Universidade do Minho
Escola de Engenharia
Departamento de Informática

Diogo Filipe Lopes Soares

Python-Tutor on program comprehension

December 2020

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Diogo Filipe Lopes Soares

Python-Tutor on program comprehension

Master dissertation
Intregrated Master’s in Informatics Engineering

Dissertation supervised by
Pedro Manuel Rangel Santos Henriques
Maria João Varanda

December 2020

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Diogo Soares

ABSTRACT

The time spent analysing a software with the goal of comprehending it is huge and expensive. Reduce the time necessary to a professional understand a program is essential for the advance of technology. Therefore, the program comprehension has always been an area of interest as realizing how a programmer thinks can help facilitate many of their daily activities, making the developer a more productive worker. As the world begins to reshape itself thanks to the advances of technology, this area of research gains more and more relevance. This project aim to study the tools developed within the comprehension of programs that usually are associated to software maintenance and analysing the animation web tool Python-Tutor. After this study, it's required to explore Python-Tutor to understand how it can be improved with the addition of important features to program comprehension as Control Flow Graph (CFG), Data Flow Graph (DFG), Function Call Graph (FCG) and System Control Graph (SCG). The idea behind this is to allow new programmers to view their programs and create a visual image of them in order to understand them and improving their skills to understand someone else's programs.

Keywords: program comprehension, software visualization, Python-Tutor, graphs, software animation

RESUMO

O tempo despendido a analisar um programa de forma a compreendê-lo é enorme e dispendioso. Reduzir o tempo necessário para um profissional compreender um programa é fulcral para o avanço da tecnologia. Assim, a compreensão de programas sempre foi uma área de interesse pois perceber como um programador pensa pode ajudar a facilitar muitas atividades diárias deste, tornando o programador num trabalhador mais produtivo. À medida que o mundo se vai moldando à informática, esta área de pesquisa tem ganho cada vez mais relevância. Neste projecto iremos estudar as ferramentas desenvolvidas no âmbito da compreensão de programas associadas à manutenção de software e analisar a ferramenta de animação *web Python-Tutor*. Iremos explorar esta ferramenta de modo a perceber como a podemos melhorar através da inclusão de novos recursos importantes para a compreensão de programas, tais como: o Grafo de Controlo de Fluxo, Grafo de Fluxo de Dados e o Grafo de Chamadas de Funções. A ideia base passa então, por permitir aos novos programadores visualizar os seus programas e criar uma imagem visual destes de modo a os compreenderem e a melhorarem as suas competências para compreenderem programas de outrem.

Palavras-chave: compreensão de programas, visualização de programas , Python-Tutor, grafos, animação de programas

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

This first chapter will serve as an introduction overview on the work that will be pictured here. It'll start by presenting the problem and the context in which this project is inserted, followed by its motivation. It'll end with a list of the goals of this thesis and its structure.

1.1 CONTEXT AND MOTIVATION

We live in a time where the comprehension of the programming world is more valuable than ever and it's expected to continue like that as the world tends to became more and more dependent of technology. However, the majority of people never had contact with the skeleton of this world, in other words, the code that makes technology run, and those who had, usually began this journey on their college years. But if this type of knowledge is becoming more and more valuable, isn't it urgent to teach people sooner? And people that are already on the job market, wouldn't it be important to them to learn how this world works? But how can we teach some apparent complicated notions to people that are not familiarized with them?

There's also the self taught programmers that use the various tutorials along the web to learn and refine their skills. The methods of these tutorials are usually very similar and the majority of them use the same formula: displays an example and makes you write a similar code. This method allows the user to learn how the language in question works and gives the users the ability to write programs with it. But to being capable of full understanding any language, you don't need just to be shown how it's done but also understand what truly happens behind it and the correlations within a program.

With this type of knowledge and familiarity with coding, people will be able and feel more comfortable to face nowadays problems and comprehend the until then unknown, side of technology.

After learning how to code and becoming a professional in this area, is expected to a normal developer spend much of its time working and looking to others programs and code that isn't their ones. If sometimes it's already hard to understand your own code after some time without looking at it, imagine to a software engineer that has to do the maintenance of code he never saw before. The average developer spends about 60% to 90% of their time understanding the program and this data isn't changing over time[Sie16]. This data is understandable because normally the programs tend to be complex and with numerous lines and files. Also, everyone thinks different so everyone programs different, hindering the process of understanding programs written by others. As a program is usually an attempt to solve a real life problem[You14], programmers write their code according to the solution they find to that problem. So, if you would take a different approach to that problem it can also prejudice you on the comprehension of that program.

One way that could help understand the approach the original programmer took and even understanding all the components of the program is through documentation but unfortunately a good documentation is rare and when there is one, its stays quickly outdated.

Thereby, the idea behind this thesis is a platform where you can not only learn how to program but also be capable to observe the entire structure of a program and how it is organised, improving the perception of a new programmer of existing softwares and the ability to comprehend a program faster. A tool where beginners can learn and understand their code and maybe a tool that can even easy up the job of a software developer with smaller programs and make him more productive. The way to do this is by "giving programs another aspect than that of their source code"[GH01].

1.2 GOALS

This master thesis aims to improve the program comprehension of a program by using software visualization. With this in mind, the platform Python-Tutor will be improved by incorporating a new and innovative feature to it.

It is intended to make this the platform that teachers will use and recommend to their students in order to improve their programming skills. Furthermore, this feature is not only aimed to people that are learning but also to be able to help experienced software developers on their daily jobs.

The feature idealized for this will work with an input that will be the user code and it will be inserted in Python-Tutor, where it will be analysed and then produce graphs allowing the user to acquire a better comprehension of the program in question.

1.3 RESEARCH HYPOTHESIS

The appropriate combination of software animation techniques with visualization tools contributes to facilitate the comprehension of programs.

1.4 DOCUMENT STRUCTURE

After the presentation of the motivation and the goals of this project in the present chapter, the next one will contain the State of Art and the definition of some important concepts.

The state of art is composed by the definition of Program Comprehension, a fundamental concept to understand in order to successfully achieve this project's goal, and Software Visualization. Still on this chapter it will be presented the web platform Python-Tutor from the point of view of an user as well as some similar tools to the one that is aimed to be built.

On the next chapter it's displayed how it is pretended to accomplish this project's goals and its challenges.

On chapter 4 it will be exposed how the tool developed works explaining all the steps since the input as code until its final transformation in the various graphs available.

Still on the development side, the chapter 5 will contain the explanation of the process of integration between the new feature with the platform already available.

The chapter 6 presents the results gathered in a survey about the feature developed and has a analysis to that results.

On the final chapter lies the conclusion of this document as well as the work plan for the future.

2

STATE OF THE ART

This chapter will contain the research that was done to be able to execute this master's project goal as well as possible. For being capable to build a tool that can help understand a program it was needed to understand the concepts behind the construction of one tool like this and to research what was being made and what is already done.

Thereby, this chapter will contain the research on the concepts: Program Comprehension and Software Visualisation. The two pillars necessary for the successful construction of this tool.

It will also contain research on Python-Tutor, the platform where it is intended to implement this feature on, as well as other tools in order to visualize what is already on the market and what can work or not. In other words, see what can be used from the different tools available to build the best possible tool.

2.1 PROGRAM COMPREHENSION OVERVIEW

Researches on program comprehension were analysed to understand what are their conclusions and what's already accomplished in this area. The purpose of this study is to recognize the best strategy to build a tool that can be actually helpful but first, what is program comprehension?

2.1.1 *What is Program Comprehension?*

Program comprehension is the cognitive process of understanding a software's goal and how it is accomplished. This process involves detecting or inferring different kinds of relations between program parts [Pen87]. As a cognitive process, it is difficult to formulate an accurate conclusion since it's dependent of the human factor. An easy example is the difference of level of expertise of two programmers. The expertest one would usually comprehend the program faster than the other.

This is due to the fact that the most experienced programmer has more general knowledge than the other. This type of knowledge is one of the two types needed to comprehend a program. Also known as basic knowledge, it allows the programmer to recognise code structures and algorithms. It's related to a developer programming languages skills and programming principles.

The other one is the software-specific knowledge that represents their level of understanding of the software in question[MV95]. This second type of knowledge can be nonexistent when the programmer first looks at the software or he can had already acquired some of this knowledge, if he already knows the goal of the software, for example. If on one hand the general knowledge is independent, on the other hand the specific knowledge will improve as the programmer analyses the code. This process will be scrutinized with more detail further on the next chapter.

2.1.2 *Evolution of Program Comprehension*

When program comprehension started to be a research field there were less information and much less technology than now, so researchers had to be pragmatic and began these studies with simple methods like think-aloud protocols. Think-aloud protocol is a method that was used to understand the cognitive process[Yos08] where the person in study verbalize all his thoughts.

The use of this method allowed us to realise that the thoughts of a developer differ consonant the familiarity of a program's domains[SV98]. With this in mind, it was derived two concepts of comprehension models, the top-down comprehension model and the bottom-up comprehension models.

The top-down approach is used when the programmer is familiar with the program he's trying to understand. He starts by creating a general hypotheses about the program goal and how it is achieved by using his previous knowledge about the program. This is called the 'expectation-based' comprehension, where the programmer has pre-generated expectations of the code's meaning[OBS04].

After creating his hypotheses, the developer starts searching in the code for algorithms/structures that he can link to his theory and refines his hypotheses as he does that[Bro83]. This is the 'inference-based' comprehension[OBS04].

If the programmer has no previous knowledge about a program, he has no other choice than analyse line by line to create a hypothesis about the program purpose. In this method, the developer starts aggregating functions by their goals.

What usually happens is that developers end up by using an integrated model[MV93] where they use top-down approach when possible and bottom-up when necessary.

Another method used on the first researches was memorization where the programmer have to rewrite the program after seeing it. Researchers conclude that this could be only possible if the programmer had understood the program. They verified that developers could easily rewrite the program when it was written in order of execution. This means that how the code is written affects the comprehension of it, which is good news right? Not exactly, although it would be great have an uniformed structure that allowed a better comprehension of programs, how that structured should be varies from person to person. A different form of coding can really slow down the comprehension of a program for the most experienced programmer[SE84]. On the other hand, there are global concepts that helps achieving this like variables names or indentation rules[SMMH77].

Nowadays, there's even more factors that can affect program comprehension. For example, a simple color schema helps the programmers comprehending a program [Ram86] faster and each text editor/IDE has its own so, this may imply discrepancies on the researches results, just because the test was conducted on an environment more familiar to one person that uses a software more similar to the one where the tests were conducted, as opposed to others participants who prefer others tools.

Therefore, although difficult it is essential to create a neutral environment in order to get the bests results and truly understand what helps in program comprehension.

On the other side, the evolution of technology also brought to us new ways to try to decode program comprehension. One tool that is being tested is the eye tracking technology where researchers can visualize where the developers focus when trying to comprehend a program. Unfortunately, there is no conclusions for now because researches are still in progress.

New studies will not only focus on the comprehension of the source-code, but also on the comprehension of the programmer on structures, hierarchies and architecture of the program as well on the relations between components[Sie16]. And these are aspects where we think we can help to understand with the visualization of our diagrams.

2.2 SOFTWARE VISUALIZATION

Software visualization is the process of giving to a program a more visual aspect. It takes the information that is presented in the source code and converts it to a more graphical representation with the goal of improving the comprehension of that program.

Although a visual image can help to understand a program, it's not that simple since a basic program can have multiple representations. The best one will depend on the task in cause and what the developer wants to know. This also means that each representation has to be thought for specific tasks and a good portray of the system is imperative for it to be helpful. ″Simply repackaging massive textual information into a massive graphical representation is not helpful″, so a linear translation is not ideal since ″experts want to see software visualised in context – not just what the code does, but what it means″.[Pet02]

This type of representations that portray the source code can be helpful for new programmers because it helps them to understand in a more visual way what the code does, but on the other hand, for experts who are comfortable with the code itself, it does not bring new information that can speed up the process of program comprehension.

Sometimes the representation of a big program is so compressed that doesn't allow the developer to gather the information he needs. This obstacle can be surpassed by making the depiction interactive. One simple interaction that can make great difference is the ability of zooming, allowing the user to focus on what he's really interested in[Had18]. The addition of interactivity also improves the user experience since he can choose what to do and what to watch.

2.2.1 *Visualization Techniques*

Here it will be displayed some techniques for software visualization. There are two types of software visualization, static and dynamic[LS06]. A static software visualization it's when it's representation doesn't change over time and it's based on static information, so the representation is just an image of the program. The techniques shown in this section will all be static.

It's dynamic when that representation is animated and the information that it displays change as the program is executed. A good example of an animated visualization is Python-Tutor itself that will be presented on the next section.

2.2.1.1 *Nassi–Shneiderman diagram*

This diagram is used to represent a control-flow of a program and it was created by Nassi and Shneiderman[NS73]. It shows all the paths possible within a program with a simple approach. It follows a top-down design and uses nested boxes to represent sub-problems. In the example presented bellow it's possible to visualize the different types of blocks that this diagrams offer. There is the process blocks that represent the simple actions and when that action is performed it advances to the next block. There's also the branching blocks that represent the conditional statements and divide the path in two. The last representation that is possibly to identify is the loop one, where all actions inside the subset with a side-bar extending out from the condition are executed in each loop. It's not a common representation nowadays.

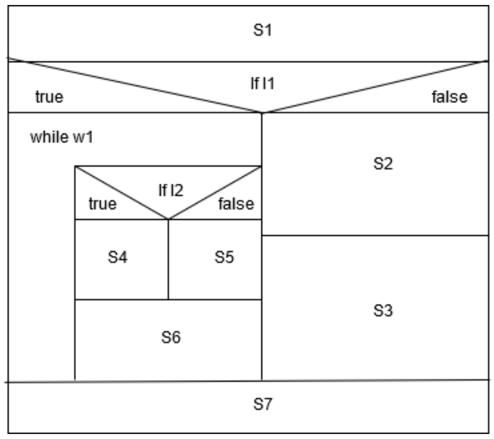


Figure 1: Example of Nassi–Shneiderman diagram

2.2.1.2 *Space-Filling*

Space-filling aims to compress the structured information of a program[LS06]. This type of representation is used to present hierarchies like computer directory and file structures. Hierarchies are one of the most common and important information structures in computing[SCGM00]. This technique also allow us to visualize code metrics and other code related statistics[BE95].

2.2.1.2.1 Tree-maps

The tree-map visualization method maps hierarchical information to a rectangular 2-D display in a space-filling manner[JS91]. The size of each component can be set to be proportional to a given metric like its number of lines for example. Its color can be used to transmit other data of the program as well.

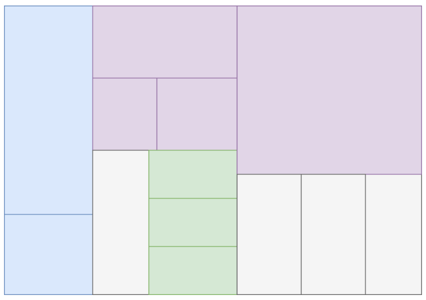


Figure 2: Example of Tree-map diagram

2.2.1.2.2 Sunburst

Sunburst is another space-filling technique but instead of a rectangular layout it uses a radial one where the higher-level hierarchy items stay close to the center. Besides colors, the radial angle corresponds to a directory or file size[SCGM00].

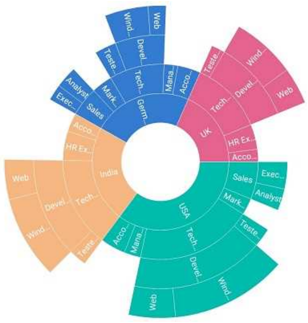


Figure 3: Example of a sunburst diagram[Syn]

2.2.1.3 *Graphs*

Graphs are the most common representation and most recognizable one. One graph consists of nodes and arcs where the nodes represent blocks of code and the arcs its flow. How the nodes are disposed can affect the readability and effectiveness of a graph[LS06] making the choice of its layout an important one. It will now be presented some graphs layouts.

2.2.1.3.1 Tree Layout

The tree layout hasn't necessary the shape of a tree since it can be disposed in a radial way. This layout has essentially two relations between nodes, ancestor or children. If node 1 is above node 2, then 1 is the ancestor of node 2 and 2 is children of 1. The ancestor can have various children but each children only has one ancestor.

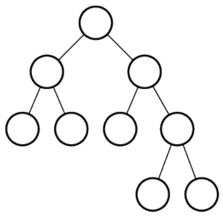


Figure 4: Example of a tree layout graph

2.2.1.3.2 Hierarchical Layout

This style represents the flow with a directed graph where the nodes, as the name indicates, are disposed hierarchically by layers in a top-to-bottom orientation.

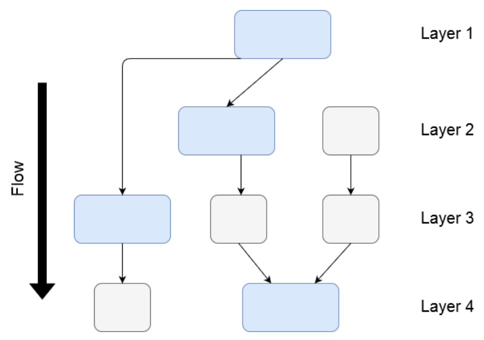


Figure 5: Example of a hierarchy graph

2.2.1.3.3 Orthogonal Layout

This layout is not as strict as the others since nodes can have a free disposition and the relations between them are not limited. Useful to represent complex networks.

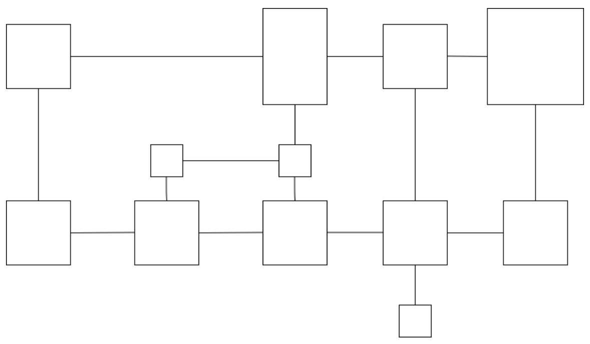


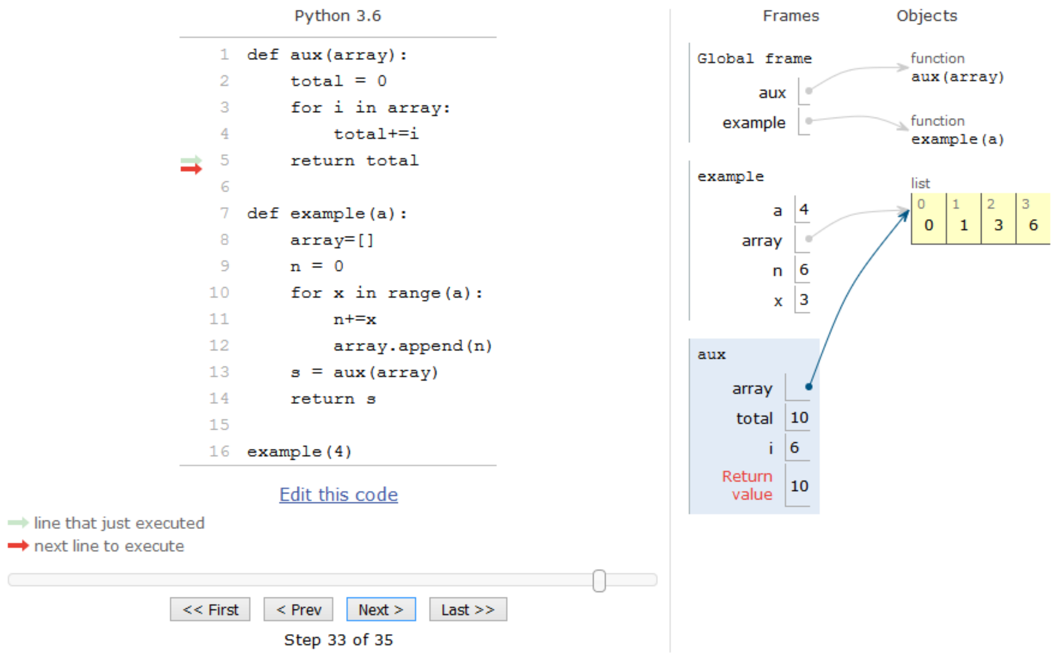
Figure 6: Example of an orthogonal graph

2.3 PYTHON-TUTOR

In this subsection it will be done a detailed presentation on Python-Tutor exposing all of its features. Python-Tutor is an animation web tool that intends to help new programmers understand their code and currently supports some of the most popular languages at the moment like Python, Java, C, C++, JavaScript, TypeScript and Ruby[Put]. This software is designed to help beginners to try their code and test it.

2.3.1 *Visualize Execution*

This is the main feature of Python-Tutor, an interactive step by step presentation that shows the user what is really happening behind every line of code. This allows the user to keep up with the modifications of values that each variable suffers as each line of code is executed. The user has total control of this presentation as he can choose if he wants to go to the next or previous step. As it can be seen on the next image the variables are aggregated by function. If there's a global variable it will appear on the global frame like the functions in this example do. This feature allows the user to visualize data dependencies and also follow its value modifications as the functions run.



```
def aux(array):
    total = 0
    for i in array:
        total+=i
    return total

def example(a):
    array=[]
    n = 0
    for x in range(a):
        n+=x
        array.append(n)
    s = aux(array)
    return s

example(4)
```

Figure 7: Example of visualize execution on Python-Tutor

2.3.2 *Live Help*

Another groundbreaking feature that this platform offers is the possibility to discuss and try to achieve a solution for a problem you have with others users. This is the way it works:

- An user that has a code goal ask to get live help;
- Others users can join his session;
- Each session has its own private chat;
- The session is synchronized for all users.

Besides this, the user responsible for the session can close the session for others anytime he wants or share his session by link. This is a great feature for someone who is struggling on how to write determined function by himself because allows the communication and discussions with others users allowing them to help each other.

2.3.3 *Tests*

Python-Tutor also allows you to validate your own program with comparisons between the output you say it should achieve with a certain input and the actual output your functions calculate. This feature isn't compatible with all languages that Python-Tutor supports as C, C++ and Java are not supported.

2.4 OTHER TOOLS

In these subsections will be presented some other tools that use software visualization in order to improve the program comprehension of users.

2.4.1 *AgileJ StructuredViews*

AgileJ StructuredViews is a plugin for eclipse that generates UML class diagrams from the source code. UML is a language used to structure software projects that is easy to understand so it can be shown to the developers and even to the client. The problem with this language is that unlike other fields, in programming it's not possible to build a software following a plan in detail because of the constant changes to which it is subject to. This makes the UML diagrams quickly out of date.

So, the strategy of this plugin is making things in the reverse way, it creates the UML diagrams from the code creating an easy to read and updated representation of the code. This is a plugin for Java programs that allows the user to visualize dynamically the program structure by presenting it as class diagrams in a web browser. Unlike the Python-Tutor, this tool is not about making the user understand how the functions in a program work but to give the user an overview of the classes and their relationship in a program. This plugin can be worth for someone who is looking for a program for the first time by helping them understand how it is organised.

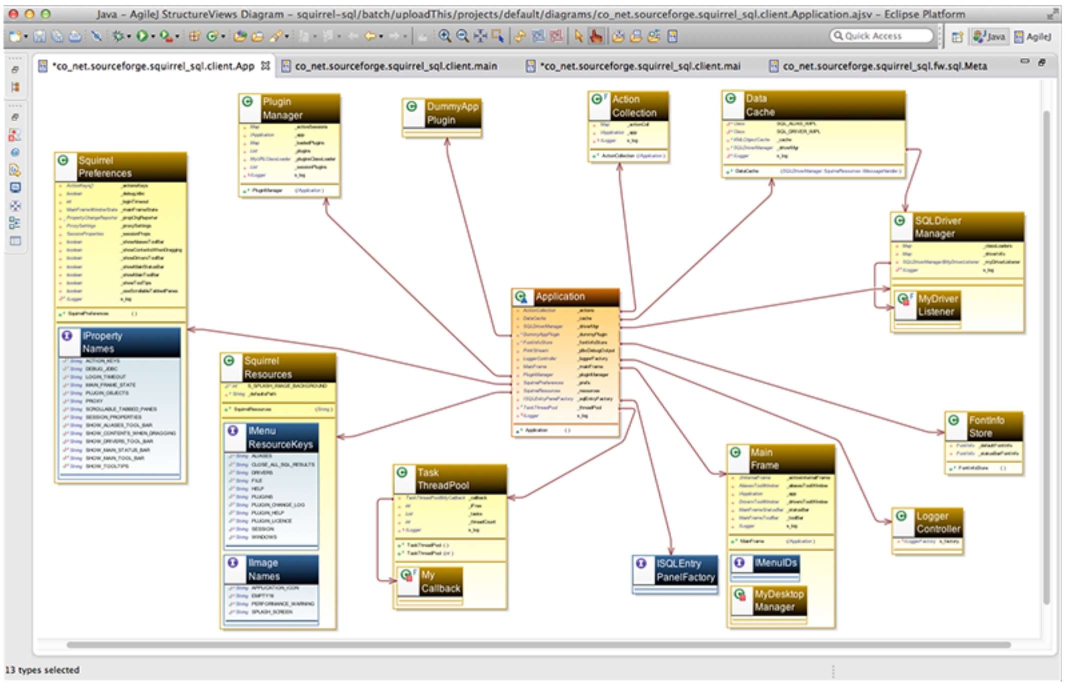
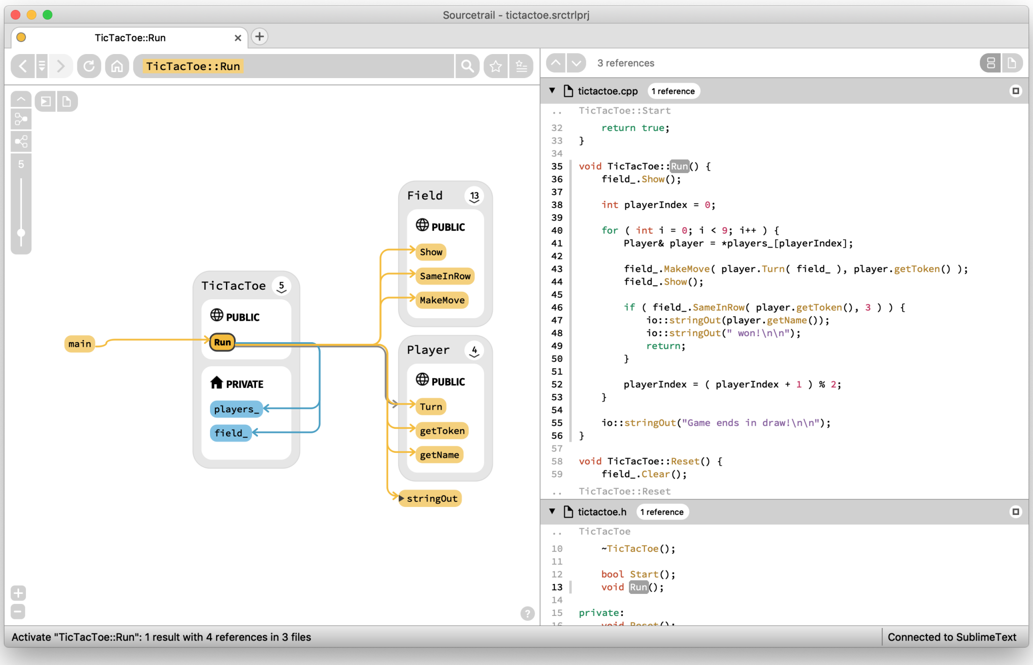


Figure 8: Example of a class diagram generated by AgileJ StructuredView

2.4.2 *Sourcetrail*

Sourcetrail is a tool that can be connected to an IDE or a text editor that displays an interactive dependency graph. It also offers to the user the possibility to see the code that is represented on the graph and a search bar to search functions, classes or variables for example. Its interactivity allows the user to see both of an overview of the program and the detail of a function/variable. You can also see all the calls to a determined function or the general function call graphs and the information of classes like its methods and variables.

So, Sourcetrail allows the user to visualize the relationships between classes as the previous tool but offers much more besides that. The interactivity and the dynamism present in the tool are what makes this tool looking so great. The user can navigate by all of his program and goes direct to what he wants to see. There's features of this tool that are offered to the user for making it more easy to navigate and access every part of his program, but what is connected and can help on the program comprehension is the fact that the user can see all the dependencies of any class, variable or function and see how they relate to each other in a very pleasant way.



```
..  TicTacToe::Start
        return true;
    }

    void TicTacToe::Run() {
        field_.Show();

        int playerIndex = 0;

        for ( int i = 0; i < 9; i++ ) {
            Player& player = *players_[playerIndex];

            field_.MakeMove( player.Turn( field_ ), player.getToken() );
            field_.Show();

            if ( field_.SameInRow( player.getToken(), 3 ) ) {
                io::stringOut(player.getName());
                io::stringOut(" won!\n\n");
                return;
            }

            playerIndex = ( playerIndex + 1 ) % 2;
        }

        io::stringOut("Game ends in draw!\n\n");
    }

    void TicTacToe::Reset() {
        field_.Clear();
..  TicTacToe::Reset
```

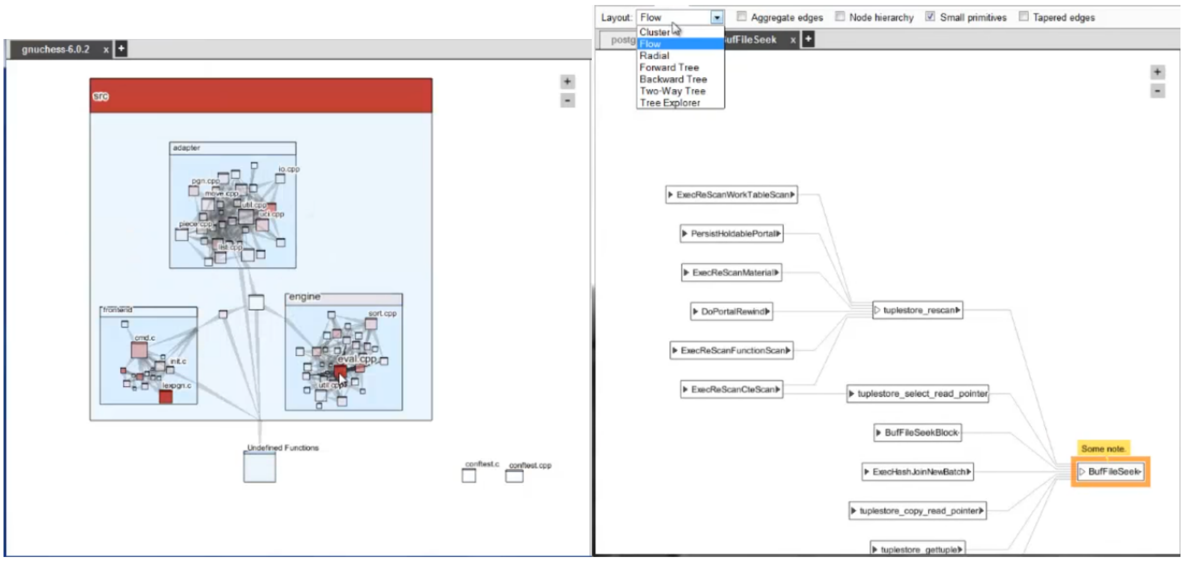
```
..  TicTacToe
        ~TicTacToe();

        bool Start();
        void Run();

    private:
```

Activate "TicTacToe::Run": 1 result with 4 references in 3 files

Figure 9: Sourcetrail example

2.4.3 *CodeSonar's visualization software*

CodeSonar offers a visualization software that shows an interactive call graph where you can see the directories, files and functions rearranged hierarchically. It allows you to choose which metrics you want to be displayed in the graph as well compare functions/files based on that metrics. One of this metrics are warnings, you can see how many warnings there is by file and it even suggest which path may be the origin of a warning. It also allows the user to change the layout of the graph and add notes to the nodes, like sourcetrail.



(a) Overview call graph (b) Call graph of function

Figure 10: Codesonar examples

3 PROPOSAL

In this chapter it will be presented the expected difficulties to accomplish this project goal as well as the proposed solution. The graphs that will be implemented will be explained here and its utilities to the user. It will also be shown the system workflow and its architecture.

3.1 EXPECTED DIFFICULTIES

This project requires adaptability because it is intended to be implemented on other software and as previous said on this report, it's not easy to analyse someone else software and that's the first step of the implementation of this project. Therefore, it's expected some difficulty on the recognition of Python-Tutor structure and on finding the best way to incorporate the new features in.

Another challenge that this project imposes is not how to generate the graphs wanted, but make them different of the ones that already exist and make them appealing to the users so that they can accomplish their purpose.

3.2 PROPOSED APPROACH

This master thesis main goal is to improve the platform Python-Tutor by incorporating new and innovative features to it. The objective is an interactive tool where users can select what kind of graphs/diagrams want to be displayed and be able to focus just on the desired part of the code. Some graphs that can be helpful and can possible integrate this feature are:

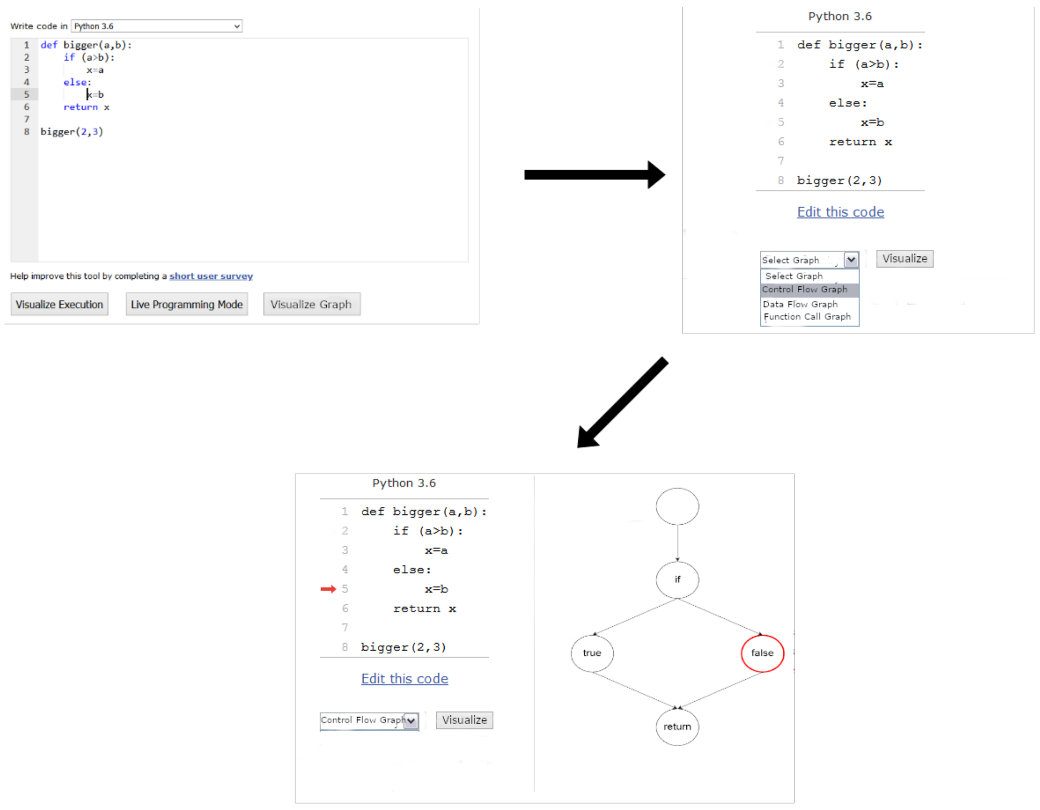
– Control Flow Graph (CFG): It's a representation of all paths that might be traversed through a program during its execution where which node represents a basic block of code. By observing a control flow graph we can see to where the values go and from where they came. It's a very useful graph to realize the dependencies that exist in the program and which statements are influenced by others. A statement B is control dependent on a statement A if B execution depends of the A outcome [Zel09].

– Data Flow Graph (DFG): In this graph we have two types of nodes where one represents the values/variables and other the operators[Wol12]. This graph allows the user to observe all operations that any variable suffers in its lifetime and the existing data dependencies. It's said that B is data dependent of A when A's data is used in B.

– Function Call Graph (FCG): This type of graph displays the relationships between a function and the functions it calls[Hol16]. It allows the user to identify the most used functions and the ones that are not being used at all. Knowing which functions are most called can be useful to improve the program performance since improving these functions will affect a big part of the program.

3.3 SYSTEM WORKFLOW

This feature will be incorporated on Python-Tutor so it has to be adapted to it. Currently the Python-Tutor displays buttons to execute the features previously mentioned, so what is intended to do is to add another one for the visualization of graphs. It will now be shown a mock-up of the desired flow.


Write code in Python 3.6

```
def bigger(a,b):
    if (a>b):
        x=a
    else:
        x=b
    return x

bigger(2,3)
```

Help improve this tool by completing a short user survey

Python 3.6

```
def bigger(a,b):
    if (a>b):
        x=a
    else:
        x=b
    return x

bigger(2,3)
```

Edit this code

Python 3.6

```
def bigger(a,b):
    if (a>b):
        x=a
    else:
        x=b
    return x

bigger(2,3)
```

Edit this code

Figure 11: System Workflow

3.4 SYSTEM ARCHITECTURE

The basis of this project will be the current Python-Tutor architecture, so it will be developed in python, like Python-Tutor. The user input will be passed through the Python-Tutor front-end and then it will be parsed by Python-Tutor and follow the existent flow, or by the new feature to be developed. The new feature will require the input to be parsed to collect the information needed to build each graph. Depending of the requirements, the user input will be parsed by a existent python parser or it will be required to build one. In case of it be the last option, the new parser will be build with *ANTLR*. Once the information about the user program is all gathered the next step is to build the graphs. Initially the graphs will be build in python with the help of some libraries like *igraph* for example. Then it may be required using external tools to accomplish a better visual graphs, interactive and more appealing for users.

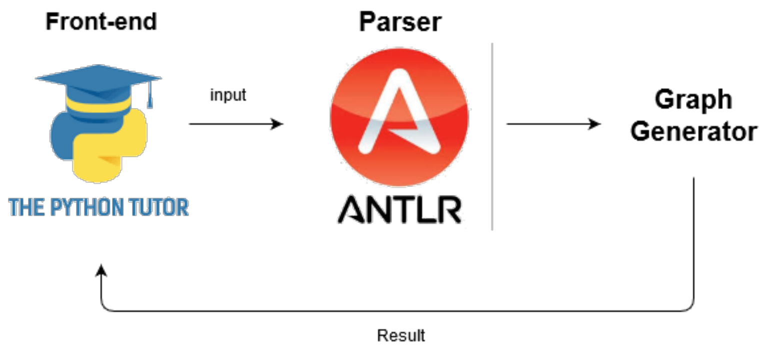


Figure 12: System Architecture

4 GRAPH GENERATOR

In this chapter it will be presented all the steps required to build any type of graph that is proposed on this thesis since the more technical work like the gathering of information to the more visual one, like the drawing of graphs and examples of its result.

4.1 GRAMMAR

To gather the information that is presented in a input written in python language it's needed to understand that language and its rules. To do this it is needed to build a grammar. A grammar is a set of rules that the input has to follow to be recognized. In this specific case, it is required a grammar that follows python rules. Thereby, in this project it is used a combination of two grammars made available on ANTLR's github. One grammar[KLSK14] was used because of how well and simple it is written and because it is prepared to python 2 as well as python 3, and the other one[Kie14] was used as a guideline to use the first one with python programs.

In some cases, related to the python language and the information necessary to extract, it was required to make some minor modifications on the original grammar like splitting one grammar rule into three different ones in order to treat the data more specifically.

On the other hand, the actions of each rule had to be constantly updated. Every time some piece of code allowed to detect an unhandled case of at least one of the graphs, actions had to be added or changed to one or more rules.

How the information was processed had to be the most accurate as possible and the data passed the necessary to build the graphs faithfully, so the structures and what they contain also had to be updated in order to match the needs. On the next section, the final form of this structures to each graph will be presented and explained as well the information they hold.

In order to cover all cases, the rules had to ”communicate” between themselves because each rule had some important information that another rule would need to know in order to save the data accurately. This communication couldn't be done in a direct way because this types of rules weren't always executed one after another and some rules handled information for more than one graph. So, in order to kept all the data available for each rule, it was necessary to create a big amount of local variables.

This local variables were divided in two types, there are variables with the purpose of saving information that is not complete yet and others that just hold a boolean value. The last type were created just to pass informative value for example, the data presented in a segment of code will be treated different depending if it's in a conditional body or not, so there's a variable to allow the program to know in what context that segment had appeared.

4.2 GRAPHS

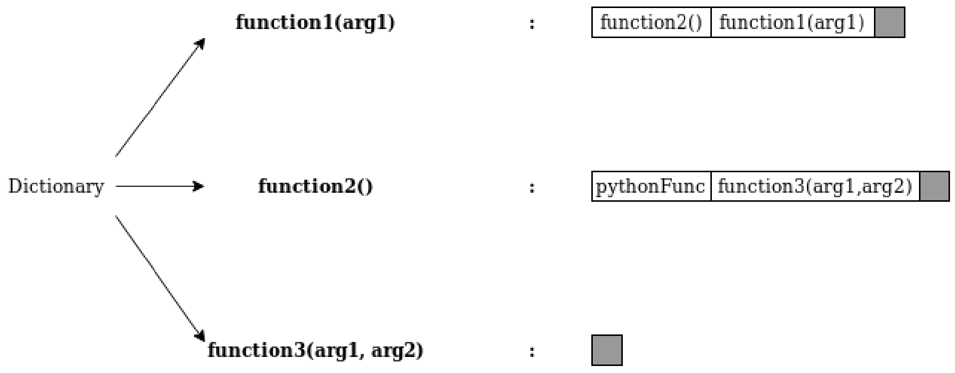
In this section it will be shown how the python programs have its information recognized by the grammar and how it is treated and gathered for each type of graph.

4.2.1 *Function Call Graph*

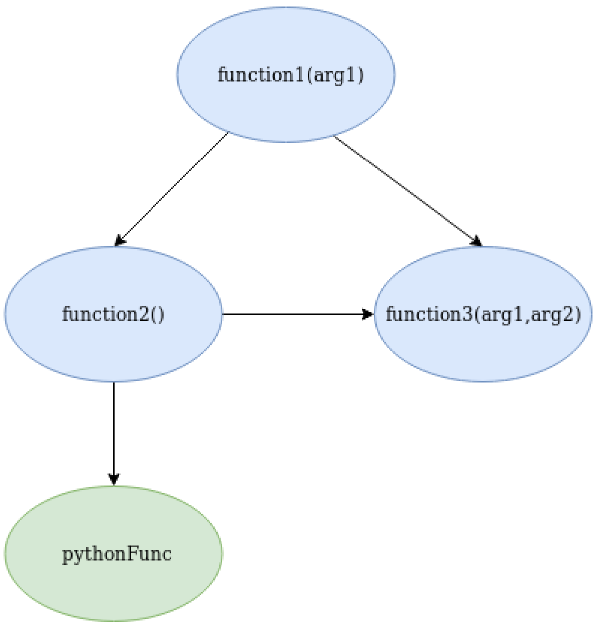
Let's begin with function call graph. There are two types of functions that can be called in any program, functions defined by the developer or functions that are available in python libraries. A function is recognized by a word (its name) followed by parenthesis or, in case of being a function written by the programmer and its first appearance being its definition, by the keyword "def".

The functions calls present in a program are gathered in a python dictionary where the key is the name of the function that calls the others functions and the value the functions that are called.

On figure 13 it can be observed how it's made the storage of the information required to create a Function Call Graph and its result. On the final graph the two types of functions are distinguished by color, blue for the functions created by the user and green for python's functions. As it is shown, the defined functions allow the program to identify its arguments.



(a) Visual representation of call graph structure



(b) Result of the information saved on structure

Figure 13: A visual example of how the data of function call graph is stored and its result

It will be now presented a graph generated by the software built to one program that simulates the positions (by coordinates) of hockey players during a game.

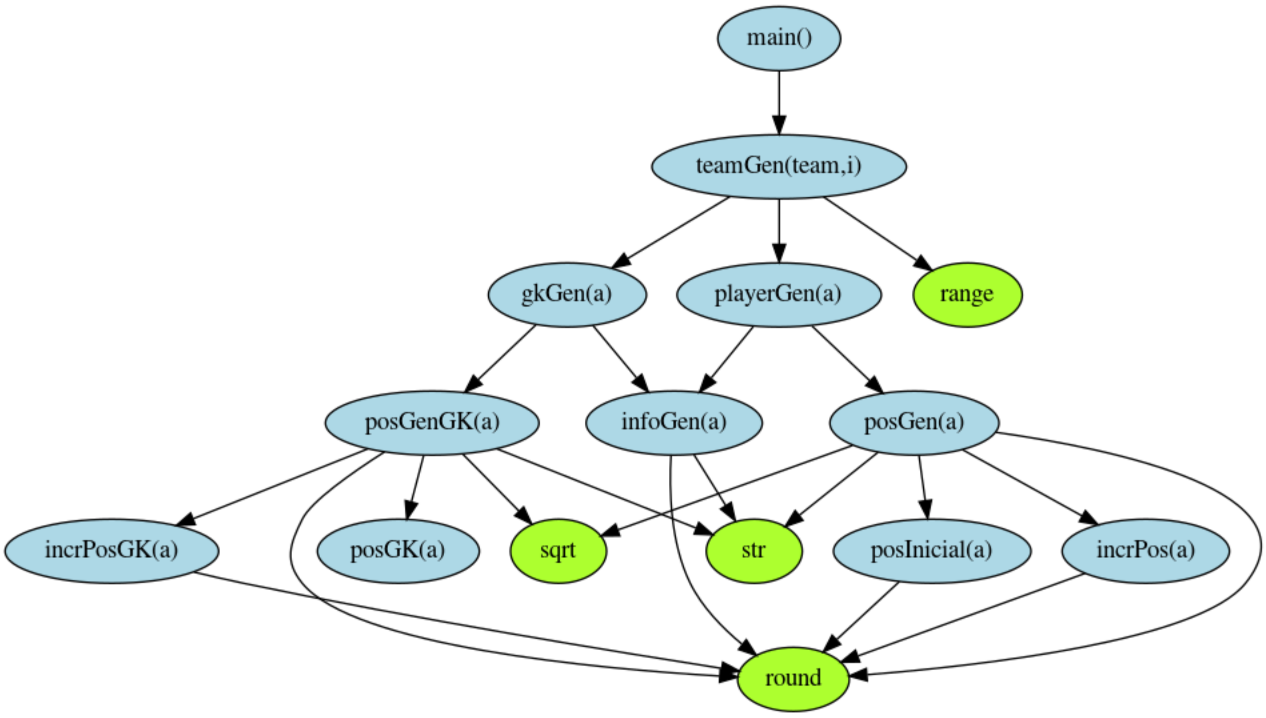


Figure 14: Function Call Graph example

4.2.2 *Control Flow Graph*

The data for the construction of a CFG is stored in a similar way of the previous graph, that is in a dictionary with an entry for each defined function. The information linked with each function is the part that changes. As the construction of this graph is not so linear as the previous one, it is required to save more information on the dictionary. So, instead of saving just the statement that is in the code it also saves the type of that statement and the number of body where it appears.

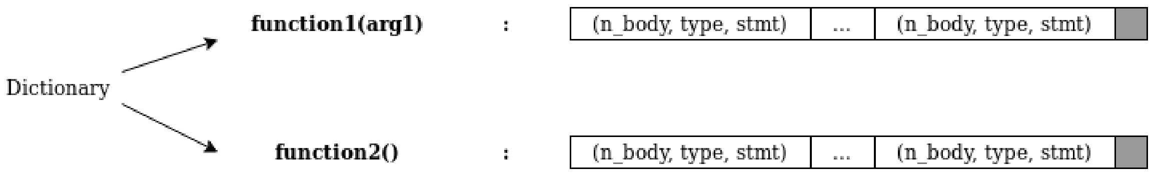
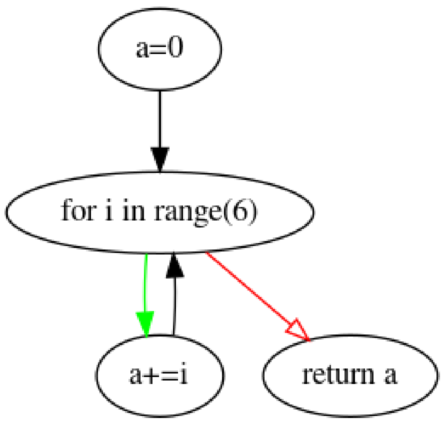
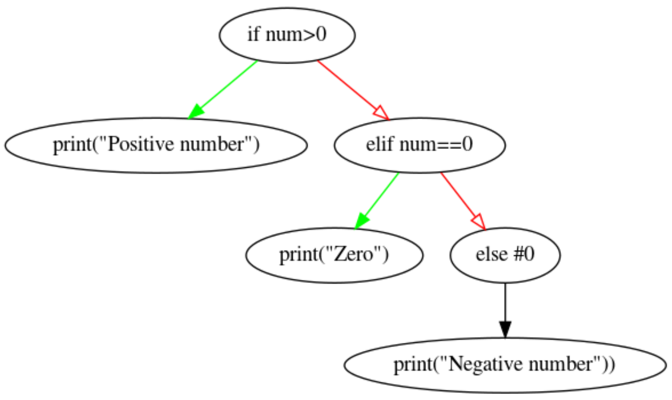


Figure 15: Visual representation of data structure of Control Flow Graph

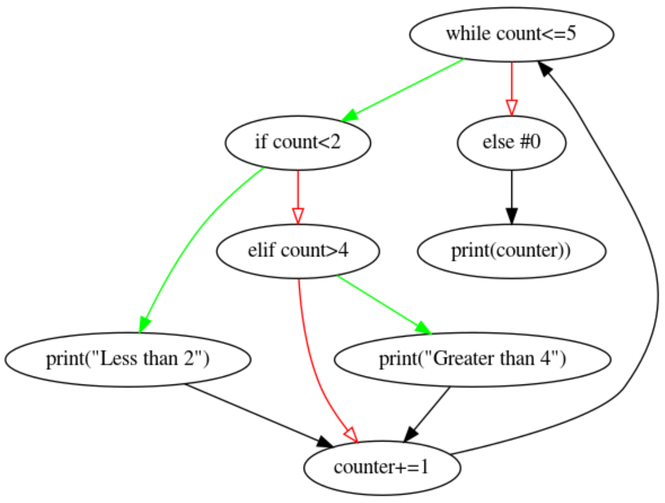
A statement is each line of code that is in the input and can have four different types, it can be of the **simple** type that represents statements that not affect the flow of execution. It can be of the **loop** type (a ”for” or a ”while”), a **conditional** which is separated in two types, **”if”** or **”elif”**, or the last case, it can be of an **”else”** type. These types help building this graph because it adds information to the node that will represent each statement. For example, a node of one statement of the ”loop” or ”conditional” type will have to have two edges coming out of him, one in case of the condition being true and the other in case of being false. A ”loop” node will also have to have at least one more edge coming in from the last statement inside the loop. An ”else” node will always be preceded of one node of the others non-simple types.



(a) A loop representation



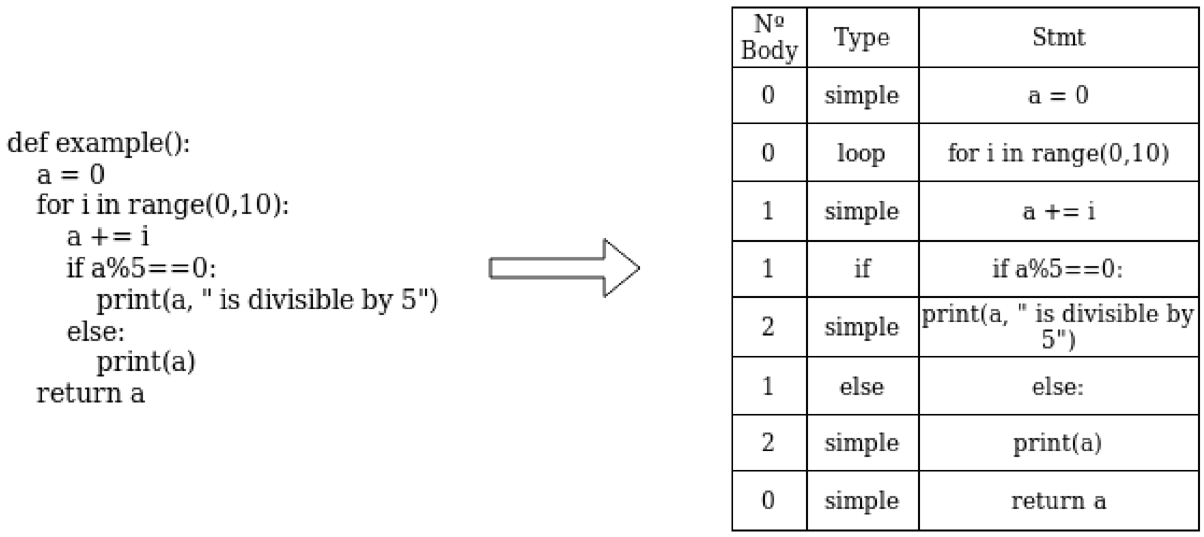
(b) Conditional statements



(c) A loop with conditional statements in its body and an else for loop condition

Figure 16: Examples of CFG with loops and conditionals statements

The number of body represents the number of tabs before the statement, in other words, if that statement depends of a condition. The body (group of statements that depends of one statement) of non-simple statements will always have at least one more number of body than that statement itself. When the number of body changes from one statement to another, it allows the software to recognize that it will start a new body of statements or that one just ended.



```
def example():
    a = 0
    for i in range(0,10):
        a += i
        if a%5==0:
            print(a, " is divisible by 5")
        else:
            print(a)
    return a
```

Nº Body	Type	Stmt
0	simple	a = 0
0	loop	for i in range(0,10)
1	simple	a += i
1	if	if a%5==0:
2	simple	print(a, " is divisible by 5")
1	else	else:
2	simple	print(a)
0	simple	return a

Figure 17: Transformation of code in data

There's other statements that affect the flow of a program but do not have a special type, like for example the reserved words **break** and **continue**. Statements composed with one of these words are of simple type because the statement is just the word itself. As they are easily recognizable there was no need of creating a special type for them.

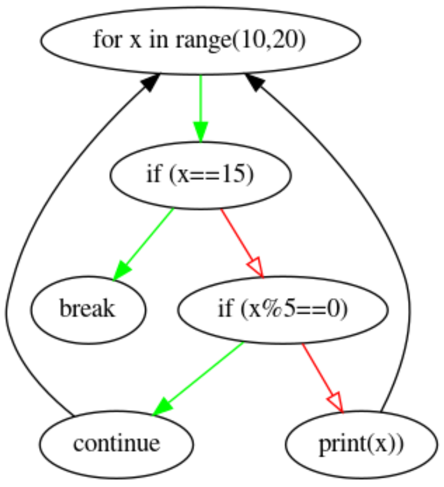


Figure 18: Example of a CFG with break and continue statements

As it is observable in the figure 16 and 18, the CFG also contains coloured arrows in order to be more intuitive for the user to identify which path is followed if the condition present in the node is true (green arrow) or false (red arrow). Besides the color difference, the true path arrowhead is filled unlike the false one, which is non-filled. This difference was implemented in order to the feature become more accessible to different users.

4.2.3 *Data Flow Graph*

Unlike the others graphs, this one is not so linear because the dependencies of one variable depends not only of the other variables or functions results but also of loops or conditional statements. There's also variables that only exist on a determined context like inside of the bodies of the statements previously mentioned.

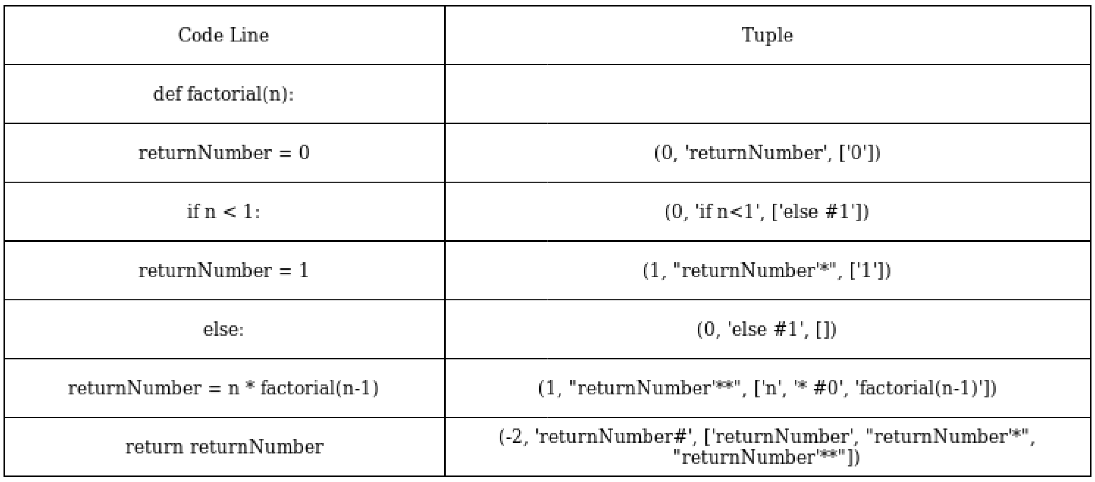
Although this conditions, the information needed to build this type of graph is gathered in the same way of the previous ones, more specifically like is shown on figure 15 but the data on the tuples is slightly different. Moreover, there are three types of tuples in the construction of this graph, one that represents a change of value of a variable, one that serves to control the loops and conditionals dependencies and the other one it will be explained later.

The first type of tuple is composed by the number of body, the name of the variable that had its value changed and the new value of the variable or operations that produce that new value. The last data is passed inside a list where each of its positions corresponds to a different node.

The second type is also composed by the number of body but, instead of the name of the variable, it has the conditional or loop statement and on the third position of the tuple the alternative path, if exists, of that statement.

Thanks to conditionals and loops statements, there are points of the program that one variable can have multiple values depending on the path that the execution of the program took. For example, if we have a program that has the following line of code $a = 5$ and inside an condition statement has $a \mathrel{+}= 1$, at the end of the program the value of a can be 5 or 6 depending on the value of the condition statement. And that's the reason why is a third type of tuple, to gather all the possibles values in just on node. So, the third type is composed by a number that identifies this type of tuple, which in this case is "-2", the name of the "final" variable and a list of the names of variables that contain the possible values that the "final" one can have.

There's shown on figure 19 an example of how these tuples are created as the code are being parsed. Note to the names of variables that contain symbols that help to identify the nodes referring to the same variable. This symbols are not shown on the final graph as we can see on figure 20.



Code Line	Tuple
def factorial(n):	
returnNumber = 0	(0, 'returnNumber', ['0'])
if n < 1:	(0, 'if n<1', ['else #1'])
returnNumber = 1	(1, "returnNumber'*", ['1'])
else:	(0, 'else #1', [])
returnNumber = n * factorial(n-1)	(1, "returnNumber'**", ['n', '* #0', 'factorial(n-1)'])
return returnNumber	(-2, 'returnNumber#', ['returnNumber', "returnNumber'*", "returnNumber'**"])

Figure 19: Transformation of code in data

In this graph was fundamental to use colors on the nodes for it became more intuitive for the user. Therefore it is used the colour green on the nodes that represent the variables which value changed and in red the node that aggregates all possible values of that variable at determined moment. The other nodes illustrate values or operations.

For it became more intuitive and informative there's also boxes on the graph representing loops or conditional statements. The nodes inside these boxes are dependent of these statements. In the figure 20 is possible to visualize how the information displayed in the figure 19 is demonstrated.

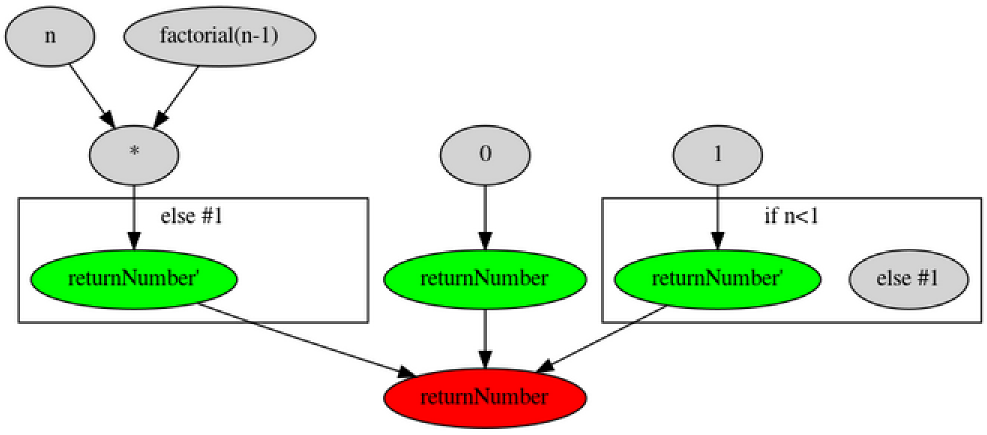


Figure 20: Example of one DFG

Analysing the code with the graph generated it is possible to conclude that the graph is not 100% accurate because the final state of the variable *returnNumber* never will be zero like the graph suggest it can be.

This happens because this variable's value change inside the *if statement* as well as in the *else statement*, so the final value it will be one of that cases.

On the other hand it's hard to keep tracking of all variables that may appear on all conditional statements because one variable may appear on the *if*'s body but not on the *else*'s one and vice-verse. There's also possible *elif* statements that can be numerous, making it even more difficult the process of tracking the variables.

Therefor it was made the decision of showing as one possibility of the final value of a variable be its value before the conditional statements, which in some cases it will be true but on the others it will be not.

5 INTEGRATION ON PYTHON-TUTOR

In this chapter it will be explained how the integration of the new feature with Python-Tutor was done. It was necessary to adapt to the existing system and find the best way to integrate it.

5.1 SYSTEM ARCHITECTURE

After the presentation of the features of Python-Tutor its now time to analyse how it was build to understand how the new feature can be integrated. The system architecture of the software it's really well explained in its *github* page so the best way to describe it is quoting it:

"The Online Python Tutor is implemented as a web application, with a JavaScript front-end making AJAX calls to a pure-Python back-end. The front-end is HTML/JavaScript (using the jQuery library). It's responsible for the input text box, submitting the Python code (as plaintext) to the back-end, receiving an execution trace from the back-end, and then rendering that trace as data structure visualizations. The back-end is a server-side CGI application that takes Python script source code as input, executes the entire script (up to 200 executed lines, to prevent infinite loops), and collects a full trace of all variable values (i.e., data structures) after each line has been executed. It then sends that full trace to the front-end in a specially-encoded JSON format. The front-end then parses and visualizes that trace and allows the user to single-step forwards AND backwards through execution."[Ste11].

The input to use on the graph visualization is inserted on the home page of Python-Tutor as well. It is used the maximum of the existing software as possible, so the input text is also send to the back-end in order to receive the execution trace to check if the code passed as the input is correct and compilable. Once the input is validated, it is time to start using the new code present in the software. The system it will open a new web page created only to visualize the graphs of the input. After the user choose the function and graph that he wants to see the input is now passed through a new route that will pass it to the graph generator. The graph generator will create an image with the graph in question and return its path to the front-end. The front-end it will display the image in its page so the user can visualize it and then remove the image in order to avoid an overcrowding of files.

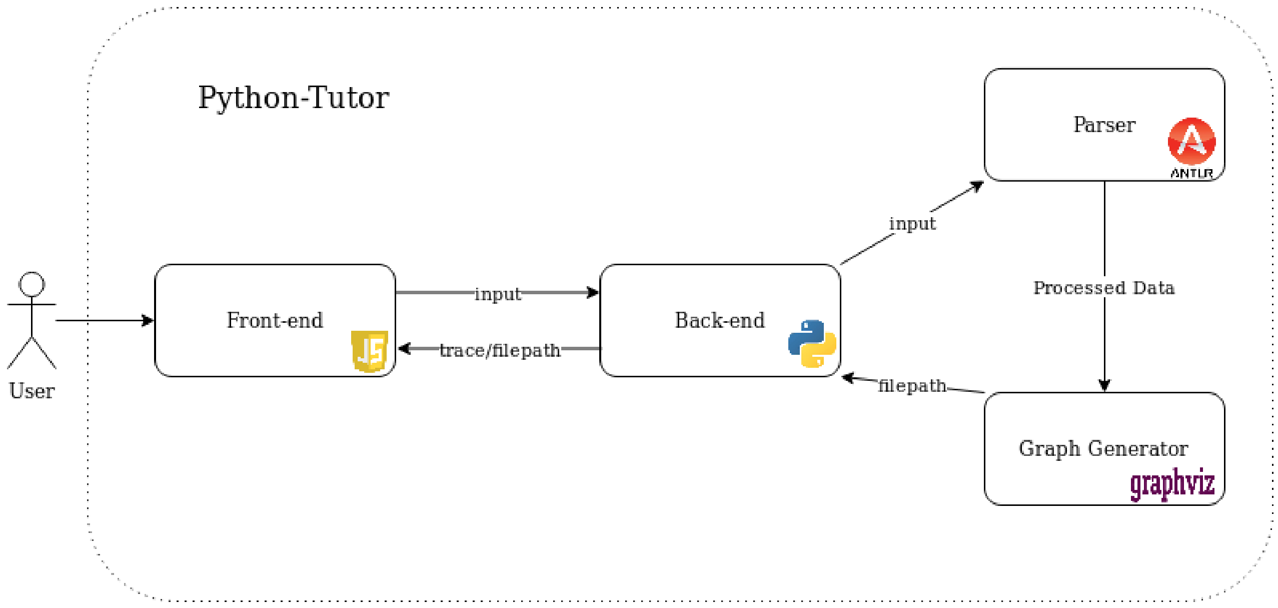
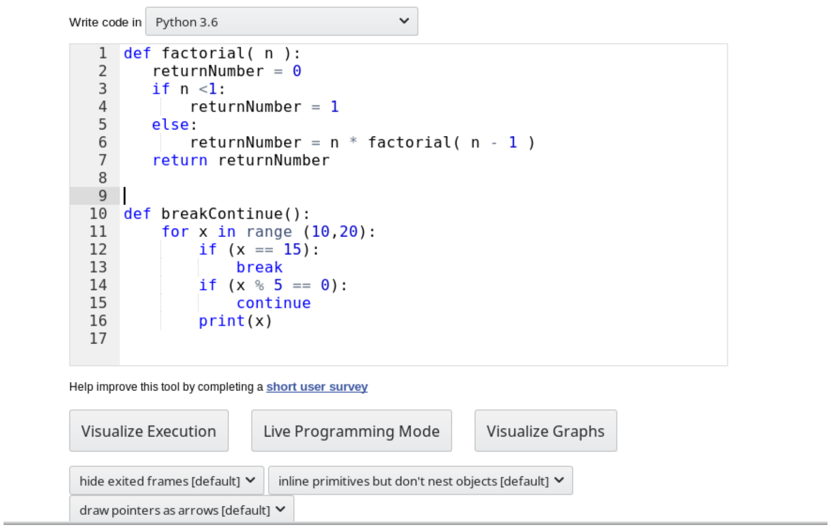


Figure 21: Architecture of Python-Tutor's new feature

5.2 VISUAL

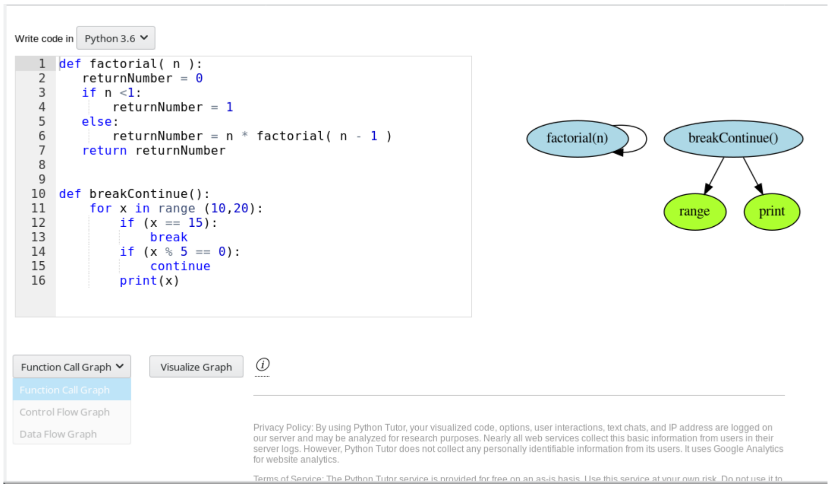
The final version of the visual side of the front-end is really close to the one presented on the proposal. The user can choose the type of graph that wants to visualize. The FCG is the only that is independent of functions because it's a graph to visualize the relationships between the functions (if there's one). The other two are specified to one function so the user has to choose which function he wants to see that graph being applied to.


Write code in
Python 3.6
Visualize Execution
Live Programming Mode
Visualize Graphs
hide exited frames [default]
inline primitives but don't nest objects [default]
draw pointers as arrows [default]

```python
def factorial( n ):
    returnNumber = 0
    if n <1:
        returnNumber = 1
    else:
        returnNumber = n * factorial( n - 1 )
    return returnNumber


def breakContinue():
    for x in range (10,20):
        if (x == 15):
            break
        if (x % 5 == 0):
            continue
        print(x)
```

Help improve this tool by completing a short user survey

(a)

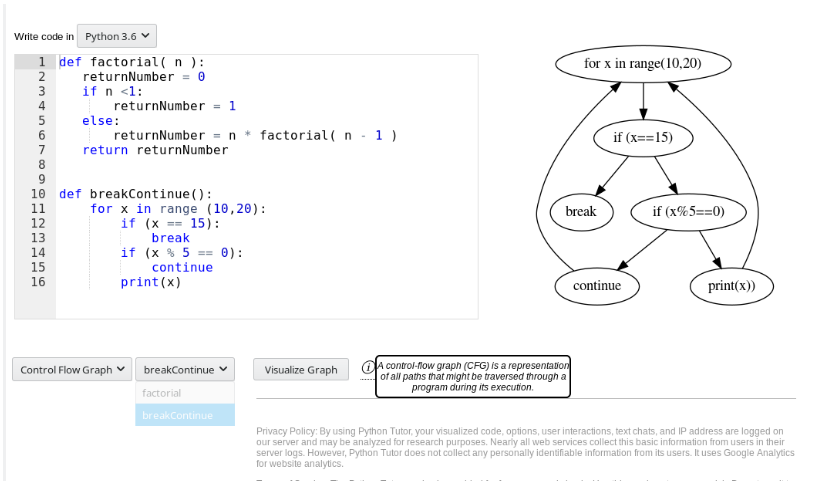


```python
def factorial( n ):
    returnNumber = 0
    if n <1:
        returnNumber = 1
    else:
        returnNumber = n * factorial( n - 1 )
    return returnNumber


def breakContinue():
    for x in range (10,20):
        if (x == 15):
            break
        if (x % 5 == 0):
            continue
        print(x)
```

Privacy Policy: By using Python Tutor, your visualized code, options, user interactions, text chats, and IP address are logged on our server and may be analyzed for research purposes. Nearly all web services collect this basic information from users in their server logs. However, Python Tutor does not collect any personally identifiable information from its users. It uses Google Analytics for website analytics.

Terms of Service: The Python Tutor service is provided for free on an as-is basis. Use this service at your own risk. Do not use it to

(b)



```python
def factorial( n ):
    returnNumber = 0
    if n <1:
        returnNumber = 1
    else:
        returnNumber = n * factorial( n - 1 )
    return returnNumber


def breakContinue():
    for x in range (10,20):
        if (x == 15):
            break
        if (x % 5 == 0):
            continue
        print(x)
```

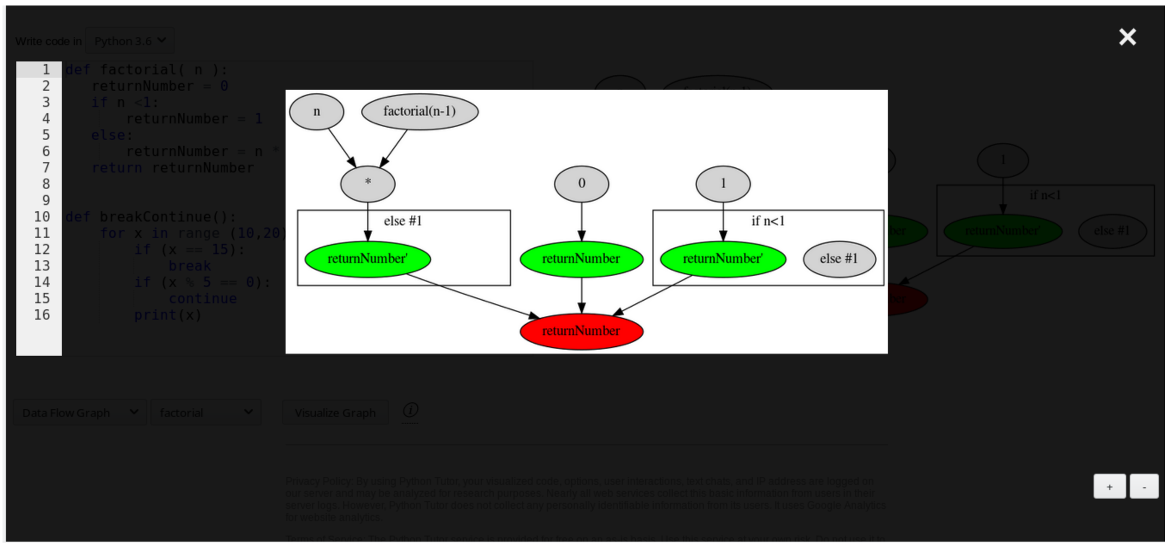
A control-flow graph (CFG) is a representation of all paths that might be traversed through a program during its execution.

Privacy Policy: By using Python Tutor, your visualized code, options, user interactions, text chats, and IP address are logged on our server and may be analyzed for research purposes. Nearly all web services collect this basic information from users in their server logs. However, Python Tutor does not collect any personally identifiable information from its users. It uses Google Analytics for website analytics.

(c)

Figure 22: Visual representation of the new feature

As it can be seen on the figure 23 there's also new features in the front-end that makes the graph clickable so that the user can interact with it, zooming-in and zooming-out allowing the user a better visualisation and comprehension of the graph in question.



```
def factorial( n ):
    returnNumber = 0
    if n <1:
        returnNumber = 1
    else:
        returnNumber = n *
    return returnNumber


def breakContinue():
     for x in range (10,20)
        if (x == 15):
            break
        if (x % 5 == 0):
            continue
        print(x)
```

Figure 23: Zoom feature

6 TESTS AND RESULTS

In order to test and evaluate the new feature of Python-Tutor it was created a survey that contained instructions for the testers to follow and then give their opinion and answer some questions about the feature.

The testers had to satisfy a simple requisite that is they had to be familiarized with *python* in order to be able to write *python* functions and compare it to the resulting graphs. This survey was completed by thirty one participants.

The survey was performed with a tool called *survio* and it was also used a tool called *ngrok* that allowed to share the local-host where the feature was running with the testers.

This survey was made with two purposes:

- test technical aspects like the accuracy of the generated graphs and possible bugs;
- get the insights of the testers on the opinion of the tool for example, if really can help someone understand a program.

These two goals were accomplished as it was detected some errors in the graphs that were generated related to specific ways to code in *python* that were corrected afterwards, like for example multiple assignment (assign multiple values or the same value to multiple variables). This survey also allowed to gather opinions on what could be done to improve the intuition of the feature and some of these considerations were taken like the colourful arrows of the CFG.

In the next section it will be made a more detailed analysis of the results.

6.1 RESULT ANALYSIS

This section will be sub-divided in two categories that corresponds to the two purposes mentioned previously. Firstly, it will be presented the results related to the technical side like the accuracy of the tool and the level of the complexity of the functions used to test the feature.

Then it will be analysed the opinion of the testers of the feature. This results, unlike the first ones, will be analysed individually to each graph so that we can understand which graphs can help on a better program comprehension.

6.1.1 *Programs Complexity*

The first question of the survey was related to the complexity of the code the tester were writing. This question was asked with the intuition of relate the accuracy of the resulting graph with its complexity, but as it is observable on the pie-chart of the figure 24, the majority of the functions written to test were of low complexity which, in this case, it's not bad because as Python-Tutor is directed to new programmers the expected inputs are of low to medium complexity.

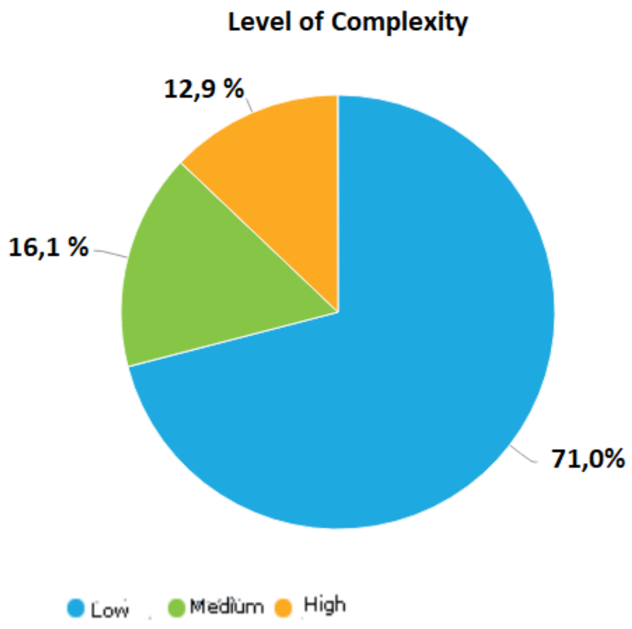
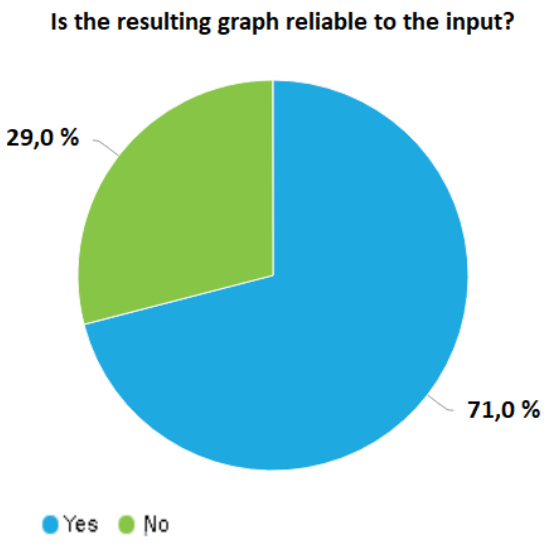


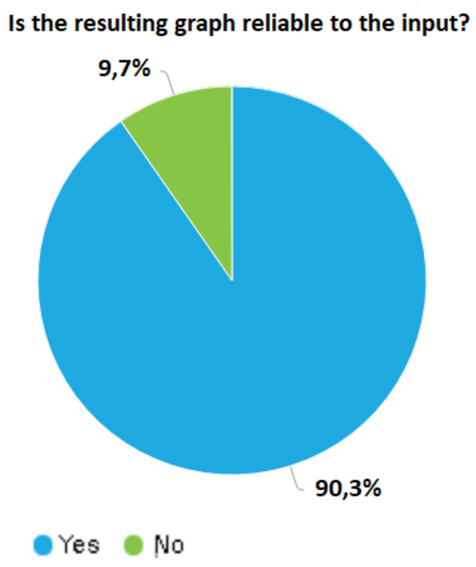
Figure 24: Complexity level of functions used to test the feature

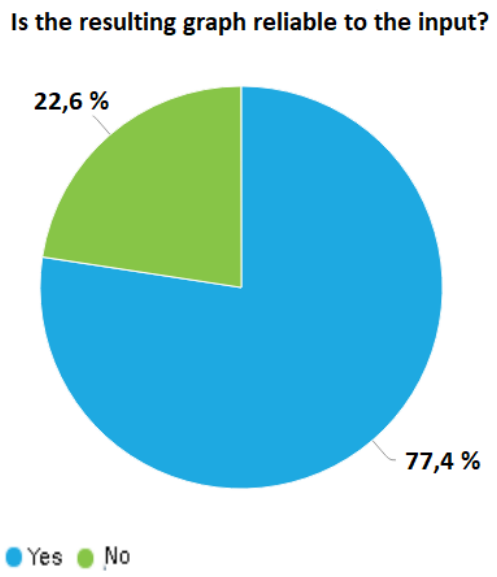
6.1.2 *Graphs Accuracy*

After having an overview of the type of programs that were tested, we will now analyse the tool accuracy by each graph.



(a) FCG Accuracy





(b) CFG Accuracy

(c) DFG Accuracy

Figure 25: Accuracy of each type of graph

The results indicate a good percentage of accuracy of each graph, with the most surprisingly result being the FCG where it was expected to have the most high accuracy but ended up being the one with the least. On the other hand, it was refreshing realise how accurate was the CFG graph. This results were very satisfying mainly because the bugs reported were corrected which means that this accuracy is now increased.

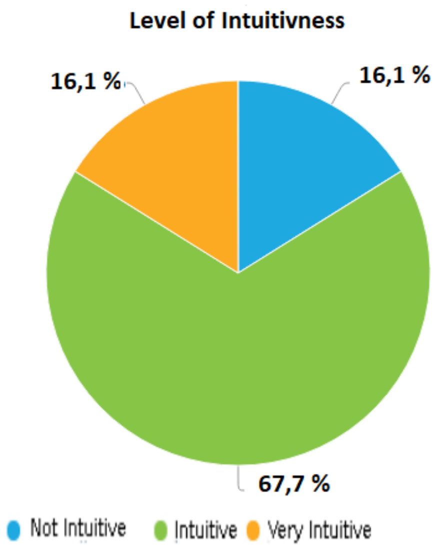
6.1.3 *Graphs on Program Comprehension*

It is time now to see if this graphs can really help people on comprehending a program. There are two questions for each graph involving this area, first the intuitiveness of each graph, or in other words, if the graphs are easily read. For it be capable of helping one understand their program, the graphs can't be harder to understand than the program, so an intuitive graph was always the goal and now we'll see if it was accomplished.

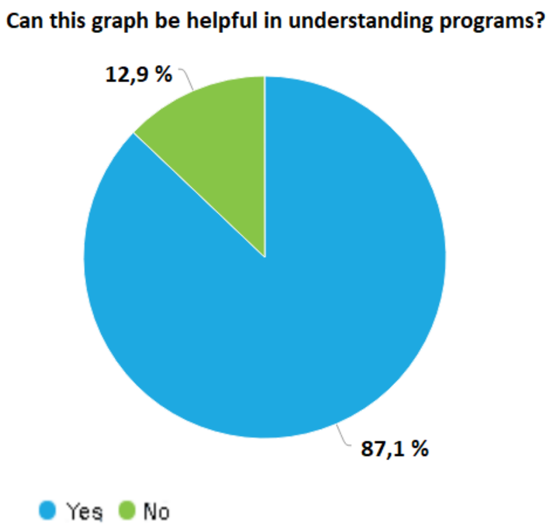
The second question is a really direct one that is if the tester thinks the graphs can help the comprehension of a program which is the whole idea behind of this thesis.

6.1.3.1 *Function Call Graph*

Following the order presented on this whole paper, let's start with the FCG. With an 83.8% of positive response on intuitiveness, the people who tested the tool think that, although of it being a very simple graph displaying minor information as it only shows the functions who are called by other functions, it really can improve a program comprehension.



(a) FCG Level of Intuitiveness

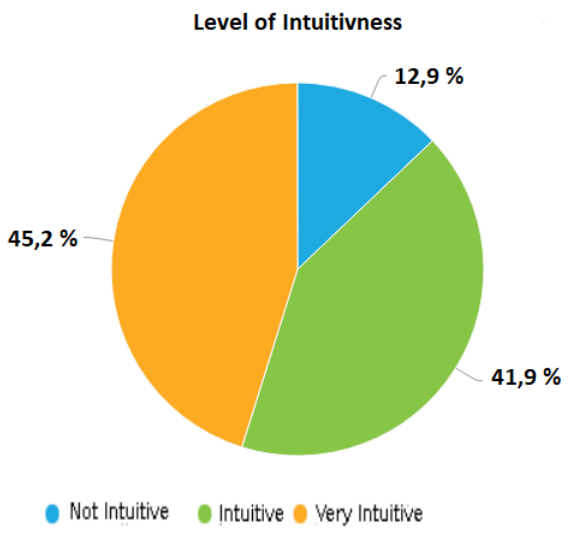


(b) FCG On Program Comprehension

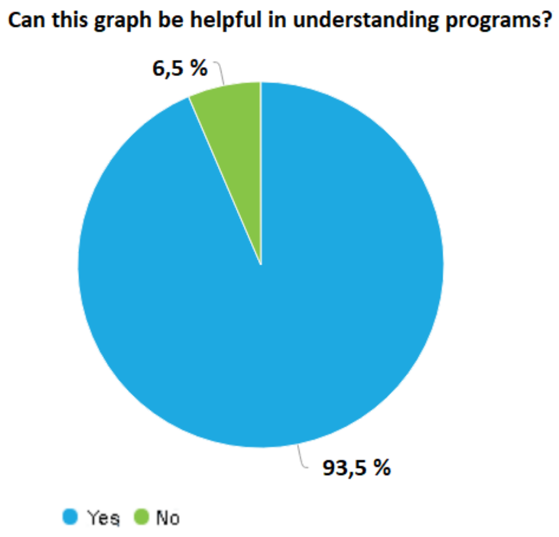
Figure 26: Perceptions on FCG

6.1.3.2 *Control Flow Graph*

These results come to reinforce the preconceived idea that this graph will be the strength of the new feature. This vastly positively results boost the confidence that this graph can really be helpful on program comprehension.



(a) CFG Level of Intuitiveness

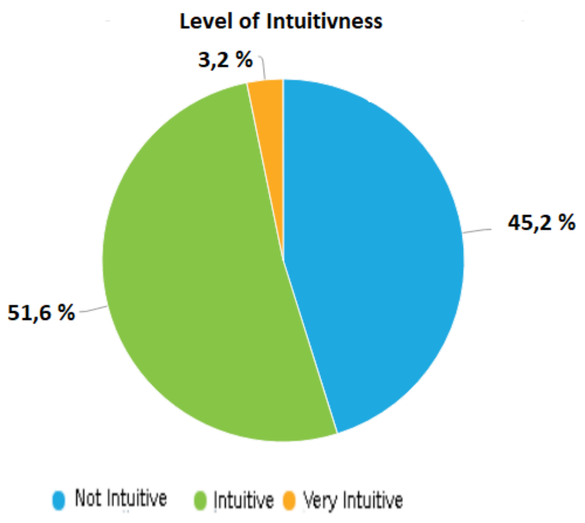


(b) CFG On Program Comprehension

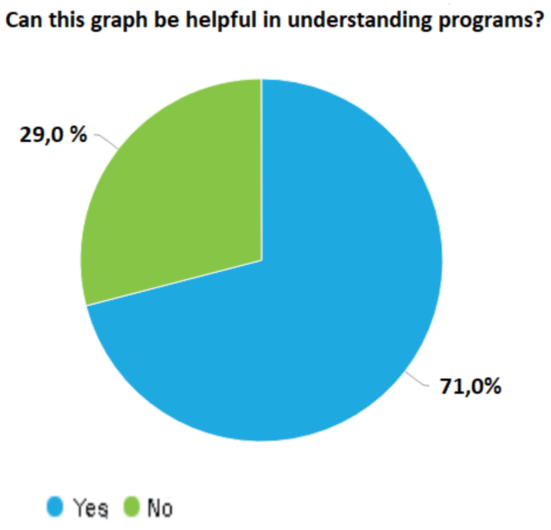
Figure 27: Perceptions on CFG

6.1.3.3 *Data Flow Graph*

As expected, this one was considered the least intuitive graph. This graph was the most difficult to conceive and it really shows. Although the lack of intuitiveness of this graph, a considerable high amount of people still thinks that it can improve the program comprehension, which means that trying to improve the presentation of this graph as to be a priority in the future.



(a) DFG Level of Intuitiveness



(b) DFG On Program Comprehension

Figure 28: Perceptions on DFG

6.1.4 *Final thoughts*

At the end, these results were very positive and increased the confidence on this feature and what it can do by program comprehension. This survey let us identify some minor bugs on the construction of the graphs that after corrected increased the accuracy and the intuitiveness of the graphs, allowing ending up with a better final version of this tool.

7

CONCLUSION

The last chapter of the dissertation presents the final considerations of all the work carried out and also on the objectives that have been achieved. In addition, some improvements are defined as important for the future of the project.

To understand a program after it has been fully developed is an hard task for programmers in general, no matter if he is an expert or even its author. This fact creates big difficulties in the area of program maintenance that is a necessary activity which takes a lot of time and consumes many resources in the software companies. To overcome this actual difficulty, researchers in various areas of computing started a joint effort to work out a new topic called Program Comprehension aiming at the creation of theoretical analytic instruments and the development of tools to make this process easier.

The use of CFG, DFG and FCG are some of those instruments that help to visualize statically some program perspectives that depict its behavior.

On the other hand, program animation tools were developed to help in Program Comprehension as well as in teaching introductory programming courses. Program Animators are a kind of high-level program debuggers that simulate the running of a program while showing the internal (in terms of variables and computer memory) effect of each instruction executed. The hope of Program Animators' builders is that this visual representation of the effects produced by each instruction can effectively help program Analysers to comprehend them faster.

The Master's work performed and described along the present dissertation is located in the intersection of the above referred two areas: the use of graphs for Program Comprehension, and the use of Program Animation tools.

As can be expected, at the beginning a wide and deep research was done to build the state-of-the-art in Program Comprehension, Software Visualization, and respective tools, as was written in Chapter 2. That chapter also describes in details Python-Tutor, the animator chosen for the integration of graphs due to its large usage and nice capabilities.

After understanding clearly the project objectives and research approach, and after the literature review, it was planned how to solve the problem and then an architecture of the system to be developed was drawn as discussed in Chapter 3.

Chapter 4 and 5 describe the implementation of the functions needed to create the various graphs intended to be incorporated into the animator, as well as the integration process.

To support the thesis below, an experiment with real programmers was designed and conducted to assess the developed system. The description of the experiment, results so far attained and their discussion are the content of chapter 6.

At the end, the Master's Thesis that was proven with the project can be stated as: It is possible to include control-flow and data-flow graphs into Program Animation systems, providing in that way to the programmer a better aid in the comprehension of programs' behavior.

7.1 FINAL CONSIDERATIONS

In the end of this master's project, it can be said that the main goal was achieved: to build a tool that transforms a function to its corresponding control and data flow graphs and to integrate it into the program animator tool Python-Tutor. The process of building this tool had obviously some obstacles that had to be overtaken. The biggest difficulty faced was related to the Python Grammar used that does not cover all the real program situations causing many compiling exceptions that frequently arose. Other challenge was the integration with Python-Tutor as this platform resorted to some outdated libraries. However, all these troubles have been overcome ending up with a functional tool to help understand a program like it was meant to be.

7.2 FUTURE WORK

As future work, the first proposal is to implement the same control-flow and data-flow graphs for the other languages presented in the Python-Tutor like *Java* and *C*. It is expected to be easier to implement it now as the new languages will only need the construction of a grammar for each language as the graph generator can be reused and the integration with Python-Tutor would only need a few adaptations.

The other direction for the future of this Python-Tutor add-on is to improve the aspect of the DFG as the testers inquired said that it is not very intuitive and easy to comprehend.

BIBLIOGRAPHY

[BE95] Marla Baker and Stephen Eick. Space-filling software visualization. *Journal of Visual Languages Computing*, 6:119–133, 06 1995.

[Bro83] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543 – 554, 1983.

[GH01] Luis M. Gómez-Henríquez. Software visualization: An overview, 2001.

[Had18] Martin Hadley. 3 benefits of interactive visualization. 01 2018.

[Hol16] Ben Holland. Call graph construction algorithms explained. 03 2016.

[JS91] B. Johnson and B. Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *Proceeding Visualization '91*, pages 284–291, 1991.

[Kie14] Bart Kiers. Python 3 parser. https://github.com/antlr/grammars-v4/tree/master/python/python3-py, 2014.

[KLSK14] Bart Kiers, Dmitriy Litovchenko, Nikita Subbotin, and Ivan Kochurkin. Python 2 and 3 universal grammar. https://github.com/antlr/grammars-v4/tree/master/python/python, 2014.

[LS06] François Lemieux and Martin Salois. Visualization techniques for program comprehension - a literature review. In *SoMeT*, 2006.

[MV93] A. Mayrhauser and A. Marie Vans. From program comprehension to tool requirements for an industrial environment. pages 78 – 86, 08 1993.

[MV95] Anneliese Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28:44 – 55, 09 1995.

[NS73] Ike Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8:12–26, 08 1973.

[OBS04] Michael O'Brien, Jim Buckley, and Teresa Shaft. Expectation-based, inference-based, and bottom-up software comprehension. *Journal of Software Maintenance*, 16:427–447, 11 2004.

[Pen87] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[Pet02] Marian Petre. Mental imagery, visualisation tools and team work. 01 2002.

[Put] Ben Putano. A look at 5 of the most popular programming languages of 2019. https://stackify.com/popular-programming-languages-2018/. Accessed: 26-9-2019.

[Ram86] Gerard Rambally. The influence of color on program readability and comprehensibility. volume 18, pages 173–181, 02 1986.

[SCGM00] John Stasko, RICHARD CATRAMBONE, Mark Guzdial, and KEVIN MCDONALD. Evaluation of space-filling information visualizations for depicting hierarchical structures. *International Journal of Human-Computer Studies*, 53:663–694, 11 2000.

[SE84] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, Sep. 1984.

[Sie16] J. Siegmund. Program comprehension: Past, present, and future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 13–20, March 2016.

[SMMH77] Ben Shneiderman, Richard Mayer, Don McKay, and Peter Heller. Experimental investigations of the utility of detailed flowcharts in programming. *Commun. ACM*, 20:373–381, 06 1977.

[Ste11] Michael Stewart. Online python-tutor. https://github.com/hcientist/OnlinePythonTutor, 2011.

[SV98] Teresa M. Shaft and Iris Vessey. The relevance of application domain knowledge: Characterizing the computer program comprehension process. *J. Manage. Inf. Syst.*, 15(1):51–78, June 1998.

[Syn] Syncfusion. Hierarchical levels.

[Wol12] Marilyn Wolf. *Computers As Components, Third Edition: Principles of Embedded Computing System Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2012.

[Yos08] Mami Yoshida. Think-aloud protocols and type of reading task: The issue of reactivity in l2 reading research. 01 2008.

[You14] Stephen Young. Why your code is so hard to understand. 11 2014.

[Zel09] Andreas Zeller. Chapter 7 - deducing errors. In Andreas Zeller, editor, *Why Programs Fail (Second Edition)*, pages 147 – 173. Morgan Kaufmann, Boston, second edition edition, 2009.

A

SURVEY

In this appendix is presented the Survey (questions, introduction and comments) which answers were analysed in Chapter 6.

A.1 SURVEY FULFILLMENT INSTRUCTIONS:

Write one or more functions in python and click on the Visualize Graph functionality. To be able to evaluate all types of graphs the functions follow the following requirements:

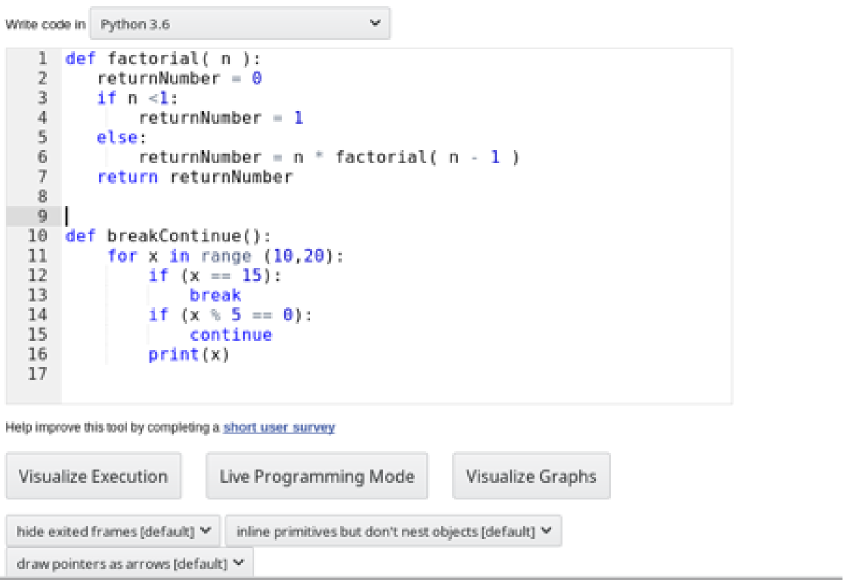
- contain function calls;
- contain assignment / changes of variable values;

These requirements can be present in a single function or spread over several functions. In the following question you can see the type of entry we want.

After exploring the features of the tool, proceed to the questions. Always respond in accordance with your functions/graphs and not with examples displayed.

A.2 SURVEY QUESTIONS: GENERAL

Q1. Level of complexity of your functions:*


Write code in
Python 3.6
Help improve this tool by completing a short user survey
Visualize Execution
Live Programming Mode
Visualize Graphs
hide exited frames [default]
inline primitives but don't nest objects [default]
draw pointers as arrows [default]

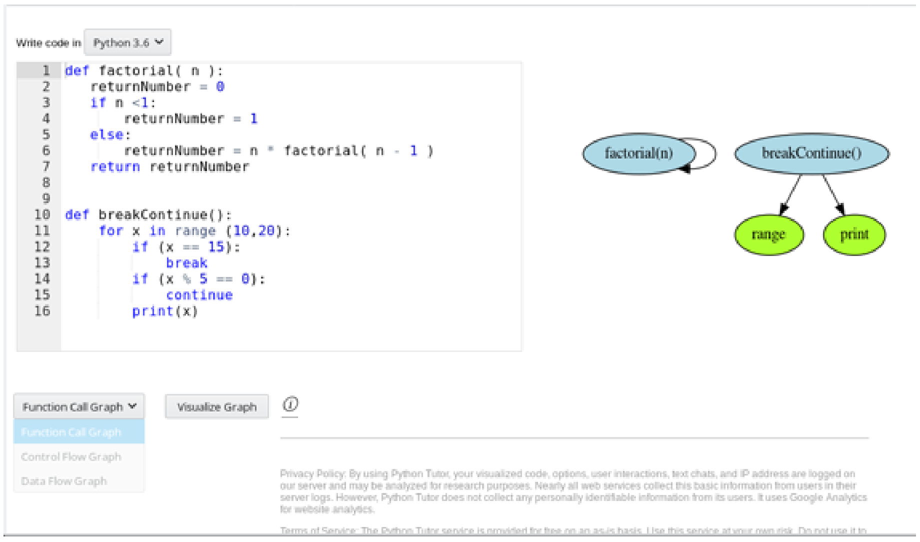
```python
def factorial( n ):
    returnNumber = 0
    if n <1:
        returnNumber = 1
    else:
        returnNumber = n * factorial( n - 1 )
    return returnNumber


def breakContinue():
     for x in range (10,20):
         if (x == 15):
             break
         if (x % 5 == 0):
             continue
         print(x)
```

Figure 29: Example of a valid input

- Low
- Medium
- High

A.3 Survey Questions: Function Call Graph Evaluation

Q2. Do you think that this type of graphs is intuitive and easy to understand?*



```
def factorial( n ):
    returnNumber = 0
    if n <1:
        returnNumber = 1
    else:
        returnNumber = n * factorial( n - 1 )
    return returnNumber


def breakContinue():
    for x in range (10,20):
        if (x == 15):
            break
        if (x % 5 == 0):
            continue
        print(x)
```

Privacy Policy: By using Python Tutor, your visualized code, options, user interactions, text chats, and IP address are logged on our server and may be analyzed for research purposes. Nearly all web services collect this basic information from users in their server logs. However, Python Tutor does not collect any personally identifiable information from its users. It uses Google Analytics for website analytics.

Terms of Service: The Python Tutor service is provided for free on an as-is basis. Use this service at your own risk. Do not use it to

Figure 30: Example of a Function Call Graph

- Not Intuitive
- Intuitive
- Very Intuitive

Q3. Do you believe that this graphs can help to comprehend a program?*

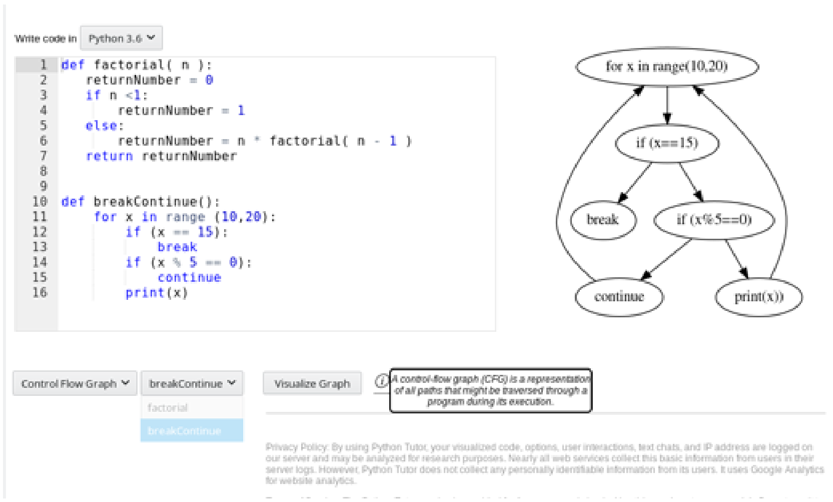
- Yes
- No

Q4. Is the resulting graph reliable to the input?*

- Yes
- No

A.4 SURVEY QUESTIONS: CONTROL FLOW GRAPH EVALUATION

Q5. Do you think that this type of graphs is intuitive and easy to understand?*



```
def factorial( n ):
    returnNumber = 0
    if n <1:
        returnNumber = 1
    else:
        returnNumber = n * factorial( n - 1 )
    return returnNumber


def breakContinue():
     for x in range (10,20):
         if (x == 15):
             break
         if (x % 5 == 0):
             continue
         print(x)
```

A control-flow graph (CFG) is a representation of all paths that might be traversed through a program during its execution.

Privacy Policy: By using Python Tutor, your visualized code, options, user interactions, text chats, and IP address are logged on our server and may be analyzed for research purposes. Nearly all web services collect this basic information from users in their server logs. However, Python Tutor does not collect any personally identifiable information from its users. It uses Google Analytics for website analytics.

Figure 31: Example of a Control Flow Graph

- Not Intuitive
- Intuitive
- Very Intuitive

Q6. Do you believe that this graphs can help to comprehend a program?*

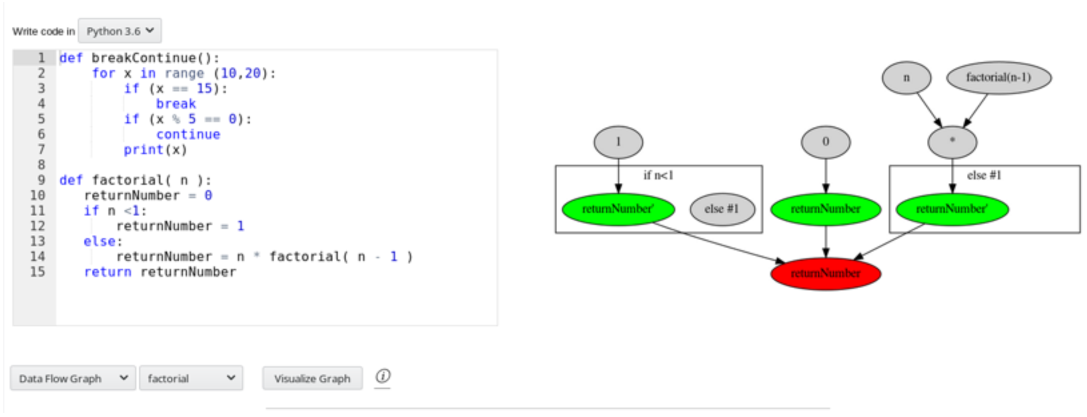
- Yes
- No

Q7. Is the resulting graph reliable to the input?*

- Yes
- No

A.5 Survey Questions: Data Flow Graph Evaluation

Q8. Do you think that this type of graphs is intuitive and easy to understand?*



```
def breakContinue():
    for x in range (10,20):
        if (x == 15):
            break
        if (x % 5 == 0):
            continue
        print(x)

def factorial( n ):
    returnNumber = 0
    if n <1:
        returnNumber = 1
    else:
        returnNumber = n * factorial( n - 1 )
    return returnNumber
```

Figure 32: Example of a Data Flow Graph

- Not Intuitive
- Intuitive
- Very Intuitive

Q9. Do you believe that this graphs can help to comprehend a program?*

- Yes
- No

Q10. Is the resulting graph reliable to the input?*

- Yes
- No

Thanks for your time.

B

INDIVIDUAL ANSWERS TO THE SURVEY

Here is presented all the answers that each participant gave to each question.

#	Q.1	Q.2	Q.3	Q.4	Q.5	Q.6	Q.7	Q.8	Q.9	Q.10
1	Low	Intuitive	Yes	Yes	Very Intuitive	Yes	Yes	Intuitive	Yes	Yes
2	Low	Intuitive	Yes	Yes	Intuitive	Yes	Yes	Not Intuitive	Yes	Yes
3	Low	Intuitive	No	Yes	Intuitive	Yes	Yes	Not Intuitive	Yes	Yes
4	Medium	Intuitive	Yes	Yes	Intuitive	Yes	Yes	Intuitive	Yes	Yes
5	Low	Not Intuitive	Yes	No	Very Intuitive	Yes	Yes	Not Intuitive	Yes	Yes
6	Low	Intuitive	Yes	Yes	Very Intuitive	Yes	Yes	Intuitive	Yes	Yes
7	Low	Intuitive	Yes	Yes	Very Intuitive	Yes	Yes	Intuitive	Yes	Yes
8	Low	Not Intuitive	No	Yes	Very Intuitive	Yes	Yes	Intuitive	Yes	Yes
9	High	Very Intuitive	Yes	Yes	Very Intuitive	Yes	Yes	Very Intuitive	Yes	No
10	Low	Intuitive	Yes	Yes	Intuitive	Yes	Yes	Intuitive	Yes	Yes
11	Low	Not Intuitive	Yes	Yes	Intuitive	Yes	Yes	Not Intuitive	Yes	Yes
12	Low	Very Intuitive	Yes	Yes	Not Intuitive	Yes	Yes	Intuitive	Yes	Yes
13	Low	Intuitive	Yes	No	Intuitive	Yes	Yes	Intuitive	Yes	Yes
14	Low	Very Intuitive	Yes	Yes	Very Intuitive	Yes	Yes	Intuitive	Yes	Yes
15	Medium	Not Intuitive	No	No	Intuitive	Yes	Yes	Not Intuitive	No	No
16	Low	Intuitive	Yes	No	Not Intuitive	No	No	Not Intuitive	No	No
17	High	Intuitive	Yes	Yes	Not Intuitive	Yes	Yes	Not Intuitive	Yes	Yes
18	Low	Very Intuitive	Yes	Yes	Very Intuitive	Yes	Yes	Not Intuitive	No	Yes
19	Low	Very Intuitive	Yes	Yes	Very Intuitive	Yes	Yes	Not Intuitive	No	Yes
20	Low	Intuitive	Yes	No	Intuitive	No	No	Intuitive	No	Yes
21	Low	Intuitive	Yes	Yes	Intuitive	Yes	Yes	Not Intuitive	No	No
22	Low	Not Intuitive	No	No	Intuitive	Yes	Yes	Intuitive	Yes	Yes
23	High	Intuitive	Yes	Yes	Very Intuitive	Yes	Yes	Intuitive	Yes	Yes
24	Medium	Intuitive	Yes	No	Very Intuitive	Yes	Yes	Not Intuitive	No	No

Continued on next page

Table 1 – *Continued from previous page*

#	Q.1	Q.2	Q.3	Q.4	Q.5	Q.6	Q.7	Q.8	Q.9	Q.10
25	Low	Intuitive	Yes	No	Very Intuitive	Yes	No	Not Intuitive	No	No
26	Low	Intuitive	Yes	Yes	Very Intuitive	Yes	Yes	Not Intuitive	Yes	Yes
27	Medium	Intuitive	Yes	Yes	Intuitive	Yes	Yes	Intuitive	Yes	No
28	High	Intuitive	Yes	No	Intuitive	Yes	Yes	Not Intuitive	No	Yes
29	Medium	Intuitive	Yes	Yes	Very Intuitive	Yes	Yes	Intuitive	Yes	Yes
30	Low	Intuitive	Yes	Yes	Not Intuitive	Yes	Yes	Not Intuitive	Yes	Yes
31	Low	Intuitive	Yes	Yes	Intuitive	Yes	Yes	Intuitive	Yes	Yes
32	Low	Intuitive	Yes	Yes	Intuitive	Yes	Yes	Intuitive	Yes	Yes