

debugx

The most comprehensive offline Python debugging snapshot tool

```
pip install debugx
```

Feature Overview | Version 0.1.0

Captures a complete snapshot of your Python runtime, host system, project, and application environment in a single command -- entirely offline.

1. Overview

debugx collects a complete debugging snapshot of the current system, Python runtime, project, and application environment in a single command, and renders it as a beautiful HTML dashboard, a GitHub-ready Markdown file, and a schema-versioned JSON document.

Everything runs 100% offline. debugx never uploads data anywhere -- there is no telemetry, no analytics, and no phone-home of any kind.

```
from debugx import capture

capture()
```

That single call writes `debug_report.html`, `debug_report.json`, and `debug_report.md` to the current directory.

Why debugx?

When a bug report says "it works on my machine," or a teammate needs to understand exactly what environment produced a stack trace, debugx answers almost every follow-up question before it's asked: Python version, installed packages, git branch and dirty state, OS/CPU/memory, Docker/Kubernetes context, network reachability, recent log lines, and the active exception -- masked for secrets, formatted for humans, generated in under two seconds.

2. Core Features

Runtime & System

- **Python runtime** -- version, executable path, virtual environment / conda detection, pip version
- **Operating system** -- OS name/release, hostname, CPU architecture, byte order, libc version
- **CPU** -- logical/physical core counts, frequency, load average, live usage percent
- **Memory** -- system total/available/used percent, current process RSS
- **Disk** -- usage of the current volume plus all mounted partitions
- **Runtime stats** -- PID, parent PID, process uptime, garbage collector stats, warnings filters
- **Threads** -- every active Python thread: name, daemon flag, alive state
- **Loaded modules** -- count and list of top-level imported modules

Project & Environment

- **Environment variables** -- full os.environ snapshot, secret-masked by default
- **Git** -- branch, commit hash, last commit message/author/date, dirty state, modified/staged/untracked files, sanitised remote URL
- **Project structure** -- pyproject.toml / requirements.txt / setup.py detection, Poetry and uv detection, declared name & version, shallow file tree
- **Installed packages** -- every installed distribution and its version
- **Logs** -- tail of recently modified log files near the project root, secret-masked

Network & Containers

- **Network** -- hostname, local IP, DNS resolution status, internet reachability -- via a bare TCP/DNS check only, no data ever sent
- **Ports** -- locally listening TCP ports (via psutil, or a bounded localhost probe as fallback)
- **Processes** -- running processes visible to the current user (via psutil)
- **Docker** -- container detection via /.dockerenv and cgroup inspection, container ID
- **Kubernetes** -- in-cluster detection, namespace, pod name, node name -- service account token presence only, never its contents

Diagnostics

- **Call stack** -- the full Python call stack at the moment capture() runs
- **Active exception** -- captures either an explicitly-passed exception or the exception being handled in an except block, with a full formatted, syntax-highlighted traceback and exception chain
- **Performance** -- per-collector and per-plugin execution time, total capture duration
- **Custom plugins** -- your own health checks (e.g. "is my database reachable?") via @plugin

3. The capture() API

Every aspect of a capture can be tuned via keyword arguments:

```
capture(  
    output=["html", "json", "markdown"],  
    include_packages=True,  
    include_git=True,  
    include_network=True,  
    include_processes=True,  
    include_ports=True,  
    include_logs=True,  
    include_runtime=True,  
    include_docker=True,  
    include_kubernetes=True,  
    include_project=True,  
    include_environment=True,  
    include_stacktrace=True,  
    include_performance=True,  
    include_threads=True,  
    mask_secrets=True,  
    output_directory="./reports",  
    compress=False,  
    verbose=False,  
)
```

Key parameters

Parameter	Default	Description
output	["html","json","markdown"]	Which report formats to write
output_directory	""	Where reports are written (created if missing)
compress	False	Also bundle reports into debug_report.zip
mask_secrets	True	Redact detected secrets everywhere
exception	None	Capture a specific exception explicitly
plugins_enabled	True	Run registered @plugin functions
verbose	False	Emit debugx's own debug logs during capture

Return value

capture() returns a CaptureResult with two attributes: snapshot (the full structured Snapshot object -- summary, collectors, plugins, performance) and files (a dict mapping format name to the path written), so results can be inspected programmatically without re-reading the JSON file.

4. Command-Line Interface

debugx ships a full Typer-based CLI, installed as the debugx command (and runnable via `python -m debugx`):

```
debugx capture          # write html + json + markdown
debugx capture --html -o ./reports
debugx capture --all --compress
debugx capture --no-network --no-processes --verbose
debugx capture --config debugx.toml

debugx report debug_report.json --html  # re-render from a saved snapshot

debugx doctor           # environment sanity checks
debugx plugins          # list discovered plugins
debugx system           # quick system summary, no files written
debugx version
```

Flags

Flag	Effect
--html / --json / --markdown / --all	Select output format(s)
--output / -o	Output directory
--compress	Bundle reports into a .zip
--verbose / -v	Verbose debugx logging
--config	Load defaults from a TOML/JSON file
--no-git / --no-network / --no-packages	Skip individual collectors
--no-logs / --no-processes	Skip individual collectors
--no-mask-secrets	Disable masking (not recommended)

5. Output Formats

HTML dashboard

A self-contained, single-file dashboard with no external requests of any kind -- no CDN scripts, no web fonts, works opened directly from disk. Features:

- Light & dark mode (follows OS preference, plus a manual toggle, persisted via localStorage)
- Sticky sidebar navigation with scroll-spy highlighting
- Live search across every collector's data
- Collapsible sections for every collector and plugin
- Syntax-highlighted, color-coded tracebacks
- Summary cards and usage meters (memory / disk) built with lightweight vanilla JS -- no chart library
- One-click copy and one-click download buttons for the JSON/Markdown payloads

Markdown report

Optimised for pasting directly into a GitHub issue: a summary table, then one collapsible <details> block per collector using native GitHub Markdown syntax.

JSON report

Machine-readable and schema-versioned (schema_version field), so tooling can depend on its shape across debugx releases. This is the exact same Snapshot object the HTML and Markdown renderers consume -- one source of truth, three views.

6. Plugin System

A plugin is any zero-argument callable returning a JSON-serialisable dict -- no base class, no interface, just a decorated function:

```
from debugx import plugin

@plugin(name="database", priority=10, timeout=2.0)
def check_database() -> dict:
    return {"status": "healthy", "latency_ms": 4.2}
```

Plugin capabilities

- **Automatic discovery** -- any @plugin-decorated function registers itself at import time
- **Entry points** -- third-party packages can register plugins via the debugx.plugins entry-point group, with zero import-time coupling to debugx
- **Metadata** -- name, version, description shown by debugx plugins
- **Priority** -- controls display/listing order
- **Timeout** -- a hung plugin stops blocking capture() once its timeout elapses
- **Isolation** -- a plugin that raises is recorded as a failed result, never crashes capture()
- **Enable / disable** -- plugins can be registered disabled by default, or toggled at runtime

7. Security & Secret Masking

Hard guarantees

- Never uploads data -- the only network activity is a bare TCP/DNS reachability check, no payload sent
- Never executes arbitrary shell commands -- git is invoked with a fixed argument list only, never shell=True
- Never requires administrator or root privileges
- Kubernetes service account tokens: presence is noted, contents are never read

Secret masking

Enabled by default (mask_secrets=True) and applied recursively across dicts, lists, dataclasses, environment variables, and arbitrary Python objects. Detected both by key name and by value pattern:

- Key names: password, secret, token, jwt, cookie, authorization, private_key, api_key, client_secret, database_url, connection_string, and more
- Value patterns: AWS access/secret keys, Google/OpenAI/Anthropic/Gemini API keys, GitHub/Slack tokens, JWTs, Bearer tokens, PEM private key blocks, and connection strings with embedded credentials

Detected values are replaced with ********* before ever being written to a report.

8. Architecture

debugx is built from a small number of composable pieces, each with one job:

Component	Responsibility
CaptureConfig	Validated, immutable configuration for one capture() call
Collector (Strategy)	One class per data source; every collector is isolated by CollectorRunner
PluginManager	Runs @plugin / entry-point plugins concurrently, with per-plugin timeouts
Snapshot	The single versioned data model every output format renders from
SnapshotFormatter	Builds the Snapshot's summary cards and performance section
Reporter	The only component that writes files to disk (HTML/JSON/Markdown/zip)

Design principles

- Total isolation: a collector or plugin can never crash capture() -- failures are recorded, not raised
- Bounded concurrency: collectors and plugins run in a ThreadPoolExecutor, keeping full snapshots under the ~2 second budget even with 20 collectors
- One config object threaded through the whole pipeline, shared by both the Python API and the CLI
- One schema, three renderers -- HTML/Markdown/JSON never drift out of sync

9. Quality, Testing & Packaging

Test suite

- 212 tests across unit, integration, CLI, and plugin test suites (pytest)
- 96% statement coverage
- Regression tests proving a broken collector never stops the rest of the capture

Code quality tooling

- black -- formatting
- ruff -- linting (security rules included via flake8-bandit's S-prefixed checks)
- mypy -- static type checking, fully typed public API
- pre-commit -- runs all of the above automatically before each commit

Packaging & CI/CD

- pyproject.toml (Hatchling build backend), MANIFEST.in, MIT LICENSE
- GitHub Actions CI: lint + type-check + test matrix across 5 Python versions x 3 operating systems
- GitHub Actions release workflow using PyPI Trusted Publishing (OIDC, no stored API tokens)
- Dependabot, issue templates (bug report / feature request), pull request template
- Full documentation set: README, architecture guide, API reference, plugin development guide, CONTRIBUTING, SECURITY policy, CODE_OF_CONDUCT, CHANGELOG, ROADMAP

Examples included

basic_capture.py, exception_capture.py, plugin_creation.py, html/markdown/json_generation.py, plus framework integrations for FastAPI, Flask, Django, Celery, Docker, and Kubernetes.