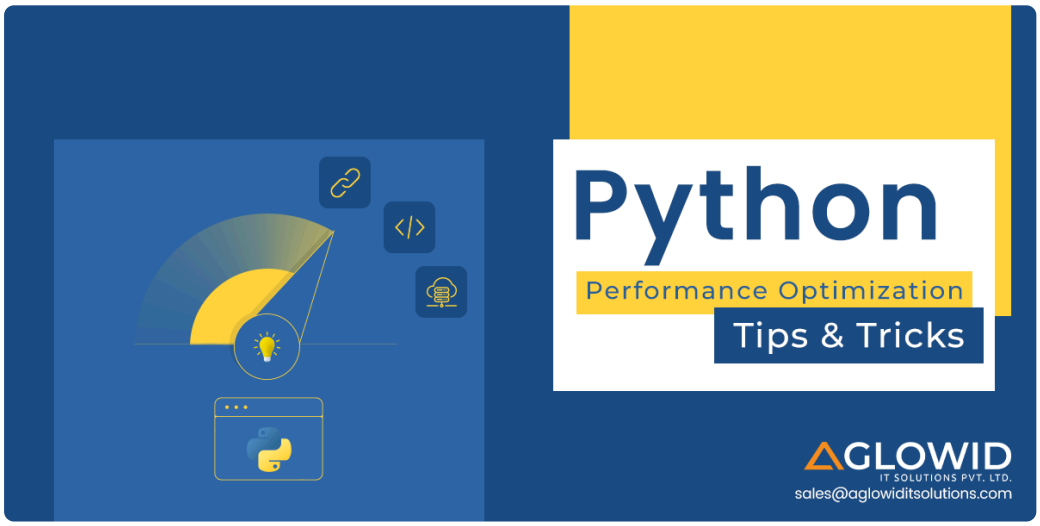
AGLOWID

Python Optimization: Performance, Tips & Tricks in 2024

SAURABH BAROT
4 May 2023



Quick Summary:

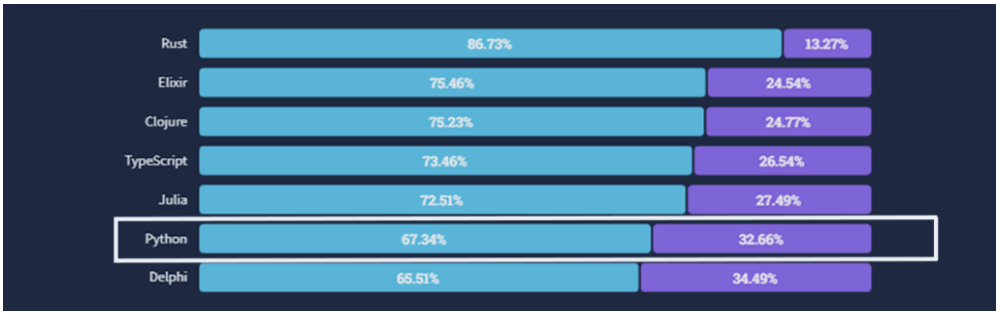
Python optimization is locating and fixing frequent performance problems that can make your code run slowly. Several elements, including improper use of external libraries, memory leaks, garbage collection costs, and inefficient algorithms and data structures, may cause these problems. The effectiveness and output of your Python code can be significantly increased by comprehending and tackling these problems.

AGLOWID

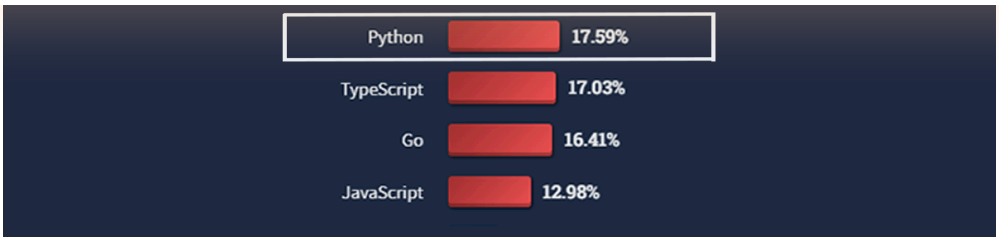
1991. Python has become a universal programming language for data analysis, machine learning, web development, and system scripts and utilities. One of the best features of this language is how easy it is to learn and how consistent it is across a wide range of applications. Although it is easy to understand, it is essential to remember that it doesn't mean it's trouble-free.

The easy-to-remember keywords and syntax require an additional pre-processing level, and this pre-processing adds a tremendous amount of load on the overall throughput of the compiler/interpreter. If this is not taken care of properly, you can experience performance lag in your Python applications. Python performance testing is an important skill. The popularity of Python is increasingly astounding, as evidenced by the following statistics:

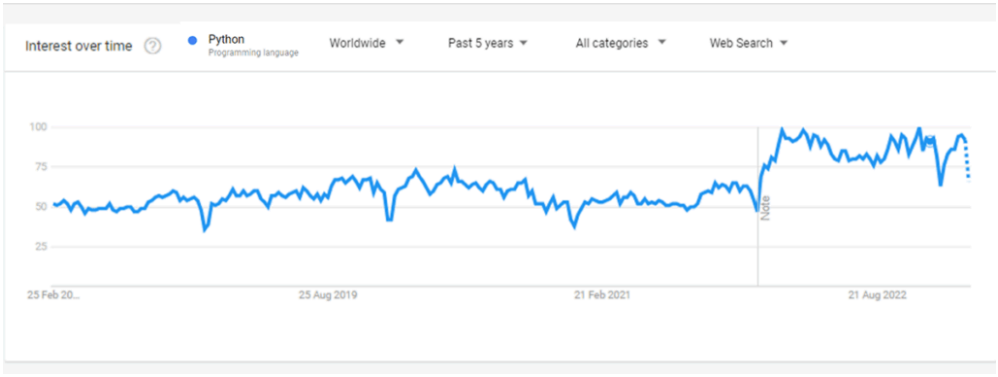
Stack Overflow – In the Stack Overflow Survey 2022, 67.34% of 71,467 respondents voted that they 'Loved' Python while '32.66%' responded dreaded.



When the same group of respondents were asked which technology, they aren't using right now but want to develop in; 17.59% of the respondents voted with Python, making it the second most popular and wanted framework right below Rust.



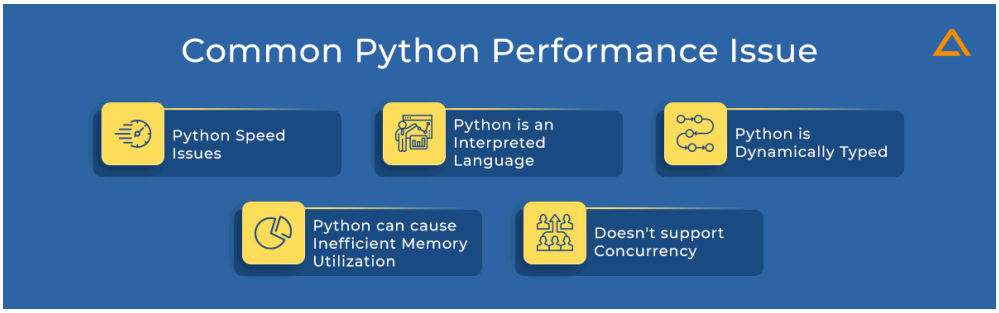
Google Trends: As we can see on Google Trends there has been a constant rise in popularity of Python with sudden boost in the overall interest over time around December 2022 to February 2023.



Now that we know the popularity of Python let's look at the issues faced while working with python. Common Python Performance Issues

Common Python Performance Issue

Before we get into the python optimization tips and tricks, it's critical to look into the reasons that cause Python to be slower than its competitors.



Many hardware features were created with the hardware available at the time in mind. Adapting the language design to accommodate advancements in hardware technologies is a time-consuming undertaking. Although python applications can already be torn down and rebuilt, making changes can improve its core design, making it compatible with future Python versions.

2. Python is an interpreted Language

Python is frequently interpreted, which is one of the most significant differences between it and other languages. Interpreted languages are slower than compiled languages because each command requires more machine instructions to execute than their compiler counterparts. In compiled languages, executables are created ahead of time and usually contain byte code or its equivalent, requiring no further compilation to run the program.

Python, on the other hand, necessitates constant interpretation. When .pyc files are compiled, there is some byte code caching that occurs. However, because java and.net employ Just-In-Time-Complier, there are no mah for these languages (JIT). PyPy was designed with JIT in mind, and the benefits are clear: it runs 4-5 times quicker than regular CPython.

3. Python is Dynamically Typed

One of the best features of Python is that you don't have to define the type of a variable each time you declare it. This is a relatively popular Python code type:

```
Test = 0
Test = "bear"
```

However, such a proclamation will have a significant impact on the language's performance. The interpreter in dynamically typed languages like Python has no idea what kind of variables are defined when the program is run. This means that more work will be required to identify the type of data stored in a variable in Python before it can be used in a statement. For example, suppose we had such a piece in C:

```
int a = 0;
int b = 1;
int sum = a + b;
```

In Python, we have an equivalent:

```
a = 0
b = 0
sum = a + b
```

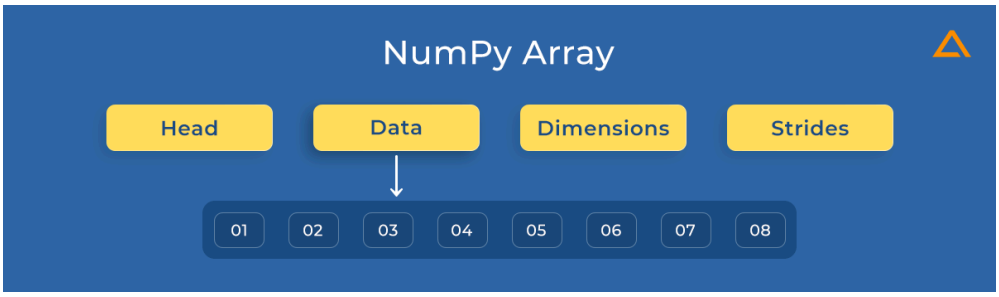
The C code would consistently outperform the Python version in terms of performance. This rationale is that when a variable's data type is explicitly declared to the runtime environments, it can easily apply optimizations related to various data types to increase efficiency. Furthermore, when a variable is accessed, the runtime is not necessary to evaluate the type of data contained in it. This information is already known to it, which eliminates a large number of otherwise duplicate actions. This means that many extra steps are necessary when addressing such scenarios in dynamically written languages like Python, which reduces the environment's efficiency.

4. Python can cause Inefficient memory utilization

Because Python objects lack correct types, performing simple activities necessitates performing several redundant

than traditional arrays; a list object in Python contains a reference to a contiguous array of references in memory. These links will take you to the actual data. This can lead to a tangled situation, similar to this:

NumPy Arrays, which are objects built around primitive C arrays, were created to combat this. As a result, their structure is essential, and changing and accessing data is a breeze.



5. Doesn't support Concurrency

This isn't so much a setback as it is an area where Python could have improved. Most programming languages allow you to take advantage of the several processing cores that current machines come equipped with. While multi-core processing is complex, it does provide significant performance gains. It also provides for better load balancing in the case of high-performance jobs, resulting in a more pleasant user experience.

brings. Situations like race conditions, deadlock, and livelock, if not managed effectively, can render a functioning application unusable in seconds. To prevent this, the concept of locks is employed. A lock is used to prevent multiple threads from accessing sensitive variables at the same time.

The CPython runtime, on the other hand, implements a lock called the GIL (Global Interpreter Lock) that prevents any two threads from operating at the same time. In systems with single-core CPUs, this usually makes little difference. However, it negates the advantage of having a multi-core processor because only one code may be used at any given moment. One thing to remember is that there is always space for improvement.

Now that we've learned about the most typical **python optimization** concerns, we can go on to the next step. Let's look at how Python performance can be measured.

> ***Also Read: –*** *Python Best Practices to Follow in 2024*

Common Python Optimization Bottleneck's

It's vital to recognize potential bottlenecks while optimizing Python code. These blockheads might make your software run more slowly because they can consume a lot of time and resources. The following are some examples of common bottlenecks in Python:

AGLOWID

Inefficient Algorithms & Data Structure

Selecting the appropriate algorithm and data structure can significantly impact your code's performance. For instance, a linear search algorithm may need to be more active when used to search through an extensive dataset for a given value, whereas a binary search method may be substantially faster in this situation. Likewise, utilizing a list instead of a set or a dictionary may be slower if you need to access items in a precise order. It's crucial to comprehend how various algorithms and data structures perform so that you may select the most suited to the task at hand.

Memory Leaks

The performance may gradually suffer due to memory leaks, which happen when the code does not free up memory that is no longer needed. The program's memory use will rise over time, for instance, if a large object is created or deleted, and its memory space is not freed. Ensuring that every object is correctly removed and that whatever memory it occupies is freed up is crucial to prevent memory leaks.

Garbage Collection Overhead

Garbage collection is the process of releasing memory that is no longer used by the program. However, if the garbage collection process is not optimized, it can add significant overhead to the program. For example, if the garbage

it works and how to configure it to suit the program's needs.

Optimal Use of External Libraries

External libraries can significantly speed up your code, but if they are not used properly, they can also cause it to run more slowly. The code can be considerably slowed down by, for instance, loading an extensive library that is not required for the task. Selecting the appropriate libraries for the job and simply utilizing the necessary functionalities are crucial. It's critical to comprehend the libraries' performance traits and pick the ones most suited to the job.

Methods to test python performance

If you've never done performance testing before, you might be unsure where to begin. Typically, we'll capture a timestamp of the code before and after it runs and when we run our code snippet. Let's look at a few approaches for evaluating the performance of python code.

Here it is divided into 2 parts:

1. Performance Testing by Brute Force
2. Performance testing using libraries & Profilers.

Performance Testing with Brute Force

You may use the DateTime module in Python to accomplish this:

AGLOWID

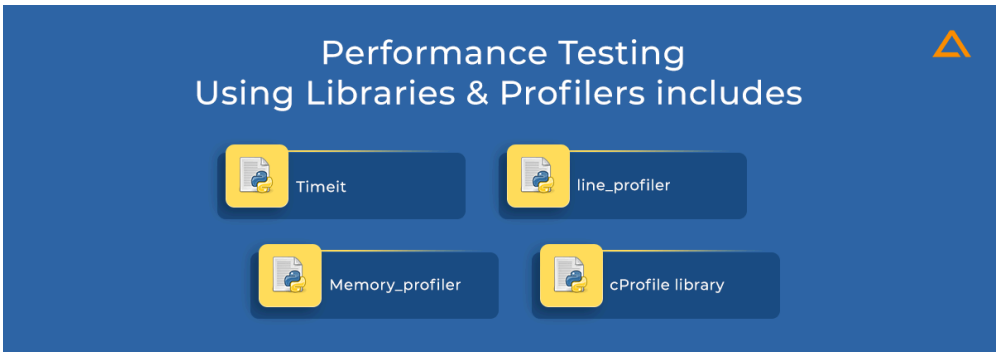
```
# insert code snippet here
end_time = datetime.datetime.now()
print(end_time - start_time)
```

This solution, of course, leaves a lot to be desired. It merely gives us one data point, for example. We'd like to run it a few times to get an average or lower bound.

Performance Testing using Libraries & Profilers

Libraries included are :

1. **timeit**
2. **line_profiler**
3. **memory_profiler**
4. **cProfile library**



1. timeit Library

You have the option of having all of the time crap abstracted away with the addition of perks. Take a look at the timeit library. The timit library has two main ways to test code: command line and inline.

To test code that uses the timeit library, you must use either the timeit or the repeat methods. Both are fine, but the repeat feature gives you a bit more control.

In this example, we're creating a list of pairs from two tuples. To test it, we may use the timeit function:

```
import timeit
timeit.timeit("[(a, b) for a in (1, 3, 5) for b in (2, 4
```

If everything goes as planned, this snippet will execute a million times and report an average execution time. You can, of course, alter the inputs and the number of iterations.

```
import timeit
timeit.timeit("[(a, b) for a in (1, 3, 5) for b in (2, 4
```

Using the repeat function, the process is repeated numerous times.

```
import timeit
timeit.repeat("[(a, b) for a in (1, 3, 5) for b in (2, 4
```

The function produces a list of execution times rather than an execution time. The list will contain three different execution times in this situation. We don't need all of those times; all we need is the shortest execution time to determine the snippet's lower bound.

```
import timeit
min(timeit.repeat("[(a, b) for a in (1, 3, 5) for b in (
```

2. line_profiler

The second library we'll look at is called line profiler, and it has a little different application than the others. You can get

profiler module. If you're having difficulties limiting down slow routines or if a third party calls the larger file, this is quite handy.

Instead of going through line after line of dense code, seeing the time spent on each line allows you to quickly spot the faults.

The normal usage of line profiler can be a little puzzling at first, but after a few uses, it becomes second nature. You must add the @profile decorator to each function in order to profile it. For a better understanding, consider the following example.

```
#!/usr/bin/env python3
# test.py
import time
@profile
def long_function():
    print('function start')
    time.sleep(5)
    print('function end')
long_function()
```

Isn't it straightforward? That's because you don't need to import anything or change your code to use a line profiler; all you have to do is add the decorator. You must do two things outside of code to use line profiler:

```
kernprof -l test.py
python -m line_profiler test.py.lprof
```

Line profiler will be run on your file and a separate. lprof file will be generated in the same directory as the first command. The results of this.lprof file can generate a report using the module itself in the second command. Let's have a look at the second command's output:

AGLOWID

```
File: test.py
Function: long_function at line 6

Line #      Hits         Time  Per Hit   % TimeLine Cont
==============================================================

6                                           @profile
7                                           def long_fun
8         1         15.0     15.0      0.0      print('f
9         1    5004679.0 5004679.0    100.0       time.sl
10        1         21.0     21.0      0.0      print('
```

The statistics for each line of the profiled functions are listed. Because we spend so much time in long function sleeping, which consumes nearly all of the execution time, you'll be able to quickly establish where you should focus your efforts, whether it's refactoring or speeding up the slow jobs.

3. Memory_profile

Memory profiler is similar to line profiler, except it concentrates on producing statistics about memory utilization. When you run a memory profiler on your code, it gives you a line-by-line breakdown, focusing on total and incremental memory usage by line.

We'll use the same decorator structure to test our code as we did in the line profiler. Here's a modified version of the sample code.

```
#!/usr/bin/env python3
# test.py
@profile
def long_function():
    data = []
```

AGLOWID

```
    return data
long_function()
```

In the preceding example, we establish a test list and populate it with many integers. This gradually expands the list, allowing us to track the increase in memory usage over time. Run the following command to see the memory profiler report:

```
python -m memory_profiler test.py
```

This should generate the following report, which contains memory statistics on a line-by-line basis:

```
Filename: tat.py

Line #    Mem usage    Increment  Occurences   Line Cont

============================================================

3   38.207 MiB   38.207 MiB           1   @profile

4                                         def long_funct

5   38.207 MiB    0.000 MiB           1       data = []

6   41.934 MiB    2.695 MiB      100001       for i in r

7   41.934 MiB    1.031 MiB      100000           data.c

8   41.934 MiB    0.000 MiB           1       return dat
```

As you can see, our function uses roughly 38MB of memory at first and then climbs to 41.9MB once our list is filled. Although the resource library provides memory consumption metrics, it does not provide a thorough line-by-line breakdown like a memory profiler. This is the way to go if you're looking for a memory leak or dealing with a particularly fat application.

AGLOWID

to timeit and brute force. We can use cProfile to collect runtime statistics from a portion of code, just like we can with timeit. cProfile, on the other hand, is a lot more thorough. For instance, we can use the same list comprehension as before.

```
import cProfile
cProfile.run("[(a, b) for a in (1, 3, 5) for b in (2, 4,
```

As a result of output, you can get a report that looks like this.

```
4 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:line

1    0.000    0.000    0.000    0.000 <string>:1(<listcc

1    0.000    0.000    0.000    0.000 <string>:1(<module

1    0.000    0.000    0.000    0.000 {built-in method b

1    0.000    0.000    0.000    0.000 {method 'disable'
```

We obtain a fabulous table with a lot of helpful information in this section. Each column decomposes a different runtime segment, and each row denotes a function that was executed. The listcomp> function, for example, was called once (ncalls) and took 0.000 seconds (tottime), ignoring subfunction calls. Check out the following analysis of all six columns to understand everything else in this table:

ncalls

The number of times a specific method was invoked. This number can be expressed as a fraction (e.g., 3/1). The first value represents the total number of calls, and the second value represents the number of primitive calls (not recursive)

Tottime

Total time spent performing the function, excluding calls to subfunctions

Percall

The ratio of tottime to ncalls is called per call (first) (i.e., the average amount of time spent in this function excluding subfunctions)

Cumtime

Includes call to subfunctions, the total amount of time the function spent executing

Percall

The proportion of cumtime calls to primitive calls (i.e., the average amount of time spent in this function)

Filename:lineno(function)

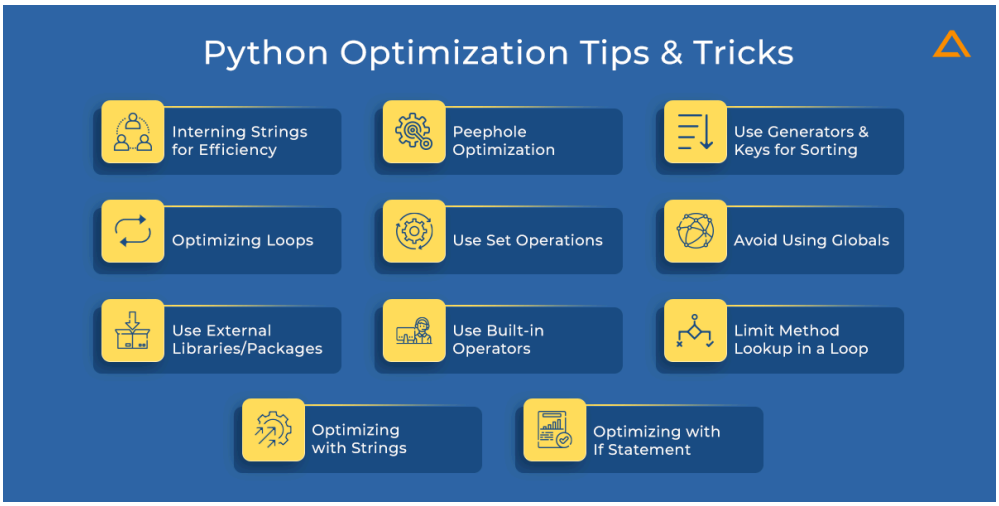
The requested filename, line number, and function

this is more of a supplement than a replacement for timeit. cProfile would be ideal for substantial profiling scripts. That way, you'll be able to see which functions need to be improved.

> ***Also Read: –*** *Python Vs Java*

Python Optimization Tips & Tricks

These tips and tricks for **python code performance optimization** lie within the realm of python. The following is the list of **python performance tips.**



1. Interning Strings for Efficiency

Interning a string is a technique for storing only one copy of each unique string. We can also have the Python interpreter reuse strings by modifying our code to cause string interning. When we construct a string object, the python interpreter usually decides whether or not the string should be cached. The interpreter's underlying essence comes out in particular circumstances, such as when processing identifiers.

underscores, and integers causes Python to intern the string and generate a hash for it. Python contains a lot of internal code that uses dictionaries, which causes it to execute a lot of identifier searches. As a result, interning the identification strings accelerates the entire procedure. Simply said, Python stores all identifiers in a table and created unique keys (hashes) for each item for future lookups. This optimization happens throughout the compilation process. It also includes the intertwining of string literals that seem like identifiers.

As a result, it's a beneficial feature in Python that you may take advantage of. This type of functionality can aid in processing massive text mining or analytics applications because they necessitate regular searches and message flip-flopping for bookkeeping.

The auto-interning in Python does not include strings read from a file or received through network communication. Instead, you can delegate this work to the intern() function responsible for handling such situations.

2. Peephole Optimization

Peephole optimization is a method that optimizes a small segment of instructions from a program or a section of the program. This segment is then known as <Peephole> or <windows>. It helps in spotting the instructions that you can replace with a minified version.

With the example below, we can get comprehensive knowledge of Python deals with peephole optimization.

Example 1:

A function in the example initializes two of its members. One of them is a string, and the other is a number. Following that,

will remain as constants in your memory. Please see the graphic below for further information.

You can notice that we used the constant. code .co consts> in the accompanying screenshot. It's one of three tuples that every Python function object holds. In Python, a function is also an object. It is made up of the three tuples listed below.

1. `The <__code__.co_varnames>:` Holds local variables including parameters.
2. `The <__code__.co_names>:` Stores global literals.
3. `The <__code__.co_consts>:` References to all the constants.

Example 2:

We're using the "in" operator to find a specific element in a set in this example. Python will recognize that the set is being used to validate an element's membership. As a result, regardless of the size of the set, it will consider the instructions as a constant cost operation. And it will handle them faster than a tuple or a list would. In Python, this is referred to as membership testing. Please see the screenshot attached.

3. Use Generators and Keys for Sorting

Generators are a fantastic way to save space in your memory. They make it easier to write functions that return one thing at a time (the iterator) rather than all at once. When you're making a long list of numbers and adding them all up, this is an excellent illustration.

You should also use keys and the default sort() function as feasible when sorting items in a list. Check that the list is sorted according to the index specified in the critical

AGLOWID

```
import operator

test = [(11, 52, 83), (61, 20, 40), (93, 72, 51)]
print("Before sorting:", test)

test.sort(key=operator.itemgetter(0))
print("After sorting[1]: ", test)

test.sort(key=operator.itemgetter(1))
print("After sorting[2]: ", test)

test.sort(key=operator.itemgetter(2))
print("After sorting[3]: ", test)
```

Output

```
Before sorting: [(11, 52, 83), (61, 20, 40), (93, 72, 51

After sorting[1]:  [(11, 52, 83), (61, 20, 40), (93, 72,

After sorting[2]:  [(61, 20, 40), (11, 52, 83), (93, 72,

After sorting[3]:  [(61, 20, 40), (93, 72, 51), (11, 52,
```

4. Optimizing Loops

Most programming languages emphasize the need for optimizing loops. We do have a mechanism to make loops run faster in Python. Consider the way of prohibiting the usage of dots within a loop, which many programmers overlook.

There are a few looping-supporting building components in Python. The employment of a “for” loop is common among these few. While you may enjoy employing loops, they do come at a price. The Python interpreter expends a lot of time

Following then, the level of code optimization is determined by your understanding of Python's built-in capabilities. We'll try to illustrate how different structures can aid in loop optimization in the instances below.

4.1. Examples for Optimizing a Loop in Python

Example:

Consider a function that utilizes a for loop to update a list of Zipcodes, trims the trailing spaces, and updates the Zipcodes list.

```
newZipcodes = []
for zipcode in oldZipcodes:
newZipcodes.append(zipcode.strip())
```

Look at how you can use the map object to convert the above into a single line. It'll also be less expensive now.

```
newZipcodes = map(str.strip, oldZipcodes)
```

List comprehensions can even be used to make the syntax more linear.

```
Zipcodes += [iter.strip() for iter in newZipcodes]
```

Finally, converting the for loop into a generator expression is the quickest method.

```
itertools.chain(Zipcodes, (iter.strip() for iter in
newZipcodes))
```

4.2. Let's See What have We Optimized?

As previously stated, the fastest technique to optimize the for loop in the given use case is to use generator expression (and in general). We've combined the code from four

AGLOWID

```
import timeit
import itertools

Zipcodes = ['121212','232323','434334']
newZipcodes = ['  131313 ',' 242424   ',' 212121 ','  32

def updateZips(newZipcodes, Zipcodes):
    for zipcode in newZipcodes:
        Zipcodes.append(zipcode.strip())

def updateZipsWithMap(newZipcodes, Zipcodes):
    Zipcodes += map(str.strip, newZipcodes)

def updateZipsWithListCom(newZipcodes, Zipcodes):
    Zipcodes += [iter.strip() for iter in newZipcodes]

def updateZipsWithGenExp(newZipcodes, Zipcodes):
    return itertools.chain(Zipcodes, (iter.strip() for i

print('updateZips() Time            : ' + str(timeit.tim

Zipcodes = ['121212','232323','434334']
print('updateZipsWithMap() Time     : ' + str(timeit.tim

Zipcodes = ['121212','232323','434334']
print('updateZipsWithListCom() Time : ' + str(timeit.tim

Zipcodes = ['121212','232323','434334']
print('updateZipsWithGenExp() Time  : ' + str(timeit.tim
```

Output:

```
updateZips() Time            : 1.525283

updateZipsWithMap() Time     : 1.4145331

updateZipsWithListCom() Time : 1.4271637

updateZipsWithGenExp() Time  : 0.6092696999999996
```

Python manages sets through hash tables. When we add an element to a set, the Python interpreter uses the hash of the target element to determine its location in the RAM allotted for the set.

Because Python resizes the hash table automatically, the speed remains constant (O(1)) regardless of the size of the set. This is what allows the set operations to run more quickly.

Union, intersection, and difference are examples of set operations in Python. As a result, you can try incorporating them into your code when appropriate. These are frequently faster than going through the lists one by one.

```
Syntax       Operation    Description

------       ---------    -----------

set(l1)|set(l2) Union      Set with all l1 and l2 items.

set(l1)&set(l2) Intersection Set with commmon l1 and l2

set(l1)-set(l2) Difference   Set with l1 items not in l2
```

6. Avoid Using Globals

Excessive or haphazard use of globals is frowned upon in practically all programming languages. The reason for this is that they may have unintended consequences that result in the Spaghetti code. Furthermore, Python is relatively slow when it comes to accessing foreign variables.

It does, however, allow for limited use of global variables. The global keyword can be used to declare an external variable. Also, before utilizing them inside loops, make a local duplicate.

AGLOWID

Some Python libraries have a "C" equivalent that has the same functionality as the original. They run quicker since they are written in "C." Try using cPickle instead of a pickle as an example.

Then you can use Cython>, which is a static optimizing compiler for both Python and C++. It's a Python superset that adds support for C functions and types. It instructs the compiler to generate code that is both quick and efficient.

Consider utilizing the PyPy package as well. It comes with a JIT (Just-in-time) compiler that allows Python code to dash. You may even adjust it to give it a boost in processing power.

8. Use Built-in Operators

Python is a high-level abstraction-based interpreted language. As a result, whenever possible, you should use the built-ins. Because the built-ins are precompiled and quick, it will improve the efficiency of your code. Long iterations with interpreted steps, on the other hand, become pretty sluggish. Similarly, use built-in tools such as the map, which boost speed significantly.

9. Limit Method Lookup in a Loop

following example, you will be able to grasp the concept.

```
>>> for it in xrange(10000):
>>>     myLib.findMe(it)
>>>findMe = myLib.findMe
>>> for it in xrange(10000):
>>>     findMe(it)
```

10. Optimizing with Strings

Concatenating strings is slow, therefore don't do it inside a loop. Instead, use the join method in Python. Alternatively, you can utilize the formatting tool to create a unified string.

RegEx operations in Python are quick because they are delegated to C code. Basic string techniques such as isalpha(), isdigit(), startswith(), and endswith() perform better in some instances.

The <timeit> module can also be used to test different ways. It will assist you in determining which strategy is the most efficient.

11. Optimizing with If Statement

Python, like most programming languages, supports lazy-if evaluation. It indicates that if there are many 'AND' conditions, not all of them will **be tested if one fails.**

1. You can modify your code to take advantage of Python's behavior. For example, if you're looking for a specific pattern in a list, you can narrow the scope by adding the following condition.Add a 'AND' condition that returns false if the target string's length is smaller than the

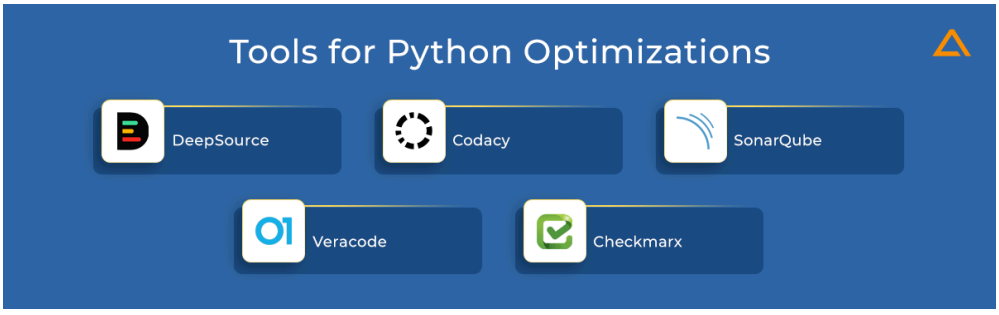
"string should end with a dot."

2. Instead of using `<if done!= None>`, you can test a condition like `< if done is None>`.

> ***Also Read: –*** *Ruby vs Python*

Tools for Python Optimization

As we have established already, Python is a robust, high-level general-purpose programming language with many use cases and potential benefits. There are many market-tested and popular Python Optimization tools that can be leveraged for truly bringing out the full potential of Python programming and reducing development time while improving the efficiency and accuracy of your Python projects. Here is a curated list of top 5 Python performance optimization tools you can use:



1. DeepSource

DeepSource analyses static code in various programming languages, including Python, Javascript, Golang, and others. DeepSource offers flexibility as well as the ability to do static analysis on Python code with simplicity. DeepSource creates

AGLOWID

Some of the Features of DeepSource include:

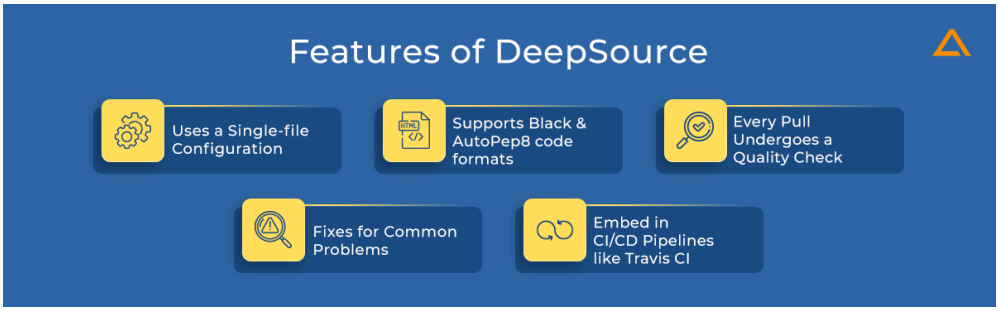
For continuous analysis, use a single-file configuration.

Code formats such as Black and AutoPep8 are supported.

Every pull undergoes a quality check.

Fixes for common problems that are automated

To test coverage, embed in CI/CD pipelines like Travis CI.



Compared to other static code analysis tools, DeepSource has a low false-positive rate and a quick resolution time. DeepSource makes it simple for maintainers to assess framework-related issues by giving them access to them.

Aside from that, DeepSource makes it simple to work with private repositories. A private token is used to fetch the code with each pull request or commit. After that, the analysis is carried out in a separate environment. The analysis is then performed in a separate context. The codebase is purged once the analysis is finished, reducing the chance of security breaches.

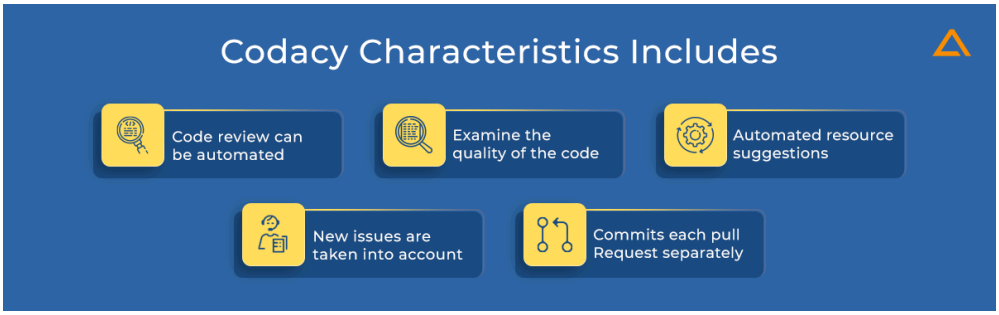
2. Codacy

Codacy is another tool that delivers code coverage and review reports for various general-purpose programming

complexity. It aids developers in maintaining code quality and a clean code review.

Codacy has several characteristics, including:

- Code review can be automated.
- Examine the quality of the code over time.
- Developers will benefit from automated resource suggestions.
- To avoid noise, only new issues are taken into account.
- Analyzes and commits each pull request separately.



The following are some of the disadvantages:

- There is a lack of issue priority, which should assist developers in focusing their efforts.
- There is no way to export code patterns.
- It’s challenging to create setup pages.

Codacy, on the other hand, is a complicated setup that necessitates a lot of settings and has a high false-positive rate.

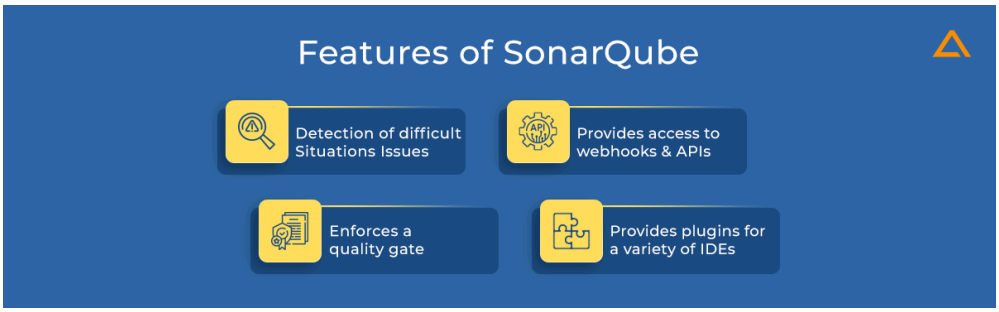
AGLOWID

SonarQube performs automatic reviews by continuously inspecting code quality. Its Static Code Analysis Tool may detect Python defects, anti-patterns, and even security flaws. For effective code quality management, SonarQube is extremely simple to integrate with **a CI/CD pipeline.**

Two of SonarQube's tools are used to implement the Code Analysis feature. The Sonar Scanner enables the analysis to be carried out, with the results being handled and saved on the SonarQube Server.

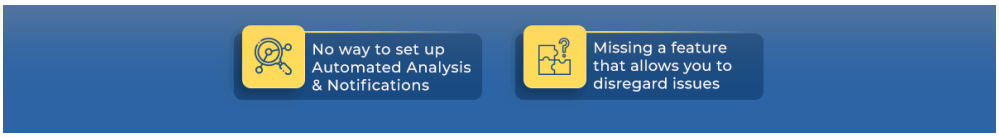
Features of SonarQube:

- Detection of difficult situations Issues such as security flaws and defects in execution paths.
- To automate the code review process, it provides access to webhooks and APIs.
- Enforces a quality gate following the criteria and procedures.
- Provides plugins for a variety of popular IDEs, reducing the need for the complete package.



Some drawbacks include:

- There is no way to set up automated analysis and notifications.
- It is missing a feature that allows you to disregard issues that should not be solved.



It's challenging to set up SonarQube for a Python project because it necessitates the installation of packages and plugins to set up client analysis and server storage. To learn more about SonarQube settings for a Python project, consult the official documentation.

4. Veracode

Veracode is another popular Python code review tool. It not only scans for typical vulnerabilities and exposures, but it can also uncover flaws using static analysis, making it simple to report bugs and anti-patterns. Veracode also offers other services through its enterprise service, such as interactive and dynamic **analysis.**

The following are some of the essential features:

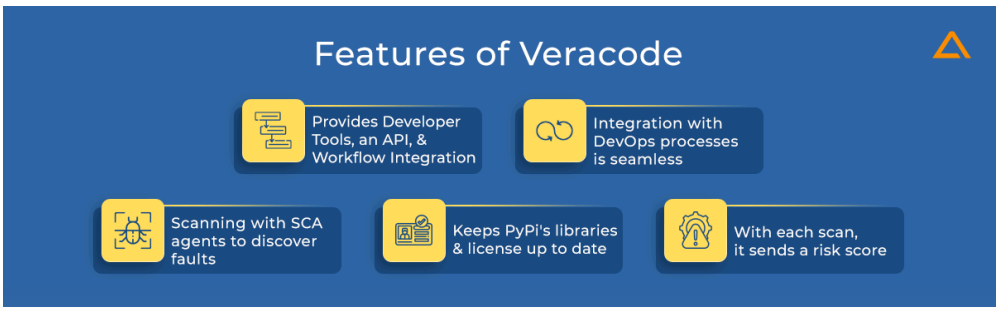
To make code quality checks easier, it provides developer tools, an API, and workflow integration.

Integration with DevOps processes is seamless.

Scanning with SCA agents to discover faults and vulnerabilities.

Keeps PyPi's libraries and licen**s**e up to date.

With each scan, it sends a risk score.



AGLOWID

It is not as simple as it appears. Setups for Continuous Integration

There is a lack of a user experience that is intuitive.

5. Checkmarx

Checkmarx is a static code analysis and application security testing tool. It includes capabilities such as static application testing, runtime testing, interactive testing, and dependency scanning, which allows for quick scanning of source code and the elimination of flaws.

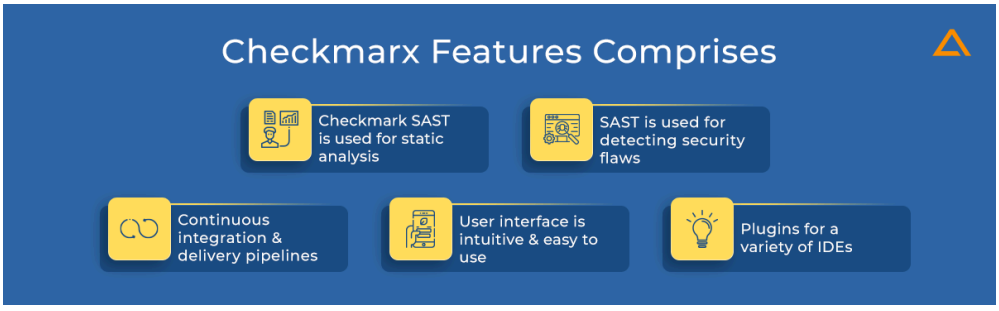
The following are some of the essential features:

Checkmarx SAST is used for static analysis and detecting security flaws.

Integrates with continuous integration and delivery pipelines.

The user interface is intuitive and straightforward to use.

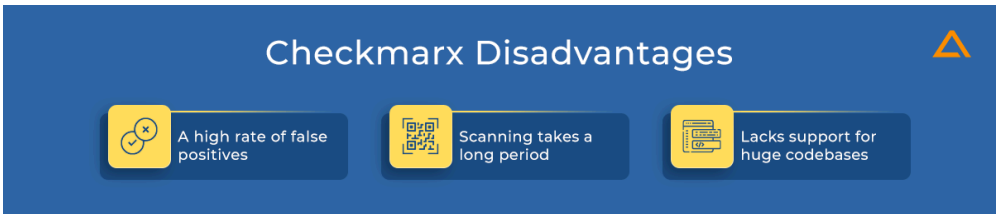
Plugins for a variety of popular IDEs are available.



The following are some of the disadvantages:

integration.

The best thing about Checkmarx is that it comes with built-in support for most general-purpose programming languages. Checkmarx, on the other hand, has problems with false positives and a lack of support for huge codebases.



Wrapping up!

We hope the Python performance optimization Tips and Tricks given in this article can help you build faster Python applications. These python optimization tips will help you run Python code more reliably at numerous levels of granularity, from profiling to data structures to string concatenation and memory optimization with the xrange function. These points were made to assist Python programmers in their day-to-day programming tasks and assist them in writing high-quality code.

AGLOWID

Check:

> *Why Digital Transformation is Important for Small Businesses?*
>
> *App Security – Vulnerability, Best Practices, Testing Tools & Checklist*
>
> *Python Sentiment Analysis Libraries*
>
> *IaaS Vs PaaS Vs SaaS – Detailed Comparison of Cloud Technologies*
>
> *How to Secure Microservices Architecture?*

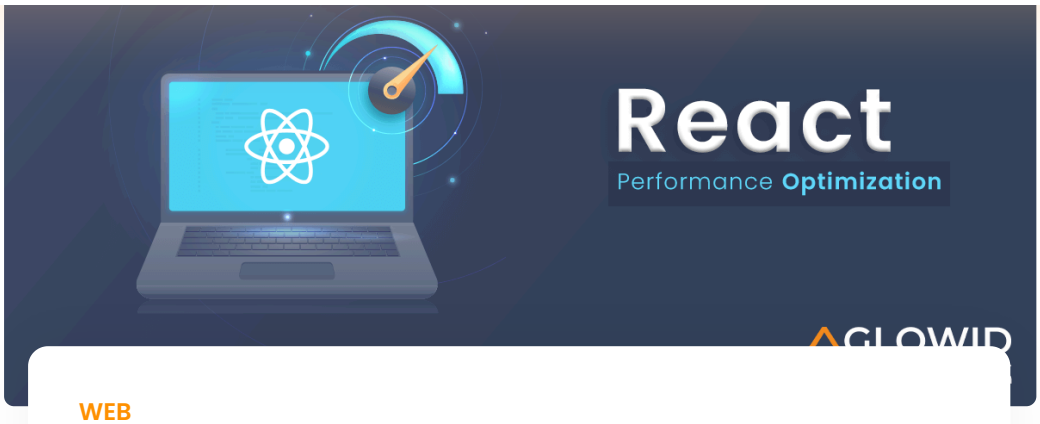
SAURABH BAROT

Saurabh Barot, the CTO of Aglowid IT Solutions, leads a team of 50+ IT experts across various domains. He excels in web, mobile, IoT, AI/ML, and emerging tech. Saurabh's technical prowess is underscored by his contributions to Agile, Scrum, and Sprint-based milestones. His guidance

AGLOWID

RELATED POSTS

AGLOWID



WEB

Top React Performance Optimization Tips in 2024

By Saurabh Barot

WEB

Top Laravel Packages to Use in 2024

By Saurabh Barot

AGLOWID

WEB

React Router : A Complete Guide

By Saurabh Barot

AGLOWID

TESTIMONIALS

Our Slam Book

Tony Lehti

DIRECTOR -

Spain

Very professional, accurate and efficient team despite a
working with them

TALK TO US

Let’s Get In Touch

AGLOWID

Contact number

+91 8487981277

+1 770 796 0077

Email Address

sales@aglowiditsolutions.com

Say Hello

Tell us about your project

Name*

Email*

Phone*

Message*

By sending this form I confirm that I have read and accept the Privacy Policy

AGLOWID

Media Coverage

AGLOWID
IT SOLUTIONS PVT. LTD.

Aglowid is a leading IT Development Strategy and Consulting Company serving clientele from USA, UK, Australia, Canada, UK, Singapore etc. With a team of 50+ developers in fields like web, mobile, cloud, AI/ML, and other cognitive technologies, we aim to deliver highly performant, secure, future-ready IT solutions to our clients.

USA

1000 Whitlock Ave NW, Marietta, Georgia - 30064

AGLOWID

INDIA

501, City Center, Science City Rd, Ahmedabad - 380060

FOLLOW US ON

Quick Links

- Careers
- Life @Aglowid
- Blog
- Contact Us
- Brochure
- Partner with us

Services

- Mobile App Services
- Web Services
- IT Staff Augmentation
- IT Strategy & Consulting
- Set up an ODC
- Digital Transformation
- Enterprise Mobility
- Dedicated Development Team

Hire Developers

- Hire Angular Developers
- Hire Ruby On Rails Developers
- Hire NodeJS Developers
- Hire ReactJS Developers
- Hire React Native Developers
- Hire iOS Developers
- Hire Android App Developers
- Hire Flutter Developers

AGLOWID

Contact number

+91 9016227777

Email Address

hr@aglowiditsolutions.com