

Using Julia Well

perspectives, practices, pragmatics

Jeffrey A Sarnoff

Julia Innovator

Julia Users Group NYC meetup

Thursday, 2022-Aug-28



Being around people here deeply deeply accelerates the learning process.
A lot of really cool folks here did the same for me ... and I am eternally grateful for their guidance. Glad you got to do the same 😊 .

– otde on Zulip



Learn from us

The Julia Community is welcoming, helpful, and self-respecting.

Your question is a good one. Ask. We will not think less of you.

- The people who are experts now **all** asked questions.

do see if the answer is readily available (docs, Discourse, Zulip, SO)

ask about technique, ask to clarify, ask for explanation

- where to ask: Discourse, Zulip, Slack
- how to ask: discourse.julialang.org/t/please-read-make-it-easier-to-help-you/14757
- what to ask: explain what it is that you want to know, what you seek to have happen



with Julia

To consider Julia merely a programming language is to lose advantage.

Enhance your own effectiveness

look for ways that simplify, clarify, and engage ... use them often.

Elevate aspects of your professional style

Read your own work, even when it works correctly – especially then.

Explain with words and design. Persuade with code. Convince with tests.



with Julia

To consider Julia merely a programming language is to lose advantage.

Enhance your own effectiveness

look for ways that simplify, clarify, and engage ... use them often.

keep it simple. get it working. note what you want it to be doing.

clear away the overdone. revisit, reflow. only then address speed.

Elevate aspects of your professional style

Read your own work, even when it works correctly – especially then.

Explain with words and design. Persuade with code. Convince with tests.



Tuples

Tuples are one of the core datatypes in Julia.

- They should be relatively small
 - ≤ 32 items is optimized in all sorts of ways
 - ≤ 64 items is optimized in important ways
- They are most performant when of uniform concrete type if that is a bitstype, so much the better
- They are still worthwhile when of different concrete types
 - if there are ≤ 3 different concrete types, good things happen (really its ≤ 4 different concrete types, but think of it as 3)
- Tuples look like this
 - `()` `(1,)` `(1, 2)` `("abc", pi)`



NamedTuples

- All the fun of Tuples, now with names enfolded.
- More trustworthy: `routing.destination` is more helpful than `routing[2]`
- More easily shared, maintained, **explained**



NamedTuples

- All the fun of Tuples, now with names enfolded.
- More trustworthy: `routing.destination` is more helpful than `routing[2]`
- More easily shared, maintained, **explained**

```
> emily_rey = (firstname = "Emily", lastname = "Rey", badge = 12)
(firstname = "Emily", lastname = "Rey", badge = 12)
```

```
> emily.firstname, emily[:firstname], emily[1]
("Emily", "Emily", "Emily")
```

What about people who are not Emily?



NamedTuples at a party with the cool kids

```
newhire( firstname, lastname, badge ) =  
    ( ; firstname, lastname, badge )
```

```
emily_rey = newhire( "Emily", "Rey", 12 )  
    (firstname = "Emily",  
      lastname = "Rey",  
      badge = 12 )
```



NamedTuples

```
emily_rey == (firstname = "Emily", lastname = "Rey", badge = 12)
```

```
struct NewHire{AkoString}  
  firstname::AkoString  
  lastname::AkoString  
  badge::Int  
end
```

```
NewHire(nt::NamedTuple) = NewHire( nt... )
```

```
EmilyRey = NewHire(emily_rey)
```

```
EmilyRey.badge == emily_rey.badge
```



NamedTuples

```
> well_paid_employee = newhire( "Emily", "Rey", 12 )  
( firstname = "Emily", lastname = "Rey", badge = 12 )  
  
> firstname, lastname, badge = well_paid_employee  
( "Emily", "Rey", 12 )  
  
> keys( well_paid_employee )  
( :firstname, :lastname, :badge )  
  
> values( well_paid_employee )  
( "Emily", "Rey", 12 )
```



NamedTupleTools

```
> using NamedTupleTools

> select(employee, (:firstname, :lastname))
(firstname = "Emily", lastname = "Rey")

> delete(employee, :badge)
(firstname = "Emily", lastname = "Rey")

> id, name = split(employee, :badge)
((badge = 12,), (firstname = "Emily", lastname = "Rey"))

> merge(id, name)                                     # create new
(badge = 12, firstname = "Emily", lastname = "Rey")
```



write clean code

not there yet, rewrite it (developing good habits)

not sure what to do, look at other solutions or **ask**

iteration

```
for current_value in xs .. end      # avoid index nums
for current_index in eachindex(xs) .. end  # these are fast
for (index, value) in enumerate(xs) .. end  # and future proof
for current_column in eachcol(amat) .. end  # prefer bycol 2x+
```

lazy comprehension

```
ys = (x^2 for x in xs) # Base.Generator{Vector{Int64}, ..
zs = zip(xs, ys)       # zip is lazy and surprisingly fast
```



Integers

overflow and underflow happen when Int types wrap

```
> typemin(Int8), typemax(Int8)  
(-128, 127)
```

```
> typemin(Int8) - one(Int8), typemax(Int8) + one(Int8)  
( 127, -128)
```

What to do?

- look out for logic that may wrap, widen your type

- test the domain – sample everywhere, corners, combinations

What about mission critical code, math & physics research, money?

betting the farm? use SaferIntegers.



SaferIntegers

```
> using SaferIntegers
```

```
> zero = SafeInt16(0)
```

```
> a = 32_000 + zero;
```

```
> a + 999
```

```
ERROR: OverflowError: 32000 + 999 overflowed for type Int16
```

```
> typemin(SafeInt16) - 1 # underflow is an OverflowError
```

```
ERROR: OverflowError: -32768 - 1 overflowed for type Int16
```



using floats

Please do. Just do not take the trailing digits of your results too seriously.

However, if you want reliable trailing digits .. there are helpful packaged types.

- Quadmath.jl exports Float128
(calculate using Float128, convert the result to Float64)
- DecFP.jl exports Dec128, Dec64
(calculate using Dec128, convert the result to Float64)
- DoubleFloats.jl exports Double64
(calculate using Double64, that is all you need to do)



comparing floats

never compare floats for equality

- almost never (testing derived values exactly match rounded constants)
 - and then use `===` so others will know what you intend
- use `isapprox` (`≈` for `isapprox` with defaults) rather than `==`
 - `atol` sets the absolute difference required to match
 - `rtol` sets the proportional difference (# of sigbits) required to match
 - it is ok to use both, with `atol` set for values near 0.0



isapprox

```
tolerance(T::Type, proportion=0.618034) =          # books use = 1/2  
    map(T, tolerance(relbits(T, proportion)))
```

```
tolerance(nbits; abstol_power = 2.125) =  
    ( rtol = 2.0^(-nbits), atol = 2.0^(-nbits * abstol_power) )
```

```
relbits(T::Type, proportion) =  
    floor{Int, proportion * Base.significand_bits(T)}
```

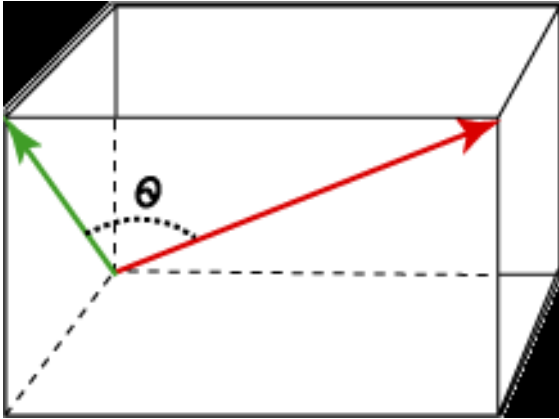
```
const RTOL = ldexp(2.0, -33)      # 2.328e-10 ~ 2.0^(-33)
```

```
const ATOL = ldexp(2.0, -70)      # 3.388e-21 ~ 2.0^(-70)
```

```
≈(x, y) = isapprox(x, y; rtol=RTOL, atol=ATOL)      # if a ≈ b ..
```



Angle Between Vectors



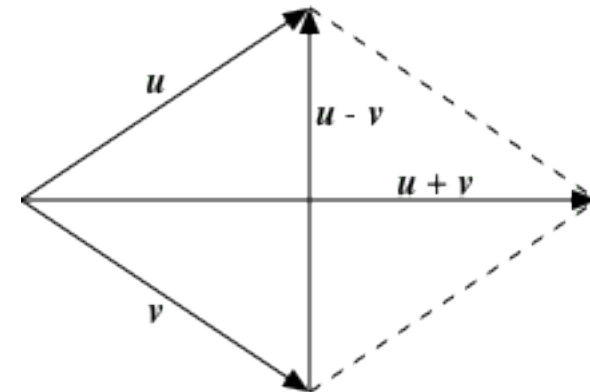
Start with Math. Finish with Numerics.

The dot product of two normalized vectors equals the cosine of their separating angle.

unstable and inaccurate at very small angles

Start with Math. Finish with Numerics.

The sum and difference of two equilength vectors are orthogonal. We use this to find the angle in a stable and robust manner.



AngleBetweenVectors

```
using AngleBetweenVectors    # with Tuples, NamedTuples, Vectors
```

```
smaller_angle = angle(point1, point2)
```

```
                                # to add a new point representation
struct Point2D{T}              # provide a point constructor
    x::T
    y::T
end                             # define a Tuple(::point) method
```

```
Base.Tuple(p::Point2D{T}) where T = (p.x, p.y)
```



AngleBetweenVectors

```
@inline norm2(p::P) where {P<:NTuple{N,T}} where {N,T} =  
    sqrt(foldl(+, abs2.(p)))
```

```
@inline normalize(p ::P) where {P<:NTuple{N,T}} where {N,T} =  
    p ./ norm2(p)
```

```
# works with any point type that has Tuple(point) defined  
Base.angle(pt1::T, pt2::T) where T =  
    angle(Tuple(pt1), Tuple(pt2))
```



AngleBetweenVectors

```
function Base.angle(pt1::P, pt2::P) where {N,T, P<:NTuple{N,T}}
    unitpt1 = normalize(pt1)           # map pts to unit length
    unitpt2 = normalize(pt2)
    y = norm2(unitpt1 .- unitpt2)      # sin of halfangle 0..1
    x = norm2(unitpt1 .+ unitpt2)      # cos of halfangle 1..0
    a = 2 * atan(y, x)                 # a is the answer
    isallgood(a) ? a : clip(a)         # 0 <= a <= T(pi) < pi
                                       # a < 0 ? zero(T) : T(pi)
end
```



abstractions, concrete unions

```
> x = [59, "two"]; typeof(x) == Vector{Any}
> c = concrete(x); typeof(x) == Vector{Union{Int64, String}}
```

```
function concrete(x::AbstractArray)
    ConcreteTypes = Union{typeof.(x)...}
    length(Base.uniontypes(ConcreteTypes)) > 3 && return x

    BaseType = eval(typeof(x).name.name)
    ndim = length(size(x))
    BaseType{ConcreteTypes, ndim}(x)
end
```



abstractions, concrete unions

```
> x = [59, "two"]; typeof(x) == Vector{Any}  
> c = concrete(x); typeof(x) == Vector{Union{Int64, String}}
```

```
function concrete(x::AbstractArray)  
    ConcreteTypes = Union{typeof.(x)...}  
    length(Base.uniontypes(ConcreteTypes)) > 3 && return x  
  
    BaseType = eval(typeof(x).name.name)  
    ndim = length(size(x))  
    BaseType{ConcreteTypes, ndim}(x)  
end
```



Abstract Types

```
abstract type SpaceTime end                # conceptual whole

abstract type AbstractSpace  <: SpaceTime  end # enfolded constituent
abstract type AbstractTime   <: SpaceTime  end # enfolded constituent
abstract type ReferenceFrame <: SpaceTime  end # specialization

abstract type Clock{Frame<:ReferenceFrame} end # constraint
```



Abstract Types

```
abstract type SpaceTime end                # conceptual whole

abstract type AbstractSpace  <: SpaceTime end # enfolded constituent
abstract type AbstractTime   <: SpaceTime end # enfolded constituent
abstract type ReferenceFrame <: SpaceTime end # specialization

abstract type Clock{Frame<:ReferenceFrame} end # constraint

# singleton types

struct FrameIsUTC    <: ReferenceFrame end    # ako GMT, Mean Time
struct FrameIsLocal  <: ReferenceFrame end    # ako wallclock time
```



Abstract Types

```
abstract type Period <: AbstractTime end
```

```
struct Hour    <: Period value::Int64 end
```

```
struct Minute  <: Period value::Int64 end
```



Abstract Types

```
abstract type Period <: AbstractTime end
```

```
struct Hour <: Period value::Int64 end
```

```
struct Minute <: Period value::Int64 end
```

a good way to use `eval`

```
for T in (:Year, :Month, :Day, :Hour, :Minute, :Second)
```

```
  @eval begin
```

```
    struct $T <: Period
```

```
      value::Int64
```

```
    end
```

```
  end
```

```
end
```



Abstract Types

```
struct HourMin{Frame} <: Clock{Frame}  
  hour::Hour  
  minute::Minute  
end
```

```
HourMin(frame::ReferenceFrame, hr::Hour, mn::Minute) =  
  HourMin{frame}(hour, min, sec)
```

```
HourMin(frame::ReferenceFrame, hr::T, mn::T) where {T<:Integer} =  
  HourMin(frame, Hour(hr), Minute(mn))
```

```
HourMin(hr::Hour, mi::Minute) = HourMin{FrameIsUTC}(hr, mi)
```

```
HourMin(hr::T, mn::T) where {T<:Integer} = HourMin(Hour(hr), Minute(mn))
```



Parametrics

```
struct AnyValue{T}  
    value::T  
end
```

```
intval = AnyValue(8)  
strval = AnyValue("abc")
```

```
# AnyValue{Int64}(8)  
# AnyValue{String}("abc")
```

`doubleneg(x::FP{+1,T}) where {T} = x`



Parametrics

```
struct AnyValue{T}
    value::T
end
```

```
intval = AnyValue(8)
strval = AnyValue("abc")
```

```
# AnyValue{Int64}(8)
# AnyValue{String}("abc")
```

```
struct AnyNumber{T<:Number}
    value::T
end
```

```
intval = AnyNumber(8.0)
AnyNumber("abc")
```

```
# AnyNumber{Float64}(8.0)
# MethodError
```

`doubling(x::FP{+1,T}) where {T} = x`



Parametrics

```
struct FP{SGN,T}  
  value::T  
end
```

```
FP(x::T) where T = x<0 ? FP{-1,T}(x) : FP{1,T}(x)
```

```
doubleneg(x::FP{+1,T}) where {T} = x  
doubleneg(x::FP{-1,T}) where {T} = FP{-1,T}(2 * x.val)  
  
two = FP(+2.0); negthree = FP(-3.0); negsix = FP(-6.0)  
doubleneg(two) == two && doubleneg(negthree) == negsix  
  
using Test  
@inferred doubleneg(negthree) == negsix
```



Parametrics

```
struct FP{SGN,T}  
  value::T  
end
```

$\text{FP}(x::T)$ where $T = x < 0 \ ? \ \text{FP}\{-1,T\}(x) : \text{FP}\{1,T\}(x)$

$\text{doubleneg}(x::\text{FP}\{+1,T\})$ where $\{T\} = x$

$\text{doubleneg}(x::\text{FP}\{-1,T\})$ where $\{T\} = \text{FP}\{-1,T\}(2 * x.\text{val})$

$\text{two} = \text{FP}(+2.0)$; $\text{negthree} = \text{FP}(-3.0)$; $\text{negsix} = \text{FP}(-6.0)$

$\text{doubleneg}(\text{two}) == \text{two}$ && $\text{doubleneg}(\text{negthree}) == \text{negsix}$



Parametrics

```
struct FP{SGN,T}  
  value::T  
end
```

```
FP(x::T) where T = x<0 ? FP{-1,T}(x) : FP{1,T}(x)
```

```
doubleneg(x::FP{+1,T}) where {T} = x
```

```
doubleneg(x::FP{-1,T}) where {T} = FP{-1,T}(2 * x.val)
```

```
two = FP(+2.0); negthree = FP(-3.0); negsix = FP(-6.0)
```

```
doubleneg(two) == two && doubleneg(negthree) == negsix
```

```
using Test: @inferred
```

```
@inferred doubleneg(negthree) == negsix
```



Some Packages

- Tables, TableOperations, DataFrames[Meta], TimeSeries
- Statistics, StatsFuns, Distributions, Random
- Interpolations, Dierckx, LsqFit, BlackBoxOptim, Optimization
- SpecialFunctions, Quadmath, DecFP, [Generic]LinearAlgebra
- JuMP, SciML, Symbolics, ModelingToolkit, DrWatson
- Lazy, Chain, TOML, JSON3, JSONTables, CSV
- MLStyle, IterTools, FastBroadcast, InlineStrings, TupleTools

<https://julialang.org/community/organizations/>

<https://juliahub.com/ui/Search>



Tooling

- GitHub or GitLab with GitHub desktop [free on all platforms] or **GitKraken**
 - VSCode with Julia extension [free on all platforms]
 - Documenter, Revise, TestEnv, Infiltrator
 - BenchmarkTools, PkgBenchmarks, PkgTemplates
 - @edit, @which, methods
-
- Branches – try out an approach without committing to it
 - Labels – easily locate the last coherent revision
 - Commit messages – really annoying, occasionally worth the arrgh (squash)



What is and is not “type piracy”

your type, your rules

- major version convention

built-in types and other developers' exported types are theirs

- do not redefine methods (exported or not)
- use the type and its methods, do not alter or amend their working
 - do you see an omission, an improvement? post an issue or a PR.
 - there should be a length method, we have the count of elements

your own multimethods are not piracy because they are not theirs

- just use names that are **not** in Base and **try hard not to clash** with imports



sketch what you feel

```
src/runningsum.jl
```

```
"""
```

```
    runningsum(source, winsize)
```

Provides the windowed running sum over source.

- result has $\text{length}(\text{source}) - \text{winsize} + 1$ elements

```
"""
```

```
function runningsum(source::AbstractVector{T}, winsize) where {T}
end
```

```
test/runningsum.jl
```

```
@test runningsum([1,2,3,4,5,6], 3) == [6,9,12,15]
```



design concept

```
struct Window{V,I,F}  
  source::V  
  span::I  
  apply::F  
end
```

```
# allow many different functions  
# data to run window over  
# width of the window (nelements)  
# function to apply over window
```

```
mutable struct Running{V,F,T}  
  window::Window{V,F}  
  firstidx::Int  
  finalidx::Int  
  lastvalue::T  
end
```

```
# support running over windows  
# generalized window specifier  
# where current window starts  
# where current window ends  
# prior (or first) summary value
```



design refinement

```
struct Window{V,I,F}  
  source::V  
  span::I  
  apply::F  
end
```

```
mutable struct Running{V,F,T}  
  window::Window{V,F}  
  firstidx::Int  
  finalidx::Int  
  lastvalue::T  
end
```

```
struct Window{V}      struct Runner{F1,F2}  
  source::V           setup::F1  
  span::Int           update::F2  
end                   end
```

```
struct Running{V,F1,F2}  
  runner::Runner{F1,F2} # applicative  
  window::Window{V}     # data surface  
  current_start::Int     # running start  
end
```

```
present(idx, value) = (; idx, value)
```


coding

```
function runningsum(source::AbstractArray{T,N}, winsize) where {T,N}
    # provides a view given a concrete Array
    runningsum(view(source,:), winsize)
end

const ArrayView = SubArray{T,N,P,I,L} where {T,N,P,I,L}

function runningsum(source::ArrayView, winsize)
    # works with a view of the source, not the source directly
end
```



coding

```
function runningsum(source::ArrayView, winsize)
    n = length(source) - winsize + 1           # how many results
    result = Vector{T}(undef, n)               # fast allocation
    current = sum(view(source, 1:winsize))      # initialize
    result[1] = current                         # set up result

    @inbounds for idx in 1:n-1                 # proper for algorithm
        current += source[winsize+idx] - source[idx] # update
        result[idx+1] = current                 # remember
    end
    isallgood(result) ? result : clip(result)   # ↩ allisgood ↪
end
```

I recommend working for this person

"Your work always delivers \$. It is a tomorrow key for me."
"As a professional , you are ready to .. I'll see to that."



shuffle up

```
# how a data processing center overcharged my client millions
```

```
oldsystem = (; flops = 4)
```

```
newsystem = (; flops = 16)
```

```
performance_change = newsystem.flops - oldsystem.flops      # 12
```

```
performance_multiplier = performance_change / oldsystem.flops # 3.00
```

```
performance_adjusted_unit_cost = 1 + performance_multiplier  # 4.00
```

shuffle up and deal

how the overcharges happened

```
oldsystem = (; flops = 4)
```

```
newsystem = (; flops = 16)
```

```
performance_change = newsystem.flops - oldsystem.flops      # 12
```

```
performance_multiplier = performance_change / oldsystem.flops # 3.00
```

```
performance_adjusted_unit_cost = 1 + performance_multiplier # 4.00
```

```
comparative_advantage = performance_change / newsystem.flops # 0.75
```

```
performance_adjusted_unit_cost = 1 + comparative_advantage  # 1.75
```

Big Picture

- Julia takes some familiarity
 - mostly time to unlearn approaches unhelpful with Julia
 - some time (practice time) to gain ease with the helpful ones
- truly provides community help
 - *no more tears* -- just ask, we are inclined to answer
- speeds good work, encourages cooperation
- less tension, much less self-recrimination

lather, rinse, repeat

Portmanteaus are the paradigmatics of programming.

This is our profession – we make what is not become that which is.

-- we focus, ennoble, and provide outcomes to work.

Consider, Shape, Describe, Design, Reframe, Reconsider, Design, Sleep

Sketch, Scramble, Sculpt,

WalkTomorrowsWalk before you TalkTodayTalk

Scramble and be steady with your footing Sculpt with your own flavorful intent

The best of Julia is what you do with Julia

The best of Julia is what you do with Julia

these are the slides

github.com/JeffreySarnoff/JuliaCon2022meetup