Week 12 Tutorial: Complexity and Performance

POP77001 Computer Programming for Social Scientists

Module website: tinyurl.com/POP77001

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 returnes

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 returnes

In [2]:

```
library("microbenchmark")
```

Exercise 1: Compare performance

- Consider a data frame with 20 different variables below.
- We want to know the mean of each variable in the matrix.
- 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?

In [3]:

```
set.seed(2021)
# Here we create a data frame of 1000 observations of 50 variables
# where each variable is a random draw from a normal distribution with
# drawn from a uniform distribution between 0 and 10 and standard devia
dat <- data.frame(mapply(
  function(x) cbind(rnorm(n = 1000, mean = x, sd = 1)),
  runif(n = 50, min = 0, max = 10)
))
```

In [3]:

```
set.seed(2021)
# Here we create a data frame of 1000 observations of 50 variables
# where each variable is a random draw from a normal distribution with
# drawn from a uniform distribution between 0 and 10 and standard devia
dat <- data.frame(mapply(
  function(x) cbind(rnorm(n = 1000, mean = x, sd = 1)),
  runif(n = 50, min = 0, max = 10)
))
```

In [4]:

```
dim(dat)
```

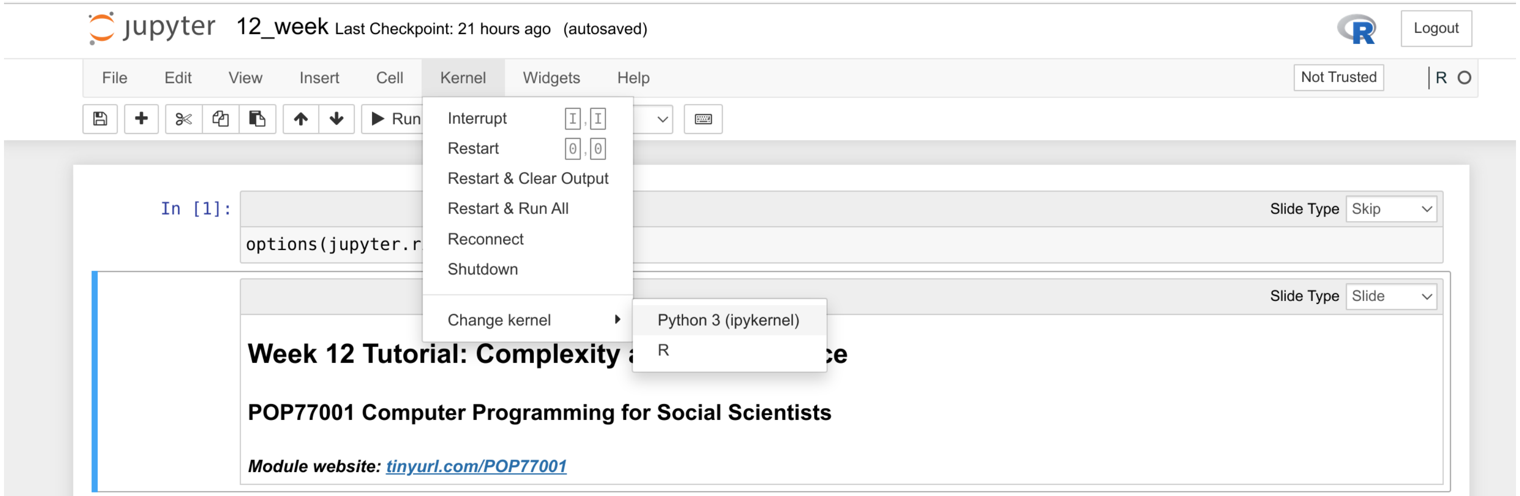
```
[1] 1000   50
```

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 an 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.



In [1]:

```
import random
import numpy as np
import pandas as pd
```

In [1]:

```
import random
import numpy as np
import pandas as pd
```

In [2]:

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

Out[2]:

```
-0.7261368325293743
```

In [1]:
```
import random
import numpy as np
import pandas as pd
```

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

Out[2]:
```
-0.726136832529374З
```

In [3]:
```
# Instead of just a float number it returns an array
np.random.randn(1)
```

Out[3]:
```
array([-0.88354514])
```

In [4]:

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

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

In [5]:
```
N = 1000000
```

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

In [5]:
```
N = 1000000
```

In [6]:
```
# We can use `for _` expression to indicate that returned value is bei
%timeit [random.gauss(mu = 0, sigma = 1) for _ in range(N)]
```

```
358 ms ± 2.62 ms per loop (mean ± std. dev. of 7 runs, 1 loop e
ach)
```

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

In [5]:
```
N = 1000000
```

In [6]:
```
# We can use `for _` expression to indicate that returned value is bei
%timeit [random.gauss(mu = 0, sigma = 1) for _ in range(N)]
```

```
358 ms ± 2.62 ms per loop (mean ± std. dev. of 7 runs, 1 loop e
ach)
```

In [7]:
```
# `numpy` is order of magnitude faster than built-in module
%timeit np.random.normal(size = N)
```

```
18.2 ms ± 99.3 µs per loop (mean ± std. dev. of 7 runs, 100 loo
ps each)
```

Exercise 2:

- 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?

In [8]:

```
from statistics import mean
```

In [8]:

```
from statistics import mean
```

In [9]:

```
# Setting seed using 'numpy' is slightly more involved than with 'rando
# We first need to create a random number generator object, that we car
# to generate random draws from distributions that are consistent acros
rng = np.random.default_rng(2021)

# Here we are, essentially, replicating the process of data frame creat
# each variable is a random draw from a normal distribution with mean
# drawn from a uniform distribution between 0 and 10 and standard devia
dat2 = pd.DataFrame(np.concatenate([
    rng.normal(loc = x, scale = 1, size = (1000, 1))
    for x
    in rng.uniform(low = 0, high = 10, size = 50)
], axis = 1))
```

In [8]:
```
from statistics import mean
```

In [9]:
```
# Setting seed using 'numpy' is slightly more involved than with 'rando
# We first need to create a random number generator object, that we car
# to generate random draws from distributions that are consistent acros
rng = np.random.default_rng(2021)

# Here we are, essentially, replicating the process of data frame creat
# each variable is a random draw from a normal distribution with mean
# drawn from a uniform distribution between 0 and 10 and standard devia
dat2 = pd.DataFrame(np.concatenate([
    rng.normal(loc = x, scale = 1, size = (1000, 1))
    for x
    in rng.uniform(low = 0, high = 10, size = 50)
], axis = 1))
```

In [10]:
```
dat2.shape
```

Out[10]:
```
(1000, 50)
```

Next

- Final project: Due at 12:00 on Monday, 19th December (submission on Blackboard)