

# Week 12 Tutorial: Complexity and Performance

POP77001 Computer Programming for Social Scientists

# Benchmarking in R

- In the lecture we used `system.time()` function to analyse function performance.
- Albeit conveniently built-in, the main drawback is that it's rather coarse.
- While useful for detecting large performance gaps, it often doesn't capture more subtle differences.
- The reason is that it only runs once and uses seconds as a standard unit of measurement.
- Here we will use `microbenchmark` package and identically named function to time function calls.
- Remember to print out the results of `microbenchmark`, otherwise times of individual runs are returned.

```
1 library("microbenchmark")  
2  
3 # Here we run 1000 times the same function call  
4 # and time how long it takes to run  
5 microbenchmark::microbenchmark(  
6   mean(rnorm(n = 1000)),  
7   times = 1000  
8 )
```

Unit: microseconds

|  | expr                  | min    | lq      | mean     | median | uq      | max     | neval |
|--|-----------------------|--------|---------|----------|--------|---------|---------|-------|
|  | mean(rnorm(n = 1000)) | 46.233 | 49.0625 | 58.18133 | 51.231 | 58.2125 | 170.677 | 1000  |

# Exercise: Compare Performance in R

- Consider a data frame with 50 different variables below.
- We want to know the mean of each of those variables.
- There are 2 principal ways of estimating them:
  - One using `apply()` function.
  - Or using built-in `colMeans()` function.
- Apply each of those function to calculate means.
- Benchmark the time it took to run using `system.time()` benchmark and `microbenchmark` package.
- What do you find?

```
1 set.seed(1234)
2 # Here we create a data frame of 1000 observations of 50 variables
3 # where each variable is a random draw from a normal distribution with mean
4 # drawn from a uniform distribution between 0 and 10 and standard deviation
5 df <- data.frame(mapply(
6   function(x) cbind(rnorm(n = 1000, mean = x, sd = 1)),
7   runif(n = 50, min = 0, max = 10)
8 ))
```

```
1 dim(df)
```

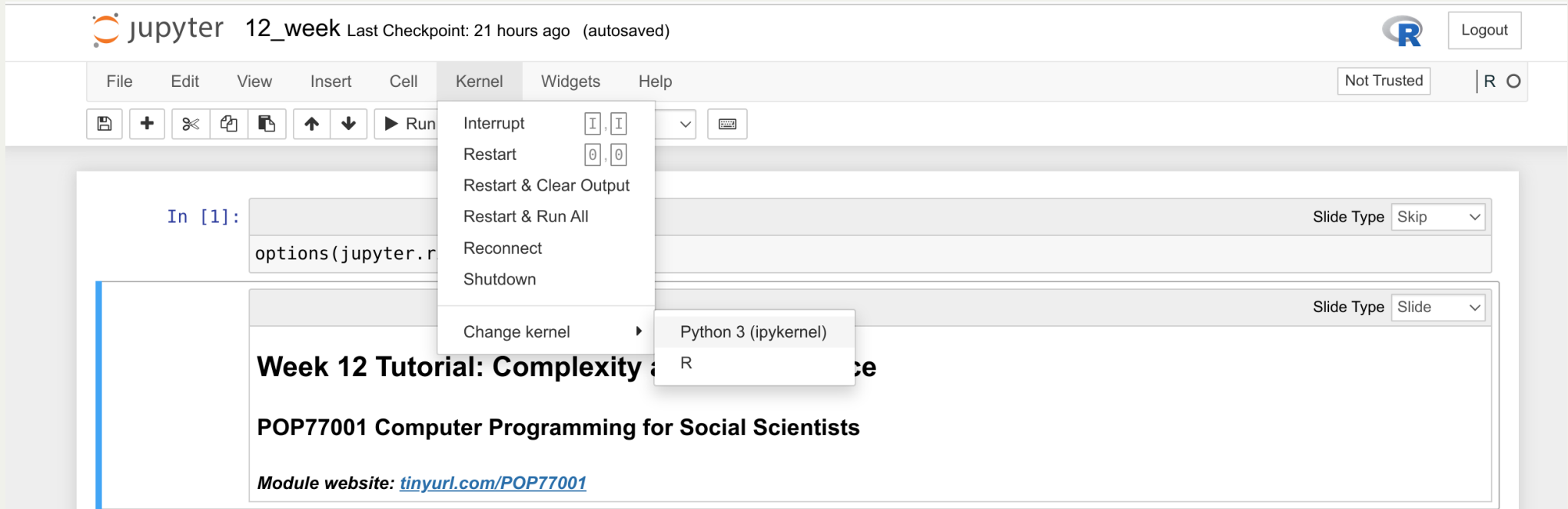
```
[1] 1000    50
```

# Benchmarking in Python

- It is possible to measure timing of operation in Python with built-in `time` module.
- But it would require recording time before a call and after and then taking a difference.
- Python's built-in `timeit` module provides a better alternative as it does it automatically and more.
- It behaves similar to `microbenchmark` in R in that it averages over many runs.
- It is also available in IPython (and, as a result, in Jupyter) as a magic command that can be called with `%timeit`.

# Switching kernels in Jupyter

- In order to be able to continue with Python part of the exercises you can switch your kernel.
- Got to **Kernel, Change kernel** and pick Python from the drop-down menu.



The screenshot shows the JupyterLab interface. At the top, the header displays the Jupyter logo, the text "jupyter 12\_week", and "Last Checkpoint: 21 hours ago (autosaved)". On the right, there is an R logo and a "Logout" button. Below the header is a menu bar with "File", "Edit", "View", "Insert", "Cell", "Kernel", "Widgets", and "Help". The "Kernel" menu is open, showing options: "Interrupt", "Restart", "Restart & Clear Output", "Restart & Run All", "Reconnect", "Shutdown", and "Change kernel". The "Change kernel" option is selected, opening a sub-menu with "Python 3 (ipykernel)" and "R". The background shows a code editor with a cell containing "In [1]: options(jupyter.r" and a slide viewer displaying "Week 12 Tutorial: Complexity" and "POP77001 Computer Programming for Social Scientists".

```
1 import random
2 import numpy as np
3 import pandas as pd
```

```
1 # Random numbers in Python can be generated either using
2 # the built-in `random` module or using `numpy` external
3 # module (which is underlying a lot of `pandas` operations)
4 random.gauss(mu = 0, sigma = 1)
```

0.15364381657690476

```
1 # Instead of just a float number it returns an array
2 np.random.randn(1)
```

array([0.06650689])



```
1 # Let's start our benchmarking experiments from looking
2 # at random number generation in Python.
3 # First let's draw a sample of 1M using both built-in `random` module
4 # And `numpy`'s methods
```

```
1 N = 1000000
```

```
1 # We can use `for _` expression to indicate that returned value is being di
2 %timeit [random.gauss(mu = 0, sigma = 1) for _ in range(N)]
```

invalid syntax (<string>, line 2)

```
1 # `numpy` is order of magnitude faster than built-in module
2 %timeit np.random.normal(size = N)
```

invalid syntax (<string>, line 2)

# Exercise: Compare Performance in Python

- Now let's replicate the calculation of some summary statistics in `pandas` `DataFrame`.
- As in the case of R, there are 2 principal ways of doing this:
  - First, is iterating over columns in a data set with a list comprehension and applying some function to each of columns (e.g. `mean()` from `statistics` module).
  - Alternatively, you can apply one of the built-in statistical summary methods (check Week 10 for the list).
- Apply each of those approaches to the data frame below.
- How do these two approaches compare?

```
1 from statistics import mean

1 # Setting seed using 'numpy' is slightly more involved than with 'random' m
2 # We first need to create a random number generator object, that we can tha
3 # to generate random draws from distributions that are consistent across re
4 rng = np.random.default_rng(1234)
5
6 # Here we are, essentially, replicating the process of data frame creation
7 # each variable is a random draw from a normal distribution with mean
8 # drawn from a uniform distribution between 0 and 10 and standard deviation
9 df2 = pd.DataFrame(np.concatenate([
10     rng.normal(loc = x, scale = 1, size = (1000, 1))
11     for x
12     in rng.uniform(low = 0, high = 10, size = 50)
13 ], axis = 1))
```

```
1 df2.shape
```

```
(1000, 50)
```

# Next

- Final project: Due by 23:59 on Friday, 13th December (submission on Blackboard)