Scientific Programming in Julia

Package Management, Advanced Concepts, and Worflows

Nicholas Gale & Stephen Eglen

Types and Hierarchy

- Julia has a hierarchy of types e.g. Int <: Real <: Number.
- Function definitions can apply to all subtypes.
- Sub/super type can be checked with `<:`.

```julia
println(Int64 <: Real)
println(Float64 <: Real)
println(Int64 <: Float64)
println(subtypes(Number))
```

```
true
true
false
Any[Complex, Real]
```

Custom Types

- Julia supports custom types which can be inserted into a hierarchy.
- Custom types are defined with the `struct` keyword and have named fields.
- An object is called using the name of the `struct`

```julia
struct Stats
    xbar::Real
    xsig::Real
    xkur::Real
end
Stats(4,0.4,0.1)
```

```
Stats(4, 0.4, 0.1)
```

Type Constructors

- An object is created by calling the struct name with fields.
- A type constructor may also be defined in the `struct` definition and the `new` method.
- Objects can now be created with a function constructor.

```
using StatsBase, Random
struct StatsGenerator
    xmean::Real
    xvar::Real
    xkur::Real
    function StatsGenerator(n::Int
        # take a sample of n draws
        sample = randexp(n)
        av = round(mean(sample), d
        v = round(var(sample), dig
        k = round(kurtosis(sample)
        new(av, v, k)
    end
end

println([StatsGenerator(100), Stat
```

```
StatsGenerator[StatsGenerator(1.249,
1.213, 1.652), StatsGenerator(1.042,
1.102, 4.03), StatsGenerator(1.073,
1.18, 3.531)]
```

Changing Objects

- Julia passes objects by reference - be careful with copying mutable types.

```
a = Any[Any[1,0], Any[2,0], Any[3,0], Any[4,0]]
## reference
b1 = a
b1[1] = ["*",2]
println(a)
```

```
Any[Any["*", 2], Any[2, 0], Any[3, 0], Any[4, 0]]
```

- `copy` deferences the first layer.

```julia
## shallow copy (first layer mutations don't change original object)
b2 = copy(a)
println(b2 === a)
b2[1] = ["A",2]
b2[2][1] = "b"
println(a)
```

```
false
Any[Any["*", 2], Any["b", 0], Any[3, 0], Any[4, 0]]
```

- `deepcopy` recursively copies.

```julia
## deep copy (object is completely indistinguishable)
b3 = deepcopy(a)
println(b3 === a)
b3[3][1] = "c"
println(a)
```

```
false
Any[Any["*", 2], Any["b", 0], Any[3, 0], Any[4, 0]]
```

Pass by reference

- This applies to objects passed to functions whereas in R they are passed by *value*.
- In R a copy of an object's data is used by the functions local environment, in Julia it is the object itself.

- R Code

```r
#| eval: false
x <- c(1,2,3,4)
f <- function(x){x[1] = 10}
f(x)
# x is unchanged.
```

- Julia Code

```julia
x = [1,2,3,4]
function f(x)
    x[1] = 10
end
f(x)
println(x)
```

```
[10, 2, 3, 4]
```

- Functions that mutate their objects are usually denoted by a bang operator.

```julia
x = [5.0, 1.8, 2.1, 3.7, 4.1]
sort!(x)
println(x)
```

```
[1.8, 2.1, 3.7, 4.1, 5.0]
```

Multiple Dispatch

- Functions in Julia may be overloaded with multiple definitions; Packages often do this.
- The compiler decides which definition to use based on type inference.

```julia
function custom_modulus(x::Real)
    return abs(x)
end

function custom_modulus(x::Complex)
    res::Complex = sqrt(x.re^2 + x.im^2)
    return res
end

a = custom_modulus(-2)
b = custom_modulus(-2+1im)
```

```julia
println([a, typeof(a)])
println([b, typeof(b)])
```

```
Any[2, Int64]
Any[2.23606797749979 + 0.0im, ComplexF64]
```

Functions: Broadcasting

- Broadcasting is done with the `@broadcast` macro, or the `.` notation.
- Using the `.` notation any function can be broadcast onto an array.
- This is native vectorisation as you might expect in R.

```
using StatsBase
samples = [rand(100)*i for i in 1:5]
variances = var.(samples)
println(variances)
```

```
[0.0849030739394276, 0.3376312623058097, 0.7356963876919044,
1.344597763777424, 2.012684813134627 3]
```

Functions: Vectorisation

- There is no performance hit to *not* vectorising unlike R which *must* be vectorised.
- Choose the format that comes naturally when writing code.
- *Don't* spend time posing the problem in linear algebra format unless it make sense.

Introduction to Packages

- Packages are a collection of function and type defintions.
- About 7,400 packages registered (October 2022). https://julialang.org/packages/ has several methods to navigate the ecosystem.
- Packages in Julia are loaded with the `using` keyword. Some are included in the base installation.

```julia
using Random
randperm(5)'
```

```
1×5 adjoint(::Vector{Int64}) with eltype Int64:
 5  3  4  1  2
```

- Package functions are imported into the namespace; and can also be accessed using `PkgName.function`.

```
using StatsBase, Random
vec = Random.randperm(100)
stdv100 = std(vec)
```

```
29.011491975882016
```

Package manager

- Package management in Julia is easing using the Pkg package, or the package environment.
- To access package press `]` in the REPL. To exit press `:esc` in the enviroment.
- To add/remove/build a package use the `add`/`remove`/`build` keywords in the package manager followed by the package name.
- Alternatively Pkg.add("PkgName"). Equivalently: `Pkg.remove`, `Pkg.build`.

Interesting packages

Macros

- Macros are useful shorthands for blocks of code; often packages export their own macros
- Macros are used with the `@macro` syntax placed before a code block.
- Useful benchmarking macros are: @time (Base), @btime & @benchmark (Benchmarking Tools), @profile (Profile).
- Often used to do magic, e.g. @fastmath @inbounds

```
x = 0
@time for i = 1:100000
    x *= x^i
end
```

```
0.011881 seconds (99.49 k allocations: 1.518 MiB)
```

Data I/O

- Julia supports low level IO through `read` and `write` functions.
- Julia objects can be saved through the `FileIO` package. `save("path", object, "save_name")`, and `load("path", "save_name")`.
- CSV is a package for CSV with readcsv with `CSV.read`, and `CVS.write` analogous to `read_csv` and `write_csv` in R. `CSV.File("path")` creates an object useful for piping.
- DelimitedFiles is a package for generic delimiters with `writedlm("dir", obj, delim)` and `readdlm("path", delim)`.

Workflow: Module

- Having a large script with many static function definitions is unwieldy.
- A module can be used to abstract many functions and types away and export only necessary functions.
- It acts like a local package.

Creating a Module

- A module enviroment is created using the syntax: `module PackageName`.
- Definitions are exported using the `export` keyword.
- A module is included in a script with `include("path/to/module")` and `using .PackagName` or `using Main.PackageName`.

```
module BasicStats
export std
mean(x) = sum(x)/length(x)
std(x) = sqrt(sum((x .- mean(x)).^2)/length(x))
end
```

```
using .BasicStats
std([1.0, 2.0, 3.0])
```

```
1.0
```

Workflow: Revise

- Including a module imports new functions to the namespace which results in conflicts.
- Starting a new session resolves this, but incurs start-up time penalty.
- The `Revise` package tracks changes in files included with `includet("path/to/file")`.
- This allows for developing modules.

Workflow: Environments

- An environment is a version-controlled local version of Julia.
- It includes a Project.toml, and Manifest.toml which list package dependencies.
- They are good for reproducible code.
- They can be upgraded into a package easily.

Creating Environments

- Start a Julia session in the parent directory of an environment.
- Generate the enviroment with the package manager `]generate PackageName`
- This creates a directory called PackageName with a Manifest, Project, and src files.

Activating Enviroments

- Navigate to the enviroment directory and start a Julia session with `julia --project=.`
- Alternatively, start julia in the directory and use the package manager: `]activate .`
- Once the environment is activated the package may be included with `using PackageName`
- There is *no* dot.

Environements and Revise

- In the enviroment `using Revise` will track all changes.
- This allows splitting a module into several files.
- This can be useful for organisation in large projects.