Ch5_Notes_Advanced_topics_on_collections

May 20, 2024

1 Chapter 5 - Advanced topics on handling collections

1.1 This chapter covers

- Vectorizing your code, aka broadcasting
- Understanding subtyping rules for parametric types
- Integrating Julia with Python
- Performing t-SNE dimensionality reduction

1.2 Broadcasting!

Broadcasting is another way to 'broadcast' a function to a set of elements, to iterate over a collection, the way a map() function, or comprehension loop would.

A very basic example - multiple every element of x with every element of y, index wise, meaning element of index 1 in x is multiplied with element of index 1 in y.

[1]:
```
x = [2, 3, 4] ; y = [2, 3, 4]
```

[1]:
```
3-element Vector{Int64}:
 2
 3
 4
```

The "." dot operator is the magic symbol of broadcasting

[2]:
```
x .* y
```

[2]:
```
3-element Vector{Int64}:
  4
  9
 16
```

The same thing can be achieved with **map()** and **comprehension**

[3]:
```
map(*, x, y)
```

[3]:
```
3-element Vector{Int64}:
  4
  9
 16
```

Comprehension is much uglier in my opinion

```
[5]: [x[i] * y[i] for i in eachindex(x,y)] # peep the eachindex() function which␣
     ↪listed the index size
```

```
[5]: 3-element Vector{Int64}:
       4
       9
      16
```

If the sizes of a and b do not match, we get an error

1.2.1 Breaking the index equivalence rule whereby both collections must be of equal size to perform broadcasting

There is one exception to the rule that dimensions of all collections taking part in broadcasting must match. This exception states that single-element dimensions get expanded to match the size of the other collection by repeating the value stored in this single element: If the second collection **ONLY** has a single element, than iteration is possible on this.

```
[6]: x .* [2]
```

```
[6]: 3-element Vector{Int64}:
      4
      6
      8
```

1.2.2 Another example using matrices

```
[7]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] .* [1 2 3 4 5 6 7 8 9 10]
```

```
[7]: 10×10 Matrix{Int64}:
       1   2   3   4   5   6   7   8   9   10
       2   4   6   8  10  12  14  16  18   20
       3   6   9  12  15  18  21  24  27   30
       4   8  12  16  20  24  28  32  36   40
       5  10  15  20  25  30  35  40  45   50
       6  12  18  24  30  36  42  48  54   60
       7  14  21  28  35  42  49  56  63   70
       8  16  24  32  40  48  56  64  72   80
       9  18  27  36  45  54  63  72  81   90
      10  20  30  40  50  60  70  80  90  100
```

“This technique is often used in practice to get a Cartesian product of all inputs. For instance, in part 2, you will learn that when you write”x” => sum in DataFrames.jl, you ask the package to apply the sum function to the column x of the data frame. A common scenario is that we want to apply several functions to several columns of a data frame. Using broadcasting, this can be written concisely as follows:”

```
[8]: ["x", "y"] .=> [sum minimum maximum]
```

```
[8]: 2×3 Matrix{Pair{String}}:
      "x"=>sum  "x"=>minimum  "x"=>maximum
      "y"=>sum  "y"=>minimum  "y"=>maximum
```

recall that vectors in Julia are columnar; in this case, the vector has one column and two rows), and the [sum minimum maximum] expression creates a matrix with one row and three columns. More from this blog post https://julialang.org/blog/2013/09/fast-numeric/

"Julia arrays are stored in column-major order, which means that the rows of a column are contiguous, but the columns of a row are generally not. It is therefore generally more efficient to access data column-by-column than row-by-row."

1.2.3 Some nice example of using broadcasting with functions

If we want to prefix 22 values with a certain element, say "chr", we can do

```
[9]: string.("chr", 1:22)
```

```
[9]: 22-element Vector{String}:
      "chr1"
      "chr2"
      "chr3"
      "chr4"
      "chr5"
      "chr6"
      "chr7"
      "chr8"
      "chr9"
      "chr10"
      "chr11"
      "chr12"
      "chr13"
      "chr14"
      "chr15"
      "chr16"
      "chr17"
      "chr18"
      "chr19"
      "chr20"
      "chr21"
      "chr22"
```

```
[10]: f(i::Int) = string("got integer ", i) ; f(s::String) = string("got string ", s)
```

```
[10]: f (generic function with 2 methods)
```

```
[11]: f.([1, "1"])
```

```
[11]: 2-element Vector{String}:
       "got integer 1"
       "got string 1"
```

1.3 Protecting collections from being broadcasted over

Let's say we have a vector of values, any we want to know whether any of the values are in another collection, what ought we do? We could try use the **in()** function?

```
[12]: in([1, 3, 5, 7, 9], [1, 2, 3, 4])
```

```
[12]: false
```

But that doesn't work even though 1 and 3 are matches, so what's the reason? The first array is treated as a whole, and so ALL the elements must be matching between the arrays. What if we wrap the second array in two brackets?

```
[13]: in([1, 3, 5, 7, 9], [[1, 2, 3, 4]])
```

```
[13]: false
```

Nope, still treated as a whole. Let's try using the function where we know all elements are matching. in([1, 3, 5, 7, 9], [1, 2, 3, 4, [1, 3, 5, 7, 9]])

```
[14]: in([1, 3, 5, 7, 9], [1, 2, 3, 4, [1, 3, 5, 7, 9]])
```

```
[14]: true
```

This still didn't solve our original problem, whereby we want to know if and which ANY of the elements in the first set are in the second. What if we use broadcasting?

```
[15]: in.([1, 3, 5, 7, 9], [1, 2, 3, 4])
```

```
DimensionMismatch: arrays could not be broadcast to a common size; got a␣
  ↪dimension with lengths 5 and 4

Stacktrace:
 [1] _bcs1
   @ ./broadcast.jl:555 [inlined]
 [2] _bcs
   @ ./broadcast.jl:549 [inlined]
 [3] broadcast_shape
   @ ./broadcast.jl:543 [inlined]
 [4] combine_axes
   @ ./broadcast.jl:524 [inlined]
 [5] instantiate
   @ ./broadcast.jl:306 [inlined]
 [6] materialize(bc::Base.Broadcast.Broadcasted{Base.Broadcast.
  ↪DefaultArrayStyle{1}, Nothing, typeof(in), Tuple{Vector{Int64},␣
  ↪Vector{Int64}}})
   @ Base.Broadcast ./broadcast.jl:903
 [7] top-level scope
   @ In[15]:1
```

Nope. Hmmmm.

Well we know that using the set operations should work, let's perform an intersection.

[16]:
```
intersect([1, 3, 5, 7, 9], [1, 2, 3, 4])
```

[16]:
```
2-element Vector{Int64}:
 1
 3
```

As expected, but, there is another way to still use the **in()** function.

"How should we resolve this issue? The solution is to wrap the vector that we want to be reused as a whole with Ref. In this way, we will protect this object from being iterated over. Instead, it will be unwrapped from Ref and treated by broadcasting as if it were a scalar, and thus this value will be repeated to match the dimension of the other container:"

[17]:
```
in.([1, 3, 5, 7, 9], Ref([1, 2, 3, 4]))
```

[17]:
```
5-element BitVector:
 1
 1
 0
 0
 0
```

Now it works, I suppose..

1.3.1 wtf is Ref?

Ref can be seen as shorthand for "Reference" - it takes the values of a collection, and creats a single value reference of the value, which it stores in an array, almost like a little iterator of it's own. So when we run broadcasting in combination with **Ref()**, Ref will operator on each element of the array, and thus the size of the first and second arrays will be equal (1 each), allowing for the broadcasting to work as intended…. a bit random but hey this is effective.

1.4 Exercise 5.1

The parse function can be used to convert a string into a number. For instance, if you want to parse a string as an integer, write parse(Int, "10") to get the integer 10. Assume you are given a vector of strings ["1", "2", "3"]. Your task is to create a vector of integers by parsing the strings contained in the given vector.

[23]:
```
string_vec = ["1", "2", "3"]
```

```
[23]: 3-element Vector{String}:
 "1"
 "2"
 "3"
```

```
[24]: int_vec = parse.(Int, string_vec)
```

```
[24]: 3-element Vector{Int64}:
 1
 2
 3
```

1.5 Analyzing Anscombe's quartet data using broadcasting

aq = [10.0 8.04 10.0 9.14 10.0 7.46 8.0 6.58 8.0 6.95 8.0 8.14 8.0 6.77 8.0 5.76 13.0 7.58 13.0 8.74 13.0 12.74 8.0 7.71 9.0 8.81 9.0 8.77 9.0 7.11 8.0 8.84 11.0 8.33 11.0 9.26 11.0 7.81 8.0 8.47 14.0 9.96 14.0 8.1 14.0 8.84 8.0 7.04 6.0 7.24 6.0 6.13 6.0 6.08 8.0 5.25 4.0 4.26 4.0 3.1 4.0 5.39 19.0 12.50 12.0 10.84 12.0 9.13 12.0 8.15 8.0 5.56 7.0 4.82 7.0 7.26 7.0 6.42 8.0 7.91 5.0 5.68 5.0 4.74 5.0 5.73 8.0 6.89]

```
[8]: using Pkg
```

```
[9]: Pkg.add("Statistics") ; using Statistics
```

```
   Resolving package versions…
  No Changes to `~/.julia/environments/v1.10/Project.toml`
  No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

Calculate the mean of each column in the matrix, using broadcasting of the mean function combined with the eachcol() function

```
[10]: mean.(eachcol(aq))
```

```
[10]: 8-element Vector{Float64}:
 9.0
 7.500909090909093
 9.0
 7.500909090909091
 9.0
 7.500000000000001
 9.0
 7.50090909090909
```

If we don't use broadcasting, we get a row based estimate

```
[11]: mean(eachcol(aq))
```

```
[11]: 11-element Vector{Float64}:
  8.6525
  7.4525
```

```
10.47125
 8.56625
 9.35875
10.492500000000001
 6.3375
 7.03125
 9.71
 6.92625
 5.755000000000001
```

We'll just show a brief comparison between the correlation function from chapter 4, and an updated correlation function which no incorporates broadcasting.

```
[12]: function R²(x, y)
         X = [ones(11) x]
         model = X \ y
         prediction = X * model
         error = y - prediction
         SS_res = sum(v -> v ^ 2, error)
         mean_y = mean(y)
         SS_tot = sum(v -> (v - mean_y) ^ 2, y)
         return 1 - SS_res / SS_tot
      end
```

```
[12]: R² (generic function with 1 method)
```

Pay attention to the slight differences and the streamlining it allows

```
[13]: function R²(x, y)
         X = [ones(11) x]
         model = X \ y
         prediction = X * model
         SS_res = sum((y .- prediction) .^ 2)
         SS_tot = sum((y .- mean(y)) .^ 2)
         return 1 - SS_res / SS_tot
      end
```

```
[13]: R² (generic function with 1 method)
```

"By now you know four ways of iteratively applying operations to elements of collections:

- Using for loops
- Using comprehensions
- Using the map function (and other similar higher-order functions that take functions as their arguments)
- Using broadcasting

You're probably asking yourself in which cases you should use which option. Fortunately, this is mostly a matter of convenience and code readability. In your projects,

use the option that is easiest for you to use and that results in the most readable code. One of the great features of Julia is that all these options are fast. Most of the time, you won't sacrifice performance by choosing one over the other."

Another brief reminder of comprehension - much more can be done ofcourse

```
[17]: [sum(x) for x in [[1,2,3]]]
```

```
[17]: 1-element Vector{Int64}:
       6
```

1.6 Defining collections with parametric types

When we're at the stage where we are ready to start defining completely new, custom methods for our work, and understanding of the Types which our methods accept is a foundational key. Does it work on Intergers? All Intergers or only 8bit? Can it also work if the user accidently plugs in a Float with a single zero digit? Does it take vectors, dicts etc.? What are the parameters for our methods.

A simple demonstration in which we create a vector holding float values - even though we provide intergers, they are automatically parsed to floats.

```
[34]: float_vec = Float64[1,2,3]
```

```
[34]: 3-element Vector{Float64}:
       1.0
       2.0
       3.0
```

If we want a dict in which the keys are UInt8 (8bit unicode interger) and the values are Float64, we can define this immedietly after the **Dict** term

```
[33]: Dict{UInt8, Float64}(0 => 0, 1 => 1)
```

```
[33]: Dict{UInt8, Float64} with 2 entries:
        0x00 => 0.0
        0x01 => 1.0
```

1.6.1 The eltype function

The function **eltype** will tell us the type of elements/values that a **collection** can store - perhaps it's an array only storing floats, or a Tuple storing everything and so on and so on....

```
[35]: eltype(float_vec)
```

```
[35]: Float64
```

This is the cousin to the **typeof()** function which works on the entire collection and gives us a bit more information

```
[37]: typeof(float_vec)
```

```
[37]: Vector{Float64} (alias for Array{Float64, 1})
```

1.6.2 Nuances of types and subtyping

Typing our methods in the correct fashion is essential if we want to avoid naive errors - and by naive I mean using a Type which is almost correct, but just slightly off the mark - for instance, naively restricting our method types to Strings, without really having a reason to, when it's likely better suited to AbstractString, which allows us to input SubStrings, which look identical to Strings in the first place. Or perhaps we want type stability but we aren't restrictive enough in our definition.

Getting a firm grip on the way Julia handles typing is thus always fruitful for us.

Things can definitely get tricky, here is an example of using a collection which is an AbstractVector{Int} into a method which takes AbstractVector{Real}, simply assuming that since Int is a subtype of Real numbers, than by extension AbstractVector{Int} is also likely a subtype of AbstractVector{Real} – but this isn't the case!

```
[38]: isa([1,2,3], AbstractVector{Int})
```

```
[38]: true
```

```
[61]: isa([1,2,3], AbstractVector{Real})
```

```
[61]: false
```

Let's take a look at the all the subtypes of AbstractVector{Real}

```
[62]: subtypes(AbstractVector{Int})
```

```
[62]: 16-element Vector{Any}:
       AbstractRange{Int64}
       AbstractSlices{Int64, 1}
       Base.LogicalIndex{Int64}
       Base.ReinterpretArray{Int64, 1, S} where S
       Base.ReshapedArray{Int64, 1}
       Base.Sort.WithoutMissingVector{Int64}
       Core.Compiler.AbstractRange{Int64}
       Core.Compiler.LinearIndices{1, R} where
      R<:Tuple{Core.Compiler.AbstractUnitRange{Int64}}
       Core.Compiler.TwoPhaseVectorView
       DenseVector{Int64} (alias for DenseArray{Int64,
      1})
       LinearIndices{1, R} where R<:Tuple{AbstractUnitRange{Int64}}
       PermutedDimsArray{Int64, 1}
       AbstractSparseVector{Int64} (alias for
      SparseArrays.AbstractSparseArray{Int64, Ti, 1} where
      Ti)
       SparseArrays.ColumnIndices{Int64, S} where
      S<:(SparseArrays.AbstractSparseMatrixCSC{<:Any, Int64})
```

```
SparseArrays.ReadOnly{Int64, 1, V} where V<:AbstractVector{Int64}
SubArray{Int64, 1}
```

[63]:
```
supertypes(AbstractVector{Real})
```

[63]:
```
(AbstractVector{Real}, Any)
```

When defining our collections and methods, we can specify the types of elements that it is allowed to store/contain - in the following case we want to create an AbstractVector which stores elements of the Real type

[66]:
```
q = AbstractVector{<:Real}
```

[66]:
```
AbstractVector{<:Real} (alias for AbstractArray{<:Real,
1})
```

If we create an AbstractVector of the type Int, we won't be able to house anything but elements of this type inside of it

"Vector{Int} is not a subtype of Vector{Real}. This is because both Vector{Int} and Vector{Real}, as you have seen in this section, can have instances. One is a container that can store only integers. The other is a container that can store any Real values. These are two concrete and different containers. Neither is a subtype of the other."

1.6.3 Using subtyping rules to define functions

Some fun!!!!

[69]:
```
function ourcov(x::AbstractVector{<:Real},
                      y::AbstractVector{<:Real})
           len = length(x)
           @assert len == length(y) > 0
           return sum((x .- mean(x)) .* (y .- mean(y))) / (len - 1)
end
```

[69]:
```
ourcov (generic function with 1 method)
```

[70]:
```
ourcov(1:4, [1.0, 3.0, 2.0, 4.0])
```

[70]:
```
1.3333333333333333
```

We can see that so long as we provide arguments to the function which are of the correct type, we should be OK. What happens if we provide an argument of an incorrect type, which *seems* to look OK but in fact isn't?

"Note that in the code, we mix a range of integers with a vector of floating-point values, and they get accepted and are handled correctly. However, if we pass a collection whose element type is not a subtype of Real, the function will fail, even if we do not change the specific values stored by the collection:"

[71]:
```
ourcov(1:4, Any[1.0, 3.0, 2.0, 4.0])
```

```
MethodError: no method matching ourcov(::UnitRange{Int64}, ::Vector{Any})

Closest candidates are:
  ourcov(::AbstractVector{<:Real}, ::AbstractVector{<:Real})
   @ Main In[69]:1


Stacktrace:
 [1] top-level scope
   @ In[71]:1
```

This fails as Any is not a subtype of Real, in fact, Real is a Subtype of Any.

[74]:
```
typejoin(Int, Float64)
```

[74]:
```
Real
```

[75]:
```
typeof([1,2,3])
```

[75]:
```
Vector{Int64} (alias for Array{Int64, 1})
```

1.6.4 Determing the Type our methods should used based upon the collection at hand

If we have a sample collection of elements which will serve as a reference for our methods/functions, we can determine the Type of the collection by broadcasting the **identity()** function over our collection

[76]:
```
identity.([1.0, 1])
```

[76]:
```
2-element Vector{Float64}:
 1.0
 1.0
```

[77]:
```
identity.(Any[1.0, 1])
```

[77]:
```
2-element Vector{Real}:
 1.0
 1
```

1.7 Using Python code in Julia - calling Python from Julia

We'll use t-SNE to reduce the dimensions of our dataset and visualise them in 2D space. "The t-SNE performs a mapping in such a way that similar objects in the high-dimensional source space are nearby points in the low-dimensional target space, and dissimilar objects are distant points."

[78]:
```
using Random
```

We're gonna generate some random data and have a play

Seed the random generator

[79]:
```
Random.seed!(1234);
```

[80]:
```
cluster1 = randn(100, 5) .- 1
```

[80]:
```
100×5 Matrix{Float64}:
 -0.0293437  -0.737544   -0.613869  -1.31815    -2.95335
 -1.97922    -1.02224    -1.74252   -2.33397    -2.00848
 -0.0981391  -1.39129    -1.87533   -1.76821    -1.23108
 -1.0328     -0.972379    0.600607  -0.0713489  -1.16386
 -1.60079    -3.29076     0.521804   1.71145    -0.113714
 -2.44518    -1.66854    -0.715754  -1.37086     0.630318
  1.70742    -1.78469    -1.87393   -1.44342     0.521595
  0.524448    0.128985   -1.8989    -1.20746    -0.417508
 -0.240196   -0.788823   -1.16236   -0.0609412  -0.30224
 -1.88144    -0.285358   -0.382139  -1.26819    -1.58575
 -0.294007   -0.965934   -0.890225  -2.73204    -2.00704
  0.0915553  -0.431329   -0.896389  -1.78917    -3.25478
 -0.128502   -3.62623    -2.29636   -2.58389    -1.03068
 ⋮
  0.349637   -1.35364    -1.89788    0.600061    0.371505
  0.131929   -0.868554   -0.737022  -1.75019    -1.40852
 -2.3637     -1.03082    -1.12565   -2.20938    -0.321452
 -1.06856    -0.306033   -3.40849   -2.19015    -1.28519
  1.5139     -0.771895   -1.65267   -2.90374    -1.55961
 -0.875938   -2.45727    -1.59571    0.261639   -0.579293
 -0.557989   -1.17941    -0.310314  -0.886134   -1.31935
 -0.344089   -2.66339    -3.08627   -2.44066    -1.27028
 -0.814052   -0.0632304  -1.01902   -0.322153   -1.20563
  2.22591    -0.164603   -0.614049  -1.03229    -0.229297
 -1.55273    -1.09341    -0.823972  -3.41422    -2.21394
 -4.40253    -1.62642    -1.01099   -0.926064    0.0914986
```

[81]:
```
?randn
```

```
search: randn
randn!
randstring
rand
rand!
randexp
Random
randperm
randexp!
```

[81]:

```
randn([rng=default_rng()], [T=Float64], [dims...])
```

Generate a normally-distributed random number of type `T` with mean 0 and standard deviation 1. Optionally generate an array of normally-distributed random numbers. The `Base` module currently provides an implementation for the types `Float16`, `Float32`, and `Float64` (the default), and their `Complex` counterparts. When the type argument is complex, the values are drawn from the circularly symmetric complex normal distribution of variance 1 (corresponding to real and imaginary part having independent normal distribution with mean zero and variance `1/2`).

See also `randn!` to act in-place.

2 Examples

```
julia> using Random

julia> rng = MersenneTwister(1234);

julia> randn(rng, ComplexF64)
0.6133070881429037 - 0.6376291670853887im

julia> randn(rng, ComplexF32, (2, 3))
2×3 Matrix{ComplexF32}:
 -0.349649-0.638457im  0.376756-0.192146im  -0.396334-0.0136413im
  0.611224+1.56403im   0.355204-0.365563im  0.0905552+1.31012im
```

[82]:
```
cluster2 = randn(100, 5) .+ 1
```

[82]:
```
100×5 Matrix{Float64}:
  0.910428    2.13668    0.852595   -0.450324    0.279842
 -0.203334    0.993725   1.86318     0.410499   -0.0472934
 -0.310062    0.608036  -0.0537928   1.48085     1.51439
  1.57447     1.40369    1.44851     1.27623     0.942008
  2.16312     1.88732    2.51227     0.533175   -0.520495
 -0.297068   -0.294909   1.69599    -0.955542    0.460474
  0.326462    1.73068   -0.107294   -0.173673    2.16299
  2.06125     1.23118    2.39091     1.94137     0.99571
  2.04321     0.655377   0.752083    1.51127    -1.03125
  1.50997     0.53043    0.855706    1.14648    -0.473419
  0.283238    1.66014    1.30692     1.63794     2.61739
 -0.194558    1.54699    0.929476   -1.25768     1.16355
  0.852959    3.43833    0.209476    0.0278118   0.657448

  1.745       0.264627  -0.48647     1.7736      1.85603
  1.92047     1.90824    1.80222    -1.18045     1.08781
  0.0598871   0.375741  -0.0903753  -0.0878099   3.18596
  0.341124    1.66711    1.03731     1.60777     0.825171
  1.4519      0.619694   1.31183    -0.281327    1.45072
 -0.601915    1.13995    2.14423     2.00186     1.29112
```

```
  1.43014    -0.124389   1.17537     0.67043     1.46802
  0.364553    1.0528     2.95952     0.526816    0.799745
  1.2327      2.37472    1.31467    -0.290594    3.00592
 -0.198159   -0.211778  -0.726857    0.194847    2.65386
  1.47109     2.61912    1.80582     1.18953     1.41611
  2.77582     1.53736   -0.805129   -0.315228    1.35773
```

Let's concatenate (cat) these matrices vertically using **vcat()** - imagine stacking one matrix on top of the other. Also try using horizontal **hcat()** to see the difference.

```
[83]: vcat(cluster1, cluster2)
```

```
[83]: 200×5 Matrix{Float64}:
 -0.0293437  -0.737544  -0.613869   -1.31815    -2.95335
 -1.97922    -1.02224   -1.74252    -2.33397    -2.00848
 -0.0981391  -1.39129   -1.87533    -1.76821    -1.23108
 -1.0328     -0.972379   0.600607   -0.0713489  -1.16386
 -1.60079    -3.29076    0.521804    1.71145    -0.113714
 -2.44518    -1.66854   -0.715754   -1.37086     0.630318
  1.70742    -1.78469   -1.87393    -1.44342     0.521595
  0.524448    0.128985  -1.8989     -1.20746    -0.417508
 -0.240196   -0.788823  -1.16236    -0.0609412  -0.30224
 -1.88144    -0.285358  -0.382139   -1.26819    -1.58575
 -0.294007   -0.965934  -0.890225   -2.73204    -2.00704
  0.0915553  -0.431329  -0.896389   -1.78917    -3.25478
 -0.128502   -3.62623   -2.29636    -2.58389    -1.03068

  1.745       0.264627  -0.48647     1.7736      1.85603
  1.92047     1.90824    1.80222    -1.18045     1.08781
  0.0598871   0.375741  -0.0903753  -0.0878099   3.18596
  0.341124    1.66711    1.03731     1.60777     0.825171
  1.4519      0.619694   1.31183    -0.281327    1.45072
 -0.601915    1.13995    2.14423     2.00186     1.29112
  1.43014    -0.124389   1.17537     0.67043     1.46802
  0.364553    1.0528     2.95952     0.526816    0.799745
  1.2327      2.37472    1.31467    -0.290594    3.00592
 -0.198159   -0.211778  -0.726857    0.194847    2.65386
  1.47109     2.61912    1.80582     1.18953     1.41611
  2.77582     1.53736   -0.805129   -0.315228    1.35773
```

```
[84]: hcat(cluster1, cluster2)
```

```
[84]: 100×10 Matrix{Float64}:
 -0.0293437  -0.737544   -0.613869  …   0.852595   -0.450324    0.279842
 -1.97922    -1.02224    -1.74252       1.86318     0.410499   -0.0472934
 -0.0981391  -1.39129    -1.87533      -0.0537928   1.48085     1.51439
 -1.0328     -0.972379    0.600607      1.44851     1.27623     0.942008
 -1.60079    -3.29076     0.521804      2.51227     0.533175   -0.520495
 -2.44518    -1.66854    -0.715754  …   1.69599    -0.955542    0.460474
  1.70742    -1.78469    -1.87393      -0.107294   -0.173673    2.16299
  0.524448    0.128985   -1.8989        2.39091     1.94137     0.99571
 -0.240196   -0.788823   -1.16236       0.752083    1.51127    -1.03125
 -1.88144    -0.285358   -0.382139      0.855706    1.14648    -0.473419
 -0.294007   -0.965934   -0.890225  …   1.30692     1.63794     2.61739
  0.0915553  -0.431329   -0.896389      0.929476   -1.25768     1.16355
 -0.128502   -3.62623    -2.29636       0.209476    0.0278118   0.657448

  0.349637   -1.35364    -1.89788      -0.48647     1.7736      1.85603
  0.131929   -0.868554   -0.737022      1.80222    -1.18045     1.08781
 -2.3637     -1.03082    -1.12565   …  -0.0903753  -0.0878099   3.18596
 -1.06856    -0.306033   -3.40849       1.03731     1.60777     0.825171
  1.5139     -0.771895   -1.65267       1.31183    -0.281327    1.45072
 -0.875938   -2.45727    -1.59571       2.14423     2.00186     1.29112
 -0.557989   -1.17941    -0.310314      1.17537     0.67043     1.46802
 -0.344089   -2.66339    -3.08627   …   2.95952     0.526816    0.799745
 -0.814052   -0.0632304  -1.01902       1.31467    -0.290594    3.00592
  2.22591    -0.164603   -0.614049     -0.726857    0.194847    2.65386
 -1.55273    -1.09341    -0.823972      1.80582     1.18953     1.41611
 -4.40253    -1.62642    -1.01099      -0.805129   -0.315228    1.35773
```

"We will want to see if, after using the t-SNE algorithm to perform dimensionality reduction to two dimensions, we will be able to visually confirm that these two clusters are indeed separated."

"If you refer to the examples of using the t-SNE algorithm in the scikit-learn documentation (http://mng.bz/K0oZ), you can see that using Python in Julia is essentially transparent:

- You can call Python functions in exactly the same way as you would call them in Python. In particular, you can use dot (.) to refer to objects in the same way as in Python.
- An automatic conversion occurs between Julia and Python objects, so you do not have to think about it.

This level of integration means that using Python from Julia requires little mental effort for a developer. From my experience, most of the time, fixing the syntax differences is enough if you want to port some Python code to Julia, and things just work. For example, in Julia, string literals require double quotes ("), while typically in Python, a single quote (') is used."

"Julia provides four important ways to iterate over collections and transform them: loops, the map function (and other similar higher-order functions), comprehensions, and broadcasting. Each has slightly different rules of processing data. Therefore, you should choose one depending on your needs in a given situation."