Ch3_Scaling_projects_in_Julia

March 21, 2024

1 Chapter 3 - Julia's support for scaling projects

"In this chapter, you will learn elements of the Julia language that are important when creating larger projects. We start with exploring Julia's type system. Understanding how type hierarchy works is essential to learning how to define multiple methods for a single function, a topic we started discussing in section 2.4. Similarly, when you use an existing function, you must know how to find out which types of arguments it accepts. Getting an exception because you tried to pass an argument of incorrect type when calling a function is one of the most common errors when working in Julia. To avoid such problems, you must have a good understanding of how Julia's type system is designed."

1.1 Julia's type system

A highly touted feature of julia is *multiple dispatch* - essentially the ability to define multiple methods for a function, allowing it to work on different types. It's power is ground breaking.

Let's look at the several methods for **cd()** which changes directories. Programming multiple methods reduces the chaces of your code breaking and anticipates multiple use scenarios.

[1]:
```
methods(cd)
```

[1]:
```
# 4 methods for generic function "cd" from Base.Filesystem:
 [1] cd()
     @ file.jl:94
 [2] cd(f::Function)
     @ file.jl:147
 [3] cd(f::Function, dir::AbstractString)
     @ file.jl:107
 [4] cd(dir::AbstractString)
     @ file.jl:89
```

As we can see there are two types for cd, the *Function* and *AbstractString* type - naturally, every function which is defined, has a *Function* supertype in Julia. An abstract string could be any string consists of any characters, a string in its most abstract form

[3]:
```
cd isa Function
```

[3]:
```
true
```

But if we ask if cd is a **typeof()** Function then we get false! What? It seems contradictory no?

[4]:
```
typeof(cd) == Function
```

[4]:
```
false
```

The reason for this, is because Julia organises it's types in a hierarchy, with the supertype **Function** being the parents, and other subtypes belogning to it. When we execute the statement above, it evaluates as false because it defaults to the specific subtype, rather than the supertype.

[9]:
```
supertype(typeof(cd))
```

[9]:
```
Function
```

We have a "type tree" that we need to visualise - the root is the *Any* type - this is the default type that Julias compiler asigns to standard functions.

"Only the types that are leaves can have instances (that is, have objects that are of that specific type). The types that can be instantiated are called concrete. In other words, if you have a value, you can be sure that its type is concrete and that it is a leaf type. For this reason, there is no function whose type is Function. Every function has its own unique concrete type that is a subtype of the Function type."

1.1.1 Concrete vs Abstract types

Only concrete types can be instantiated and cannot have concrete subtypes. You can check whether a given type is concrete by using the **isconcretetype()** function. Abstract types cannot have instances but can have subtypes. You can check whether a given type is abstract by using the **isabstracttype()** function. Therefore, it is not possible for a type to be both abstract and concrete.

However, some types are neither abstract nor concrete. You will encounter these types in chapter 4 when you learn more about parametric types. An example of such a type is Vector. (Note that this type has its parameter left out, and this is why it is not concrete; in section 2.1, you saw an example of a value having Vector{Int}, which is a concrete type as it has a fully specified parameter, Int in that case.)

[15]:
```
supertypes(AbstractFloat)
```

[15]:
```
(AbstractFloat, Real, Number, Any)
```

Iterate through the type tree

[37]:
```
function sub_types(type)
    println(type)
    for t in subtypes(type)
        sub_types(t)
    end
    return nothing
end
```

[37]:
```
sub_types (generic function with 1 method)
```

[36]:
```
sub_types(Integer)
```

```
Integer
Bool
Signed
BigInt
Int128
Int16
Int32
Int64
Int8
Unsigned
UInt128
UInt16
UInt32
UInt64
UInt8
```

1.1.2 Union types in function definitions

Say we want to define a single function (not redefine it with another type), which accepts two different types, we would specify to the function that we're acception a Union of the two types, meaning the argument could be either/or - Union{Int64, Int32} would allow either 64 or 32 bit integers. Or Singed and Unsigned integers but not booleans. So where would we insert this information?

`function fun(x::Union{Signed, Unsigned})` is what we may do

Now, if we're specifying the types that our function can take, we need to be quite careful in assigning the correct level of abstraction or generality. For instance, 1:3 is the same as [1,2,3] and for most cases, the same as [1.0,2.0,3.0] - BUT these will likely have different type tree structures, and so choosing a type which is not shared by these three different representations will lead to error. Let's example it!

[42]:
```
supertypes(typeof(1:3))
```

[42]:
```
(UnitRange{Int64}, AbstractUnitRange{Int64}, OrdinalRange{Int64, Int64},
AbstractRange{Int64}, AbstractVector{Int64}, Any)
```

[44]:
```
supertypes(typeof([1,2,3]))
```

[44]:
```
(Vector{Int64}, DenseVector{Int64}, AbstractVector{Int64}, Any)
```

[45]:
```
supertypes(typeof([1.0,2.0,3.0]))
```

[45]:
```
(Vector{Float64}, DenseVector{Float64}, AbstractVector{Float64}, Any)
```

We can see that indeed, they are represented differently. However, they do share a common node/branch in the tree, and so this would be the Type we would assign to our function in order to allow the input of all three.

```
function stepper(step::AbstractVector)
end
```

Julia has a very handy base function called **typejoin(typeof(x), typeof(y))** which will decide this for us, finding the intersection between the different types!

[47]:
```
typejoin(typeof(1:3), typeof([1,2,3]))
```

[47]:
```
AbstractVector{Int64} (alias for AbstractArray{Int64,
1})
```

1.2 Modules

Modules allow the encapsulation and export of code to other workspaces where julia is being executed. When we specify that we want to import a package using `using X`, we are importing this **module** X. We can declade that our programs are part of a parcelised module using the `module MyModule; end` specification.

[2]:
```
module ExampleModule

function funthings(x)
    println(x)
end

end # ExampleModule
```

```
WARNING: replacing module ExampleModule.
```

[2]:
```
Main.ExampleModule
```

If someone creates a module that is intended to be shared with other Julia users, it can be registered with the Julia general registry (https://github.com/JuliaRegistries/General). These modules must have a special structure, and after being registered, they become available as packages. You can find instructions for managing packages in appendix A.

Knowing how to use modules that are bundled into packages is important for a data scientist. You have two basic ways to make the functionality of an installed package usable in your code: using the import or using keyword arguments. When you use import, only the module name is brought into the scope of your code. To access variables and functions defined by the module, you need to prefix their names with the module name, followed by a dot. Here is an example:

[3]:
```
import Statistics
```

[4]:
```
x = [1,2,3] ; mean(x)@edit winsor(x, count=10^5)
```

```
UndefVarError: `mean` not defined

Stacktrace:
 [1] top-level scope
```

```
@ In[4]:1
```

As we can see, an error is thrown, since we brought the package into our workspace using the **import** function, we have to call the mean function using the package name and dot syntax

[6]:
```
Statistics.mean(x)
```

[6]:
```
2.0
```

Now if we added the packaged to this workspace using the **using** function, than we could call the mean function simply using `mean(x)` - this would bring the packages scope into the current global sphere

In most Julia code, you can safely employ the using statement, and this is what people normally do. You already know the reason: the Julia language can automatically detect if a name you are trying to use conflicts with an identical name already introduced with, for example, the using keyword. In such cases, you will be informed that there is a problem.

1.3 Macros

To reveal the code transfromation that has taken place when calling a macro, we can reveal what's behind the curtains and expand the macro using the macro `@macroexpand`

An important aspect of using the @benchmark macro is that we use `$x` instead of just x. This is needed to get a correct assessment of execution time of the expressions we check. As a rule, remember to prefix with `$` all global variables you use in the expressions you want to benchmark (this applies only to benchmarking and is not a general rule when using macros). For details about this requirement, refer to the documentation of the BenchmarkTools.jl package (https://github.com/JuliaCI/BenchmarkTools.jl). The short explanation is as follows. Recall that since x is a global variable, code using it is not type stable. When the @benchmark macro sees the `$x`, it is instructed to turn the x variable into one that is local (and thus type stable) before running the benchmarks.

[7]:
```
x = rand(10^6)
```

[7]:
```
1000000-element Vector{Float64}:
 0.5759691287696864
 0.30891945247573427
 0.7510091297588427
 0.7046815227435752
 0.9400881257332492
 0.2480082205772437
 0.2218067144050777
 0.8288235895313576
 0.8731878495825949
 0.31828501427159295
 0.5959072514933209
 0.8112912703600579
```

```
0.4483970959651261

0.8760315502180783
0.4091799004162652
0.058765736792229184
0.6264302991227584
0.053908091953408954
0.16808272679418423
0.45481189935269795
0.049276215552853664
0.5052723471844284
0.9252416483641728
0.7468276777790201
0.9633311675838864
```

"Types of variables have a hierarchical relationship and form a tree. The root of the tree is the Any type that matches any value. Types that have subtypes are called abstract and cannot have instances. Types that can have instances cannot have subtypes and are called concrete."

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]: