*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)

*Named Tuples*

* All the fun of Tuples, now with names enfolded.
* More trustworthy**:** routing.destination is
* More easily shared, maintained, **explained**

more helpful than routing[2]

*Named Tuples*

* All the fun of Tuples, now with names enfolded.
* More trustworthy**:** routing.destination is
* More easily shared, maintained, **explained**

more helpful than routing[2]

> 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?*

*Named Tuples*

*at a party with the cool kids*

newhire( firstname,

lastname,

lastname,

badge )

badge )

=

( ;

firstname,

emily\_rey

=

newhire( "Emily", "Rey", 12 )

(firstname =

"Emily",

lastname = "Rey",

badge =

12 )

*Named Tuples*

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

*Named Tuples*

>

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 )

*Named TupleTools*

> using NamedTupleTools

> select(employee, (:firstname, :lastname))

(firstname = "Emily", lastname = "Rey")

> delete(employee, :badge)

(firstname = "Emily", lastname = "Rey")

> id, name = split(employee,

((badge = 12,), (firstname =

:badge)

"Emily", lastname

= "Rey"))

> merge(id, name)

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

# create new

*write clean code*

not there yet, rewrite it (developing good habits) not sure what to do, look at other solutions or ask

iteration

for

for for

for

current\_value

current\_index (index, value)

current\_column

in

in in

in

xs

eachindex(xs) enumerate(xs)

eachcol(amat)

..

..

..

..

end

end end

end

#

# #

#

avoid index nums

these are fast and future proof

prefer bycol 2x+

lazy comprehension

ys =

zs =

(x^2 for x

zip(xs, ys)

in xs)

# Base.Generator{Vector{Int64}, ..

#

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 Safer ntegers.

*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:

*Overflow*Error:

-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)

* DoubleFloatsjl 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

const

RTOL

ATOL

=

=

ldexp(2.0,

ldexp(2.0,

-33)

-70)

#

#

2.328e-10

3.388e-21

~

~

2.0^(-33)

2.0^(-70)

≃(x,

y)

=

isapprox(x,

y;

rtol=RTOL,

atol=ATOL)

#

if

a

≃

b

..

*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. 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

provide a

new

point representation

struct Point2D{T}

x::T

y::T

end

point constructor

#

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}}

sqrt(foldl(+, abs2.(p)))

where {N,T} =

@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) unitpt1 = normalize(pt1)

unitpt2 = normalize(pt2)

where {N,T, P<:NTuple{N,T}}

# map pts to unit length

y x

2

end

=

=

\*

norm2(unitpt1 norm2(unitpt1

atan(y, x)

.-

*.+*

unitpt2) unitpt2)

#

#

if

lsb[s] are off for the precision

given

stable

result remains robustly consistent +

function

# …

Base.angle(pt1::P,

pt2::P)..

# # # #

#

protecting against the is the math expressing

be runtime savvy to isallgood(a) = 0 <=

almost certainly never impeccable

do this, and do a <= T(pi)

never (no)

this

a = 2

\* atan(y, x)

isallgood(a) ? a : clip(a)

clip(a) = a < 0 ? zero(T) : T(pi)

end

*abstractions, concrete unions*

>

>

x

c

=

=

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

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

c

=

=

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

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

abstract

abstract

type

type

type

AbstractSpace

AbstractTime

ReferenceFrame

<:

<:

<:

SpaceTime

SpaceTime

SpaceTime

end

end

end

#

#

#

enfolded constituent

enfolded constituent

specialization

abstract

type

Clock{Frame<:ReferenceFrame}

end

#

constraint

*Abstract Types*

*abstract*

*type*

SpaceTime

*end*

#

conceptual whole

abstract

abstract

abstract

type

type

type

AbstractSpace

AbstractTime

ReferenceFrame

<:

<:

<:

SpaceTime

SpaceTime

SpaceTime

end

end

end

#

#

#

enfolded constituent

enfolded constituent

specialization

abstract

type

Clock{Frame<:ReferenceFrame}

end

#

constraint

#

singleton *types*

struct FrameIsUTC

<: ReferenceFrame end

#

#

ako

ako

GMT, Mean Time

wallclock time

struct FrameIsLocal <: ReferenceFrame end

*Abstract Types*

abstract type

Period <:

AbstractTime

end

struct Hour

struct Minute

<: Period

<: Period

value::Int64

value::Int64

end

end

*Abstract Types*

abstract type

Period <:

AbstractTime

end

struct Hour

struct Minute

<:

<:

Period

Period

value::Int64

value::Int64

end

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,

HourMin{frame}(hour, min, sec)

mn::Minute) =

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

strval

=

=

AnyValue(8)

AnyValue("abc")

#

#

AnyValue{Int64}(8)

AnyValue{String}("abc")

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

{T}

=

x

*Parametrics*

struct AnyValue{T} value::T

end

intval

strval

= AnyValue(8)

#

#

AnyValue{Int64}(8)

AnyValue{String}("abc")

=

AnyValue("abc")

struct

AnyNumber{T<:Number}

value::T

end

intval =

AnyNumber(8.0)

#

#

AnyNumber{Float64}(8.0)

MethodError

AnyNumber("abc")

doubleneg(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})

doubleneg(x::FP{-1,T})

where {T}

where {T}

=

=

x

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})

doubleneg(x::FP{-1,T})

where {T}

where {T}

=

=

x

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})

doubleneg(x::FP{-1,T})

where {T}

where {T}

=

=

x

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
* Labels
* Commit messages
* try out an approach without committing to it
* easily locate the last coherent revision
* 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 length(source) –

winsize +

1

elements

"""

function

end

runningsum(source::AbstractVector{T}, winsize)

where {T}

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}

# # # #

#

support running over windows

generalized window specifier

window::Window{V,F} firstidx::Int finalidx::Int lastvalue::T

end

where where

prior

current window starts current window ends

(or first) summary value

*design refinement*

struct Window{V,I,F} source::V

span::I apply::F

end

struct Window{V} source::V span::Int

end

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

end

mutable struct

Running{V,F,T}

struct Running{V,F1,F2} runner::Runner{F1,F2} window::Window{V} current\_start::Int

end

window::Window{V,F} firstidx::Int finalidx::Int lastvalue::T

end

# #

#

applicative data *surface*

running start

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)

#

end

works with a

view of the source, not the source directly

*coding*

function runningsum(source::ArrayView, n = length(source) - winsize + 1

winsize)

#

# # #

#

how many results

fast allocation initialize

set up result

proper for algorithm # update

# remember

result =

current = result[1]

@inbounds

current

Vector{T}(undef,

sum(view(source,

= current

for idx in 1:n-1

n)

1:winsize))

+= source[winsize+idx]

- source[idx]

result[idx+1] = current

end

isallgood(result) ?

end

# ↜ allisgood ↝

result :

clip(result)

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 =

newsystem =

(; flops

(; flops

=

=

4)

16)

performance\_change =

newsystem.flops -

oldsystem.flops

#

12

performance\_multiplier =

performance\_change /

oldsystem.flops

#

#

3.00

4.00

performance\_adjusted\_unit\_cost =

1

+

performance\_multiplier

*shuffle up and deal*

#

how the

overcharges happened

oldsystem

newsystem

=

=

(; flops

(; flops

=

=

4)

16)

performance\_change =

newsystem.flops

-

oldsystem.flops

#

12

performance\_multiplier =

performance\_change /

oldsystem.flops

#

#

3.00

4.00

performance\_adjusted\_unit\_cost =

1

+

performance\_multiplier

comparative\_advantage =

performance\_change /

newsystem.flops

#

#

0.75

1.75

performance\_adjusted\_unit\_cost

=

1

+

comparative\_advantage

*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

*The best of Julia is what you do with Julia*