Efficient coding patterns in R and Python

2024-09-28

1. Do less
2. Benchmark and experiment
3. Vectorize
4. Profile to find bottlenecks
5. Parallelize

1. Use faster tools
2. Write in a faster language

Do less

Do less

- Subset and aggregate in SQL
- Subset rows earlier (before calculations)

Your Turn 1 (`exercises_r.qmd`, `exercises_py.qmd`)

Read in the `fd_calls.csv` file. Create a new variable called `log_delay` that is the log of `Delay`. Subset the data frame to just use rows where `year` is 2015.

Benchmark and experiment

Benchmarking

- Write your code as a function
- Run it repeatedly to get information on speed

R: bench

```r
x <- runif(1000)
sqrt2 <- function(x) x ^ 0.5

bench::mark(
  sqrt(x),
  sqrt2(x)
)
```

```
# A tibble: 2 × 6
  expression      min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
1 sqrt(x)       984ns   1.44µs   538963.    7.86KB     53.9
2 sqrt2(x)     9.35µs   9.92µs    98246.    7.86KB     19.7
```

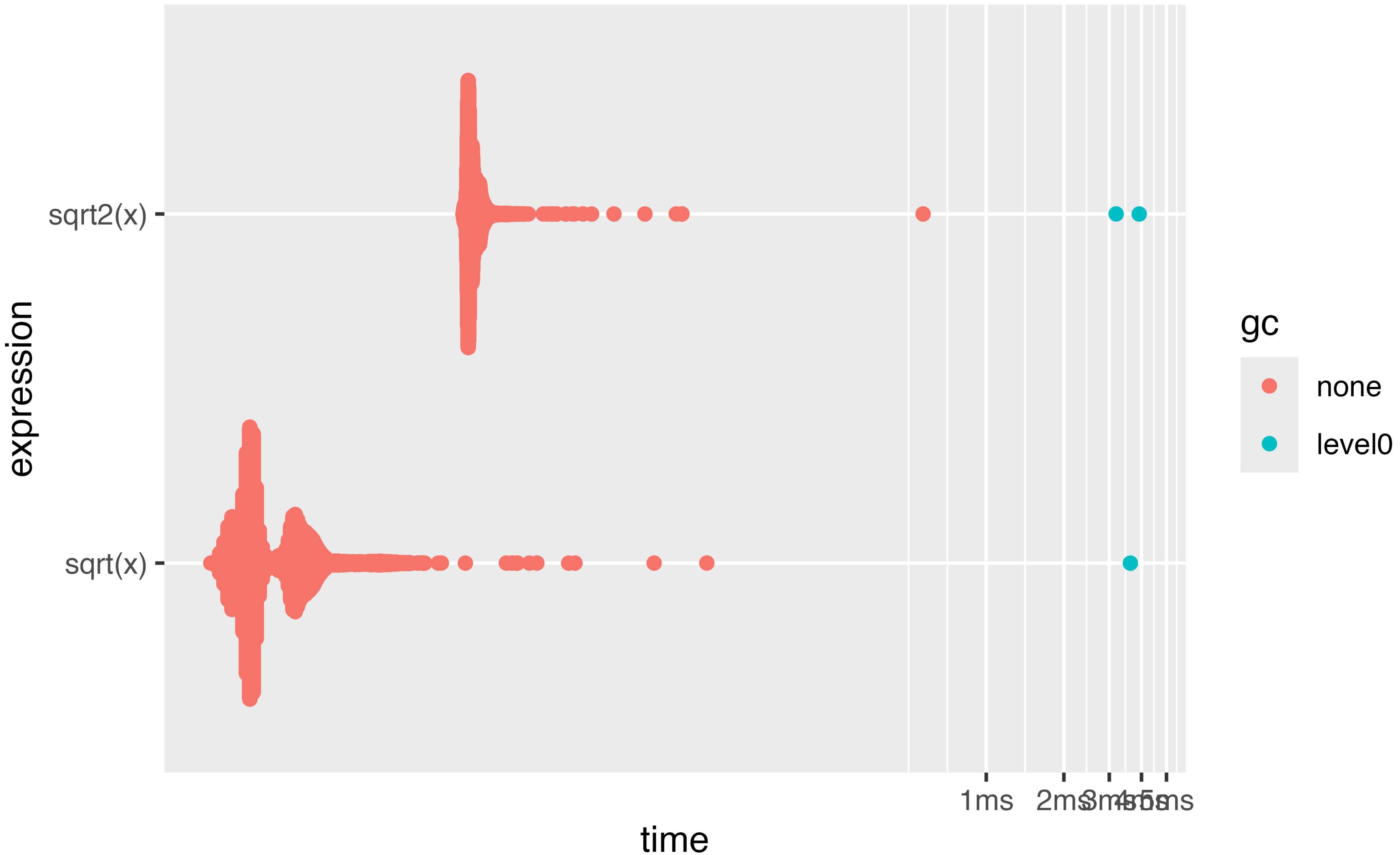
R: bench

```r
x <- runif(1000)
sqrt2 <- function(x) x ^ 0.5

bm <- bench::mark(
  sqrt(x),
  sqrt2(x),
  relative = TRUE
)

bm
```

```
# A tibble: 2 × 6
  expression   min median `itr/sec` mem_alloc `gc/sec`
  <bch:expr> <dbl>  <dbl>     <dbl>     <dbl>    <dbl>
1 sqrt(x)     1      1          6.04         1     3.02
2 sqrt2(x)    9.50   7.03       1            1     1
```

R: bench

```r
plot(bm)
```



Python: timeit

```python
import numpy as np
import timeit

x = np.random.uniform(size=1000)

def sqrt2(x):
    return x ** 0.5

%timeit np.sqrt(x)
%timeit sqrt2(x)
```

```
658 ns ± 4.33 ns per loop (mean ± std. dev. of 7 runs, 1,000,000
loops each)
691 ns ± 7.46 ns per loop (mean ± std. dev. of 7 runs, 1,000,000
loops each)
```

Python: timeit

```python
import numpy as np
import timeit

x = np.random.uniform(size=1000)

def sqrt2(x):
    return x ** 0.5

nbm = timeit.timeit("np.sqrt(x)", globals=globals())
cbm = timeit.timeit("sqrt2(x)", globals=globals())

nbm/cbm
```

```
0.9511412986201114
```

Your Turn 2

Benchmark the two approaches you wrote in Your Turn 1 using benchmarking. First, write a function for each approach, then call the function in the benchmarking tool

Arrow

```
library(arrow)
fd_calls <- read_csv("fd_calls.csv")
fd_calls |>
  group_by(year) |>
  write_dataset("fd_calls")
```

Arrow

```
open_dataset("fd_calls") |>
  filter(year == 2015) |>
  mutate(log_delay = log(Delay)) |>
  group_by(Neighborhood) |>
  summarise(log_delay = mean(log_delay)) |>
  collect()
```

Vectorize

Vectorization: R

```r
xs <- runif(100)
out <- xs[1]
for (x in xs[-1]) {
  out <- c(out, out[length(out)] + x)
}

head(out)
```

```
[1] 0.3534023 1.1171094 1.6104174 2.2896895 3.1235836 4.0780196
```

Vectorization: R

```r
cumsum(xs) |>
  head()
```

```
[1] 0.3534023 1.1171094 1.6104174 2.2896895 3.1235836 4.0780196
```

Vectorization: Python

```
xs = np.random.uniform(size=100)
np.cumsum(xs)[:6]
```

```
array([0.2286732 , 0.7722714 , 1.13190394, 1.35082272, 1.65016753,
       1.96338569])
```

Of course someone has to write loops. It doesn’t have to be you.
—Jenny Bryan

Vectorising is about taking a whole-object approach to a problem, thinking about vectors, not scalars.

—Hadley Wickham

Vectorization: smells and solutions

- iterating by row: look for a way to work with the whole column as a vector

- iterating by groups: use grouping and aggregating

Your Turn 3: Challenge!

This exercise contains a simulation with a population and two tables of effects. It uses a for loop to apply the effects to calculate a cost for each person in the population. For this exercise, vectorize this for loop to make it more efficient. Benchmark the two approaches and compare.

List-comprehensions

```
sentences = [
    "The better part of Valour, is Discretion.",
    "I had rather have a fool to make me merry than " +
    "experience to make me sad.",
    "I wasted time, and now doth time waste me.",
    "The empty vessel makes the loudest sound.",
    "Give every man thy ear, but few thy voice."
]

len(sentences[0].split())
```

7

List-comprehensions

```
n_words = []
for sentence in sentences:
    n_words.append(len(sentence.split()))

n_words
```

```
[7, 16, 9, 7, 9]
```

List-comprehensions

```
n_words = [len(sentence.split()) for sentence in sentences]

n_words
```

```
[7, 16, 9, 7, 9]
```

Case when-style statements (Python)

```
import pandas as pd

# Sample dataframe
df = pd.DataFrame({
    'patient': ['Patient1', 'Patient2', 'Patient3', 'Patient4'],
    'a1c': [5.2, 5.9, 6.8, 5.6]
})
```

Case when-style statements (Python)

```python
df['diabetes_status'] = None
df.loc[df['a1c'] < 5.7, 'diabetes_status'] = 'Normal'
df.loc[(df['a1c'] >= 5.7) & (df['a1c'] < 6.5), 'diabetes_status'
df.loc[df['a1c'] >= 6.5, 'diabetes_status'] = 'Diabetes'

df
```

```
    patient  a1c diabetes_status
0  Patient1  5.2          Normal
1  Patient2  5.9    Pre-diabetes
2  Patient3  6.8        Diabetes
3  Patient4  5.6          Normal
```

Case when-style statements (Python)

```
import numpy as np

conditions = [
    (df['a1c'] < 5.7),
    (df['a1c'] >= 5.7) & (df['a1c'] < 6.5),
    (df['a1c'] >= 6.5)
]

choices = ['Normal', 'Pre-diabetes', 'Diabetes']

df['diabetes_status'] = np.select(conditions, choices)
df
```

```
    patient  a1c diabetes_status
0  Patient1  5.2          Normal
1  Patient2  5.9    Pre-diabetes
2  Patient3  6.8        Diabetes
3  Patient4  5.6          Normal
```

Case when-style statements (Tidyverse)

```r
df <- data.frame(
  patient = c("Patient1", "Patient2", "Patient3", "Patient4"),
  a1c = c(5.2, 5.9, 6.8, 5.6)
)

df |>
  mutate(
    diabetes_status = case_when(
      a1c < 5.7 ~ "Normal",
      a1c >= 5.7 & a1c < 6.5 ~ "Pre-diabetes",
      a1c >= 6.5 ~ "Diabetes"
    )
  )
```

```
   patient a1c diabetes_status
1 Patient1 5.2          Normal
2 Patient2 5.9    Pre-diabetes
3 Patient3 6.8        Diabetes
4 Patient4 5.6          Normal
```

Case when-style statements (data.table)

```
dt <- data.table(
  patient = c("Patient1", "Patient2", "Patient3", "Patient4"),
  a1c = c(5.2, 5.9, 6.8, 5.6)
)

dt[, diabetes_status := fcase(
  a1c < 5.7, "Normal",
  a1c >= 5.7 & a1c < 6.5, "Pre-diabetes",
  a1c >= 6.5, "Diabetes"
)]
```

Functional programming (R)

```r
# base R
lapply(a_list, \(.x) do_something(.x))

# purrr
map(a_list, \(.x) do_something(.x))
```

Functional programming (Python)

```
list(map(lambda x: do_something(x), a_list))
```

Functional programming: reduce

```
temp_df <- left_join(df1, df2, by = "key")
temp_df <- left_join(temp_df, df3, by = "key")
df <- left_join(temp_df, df4, by = "key")
```

Functional programming: reduce

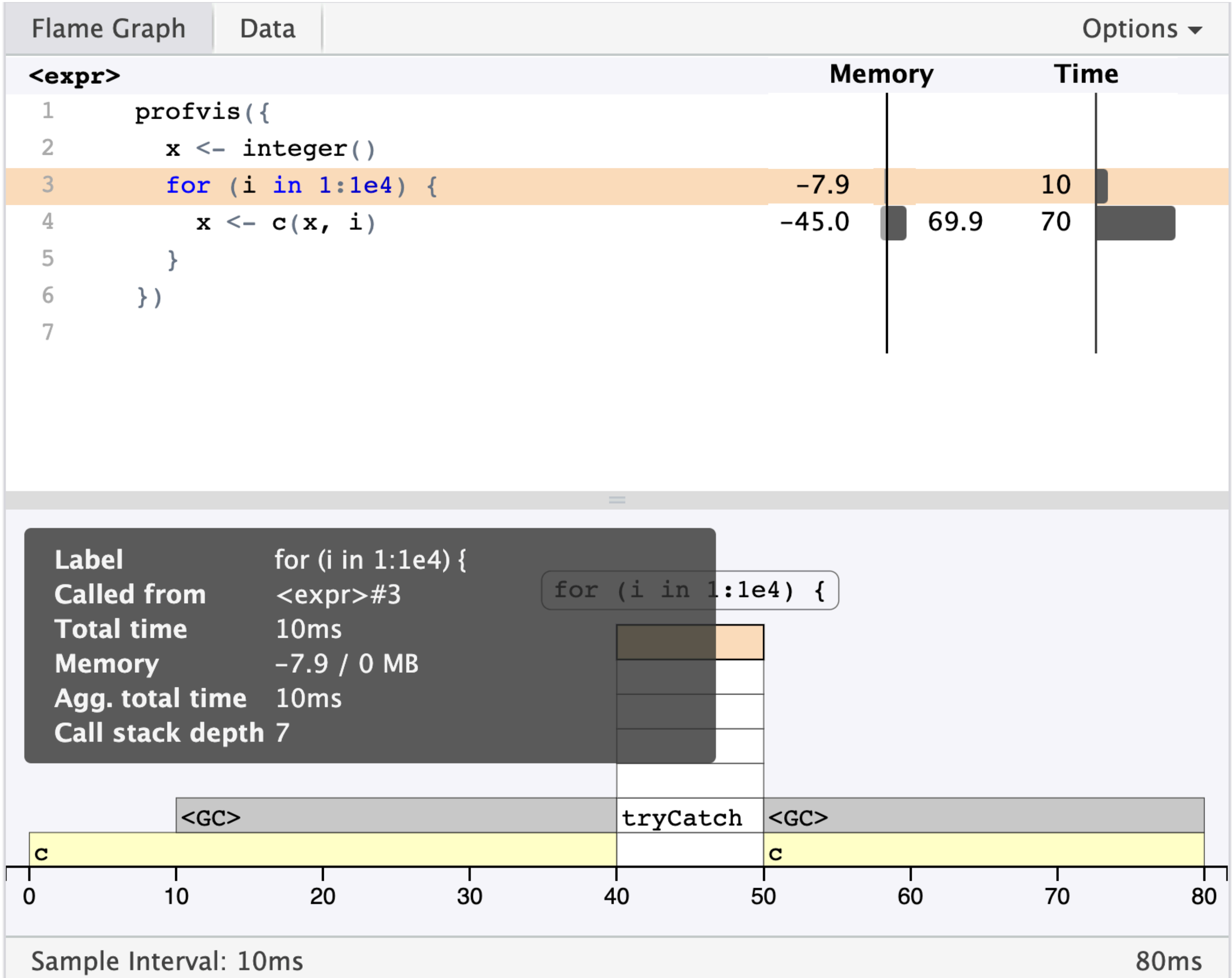
```
df <- list(df1, df2, df3, df4) |>
  reduce(left_join, by = "key")
```

Profile to find bottlenecks

Profiling code: R

```r
library(profvis)

profvis({
  x <- integer()
  for (i in 1:1e4) {
    x <- c(x, i)
  }
})
```

Profiling code: R



Profiling code: R

```
library(profvis)
source("your_code.R")
profvis({
  do_something_to(thing)
})
```

Profiling code: Python

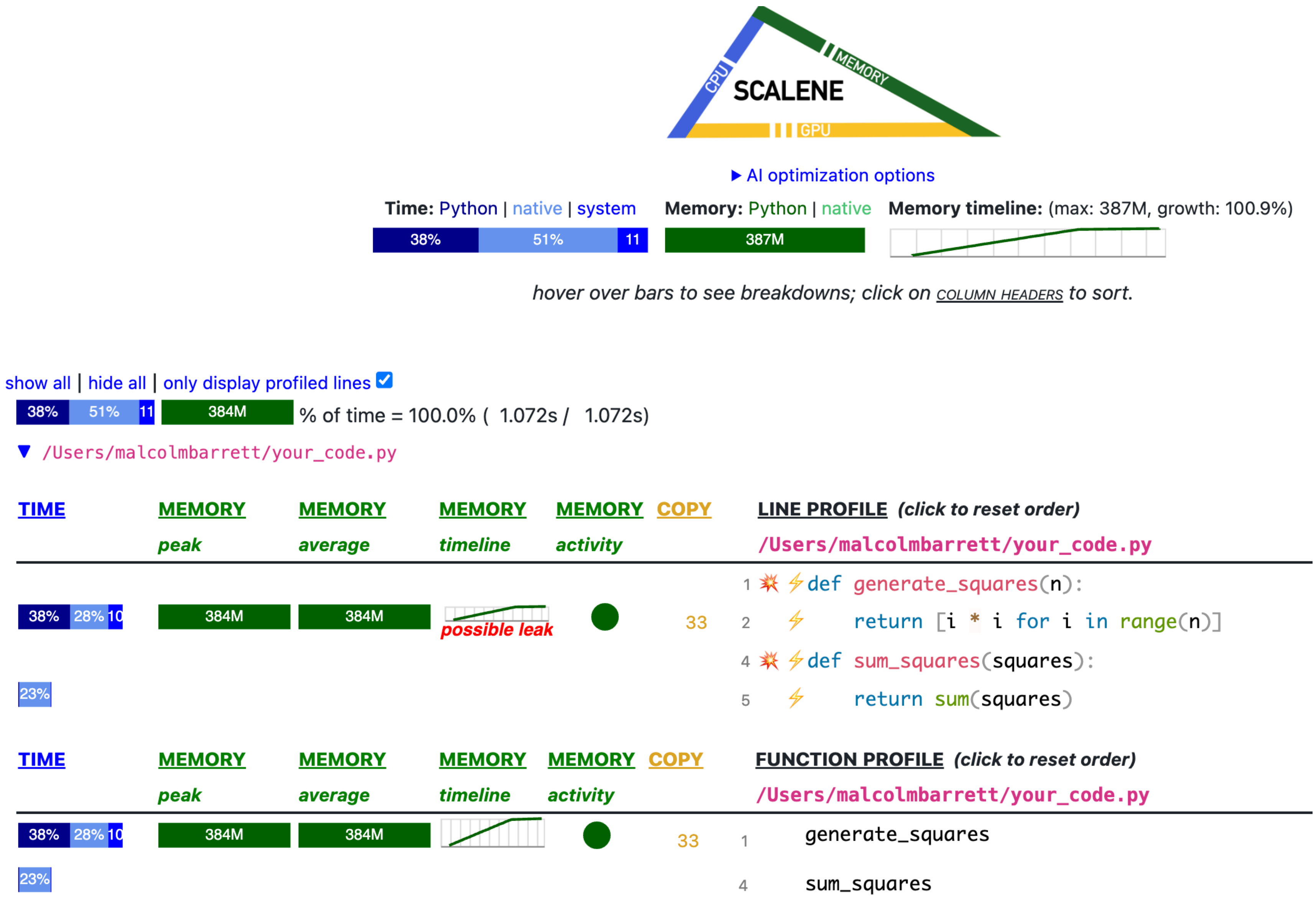
your_code.py

```python
def generate_squares(n):
    return [i * i for i in range(n)]

def sum_squares(squares):
    return sum(squares)

n = 10_000_000
squares = generate_squares(n)
result = sum_squares(squares)

print(result)
```

terminal

```
scalene your_code.py
```

Profiling code: Python



Profiling code: Python

```python
def generate_squares(n):
    return (i * i for i in range(n))

def sum_squares(squares):
    return sum(squares)

n = 10_000_000
squares = generate_squares(n)
result = sum_squares(squares)

print(result)
```

Your Turn 4: Challenge!

Profile the code in the stated file. Try to improve the speed of the code based on your findings. The R and Python versions use different examples for this exercise.

Parallelize

Parallelizing code: R

```
library(future)
n_cores <- availableCores() - 2
plan(multisession, workers = n_cores)
```

Parallelizing code: Base R

```r
library(future.apply)
future_lapply(a_list, do_something)
```

Parallelizing code: Tidyverse

```r
library(furrr)
future_map(a_list, do_something)
```

Parallelizing code: Python

```
from concurrent.futures import ProcessPoolExecutor
from os import os.cpu_count
from your_script import do_something

n_cores = cpu_count() - 2

with ProcessPoolExecutor(max_workers=n_cores) as exec:
    results = list(exec.map(do_something, a_list))
```

Tools that paralellize automagically

- polars
- duckdb
- (sometimes) data.table

Your Turn 5

In this exercise, modify the bootstrap procedure to use parallel processing

Use faster tools

Faster tools

- duckdb
- polars
- data.table

Faster backends

- Tidyverse: duckplyr, dtplyr, tidypolars, etc.
- Pandas: dask, fireducks

Write in a faster language

Compiled languages

- C, C++
- Rust
- Look for tools already doing this!

1. Do less
2. Benchmark and experiment
3. Vectorize
4. Profile to find bottlenecks
5. Parallelize

1. Use faster tools
2. Write in a faster language