Contents

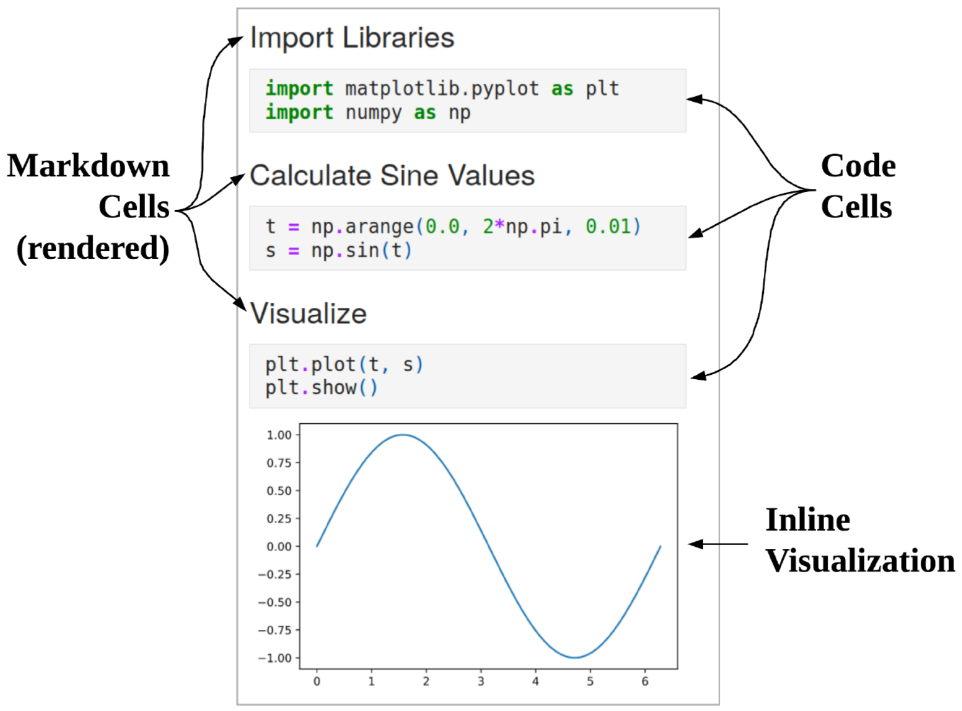
1 Introduction

`Python`, being an open-sourced language, is supported by a large community of developers and high-quality documentation. The easy readability of `Python` code allows for collaborative programming. The multitude of machine learning and data analysis specific libraries ease the programming process for researchers, data scientists, and other related professions. `Python` is used extensively in various disciplines of Machine Learning. From Data Science and Computer Vision to Deep Learning, the complex theory of Mathematics such as Linear Algebra, Probability, Statistics and Calculus can be implemented using `Python's` libraries and implicitly defined functions.

Literate Programming defines the programs as works of literature [Knu84]. In literate programming the focus is on writing computer programs with a narrative structure that enhances code comprehension and sharing. Literate programming intends to write computer programs as a document. This approach facilitates enhanced understanding of the code and allows for sharing of the code files amongst multiple users. Such programs are more comprehensible as the concepts are introduced in an order that is best suitable for human understanding.

Jupyter notebook is an open-source literate programming tool. Jupyter notebooks provide a programming environment for developing Literate Programming documents augmenting rich text, code snippets, and visualizations [VB21]. A Jupyter notebook comprises of 3 types of cells: markdown, code, or raw. Markdown cells display the rich-text representing the explanatory text. The code cells contain the executable code and the raw cells are used for rendering different code formats into HTML or LaTeX. Jupyter notebooks are widely adopted by data scientists and machine learning engineers [Per18]. Fig. 1.1 illustrate a Jupyter notebook with 3 markdown cells containing HTML H3 headers and a visualization of the output of the last cell. The literate programming approach merges the code with the rich-text explanatory texts and inline visualizations.

Jupyter notebook's literate programming approach is intended to provide computational results with sufficient self-explanatory texts for understandability and reproduction. However, recent studies have proven that the implementation of literate programming principles is lacking in practice. This paper puts forth a proposal for the thesis work to develop a Proof-of-Concept Jupyter notebook annotator called HeaderGen with an extended version of a static analysis tool for `Python` known as PyCG. PyCG analyses the `Python` code using an intermediate representation and generates an Abstract Syntax Tree (AST) which is then used to build a Call Graph. HeaderGen identifies the call-sites and generates descriptive markdown header cells as annotations to the corresponding code cells of the Jupyter notebook.



Import Libraries

```
import matplotlib.pyplot as plt
import numpy as np
```

Calculate Sine Values

```
t = np.arange(0.0, 2*np.pi, 0.01)
s = np.sin(t)
```

Visualize

```
plt.plot(t, s)
plt.show()
```

Figure 1.1: A Jupyter Notebook [VB21]

1.1 Structure

This section is followed by an introduction of `PyCG` and `HeaderGen` as the background in Chapter 2. Chapter 3 describes the limitations and missing features in the implementation of the above mentioned tools. In Chapter 4, The goals of the thesis along with the solution ideas are discussed. Chapter 5 lays down the preliminary thesis structure. Finally, Chapter 6 represents the time-plan of the thesis.

2 Background

2.1 PyCG

Call Graphs illustrate relationships between different subroutines of a computer program. A call graph plays an important role in various tasks such as identification of procedures that are never called, detection of code injection attacks and vulnerability propagation analysis, among others. Generation of an efficient call graph for high-level dynamic programming languages such as `Python` can be challenging. This is due to the presence of high-order functions, dynamic and meta-programming features such as *eval()*, libraries and modules, etc.

The main goal of generating a call-graph through static analysis of such dynamic programming languages is completion. However, deduction of all the facts in a program for dynamic languages is accompanied with a performance cost. Therefore, the implementation of such approaches is restricted due to scalability [KLDR15]. *Pyan* [F$^+$21] extracts the program's Abstract Syntax Tree to generate the respective call graph. However, it has limitations related to the handling of module imports and inter-procedural code flow. *Code2Graph* [GTL18] uses *Pyan's* call graphs for visualizations, which come along with the existing drawbacks. *Depends* [ZW21] extracts syntactic information from the code to generate a call graph. However, it does not handle functions passed as variables to other functions or assigned to variables, thus fails with respect to the high-order functions.

`PyCG` is a tool developed for static analysis of python code [SSL$^+$21]. `PyCG` builds a Call Graph by identifying relationships among program identifiers and performing context-insensitive inter-procedural analysis on a simple intermediate representation for Python. The approach is implemented as a two-step process:

1. **Assignment Graph Computation** An assignment graph depicts the assignment relationships among the program identifiers. In addition, it also captures the program's inter-procedural flow. `PyCG` can identify high-order functions, multiple inheritance, modules, etc.

2. **Call Graph Generation** All the functions that can be called by the callee variables are deduced from the assignment graph. The results are then used to construct a call graph.

2.2 HeaderGen

Programming platforms such as Kaggle use Jupyter notebooks as their coding environment. Jupyter notebooks also serve as the guiding notebooks for programming tasks for beginners. Therefore, maximizing code comprehension in such notebooks could yield high understanding and raise users affinity towards literate programming. However, studies prove that the coding practices used by data scientists and data analysts deflect from literate programming. For instance, Jupyter notebooks often contain deprecated APIs and unused variables [WLZ20]. Furthermore, [RTH18] observed that approximately a quarter of Jupyter notebooks used for their study did not have markdown cells. The lack of markdown cells points to an inefficient narrative structure.

To address such limitations, studies such as [PMBF19] found that more than 90% of 1.4 million Jupyter notebooks under observation used headers in markdown cells for creating a narrative structure. HeaderGen is an automated Jupyter notebook annotator and is being developed as a Proof-of-Concept currently [VB21]. HeaderGen statically analyzes the notebook and generates descriptive markdown header cells. Headers refer to the annotations to the following code cells. HeaderGen annotates Jupyter notebooks implementing Python code as Python is used in a vast majority of notebooks. In addition, header cells are associated with a clickable *Table of Contents* for effortless navigation through the notebook. The *Table of Contents* is positioned at the top of the notebook.

HeaderGen focusses on annotating notebooks implementing Machine Learning algorithms and models using `Python`. A typical Machine Learning program flow consists of the following three phases:

1. **Data Phase** refers to the data pre-processing phase. This includes the ingestion, validation, scaling of the dataset, among other steps required before analysing the data.

2. **Model Phase** refers to the training of a Machine Learning model on a training set extracted from the dataset. The model is further validated and evaluated against a test dataset.

3. **Post Training Phase** refers to the phase where the trained model is deployed or saved for future use, as per the requirements.

HeaderGen implements an *Function-to-ML-Phase-Mapping* approach and annotates the code cells with the corresponding Machine Learning phases. For example, a call to *"keras.layers.LSTM"* is mapped to a single "MODEL TRAINING" phase and the call to *"numpy.array"* is mapped to "DATA INGESTION, DATA PROCESSING" phases. Similarly, all the phases depicted in Fig. 2.1 are mapped in the function-to-ML-Phase-Mapping. Since the existing static analyzer in HeaderGen is an early prototype, incorporating `PyCG` into HeaderGen will result in a more robust and accurate static analysis of the notebooks.

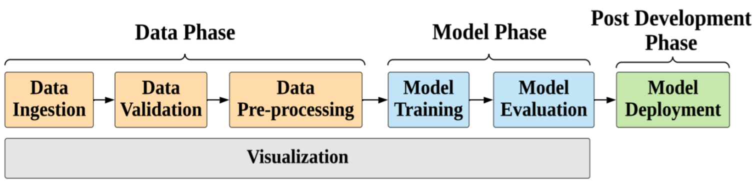


Figure 2.1: Machine Learning phases [VB21]

3 Problem Description

3.1 Limitations of PyCG

Cell-Sensitive Call-Site Identification. PyCG generates a call graph for a given Python code through a context-insensitive inter-procedural static analysis based on the intermediate representation defined for Python. For instance, let us consider a Python code in Listing 3.1. The call graph extracted using `PyCG` can be illustrated in the following Listing 3.2.

```
# %%
class MyClass:
def func(self):
pass

class MyClass2:
def func(self):
pass

# %%
a = MyClass()
b = a.func
b()

# %%
a = MyClass2()
b = a.func
b()

```

Listing 3.1: Python Code: Re-initialization of variable names

```
{
  "main": ["main.A.func", "main.B.func"],
  "main.A.func": [],
  "main.B.func": []
}

```

Listing 3.2: PyCG generated call-graph JSON for the code in Listing 3.1

PyCG extracts the call-sites related to dynamic programming features of Python to build a call graph for the given code. However, the Python code in the Listing 3.1 uses the same variables to call two different functions with the same name, *func*, from two different classes *A* and *B*. The first goal of the thesis is to identify such patterns where the variables, classes, and functions are re-used with the same name but different definitions and/or implementations. This approach will conclude the frequency of such patterns in Jupyter notebooks with which the programmers use same identifiers in different code cells based on their logic and business requirements. The identification of the cell that is calling a function will provide enhanced overview and increase understanding of the functions call sites. One way to achieve this is by including the line number, of the function's call-site in the code, as a part of the analysis domain.

Missing support for `eval()` and `walrus`. In addition to the above mentioned limitation of PyCG, the tool does not support dynamic code-generation schemes at the current stage. One such built-in function of Python is the `eval` function. The `eval` function takes in a string expression as an argument, parses the expression, and executes the code in the python expression within the program. For example, let us consider the Python code in Listing 3.3 calling the built-in *eval()* function. The *eval()* function takes in the expression *"func()"* as a parameter and executes the code, i.e. the call to the *func()* function, after parsing the expression. The expected call-graph for the given code can be shown in Listing 3.4. However, the current implementation of PyCG generates the call-graph shown in Listing 3.5. Evidently, PyCG currently does not create the required nodes for the *eval()* function call.

```python
# %%
def func():
  pass

# %%
eval("func()")

```

Listing 3.3: Python Code: Call to eval()

```json
{
  "main": ["main.func"],
  "main.func": ["<builtin>.eval"],
  "<builtin>.eval": []
}

```

Listing 3.4: Expected call-graph JSON for the code in Listing 3.3

```
{
  'main': ['<builtin>.eval'],
  'main.func': [],
  '<builtin>.eval': []
}

```

Listing 3.5: PyCG generated call-graph JSON for the code in Listing 3.3

Furthermore, the `walrus` operator has been introduced in `Python 3.8`. This operator is denoted as `:=` and allows the user to introduce and return a value within the same assignment expression. To understand the problem with the **walrus** operator, let us first look at the code in Listing 3.6. The Python code initializes the variable *walrus_op* with the return value *return_true* of the function *return_func()* in line number 8. The following line number 9 calls the function *return_true()* through the variable *walrus_op*. Therefore, the expected call-graph as generated by PyCG is shown in Listing 3.7. The same logic can be implemented using the newly introduced **walrus** operator as shown in Listing 3.8. The expected call-graph remains the same as in Listing 3.7. However, PyCG generated the call-graph for the implementation with the **walrus** operator as shown in the Listing 3.9. Notably, the call to the function *return_true()* is not identified by PyCG and hence, the generated call-graph does not contain the corresponding node.

```
 1  def return_true():
 2  return True
 3
 4  def return_func():
 5  return return_true
 6
 7  # %%
 8  walrus_op = return_func()
 9  walrus_op()
10
```

Listing 3.6: Python Code: without walrus operator

```
{
  'main': ['main.return_func', 'main.return_true'],
  'main.return_func': [],
  'main.return_true': []
}
```

Listing 3.7: PyCG generated call-graph JSON for the code in Listing 3.6

```
def return_true():
return True

def return_func():
return return_true

# %%
if (walrus_op := return_func()):
walrus_op()
```

Listing 3.8: Python Code: with walrus operator

```
{
  'main': ['main.return_func'],
  'main.return_func': [],
  'main.return_true': []
}
```

Listing 3.9: PyCG generated call-graph JSON for the code in Listing 3.8

Analysis of External Libraries. Another existing problem with the current implementation of PyCG is the inability to identify the function calls made through the external libraries. For instance, Listing 3.10 shows a Python code in which the *regressor* object of the *keras.models.Sequential* class calls an implicitly defined function *add()*. The *add()* function further takes in an object of the *keras.layers.Dropout* class as its parameter. However, the PyCG generated call-graph, as depicted in the Listing 3.11, can not capture the call to the *add()* function. This limitation leads towards an observation that the calls to external libraries are not recognized by the current PyCG implementation.

```python
from keras.models import Sequential
from keras.layers import Dropout

regressor = Sequential()
regressor.add(Dropout(0.2))
```

Listing 3.10: Python example: External Libraries

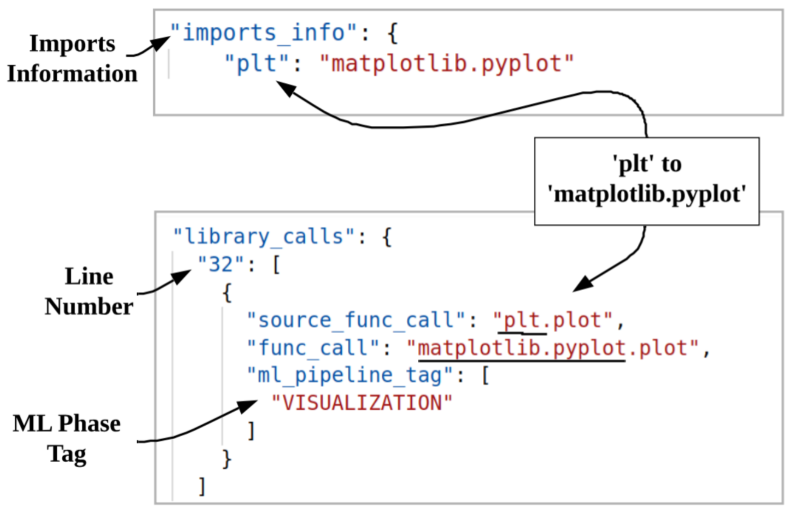
```
{'keras.layers.Dropout': [],
  'keras.models.Sequential': [],
  'main': ['keras.layers.Dropout', 'keras.models.Sequential']}

```

Listing 3.11: PyCG generated call-graph JSON for the code in Listing 3.10

3.2 Enhancements to HeaderGen

Primitive Static Analysis. HeaderGen aims at creating a narrative structure for ML notebooks. The task required for the generation of narrative markdown header cells is the identification of all the call-sites and determination of the libraries and modules they belong to through static analysis of the Python code. The process starts by taking a Jupyter notebook as an input



```
"imports_info": {
    "plt": "matplotlib.pyplot"

"library_calls": {
  "32": [
    {
      "source_func_call": "plt.plot",
      "func_call": "matplotlib.pyplot.plot",
      "ml_pipeline_tag": [
        "VISUALIZATION"
      ]
    }
  ]
```

Figure 3.1: Analysis Report from HeaderGen [VB21]

and converting it into a Python script. In the following step, an Abstract Syntax Tree (AST) is generated using the built-in AST library. The assignment nodes in the generated AST are then analyzed for recognizing the respective libraries and tagging the ML phases. Fig. 3.1 illustrates the analysis of a sample notebook with a call to the matplotlib.pyplot.plot function. Evidently, the static analyzer of HeaderGen is an early prototype and misses various complex features of `Python`. Incorporating PyCG as the back-end analyzer for HeaderGen can significantly increase the static analysis efficiency and expand the effective range of HeaderGen's categorization and recognition of function calls.

Early Stage Function-to-ML-Phase Mapping. HeaderGen relies upon the the *function-to-ML-phase* mapping for the accurate realization of the ML phase being employed in a code cell and generating a markdown header for the respective code cell. However, the ML phases and the respective mappings defined at the present are not complete for real-world use-cases. For instance, the mapping exists for only a few libraries namely: keras, math, matplotlib, numpy, pandas, sklearn, and tenserflow. Therefore, there is a scope to extend and improve the existing *function-to-ML-phase* mapping.

4 Thesis Goals

The overall goal of the thesis is to develop `HeaderGen` as a robust cell-header generator with `PyCG` as the efficient static analysis tool implemented as the backend. To achieve this goal, the thesis work can be divided into three main objectives with various sub-goals. The three goals can be summarised as:

1. Improving the existing PyCG implementation.
2. Incorporating the extended PyCG as the backend with HeaderGen
3. Performing a usability study to assess the effectiveness of the final product

These three goals can be further divided into different sub-goals. This sections further describes the goals in details along with their respective sub-goals. In addition, the solution ideas to achieve these goals are also elucidated with each subsection.

4.1 Extending PyCG

4.1.1 Cell-Sensitive Call Site Identification

Initial evidence shows that programmers tend to use same identifiers for variables, classes, and functions in different cells throughout the notebook. As discussed in the previous sections, `PyCG` performs a context-insensitive static code analysis. The problem described in section 3.1 involves two variables used twice. Although these variables, `a` and `b`, have the same name, they instantiate the objects of two different classes `A` and `B` and call their respective functions. Existing analysis output of PyCG does not include information about the line numbers of call sites, therefore, the goal is to add the line numbers of the function calls from the code to the existing implementation of PyCG. The resulting analysis will enable linking the call site to the respective cell.

Solution Idea

The solution here is to modify the pre-processor of PyCG to include the line number information and add an additional cell-sensitive call site processor, similar to the cgprocessor.py in PyCG, to generate the information required.

4.1.2 Analyzing External Libraries

Although PyCG can theoretically analyze external libraries, it has certain engineering limitations. Current PyCG implementation can not analyze imported modules that are not in the same folder, that is, it cannot analyze external libraries that are installed at the system level. The goal is therefore to add the capability to recognize function calls from external libraries.

Solution Idea

When a function from a class belonging to an external library is called, PyCG fails to recognize the library name and does not report it as the output in the generated call-graph. The solution idea is to resolve this limitation and make PyCG capable of identifying external libraries as a part of the function calls. This can be achieved by implementing a class in the PyCG machinery that analyzes the calls made to the external libraries. These function calls will be added to the AST as a node as the AST serves as the ground for generating the call-graph.

4.1.3 Optional: Analyzing eval function and walrus operator

The *eval()* function is one of the Python's built-in functions and it allows the programmer to execute Python expressions dynamically. These expressions can be in the form of a compiled-code-based input or a string-input. On the other hand, the *walrus* operator allows the user to execute two operations with a single command. The first operation is to assign a value to a variable. The second is to return that value. The current implementation of PyCG does not reason about these operators, therefore, the goal is to design and implement a proof-of-concept analysis to include *eval()* function and the *walrus* operator.

Solution Idea

The solution idea is to add initial support for dynamic function calls. PyCG will be extended in order to support the *eval* function calls and the *walrus* operator commands and include corresponding nodes in the call graph. This approach can be implemented by adding support for both *eval()* and *walrus* as individual classes in the PyCG machinery. The approach is to modularize the implementation so as to facilitate easy identification and implementation of such dynamic features in the future.

4.2 HeaderGen: Automated Cell Header Generation

4.2.1 Hybrid: PyCG Static Analysis With HeaderGen

HeaderGen annotates the cells after the identification of call sites in Jupyter notebooks through static analysis. HeaderGen identifies the call sites and traces the function calls back to their library of origin. As the first step, a Jupyter notebook is taken as an input and converted into a native `Python` script. Thereafter, the identification of call sites is done by generating an Abstract Syntax Tree (AST) for the converted `Python` script and iteratively analyzing the AST.

However, the static analyzer implemented in HeaderGen to identify these call sites is an early prototype. Therefore, incorporating PyCG as the static analyzer for HeaderGen will enhance the precision of call-site identification.

Solution Idea

HeaderGen currently employs the built-in AST library to extract the Abstract Syntax Tree from the python script. The idea to achieve the goal is by replacing the built-in AST library with the PyCG static analysis. This is to be achieved by using the abstract syntax tree generated by the PyCG backend for the analysis in HeaderGen.

4.2.2 Enhanced Function-To-ML-Phase Mapping

HeaderGen is being developed to annotate code cells with respective Machine Learning phase according to the code snippet contained in the respective code cells in a Jupyter notebook. HeaderGen recognizes the library related to a function call after analyzing the generated AST. To achieve this, a *function-to-ML-phase* mapping is implemented in HeaderGen. The goal is to improve this mapping and include more libraries for identification and annotation.

Solution Idea

The existing *function-to-ML-phase* mapping will be improved by conducting surveys and interviews. The solution idea for this goals involves literature research on related studies and surveys for classing of the Machine Learning phases and tagging them to the Python libraries. The results of the surveys and interviews will then be incorporated to add more ML libraries to HeaderGen and tag respective ML phases to each of them.

4.3 Evaluation

As HeaderGen is a Proof-of-Concept, it is important to assess the performance of the prototype after successful implementation. Therefore, the final goal of the thesis is to evaluate the pipeline classifier for common classification evaluation metrics such as, accuracy, precision, and recall. As an optional goal, a usability study to measure the increase in code comprehension when using HeaderGen is planned.

4.3.1 Classifier Evaluation

The aim of classifier evaluation is to assess the performance of HeaderGen's *function-to-ML-phase* mapping. A group of data scientists will be contacted and a collection of Jupyter notebooks will be provided to them. These users will then be asked to annotate a few of these notebooks without the help of HeaderGen. Subsequently, HeaderGen will also be executed on these notebooks to yield automated annotated notebooks. After the tasks, both the output notebooks will be presented to the users and their opinions on HeaderGen's performance will be captured. In addition, a synthetic benchmark can be implemented to generate the automated headers.

4.3.2 Synthetic Benchmark

To assess the functionality after implementing the above mentioned goals and assure effectiveness of the extended static analyzer and the cell header generator, it is important to perform some tests as a part of the development process. Therefore, a synthetic benchmark consisting of various tests is included in the development life-cycle as a goal of the thesis. This benchmark will be structured in a way similar to PyCG's micro-benchmark. The benchmark directory will be divided into various sub-directories based on different features that will be implemented during the thesis work. These sub-directories will contain an input jupyter notebook, a result json, and a brief description of the code. The analysis result from the HeaderGen's static analyzer will be validated against the provided result json.

4.3.3 Optional: Usability Study

The usability study can be conducted after the implementation of above mentioned goals. The idea is to gather users from different professional backgrounds related to the field of Machine Learning. The target groups would be Data Scientists, Data Analysts, Machine Learning engineers, professors, researchers, and university students. After the task completion, all the users will answer a questionnaire related to the technical and personal aspects of successfully completing the task. The answers to the questionnaire will be analyzed to study the effect of HeaderGen as the automated cell header generator in practical use.

5 Thesis Structure

1 Introduction

   (a) Motivation
   (b) Problem Statement
   (c) Goals

2 Related Work

3 Problem Analysis

   (a) Shortcomings in PyCG
   (b) Functions to Phase Mapping in HeaderGen

4 Methodology

   (a) Extended PyCG
      i. Flow Sensitive Call-Site Identification
      ii. Dynamic Function Call: eval()
   (b) Enhanced HeaderGen
      i. Fusing PyCG and HeaderGen
      ii. Advanced Functions to Machine Learning Phase Mapping

5 Implementation and Validation

   (a) Proof-of-Concept

6 Evaluation

   (a) Is HeaderGen useful?

7 Conclusion and Future Work

- Abbreviations
- Bibliography

6 Time Plan

TASK	WEEKS
GOAL-PCG: EXTENDING PYCG	
Cell-sensitive call-site identification	3
Analyzing External libraries	3
GOAL-EHG: ENHANCING HEADERGEN	
Fusing PyCG with HeaderGen	3
Extending function-to-ML-phase function	2
GOAL-EVL: EVALUATION	
Synthetic benchmark	2
Classifier evaluation user study	1
Results analysis and report	1
THESIS REPORT	
Writing thesis report	5
TIMELINE OVERVIEW	
Total	20

Bibliography

[F^{+}21] David Fraser et al. Pyan3: Offline call graph generator for python 3, Accessed on 16. Aug. 2021. `https://github.com/davidfraser/pyan`.

[GTL18] Gharib Gharibi, Rashmi Tripathi, and Yugyung Lee. Code2graph: Automatic generation of static call graphs for python source code. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 880–883, 2018. `https://doi.org/10.1145/3238147.3240484`.

[KLDR15] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. Practically tunable static analysis framework for large-scale javascript applications (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 541–551, 2015. `https://doi.org/10.1109/ASE.2015.28`.

[Knu84] Donald E. Knuth. Literate programming. *Comput. J. 27,2 (Jan. 1984), 97 - 111*, 1984. `https://doi.org/10.1093/comjnl/27.2.97`.

[Per18] Jeffrey M. Perkel. Why jupyter is data scientists' computational notebook of choice, 2018. `https://www.nature.com/articles/d41586-018-07196-1`.

[PMBF19] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. A large-scale study about quality and reproducibility of jupyter notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 507–517, 2019. `https://doi.org/10.1109/MSR.2019.00077`.

[RTH18] Adam Rule, Aurélien Tabard, and James D. Hollan. *Exploration and Explanation in Computational Notebooks*, page 1–12. Association for Computing Machinery, New York, NY, USA, 2018. `https://doi.org/10.1145/3173574.3173606`.

[SSL^{+}21] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. Pycg: Practical call graph generation in python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1646–1657, 2021. `https://doi.org/10.1109/ICSE43902.2021.00146`.

[VB21] Ashwin P. S. Venkatesh and Eric Bodden. Automated cell header generator for jupyter notebooks. `https://doi.org/10.1145/3464968.3468410`, 2021.

[WLZ20] Jiawei Wang, Li Li, and Andreas Zeller. Better code, better sharing: On the need of analyzing jupyter notebooks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '20)*, 2020. Association for Computing Machinery, New York, NY, USA, 53–56. `https://doi.org/10.1145/3377816.3381724`.

[ZW21] Gang Zhang and Jin Wuxia. Depends is a fast, comprehensive code dependency analysis tool, Accessed on 18. Aug. 2021. `https://github.com/multilang-depends/depends`.