube = Cube(2)
cube.surface_area()
cube.volume()
```

(a) super_test_5.py文件

body	**local**	• super_test_5.Cube • super_test_5.Cube.surface_area • super_test_5.VolumeMixin.volume	
Square	**__init__**	**local**	• super_test_5.Rectangle.__init__
VolumeMixin	**volume**	**local**	• super_test_5.VolumeMixin.area
Cube	**__init__**	**local**	• super_test_5.Square.__init__
	face_area	**local**	• super_test_5.Rectangle.area
	surface_area	**local**	• super_test_5.Rectangle.area

(b) 调用列表 示例2

图 16: *继承基准*。为super_test_5.py中的每个类的方法和主体生成的调用列表。注意在这个例子中，*Cube*类继承了两个类（*VolumeMixn*和*Square*），当我们调用*cube.volume()*时，调用列表能够找到正确的'volume'方法。同样在这个例子中，我们可以看到'super'是如何被正确的类名所取代的。

```python
import test_dynamic_func

def func_3(func):
    return func()


a=func_3(test_dynamic_func.func_1)
print(a)
```

(a) test_dynamic_import.py 文件

functions	**func_3**	**local**	• test_dynamic_func.func_1
body	**local**	• test_dynamic_import.func_3 • print	

(b) 呼叫列表示例3

图17：*动态基准*。为test_dynamic_import.py中的每个函数生成的调用列表。注意我们如何能够推断出func_3调用了test_dynamic_funcs.func_1，而不是调用func。因此，它能够推断出哪些函数是作为参数传递的。

9 软件类型评估

对 inspect4py 的初步评估将软件类型和调用结果与人工注释的语料库进行比较。

9.1 有注释的语料库

我们已经创建了两个不同的语料库来评估我们的方法。对于主要的软件类型检测，我们选择了一个由95个不同的Python代码库组成的语料库（分布在24个包、27个库、13个服务和31个脚本中）[26]。

[26] 基准可在以下网站获得： https://doi.org/10.5281/zenodo.5907936

软件类型	精度	召回	F1分值
包装	1	0.916	0.956
图书馆	0.93	1	0.9637
服务	1	1	1
脚本	0.967	0.967	0.967

表2:软件类型分类的结果。

每个作者都分别对资源库进行了注释，并进行了比较，直到达成一致。这些资源库的范围和领域各不相同，从用于机器学习的研究资源库[27]到流行的社区工具，如Apache Airflow[28]或特定领域的库（如Astropy[29]）。为了获得更广泛的代表性样本，我们还包括了作者所在机构开发的、可用的或正在开发的文档较少的资源库。

第二个语料库是为了评估排名结果而设计的，它是作为第一个语料库的一个子集而产生的。对于所有具有多种调用方法的资源库，作者根据资源库的结构和文档，手动注释了最相关的文件。结果，44个资源库被注释了。

9.2 结果

表2显示了我们对主要软件类型分类的结果概览，支持的每个类别。我们的启发式方法，基于Python应用程序开发的最佳实践，在所有类别中产生了超过95%的F1分数。我们遇到的唯一错误发生在开发者超出常规做法的时候，比如为他们的库创建自定义元数据文件。

为了评估我们的排名结果，我们选择了归一化折现累积增益（NDGC）[7]，这是信息检索中用来评估排名质量的指标。NDGC的范围从0（最小）到1（最大）。我们的综合排名为0.87，这被认为是初步评估中令人满意的。

10 相关工作

许多静态代码分析工具已经被开发出来，用于提取代码元数据、文档或特征[3]。从机器学习的角度来看。

[8]描述了一项关于代码特征提取技术的调查，以便训练源代码模型。libsa4py[9]等工具从代码中创建特征，以补充 inspect4py 中已经提取的信息。其他工具与我们框架内的功能有一些重叠。例如，pydeps[30]和modulegraph2[31]提取模块的依赖性。

[27] https://paperswithcode.com/ [28] https://github.com/apache/airflow [29] https://github.com/astropy/astropy [30] https://github.com/thebjorn/pydeps [31] https://pypi.org/project/modulegraph2/

像pydoc[32]这样的库可以生成HTML代码文档，像pigar[33]这样的库可以提取代码需求，像*pyan*[5]和*pycg*[6]这样的包可以使用静态分析生成Python代码的调用图，包括高阶函数、类继承方案和嵌套函数定义。Inspect4py建立在其中一些工具（例如pygar）的基础上，并将所有这些功能整合在一个框架下，对其所有的结果使用一个独特的表示法。
据我们所知，目前还没有检测软件类型和调用方法的框架。

11 安装和执行

Inspect4py可以通过pip（pip install inspect4py）和Docker来安装。要调用该工具的基本功能（即提取类、功能和它们的文档），需要运行以下命令。

识别和控制的方法是：<input file.py | directory>。

这个命令可以用不同的选项来限定，[34] 以便执行不同的功能。例如，下面的命令提取了类、函数和方法的文档，同时也存储了软件的调用信息（标志-*si*）。

inpsect4py -i repository_path -o out_path -si -html

12 结论和未来工作

本文介绍了 inspect4py，一个静态代码分析框架，旨在从代码库中提取常见的代码特征和文档，以便于理解。初步评估显示了评估Python代码库类型的有希望的再结果，以及识别和排列运行它们的替代方法（从而为潜在用户节省时间）。
我们将 inspect4py 与软件元数据提取工具[10]结合使用，为完成 README 文件提出建议。[35] 此外，我们还计划将我们的结果用于促进函数的并行化和组合，以及在寻找类似代码时比较软件的其他表示方法。
至于未来的工作，我们正在扩展 inspect4py，以便 1）通过注释更多的资源库来改进我们的评估结果，2）纳入新的特征提取工具（例如 libsa4py），以及 3）改进我们的调用检测，以包括说明性的参数例子。

[32] https://docs.python.org/3/library/pydoc.html
[33] https://github.com/damnever/pigar
[34] 见 https://github.com/SoftwareUnderstanding/inspect4py/blob/main/README.md
[35] https://github.com/SoftwareUnderstanding/completeR

参考文献

[1] A.Von Mayrhauser, A. Vans, 软件维护和进化过程中的程序理解, Computer 28 (8) (1995) 44-55. doi:10.1109/. 2.402076.

[2] I.Neamtiu, J. S. Foster, M. Hicks, Understanding source code evolution using abstract syntax tree matching, SIGSOFT Softw.工程。Notes 30 (4) (2005) 1-5. doi:10.1145/1082983.1083143. URL https://doi.org/10.1145/1082983.1083143

[3] J.Novak, A. Krajnc, R. Žontar, Taxonomy of static code analysis tools, in: 第33届MIPRO国际会议，2010，第418-422页。

[4] J.Ellson, E. Gansner, L. Koutsofios, S. C. North, G. Woodhull, Graphviz-open source graph drawing tools, in:P. Mutzel, M. Jünger, S. Leipert (Eds.), Graph Drawing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp.483-484。

[5] J.J. D. Fraser, E. Horner, P. Massot, Pyan3:Offline call graph generator for python 3, https://github.com/davidfraser/pyan, accessed: accessed 09-January-2020 (2020).

[6] V.Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, D. Mitropoulos, Pycg: 在 python 中生成实用的调用图， CoRR abs/2103.00587 (2021). arXiv:2103.00587. URL https://arxiv.org/abs/2103.00587

[7] K.Järvelin, J. Kekäläinen, Cumulated gain-based evaluation of ir technology- niques, ACM Trans.Inf.Syst.20 (4) (2002) 422-446. doi:10.1145/ 582415.582418. URL https://doi.org/10.1145/582415.582418

[8] T.Sharma, M. Kechagia, S. Georgiou, R. Tiwari, F. Sarro, A survey on machine learning techniques for source code analysis (2021). arXiv: 2110.09610.

[9] A.M. Mir, E. Latoškinas, G. Gousios, Manytypes4py:A benchmark python dataset for machine learning-based type inference, in:2021年IEEE/ACM 第18届采矿软件库国际会议（MSR），2021年，第585-589页。Doi:10.1109/MSR52588.2021.00079。

[10] A.Kelley, D. Garijo, A Framework for Creating Knowledge Graphs of Sci- entific Software Metadata, Quantitative Science Studies (2021) 1-37arXiv: https://direct.mit.edu/qss/article-pdf/doi/10.1162/qss_a\ _00167/1971225/qss_a_00167.pdf, doi:10.1162/qss_a_00167.URL https://doi.org/10.1162/qss_a_00167