Scripting Laboratory 2

reading code and generating PDF

Pol Fernàndez Guerra [Pol420 on Git]
Joel Ramos Cano [Revskell on Git]

Operative Systems
Group P
28/02/2020

Index

1. Introduction

We have been task to deliver a Python 3 program that asks the user to input the name of folder with a series of python scripts to read in it and then generate a PDF file with some information about the files that have been read.

We will be using four libraries to help us deal with the task, these are pyPDF, Redbaron, NetworkX and Matplotlib. Each one of them will help us with a specific task in the program:

- **pyPDF:** PyPDF is a library that allows you to generate a PDF and write strings or images into it through code. This will let us create the PDF that we need to generate in order to complete the task.

- **Redbaron:** By opening python code with redbaron, we are able to use functions from the redbaron library in order to read and extract code from python. Redbaron is a library which allows you to search for specific lines of code in python. By using the function .search_all(redbaron_node), redbaron will return us a node with code following the specified redbaron_node in the function call. Redbaron nodes are a way to ask redbaron for specific insances, calls or statements of python code. These can all be found in the Redbaron documentation nodes page: https://redbaron.readthedocs.io/en/latest/nodes_reference.html#nodes

- **NetworkX:** In order to generate an image that renders the structure of a directory with python code, we have decided to use the library NetworkX. NetworkX is a library which lets you create graphs with nodes and edges from code and by also using the library **Matplotlib** we can draw these graphs and save them as a .png image in a directory.

2. CodeReader.py

This script manages everything related to reading python code. In it you will find a series of defined functions which will be used in the other script PDFGenerator.py to write things on the PDF as well as a defined object called variable which will let us store information about variables so that it can be more easily read and sorted when used with PDFGenerator.py.

Let's first start with the defined object variable:

```python
class variable:
    reassignment = []  # stores the line of every variable reassignment
    use = []  # stores the line of every used variable
    with_functions = []  # stores the line of every variable used within functions
    operators = []  # stores the line of every variable used with operators

    def __init__(self, name: str, line: int):
        self.name = name  # stores the name of the variable
        self.line = line  # stores the line of the first use in the variable
        self.reassignment = []
        self.use = []
        self.with_functions = []
        self.operators = []

    def add_reassignment(self, var: str):
        if var not in self.reassignment:
            self.reassignment.append(var)

    def add_use(self, var: str):
        if var not in self.use:
            self.use.append(var)

    def add_with_functions(self, var: str):
        if var not in self.with_functions:
            self.with_functions.append(var)

    def add_operators(self, var: str):
        if var not in self.operators:
            self.operators.append(var)
```

In it you can see four arrays, each one representing every case in which a variable can be used. You can also see that the name and the first line in which the variable is used are also there too. In addition to that we also have 4 defined functions which will add more information into each of the arrays.

```python
# returns a string list with all the imported libraries (including ones with the operation "from")
def importedLibraries(red):
    nodes = red.find_all("import")
    result = []
    counter = 0
    for node in nodes:
        result.insert(counter, "    >   " + str(node).split("import")[1] + ": line " + str(
            node.absolute_bounding_box.top_left.line))
        counter += 1
    nodes = red.find_all("fromimport")
    for node in nodes:
        result.insert(counter, "    >   " + str(node).split("import")[1] + ": line " + str(
            node.absolute_bounding_box.top_left.line))
        counter += 1
    return result


# returns a string list with all the defined functions
def definedFunctions(red):
    nodes = red.find_all("def")
    result = []
    counter = 0
    for node in nodes:
        result.insert(counter, "    >   " + str(node).split("def")[1].split(":")[0] + ": line " + str(
            node.absolute_bounding_box.top_left.line))
        counter += 1
    return result


# returns a string list with all the comments in the code
def commentsOnCode(red):
    nodes = red.find_all("comment")
    result = []
    counter = 0
    for node in nodes:
        result.insert(counter,
                      "    >   " + str(node).split("#")[1] + ": line " + str(node.absolute_bounding_box.top_left.line))
    return result
```

Next up we have these three functions which will each one return a string list with the information we need, in this cases we obtain all the imported libraries, all the defined functions and finally all the comments in the code. Each one of them has a redbaron variable called red as parameters in order to be able to read python code from them. The file has already been opened so we don’t have to worry about which file we are in right now.

```
# returns a variable list with all the used variables
def usedVariables(red):
    vList = []  # stores every found variable
    vNames = []  # stores the name of every variable without repetitions
    nodes = red.find_all("assignment")
    for node in nodes:
        var = str(node.name)
        if '+' and '-' and '*' and '/' not in var:
            if var not in vNames:
                v = variable(var, int(node.absolute_bounding_box.top_left.line))
                vList.append(v)
                vNames.append(var)
    nodes = red.find_all("tuple")
    for node in nodes:
        var = str(node.name)
        if var not in vNames:
            v = variable(var, int(node.absolute_bounding_box.top_left.line))
            vList.append(v)
            vNames.append(var)
    nodes = red.find_all("for")
    for node in nodes:
        if len(str(node.iterator).split(",")) > 1:
            var = str(node.iterator).split(", ")
            for v in var:
                if v not in vNames:
                    vr = variable(v, int(node.absolute_bounding_box.top_left.line))
                    vList.append(vr)
                    vNames.append(var)
        else:
            if str(node.iterator) not in vNames:
                v = variable(node.iterator, int(node.absolute_bounding_box.top_left.line))
                vList.append(v)
                vNames.append(str(node.iterator))
```

Here we get to the functions related with used variables in the code. This function will return a list of variable objects with all the used variables in the code. Just like the previous functions we get a redbaron variable called red in order to read python code.

First of all we search for every variable in the code and we put them in the vList array as a variable object and in the vNames array as a string with the name of the variable in it. We need to search for the first instance of every different variable assignment, tuple and for iterations.

There are 4 ways in which a variable can be used, these are **Reassignments**, **Use**, **With functions** and **Operators**.

- **Reassignment:**

```
# we search for every variable reassignment
nodes = red.find_all("assignment")
for node in nodes:
    var = str(node.name)
    if '+' and '-' and '*' and '/' not in var:
        for v in vList:
            if v.name == var:
                if int(v.line) != int(node.absolute_bounding_box.top_left.line):
                    v.add_reassignment(node.absolute_bounding_box.top_left.line)
                    break
```

In order to search for every reassignment, we need to check every assignment in the code and then check if that assignment is the first instance of a variable or not. If it isn't, then that means that that value has been reassigned. We don't take into account +=, -=, *= or /= as those are classified in another way.

- **Use:**

```
# we search for every used variable
nodes = red.find_all("comparison")
for node in nodes:
    var = str(node)
    if ' < ' in var:
        var = var.split(' < ')
    elif ' > ' in var:
        var = var.split(' > ')
    elif ' == ' in var:
        var = var.split(' == ')
    elif ' != ' in var:
        var = var.split(' != ')
    for v in var:
        for vb in vList:
            if str(v) == vb.name:
                if int(vb.line) != int(node.absolute_bounding_box.top_left.line):
                    vb.add_use(node.absolute_bounding_box.top_left.line)
nodes = red.find_all("binaryOperator")
for node in nodes:
    var = str(node)
    if ' + ' in var:
        var = var.split(' + ')
    elif ' - ' in var:
        var = var.split(' - ')
    elif ' * ' in var:
        var = var.split(' * ')
    elif ' / ' in var:
        var = var.split(' / ')
    for v in var:
        if '[' in v:
            v = v.split('[')[0]
        if '(' in v:
            v = v.split('(')[0]
        for vb in vList:
            if str(v) == vb.name:
                if int(vb.line) != int(node.absolute_bounding_box.top_left.line):
                    vb.add_use(node.absolute_bounding_box.top_left.line)
```

Next up are used variables. A variable can be used in a variety of ways:

- When a variable is compared
- When an operation (+, -, *, /) is used on a variable
- When a variable is used in a function call
- When a +=, -=, *= or /= is used on a variable

Since we have stored all the variables found in the code, we can begin searching for these operations. In order to find if a variable has been compared, we need to use the redbaron node “comparison”.

After this we filter the found results and add them to our lists if necessary. We proceed to do the same thing but with the redbaron node “binaryOperator” to find if an operation has been used on a variable. We also take into account if an operation has been done on an array or a string has been expanded through a ‘+’ operator.

Just like before, we proceed to do the same thing but with the redbaron nodes “getItem” for variables inside call functions, “atomTrailers” for every call function and “assignment” again for +=, -=, *= and /= operators. AtomTrailer in specific returns a “trail” of a call functions such as if we were to find a.b.c[x].f(thing) it would return the entire thing. It’s also worth mentioning that in this case we always take into account if a variable has been called in a function call or not.

```
nodes = red.find_all("getItem")
for node in nodes:
    var = str(node).split('[')[1]
    var = var.split(']')[0]
    if '(' in var:
        var = var.split('(')[1]
        var = var.split(')')[0]
    for v in var:
        for vb in vList:
            if str(v) == vb.name:
                if int(vb.line) != int(node.absolute_bounding_box.top_left.line):
                    vb.add_use(node.absolute_bounding_box.top_left.line)
nodes = red.find_all("atomTrailers")
for node in nodes:
    counter = 0
    for n in node:
        var = str(n)
        if '(' in var:
            var = var.split('(')[1]
            var = var.split(')')[0]
        if '[' in var:
            var = var.split('[')[1]
            var = var.split(']')[0]
        if ',' in var:
            var = var.split(', ')
            for vr in var:
                for v in vList:
                    if str(v.name) == vr and counter != 0:
                        v.add_use(node.absolute_bounding_box.top_left.line)
        else:
            for v in vList:
                if str(v.name) == var and counter != 0:
                    v.add_use(node.absolute_bounding_box.top_left.line)
        counter += 1
nodes = red.find_all("assignment")
for node in nodes:
    var = str(node)
    if '+' or '-' or '*' or '/' in var:
        for v in vList:
            if var.split("= ")[1] == v.name:
                v.add_use(node.absolute_bounding_box.top_left.line)
                break
```

- **With functions:**

```
# we search for every variable used within functions
variab = []
nodes = red.find_all("callArgument")
for node in nodes:
    var = str(node)
    if '"' not in var:
        vr = []
        if '+' in var:
            vr.extend(var.split(' + '))
        elif '-' in var:
            vr.extend(var.split(' - '))
        elif '*' in var:
            vr.extend(var.split(' * '))
        elif '/' in var:
            vr.extend(var.split(' / '))
        elif '.' in var:
            vr.extend(var.split('.'))
        elif '[' in var:
            v1 = var.split('[')[0]
            v2 = var.split('[')[1]
            v2 = v2.split(']')[0]
            vr.extend(v1)
            vr.extend(v2)
        else:
            vr.extend(var)
        for vari in vr:
            for v in vList:
                if str(v.name) == vari:
                    variab.append(vari)
                    break
```

As for with functions, we need to search for every variable in a call function so we search for every callArgument redbaron node and store every found variable in our list into another variab list while also filtering it looking for +, -, *, /, ., or [signs.

```
nodes = red.find_all("atomTrailers")
for node in nodes:
    string = ""
    counter = 0
    for n in node:
        var = str(n)
        if '(' not in var:
            if counter == 0:
                string = var
            else:
                string += "." + var
        else:
            var = var.split("(")[1]
            var = var.split(")")[0]
            vr = []
            if '+' in var:
                vr.extend(var.split(' + '))
            elif '-' in var:
                vr.extend(var.split(' - '))
            elif '*' in var:
                vr.extend(var.split(' * '))
            elif '/' in var:
                vr.extend(var.split(' / '))
            elif '.' in var:
                vr.extend(var.split('.'))
            elif '[' in var:
                v1 = var.split('[')[0]
                v2 = var.split('[')[1]
                v2 = v2.split(']')[0]
                vr.extend(v1)
                vr.extend(v2)
```

Now we make another atomTrailer node search and again filter it with our parameters. After this we start iterating on our results and try to find if one of the values also belongs to our current variable list. We also need to take into account the called function that we will need to store alongside the variable and the line of use. **I also need to note that since use operator takes into account if a variable is used in a system call, every time the program finds a with function variable then that variable will also be a use variable.**

```
        else:
            vr.extend(var)
        for vari in vr:
            for v in vList:
                if str(v.name) == vari:
                    found = False
                    for function in v.with_functions:
                        if string == str(function).split(':')[0]:
                            function += ", " + str(node.absolute_bounding_box.top_left.line)
                            found = True
                            break
                    if not found:
                        v.add_with_functions(
                            string + ": line " + str(node.absolute_bounding_box.top_left.line))
                    break
    counter += 1
```

- **Operators:**

```
# we search for every variable used with operators
nodes = red.find_all("atomTrailers")
for node in nodes:
    found = False
    for n in node:
        if found:
            if '[' not in str(n):
                v.add_operators("." + str(n) + ": line " + str(node.absolute_bounding_box.top_left.line))
                break
        for v in vList:
            if str(n) == v.name:
                found = True
                break
```

A variable is used as an operator if there's a called function from it in the code (as in a.write(b) for variable a). Since we already have all the variables stored in vList the only thing we need to do is look for atomTrailers and see if one of the values is a variable of our list. Finally after all of this, we return vList as a value.

Now we move into functions. Just like with variables, functions can be used in a variety of ways:

- **From libraries**
- **Generic**
- **With variable**

We have a usedFunctions function ready that returns a string list with all the found used functions in the code.

```
# returns a string list with all the used functions
def usedFunctions(red):
    usedFunctions = [] # here we will store all the arrays for the result
```

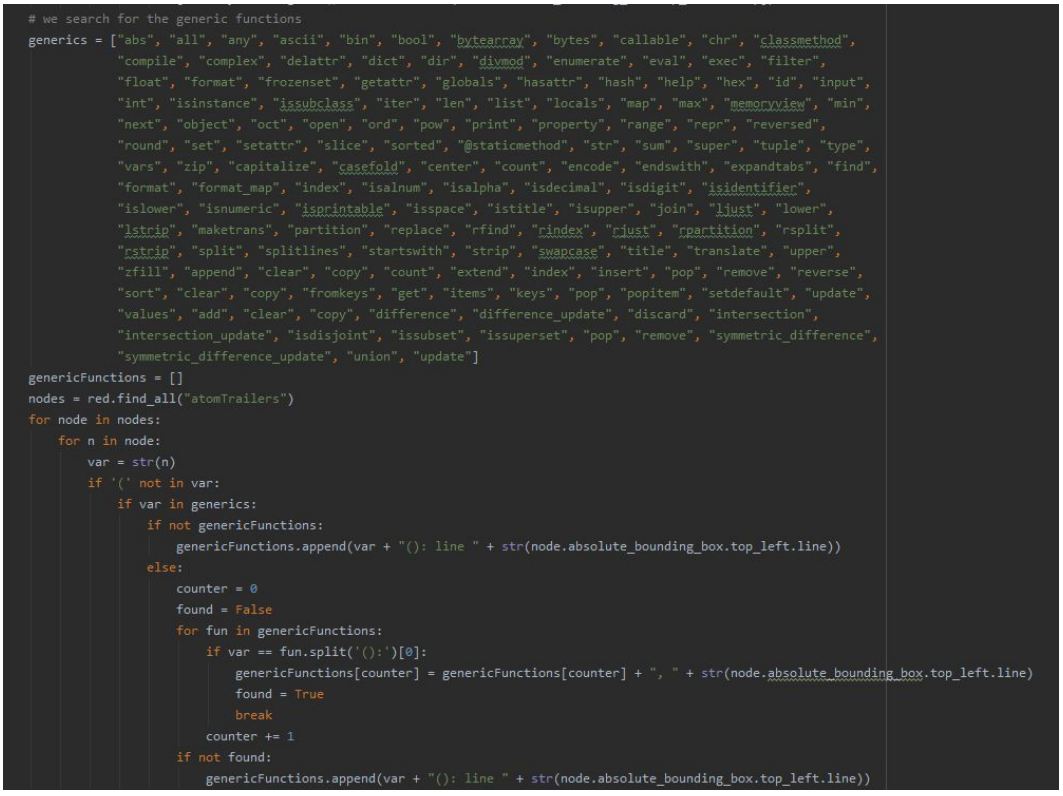
- **From libraries**

```
# we search for all the imported functions
fromFunctions = []
libraries = []
lib = importedLibraries(red)
for library in lib:
    library = library.split('    >    ')[1]
    library = library.split(':')[0]
    library = library.split(" ")[1]
    libraries.append(library)
nodes = red.find_all("atomTrailers")
for node in nodes:
    string = ""
    library = ""
    found = False
    for n in node:
        var = str(n)
        if '(' not in var:
            if not found:
                for lib in libraries:
                    if var == lib:
                        found = True
                        library = var
                        break
            else:
                string += "." + var
    if found:
        if not fromFunctions:
            fromFunctions.append([library, string + "():  line " + str(node.absolute_bounding_box.top_left.line)])
        else:
            found = False
            for lb in fromFunctions:
                if lb[0] == library:
                    found = True
                    counter = 0
                    for fn in lb:
                        if counter != 0:
                            if fn.split('():')[0] == string:
                                lb[counter] = lb[counter] + ", " + str(node.absolute_bounding_box.top_left.line)
                                break
                        counter += 1
                    break
            if not found:
                fromFunctions.append(
                    [library, string + "():  line " + str(node.absolute_bounding_box.top_left.line)])
```

First we get all the imported libraries through our importedLibraries() function and we filter it. After this we start searching for every atomTrailer in the code and check if one of the values has the same name as one of our imported libraries. We also need to make sure that the library and function aren't already on our list and if they are, add the current position of our line in the code to that function in the list.

- **Generic**

I have recompiled every possible generic function in python into an array that I then use to filter out values in atomTrailer nodes. If a generic function has been found in an atomTrailer node then I add it into our genericFunction list and so on.



```
# we search for the generic functions
generics = ["abs", "all", "any", "ascii", "bin", "bool", "bytearray", "bytes", "callable", "chr", "classmethod",
            "compile", "complex", "delattr", "dict", "dir", "divmod", "enumerate", "eval", "exec", "filter",
            "float", "format", "frozenset", "getattr", "globals", "hasattr", "hash", "help", "hex", "id", "input",
            "int", "isinstance", "issubclass", "iter", "len", "list", "locals", "map", "max", "memoryview", "min",
            "next", "object", "oct", "open", "ord", "pow", "print", "property", "range", "repr", "reversed",
            "round", "set", "setattr", "slice", "sorted", "@staticmethod", "str", "sum", "super", "tuple", "type",
            "vars", "zip", "capitalize", "casefold", "center", "count", "encode", "endswith", "expandtabs", "find",
            "format", "format_map", "index", "isalnum", "isalpha", "isdecimal", "isdigit", "isidentifier",
            "islower", "isnumeric", "isprintable", "isspace", "istitle", "isupper", "join", "ljust", "lower",
            "lstrip", "maketrans", "partition", "replace", "rfind", "rindex", "rjust", "rpartition", "rsplit",
            "rstrip", "split", "splitlines", "startswith", "strip", "swapcase", "title", "translate", "upper",
            "zfill", "append", "clear", "copy", "count", "extend", "index", "insert", "pop", "remove", "reverse",
            "sort", "clear", "copy", "fromkeys", "get", "items", "keys", "pop", "popitem", "setdefault", "update",
            "values", "add", "clear", "copy", "difference", "difference_update", "discard", "intersection",
            "intersection_update", "isdisjoint", "issubset", "issuperset", "pop", "remove", "symmetric_difference",
            "symmetric_difference_update", "union", "update"]
genericFunctions = []
nodes = red.find_all("atomTrailers")
for node in nodes:
    for n in node:
        var = str(n)
        if '(' not in var:
            if var in generics:
                if not genericFunctions:
                    genericFunctions.append(var + "(): line " + str(node.absolute_bounding_box.top_left.line))
                else:
                    counter = 0
                    found = False
                    for fun in genericFunctions:
                        if var == fun.split('():')[0]:
                            genericFunctions[counter] = genericFunctions[counter] + ", " + str(node.absolute_bounding_box.top_left.line)
                            found = True
                            break
                        counter += 1
                    if not found:
                        genericFunctions.append(var + "(): line " + str(node.absolute_bounding_box.top_left.line))
```

- **With variable**

Just like some of the previous examples, we need to find every variable in the code. Here I just reuse and rearrange some of the code I wrote earlier but this time I just store the name of each value on an array instead of creating a variable object and storing it.

```
# we search for all the used functions with variables
withVariable = []
vNames = []  # stores the name of every variable without repetitions
nodes = red.find_all("assignment")
for node in nodes:
    var = str(node.name)
    if '+' and '-' and '*' and '/' not in var:
        if var not in vNames:
            vNames.append(var)
nodes = red.find_all("tuple")
for node in nodes:
    var = str(node.name)
    if var not in vNames:
        vNames.append(var)
nodes = red.find_all("for")
for node in nodes:
    if len(str(node.iterator).split(",")) > 1:
        var = str(node.iterator).split(", ")
        for v in var:
            if v not in vNames:
                vNames.append(v)
    else:
        if str(node.iterator) not in vNames:
            vNames.append(str(node.iterator))
```

```
nodes = red.find_all("atomTrailers")
for node in nodes:
    string = str(node)
    n = str(node).split(".")
    for nd in n:
        var = str(nd)
        if '(' not in var:
            for v in vNames:
                if str(v) == var:
                    if not withVariable:
                        withVariable.append([var, string + ": line " + str(node.absolute_bounding_box.top_left.line)])
                    else:
                        counter = 0
                        foundVar = False
                        foundFun = False
                        for vr in withVariable:
                            if vr[0] == var:
                                foundVar = True
                                for ln in vr:
                                    if counter != 0:
                                        if ln == string.split(':')[0]:
                                            vr[counter] = vr[counter] + str(node.absolute_bounding_box.top_left.line)
                                            foundFun = True
                                            break
                                    counter += 1
                                break
                        if not foundVar:
                            if '(' in string:
                                string = string.split("(")[0]
                                withVariable.append([var, string + "(): line " + str(node.absolute_bounding_box.top_left.line)])
                            else:
                                withVariable.append([var, string + ": line " + str(node.absolute_bounding_box.top_left.line)])
                        if not foundFun and foundVar:
                            for vr in withVariable:
                                if vr[0] == var:
                                    if '(' in string:
                                        string = string.split("(")[0]
                                        vr.append(string + "(): line " + str(node.absolute_bounding_box.top_left.line))
                                    else:
                                        vr.append(string + ": line " + str(node.absolute_bounding_box.top_left.line))
                                    break
                        break
```

Now we can start looking for atomTrailers. I keep searching for every node in an atomTrailer that does not contain an “(“ character since we are only looking for function calls from variables not the variables in the call themselves. Since a variable can have more than one different function call, I have to store every result into an array of arrays in which on the first position of the second iteration of arrays, I store the name of the found variable and the called function and corresponding lines on the rest of them. Every time I find a new function call on a variable that I already have, I need to check if that function isn’t already on the list. If it is then I need to add the line position of that function on the existing function of the same name.

```
# we recollect all the data we've found
usedFunctions.append(fromFunctions)
usedFunctions.append(genericFunctions)
usedFunctions.append(withVariable)
return usedFunctions
```

Finally I add every array into our big usedFunctions array that I then return as a value.

Next up is our graph generator that we use to generate the structure of our program. As we previously said, we are using NetworkX in order to create and fill graphs with nodes and edges which then we can draw with the library matplotlib and save them as .png to further insert them into our PDF later.

```
# generates the graph of the structure of the directory
def graphGenerator(directory):
    G = nx.Graph()
    totalLibraries = []  # here we store all the used libraries
    functions = []  # here we store all the defined functions
    scripts = []  # here we store lists with the libraries and the filename they are used in
    filenames = []
```

First we create a new graph with nx.Graph() and initialize the arrays we need to sort out the structure of the project. We have an array to store all the used libraries, another to store all the defined functions, another one to store the name of every python script in our directory and another final one to sort out the files that we have already added to our graph.

```
for root, dirs, files in os.walk(directory):
    for file in files:
        if file.endswith(".py"):
            filename = os.path.join(root, file)
            filename = filename.split("\\")[1]
            if filename not in G.nodes:
                G.add_node(filename)
                if filename not in filenames:
                    filenames.append(filename)
            with open(os.path.join(root, file), "r") as f:
                red = redbaron.RedBaron(f.read())
                # we search for all the imports and from imports and store them
                nodes = red.find_all("import")
                fileLibraries = []
                for node in nodes:
                    fileLibraries.append(str(node).split("import ")[1])
                nodes = red.find_all("fromimport")
                for node in nodes:
                    if str(node).split("import ")[1] not in fileLibraries:
                        fileLibraries.append(str(node).split("import ")[1])
                for library in fileLibraries:
                    if library not in totalLibraries:
                        totalLibraries.append(library)
                libraries = [filename]
                libraries.extend(fileLibraries)
                scripts.append(libraries)
                # we search for all the defined functions and store them
                nodes = red.find_all("def")
                for node in nodes:
                    funct = str(node).split("def")[1].split(":")[0]
                    while funct in functions:
                        funct += "\u2028"
                    functions.append(funct)
                    G.add_node(funct)
                    G.add_edge(filename, funct)
```

NetworkX has a function called .add_node(identifier) that lets you add a node to a graph. This node will have the information inside the identifier and you can use anything as an identifier to that node but since the .draw() function will draw the identifier in the graph then we need to use a string in order to draw the filename of every python script.

For starters we check every file in the directory that ends with a “.py” and we open it up. If the file isn’t already in the graph then we add it as a node. Then we search for every imported library and we add them into our arrays.

For functions, we search for every defined function in the code and every time we find a new function we add them as a node and we create an edge between it and the current file with the .add_edge(function1, function2). Since we are using strings as identifiers with our nodes we need to be careful since generic functions such as “main()” will have many edges between file nodes and we want to avoid that. In order to do that, every time we find that a function with the same identifier has already been added to the graph, we add an invisible UTF character into it to avoid having the same identifier.

```
# we generate the edges for the libraries
for script in scripts:
    counter = 0
    for lib in script:
        if counter != 0:
            if lib + ".py" in filenames:
                G.add_edge(script[0], lib + ".py")
        counter += 1

nx.draw(G, pos=nx.spring_layout(G, k=1000), with_labels=True, node_size=2000, node_color='#00ecff', node_shape='s')
plt.savefig("../resources/graph.png")
```

Once we have everything we need we start to iterate on our scripts and if we find that one of the libraries that has been used has the name of one of the scripts then we create an edge between them. To draw our graph, we use the function nx.draw() and save it as a png in our resources directory.

The remaining functions on the script will be used to manage the statistics part of the pdf report and each one of them will return an integer with the number of found functions, variables, libraries and lines in the code.

3. PDFGenerator.py

The PDFGenerator script will manage everything related to writing and inserting things into a PDF and will also act as a sort of “main” function of our program. The program will first ask for a directory with the python scripts we want to analyze and from there will generate a PDF with a series of statistics and reports about it. We will be using the library pyPDF to insert, write and generate our PDF.

We first start by setting up the configuration of our PDF. The PDF is divided into a series of categories:

- Cover
- Index
- Introduction
- Programs
- Statistics

Since we wanted to avoid hardcoding strings as much as we could, we have a resources folder with a series of .txt files that will be used to avoid writing some of the hardcoded parts of our code. You can also find there the images that we will be inserting later into our PDF. The following code is used for the setup of the cover. Since the cover is static, We didn’t bother with the function .write() here since we don’t need to worry about breaking pages here.

```
# setting up the pdf configuration
pdf = PDF(orientation='P', unit='mm', format='A4')
pdf.set_left_margin(25)
pdf.set_top_margin(35)
pdf.alias_nb_pages()
pdf.add_page()
pdf.set_font("Arial", size=12)

# setting up the cover line
pdf.set_line_width(4)
pdf.set_draw_color(255, 200, 0)
pdf.line(20, 275, 20, 10)

# setting up the cover image
pdf.image("../resources/tecnocampus.png", x=140, y=50, w=60)
```

```
# setting up the rest of the cover
f = open("../resources/Cover.txt", "r")
if f.mode == "r":
    lines = f.readlines()
    counter = 1
    for line in lines:
        if counter == 1:
            pdf.set_font_size(24)
            pdf.text(30, 150, line)
        elif counter == 2:
            pdf.text(30, 170, line)
        elif counter == 3:
            pdf.set_font_size(12)
            pdf.text(30, 200, line)
        elif counter == 4:
            pdf.set_font("Arial", 'B', 8)
            pdf.set_xy(30, 254)
            pdf.cell(0, 10, line)
        elif counter == 5:
            pdf.set_xy(30, 258)
            pdf.cell(0, 10, line)
        else:
            pdf.set_xy(30, 262)
            pdf.cell(0, 10, line)
        counter += 1
f.close()
```

Next we set up our index.Since the PDF still hasn't been completed, we can't get each one of the category foot pages since they don't exist yet. We will rearrange the index just before we finish the PDF later.

```
# creating the index
pdf.add_page()
f = open("../resources/Index.txt", "r")
if f.mode == "r":
    lines = f.readlines()
    counter = 1
    for line in lines:
        if counter == 1:
            pdf.set_font("Arial", 'B', 12)
            pdf.text(25, 35, line)  # Index
        elif counter == 2:
            pdf.text(25, 42, line)  # 1. INTRODUCTION
        elif counter == 3:
            pdf.set_font("Arial", '', 10)
            pdf.text(30, 47, line)  # 1.1 DESCRIPTION
        elif counter == 4:
            pdf.text(30, 52, line)  # 1.2 OBJECTIVES
        elif counter == 5:
            pdf.text(30, 57, line)  # 1.3 STRUCTURE OF THE PROGRAM
        elif counter == 6:
            pdf.set_font("Arial", 'B', 12)
            pdf.text(25, 64, line)  # 2. THE PROGRAMS
            pdf.set_font("Arial", '', 10)
            yCounter = 67
            fileCounter = 1
            for root, dirs, files in os.walk(directory):
                for file in files:
                    if file.endswith(".py"):
                        filename = os.path.join(root, file)
                        filename = "2." + str(fileCounter) + " " + filename.split('\\')[1]
                        pdf.set_xy(30, yCounter)
                        pdf.write(5, filename)
                        yCounter += 5
                        fileCounter += 1
        elif counter == 7:
            pdf.set_font("Arial", 'B', 12)
            yCounter += 5
            pdf.text(25, yCounter, line)  # 3. SOME STATISTICS
        counter += 1
f.close()
```

After that we set up the introduction category. Most of the information in the introduction category is static and we only need to worry about the files of the project and the structure of the program. To find out the files in the project we just use the function os.walk from the python library os with our existing directory. After that we just write what we need in the PDF.

```
elif counter == 3:  # Files of the project
    pdf.set_font("Arial", '', 10)
    pdf.set_text_color(0)
    pdf.write(5, line.split('!')[0] + directory + line.split('!')[1])
    pdf.ln()
    for root, dirs, files in os.walk(directory):
        for file in files:
            if file.endswith(".py"):
                filename = os.path.join(root, file)
                filename = "    >   " + filename.split('\\')[1]
                pdf.set_font("Arial", 'B', 10)
                pdf.write(5, filename)
                pdf.ln(10)
```

As for the structure of the project, we just use the graphGenerator function we created earlier and insert the generated .png image into the PDF.

```
elif counter == 8:  # 1.3 Structure of the project
    pdf.add_page()
    pdf.set_text_color(255, 200, 0)
    pdf.set_font("Arial", 'B', 12)
    pdf.write(5, line)
    pdf.ln()
    pdfPages.append(pdf.page_no())
elif counter == 9:
    pdf.set_font("Arial", '', 10)
    pdf.set_text_color(0)
    pdf.write(5, line)
    pdf.ln(10)
    graphGenerator(directory)
    pdf.image("../resources/graph.png", x=30, y=50, w=150)
counter += 1
```

The next two categories are the programs and statistics sections which again we mostly create by calling in the functions from scriptReader.py we created previously. This is the most extensive part of the code in PDFGenerator but we don't think it requires much explanation since yet again most of it is just calling in our own functions.

To end our program, we reformat the index by indexing our program to the page where the index is which will always be the second page and after this inserting the foot page value into each of every single one of our categories and subcategories. Finally we output our PDF into our directory and we close it.

It is important to note that I have changed every instance of “Català” in every single one of the examples into “Catala” since it sometimes was giving me problems when trying to write latin characters into the PDF.

From our experience, the program takes about 20 seconds to run the first example and about a minute to run the second one. We know the code isn’t exactly optimized or optimal so we ask you to please be patient with it.

```
# reformatting the index
lastPage = pdf.page
pdf.page = 2
y = 42
yCounter = 7
for page in pdfPages:
    if str(page) == "!":
        yCounter = 5
        pdf.set_font("Arial", '', 10)
    elif str(page) == "?":
        yCounter = 7
        pdf.set_font("Arial", 'B', 10)
    else:
        pdf.set_font("Arial", 'B', 10)
        pdf.text(185, y, "pg " + str(page))
        y += yCounter
pdf.page = lastPage

# finish the PDF
pdf.output('../Report_TCM.pdf', 'F')
pdf.close()
```

4. Execution of the program