# Titus

Developed by @danii_mor

## DEVELOPER MANUAL

### *LEXICAL ANALYSIS*

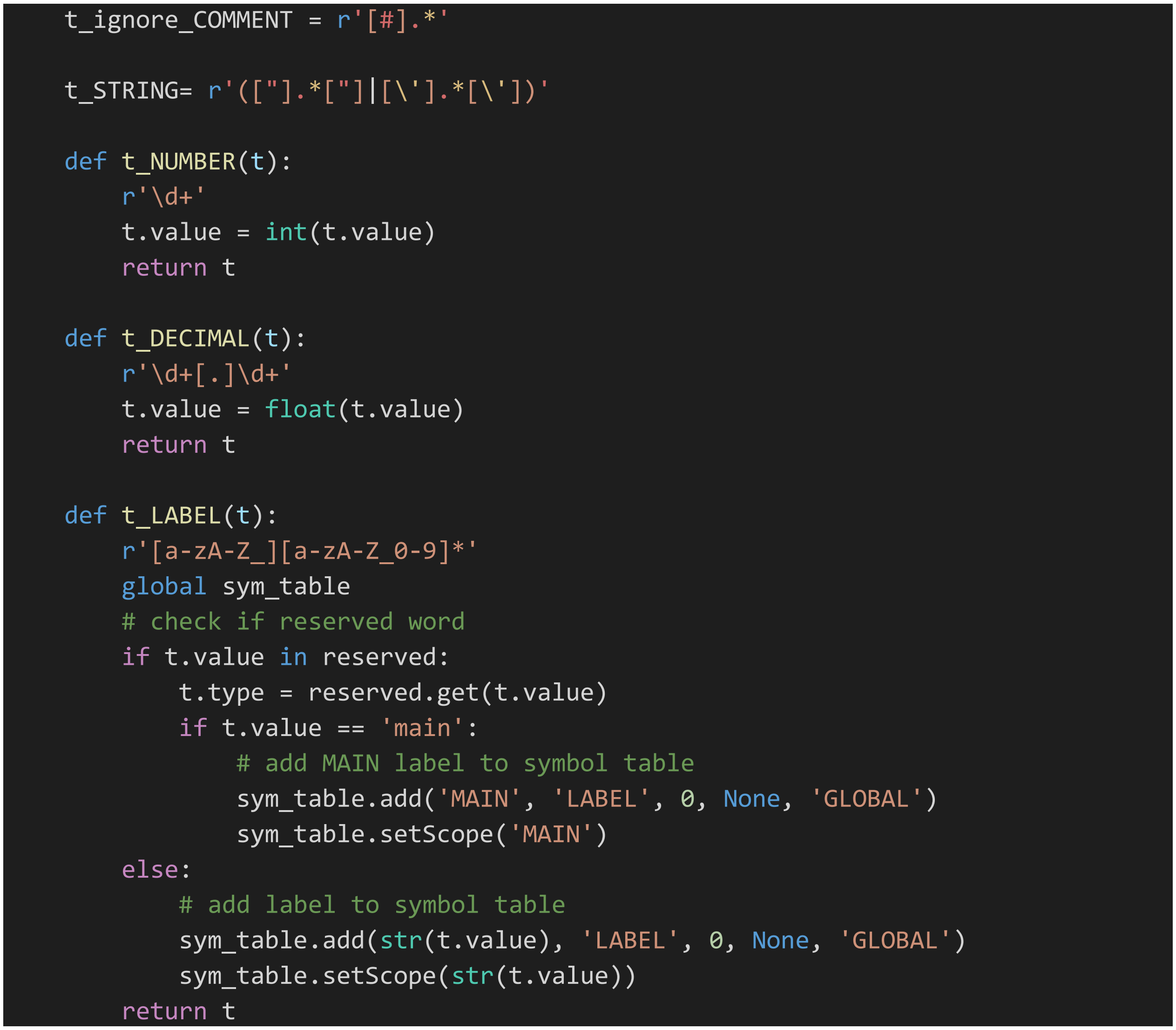
First of all we start by creating our keywords. In them you can detail the tokens that you could not afford to use in any other way. They usually identify native language functions.

```
# tokenizing rules
    reserved = {
                'main'   : 'MAIN',
                'if'     : 'IF',
                'print'  : 'PRINT',
                'abs'    : 'ABS',
                'goto'   : 'GOTO',
                'xor'    : 'XOR',
                'read'   : 'READ',
                'unset'  : 'UNSET',
                'array'  : 'ARRAY',
                'int'    : 'INT',
                'float'  : 'FLOAT',
                'char'   : 'CHAR',
                'exit'   : 'EXIT'
                }
```

To later declare the values with which our application will be interacting.

```
tokens = [
                'VAR',
                'NUMBER',
                'DECIMAL',
                'STRING',
                'LABEL'
            ] + list(reserved.values())

    literals = ['=', '\'', '"', '+', '-', '*', '/', '%', '&', '|', '^', '<', '>', '!', '~', '(', ')', '[', ']', ';', ':']

    t_ignore = " \t"
```

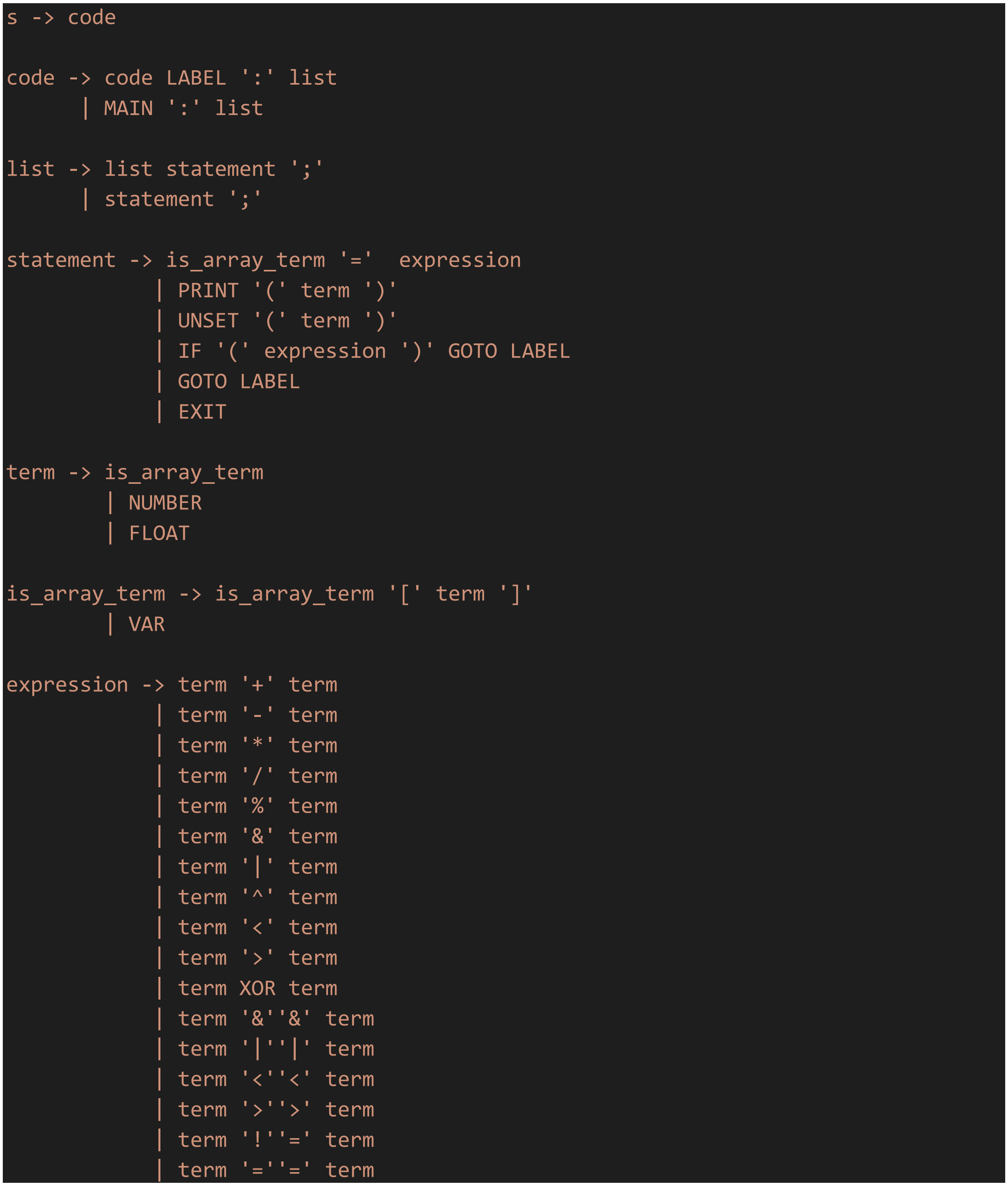
For this we need to indicate what patterns this data will follow. Therefore each token needs a function that defines the pattern to follow and some action that is required to be done just when it is found.



```
    t_ignore_COMMENT = r'[#].*'

    t_STRING= r'(["].*["]|[\'].*[\'])'

    def t_NUMBER(t):
        r'\d+'
        t.value = int(t.value)
        return t

    def t_DECIMAL(t):
        r'\d+[.]\d+'
        t.value = float(t.value)
        return t

    def t_LABEL(t):
        r'[a-zA-Z_][a-zA-Z_0-9]*'
        global sym_table
        # check if reserved word
        if t.value in reserved:
            t.type = reserved.get(t.value)
            if t.value == 'main':
                # add MAIN label to symbol table
                sym_table.add('MAIN', 'LABEL', 0, None, 'GLOBAL')
                sym_table.setScope('MAIN')
        else:
            # add label to symbol table
            sym_table.add(str(t.value), 'LABEL', 0, None, 'GLOBAL')
            sym_table.setScope(str(t.value))
        return t
```

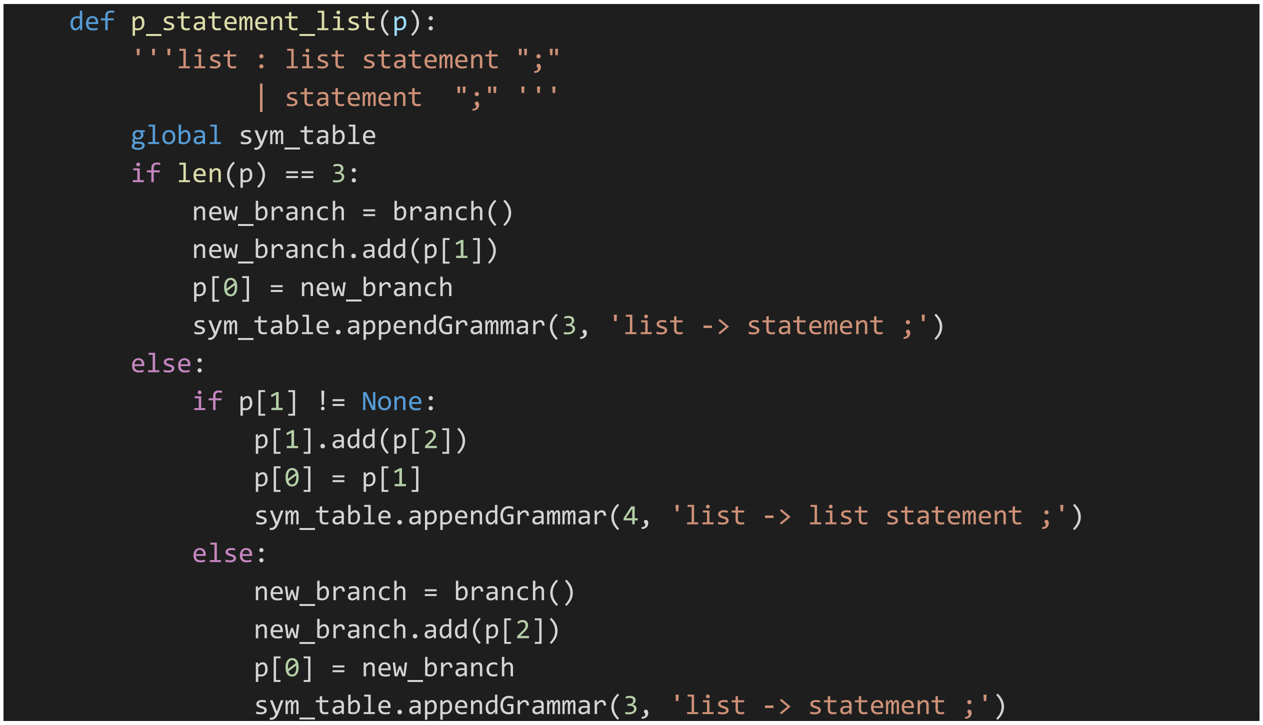
***SYNTACTIC ANALYSIS***

In this region we must indicate the patterns that our list of tokens must follow, already generated by our previous analysis.



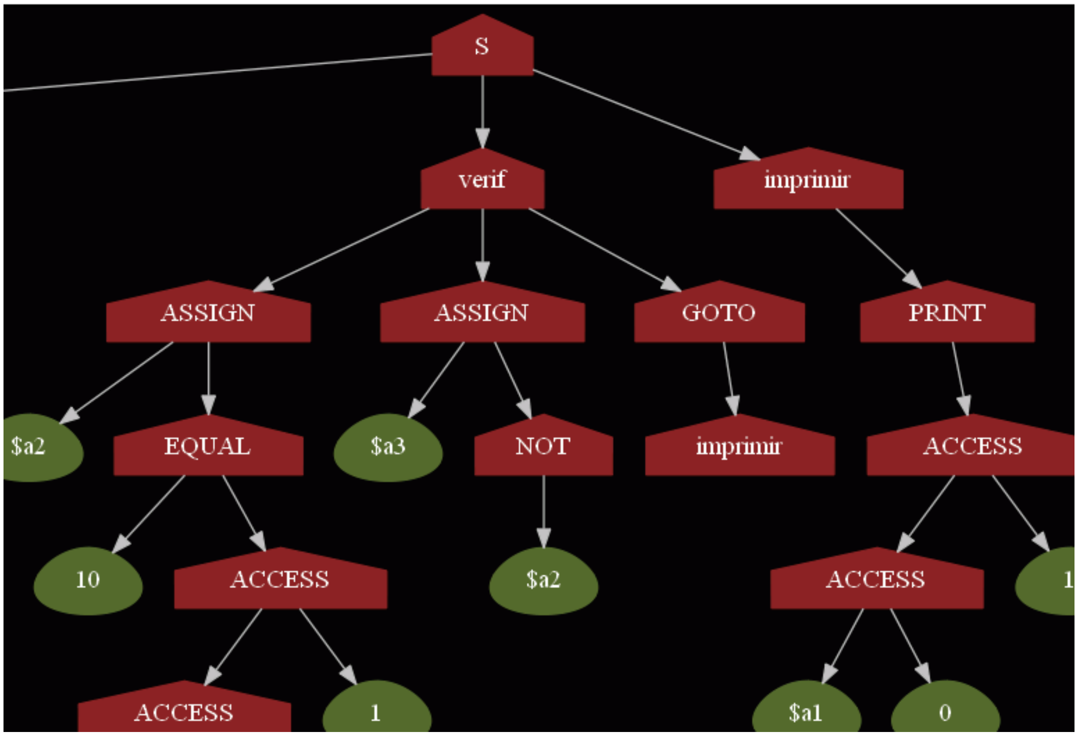
```
s -> code

code -> code LABEL ':' list
      | MAIN ':' list

list -> list statement ';'
      | statement ';'

statement -> is_array_term '='  expression
            | PRINT '(' term ')'
            | UNSET '(' term ')'
            | IF '(' expression ')' GOTO LABEL
            | GOTO LABEL
            | EXIT

term -> is_array_term
        | NUMBER
        | FLOAT

is_array_term -> is_array_term '[' term ']'
        | VAR

expression -> term '+' term
            | term '-' term
            | term '*' term
            | term '/' term
            | term '%' term
            | term '&' term
            | term '|' term
            | term '^' term
            | term '<' term
            | term '>' term
            | term XOR term
            | term '&''&' term
            | term '|''|' term
            | term '<''<' term
            | term '>''>' term
            | term '!''=' term
            | term '=''=' term
```

it can be seen how a syntactic analysis tree is being generated among the grammatical productions. This will then help us to generate the orderly execution of our code.



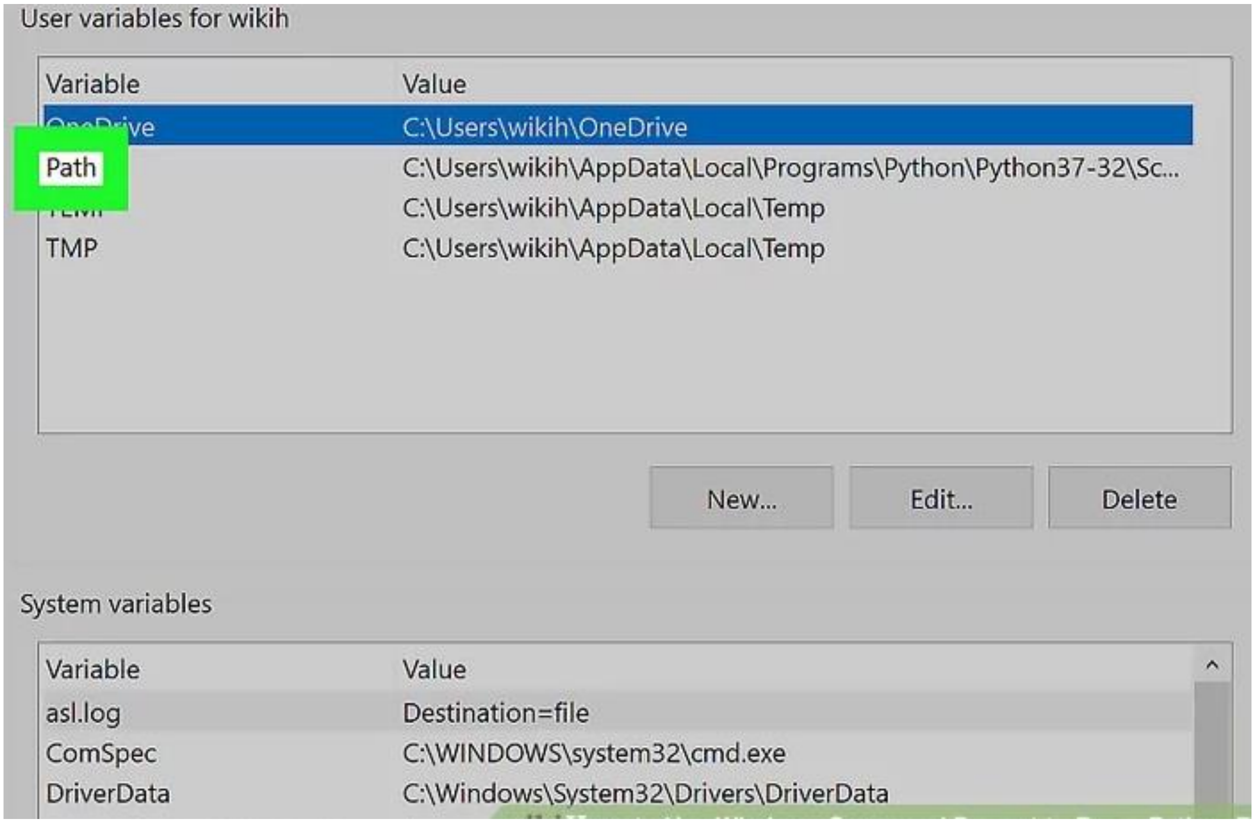
```
    def p_statement_list(p):
        '''list : list statement ";"
                | statement  ";" '''
        global sym_table
        if len(p) == 3:
            new_branch = branch()
            new_branch.add(p[1])
            p[0] = new_branch
            sym_table.appendGrammar(3, 'list -> statement ;')
        else:
            if p[1] != None:
                p[1].add(p[2])
                p[0] = p[1]
                sym_table.appendGrammar(4, 'list -> list statement ;')
            else:
                new_branch = branch()
                new_branch.add(p[2])
                p[0] = new_branch
                sym_table.appendGrammar(3, 'list -> statement ;')
```

Here you can clearly see how our tree is being generated as we go forward in our productions.



*USAGE*

We need Python3.8 + and added our installation of Graphviz / bin to our windows PATH. For linux we only need to update our Python + to the latest version and finally verify that the DOT compilation is accessed from BASH.

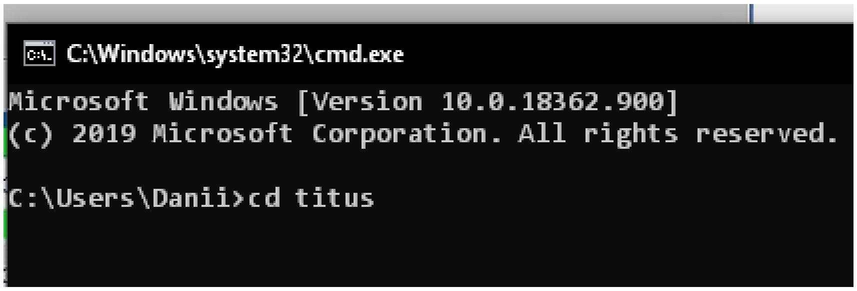


User variables for wikih

| Variable | Value |
|---|---|
| OneDrive | C:\Users\wikih\OneDrive |
| Path | C:\Users\wikih\AppData\Local\Programs\Python\Python37-32\Sc... |
| TEMP | C:\Users\wikih\AppData\Local\Temp |
| TMP | C:\Users\wikih\AppData\Local\Temp |

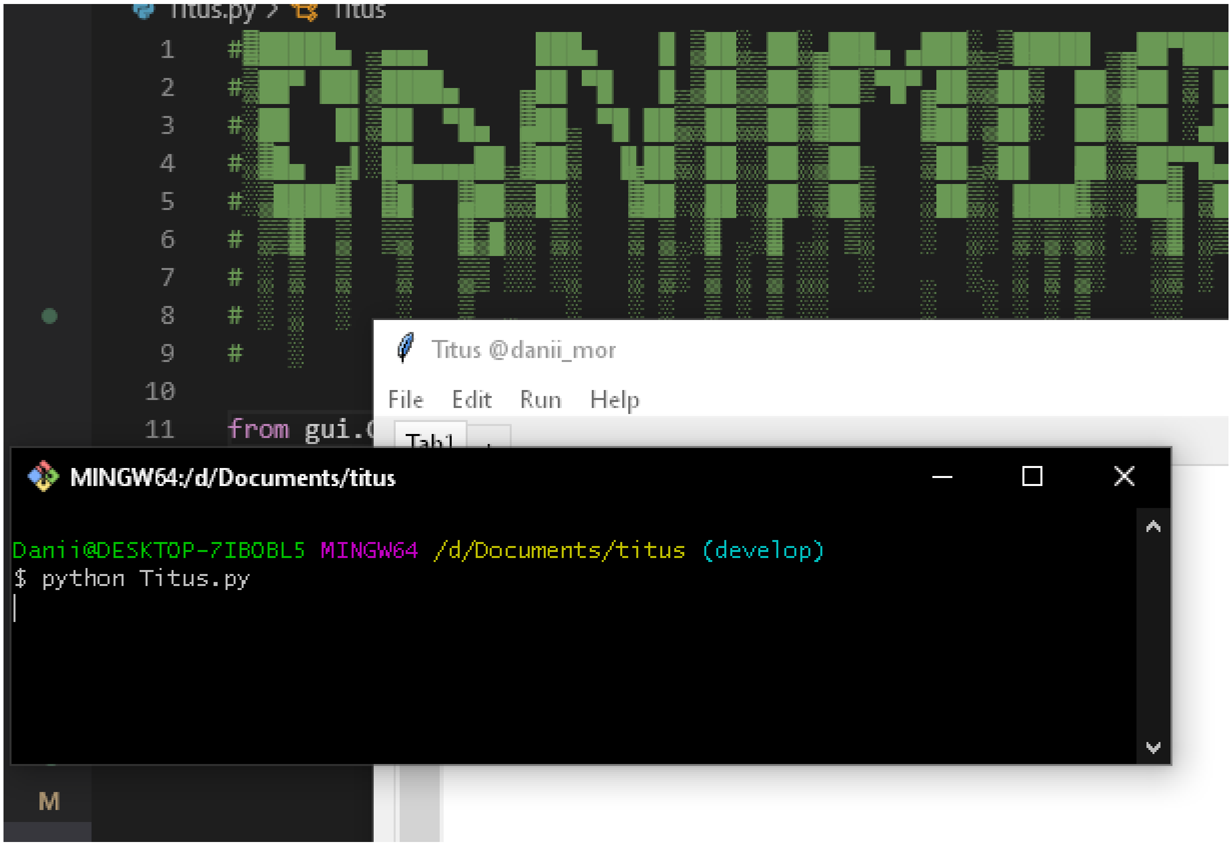
System variables

| Variable | Value |
|---|---|
| asl.log | Destination=file |
| ComSpec | C:\WINDOWS\system32\cmd.exe |
| DriverData | C:\Windows\System32\Drivers\DriverData |



```
Microsoft Windows [Version 10.0.18362.900]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Danii>cd titus
```

***EXECUTE***



```
from gui.(
```

```
Danii@DESKTOP-7IBOBL5 MINGW64 /d/Documents/titus (develop)
$ python Titus.py
```