Julia Tutorial

Contents

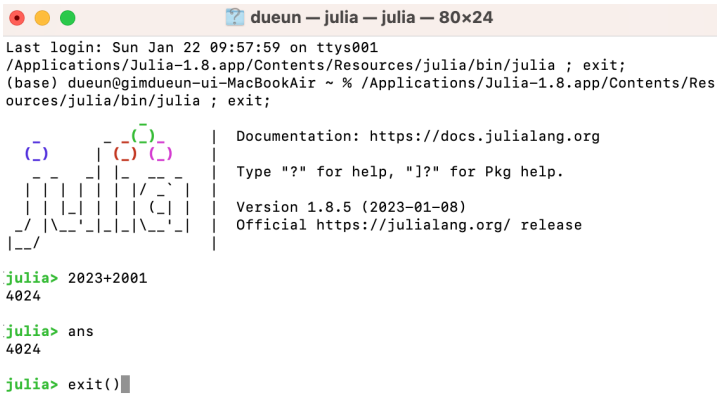
1. 설치 방법과 패키지
2. 기본 문법과 데이터 타입 I
3. 데이터 타입 II
4. 연습문제들
5. Convex.jl

이 문서는 https://julia.quantecon.org/intro.html (https://julia.quantecon.org/intro.html) 를 참고해 만들어졌습니다.
이 문서 외에 줄리아의 시스템적인 부분에 대해 더 궁금하시다면 https://docs.julialang.org/en/v1/ (https://docs.julialang.org/en/v1/)(공식 매뉴얼)을 참고하시기 바랍니다.

1. 설치와 주피터 노트북 연결

만약 jupyter notebook이 설치되지 않았다면 https://www.anaconda.com/ (https://www.anaconda.com/) 에서 anaconda를 먼저 다운로드 받고, 기본 환경을 세팅한 다음 진행해 주세요.

https://julialang.org/ (https://julialang.org/) 에서 자신의 컴퓨터에 맞는 버전을 다운로드 받는다.



실행시킨 뒤 다음과 같이 잘 작동되는지 확인한다. `exit()` 을 입력하면 줄리아가 종료된다.

패키지

줄리아에는 패키지 관리자 역할을 하는 Pkg라는 패키지가 있다. 문자 `]` 를 입력하면 줄리아 REPL(대화창): `>Julia` 에서 Pkg 내부의 REPL: `>Pkg` 로 넘어간다. 백스페이스를 누르면 원래 줄리아의 REPL로 다시 넘어간다.
`Pkg> status` 를 입력하면 설치된 패키지의 목록을 확인할 수 있다.
패키지를 설치하고 싶으면 `Pkg> add 패키지명` 으로 설치한다. 주피터 노트북 상에서는 `] add 패키지명` 으로 입력하면 된다. `Julia> using 패키지명` 을 입력하면 R에서 library를 하는 것처럼 설치된 패키지를 불러올 수 있다.

주피터 노트북과 연결하기

줄리아를 jupyter notebook에서 사용하는 방법은 간단하다.

`Pkg> add IJulia` 를 입력해 IJulia 패키지를 설치하면 줄리아와 주피터 노트북이 자동으로 연결된다. 주피터 노트북을 실행하고 ipynb파일을 열 때, 커널을 기존의 Python 대신 Julia 1.8.5로 선택하면 된다.

*만약 IJulia를 설치했는데, 주피터 노트북에서 커널이 제대로 동작하지 않을 경우 다음 방법으로 해결할 수 있다.
에러 메시지: kernel error ("Failed to precompile in IJulia")

1. 명령 프롬프트에서
   ```
   rm -rfd ~/.julia/compiled
   ```
2. 줄리아 REPL에서
   ```
   using Pkg
   Pkg.update()
   Pkg.precompile()
   ```
3. 줄리아 REPL에서
   ```
   using IJulia
   installkernel("Julia")
   ```

2. 기본 문법, 데이터 타입 I

기본적인 문법은 Python과 거의 동일해 어렵지 않지만, 약간씩 다른 점에 주목해 읽어 보면 좋다.

In [ ]:
```
using LinearAlgebra, Statistics, Plots
```

Primitive Data Types

boolean, float, int, complex

매우 익숙한 형태이다.

In [ ]:
```
x = true
```

In [ ]:
```
typeof(x)
```

In [ ]:
```
y = 1 > 2
```

In [ ]:
```
x = 2; y = 1.0;
```

; 는 출력을 보이지 않게 하거나 여러 변수를 함께 선언할 때 쓰인다.

In [ ]:

```
x = 1 + 2im; y = 1 - 2im;
```

In [ ]:

```
x * y
```

In [ ]:

```
2x - 3y
```

다항식을 쓸 때, `*` 를 생략 가능하다.

In [ ]:

```
@show 2x - 3y
@show x + y;
```

`@show` 를 통해 `print` 를 쓰듯 출력을 전부 보이게 할 수 있다.

Strings

`"` 로 정의한다.

In [ ]:

```
x = "foobar"; typeof(x)
```

문자열 포맷팅도 파이썬과 동일하게 가능하다.

In [ ]:

```
x = 10; y = 20;
```

In [ ]:

```
"x + y = $(x + y)"
```

아래와 같은 전용 함수들 역시 비교적 익숙한 형태이다.

In [ ]:

```
"foo" * "bar" #서로 붙일 때 +가 아닌 *를 사용
```

In [ ]:

```
s = "Charlie don't surf"
```

In [ ]:

```
split(s)
```

In [ ]:

```
replace(s, "surf" => "ski")
```

In [ ]:

```
split("fee,fi,fo", ",")
```

In [ ]:

```
strip(" foobar ")  # 공백 지우기
```

containers

Tuple은 immutable(내부 수정 불가)하고, 여러 타입의 원소를 동시에 포함할 수 있다.

In [ ]:

```
x = ("foo", "bar")
y = ("foo", 2);
```

In [ ]:

```
typeof(x), typeof(y)
```

괄호 없이 생성 가능하며, 각각의 원소를 변수로 취급할 수 있다.

In [ ]:

```
x = "foo", 1
```

In [ ]:

```
word, val = x
println("word = $word, val = $val")
```

`println()` 은 `print()` 뒤에 줄바꿈을 추가한다.

가장 기본적인 container인 array 역시 list와 같이 익숙하지만, Python과 다른 점은 Julia에서는 인덱스가 1부터 시작하고, 슬라이싱을 할 때 마지막 번호까지 포함한다는 것이다.
또 마지막 원소에 접근할 때는 `end` 를 사용하면 된다.

In [ ]:

```
x = [10, 20, 30, 40]
```

In [ ]:

```
x[end]
```

In [ ]:

```
x[end-1]
```

In [ ]:

```
x[1:3]
```

In [ ]:

```
x[2:end]
```

In [ ]:

```
"foobar"[3:end]
```

Dictionary

In [ ]:

```
d = Dict("name" => "Frodo", "age" => 33)
```

In [ ]:

```
d["age"]
```

In [ ]:

```
@show keys(d)
@show values(d);
```

In [ ]:

```
collect(keys(d))
```

key들을 array로 만들 수 있다.

Iterating

반복문의 경우 `for i in ~` 뒤에 `:` 를 쓰지 않고, 끝 부분에 `end` 로 반드시 닫아 주어야 한다.

In [ ]:

```
actions = ["surf", "ski"]
for action in actions
    println("Charlie doesn't $action")
end
```

In [ ]:

```
for i in 1:3
    print(i)
end
```

i번째 원소에 접근하는 방식을 택할 수도, 원소 하나하나에 직접 접근하는 방식을 택할 수도 있다.

In [ ]:

```
x_values = 1:5
```

In [ ]:

```
for x in x_values
    println(x * x)
end
```

In [ ]:

```
for i in eachindex(x_values)
    println(x_values[i] * x_values[i])
end
```

`zip()` 과 `enumerate()` 역시 사용 가능하다.

In [ ]:

```
countries = ("Japan", "Korea", "China")
cities = ("Tokyo", "Seoul", "Beijing")
for (country, city) in zip(countries, cities)
    println("The capital of $country is $city")
end
```

In [ ]:

```
countries = ("Japan", "Korea", "China")
cities = ("Tokyo", "Seoul", "Beijing")
for (i, country) in enumerate(countries)
    city = cities[i]
    println("The capital of $country is $city")
end
```

Comprehensions

매우 편리하고 유려한 방법이다.

In [ ]:

```
doubles = [ 2i for i in 1:4 ]
```

iterable이 2D 이상일 때, element wise하게 더하는 것이 아니라 모든 순서쌍에 대해 전부 더해지는 것에 유의하자.

In [ ]:

```
[ i + j for i in 1:3, j in 4:6 ]
```

In [ ]:

```
[ i + j + k for i in 1:3, j in 4:6, k in 7:9 ]
```

다음처럼 tuple이 들어간 array를 생성할 수도 있다.

In [ ]:

```
animals = ["dog", "cat", "bird"]
```

In [ ]:

```
[ (num = i, animal = j) for i in 1:2, j in animals]
```

Generators

반복 객체(iterable)를 명시적인 객체(array)로 만들지 않으면 속도가 빨라지는 장점이 있다.

In [ ]:

```
xs = 1:10000
f(x) = x^2
f_x = f.(xs)
sum(f_x)
```

`.` 을 사용해 xs의 원소 각각에 f를 적용해 제곱하고 합을 구한다. 이렇게 하면 `f_x` array를 중간에 임시로 생성해야 한다.

In [ ]:

```
f_x2 = [f(x) for x in xs]
@show sum(f_x2)
@show sum([f(x) for x in xs]); # still allocates temporary
```

comprehension을 사용하더라도 중간에 임시적인 array를 생성해야 한다.

In [ ]:

```
using BenchmarkTools
@btime sum([f(x) for x in $xs])
@btime sum(f.($xs))
@btime sum(f(x) for x in $xs);
```

임시 array를 생성하지 않고, 3번째 줄처럼 f(x)를 생성하는 즉시 더하는 방식으로 코드를 짜면 1,000배 이상 빨라진다.

Operators

Python과 거의 동일하다.

In [ ]:

```
x = 1
```

In [ ]:

```
x == 2
```

In [ ]:

```
x != 3
```

In [ ]:

```
1 + 1E-8 ≈ 1
```

`\approx` 를 입력하고 `<TAB>` 을 입력하면 대략적인 값을 비교하는 ≈ 연산자가 된다.

In [ ]:

```
true && false
```

In [ ]:

```
true || false
```

and와 or는 두 개씩 입력해야 한다.

Functions

입력값의 타입은 number, string, array, function 등이 모두 가능하다. 함수를 만들 때에도 반복문처럼 끝에 `end` 로 반드시 닫아 주어야 한다.

In [ ]:

```
function foo(x)
    if x > 0
        return "positive"
    end
    return "nonpositive"
end
```

다만 Python의 lambda처럼 간단한 함수는 아래와 같이 만들 수도 있다.

In [ ]:

```
f(x) = sin(1 / x)
```

In [ ]:

```
f(1 / pi)
```

`map` 을 이용해 주어진 원소 각각에 직접적으로 적용할 수도 있다.

In [ ]:

```
map(x -> sin(1 / x), randn(3))
```

keyword arguments

함수의 입력값에 default 값을 지정할 수 있다.

In [ ]:

```
f(x, a = 1) = exp(cos(a * x))
```

In [ ]:

```
f(pi)
```

In [ ]:

```
f(pi, 2)
```

기본 argument 외에 keyword argument를 사용할 수 있는데, 이것은 순서에 상관없이 keyword에 걸리는 입력값이다. `;` 뒤에 parameter의 이름을 지정해야 한다.

In [ ]:

```
f(x; a) = exp(cos(a * x))
f(pi; a = 2)
```

함수 외부에서 정의된 변수나, named tuple안의 원소 이름이 함수를 정의할 때 사용한 keyword와 같으면 함수의 입력값으로 사용할 수 있다.

In [ ]:

```
nt = (;a=3, b=10)
```

In [ ]:

```
f(pi; nt.a) #만약 f(pi; nt.b)를 입력하면 키워드와 맞지 않아 에러가 난다.
```

Broadcasting

Array 단위로 함수 연산을 하는 것은 아래와 같이 반복문을 통해 쉽게 수행할 수 있다.

In [ ]:

```
x_vec = [2.0, 4.0, 6.0, 8.0]
y_vec = similar(x_vec)
for (i, x) in enumerate(x_vec)
    y_vec[i] = sin(x)
end
```

그러나 위에서 잠시 살펴봤듯, `.` 을 쓰면 간단하게 array의 원소 각각에 함수를 적용할 수 있다. 이것은 우리가 임의로 만든 함수에도 똑같이 사용 가능하다.

In [ ]:

```
y_vec = sin.(x_vec)
```

In [ ]:

```
function chisq(k)
    @assert k > 0
    z = randn(k)
    return sum(z -> z^2, z)  # same as `sum(x^2 for x in z)`
end
```

`@assert` 는 입력값 범위를 제한해주는 매크로이다. 이 함수를 array에 적용하면 아래와 같다.

In [ ]:

```
chisq.([2, 4, 6])
```

이 개념을 잘 활용하면 코드를 굉장히 효율적으로 구성할 수 있다.

In [ ]:

```
x = 1.0:1.0:5.0
y = [2.0, 4.0, 5.0, 6.0, 8.0]
z = similar(y)
z .= x .+ y .- sin.(x)
```

`x` 는 python의 `seq(1.0, 5.0, 1.0)` 과 동일하다. 위에서 보듯, operator 역시 함수와 같기 때문에 `.` 을 활용할 수 있다. 마지막 줄은 아래 매크로로 똑같이 작성할 수 있다.

In [ ]:

```
@. z = x + y - sin(x)
```

`.` 을 사용했을 경우, scalar가 아닌 모든 것에 대해서 broadcasting이 강제로 적용되기 때문에 원하지 않는 부분에 대해서 `Ref()` 로 감싸주어야 한다.

In [ ]:

```
f(x, y) = [1, 2, 3] ⋅ x + y    # "⋅"는 \cdot<tab>으로 입력
@show f([3, 4, 5], 2)
@show f.(Ref([3, 4, 5]), [2, 3]);
```

위의 마지막 줄처럼 두 번째 입력값에 array를 넣고 싶으면 1) `f.` 로 broadcasting 명령을 내리고, 2) `Ref()` 로 첫 번째 입력값을 반드시 감싸 주어야 오류가 나지 않는다.

Closure

어떤 함수의 내부에 중첩되어 있으며, 외부 함수 범위(outer scope)에서 정의되는 값을 참조해 동작하는 함수를 closure라고 한다.

In [ ]:

```
function g(a)
    f(x) = a * x^2
    f(1)
end
g(0.2)
```

함수 g는 a에 대한 함수이다. 그런데, 이 g 내부에 있는 f 함수는 f의 외부값인 a를 참조해 동작하는 것을 확인할 수 있다. 이때 f 함수를 closure라고 한다.
예시를 더 살펴보자. 이런 구조의 코드는 다소 복잡해 보이지만 유용할 때가 많다.

In [ ]:

```
function multiplyit(a, g)
    return x -> a * g(x) # a * g(x)는 closure
end

f(x) = x^2
h = multiplyit(2.0, f)
h(2)
```

아래 두 함수는 동치이다.

In [ ]:

```
function snapabove(g, a)
    function f(x)
        if x > a          # f는 closure, 외부값 a를 참조해 동작
            return g(x)
        else
            return g(a)
        end
    end
    return f # 외부 함수의 return 값은 closure가 됨.
end

f(x) = x^2
h = snapabove(f, 2.0)
plot(h, 0.0:0.1:3.0)
```

In [ ]:

```
function snapabove2(g, a)
    return x -> x > a ? g(x) : g(a) # closure를 반환
end
plot(snapabove2(f, 2.0), 0.0:0.1:3.0)
```

3. 데이터 타입 II

이제 앞에서 배운 데이터 타입에 대해 좀 더 자세하게 알아보자.

Arrays

array는 기본적으로 column vector로 취급된다. 1d array는 vector, 2d array는 matrix라고 한다. 기본 단위는 항상 column vector이다.

In [ ]:

```
Array{Int64, 1} == Vector{Int64}
Array{Int64, 2} == Matrix{Int64}
```

In [ ]:

```
a = [1.0, 2.0, 3.0]
```

array의 형태는 아래 두 함수로 확인할 수 있다.

In [ ]:

```
ndims(a)
```

In [ ]:

```
size(a)
```

array에서 행을 나누는 기준은 `;` 이다. `;` 를 사용해 다양한 형태의 array들을 만들 수 있다.

In [ ]:

```
[1, 2, 3] == [1; 2; 3]  # 둘 다 column vector이다.
```

In [ ]:

```
a = [10 20 30 40]  # row vector는 2d array이다.
```

In [ ]:

```
a = [10 20; 30 40]  # 2 x 2
```

만약 n x 1의 2d array를 만들고 싶다면 row vector를 transpose하면 된다. 다만 이때 만들어진 것은 일반적인 array와 type이 다르다. 따라서 그냥 array로 만들고 싶다면 한번 더 처리해줘야 한다.

In [ ]:

```
a = [10 20 30 40]'
```

In [ ]:

```
b = Matrix(a)
c = collect(a)
b == c
```

; 를 이용해 array의 dimension을 원하는 대로 설정할 수 있다.

In [ ]:

```
[1; 2;] # vector
```

In [ ]:

```
[1; 2;;] # matrix
```

In [ ]:

```
[1; 2;;;] # 3d array
```

In [ ]:

```
[1;;] # 단일 원소 matrix
```

In [ ]:

```
[1 2; 3 4;;;] # 2 x 2 x 1 array
```

reshape() , dropdims 로 차원을 조정할 수 있다. reshape() 함수의 return값은 @views 로 취급함에 주의하자. 따라서 b의 원소 값을 바꾸면 a의 원소도 바뀐다.

In [ ]:

```
a = [10, 20, 30, 40]
```

In [ ]:

```
b = reshape(a, 2, 2)
```

In [ ]:

```
b[1, 1] = 100
```

In [ ]:

```
b
```

In [ ]:

```
a
```

In [ ]:

```
a = [1 2 3 4]
```

In [ ]:

```
dropdims(a, dims = 1)
```

더미 array를 만드는 함수들은 아래와 같다.

In [ ]:

```
zeros(3) # 0.0으로 채워진 array
```

In [ ]:

```
zeros(2, 2) # 0.0으로 채워진 matrix
```

In [ ]:

```
fill(5.0, 2, 2) # 5.0으로 채워진 matrix
```

In [ ]:

```
x = Array{Float64}(undef, 2, 2) # np.empty()와 같음
```

In [ ]:

```
fill(0, 2, 2)  # 0으로 채워진 matrix
```

In [ ]:

```
fill(false, 2, 2)  # boolean matrix
```

기존 데이터를 복사하려면 `copy()` 를 쓴다.

In [ ]:

```
x = [1, 2, 3]
y = copy(x)
y[1] = 2
x
```

기존 데이터와 데이터 타입만 같은 array를 생성하려면 `similar()` 를 쓰면 된다.

In [ ]:

```
x = [1, 2, 3]
y = similar(x)
y
```

In [ ]:

```
x = [1, 2, 3]
y = similar(x, 4)  # int type의 길이 4인 vector
```

In [ ]:

```
x = [1, 2, 3]
y = similar(x, 2, 2)  # int type의 2 x 2 matrix
```

Indexing and Slicing

Array 인덱싱, 슬라이싱의 예를 조금 더 살펴보자. 우리가 이미 아는 것과 별반 다르지는 않다.

In [ ]:

```
a = randn(2, 2)
b = [true false; false true]
```

In [ ]:

```
@show a[1, 1]
@show a[1, :]
@show a[:, 1]
@show a[b];
```

In [ ]:

```
a = zeros(4)
```

In [ ]:

```
a[2:end] .= 42 # .= 으로 broadcasting이 필요
```

In [ ]:

```
a
```

In [ ]:

```
a = [1 2; 3 4]
b = a[:, 2]
@show b
a[:, 2] = [4, 5] # modify a
@show a
@show b;
```

`@views` 매크로를 통해 a의 값을 슬라이싱할 때 값을 따로 저장하지 않고 바로 베껴올 수 있다. 이때 `b` 와 `@views b` 의 타입이 다름에 주의하자.

In [ ]:

```
a = [1 2; 3 4]
@views b = a[:, 2]
@show b
a[:, 2] = [4, 5]
@show a
@show b;
```

Some other features

1. Julia에서는 adjoint처럼 특수한 타입을 일반적인 matrix와 같게 취급하지 않고 따로 기억한다. 이런 특성 덕분에 더 효율적인 연산이 가능하다.

In [ ]:

```
d = [1.0, 2.0]
a = Diagonal(d)
```

In [ ]:

```
@show 2a
b = rand(2,2)
@show b * a;
```

In [ ]:

```
b = [1.0 2.0; 3.0 4.0]
b - I
```

In [ ]:

```
typeof(I)
```

2. assignment: `a = b` 와 `a .= b` 는 단순히 a와 b가 같다는 의미(name bind) / mutable a에 b의 값을 assign해주는 의미로 서로 다르다.

In [ ]:

```
x = [1 2 3]
y = x
z = [2 3 4]
y = z       # y = x -> y = z로 bind만 바꿈.
@show (x, y, z);
```

In [ ]:

```
x = [1 2 3]
y = x
z = [2 3 4]
y .= z      # y의 값을 바꾸면 y = x bind에 의해 x 값도 바뀜.
@show (x, y, z);
```

scalar는 immutable이어서 assignment가 불가능하다.

In [ ]:

```
y = [1 2]
y .-= 2    # y .= y .- 2

x = 5
# x .-= 2  # fails!
x = x - 2  # 새로운 값을 생성해 x를 다시 bind하는 것으로 조금 다르다.
```

tuple, static array 등의 타입도 immutable로 assignment가 불가능하다.

In [ ]:

```
using StaticArrays
xdynamic = [1, 2]
xstatic = @SVector [1, 2]  # tuple과 유사한 성질의 array이다.

f(x) = 2x
@show f(xdynamic)
@show f(xstatic)

# inplace version
function g(x)
    x .= 2x
    return "Success!"
end
@show xdynamic
@show g(xdynamic)
@show xdynamic;

# g(xstatic) # fails!
```

Operations

몇 가지 유용한 method들과 연산에 대해 좀 더 알아보자.

In [ ]:

```
a = [-1, 0, 1]

@show length(a)
@show sum(a)
@show mean(a)
@show std(a)      # standard deviation
@show var(a)      # variance
@show maximum(a)
@show minimum(a)
@show extrema(a)  # (mimimum(a), maximum(a))
```

In [ ]:

```
b = sort(a, rev = true)
```

In [ ]:

```
b = sort!(a, rev = true) # inplace = True
```

In [ ]:

```
b == a # same values?
```

In [ ]:

```
b === a # identical? (same values + stored in same memory)
```

* 는 matrix multiplication을 의미한다.

In [ ]:

```
a = ones(1, 2)
```

In [ ]:

```
b = ones(2, 2)
```

In [ ]:

```
a * b
```

In [ ]:

```
b * a'
```

In [ ]:

```
b = ones(2, 2)
b * ones(2)
```

In [ ]:

```
ones(2)' * b
```

선형 방정식 $AX = B$를 풀려면 `A \ B` 를 쓰면 된다.

In [ ]:

```
A = [1 2; 2 3]; B = ones(2, 2);
```

In [ ]:

```
A \ B == inv(A) * B
```

행렬식, 대각합, 고유값, 랭크도 편리하게 구할 수 있다.

In [ ]:

```
A = [1 2; 3 4]
```

In [ ]:

```
det(A)
```

In [ ]:

```
tr(A)
```

In [ ]:

```
eigvals(A)
```

In [ ]:

```
rank(A)
```

내적은 `dot()` 나 `\cdot<TAB>` 을 통해서 가능하다.

In [ ]:

```
@show ones(2)' * ones(2)
@show dot(ones(2), ones(2))
@show ones(2) ⋅ ones(2);
```

Julia가 기존과 가장 다른 점 중 하나는 위에서도 이미 언급했듯 `.` 을 사용한 elementwise operation이다.

In [ ]:

```
ones(2, 2) * ones(2, 2)
```

In [ ]:

```
ones(2, 2) .* ones(2, 2)
```

In [ ]:

```
A = -ones(2, 2)
```

In [ ]:

```
A.^2
```

In [ ]:

```
x = [1, 2]
x .+ 1     # not x + 1
x .- 1     # not x - 1
```

값 비교 역시 동일하게 가능하다.

In [ ]:

```
a = [10, 20, 30]
```

In [ ]:

```
b = [-100, 0, 100]
```

In [ ]:

```
b .> a
```

In [ ]:

```
a .== b
```

In [ ]:

```
b .> 1
```

In [ ]:

```
a = randn(4)
```

In [ ]:

```
a[a .< 0]
```

In [ ]:

```
a = [10, 20, 30, 40]
```

Ranges

벡터와 동일한 성질을 가진다.

In [ ]:

```
a = 10:12        # 10:1:12 = 10, 11, 12
@show Vector(a)  # 바꿀 순 있지만 바꿀 이유가 없다.

b = Diagonal([1.0, 2.0, 3.0])
b * a .- [1.0; 2.0; 3.0]
```

In [ ]:

```
a = 0.0:0.1:1.0 # 0.0, 0.1, ... , 1.0
```

In [ ]:

```
maxval = 1.0
minval = 0.0
stepsize = 0.15
a = minval:stepsize:maxval # 0.0, 0.15, 0.3, ... 이 경우에 0.9에서 끊어짐.
maximum(a) == maxval
```

아래처럼 `np.linspace()` 와 동일하게 사용할 수 있다.

In [ ]:

```
maxval = 1.0
minval = 0.0
numpoints = 10
a = range(minval, maxval, length=numpoints)
# or range(minval, stop=maxval, length=numpoints)

maximum(a) == maxval
```

Tuples and Named Tuples

Immutable이지만 array보다 더 빠르다.

In [ ]:

```
t = (1.0, "test")
t[1]             # index로 접근
a, b = t         # unpack
# t[1] = 3.0     # fails!
println("a = $a and b = $b")
```

In [ ]:

```
t = (;val1 = 1.0, val2 = "test")
t.val1      # name으로 접근
println("val1 = $(t.val1) and val2 = $(t.val2)")
(;val1, val2) = t  # unpacking notation (note the ;)
println("val1 = $val1 and val2 = $val2")
```

In [ ]:

```
t2 = (;val3 = 4, val4 = "test!!")
t3 = merge(t, t2)  # new tuple
```

named tuple은 많은 parameter들을 가진 함수를 만들 때 편리하다.

In [ ]:

```
function f(parameters)
    (;α, β) = parameters
    return α + β
end

parameters = (;α = 0.1, β = 0.2)
f(parameters)
```

In [ ]:

```
using Parameters
paramgen = @with_kw (α = 0.1, β = 0.2)  # default값을 가진 named tuple을 생성

@show paramgen()
@show paramgen(α = 0.2)
@show paramgen(α = 0.2, β = 0.5);
```

In [ ]:

```
function paramgen2(;α = 0.1, β = 0.2)
    return (;α, β)
end
@show paramgen2()
@show paramgen2(;α = 0.2);
```

Nothing, Missing

개념적으로 각각 값이 아예 필요없을 때, 결측치일 때를 의미한다고 보면 좋다. 고유의 type을 가진다.

In [ ]:

```
typeof(nothing)
```

보통 함수에서 $x \notin \mathcal{D}$ 일때 많이 사용하게 된다.

In [ ]:

```
function f(y)
    x = nothing
    if y > 0.0
        x = y
    end

    if isnothing(x)
        println("x was not set")
    else
        println("x = $x")
    end
    x
end

@show f(1.0)
@show f(-1.0);
```

다음과 같이 `ifelse` 와 유사한 형태로 함수를 만들 수도 있다.

In [ ]:

```
function f(x)
    x > 0.0 ? sqrt(x) : nothing  # the "a ? b : c" pattern is the ternary
end

f(1.0)
```

In [ ]:

```
x = [0.1, -1.0, 2.0, -2.0]
y = f.(x)
```

이때 return type은 union이라는 새로운 type임을 확인할 수 있다. 이런 domain check 상황에서는 `@assert` 를 사용해도 좋다. 다만 이 경우 fail시에 에러가 발생한다는 단점이 있다.

In [ ]:

```
function f(x)
    @assert x > 0.0
    sqrt(x)
end

f(-1.0)
```

error를 다루는 또다른 방법으로 `try` 와 `catch` 가 있다. `catch` 로 에러를 잡아낸 다음 `err end` 를 입력하면 알 수 있는 에러의 종류를 프린트해 준다. 그 에러가 예상할 수 있는 것이라면 적절한 조치를 취해줄 수도 있다.

In [ ]:

```
try sqrt(-1.0); catch err; err end
```

In [ ]:

```
function f(x)
    try
        sqrt(x)
    catch err
        sqrt(complex(x, 0))
    end
end

f(0.0)
f(-1.0)
```

마지막으로 missing을 살펴보자. 위에서 언급했듯 R의 NA에 대응된다고 생각하면 된다.

In [ ]:

```
typeof(missing)
```

In [ ]:

```
x = [3.0, missing, 5.0, missing, missing]
```

missing은 nothing과 달리 함수에 집어넣을 수 있다. 다만 값은 missing이다.

In [ ]:

```
f(x) = x^2

@show missing + 1.0
@show missing * 2
@show missing * "test"
@show f(missing);
@show mean(x);
@show x == missing # 같은 missing과 비교해도 missing이 나온다.
@show ismissing(x);
```

missing을 무시하거나 없애고 싶다면 아래처럼 하면 된다.

In [ ]:

```
@show mean(skipmissing(x))
@show coalesce.(x, 0.0);  # replace missing with 0.0;
```

4. 연습문제들

솔루션은 맨 밑에 있습니다.

Exercise 1

$X \sim B(n, p)$를 generate하는 함수 `binomial_rv(n, p)` 를 $U(0, 1)$에서 random number를 generate하는 함수인 `rand()` 만을 사용해 작성하세요.

In [ ]:

```
# your solution
```

Exercise 2

몬테카를로 시뮬레이션을 활용해 π의 값을 추정하는 코드를 작성하세요. 단 random number를 이용할 때 `rand()` 만을 사용하세요.

힌트는 다음과 같습니다:

- U 가 unit square $(0, 1)^2$ 영역의 bivariate uniform random variable일 때, U가 $(0, 1)^2$의 부분집합 B 위의 점이 될 확률은 B의 넓이와 같습니다.
- $U_1, \dots, U_n \overset{iid}{\sim} U$라고 할 때, n이 커질수록 $\{U_n\}$이 B 위의 점으로 나타난 비율은 U가 B 위의 점이 될 확률로 수렴합니다.
- 원의 넓이는 πr^2 입니다.

In [ ]:

```
# your solution
```

Exercise 3

아래의 데이터는 미국 도시의 인구 수입니다. 이것을 txt파일로 만들고 저장하는 코드는 아래와 같습니다.

In [ ]:

```
pwd() # current working directory
```

In [ ]:

```
open("us_cities.txt", "w") do f
  write(f,
"new york: 8244910
los angeles: 3819702
chicago: 2707120
houston: 2145146
philadelphia: 1536471
phoenix: 1469471
san antonio: 1359758
san diego: 1326179
dallas: 1223229")
end
```

이 파일을 읽어와서 합을 구하는 코드를 작성하세요.

hint:

- `eachline(f)` 를 통해 파일 한줄한줄에 iterable로 접근할 수 있습니다. ex) `for line in eachline(f)`
- `parse(Int, "100")` 함수로 `"100"` 을 정수로 바꿀 수 있습니다.
- Python과 매우 유사합니다!

In [ ]:

```
# your solution
```

Exercise 4

다음의 stochastic differential equation이 있습니다. 행렬 계산 연습용이므로 배경지식은 무시하셔도 좋습니다.

$$X_{t+1} = AX_t + b + \Sigma W_{t+1} \tag{4.1}$$

- X_t, b and X_{t+1} are $n \times 1$
- A is $n \times n$
- Σ is $n \times k$
- W_t is $k \times 1$ and $\{W_t\}$ is iid with zero mean and variance-covariance matrix equal to the identity matrix

$n \times n$행렬 S_t를 X_t의 공분산행렬로 정의합시다.

$\{S_t\}$에 의해 다음 시점의 공분산행렬 $\{S_{t+1}\}$은 다음과 같이 정의됩니다.

$$S_{t+1} = AS_tA' + \Sigma\Sigma' \tag{4.2}$$

A의 eigenvalue가 모두 unit circle 안에 속하면(여기에서는 신경쓸 필요가 없습니다), $\{S_t\}$는 유일한 극한값 S로 수렴하게 됩니다. 이론적인 값으로 아래의 (4.3)이 주어져 있습니다.

위의 (4.2)와 맨 아래 주어진 실제 행렬 값을 활용해 S를 추정하는 함수 `compute_aymptotic_var` 를 만들어 주세요.

반복을 이어가다 값이 `1e-6` 이내로 변할 때, 즉 수렴할 때 반복을 멈추면 됩니다. 마지막으로 (4.3)값과 우리가 수렴시킨 값을 비교해 보세요.

$$S = ASA' + Q \quad \text{where} \quad Q := \Sigma\Sigma' \tag{4.3}$$

$$A = \begin{bmatrix} 0.8 & -0.2 \\ -0.1 & 0.7 \end{bmatrix} \qquad \Sigma = \begin{bmatrix} 0.5 & 0.4 \\ 0.4 & 0.6 \end{bmatrix}$$

In [ ]:

```
# your solution

```

solutions

먼저 안 보고 20분만 고민해 보기!

exercise 1

In [ ]:

```
function binomial_rv(n, p)
    U = rand(n)
    return sum(U .< p)
end
```

exercise 2

자세한 설명은 https://bigdata-doctrine.tistory.com/15 (https://bigdata-doctrine.tistory.com/15) 참고

In [ ]:

```
n = 1000000
U = rand(n,2)
circle = 0
for i in 1:n
    if U[i,1]^2 + U[i,2]^2 <= 1
        circle += 1
    end
end
circlearea = circle/n
pi = 4*circlearea
```

exercise 3

In [ ]:

```
f_ex = open("us_cities.txt", "r")
total_pop = 0
for line in eachline(f_ex)
    city, population = split(line, ':')  # tuple unpacking
    total_pop += parse(Int, population)
end
close(f_ex)
println("Total population = $total_pop")
```

exercise 4

In [ ]:

```
function compute_asymptotic_var(A, Σ;
                                S0 = Σ * Σ',
                                tolerance = 1e-6,
                                maxiter = 500)
    V = Σ * Σ'
    S = S0
    err = tolerance + 1
    i = 1
    while err > tolerance && i ≤ maxiter
        next_S = A * S * A' + V
        err = norm(S - next_S)
        S = next_S
        i += 1
    end
    return S
end
```

In [ ]:

```
A = [0.8  -0.2;
     -0.1  0.7]

Σ = [0.5 0.4;
     0.4 0.6]
```

In [ ]:

```
our_solution = compute_asymptotic_var(A, Σ)
```

5. Convex.jl

교재 저자인 Stephan Boyd 교수가 matlab 기반으로 만든 패키지인데, 누군가 그것을 줄리아로 번역해 놓았다. 이름 그대로 convex optimization 문제를 편리하게 formulation 하고 풀 수 있는 패키지이다. Python 버전인 CVXPY도 있다.

설치

In [ ]:

```
] add Convex
```

In [ ]:

```
] add ECOS # numerical solver
```

In [ ]:

```
] add SCS # numerical solver
```

In [ ]:

```
using Convex, SCS
```

Variable

문제를 풀어서 optimum을 찾고 싶은 변수를 뜻한다.

In [ ]:

```
# Scalar variable
x = Variable()

# Column vector variable
x = Variable(5)

# Matrix variable
x = Variable(4, 6)
```

Variable들은 아래 예시처럼 어떤 특성과 함께 선언될 수 있다.

- (entrywise) positive: `x = Variable(4, Positive())`
- (entrywise) negative: `x = Variable(4, Negative())`
- integral: `x = Variable(4, IntVar)`

- binary: `x = Variable(4, BinVar)`
- (for a matrix) being symmetric, with nonnegative eigenvalues (ie, positive semidefinite): `z = Semidefinite(4)`

순서는 size, sign, Convex.VarType (i.e., integer, binary, or continuous) 순이며 필요없으면 생략해도 된다. variable x의 현재 값은 `evaluate(x)` 로 확인할 수 있다. 문제를 풀면 x의 값은 optimal이 될 것이다.

Expressions

variable에 연산을 수행한 것을 expression이라고 한다.

In [ ]:

```
x = Variable(5)
# The following are all expressions
y = sum(x)
z = 4 * x + y
z_1 = z[1]
```

Convex 패키지로 문제를 푸는 기본 구조는 다음과 같이 variable과 expression, problem을 정의하고 `solve!` 로 푸는 것이다.

Constants

expression 안의 variable을 제외한 숫자, 벡터, 행렬들을 의미한다.

In [ ]:

```
x = Variable()
y = Variable()
z = Variable()
expr = x + y + z
problem = minimize(expr, x >= 1, y >= x, 4 * z >= y)
solve!(problem, SCS.Optimizer)

# Once the problem is solved, we can call evaluate() on expr:
evaluate(expr)
```

Constrains

부등식 꼴로 정의되며, strict한 경우(등호가 없는 경우)와 아닌 경우가 구분되지 않는다.

In [ ]:

```
x = Variable(5, 5)
# Equality constraint
constraint = x == 0
# Inequality constraint
constraint = x >= 1
```

In [ ]:

```
x = Variable(3, 3)
y = Variable(3, 1)
z = Variable()
# constrain [x y; y' z] to be positive semidefinite
constraint = ([x y; y' z] in :SDP)
# or equivalently,
constraint = ([x y; y' z] ⪰ 0)
```

아래처럼 variable을 construct한 다음 constraint를 추가하면, x를 사용하는 모든 문제에 자동으로 implicit constraint가 추가된다.

In [ ]:

```
x = Variable(3)
add_constraint!(x, sum(x) == 1)
```

Objective

Objective는 목적 함수로 `maximize` 나 `minimize` 에 묶인 expression을 뜻한다. Feasibility problem은 constant를 objective로 설정하거나, `problem = satisfy(constraints)` 로 표현 가능하다.

Problem

Problem은 sense (minimize, maximize, or satisfy), objective, constraints로 구성된다.

In [ ]:

```
problem = minimize(objective, constraints)
# or
problem = maximize(objective, constraints)
# or
problem = satisfy(constraints)
```

Constraints를 정의한 후 문제에 추가할 수 있다.

In [ ]:

```
# No constraints given
problem = minimize(objective)
# Add some constraint
problem.constraints += constraint
# Add many more constraints
problem.constraints += [constraint1, constraint2, ...]
```

정의된 문제는 solve!를 통해 풀 수 있다. 문제가 solve된 뒤, `problem.status` 가 문제의 status를 분류한다. 분류 목록은 `:Optimal` , `:Infeasible` , `:Unbounded` , `:Indeterminate` , `:Error` 가 있다. 만약 `:Optimal` 이면 `problem.optval` 에 optimum value가 기록된다. `evaluate()` 를 통해 variable의 optimum, objective의 optimum을 둘 다 확인할 수 있다.

In [ ]:

```
solve!(problem, solver)
```

Example

다음 least squares problem을 풀어 보자.

$$minimize \ \|Ax - b\|_2^2$$

$$subject\ to\ x \succeq 0$$

이때 variable $x \in \mathbb{R}^n$, 데이터 $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$이다.
이 문제를 Convex 패키지를 이용해 푸는 코드는 아래와 같다.

In [ ]:

```
# Generate random problem data
m = 4;  n = 5
A = randn(m, n); b = randn(m, 1)

# Create a (column vector) variable of size n x 1.
x = Variable(n)

# The problem is to minimize ||Ax - b||^2 subject to x >= 0
# This can be done by: minimize(objective, constraints)
problem = minimize(sumsquares(A * x - b), [x >= 0])

# Solve the problem by calling solve!
solve!(problem, SCS.Optimizer; silent_solver = true)

# Check the status of the problem
@show problem.status # :Optimal, :Infeasible, :Unbounded etc.

# Get the optimum value
@show problem.optval;
```

수고하셨습니다!