Antipatterns and idiomatic Julia versions

Fredrik Bagge Carlson
fredrik.bagge@nus.edu.sg

October 17, 2019

Introduction

How to use multiple dispatch to structure your code

Some common patterns and anti-patterns

How to use multiple dispatch to structure your code

Some common patterns and anti-patterns

Code reuse

- Julia is excellent on code reuse
  - Interoperability between packges and types
  - In your own code
- Requires good and thoughtful design!

Dispatch on types

Antipattern

```
class Shape:
    @property
    def name(self):
        return self.__class__

class Rectangle(Shape): pass
class Ellipse(Shape): pass

def intersect(s1, s2):
    if isinstance(s1, Rectangle) and isinstance(s2, Ellipse):
        print('Rectangle x Ellipse [names s1=%s, s2=%s]' % (s1.name, s2.name))
    elif isinstance(s1, Rectangle) and isinstance(s2, Rectangle):
        print('Rectangle x Rectangle [names s1=%s, s2=%s]' % (s1.name, s2.name))
    else:
        # Generic shape intersection.
        print('Shape x Shape [names s1=%s, s2=%s]' % (s1.name, s2.name))
```

Drawback: To implement a new geometry, the user has to modify the source code pf `intersect`

Can only dispatch based on one argument

Dispatch on types

Multiple dispatch approach

```julia
abstract type AbstractShape end
struct Rectangle <: AbstractShape end
struct Ellipse <: AbstractShape end

intersect(s1::Rectangle, s2::Ellipse) = do_something...
intersect(s1::Rectangle, s2::Rectangle) = do_something...
intersect(s1, s2) = do_something...
```

Benefit: The user can send in an arbitrary existing subtype of `AbstractShape` as well as creating new such subtypes without modifying the function `intersect`.

Can dispatch based on both types.

Dispatch on strings

Antipattern

```
function process(data, preprocess="sumto1")
    if preprocess == "sumto1"
        data = data ./ sum(data)
    elseif preprocess == "norm1"
        data = data ./ norm(data)
    elseif preprocess == "filter"
        data = filter(data)
    end
    do_something_with(data)
end
```

Drawback: To implement a new form of preprocessing, the user has to modify the source code pf `process`

Dispatch on strings

Functional approach

```julia
function process(data, preprocess)
    data = preprocess(data)
    do_something_with(data)
end
```

Benefit: The user can send in an arbitrary function

Dispatch on strings

Multiple dispatch approach

```julia
abstract type AbstractPreprocesser end

struct SumToOne <: AbstractPreprocesser end
preprocess(::SumToOne, data) = data ./ sum(data)

struct NormOne <: AbstractPreprocesser end
preprocess(::NormOne, data) = data ./ norm(data)

struct Filter <: AbstractPreprocesser
    cutoff
end
preprocess(f::Filter, data) = filter(data, f.cutoff)

function process(data, processor)
    data = preprocess(processor, data)
    do_something_with(data)
end
```

Benefit: The user can send in an arbitrary existing subtype of `AbstractPreprocesser` as well as creating new such subtypes without modifying the function `preprocess`.

Several names for the same function

Antipattern

```
function optimize_gd(problem, x)
    for i in iterations
        x = take_gd_step(x, problem)
        convergence_check_x(x) && break
        convergence_check_gradient(x, problem) && break
        convergence_check_funval(x) && break
    end
    x
end
function optimize_bfgs(problem, x)
    for i in iterations
        x = take_bfgs_step(x, problem)
        convergence_check_x(x) && break
        convergence_check_gradient(x, problem) && break
        convergence_check_funval(x) && break
    end
    x
end
function optimize_newton(problem, x)
    for i in iterations
        x = take_newton_step(x, problem)
        convergence_check_x(x) && break
        convergence_check_gradient(x, problem) && break
        convergence_check_funval(x) && break
```

Drawback: The *function* is the same, it find the optimum, what differs is the *method*.

The different names are only used for dispatch. Lots of code is repeated, all algorithms have the same structure.

Several names for the same function

Multiple dispatch approach

```julia
abstract type AbstractOptimizer end

struct GradientDescent <: AbstractOptimizer
    stepsize
end
struct BFGS <: AbstractOptimizer end
struct Newton <: AbstractOptimizer
    preconditioner
end

step(::GradientDescent, x, problem) = take_gd_step(x, problem)
step(::BFGS, x, problem) = take_bfgs_step(x, problem)
step(f::Newton, x, problem) = take_newton_step(x, problem)

function process(problem, x, algorithm)
    for i in iterations
        x = step(algorithm, x, problem)
        convergence_check_x(x) && break
        convergence_check_gradient(x, problem) && break
        convergence_check_funval(x) && break
    end
    x
end
```

Benefit: The outer algorithm is on implemented once, the behaviour differs is the step, dispatched usir

No need to copy the code in outer to implement a new optimization r

Other antipatterns

Antipatterns

```julia
repmat(v,1,10) .* A
for i ∈ collect(1:100)
randn(10,1)
0
Vector{typeof(x)}(undef, length(x))
mean(x.^2)
f(x::Float64) = x^2
middle(x::Vector) = x[end÷2]
```

Functions that operate elementwise over arrays

Rolling your own types

Nice patterns

```julia
v .* A
for i ∈ 1:100
randn(10)
zero(x)
similar(x)
mean(abs2, x)
f(x) = x^2
middle(x::AbstractVector) = x[end÷2]
```

Broadcast a scalar function

Reusing ecosystem types

Common Julia idioms and types

```
condition && return x
```

Packages implementing commonly used types

- Colors.jl
- Distances.jl
- RecipesBase.jl
- ChainRules.jl