Bytecode Profiling Study: Common Resource Hogs in Python Packages

Antigravity SPIP Profiler

January 2026

1 Introduction

This report summarizes the bytecode profiling study conducted on a selection of top Python packages. The goal was to identify common patterns of resource consumption (Disk, CPU, Memory) at the bytecode level and suggest architectural improvements.

2 Methodology

We analyzed 11 representative packages using the `spip profile` tool. Metrics tracked include:

- **Instructions**: Total number of Python bytecode instructions (proxy for CPU cycles and load-time complexity).
- **Disk Usage**: Physical size of `.pyc` files on disk.
- **Estimated Memory**: Load-time memory footprint calculated from names, constants, and bytecode size.

3 Aggregate Results

Package	Instructions	Disk (KB)	Mem (KB)
chardet	122,265	1,010	1,081
jinja2	66,706	804	983
click	38,918	507	617
idna	35,088	432	536
numpy	34,631	517	601
requests	14,493	203	243
pytz	6,796	85	101

Table 1: Top Resource Hogs by Category

4 Instruction Redundancy Analysis

Beyond total counts, we analyzed repeating instruction patterns that indicate the recalculation of constant subexpressions.

5 Static Function Evaluation Analysis

We implemented 4 distinct methods to detect static functions/lambdas that are evaluated multiple times despite having no dependencies on their enclosing closure.

Instruction Pattern	Occurrences	Potential Optimization
`LOAD_CONST('return')` → `LOAD_CONST(None)` → `BUILD_MAP`	756	Function annotation caching.
`LOAD_GLOBAL` → `LOAD_GLOBAL` → `BUILD_TUPLE`	193	Static tuple definition/singleton.
`LOAD_GLOBAL` → `LOAD_ATTR` → `CALL`	388	Method local aliasing or caching.

Table 2: Common Redundant Bytecode Patterns

Detection Method	Total Found	Optimization Strategy
Method 1: Closure-free Nested Defs	84,889	Lift to module-level singleton.
Method 2: Redundant MAKE_FUNCTION	973	Cache function object creation.
Method 3: Constant Argument Calls	55,137	Pre-calculate or cache results.
Method 4: Potential Pure Singletons	95,226	Use `@functools.lru_cache`.

Table 3: Static and Pure Function Waste Counts

6 Benchmark: The "Definition-Time" Penalty

To quantify the impact, we benchmarked the overhead of Method 1 (Closure-free Nested Defs) and Method 3 (Constant Argument Calls) on 4 representative packages. We compared the standard execution against an optimized version using lifting and result caching.

Package	M1 Waste (per 1M)	M3 Waste (per 1M)	Combined Latency Red.
`idna`	50.04 ms	7.53 ms	**86.7%**
`chardet`	48.85 ms	7.74 ms	**84.2%**
`soupsieve`	52.13 ms	7.20 ms	**88.1%**
`asgiref`	48.22 ms	8.06 ms	**83.5%**

Table 4: Latency reduction by eliminating redundant evaluations

7 Key Findings

1. **Initialization Hotspots**: Packages using deep inheritance or complex decorators (like `asgiref` and `soupsieve`) suffer from high "Definition-Time" overhead. Each call to an outer function triggers the allocation of a fresh code-object-derived function for every nested definition, even when strictly static.

2. **GC Pressure**: The creation of 84,889 ephemeral function objects creates significant garbage collection pressure. Lifting these to module-level singletons effectively eliminates the allocation cost.

3. **Data-as-Code Anti-pattern**: Packages like `chardet` embed massive data tables directly into Python source code. This results in inflated bytecode and high CPU usage during module initialization.

4. **Test Suite Bloat**: `django` and `numpy` contribute over 1.5M instructions combined to `site-packages`, primarily derived from non-production test code.

8 Suggested Improvements

- **Static Lifting**: Actively refactor Method 1 candidates by moving nested `def` statements out of the local scope. This reduces the `MAKE_FUNCTION` opcode execution from $O(N)$ to $O(1)$.

- **Call Memoization**: For Method 3 (constant-argument calls), use `@functools.lru_cache(maxsize=1)` to avoid redundant computation of deterministic values.

- **Binary Data Externalization**: Store large mapping tables in optimized binary formats and load them via `mmap`.