Object Orientation

In Julia object orientation is done using structs - same as classes in python, matlab etc. They are a way of compartmentalising data, for example

In [1]:
```
struct Dog
    name::String
    age::Int64
    friends::Vector{Dog}
end
```

To initialise an instance of the struct, in this case a dog:

In [2]:
```
dog1 = Dog( "Kevin", 1, [] )
```

Out[2]:
```
Dog("Kevin", 1, Dog[])
```

In [3]:
```
dog1.name
```

Out[3]:
```
"Kevin"
```

In [4]:
```
dog1.age
```

Out[4]:
```
1
```

In [5]:
```
dog1.age = 2
```

```
setfield! immutable struct of type Dog cannot be changed

Stacktrace:
 [1] setproperty!(x::Dog, f::Symbol, v::Int64)
   @ Base ./Base.jl:34
 [2] top-level scope
   @ In[5]: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
```

Once a field of a struct is set, it can't be changed. This is a design thing in Julia that differs from python, this is due to speed.

In [6]:
```
dog2 = Dog( "Anna", 2, [] )
```

Out[6]:
```
Dog("Anna", 2, Dog[])
```

In [7]:
```
dog1.friends
```

Out[7]:
```
Dog[]
```

In [8]:
```
push!( dog1.friends, dog2 )
```

Out[8]:
```
1-element Vector{Dog}:
 Dog("Anna", 2, Dog[])
```

In [9]:
```
dog1.friends
```

Out[9]:
```
1-element Vector{Dog}:
 Dog("Anna", 2, Dog[])
```

But note you can push to arrays that are fields of structs. This is because the array is mutable, however

In [10]:
```
dog1.friends = []
```

```
setfield! immutable struct of type Dog cannot be changed

Stacktrace:
 [1] setproperty!(x::Dog, f::Symbol, v::Vector{Any})
   @ Base ./Base.jl:34
 [2] top-level scope
   @ In[10]: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
```

So you can mutate the object that the field is storing if it is mutable but you can't change the field itself.

It's quite tedious to always have to initialise the empty friend array. You can bypass this by defining the constructor:

In [11]:
```
struct Dog
    name::String
    age::Int64
    friends::Vector{Dog}
    Dog( name, age, friends=[] ) = new( name, age )
end
```

```
invalid redefinition of constant Dog

Stacktrace:
 [1] top-level scope
   @ In[11]:1
 [2] eval
   @ ./boot.jl:360 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, cod
e::String, filename::String)
   @ Base ./loading.jl:1116
```

Note that in Julia you can't redefine structs. You either have to make a new name or restart the kernel to redefine a struct.

In [12]:
```
struct Dog2
    name::String
    age::Int64
    friends::Vector{Dog2}
    Dog2( name, age ) = new( name, age, Vector{Dog2}() )
end
```

In [13]:
```
dog1 = Dog2( "Kevin",1 )
```

Out[13]:
```
Dog2("Kevin", 1, Dog2[])
```

In [14]:
```
dog1.friends
```

Out[14]:
```
Dog2[]
```

So this time it automatically creates a the friends array without us having to input it as an argument.

Function arguments

It is common for functions to take in arguments that must be of certain type. Why? Because of efficiency and also sometimes it just makes sense. For example if we have a function multiply, it should take it only numbers, and wouldn't make sense to take in a string or a colour for instance.

In [15]:
```
function makeFriends!( d1::Dog2, d2::Dog2 )

    push!( d1.friends, d2 );
    push!( d2.friends, d1 );

end
```

Out[15]:
```
makeFriends! (generic function with 1 method)
```

In [16]:
```
dog2 = Dog2( "Anna",2 );
```

In [17]:
```
makeFriends!( 1,2 )
```

```
MethodError: no method matching makeFriends!(::Int64, ::Int64)

Stacktrace:
 [1] top-level scope
   @ In[17]:1
 [2] eval
   @ ./boot.jl:360 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, cod
e::String, filename::String)
   @ Base ./loading.jl:1116
```

In [18]:
```
makeFriends!( dog1, dog2 )
```

Out[18]:
```
1-element Vector{Dog2}:
 Dog2("Kevin", 1, Dog2[Dog2("Anna", 2, Dog2[#= circular reference @-4
=#])])
```

So the function only works on the correct types.

Inheritance/Abstract Classes

Classes can inherit/be a subclass of a parent class. HOWEVER IN JULIA PARENT CLASSES MUST BE ABSTRACT CLASSES AND NOT CONCRETE CLASSES.

An abstract class is a class that cannot be instantiated. Dog and Dog2 above are not abstract classes as they can be instantiated.

In [23]:
```
abstract type Animal end;

struct Dog3 <: Animal
    name::String
    age::Int64
    friends::Vector{Animal}
    Dog3( name, age ) = new( name, age, Vector{Animal}() )
end

struct Cat3 <: Animal
    name::String
    age::Int64
    friends::Vector{Animal}
    Cat3( name, age ) = new( name, age, Vector{Animal}() )
end
```

In [20]:
```
dog1 = Dog3( "Kevin", 1 )
```

Out[20]:
```
Dog3("Kevin", 1, Animal[])
```

In [21]:
```
isa( dog1, Dog3 )
```

Out[21]:
```
true
```

In [24]:
```
isa( dog1, Animal )
```

Out[24]:
```
true
```

In [27]:
```
cat1 = Cat3( "Steve", 10 )
```

Out[27]:
```
Cat3("Steve", 10, Animal[])
```

So here both Dog3 and Cat3 are types of animals. Note you can't actually create an instance of animal as it is an abstract class. Why is this useful?

In [25]:
```
function makeFriends!( a1::Animal, a2::Animal )

    push!( a1.friends, a2 );
    push!( a2.friends, a1 );

end
```

Out[25]:
```
makeFriends! (generic function with 2 methods)
```

In [28]:
```
makeFriends!( dog1, cat1 )
```

Out[28]:
```
1-element Vector{Animal}:
 Dog3("Kevin", 1, Animal[Cat3("Steve", 10, Animal[#= circular referen
ce @-4 =#])])
```

It means for these functions you can actually act on more than one type.

In [29]:
```
cat1.friends
```

Out[29]:
```
1-element Vector{Animal}:
 Dog3("Kevin", 1, Animal[Cat3("Steve", 10, Animal[#= circular referen
ce @-4 =#])])
```

Overloading Operators

It is possible to redefine Julia built in functions to work on your own types too. This is called overloading

In [42]:
```
function Base.show(io::IO, a::Animal)
    print("Name: ", a.name, ", Age: ", a.age, ", Number of friends: ",
end
```

In [39]:
```
println( dog1 )
```

```
Name: Kevin, Age: 1, Number of friends: 1
```

To do it you just write Base.(some function name) and redefine what you want it to do. Here's another example

In [40]:
```
function Base.:+( a1::Animal, a2::Animal )
    if a1.age > a2.age
        return a1
    end
    return a2
end
```

In [41]:
```
dog1 + cat1
```

Out[41]:

```
Name: Steve, Age: 10, Number of friends: 1
```

So I have defined add on animals to return the older one. Note for the binary operators you need the colon before it.

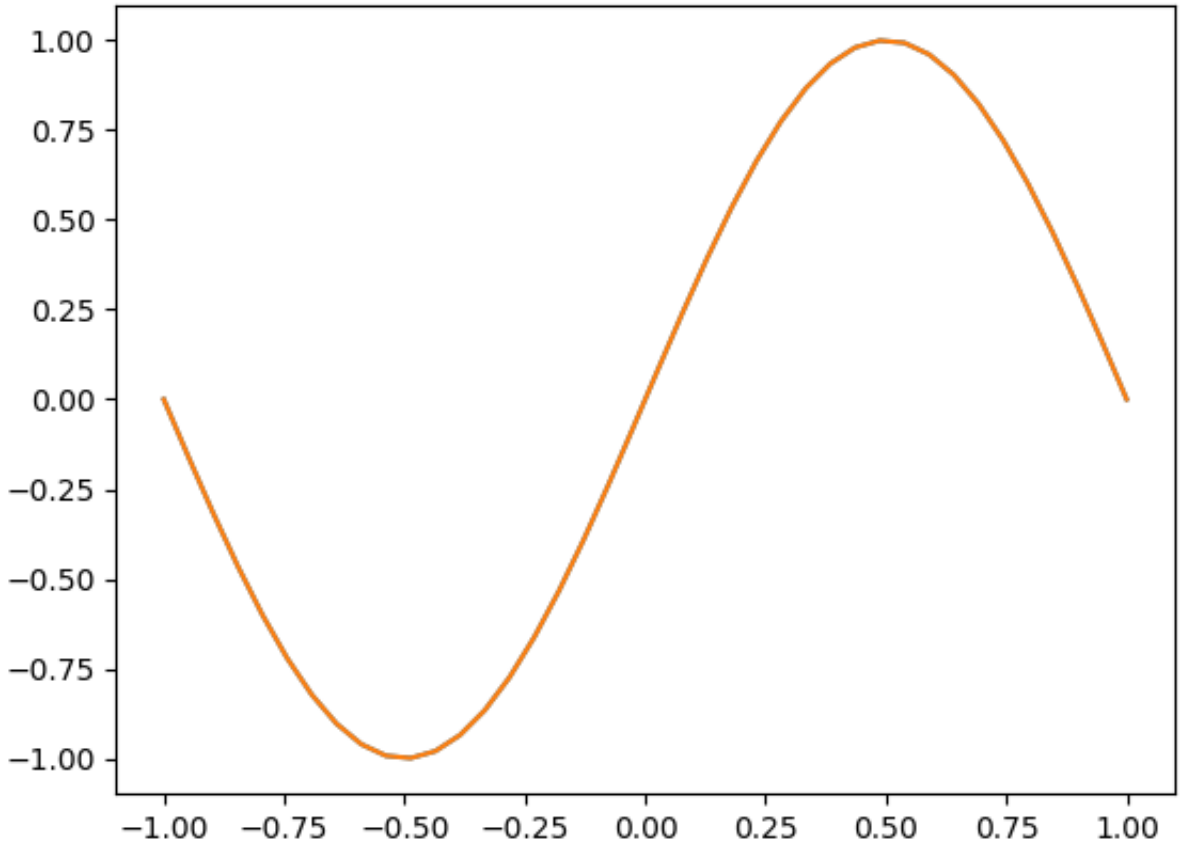
Filtering

Filtering is a way to reduce noise in input data (it actually has lots of other uses as well, for example in PDEs, optics, ... a very long list). The idea is that if we have some sort of error, we can hope that the error is not overly biased, meaning that if we sum all the errors up it should be some predictable value (like 0). So by taking various averages of our input data, we can hopefully smooth out the error.

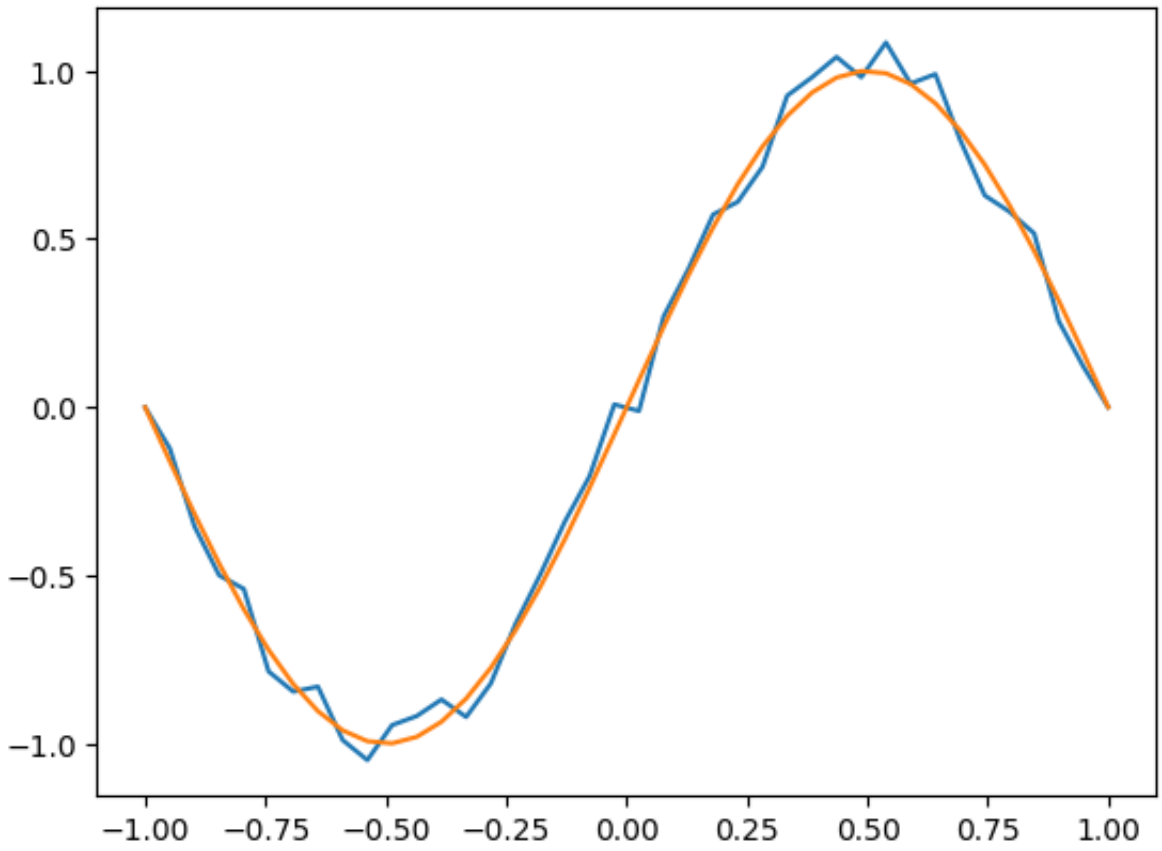
In [43]:
```
using PyPlot
```

In [69]:
```
xpts = LinRange( -1,1,40 )
ypts = [ sin(x*pi) for x in xpts ]

plot( xpts, ypts );
plot( xpts, ypts );
```



In [70]:
```
strength = 0.1;
data = ypts .+ 0.0;
data[2:end-1] += 2.0*(rand( 38 ).-0.5)*strength;
plot( xpts, data );
plot( xpts, ypts );
```



Let's now apply the mean filter by taking the mean of each point and it's neighbours:

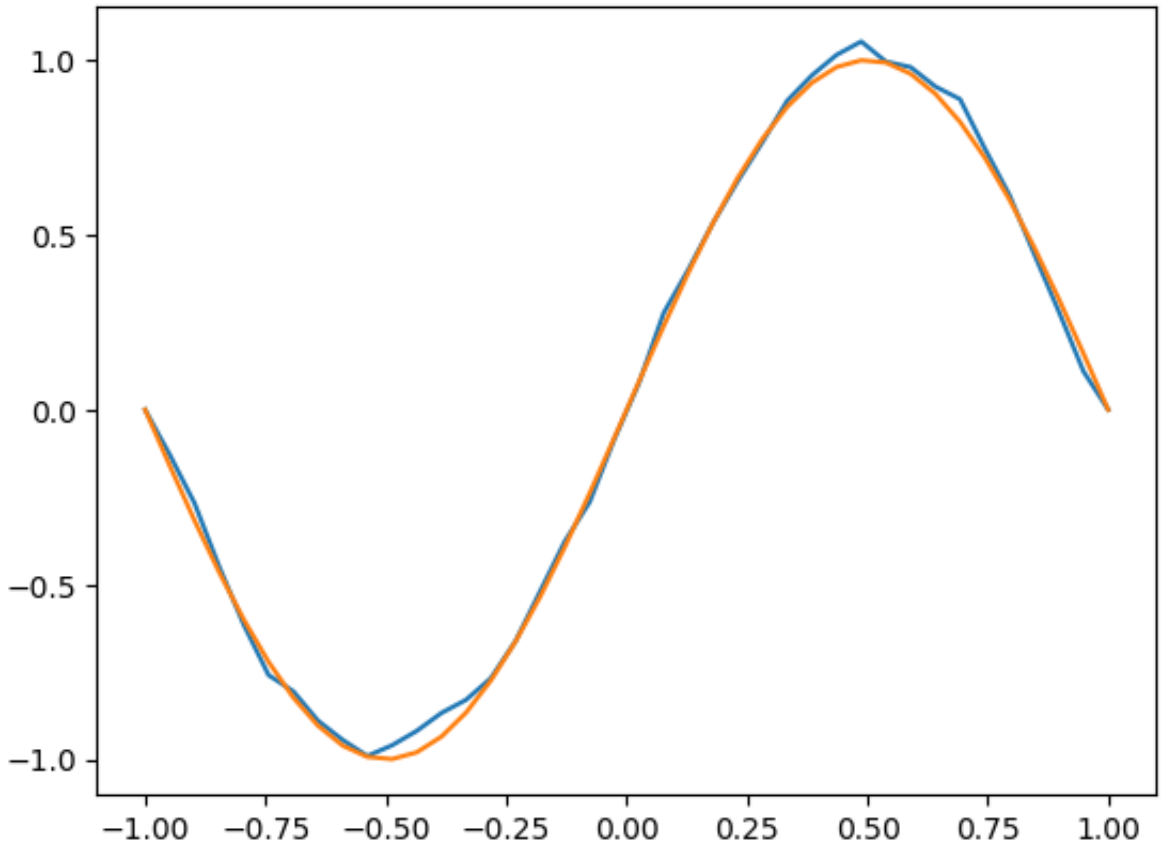
In [64]:
```
function meanFilter( data )

    ndata = [ data[1] ]
    for ii = 2:length(data)-1
        push!( ndata, ( data[ii-1]+data[ii]+data[ii+1] ) / 3 )
    end
    push!( ndata, data[end] )
    return ndata

end
```

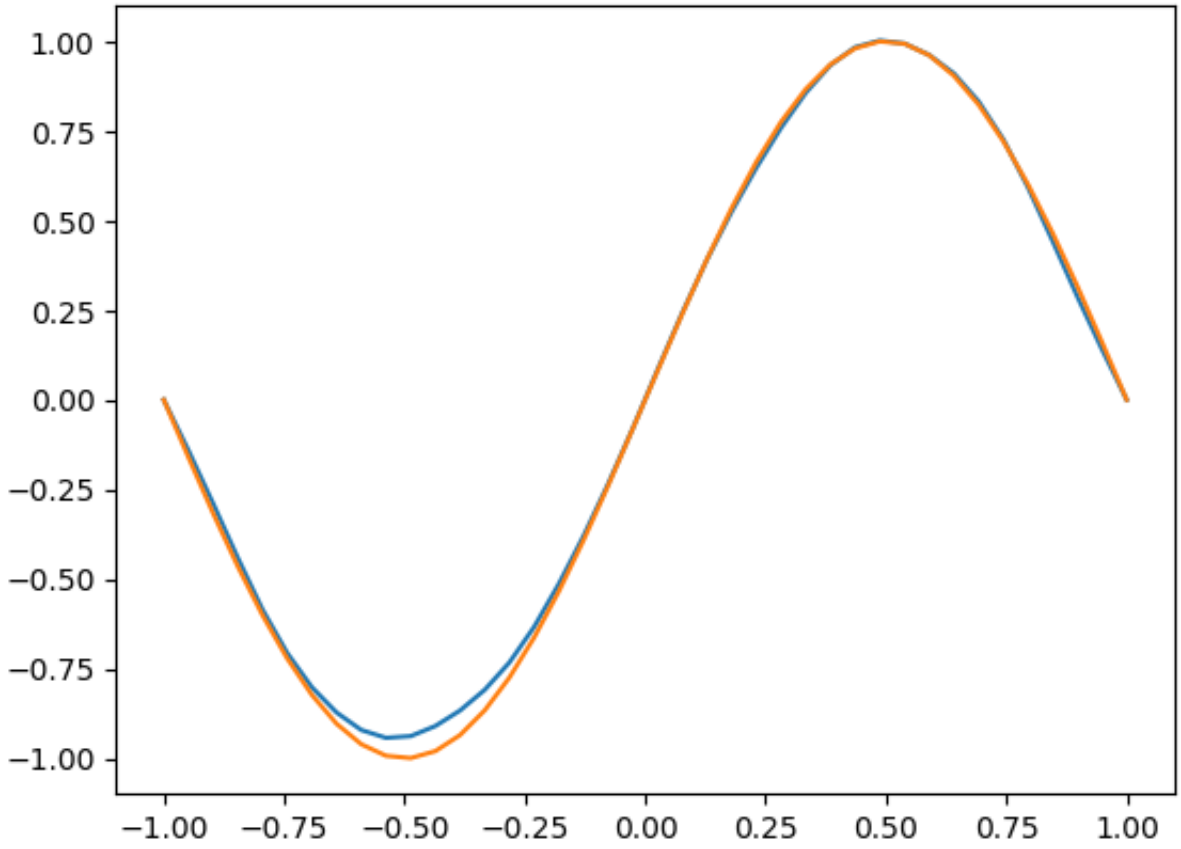
Out[64]:
```
meanFilter (generic function with 1 method)
```

In [66]:
```
meandata = meanFilter( data );
plot( xpts, meandata );
plot( xpts, ypts );
```



So it's clearly better. You can actually filter again:

In [68]:
```
meandata = meanFilter( meandata );
meandata = meanFilter( meandata );
plot( xpts, meandata );
plot( xpts, ypts );
```



Will it ever be perfect? Of course not in general, as if we have really bad noise, we can't really expect it to give us the exact answer back. But it can give us data that is smooth enough in general that it is workable.

A final note - there are way better filters than this in real life of course. They can filter out different things like high frequencies, low frequencies, or even specific ones. This is useful when for example you know your channel adds noise of a specific range of frequencies which is how most things in the real world work.

This is way out of scope for this course - but if you are interested feel free to come ask me. It is also a very active area of research so there is plenty to read about/investigate if you are interested!

In [ ]: