

# Package: debrief (via r-universe)

May 22, 2026

**Title** Text-Based Summaries for 'profvis' Profiling Data

**Version** 0.1.0.9000

**Description** Provides text-based summaries and analysis tools for 'profvis' profiling output. Designed for terminal workflows and artificial intelligence (AI) agent consumption, offering views including hotspot analysis, call trees, source context, caller/callee relationships, and memory allocation breakdowns.

**License** MIT + file LICENSE

**URL** <https://github.com/r-lib/debrief>, <https://r-lib.github.io/debrief/>

**BugReports** <https://github.com/r-lib/debrief/issues>

**Depends** R (>= 4.1)

**Suggests** knitr, profvis, rmarkdown, testthat (>= 3.0.0)

**Config/testthat/edition** 3

**VignetteBuilder** knitr

**Encoding** UTF-8

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.3.3

**Repository** <https://r-lib.r-universe.dev>

**Date/Publication** 2026-04-22 00:32:17 UTC

**RemoteUrl** <https://github.com/r-lib/debrief>

**RemoteRef** HEAD

**RemoteSha** ce2a45e3a5e8d6e609fbb43ce8233ff4996acfab

## Contents

|                         |   |
|-------------------------|---|
| pv_call_depth . . . . . | 2 |
| pv_call_stats . . . . . | 3 |
| pv_callees . . . . .    | 4 |
| pv_callers . . . . .    | 5 |

|                                    |    |
|------------------------------------|----|
| pv_compare . . . . .               | 5  |
| pv_compare_many . . . . .          | 6  |
| pv_debrief . . . . .               | 7  |
| pv_example . . . . .               | 8  |
| pv_file_summary . . . . .          | 9  |
| pv_flame . . . . .                 | 9  |
| pv_flame_condense . . . . .        | 10 |
| pv_focus . . . . .                 | 11 |
| pv_gc_pressure . . . . .           | 11 |
| pv_help . . . . .                  | 12 |
| pv_hot_lines . . . . .             | 13 |
| pv_hot_paths . . . . .             | 14 |
| pv_memory . . . . .                | 15 |
| pv_memory_lines . . . . .          | 15 |
| pv_print_call_depth . . . . .      | 16 |
| pv_print_call_stats . . . . .      | 17 |
| pv_print_callers_callees . . . . . | 17 |
| pv_print_compare . . . . .         | 18 |
| pv_print_compare_many . . . . .    | 19 |
| pv_print_debrief . . . . .         | 19 |
| pv_print_file_summary . . . . .    | 20 |
| pv_print_gc_pressure . . . . .     | 21 |
| pv_print_hot_lines . . . . .       | 21 |
| pv_print_hot_paths . . . . .       | 22 |
| pv_print_memory . . . . .          | 23 |
| pv_print_recursive . . . . .       | 23 |
| pv_print_suggestions . . . . .     | 24 |
| pv_recursive . . . . .             | 25 |
| pv_self_time . . . . .             | 26 |
| pv_source_context . . . . .        | 27 |
| pv_suggestions . . . . .           | 27 |
| pv_to_json . . . . .               | 28 |
| pv_to_list . . . . .               | 29 |
| pv_total_time . . . . .            | 30 |
| pv_worst_line . . . . .            | 31 |

|              |           |
|--------------|-----------|
| <b>Index</b> | <b>32</b> |
|--------------|-----------|

---

|               |                             |
|---------------|-----------------------------|
| pv_call_depth | <i>Call depth breakdown</i> |
|---------------|-----------------------------|

---

### Description

Shows time distribution across different call stack depths. Useful for understanding how deeply nested the hot code paths are.

**Usage**

```
pv_call_depth(x)
```

**Arguments**

x                    A profvis object.

**Value**

A data frame with columns:

- depth: Call stack depth (1 = top level)
- samples: Number of profiling samples at this depth
- time\_ms: Time in milliseconds
- pct: Percentage of total time
- top\_funcs: Most common functions at this depth

**Examples**

```
p <- pv_example()
pv_call_depth(p)
```

---

pv\_call\_stats                    *Call statistics summary*

---

**Description**

Shows call count, total time, self time, and time per call for each function. This is especially useful for identifying functions that are called many times (where per-call optimization or reducing call count would help).

**Usage**

```
pv_call_stats(x, n = NULL)
```

**Arguments**

x                    A profvis object.

n                    Maximum number of functions to return. If NULL, returns all.

**Value**

A data frame with columns:

- label: Function name
- calls: Estimated number of calls (based on stack appearances)
- total\_ms: Total time on call stack
- self\_ms: Time at top of stack (self time)
- child\_ms: Time in callees
- ms\_per\_call: Average milliseconds per call
- pct: Percentage of total profile time

**Examples**

```
p <- pv_example()
pv_call_stats(p)
```

---

pv\_callees

*Get callees of a function*

---

**Description**

Returns the functions that a specified function calls, based on profiling data. Shows what the target function invokes.

**Usage**

```
pv_callees(x, func)
```

**Arguments**

|      |                               |
|------|-------------------------------|
| x    | A profvis object.             |
| func | The function name to analyze. |

**Value**

A data frame with columns:

- label: Callee function name
- samples: Number of times this callee appeared
- pct: Percentage of calls to this callee

**Examples**

```
p <- pv_example()
pv_callees(p, "outer")
```

---

|            |                                  |
|------------|----------------------------------|
| pv_callers | <i>Get callers of a function</i> |
|------------|----------------------------------|

---

**Description**

Returns the functions that call a specified function, based on profiling data. Shows who invokes the target function.

**Usage**

```
pv_callers(x, func)
```

**Arguments**

|      |                               |
|------|-------------------------------|
| x    | A profvis object.             |
| func | The function name to analyze. |

**Value**

A data frame with columns:

- label: Caller function name
- samples: Number of times this caller appeared
- pct: Percentage of calls from this caller

**Examples**

```
p <- pv_example()
pv_callers(p, "inner")
```

---

|            |                                     |
|------------|-------------------------------------|
| pv_compare | <i>Compare two profvis profiles</i> |
|------------|-------------------------------------|

---

**Description**

Compares two profiling runs to show what changed. Useful for measuring the impact of optimizations.

**Usage**

```
pv_compare(before, after, n = 20)
```

**Arguments**

|        |   |
|--------|---|
| before | A profvis object (before optimization). |
| after  | A profvis object (after optimization).  |
| n      | Number of top functions to compare.     |

**Value**

A list with:

- `summary`: Overall comparison summary
- `by_function`: Function-by-function comparison
- `improved`: Functions that got faster
- `regressed`: Functions that got slower

**Examples**

```
p1 <- pv_example()
p2 <- pv_example()
pv_compare(p1, p2)
```

---

|                 |  |
|-----------------|--|
| pv_compare_many | <i>Compare multiple profvis profiles</i> |
|-----------------|--|

---

**Description**

Compares multiple profiling runs to identify the fastest. Useful for comparing different optimization approaches.

**Usage**

```
pv_compare_many(...)
```

**Arguments**

...                   Named profvis objects to compare, or a single named list of profvis objects.

**Value**

A data frame with columns:

- `name`: Profile name
- `time_ms`: Total time in milliseconds
- `samples`: Number of samples
- `vs_fastest`: How much slower than the fastest (e.g., "1.5x")
- `rank`: Rank from fastest (1) to slowest

## Examples

```
p1 <- pv_example()
p2 <- pv_example("gc")
p3 <- pv_example("recursive")
pv_compare_many(baseline = p1, gc_heavy = p2, recursive = p3)

# Or pass a named list
profiles <- list(baseline = p1, gc_heavy = p2)
pv_compare_many(profiles)
```

---

pv\_debrief

*Comprehensive profiling data*

---

## Description

Returns all profiling analysis in a single list for programmatic access. This is the primary function for AI agents and scripts that need comprehensive profiling data without printed output.

## Usage

```
pv_debrief(x, n = 10)
```

## Arguments

|   |   |
|---|---|
| x | A profvis object from <code>profvis::profvis()</code> .           |
| n | Maximum number of items to include in each category (default 10). |

## Value

A list containing:

- `total_time_ms`: Total profiling time in milliseconds
- `total_samples`: Number of profiling samples
- `interval_ms`: Sampling interval in milliseconds
- `has_source`: Whether source references are available
- `self_time`: Data frame of functions by self-time
- `total_time`: Data frame of functions by total time
- `hot_lines`: Data frame of hot source lines (or NULL if no source refs)
- `hot_paths`: Data frame of hot call paths
- `suggestions`: Data frame of optimization suggestions
- `gc_pressure`: Data frame of GC pressure analysis
- `memory`: Data frame of memory allocation by function
- `memory_lines`: Data frame of memory allocation by line (or NULL)

## Examples

```
p <- pv_example()
d <- pv_debrief(p)
names(d)
d$self_time
```

---

pv\_example

*Example profvis data*

---

## Description

Creates an example profvis object for use in examples and testing. This avoids the need to run actual profiling code in examples.

## Usage

```
pv_example(type = c("default", "no_source", "recursive", "gc"))
```

## Arguments

|      |  |
|------|--|
| type | Type of example data to create:  |
|      | <ul style="list-style-type: none"><li>• "default": A real profile captured from example code with source refs</li><li>• "no_source": A synthetic profile without source references</li><li>• "recursive": A synthetic profile with recursive function calls</li><li>• "gc": A synthetic profile with garbage collection pressure</li></ul> |

## Value

A profvis object that can be used with all debrief functions.

## Examples

```
# Get default example data
p <- pv_example()
pv_self_time(p)

# Get example with recursive calls
p_recursive <- pv_example("recursive")
pv_recursive(p_recursive)
```

---

|                 |                                |
|-----------------|--------------------------------|
| pv_file_summary | <i>File-level time summary</i> |
|-----------------|--------------------------------|

---

**Description**

Aggregates profiling time by source file. Requires source references (use `devtools::load_all()` for best results).

**Usage**

```
pv_file_summary(x)
```

**Arguments**

x                    A profvis object.

**Value**

A data frame with columns:

- filename: Source file path
- samples: Number of profiling samples
- time\_ms: Time in milliseconds
- pct: Percentage of total time

**Examples**

```
p <- pv_example()
pv_file_summary(p)
```

---

|          |                               |
|----------|-------------------------------|
| pv_flame | <i>Text-based flame graph</i> |
|----------|-------------------------------|

---

**Description**

Generates an ASCII representation of a flame graph showing the hierarchical breakdown of time spent in the call tree.

**Usage**

```
pv_flame(x, width = 70, min_pct = 2, max_depth = 15)
```

**Arguments**

|           |  |
|-----------|--|
| x         | A profvis object.                                  |
| width     | Width of the flame graph in characters.            |
| min_pct   | Minimum percentage to show (filters small slices). |
| max_depth | Maximum depth to display.                          |

**Value**

Invisibly returns a `debrief_flame` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p <- pv_example()
pv_flame(p)
```

---

pv\_flame\_condense      *Condensed flame graph*

---

**Description**

Shows a simplified, condensed view of the flame graph focusing on the hottest paths.

**Usage**

```
pv_flame_condense(x, n = 10, width = 50)
```

**Arguments**

|       |                              |
|-------|------------------------------|
| x     | A profvis object.            |
| n     | Number of hot paths to show. |
| width | Width of bars.               |

**Value**

Invisibly returns a `debrief_flame_condense` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p <- pv_example()
pv_flame_condense(p)
```

---

|          |  |
|----------|--|
| pv_focus | <i>Focused analysis of a specific function</i> |
|----------|--|

---

**Description**

Provides a comprehensive analysis of a single function including time breakdown, callers, callees, and source context if available.

**Usage**

```
pv_focus(x, func, context = 5)
```

**Arguments**

|         |   |
|---------|---|
| x       | A profvis object.                               |
| func    | The function name to analyze.                   |
| context | Number of source lines to show around hotspots. |

**Value**

Invisibly returns a `debrief_focus` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p <- pv_example()
pv_focus(p, "inner")
```

---

|                |                           |
|----------------|---------------------------|
| pv_gc_pressure | <i>Detect GC pressure</i> |
|----------------|---------------------------|

---

**Description**

Analyzes the profile to detect excessive garbage collection, which is a universal indicator of memory allocation issues in R code.

**Usage**

```
pv_gc_pressure(x, threshold = 10)
```

**Arguments**

|           |  |
|-----------|--|
| x         | A profvis object.  |
| threshold | Minimum GC percentage to report (default 10). Set lower to detect smaller GC overhead. |

## Details

GC pressure above 10% typically indicates the code is allocating and discarding memory faster than necessary. Common causes include:

- Growing vectors with `c(x, new)` instead of pre-allocation
- Building data frames row-by-row with `rbind()`
- Creating unnecessary copies of large objects
- String concatenation in loops

## Value

A data frame with columns:

- `severity`: "high" (>25%), "medium" (>15%), or "low" (>threshold%)
- `pct`: Percentage of total time spent in GC
- `time_ms`: Time spent in garbage collection
- `issue`: Short description of the problem
- `cause`: What typically causes this issue
- `actions`: Comma-separated list of things to look for

Returns an empty data frame (0 rows) if GC is below the threshold.

## Examples

```
p <- pv_example("gc")
pv_gc_pressure(p)

# More sensitive detection
pv_gc_pressure(p, threshold = 5)

# No GC pressure in default example
p2 <- pv_example()
pv_gc_pressure(p2)
```

---

pv\_help

*List available debrief functions*

---

## Description

Prints a summary of all available functions in the debrief package, organized by category. Useful for discovering the API.

## Usage

```
pv_help(category = NULL)
```

**Arguments**

`category` Optional category to filter by. If NULL, shows all categories. Options: "overview", "time", "hotspots", "memory", "calls", "source", "visualization", "comparison", "diagnostics", "export".

**Value**

Invisibly returns a `debrief_help` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
pv_help()
pv_help("time")
pv_help("hotspots")
```

---

|                           |                                      |
|---------------------------|--------------------------------------|
| <code>pv_hot_lines</code> | <i>Hot source lines by self-time</i> |
|---------------------------|--------------------------------------|

---

**Description**

Returns the source lines where the most CPU time is spent. Requires source references (use `devtools::load_all()` for best results).

**Usage**

```
pv_hot_lines(x, n = NULL, min_pct = 0, min_time_ms = 0)
```

**Arguments**

`x` A profvis object.

`n` Maximum number of lines to return. If NULL, returns all that pass the filters.

`min_pct` Minimum percentage of total time to include (default 0).

`min_time_ms` Minimum time in milliseconds to include (default 0).

**Value**

A data frame with columns:

- `location`: File path and line number (e.g., "R/foo.R:42")
- `label`: Function name at this location
- `filename`: Source file path
- `linenum`: Line number
- `samples`: Number of profiling samples
- `time_ms`: Time in milliseconds
- `pct`: Percentage of total time

### Examples

```
p <- pv_example()
pv_hot_lines(p)

# Only lines with >= 10% of time
pv_hot_lines(p, min_pct = 10)
```

---

pv\_hot\_paths

*Hot call paths*

---

### Description

Returns the most common complete call stacks. This shows which execution paths through the code consume the most time.

### Usage

```
pv_hot_paths(x, n = NULL, include_source = TRUE)
```

### Arguments

|                |  |
|----------------|--|
| x              | A profvis object.  |
| n              | Maximum number of paths to return. If NULL, returns all.                                       |
| include_source | If TRUE and source references are available, include file:line information in the path labels. |

### Value

A data frame with columns:

- `stack`: The call path (functions separated by arrows)
- `samples`: Number of profiling samples with this exact path
- `time_ms`: Time in milliseconds
- `pct`: Percentage of total time

### Examples

```
p <- pv_example()
pv_hot_paths(p)
```

---

pv\_memory                      *Memory allocation by function*

---

**Description**

Returns memory allocation aggregated by function name.

**Usage**

```
pv_memory(x, n = NULL)
```

**Arguments**

x                      A profvis object.  
n                      Maximum number of functions to return. If NULL, returns all.

**Value**

A data frame with columns:

- label: Function name
- mem\_mb: Memory allocated in megabytes

**Examples**

```
p <- pv_example()
pv_memory(p)
```

---

pv\_memory\_lines                      *Memory allocation by source line*

---

**Description**

Returns memory allocation aggregated by source location. Requires source references (use devtools::load\_all() for best results).

**Usage**

```
pv_memory_lines(x, n = NULL)
```

**Arguments**

x                      A profvis object.  
n                      Maximum number of lines to return. If NULL, returns all.

**Value**

A data frame with columns:

- location: File path and line number
- label: Function name at this location
- filename: Source file path
- linum: Line number
- mem\_mb: Memory allocated in megabytes

**Examples**

```
p <- pv_example()
pv_memory_lines(p)
```

---

pv\_print\_call\_depth    *Print call depth breakdown*

---

**Description**

Print call depth breakdown

**Usage**

```
pv_print_call_depth(x)
```

**Arguments**

x                    A profvis object.

**Value**

Invisibly returns a `debrief_call_depth` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p <- pv_example()
pv_print_call_depth(p)
```

---

pv\_print\_call\_stats *Print call statistics*

---

**Description**

Print call statistics

**Usage**

```
pv_print_call_stats(x, n = 20)
```

**Arguments**

|   |                              |
|---|------------------------------|
| x | A profvis object.            |
| n | Number of functions to show. |

**Value**

Invisibly returns a debrief\_call\_stats object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p <- pv_example()
pv_print_call_stats(p)
```

---

pv\_print\_callers\_callees  
*Print caller/callee analysis for a function*

---

**Description**

Shows both callers (who calls this function) and callees (what this function calls) in a single view.

**Usage**

```
pv_print_callers_callees(x, func, n = 10)
```

**Arguments**

|      |  |
|------|--|
| x    | A profvis object.                          |
| func | The function name to analyze.              |
| n    | Maximum number of callers/callees to show. |

**Value**

Invisibly returns a `debrief_callers_callees` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p <- pv_example()
pv_print_callers_callees(p, "inner")
```

---

|                  |                                 |
|------------------|---------------------------------|
| pv_print_compare | <i>Print profile comparison</i> |
|------------------|---------------------------------|

---

**Description**

Print profile comparison

**Usage**

```
pv_print_compare(before, after, n = 15)
```

**Arguments**

|        |   |
|--------|---|
| before | A profvis object (before optimization).             |
| after  | A profvis object (after optimization).              |
| n      | Number of functions to show in detailed comparison. |

**Value**

Invisibly returns a `debrief_compare` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p1 <- pv_example()
p2 <- pv_example()
pv_print_compare(p1, p2)
```

---

pv\_print\_compare\_many *Print comparison of multiple profiles*

---

**Description**

Print comparison of multiple profiles

**Usage**

```
pv_print_compare_many(...)
```

**Arguments**

...                   Named profvis objects to compare, or a single named list.

**Value**

Invisibly returns a `debrief_compare_many` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p1 <- pv_example()
p2 <- pv_example("gc")
pv_print_compare_many(baseline = p1, gc_heavy = p2)
```

---

pv\_print\_debrief           *Print profiling summary*

---

**Description**

Prints a comprehensive text summary of profiling data suitable for terminal output or AI agent consumption.

**Usage**

```
pv_print_debrief(x, n_functions = 10, n_lines = 10, n_paths = 5, n_memory = 5)
```

**Arguments**

`x`                    A profvis object from `profvis::profvis()`.  
`n_functions`        Number of top functions to show (default 10).  
`n_lines`            Number of hot source lines to show (default 10).  
`n_paths`            Number of hot paths to show (default 5).  
`n_memory`           Number of memory hotspots to show (default 5).

**Value**

Invisibly returns a `debrief_debrief` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p <- pv_example()
pv_print_debrief(p)
```

---

`pv_print_file_summary` *Print file summary*

---

**Description**

Print file summary

**Usage**

```
pv_print_file_summary(x)
```

**Arguments**

`x` A `profvis` object.

**Value**

Invisibly returns a `debrief_file_summary` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p <- pv_example()
pv_print_file_summary(p)
```

---

pv\_print\_gc\_pressure *Print GC pressure analysis*

---

**Description**

Print GC pressure analysis

**Usage**

```
pv_print_gc_pressure(x, threshold = 10)
```

**Arguments**

x                    A profvis object.  
threshold            Minimum GC percentage to report (default 10).

**Value**

Invisibly returns a `debrief_gc_pressure` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p <- pv_example("gc")  
pv_print_gc_pressure(p)
```

---

pv\_print\_hot\_lines *Print hot lines with source context*

---

**Description**

Prints the hot source lines along with surrounding code context.

**Usage**

```
pv_print_hot_lines(x, n = 5, context = 3)
```

**Arguments**

x                    A profvis object.  
n                    Number of hot lines to show.  
context             Number of lines to show before and after each hotspot.

**Value**

Invisibly returns a `debrief_hot_lines` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p <- pv_example()
pv_print_hot_lines(p, n = 5, context = 3)
```

---

`pv_print_hot_paths`      *Print hot paths in readable format*

---

**Description**

Print hot paths in readable format

**Usage**

```
pv_print_hot_paths(x, n = 10, include_source = TRUE)
```

**Arguments**

`x`                      A profvis object.  
`n`                        Number of paths to show.  
`include_source`      Include source references in output.

**Value**

Invisibly returns a `debrief_hot_paths` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p <- pv_example()
pv_print_hot_paths(p, n = 3)
```

---

pv\_print\_memory      *Print memory allocation summary*

---

**Description**

Print memory allocation summary

**Usage**

```
pv_print_memory(x, n = 10, by = c("function", "line"))
```

**Arguments**

x                    A profvis object.  
n                    Number of top allocators to show.  
by                   Either "function" or "line".

**Value**

Invisibly returns a debrief\_memory object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p <- pv_example()
pv_print_memory(p, by = "function")
pv_print_memory(p, by = "line")
```

---

pv\_print\_recursive      *Print recursive functions analysis*

---

**Description**

Print recursive functions analysis

**Usage**

```
pv_print_recursive(x)
```

**Arguments**

x                    A profvis object.

**Value**

Invisibly returns a `debrief_recursive` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p <- pv_example("recursive")
pv_print_recursive(p)
```

---

`pv_print_suggestions` *Print optimization suggestions*

---

**Description**

Print optimization suggestions

**Usage**

```
pv_print_suggestions(x)
```

**Arguments**

`x`                    A `profvis` object.

**Value**

Invisibly returns a `debrief_suggestions` object. Use `capture.output()` to capture the formatted text output.

**Examples**

```
p <- pv_example("gc")
pv_print_suggestions(p)
```

---

|              |                                   |
|--------------|-----------------------------------|
| pv_recursive | <i>Detect recursive functions</i> |
|--------------|-----------------------------------|

---

### Description

Identifies functions that call themselves (directly recursive) or appear multiple times in the same call stack. Recursive functions in hot paths are often optimization targets.

### Usage

```
pv_recursive(x)
```

### Arguments

x                    A profvis object.

### Value

A data frame with columns:

- label: Function name
- max\_depth: Maximum recursion depth observed
- avg\_depth: Average recursion depth when recursive
- recursive\_samples: Number of samples where function appears multiple times
- total\_samples: Total samples where function appears
- pct\_recursive: Percentage of appearances that are recursive
- total\_ms: Total time on call stack
- pct\_time: Percentage of total profile time

### Examples

```
p <- pv_example("recursive")
pv_recursive(p)
```

---

|              |                                      |
|--------------|--------------------------------------|
| pv_self_time | <i>Self-time summary by function</i> |
|--------------|--------------------------------------|

---

### Description

Returns the time spent directly in each function (at the top of the call stack). This shows where CPU cycles are actually being consumed.

### Usage

```
pv_self_time(x, n = NULL, min_pct = 0, min_time_ms = 0)
```

### Arguments

|             |  |
|-------------|--|
| x           | A profvis object.  |
| n           | Maximum number of functions to return. If NULL, returns all that pass the filters. |
| min_pct     | Minimum percentage of total time to include (default 0).                           |
| min_time_ms | Minimum time in milliseconds to include (default 0).                               |

### Value

A data frame with columns:

- label: Function name
- samples: Number of profiling samples
- time\_ms: Time in milliseconds
- pct: Percentage of total time

### Examples

```
p <- pv_example()
pv_self_time(p)

# Only functions with >= 5% self-time
pv_self_time(p, min_pct = 5)
```

---

pv\_source\_context      *Show source context for a specific location*

---

### Description

Displays source code around a specific file and line number with profiling information for each line.

### Usage

```
pv_source_context(x, filename, linenum = NULL, context = 10)
```

### Arguments

|          |  |
|----------|--|
| x        | A profvis object.  |
| filename | The source file to examine.  |
| linenum  | The line number to center on. If NULL, shows the hottest line in the file. |
| context  | Number of lines to show before and after.                                  |

### Value

Invisibly returns a `debrief_source_context` object. Use `capture.output()` to capture the formatted text output.

### Examples

```
p <- pv_example()
pv_source_context(p, "R/main.R", linenum = 10)
```

---

pv\_suggestions      *Generate optimization suggestions*

---

### Description

Analyzes the profile and generates specific, actionable optimization suggestions based on detected patterns and hotspots.

### Usage

```
pv_suggestions(x)
```

### Arguments

|   |                   |
|---|-------------------|
| x | A profvis object. |
|---|-------------------|

**Value**

A data frame with columns:

- priority: 1 (highest) to 5 (lowest)
- category: Type of optimization (e.g., "data structure", "algorithm")
- location: Where to apply the optimization
- action: What to do
- pattern: Code pattern to look for (or NA)
- replacement: Suggested replacement (or NA)
- potential\_impact: Estimated time that could be saved

**Examples**

```
p <- pv_example("gc")
pv_suggestions(p)
```

---

pv\_to\_json

*Export profiling results as JSON*

---

**Description**

Exports profiling analysis results in JSON format for consumption by AI agents, automated tools, or external applications.

**Usage**

```
pv_to_json(
  x,
  file = NULL,
  pretty = TRUE,
  include = c("summary", "self_time", "total_time", "hot_lines", "memory", "gc_pressure",
    "suggestions", "recursive"),
  system_info = FALSE
)
```

**Arguments**

|             |  |
|-------------|--|
| x           | A profvis object.  |
| file        | Optional file path to write JSON to. If NULL, returns the JSON string.   |
| pretty      | If TRUE, formats JSON with indentation for readability.  |
| include     | Character vector specifying which analyses to include. Options: "summary", "self_time", "total_time", "hot_lines", "memory", "callers", "gc_pressure", "suggestions", "recursive". Default includes all. |
| system_info | If TRUE, includes R version and platform info in metadata. Useful for reproducibility.   |

**Value**

If file is NULL, returns a JSON string. Otherwise writes to file and returns the file path invisibly.

**Examples**

```
p <- pv_example()
json <- pv_to_json(p)
cat(json)

# Include only specific analyses
json <- pv_to_json(p, include = c("self_time", "hot_lines"))

# Include system info for reproducibility
json <- pv_to_json(p, system_info = TRUE)
```

---

pv\_to\_list

*Export profiling results as a list*

---

**Description**

Returns all profiling analysis results as a nested R list, useful for programmatic access to results without JSON serialization.

**Usage**

```
pv_to_list(
  x,
  include = c("summary", "self_time", "total_time", "hot_lines", "memory", "gc_pressure",
    "suggestions", "recursive"),
  system_info = FALSE
)
```

**Arguments**

x                    A profvis object.

include             Character vector specifying which analyses to include. Same options as [pv\\_to\\_json\(\)](#).

system\_info        If TRUE, includes R version and platform info in metadata.

**Value**

A named list containing the requested analyses.

## Examples

```
p <- pv_example()
results <- pv_to_list(p)
names(results)
results$self_time
```

---

|               |                                       |
|---------------|---------------------------------------|
| pv_total_time | <i>Total time summary by function</i> |
|---------------|---------------------------------------|

---

## Description

Returns the time spent in each function including all its callees. This shows which functions are on the call stack when time is being spent.

## Usage

```
pv_total_time(x, n = NULL, min_pct = 0, min_time_ms = 0)
```

## Arguments

|             |  |
|-------------|--|
| x           | A profvis object.  |
| n           | Maximum number of functions to return. If NULL, returns all that pass the filters. |
| min_pct     | Minimum percentage of total time to include (default 0).                           |
| min_time_ms | Minimum time in milliseconds to include (default 0).                               |

## Value

A data frame with columns:

- label: Function name
- samples: Number of profiling samples where function appeared
- time\_ms: Time in milliseconds
- pct: Percentage of total time

## Examples

```
p <- pv_example()
pv_total_time(p)

# Only functions with >= 50% total time
pv_total_time(p, min_pct = 50)
```

---

|               |                                    |
|---------------|------------------------------------|
| pv_worst_line | <i>Get the single hottest line</i> |
|---------------|------------------------------------|

---

**Description**

Returns the hottest source line with full context. Useful for quickly identifying the #1 optimization target.

**Usage**

```
pv_worst_line(x, context = 5)
```

**Arguments**

|         |   |
|---------|---|
| x       | A profvis object.                                   |
| context | Number of source lines to include before and after. |

**Value**

A list with:

- location: File path and line number (e.g., "R/foo.R:42")
- label: Function name
- filename: Source file path
- linenum: Line number
- time\_ms: Time in milliseconds
- pct: Percentage of total time
- code: The source line
- context: Vector of surrounding source lines
- callers: Data frame of functions that call this location

Returns NULL if no source references are available.

**Examples**

```
p <- pv_example()
pv_worst_line(p)
```

# Index

profvis::profvis(), [7](#), [19](#)  
pv\_call\_depth, [2](#)  
pv\_call\_stats, [3](#)  
pv\_callees, [4](#)  
pv\_callers, [5](#)  
pv\_compare, [5](#)  
pv\_compare\_many, [6](#)  
pv\_debrief, [7](#)  
pv\_example, [8](#)  
pv\_file\_summary, [9](#)  
pv\_flame, [9](#)  
pv\_flame\_condense, [10](#)  
pv\_focus, [11](#)  
pv\_gc\_pressure, [11](#)  
pv\_help, [12](#)  
pv\_hot\_lines, [13](#)  
pv\_hot\_paths, [14](#)  
pv\_memory, [15](#)  
pv\_memory\_lines, [15](#)  
pv\_print\_call\_depth, [16](#)  
pv\_print\_call\_stats, [17](#)  
pv\_print\_callers\_callees, [17](#)  
pv\_print\_compare, [18](#)  
pv\_print\_compare\_many, [19](#)  
pv\_print\_debrief, [19](#)  
pv\_print\_file\_summary, [20](#)  
pv\_print\_gc\_pressure, [21](#)  
pv\_print\_hot\_lines, [21](#)  
pv\_print\_hot\_paths, [22](#)  
pv\_print\_memory, [23](#)  
pv\_print\_recursive, [23](#)  
pv\_print\_suggestions, [24](#)  
pv\_recursive, [25](#)  
pv\_self\_time, [26](#)  
pv\_source\_context, [27](#)  
pv\_suggestions, [27](#)  
pv\_to\_json, [28](#)  
pv\_to\_json(), [29](#)  
pv\_to\_list, [29](#)  
pv\_total\_time, [30](#)  
pv\_worst\_line, [31](#)