

Programming best practices

Agent-based modelling, Konstanz, 2024

Henri Kauhanen

21 May 2024

Plan

- This week, we will look at a few practices that have the potential to make your code better
- Here, “better” can mean:
 - more logical organization of code
 - better performance (faster running, and/or less memory consumption)
- In addition, we will wrap up the first half of the course and talk about any issues/challenges you may have run into

Note

Today’s lecture requires the following Julia packages:

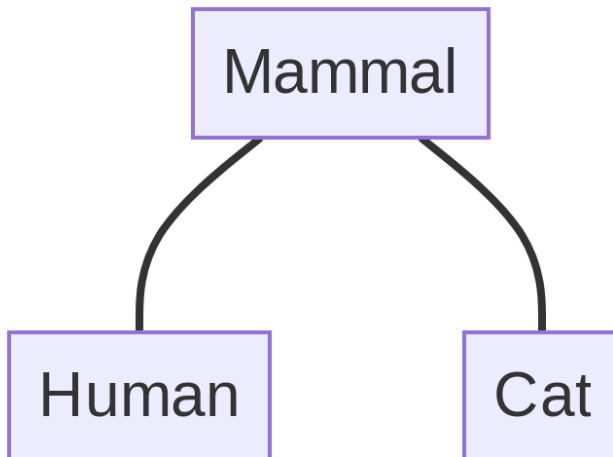
- Agents
- BenchmarkTools
- Random

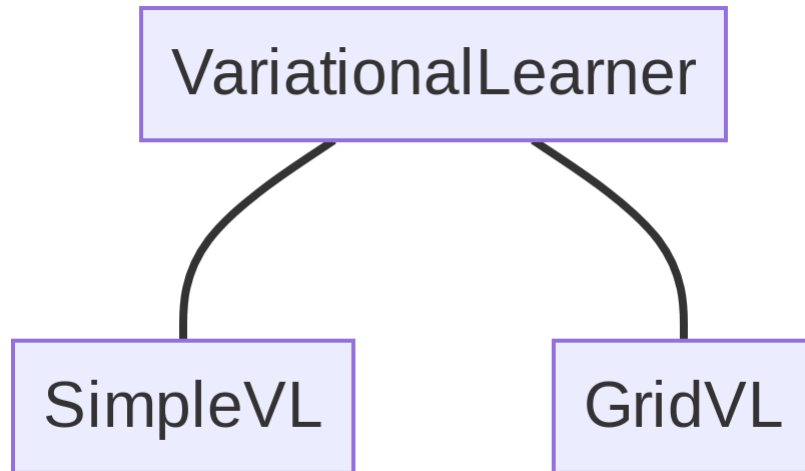
It would be a good idea to install them now, if your system does not already have them.

- The “best practices” bit of today’s session is broken down into three major topics:
 1. Abstract types, inheritance and multiple dispatch
 2. Benchmarking
 3. Random numbers

Abstract types and inheritance

- Some weeks ago, we defined a variational learner type which lives in an unstructured population
- Last week, we defined one that lives in a grid space
- Two possible strategies:
 1. Name both types `VariationalLearner`
 - pro: we can reuse the functions we've written that take `VariationalLearner` objects as arguments, such as `speak`, `learn!` and `interact!`
 - con 1: we can't use both types in the same code
 - con 2: Julia does not deal well with type redefinitions, forcing a restart when moving from one definition to the other
 2. Give the new type a new name, such as `GridVL`
 - pro: no complaints from Julia
 - con: we can't reuse our functions, since they're defined for `VariationalLearner` objects
- A neat solution to this problem is to start thinking about **type hierarchies**
- Intuitively: types can have hierarchical relationships, a bit like biological taxonomies





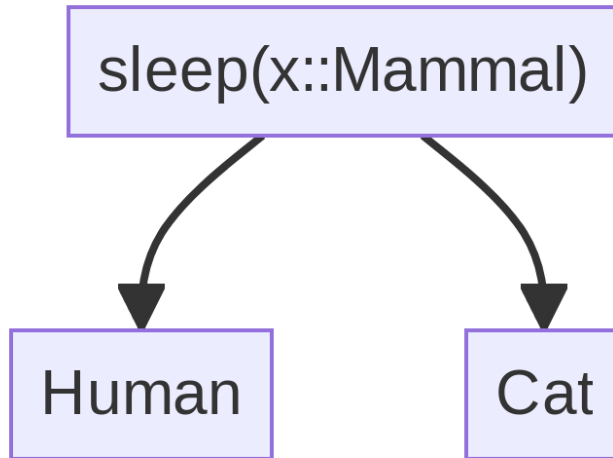
! Important

From now on, I will use `SimpleVL` to refer to our original `VariationalLearner`, i.e. the type that lives in an unstructured population.

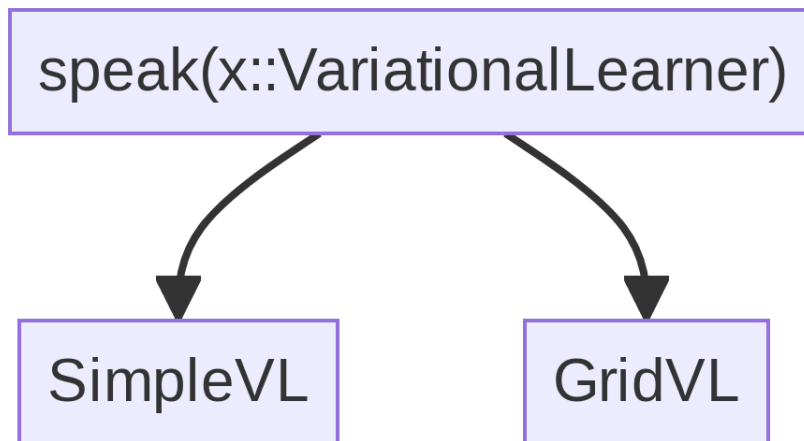
`VariationalLearner` from now on will denote the **supertype** of all “variational learnery” things.

Abstract types and inheritance

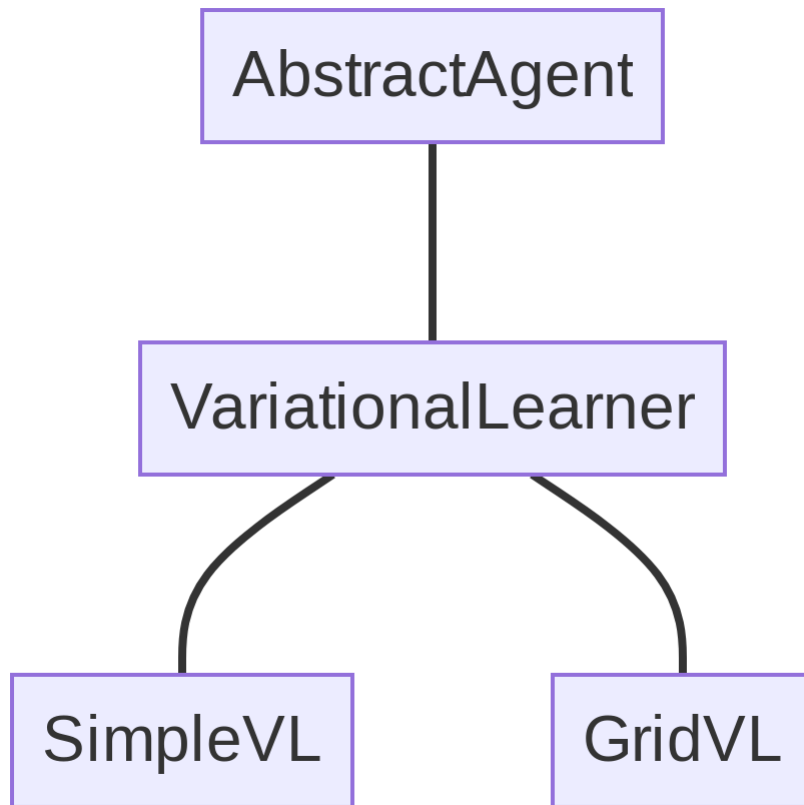
- The point of this is: a function can be defined for the supertype, whose subtypes then **inherit** that function
- E.g. we can define a `sleep` function for `Mammal`
- Both `Human` and `Cat` inherit this function, and so we don’t need to define one for them separately



- Similarly, we can define `speak` for the supertype `VariationalLearner`
- Then both `SimpleVL` and `GridVL` have access to this function



- In Julia such “supertypes” are known as **abstract types**
- They have no fields; they only exist to define the type hierarchy
- Inheritance relations are defined using a special `<:` operator
- To use `Agents.jl`, our `VariationalLearner` abstract type itself needs to inherit from `AbstractAgent`



```
abstract type VariationalLearner <: AbstractAgent end

mutable struct SimpleVL <: VariationalLearner
    # code goes here...
end

@agent struct GridVL(GridAgent{2}) <: VariationalLearner
    # code goes here...
end

function speak(x::VariationalLearner)
    # code goes here...
end
```

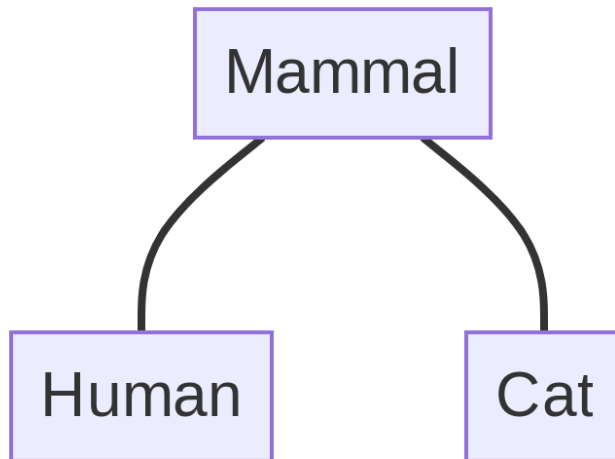
- We can now do things like:

```
bob = SimpleVL(0.1, 0.01, 0.4, 0.1)

speak(bob)
```

even though `speak` wasn't defined for `SimpleVL`

Multiple dispatch



- What if `Cat` needs to sleep differently from other `Mammals`?
- Easy: we simply define a function `sleep(x::Cat)`
- Other `Mammals` will use the default function `sleep(x::Mammal)`
- In Julia, this is called **multiple dispatch**
- One and the same function (here, `sleep`) can have multiple definitions depending on the argument's type
- These different definitions are known as **methods** of the function
- When figuring out which method to use, the compiler tries to apply the method that is deepest in the type hierarchy, moving upwards if such a definition isn't found
 - e.g. in our example calling `sleep` on `Human` will trigger the `sleep` method defined for `Mammal`, since no `sleep` method specific to `Human` has been defined

Our VL code so far

Tip

You can also download this code: [VL.jl](#). To use:

```
include("VL.jl")
using .VL
```

If VSCode complains about modules, simply delete the first and last lines of the file and include it like so:

```
include("VL.jl")
```

```
module VL

# Agents.jl functionality
using Agents

# we need this package for the sample() function
using StatsBase

# we export the following types and functions
export VariationalLearner
export SimpleVL
export GridVL
export speak
export learn!
export interact!
export VL_step!

# abstract type
abstract type VariationalLearner <: AbstractAgent end

# variational learner type on a 2D grid
@agent struct GridVL(GridAgent{2}) <: VariationalLearner
    p::Float64      # prob. of using G1
    gamma::Float64  # learning rate
    P1::Float64     # prob. of L1 \ L2
    P2::Float64     # prob. of L2 \ L1
end
```

```

# "simple" variational learner in unstructured population
mutable struct SimpleVL <: VariationalLearner
    p::Float64      # prob. of using G1
    gamma::Float64  # learning rate
    P1::Float64     # prob. of L1 \ L2
    P2::Float64     # prob. of L2 \ L1
end

# makes variational learner x utter a string
function speak(x::VariationalLearner)
    g = sample(["G1", "G2"], Weights([x.p, 1 - x.p]))

    if g == "G1"
        return sample(["S1", "S12"], Weights([x.P1, 1 - x.P1]))
    else
        return sample(["S2", "S12"], Weights([x.P2, 1 - x.P2]))
    end
end

# makes variational learner x learn from input string s
function learn!(x::VariationalLearner, s::String)
    g = sample(["G1", "G2"], Weights([x.p, 1 - x.p]))

    if g == "G1" && s != "S2"
        x.p = x.p + x.gamma * (1 - x.p)
    elseif g == "G1" && s == "S2"
        x.p = x.p - x.gamma * x.p
    elseif g == "G2" && s != "S1"
        x.p = x.p - x.gamma * x.p
    elseif g == "G2" && s == "S1"
        x.p = x.p + x.gamma * (1 - x.p)
    end

    return x.p
end

# makes two variational learners interact, with one speaking
# and the other one learning
function interact!(x::VariationalLearner, y::VariationalLearner)
    s = speak(x)
    learn!(y, s)
end

```



```
# steps a model
function VL_step!(agent, model)
    interlocutor = random_nearby_agent(agent, model)
    interact!(interlocutor, agent)
end

end # this closes the module
```

Benchmarking

- When working on larger simulations, it is often important to know how long some function takes to run
- It may also be important to know how much memory is consumed
- Both of these things can be measured using the `@benchmark` macro defined by [BenchmarkTools.jl](#)
- Example:

```
using BenchmarkTools

@benchmark sum(1:1_000_000_000)
```

BenchmarkTools.Trial: 10000 samples with 1000 evaluations.

| | | | |
|----------------------|------------------------|-------------------|-----------------|
| Range (min ... max): | 1.336 ns ... 22.348 ns | GC (min ... max): | 0.00% ... 0.00% |
| Time (median): | 1.528 ns | GC (median): | 0.00% |
| Time (mean ±): | 1.600 ns ± 0.520 ns | GC (mean ±): | 0.00% ± 0.00% |

1.34 ns Histogram: frequency by time 2.45 ns <

Memory estimate: 0 bytes, allocs estimate: 0.

All roads lead to Rome, but they're not all equally fast...

- Suppose we want to calculate the square root of all numbers between 0 and 100,000 and put them in an array
- One way of doing this:

```

result = [] # empty array
for x in 0:100_000
    append!(result, sqrt(x)) # put  $\sqrt{x}$  in array
end

```

```

@benchmark begin
    result = [] # empty array
    for x in 0:100_000
        append!(result, sqrt(x)) # put  $\sqrt{x}$  in array
    end
end

```

BenchmarkTools.Trial: 2946 samples with 1 evaluation.

| | | | |
|----------------------|-----------------------|-------------------|------------------|
| Range (min ... max): | 1.181 ms ... 6.330 ms | GC (min ... max): | 0.00% ... 67.11% |
| Time (median): | 1.517 ms | GC (median): | 0.00% |
| Time (mean ±): | 1.693 ms ± 564.011 s | GC (mean ±): | 8.64% ± 14.58% |

1.18 ms Histogram: log(frequency) by time 4.2 ms <

Memory estimate: 3.35 MiB, allocs estimate: 100012.

- Another way:

```

result = zeros(100_000 + 1)
for x in 0:100_000
    result[x+1] = sqrt(x) # put  $\sqrt{x}$  in array
end

```

```

@benchmark begin
    result = zeros(100_000 + 1)
    for x in 0:100_000
        result[x+1] = sqrt(x) # put  $\sqrt{x}$  in array
    end
end

```

BenchmarkTools.Trial: 10000 samples with 1 evaluation.

| | | | |
|----------------------|-------------------------|-------------------|------------------|
| Range (min ... max): | 100.031 s ... 675.456 s | GC (min ... max): | 0.00% ... 47.55% |
| Time (median): | 103.037 s | GC (median): | 0.00% |
| Time (mean ±): | 115.640 s ± 46.187 s | GC (mean ±): | 4.09% ± 8.74% |

100 s Histogram: log(frequency) by time 386 s <

Memory estimate: 781.36 KiB, allocs estimate: 2.

- A third possibility:

```
result = [sqrt(x) for x in 0:100_000]
```

```
@benchmark result = [sqrt(x) for x in 0:100_000]
```

BenchmarkTools.Trial: 10000 samples with 1 evaluation.

Range (min ... max): 78.260 s ... 655.362 s GC (min ... max): 0.00% ... 50.67%

Time (median): 79.524 s GC (median): 0.00%

Time (mean ±): 87.558 s ± 32.717 s GC (mean ±): 3.75% ± 8.62%

78.3 s Histogram: log(frequency) by time 282 s <

Memory estimate: 781.36 KiB, allocs estimate: 2.

- A fourth way:

```
result = sqrt.(0:100_000)
```

```
@benchmark result = sqrt.(0:100_000)
```

BenchmarkTools.Trial: 10000 samples with 1 evaluation.

Range (min ... max): 78.419 s ... 690.822 s GC (min ... max): 0.00% ... 45.35%

Time (median): 79.265 s GC (median): 0.00%

Time (mean ±): 87.849 s ± 36.867 s GC (mean ±): 4.04% ± 8.78%

78.4 s Histogram: log(frequency) by time 310 s <

Memory estimate: 781.36 KiB, allocs estimate: 2.

Summing up the findings

| Procedure | Median time | Mem. estimate |
|---------------------|-------------|---------------|
| Growing an array | ~1.4 ms | ~3.4 MiB |
| Adding to 0-array | ~0.1 ms | ~0.8 MiB |
| Array comprehension | ~80 μ s | ~0.8 MiB |
| Broadcasting | ~80 μ s | ~0.8 MiB |

- Lesson: try to avoid growing (and shrinking!) arrays whenever possible
- Of course, sometimes this is practically unavoidable (such as when adding and removing agents from a population)
- Another lesson: if procedure X gets repeated very many times in a simulation, try to make X as efficient as possible
 - Procedures which are only carried out once or a few times (such as initializing a population) don't matter so much

Random numbers

- In the [first lecture](#), we talked about the importance of (pseudo)random numbers in ABM simulations
- E.g. whenever an agent needs to be sampled randomly, the computer needs to generate a random number
- There are two important issues here:
 1. Reproducibility – how to obtain the same sequence of “random” numbers if this is desired
 2. Consistency – making sure that whenever a random number is drawn, it is drawn using the same generator (i.e. from the same sequence)

Reproducibility

- Recall: a PRNG (pseudorandom number generator) generates a **deterministic** sequence which appears random
- The sequence is generated from an initial **seed** number
- If you change the seed, you obtain different sequences
- Normally, when Julia is started, the PRNG is seeded with a different seed every time
 - Hence, you obtain different sequences

Reproducibility: illustration

- To illustrate this, suppose you want to toss a coin 10 times. This is easy:

```
rand(["heads", "tails"], 10)
```

```
10-element Vector{String}:
```

```
"tails"  
"tails"  
"tails"  
"heads"  
"tails"  
"heads"  
"heads"  
"tails"  
"tails"  
"heads"
```

- Now restart Julia and execute the same thing. You will get a different result:

```
# here, restart Julia...
```

```
rand(["heads", "tails"], 10)
```

```
10-element Vector{String}:
```

```
"tails"  
"tails"  
"heads"  
"heads"  
"heads"  
"heads"  
"tails"  
"tails"  
"tails"  
"heads"
```

- If you want to make sure the exact same sample is obtained, you can seed the PRNG manually after startup
- For example, seed with the number 123:

```
using Random
Random.seed!(123)

rand(["heads", "tails"], 10)
```

```
10-element Vector{String}:
 "tails"
 "tails"
 "tails"
 "heads"
 "tails"
 "heads"
 "heads"
 "tails"
 "tails"
 "heads"
```

Reproducibility

- Why would you do this? Wasn't randomness kind of the point?
- Suppose someone (e.g. your supervisor, or an article reviewer) wants to check that your code actually produces the results you have reported
- Using a manually seeded PRNG makes this possible

Consistency

- It is possible to have multiple PRNGs running simultaneously in the same code
- This is rarely desired, but may happen by mistake...
- For example, when you call `StandardABM`, `Agents.jl` will set up a new PRNG by default
- If your own functions (such as `speak` or `learn!`) utilize a different PRNG, you may run into problems
 - For one, it will be difficult to ensure reproducibility
- To avoid this, pass `Random.default_rng()` as an argument to `StandardABM` when creating your model:

```
using Agents
using Random
include("VL.jl")
using .VL
```

```
Random.seed!(123)

space = GridSpace((10, 10))

model = StandardABM(GridVL, space; agent_step! = VL_step!,
                    rng = Random.default_rng())
```

Reminder: not all agents are humans!

<https://youtu.be/UzgMw3SJn2s>

Looking ahead

- Homework:
 1. **Keep thinking about your project!**
 2. Read Smaldino (2023), chapter 10
- The following two weeks constitute a break for us: the first one is the lecture-free period, the second one is consolidation week (“Vertiefungswoche”)
- After this break, you need to
 1. have a project team (or have decided to work on your own)
 2. have at least an initial idea about your project topic
- If you struggle, I’m happy to help! You can always write me an email, and/or come to see me in my office.

Smaldino, Paul E. 2023. *Modeling Social Behavior: Mathematical and Agent-Based Models of Social Dynamics and Cultural Evolution*. Princeton, NJ: Princeton University Press.