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, how, what *to ask*?

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.

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. Persuade with code. Convince with tests.

NamedTuples

- All the fun of Tuples `(1, 2, 3)`, `(1, 2.0, "three")`, now with names enfolded.
- More trustworthy: `routing[2]` vs `routing.destination`
- More easily shared, maintained, **explained**

```
> emily = (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

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

> employee = newhire("Emily", "Rey", 12)
(firstname = "Emily", lastname = "Rey", badge = 12)

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

> keys(employee)
(:firstname, :lastname, :badge)

> values(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")
```

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
```

isapprox

never compare floats for equality

- almost never (are derived values exactly these known 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` for values near 0.0

what to do when last digit[s] accuracy matters?

isapprox

```
tolerance(T::Type, proportion=0.618034) =
  map(T, tolerance(relbits(T, proportion)))

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

const AbsTolScale = 2.618034

tolerance(nbits; atolscale=AbsTolScale) =
  (atol = 2.0^(-nbits*atolscale), rtol = 2.0^(-nbits))

const ATOL = tolerance(Float64, 5/8).atol
const RTOL = tolerance(Float64, 5/8).rtol

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

write clean code (not there yet, rewrite it)

iteration

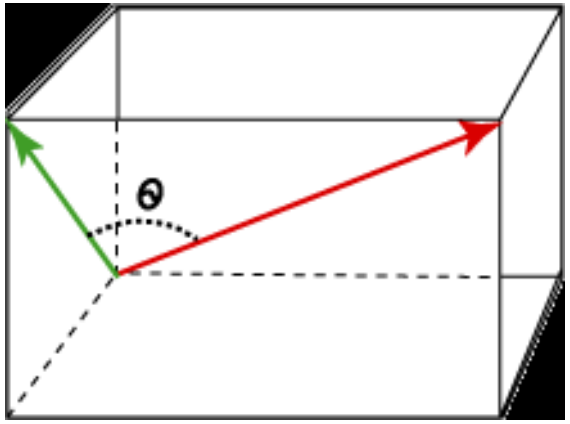
```
for current_value  in           xs  .. end    # avoid index nums
for current_index  in eachindex(xs) .. end    # these are fast
for (index, value) in enumerate(xs) .. end    # and future proof
for current_column in eachcol(amat) .. end    # prefer bycol 2x+
```

lazy comprehension

```
xs = [3, 5, 7]          # keep fn.(xs) lazy as long as possible
ys = (x^2 for x in xs) # Base.Generator{Vector{Int64}, ..

zs = zip(xs, ys)        # zip is lazy and surprisingly fast
```

AngleBetweenVectors



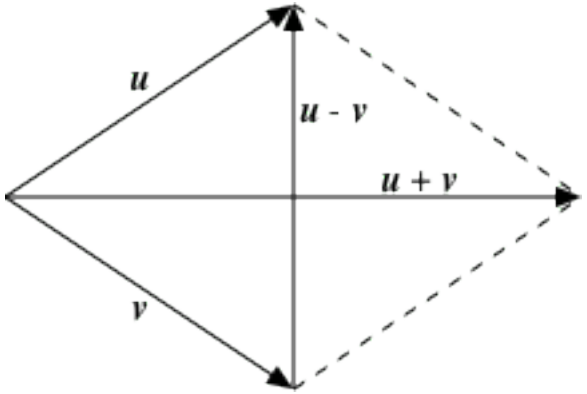
Start with Math. Finish with Numerics.

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. vectorstwo argument atan of the difference and the sum of two normalized vectors approximates half their separating angle in a stable and robust manner.



AngleBetweenVectors

```
using AngleBetweenVectors
                              # point as Tuple, NamedTuple, Vector
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)                    # 2 atan(tan(halfangle))
  zero(T) <= a <= T(pi) && return a     # protect the expected
  a < 0 ? zero(T) : T(pi)               # correct the can't happen
end
```

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
```

abstract types and concrete unions

```
> x = [59, "two"]; typeof(x) == Vector{Any}
> x = 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 and concrete unions

```
> x = [59, "two"]; typeof(x) == Vector{Any}
> x = 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

struct FrameIsUTC   <: ReferenceFrame end      # was GMT
struct FrameIsLocal <: ReferenceFrame end      # local or wallclock time

const TimeIsUTC   = FrameIsUTC()               # consts are in a module
const TimeIsLocal = FrameIsLocal()             # these are singletons
```

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

```
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

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

HourMin(frame::ReferenceFrame, hr::T, mn::T) where {T<:Int64}  =
  HourMin(frame, Hour(hr), Minute(mn))

HourMin(hr::Hour, mi::Minute) where T =
  HourMin(TimeIsUTC, hr, mi)
```

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

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
```

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
```

function signatures

```
rhyme(word) # rhyme("rhyme") == ("chime", "prime", "time")
rhyme(word, matches=1) # ("chime",) # trailing args may default

rhyme(word::String, matches::Int)
rhyme(word::String, matches::Integer=1)
rhyme(word::AbstractString, matches::Signed=1)
rhyme(word::T, matches::Integer=4) where {T<:AbstractString}

methods(rhyme), methodswith((String, Int))
MethodAnalysis.methodinstances(rhyme)
```

Some Packages

- Tables, TableOperations, DataFrames[Meta], TimeSeries
- Statistics, StatsFuns, Distributions, Random
- Interpolations, Dierckx, LsqFit, Optim, BlackBoxOptim
- 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]
- BenchmarkTools, Revise, Infiltrator, TestEnv
- 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 and other developers' exported types are not yours
  - 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

Performance Tips

- keep your focus on clarity, transferability, obviousness

- small functions are more pleasant, more performant
- use `@code_warntype` to find type instability (fix what is fixable)
- watch your nested loops
  - column iteration inside row iteration (and so on)

- maintain your own notes, your own snippets and helpful reminders
- keep organized bookmarks to references and examples and answers you like

development

src/runningsum.jl

```
"""
    runningsum(source, windowsize)

Provides the windowed running sum over source.
- result has length(source) - windowsize + 1 elements
"""
function runningsum(source::Vector{T}, windowsize) where {T}
end
```

test/runningsum.jl

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

coding

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

  for idx in 1:n-1                                      # firstindex
    current += source[windowsize+idx] - source[idx]     # update
    result[idx+1] = current                             # remember
  end
  result
end
```

design concept

```
struct Window{V,I,F}                    # allow many different
  source::V                             # windowed functions
  span::I
  apply::F
end

mutable struct Running{V,F,T}           # support running over windows
  window::Window{V,F}                   # with a generalized approach
  firstidx::Int
  finalidx::Int
  lastvalue::T
end
```

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}
  source::V
  span::Int
end

struct Runner{F1,F2}
    setup::F1
    update::F2
end

struct Running{V,F1,F2}
  runner::Runner{F1,F2}
  window::Window{V}
end

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

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

shuffle up and deal

```
# 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 # 3.0
performance_adjusted_unit_cost = 1 + performance_multiplier   # 4.0
```

shuffle up and deal

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

oldsystem = (; flops =  4)
newsystem = (; flops = 16)
performance_change = newsystem.flops - oldsystem.flops        # 12

performance_multiplier = performance_change / oldsystem.flops # 3.0
performance_adjusted_unit_cost = 1 + performance_multiplier   # 4.0

comparative_advantage = performance_change / newsystem.flops  # 0.75
performance_adjusted_unit_cost = 1 + comparative_advantage    # 1.75
```

the best of Julia is what you do with Julia

Thank you for being a part of this.

the slides are available at
github.com/JeffreySarnoff/JuliaCon2022meetup