inspect4py: A novel framework for extracting automatically knowledge from software repositories

January 2022

1 Abstract

This works presents a new static code analysis framework, **inspect4py**, that allows users to automatically extract the main software features and components (e.g. functions, classes, documentation, call lists and control graphs) of Python repositories, to analyse them to detect automatically hierarchies, dependencies and requirements, and to infer new knowledge such as software invocation or software classification. With this framework, we aim to increase the understandability of users' software to improve the sustainability, adoption and validity of the software researchers. We have validated our framework by: 1) using a set of 100 Python software repositories, which we have previously annotated; 2) creating a synthetic benchmarks suite containing small Python programs to evaluate call lists. Our results show that 90% of the time **inspect4py** classifies correctly the type of software, and 70% of the time it detects correctly the software invocation.

2 Introduction

Modern research depends on software. Software analyses data simulates the real and envisaged systems and visualizes the results; just about every step of scientific work depends on correctly employing software. Although the use of commercial software is important in science, scientists, software engineers, and students themselves are developing a significant amount of software used today in academic settings.

However, understanding, adopting, comparing, executing, reproducing or scaling scientific software (from scientific communities, or even scientists' own software) can be a difficult task. Therefore, we are currently working on new ways to understand software automatically, by creating an 'intelligent' framework that enables code comprehension [1] and detects (without human inter-

vention) the main features, components, goals, dependencies, control and call flow, type and execution hierarchies from a software repository.

This work presents **inspect4py** [1], a new static code analysis framework, that allows users to inspect automatically a Python software project (i.e., a directory and its sub-directories contained in a software repository) and to extract the most relevant information (by using Abstract Syntax Trees), such as functions, classes, methods, documentation and imports. The framework also generates automatically call lists and control graphs.

inspect4py, not only extracts the previous information, it also analyses it and translates it into actionable insights. **inspect4py** is able to detect automatically hierarchies, dependencies and requirements, and to infer new knowledge by employing several novel heuristics. For example, given a software project (e.g. GitHub repository), currently **inspect4py** is able to classify the type of project (i.e, package, library, script or service), and to detect how to invoke it . This information is stored locally in JSON and HTML formats, having at the end a summary of the different software features extracted, as well as the information of how to install it and/or run it.

The rest of the paper is structured as follows. Section 3 presents the relevant background. Section 4 introduces how we extract several software features with **inspect4py**. Section 5 details our heuristics for generating *call_lists*. Section 6 and Section 7 presents the novel methods for obtaining the software invocation and software classification of a given Python repository. We present the different evaluations performed in this work in Section 8 and Section 9. A description of related work is described at Section 10, while the instructions for installing our software can be found at Section 11. We conclude in Section 12 with a summary of achievements and outline some future work.

3 Background

inspect4py falls under the category of static code analysis tools/frameworks, since it aims to extract and analyse automatically the information of Python repositories without executing them by parsing them to abstract syntax tree (AST). In the next subsections both, AST and Python static code analysis tools are reviewed.

3.1 Abstract Syntax Trees)

Abstract syntax tree (AST) is a tree representation of source code [2]. Compilers use ASTs when transforming source code into binary code (see Figure 1):

- Given some text source code, the compiler first tokenizes the text to identify programming language keywords, variables, literals, etc. Each token represents an "atom" of an instruction.

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

- Tokens are then rearranged into an *AST*, a tree where nodes are the "atoms" of the instructions, and edges the relationships between the atoms based on the programming language grammar. For instance, the *AST* make explicit the presence of a function call, the related input arguments, the instructions composing the function, etc.
- The compiler then can apply multiple optimizations to the *AST*, and ultimately converts it into binary code.

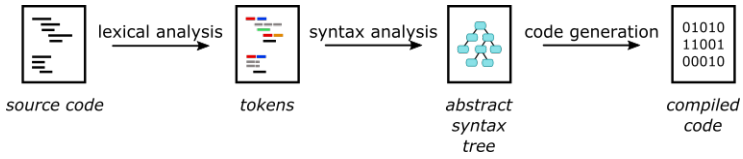


Figure 1: Pipeline used by compilers to transform code into bynary code.

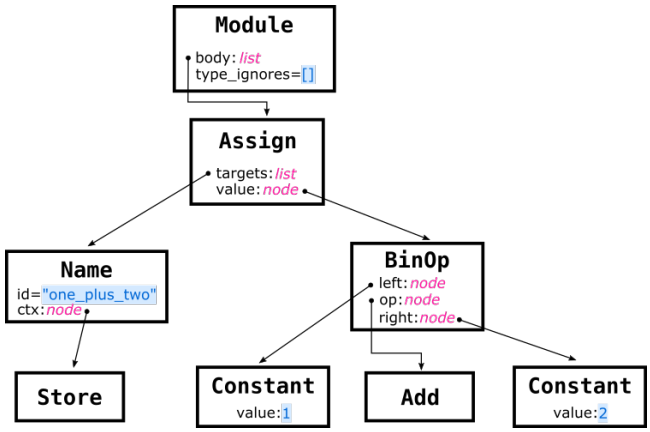


Figure 2: *AST* of '`one_plus_two = 1+2`'.

In **inspect4py** we use *ASTs* to analyse and extract the software features of a given file or files within a directory/repository, without actually executing any code. More specifically, we used the *ast* Python package to generate our *ASTs*. Note that an *AST* represents each element in a source code as an object. These are instances of the various *ast* subclasses [2], such as `Call`, `ClassDef`, `Name`, `BinOp` or `FunctionDef` among others. Figure 2 shows the *AST* of the code: `one_plus_two = 1+2`.

Once the source code of a repository is parsed as an *AST*, **inspect4py** traverses it to get different software features (e.g. classes, methods, functions, dependencies, call list, etc.), as is described in detail in Section 4 and Section 5.

[2]The complete list of *ast* classes and subclasses can be found at `https://docs.python.org/3/library/ast.html`

3.2 Static Code Analysis Tools

Static code analysis [3] is the process of analyzing a computer program to extract information about it without actually executing it. Generally, static analysis is performed on the source code of the program with tools that convert the program into *AST* (see Section 3.1) to understand the code's structure and then find problems in it. There are several static code analysis tools for analysing Python codes. These can be classified according to the type of information that they extract, as we can see as follows:

- Modules (and packages) dependency graph: It shows the dependencies among modules, by finding the relationship among the imports for a given repository or file. In other words, it shows how the 'imports' are interconnected. Some example of tools for generating this type of graphs are listed bellow:
  - *pydeps* [3]: it is a Python module which finds imports by looking for import-opcodes in python byte code. It also enables to create dependency graphs and visualize them with *Graphviz* [4].
  - *modulegraph2* [4]: it is a library for creating and introspecting the dependency graph between Python modules. The graph is created using static analisys from source and byte code.
- Call Graphs (CG): It shows the relationships among the functions and methods for a given repository or file, by creating a dependency graph that shows which functions within a program are calling other functions. A call graph can also show other relationships such as definitions and file groupings). Some example of tools for generating CGs are listed bellow:
  - *pyan* [5]: it is a Python module that performs static analysis of Python code to determine a call dependency graph between functions and methods. It performs an static analysis, and constructs a directed graph of the objects in the combined source, and how they define or use each other.
  - *pycg* [6]: it generates call graphs for Python code using static analysis. It efficiently supports Higher order functions, Twisted class inheritance schemes, Automatic discovery of imported modules for further analysis and Nested definitions.
- Call List: It lists all functions and methods called by another functions, methods or even a 'body' of a given source file. Some example of tools for generating Call Lists are listed bellow:
  - *inspect* [5]: The inspect module provides several useful functions to help get information, including call lists, about live objects such as

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

modules, classes, methods, functions, tracebacks, frame objects, and code objects.

  - *pydoc* [6]: pydoc module generates documentation from Python modules. It can be used for generating call lists of different classes, methods and functions.

- Control Flow Graph (CFG): It shows the flow or computation during the execution of programs or applications. It provides finer 'details' into the structure of the program as a whole, and of the subroutines in particular. Some example of tools for generating CFGs are listed bellow:

  - ***staticfg*** [7]: it is a package that can be used to produce CFGs for Python3 programs. The CFGs that generates can be easily visualised with graphviz and used for static analysis.
  - ***cdm-flowparser*** [8]: it is a module which can takes a file with a python code or a character buffer, parse it and provide back a hierarchical representation of the code in terms of fragments. Each fragment describes a portion of the input: a start point (line, column and absolute position) plus an end point (line, column and absolute position).

- Python Dependency Management (PDM): PDM is essential to the well-being of Python applications. If a Python application relies on third-party libraries and frameworks to function, properly managing them can help us reap the rewards of security, sustainability, and consistency. Some example of tools for generating this type of graphs are listed bellow:

  - ***pigar*** [9]: it is a PDM tool to generate the list of requirements for given Python project. The tool is able to handle different Python versions, is supports Jupyter notebooks. Pigar uses the method of parsing *AST*, rather than regular expression, and can easily extract the dependency library from the document test of exec / Eval parameters and document strings. It is also able to search packages by import name.
  - *pipreqs* [10]: this is another PDM tool that generates dependent files based on the project path, and lists where the dependent libraries are used in the file.

Unlike many other static code analysis tools, **inspect4py** extracts as much as information as possible from a given repository, obtaining different types of information (or software features). And it analyses those for obtaining further knowledge.

[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/

For extracting some software features (e.g. control flow graph, requirements list, etc.), we have employed some of the state-of-the-art static code analysis tools reviewed above (the ones highlighted in bold). In order to decide which tools to integrate in **inspect4py**, we conducted several experiments to compare them, choosing the ones that fit better to our needs. But for most of the software features gathered by **inspect4py** (such as functions, classes, documentation, call list, software invocation, etc.) we have developed our own methods, which are described in the next sections of the paper.

4 inspect4py

As we introduced in Section 3.1, in order to extract the main software features of a given Python repository, **inspect4py** manipulates the *ASTs* nodes of a parsed source code. It uses the `ast.walk()` [11] function, which recursively yields all descendant nodes in the tree. Listing 2 shows a simplified code snippet in which we traverse the *AST* tree to extract all functions names of the *"Example.py"* file shown in Listing 1.

```
import os

path=os.path.join("/User", "/home", "file.txt")

def width():
    return 5

def area(length, func):
    print(length * func())

area(5, width)
```

Listing 1: *Example.py* program that calculates the area on an object.

```
import ast

with open("Example.py", 'rb') as source:
     tree = ast.parse(source.read())

functions_list = [node for node in ast.walk(tree) \
                  if isinstance(node, ast.FunctionDef)]
for f in functions_list:
        print("Function Name: %s" % f.name)
```

Listing 2: Extracting the functions names of *Example.py* file, by checking which nodes are *ast.FunctionDef* instances. This code prints: *Function Name: width* and *Function Name: area*.

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

inspect4py employs a similar (but more sophisticated) method to the one shown in Listing 2. In our case, **inspect4py** uses different *ast* classes (e.g `FunctionDef`, `Call`, `Assign`, `Return`, etc.) to extract automatically all the details of classes, methods, functions, and documentation among other features of a given source code.
inspect4py enables us to represent the code structure of the whole system (Python repository), preserves as many of the semantic details as possible, and stores information about the exact place of a given code piece in the source code. As follows, we describe each software feature extracted by **inspect4py**, which are stored in *dir_info*, *dir_tree_info* and *call_list* dictionaries:

- File: It extracts features at file level. For each of the files found, **inspect4py** stores (inside `dir_info[filePath][file]`) the following features:
  - *path*: path of the file;
  - *fileNameBase*: name of the file without the extension;
  - *extension*: extension of the file; and
  - *doc*: documentation included at file level. The method supports several levels of *docstrings* extraction, such as file's *long_description*, *short_description* and *full*.

  .
- Dependencies: It extracts features at dependency level. For each of the dependencies detected, **inspect4py** stores (inside `dir_info[filePath][dependencies]`) a *dependency* entry with the following features:
  - *import*: module name (e.g `import os`);
  - *from_import*: function or method imported from a module (e.g. `from datetime import datetime`) ;
  - *alias*: each module imported as an alias (e.g. `import numpy as np`); and
  - *type*: we define as 'local' dependency, when the module belongs to the same repository; e.g.`import inspect4py.utils`. On the contrary, we define as 'external' dependency, when the module belongs to an external repository; e.g. `import os`. The possible values are: $<local \| external>$
- Functions: It extracts features at function level. For each of the functions detected, **inspect4py** stores (inside `dir_info[filePath][functions]`) a *function* entry with the following features:
  - *name*: function name;

- *args*: function arguments. Per argument it extracts their *description*, *type*, *default_value*, and if they are *optional* or not.
- *doc*: documentation included at function level. The method supports several levels of *docstrings* extraction [12], such as such as functions *long_description* and *short_description*.
- *returns*: each function return value. Per return value, it extracts their *description*, *type*, or if it is a generator or not (*is_generator*).
- *min_max_lineno*: function starting and ending lines; and
- *call_list*: list of calls per function.
- *store_vars_calls*: variables which store values of functions calls (e.g. `a=functionA()`) or class instances (e.g. `b=MyClass()`). It stores both, the variable name (e.g. `a`) as well as the function which is called (`functionA`).
- *nested*: function definitions can be nested, meaning that a function can be defined and invoked within the context of another function. So, *nested* stores a list of nested functions (if they exit) per function. Each entry inside this list is a *function* entry.

.

- Classes: It extracts features at class and method level. For each of the classes detected, **inspect4py** stores (inside `dir_info[filePath][classes]`) a *class* entry with the following features:
  - *name*: class name;
  - *extend*: class(es) name which this class inherits from.
  - *min_max_lineno*: class starting and ending lines
  - *doc*: documentation included at class level. The method supports several levels of docstrings extraction, such as such as class' long_description and short_description.
  - *methods*[13]: list of methods defined in each class. Per class method, it stores (inside `dir_info[filePath][classes][className][methods]`) a *method* entry with the same features as a *function* entry.
- Main: It extracts features at 'main' level. When **inspect4py** detects this code `if __name__ == __main__` , **inspect4py** stores (inside `dir_info[filePath]`) a *main* entry. We refer to those files as scripts with 'main'. In most cases, a script with 'main' call to a function (i.e `main()`) to run the 'main' functionality of the script, but this is optional.

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

[13] A method refers to a function which is part of a class. A function just refers to a standalone function.

In those cases, we store in `dir_info[filePath][main]` the name of such function.

- Body: It extracts features at 'body' level. If a 'body' is detected, **inspect4py** stores inside *dir_info* a *body* entry as `dir_info[filePath][body]`. We define as 'body' everything that is not a function/class/method or an import within a file/module. We refer to those files as scripts with 'body'. The *body* entry has the following features:
  - *call_list*: list of body calls.
  - *store_vars_calls*: variables which store values of body calls.
- Control Flow: It extracts the control flow per file as a text file or figure (png/pdf/dot), depending the user preferences. We make use of *cdmcf-parser* and *staticfg* tools. Those tools generate the control flow per file.
- Call list: It gathers functions, methods and 'body' calls inside their *call_list* dictionaries (e.g. `dir_info[filePath][body][call_list]`). **inspect4py** also stores the context of each function call (*nested* or *local*). More details about the method for creating those *call_lists* are described in Section 5.
- Requirements: It extracts all the requirements of a given source code (file, directory or repository) by making use of the *pigar* Python package. Per requirement found, it stores (inside `[dir_info][filePath][requirements]`) a *requirement* entry with the following features.
  - *name*: requirement (library/module/tool) name.
  - *version*: version of the requirement.
- File Hierarchy: It extracts the file hierarchy of a given directory or software repository and stores it inside *dir_info_tree*.
- Software invocation: It infers how the software contained within a directory or a software repository can be invoked and tags the most relevant files as package, library, service or script. Details about this features are described in Section 6. For each software detected, **inspect4py** stores a *software_invocation* entry as `dir_info[software_invocation]`, with the following features:
  - *type*: software type.
    The possible values are: $<package \| library \| service | script>$
  - *installation*: command to install a package or library
  - *run*: command to invoke a software

  - *has_structure*: Indicates if a file has 'main', or not. In case of not having a 'main', it indicates if the file has 'body' or not. This feature only applies services, scripts and tests. The possible values are: $<main||body||without\_body>$
  - *mentioned_in_readme*: indicates if the software is mentioned in the readme [14]. Only for services, scripts and tests. The possible values are: $<True|False>$

- Tests: It infers if the software is a test or not. Details about this features are described in Section 6. Per test found, it stores a *test* entry as `dir_info[test]`, with the same features as a *software_invocation*.

- Software type: It infers the type of software contained within a directory or a software repository (i.e package, library, or script). It stores a *software_type* entry as `dir_info[software_type]`. Details about this feature are described in Section 7.

File, *Dependencies*, *Functions*, *Classes*, *Main* and *Body* features are stored per file (using the file's full path) inside *dir_info* to keep track of the different files' features of a given source code (e.g. `dir_info["/User1/home/file1"][functions]`). On the other hand, the last two features, *software_invocation* and *software_type*, need all the previous features in order to be calculated. Furthermore, these two features along with *dir_info_tree* (file hierarchy), are only calculated in case we indicate a directory or a software as input source code to explore.

5 Call List

As we mentioned in Section 4, **inspect4py** generates a call list per function, method or 'body' given a file or a repository (with several files). After reviewing several tools (see Section 3.2) for generating call lists, and given the level of information that we have already extracted from *AST* (e.g functions, classes, main, etc), we decided to create our own method for generating those call lists. This method consists in two main parts:

- detecting all *direct*[15], *argument*[16] , *assignment*[17] and *dynamic*[18] calls made per function, method or 'body'. We have developed the **detection** heuristic to this end. See Section 5.1.

[14] Readme files are usually markdown documents that provide basic descriptions of the functionality of a software component, how to run it, and how to operate it

[15] when we call directly to a function or method using its name.

[16] when we pass a function as an argument/parameter.

[17] when we assign a function or method to a variable, and we use that variable to make the call.

[18] when a function execution is determined only at runtime by the input parameter of another function call.

- completing the calls name with the modules/files where functions or methods have been defined. We have developed the **completion_name** heuristic to this end. See Section 5.2.

Our heuristics allow us to handle higher-order functions calls, modules, function closures, nested functions, and multiple inheritance. Once the call lists are calculated, the information is gathered in the *call_list* dictionary. This information is also available inside each `dir_info[filePath]` across *functions*, *classes* and *body* features (e.g. `dir_info["/User1/home/file1"][functions]["func1"][call_list]`).

Note that we have also developed a benchmark suite (see Section 8) to evaluate our heuristics for generating call lists.

5.1 Detection Heuristic

Our **detection** heuristic consists in two main phases. During the first phase, the heuristic starts by detecting all *direct*, *argument* and *assignment* calls (see Section 5.1.1) per classes methods, functions and 'body' (if we have one). Then, the **detection** heuristic applies the **completion_name** heuristic to complete the detected calls' names (see Section 5.2). At the end of the first phase, the heuristic creates the preliminary version of the *call_list*, which contains several sub-call lists (per class method, function and 'body').
During the second phase, the heuristic detects all *dynamic* calls using the preliminary *call_list*. As a result, the *call_list* is updated (with the *dynamic* calls information) and finalized. In this phase, the **completion_name** heuristic is also applied to complete the *dynamic* calls names. Section 5.1.2 explains in detail our strategy for detecting *dynamic* calls.

5.1.1 Detecting calls

As we introduced before, during the first phase of the **detection** heuristic it detects all *direct*, *argument*, and *assignment* calls of a given source code. The heuristic detects them by traversing and inspect the AST tree to detect which nodes are `ast.Call` instances. A simplified code is shown in Listing 3.

```
import ast
with open("Example.py", 'rb') as source:
    tree = ast.parse(source.read())

calls_list = [node for node in ast.walk(tree)\
              if isinstance(node, ast.Call)]
```

Listing 3: Simplified code for extracting the calls of *Example.py* file. The source code of *Example.py* is displayed in Listing 1.
For each detected call, the **detection** heuristic performs two further checks:

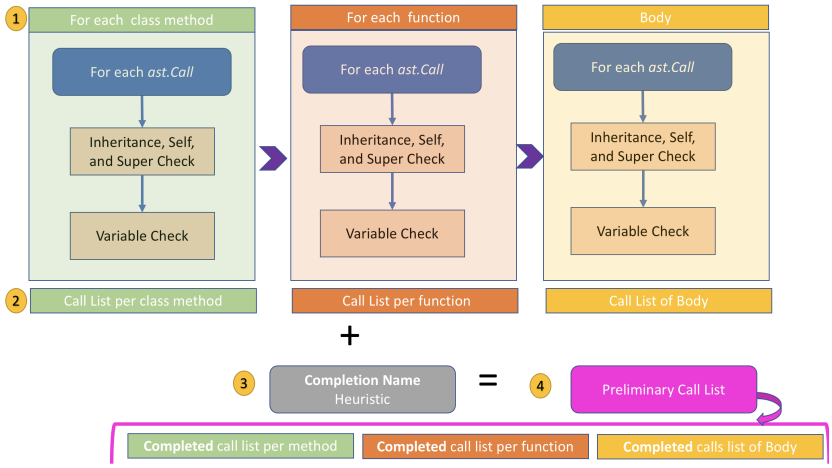


Figure 3: Representation of the first phase of the **detection** heuristic: (1) For each method, function and body, it detects all calls (except *dynamic* calls). (2) As a result, it generates a call list per method, function and body. (3) Then, all call names are completed. (4) Finally the first preliminary call list is obtained, which compromises several sub-call lists.

- Inheritance: As an object-oriented language, Python allows for the creation of classes that inherit attributes and methods from other classes. In order to resolve inherited methods from parent classes we have implemented in the **detection** heuristic a *Method Resolution Order* (*MRO*) algorithm. When we have a call to a class method, this heuristic checks if the class inherits from several base classes (multiple inheritance). If that is the case, it applies the *MRO* algorithm to search across the base classes (this information is captured in `dir_info[file][classes][className][extend]` feature). *MRO* uses a *depth-first left-to-right scheme*, returning the first matching object found during this search. Figure 16 shows an example of multiple inheritance. Furthermore, the heuristic also detects if a call uses:
  - `self.functionName`: in which case the call is stored in *call_list* as `className.functionName`.
  - `super.functionName`: in which case the call is stored in *call_list* as `baseClassName.functionName`. See Figure 16 for an example of this.
- Variables: Functions or methods can be also assigned to variables (e.g. `cube=Cube(2)`). These are also known as *assignment calls*. And using those variables we can call functions as many as times we want. When **inspect4py** extracts functions, methods and body features of a given source code, it also stores in those cases, both, variables and

their assigned functions in the corresponding *store_vars_calls* field (e.g `dir_info['/User1/home/super_test_5'][body][store_var_calls]`). So, this heuristic checks if variables are used as part of calls by checking for matches in the corresponding *store_vars_calls*. If matches are detected, the heuristic updates those calls by replacing the variables names with their assigned functions in the corresponding *call_list*. For example, in the example shown in Figure 16, the variable 'cube' stores the call to the 'Cube' class (e.g. `cube=Cube(2)`). This information is stored in the 'body' *store_var_calls*, so when the call `cube.surface_area()` is detected, this heuristic stores is as `Cube.surface_area()` instead.

All steps performed during the first phase of the **detection** heuristic are captured in Figure 3. At the end of this phase, we to obtain the preliminary *call_list*, which will be updated with the *dynamic* calls (if any) information during the second phase of the heuristic. This is introduced in the next Section 5.1.2 .

5.1.2 Dynamic calls

We refer as a *dynamic* call, when a function(s) execution is not determined when the code is written. It is determined instead at runtime by the input parameter(s) of the function(s)' call. For example, in Listing 1 (which shows the source code of *Example.py*), `area()` calls to `func()`, which is replaced at runtime by `width()`.
During the first phase of the **detection** heuristic, it detects `func()` as a call made in `area()` function. However, this is inaccurate, since we are calling to `width()` when we call `area()` (passing `width()` as an input parameter) and not to `func()`.

Therefore, the second phase of the **detection** heuristic aims for: detecting when functions or methods that are called *dynamically*; and updating their corresponding function or method or body call lists with their functions or methods passed as input parameters. These are described at follows:

- Detecting *dynamic* calls:
  - For each of the calls contained in the preliminary *call_list*, it selects only the calls to functions and methods developed within the repository. We refer to those calls as *local calls* and the rest as *external calls*. Therefore, it skips the rest of *external calls*. For example, given the following call `os.path.join("/User", "/home", "file.txt")`, the method detects that it is an external call to the library `os`, so it does not explore its parameters.
  - For each of the selected *local calls*, it checks their input parameters, and detects if any of them are functions or methods, selecting again only *local calls* with **functions or methods** as input parameters.

Therefore, in this point we have detected all functions and/or methods which are called *dynamically* and for which we will have to update their call lists taking into account their input parameters passed when those calls are made. Given our previous example, this would be only the case for `area()` function and not the case for `width()` function.

- Updating call lists:
  - For each of the functions and methods previously selected (e.g. `area()`) and for each of their input **functions or methods parameters** (e.g. `width()`), the method extracts the completed call name of these **input functions or methods parameters** by applying the **completion_name** heuristic (see 5.2). For our example, it would give us `Example.width` as the completed call name for the input parameter `width()`.
  - The call list of each of the functions or methods previously filtered are next updated with the previous information (**input functions or methods parameters** completed call names). Continuing with our example, it would update the `area` call list with `Example.width`.
  - Finally, this method deletes previous erroneous entries from those call lists. In our example, it would delete `func` from `area's` call list.

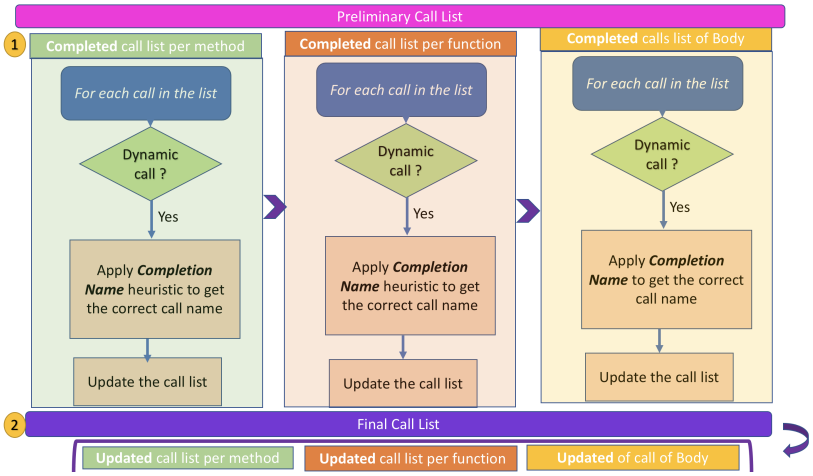


Figure 4: Representation of the second phase of the **detection** heuristic: (1) Given the preliminary call list, it detects all *dynamic calls*, completes their call names and updates the appropriate sub-call list. (2) It obtains the final *call_list*, which is also composed by several sub-call lists.

All steps performed during the second phase of the **detection** heuristic to obtain the final *call_list* are captured in Figure 4. Figure 5 shows how this second

phase of the *detection* heuristic has successfully updated the `area()` call list with `Example.width` instead of `func()`.

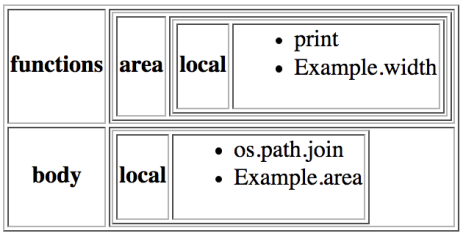


Figure 5: *Example.py* call list. The source code of *Example.py* is displayed in Listing 1.

5.2 Completion Name Heuristic

For each call identified by any of the phases of the **detection** heuristic, the heuristic returns its method or function name ($<className.[methodName]||functionName>$).

Therefore, the aim of **completion_name** heuristic is to find the functions or methods modules (in which they have defined) to complete their call names. As a result, **completion** heuristic returns the following information for each call: $<moduleName>.<className.[methodName]||functionName>$.

Note that there are cases in which the **detection** heuristic already returns the modules name. For example in the code snippet shown in Listing 1, it imports the 'os' module (`import os`), and later it calls a function using the module name (e.g. `os.path.join()`). In those cases, the call name is already completed, and the **completion_name** heuristic does not have to perform any additional steps. However, this is not always the case for many calls, and in most the cases (see Figure 5), the **completion_name** heuristic has to find the correct module for each function or method called.

Python is highly extensible, allowing applications to import different modules. The **completion_name** heuristic is able to identify functions and methods from the different modules that are imported in a given source code, as well as taking into account their resolution order by applying the *MRO* algorithm introduced.
For each call, in which the module name has not been specified, the **completion_name** heuristic perform the following steps (see Figure 6):

- step 1: it checks if the function or method has been defined locally (in the current module/file) by checking the information stored in *methods* or *functions* features.

- step 2: If the search does not return any results, it proceeds to check the modules imported using the information stored in *dependency* feature, filtering out the external imports (a module outside the repository). For each filtered module, the heuristic perform the step 1, checking if the function or module are defined in their *methods* and *function* features.

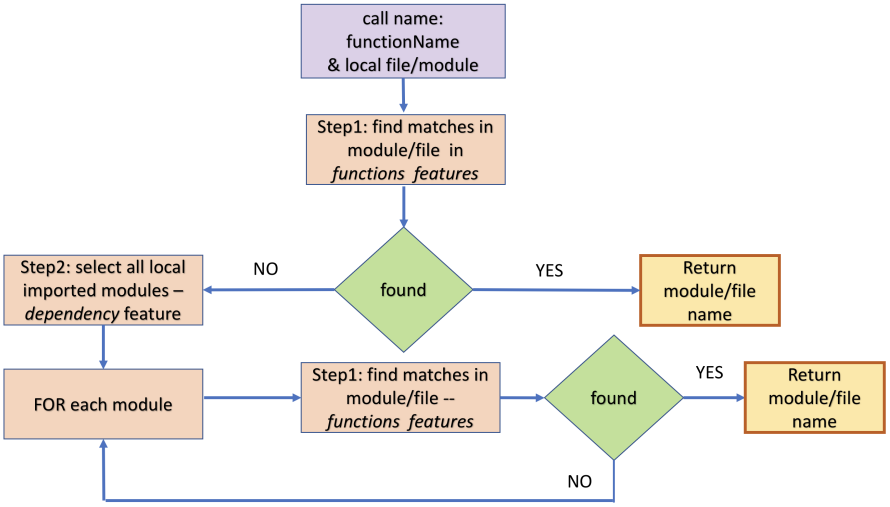


Figure 6: **Completion_name** heuristic for finding the module of the function detected in a call.

6 Software Invocation

Once we have extracted all features of a given directory or software repository, **inspect4py** analyses them further to detect if their software should be invoked as *package*, *library*, *service* or *script(s)*. Note that in case we run **inspect4py** with a single file, this analysis (neither the software type) are performed.

Therefore, we have designed a new **software_invocation** heuristic that explores all the previous information to infer new information about the software contained within a directory or a software repository.

From this section on wards we are going to focus on analysing software repositories. Therefore, we will be using the term **repository** as the input source code for **inspect4py**.

This heuristic uses three structures previously calculated (see Section 4):

- *dir_info*: dictionary containing all the extracted software features.

- *dir_tree_info*: dictionary containing the directory and file hierarchy.
- *call_list*: dictionary containing all calls lists.

The **software_invocation** heuristic consists in:

1. exploring packages and libraries;
2. exploring 'main' files. We refer as 'main' files to those which contain a 'main' function. In those files, we are going to explore for services, tests and scripts.
3. exploring 'body' files. 'Body' files do not contain a 'main' function, but they can call to functions/methods and/or instantiate classes. In those files we explore for services and scripts.
4. exploring files without 'main' nor 'body' to detect scripts. Note that this step is only performed only for repositories that do not have 'main' or 'body' files.

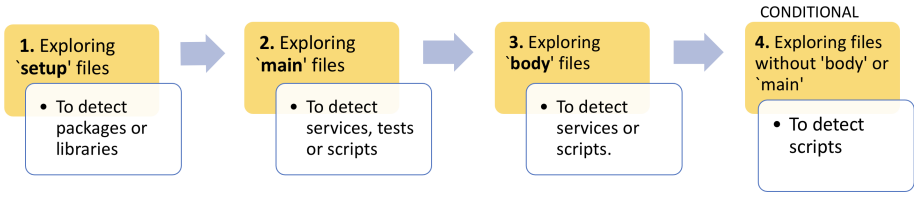


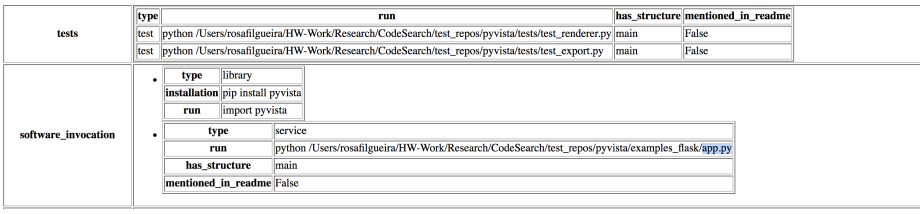
Figure 7: Representation of the **Software_invocation** heuristic. Notice the heuristic always performs the steps from 1 to 3. The step 4 is only performed in case that none packages, libraries, services or scripts with 'main' or 'body' files have been found in the repository

The schema of this heuristic is shown in Figure 7. Furthermore, for each script, service and test the heuristic also checks if it is mentioned in the 'readme' document (in case there is a 'readme' file).

After performing all the previous steps, this heuristic stores the inferred information in two dictionaries:

- *tests*: it stores the information of files with 'main' classified as tests.
- *software_invocation*: it stores the rest of the information about other files.

Both dictionaries (*tests* and *software_invocation*) can have several entries, with the information analysed from different files of a given repository. Figure 8 shows how the **software_invocation** heuristic has detected two tests



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

Figure 8: Information extracted from 'pyvista' repository by the **software_invocation** heuristic.

('test_renderer.py' and 'test_export'.py), a library ('pyvista'), and a service ('app.py') for **pyvista** repository [19].

The next subsections explain in details each of the explorations made by the **software_invocation** heuristic.

6.1 Package and/or library exploration

The **software_invocation** heuristic first looks if the repository has either 'setup.py' or 'setup.cfg' files (or both) inside the repository by making use of *dir_tree_info* dictionary.
In case it finds any of both setup files, it continue to parser them giving priority to 'setup.py'. It only parses 'setup.cfg' if it gets an error parsing the 'setup.py' or if 'setup.py' does not exits.

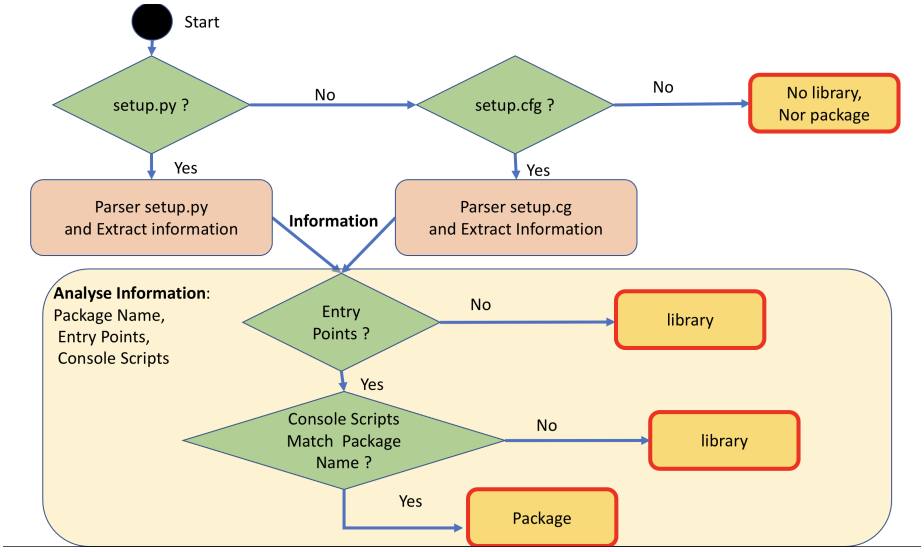


Figure 9: Exploration of 'setup' files for detecting packages and libraries.

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

When the heuristic parses any of the setup files, it obtains: a) the package name ; b) entry points [20]; and c) console scripts [21]. If it does not find any entry points, then it considers that it has found a **library**. If it finds an entry point, then it checks the console scripts names and compare them with the package name (after normalising them). If it finds a console script with a name match (or matching partially) with the package name, then it considers that it has found a **package**. Otherwise, it considers that it has found a **library**.
We have developed two methods for parsing 'setup.py' files within the **software_invocation** heuristic. The first mocks [22] out the 'setuptools.setup' and inspects its arguments to extract name, entry points and console scripts. In some (few) cases, this method is not able to obtain the desired information due to incompatibilities between the `mock` library and the 'setup.py' files . For those cases, we have created an alternative method, in which we read the 'setup.py' file line by line, and extract those values. Furthermore, we have also developed a method for parsing 'setup.cfg' files. Figure 9 shows the steps described above to detect packages and libraries.

Once collected this information, the heuristic stores in the `dir_info[software_invocation]` dictionary an entry with **type:** $<package|library>$, **installation**: `pip install 'package_name'` (for packages) or `pip install library_name` (for libraries), and **run**: `package_name` (for packages) or `import 'library_name'` (for libraries).

6.2 Main files exploration

The *dir_info* structure contains the information about which files are scripts with 'main'. In this point, the heuristic selects only the scripts with 'main', which means the files that have an entry in `dir_info[filePath][main]`. And performs three further explorations as we can see in Figure 10.
As follows, details of each explorations are detailed:

1. services: the heuristic checks if the modules and libraries imported by previously selected files (using the `dir_info[filePath][dependencies]` feature) match with any of our listed services [23]. For each of the matches, the heuristic stores in the `dir_info[software_invocation]` feature an entry with **type: service**, **has_structure: main**, and **mentioned_in_readme:** $<True|False>$.

2. test: the heuristic checks if any of the selected files was previously classified as a test. This classification happens at the time **inspect4py**

[20] An entry point is typically a function (or other callable function-like object) that a developer or user of a Python package might want to use, though a non-callable object can be supplied as an entry point as well.
[21] Console scripts are a type of entry point. They points to functions that developer make available as a command-line tools to users for installing packages
[22] mock library: https://docs.python.org/3/library/unittest.mock.html
[23] list of services: flask, flask_restful, falcon, falcon_app, aiohttp, bottle, django, fastapi, locust, pyramid, hug, eve, connexion

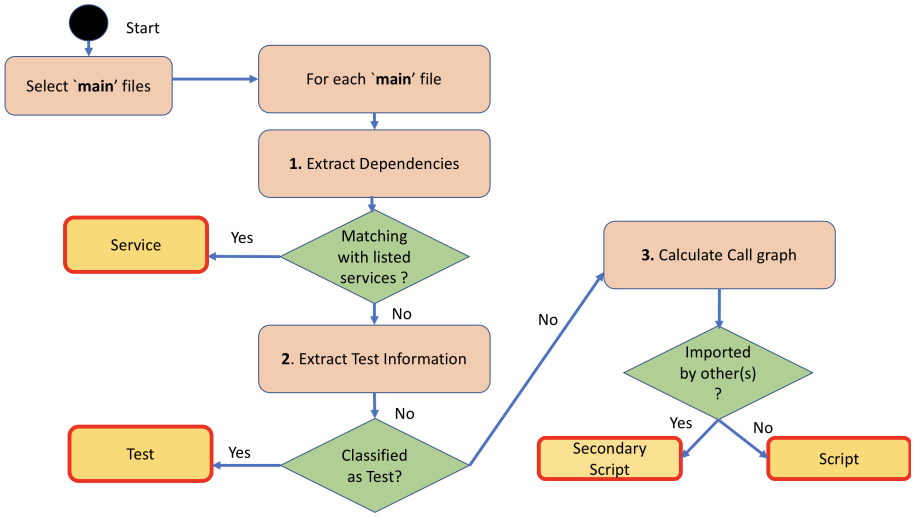


Figure 10: Exploration of 'main' files for detecting services, tests and scripts.

extracts the software features (at the creation of *dir_info* structure). For each file which 'main', it checks if they import any of our listed test libraries [24]. For each of those files previously classified as a test, the **software_invocation** heuristic stores in the `dir_info[tests]` feature an entry with **type: test**, **has_structure: main**, and **mentioned_in_readme:** $<True|False>$.

3. script: the heuristic explores files that have not been classified yet as a service or tests. And then, it explores the relationship between them to detect which scripts with 'main' are imported by others (directly or indirectly). Note that if a script with 'main' is imported by another script with 'main' (directly or indirectly), then the imported script with 'main' is acting as a normal script.

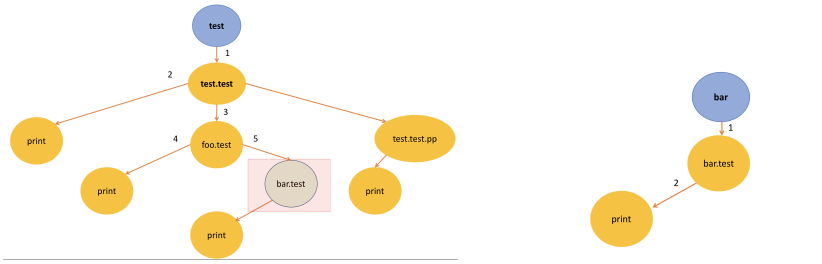
   In order to explore those relationships, the heuristic uses the *call_list* structure to create a *call_graph* on-the-fly per selected file[25]. A *call_graph* represents not only the calls that are made by each file, as well the calls that each of those calls make, which can imply other files. *Call_graph* nodes have this naming schema: $<module\_name>.<class\_name.[method\_name]||function\_name>$.

   For each of the scripts with 'main', it searches their *call_graph* (see Figure 11) by employing Deep-First-Search(DFS) algorithm. The aim is to find a node which its module name matches with any of other scripts with 'main'. In the example shown in Figure 11, we have the *call_graphs* of two

[24] list of test libraries: unittest, pytest, nose, nose2, doctest, testify, behave, lettuce

[25] We currently do not store or pre-calculate the *call_graph*, but in the future we plan to add this functionality.

scripts with 'main', *test.py* and *bar.py*. When we search in the *call_graph* of *test.py* 11.a, the DFS algorithm finds in the 5th iteration a match with *bar.py* in the form of *bar.test*. Note that we are looking for module name matching, so the DFS algorithm only looks in the left side for each node name. On the other hand, when we search in the *call_graph* of *bar.py* we can not not find a match with *test.py*. In order to limit our



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

Figure 11: Generated *call_graphs* for *test.py* and *bar.py*, which are two scripts with 'mains'. The DFS algorithm finds that *test.py* has a relation with *bar.py*, but not other way around. The blue nodes represent modules/files (e.g. test or bar), while the yellow nodes represent calls.

If the heuristic finds that a script with 'main' calls to another script with 'main', it marks the script as a 'secondary' (e.g. in this case *bar.py* is marked as a secondary). Finally, the heuristic stores an entry in our `dir_info[software_invocation]` per script with 'main', with **type: script**, **has_structure: main**, **mentioned_in_readme:** $<True|False>$, **import_mains**: [list of scripts with 'main' that it imports] and **imported_by**:[list of scripts with 'main' that it is imported by]). Following with the previous example, the *software_information* generated can be seen in Figure 12.

6.3 Body files exploration

The next step of the **software_invocation** heuristic is to explore files with 'body'. Therefore, it selects the files that have an entry in `dir_info[filePath][body]`. And performs two further explorations with them, as it is shown in Figure 13.
Both steps are detailed bellow:

1. The main reason for exploring scripts with 'body' is to find more services. And we do so, by applying the technique introduced in Section 6.2 for detecting services, but using the files with 'body' previously selected. For each match, the heuristic stores in `dir_info[software_invocation]` an entry with **type: service**, **has_structure: body**, and **mentioned_in_readme:** $<True|False>$.

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

Figure 12: Example of information extracted by the software invocation method using 'test/test_files/test_multiple_mains_indirect' directory. Here we can see that two scripts with mains have been detected ('test.py' and 'bar.py'), and 'bar.py' is imported by 'test.py'.

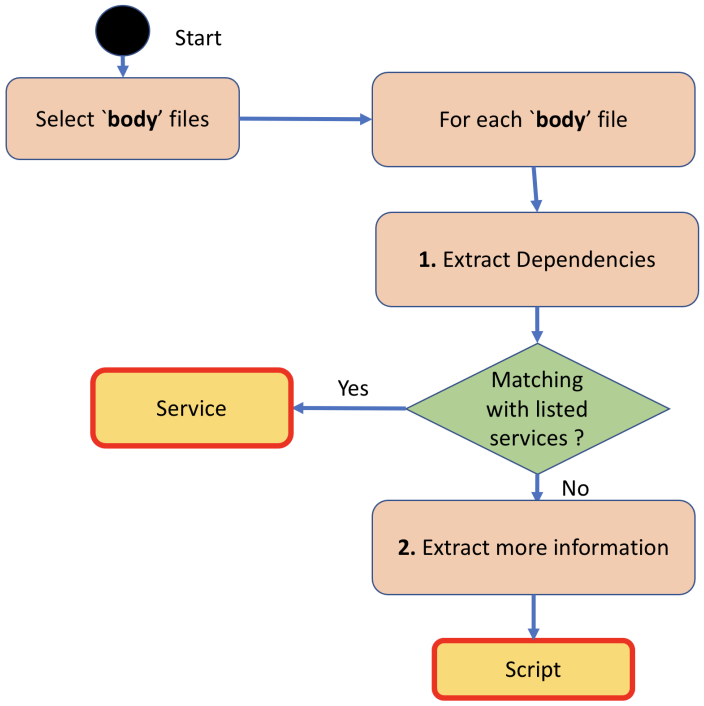


Figure 13: Exploration of 'body' files for detecting services and scripts.

2. Only, in the case that the heuristic has not found at this point any packages, library or services (in the files with 'main' or with 'body'), then, it continues the exploration by extracting the information of this type of files. For each them, it stores in `dir_info[software_invocation]` an entry with the **type: script**, **has_structure: body**, and **men-**

tioned_in_readme: $<True|False>$.

6.4 Rest of files exploration

Finally, there are repositories which only contain files without a 'main' function or a 'body'. Those are usually a collection of functions or classes.

Only in those cases, the heuristic explores them and creates an entry in our `dir_info[software_invocation]` for each of these files with the following information: **type: script**, **has_structure: without_body**, and **mentioned_in_readme:** $<True|False>$. Figure 14 represents this last exploration performed (in some occasions) by the **software_invocation** heuristic.

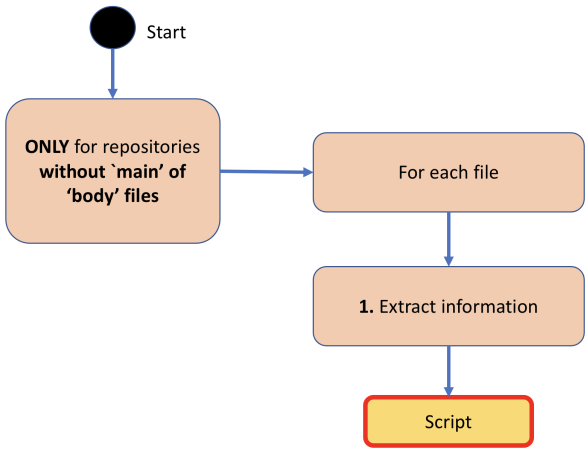


Figure 14: Exploration of files without 'body' nor 'main' to detect scripts.

7 Software Type and Ranking

Once our file analysis finishes, we rank our invocation results by estimating their relevance. We have a scoring function, which assigns weights to each result based on the features resulting from the previous analysis. Different features have different weights, according to the common practices observed empirically by the authors in over 90 code repositories. The scoring function is defined as:

$$score(x) = w_{lib} * lib(x) + w_{readme} * readme(x) + + w_{service} * service(x) + + w_{main} * main(x) + w_{body} * body(x) \quad (1)$$

where x denotes the invocation being scored, *lib(x)*= 1 if the code repository was identified as a library or package (zero otherwise), *service(x)* = 1 if x is identified as a service, and *main(x)* and *body(x)* equal to 1 if x has a main function or only body (they equal to zero otherwise). We also consider if a service or script

is mentioned in the README file (*readme(x)* = 1), as this indicates that the it was identified as significant by the authors. The weights associated with each of these features (represented with a w, like w_{lib}), are highest for package or libraries, followed by services and finally, scripts. If an executable file has a *main* function, its weight will be higher than a script with just *body*.

Listing 4: JSON snippet showing two invocation alternatives

```
"software_invocation": [
    {
      "run": [
        [
          "pylode ",
          "pylode.cli:main,"
        ]
      ],
      "type": "package",
      "installation": "pip install pyLODE",
      "ranking": 1
    },
    {
      "type": "service",
      "run": "python pyLODE/pylode/server.py",
      "has_structure": "body",
      "mentioned_in_readme": true,
      "ranking": 2
    }
  ],
  "software_type": "package"
```

Once all invocation alternatives have been assessed, we sort them by their score in descending order. To improve readability, we create a ranking in ascending order, where the first position is assigned to the entry with the highest score. The first ranked element is returned as the main software type. If two invocation alternatives have the same score, they are assigned the same ranking number. Listing 4 shows a simple example of a package where the main invocation is through the command line, but that also has an alternative invocation as a service. Inspect4py detects the main command to run the package, (available in the *setup.cfg* found in the code repository), together with another a script (*server.py*), mentioned in the main README.

8 Call List Evaluation

We have created a benchmark suite for testing and evaluating the heuristics presented in this paper for generating *call_lists*, as we can see in Table 1.
The results show that **inspect4py** correctly generates the *call_list* from all benchmarks. From this table, we have selected three benchmarks to show in this paper along with their *call_lists* generated by our heuristics. The code snippet of Figure 15 illustrates how our heuristics are able to identify local and imported functions. While the code snippet shown in Figure 16 illustrates how inheritance (simple and multiple) is successfully handled. Finally, Figure 17 shows the third

Category	#Number	Description
Inheritance	6	resolution of inherited methods
Class	6	method functions calls
Imports	4	imported modules, functions classes
Variables	3	assignment of functions to parameters
Nested	2	nested functions calls
Function	3	vanilla function calls
Dynamic	12	dynamic calls
Argument	1	passing functions as arguments
Alias	2	importing functions as alias

Table 1: Benchmark suite for testing **inspect4py** *call_lists*.

code snipped which contains *dynamic* calls. From these examples, we can check that the generated *call_lists* are correct.

```
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 file

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) Call list Example 1

Figure 15: *Import Benchmark*. Generated call list for each of the functions, methods, and body for the python code contained in test_import.py. The example makes use of another two files: test_functions.py and test_classes.py. Here we can see how the names have been filled accordingly, being able to identify if a method or function belongs to an imported module or not, capturing the corresponding module each time.

```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 file

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) Call list Example 2

Figure 16: *Inheritance Benchmark*. Generated call list for each of the classes methods and body in super_test_5.py. Notice how in this example, in which *Cube* class inherits from two classes (*VolumeMixn* and *Square*), the call list is able to find the correct 'volume' method when we call to *cube.volume()*. Also in this example, we can see how 'super' is replaced by the correct class name.

```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 file

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

(b) Call list Example 3

Figure 17: *Dynamic Benchmark*. Generated call list for each of the functions in test_dynamic_import.py. Notice how we are able to infer that the `func_3` calls to `test_dynamic_funcs.func_1`, instead of calling to `func`. Therefore, it is able to infer which functions are passed as arguments.

9 Software Type Evaluation

An initial assessment of inspect4py compares software type and invocation results against a manually annotated corpus.

9.1 Annotated corpora

We have created two different corpora to assess our approach. For the main software type detection, we selected a corpus of 95 different Python code repositories (distributed in 24 packages, 27 libraries, 13 services and 31 scripts).[26] All

[26] Benchmark is available at: `https://doi.org/10.5281/zenodo.5907936`

Software type	Precision	Recall	F1-score
Package	1	0.916	0.956
Library	0.93	1	0.9637
Service	1	1	1
Script	0.967	0.967	0.967

Table 2: Results for software type classification.

repositories have been annotated by each of the authors, separately, and compared until agreement was reached. The repositories are heterogeneous in scope and domain, and range from research repositories used in Machine Learning[27] to popular community tools such as Apache Airflow[28] or domain-specific libraries (e.g., Astropy[29]). We also included repositories developed at the author home institutions, with less documentation available or in development, in order to obtain a wider representative sample.

The second corpus was designed to assess the ranking results, and it was generated as a subset of the first one. For all repositories with multiple invocation methods, the authors manually annotated the most relevant files, based on the repository structure and documentation. As a result, 44 repositories were annotated.

9.2 Results

Table 2 shows an overview of our results for main software type classification, for each of the categories supported. Our heuristics, based on best practices for Python application development, yield over 95% F1-score for all categories. The only errors we encounter occur when developers step outside the common practices, such as creating custom metadata files for their libraries.

For assessing our ranking results, we selected the normalized discounted cumulative gain (NDGC) [7], a metric used in information retrieval to assess ranking quality. NDGC ranges from 0 (min) to 1 (max). Our aggregated ranking amounts to 0.87, which is considered satisfactory for a preliminary evaluation.

10 Related Work

A number of static code analysis tools have been developed for extracting code metadata, documentation or features [3]. From a Machine Learning perspective, [8] describes a survey of techniques for code feature extraction, in order to train source code models. Tools like libsa4py [9] create features from code that would complement the information already extracted in inspect4py.

Other tools present some overlap with the functionality included in our framework. For example, pydeps[30] and modulegraph2[31] extract module dependencies,

[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/`

libraries like *pydoc*[32] generate HTML code documentation, libraries like *pigar*[33] extract code requirements, and packages like *pyan* [5] and *pycg* [6] generate call graphs for Python code using static analysis, including high order functions, class inheritance schemes and nested function definitions. Inspect4py builds on some of these tools (e.g., pygar), and integrates all these functionalities under a single framework, using a unique representation for all of their results.
To the best of our knowledge, there are no frameworks for detecting software type and invocation methods.

11 Installation and Execution

Inspect4py can be installed through pip (`pip install inspect4py`) and Docker. To invoke the tool with basic functionality (i.e., extract classes, functions and their documentation), one needs to run the following command:

```
inspect4py -i <input file.py | directory>
```

This command can be qualified with different options,[34] in order to perform different functionalities. For example, the following command extracts classes, function and method documentation and also stores the software invocation information (flag *-si*):

```
inpsect4py -i repository_path -o out_path -si -html
```

12 Conclusions and Future work

This paper introduces inspect4py, a static code analysis framework designed to extract common code features and documentation from a code repository in order to ease its understanding. A preliminary evaluation shows promising results evaluating Python code repository types, as well as identifying and ranking alternatives ways of running them (thus saving time to potential users).
We are using inspect4py in combination with software metadata extraction tools [10] to suggest recommendations for completing README files.[35] In addition, we are planning to use our results for facilitating function parallelization and composition, as well as to compare alternative representations of software when finding similar code.
As for future work, we are expanding inspect4py to 1) improve our evaluation results, by annotating additional number of repositories, 2) incorporating new feature extraction tools (e.g., libsa4py), and 3) improving our invocation detection to include illustrative parameter examples.

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

References

[1] A. Von Mayrhauser, A. Vans, Program comprehension during software maintenance and evolution, 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. Eng. 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: The 33rd International Convention MIPRO, 2010, pp. 418–422.

[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: Practical call graph generation in python, 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 techniques, 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 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), 2021, pp. 585–589. doi:10.1109/MSR52588.2021.00079.

[10] A. Kelley, D. Garijo, A Framework for Creating Knowledge Graphs of Scientific 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