LU factorization with Julia.

```
julia> using LinearAlgebra
```

load linear algebra subroutine library...

BLAS
LAPACK
UMFPack

```
julia> lu
lu (generic function with 9 methods)

help?> lu
```

calls different routines depending on the arguments...

press the [?] key to get help mode

Component	Description
F.L	L (lower triangular) part of LU
F.U	U (upper triangular) part of LU
F.p	(right) permutation Vector
F.P	(right) permutation Matrix

Note matrices with lots of 0 entries are called sparse, in this case one can work with much larger matrices by not storing the zeros....

```
julia> A=[1 2 3; 4 5 6; 2 0 -3]
3×3 Matrix{Int64}:
 1  2   3
 4  5   6
 2  0  -3
```

from last time

$$A = P^T L U = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}^T \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{4} & -\frac{3}{10} & 1 \end{bmatrix} \begin{bmatrix} 4 & 5 & 6 \\ 0 & -5/2 & -6 \\ 0 & 0 & -3/10 \end{bmatrix}$$

```
julia> r=lu(A)
LU{Float64, Matrix{Float64}}
L factor:
3×3 Matrix{Float64}:
 1.0    0.0   0.0
 0.5    1.0   0.0
 0.25  -0.3   1.0
U factor:
3×3 Matrix{Float64}:
 4.0   5.0   6.0
 0.0  -2.5  -6.0
 0.0   0.0  -0.3
```

L =

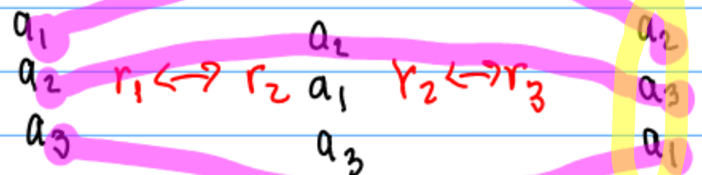
U =

```
julia> r.P
3×3 Matrix{Float64}:
 0.0  1.0  0.0
 0.0  0.0  1.0
 1.0  0.0  0.0
```

```
julia> typeof(r)
LU{Float64, Matrix{Float64}}
```

```
julia> r.p
3-element Vector{Int64}:
 2
 3
 1
```

1 → 2, 2 → 3, 3 → 1



follow the arrows the other direction...

```
julia> invperm(r.p)
3-element Vector{Int64}:
 3
 1
 2
```

1 → 3, 2 → 1, 3 → 2

```
julia> r.L*r.U
3×3 Matrix{Float64}:
 4.0  5.0   6.0
 2.0  0.0  -3.0
 1.0  2.0   3.0
```

← A with permuted rows

```
julia> r.P'*r.L*r.U
3×3 Matrix{Float64}:
 1.0  2.0   3.0
 4.0  5.0   6.0
 2.0  0.0  -3.0
```

← A

```
julia> (r.L*r.U)[invperm(r.p),:]
3×3 Matrix{Float64}:
 1.0  2.0   3.0
 4.0  5.0   6.0
 2.0  0.0  -3.0
```

← another way to find A using the permutation vector rather than the matrix...

Internal representation of LU factorization...

```
julia> fieldnames(typeof(r))
(:factors, :ipiv, :info)

julia> r.factors
3×3 Matrix{Float64}:
 4.0    5.0   6.0
 0.5   -2.5  -6.0
 0.25  -0.3  -0.3
```

$$L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{4} & -\frac{3}{10} & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 4 & 5 & 6 \\ 0 & -5/2 & -6 \\ 0 & 0 & -3/10 \end{bmatrix}$$

Idea keep track of the multipliers used in the elimination steps by storing them in place of the zeros in U. Then when row swaps are performed during pivoting the multipliers automatically get swapped as well.

```
julia> A=rand(10000,10000);

julia> @time lu(A);
  4.619843 seconds (5 allocations: 763.016 MiB, 0.75% gc time)
```

If you write your own code for Gaussian elimination with partial pivoting, it will (almost surely) run slower...

Finding LU factorization

each variable has to be eliminated.

n variables
n equations
n length of a row operation

roughly n^3 amount of work

more accurately accounted...

$$\frac{2}{3}n^3 - \frac{1}{2}n^2, \approx 6.6661\ldots \times 10^{11}$$ operations

```
julia> (2/3*n^3-1/2*n^2)/4.619843
1.4429422529438046e11
```

Flops ⟸ floating point operation per second.

144 G FLOPS

Peta- means 1,000,000,000,000,000; a Pet
Exa- means 1,000,000,000,000,000,000; ar

1	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786

1206,000,000 GFLOPS

Solving $Ax=b$ using Gauss elimination