Session 3: Types, type inference and stability

OBJECTIVE: Demonstrate the dynamic programming features of Julia.

KR1: Shown or demonstrated the hierarchy of Julia's type hierarchy using the command subtypes(). Start from Number and use subtypes() to explore from down to . Use supertype() to determine the abstract type.

The root type is `Any`. When I use subtypes on it, it shows that it has 513 subtypes.

Interestingly, `Any` is a subtype of `Any`.

In [1]:
```
subtypes(Any)
```

Out[1]:
```
513-element Vector{Any}:
 AbstractArray
 AbstractChannel
 AbstractChar
 AbstractDict
 AbstractDisplay
 AbstractMatch
 AbstractPattern
 AbstractSet
 AbstractString
 Any
 Base.AbstractBroadcasted
 Base.AbstractCartesianIndex
 Base.AbstractCmd
 ⋮
 Tuple
 Type
 TypeVar
 UndefInitializer
 Val
 Vararg
 VecElement
 VersionNumber
 WeakRef
 ZMQ.Context
 ZMQ.Socket
 ZMQ._Message
```

so it makes sense that `Any` is also a supertype of `Any`

In [2]:
```
supertype(Any)
```

Out[2]:
```
Any
```

I want to show the hierarchy starting from type `Number`. The output of supertype is a vector so I'll create a function to print all the branches from the type `Number`.

I want the input of my function to be whatever `Any` / `Number` is which is a `DataType`

In [3]:
```
typeof(Number)
```

Out[3]:
```
DataType
```

`DataType` is not a subtype of `Any` and its type is still `DataType`.

It is a subtype of `Type` which is a subtype of `Any`

In [4]:
```
DataType in subtypes(Any)
```
Out[4]:
```
false
```

In [5]:
```
typeof(DataType)
```
Out[5]:
```
DataType
```

In [6]:
```
supertype(DataType)
```
Out[6]:
```
Type{T}
```

In [7]:
```
subtypes(DataType)
```
Out[7]:
```
Type[]
```

In [8]:
```
subtypes(Type)
```
Out[8]:
```
4-element Vector{Any}:
 Core.TypeofBottom
 DataType
 Union
 UnionAll
```

In [9]:
```
supertype(Type)
```
Out[9]:
```
Any
```

In [10]:
```
length(subtypes(DataType))
```
Out[10]:
```
0
```

In [11]:
```
supertype(DataType)
```
Out[11]:
```
Type{T}
```

Now we want to explore the type hierarchy of `Number`. I made a function to make this a bit easier.

In [12]:
```
supertype(Number)
```
Out[12]:
```
Any
```

In [13]:
```
function get_hierarchy(x; tabs::Integer=0)
    println(repeat("    ", tabs), "└--", x)
    st = subtypes(x)

    if length(st) > 0
        for i in st
            get_hierarchy(i, tabs=tabs+1)
        end
```

```
    end
end
```

Out[13]:
```
get_hierarchy (generic function with 1 method)
```

In [14]:
```
get_hierarchy(Number)
```

```
└--Number
    └--Complex
    └--Real
        └--AbstractFloat
            └--BigFloat
            └--Float16
            └--Float32
            └--Float64
        └--AbstractIrrational
            └--Irrational
        └--Integer
            └--Bool
            └--Signed
                └--BigInt
                └--Int128
                └--Int16
                └--Int32
                └--Int64
                └--Int8
            └--Unsigned
                └--UInt128
                └--UInt16
                └--UInt32
                └--UInt64
                └--UInt8
        └--Rational
```

KR2: Implemented and used at least one own composite type via struct. Generate two more versions that are mutable type and type-parametrized of the custom-built type.

One thing I've used commonly is generating Gaussians so I'll make a composite type for that. Simplest way to describe them is just via mean / variance. I'll add N to pretend it's number of particles or something just so I can add an Int type that sort of makes sense.

In [15]:
```
struct gaussian
    μ::Float64
    σ::Float64
    N::Int
end
```

In [16]:
```
typeof(gaussian)
```

Out[16]:
```
DataType
```

In [17]:
```
g = gaussian(0.0, 1.0, 1000)
```

Out[17]:
```
gaussian(0.0, 1.0, 1000)
```

In [18]:
```
typeof(g)
```

Out[18]:
```
gaussian
```

In [19]:
```
print("μ is $(g.μ), σ is $(g.σ), N is $(g.N)")
```

```
μ is 0.0, σ is 1.0, N is 1000
```

Let's see if I can change the value once it's set. Since this isn't mutable, I'm assuming that I can't.

In [20]:
```
g.μ = 0.5
```

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

Stacktrace:
 [1] setproperty!(x::gaussian, f::Symbol, v::Float64)
   @ Base ./Base.jl:34
 [2] top-level scope
   @ In[20]:1
 [3] eval
   @ ./boot.jl:360 [inlined]
 [4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::
String)
   @ Base ./loading.jl:1094
```

Now I want to try to create a mutable struct

In [21]:
```
mutable struct gaussian_mutable
    μ::Float64
    σ::Float64
    N::Int
end
```

Seems like there's no distinction using `typeof` if it is mutable or not.

In [22]:
```
typeof(gaussian_mutable)
```

Out[22]:
```
DataType
```

In [23]:
```
g_m = gaussian_mutable(0.0, 1.0, 1000)
```

Out[23]:
```
gaussian_mutable(0.0, 1.0, 1000)
```

In [24]:
```
typeof(g_m)
```

Out[24]:
```
gaussian_mutable
```

With a mutable structure, I can change the value!

In [25]:
```
g_m.μ
```

Out[25]:
```
0.0
```

In [26]:
```
g_m.μ = 0.5
```

Out[26]:
```
0.5
```

In [27]:
```
g_m
```

Out[27]: gaussian_mutable(0.5, 1.0, 1000)

Last I need to show a type parametrized struct

In [28]:
```
struct gaussian_parametrized{T}
    μ::T
    σ::T
    N::Int
end
```

Interestingly, when it's parametrized, its type is `UnionAll`

In [29]:
```
typeof(gaussian_parametrized)
```

Out[29]: UnionAll

Here we see that the type depends on the inputs for each parameter

In [30]:
```
gaussian_parametrized(0.0, 1.0, 1000)
```

Out[30]: gaussian_parametrized{Float64}(0.0, 1.0, 1000)

In [31]:
```
gaussian_parametrized(0, 1, 1000)
```

Out[31]: gaussian_parametrized{Int64}(0, 1, 1000)

Wondering what would happen if two are parametrized to `T` with different datatypes; Result: It doesn't work!

In [32]:
```
gaussian_parametrized(0.0, 1, 1000)
```

```
MethodError: no method matching gaussian_parametrized(::Float64, ::Int64, ::Int64)
Closest candidates are:
  gaussian_parametrized(::T, ::T, ::Int64) where T at In[28]:2

Stacktrace:
 [1] top-level scope
   @ In[32]:1
 [2] eval
   @ ./boot.jl:360 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::
String)
   @ Base ./loading.jl:1094
```

In [33]:
```
g_p = gaussian_parametrized(0, 1, 1000)
```

Out[33]: gaussian_parametrized{Int64}(0, 1, 1000)

I didn't declare this as a mutable struct so value shouldn't be changeable.

In [34]:
```
g_p.μ = 0.0
```

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

Stacktrace:
```

```
 [1] setproperty!(x::gaussian_parametrized{Int64}, f::Symbol, v::Float64)
   @ Base ./Base.jl:34
 [2] top-level scope
   @ In[34]:1
 [3] eval
   @ ./boot.jl:360 [inlined]
 [4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::
String)
   @ Base ./loading.jl:1094
```

Lastly, let's make a parametrized, mutable struct.

In [35]:
```julia
mutable struct gaussian_parametrized_mutable{T}
    μ::T
    σ::T
    N::Int
end
```

In [36]:
```julia
g_pm = gaussian_parametrized_mutable(0, 1, 1000)
```

Out[36]:
```
gaussian_parametrized_mutable{Int64}(0, 1, 1000)
```

In [37]:
```julia
g_pm.μ = 1;
g_pm
```

Out[37]:
```
gaussian_parametrized_mutable{Int64}(1, 1, 1000)
```

It works! And just to see that when you change the value it should still be of the same type `T` that is initialized:

In [38]:
```julia
g_pm.μ = 0.5
```

```
InexactError: Int64(0.5)

Stacktrace:
 [1] Int64
   @ ./float.jl:723 [inlined]
 [2] convert
   @ ./number.jl:7 [inlined]
 [3] setproperty!(x::gaussian_parametrized_mutable{Int64}, f::Symbol, v::Float64)
   @ Base ./Base.jl:34
 [4] top-level scope
   @ In[38]:1
 [5] eval
   @ ./boot.jl:360 [inlined]
 [6] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::
String)
   @ Base ./loading.jl:1094
```

KR3: Demonstrated type inference in Julia. Generator expressions may be used for this.

Following the example in the textbook, we can show type inference:

Julia knows when it's either an integer or a float:

In [39]:
```julia
[x for x in 1:5]
```

Out[39]:
```
5-element Vector{Int64}:
 1
 2
```

```
3
4
5
```

In [40]:
```
[x for x in 1.0:5.0]
```

Out[40]:
```
5-element Vector{Float64}:
 1.0
 2.0
 3.0
 4.0
 5.0
```

Seems like float takes precedence (which makes sense)

In [41]:
```
[x for x in 1.0:5]
```

Out[41]:
```
5-element Vector{Float64}:
 1.0
 2.0
 3.0
 4.0
 5.0
```

In [42]:
```
[x for x in 1:5.0]
```

Out[42]:
```
5-element Vector{Float64}:
 1.0
 2.0
 3.0
 4.0
 5.0
```

Also interesting to note that type inference can still be shown when adding things to the generator expression

In [43]:
```
[x + 1.0 for x in 1:5]
```

Out[43]:
```
5-element Vector{Float64}:
 2.0
 3.0
 4.0
 5.0
 6.0
```

In [44]:
```
[x + 1 for x in 1:5]
```

Out[44]:
```
5-element Vector{Int64}:
 2
 3
 4
 5
 6
```

In [45]:
```
[x + 1 for x in 1.0:5]
```

Out[45]:
```
5-element Vector{Float64}:
 2.0
 3.0
 4.0
```

```
5.0
6.0
```

KR4: Created a function with inherent type-instability. Create a version of the function with fixed issues.

Doing something similar to the expression on the book `pos`, where

$$pos(x) = \begin{cases} x, & \text{if } x > 0. \\ 0, & \text{otherwise.} \end{cases}$$

Learned something new from https://docs.julialang.org/en/v1/manual/control-flow/ which is the ternary operator and makes the function more compact code-wise.

In [46]:
```
pos_unstable(x) = x > 0 ? x : 0
```

Out[46]:
```
pos_unstable (generic function with 1 method)
```

According to the textbook, we can demonstrate type instability when the output of a function depends on the value of the input and not just its type.

Consider float inputs, one positive and one negative. We can see that because of how we wrote the function, the output is of type Int64 no matter the type of the input if the input is a number less than or equal to 0.

In [47]:
```
typeof(42.0)
```

Out[47]:
```
Float64
```

In [48]:
```
typeof(-42.0)
```

Out[48]:
```
Float64
```

In [49]:
```
pos_unstable(42.0)
```

Out[49]:
```
42.0
```

In [50]:
```
typeof(pos_unstable(42.0))
```

Out[50]:
```
Float64
```

In [51]:
```
pos_unstable(-42.0)
```

Out[51]:
```
0
```

In [52]:
```
typeof(pos_unstable(-42.0))
```

Out[52]:
```
Int64
```

We can fix this by returning a 0 with the same type as the input

In [53]:
```
pos_stable(x) = x > 0 ? x : zero(x)
```

Out[53]: pos_stable (generic function with 1 method)

In [54]:
```
typeof(pos_stable(42.0))
```

Out[54]:
```
Float64
```

In [55]:
```
typeof(pos_stable(-42.0))
```

Out[55]:
```
Float64
```

In [56]:
```
typeof(pos_unstable(-42))
```

Out[56]:
```
Int64
```

KR5: Demonstration of how @code_warntype can be useful in detecting type-instability .

For type unstable functions with a possible float input and int output, there's a red warning that highlights possible type-instability when using `@code_warntype` , regardless of the input value.

In [57]:
```
@code_warntype pos_unstable(42.0)
```

```
Variables
  #self#::Core.Const(pos_unstable)
  x::Float64

Body::Union{Float64, Int64}
1 ─ %1 = (x > 0)::Bool
└──      goto #3 if not %1
2 ─      return x
3 ─      return 0
```

In [58]:
```
@code_warntype pos_unstable(-42.0)
```

```
Variables
  #self#::Core.Const(pos_unstable)
  x::Float64

Body::Union{Float64, Int64}
1 ─ %1 = (x > 0)::Bool
└──      goto #3 if not %1
2 ─      return x
3 ─      return 0
```

For the same unstable code, if the input is also an int, there is no red "warning".

In [59]:
```
@code_warntype pos_unstable(42)
```

```
Variables
  #self#::Core.Const(pos_unstable)
  x::Int64

Body::Int64
1 ─ %1 = (x > 0)::Bool
└──      goto #3 if not %1
2 ─      return x
3 ─      return 0
```

In [60]:
```
@code_warntype pos_unstable(-42)
```

```
Variables
  #self#::Core.Const(pos_unstable)
  x::Int64

Body::Int64
1 ─ %1 = (x > 0)::Bool
└──      goto #3 if not %1
2 ─      return x
3 ─      return 0
```

Further highlighting the that when the function does not have type-instability, there is no warning for `@code_warntype` and the type or value of the inputs don't really change the output at all.

In [61]:

```
@code_warntype pos_stable(-42.0)
```

```
Variables
  #self#::Core.Const(pos_stable)
  x::Float64

Body::Float64
1 ─ %1 = (x > 0)::Bool
└──      goto #3 if not %1
2 ─      return x
3 ─ %4 = Main.zero(x)::Core.Const(0.0)
└──      return %4
```

In [62]:

```
@code_warntype pos_stable(-42)
```

```
Variables
  #self#::Core.Const(pos_stable)
  x::Int64

Body::Int64
1 ─ %1 = (x > 0)::Bool
└──      goto #3 if not %1
2 ─      return x
3 ─ %4 = Main.zero(x)::Core.Const(0)
└──      return %4
```

In [63]:

```
@code_warntype pos_stable(42.0)
```

```
Variables
  #self#::Core.Const(pos_stable)
  x::Float64

Body::Float64
1 ─ %1 = (x > 0)::Bool
└──      goto #3 if not %1
2 ─      return x
3 ─ %4 = Main.zero(x)::Core.Const(0.0)
└──      return %4
```

In [64]:

```
@code_warntype pos_stable(42.0)
```

```
Variables
  #self#::Core.Const(pos_stable)
  x::Float64

Body::Float64
1 ─ %1 = (x > 0)::Bool
└──      goto #3 if not %1
2 ─      return x
```

```
3 ─ %4 = Main.zero(x)::Core.Const(0.0)
└──      return %4
```

KR6: Demonstration of how Arrays containing ambiguous/abstract types often results to slow execution of codes. The BenchmarkTools may be useful in this part.

In [65]:
```julia
using BenchmarkTools
```

We know that `Number` is a more abstract type and `Int64` is a more specific type so we can compare two arrays with different declared types.

In [66]:
```julia
abstract_ = Number[1,2,3,4]
```

Out[66]:
```
4-element Vector{Number}:
 1
 2
 3
 4
```

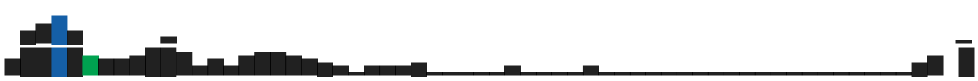
In [67]:
```julia
concrete = Int64[1,2,3,4]
```

Out[67]:
```
4-element Vector{Int64}:
 1
 2
 3
 4
```

The simplest thing to do would be to just get the sum of the contents of both arrays and even if there are only four elements in each, we can see the speed improvement from the benchmarks of around 3x for the example whent the variable type is not abstract.

In [68]:
```julia
@benchmark sum(abstract_)
```

Out[68]:
```
BenchmarkTools.Trial: 10000 samples with 957 evaluations.
 Range (min … max):  92.258 ns … 220.446 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     92.698 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   92.951 ns ±   3.206 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```



```
 92.3 ns        Histogram: log(frequency) by time       100 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

In [69]:
```julia
@benchmark sum(concrete)
```

Out[69]:
```
BenchmarkTools.Trial: 10000 samples with 996 evaluations.
 Range (min … max):  24.889 ns … 114.379 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     24.917 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   25.060 ns ±   1.458 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
 24.9 ns        Histogram: log(frequency) by time      27.4 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

Also wanted to test if just generating a struct would have speed improvements by specifying types.

In [70]:
```julia
struct point
```

```
    x::Float64
    y::Float64
end
```

In [71]:

```
@benchmark point(rand(), rand())
```

Out[71]:

```
BenchmarkTools.Trial: 10000 samples with 998 evaluations.
 Range (min … max):  14.729 ns … 167.199 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     15.615 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   16.105 ns ±   2.395 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
 14.7 ns         Histogram: log(frequency) by time       25.2 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

In [72]:

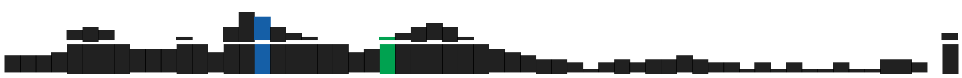
```
struct point_
    x
    y
end
```

In [73]:

```
@benchmark point_(rand(), rand())
```

Out[73]:

```
BenchmarkTools.Trial: 10000 samples with 996 evaluations.
 Range (min … max):  25.085 ns …   2.209 μs  ┊ GC (min … max): 0.00% … 98.47%
 Time  (median):     28.275 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   29.929 ns ± 49.758 ns  ┊ GC (mean ± σ):  4.37% ±  2.60%
```



```
 25.1 ns         Histogram: log(frequency) by time       36.9 ns <

 Memory estimate: 32 bytes, allocs estimate: 2.
```

Based on this, even with the same input types, there's ~a 2x speed improvement when we declare Float64 for the struct parameters.

In [ ]: