GAP-Gen: Guided Automatic Python Code Generation

Junchen Zhao * junchez3@uci.edu
Yurun Song * yuruns@uci.edu
Junlin Wang junliw1@uci.edu
Ian G. Harris harris@ics.uci.edu
University of California, Irvine

Abstract

Automatic code generation from natural language descriptions can be highly beneficial during the process of software development. In this work, we propose GAP-Gen, a **G**uided **A**utomatic **P**ython **C**ode **G**eneration method based on Python syntactic constraints and semantic constraints. We first introduce Python syntactic constraints in the form of **Syntax-Flow**, which is a simplified version of Abstract Syntax Tree (AST) reducing the size and high complexity of real python AST but maintaining crucial syntactic information of Python code. In addition to Syntax-Flow, we introduce **Variable-Flow** which abstracts variable and function names consistently throughout the code. In our work, rather than pre-training, we focus on modifying the fine-tuning process which reduces computational requirements but retains high generation performance on automatic Python code generation task. GAP-Gen fine-tunes the transformer-based language models T5 and CodeT5 using the Code-to-Docstring datasets CodeSearchNet, CodeSearchNet AdvTest and Code-Docstring-Corpus from EdinburghNLP. Our experiments show that GAP-Gen achieves better results on automatic Python code generation task than previous works. Our implementation is available on the github[1].

1 Introduction

With billions of people relying on software for their everyday work and life, developers face an ongoing challenge to create programs efficiently. One potential solution to this challenge is to use human descriptions to generate the corresponding source code. By using this approach, developers can write software specifications in natural language, which are then translated into code through code generation mechanism. Early attempts to tackle this problem were rule-based, identifying syntactic patterns in text and using handcrafted rules to map the patterns to code. Methods used to recognize syntactic structure include regular patterns (Gulwani and Marron, 2014; Kate et al., 2005; Le et al., 2013) and parse trees produced using context-free grammars (Kate et al., 2005; Le et al., 2013; Ballard and Biermann, 1979; Price et al., 2000). Several previous approaches convert a sentence into a formal statement by mapping verbs to functions in the formal language, and mapping the objects of the verb in the sentence to function arguments in the formal language (Ballard and Biermann, 1979; Price et al., 2000; Little and Miller, 2006). For example, the sentence, "Add r1 to r2" might be mapped to add(r1, r2) in a procedural language. The problem of finding the objects of the verb to use as function arguments is simple if the sentence structure is strictly limited. Several approaches use regular expressions (Le et al., 2013) or context-free grammars (Kate et al., 2005) to identify the objects in the sentence.

More recent approaches are data-driven and leverage machine learning methods, e.g., (Desai et al., 2016) uses a Naive Bayesian Classifier to map English words to a domain-specific language, and (Quirk et al., 2015) learns production rules for a semantic parser. (Rahit et al., 2019) uses a Long-Short Term Memory (LSTM) Recurrent Neural Network (RNN) architecture to implement their neural machine translation approach . Work presented in (Ling et al., 2016) introduces the Latent Predictor Network (LPN) architecture which treats code generation as a sequence-to-sequence modeling problem. (Yin and Neubig, 2017) builds upon this approach by leveraging the grammar model of the target language as prior knowledge.

Research presented in (Clement et al., 2020; Feng et al., 2020; Lu et al., 2021) introduced transformer-based language model pre-training methods to map the natural language semantic with

* Equally contributed
[1] https://github.com/Rain9876/Auto-Code-Generator

Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop, pages 37–51

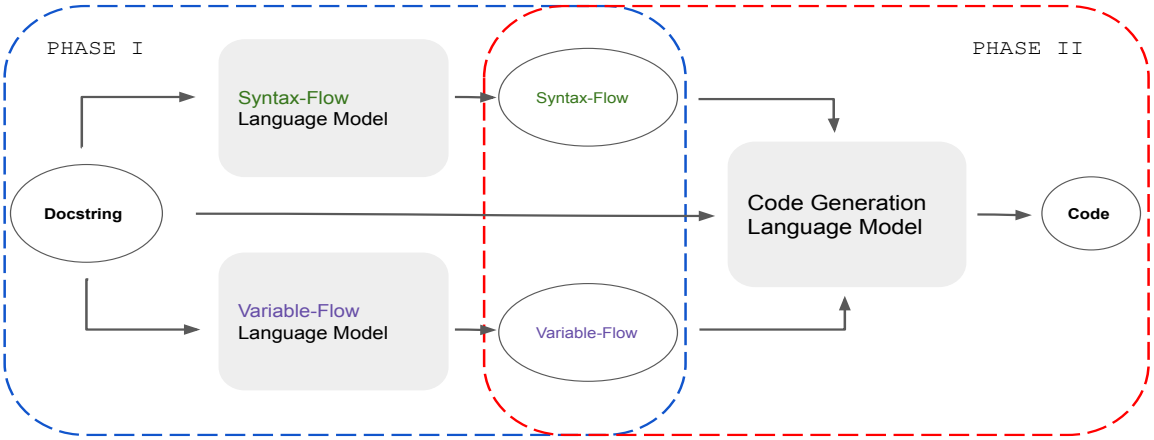

Figure 1: **An overview of the proposed approach** Phase I has two language models generating Syntax-Flow and Variable-Flow. In Phase II, another language model encodes these two types of information as well as the docstring to generate code.

the code. Although these works have relatively good performance on source code generation task, they require high computational resources, which are difficult to acquire. They also usually consider the code as a sequence of tokens (Feng et al., 2020; Kanade et al., 2020; Lu et al., 2021) and ignore either the source code's syntactic-level or semantic-level information, which could improve the language models' code understanding capability, during their pre-training process.

In this work, we present GAP-Gen, a method to improve automatic Python source code generation from natural language description. Our GAP-Gen is fine-tuning of the pre-trained T5-English (Raffel et al., 2020a) and CodeT5 (Wang et al., 2021) language models that employ Syntax-Flow and Variable-Flow as guidance and has shown on being able to understand the relationship between natural language description and Python code from syntactic and semantic level of the Python code.

Our GAP-Gen training pipeline is composed of two phases. As shown in Figure 1, **Phase I**, fine-tunes our pre-trained language model for the purpose of generating Python code's syntactic constraints and semantic-level structure, the Syntax-Flow and the Variable-Flow. **Phase II**, fine-tunes a separate language model by encoding natural language description of the code, the generated code syntactic constrains (Syntax-Flow) and abstracted variable names (Variable-Flow) from Phase I to generate Python code. By doing so, language models fine-tuned with GAP-Gen training pipeline are able to surpass many previous works' performances, which rely on the pre-training process of language models without considering code's syntactic and semantic information.

Our **main contributions** are:

- We introduce Syntax-Flow and demonstrate the importance of the source code's syntactic information in the automatic Python code generation task.
- We show that abstracting variable and function names through Variable-Flow is effective in maintaining the naming semantics of the code.
- We achieve high performance on automatic Python source code generation task without language model pre-training.

2 Related Works

Language Models for Programming Languages. Transformer-based language models that utilize attention mechanisms have been dominating NLP benchmarks (Vaswani et al., 2017; Wang et al., 2018). The novel attention-based message passing techniques plus multi-task pre-training (Devlin et al., 2019) have been through extensive studies. This leads to a deeper understanding of the representational power of transformer-based models (Ethayarajh, 2019; Kovaleva et al., 2019; Jain and Wallace, 2019).

At the same time, transformer-based auto-regressive language models consisting of encoder/decoder demonstrate stellar performances on many NLP generative tasks (Radford et al., 2019; Lewis et al., 2020; Raffel et al., 2020b). These tasks include but not limited to story generation (See et al., 2019), dialogue (Budzianowski and Vulic, 2019), summarization (Lewis et al., 2020), Entity

Retrieval (Cao et al., 2021), Question Answering (Guu et al., 2020), and so on. Similar advances have also been made in Programming Language relevant tasks.

Programming language generation tasks, although not considered as natural language generation tasks, have been demonstrated to have great results when they are modeled similarly as natural language generation tasks. (Feng et al., 2020) pre-trains on Mask Language Modeling (MLM) and replaced-token detection for code understanding tas. In (Liu et al., 2020), the authors develop a code completion transformer-based model by jointly predicting the probability and type of the next token. For the task of code summarization, transformer-based models outperform the other neural approaches (Yu et al., 2020; Ahmad et al., 2020); Svyatkovskiy et al., 2020; Liu et al., 2020) use GPT and UniLM respectively for code completion. More related to our work, (Husain et al., 2019; Clement et al., 2020) explore pre-training methodologies for learning better structural and syntactical information for automatic code generation. Moreover, (Wang et al., 2021; Guo et al., 2021) incorporates Variable-Flows and identifier information into their pre-training process for better code generation performance.

Guided Text Generative Models. Generative modeling is powerful but often falls short in many conditions. The behavior of auto-regressive language models cannot be explicitly controlled, and was shown to be very easy to degenerate (Holtzman et al., 2020; Welleck et al., 2020; Meister et al., 2020). This is also the case for code generation. This prompts researchers to combat this issue by looking at either the training time or the decoding time. Work in (Fan et al., 2018) constrains the sample space to top-k tokens in the softmax logistics to avoid introducing highly unlikely tokens. (Holtzman et al., 2020) instead restricts the sampling space to the smallest set of space above some probability mass. Using simple decoding variants is lightweight to implement, but does not change the predicted likelihood of each token. (Welleck et al., 2020) argues that the likelihood objectives is at fault, and proposes unlikelihood training objective, which forces lower probabilities on unlikely generations.

Furthermore, practitioners have also injected priors or structural information into the language model for better generation. (Zhang et al., 2020; Lagutin et al., 2021) utilizes policy learning to control model behaviors. However, this approach suffers from high variance (Choshen et al., 2020). Recently work in story generation (Yao et al., 2019; Rashkin et al., 2020; Goldfarb-Tarrant et al., 2020) uses a plotline/storyline as an intermediate state for generation. This alleviates the language modeling tasks and sets up the model to better learn the structure of the stories. Being motivated by story generation, our work injects syntactic and semantic structural information in a setup that is similar to this line of works. For the code generation task, we utilize our proposed Syntax-Flow and Variable-Flow as the intermediate state to help language model better understand code's syntactic and semantic structure information and improve its performance.

3 Method

In this section, we describe our method by introducing Syntax-Flow and Variable-Flow. Then we present the generation process of Syntax-Flow and Variable-Flow. Finally, we present the Python code generation process guided by Syntax-Flow and Variable-Flow.

3.1 Syntax-Flow

Unlike other methods that generate code directly from source input with pre-training, our approach works in a pipeline by generating the structure of the code as an intermediate state first and then generating the detailed code using the code structure.

Procedural structure can be expressed formally as an Abstract Syntax Tree (AST). An AST contains two major components: **STMT (Statement)** and **Expr (Expression)**. STMT describes the general structure of code including the high-level Python code syntactic constraints. Expr is the detailed content of the code, mainly including the function variables and operations. Additionally, there are some special components in an AST such as the exceptional handler, import alias, arguments, etc.

Due to the AST's formality and rich expressiveness regarding the syntactic information of code, there are works that generate an AST first and then use it to aid code generation, such as (Yin and Neubig, 2017; Ling et al., 2016), but these works usually require composite model architecture changes. Also, the AST is too complex for models to directly generate information. The length of ASTs typically

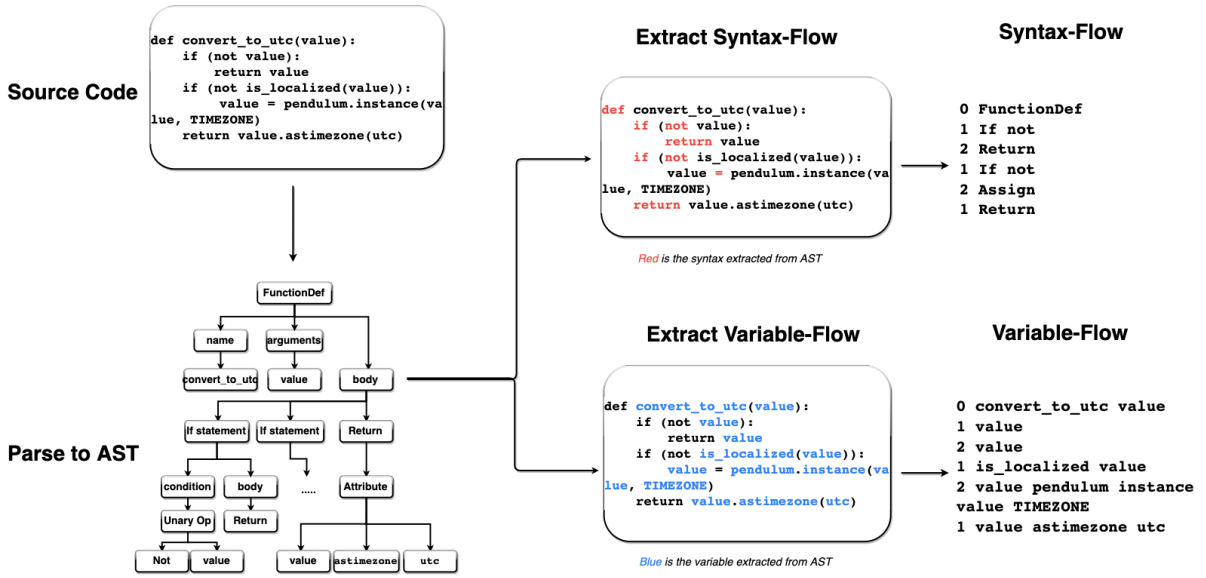

Figure 2: **An overview of the Syntax-Flow and Variable-Flow Generation** . The numbers refer to the number of indentations (4 spaces) required at the beginning of the command line. For Syntax-Flow, statements (STMTs) immediately follow the required indentation and are then followed by several built-in expressions (Exprs). In contrast, Variable Flow follows the function and variable names.

makes it impractical to directly input them into transformer-based language models due to their input sequence length limitation. As a result, we propose a simplified version of an AST, namely Syntax-Flow.

Instead of using the entire tree structure of the AST, we only extract crucial information including Indentation, STMT, and some parts of Expr. By doing so, we reduce the complexity of AST but retain its crucial syntactic structure of Python code, and is small enough to be compatible with transformer-based language models.

In our proposed Syntax-Flow, there are three critical components: **Indentation**, **STMT** and **Default Functions**. These three components are viewed as invariants which means that these components are kept unchanged for maintaining code's correct functionality.

3.2 Variable-Flow

Variable-Flow is another indispensable component in automatic code generation task. It can be effectively applied to maintain the naming semantics of the code during the code generation process. (Wang et al., 2021; Guo et al., 2021) use Variable-Flow during their pre-training process and achieve good performances on programming language-relevant tasks. In their works, they extract function variables names as Variable-Flow which is integrated into their pre-training process for improving language models' capability on understanding the code semantic structure.

In our work, rather than extract variable names only as Variable-Flow, our Variable-Flow contains **Indentation**, **variable names**, and **function names**. Variable names and function names are uniformly free to change. In other words, Python code's functionality remains correct regardless of the changes in these two Variable-Flow components. Therefore, compared with Syntax-Flow, our Variable-Flow contains variant components and is more dynamic.

3.3 Phase I - Generation of Syntax-Flow and Variable-Flow

3.3.1 Generation of Syntax-Flow

Figure 2 shows the Syntax-Flow and Variable-Flow generation pipeline. With regard to Syntax-Flow, the numbers represent the number of indentations (4 spaces) required at the beginning of the command line. Then, a statement is immediately next to the Indentation, followed by several built-in expressions. This feature is extracted from Python code through AST syntax visitor method, a method to go through every detail of the AST nodes recursively and extracts all necessary nodes for use, such as FunctionDef, STMT, exception handler etc, with the count of indentation at the same time.

The simplified generation process of Syntax-Flow is shown in Algorithm 1 Appendix B.4. For each line of source code, we generate one line of Syntax-Flow as you can see in Figure 2. Formally we denote the source code to be $y = (y_1, y_2, ..., y_n)$, and let $E = [e_1, e_2, ..., e_L]$ be the list of indexes of the newline character. Hence, y_{e_1} is the first line break, and $Y_1 = (y_1, ..., y_{e_1})$ is the first line of the source code, $Y_2 = (y_{e_1+1}, ..., y_{e_2})$ the second and so on. Then for each such line of the code, we generate a pair $a = (t, c)$. t is the indentation of the current line of code or number of tabs, and c is the code logic which includes control flow or function definitions. In other words, we are looking for,

$$p(t_i, c_i|Y_i) = p(a_i|Y_i) \tag{1}$$

where i denotes the ith line. Both properties are derived from an AST, which is generated by a standard toolkit. For more detailed steps, refer to Algorithm 1 in Appendix B.4.

3.3.2 Syntax-Flow Language Model

To better learn and utilize the syntactical information of the source code, we use the Syntax-Flow language model to first encode docstrings and then generate Syntax-Flow. Here we use a pre-trained auto-regressive language model. We do not do any additional pre-training, so computing resource is restricted to a manageable amount. As shown in Algorithm 2 in Appendix B.4, to fine-tune the language model, we first generate AST from the ground golden source code $|\boldsymbol{y} = (\boldsymbol{y_1}, \boldsymbol{y_2}, \ldots, \boldsymbol{y_n})$. Then we transform the AST of the source code to Syntax-Flow in a deterministic process,

$$a = \text{SynParse}(\text{AST-Parse}(y)) \tag{2}$$

where SynParse stands for Syntax-Flow Parse. Both SynParse and ASTParse are deterministic functions that generate the Syntax-Flow a. We take this as our true reference and model the process as a standard generative task $P_{LM_S}(\hat{a}_i|x, \hat{a}_1...\hat{a}_{i-1})$, namely,

$$\hat{a} = LM_S(x) \tag{3}$$

where x is the input (docstring for code generation). During inference, given a docstring, this language model is able to generate Syntax-Flow directly for the latter use.

3.3.3 Generation of Variable-Flow

We define the format of Variable-Flow in our work similar to that of Syntax-Flow as shown in Figure 2. For each line of code, it has an indentation t followed by $V = [v_1, ..., v_j]$. V is the list of Variable-Flow which can be either variable names or function names. Multiple variable names can exist in the same line. Its sequential nature alleviates language models like T5 during the generation process. Similar to the setup of Syntax-Flow, we are looking for

$$p(t_i, V_i|Y_i) = p(b_i|Y_i) \tag{4}$$

where Y_i is the ith line of source code and $b_i = (t_i, V_i)$.

3.3.4 Variable-Flow Language Model

We generate Variable-Flow from source code $\boldsymbol{y} = (\boldsymbol{y_1}, \boldsymbol{y_2}, \ldots, \boldsymbol{y_n})$. Then we can safely extract Variable-Flow determinedly from AST:

$$b = \text{VarParse}(\text{ASTParse}(y)) \tag{5}$$

where VarParse stands for Variable-Flow Parse. We take this as our true reference and fine-tune the Variable-Flow Language Model on the source code and Variable-Flow pairs. Specifically, we are modeling $P_{LM_V}(\hat{b}_i|x, \hat{b}_1...\hat{b}_{i-1})$. Hence,

$$\hat{b} = LM_V(x) \tag{6}$$

for the ith line. At inference time, the model is expected to generate Variable-Flow for the latter models to encode.

3.4 Phase II – Generation of Code

Code generation is built on the same language model as Syntax-Flow and Variable-Flow language model. However, unlike the generation of Syntax-Flow or Variable-Flow, an issue in the code generation task is that the length of code description is usually much shorter than the length of generated code. For example, in the CodeSearchNet dataset, many function code data length is over 128 tokens while the description only has an average length is about 50 tokens per sequence. This means that the input information is limited and not enough to generate plausible code unless the language model is available to have more prepared features during the code generation process. For this reason, we use the information generated from Phase I as intermediate features to guide the language model generating Python code in Phase II.

3.4.1 Guided Code Generation Language Model

Our Code Generation Language Model depends on the docstring and the corresponding Syntax-Flow and the Variable-Flow. The language model is obtained from a pre-trained auto-regressive language model T5. In our work, we use the T5-based language model as our guided Code Generation Language Model LM_G.

$$\hat{y} = LM_G(x, LM_S(x), LM_V(x)) \quad (7)$$

The Guided Code Generation Language Model takes in the input docstrings x as well as the outputs of the Syntax-Flow Language Model and Variable-Flow Language Model.

4 Experiment

In this section, we present our experiment in detail. First, we introduce the datasets we use and our data processing approach in our experiment. Then, we present our experimental setup. Finally, we introduce our evaluation metrics in the last subsections.

4.1 Datasets

Code Search Net (CSN)[2] (Husain et al., 2019) is collected from publicly available open-source non-forked GitHub repositories. Only projects that are referenced by at least one other project are included. The original paper filters around 500k code-documentation pairs for Python. They removed pairs where either the documents are less than 3 words or methods less than 3 lines. They also removed duplicate code, constructor and extension methods. After processing, there are 412k training data, 22k validation data and 22k test data.

Edinburgh Code-to-Docstring dataset (CDC)[3] (Barone and Sennrich, 2017) is a parallel Python function-to-docstring corpus collected and processed from Github. The Edinburgh Code-to-Docstring dataset contains 150,370 triples of function declarations, docstrings and bodies in the main parallel corpus. This parallel corpus is partitioned into training/ validation/ testing data, in which the training data contains 109,108 training data, 2,000 validation data and 2000 testing data.

CodeSearchNet AdvTest (Adv)[4] (Lu et al., 2021) is a Python dataset derived from the CodeSearchNet (CSN) corpus. The individual example in CodeSearchNet AdvTest is designed for the code search task. (Lu et al., 2021) took the first paragraph of the docstring as the query for the corresponding Python function. The function names and variables are replaced by special tokens, which we recover back with the original variables name. The CodeSearchNet Advtest dataset contains 251,820 training data, 9,640 validation data, and 19,210 testing data.

4.2 Data Processing

In our experiment, we process our data in 3 steps. **(1)** Clean up Raw Code: All Python 2 code is converted to Python 3 using package 2to3[5], and all Python code styles remain consistent with package pep8[6]. Similar with the step of (Clement et al., 2020). We also remove all invalid code samples that cannot be parsed to AST. After cleaning up the raw code, 99.92% code data is remaining. **(2)** Remove comments and docstrings: Comments and docstrings are removed from the code, since these will not be predicted. **(3)** Replace indentation and newline: Indentation and newline is critical for generation a structured Python code. In our work, we replace them with special symbol § for Indentation and δ for newline.

4.3 Experimental Set Up

In both Phases, we use T5-based models. For Phase I and II, the code description is the main source inputs for the encoder.

Encoding Setup. We use the AdamW optimizer for all the T5 models and assign learning rate 1e-4 for Phase I and Phase II. The training step for Phase I is kept at 75K and batch size at 32. The training step for Phase II is kept at 100K and batch size at 32.The learning scheduler is inverse root square and has warm-up step of 5000 for phase I and 10000 for Phase II.

Decoding Setup. Both Phase I and Phase II take length 512 as input and have output length of 128 for phase I and 256 for Phase II. The beam size is 5 for all Phases fine-tuning. We add repetition penalty of 2 for Syntax-Flow and Variable-Flow generation considering the case that repeated statements occur frequently. All the tasks are run on the two Nvidia GeForce A6000 with 48GB GPU memory each.

Evaluation. For our experimental evaluation, we use the metrics BLEU (Papineni et al., 2002),

[2]https://github.com/github/CodeSearchNet

[3]https://github.com/EdinburghNLP/code-docstring-corpus

[4]https://github.com/microsoft/CodeXGLUE

[5]https://pypi.org/project/2to3/

[6]https://pypi.org/project/autopep8/

	Rouge1-F1	Rouge2-F1	RougeL-F1	BLEU
CSN Syntax-Flow	49.1	35.8	47.7	12.7
CSN Variable-Flow	36.7	15.7	33.7	11.4
CDC Syntax-Flow	51.8	41.4	50.4	15.2
CDC Variable-Flow	37.4	18.9	34.8	11.9
Adv Syntax-Flow	50.4	36.9	48.9	13.6
Adv Variable-Flow	37.3	15.6	34.0	11.1

Table 1: The results of Syntax-Flow and Variable-Flow generation for all three datasets in Phase I with T5. The performance is evaluated through Rouge and BLUE.

	Rouge1-F1	Rouge2-F1	RougeL-F1	BLEU	CodeBLEU
CSN	31.1	12.1	27.9	21.2	22.1
CDC	**32.3**	**15.7**	**29.3**	**22.6**	**22.4**
Adv	29.8	11.0	26.7	20.7	20.9

Table 2: The results of Python code generation for CSN, CDC and Adv in Phase II using GAP-Gen pipeline with T5. The performance is evaluated through Rouge, BLUE and CodeBLEU.

ROUGE (Lin, 2004) and CodeBLUE (Ren et al., 2020). BLUE and Rouge are the most common metrics to evaluate generated text. CodeBLUE is a metric specifically designed for the evaluation of generated programming languages. Apart from the similarity of the tokens, it also considers the syntax of commands and logic.

5 Results and Analysis

In this section, we first present our Phase I experimental results, which contain the performance of Syntax-Flow and Variable-Flow generation on the CSN, CDC, and Adv Test datasets. Then, we present our Phase II experimental results on CSN, CDC, and Adv Test datasets. We train our models on each dataset's training data, and run evaluations on the corresponding testing data. Finally, we compare our approach's performance on automatic Python code generation task with previous works.

5.1 Results of Phase I

Syntax-Flow Results. We first show our results on generating Syntax-Flow using T5 language model. We evaluate the generated Syntax-Flow with Rouge and BLEU metrics, as shown in Table 1. The Syntax-Flow performance of CSN, CDC, and Adv is around 50% in Rouge-F1 and Rouge-F2, and over 35% in Rouge-F2. These results are good considering the real vocabulary size used in Syntax-Flow is relatively smaller and syntax tokens are generally similar. When we make a comparison among the three corpora, results from CDC are slightly better than that of Adv and CSN for all the metrics consistently. CDC is a well-organized dataset that's specifically designed for Python automatic code generation task. Considering Adv is derived from CSN and thus more organized, there is only 1.3% in Rouge score and 1% in BLEU improvement.

Variable-Flow Results. We evaluate our generated Variable-Flow results from code docstrings using the Rouge and BLEU metrics. Our evaluation results regarding the generated Variable-Flow are shown in Table 1. Similar to the results in Syntax-Flow, the performance of Variable-Flow in CDC is slightly better than the other two datasets for all the metrics scores. The average results of the Variable-Flow are not as good as those of Syntax-Flow because the generation of Variable-Flow variant components is much more difficult than the Syntax-Flow invariant components. Moreover, over 95% of Syntax-Flow samples' lengths are shorter than 125 tokens. The Rouge F1 is over 35% and Rouge F2 is over 15% on average.

5.2 Results of Phase II

From Table 2, we observe the performance of final Python code generation with Rouge, BLEU and CodeBLEU. The result of CDC is the best among the three corpora because of its cleaner data as well as the effect of better Phase I performance (Syntax-Flow and Variable-Flow). CSN's results were slightly better than Adv's since CSN had about twice as much training data as Adv. As we can see from Table 3, the performance of GAP-Gen slightly outperforms the T5 model that's directly

	Rouge1-F1	Rouge2-F1	RougeL-F1	BLEU	CodeBLEU
GPT2 (Clement et al., 2020)	20.9	7.6	21.9	2.8	–
PyMT5 (Clement et al., 2020)	28.4	13.5	24.8	8.6	–
T5	30.4	11.7	27.4	20.7	21.7
GAP-Gen T5	**31.1**	**12.1**	**27.9**	**21.2**	**22.1**
CodeT5 (Wang et al., 2021)	34.6	14.6	30.2	21.6	23.4
GAP-Gen CodeT5	**35.1**	**14.9**	**30.6**	**22.3**	**24.1**

Table 3: The results of GAP-Gen with other models fine-tuning on CSN datasets for Python code generation task. We report the Rouge, BLUE and CodeBLEU score for all different models, where GAP-Gen T5 and GAP-Gen CodeT5 are the models built on the T5 and CodeT5 model separately using GAP-Gen pipeline.

trained to generate Python code for both Rouge and BLEU metrics. It indicates that our pipeline approach is effective in improving Python code generation. Similar conclusion can be proved by fine-tuning the CodeT5 language model with our GAP-Gen training pipeline. We apply our training pipeline with CodeT5 in Phase II and show that GAP-Gen CodeT5 achieves the best Rouge, BLEU and CodeBLEU scores compared with other models on the same fine-tuning task.

There is a large gap between GAP-Gen and PyMT5 on BLEU and CodeBLEU, which is because PyMT5 generates sequence with max tokens 1024. We limit the maximum target length to 256, which covers about 75% of code lengths. Based on our comparison between T5 and GAP-Gen, the results of GAP-Gen have improved due to the prerequisite of Syntax-Flow and Variable-Flow generation.

5.3 Discussion

Unlike other works focused on pre-training, we design a pipeline approach to achieve a better fine-tuning result. Given the same training configuration, our results prove that there is an improvement derived from using docstring, Syntax-Flow and Variable-Flow together, as compared to using the docstring only. Code generation is a translation task but has its own difficulties. First, our docstring inputs are usually very short, while code outputs are long. For example, there is about 85% of the input sequences in CSN, CDC, and Adv are less than 128 tokens while over half of codes that are longer than 128 tokens. Moreover, code has stricter syntax and less ambivalent semantics. Our pipeline, by dividing the load of generating syntax and semantic information to multiple language models, bypasses the above difficulties and achieves better generation results.

The data leaking issue exists in many previous works using the pre-training technique on the automatic code generation task. For example, in previous work (Clement et al., 2020), the dataset CodeSearchNet used for fine-tuning overlaps with their data used for pre-training. Both of them are collected from the public github repositories. Data leaking will tend to result in high performance on the fine-tuning task but usually is dubious in practice because model should generalize on the unseen data. In our work, we fine-tune our model using T5 which is not pre-trained on existing Code-to-Docstring datasets. Hence, T5 does not have the data leaking problem. However, CodeT5 is pre-trained on the CSN dataset, which may lead to the data leaking problem in code generation task. This can be the reason that CodeT5 alone without using our training pipeline can achieve very good results. However, after we fine-tune CodeT5 using our training pipeline, CodeT5 shows better performance on the Python code generation task, as you can see in Table 2.

At the same time, due to the computational resources limitation, the maximum batch size we can use is 32. Although we are limited by computational resources, we still achieve result improvements in the code generation task. In the future, better computational resources would probably increase the performance further.

6 Conclusions

In this work, we demonstrate the effectiveness of injecting Python syntactic and semantic information into the code generation tasks. We design and implement two different types of information components: Syntax-Flow and Variable-Flow. To incorporate this information, we encode them using separate language models and then feed them along with the docstring input into the final lan-

guage model. Pre-trained language models fine-tuned with our proposed pipeline show better performances over state-of-the-art code generation models. For future directions, new strategies for incorporating that information can be explored.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *ArXiv*, abs/2005.00653.

Bruce W. Ballard and Alan W. Biermann. 1979. Programming in natural language: "nlc" as a prototype. In *ACM '79*.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In *IJCNLP*.

Paweł Budzianowski and Ivan Vulic. 2019. Hello, it's gpt-2 - how can i help you? towards the use of pre-trained language models for task-oriented dialogue systems. In *EMNLP*.

Nicola De Cao, Gautier Izacard, Sebastian Riedel, and Fabio Petroni. 2021. Autoregressive entity retrieval. *ArXiv*, abs/2010.00904.

Leshem Choshen, Lior Fox, Zohar Aizenbud, and Omri Abend. 2020. On the weaknesses of reinforcement learning for neural machine translation. *ArXiv*, abs/1907.01752.

Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. Pymt5: Multi-mode translation of natural language and python code with transformers. *ArXiv*, abs/2010.03150.

Aditya Desai, Sumit Gulwani, Vineeta Lokhande Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, R Sailesh, and Subhajit Roy. 2016. Program synthesis using natural language. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 345–356.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*.

Kawin Ethayarajh. 2019. How contextual are contextualized word representations? comparing the geometry of bert, elmo, and gpt-2 embeddings. In *EMNLP*.

Angela Fan, Mike Lewis, and Yann Dauphin. 2018. Hierarchical neural story generation. In *ACL*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. *ArXiv*, abs/2002.08155.

Seraphina Goldfarb-Tarrant, Tuhin Chakrabarty, Ralph M. Weischedel, and Nanyun Peng. 2020. Content planning for neural story generation with aristotelian rescoring. In *EMNLP*.

Sumit Gulwani and Mark Marron. 2014. Nlyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD Conference*, pages 803–814. ACM.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. Realm: Retrieval-augmented language model pre-training. *ArXiv*, abs/2002.08909.

Ari Holtzman, Jan Buys, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. *ArXiv*, abs/1904.09751.

Hamel Husain, Hongqi Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *ArXiv*, abs/1909.09436.

Sarthak Jain and Byron C. Wallace. 2019. Attention is not explanation. In *NAACL*.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code.

Rohit J. Kate, Yuk Wah Wong, and Raymond J. Mooney. 2005. Learning to transform natural to formal languages. In *AAAI*, pages 1062–1068. AAAI Press / The MIT Press.

Olga Kovaleva, Alexey Romanov, Anna Rogers, and Anna Rumshisky. 2019. Revealing the dark secrets of bert. *ArXiv*, abs/1908.08593.

Evgeny Lagutin, Daniil Gavrilov, and Pavel Kalaidin. 2021. Implicit unlikelihood training: Improving neural text generation with reinforcement learning. *ArXiv*, abs/2101.04229.

Vu Le, Sumit Gulwani, and Zhendong Su. 2013. Smartsynth: synthesizing smartphone automation scripts from natural language. In *MobiSys '13*.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *ArXiv*, abs/1910.13461.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *ACL 2004*.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew W. Senior. 2016. Latent predictor networks for code generation. *ArXiv*, abs/1603.06744.

Greg Little and Rob Miller. 2006. Translating keyword commands into executable code. In *UIST*.

F. Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 473–485.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *ArXiv*, abs/2102.04664.

Clara Meister, Tim Vieira, and Ryan Cotterell. 2020. If beam search is the answer, what was the question? *ArXiv*, abs/2010.02650.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

David Price, Ellen Riloff, Joseph L. Zachary, and Brandon Harvey. 2000. Naturaljava: a natural language interface for programming in java. In *IUI '00*.

Chris Quirk, Raymond J. Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *ACL*.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020a. Exploring the limits of transfer learning with a unified text-to-text transformer.

Colin Raffel, Noam M. Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020b. Exploring the limits of transfer learning with a unified text-to-text transformer. *ArXiv*, abs/1910.10683.

K. M. Tahsin Hassan Rahit, Rashidul Hasan Nabil, and Md Hasibul Huq. 2019. Machine translation from natural language to code using long-short term memory. *ArXiv*, abs/1910.11471.

Hannah Rashkin, Asli Çelikyilmaz, Yejin Choi, and Jianfeng Gao. 2020. Plotmachines: Outline-conditioned generation with dynamic plot state tracking. In *EMNLP*.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, M. Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *ArXiv*, abs/2009.10297.

A. See, Aneesh S. Pappu, Rohun Saxena, Akhila Yerukola, and Christopher D. Manning. 2019. Do massively pretrained language models make better storytellers? *ArXiv*, abs/1909.10705.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1433–1443, New York, NY, USA. Association for Computing Machinery.

Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *ArXiv*, abs/1706.03762.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. *ArXiv*, abs/1804.07461.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation.

Sean Welleck, Ilia Kulikov, Stephen Roller, Emily Dinan, Kyunghyun Cho, and Jason Weston. 2020. Neural text generation with unlikelihood training. *ArXiv*, abs/1908.04319.

Lili Yao, Nanyun Peng, Ralph M. Weischedel, Kevin Knight, Dongyan Zhao, and Rui Yan. 2019. Plan-and-write: Towards better automatic storytelling. *ArXiv*, abs/1811.05701.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *ACL*.

Xiaohan Yu, Quzhe Huang, Zongge Wang, Yansong Feng, and Dongyan Zhao. 2020. Towards context-aware code comment generation. In *FINDINGS*.

Yuhao Zhang, Derek Merck, Emily B Tsai, Christopher D. Manning, and C. Langlotz. 2020. Optimizing the factual correctness of a summary: A study of summarizing radiology reports. In *ACL*.

A Appendix

B Ablation Study

B.1 Effectiveness of Syntax-Flow

In order to show the effectiveness of Syntax-Flow on improving the language model's capability for the Python code generation task, we make comparisons between the results of T5 fine-tuned with docstrings only and T5 fine-tuned with docstrings and Syntax-Flow shown in Table 4. Based on the comparisons of the results, we can observe that T5$_{Syntax-Flow}$ has outperformed the performance of T5 on the majority of evaluation metric scores. Since code has a tree structure and needs to be compiled based on the corresponding AST, it is particularly important to make sure that the syntactic structure included in the generated code is correct. In T5$_{Syntax-Flow}$, we inject the syntactic structure of code, the Syntax-Flow, into the fine-tuning process of the T5 model so that the T5 can learn how the syntactic structure of code should be incorporated to generate higher quality code, a fact which we believe is the reason that T5$_{Syntax-Flow}$ has better code generation performance.

B.2 Effectiveness of Variable-Flow

We also make experiments to show the effectiveness of Variable-Flow for the Python code generation task. Similarly, we make comparisons between the results of T5 fine-tuned with docstrings only and T5 fine-tuned with docstrings and Variable-Flow shown in Table 4. As we can observe from the result comparison, T5$_{Variable-Flow}$ only does not achieve significant improvements regarding the evaluation metric scores and we believe that there are two potential reasons causing this to happen. First, comparing the evaluation score of the generated Syntax-Flow with that of Variable-Flow shown in Table 1, we can see that the generated Variable-Flow's evaluation scores are worse than that of the Syntax-Flow. It happens because the length of generated Variable-Flow is much longer than that of Syntax-Flow due to the characteristics of Variable-Flow that it supposes to contain general semantic information of code. Second, due to the longer length of the generated Variable-Flow, the inputs to T5$_{Variable-Flow}$ are much longer than that of T5$_{Syntax-Flow}$, and T5$_{Variable-Flow}$ does not know which line of generated code the variable should be assigned to because of the lack of syntac-

	Rouge1-F1	Rouge2-F1	RougeL-F1	BLEU	CodeBLEU
T5	30.4	11.7	27.4	20.7	21.7
T5-Syntax-Flow	30.9	12.1	27.7	20.7	21.9
T5-Variable-Flow	30.5	11.8	27.4	20.6	21.7
GAP-Gen-T5	**31.1**	**12.1**	**27.9**	**21.2**	**22.1**

Table 4: Results Comparisons of GAP-Gen-T5 components on CSN datasets for Python code generation task.

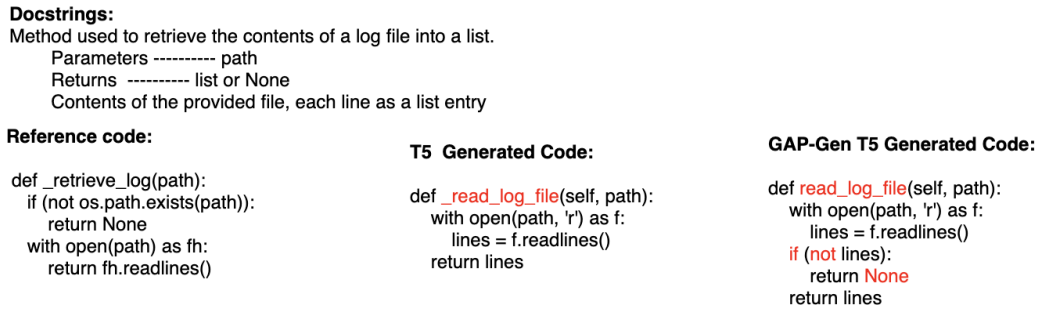

Figure 3: Sample code generated from the docstring in CSN datasets. **The most left code** is the golden standard reference code. **The middle code** is generated directly from T5 fine-tuned with docstring. **The most right code** is generated using our GAP-Gen fine-tuning pipeline.

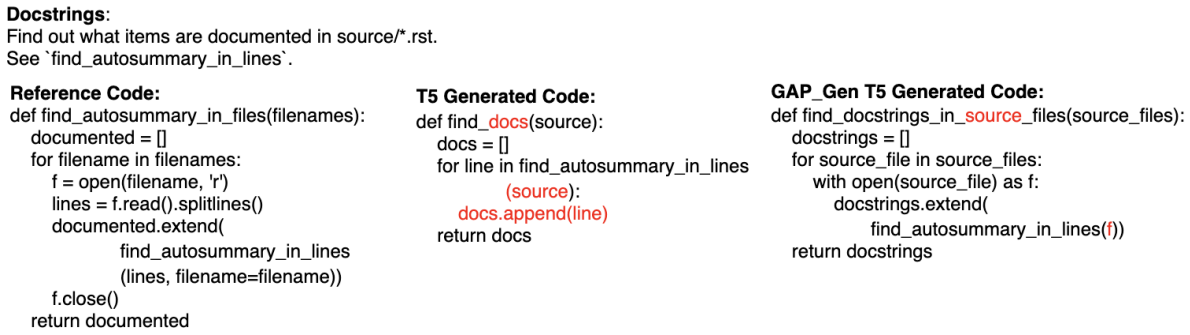

Figure 4: Sample code generated from the docstring in CSN datasets.

tic structure information, then much longer code is likely to be generated. However, the effectiveness of Variable-Flow can also be reflected from the loss of T5 with Variable-Flow only from Figure 6. A lower perplexity can be obtained, and the generated code is more fluent on average.

B.3 Samples Analysis

To illustrate the usefulness of our proposed Syntax-Flow and Variable-Flow components, we have attached the generated Python code samples using Syntax-Flow and Variable-Flow with the corresponding docstrings in the CSN dataset. We have also provided sample analysis of them in the following paragraphs.

As we have demonstrated in our paper, in order to generate well-working Python code, the language model should not only understand the text semantic information from a given docstring but also should be capable of considering the code syntactic information and the code variable semantic information.

Based on the given sample codes shown in Figure 3, it is clear that the code, which is generated directly from T5 without having Syntax-Flow and Variable-Flow injected, cannot properly handle both the code syntactic information and the code variable semantic information. For example, the docstring specifies that the code should return a list or None variable, suggesting that there are 2 different return values that should be generated under different conditions. As a result, the fine-tuned model should consider both the code syntax logic, the boolean operation, and the variable semantic,

the generated variables, during the code generation process. However, due to the lack of Syntax-Flow and Variable-Flow components, the T5 model fine-tuned with docstring only is unable to learn the code syntactic information and the code variable semantic information, resulting in the fine-tuned model generates code that is not able to determine where the boolean operation should be generated to handle multiple return values. Similar trends happen in the sample codes shown in Figure 4 as well.

In contrast, in our work GAP-Gen, we consider the Syntax-Flow and Variable-Flow during the code generation process. Due to the support of these two components, we can successfully generate a higher-quality code with the boolean operation and different return values.

B.4 Training Algorithms

In this subsection, we include the training algorithms for **1.** generating the Syntax-Flow and Variable-Flow, and **2.** fine-tuning the pre-trained Language Model with Syntax-Flow and Variable-Flow.

Algorithm 1 Generate Syntax-Flow & Variable-Flow

Require:
$x = (x_1, x_2, ..., x_n) \in X$: input docstring
LM_S: Language Model being used for Syntax-Flow
LM_V: Language Model being used for Variable-Flow
Ensure:
$A = (a_1, a_2, ..., a_n)$: Syntax-Flow
$B = (b_1, b_2, ..., b_k)$: k Variable-Flow
Initialize A, B to be empty arrays
for each docstring:$x \in X$ **do**
  $a \leftarrow LM_S(x)$
  $b \leftarrow LM_V(x)$
  Append(A,a)
  Append(A,a)
end for
return A,B

B.5 Code Generation Loss Analysis

We further analyze the loss trend for generating the Python code using T5 and our GAP-Gen training pipeline. In our analysis, we show three loss trend comparisons:

Algorithm 2 Fine-tuning Language Model with Syntax-Flow

Require:
$x = (x_1, x_2, ..., x_n) \in X$: input docstring
LM_S: Pre-trained Language Model being used for Syntax-Flow
Ensure:
LM_S: Language Model fine-tuned for generating Syntax-Flow
Initialize D to be empty array
for each docstring:$x \in X$ **do**
  $p \leftarrow$ ASTPARSE(X) AST parsed by standard Python AST parser
  $d \leftarrow$ SYNPARSE(p)
  Append(D,d)
end for
for i = 1 to |X| **do**
  $a' \leftarrow LM_S(X[i])$
  $l = loss(D[i], a')$
  $LM_S.backwards(l)$
end for
return LM_S

- T5 vs GAP-Gen with Syntax-Flow only in Figure 5,
- T5 vs GAP-Gen with Variable-Flow only in Figure 6,
- T5 vs GAP-Gen with both Syntax-Flow and Variable-Flow in Figure 8.

Based on our observations, we find the global loss trends between T5 and GAP-Gen are similar. However, when we zoom into the last 10k steps, the training and validation loss of GAP-Gen are consistently lower than those of T5 on the Python code generation task from all three scenarios.

By comparing with the loss trend between T5 and GAP-Gen with Syntax-Flow only and GAP-Gen with Variable-Flow only, we find both scenarios have lower training and validation loss in the last 10k steps than those of T5. This fact shows that both of Syntax-Flow and Variable-Flow are contributing to the Language Model fine-tuning process. At the same time, we find GAP-Gen with both the Variable-Flow and Syntax-Flow results in the lowest training and validation loss compared with those of the other two scenarios. This finding further illustrates that our method GAP-Gen does have improvement on the Python code generation task.

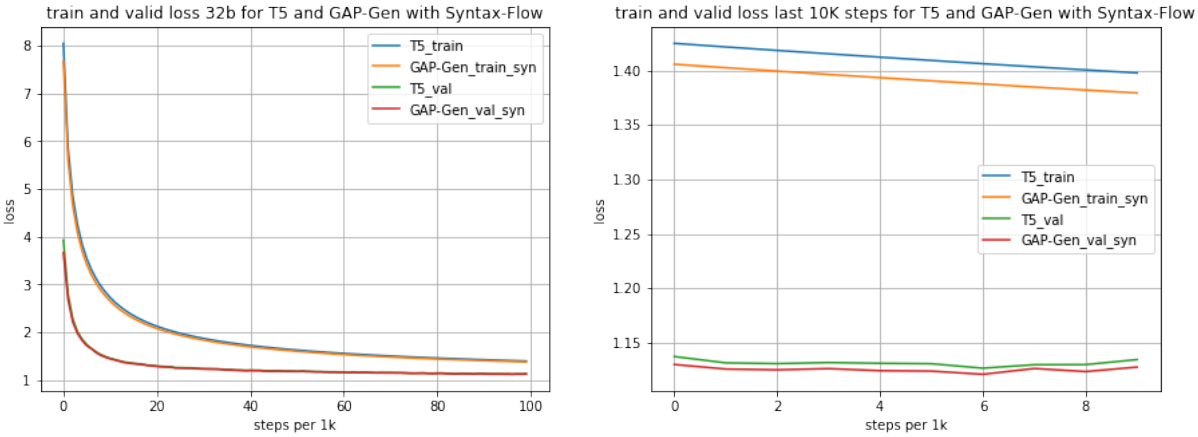

Figure 5: Loss comparison visualization between T5 and GAP-Gen using Syntax-Flow only.

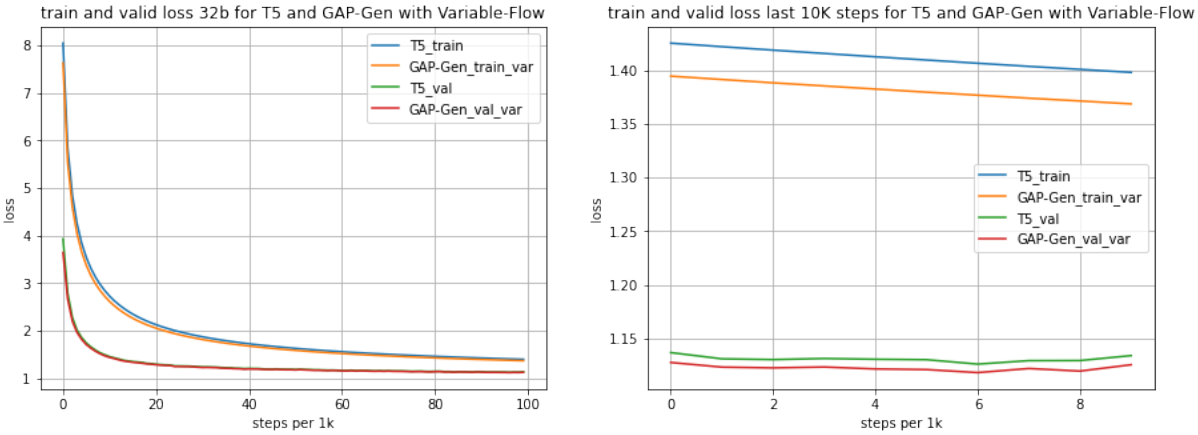

Figure 6: Loss comparison visualization between T5 and GAP-Gen using Variable-Flow only.

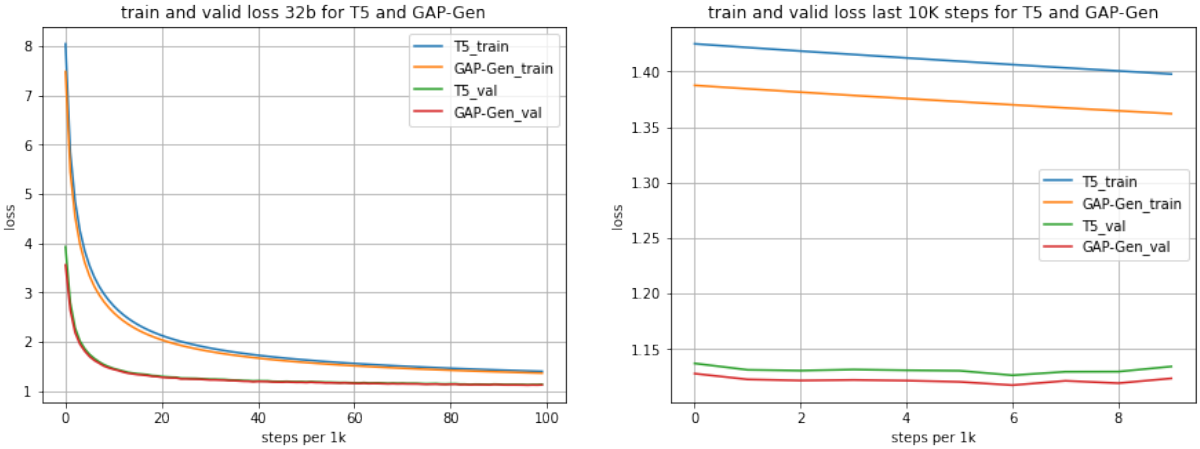

Figure 7: Loss comparison visualization between T5 and GAP-Gen.

Docstrings:
Lists all of the reports currently available.

Reference Code:

```
def reports_list(request):
    reports = []
    for report in get_all_reports():
        reports.append({'name': report.__name__, 'url':
                reverse(report), 'description': report.__doc__})
    reports.sort(key=(lambda report: report['name']))
    ctx = {'reports': reports}
    return render(request,'registrasion/reports_list.html', ctx)
```

T5 Generated Code:

```
def list_reports(self):
    if (self._reports is None):
        self._reports = []
        for report in self.get_reports():
            self._reports.append(report)
```

GAP_Gen T5 Generated Code:

```
def list_reports(self):
    report_list = []
    for report in self.reports:
        report_list.append(report)
    return sorted(report_list, key=(lambda x: x[0]))
```

Docstrings:
Build a file path from *paths* and return the contents

Reference Code:

```
def read(*paths):
    with open(os.path.join(*paths), 'r')
        as file_handler:
    return file_handler.read()
```

T5 Generated Code:

```
def read(*paths):
    for path in paths:
        path = os.path.expanduser(path)
        if os.path.exists(path):
            with open(path, 'r') as f:
                return f.read()
    return ''
```

GAP_Gen T5 Generated Code:

```
def read(*paths):
    with open(os.path.join(*paths), 'r') as f:
        return f.read()
```

Docstrings:

```
Validate that all required type methods are implemented.
        At minimum a type must have:
        - a convert() or convert_binary() function
        - a default_formatter() function
        Raises an ArgumentError if the type is not valid
```

Reference Code:

```
def _validate_type(cls, typeobj):
    if (not (hasattr(typeobj, 'convert') or hasattr(typeobj, 'convert_binary'))):
        raise ArgumentError('type is invalid, does not have convert or convert_binary function',
                type=typeobj, methods=dir(typeobj))
    if (not hasattr(typeobj, 'default_formatter')):
        raise ArgumentError('type is invalid, does not have default_formatter function',
                type=typeobj, methods=dir(typeobj))
```

T5 Generated Code:

```
def validate_type_methods(self):
    for method_name in dir(self):
        method = getattr(self, method_name)
        if (not hasattr(method, 'convert')):
            raise ArgumentError(('Method %s must have a convert() or convert_binary() function'
                    % method_name))
        if (not hasattr(method, 'default_formatter')):
            raise ArgumentError(('Method %s must have a default_formatter function' % method_name))
```

T5-GAP-Gen Generated Code:

```
def _validate_type(self, type):
    if (not hasattr(type, 'convert')):
        raise ArgumentError('Type must have a convert() or convert_binary() function')
    if (not hasattr(type, 'default_formatter')):
        raise ArgumentError('Type must have a default_formatter function')
```

Figure 8: Additional generated sample codes from our experiments.