REPORT ON CPYTHON

Goal of the Project / Abstract

The goal of this project is to practice building and testing large projects and analyzing large software projects and providing our assessment of testing in the following project. Building, Testing and Coverage which is defined as a metric to evaluate the amount of testing performed by a set of tests. Analyzing through Assert and Debug Statements in Test and Production Files, and GitHub analysis of CPython contributions.

The project aims at to investigate all the above-mentioned details in both Developer and Tester perspectives, considering adequacy of the tests, and the appropriateness of testing process.

1. Introduction

1.1 What is Cpython?

The original Python implementation is CPython. People refer to it as CPython to separate it from later Python implementations, as well as to distinguish the language engine implementation from the Python programming language itself.

CPython is also the first to provide new features; Python-the-language development starts with CPython, and other implementations follow.

CPython is written in the C programming language. That is, after all, simply a minor implementation detail. CPython compiles your Python code (transparently) into bytecode, which it then interprets in an evaluation loop.

1.2 Memory Management in CPython:

The whole concept of memory management has a reference point called PyArena object. The wrapper for C's memory allocation and deallocation routines is found in Python/pyarena.c.

When the allocated memory is no longer in use, the developer must deallocate, or free, it and restore it to the operating system's block table of free memory. When a process allocates memory for a variable, such as within a function or loop, the memory is not automatically returned to the operating system in C when the function is completed. It causes a memory leak if it hasn't been explicitly deallocated in the C code. Each time that function is executed, the process will consume more memory until the system runs out of memory and crashes.

Now, python comes play a major role by utilizing two techniques to take this burden away from the programmer: a reference counter and a garbage collector.

1.3 Subdirectories in Cpython:

Doc	Source for the documentation
Grammar	The computer-readable language definition
Include	The C header files
Lib	Standard library modules written in Python
Mac	macOS support files
Misc	Miscellaneous files
Modules	Standard Library Modules written in C
Objects	Core types and the object model
Parser	The Python parser source code
PC	Windows build support files
PCbuild	Windows build support files for older Windows versions
Programs	Source code for the python executable and other binaries
Python	The CPython interpreter source code
Tools	Standalone tools useful for building or extending Python

2. Building and Coverage:

The applications needs to be build for further testing and analysis.

2.1 Steps To Build:

The Application is built by using Linux, there are several dependencies we have faced while building this application, for example:

There are several dependencies and commands used to build the application, which were included in Steps_to_build.txt

2.2 Main test command:

make test

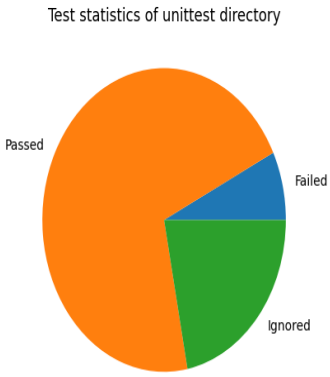
The above Command **make test** runs all the tests, The test Result shows success, if all the tests have been run successfully. The time taken to run these tests is approximately 5 min 10 sec, there are a total of 420 tests that have run successfully and 15 tests that have been skipped. Every other day the test cases are being added by the testers.

2.3 Test Metrices:

Test activities are measured and monitored by software testing metrices. They give insights about the time taken to run the tests cases, information about the number of bugs found, fixed and closed. It also gives information about how much software was tested. The key metrics for test coverage are Requirement Coverage, Test Execution Coverage, Test Execution Summary. And the absolute numbers in the test metrices are total number of test files and test cases,

number of test cases passed and failed. We can also calculate and document the number of test hours planned and number of actual test hours. Another test metrics are economic metrics like total cost estimated and actual cost of testing, cost per bug fix.

Example of a test metric – Unittest directory of the project cpython:



Tests failed are 16, passed are 155, ignored are 48 of 219 items in unittest directory. Time taken to run the tests is 304 ms.

Total Number of test cases and passed cases:

```
0:05:10 load avg: 4.30 [435/435] test_concurrent_futures passed (3 min 8 sec)

== Tests result: SUCCESS ==

420 tests OK.

15 tests skipped:
    test_devpoll test_gdb test_ioctl test_kqueue test_launcher
    test_msilib test_ossaudiodev test_startfile test_tix test_tk
    test_ttk_guionly test_winconsoleio test_winreg test_winsound
    test_zipfile64

Total duration: 5 min 10 sec
Tests result: SUCCESS
harika@DESKTOP-LDHCK56:~/cpython/Python-3.11.0a7$
```

2.4 Coverage:

After successfully building and testing the application, coverage is the next definite step to see if the files have been covered atmost during testing.

It is necessary to know if the project has been tested thoroughly with different method of coverages. Three practical coverage methods used in this project are Statement Coverage, Branch Coverage and Function Coverage.

Code coverage is a measure which describes the degree of which the source code of the program has been tested. It assists you in determining the effectiveness of test implementation. It provides a quantitative evaluation. It indicates how thoroughly the source code has been tested.

2.5 There are different types of coverage:

Statement Coverage - is a white box testing technique in which all the executable statements in the source code are executed at least once.

Statement Coverage main goal is to make use of all the files, paths, and statements to maximum, The Statement Coverage is generally calculated by:

Statement Coverage = (Number of Executed Statements / Total Number of Statements) * 100

Commands used in Linux to obtain Statement Coverage are

python-coverage run -m pytest

python-coverage report

We have obtained a Coverage of 12% on overall.

Branch Coverage - The purpose of branch coverage is to ensure that each condition from every branch is executed at least once. It aids in the calculation of fractions of independent code segments and the identification of sections with no branches.

Commands used in Linux to obtain Branch Coverage are

python-coverage run --branch -m pytest

python-coverage report >> coverage_branch.csv → The Report has been saved in .csv File

python-coverage html → This Generates a htmlcov Folder with .html files for every file and contains code coverage of every file.

The formula to calculate Branch Coverage:

Branch Coverage = Number of Executed Branches / Total Number of Branches

We have obtained a Branch Coverage of 12% on overall

Function Coverage - The statistic of how much design functionality has been exercised/covered by the testbench or verification environment, as described by the verification engineer in the form of a functional coverage model, is known as functional coverage.

In its most basic form, it is a user-defined mapping of each functional feature to be checked to a so-called **cover point**, which has requirements (ranges, defined transitions or cross etc.)

Total count of the test file, assert are found here.

2.6 Plot to Visualize Coverage

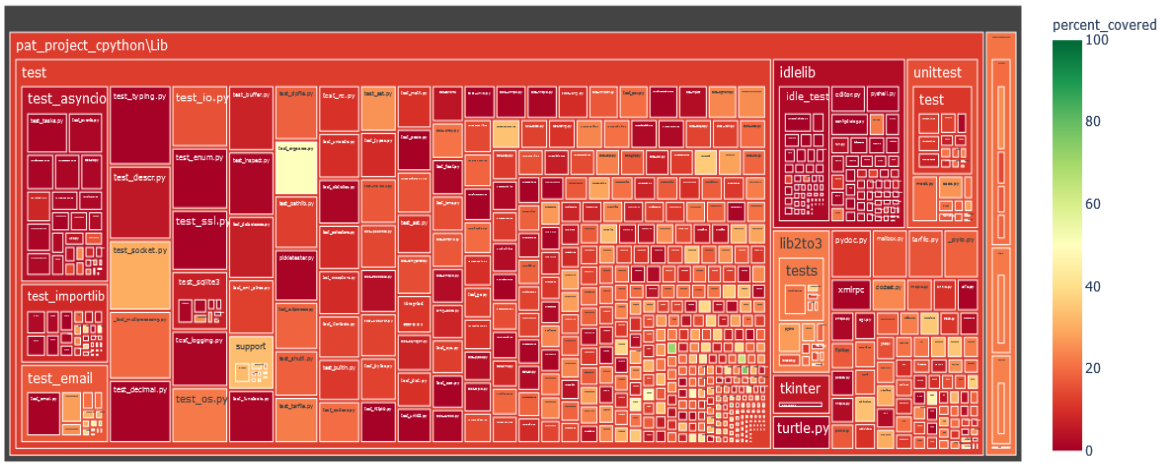
coverage-plot in python is used to plot the coverage obtained, the following are the commands to obtain the coverage plot:

- pip install coverage-plot
- coverage-plot coverage.xml

When the coverage is obtained a .xml file with name coverage will be generated, that is used in plotting the coverage.

The plot visualizes all the subfolders, files and coverage of each file, as we can see mostly of the plot is colored in red and orange, which indicates most number of files have coverage in the range of 0 to 20 and 20 to 40 percent when we see the scale.

As discussed above, the overall percent is 12 percent, the overall percent is effected as there are many individual files, whose coverage is in the range of 0 to 20 percent.



3. Appropriateness of testing process:

There are two points in testing process that should use to determine the best time to write test cases.

1. **Requirements:** Creating test cases before requirements are ready. In this case, test cases are created for scenarios that will ultimately be abandoned or changed dramatically.

2. **Development:** Creating test cases when development is scheduled to begin and end. It would likely make the most sense to write the test cases for tasks during the development cycle.

When analyzing the dates and frequency of test files added to the project and test files modified in the project, we can observe authors are more involved in modifying the existing

files rather than adding the files. If observed, in the year 2001, modification done to the test files are 8237 while test files added were only 171.It is a clear sign that authors are writing and modifying the test cases in parallel to the development of the project.

4. Assert Statements and Usage:

To further analyze the test files and production files we consider the usage of assert statements.

An assertion is a statement that allows you to test your program's assumptions. If we build a method to see if it is a valid division, we can assert that the denominator should not be zero.

Each assertion has a Boolean expression that you expect to be true when the assertion is executed. The system will throw an error if it is not true. The assertion supports your assumptions about the behavior of your program by checking that the Boolean expression is true, boosting your confidence that the program is error-free.

4.1 Assert statements in Python and C

To employ assertion conditions in a program, Python offers a built-in assert statement. The assert statement has a condition or phrase that should always be true. If the condition is false, the program is terminated, and an Assertion Error is thrown.

Python Syntax: assert <condition> or assert <condition>, <error message>

C Syntax: void assert(int expression);

To calculate the number of assert statements in **Test files**, we have used RegEx (Regular Expressions), that searches for assert statements in each file (python and C) using **re.search()** function. And counted the number of test files.

4.2 How Assertions Sometimes Can Make Testing More Difficult

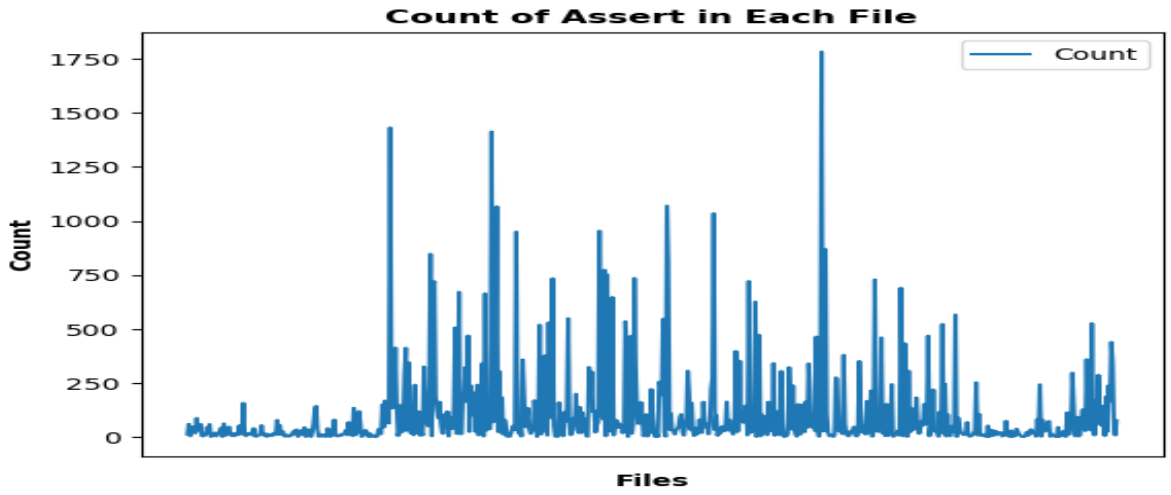
When an assumption fails for one reason or another, the ramifications might be disastrous. An Assertion could become an obstacle, causing testing to be halted for the entire day. Some of the conditions we want to examine are conceptually easy, but they are quite difficult to verify in practice.

4.3 Analysis of Assert Statements in Production Files

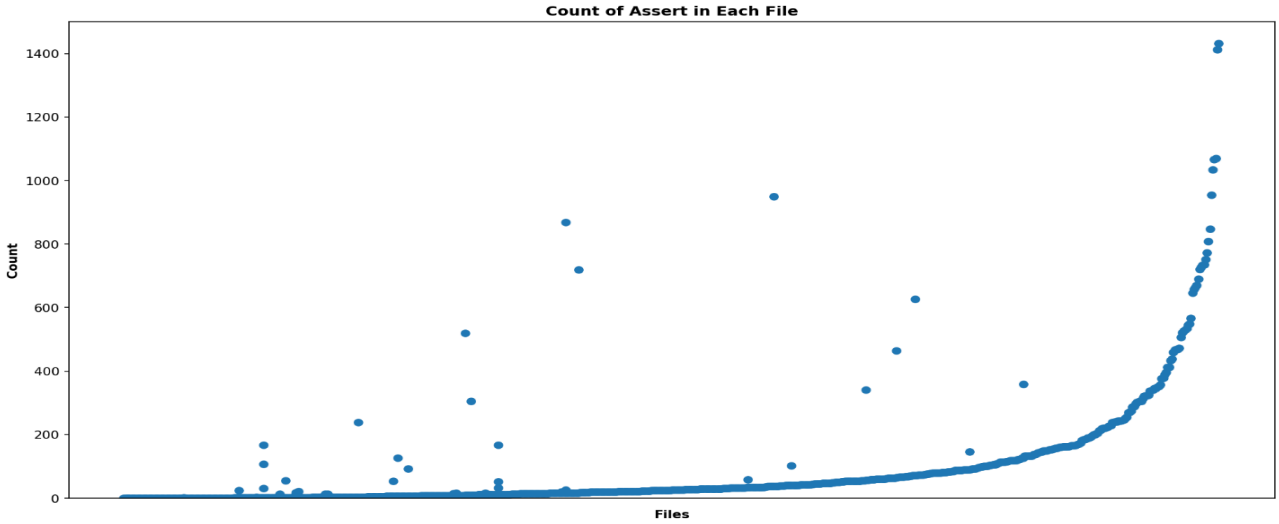
All the other files other than test files are assumed as production files.To calculate the count of Assert statements in **Production files**, we have used RegEx (Regular Expressions), that searches for Debug statements in each file using **re.search()** function.

4.5 Visualizing the Number of assert statements in each test file:

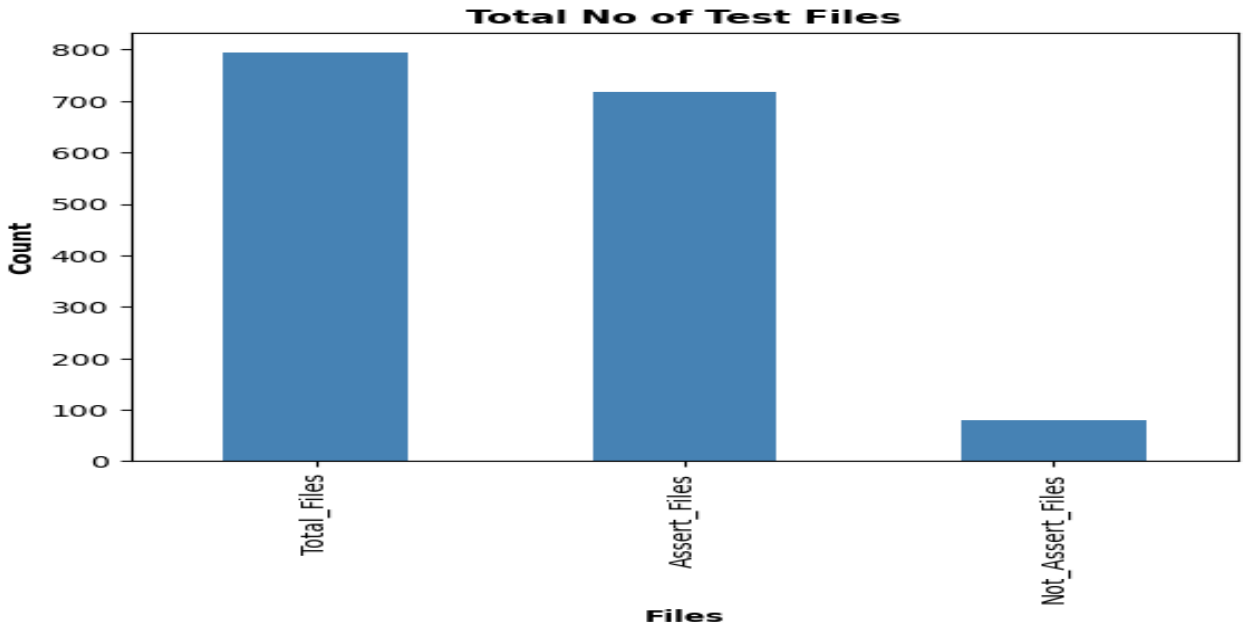
The below plot depicts the trend on how many assert statements each test file is having, from below it is clear that almost every test file have assert conditions in huge number. which indicates the there are many program assumptions which are being tested in each file.



The below plot depicts the same as above, but for a better visualization, we have plotted it in increasing order of count.

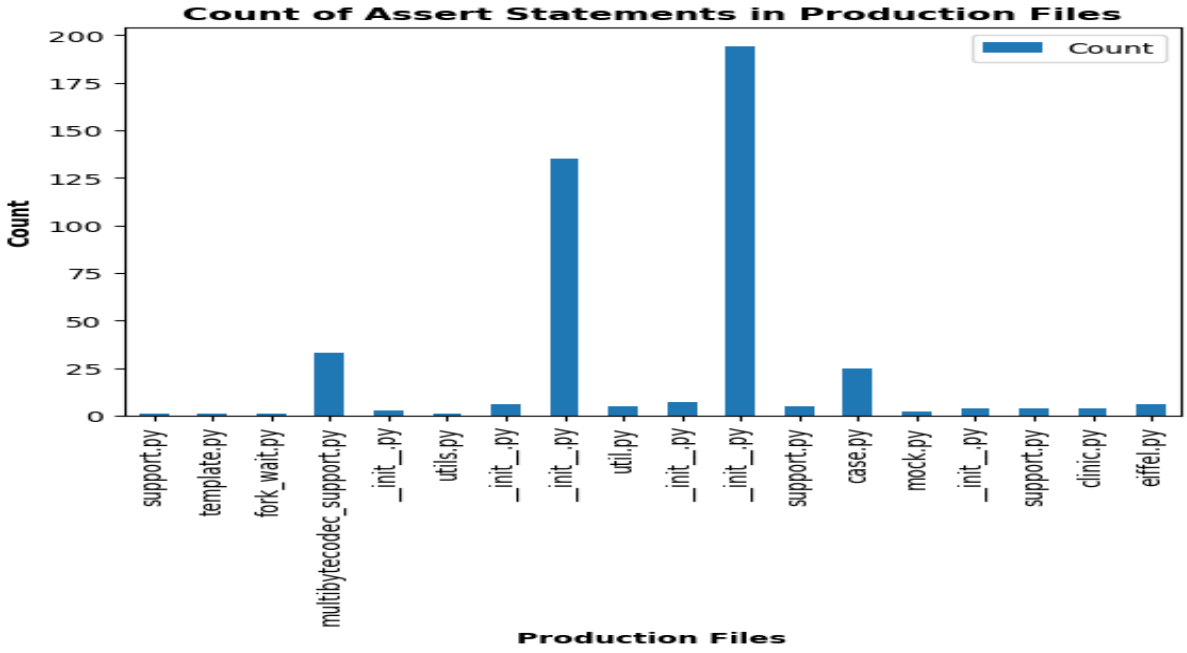


The below plot depicts the total number of files, and number of files with assert statements and without assert statements in test files, we can see that there are around 700 test files.



4.6 Visualizing the Number of assert statements in each Production file:

The below plot depicts the trend on how many assert statements each Production file is having, it is clear that there are very few number of assert statements in these files.Which concludes that development side has less assertion situations.



5. Debug Statements and Usage:

To further analyze the production files we consider the usage of debug statements.

5.1 What is Debugging?

The developer must discover the cause of a specific fault or flaw during the debugging process, which is done by thoroughly reviewing the coding. When a flaw or error is discovered, the developer fixes the code and then tests to see if the defect has been removed.

5.2 Testing vs Debugging

Debugging is done by either programmer or developer. The goal of debugging is to find and fix the problem. Testing the software doesn't reveal the cause of a bug, it just shows us the effect. But debugging does. The testing technique does not assist the developer in determining the nature of the coding issue. It finds out how the program is affected due to errors or issues in coding.

5.3 Debug statements in python

Logger.debug() is one of the most used statements in python for Debugging.

Python includes a logging system in its standard library, allowing you to quickly incorporate logging into your program.

In a programmer's arsenal, logging is an extremely useful tool. It can aid in the development of a better knowledge of a program's flow and the discovery of scenarios that you may not have considered while designing. Logs give developers with an extra pair of eyes that are always monitoring the flow of an application.

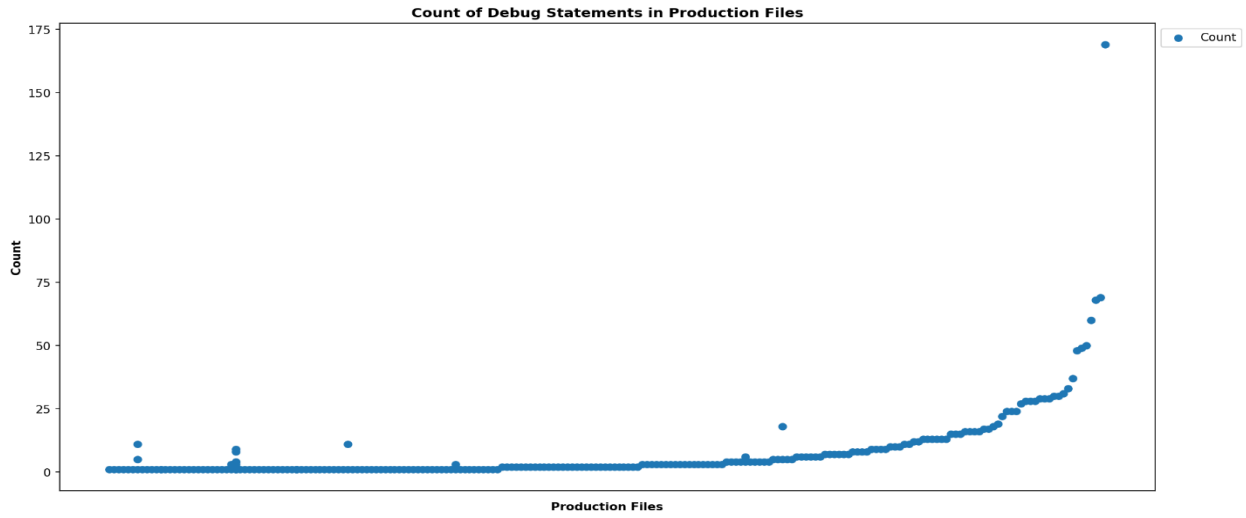
You can not only simply troubleshoot mistakes by recording relevant data from the correct locations, but you can also utilize the data to assess the performance of the application to plan for scaling or look at usage patterns to plan for marketing by logging useful data from the right places.

5.4 Analysis of Debug Statements in Production Files

To calculate the count of number of Debug statements in **Production files**, we have used RegEx (Regular Expressions), that searches for Debug statements in each file using **re.search()** function.

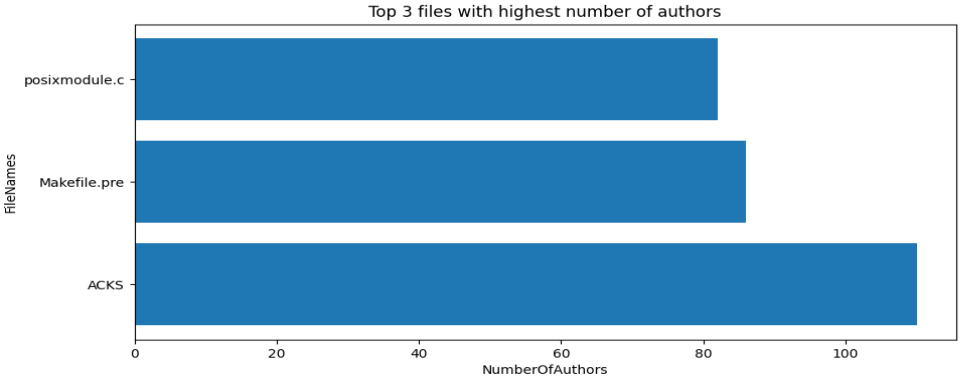
5.5 Visualizing the Number of Debug statements in each Production file:

The below plot depicts the trend on how many Debug statements each Production file is having, it is clear that there are quite a number of Debug statements in Production files.As the debugging is done by the developer, It is obvious that the production files have more number of Debug statements.

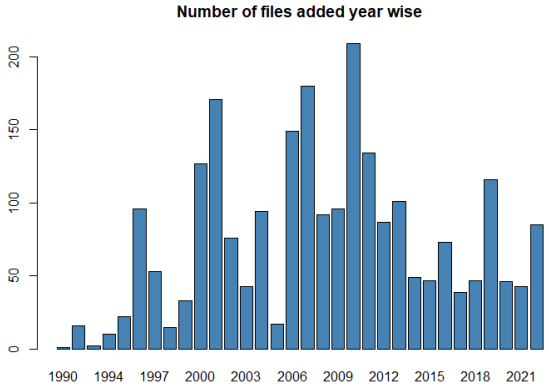


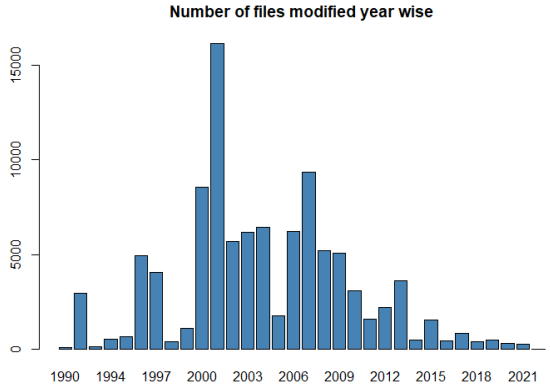
6. Visualizing and Analysis of the data extracted from Pydriller:

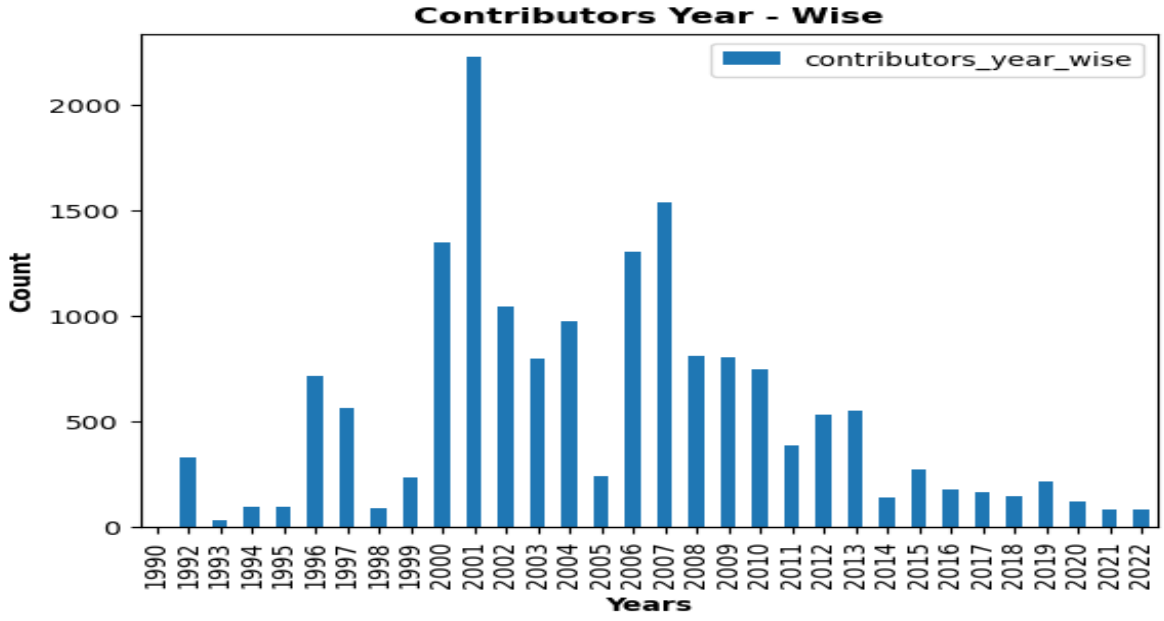
Out of all the years from 1990 to 2021, the highest number of test files were added in the year 2010 with 209 new test files. Although there are highest number of files added in 2010, the commits/modifications done are only 1756. More number of commits are done in the year 2001 with 8237 modifications by various authors. And from the data generated by pydriller, we can observe that for the file ACKS, 110 authors contributed. It is the file with highest number of contributors. As most of the authors are working on the same file (ACKS) in 1994, we can observe that there are not many commits and new files added in the year 1994. Top 3 files with highest number of authors are ACKS, Makefile.pre.in and posixmodule.c with 110, 86 and 82 authors respectively. In the recent years, from 2015- 2021, we can observe there are more files being added rather than modifying existing files. Also, from the data extracted using pydriller, the top 5 contributors out of 100 are Guido van Rossum, Fred Drake, Georg Brandl, Victor Stinner and Benjamin Peterson. Guido van Rossum is the only author who worked on 144 files single handedly. Rest all the authors worked on files teaming up with other authors. Also, Guido van Rossum is the author with highest number of commits on the project level with 10,921.



6.1 commits from 1990-2021.







7. Adequacy of the tests:

Testing is all about, giving assurance about a software component or product by running test cases on the component. The product is said to be legit when it passes all the test cases.

But when we dive deep into the testing process, to test a program and found out that there are some bugs or issues, we modify the program or component code and then add a few more test

cases relevant to test the modified code. We will keep on testing until there is no issue on the development side.

One of the important things that we should consider here is that testing reveals the faults only if they are present, but not their absence. So, we cannot conclude a program is issue free unless we can agree that the testing is performed with an adequate set of test cases.

From the analysis performed above, we can consider that there are an adequate set of tests to test the CPython, the contributors might have considered different scenarios while testing, as there are many modifications done to the test files in all the years.

8. Conclusion:

When we analyze the state of testing in both developer and tester perspective (through assert, debug and contribution analysis), It is observed that there are quite a number of additions, modifications and deletions of tests. Proving that there are enough number of changes to the test files that might have covered all the different cases, by which we can conclude that there are adequate set of tests. This might arise a question to come up with the adequate set of tests initially itself. But the main goal of testing is to find bugs or errors which might be compromised if that case is considered. And, when coming to the appropriateness of testing process there are more modifications than addition of files. Which specifies that the authors are writing and modifying the test cases in parallel to the development of the project. Which is more appropriate.

9. References:

https://realpython.com/cpython-source-code-guide/#compiling-cpython-linux

https://pypi.org/project/coverage-plot/

https://blog.testlodge.com/when-to-write-test-cases/

https://www.tricentis.com/blog/64-essential-testing-metrics-for-measuring-quality-assurance-success/

https://www.zenq.com/blogs/testing-metrics-influencing-major-test-release-decisions/

https://text-id.123dok.com/document/nzwmnrxlq-adequacy-of-testing-software-testing-and-quality-assurance-theory-and-practice-kshirasagar-naik-2008.html

10. Git Link:

https://github.com/harika175/PAT_Project_CPython

https://github.com/DineshNarlakanti/PAT_Project_CPython