

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

```
1 x <- runif(1000)
2 sqrt2 <- function(x) x ^ 0.5
3
4 bench::mark(
5   sqrt(x),
6   sqrt2(x)
7 )
```

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)	1.02μs	1.52μs	517584.	7.86KB	51.8
2	sqrt2(x)	9.39μs	10.04μs	96454.	7.86KB	19.3

R: bench

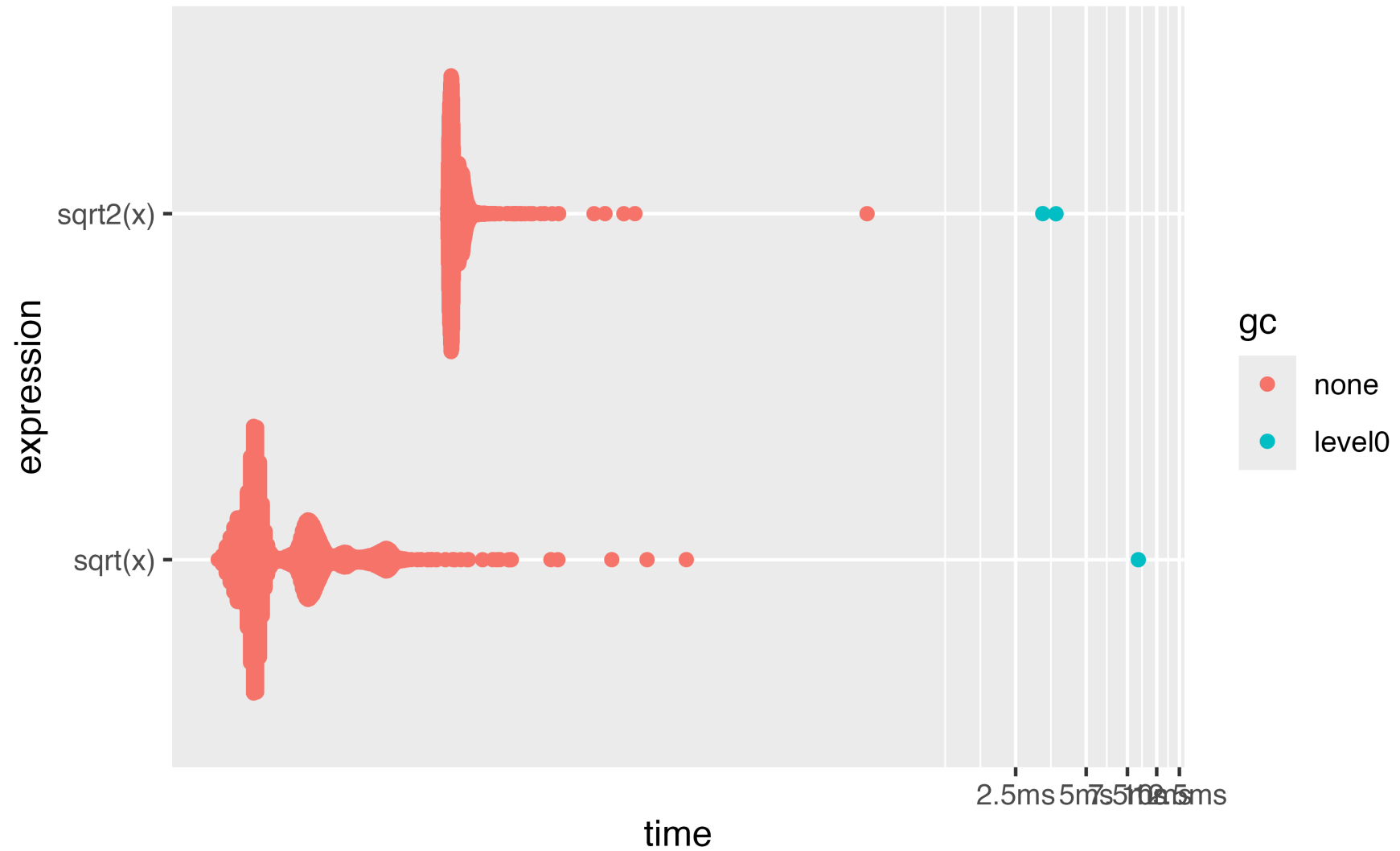
```
1 x <- runif(1000)
2 sqrt2 <- function(x) x ^ 0.5
3
4 bm <- bench::mark(
5   sqrt(x),
6   sqrt2(x),
7   relative = TRUE
8 )
9
10 bm
```

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	5.28	1	2.64
2	sqrt2(x)	9.54	6.91	1	1	1

R: bench

```
1 plot(bm)
```



Python: timeit

```
1 import numpy as np
2 import timeit
3
4 x = np.random.uniform(size=1000)
5
6 def sqrt2(x):
7     return x ** 0.5
8
9 %timeit np.sqrt(x)
10 %timeit sqrt2(x)
```

658 ns \pm 4.33 ns per loop (mean \pm std. dev. of 7 runs, 1,000,000 loops each)

691 ns \pm 7.46 ns per loop (mean \pm std. dev. of 7 runs, 1,000,000 loops each)

Python: timeit

```
1 import numpy as np
2 import timeit
3
4 x = np.random.uniform(size=1000)
5
6 def sqrt2(x):
7     return x ** 0.5
8
9 nbm = timeit.timeit("np.sqrt(x)", globals=globals())
10 cbm = timeit.timeit("sqrt2(x)", globals=globals())
11
12 nbm/cbm
```

0.931157519257646

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

```
1 library(arrow)
2 fd_calls <- read_csv("fd_calls.csv")
3 fd_calls |>
4   group_by(year) |>
5   write_dataset("fd_calls")
```

Arrow

```
1 open_dataset("fd_calls") |>
2   filter(year == 2015) |>
3   mutate(log_delay = log(Delay)) |>
4   group_by(Neighborhood) |>
5   summarise(log_delay = mean(log_delay)) |>
6   collect()
```


Vectorize

Vectorization: R

```
1 xs <- runif(100)
2 out <- xs[1]
3 for (x in xs[-1]) {
4   out <- c(out, out[length(out)] + x)
5 }
6
7 head(out)
```

```
[1] 0.07301948 0.26309714 1.19924931 2.02962025 2.81384381
3.03295182
```

Vectorization: R

```
1 cumsum(xs) |>  
2   head()
```

```
[1] 0.07301948 0.26309714 1.19924931 2.02962025 2.81384381  
3.03295182
```

Vectorization: Python

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

```
array([0.21228289, 0.95770549, 1.48998783, 2.41253633, 2.88187069,  
       2.95438318])
```

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

```
1 sentences = [  
2     "The better part of Valour, is Discretion.",  
3     "I had rather have a fool to make me merry than " +  
4     "experience to make me sad.",  
5     "I wasted time, and now doth time waste me.",  
6     "The empty vessel makes the loudest sound.",  
7     "Give every man thy ear, but few thy voice."  
8 ]  
9  
10 len(sentences[0].split())
```

7

List-comprehensions

```
1 n_words = []  
2 for sentence in sentences:  
3     n_words.append(len(sentence.split()))  
4  
5 n_words
```

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

List-comprehensions

```
1 n_words = [len(sentence.split()) for sentence in sentences]  
2  
3 n_words
```

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

Case when-style statements (Python)

```
1 import pandas as pd
2
3 # Sample dataframe
4 df = pd.DataFrame({
5     'patient': ['Patient1', 'Patient2', 'Patient3', 'Patient4'],
6     'alc': [5.2, 5.9, 6.8, 5.6]
7 })
```

Case when-style statements (Python)

```
1 df['diabetes_status'] = None
2 df.loc[df['a1c'] < 5.7, 'diabetes_status'] = 'Normal'
3 df.loc[(df['a1c'] >= 5.7) & (df['a1c'] < 6.5), 'diabetes_status'] = 'Pre-diabetes'
4 df.loc[df['a1c'] >= 6.5, 'diabetes_status'] = 'Diabetes'
5
6 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)

```
1 import numpy as np
2
3 conditions = [
4     (df['a1c'] < 5.7),
5     (df['a1c'] >= 5.7) & (df['a1c'] < 6.5),
6     (df['a1c'] >= 6.5)
7 ]
8
9 choices = ['Normal', 'Pre-diabetes', 'Diabetes']
10
11 df['diabetes_status'] = np.select(conditions, choices)
12 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)

```
1 df <- data.frame(  
2   patient = c("Patient1", "Patient2", "Patient3", "Patient4"),  
3   a1c = c(5.2, 5.9, 6.8, 5.6)  
4 )  
5  
6 df |>  
7   mutate(  
8     diabetes_status = case_when(  
9       a1c < 5.7 ~ "Normal",  
10      a1c >= 5.7 & a1c < 6.5 ~ "Pre-diabetes",  
11      a1c >= 6.5 ~ "Diabetes"  
12    )  
13  )
```

	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)

```
1 dt <- data.table(  
2   patient = c("Patient1", "Patient2", "Patient3", "Patient4"),  
3   a1c = c(5.2, 5.9, 6.8, 5.6)  
4 )  
5  
6 dt[, diabetes_status := fcase(  
7   a1c < 5.7, "Normal",  
8   a1c >= 5.7 & a1c < 6.5, "Pre-diabetes",  
9   a1c >= 6.5, "Diabetes"  
10 )]
```


Functional programming (R)

```
1 # base R
2 lapply(a_list, \(.x) do_something(.x))
3
4 # purrr
5 map(a_list, \(.x) do_something(.x))
```

Functional programming (Python)

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

Functional programming: reduce

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

Functional programming: reduce

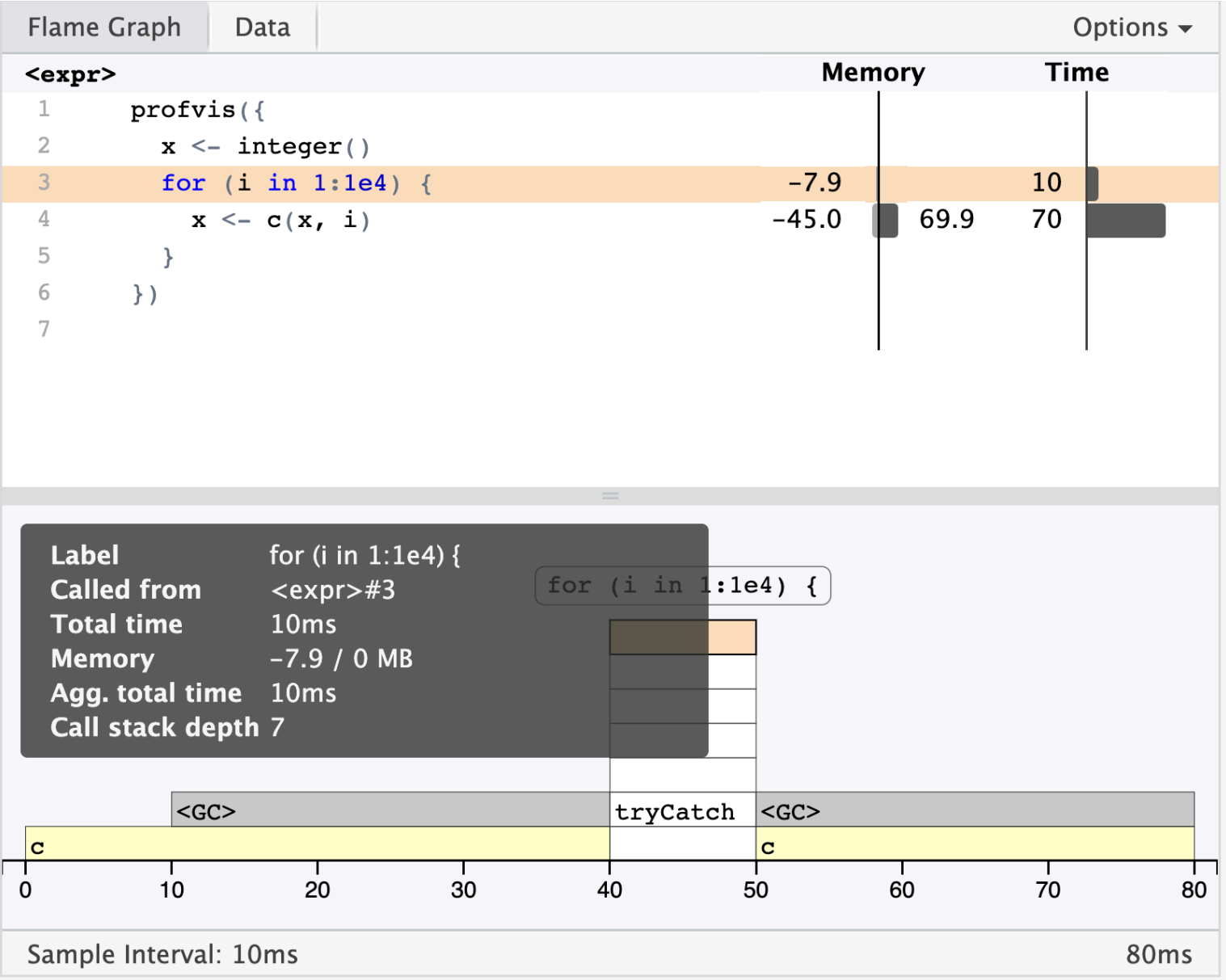
```
1 df <- list(df1, df2, df3, df4) |>  
2   reduce(left_join, by = "key")
```

Profile to find bottlenecks

Profiling code: R

```
1 library(profvis)
2
3 profvis({
4   x <- integer()
5   for (i in 1:1e4) {
6     x <- c(x, i)
7   }
8 })
```

Profiling code: R



Profiling code: R

```
1 library(profvis)
2 source("your_code.R")
3 profvis({
4   do_something_to(thing)
5 })
```


Profiling code: Python

your_code.py

```
1  def generate_squares(n):  
2      return [i * i for i in range(n)]  
3  
4  def sum_squares(squares):  
5      return sum(squares)  
6  
7  n = 10_000_000  
8  squares = generate_squares(n)  
9  result = sum_squares(squares)  
10  
11 print(result)
```

terminal

```
1  scalene your_code.py
```

Profiling code: Python



▶ AI optimization options

Time: Python | native | system

Memory: Python | native

Memory timeline: (max: 387M, growth: 100.9%)



hover over bars to see breakdowns; click on COLUMN HEADERS to sort.

show all | hide all | only display profiled lines ☒

38% 51% 11 384M % of time = 100.0% (1.072s / 1.072s)

▼ /Users/malcolmbarrett/your_code.py

TIME	MEMORY peak	MEMORY average	MEMORY timeline	MEMORY activity	COPY	LINE PROFILE (click to reset order)
						/Users/malcolmbarrett/your_code.py
38% 28% 10% 23%	384M	384M	 possible leak		33	1 ⚡ def generate_squares(n): 2 ⚡ return [i * i for i in range(n)] 4 ⚡ def sum_squares(squares): 5 ⚡ return sum(squares)
TIME	MEMORY peak	MEMORY average	MEMORY timeline	MEMORY activity	COPY	FUNCTION PROFILE (click to reset order)
						/Users/malcolmbarrett/your_code.py
38% 28% 10% 23%	384M	384M	 possible leak		33	1 generate_squares 4 sum_squares

Profiling code: Python

```
1  def generate_squares(n):  
2      return (i * i for i in range(n))  
3  
4  def sum_squares(squares):  
5      return sum(squares)  
6  
7  n = 10_000_000  
8  squares = generate_squares(n)  
9  result = sum_squares(squares)  
10  
11 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

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

Parallelizing code: Base R

```
1 library(future.apply)
2 future_lapply(a_list, do_something)
```

Parallelizing code: Tidyverse

```
1 library(furrr)
2 future_map(a_list, do_something)
```


Parallelizing code: Python

```
1 from concurrent.futures import ProcessPoolExecutor
2 from os import os.cpu_count
3 from your_script import do_something
4
5 n_cores = cpu_count() - 2
6
7 with ProcessPoolExecutor(max_workers=n_cores) as exec:
8     results = list(exec.map(do_something, a_list))
```

Tools that paralellize automatically

- 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