Arrays

Basically lists of things

In [1]:
```
list1 = [0 ,1, 3, "asd"]
```

Out[1]:
```
4-element Vector{Any}:
 0
 1
 3
  "asd"
```

Access to it is done using square brackets - rememeber julia has 1 indexing instead of 0 indexing unlike python/c/...

In [2]:
```
list1[0]
```

```
BoundsError: attempt to access 4-element Vector{Any} at index [0]

Stacktrace:
 [1] getindex(A::Vector{Any}, i1::Int64)
   @ Base ./array.jl:801
 [2] top-level scope
   @ In[2]:1
 [3] eval
   @ ./boot.jl:360 [inlined]
 [4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, cod
e::String, filename::String)
   @ Base ./loading.jl:1116
```

In [3]:
```
list1[1]
```

Out[3]:
```
0
```

You can also access elements at the end

In [4]:
```
list1[end]
```

Out[4]:
```
"asd"
```

In [5]:
```
list1[end-1]
```

Out[5]:
```
3
```

The type of a list is the most common ancestor of all the types of things in the list

In [6]:
```
intList = [0,1,2,3,4]
```

Out[6]:
```
5-element Vector{Int64}:
 0
 1
 2
 3
 4
```

In [7]:
```
floatList = [0,1.1,2341.12]
```

Out[7]:
```
3-element Vector{Float64}:
    0.0
    1.1
 2341.12
```

In [8]:
```
complexList = [0, 1+im, 0.1]
```

Out[8]:
```
3-element Vector{ComplexF64}:
 0.0 + 0.0im
 1.0 + 1.0im
 0.1 + 0.0im
```

In [9]:
```
anyList = [0, 1, "string"]
```

Out[9]:
```
3-element Vector{Any}:
 0
 1
  "string"
```

Although this doesn't matter as much as in C as it is dynamically typed, meaning you can just insert a string into an int list for example without impunity in Julia, this will actually make your code run slower. This is because Julia is just-in-time compiled (if you want more details I can expand in office hours) so keeping a list to a specific type will speed things up.

Vectors and Matrices

For this class an in general for mathematical computing, we mainly care about 1D/2D arrays of numbers (vectors/matrices). The package in Julia for a lot of the operations you will be using (e.g. inverse, determinants, ...) is called LinearAlgebra

In [10]:
```
using LinearAlgebra
```

In [11]:
```
vec = [1, 0, 0]
```

Out[11]:
```
3-element Vector{Int64}:
 1
 0
 0
```

In [12]:
```
A = [ [1,0,0] [0,1,1] [0,2,3] ]
```

Out[12]:
```
3×3 Matrix{Int64}:
 1  0  0
 0  1  2
 0  1  3
```

Indexing is done as follows, not for a matrix you need two index numbers as it is a 2D array. The index is [row,column]

In [13]:
```
vec[1]
```

Out[13]:
```
1
```

In [14]:
```
A[1,2]
```

Out[14]:
```
0
```

You can do the expected operations, such as matrix multiply, addition subtraction of vectors and matrices:

In [15]:
```
A*vec
```

Out[15]:
```
3-element Vector{Int64}:
 1
 0
 0
```

In [16]:
```
vec2 = [2,3,1]
```

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

In [17]:
```
vec+vec2
```

Out[17]:
```
3-element Vector{Int64}:
 3
 3
 1
```

One of the most common commands is to solve the equation Ax = b (i.e. calculate $A^{-1}b)$, you could do this explicitly

In [18]:
```
invA = inv(A)
invA * vec
```

Out[18]:
```
3-element Vector{Float64}:
 1.0
 0.0
 0.0
```

Or there is special syntax for this that does not explicitly compute the inverse of A:

In [19]:
```
A \ vec
```

Out[19]:
```
3-element Vector{Float64}:
 1.0
 0.0
 0.0
```

It turns out computing the inverse explicitly is a really bad idea! Let's see why with a randomly generated matrix of size 5000:

In [20]:
```
Arand = rand( 5000, 5000 );
brand = rand( 5000, 1 );
```

In [21]:
```
@time Arand \ brand;
```

```
  2.055235 seconds (436.01 k allocations: 215.919 MiB, 0.61% gc time,
8.77% compilation time)
```

In [22]:
```
@time inv(Arand) * brand;
```

```
  5.123473 seconds (2.22 M allocations: 310.770 MiB, 0.95% gc time, 1
1.81% compilation time)
```

So it looks like backslash is 2-3 times faster! In fact for a lot of matrices backslash can be even faster than that compared to direct inversion (this is because \ has some very fancy algorithms behind it QR, Cholesky, PLU, ..., ask me if you want to know more)!

Plotting

I believe for this course we will mainly be using the PyPlot package. There do exist others in Julia (e.g. Plots, GR, Makie, ...) but we will stick with PyPlot for ease of use.

In [23]:
```
using PyPlot
```

To generate a simple line plot, we need an array of x and y values:

In [24]:
```
xvals = LinRange(0,1,20)
```

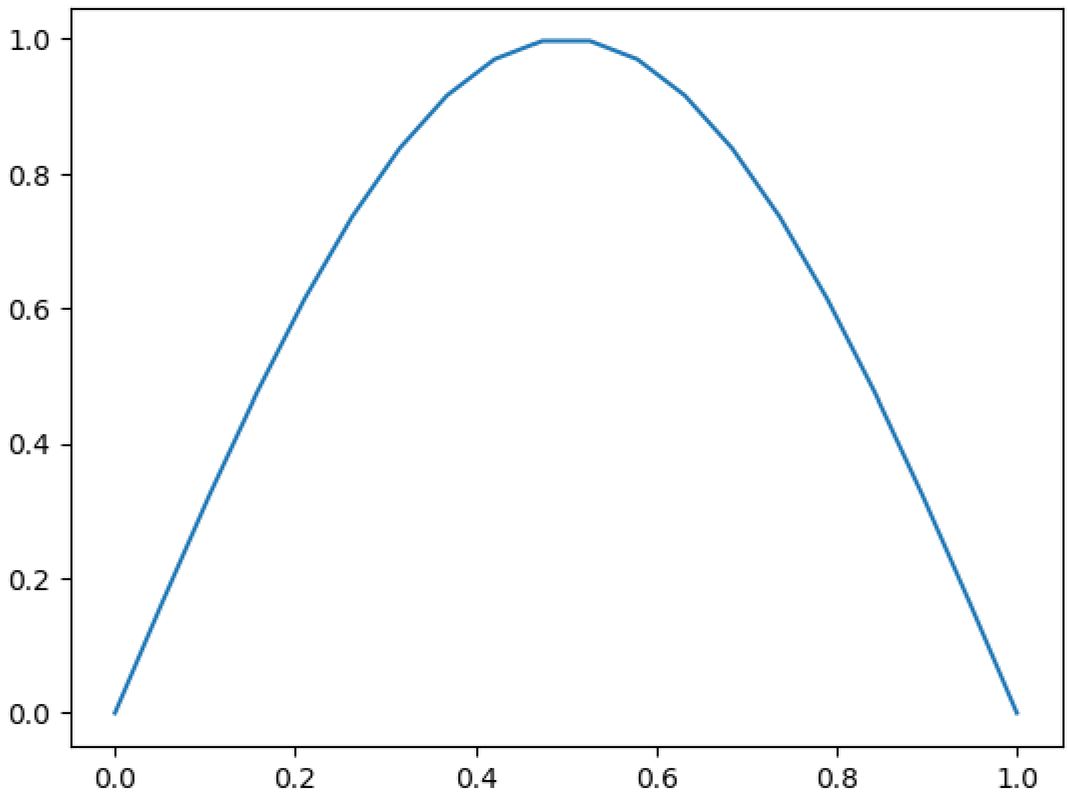
Out[24]:
```
20-element LinRange{Float64}:
 0.0,0.0526316,0.105263,0.157895,0.210526,…,0.842105,0.894737,0.94736
8,1.0
```

The above code generates 20 points equidistant from 0 to 1. Now to get some y values, let's say of the $\sin(\pi x)$ function:

In [25]:
```
sinvals = [ sin(pi*x) for x in xvals ];
```

I used something here called a list comprehension, it is basically a shortcut for a for loop in that it applies the specified function (here sin) to all x in xvals. To plot it:

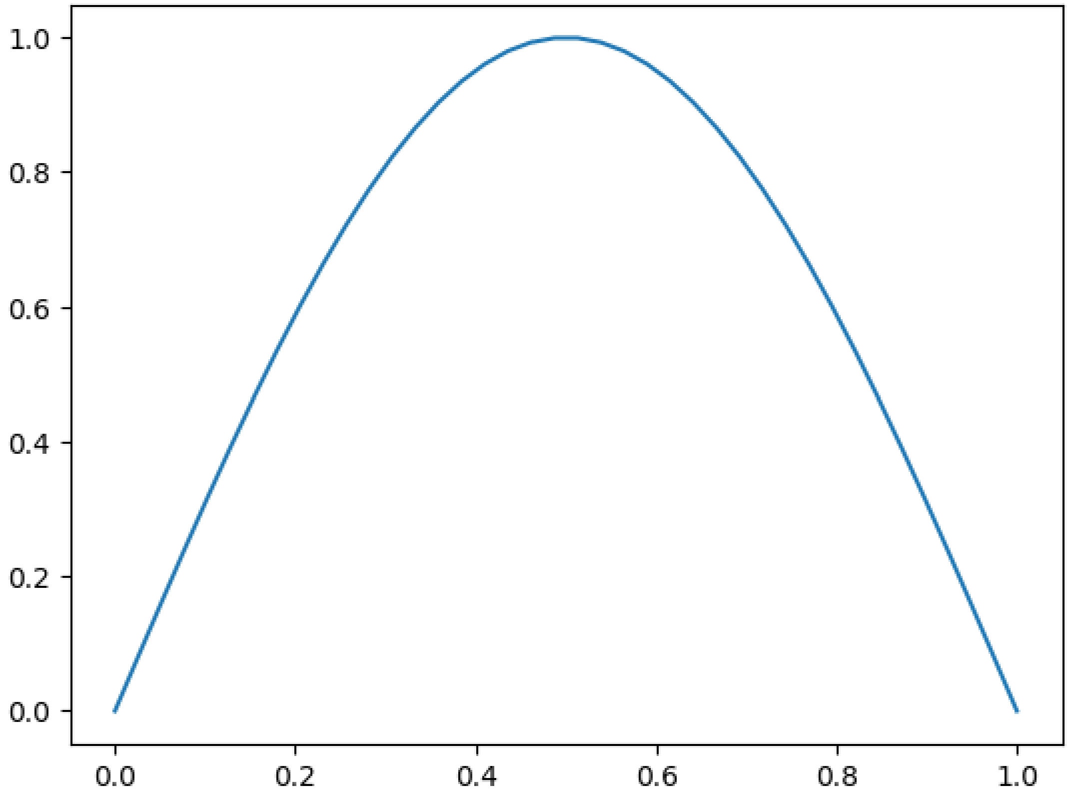
In [26]:
```
plot( xvals, sinvals )
```



Out[26]:
```
1-element Vector{PyCall.PyObject}:
 PyObject <matplotlib.lines.Line2D object at 0x7f82e1cf3b50>
```

It looks a bit rough so can increase number of sample points:

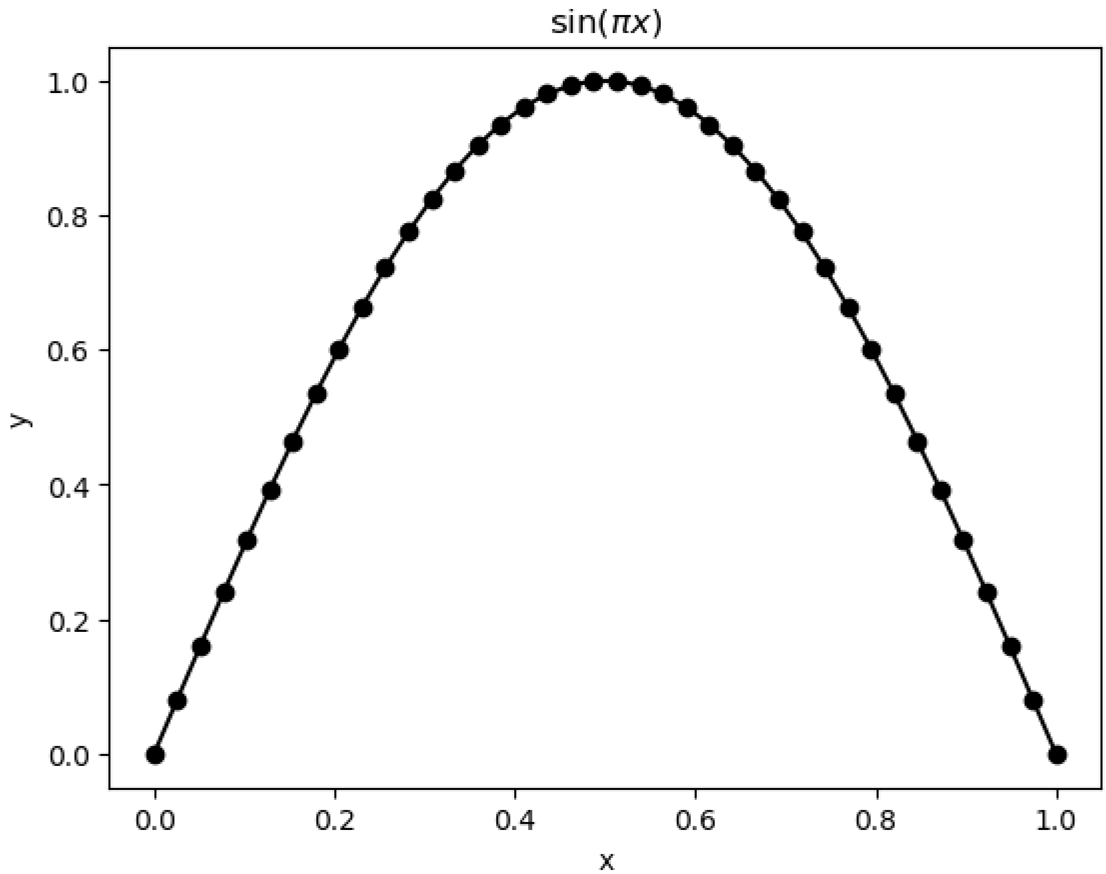
In [27]:
```
xvals = LinRange(0,1,40);
sinvals = [ sin(pi*x) for x in xvals ];
plot( xvals, sinvals )
```



Out[27]:
```
1-element Vector{PyCall.PyObject}:
 PyObject <matplotlib.lines.Line2D object at 0x7f82e20859d0>
```

And that looks better! To label the axes:

In [28]:
```
plot( xvals, sinvals, "ko-" ) #The "k" means its a black curve, o turns
xlabel("x")
ylabel("y")
title(L"\sin(\pi x)")
```



Out[28]:
```
PyObject Text(0.5, 1, '$\\sin(\\pi x)$')
```

The `L` in front of the string tells PyPlot that it is LaTeX notation.

There are lots of other possible commands to customise your plots - google them!

In [ ]: