

# Python Code Visualization

## Text to Flowcharts

Sergey Satskiy

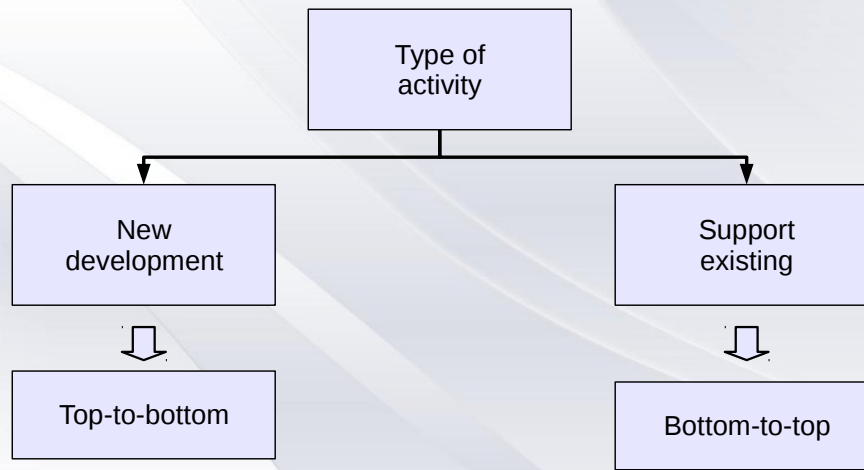
Hello everybody. My name is Sergey Satskiy. I am a software engineer and I met python in 2008. Since then I have been using it for my scripting needs however I must say that python is not my main language. I found python to be a simple, an effective and even a charming language. As a matter of fact this combination inspired me to create a technology and then a tool which supports two ways of working with code depending on one's mood or a certain task: via text or via graphics. If I had known how much time this project would take I might never have started it. But now that it's done, I'd like to present it to you and hopefully you can benefit from it - or even better, take it to the next level.

## Agenda

- Flowcharts in the wild
- Text to graphics: primitives
- Codimension Python IDE
- Implementation
- Further thoughts: features + CML
- Q and A

Here is the agenda of the presentation. I will talk about the role of flowchart diagrams in my experience and give an example of a large scale project which looks quite convincing to me. Then we will discuss the graphics primitives which could be used to represent the code on a diagram. A tool which implements this feature and proves the feasibility of it will be presented after that. The following discussion will be about how the flowchart generation is implemented. And the final topic will be about the other functionality which could be relatively easily added to graphics but could not be added to text.

## Flowcharts: My Experience



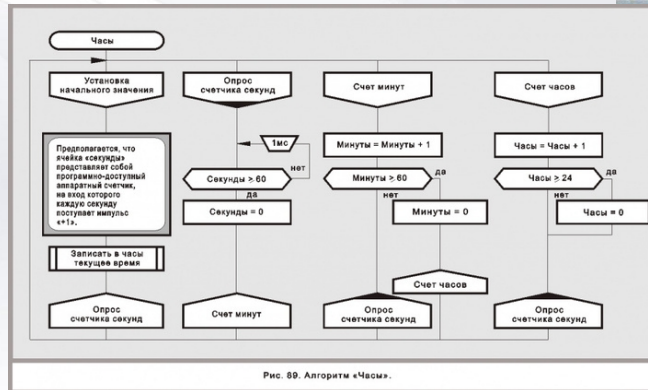
3

Generally speaking, the professional activity of an engineer like me consists of two significantly different scenarios: developing new software from scratch or supporting existing software which in most cases is developed by somebody else. In both scenarios I use diagrams, though in different fashions. In the case of a new development it is usually a top-to-bottom approach. I draw the architecture of the future software as a set of abstract blocks or actors and then I move to more detailed levels eventually reaching the level of the chosen programming language. Certainly, I do not draw everything. I draw only the parts which are of interest to me. Tools like MS Visio help me only when I work on a high level. I wish there was a tool which would help me with the lower level, and preferably in two complementary ways: generating a diagram from code and vice versa. I have never found such a tool.

In the case of maintaining existing code it is the usual – and sad – situation that there is no documentation and I have to reverse engineer the ideas behind the code. So I proceed in a bottom-to-top way. I read the code and when I understand what a chunk of it does I draw – physically on a piece of paper or mentally – a block with an appropriate label on it. Thus I get a single block (or a chunk) with a label instead of a group of statements. Eventually I get a flowchart which corresponds to the code. Again, I wish there was a tool which supports this process conveniently but I have not found it yet.

# Flowcharts: Large Scale Projects

DRAKON language



Buran Spaceship

All right, I was talking about my habits, but what about large scale industrial projects? Are there any which rely heavily, if not entirely, on flowcharts?

The answer is yes and it comes from rocket science. Probably the largest project I am aware of that used flowcharts is the software for the Russian space shuttle called Buran. Unlike the US space shuttles, the Russian one was unmanned and the software did all the work on the way to space and back. During the whole software development process they used the programming language called DRAKON and the language uses flowchart-like diagrams on all stages. The developers could not use text at all.

The project proved to be very successful: Buran reached space and came back safe and sound. The reports claim that the low level of software mistakes in the project and the speed of development should thank the chosen approach of using diagrams instead of text. A similar approach is still in use for some space related projects as far as I know.

The successful use of flowcharts in lieu of text for such a large scale project supports the idea that the same approach could benefit every developer.

Unfortunately some specific constraints prevent me from using DRAKON in my professional area.

## Available Tools

- Engineering graphics (e.g. Dia, MS Visio)
- UML tools
- Specific application domain tools (e.g. DRAKON)
- Code generators
  
- None support both text and graphics without sacrificing one or the other part
- Therefore why not to try to develop a new one?

My understanding of the software engineering is that sometimes it makes more sense to work with the code in a text editor while sometimes it is graphics which delivers the best performance. So ideally I would like to have a tool which supports both: text and graphics, without sacrificing any of the two ways to look at the code. That ideal tool would also provide a smooth and integrated way to switch between the views of the same code. Is there such a tool on the market? I have not found any so far. There are generic engineering graphics tools like Dia or MS Visio which are very good at what they are designed for. They can help at some stages but it is hard to use them when there are frequent changes. There are tools which support a design stage - e.g. UML oriented tools - but it is hardly ever possible to use them on a lower level. There are code generators but the generated code in many case is not really for reading it. There are domain specific graphics oriented tools - like DRAKON - and I cannot use them for my general purpose projects. They usually sacrifice the text and concentrate on drawing or even do not provide access to the text at all. So if a suitable tool does not exist why not to develop it? Basically such a tool would make it possible to look at any existing project via two views, together with ability to edit any of them with an automatic synchronization between each other. My research is about how to do it for Python and mostly in Python.

## Graphics Primitives

- Code blocks
- Comments
- Imports
- Raise / Assert
- If / elif / else
- Functions
- Returns
- Classes
- Sys.exit()
- For / while / else
- Break / Continue
- Try / except / else / finally
- With
- Decorators
- A file

A start point for the discussion of such a tool could be a set of graphics primitives that could be used to represent an arbitrary python code. Let's talk first of moving from text to graphics. At the beginning we have a file with some python code and at the end we need graphics primitives, appropriately drawn and connected to each other. What language elements should be recognized? How exactly the recognized elements should be drawn on a flowchart-like diagram? The next chapter will discuss all the required graphics primitives.

## Code Blocks

```
c = MyClass( 154 )  
c.member = f( 17 )  
c.doSomething()  
print c
```

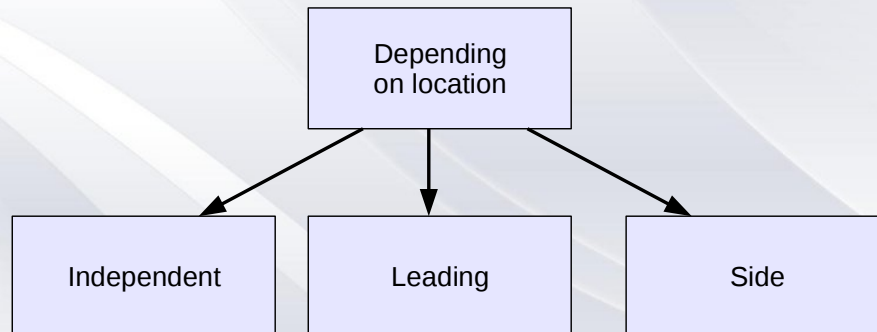
```
c = MyClass( 154 )  
c.member = f( 17 )  
c.doSomething()  
print c
```

```
c = MyClass( 154 )  
c.member = f( 17 )  
  
c.doSomething()  
print c
```

```
c = MyClass( 154 )  
c.member = f( 17 )  
  
c.doSomething()  
print c
```

Certainly not all the language statements affect the control flow directly. The statements which do not affect it could be drawn as code blocks. Why blocks but not just individual statements? Well, developers tend to group statements into chunks which help to understand the code for the future readers. The chunks are separated from each other with blank lines. This fact should be respected on a flowchart diagram. As for the shape, a plain rectangle seems to be a very reasonable graphics primitive for a code block. The examples on the slide show a single code block and a couple of blocks one after another with the only difference in the code of a blank line between the statements.

# Comments



8

Let's talk about the comments.

If we take a closer look it is easy to notice that a few types of comments could be identified basing on how a developer located the comments in the code. Similarly to the code blocks, empty lines should be respected because they define chunks of information. The three comment types are: independent, leading and side comments. The independent comments are those which occupy one or many lines and separated from anything else by at least one empty line. The independent comment lines do not contain anything but comments.

The leading comments are quite similar to the independent ones with one exception. The very next line after a leading comment is a python statement. A developer did not insert an empty line between the comment and the following block and most probably this is meaningful - the comment is for the following block.

The side comments are those which are located to the right of the statements. There are a few important details about side comments. A code block may occupy a few lines and a developer may want to provide a comment for a certain line in the block. This fact should be respected. Another detail is that sometimes a developer may want to provide more than one line of comments for the last statement in a code block. Cases like that should also be respected in graphics.

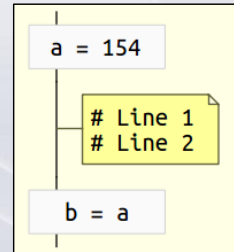
Theoretically it is possible to introduce a trailing type of comments - similarly to the leading type with the difference that there is no empty line between a statement above and the comment. This however seems to have a very minor practical sense. Developers rarely comment something which is located above. They rather use a leading comment or a side one. So I decided not to introduce a trailing type of comments.

So, how could these three distinguished types of comments be drawn on a diagram if they are properly identified?



## Independent Comments

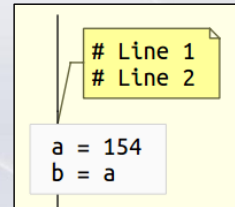
```
a = 154  
# Line 1  
# Line 2  
b = a
```



What we have here is an independent comment comprised of two lines. Indeed the comment is between two code blocks and that position corresponds to a connector between the blocks on the diagram. So a reasonable graphics for independent comments would be a note with a horizontal connector to the appropriate inter block connector.

## Leading Comments

```
# Line 1  
# Line 2  
a = 154  
b = a
```



This leading comment is for a code block and has two lines. So what we can do is to draw the comment above the code block and to change a comment connection directing it not to an inter block connector but to the block itself.

## Side Comments

```
a = 154
b = a      # No comment for the first line
c = b + 1  # Comment for c
           # A tail -----^
```

```
a = 154
b = a
c = b + 1  # No comment for the first line
           # Comment for c
           # A tail -----^
```

11

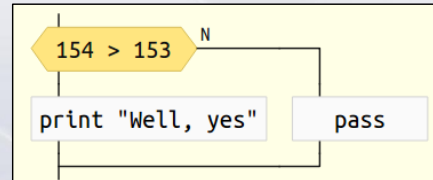
The side comments require to pay attention to a couple of things. The first is that there is usually a line correspondence between the comment and the statement. In this example the code author provided a comment for certain statements. Therefore the graphics representation must keep the line-to-line correspondence between a drawn code block and its side comment.

The second thing to consider is a tail of a side comment. Sometimes a side comment for the last block statement takes more than a single line as is in the example. The last comment line looks like an independent comment because it does not have a statement before the '#' character however the author wanted the comment to be for the last statement in the block. A criteria to consider the side comment continue could be as follows:

- the comment continue line is the very next one and
- the '#' character is at the same position in the line as in the line above

## If

```
if 154 > 153:  
    print "Well, yes"  
else:  
    pass
```



There are a few things to discuss when we talk about the if statements.

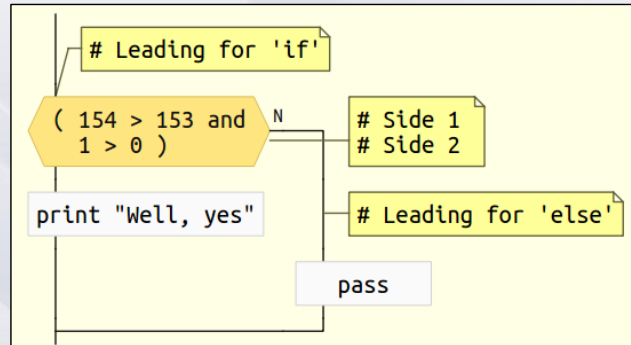
The very first one is the shape of the decision block. A traditionally recommended shape is a diamond. The diamond shape probably works just fine if a condition is very short. In practice, however, a lot of code has complicated and quite often multilined conditions which are hard to squeeze into a reasonably sized diamond. The diamond will either occupy too much space on the screen or the font size will be too small if readable at all. So my suggestion is to use compromise graphics which have the left and right edges resembling a diamond with the top and bottom edges flat to better use the screen pixel estate. That shape can be easily and naturally scaled to accommodate a condition of an arbitrary complexity.

The second thing to discuss is how to draw the 'yes' and 'no' branches. One of the alternatives here would be to draw them as shoulders on the left and right of a decision block. However, this may lead to a diagram which is hard to read and which does not look nice. The problem comes from the fact that the branches may have arbitrary complexity and in graphics it may lead to very wide shoulders. Consequently it may make browsing the diagram inconvenient because both, the vertical and horizontal scrolling would be required.

The other consideration is a thing which I call a main line of execution. Imagine that you are working with a code which you are not familiar with. Your natural wish would be to follow what the program main purpose is and then investigate error handling, special cases etc. It would be nice if on graphics the main line of execution goes always on the most left hand side and preferably in terms of positive logic. To emphasize the main line of execution on graphics I decided to locate the branches as shown on the example, i.e. one branch is located beneath the decision block and the other branch always grows to the right.

## If (continue)

```
# Leading for 'if'
if ( 154 > 153 and      # Side 1
    1 > 0 ):           # Side 2
    print "Well, yes"
# Leading for 'else'
else:
    pass
```



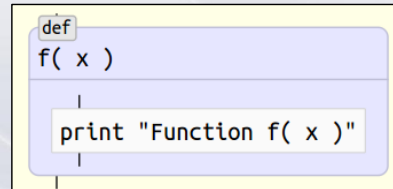
The last detail to discuss about the if statements is comments. Here is an example.

Two things require an attention here. The first one is the condition side comment. The side comment needs to be aligned with the condition lines so it has to be exactly to the right of the decision block. On the other hand there is a connector of the branch at the right which will be crossed by the side comment one way or another while it definitely would be better to avoid any kind of crossing. So a tough decision was made: there will be crossing connectors and the comment connector will be drawn shifted down. A mitigating factor to this decision is that the side comments for the if statements are rare.

The second detail is the 'else' leading comment. The 'else' keyword in the text editor corresponds to a connector in graphics so the 'else' leading comment becomes looking exactly as an independent one.

# Functions

```
def f( x ):  
    print "Function f( x )"
```



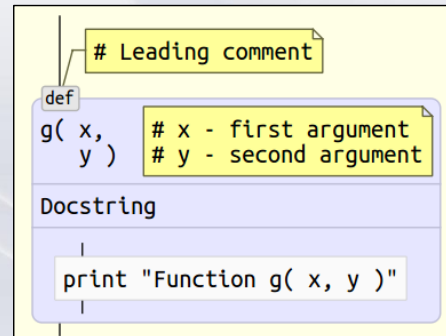
A single python file may contain many function definitions and even nested function definitions. A commonly accepted graphics for the flowchart diagrams however does not offer anything well suitable for the real life python functions. So something new should be suggested.

One of the thoughts which came to my mind was the idea of a scope. The term scope comes up when, for example, accessing variables within functions is discussed. A scope plays a role of borders where a variable is available if it is created in a function. Certainly a function has very definitive points where it starts and where it ends. So the graphics may use a sort of a rectangle - a scope - within which the function body is drawn. Actually sometimes it is a problem for me when I look at a middle of an unfamiliar python code and I do not really understand if it is a function, a class member or some other nesting level - I have to scroll up to understand it. With the idea of a scope for a function it is possible to explicitly show the borders of a function with a distinctive background color so it is obvious immediately what the context is.

A function scope has a header where the function name and arguments are provided. Below a dividing line the function body is drawn. I also decided to add a small badge at the upper left corner of the scope rounded rectangle.

## Functions (continue)

```
# Leading comment
def g( x,      # x - first argument
      y ):    # y - second argument
    " Docstring "
    print "Function g( x, y )"
```



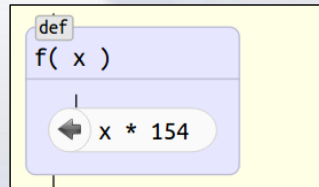
Here we have a function leading comment, comments for multilined list of arguments and a docstring.

The leading comment graphics is more or less obvious while a side comment brings a challenge. The side comment must be aligned with the lines in the function prototype and therefore I decided to draw it within the function header.

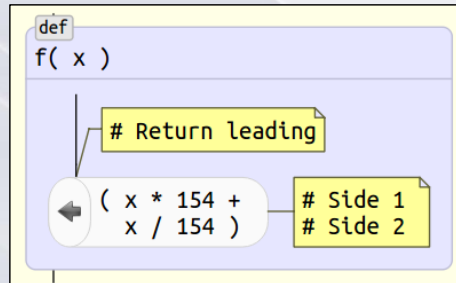
To draw a docstring the scope header is extended with one more horizontal section as shown on the example.

## Returns

```
def f( x ):  
    return x * 154
```



```
def f( x ):  
    # Return leading  
    return ( x * 154 +      # Side 1  
            x / 154 )      # Side 2
```



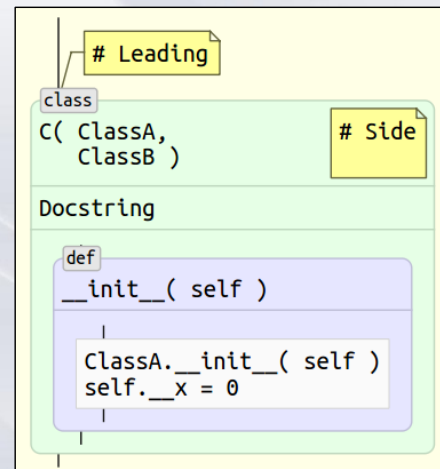
The flowchart diagrams offer a nice graphics for the return statements and this shape could be used. I decided to introduce one improvement to the shape - to add an icon on the left. The icon serves the purpose of catching the returns even at a brief glance at a diagram and I consider this as an important touch. The return statements affect the control flow heavily.

As usual, a return expression could occupy many lines and some comments could be supplied. The second example covers this case.



# Classes

```
# Leading
class C( ClassA,      # Side
        ClassB ):
    """ Docstring """
    def __init__( self ):
        ClassA.__init__( self )
        self.__x = 0
```

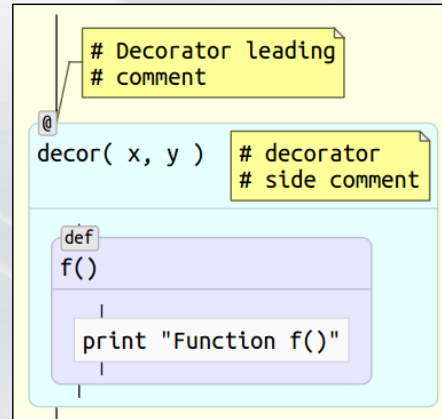


17

Coming from functions to classes, it seems more than logical to use the same idea of scopes for classes. The class graphics layout could be very similar to the function's one with an exception of a background color and a text in the badge.

## Decorators

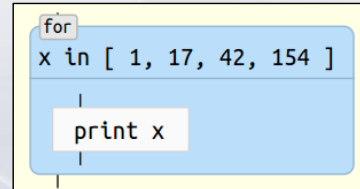
```
# Decorator leading
# comment
@decor( x, y ) # decorator
               # side comment
def f():
    print "Function f()"
```



One more python entity which may appear in the context of python functions and classes is a decorator. Essentially a decorator is an outer function so a scope idea could be used again. Here we have a distinctive background color for the decorator scope, a badge to reflect the nature of the scope and a familiar header with the decorator declaration and its comment.

# Loops

```
for x in [ 1, 17, 42, 154 ]:  
    print x
```



19

Python supports two types of loops: for and while. Both of them have a condition and may have break and continue statements inside. The decision of what graphics to use to draw the loops was a hard one. Here are a few considerations.

A traditional flowchart graphics primitive used for loops is already occupied by the if statements and I rather keep it this way because I don't see any good alternatives for if statements.

On the other hand a loop has a very definitive point where it begins and where it ends so it reminds a scope for the loop body with a loop condition in the scope header. Another consideration is that in case of a scope primitive for loops the idea of the main line of execution is expressed more naturally than in case of a traditional flowchart primitive. This is actually about the connectors. The scope primitive would have an exit point at the bottom while the traditional primitive would have it at the right hand side of a header. Another problem with the traditional graphics comes up when we talk about the 'else' clauses supported in python for loops. There is nothing like that in the traditional flowchart diagrams and it is hard to suggest something for them.

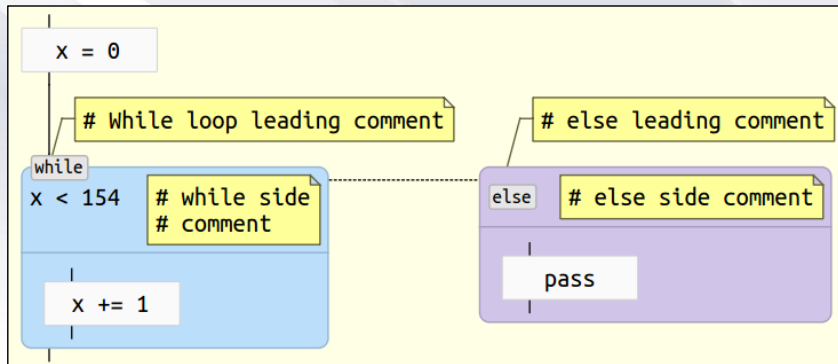
The last consideration is the break and continue statements. If a scope primitive is used then the points to where break and continue should lead become very well visible: at the bottom and at the top of the rounded rectangle. Otherwise explicit connectors would be required and it would be hard to draw them automatically without crossing the other graphics.

Having the scope idea for the loops the question with the leading and side comments could also be resolved easily.

So I decided to stick on the scope primitives for python loops.

## Loops (continue)

```
x = 0
# While loop leading comment
while x < 154:    # while side
                  # comment
    x += 1
# else leading comment
else:            # else side comment
    pass
```

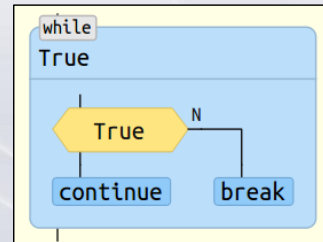


20

Here we have an 'else' part for a 'while' loop. The 'else' part has its own scope and is drawn at the right hand side. Both, the loop and the else part, have a side comment and there is a good place for them. Leading comments also look natural to me. To emphasize the association between the loop and the else part the graphics has a dotted connection line. The 'else' part scope badge has been moved into the header part because there is nothing to draw in the else part header and it seems look better this way.

## Break / Continue

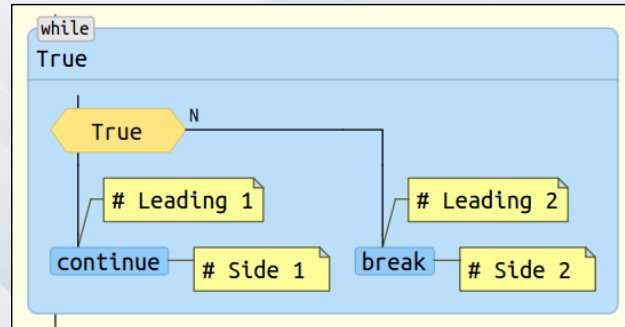
```
while True:
    if True:
        continue
    else:
        break
```



Traditional flowchart diagrams do not offer any graphics for the break and continue statements. These statements correspond to connectors and that introduces a potential problem. Both break and continue statements may have comments tied to them and it would be hard to show comments distinctive enough to highlight that fact. Another problem is that the logic within a loop could be very complicated and there could be many continue and break statements. This may lead to crossing connections which I would prefer to avoid as much as possible. Thus I decided to introduce new graphics for the statements. To highlight that they are essentially jumps to certain points I chose an idea of a label without any connectors. A person who knows python definitely knows anyway where continue and break will jump to.

## Break / Continue (continue)

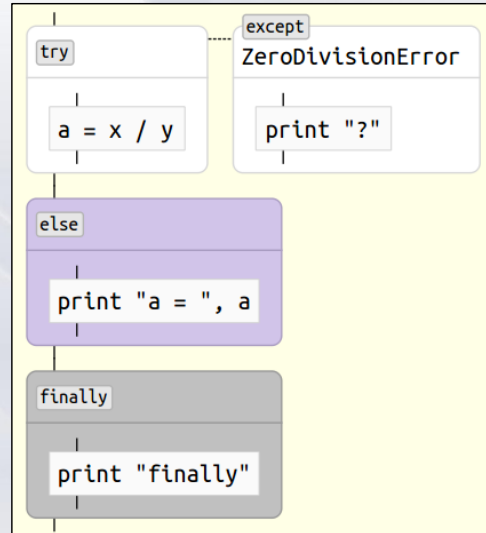
```
while True:
    if True:
        # Leading 1
        continue    # Side 1
    else:
        # Leading 2
        break        # Side 2
```



Here is a more extensive example with comments. As it was mentioned earlier there is good notation of what the comments are linked to.

## Try / except / else / finally

```
try:  
    a = x / y  
except ZeroDivisionError:  
    print "?"  
else:  
    print "a = ", a  
finally:  
    print "finally"
```



23

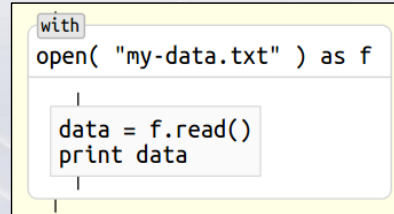
This is probably the most complicated language statement. It may have try, many except, finally and else blocks. As soon as all these parts have their own suits it was decided to use the idea of a scope for each of them.

The except scopes are for error handling which are usually not on the main path of a program execution. That is why they are on the right hand side. The else and finally blocks on the other hand are rather on the main line of execution so they are right under the try block. The except blocks have a dotted connection to the corresponding try block. This is done to emphasize the connection between them. If there are more except blocks, then they will be one after another on the right hand side.

There is not much more to say about the try statement. Obviously each element could have both leading and side comments and if so then the comments will be shown the very same way as for the other statements which use the scope shape graphics.

## With

```
with open( "my-data.txt" ) as f:  
    data = f.read()  
    print data
```

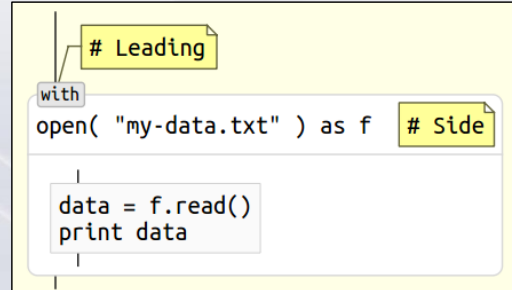


The with statement defines a context in which its suit is executed. Thus the idea of a scope for the with statement looks appropriate.



## With (continue)


```
# Leading  
with open( "my-data.txt" ) as f: # Side  
    data = f.read()  
    print data
```



The second example just adds a leading and a side comments which the with statement may have. There is nothing special here.


## Raise

```
raise Exception( "Oops" )
```



```
Exception( "Oops" )
```

```
# Leading  
raise Exception( "First line " # Side 1  
                "Second line" ) # Side 2
```



```
# Leading
```

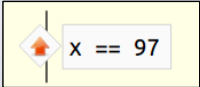
```
Exception( "First line " # Side 1  
          "Second line" ) # Side 2
```

Undoubtedly, an exception generation affects the control flow considerably. So the graphics for it should be identifiable at first glance. The other consideration is that there is something similar between a return and a raise statements. Both lead the control flow out of the current scope. So I decided to use the shape from the return statements and to add a red arrow icon for eye catching purposes.

As usual the raise statement may occupy many lines and may have leading and side comments as shown in the second example.

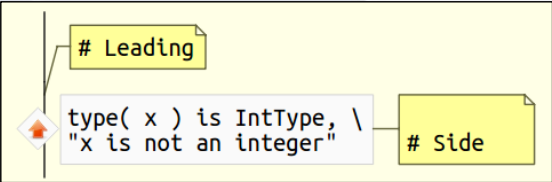
## Assert

```
assert x == 97
```



```
x == 97
```

```
# Leading  
assert type( x ) is IntType, \  
    "x is not an integer"      # Side
```



```
# Leading  
type( x ) is IntType, \  
"x is not an integer" # Side
```

27

Asserts generate exceptions conditionally depending on the provided condition. So they affect the control flow similarly to the raise statements. This is why I wanted to have an easy to catch graphics for them and this is why they have a red arrow icon. In order to highlight the fact that the exception is conditional I put the arrow icon into a diamond shape separate element as shown on the example.

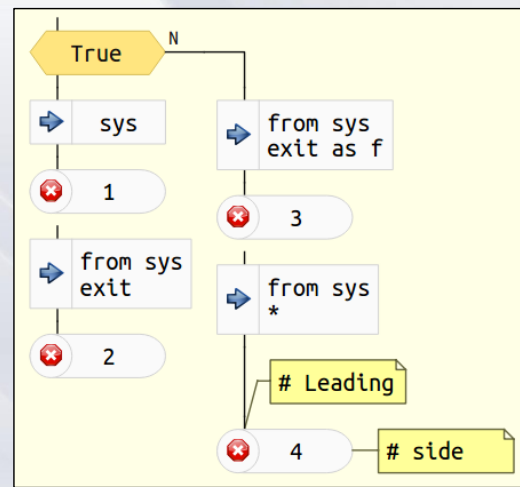
Obviously the comments must not be forgotten. The second example shows how the diagram will look like if an assert has both leading and side comments as well as occupies more than one line.

## sys.exit()

```
if True:
    import sys
    sys.exit( 1 )

    from sys import exit
    exit( 2 )
else:
    from sys import exit as f
    f( 3 )

    from sys import *
    # Leading
    exit( 4 ) # side
```



Strictly speaking the `sys.exit()` call is not a part of the language but a library function. It however affects the control flow not less than exceptions and probably even severer. So I decided that it would be a good idea to recognize the `sys.exit()` calls and highlight them explicitly.

A peculiar thing here is that the `sys.exit()` could look different depending on how the import is done.

Certainly, there is `eval("...")` call as well which is very hard to cover (and this case is not covered) but handling the most common cases is better than nothing.

As a matter of fact `sys.exit()` is internally implemented using exceptions so it makes sense to use similar graphics for `sys.exit()` and exceptions. This is what is done: the exception arrow icon is replaced with a stop sign icon and the graphics shape is shared. The result is the `sys.exit()` calls are easily recognizable.

## File

```
#!/usr/bin/env python
# encoding: utf-8

"""
A file docstring may occupy
a few lines
"""

print "Hello flowcharts"
```

```
Encoding: utf-8
Bang line: /usr/bin/env python

A file docstring may occupy
a few lines
```

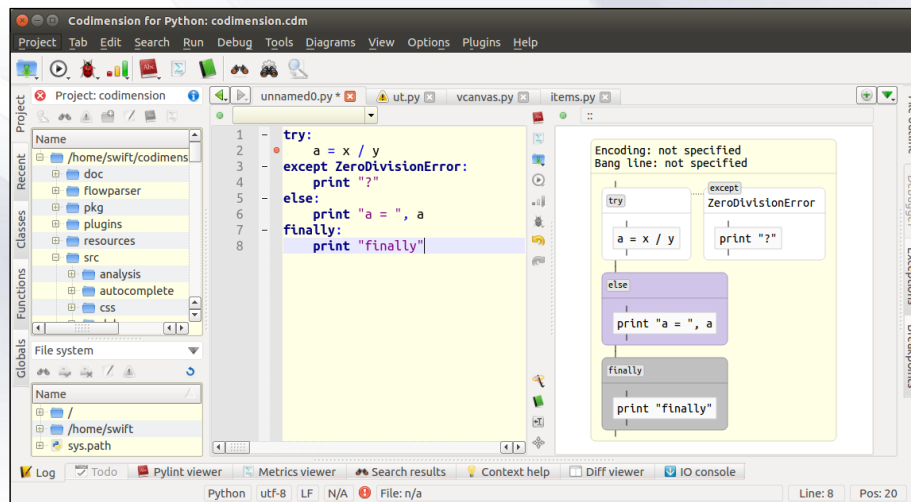
```
|
print "Hello flowcharts"
|
```

The last primitive to reflect the whole module content is for a file. A python file has a few attributes which should be shown. Namely, a file may have:

- a docstring
- an encoding line
- a hash bang line

Naturally, a file forms a scope within which all the other items are located. So a scope primitive could be used as shown here.

# Codimension Python IDE



30

So, having a good idea of what information should be collected from a source code and how it could be drawn, a tool could be implemented. My thoughts on the tool started at the point of how the text and the graphics should collaborate with each other. One of the options was to support graphics only and I rejected this idea. I am sure that depending on the situation a text could be a better representation while in other cases graphics would be. So I wanted a tool which does support both ways and does not sacrifice any of them. In a typical IDE a text editor occupies the main area so now this area is going to be equally divided between the text and the graphics. Before starting a new project, I looked at the available open source IDEs with an intention to develop a plugin for them which supports the graphics. Unfortunately nothing suitable was found. This is why I started a new experimental open source project which I called Codimension. Codimension did not start from an absolute scratch. I used another python IDE called Eric 4 as a source of ideas and some code for Codimension.

At the moment Codimension implements an automatic drawing of flowchart diagram for the python code. A pause in typing code is detected and the diagram is automatically re-drawn on the right hand side. If the code becomes broken at some stage, the diagram is simply not updated and an indicator on the top tells about the current state. I also implemented a feature of showing a navigation-like path for a scope under the mouse cursor. A double click on the diagram is also supported: the focus is passed to the text editor part and the corresponding line is set as the current one. The opposite way of synchronizing the views is supported via a hot key combination. When the user invokes it the IDE detects what graphics primitive corresponds to the current line in the editor and scrolls the graphics view appropriately. The last few implemented graphics features are zooming and exporting to an SVG, a PDF and a PNG. Obviously, I mention here not all the IDE features and more to come!

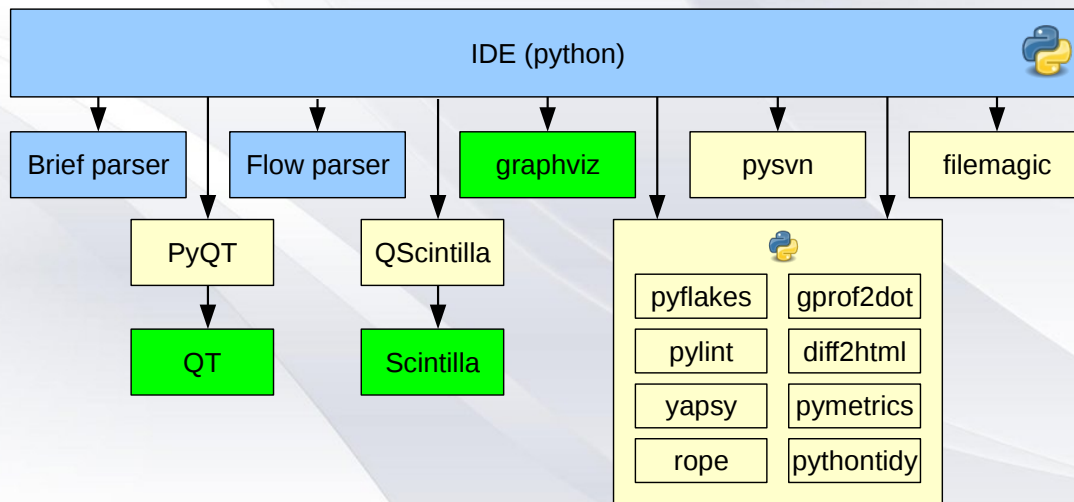
## Codimension Python IDE

- Experimental IDE
- Integration with flowchart generation
- Written in mostly in Python + 2 modules written in C/C++
- PyQt bindings for the UI
- Python 2
- GPL v.3
- Linux (development is on Ubuntu)
- <https://github.com/SergeySatskiy/codimension>

31

Here are some basic facts about the IDE. It is a GPL v3 project hosted on GitHub so anybody can try it or join the project. The project is mostly written in Python and there are two modules written in C/C++. The development is done on Linux, namely I use Ubuntu and it is a Python 2 project. The UI part of the IDE uses the QT library and its python bindings PyQt.

## Codimension IDE: architecture



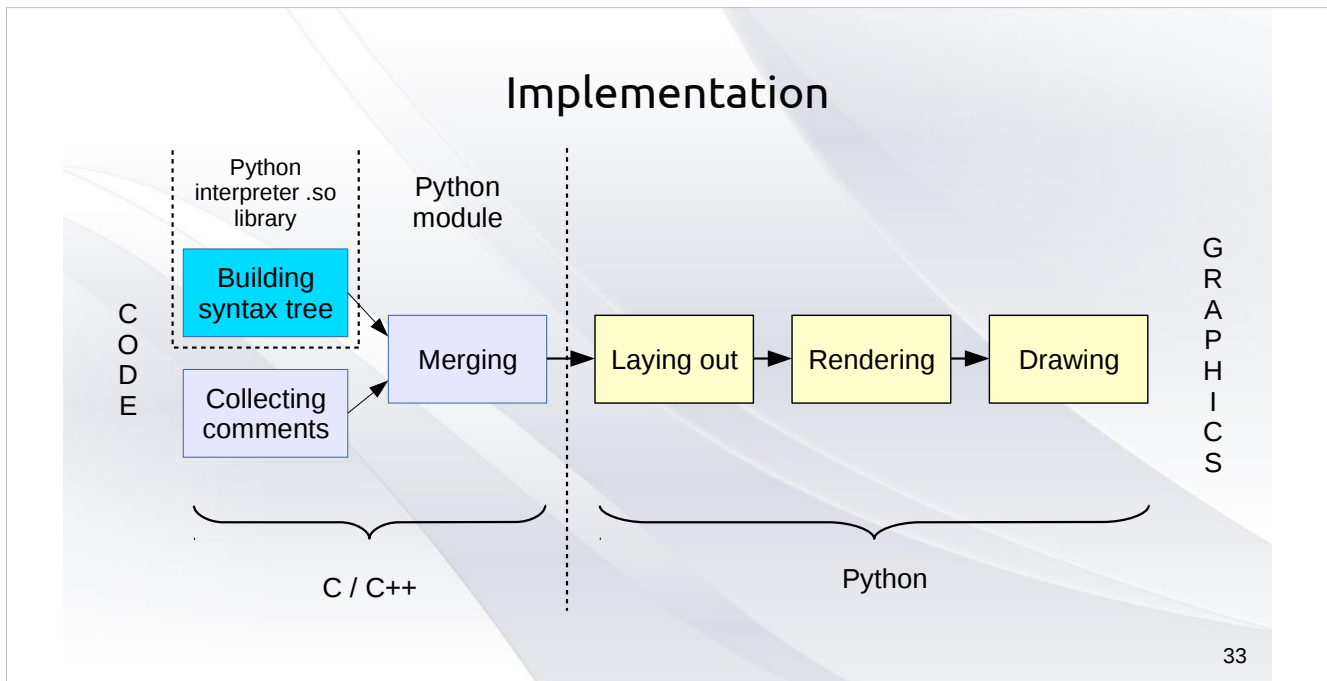
32

Here is the overall Codimension architecture. When I started the project I clearly understood that it is nearly impossible for one developer to develop every single required piece from scratch. So the idea was to use the existing python packages as much as possible. This architecture diagram reflects the decision quite well.

There are only three pieces developed within the Codimension project: the IDE, the brief parser module and the flow parser module. The IDE is written in python and the modules are written in C/C++ for better performance. The brief parser tells what entities are found in the code like imports, classes, functions, docstrings, global variables etc. This lets the IDE to display a structural information about a file and also to implement some analysis features like identifying unused global variables or functions in the project. The flow parser provides the information for drawing the flowchart diagrams.

All the other modules are thirdparty though some of them required some patching. I used PyQt QT library bindings for drawing the UI, Scintilla for the text editor component, graphviz for providing a layout of some diagrams like a dependency graph. Various python written packages were used like pyflakes, pylint, rope, pymetrics, yapsy, etc.





My implementation of the flowchart generator for a given valid python code is a multistage process. On some stages I used C/C++ to squeeze the best performance while on the others I used Python to make it easier to try various ideas without irritating necessity to re-build an application after each minor change.

The pipeline starts with parsing the source code into a syntax tree. Then the syntax tree is analyzed and all the code blocks, functions, classes etc are created as a hierarchical data structure. Then there is another pass over the source code to collect comments. Then the comments and the recognized language elements are merged into a single data structure. These stages are written in C/C++ and provided as a python module. The further stages are written in python. First of all the graphics items are laid out on a virtual canvas, then there is a rendering stage which is followed by drawing and painting stages. At the end we have a graphics flowchart-like diagram. Now, let's discuss all these stages in more details.

## Implementation: syntax tree

```
#!/bin/env python
# encoding: latin-1
```

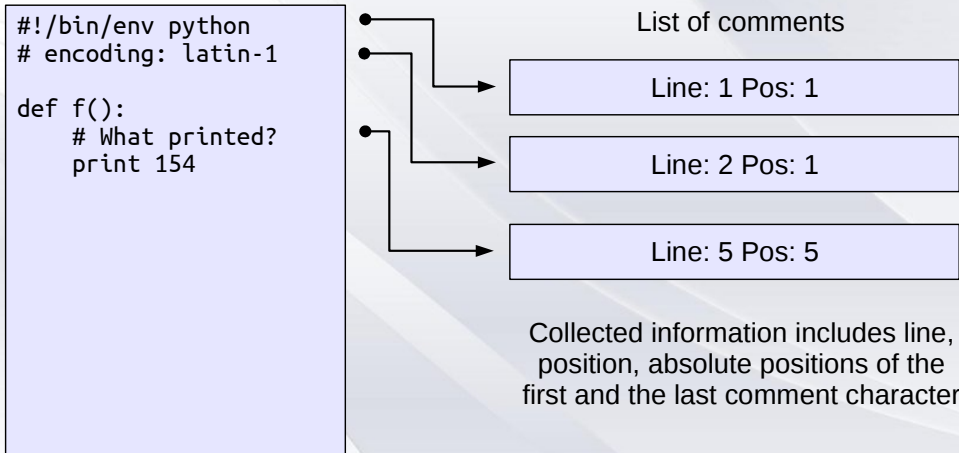
```
def f():
    # What printed?
    print 154
```

```
$ ./tree test.py
Type: encoding_decl line: 0 col: 0 str: iso-8859-1
Type: file_input line: 0 col: 0
Type: stmt line: 4 col: 0
Type: compound_stmt line: 4 col: 0
Type: funcdef line: 4 col: 0
Type: NAME line: 4 col: 0 str: def
Type: NAME line: 4 col: 4 str: f
Type: parameters line: 4 col: 5
Type: LPAR line: 4 col: 5 str: (
Type: RPAR line: 4 col: 6 str: )
Type: COLON line: 4 col: 7 str: :
Type: suite line: 4 col: 8
Type: NEWLINE line: 4 col: 8 str:
Type: INDENT line: 6 col: -1 str:
Type: stmt line: 6 col: 4
. . .
```

34

The first stage on the way from text to graphics is to parse the python code and to collect everything what is there in a form of data structures which could be used for generating graphics. I definitely wanted to use something which has already been developed. It turned out that there is a function in the python interpreter dynamic library which can help. It's a C function which provides a syntax tree. While working on this part I wrote a simple utility which prints the data structure produced by the python interpreter function. Here is an example of a very simple python file and the produced data structure. Generally it looks nice: there are line numbers and column numbers, the node types correspond to the formal python grammar specification. However there are some problems too. The sample code has a few comments - the syntax tree lost them. The encoding line number and column number are wrong. Even the encoding name is wrong: the file says it is latin-1 but the syntax tree reports iso-8859-1. It turned out that the python interpreter code has a normalization procedure for the encoding spec. There are some problems with multiline string literals as well - the line numbers are not supplied. All these surprises had to be considered in the parser module. On the other hand all the text parsing complexity is gone, all I had to do is to walk the syntax tree and build data structures convenient for generating graphics.

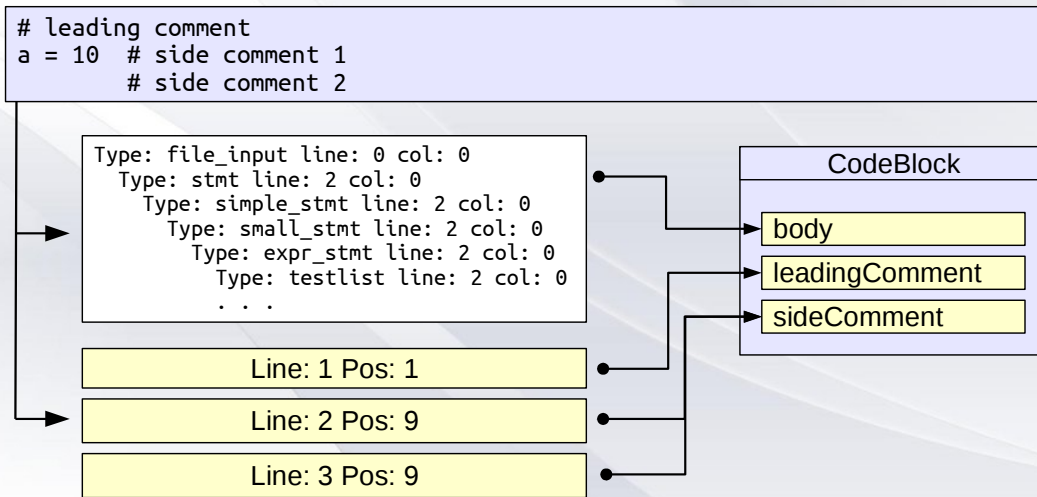
## Implementation: comments



35

As I already mentioned the parser from the python interpreter shared library strips the comments. Certainly, the interpreter does not need them however it is critical not to lose them when a flowchart diagram is generated. Therefore a second pass over the input buffer was required to collect comments. Luckily python has a very simple grammar: there are no multi line comments and there is no pre-processor. The only thing which needs an attention is whether the `#` character is within a string literal or not. The comments are collected as a list of fragments. Each fragment describes a single line and holds the line number, the start and the end positions in the line as well as their absolute positions.

## Implementation: merging



36

Having all the comments separately from all the other flowchart items is not very convenient for the further stages, because the leading and side comments are tightly related to the corresponding items. So there is the stage of merging the comments with the other recognized items. For example, a CodeBlock structure has fields for leading and side comments which have the None value after a syntax tree analysis but at the comments merging stage they get filled as in this example. The merging is done basing on the line numbers which are coming from the syntax tree and from the extracting comments pass.

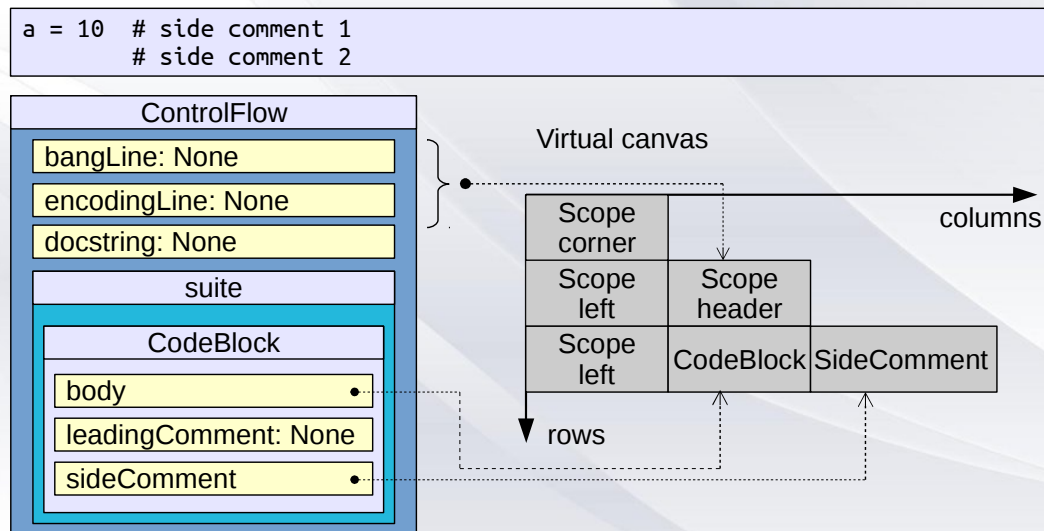
## Implementation: module performance

- Hardware: Intel Core i5-3210M laptop
- OS: Ubuntu 14.04 LTS
- Python 2.7.6 installation: 5709 files
- Time to parse: 6 seconds (~1ms per file)

37

The stages we have just discussed are implemented in a Python 2 module written in C/C++. Generally speaking it was quite important to make the parsing very quick because it opens a possibility to draw a flowchart nearly as the user types the code – via catching pauses – and not introducing irritating hang ups. I believe that the goal was reached successfully. Here are some numbers for my not such a new laptop. Certainly the python installation files are of different size but nevertheless the average time of around 1 millisecond for a file is more than acceptable.

## Implementation: laying out



38

The input for the laying out stage is a hierarchical data structure which is essentially a lossless representation of the source code. The purpose of this stage is to lay out all the items from the input as graphics elements on a canvas for further drawing. This is done with a help of a data structure which I called a virtual canvas. A virtual canvas is a two dimensional array which starts as an empty one and which can grow right and down as needed. Each cell in a virtual canvas contains an instance of a class which in most of the cases represents a graphics primitive we have discussed earlier or another virtual canvas. Nested virtual canvases are used for example for all the elements that use the scope idea.

Lets see how it works for the example above. The virtual canvas is created as an empty one and we start with the input. The most outer container of the input is a file so a rounded rectangle should be drawn. The virtual canvas is extended with one row and one column in it. The cell is set to the upper left corner of the file scope rectangle. This cell is enough to draw the whole rounded rectangle later on so there is no need to allocate any cells for the top of the scope rectangle. The next item is the file header. Therefore the canvas is extended with one more row. The first allocated column in the row is for the left vertical edge of the file scope rectangle. The second column is set to the header of the file scope i.e. a text for the encoding and the hash bang line. We do not have a file docstring in the example so there is basically nothing else to be done for the file scope header.

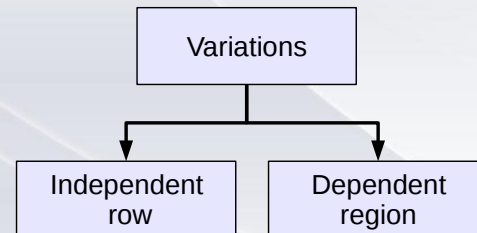
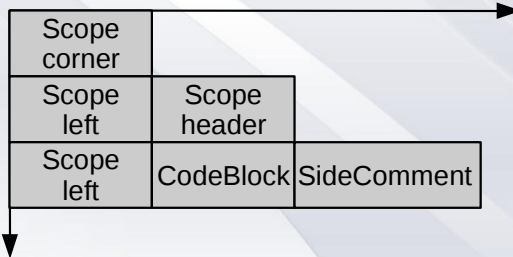
The next item in the input is a file scope suite. The first element in the suite container is a code block. So the canvas is extended with one more row. The first column in the row is set as the file scope left edge. The next column is allocated for the code block. The code block in the example has a side comment so another cell is allocated in the row and its content is set as the corresponding side comment.

There is no need to allocate cells neither for the bottom nor for the right edges of the file scope because the scope rectangle will be drawn when the top left corner is found.

## Implementation: rendering

```
a = 10 # side comment 1  
      # side comment 2
```

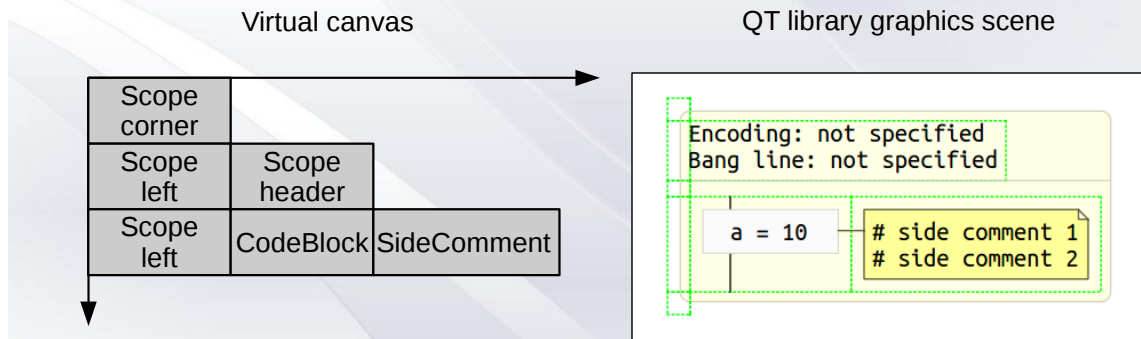
Virtual canvas



39

The rendering stage is about the sizes of the graphics primitives. Basically what this stage does is it visits every allocated cell and sets the sizes appropriately. Naturally each cell in a virtual canvas has two sets of sizes: minimum required width and height and the actual - calculated during rendering process - width and height. Let's talk about the cell heights first. The analysis is done on per row basis. In the code example the assignment statement takes just one line while the height of its side comment allocated in the same row requires two lines i.e. more pixels vertically. So the minimum required sizes differ for the items in the same row. The rendering algorithm will walk all the items in the row and pick the maximum height required. That value will be assigned as the actual height of each cell in the row. By the way, all the calculations are done basing on the font which was selected for drawing the diagram and other settings like a text or a cell padding. Slightly more complicated story is with the cell width. From this prospective there are two types of rows: those which cell widths affect the other row cell widths and those which do not. A good example of a dependent row region is an if-else portion of a diagram. A below-branch of an if statement could of an arbitrary complexity so the width of the suite could vary while on the other hand a connector for the right-branch needs to be drawn properly i.e. considering the below-branch suite width. Therefore for an independent row rendering is primitive: the actual width of a cell is assigned as the minimum required. For the dependent row regions the rendering procedure needs to walk all the rows for each column and calculate the actual required width as a maximum of the minimum required ones - quite similar to what is done for a row in terms of its height. Then the calculated width is assigned to all the cells in the column. Obviously the calculations are done recursively for all the virtual canvases. The outcome of the rendering is that the sizes of all the graphics primitives are calculated and the scene is ready for drawing.

## Implementation: drawing



40

The drawing stage is very simple. Its task is to add graphics items to the graphics scene and position them appropriately.

So the visitor starts from graphics coordinates 0, 0 and visits each cell of the virtual canvas. For each cell it adds a graphics primitive to the scene if necessary and updates the base point for the next cell. The actual painting of the graphics scene is done by the QT library.

This is it about the implementation.



## Present & Future

### Done:

- auto update of the flowchart
- manual sync to graphics and to text
- zoom
- export to PDF, PNG and SVG
- scope navigation bar

41

The next thing I would like to discuss is the features which have been implemented and which can be implemented in the future for the flowchart representation of the code.

There is not much what has been done. This is most importantly an auto update of a flowchart when the text is changed and a pause in typing is detected.

Manual synchronization in both directions is implemented as well. If you double click on a graphics item then the appropriate line will become the current one in the text editor. If you are in a text editor and a hot key is pressed then the appropriate graphics item is centered on the graphics pane.

Flowcharts zooming is implemented too.

Flowcharts could be exported in a few formats like PDF, PNG and SVG.

One more implemented feature is a scope navigation bar. When a mouse cursor is hovering a scope, a path to it in terms of scopes is shown in a navigation bar at the top of the view.

## Present & Future

Future – non intrusive:

- auto sync with text cursor
- deletion / inserts / editing on the graphics pane
- debugging support on the graphics pane
- search support on the graphics pane
- printing of the graphics pane
- collapse / expand comments / docstrings / scope suits
- import type highlight (project / system / unknown)

42

The features which are planned for the future could be divided into two groups: non-intrusive and intrusive. Non intrusive in this context means that the source code does not need any additional markups. These features may include the following.

Auto synchronization of the views. Say, when a text cursor changes its position the graphics view could be scrolled automatically to center the appropriate graphics item.

All the editing operations could be supported on the graphics view so that when something is changed on the diagram the text view is updated correspondingly.

Debugging could be supported on the graphics pane via highlighting the current graphics item and a certain line within it.

There could be controls which allow to show or to hide comments and docstrings.

And when they are hidden then the actual content could be shown in a tooltip.

One more feature which could be implemented is to detect a type of an import – from the current project, system wide or unknown – and display them in a slightly different way. An icon could be changed or a background color could be changed.

## Present & Future

### Intrusive:

- change individual colors (fg / bg)
- grouping / ungrouping
- replacing code with a text
- insert special items like links to pictures or web resources

43

Intrusive features, on the contrary, will require some changes in the source code but those changes must not affect how the code is executed. For example, suppose the code author would like to draw the reader's attention to a dangerous piece of code via changing the code block background to red. Certainly, such a block on a flowchart diagram will unavoidably catch an eye. Suppose this is done on the graphics pane. Then this information should be stored somewhere to be shown properly when the file is opened two months later. In my opinion this additional information is valuable and it deserves to be injected directly to the source code.

The same story is about another important feature which could also be implemented. Imagine that you are reading a code which was written by somebody else. Gradually you understand the certain pieces so that a group of statements makes a certain sense to you. So you could group the appropriate graphics items and replace that with a new graphics item with an appropriate text on it. For instance, a rectangle with doubled edges will appear with a text 'calculation of a checksum'. So, the start and the end point, as well as the replacement text, should be memorized somewhere. Another similar feature is to replace a piece of code on the diagram with a text. For example there could be a very long and complicated condition in an if statement which could be replaced with a description of it in plain English. Lastly some special items could be added to the diagram like links to the outside resources.

## CML v.1

- Codimension Markup Language
- Uses python comments
- Easy integration with existing VCSs
- No problems with any existing tools
- Already supported by the cdm-flowparser

44

As it was mentioned above, there is a need of storing some additional information about python files. One of the options would be to support a separate storage on per project or per python file basis – another file which stores the markup. I do not like this idea because of a few reasons:

- imagine that there is a team of developers and some of them prefer to use the approach of switching between graphics and text. However another half of the team despise graphics and principally use text only. So the line information in the separately stored markup could easily become broken and it is generally hard to support the consistency using this approach
- another file or files would require to keep track of them in a VCS system which introduces another effort for the team
- lastly, the markup information was introduced by the developers who use the graphics not for fun but because it has a value. Thus it would be nice if this additional value is also available in some way for those who accept nothing but vim.

Thus, a suggested solution is to introduce a micro markup language which uses the python comments. I called this language CML – Codimension Markup Language, currently of version 1. The markup will be stored right in the python files at appropriate points of the code so all the mentioned above problems will be addressed.

## CML v.1

- Format of the first line  
# cml <version> <type> [key=value pairs]
- Format of the further cml lines  
# cml+ <continue of the previous cml line>
- version: integer version, e.g. 1
- type: string identifier, e.g. CC
- key: identifier
- value: string value, in double quotes if needed

45

A CML comment may appear anywhere in a python file and may occupy one or many lines. The 'cml' and 'cml+' literals are what distinguish the CML comments from the others. The version is introduced for the future if CML is evolving.

The record type defines what will be done on the graphics pane. For example the record type CC can stand for Custom Color and expect parameters which describe the custom background and foreground colors.

Sometimes the only one record of a CML comment is required as in the CC case.

The CC CML comment appears as a leading comment and affects the item below. In some other cases a couple of records is required. For example, if it is grouping of the graphics items then the beginning and the end need to be marked. So two records will appear at the appropriate places in the source code.

The last thing to say here is that the chosen format is so simple and human readable that even emacs users will be able to read the CML comments essential content.

## CML v.1 use case: custom background

```
c = MyClass( 154 )  
c.member = f( 17 )  
c.doSomething()  
print c
```

```
c = MyClass( 154 )  
c.member = f( 17 )  
c.doSomething()  
print c
```

```
# cml 1 C bg="255,138,128"  
c = MyClass( 154 )  
c.member = f( 17 )  
c.doSomething()  
print c
```

```
c = MyClass( 154 )  
c.member = f( 17 )  
c.doSomething()  
print c
```

46

Let's take a simple use case for CML. Suppose that we would like to implement a feature of a custom background color for a code block. The sequence of events could be as follows:

- the user clicks right mouse button on a code block and a context menu pops up
- the user selects a custom background color for the block from the menu
- the selection triggers a CML comment insertion into the source code
- when the flowchart is drawn next time the code responsible for the box background color will first check if a custom color is specified in the CML leading comments for the block. If so then the required color will be used

So the information about the change is preserved and will be saved in the used version control system without any problems. Those who use traditional text editors or IDE which do not support flowcharts and CML will suffer a bit from this addition but I hope that the burden will not be so big for them.

## Acknowledgments

- Detlev Offenbach – the author of the Eric 4 Python IDE
- Authors of the other packages Codimension uses
- Dmitry Kazimirov – logo and initial Codimension packing
- Ilya Loginov, Sergey Fukanchik – packaging
- David McElhany – discussions, ideas, testing etc.

47

I would like to mention a few people who played a certain role in both developing the idea of two way code representation and the implementation of the tool which proves the feasibility of the approach.

First of all it is Detlev Offenbach, the author of the Eric 4 Python IDE. That was a great jump start for Codimension. Some code and some ideas were borrowed from Eric 4.

Obviously there are numerous packages which Codimension uses and I would like to thank their respective authors.

My colleague at NCBI Dima Kazimirov helped me with a Codimension logo and with initial packaging.

Ilya Loginov helped with packaging for a certain period of time. Now Sergey Fukanchik covers packaging.

And finally I would like to thank David McElhany – one more my colleague at NCBI – for the valuable discussions, sharing ideas some of which are implemented, testing Codimension, etc.

## Resources

- **Codimension IDE:**  
<http://codimension.org>
- **Buran:**  
[https://en.wikipedia.org/wiki/Buran\\_%28spacecraft%29](https://en.wikipedia.org/wiki/Buran_%28spacecraft%29)
- **DRAKON:**  
<https://en.wikipedia.org/wiki/DRAKON>
- **Visual Programming Languages:**  
<http://blog.interfacevision.com/design/design-visual-programming-languages-snapshots/>

These are a few links to the resources which were mentioned in the presentation.

Any questions?