2017 QuantEcon Workshops

Economic Modeling with Python and Julia

Introduction

August-September 2017

Personnel

- Matthew McKay (QuantEcon / ANU)
- Natasha Watkins (QuantEcon / ANU)
- John Stachurski (ARC / ANU) $\leftarrow$ **me**

Thanks to the Alfred P. Sloan Foundation

Workshop Timeline

1. **13:00-14:00** Introduction and First Steps
   - John Stachurski
2. 14:00-14:30 break
3. **14:30-15:30** Data Analysis with Python
   - Natasha Watkins
4. 15:30-16:00 break
5. **16:00-17:00** Advanced Data Analysis with Python
   - Matt McKay

Aims / Outcomes / Expectations

Please don't expect to learn Python + Julia + libraries + etc.

What we can do:

- Give an overview
- Show some examples
- Discuss
- Set you up to learn if you then wish to

Aims / Outcomes / Expectations

Please don't expect to learn Python + Julia + libraries + etc.

What we can do:

- Give an overview
- Show some examples
- Discuss
- Set you up to learn if you then wish to

Workshop Resources

Cheatsheets, downloads, code examples, etc.

All available from

`https://quantecon.org/unsw-workshop-2017`

Look for **Resources**

Get this PDF from the GitHub repo

- via `git` or the Download button

Downloads / Installation / Troubleshooting

⋆ **Python**

Step 1: Installation

- Install Anaconda from `https://www.continuum.io/downloads`
- Not plain vanilla Python

Step 2: Testing and next steps

Try following the instructions at `lectures.quantecon.org`

- Python → Setting up your Python Environment
- `https://lectures.quantecon.org/py/getting_started.html`

⋆ Julia

Step 1: Installation

- Install Julia `https://julialang.org/downloads`
- Not JuliaPro

Step 2: Testing and next steps

Try following the instructions at `lectures.quantecon.org`

- Julia → Setting up your Julia Environment
- `https://lectures.quantecon.org/jl/getting_started.html`

A Brief History of Programming

Languages are often graded as "high level" or "low level"

Low level languages give us

- **fine grained control** (hardware specific)
- **high execution speed**

High level languages give us

- **abstraction**
- **automation**
- **natural language representation**

A Brief History of Programming

Languages are often graded as "high level" or "low level"

Low level languages give us

- **fine grained control** (hardware specific)
- **high execution speed**

High level languages give us

- **abstraction**
- **automation**
- **natural language representation**

A Brief History of Programming

Languages are often graded as "high level" or "low level"

Low level languages give us

- **fine grained control** (hardware specific)
- **high execution speed**

High level languages give us

- **abstraction**
- **automation**
- **natural language representation**

Example. Computing $1 + 1$

In assembly:

```
pushq   %rbp
movq    %rsp, %rbp
movl    $1, -12(%rbp)
movl    $1, -8(%rbp)
movl    -12(%rbp), %edx
movl    -8(%rbp), %eax
addl    %edx, %eax
movl    %eax, -4(%rbp)
movl    -4(%rbp), %eax
popq    %rbp
```

In Python:

```
1 + 1
```

Example. : File IO

Task is to

1. open file `foo.txt`
2. print all lines

In C

```c
#include <stdio.h>

int main()
{
  FILE *in;
  char c;
  in = fopen("foo.txt", "r");
  if(in != NULL)
  {
    while((c = fgetc(in)) != EOF) putchar(c);

    fclose(in);
  }
  else printf("Unable to open file");
  return 0;
}
```

In Java

```
try (BufferedReader br =
     new BufferedReader(new FileReader("foo.txt"))) {
    String line = null;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
}
```

In Python

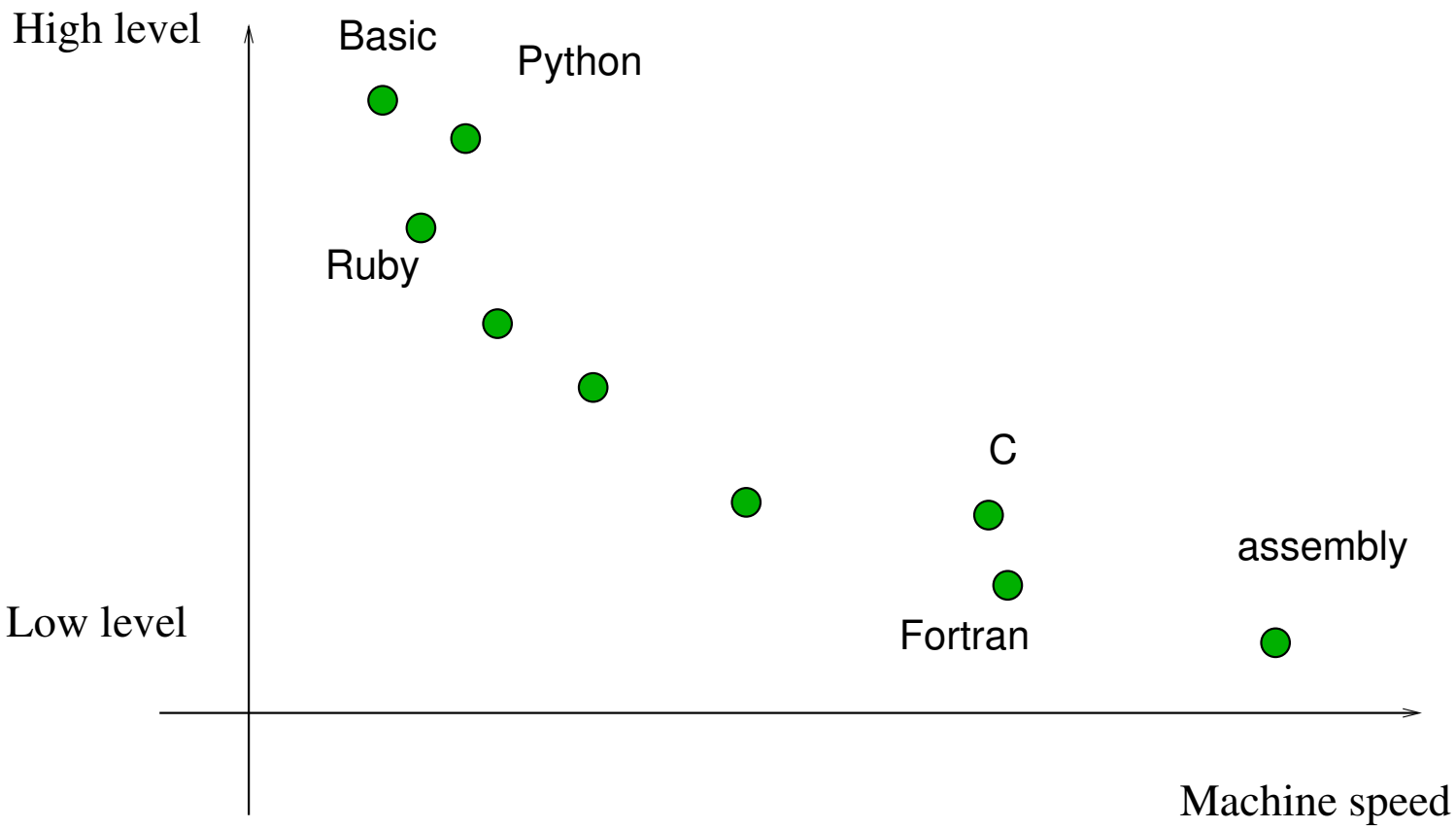
```
foo_file = open("foo.txt")
for line in foo_file
    print(line)
foo_file.close()
```

Trade-Offs

- **Low level** $\Longrightarrow$ hard for humans $\Longrightarrow$ **low productivity**
- **High level** $\Longrightarrow$ hard for machines $\Longrightarrow$ **slow execution**

Trade-Offs

- **Low level** $\Longrightarrow$ hard for humans $\Longrightarrow$ **low productivity**
- **High level** $\Longrightarrow$ hard for machines $\Longrightarrow$ **slow execution**


High level
Low level
Basic
Python
Ruby
C
Fortran
assembly
Machine speed

Trends

High level languages are ascendant

Why?

- computer time is cheap
- complexity is increasing
- fast moving technologies

Example. Instagram uses Python

- 600 million users, 400 million of whom are active daily
- 95 million images / videos **per day**

Performance speed is no longer the primary worry. Time to market speed is.

– Hui Ding, Instagram engineer

Jane Street on readability:

> There is no faster way for a trading firm to destroy itself than to deploy a piece of trading software that makes a bad decision over and over in a tight loop.
>
> Part of Jane Street's reaction to these technological risks was to put a very strong focus on building software that was easily understood—software that was readable.
>
> – Yaron Minsky, Jane Street

Example. Write a function that generates the time series

$$x_{t+1} = \alpha x_t + \beta + \sigma \xi_{t+1}$$

The first slide shows C code

Example. Write a function that generates the time series

$$x_{t+1} = \alpha x_t + \beta + \sigma \xi_{t+1}$$

The first slide shows C code

```c
int ar1(double *x, double *params, int n, unsigned long int seed)
{
    double beta = params[0];
    double alpha = params[1];
    double s = params[2];
    const gsl_rng_type * T;
    gsl_rng * r;
    gsl_rng_env_setup();
    T = gsl_rng_default;
    r = gsl_rng_alloc(T);
    gsl_rng_set(r, seed);

    x[0] = beta / (1 - alpha);
    for (int i = 1; i < n; i++)
     {
       x[i] = beta + alpha * x[i-1] + gsl_ran_gaussian(r, s);
     }
    gsl_rng_free (r);
    return 0;
}
```

Now here's the same thing in Julia

```julia
function ar1!(x, params, seed)

    srand(seed)
    α, β, σ = params

    for t in 1:(length(x)-1)

        ξ = randn()
        x[t+1] = α * x[t]  + β + σ * ξ

    end

end
```

Scientific Computing

Tasks

- Solve numerical problems
- Explore / visualize / simulate
- Manipulate data, estimate

Requirements

- Interactive — easy to explore
- Productive — easy to read, write, debug
- Fast computations

Scientific Computing

Tasks

- Solve numerical problems
- Explore / visualize / simulate
- Manipulate data, estimate

Requirements

- Interactive — easy to explore
- Productive — easy to read, write, debug
- Fast computations

In short, we want **both high speed and high productivity**

But can we have our cake and eat it too??

Let's review the state of play

In short, we want **both high speed and high productivity**

But can we have our cake and eat it too??

Let's review the state of play

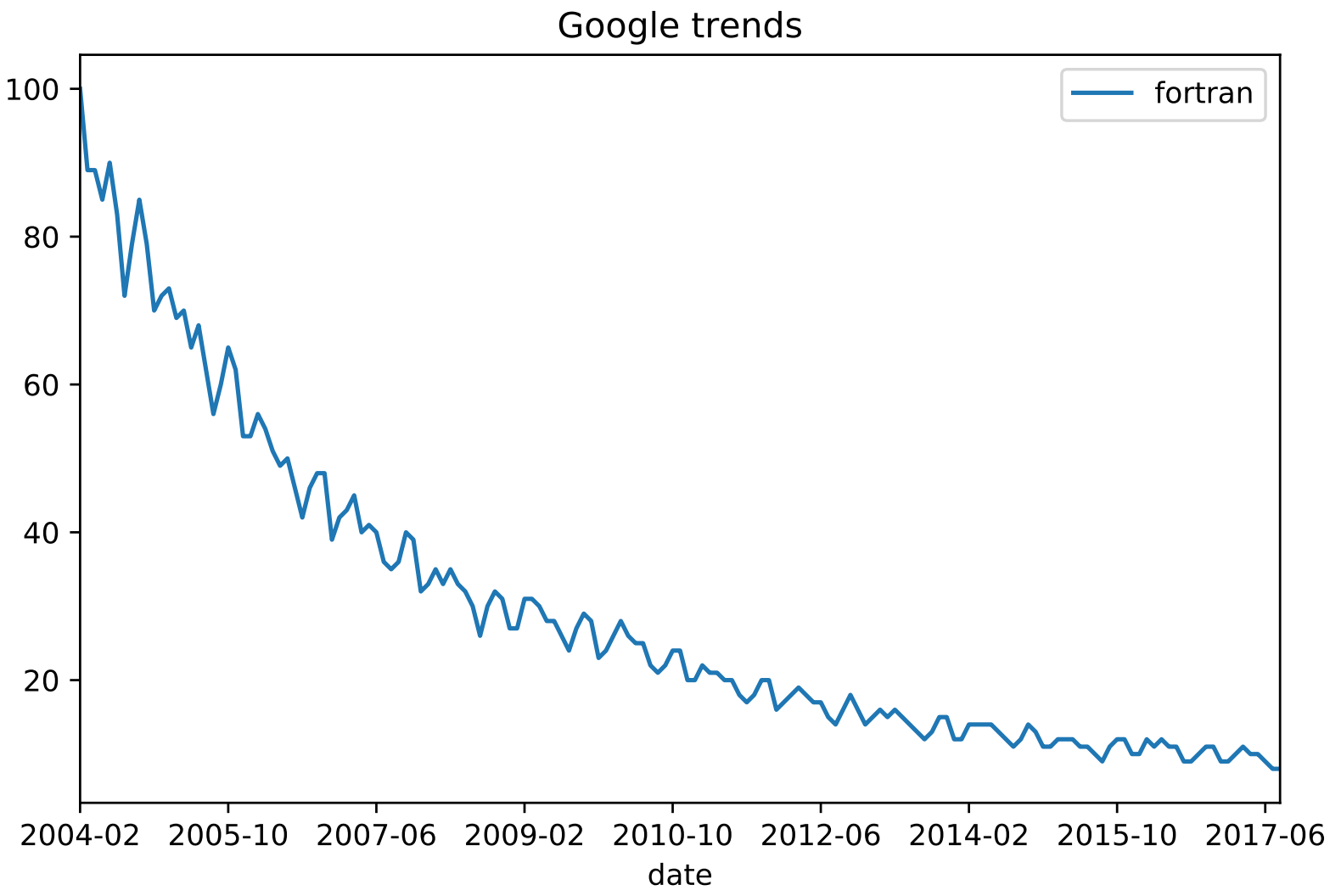
Generation 1: Low level languages

Example. Fortran, C, C++, Java

Still the backbone of many computations

But loosing popularity because

- Time consuming to write
- Error prone
- Lacking in interactivity


Google trends
fortran
100
80
60
40
20
2004-02
2005-10
2007-06
2009-02
2010-10
2012-06
2014-02
2015-10
2017-06
date

Generation 2: MATLAB and vectorization

1970s UNM prof Cleve Moler starts writing an **interactive front end to Fortran**

1984 MathWorks formed, first version released

Based around the idea of **vectorized** operations

- an interactive, productive environment
- send operations in batches to optimized precompiled C / Fortran

$\Longrightarrow$ High productivity and high speed combined

Similar approach: **Python + NumPy**

Example. Draw $X_1, \ldots X_n \sim N(0,1)$, compute $\max_i |X_i|$

```
import numpy as np
n = 1_000_000

# Non-vectorized code
m = -np.inf
for i in range(n):
    x = abs(np.random.randn())
    if x > m:
        m = x

# Vectorized code
m = max(abs(np.random.randn(n)))
```

Example. Linear algebra

```
import numpy as np

A = np.random.randn(2, 2)
b = np.ones(2)
np.linalg.solve(A, b)
```

See also GAUSS, STATA, etc.

Problems with Gen 2:

1. Vectorized array processing is actually kind of a hack
   - Example. The $\max_i |X_i|$ example
2. Many algorithms are hard to vectorized
   - Example. Consider simulating $x_{t+1} = \alpha x_t + \beta + \sigma \xi_{t+1}$

Problems with Gen 2:

1. Vectorized array processing is actually kind of a hack
   - Example. The $\max_i |X_i|$ example
2. Many algorithms are hard to vectorized
   - Example. Consider simulating $x_{t+1} = \alpha x_t + \beta + \sigma \xi_{t+1}$

Generation 3: Vectorization + JIT compilation

Gen 3 languages aim to let us have our cake and eat it

- interactive, high productivity environment
- vectorized operations
- and fast loops!

Examples: **Julia** and **Python + NumPy + Numba**

See `vectorization-py.ipynb` and `vectorization-jl.ipynb`

Generation 3: Vectorization + JIT compilation

Gen 3 languages aim to let us have our cake and eat it

- interactive, high productivity environment
- vectorized operations
- and fast loops!

Examples: **Julia** and **Python + NumPy + Numba**

See `vectorization-py.ipynb` and `vectorization-jl.ipynb`

Exercises

Option 1 generate some plots in either Python or Julia

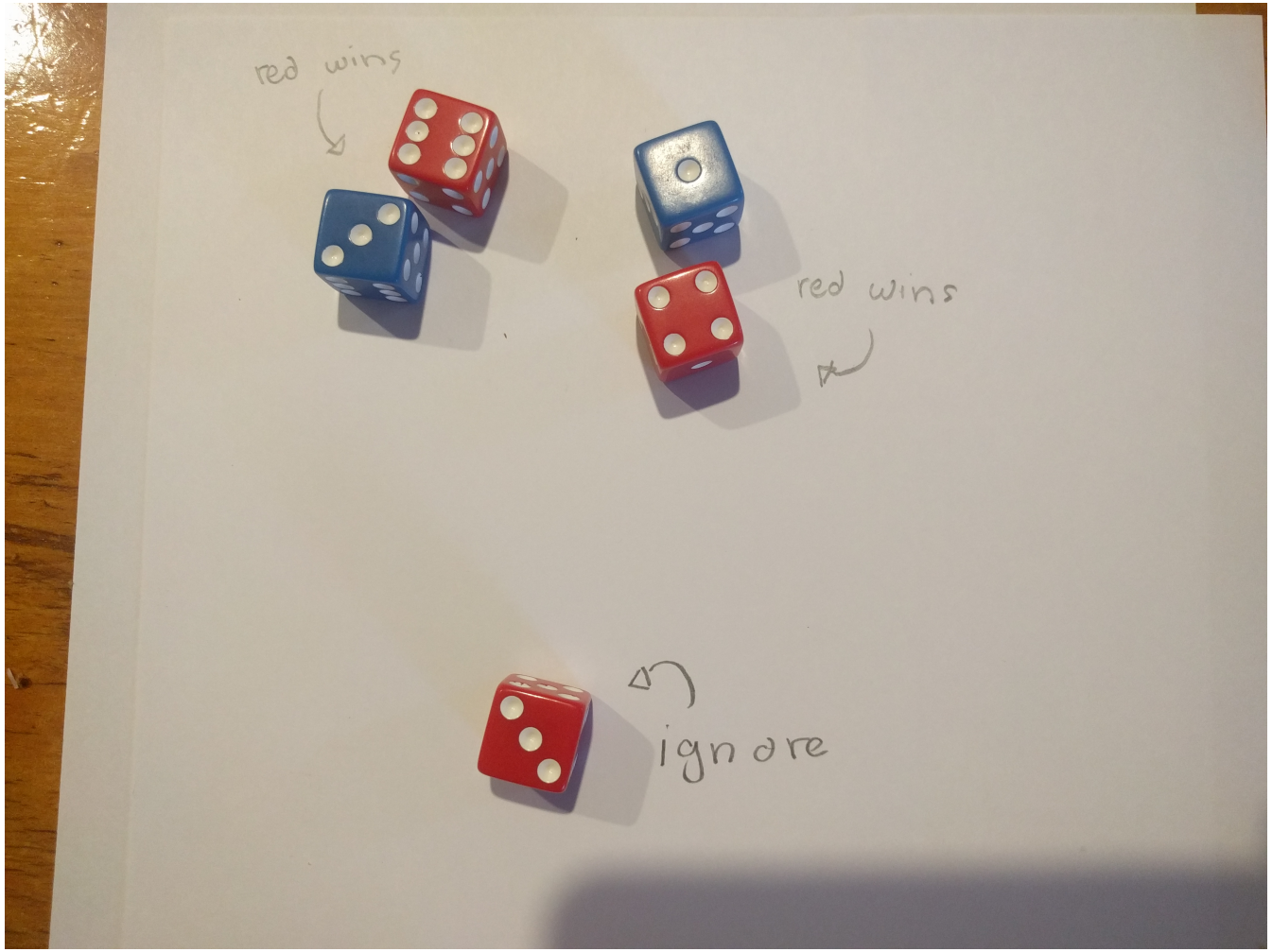
- for Python see `plots/plots-py.ipynb`
- for Julia see `plots/plots-jl.ipynb`

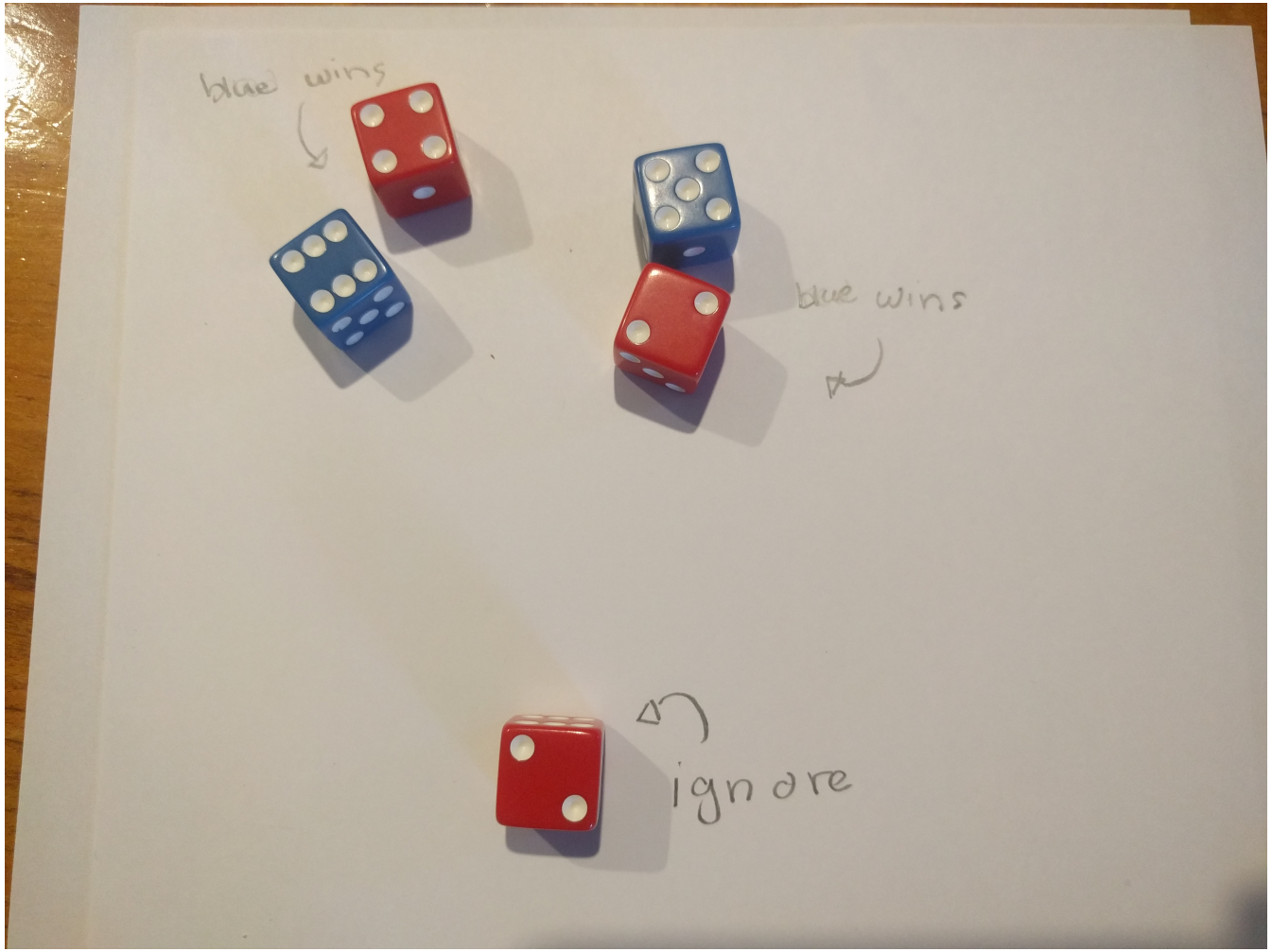
Option 2 Calculating probabilities for Risk

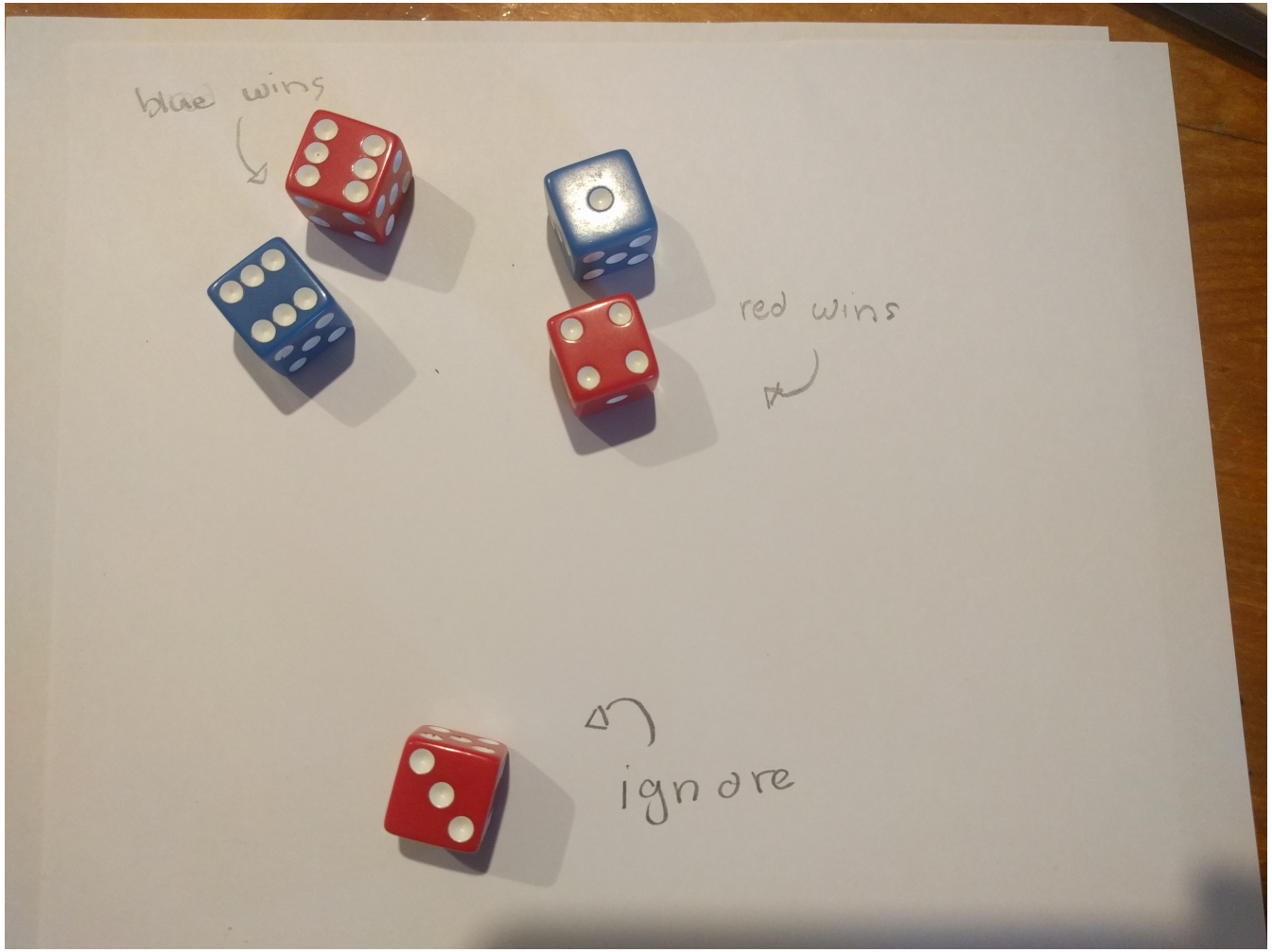
- for Python see `risk/risk-py.ipynb`
- for Julia see `risk/risk-jl.ipynb`

Risk
Risk
Risk
Risk
Risk
Atlantic Ocean
Pacific Ocean
Indian Ocean
Southern Ocean
Arctic Ocean

- Attacker rolls 3 dice, defender rolls 2


red wins
red wins
ignore


blue wins
blue wins
ignore


blue wins
red wins
ignore

Steps:

1. Discard lowest red
2. Pair highest red with highest blue
   - If $R1 > B1$, then red wins one soldier
   - else blue wins one
3. Pair second highest red with second highest blue
   - If $R2 > B2$, then red wins one soldier
   - else blue wins one

By simulating many rolls, compute the probabilities of

- red wins 2 soldiers
- blue wins 2 soldiers
- one each